CTS 2322 (Unix/Linux Administration II) Lecture Notes

By Wayne Pollock

Lecture 1 — Backups and Archives

Welcome! **Introduce course**: Admin I covered basic concepts and procedures. In this course we continue this, covering more advanced concepts and procedures: basic networking, security, licensing, kernels, logging and monitoring, and basic setup of DNS, Apache, and email services. We will also build software from source, and build, configure, and install a kernel.

Discuss Project 1 — Partition, install Linux, **pass out Fedora CDs**. Review post-install tasks (on website).

Review ***system journal*** (use file not paper, can use wiki). Old YborStudent journal is on-line as a model.

Have students create accounts (with their real names) and log into class wiki (*note: in the future, have instructor create student accounts using HCC ID*):

`YborStudent.hccfl.edu/UnixWiki/`

Review the SA's Code of Ethics. (Show LOPSA link.) Discuss case study; this is testable material. (*Expand this section!*)

Backups and Archives

Statistic: 94% of companies suffering from a catastrophic data loss do not survive; 43% never reopen and 51% close within two years. (source)

Backups are made for rebuilding a system that is identical to the current one. **Backups are thus for (disaster) recovery**, not transferring of data to another system. They do not need to be portable. In this sense, "backup" is used to mean a complete backup of an entire system: not just regular files but all owner, group, date, and permission info for all files, links, `/dev` entries, some `/proc` entries, etc.

> Backups can be used not only for "bare metal" backup and recovery, but also to install many identical computers. Clonezilla allows you to do just that, similar to the commercial product "Ghost". Clonezilla saves and restores only used blocks in the hard disk. This increases the clone efficiency. In one example, Clonezilla was used to clone one computer to 41 other computers simultaneously. It took only about 10 minutes to clone a 5.6 GiB system image to all 41 computers via multicasting.

Archives* are for transferring data to other systems**, ***operational (day-by-day, file-by-file) recovery, or making copies of files for historical or legal/regulatory purposes. As such, they should be portable so that they may be recovered on new systems when the original systems are no longer available. For example, it should be possible for an archive of the files on a Solaris Unix system to be restored on an AIX Unix, or even a Linux system. (Within limits, this portability should extend to Windows and Macintosh systems as well.)

A backup has a drawback for operational backup-and-restore needs. While backups enable rapid recovery and minimal downtime in the event of a disaster, they also backup any malware-infected or otherwise corrupted files. You generally do not have more than one backup version of storage. Archives however, are made often and many previous versions are available.

Most of the time, the two terms are used interchangeably. (In fact, the above definitions are not universally agreed upon!) In the rest of this document, the term "backup" will be used to mean either a backup or an archive as defined above. Most real-world situations call for archives, since the other objects (such as `/dev` entries) rarely if ever change on a production server once the system is installed and configured. A single "backup" is usually sufficient. For home users, the original system CDs often serve as the only backup need; all other backups are of modified files only and hence are "archives".

Using RAID is *not* a replacement for regular backups! (Imagine a disaster such as a fire on your site, an accidentally deleted file, or corrupted data.)

Service Level Agreement (SLA)

Creating backup policies (includes several sub-policies, discussed below) can be difficult. Keep in mind the operational backup requirements of the organization, often specified in an **SLA** or ***service level agreement***. Make sure users/customers are aware of what gets backed up and what doesn't, how to request a restore, and how long it might take to restore different data from various problems (very old data, a fire destroys the hardware, a DB record accidentally deleted from yesterday, ...).

Statistics from EMC.com:

- 80% of restore requests are made within 48 hours of the data loss.
- Around 15% of a storage administrator's time is spent on backup and recovery.
- Between 5% and 20% of backup/restore jobs fail.
- In 2004, backup and recovery costs were about $6,000 per TB of data, per year.

(*Show example SLA from Technion University*.)

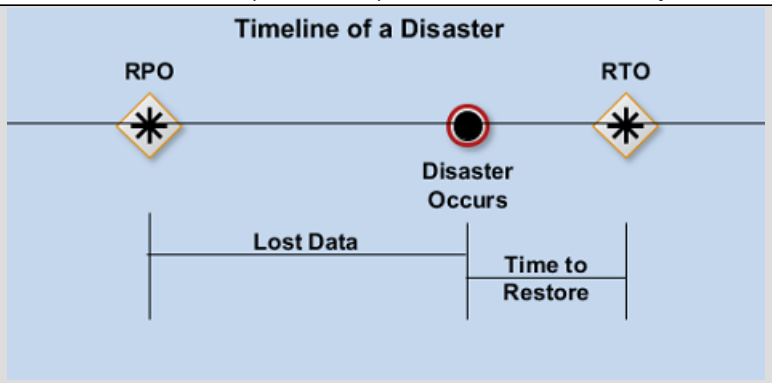
You should know these related definitions, not just for backups but in general:

- **SLI** — A *Service Level Indicator* is a measurable metric, whose values can be classified as either good or bad. An example SLI for a web service: 99% of all requests in a calendar year should have a response time of under 200 ms.
- **SLA** — A *service level agreement* is a list of SLIs that define the **required** behavior of some service. This should include all failure modes. Examples include what happens if the data center loses power, or if available network bandwidth is exhausted.
- **SLO** — A *Service Level Objective* is also a list of SLIs, but instead of listing guarantees it lists the level of service you are aiming for. For an SLA with the sample SLI from above, the SLO might be "99.99% of all requests complete in 200 ms or less". (Often, SLAs are composed of several SLOs, rather than specific SLIs.) See Wikipedia for a good discussion of these terms.

The business world often uses many acronyms with overlapping meaning. Besides SLA and SLO, you may see references to RPO and RTO when considering backups and recovery.

Recovery Point Objective (RPO) refers to the point in time in the past to which you will recover. It is the last point in time you did a backup.

Recovery Time Objective (RTO) refers to the point in time in the future at which you will be up and running again. It is the *time to restore*.



More frequent backups lead to smaller RPOs. Some technologies such as disk-to-disk backups lead to smaller RTOs. Both of these need to be specified in an SLA.

Most people underestimate how slow a restore operation can be. **It is often 10-20 times longer to restore a file than to back one up.** (One reason: operating systems are usually optimized for read operations, not write operations.)

An example SLA: Customers should be able to recover any file version (with a granularity of a day) from the past 6 months, and any file version (with a granularity of a month) from the past 3 years. Disk failure should result in no more than 4 hours of down-time, with at worst 2 business days of data lost. Archives will be full backups each quarter and kept forever, with old data copied to new medium when the older technology is no longer supported. Critical data will be kept on a separate system with hourly snapshots between 7:00 AM and 7:00 PM, with a midnight snapshot made and kept for a week. Users have access to these snapshots. Database and financial data have different SLAs for compliance reasons.

Granularity refers to how often backups are made. For example, suppose backups are made each night at midnight. If some user edits a file six times in the last two days, they can't get back any version; only the copy taken at midnight. In some cases, you want *finer* granularity (e.g., versions for each hour, or every single version) and in other cases, *coarser* granularity is fine (versions every month).

SLAs vary considerably. For example, your online sales system may require recovery granularity of one transaction, and recovery time in seconds. Emails may require per-email granularity for a day (or more) with recovery time in minutes, and 24-hour granularity for older email. On the other hand, old business records (such as stock-holder meeting minutes) can have granularity of days, and (except when regulated) recovery time in days.

You must also worry about security of your backups. Have a clear policy on who is allowed to request a restore and how to do so, or else one user might request a restore of other's files. In some cases, this may be allowed by a manager or

auditor. (In a small organization where everyone knows everyone, this is not likely to be a problem.)

In addition to access control, backups should always be encrypted with a password (or preferably, a key) not stored on the server's disks. To protect against ransomware (where data is deleted or encrypted by the attacker), recent (for example, the past 30 days) backups should be immutable (read-only).

The backup process and backup server must be made as secure as possible. Think about it: the backup server needs remote root access to all your production servers! If you automate backups, there is not even a password to protect the process!

To secure the backup server, do not use the server for anything else. Strip off unnecessary services and turn off the ones you cannot remove. Harden that server, and limit (incoming) access to SSH from a few internal IP addresses. Create a user (and group) just for the backup process to use.

To keep the process secure, have the backup process connect to the remote machines using an SSH key. (The key cannot be password protected if you wish to automate (have unattended) backups.) On the hosts to be backed-up, add the backup user with SSH key authentication only. Next, use SSH features to prevent that user from running any program except the backup software. That backup software will need root access. The best way (if possible) is to have that software run `sudo` to run the piece that actually needs to read the protected files. Finally, configure `sudo` to allow the backup user only to run (as root) that one command.

Types and Strategies of Backups and Archives

It is possible to backup only a portion of the files (and other objects in the case of a backup) on your systems. In fact, there are three types of backups (or archives):

1. **Full** (also known as "epoch" or "complete") — everything gets backed-up.
2. **Incremental** — backup everything that has been added or modified since the last backup of any type (either incremental or full).
3. **Differential** — backup everything that has been added or modified since the last full backup. Differentials can be assigned levels: *level 0* is a full backup and *level n* is everything that has changed since the last *level n-1* backup. (Differential is sometimes called *cumulative*.)

(Many people don't bother to distinguish between incremental and differential.) A system administrator must choose a ***backup strategy*** (a combination of types) based on several factors. The factors to consider are safety, required down time, cost, convenience, and speed of backups and recovery. These factors vary in importance for different situations. Common strategies include **full backups with incremental backups in-between**, and **full backups with differential backups in-between** (a two-level differential). Sometimes a three-level differential is used,

but rarely more levels. (You rarely use both incremental and differential backups as part of a single strategy.) The strategy of using only full backups is rarely used.

What with modern backup software, the differences between the strategies mentioned above aren't that large. Incrementals take less time to backup and more time to restore (since several different backup media may be needed) compared with differential backups (where at most two media, the last differential and the last full backup media, are used to recover a file). Full backups take a huge amount of time to make but recovery is very fast. (Example: disk corruption on the 25th of the month: recovery is last full then last differential, or last full then 24 daily incrementals.)

Most commercial software keeps a special file that is reset for each full backup and keeps track of which incremental tape (or disk or whatever) holds which files. This file is read from the last incremental tape during a restore, to determine exactly which tape to use to recover some file. Such information is known as *backup metadata*, or the backup *catalog*.

> ***Backup Metadata*** contains information about what has been backed up, such as file names, time of backup, size, permissions, ownership, and most importantly, tracking information for rapid location and restore. It also indicates where it has been stored, for example, which tape.

When devising a backup strategy, it is critical to understand the nature of the data and the nature of changes to the data. Granularity levels depend on several considerations. What is the aggregate weekly data change rate? If the change rate were close to or greater than 100% (daily change about 20%), it makes little sense to use an incremental backup, because the overhead for deciding which files need to be backed up may be longer than the time it takes for a full backup.

Another consideration is file size. Some applications use larger files than others. An environment with such applications tends to have a larger data change rate, because even a small change to the data results in the whole file being changed. The larger the average file size, the greater the percentage of the data set. Other applications, like software development, use many smaller files. The rate of change in these environments can be much lower. In such environments, the more mature the data set, the lower the change rate.

Another factor to consider is the properties of the files in your backup set. For instance, are they natively compressible? If not, the negative impact compression has on performance makes it less desirable.

The Backup Schedule

The frequency of backups (the ***backup schedule***) is another part of the policy. In some cases, it is reasonable to have full backups daily and incremental backups several times a day. In other cases, a full backup once a year with weekly or monthly incremental backups could be appropriate. A common strategy suitable

for most corporate environments would be monthly full backups and daily differential backups. (Another example might be quarterly full (differential level 0) backups, with monthly level 1 differentials, and daily level 2 differentials.) However more frequent full backups may save tapes (as the incremental backups near the end of the cycle may be too large for a single tape).

Note that in some cases there will be legal requirements for backups at certain intervals (e.g., the SEC for financial industries, the FBI for defense industries, or regulations for medical/personal data). Depending on your backup software, it may be required to bring the system partially or completely off-line during the backup process. Thus, there is a trade-off between convenience versus cost, versus the safety of more frequent backups.

In a large organization, it may not be possible to perform a full backup on all systems on the same weekend; there is usually too much data to backup in the time window provided. A ***staggered schedule*** is needed, where (say) 1/4 of the servers get backed up on the first Sunday of the month, 1/4 the second Sunday, and so on. Each server is still being backed up monthly but not all on the same day of the month.

> Be aware that **small changes to the schedule can result in dramatic changes in the amount of backup media needed**. For example, suppose you have 4GB to backup within this SLA: full backup every 4 weeks (28 days) and differential backups between. Now assume the differential backup grows 5% per day for the first 20 days (80% has changed) and stays the same size thereafter. Some math reveals that doing full backups each week (which still meets the SLA) will use a third the amount of the tape of a 28-day cycle, in this case.

Good schedules require a lot of complex calculation to work out (and still meet the *SLAs*). Modern backup software (such as *Amanda)* allows one to specify the SLA and will create a schedule automatically. A ***dynamic schedule*** will be adjusted automatically depending on how much data is actually copied for each backup. Such software will simply inform the SA when to change the tapes in a jukebox.

On a busy (e.g., database) server downtime will be the most critical factor. In such cases consider using **LVM *snapshots***, which very quickly makes a read-only copy of some logical volume using very little extra disk space. You can then backup the snapshot while the rest of the system remains up. This is a popular strategy, but you must be careful! **When transferring the data to your backup device, you don't want to hog the network, CPU(s), or disk I/O.** When compressing the data as you transfer it, which is common, you don't want to hog all your CPU cores. You can use `nice` and `ionice` to limit CPU and disk use, and also limit the number of threads used when compressing to use just one or two cores. For example, "`zstd -T2` ...".

Another strategy is called *disk-to-disk-to-tape*, in which the data to be backed up is quickly copied to another disk and then written to the slower backup medium later.

> Gnu `cp` command can make COW copies of files on filesystems that support such. This can be very fast to make a copy of a large file, useful when making a backup of a database file for example. The technical name for such a COW copy is *reflink*. You create the copy with "`cp --refcopy src dest`". You can control what happens when you try this on a filesystem that doesn't support COW, or try to copy from one filesystem to another. (See the Gnu `cp` man page.)

Sometimes different applications require independent backup of their data for various reasons (such as different security, retention, or SLA requirements). Examples include servers versus PCs, mobile devices (laptops and smart phones), different databases, email, log data, and so on. Even the backup procedures may be different. For example, an LVM snapshot won't correctly backup a database that was in the middle of some operation or had data cached in memory at the time.

> **Policy questions that should be answered by SLA:**
>
> - What are the restore requirements – granularity (what points in time can be requested), and time for various types of recovery?
> - Where and when will the restores occur?
> - What are the most frequent restore requests?
> - Which data needs to be backed up?
> - How frequently should data be backed up (hourly, daily, weekly, monthly), and with what strategy (full, differential, incremental)?
> - How long will it take to backup?
> - How many copies to create?
> - How long to retain backup copies?

Other Policies

Deciding **what to backup** is part of your policy too. Are you responsible for backing up the servers only? Boss' workstation? All workstations? (Users need to know!) Network devices (e.g., routers and switches)? It may be appropriate to use a different backup strategy for user workstations than for servers, for different servers, or even different partitions/directories/files of servers.

> An often-overlooked item is the MBR/GPT. Make a copy of it with:
>
> ```
> dd if=/dev/sda of=/tmp/MBR.bak bs=$SECTOR_SIZE count=1
> ```

Another part of your backup policy is determining how long to keep old backups around. This is called the ***backup retention policy***. In many cases, it is appropriate to retain the full backups indefinitely. In some cases, backups should be kept for 7 to 15 years (in case of legal action or an IRS audit). In some cases, you must not keep certain data for too long or you may face legal penalties.

> Medical records must be kept for 6 years (HIPAA), 10 years (Medicare), 40 years (OSHA in some cases), or even 75 years (Department of Veterans Affairs). Other kinds od data have similarly complex requirements. Worse, additional requirements vary by state.

Such records are often useful for more than disaster recovery. You may discover your system was compromised months after the break-in. You may need to examine old files when investigating an employee. You may need to recover an older version of your company's software. Such records can help if legal action (either by your company for by someone else suing your company) occurs.

Since Enron scandal (2001) and Microsoft scandals (when corporate officers had emails subpoenaed by DoJ), a common new policy is "if it doesn't exist it can't be subpoenaed!" These events may have led to this revision of the *FRCP*:

> **FRCP — the *Federal Rules of Civil Procedure***
>
> These include rules for handling of ESI (*Electronically Stored Information*) when legal action (e.g. lawsuits) is immanent or already underway. **You must suspend normal data destruction activities** (such as reusing backup media), possibly make "snapshot" backups of various workstations, log files, and other ESI, classify the ESI as *easy* or *hard* to produce, and the cost to produce the hard ESI (which the other party must pay), work out a "discovery" (of evidence) plan, and actually produce the ESI in a legally acceptable manner. An SA should consult the corporate lawyers well in advance to work out the procedures.

It is important to decide **where store the backup media** (***storage policy/strategy***). These tapes or CDs contain valuable information and must be secured. Also, it makes no sense to store media in the same room as the server the backup was made from; if something nasty happens to the server, such as theft, vandalism, fire, etc., then you lose your backups too. A company should store backup media in a secure location, preferably off-site. A bank safe-deposit box is usually less than $200 a year and is a good location to store backup media. If on-site storage is desirable, consider a fire-proof safe. (And keep the door shut all the time!) Consider remote storage companies but beware of bandwidth and security issues.

Different storage methods offer different levels of accessibility, security and cost for your backup data:

- **Online storage**: Sometimes called secondary storage, online storage is typically the most accessible type of data storage. A good example would be a large disk array. This type of storage is very convenient and speedy, but is relatively expensive and vulnerable to being deleted or overwritten, either by accident, or in the wake of a data-deleting virus payload. (HCC has an online storage site in Lakeland.)
- **Near-line storage**: Sometimes called tertiary storage, near-line storage is typically less accessible and less expensive than online storage. A good example would be an automatic tape library. Near-line storage is used for archival of rarely accessed information, since it is much slower than secondary storage.
- **Offline storage**: An example of offline storage is a computer storage system which must be driven by a human operator before a computer can access the information stored on the medium. For example, a media library system which uses off-line storage media, as opposed to near-line storage, where the handling of media is automatic.
- **Off-site vault**: To protect against a disaster or other site-specific problem, many people choose to send backup media to an off-site vault. The vault can be as simple as the system administrator's home office or as sophisticated as a disaster hardened, temperature controlled, high security bunker that has facilities for backup media storage.

In most cases, a mix of storage methods can be the most effective storage strategy.

Media Replacement Policy (a.k.a. Media Rotation Policy)

Backup media (tapes) will not last forever. Considering how vital the backups might be, it is a false economy to buy cheap tape or reuse the same media over and over. A reasonable ***media replacement policy*** (also known as the ***media rotation schedule*****)** is to use a new tape a fixed number of times, then toss it. The rotation schedule/replacement policy can have a major impact on the cost of backups and the speed of recovery.

Before using new media for the first time, test it and give it a unique, permanent label (number). **Annual backup tapes could be duplicated just in case the original fails.**

There are many possible schemes for media rotation. A simple policy is called ***incremental rotation*** which means different things to different folks. Basically, you should number the media used for a given cycle, such as D1–D31 for the 31 tapes used for daily backups. After one complete cycle (with each tape used once), the next cycle uses tapes D2–D32; tape D1 gets re-labeled as M1. The 12 monthly tapes M1-M12 are each used once, then M2-M13 is used the following year, etc. The old M1 tape becomes permanently retired (or archived). Thus, a given tape

will be used 31 times for daily, 12 times for monthly, and once for yearly backups, a total of 43 uses before you need to replace it.

One of the most popular schemes is called grandfather, father, son (GFS) rotation. (This term predates *political correctness*.) This scheme uses daily (son), weekly (father), and monthly (grandfather) backup sets. In each set, the oldest tape is used for the next backup. Here's an illustration (from `mckinnonsc.vic.edu.au/vceit/backups/backup_schemes.htm`):

- Monday - daily backup to tape #1
- Tuesday - daily backup to tape #2
- Wednesday - daily backup to tape #3
- Thursday - daily backup to tape #4
- Friday - weekly to tape #5. This tape is called Week 1 Backup.

(Of course, you can extend this idea to seven-day schemes as well.)

The next Monday through Thursday you would re-use tapes #1 through #4 for the daily backup set. But next Friday you do another weekly backup to Week 2 backup (tape #6). Week 3 is the same as week 2, using Week 3 backup (tape #7) on Friday. Tapes 5–7 form the weekly backup set.

At the end of the fourth week do a monthly to Month 1 backup tape (tape #8). At the end of the fifth week, Week 1 tape is re-used.

So, the daily tapes are recycled weekly. The weekly tapes are recycled monthly. Monthly tapes are recycled annually. Each year a full annual backup is kept safely stored and never re-used.

Clearly, the daily tapes (tapes 1–4) are used much more often than the weeklies (tapes 5–7) and monthlies (tapes 8–∞). This will mean they will suffer more wear and tear and may fail more readily.

The incremental scheme can be used with any rotation policy, such as GFS or Towers of Hanoi. A tape (e.g. tape 1) will be used as a weekly tape after a month (or two or more) of daily use. After a year (or two or more) of weekly use, it can be error-checked (in case it's becoming unreliable) and will be used as a monthly tape. After 12 (or 24 or more) uses as a monthly tape it could be "retired" as a permanent yearly backup tape.

Most software that automates backup uses the tower of Hanoi method, which is more complex but does result in a better policy.

Class discussion: Determine backup policies for YborStudent server. One possibility: Full backup (level 0) of `/home` one per term, level 1 once per month, level 2 each day. The SLA will specify a recovery time of a maximum of 2 working days. Backups should be kept for 6 months after the end of the semester.

For security reasons, you should completely erase the media before throwing the media in the trash. (This is harder than you think!) An alternative is to shred or burn old media, and/or encrypting backups as they are made.

> The time for a restore depends if incremental or differential backups are done for daily and weekly. In this scheme, the monthly backup is usually a full backup, but doesn't need to be if you use a 4 or 5 level differential backup scheme.

Backup Media Choices

There are too many choices to count. (And the information is likely out of date before I finish typing this!)

For smaller archives, flash or other removable disks, writable CD-ROMs or DVDs, (These are WORM media) and old fashioned DLT, DAT, DDS-{2,4,8,16} tape drives were popular. (I used a DDS-2 SCSI drive at home.) While using tape for backups is no longer popular for consumers, it is more popular than ever for the enterprise, especially those with enormous amounts of data to backup.

Consider **LTO** (linear tape open) drives, the most popular type for enterprise use. These are fast (for tape: they max out at 800 MiB/S) and have a very low cost per byte. LTO tapes have a range of densities; LTO-7 (current in 2019), holds 6 TB per tape, with a shelf life of 30-50 years, and costs around $50 to $75 each. An auto-loader LTO-7 drive can cost around $6,000; a cheap drive is still around $2,700 (2020). (See Wikipedia for details on different versions.)

LTO-8 is common today (2022) with a capacity of 12 TB per cartridge, or 30 TB when that data is compressed (which slows down the entire read/write process). LTO-8 tapes are 960 m in length, about 12.5 mm wide, and take under 10 minutes to fill one tape. LTO-9, first available in 2021, doubles storage capacities to 24 TB per cartridge and has faster I/O: the transfer rate, which currently stands at 0.75GBps for LTO-8 and 1GBps for LTO-9 (both compressed). As for pricing, LTO-9 tapes sell for around $8 per TB, with LTO-8 tapes being cheapest at $4.50 cents per TB and LTO-7 ones hitting $6.30.

The LTO consortium has a roadmap for the future up through LTO 14! Each generation is faster and denser than the previous generation. A 33% generational improvement would bring the transfer rate for LTO-14 only to about 4GBps. That sounds high but considering the amount of data to be transferred, estimated in hundreds of petabytes, it may be an issue.

One can expect prices to drop significantly by the time LTO-14 arrives. LTO tape readers are still likely to be expensive; the OWC Mercury Pro LTO-9 tape drive sells for more than $6,000 in 2022.

> [From gizmodo.com, read on 12/15/20:] Earlier this year (2020) Fujifilm revealed a breakthrough that could push tape storage capacities to 480 TB

in a decade's time. Current data tape technologies also rely on a material called Barium Ferrite (BaFe) with microscopic magnetic particles that are aligned to encode data onto the long strips of tape, but we're reaching the limitations of how far Barium Ferrite can be improved and optimized to increase storage capacities. As a result, Fujifilm has been researching a new material called Strontium Ferrite (SrFe) as an alternative because its particles are smaller than those in Barium Ferrite, allowing for increased density and, in turn, more data capacity. Tape cartridges packing 480 TB of data could be available by 2030, and unlike flash memory and hard drives, they can reliably store data for upwards of 30 years without the need for any additional power.

IBM Research announced that is has been working with researchers at Fujifilm to further advance the potential of Strontium Ferrite magnetic tape and have successfully managed to squeeze 317 GB of data into a single square inch of the material. At that density, a single tape cartridge could potentially hold up to 580 TB of uncompressed data. The breakthrough comes courtesy not only of the new magnetic coating, but also the development of new low friction tape heads that allow the tape material itself to be very smooth, improving the accuracy and reliability of what's being read and written.

(As of 2019, only Fujifilm and Sony continue to manufacture current LTO media.)

LTO formats change every few years, and an LTO-*n* drive can only work with LTO-*n*, *n-1*, or *n-2* tapes. So every 5-8 years, old data must be migrated to the newest format. This gets expensive and can be time-consuming as well, which is why LTO tape backup is better suited to data centers and larger enterprises than individuals.

Tape storage is very cheap, typically less than $20 for 80 gigabytes of storage. (DDS-2 tapes cost about $7 and hold 4 GB each. DDS-4 tapes are fast backups and hold ~100GB each.) However, tapes and other magnetic media can be affected by strong electrical and magnetic fields, heat, humidity, etc. Also, the higher density tapes require more expensive drives (some over $1,000). LTO tape delivers a 2x - 4x savings in operational costs over SSD backup.

In 2010, the record for how much data magnetic tape could store was 29.5GB per square inch. To compare, a quad-layer Blu-ray disc can hold 50GB per disk. Magnetic tapes can be hundreds of feet long. In 2014, Sony announced that it developed new magnetic tape material that can hold 148GB per square inch. With this material, a standard backup tape (the size of an old cassette) could store up to 185TB. To hold the

> equivalent amount of data would take 3,700 dual-layer 50GB Blu-rays (a stack that would be over 14 feet tall). — extremetech.com
>
> Today (2018), the density of tape storage continues to grow, doubling about every two years.

Compared to hard disks, tape is more reliable (reportedly 1,000 times or more fewer errors), takes zero power to store, and when off-line (unmounted), it is extremely safe from hackers or errors. Modern tape backup units can write data faster than disks can, and can hold much more data. And of course, tape is cheap. Recently (2018), IBM announced a new tape prototype than can hold over 300 TiB on a single tape! The only downside is that to recover the data from tape takes much longer (seconds/minutes) compared to disk (microseconds/milliseconds).

> Large data centers such as Google and Microsoft Azure cloud use both disk and tape backups: disk backup at different locations for quick recovery of some errors, and tape backup for safety. (For example, in 2011 Google lost thousands of emails from all disks due to a software bug, but was able to eventually restore all the lost data from their tape backups.)

An external hard drive (less than $100 for 1TB) connected directly to your PC can use the backup program that comes with your operating system (*Backup and Restore Center* on Windows, and *Time Machine* on OS X). Most backup software can automate backups of all new files or changed ones on a regular basis. This is a simple option if you only have one PC.

Optical media such as CDs are durable and fairly cheap but take much longer to write. They can be reused less often than magnetic media and are still susceptible to heat and humidity. Optical media can scratch if not carefully handled. Also consider the bulk of the media. If you must store seven years' worth of backups, it may be important to minimize the storage requirements and expense. A CD-ROM can hold about 700 MiB while a dual-layer Blu-ray can hold 50 GiB. However, if stored and handled correctly, such media can hold data without any maintenance for many decades at least.

> Optical media can vary greatly in quality. Some manufactures use cheap dyes that will not last 5 years. Blu-ray uses better quality dyes in general than CDs and DVDs, but the major source of rewritable Blu-rays, Sony, has ceased production of them (in 2024).
>
> An alternative is M-disc (the "M" is for millennium). It is much more durable, but also much more expensive.

A choice becoming popular (since 2008) is **on-line storage**, e.g., HP Upline, Google GDrive, etc. (for SOHO, Mozy or BackBlaze). (This market changes rapidly so do research on current companies.) The companies offer cheap data

storage and complete system backups, provided you have a fast Internet connection. Many colocation facilities ("colos", usually at network exchange points) provide this service as well to the connected ISPs. If you go this route, make sure all the data is encrypted using industry standard encryption at your site before transmission across the Internet. (Never use any company that uses "proprietary" encryption regardless of how secure they claim it is!) The major danger to this method is the company may not follow best practices to keep data safe and intact, or may simply go out of business without notice.

When backing up large transaction database files, the speed of the media transfer is important. For instance, a 6 Mbps (Megabits per second) tape drive unit will backup 10 gigabytes in about 3 hours and 45 minutes. (In most cases incremental or differential backups contain much less data!)

> For legacy IDE controllers, you only choice is a **TRAVAN** backup drive. Very slow, don't use!

Backup Performance

Performance of backups and restores matters. Performance is affected by two factors: input/output operations per second (IOPS) and bandwidth. As of 2021, regular hard disks support up to 250 IOPS, SSDs up to 60,000 IOPS. By bypassing a disk controller (SATA, SAS, etc.) and using NVMe (which uses PCIe), you can achieve up to 750,000 IOPS.

For SCSI drives (such as DDS drives from HP) there are two speeds for the SCSI controller, depending on what devices are on it. A tape drive will slow down the SCSI bus by half, so consider dual SCSI controllers.

The connection bandwidth between the server's disks and your backup medium is also critical. If you use hardware with a sufficient IOPS value, the bandwidth is the bottleneck. For example, if you use a 1 Gbps to your backup media, you can transfer about 400 GB per hour. For SOHO use that might be sufficient, but modern enterprises (2022) often have tens of terabytes to backup or restore (think of a database or a bunch of virtual machines). With 1 Gbps 10 TB of data will take over a day! Consider having faster networks, say 10-25 Gbps. With expensive networking hardware, 100 Gbps or faster is possible.

For enterprise scale backups, consider a **networked backup unit**. This would allow a single backup system to be used with many different systems. Thus, you can buy one high-speed device for about the same money as several lower-speed devices. Keep in mind however that a network backup can bring a standard Ethernet network to its knees. Even a fast Ethernet (1 Gbps) LAN might suffer noticeable delays and problems. Often a SAN solution (a separate network for data transfers) works best.

An excellent choice for single-system backup is a USB disk. Also using SAN/NAS to centralize your storage makes it easy to use a single backup system (robot tapes).

It is a good idea to **have a spare media drive** (e.g., DLT tape drive), in case the one built into a computer fails when the computer fails. This is especially true for non-standard backup devices that may not be available from a local store on a moment's notice. Regularly clean and maintain (and test) your backup drives.

(While I don't know of any organization that does this, consider copying old data to new hardware once the old drives are no longer supported or available. If you don't have a working drive (including drivers), the old backups are useless!)

> In the end, most backups still use tape as the most cost-effective solution. Keep in mind that restore from tape can be a very slow, manual process; it may require mounting several tapes to recovery a single file! And if tapes are stored off-site, it may take a day or more just to ship them back to you.
>
> Backup to disk is becoming more popular as the costs of disks go down.

As you can see, many factors must be considered when designing a backup system and its policies. In addition to the ones mentioned earlier, you need to consider the total amount of data to be backed up, total amount to retain, removing old and/or unneeded data, and sanitizing/blinding/anonymizing data that is to be used for reports, research, or training.

Keeping management reports of restore requests can help answer these questions. Suppose you guessed to keep email backups for 3 months. That's expensive, but is that too long or too short? An internal study at EMC.com in 2007 determined that for them, over 25% of restore requests were for the same day's data, and fully 100% of email restore requests were made in two weeks or less after losing email. Without regulatory/legal requirements in their case, they changed the policy and saved a considerable amount of money and effort.

Archival Storage

Magnetic media is not a good choice for very long term, or *archival* backups. The reason is, over time such media is subject to *bit rot*: loss of data just from sitting around, even unpowered. How can this happen?

Hard drives use magnetism to store bits of data in disk sectors. These bits can "flip" over time for many reasons, which can lead to data corruption. To mitigate this, hard drives have error-correcting code (ECC) that can sometimes correct flipped bits. However, if enough bits flip then corruption will occur. This will take time; some hard drives have the potential to last with their data intact for decades even if powered down; most last a year or two only.

Solid-state drives don't have any moving parts like hard drives. These drives use an insulating layer to trap charged electrons inside microscopic transistors to differentiate between 1s and 0s, in groups ("rows"). Over time, the insulating layer

degrades and the charged electrons leak out, thus corrupting data. Powered down, an SSD will see bit rot occur within a few months.

> Enterprise SLC SSD drives stored at 25C and operated at 40C have a typical retention rate of just 20 weeks without power. HDDs usually retain data for 1-2 years in this scenario.

If using such devices for archival backup, **it's a good idea to power them up periodically and let them run `fsck` or DiskFresh.** For a hard drive, you should power it up at least once every year or two to prevent the mechanical parts of the drive from seizing up. You should also "refresh" the data by recopying it or use a third-party tool like DiskFresh (which reads then writes each block, checking for bad blocks as it goes). SSDs are a little simpler since they just need to maintain their charge. You can power them up for a few minutes about twice a year, but I still recommend using some tool to check for bad rows.

Even if you kept your backups powered up all the time, bit rot can occur!

Your best choice for archival storage is likely LTO tape if you can afford it. Otherwise, consider optical media for this.

Digital preservation best practices recommend specific file formats (typically open, non-proprietary, and widely available) for long-term archival use. These formats were chosen because of their documented acceptance by the archival and digital preservation communities. Factors leading to this acceptance include format longevity and maturity, adaptation in relevant professional communities, incorporated information standards, and long-term accessibility of any required viewing software. Examples include TIFF (uncompressed) for still images, PDF for office documents (word processor, spreadsheet, etc.), mbox or XML for email, and so on. (See for example the U.S. National Archives strategy on this topic.)

> Hardware and operating systems (and drivers) go obsolete as well. To standardize digital preservation practice and provide a set of recommendations for preservation program implementation, the Reference Model for an *Open Archival Information System* (OAIS) was developed, as ISO-14721. OAIS is concerned with all technical aspects of a digital object's life cycle: ingest, archival storage, data management, administration, access and preservation planning.

Enterprise Backup Systems

Most enterprise-wide backup systems are designed with a client-server architecture. Each host holding data to be backed up runs a backup client, known as a ***backup agent***. This agent communicates with the (dedicated) ***backup server***. Each backup agent can send messages to, or answer queries from, the backup server.

The server maintains the backup metadata (such as the catalog of what is backed up onto which tapes) and the policies you select. In addition, the server sends commands to the agent to gather data and return it, when performing an actual backup operation.

Typically, the backup server is also connected with one or more ***storage nodes***. which is the hardware/software responsible to reading and writing to/from the backup media. The various backup destinations are also called *storage targets*. Such targets can be clusters of different storage locations (say in different geographical areas).

If your organization does not use a SAN for the data, the clients communicate with the server through your network. ***Network backup*** can be dangerous, as the network capacity may not be sufficient, or may cause timeouts and excessive latency for other applications using the network at the same time. It's even worse when the storage nodes are not directly attached to the backup server. Proper backup systems will limit their network utilization and use a staggered schedule.

If you do have a SAN, you attach storage nodes to it and run the server there; no backup data needs to travel on the network (but backup metadata will). If the storage node isn't directly connected to the SAN unit, it connects through the dedicated storage network (e.g., FibreChannel), which should not cause any utilization or latency problems.

In a smaller setup, ***direct attached backup*** can be used to back up data from a single client. The agent and server are just one application, running on the client, and the storage node is also directly attached to the client.

Since tape storage nodes are slow to read and especially slow to write, today many systems put a disk between the backup server and the slower storage node.

Using a combination of hardware is common for enterprise backup systems, as you can use the best attributes of different hardware for an overall backup architecture. It is common to have a *multilayer* system, where the most recent data (a.k.a. *hot data*) is kept on *tier 1* (or *performance tier*) using SSDs, and older *cold data* stored on cheaper HDDs in tier 2 (or *capacity tier*). Off-line or archival data can be considered as tier 3. Often cloud storage (such as Amazons S3) is used for tier 2 and/or tier 3. Once enough time has passed, data in tier 1 is migrated to tier 2 and eventually to tier 3.

Finally, enterprise backup regulations (or your insurance company) will require regular restores to verify your backups are present and your procedures are working. To support this, you can regularly restore full backups to VMs, alerting on any failures (and logging success).

Some technologies allow virtual machines to be started from tier 1 storage, allowing instant restores of VMs.

Managing and updating such complex backup architectures would be very difficult for most system administrators. Fortunately, there is excellent enterprise backup software that does everything for you with nice GUIs. For example Veeam is very popular, but you can use Microsoft Azure backup, IBM''s Spectrum Protect (a.k.a. TSM), VMware Data Protection (for vSphere environments), Proxmox (FOSS), and others.

Consumables Planning (*Budgeting*)

Suppose a medium to large organization uses 8 backup tapes a day, 6 days a week, means 48 tapes. If your retention policy is to keep 6 months' worth of incrementals, that's 1,248 tapes needed. High capacity DLT tapes might go for $60, so you would need $74,880.00. In the second part of the year, you only need new tapes for full backups, an additional 260 tapes (say) for $15,600, or more than $90k for the first year ($7,500 per month). (Not counting spares or the cost of drive units.) Changes to the policies can result in expense differences of over $1000 per month!

As backup technology changes over the years, it is important to keep old drives around to read old backup tapes when needed. You should keep old drives around long enough to cover your data retention policies.

Try to avoid upgrading your backup technology (drives, tapes, software) every few years, or you'll end up with many different and incompatible backup tapes. Note the budget must include amortization of the drive expenses.

Tools for Archives and Backups

Archives are easier to make than backups, so most tools create archives. A tool cannot make a "backup" without knowing the underlying filesystem intimately, i.e. it must parse the filesystem on disk. The reason is twofold:

- Different filesystems exhibit different semantics. No single tool supports all the semantics of all filesystem types. You need a different backup tool per FS type.
- The kernel interface obfuscates information about the layout of the file on disk. You have to go around the kernel, direct to the device interface, to see all the information about a file that is necessary for recording it correctly.

If you want to store the kernel's view of files along with all of the semantics the filesystem provides and none of the non-filesystem objects that might appear to inhabit the filesystem (such as sockets or `/proc` entries), **use the native `dump` program (and `restore`)** provided by your vendor specifically for that purpose (whatever they name it), for your filesystem type (note for Reiser4Fs you can just use `star`). `dump` uses `/etc/dumpdates` to track dump levels (that is, `dump` supports differential backups). Some of the differences between `dump` (for backups) and `tar`, `cpio`, or `pax` for archives are:

1 `dump` is not confused by object types that the particular operating system has defined as extensions to the standard filesystem; it also does not attempt to archive objects that do not actually reside on the filesystem, e.g. *doors* and *sockets*. Consider what GNU `tar` does to UNIX-domain sockets: it archives them as named pipes. They are not on the filesystem, so they should not be archived at all. `dump` handles this situation correctly.

2 `tar` ignores extended attributes (and ACLs unless you use the `--acls` or the `--xattrs` option when creating, adding to, or extracting from, the archive), while a native dump program will correctly archive them. (A new extensible backup format known as *pax* will archive ACLs, SELinux labels, and other meta-data stored in *extended attributes*. A tool called **`star`** uses a similar format. Find out about `star` on the web.)

3 `tar` cannot detect reliably where *holes* are. `dump` is not confused by files with holes (such as `utmp`); it will dump only the allocated blocks and `restore` will reconstruct the file with its original layout.

4 `tar` uses normal filesystem semantics to read files. That means it modifies the access times recorded in the filesystem *inodes*, when extracting files. This effectively deletes an **audit trail** which you may require for other purposes. (Modern Gnu `tar` has extra options to handle this correctly.) `dump` parses and records the filesystem outside of kernel filesystem semantics, and therefore doesn't modify the filesystem in the process of copying it.

Not all filesystem types support `dump` and `restore` utilities. When picking a filesystem type, keep in mind your backup requirements.

GNU `tar` is a popular tool for archiving the user's view of files. Another choice is `cpio`, rarely used anymore. Note neither tool is standardized by POSIX. A new standard tool, based on both (and hopefully better than either) is **`pax`**. These, combined with `find` and some compression program (such as `gzip`) are used to make portable archives.

Another choice is to use PAR files. PAR (Parchive) files use parity and checksumming to support multi-volume archive files with integrity checking. If one of the volumes in your archive set is corrupted, it can be repaired. This is generally not the case with other archive formats. PAR2 is current and PAR version 3 is in the works (as of 2021). While standard tools such as tar and pax cannot use these formats, there are many tools for Linux and Windows that can. (However, all such tools I've looked at are old and seemingly unmaintained.)

You can ask `find` to locate all files modified since a certain date and add them to a compressed `tar` archived created on a mounted backup tape drive. A backup shell script can be written, so you don't end up attempting to backup `/dev` or `/proc` files. (**See backup script on web page.**) (Note! Unix `tar` ≠ GNU `tar`;

use the GNU version. Unix `tar` doesn't handle backups that require multiple tapes.)

For either backups or archives, use `crontab` (or systemd timers) to schedule backups according to the *backup schedule* discussed earlier. (Show `ls -d /etc/*cron*`.) If your company prefers to have a human perform backups, remember that `root` permission will be needed to access the full system. Often the backup program is controlled by `sudo` or a similar facility, so the backup administrator doesn't need the root password.

The `find` command can be used to locate which files need to be backed-up. Use "`find / -mtime -`*x*" for incrementals and differentials to find files changed since *x* (you can store *x* as a timestamp on files, for example `/etc/last-backup.{full,incremental,differential}`). Use `find` with Gnu `tar` roughly like this:

```
mount /dev/removable-media /mnt
find / -mtime -1 -depth |xargs tar -rf --xattrs \
 /tmp/$$
gzip /tmp/$$; mv /tmp/$$.gz /mnt/incremental-6-20-01
touch /etc/last-backup.incremental
umount /dev/removable-media
```

(Instead of "`-mtime -1`" to mean less than 24 hours ago, you can use "`-newer` *x*", where *x* is some file.)

Commercial software is affordable and several packages are popular for Unix and Linux systems, including "BRU" (www.BRU-Pro.com), VERITAS, Seagate's BackupEXEC, and "Arkeia" (www.arkeia.com). (I haven't used these; I just use `tar` and `find`.)

Of course, there are free, open source choices as well, such as KDE `ark`, or `amanda` (network backups). One of the best is BackupPC. Another is Bacula (or Bareos, a fork of the original), which is popular (See the Bacula getting started guide for more information). Some of these can create schedules, label tapes, encrypt tapes, follow media rotation schedules, etc.

> Be careful of *bind mounts* and *private mounts* when performing backups, especially when using home-grown scripts that use `find`, `tar`, etc.! Tools such as Bareos will detect symlinks and bind mounts and not "descend into" those, but not all tools will (or may not by default). Bareos backs up the symlink, not duplicates of the files.
>
> In addition, bind mounts are only known to the kernel, in RAM, and are never backed up; you need to list those in `fstab` to restore those "views".

Finally, if you don't run the backup with root privilege, you may only back up the *polyinstantiated* (a per-user "private view") part of a directory that the process can see.

The most important tool is the documentation: the backup strategy, media types and rotation schedule, hardware maintenance schedule, location of media storage (e.g., the address of the bank and box number), and all the other information discussed above. This information is collectively referred to as the ***backup policy***. This document should clearly say to users what will be backed up and when, and what to do and who to contact if you need to recover files.

Note: Whatever tools you use, make sure you test your backup method by attempting to use the recovery procedure. (I know someone who spent 45 minutes each working day doing backups for years, only to realize none of the backups ever worked the first time he attempted to recover a file!)

(Parts of this section were adopted from Netnews (Usenet) postings in the newsgroup "`comp.unix.admin`" during 5/2001 by Jefferson Ogata. Other parts were adopted from *The Practice of System and Network Administration*, by Limoncelli and Hogan, ©Addison-Wesley.)

Backups with Solaris Zones and Other Containers — *Skip*

Solaris zones, Docker containers, and similar technology contain a complication for backup: many standard directories are actually mounted from the global zone via LOFS (loopback filesystem). These should only be backed up from the global zone. The only items in a local zone needing backup (usually) are application data and configuration files. Using an archive tool (such as `cpio`, `tar`, or `star`) will work best:

```
find export/zone1 -fstype lofs -prune -o -local \
| cpio -oc -O /backup/zone1.cpio
```

Whole zones can be fully or incrementally backed up using `ufsdump`. Shut down the zone before using the `ufsdump` command to put the zone in a quiescent state and avoid backing up shared file systems, with:

```
global# zlogin -S zone1 init 0
```

Solaris supports filesystem snapshots (like LVM does on Linux) so you don't have to shut off a zone. However, it must be *quiesced* by turning off applications before creating the snapshot. Then you can turn them back on and perform the backup on the snapshot: Create it with:

```
global# fssnap -o bs=/export /export/home #create
snapshot
global# mount -o ro /dev/fssnap/0 /mnt # then mount it.
```

You should make copies of your non-global zones' configurations in case you have to recreate the zones at some point in the future. You should create the copy of the zone's configuration after you have logged into the zone for the first time and responded to the `sysidtool` questions:

```
global# zonecfg -z zone1 export > zone1.config
```

Adding a backup tape drive

Added SCSI controller (ADAPTEC 2940)
Added SCSI DDS2 Tape drive
On reboot kudzu detected and configured SCSI controller and tape device
Verify devices found with '`dmesg`': indicate tape is `/dev/st0` and `/dev/nst0`
Verify SCSI devices with '`scsi_info`' (`/proc/scsi`)
Verify device working with: `mt -f /dev/st0 status`
Create link: `ln -s /dev/nst0 /dev/tape`
Verify link: `mt status`

Note: `/dev/st0` causes automatic tape rewind after any operation, `/dev/nst0` has no automatic rewind, but most backup software knows to rewind before finishing. If you plan to put multiple backup files on one tape, you must use `/dev/nst0`.

Common Backup and Archive Tools:

`mt` (`/dev/mt0`, `/dev/rmt0`)

`st` (`/dev/st0`, `/dev/nst0` - use `nst` for no auto rewind)

`mt` and `rmt` (remote tape backups); use like: **`mt -f /dev/tape`** ***`command`***, where command is one of: `rewind`, `status`, `erase`, `retention`, `compression`, (toggle compression on/off), `fsf` *`count`* (skip forward *count* files), `eod` (skip to end of data), `eject`, ...

`dump`/`restore` (These operate on the drive as a collection of disk blocks, below the abstractions of files, links and directories that are created by the file systems. `dump` backs up an entire file system at a time. It is unable to backup only part of a file system or a directory tree that spans more than one file system.)

`tar`, `cpio`, `dd`, `star` (and `pax` and `spax`)

A comparison of these tools (Note Gnu tar has stolen most of the good ideas from the other tools, accounting for its popularity):

- `cpio` has many more conversion options than `tar` and supports many formats, but is a legacy tool rarely used today. Gnu `cpio` does not support the `pax` format. (Use "`-H ustar`".) The default format used has many problems on modern filesystems, such as crashing with large inode numbers

- `cpio` can be used as a filter, reading names of files from stdin. (Gnu `tar` has this ability too.)
- On restore, if there is corruption on a tape `tar` will stop at that point. `cpio` will skip over corruption and try to restore the rest of the files.
- `cpio` is reportedly faster than `tar` and uses less space (because `tar` uses 512-byte blocks for every file header, `cpio` just uses whatever it needs only).
- `tar` supports multiple hard links on FSes that have 32-bit inode numbers, but `cpio` can only hand up to 18 bits *in the default format* (which can be changed with "`-H`").
- `tar` copies a file with multiple hard links once, `cpio` each time.
- Gnu `tar` can support archives that span multiple volumes; `cpio` can too but is known to have some problems with this.
- Modern `tar` (**`star`**) supports extended attributes, used for SELinux and ACLs. `cpio` doesn't *in the default format*.
- **`pax`** is POSIX's answer to `tar` and `cpio` shortcomings. `pax` attempts to read and write many of the various `cpio` and `tar` formats, plus new formats of its own. Its command set more resembles `cpio` than `tar`; with `find` and piping, this makes for a nicer interface. To use extended attributes (including SELinux labels) and ACLs, the "`pax`" archive format (or equivalent) must be used and not all systems support this (POSIX required) format. Check available formats with "`pax -x help`". Use `star` or `spax` instead, if necessary. (Note the LSB requires both `pax` and `star`.)
- `dd` is an old command that copies and optionally converts data efficiently. It can convert data to different formats, block sizes, byte orders, etc. It isn't generally used to create archives, but is often used to copy whole disks or partitions (to other disks/partitions when the geometry is different), copy large backup files, to create remote archives ("`tar ...|ssh ... dd ...`"), and to copy and create image files. (The command was part of the ancient IBM mainframe JCL utility set (and has a non-standard syntax as a result); no one knows anymore what the name originally meant.)

  There are two caveats to using `dd` to copy disks (or disk images): If the source and destination drives have different sector sizes (e.g. 512 and 4096 bytes per sector), using `dd` won't work well because partition tables (MBR or GPT) contain positions and sizes in sector counts; you'd need to manually overcome that somehow. Secondly, unless the source and destination drives have the exact same size, the GPT backup partition table won't be copied to the right location (the very end) on the destination drive; the extra space

after that won't be usable. Using tools such as `gparted` can account for such issues.

> `libarchive` is a portable library for any POSIX system, including Windows, Linux, and Unix, that provides full support for all formats. Currently it includes two front-end tools built with it: `bsdtar` and `bsdcpio`. These will support extended attributes and ACLs, when used with the correct options and with the "`pax`" (and not the default "`ustar`") format.

There are two variations of standard `dd` worth mentioning. Gnu ddrescue is designed to rebuild files from multiple passes through a corrupted filesystem and from other sources. dcfldd is designed to make verifiable copies, suitable for use as evidence. (The name comes from the Department of Defense Computer Forensics Lab, where the tool was invented.)

> Some examples of `cpio` and `pax` will be shown below. `dd` and `tar` were discussed in a previous course. Google will show many tutorials for all these commands, if needed; the man pages are only good for reference.

Additional Points

If you need to backup large (e.g., DB) files, use a larger blocksize for efficiency.

Many types of systems can use LVM, ZFS or some equivalent that supports snapshots for backup without the need to taking the filesystem off-line.

NAS (and some SANS) systems are commonly backed up with some tool that supports **NDMP** (the *Network Data Management Protocol*), which usually works by doing background backup to tape of a snapshot. This has a minimal effect on users of the storage system.

If you need to copy file hierarchies (e.g., your home directory and all subdirectories), one popular (and good) way is to use `tar`, like so:

```
    tar --xattrs -cf - some_directory | \
ssh remote_host 'cd dir && tar --xattrs -xpf -'
```

You can do this with `pax` as easily:

```
cd dir; pax -w -x pax . | \
ssh user@host 'cd /path/to/directory && pax -r -pe'
```

> To ensure the validity of backups and archives, you should compute and compare checksums. Here's one way when using `tar`:
>
> **`tar -cf - dir |tee xyz.tar |md5sum >xyz.tar.md5`**

Many archiving tools ignore extended attributes (and hence, ACLs and SELinux labels). Backup and restore by saving ACLs (or all extended attributes) to a file, then applying the file after a restore, as follows:

Backup: `cd dir; getfattr -R --skip-base . > backup.attrs`
use normal backup tool here
Restore: `cd dir;` *use normal backup tool here*
`setfattr --restore=backup.attrs`
`rm backup.attrs`

Or use an archiving tool that supports extended attributes, ext* attributes, and ACLs: **`star H=exustar -xattr -c`** ***`path`*** **`>`*****`archive`*****`.tar`**
and to restore use: **`star -xattr -x <`*****`archive`*****`.tar`**

(The POSIX standard tool `pax` can support this, but only when using the non-default archive type "`pax`". Run "`pax -x help`" to see if the `pax` archive type is available on your system. Modern Gnu `tar` has "`--xattrs`" option to use with the `-c` or `-x` options; using this forces the pax archive type.)

Additional Tools

Jörg Schilling's **`star`** program currently supports archiving of ACLs and extended attributes. IEEE Std 1003.1-2001 ("POSIX.1") defined the "pax interchange format" that can handle ACLs and other extended attributes (e.g., SELinux stuff). Gnu tar handles `pax` and `star` formats. There is also a **`spax`** tool that supports the `star` extensions.

Another tool that supposedly easily and correctly backs up ACLs, ext2/3 attributes, and extended attributes (such as for SELinux) is "**`bsdtar`**", a BSD modified version of `tar` that uses `libarchive.so` to read/write a variety of formats.

`Amanda` (a powerful network backup utility, producing backup schedules automatically but relying on other tools for the actual backups. Most other tools don't support schedule creation.)

BackupPC (an "enterprise-grade" utility)

Bacula (works very well and is popular but has a steeper learning curve than most. See also Bareos, a fork of Bacula by some of the original developers.)

`bru` (commercial sw)

`Clonezilla` **(**similar to commercial Ghost)

`HP Data Protector` (commercial sw, used at HCC)

Mondo Rescue

`unison` (uses `rsync`)

LuckyBackup (similar to Unison)

`vranger` (commercial sw, designed for VMware backups, from quest.com)

`foremost`, `ddrescue`, ... These are not backup tools, but recovery tools when a filesystem is corrupted and you need to salvage what you can.

Duplicity (Uses rsync to create encrypted tar backups.)

Rsnapshot (A wrapper around rsync. Rsync is not designed for backup, but can be used for that in some cases.) This tool makes a copy of any files modified since the last snapshot (which takes a lot of disk space), and makes a hard link to any unmodified files.

rdiff-backup (stores meta-data in a file, so can easily restore files to alternate systems. Usually produces smaller backups than Rsnapshot, is easier to use, but is slower.) This tool stores the diff between the newest version and the previous version. Thus, restoring a very old version can take a long time. This tool also can provide useful statistics.

Storix (supports AIX & Linux).

s3ql (backs up to Amazon's S3 cloud).

grsync (GUI for rsync).

Other tools can be used to backup (or migrate) data across a network, including `tar` pipped through SSH, BitTorrent (can use multiple TCP streams at once), and others.

cpio examples:

```
cd /someplace/.. # the parent of "someplace"
find someplace -depth \
 | cpio -oV --format=crc >someplace.cpio
# crc=new SysVr4 format with CRCs
```

Note that when creating an archive, "`-v`" means to print filenames as processed; "`-V`" means to print a dot per file processed.

```
# Restore all; -d means to create directories if needed;
#             -m means to preserve modification timestamps:
cpio -idm < file.cpio
```

```
# Restore; wildcards will match leading dot and slashes:
cpio -idum glob-pattern <file.cpio
```

Without the `-d` option, `cpio` won't create any directories when restoring. Without the `-u` option, `cpio` won't over-write existing files. Add `-v` to show files as they are extracted (restored).

```
cpio -tv < file.cpio # table of contents (-i not needed but allowed)
```

Command to backup all files:

```
find . -depth -print | \
  cpio -o --format=crc > /dev/tape
```

Command to restore complete (full) backup:

```
cpio -imd < /dev/tape
```

Command to get table of contents:

```
cpio -tv < /dev/tape  # -v is long listing
```

pax Examples (Note you need "-pe" when writing and reading)**:**

```
# List contents [matching patterns]:
pax -f [-v] files.pax [pattern...]

# Create archives:
find -depth ... | pax -w -pe -x pax > pax.out
pax -w -pe -x pax -f files.pax path... # recursive

# Extract from archive:
pax -r [-v] [pattern...] < files.pax
pax -r [-v] -pe < pax.out # -pe means preserve everything
```

(**spax** also has `-acl` option.)

> **Avoid using absolute pathnames in archives.** `tar` strips out a leading "/" but `cpio` and `pax` do not!
>
> Some versions of pax on Linux do not fully support "`-x pax`" (such as on Fedora currently); **use "`-x exustar`" instead**.

To duplicate some files or a whole directory tree requires more than just copying the files. Today you need to worry about ACLs, extended attributes (SELinux labels), non-files (e.g., named pipes, or FIFOs), files with holes, symlinks and hard links, etc. Gnu `cp` has options for that but cannot be used to copy files between hosts.

The best way to duplicate a directory tree on the same host is Gnu `cp -a`, or if that is not available use `pax`:

```
pax -pe -rw -x pax olddir newdir
```

To copy a whole volume to another host you can use `dump` and then transfer that, then restore it on the remote system.

`Tar` or `pax` is often used to duplicate a directory tree to the same host if Gnu `cp` isn't available. These tools can also be used to duplicate a directory tree to a different host, via `ssh`:

```
tar czf - -C sourcedir files \
| ssh remote_host 'tar xzf - -C destdir'
```

Use `tar` with `ssh` if this is a complete tree transfer. For extra performance, use different compression (e.g., "`-j`" for `bzip2` or `-I lz4` or `--zstd`). You may need extra options to control what happens with ACLs, links, etc.

Using `rsync` over `ssh` often performs better than `tar` if it is an update (i.e., some random subset of files need to be transferred). **(Show `ybsync` alias on `wpollock.com`.)**

(**Show `backup-etc` script.**)

Files or backups and archives can be copied between hosts with `scp` or `rsync`.

Using `scp`

`scp` is a simple way to copy files securely between hosts, using SSH. The syntax is simple:

```
scp file... user@host:path
```

For example, to copy the file foo to the home directory of ua00 on YborStudent, use the command "`scp foo ua00@yborstudent.hccfl.edu:.`" (Note that a relative *path* is relative to the remote user's home directory.)

You can copy in either direction, and `scp` has many options to control the attributes of the copied file; see the man page for details.

Using rsync

`rsync` is a versatile tool that does incremental archives, either locally or across a network. It can also be used to copy files across a network. `rsync` has about a zillion options but is worth learning. To understand its options (and diagnose performance problems) you need to understand how `rsync` works.

First, `rsync` reads last modified timestamp and the length of both the source and destination files. If they are the same, it does not transfer anything. If either is different, `rsync` reads both the source file and the destination file, and performs progressively smaller hashes on them, to determine which parts differ. After this analysis, `rsync` copies the similar parts of the destination file to a new file, and then copies the different parts from the source file and inserts them into the proper places in the new file. Finally, `rsync` copies the updated new file to the destination file location. Note the timestamp of the new file is the current time, not the time of the source file; usually you will need to include the "`-t`" option to update the destination file's timestamp to match the source's timestamp.

You can speed up `rsync` by eliminating the new file and updating the destination file in-place. However, that is dangerous if your network connection is unreliable. It does preserve hard links though.

`rsync` uses compression to reduce bandwidth use and make the transfer faster. You can use SSH to make the transfer secure.

You can run `rsyncd` as a daemon on the remote end (port 873). This makes the transfer go faster (no need to fork `rsync` each time, and have it calculate file lists each time), more controllable (via a config file), and allows you to push files to (say) a Windows disk that has a different layout and paths. As a server daemon, `rsync` acts like a modern, super-charged FTP server (and is often used to provide mirror sites on the Internet). You don't get SSH security however.

A serious performance concern is when copying files from different systems. The timestamps may not match, causing rsync to unnecessarily copy the whole file. This can happen when two systems have clocks that are slightly off. It can also happen **when using filesystems with different *granularity* for the timestamps**. For example, most Flash drives use FAT, which records timestamps only to within two seconds. You need to use the `rsync --modify-window=1` option in that case, to have `rsync` treat all timestamps within one second as equal.

The syntax is "`rsync` *`options source destination`*". Either the source or destination (but not both) can be to remote hosts. To specify a remote location, use "[*user@*]*host*:*path*". A relative *path* is relative to the user's home directory on *host*. (Filenames with colons can cause problems, not just with `rsync`. Colons after slashes work fine, so use "`./fi:le`" instead of "`fi:le`".)

Note! If *source* is a directory, **a trailing slash affect how `rsync` behaves**:

```
rsync -a source host:destination
```

will copy the directory *source* into the directory *destination*, so *source/foo* becomes *destination/source/foo*. On the other hand:

```
rsync -a source/ host:destination
```

copies the contents of *source* into the directory *destination,* so *source/foo* becomes *destination/foo*.

One of the reasons to use `rsync` over `tar` is the control it provides over what to copy. You can include or exclude files and directories. For example, to exclude all files or directories (and their contents) named `.git`, add "`--exclude=.git`". (If you have files with that name and only want to skip the directories, add a trailing slash.) To exclude just one specific directory, you must specify an absolute pathname. Similarly, use "`--exclude=*.o`" to skip all files ending in "`.o`". You can use glob patterns with some extensions. This can be complex; see the rsync man page or this summary. Note that unlike shell globs, there is a different meaning between patterns with trailing "`*`", "`**`", and "`***`".

If your "`--exclude`" options (you can specify this multiple times) skip too much, you can add "`--include`" options to re-include them.

The *archive* ("`-a`") option is a shorthand for several others. It means to preserve permissions, owner and group, timestamps, and symlinks. The "`-z`" option enables compression. The "`-R`" option copies pathnames, not just the filenames. The option "`-u`" says don't copy files if a newer version exists at the destination.

If you want *destination* to be an exact copy of *source*, you also want to delete any files on *destination* that weren't present in *source*; add the `--delete` option for this.

To make a backup of `/home` to `server.gcaw.org` with `rsync` via `ssh`:

rsync -avre "ssh -p 2222" /home/ server.gcaw.org:/home
rsync -azv me@server.gcaw.org:documents documents
rsync -azv documents me@server.gcaw.org:documents

rsync -HavRuzc /var/www/html/ example.com:/var/www/html/
or copy ~/public_html to/from me@example.com:public_html/

```
rsync -r ~/foo ~/bar # -r means recursive
rsync -a ~/foo ~/bar # -a means archive mode
rsync -az ~/foo remote:foo # copies foo into foo
rsync -az ~/foo/ remote:foo # copies foo's contents
rsync -azu ~/foo/ remote:foo #don't overwrite newer
files
```

The meanings of some commonly used options are:

-v = verbose,
-c = use MD4 (not just size and last-mod time) to see if dest file different than src,
-a = archive mode = -rlptgoD = preserve almost everything,
-r = recursive, -R = preserve src path at dest, -z = compress when sending,
-b = backup dest file before over-writing/deleting,
-u = don't over-write newer files,
-l = preserve symlinks, -H = preserve hard links, -p = preserve permissions,
-o = preserve owner, -g = preserve group, -t = preserve timestamps,
-D = preserve device files,
-S = preserve file "holes",
--modify-window=*X* = timestamps match if diff by less than *X* seconds
(required with FAT, which only has 2 second time precision)
--delete = remove files from destination not present in source,
-z = compress data at sending end, and decompress at destination

Some other options include using checksums to validate the transfer, renaming existing files at destination (with a trailing "`") rather than overwrite them, and a bandwidth limiting option so your backup doesn't overwhelm a LAN.

Modern `rsync` has many options to control the attributes at the destination. You can use `--chmod`, xfer ACLs and EAs. You can create `rsyncd.conf` files, to

control behavior (and use a special SSH key to run a specific command), and define new arguments via `~/.popt`. But older `rsync` versions don't have all those features. For example, old `rsync` had no options to set/change the permissions when coping new files from Windows; `umask` applies. (You can use special `ssh` tricks to work around this, to run a "`find ... |xargs chmod`..." command after each use of `rsync`.)

A good way to duplicate a website is to set a default ACL on each directory in your website. Then all uploaded files will have the `umask` over-ridden:

```
  cd ~/public_html # or wherever your web site
is.
  find . -type d -exec xargs setfacl -m d:o:rX {}
+
```

This ACL says to set a default ACL on all directories, to provide "others" `read`, plus `execute` if a directory. (New directories get default ACL too.) With this ACL, uploading a file will have 644 or 755 permissions, rather than 640 or 750.

Lecture 2 — Managing System Security — Concepts

Setup Security

There are many security mechanisms present on your system, any one of which might block some application or service from working as expected. In addition to securing your systems, a system admin must know about these different mechanisms so they can be configured properly, and to perform trouble-shooting when something isn't working.

In this section, we will cover (briefly) many security concepts and mechanisms that all sys admins must know.

Security involves protection, detection, and reaction. If there's no alarm, then sooner or later the protection will be overcome; if there's no reaction to alarms, you needn't bother. Protection is only needed for the time it takes to react.

Always have multiple layers of protection, so that if one is compromised there are others that still provide protection. Multiple layers also slows attackers, so you have a chance to stop them. Examples: use proper permissions on files, but also don't allow root to connect remotely to your machine. Use a firewall, but also turn off unnecessary services; and use *TCP Wrappers* to control access as well. (Show ringed-fort of Dún-Aonghasa.)

> In fact, this is how safes are rated: A TL-30 safe means it will resist a knowledgeable attacker with tools for 30 minutes; a TRTL-60 save means to resist that attacker with tools and an oxy-acetylene torch for an hour. You buy the safe that you need, depending on (say) security guard schedules. Computer systems' security is the same. You must have alarms and monitoring.

Define ***rings (levels) of security***: Also called *multi-layer* security, or sometimes *defense in depth*. (**Defense in depth** refers sharing the security burden over many parts of a system, rather than having a single system manage security.)

> **The balancing of costs and protections makes security an exercise in risk management.** First you need to determine what needs protection, and what the threats are that you plan on defending against. This is a *threat assessment* (and produces a *threat matrix*). Next you define *security policy* that, if implemented correctly, will reduce the risk of the identified threats, to an acceptable level. The policy is implemented using various *procedures* and *security mechanisms*.

Don't install any software you don't need. Remove unnecessary software if installed by default. (In some cases you cannot uninstall some service as it is part of a package of several services and you need one or more of them. In this case, disable such services.)

Never attach a newly installed machine to an untrusted network until after it has been ***hardened***.

Periodically (or continuously) run system and network auditing tools, which scan your system and report (or even fix) security problems, including Lynis, OpenVAS, rkhunter, Nikto2, scap-workbench, Saint (saintcorporation.com, a commercial product with trial version). Also, run system and network monitoring tools such as Nagios (nagios.org), and *intrusion detection systems* such as `ipcop` (ipcop.org). Only after running such a scan, and fixing all identified security issues, is your system considered *hardened*.

> SCAP is my favorite tool. It is collections of guides and checklists for various OSes and daemons, in a machine-readable XML format. This content can be used to automatically run security audits on your system. The various checks include info on how to remedy the issues. These checks are organized into *profiles* from an XML file in XCCDF format. Because the checks are in XML format, users can edit or add to the checks locally (this is very hard however).
>
> I like the SCAP-Workbench tool (GUI). You can also use the `oscap` CLI tool.

Trust relationships between systems: *If user U has accounts on systems s1 and s2, s2 will trust s1 to authenticate U and allow U access from S1 without further authentication*. This trust was useful for isolated LANs. The trust mechanisms include BSD style *rhosts*: A user can create `~/.rhosts` that lists trusted systems. Special versions of common commands used this: `rsh`, `rlogin`, `rcp`, ... Other similar trust mechanisms were system-wide; the file `/etc/hosts.equiv` listed trusted hosts. If a user logged in from one of the listed hosts with the same username, they wouldn't need to authenticate. Limited trust is possible too: `/etc/hosts.lpd` and `lpd.perms` were used to control printer access.

These mechanisms are too dangerous to use on modern networks. Avoid them and use ssh/VPN instead. Modern systems may use Kerberos to define a group of hosts with some trust relationships (such a group of hosts is called a *domain*).

Firewalls can prevent access from non-local users and hosts. (*Offer a quick review of IP addresses, port numbers, and protocols e.g. TCP and UDP.*) A common type is a ***packet filter***, where the administrator specifies rules that say which packets should be dropped and which should be delivered. (There are other types of firewalls too.) Every host should be running one.

Note that a secure setup uses ***deny by default*** rules, with explicit rules to say what to allow. Many times, systems have *allow by default* enabled. This means an

administrator needs to update the firewall every time a service is added to (or dropped from) the system.

For Linux, the packet filters **`iptables`** (for IPv4; `ip6tables` is similar for IPv6) is used until recently (`iptables -vL; systemctl status iptables.service`) and the more modern **firewalld** are common. For Unix, there are several different popular ones. Newer *dynamic* packet filters were developed after 2012 for both Unix and Linux that are replacing `iptables` and netfilter for Linux, such as nftables and eBPF.

Firewalld

`Firewalld` is a newer, *dynamic* packet filter firewall daemon for Linux. The idea is that you can update the firewall without a complete restart of it. (Something `iptables` cannot do.) A more useful difference is that the single set of rules works for IPv4 and IPv6. Furthermore, `firewalld` comes with a CLI tool(s) as well as a GUI tool. Finally, it can use older kernel packet filter code (netfilter) or newer code (**`nft`** and and eBPF) as a back-end. (As of 2020, it uses nftables.)

`Firewalld` replaced `iptables` and `ebtables` in Fedora by default, although you can certainly turn it off and install then enable `ip*tables` instead if you wish, or give the new `nft` command line tool a try. Unlike `iptables`, `firewalld` uses a daemon named `firewalld`. That daemon must be reloaded/restarted when you make changes to the configuration.

> Like `ip*tables-save` and `ip*tables-restore`, `firewalld` stores the firewall rules in a file between reboots. But it doesn't use the same file as the iptables tools, so make sure you save modified firewall rules correctly!

Your computer connects to various networks through the attached NICs. With `firewalld`, you can group various networks that your NICs are directly attached to, together in ***zones***. Then you can specify different firewall rules for different zones.

To see a list of zones and which NICs are included in them, use the command `firewall-cmd --get-active-zones`. This should show (by default) a single zone named `public`, containing all NICs except for "`lo`". There are other zones defined by default, but they are not ***active*** (not used). Use `--get-zones` to see them.

To see the rules for a zone, use the command "`firewall-cmd --list-all`". If you create other zones, you can see their rules using `--list-all-zones`.

By default, `firewall-cmd` options only apply to a single zone which you specify on the command line. To save typing, you can set one zone as a ***default zone***. To apply some command to some other zone, add the "`--zone=`*`name-of-zone`*" option. The default zone can be set using a command; the value is in `/etc/firewalld/firewalld.conf`. To see the default zone, use the command "`firewall-cmd --get-default-zone`".

> Several (9) zones are pre-created for you with default rules, for use in coimmon situations such as "home" and "public". These default zones are defined by XML files in `/usr/lib/firewalld/zones`. To modify one or create a new one, either use the CLI or GUI tool("`--new-zone=`*`name`*"), or you could copy one from there to `/etc/firewalld/zones` and modify it with an editor.
>
> To put a NIC's networks in a non-default zone, add "`ZONE=`*`zonename`*" to the correct config file (such as `/etc/sysconfig/network-scripts/ifcfg-p2p1`), run the command "`firewall-cmd --permanent --zone=home --change-interface=p2p1`", or you can use `nm-connection-editor` to change the zone. If you don't do this, all discovered NICs (and their networks) go into the default zone.

Changes to your firewall setup are made using `firewall-cmd` (command line tool) or `firewall-config` (GUI). **Any changes made are only temporary**, like with `iptables`. Also, some changes (such as adding new zones) don't take effect immediately; you must reload the config ("`--reload`" option).

To make the changes permanent, add the "`--permanent`" option when making changes. Note this will update the config files and the changes do not take affect immediately. Usually you will need to enter commands twice, both with and without the `--permanent` option.

`Firewalld` knows what ports are needed for many popular services. To allow one of those, use two rules like this:

```
sudo firewall-cmd --add-service=http
sudo firewall-cmd --permanent --add-service=http
```

To modify firewall rules for services not listed, you can either define a new service and then add it to a zone, or simply open up specific ports. For example:

```
sudo firewall-cmd --add-port=12345/tcp
sudo firewall-cmd --permanent --add-port=54321/udp
sudo firewall-cmd --add-port=5000-5006/tcp #range
```

The default service definitions are XML files found in `/usr/lib/firewalld/services`. To modify one, or to add a new one, copy one from there and put it into `/etc/firewalld/services`.

TCP Wrappers is related to firewalls and can be used on top of a packet filter firewall to provide extra security. Unlike a packet filter, TCP Wrappers can restrict access by arbitrary restrictions such as the time of day or current system load. Examine `/etc/hosts.allow`, `/etc/hosts.deny`. Note that TCP Wrappers can be used for both on-demand services or stand-alone services (if compiled with the `libwrap.so` DLL; check by `lddtree` (from `pax-utils`)).

TCP Wrappers is not used by all services, so setting this to a default deny policy will not block all TCP services, just those compiled with `libwrap.so` (use `lddtree` to find out). Moreover, it doesn't block any UDP services. Red Hat (Fedora) consider TCP Wrappers a legacy system and most services are no longer complied with `libwrap.so`.

SASL (Simple Authentication and Security Layer) is used by some services to negotiate the specifics of security on the connection between clients and the services. For a service to communicate securely with some client, many parameters must be agreed upon. With SASL, the service declares which security mechanisms it will accept (and with which values for parameters), and the client picks from that list (or no connection is made). For example, a server might advertise SHA512 and argon2id as the only hashing mechanisms it will accept. If the client only supports MD5 and SHA256, no connection is made. If the client and server do have mechanisms in common, they are used. (The first one matched is used, so they are listed in preference order.) SASL is a simple way for sys admins to specify which mechanisms for the server to advertise; otherwise, an application-specific way must be used (some services use hard-coded parameters that cannot be changed).

Default permissions of standard directories and any added user accounts must be checked. Often these are not set as securely as possible, perhaps to make updates easier. (But if updates are easy for you, they are easy for attackers!) These are checked by some of the security auditing tools discussed previously, or you can use the find command. *Show* ***`find-world-writable`*** *script.*

Login account defaults must be configured (`/etc/login.defs` on Linux if using the shadow suite) and the tools `user{add,mod,del}`, including account and password expiration (aging) defaults. The `/etc/default/login` on Solaris applies no matter which password store is used. (Use `chage` (or `passwd` on Solaris) to view and change.)

> Note that on modern Linux, many account management tools use the `libuser.so` (DLL). This library has its own config file,

`/etc/libuser.conf`, and that may override settings from other files. Admins should make sure *all* account default settings files are configured with the same policy.

Add/edit/delete files from **`/etc/skel`** (the template used for new accounts) and set the correct permissions; change owner to root. This is best done by creating a fresh user, logging in, and configuring web browsers, email, etc. Then (as root) copy all the dot files from there to `/etc/skel`.

Password strength policies should be controlled by PAM. Although different systems use different PAM modules and use different names for the config file, on Fedora edit the options (or include if missing) for the `required` module `pam_pwquality.so` (compatible with the older `pam_cracklib.so`) in the file `/etc/pam.d/system-auth`. **Use the (non-standard) commands `pwgen`, `pwqgen`, and `apg` to generate strong passwords, and `pwscore` to check password quality.** Note that some systems set these defaults through other means. (PAM is discussed further, below.)

Change default passwords; install and use `pwgen`, `apg`, or pwqgen (passphrases). Safe passphrases are: 3-5 random words, separated by symbols and digits. **Good passwords**: first letter of words of a quote or poetry, plus symbols or digits.

Remote access should be set up using **`ssh`** (`/etc/ssh/sshd_config`, ...). This includes `scp` and `sftp` access. Disable `telnet` and `FTP`. Make sure **remote root access is disabled**. Make sure remote users cannot use `sudo` to run any command, or at least be sure that user has a very hard to guess password (or uses keys). Remote access should require keys, not passwords. (Local logins using passwords are usually safe though.)

While we don't go into OpenSSH configuration in this course, one "footgun" to know is that, unlike most configuration files, `sshd_config` uses a *first mention wins* policy; once you set something, latter assignments to the same setting are ignored! If you make any changes to the defaults, be sure to add them to the top of the configuration file. (If using `sshd_config.d/*` files, make sure any added file will be used before any existing default files.)

SSSD is a framework used by many developers. It provides a single front-end to many different authentication and authorization systems. It also includes the ability to cache a user's credentials for a time (similar to `sudo`). If using any services that use SSSD, a sys admin needs to be able to configure SSSD to use the right back-end service (for example, Kerberos/Active Directory, LDAP, RADIUS, and so on).

Other security tasks include configuring PKI for server and service (web, email, ...) certificates, securing individual services (web, LDAP, database, printing, etc.), and configuring logging and monitoring. External systems may need to be configured as well, such as network routers and firewalls, Wi-Fi access points, etc.

(More security topics such as SELinux are discussed in more detail, later in the course.)

Security Organizations

cert.org and **us-cert.gov** (sign up for their weekly bulletins, or their RSS feed), cpni.gov.uk, sans.org, giac.org, **CISSP** (Certified Information System Security Professional, **cissp.com**), **isc2.org**, **EC-Council**, iapsc.org, IPSA, and LinuxSecurity.com. (Note Microsoft also provides bulletins for Windows.)

File Permissions

Review *inodes* (three time-stamps: *atime*, *mtime*, *ctime*), links (*hard* and *soft*), users, and groups (`/etc/group`). Review `mv`, `cp`, `rm`, `ln`, `touch` commands, and **`umask`**, **`chmod`**, **`chown`**, `chgrp`, commands. Also **`su`**, `su -` (Runs root's login scripts), `su [-] user`.

> `chown` (like many of these commands) has a *recursive* option of `-R`. For `chown`, this can be very dangerous is you don't include the `-h` option as well.

Review POSIX permissions: r, w, x, on files and directories, and suid, sgid, and text bits on files and directories. Discuss FilePermisions.htm doc, especially `SGID` on a directory (e.g., a web author's group). **Show `SUID` Demo**.

> While a powerful feature, **SetUID (especially to root) is dangerous and unnecessary** in many cases. Fedora has removed the SUID bit from most utilities, instead granting them just the specific rootly privileges needed.

Mention the `wheel` and `sysadmin` groups (members have *rootly* powers on some systems). Use of this on production systems should be discouraged, since such a user is the equivalent of root.

POSIX ACLs

Most systems support POSIX ACLs currently (2009). **Both the FS and kernel must support ACLs.** The FS must be mounted with `-o acl`. If a file has ACL entries the "`ls -l`" output includes a "+" after the permissions. You can view and set/change/remove POSIX ACL entries with:

`getfacl` ***`file ...`***
`setfacl` `[--test] -m` *`acl file...`*,

where: acl = `{u|g|o|m}::[rwxX]+`, '`m`' is mask (or all), and '`X`' (cap X) means execute *only if* directory or if some (other) user already has execute.

Ordinarily, a Unix file has an owning user and an owning group. The FS keeps permissions for each of these, plus an entry for all others. ACLs allow you to add permissions for additional users and groups. Examples:

```
setfacl -m u:ud00:r file  # add read for ud00,
setfacl -m d:u:ud00:r dir # set default ACL to read for ud00,
setfacl -x u:ud00 file    # remove ACL entry for ud00,
setfacl -m m::rx file     # remove write permission from all,
setfacl -b file           # remove all ACL entries from file,
u::rw-,u:ud00:rw-,g::r--,g:staff:rw- file # a complex example
```

File Attributes:

In addition to the standard permissions, some filesystems support file *attributes* or *flags*. (BSD's UFS does, Solaris UFS doesn't. Linux ext[234] filesystems do.) Use the commands: **`lsattr`**, **`chattr`**: `chattr +|-attributes`
Attackers often set these, especially "i" (immutable) on infected files.

Mandatory Access Controls — SELinux

POSIX (and NTFS) permissions are a type of security mechanism called ***discretionary access control***, or ***DAC***. With DAC, the owner of some object (usually, a file) determines who can do what with it. Each user and program run by that user has complete discretion over the user's objects. There is no protection against malicious or flawed software that is running as a normal or root user from doing anything it wants with those objects, even if those actions make no sense. For example, a web server should not need to set the system time, but it can if run as root (which is usually the case, at least initially). What's worse, with `setuid` and `setgid` processes the system cannot tell the true identity of a process.

What is needed is a policy that the system administrator can define, to specify which processes can do what actions on files, sockets, port numbers, etc. Sadly, no DAC system can enforce such a policy.

SELinux (Security-Enhanced Linux) is an implementation of ***Mandatory Access Control*** (***MAC***). A MAC system does define a system-wide policy, enforced by the kernel using the *Linux Security Modules* (LSM) framework. (There are other MACs available for Linux that can "plug-in" to the LSM framework in the kernel.) At boot time, the policy is loaded into the kernel. In high security situations, the policy is locked in and cannot be changed without a reboot. (Fedora uses a lower security policy, which can be changed by root without a reboot.)

> Note that a MAC does not replace the normal DAC; either one can prevent some action.

To begin with, each process has an ID to identify the user running that process. SELinux doesn't use UIDs since those can be changed. SELinux IDs can

only be changed according to the policy. That rarely happens. (A notable exception is the login process, which starts off as root but upon a successful login, transitions to that user's ID.) To see your shell's SELinux ID, run the command "`id -Z`".

Each process also has a label, a combination of the SELinux user ID and the process (executable) name. Such labels can be more general; this allows many processes to use the same label if they have similar security requirements (such as `cat` and `less`).

Next, **every object (socket, FIFO, file, network port, etc.) also has an unchangeable SELinux ID**. You can see those with "`ls -Z`". These IDs (process and object) are called ***labels*** (other terms such as *file context* or *fcontext* are sometimes used, just to confuse you).

The final piece is the policy, known as the ***type enforcement*** (TE) table. Think of this table as having one row for every process label; for example, user `ua00` running the `passwd` command is one row, `root` running `passwd` is a different row, etc. Also, every object label on your system has a column.

Each cell in the TE table defines what actions are legal by that process on that object. When a process tries to invoke some system call such (as read on a file), the TE table is consulted to see if the action is allowed.

After consulting the table, the decision is put into an ***access vector cache*** **(AVC)**; that greatly speeds up subsequent checks for the same access. If SELinux denies access, an "`avc`" audit message is generated and included in the audit log (or `/var/log/messages` if you don't use the audit system) at **`/var/log/audit/audit.log`**. The file may be large, so you should use the **`ausearch`** command (rather than `less` or even `grep`):

```
# ausearch -m avc -ts today
```

("`-m avc`" says to search for SELinux messages only; "`-ts`" says to only show messages from the given date and time (to the present unless you also specify "`-te`" for the ending time. See asusearch(8) for the syntax used with -ts and -te.)

The output may look something like this:

```
----
time->Sat May 20 19:00:12 2020
type=AVC msg=audit(1495926012.088:231): avc:  denied  {
read } for  pid=31578 comm="/usr/sbin/httpd"
name="moved.txt" dev="dm-0" ino=525088
scontext=system_u:system_r:httpd_t:s0
tcontext=unconfined_u:object_r:user_home_t:s0
tclass=file permissive=0
```

You can restrict the output to a particular executable with the "`-c`" option, as in:

```
# ausearch -m avc -ts today -c httpd
```

> You can also use **`aureport`**, which may produce output easier to read. However, `aureport` doesn't have all the filter options of `ausearch`. It is common to use `aureport ... | grep ...` to limit the output.

The only real problem with SELinux (or any MAC) is that when installed and enabled (*enforcing*), the policy may not be correct for the software you run. For example, by default the Apache web server `httpd` cannot read files other than from its configuration directory and the document root or use unusual port numbers. SELinux prevents user directories from having working `~/public_html` web sites. This gets tricky since the TE table is very fine-grained and thus complex. Also, if you move a file from your home directory into the document root, it likely has the wrong label. (The same is true when extracting files from archives.)

SELinux provides detailed log (audit) messages that state exactly what was tried and denied. That makes it fairly easy to update the policy to allow that action. To see the reason why something was denied, you can pipe the message into **`audit2why`**:

```
# ausearch -t avc -ts today -c httpd | audit2why
```

This will usuallly explain the problem and suggest a fix. You can have SELinux automatically create a policy rule to allow the error-causing situation as an exception:

```
# ausearch -t avc -ts today -c httpd |
    audit2allow -m local
```

The generated policy module can be compiled and then loaded (merged) into the current policy with `semanage` or `semodule` commands. **Adding policy modules is the easiest way to modify policy.**

In addition to modifying policy, some common situations were anticipated such as user directory web sites (`~/public_html`). Such files have a correct label, but not one the web server is allowed to read. For these sorts of situations, SELinux provides a set of **Booleans** you can turn on or off, to tweak the policy easily. To see them, run "**`getsebool -a`**". There can be hundreds of these, depending on what software you have installed. (You can set them with `setsebool`.)

For a list of Booleans, an explanation of what each one is, whether they are on or off, and their default state, run the **`semanage boolean -l`** command as root.

Since Booleans are per-package, there is no authoritative list with descriptions. (It is hoped the installed software's man pages will list and describe them.) In the

example given earlier, set the Boolean `httpd_enable_homedirs` to allow Apache to serve `~/public_html` web pages.

To see what port numbers processes are allowed to listen on, use the command **`semanage port -l`** (as root). For example:

```
$ sudo semanage port -l |grep '^http_port_t'
http_port_t                    tcp      80, 81, 443, 488, 8008, 8009, 8443, 9000
```

(You can also grep for a port number to see what process label can use it.)

If for some reason you reconfigured the web server to listen on, say, port 88 instead of port 80, it will be blocked by SELinux. You can use `semanage` to add or remove allowed ports from process labels, like so:

```
semanage port -a 88 -p tcp -t http_port_t
```

(Use "-d" instead to delete one.)

SELinux is controlled by the file `/etc/selinux/config`. This can be over-ridden on the GRUB (or any boot loader) command line with "`selinux=0`" (=1 for on).

SELinux includes a set of rules for labeling newly added files. If that is wrong, even if you change a file's label it will likely get reset according to the policy in the future. To fix that, you need to update the policy used for labeling files. This is done with the **`semange fcontext`** command (or by creating and installing a new policy module as discussed above).

To see a full listing of the available fcontext policies, run "`sudo semanage fcontext -l`". There are about 3-4 dozen; you can use `grep` to shorten the list. Suppose for example you decide to create a new directory `/etc/host-keys` to hold the host keys used by the SSHD daemon, instead of the default directory. Even after you update the SSHD configuration file to know about the new location of keys, SELinux will block access since the new directory has an incorrect label. The default location of server keys is under the directory `/etc/ssh`. Grep for that to see the label required:

```
sudo semanage fcontext -l | grep '/etc/ssh'
/etc/ssh/primes                    regular file      system_u:object_r:sshd_key_t:s0
/etc/ssh/ssh_host.*_key            regular file      system_u:object_r:sshd_key_t:s0
/etc/ssh/ssh_host.*_key\.pub       regular file      system_u:object_r:sshd_key_t:s0
```

```
# mkdir /etc/host-keys
# ls -dZ /etc/host-keys
unconfined_u:object_r:etc_t:s0 host-keys/
```

As you can see, the type label needed is "`sshd_key_t`" but the policy used the label "`etc_t`". If we want to use `/etc/host-keys`, we need to change the label policy for that directory and its contents. Since this is a new directory, let's add a new policy rule for it:

semanage fcontext -a -t sshd_key_t '/etc/host-keys(/.*)?'

("-a" means add, "-t *type*" is the new type part of the label, The rest is a regular expression (ERE) matching the directory and all files within.)

If you create that directory after the change to the policy, it will have the correct label. But we created the directory first and it has an incorrect label. Let's fix that.

> As a system admin, you need to know that if you restore files from an archive or backup, or install software that is unaware of SELinux, or move a file (rather than copy it), or simply have SELinux turned off (or in permissive mode) for a while, files may be created with incorrect or no SELinux labels! Once you have SELinux back on, that can obviously cause problems.

In our above example, we change the policy but not the existing label. To fix labels, run the command:

```
# restorecon -Rv /  # or any file or directory
```
Relabeled /etc/host-keys from unconfined_u:object_r:etc_t:s0 to unconfined_u:object_r:sshd_key_t:s0

This command changes the labels of all files to what the policy says they should have. (There are other commands too but they all do the same job.)

> Most SELinux problems can be fixed with `restorecon` or by setting some Boolean.

In addition to controlling access to files, SELinux controls access to network ports.

SELinux has other features as well, including role-based access control and multi-level security (labels such as *classified* or *top-secret*). These and other details of SELinux (such as changing the policy) are discussed in the security course.

Here's a **list of the most important SELinux commands** you should know:

`getenforce`
`setenforce 0` or `1` (to set permissive or enforcing)

`getsebool -a` # *Lists all Booleans and their current setting*
`semanage boolean -l` # *Run as root; gives definition of each Boolean*
`setsebool [-P] some_bool 1` or `0` (on or off; "-P" means permanent)

`semanage ports -l`

To adjust file labels to the policy:

```
restorecon -vFR /some/dir
```

To change labeling policy for `/some/dir` and its contents:

```
semanage fcontext -a -t some_type_t '/some/dir(/.*)?'
```

For example, to add `httpd file-context` for everything under `/web`:

```
semanage fcontext -a -t httpd_sys_content_t \
"/web(/.*)?"
restorecon -R -v /web
```

When searching for the cause of some failure:

```
ausearch -m AVC [-ts today] # can use other criteria too
... | audit2why
```

To update the policy to allow something that failed:

```
... | audit2allow -alrM foo; semodule -i foo.pp
```

File Locking [*Discussed fully in security. See also flock-demo from shell script course.*]

Some administrative tasks that involve updating config files (or any shared data files) can be dangerous. A conflict can arise if two or more processes attempt to modify the same file at the same time. For example, consider what might happen if you `vi` the `/etc/passwd` file and someone else (perhaps a `dnf upgrade` running from cron) updates the file (or a related one such as `opasswd`, `gpasswd`, `shadow`, ...) at the same time. Or one process modifies `shadow` while another modifies `passwd`; the files are no longer consistent.

File locking is used to prevent this sort of error. Before opening some file, a process can obtain a *lock* on it (or some other file). No other process can obtain the lock and will wait until the lock is released by the first process, before having a chance to obtain that lock itself.

A system admin must sometimes manage, troubleshoot, or setup locking. This may involve mount options, setting special permissions, creating/removing lock files, and even tracing utilities to see what locking they use, if any.

System utilities, user applications, and shell scripts often create locks by creating ***lock files***. When such software starts, it checks if the lock file exists already, and if so will either wait or abort. If not, it creates the lock file. **This scheme depends on all software accessing the same resource use the same lock.** So, always document any lock files used by your admin scripts.

Often these lock files are empty, but sometimes they contain data such as a PID. **Most system lock files are kept in `/var/lock` or `/var/run`.** You can use the utility `lslocks` (or the older `lslk`) to examine these.

While commonly used for lock files, using `/var/run` and `/var/lock` have problems since `/var` may not be mounted early enough at boot time. Some daemons use hidden files and directories under `/dev` instead, but that has never been a good idea.

A newer approach is to use a new top-level directory for both, **`/run`** (and `/run/lock`). (Often that will use a RAM disk.) The old folders have become symlinks to the new locations. Using a RAM disk for lock files works well, since any reboot clears all those files automatically.

When working with `/etc/passwd` and related shadow suite files, rather than lock multiple files, `vipw` and `vigr` create an advisory lock on the single empty file **`/etc/.pwd.lock`** (created if it isn't already there). `vipw` and `vigr` additionally use **`/etc/ptmp`** as a copy of the file being updated; when done this is "`mv`"-ed to replace the original file.

`vim` edits a copy named *`.name.swp`* but allows you to re-edit files anyway. (It's not really a lock file.) If you get a recovery message you can recover using `vim -r`, but this won't remove the `.swp` file. You must do that manually. Old vi also has such files but in `/var/preserve`.

Normally utilities and daemons clean up their lock files (and other resources). But, **when some program crashes it may not have a chance to remove its lock files**. This will prevent the program from restarting. The fix is simple; just manually remove the offending lock file.

Applications also use lock files, such as Firefox. Unlike lock files for daemons, these are usually found in a user's home directory and will not be automatically removed if the system crashes.

List of Linux Security Subsystems

Unix and Linux have many security subsystems. For some access to occur, all the relevant subsystems must allow that access. If any one or two of these is not updated when adding a new service, the service will be blocked. Some of these subsystems include:

- DAC, which is the regular POSIX permission system;
- ACLs, which augments the DAC;
- Filesystem-specific attributes;
- PAM, which allows services and applications to define security policies to be enforced by users;
- Firewalls, including proxy servers and packet filters;
- SASL, which is used by network services to negotiate which security mechanisms and protocols to use for client-server communication;

- SSSD, which is used to connect local users (and services) with remote resources that require authentication;
- MAC, which is implemented by SELinux or others to enforce security policies that not even root can violate;
- Various kernel level protections and per-service configurations.

Isolation Techniques — `chroot`, FreeBSD `jails`, and Solaris `zones` (Containers)

Ordinarily, filenames are looked up starting at the root of the directory structure, i.e., "`/`". "**`chroot`**" changes the root to some other specified directory (which must exist). The commands run this way cannot access any filename outsize of this directory. (Symlinks won't work but hard links will, so it is best if the chroot jail is on its own partition.)

This works best for a *statically linked* server. You put the server and its data and configuration files someplace, then run it as `chroot`. Even if some hacker breaks into this program, they cannot access any files outside of this directory. Not even root can break-out (in theory; however several exploits are known and still possible with some systems).

Not all server programs support chroot (check the man page and see if there is any command line option to enable that). Some that do support chroot run that as soon as they start, others wait until completely setup before running the chroot call. The first type requires a complete environment setup within the chroot jail, the second type can refer to DLLs and config files from their normal locations. There is a `chroot`(1) command too, so you can create a chroot jail even for programs that don't support it. **Note: only root can run `chroot`**.

To set up the first type of server (say "`ls`") if it uses DLLs, run "**`lddtree`** `ls`" to see what shared objects it needs. Then in addition to copying the application (to .../bin), also copy the listed files to the required positions (.../lib) under your intended new root directory. Finally, if the executable requires any other files (e.g., data, state, device files), copy them into place too. So typically a server will have this directory structure: .../*top*/{bin,lib,etc,dev}.

To write to `syslog`, a *socket* (.../`dev/log`) must be created within the directory structure that the program can access (or the syslog DLL must be used). `Syslogd` must be started with the `-a` *`socket`* option.) (**Demo on YborStudent: /var/named/chroot, /var/ftp**.)

> Fedora includes the tool `mach` to help setup chroot environments, used mostly for building packages.

FreeBSD `jails` take this idea one step further. The `jail` system call reuses `chroot`, and also separates out all processes to be restricted to that jail, adds networking restrictions to the `jail` (so processes can only access pre-selected addresses and ports), removes super-user privilege to any access outside the `jail` (say by using some copy of a device file).

> Specifically, jailed processes ("inmates") cannot interact with processes in a different jail, load/unload kernel modules (or change the securelevel), modify the network configuration, change sockets, or use raw sockets at all. A jail is bound only to specific IP address (unique to that jail) and firewall rules cannot be changed. Mounting and unmounting filesystems is prohibited. Jails cannot access files above their root directory (i.e. a jail is chroot'ed), and they cannot create device nodes.

Solaris zones (a.k.a. ***containers***) build on the `jail` concept and add even more security and control. Zones are useful to run a number of services on a single server. The cost and complexity of managing numerous machines make it cost effective to consolidate several applications on a single larger, more scalable server.

A `zone` provides an application execution environment in which processes are isolated from the rest of the system. This isolation prevents processes that are running in one `zone` from monitoring or affecting processes that are running in other `zones`, even if root. **Unlike `jails`, processes in `zones`/containers cannot hog the CPU, RAM, or disk I/O of the system, only what has been allocated to that zone/container.**

Each zone has its own set of users, name service/resolver setup, hostname, etc.) Each `zone` that requires network connectivity has one or more dedicated IP addresses. Processes that are assigned to different `zones` are only able to communicate through network APIs.

Every Solaris system contains a ***global zone***. The global `zone` is both the default `zone` for the system and also the zone used for system-wide administrative control. Other zones are called ***local zones***. Also supported are ***Branded zones*** (Brand-Z), which can use another OS instance (e.g., the "lx" brand for Linux).

> Linux also supports containers such as LXC (see LinuxContainers.org). Containers are becoming very popular alternatives to virtual machines as they allow the contained service/application to run anywhere.

Docker standardized container image files making the technology very useable. The image format was donated as an open standard, along with other Docker technologies.

For complete isolation you need separate hardware. The next best thing is ***virtual machines***, which are instances of the OS each running just one program. This is

indeed possible with products such as VServer (free: http://linux-vserver.org/), **Xen**, **VMware** and VM/SP. (This is a popular solution for big iron, i.e., larger computers and mainframes.) However, for most commercial systems the isolation provided by containers is sufficient.

It is easy to **install and play with Docker** on Linux:

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ chmod +x get-docker.sh
$ sudo systemctl enable docker
$ sudo systemctl start docker
$ sudo gpasswd -a $USER docker # only root or members can run docker
$ exec su - $USER  # to reload group memberships
```

Now that the one-time stuff is done, time to play:

```
$ docker pull alpine  # A small Linux distro in a containter; try "ubuntu"
$ docker container run -it alpine
# ... # running shell in container; type ^P^Q to detatch
# exit
$
```

UPDATE: Docker requires cgroups v1, not enabled by default since F31. Instead, on Fedora use the nearly identical podman. (There's a short tutorial at familiarizing-yourself-with-podman.) The standards that Docker invented were donated to the Open Containers Initiative (OCI), so other tools such as moby-engine and podman will all work nearly the same.

A bit more on Linux containers

Today organizations use private data centers, or public ones known as a *cloud*. Each computer in the data center can be configured as part of a cluster on which containers can run. Such computers (which could be virtual hosts) can be called *nodes*. When an admin deploys a container, the cluster decides on which node to run it.

Modern services are actually composed most often of multiple containers working together, that get deployed to the same node as a group. (Google calls these groups "pods", a joke on the Docker logo of a whale.) Additionally, a container system running on the cluster can keep track of how many replicates of each group are running and start new groups or terminate groups as needed. The system can load-balance incoming requests to the replicas. If a node gets too busy, the system can move a group to another node.

The system keeps track of which container is connected to which network, and makes virtual networks for each container to connect with, as well as port-forwarding from the node's network ports to container network ports (so the

outside world can talk with the containers). Since containers can move about, the system also manages a DNS server so the containers can find each other easily; this is called *service discovery*.

To manage all that (and additional tasks) requires the system keep track of nodes, groups of containers, metrics, and a lot more. The software that does this is called ***orchestration software***. Docker includes ***Docker Swarm***, but today (2019) the most popular software is Google's ***Kubernetes***.

> Red Hat provides a Docker + Kubernetes system with commercial support that they call *Openshift*. Macdill AFB is (as of 2019) migrating to such a data center and is in fact looking for Openshift certified admins.

Migrating existing applications to run as containers (a *containerized application*) can be difficult but doable. Applications written to run as containers in the first place are known as *cloud-native*. (This is more of an issue for developers than system admins.)

Many programs rely on hardware-based security and isolation, something containers don't provide. In public clouds, customers run containers on top of virtual machines. Some newer types of containers can use lighter-weight VMs (sometimes known as unikernels) to provide such security.

`sudo` and `/etc/sudoers`

This security system allows a sys admin to give privileges to users without giving the root password out. Users then can run a command prefixed with `sudo`. Sudo provides an audit log (using `su` doesn't). Example:

```
/home/jdoe$ sudo passwd someUser
```

Sudo prompts users for their password and remembers it for about 5 minutes so you don't have to re-authenticate for every command. Edit `sudoers` only using **`visudo`**, which performs some sanity checks on the `sudoers` file before saving to disk. Edit config (also journal, etc.) files safely with **`sudoedit`** (same as `sudo -e` *`file`*). To allow `jdoe` to edit `/etc/hosts`, you add this entry:

```
jdoe ALL = sudoedit /etc/hosts
```

Don't use `/usr/bin/sudoedit` in the file; the name `sudoedit` name is special, not a pathname. **(Show and review sample `/etc/sudoers` resource.)**

Homework: read `sudo` man page (it's long).

Lecture 3 — Managing System Security — PAM and Crypto (PGP/gpg) [*details in security course*]

PAM Background:

With **PAM (for *Pluggable Authentication Modules*)** programs would not need to be changed in any way to use different authentication modules. (Hence the name.) There are modules to check user passwords (from a variety of sources) and to check for valid (unlocked) user accounts.

Modern (and most legacy) applications and daemons that need authentication use PAM. There are many PAM modules available for every system and new ones can be easily created by programmers. A nice benefit of this design is that different programs can use different PAM modules for authentication, all on the same system. In addition, PAM modules can be used for session setup and tear-down, logging, and various similar uses. You can list for each application a set of modules to try, so if the user can't authenticate using (say) local files then PAM might try a different database such as Windows AD. PAM can also be configured to deny access if the resource is busy or based on other criteria such as the time of day.

PAM Details:

PAM is a library that applications needing authentication are compiled with (e.g., login). For each program that you want to control, add a file in **`pam.d`** with the application's name (usually). That file lists the PAM modules that will be used along with any module options.

> To see if some program is "PAM-ified" try:
> `lddtree` *`cmd`*`|grep libpam.so`

These files have the following syntax (described in `pam.conf(5)` man page):

context control-flag module options

A backslash at the end of a line is used to continue long lines on the next. If an application was compiled with PAM and there is no file in `pam.d` for it, the **`pam.d/other`** entry is used. (**Always have an appropriate `other` file using `pam_deny.so`**!)

By editing (or creating if missing) these files, an SA implements access policy for the applications and servers on the host.

> Other security systems also require configuration to implement a host's access policies, such as firewalls, TCP Wrappers, file permissions and ACLs, group memberships, SELinux policy files, service configuration files, SASL, and so on.

PAM modules are installed in **`/lib64/security`** (and are typically named `pam_XXX.so`) and are documented in the man pages (maybe! Try `man pam` and/or `man pam.conf`) and the on-line documentation: `/usr/share/doc/pam-version/*`

A PAM module may be called within one of four *types* or *contexts*:

- ***auth*** (authenticate users, most commonly from passwords)
- ***account*** (check for valid user accounts; also controls access to resources by other means: time of day, max number of users, etc.)
- ***session*** (don't control access at all, but instead are used to do some session setup and cleanup)
- ***password*** (control a user's ability to update authentication information).

Modules may define more than one type (group) of functions. So a module `pam_pwdb` may do one thing if used in an `auth` context but something different if used in a `password` context.

When a user runs an application such as `chfn`, the PAM function **`pam_authenticate`**`()` is invoked. This function locates the PAM configuration file for this application and starts running the *account* and *auth* modules listed within, in order. (*Show eject, other, xserver*).

If the user successfully authenticates, then any *session* modules are run in order. When the session ends (the application exits or the server TCP/IP session closes), the session modules are invoked to close the session (clean up stuff).

During a session, if the application permits the authenticated user to change passwords (or otherwise update their security credentials) the *password* modules are invoked.

Any module arguments listed in the configuration file are passed to the PAM functions that get invoked. These functions can be thought of as returning either success (pass) or failure (denied). The *control* field in the configuration file says what to do when a module fails, and is discussed below.

PAM Actions (the *control* field):

For each listed module in an application's configuration file there is a *control-flag* that says how PAM should react to a success or failure of individual modules.

In general, all the *auth* (and *account*) modules listed in the configuration file are tried, even if one or more fails. When done running all listed modules, the function returns an overall success or failure value back to the application.

The control-flags are: **`required`** (the module must succeed; if a `required` module fails the remaining modules are still tried but the result will be failure), **`requisite`** (a fail-fast version of `required`), **`sufficient`** (if this module succeeds and no previous `required` modules have failed, no further modules are

tried. A failure of this module won't affect overall success or failure of PAM), and **`optional`** (these don't affect the overall success or failure of PAM).

> While still widely used, modern PAM provides more control than just four flags. You can specify different actions for over a dozen module results. The advanced syntax shows in square braces as "[*result=action, ...*]". This will not be covered in this course.

RH puts common authentication in `system-auth`, which is over-written by the command `authconfig`. Always backup this file! This is a good idea (to put the default policy into a single file).

Example:

`pam_pwquality` (or a similar module) is used when changing passwords. Its arguments determine the password policy (which can also be specified in a config file). In `/etc/pam.d/passwd` put these lines:

```
password required pam_pwquality.so \
   difok=3 minlen=8 retry=3
password required pam_pwdb.so \
   use_authtok nullok md5
```

(Many people mistakenly think minimum password length policy is set in other files such as `/etc/login.defs` or `/etc/default/useradd`, but not so! Such files are only used under specific circumstances. For instance, `login.defs` is only consulted when using shadow password files, not if using LDAP or NIS or Samba. And even when such files apply, any candidate password must satisfy PAM criteria too. Therefore, it makes sense to set password policy with PAM only.)

Crypto:

> **TL;DR (the bottom line for 2021)**: Use TLS 1.3, AES-256, and either SHA-512 or (for secrets such as passwords) argon2id.

Checksums (a.k.a. message digests, CRCs, hash):

Describe *casting out nines* (checksums): sum all the digits in a set of numbers, repeating until a single-digit results. Repeat procedure on the answer. If the check digit is different, the addition was wrong. The Roman bishop Hippolytus used this as early as the third century. Today money, credit card numbers, ISBN numbers, etc. all use checksums.

Cyclic redundancy check (CRC) has far fewer errors, using repeated division by a specially chosen value. The remainder is the CRC. CRCs are used in networking. (Demo `cksum`, which uses the Ethernet CRC defined in ISO/IEC 8802-3:1996.)

Message digests (fingerprints, hashes, signatures, Message Integrity Codes, and other such terms) can be used to check on the integrity of a downloaded file (or to

check if some attacker has replaced some file). A ***secure digest*** is one that an attacker can't duplicate on a modified file. It would be easy to add a virus to a file and then add some padding at the end, to force the result to match the original checksum or CRC. If a secure digest is used instead, an attacker cannot tell what to add to make the digest the same as before.

Cryptographically secure message digests are often called by other names, such as secure digest or secure hash, or MIC (*message integrity code*). Some commonly used secure hash functions are:

MD5 - **`md5sum`** *`file`* (`md5sum -c file.md5`) Demo on YborStudent.

(Demo: `printf s3cr3t |md5sum`; copy and past that to hashdecryption.com.)

> In 2004, MD5 was shown to have some theoretical weaknesses. While still considered safe at the time, in 2008 a more severe weakness was discovered. It is possible for virus writers to craft a virus infected file to have the same MD5 sum as the real file! See http://www.phreedom.org/research/rogue-ca/ for more information.
>
> The NIST phased out acceptance of MD5 and SHA-1 by 2010. Instead it is recommended to migrate to SHA-2, SHA-3, or another more secure algorithm (see U.S. Gov. standard FIPS-180-2). For a list of approved MICs (and other ciphers and related algorithms), see *secure hashing* at `csrc.nist.gov/groups/ST/toolkit/`.

`sha1sum` works the same but provides SHA-1 message digests. (SHA-512 or SHA-256 is recommended over SHA-1 or MD5 digests. Use SHA-3 if available.)

Today (2019), best advise is to use **Blake2b** (`b2sum`) for fast hashing on standard hardware. (Blake1 was runner-up in the NIST competition; the winner (SHA-3) is faster if using custom hardware which the DoD can afford.) If not that, SHA-3 or SHA2-512 (the latter is more mature and some trust it more).

For hashing secrets such as passwords and keys, it isn't enough to use a good hashing algorithm; you must use it in a way the make it resistent to various attacks. For hashing passwords and keys, today the best advice is to use **argon2id**, followed by scrypt or yescrypt, and use the previous "best advice" choice of PBKDF2-SHA512 with 85,000 iterations as a last resort.

(Of course, you cannot use some cool hash method if it isn't supported on your system!)

> Windows treats text files specially when reading. On Windows use the `-b` (binary mode) flag with the *sum tools. This mode is indicated with "`*`" (space-star) between the checksum and the filename. The mode doesn't matter on *nix, but if you don't use `-b`, you should edit the saved

checksum file and change the two spaces to space-star, in case users try to validate your checksum on a Windows platform.

GnuPG supports many message digests in a very portable way. (This is touted as a feature since some systems use different code and may report a different digest. Some folks prefer to see such errors.) Use "`gpg --version`" to see a list of supported "Hash" algorithms. Here's how to produce an MD5 checksum for some *file*:

```
gpg --print-md md5 file
```

Today (2019), the US government has a new standard they like, SHA3.

To make password hashes you can copy and paste into the `/etc/shadow` file, use

printf `"secret" | openssl passwd -1 -stdin`

Note `openssl` did not support SHA512 or better until 2020; the "-1" (one) means to use MD5. For SHA-512, use "-6" instead of "-1". See the `sha512password` script that uses Python.) Even easier is to install if available the `mkpasswd` utility (show). See `crypt(5)` for allowed methods.

The argon2id method is harder to use on Linux (as of 2020) since you need to determine many things by hand. Here's an example use:

```
SALT_LEN=$(( $(LC_ALL=C; tr -cd '[:digit:]' \
    </dev/urandom |head -c 2) / 2 + 8))

SALT=$(LC_ALL=C; tr -cd '[A-Za-z0-9./]' \
    </dev/urandom |head -c $SALT_LEN)

THREADS=$(( $(nproc) * 2 ))

MEM=$(free -k |awk '/Mem:/ {print (0.75 * $4) }')

printf secret |
  argon2 "$SALT" -id -p $THREADS -k $MEM

Type:           Argon2id
Iterations:     3
Memory:         233661 KiB
Parallelism:    4
Hash:
6851f3239e0655b814a708842b1f53fcd4a9cfcfc58e0cbe2
9e6a467490d779a

Encoded:
$argon2id$v=19$m=233661,t=3,p=4$VVdbWHNPek02Smxhb
k5uNTBWSE1b$aFHzI54GVbgUpwiEKx9T/NSpz8/
Fjgy+KeakZ0kNd5o
```

```
0.725 seconds
Verification ok
```

Here we use salt generated of 8..58 (there's no real upper limit so I just picked this) random character of the type Linux allows in `/etc/shadow` (see `crypt(5)`). I used 3/4 of available memory and 2* number of cores; the more you use the harder it is for attackers, but the longer it takes to generate hashes. The default of 3 iterations with these settings doesn't seem to take too long.

Symmetric Encryption and Public Key Encryption

Encrypting data means to scramble the bits of the data so it is meaningless to anyone looking at it. *Decrypting* means to unscramble the data back to its exact original form. The process is controlled by a password or a key. A *password* is usually short (8-16 characters or normal) text a human can type in without error. A *key* is a large number, usually hundreds or thousands of digits long. Keys are stored in files as humans could not type such long numbers without error. The files holding keys should be made as secure as possible and are often encrypted with a password. A truly random key would take so long to guess, hackers don't usually try.

Encrypting a key means a human must enter the password for that key, so for services such as web or mail services keys are usually not encrypted so the service can start automatically. This is a trade-off between convenience and security.

Symmetric encryption uses a single password or key to encrypt and decrypt. These methods are fast and common, such as AES. These typically work by taking the data as input, and using the key or password to control the scrambling process. To decrypt, the encrypted data is fed into the program which runs the scrambling algorithm backwards. Note that even knowing the algorithm, without the password/key an attacker cannot decrypt the data.

Symmetric key encryption is used to password protect files. However, it isn't as useful for e-commerce applications such as buying stuff online. The main issue is that the password or key must be shared, and there's no secure way to do that over the Internet.

Public key encryption is used on the Internet. It works by having **each party create a pair of keys** that have an interesting mathematical property: data encrypted with one key is decrypted only with the other key. The algorithm is not run backwards.

Of the two keys, one you keep very secure; it is called the *private key*. The other can be published for the world to see and is called the *public key*. To send an encrypted message to someone, you need that person's public key. When that

person gets the encrypted message, they can decrypt it using their private key. An attacker, knowing the method, the encrypted data, and the public key, cannot decrypt the message!

Digital Signatures:

A neat application of public key systems is for ***digital signatures***. A digital signature **is an encryption of a message with the *private* key**. This means anyone with the matching public key can decrypt the message, but since nobody but the owner has access to the private key, a successful decryption proves the messages was sent by the owner of that pair of keys.

Many uses for this include digitally signing downloadable software (RPMs) so you can have confidence it came from the organization you think it did, and that it hasn't been corrupted. You can sign email, config files, log files, and even contracts (note valid legally since 2002, e-sign act). Use **gpg** (vs. crypt):

`gpg --verify file.sig`, `gpg --import keyfile`

```
gpg --keyserver wwwkeys.pgp.net --recv-keys \
   0x517D0F0E
```

Public key encryption is very useful, but also very slow even on modern computers. In practice, a large random number is generated to encrypt the data using symmetric encryption called a *session key*, and only that key is encrypted with public key encryption. For validation only (no privacy), a secure hash of the data can be encrypted using public key methods, then appended to the unencrypted data. If the data is altered, it won't match the hash. Often double encryption is used: the sender signs the data and then encrypts using the recipient's public key.

Other Related Issues:

A ***Message Authentication Code* (MAC)** or HMAC (a MAC that uses a secure hash) differs from a normal secure digest in that a shared secret (key) is used when generating the MAC. This use of a key prevents forgery (posting a fake version with a correct MIC) and ***false repudiation*** ("*I didn't dood it!*"). This is exactly what a digital signature is used for, except those use *public key* crypto rather than a shared key. MACs can be used to share data between servers in a cluster or database replicas.

Password aging (**`chage`** on Linux, `passwd` on Solaris); defaults for logins in `/etc/login.defs` on Linux, for Solaris `/etc/default/login`.

Security and cryptography are very difficult. It is practically impossible to invent your own crypto or security protocols. Just don't even try! Always use tried-and-true solutions for managing passwords and other secrets, and for secure data handling. That said, you still need to configure those systems. Crypto changes all the time as some algorithms

and mechanisms become vulnerable. You will need to keep up with that and configure your systems to use the correct mechanisms. (And more importantly, to not use ones known to be broken!)

Lecture 4 — Using and Securing `cron`, `anacron`, `at`, and Systemd Timer Services

Show `crontab`, `at` web resources.

> Since the command listed in a crontab file must be contained on a single line, `crontab` supports a percent (‘`%`’) character to expand to a newline in the command. Since several common commands use a percent in their command line arguments (e.g., `date`), you must remember to escape a literal percent with a backslash!
>
> You can use this “feature” to include a pseudo here-doc in a crontab:
>
> ```
> 1 2 3 4 5 cat%line1%line2
> ```

These facilities are used to schedule tasks or *jobs* to run at some future time. Although the `crontab` command is standard in POSIX, the daemon is not. This means different systems will have different configuration and options.

Use `cron` (`crontab`) and `anacron` (or `periodic`) to run jobs repeatedly on some schedule. Use `at` (and `batch`) to run a job once only, but not now. All output of these jobs (if any) is sent as email to the user who scheduled the job.

Review: `crontab` files, `crontab -lre`, `crontab` *`file`*.

`Cron` and `at` jobs don’t run using the same environment as when you type the commands at a shell prompt. (On some systems, the current environment is preserved for `at` jobs.) This sometimes leads to commands working when you test them, and failing when you run them from `cron`. You can test your jobs using `env` to reset the environment to the same one that will be used for the cron job:

```
env -i /bin/sh -c "USER=\"$USER\" \
LOGNAME=\"$LOGNAME\" HOME=\"$HOME\" \
MAILTO=\"$MAILTO\" \
some_command"
```

`/var/spool/{at,cron}` *Location of user’s crontab files*.

`/etc/cron.d/` *Location of system-wide crontab files (Linux, Solaris)*. The Linux version of cron uses an extra *`user`* field between time-spec and cmd in this file, and also the following.

`/etc/crontab` (Linux, FreeBSD) Generally used to run the scripts in the following directories: `/etc/cron.{hourly,daily,weekly,monthly}`

The **`anacron`** service runs jobs at boot time from `/etc/anacrontab`, by default `/etc/cron.{daily,weekly,monthly}` scripts tasks, if too many days have passed since the task last ran. (Note the `cron.hourly/*` jobs are run

by `cron`; one of those runs `anacron`.) Unlike `cron` it doesn't assume the machine is always on. Timestamps of jobs in `/var/spool/anacron`.

`anacrontab` format: *`period delay job-name command`*. The *delay* is used since `anacron` would otherwise run all jobs at once.

BSD may use **`periodic`** service instead of `anacron`, but it works similarly.

> The `/etc/cron.{daily,weekly,monthly}/*` scripts are run by anacron, which in turn is run by cron (along with the scripts in `/etc/cron.hourly/*`).

If managing a group of similar servers, it can be a problem if cron jobs start at the same time (e.g. `yum` downloads for 200 hosts). But having different cron files per host is also a maintenance headache. The answer is to insert a small cron delay, different per host. RH used `000-delay.cron` in `/etc/cron.*`, and the SA could configure the range of the delay in `sysconfig/crontab`. This delays jobs for a number of seconds depending on the MD5 hash of the hostname. (**Show web resource.**) (That file is no longer distributed; enterprises need a more scalable approach, usually called job scheduler software, and indeed, there are many to choose from.)

The newer Linux version of `cron` supports clustering; the `/var/spool/cron` directory can be network-mounted on all machines in a cluster. Then only one of the machines will actually run the jobs. See `cron(8)` for details.

With Linux cron, a number of additional security measures are used: no crontab files may be links, or linked to by any other file, and no crontab files may be executable, or be writable by any user other than their owner. The crontab files in `/etc` must be owned by root. PAM is used (see `/etc/pam.d/crond`) if available.

Using `at`: `at -l` (`atq`) to list pending jobs, `at -c` *`atJobNum`* shows contents of job (Non-POSIX), `at -r` (`atrm`) (`-d` on Gnu but not POSIX) to remove pending job. Also the `batch` command works like at but instead of running a job at a set time, it runs jobs when system load is low.

Securing at, cron

Uses the files **`{at,cron}.{allow,deny}`** to control user access to these facilities. If `.allow` file exists ignore `.deny` file, if neither then only root for `at` but all for `cron` (varies by distro). (Default: empty `.deny` file.)

Early `cron` (and `at`) ran jobs as root. Now they run as the user who created them. (The root cron files contain an extra field to indicate which user to run as.) They also are run using `/bin/sh` (i.e., no extra Bash features enabled), with very limited environment setup.

Solaris9: `*LK*` as the 1st four chars in `/etc/shadow` password field prevents `cron` or `at` jobs of that user from running.

Systemd Timer Units

Systemd includes *timer* units designed to replace crontab and anacron. `mlocate-updatedb.timer`, a typical timer unit file, looks like this:

```
[Unit]
Description=Updates mlocate database every day


[Timer]
OnCalendar=daily
AccuracySec=24h
Persistent=true

[Install]
WantedBy=timers.target
```

Not shown is that timer units automatically gain a `Before=` dependency on the service they are supposed to activate.

> As discussed in Admin I course, cron and anacron can be easier in some cases, but timer units have several advantages and you should learn them.

In the `[Timer]` section you also need to say which unit to run when the timer triggers; if "`unit=`" is omitted, the default is the service unit with the same name as this timer unit.

Note you cannot trigger a timer unit from another timer unit.

Crontab-like timers are defined using `OnCalendar=`*when*. The *when* argument is much more expressive than cron allows; see `systemd.timer(5)` for details of the syntax. Briefly, you specify dates with [*dayOfWeek*] [*year-month-day*] [*time*]. You can always omit the *dayOfWeek* component, but must have at least one of *year-month-day* or *time*. If missing the date, today is assumed; if missing the time, "00:00:00" (midnight) is assumed. Time can also include "`UTC`" or another standard timezone name.

Each of the five components can use a comma-separated list of values, a range (using x..y notation), a "*" for *any*, and for the *day*, you can also use "~" or "~*num*" to indicate the last day (or the *num*-th from last) day of the month. *DayOfWeek* values are case-insensitive English week day names, which can be abbreviated. Finally, seconds can contain fractional sections (all values rounded to six decimal places).

As seen in the example above, there are a number of common pre-defined *when* arguments, such as *minutely, hourly, daily, weekly, monthly, quarterly, semiannually, and yearly*. Here's an example

monday *-12-* 17:00 (normalized form: Mon *-12-* 17:00:00)

Which means Monday, any year, December, any day, at 5 PM.

Notice the "Persistent=true" line in the sample unit file. That causes systemd to save the time the timer was last triggered on disk. When the timer is next activated (that is, next boot), if it would have been trigged while the system was off than it is triggered immediately. This is similar to anacron.

In addition to *OnCalendar=* cron-like timers, there are also actual timers. These are specified differently and are called *monotonic timers*. You use one of *OnActiveSec=*, *OnBootSec=*, *OnStartupSec=*, *OnUnitActiveSec=*, or *OnUnitInactiveSec=*to specify a time duration in seconds, relative to some starting point. A monotonic timer can specify multiple durations. See `systemd.time(7)` for the syntax; some examples from that page include "2 h", "2hours", "2hr", "1y", "12month", "55s500ms", "300ms20s 5day".

For all timers, you can specify *AccuracySec=* to set the granularity of the timer; the default (like cron) is 1 minute, but can be specified as "1us" for Nano-second accuracy. You can also specify *RandomizedDelaySec=* to delay the trigger by a random amount up to the number of seconds specified (defaults to zero). This helps prevent your server from running too many jobs at the exact same moment; imagine the network load if every PC in HCC did a Windows update at the exact same time! Here's a final example, `fstrim.timer`:

```
[Unit]
Description=Discard unused blocks once a week
Documentation=man:fstrim

[Timer]
OnCalendar=weekly
AccuracySec=1h
Persistent=true

[Install]
WantedBy=timers.target
```

Which says to run `fstrim.service` once a week during some random time within 1 hour, or to run it within an hour at boot if it hasn't been triggered in over a week.

Systemd timer units can take experienced sys admins unawares, since some services that traditionally used crontab have switched. Be sure to look for `*.timer` files too when looking for cron jobs.

Enterprise Cron

When you move up to consider clusters and enterprise applications/services, or whole datacenters, standard cron is not sufficient. For example, suppose you need to run a cron job that attempts to run processes on multiple hosts. If the network is

partially down, only some of those hosts will run the job. If the host running cron itself is down when the job is supposed to run, none of the hosts run the job. (Using anacron, when the cron host comes up it will try to run the job then.)

Retrying cron jobs is in general dangerous. This is because the job may have run already, but then the network failed to deliver the "all done" message. Thus, the cron server may try to run the job multiple times.

This is not always dangerous. Some jobs are ***idempotent***, meaning they can be run multiple times without harm (for example, running a garbage collection task, or a server health check). However, many tasks are not idempotent and running them multiple times can corrupt data in databases or corrupt audit logs, etc. (For example, jobs to send out monthly bills to customers, or start payroll processing are not idempotent.)

What's worse, basic cron is stateless; it doesn't keep track of (or report about) failed, missed, or doubled job launches.

Regular cron doesn't scale up to datacenter use: If you run cron on every virtual machine (VM), how do you update crontab reliably as hosts are added/removed dynamically to the cluster? Given how long it might take to update hosts' crontab files, inevitably some will be running the new jobs and others will miss the job (the crontab update will arrive too late). Also, what about cron jobs that need to be run once per cluster and not once per host?

If you have 1,000 hosts and one is the cron service, a failure of 1/1000ths of your hosts would stop the cron service for all. Also, VMs or containers come and go all the time in a cluster; how does the cron service know which host to send the job to? If the job was sent to a VM that failed shortly thereafter and was replace with a fresh VM, should the job be restarted on the new VM? Running cron on every VM individually doesn't help with this issue either.

At enterprise scale, you need a distributed (and hence reliable) cron service that does track the state of jobs and does reporting that can be monitored. Several solutions are known, such as the Google cron service, Cronos, and others.

Lecture 5 — Managing man pages and other Documentation

Locating documentation on the Internet:

- `docs.oracle.com/cd/E26502_01/` (formerly `docs.sun.com`) and the sites for other vendors, e.g. IBM, FreeBSD, HP-UX)
- `www.tldp.org`
- `kernel.org`
- `oreilly.com/linux`
- `linuxtoday.com`
- `freecode.com`

Also various newsgroups (show `groups.google.com`: `comp.os.linux`, `comp.os.linux.answers`, `comp.unix.solaris`, ...) and mailing lists, FAQs, and Google search (`www.google.com`). Beside **man pages**, **info** is used for GNU software (just because they can).

Other docs in **`/usr/share/doc`** (or on older systems, `/usr/doc`; a good idea is to create a symlink) in different formats. `/usr/share` is meant to be mounted using NFS so all hosts in your network can share a single copy. Some formats you may see include **TeX** and LaTeX (view with `lyx`, a TeX word processor), PS (PostScript) and PDF files (view using **`gview`**, a part of the GhostScript tool), HTML, and plain text.

Managing Documentation

Man pages or `/usr/share/man` (on older systems, `/usr/man`). Create index with **`mandb`** (Linux), Solaris: `makewhatis /usr/share/man`, or `catman -w` (depends on the version; also HP-UX and AIX use `catman`). `mandb` used to run daily, automatically via `cron`. However, modern systems run `mandb` through systemd only when packages are added/removed/updated that change files under `/usr/share/man`. (This is an RPM *trigger*. *Show rpm -q --filetriggers man-db.*)

The database built by mandb is in the same place as the catman pages (if that is enabled), `/var/cache/man/index.db`.

> Because it can take a human-noticeable amount of time to locate and to format man pages, the formatted pages are sometimes cached as "cat-able man pages", or catman pages. The location for these (if they are used at all) varies by system. On Linux, look for `/var/cache/man/cat*/`. I prefer not to use these, as they need purging every so often, and I often view man pages from differently sized windows. The `NOCACHE` directive is usually the default setting.

Each section of the manual is stored in a subdirectory, man1, man2, ...). These subdirectories may be in a language or organizational (POSIX) subdirectory, under `/usr/share/man`. (Multiple man page hierarchies are common, such as `/usr/local/share/man`.) Common manual sections are:

AT&T/Solaris	Linux/BSD	Contents of Manual Section
1	1	User Commands
1M	8	System Administration Commands
2	2	System Calls
3	3	Basic (C) Library Functions
4	5	File Formats
5	7	Misc info: Standards, Environments, Troff Macros
6	6	Demos (Solaris only) and Games
7	4	Device and Network Interfaces

On some systems, additional sections may be available such as:

9 DDI and DKI (Device Driver Information) and other kernel info
0 Standard C header file descriptions
n Tcl/Tk (an old scripting language) documentation
x X window library functions
p POSIX compliant utilities (sometimes for Python library functions)

Sometimes the numbered sections are subdivided using a suffix letter: section `3C` is for C library calls, `3M` is for the math library, etc. Sometimes a suffix letter of "`p`" is used to indicate a POSIX version of some utility (so "`1p`" rather than "`p`") or library function (API) (i.e., "`3p`"); and sometimes "`x`" is used for X window utilities ("`1x`"). Perl man pages are usually in "`3pm`". As there is no SUS/POSIX standard for this, different *nix systems use different schemes.

(Try `whatis intro`.) Additional manual sections may be present, "`l`" for local, "`p`" for Python, POSIX, or Perl man pages, etc.

> Note all man pages are where you might think they should be! Shell built-in commands are in section `1p`, but "`man cd`" on Linux brings up a page on `bash-builtins(1)`. PostgreSQL's SQL commands are in section 7 for some reason (ex: `CREATE_ROLE`).

`MANPATH` points to additional collections of man pages; not the `man1` subdirectory (some systems have a `manpath` command), but the whole collection directory. On some systems, various places are included on `MANPATH` automatically; for example, `~/man/` is searched for man pages on some systems. Instead of setting an environment variable a user can put the manpath in the file `~/.manpath`.

There is a default `MANPATH` setting so normally there is no need to set this. Usually it is unset, and a MANPATH is generated each time you run the `man` command. The `MANPATH` used will included all standard locations, plus additional locations based on the `PATH` setting. With Linux, you can use the **`manpath`** command to generate a suitable one; run from your login script.

On some systems `MANPATH` overrides all standard locations, so it is a good idea to generate the current setting and append to it, when setting `MANPATH` in a login script:

```
  export
MANPATH=${MANPATH-$(manpath)}:/some/other/place
```

On other systems, an extra colon at the start or end of `MANPATH` means to prepend or append the standard `MANPATH`, not override it. For example, I have used this setting:

```
  export MANPATH=:/usr/java/latest/man/
```

`MANSECT` controls search order, default is (tip: remove some if non-developer) "`1:8:2:3:4:5:6:7:9:tcl:n:l:p:o`". As with MANPATH, there is a default you can change system-wide from the man config file.

On Linux older than 2010, `/etc/man.config` ; for the newer **`mandb`** package, the config file name varies with distro (find it using the command: `rpm -qc man[-]db`), usually `/etc/man*`. (On Solaris: `/usr/share/man/man.cf`). Note that on Solaris, many applications are installed throughout the filesystem and not just in `/usr/bin`. So are the related man pages, such as in `/usr/sfw/man`. An SA should locate all man page directories and configure the OS to include them.

To have multiple names display the same man page, create symlinks for the extras. (Show `ls -l /usr/share/man/man5 host*`.)

Creating Man Pages

Every organization has its own procedures and in-house scripts. These need to be documented and findable. Using man pages is one good way, since man pages are searchable. Another way would be to maintain a website or wiki, but that may be more work for the admin than just creating a man page.

Man pages can in fact be simple plain text files. However, plain text is difficult to format consistently so that is looks good on any output device or printer. Some modern ways of dealing with that are to use various conventions for plain text files, such as wikitext, Markdown ("MD" text) or ReStructuredText ("RST" text), which can be automatically converted to nice-looking HTML.

Before word processors, we had ***text formatters*** (a.k.a. *typesetters*). These are programs that process a text file containing special ***mark-up tags*** and produce good-looking output. One of the reasons for developing Unix in the first place was text processing (**`roff`** = *run-off*, a reference to the *mimeograph* and *ditto* copier technology of the time). These are popular today; beside *roff we have HTML/CSS, Docbook (tLDP std HOW-TO format), and LaTeX (many technical publications require this).

With a text formatter, the author specifies content and structure (headings, paragraphs, etc.) The software formats this source file, to produce the resulting formatted document. This allows easy global changes to the style of a document, and to format a document for different styles (e.g., for publication to different magazines).

Two descendants of `roff` are: **`nroff`** to format for screen and ASCII printers (think daisy-wheel), and **`ditroff`** (*device-independent troff*) for photo-typesetter printers (i.e. laser printer). Today, all have been replaced with the compatible Gnu version of those tools, **`groff`**.

Some sample `nroff`, from The zoo of Unix documentation formats.

```
This is running text.
.\" Comments begin with a backslash and double
quote.
.ft B
This text will be in bold font.
.ft R
This text will be back in the default
(Roman) font.
These lines, going back to "This is running
text", will be formatted as a filled paragraph.
.bp
The bp request forces a new page and a paragraph
break.
This line will be part of the second filled
paragraph.
Note all sentences start on a new line.
.sp 3
The .sp request emits the number of blank lines
given as argument.
.nf
The nf request switches off paragraph filling.
Until the ".fi" request switches it back on,
whitespace and layout will be preserved.
One word in this line will be in \fBbold\fR font.
```

```
.fi
Paragraph filling is back on.
```

Besides these (and may other) directives, *roff allows you to use (and define your own) macros. Generally, macros use one or two uppercase letters, like "`.PP`".

> Viewing a man page with the Gnu `less` command can be a problem. The `less` command will attempt to pre-process files according to their detected type. It will recognize a man page and process it with `groff` first! Either use `more` instead, or use "**`less -L`**" or "**`zless`**".

Additional ***pre-processors*** are available to further process man page source files, such as **`tbl`** (to produce complex tables that look good), **`eqn`** (to format mathematical equations), and **`pic`**, `tpic`, `grap` (rarely used vector graphics), and **`fig`** (for simple drawings). These use special mark-up tags only recognized by that processor. These pre-processors replace such tags with many `troff` tags (or possibly, raw DVI output that is concatenated with the *roff DVI output). You could use these tools in a Unix pipeline to process some file, something like:

```
cat file | eqn | tbl | nroff -man
```

The man command recognizes a special first line on the man page source, that indicates which preprocessors to use. Creating man pages with `nroff` (which is really a compatibility mode for `groff`) is easy. This is done by entering the text of your page interspersed with `nroff` mark-up tags. See `roff(7)` or `groff(7)` for language details. Such formatting tags are very low-level, and various *macro* packages have been developed. For example, a simple "`.P`" macro may include all of the `nroff` markup needed to start a new paragraph.

A macro collection specifically designed to format man pages is the "`an`" macros, usually called the "`man`" or "`mdoc`" macros. The macros may be used as any other `nroff` tags. To process a text file with `nroff` and the `an` macros, use: `nroff -man file`. (Using `-mandoc` or `-m mandoc` causes `groff` to guess which of two slightly different man macro packages to use: `an` or `andoc`.)

The output of `groff` depends on the value of the `-T` option. This could be "`html`" or "`ps`", but is usually one of several text formats: "`ascii`", "`utf8`", or "`latin1`". (When `groff` is invoked as `nroff`, it will use the current locale encoding and TERM settings, unless you override that with options.)

> In the past, much documentation was pure text, or possibly created with nroff. Today, it is not uncommon to use Markdown formatting in blog and forum posts, wikitext on wikis, and HTML for other documentation. (Man pages continue to use nroff; info pages, used with Gnu utilities, use

Texinfo files.) Sys admins should have some familiarity with all these formats, in case such documents need to be created or edited.

There is no need to learn all the `nroff` tags or `an` macros. Just examine a sample man page or two, and the tags to use are generally obvious. (Show `nusers.1`.) One tricky point is that blank lines are not ignored and *may* or may not cause strange appearances; use a line with only a dot instead of a blank line.

For more info see on Linux **`man(7)`**, `mdoc(7)`, and `mdoc.samples(7)`. On Solaris see `man(5)`. On any system also see `nroff(1)`. You can also see the *roff language in `groff(7)` (especially useful is the list of escapes).

`groff -mandoc -Tascii` *`file`* will process a man page and produce a text equivalent. (Other formats are possible with different `-T` options.) Convert a man page source file to a web page with `man2html`.

Man pages have an extension indicating the section and type of page (`.1` or `.1.gz`). It is required that the extension match the section; so in a directory of `man/man1`, all man pages must have a "`.1*`" extension. (Anything can follow the one, e.g., "`.1ssl`" or "`.1x`", and it will still be found if in the `man1` directory; "`.3pm`" files for Perl are found if located in `man/man3`, and so on.) Compressed man pages are allowed (and common). These have an additional extension (e.g., "`foo.1.gz`").

The extension *may* also show which preprocessors are needed; however the Linux `man` command at least (not sure about other *nix) examines the first line of the man page source for a special line that indicates the preprocessors to use, something such as "`'\" t`", where "`t`" means preprocess with `tbl`. (Example: `captoinfo(1)`.) Some of the preprocessors and the letters used to designate them are: `eqn` (`e`), `grap` (`g`), `pic` (`p`), `tbl` (`t`), `vgrind` (`v`), and `refer` (`r`). These generally work by embedding preprocessor-specific commands between preprocessor-specific *start* and *end* directives in the man page source.

Standards for man pages vary, but **always have these sections (in the correct order) at least in your man pages:**

- name
- synopsis
- description

Other optional sections include: options, return values (or diagnostics), examples, known bugs, files, see also, authors, and copyright. Linux provides a nice description of man page sections in **`man-pages(7)`**. (Demo `nusers.1`).

Linux also has `lexgrog(1)` to explain how the NAME section is parsed when building the man page index (via `mandb`). Generally, the name

section contains a comma-separated list of names, the dash ("\-"), and a brief description. Using `man` on any of those names will find that man page. (This does not seem to work for pages in `~/man/`.) Other man systems (non-mandb ones) simply create symlinks for each of the names to the one page. (Example: `man kibi` shows the `units(7)` page.)

Use **`col -bx`** to strip control characters from an otherwise text file. Use to create an ASCII text version of a formatted man page. `-b` = printable only, no backspacing; `-x` = no tabs. (See also `man groff_man`.)

Tip: Hyphenation in man pages is often confusing! You can disable it by editing the macro file or in groff settings. Add the following line to `/etc/groff/site-tmac/man.local`:

```
.hy 0 "/ Disable hyphenation
```

Additional Text Formatting Background

Text formatting has not been completely replaced with word processors and the like. Besides `groff`, **TeX** is popular for technical writing. There is even a "wysiwyg" TeX program called `lyx` and popular macro packages such as "LaTeX".

The original `roff` and `troff` produced output for a C/A/T phototypesetter. This first-generation typesetter was very limited and soon replaced by better ones. As some point AT&T updated `nroff` to produce device-independent output files, or "**DVI**" files. This version was called `ditroff` but was unbundled from Unix and was never too popular. *Print filters* exist to convert DVI (and "old" `troff`) files to some actual printer language.

The concept of DVI originated (I think) with the inventor of TeX, Donald Knuth. The DVI file produced by TeX is different than from `ditroff`. This means there are several different DVI formats!

Today the true DVI format is **PostScript** or **PCL**, although some claim that honor for PDF, GDI (Windows), OOD, etc. To print some document you type it in with an editor such as `vi`, then process it through a pipeline of pre-processors and then a text formatter to produce a (not-really) DVI file. This in turn is translated to (say) PostScript via a print filter and can then be printed.

Other PostScript utilities extract parts, to do n-up printing, and convert formats (a.k.a. print filters). Type "`ps`<TAB><TAB>" for a partial list.

Lecture 6 — Software Development & Tools: RCS, Git, and other SCMs/VCSs

Every system administrator must know something about software development. You must:

- Be able to build some application from source code, as not all software is available in packages.
- Be able to generate, read and apply patches.
- Know how to use a version control system.
- Know how to write descriptive changelogs.
- Know how to search a repository's logs for keywords within time frames.
- Know how to create scripts in shell (and preferably in a second language as well, such as Python). Scripts are essential for automation and other tasks.
- Know how to read a stack trace and how to report relevant errors to your software support contact. As an SA, you don't need to know how to fix software bugs but you should be able to understand some of the errors that occur and be able to communicate effectively with developers.
- Know how to test software (mostly scripts) you build.
- Monitor continuously and log all errors. If something goes wrong, you will need to know exactly what and when, or developers cannot be expected to figure out what happened. (Adding logging and metric collection statements to code is the developer's responsibility; system admins need to know how to view and use that information.)

There's more to know, but I hope you get the idea that you need to know this stuff. Sadly, very little of it is found in system admin books or even in online tutorials; mostly there are system administration tutorials for programmers but not the other way around.

DevOps

Today, system admins must also work closely with developers. Indeed, some companies only hire sys admins who can do development as well. The job title for one with those skills is ***DevOps Engineer***. Even if not doing development themselves, system admins must set up the tools for developers, and in many cases must build, configure, and deploy applications and services from source code. You may need to create installable packages and add them to a repo. You therefore need to understand the workflow and concepts behind software development.

Version Control Systems (VCS)

Source Code Control Systems are also known as ***Revision Control Systems, Version Control Systems, Content Management Systems, Source Code***

Management systems, Software Configuration Management systems, and no doubt other marketing buzz-terms. These systems also provide change tracking and logging features, and are often tied into bug-tracking systems. Change tracking is valuable to see exactly what changed over time, and to be able to reproduce old versions if needed.

Some of these systems also include other features, such as the expansion of keywords in the files, during *checkout*. You include these keywords (say in comments), and have them automatically show the version number, date of last change, author, and other data.

These tools are all similar: the files are kept in a ***repository***, or ***repo***. To view (or edit) files, you **check-out** the files, project, or entire repository. This makes copes of the files you can view, use, or edit. If making changes, you **check-in** the modified files (with a log message). The result of a check-in is referred to as a *commit*. (You can think of a repo as a collection of commits.)

SCCS was the first (and still in use; it is the only VCS required by POSIX/SUS), then came **RCS** as an improvement. To support distributed software projects with many authors, CVS (*concurrent versioning system*) became popular. CVS allows one to set up a single code ***repository*** that all authorized developers can access across the Internet. Each developer ***checks out*** a complete copy of the code, works on it, and periodically ***checks in*** any changes they have made. The check in process is actually a ***merge*** operation, as other developers have been working on the same set of files. (Merging means to adjust your changes to those already made, before commiting them.)

Using RCS

For one (or a few) person projects on one server, RCS works well (esp. for document tracking and general SA use) and is easy to use. **(Show RCS Demo.)** Here is a complete example:

```
cd ~; mkdir RCS; vi nusrs;
ci -i1.0 nusrs; co -l nusrs; vi nusrs; ci -u nusrs
```

RCS looks for a repo in the current directory, a subdirectory named `RCS`. There is no option to name a central repo to use with multiple directories, but you can name both the file and the path to file within the RCS repo on the command line. (Personally, I just create symlinks in various directories, named `RCS`, that all refer to a single repo.) Details are found on the `ci(1)` man page.

Another RCS feature is the **expansion of *keywords* in a file** during check-out. Keywords all look like "`$word$`" and these expand to "`$word: text$`".

(*See man pages for* ***ident*** (keyword list)*,* ***rcsintro****, rcs, ci, co, rcsdiff*, and *rlog*).

Good idea: `mkdir /etc/RCS`, then `ci` and `co` all changes, to allow easy back-out and change history.

> Another good idea is to create a shell script (usually called `xed` or `vir`) that checks out the file, fires up `$EDITOR`, then checks the file in. This is similar to how `visudo`, `sudoedit`, or `vipw` work.

SCCS and RCS only allow a single user to have a file checked-out for editing, at any one time. (Theu use lock files for that.) **CVS** and modern CVSs allow multiple users to edit the same files, simultaneously. It is expected that most of the time, there will be no problems or overlap, and the changes can be merged automatically. Occasionally, a problem does arise; the user who checks-in last must manually resolve (merge) the issues. That can be difficult.

Despite its popularity, CVS has some problems. It internally uses RCS, and that deals with changes on a per-file basis; a single fix that means edits to multiple files is not tracked correctly and cannot be undone in one operation (the Sys Admin must figure out each file and roll them back to the correct old version). Also, changing a file's name or permission is not tracked.

> While RCS development has stopped, Jörg Schilling ("Schily") continues to enhance and maintain SCCS at sourceforge.net. It is worth another look if you've been using RCS for a long time.

Newer systems address many of those issues. A popular one is *subversion*. Other VCS tools include Mercurial, `git`, Bazaar (written in Python), and some commercial systems.

> **Note!** SourceForge.net is now owned by Dice holdings (dice.com), and they have come up with a disturbing way to monetize that site. **Projects that appear abandoned for some months are taken over, and the software modified to install crapware and spyware.** As a result, many projects are moving their main repos to Github.com or elsewhere. However, the original cannot be taken down (Sourceforge keeps it up), so you should be careful that you obtain software from the *real* project home, if some webpage or link states it is hosted at Sourceforge. For the projects that remain at Sourceforge, be careful to not install any stuff you don't want; if you just click "next" without reading, you will end up installing the ask.com toolbar or worse.

For working privately on projects, any VCS is good enough. I use the simple RCS to manage admin files and shell scripts. But most software projects have more than one developer, all of whom must work together as a team. Originally this meant using a centralized VCS such as CVS or subversion (`svn`), the two most popular.

In a ***centralized*** **VCS,** there's only one repository. Only those with "submit" access can make changes, saving their work frequently as "checkpoints". Other developers must submit patches to someone with such rights.

It's awkward to work privately on a shared project when you don't have submit privilege, because it means building up a huge patch (set of changes), without any saving any checkpoints (intermediate steps), then submitting the whole thing at once as a surprise on the other developers, behavior referred to as "dropping a bomb".

In a ***distributed*** **VCS (a DVCS),** every user starts with a fully legitimate fork or *clone* of the project's repository. The one 'official' repo is purely a matter of social convention. (It can be hard to know about other repositories; for `git` projects, you can publish on sites such as github.com, Launchpad.net, BitBucket.org, code.google.com, and sourceforge.net.) Developers can check in their changes to their own repo as often as they want. The maintainer of the official repository can frequently *merge* the changes from the other developers' repos.

DVCSs include **Git** (Used for Linux kernel, but hardest to learn), **Bazaar** (`bzr`, easiest to learn) and **Mercurial** (all commands start with "`hg`", the atomic symbol for mercury); all of these have wide support (e.g. Google). Git is the most popular.

Working with Git

Due to its immense popularity, **a system admin must know something about Git and how to perform a few common tasks with it.** For example, much of source code is fetched, not from a tar-ball, but from a Git repository. Red Hat's new OpenShift cloud platform requires using Git to configure virtual machines. However, Git is more complex than other version control systems.

> How popular is Git? Until 2017, Microsoft used their own *SourceDepot* DVCS for Windows code. MS uses many different VCSs for different projects, including their own Team Foundation Version Control, but they are migrating (2/2017) to Git. Their Windows code base is so huge they needed to make changes to Git to support it, which should eventually find their way into the Git main stream. (Read the interesting story at Ars Technica.) Indeed, recently Microsoft bought GitHub as they now depend on it.

With Git, there is one repo per project. To make a new project is as simple as it could be:

```
mkdir myproj.git
cd myproj.git
git init
```

(Naming the directory *`something`*`.git` is common but not required.) These commands create a sub-directory "`.git`" in the project's directory (the *working directory*), containing all the repo's data. The working directory contains the currently checked-out version of the project's files.

A Git repo contains a linked list (really, a graph) of commit objects. Each *commit* is a complete snapshot (logically but not physically; Git won't waste disk space) of the working directory at a given point in time. Each commit also contains a commit message, a timestamp, a reference to the previous commit, and the committer's name and email address.

Each commit object has an ID number that uniquely identifies it throughout the world. This feat is accomplished by using the SHA-1 message digest (hash), and results in a long (40 digit) hex number. When working in your Git repos, you can use just the first six or more digits, as long as that is locally unique.

After editing/creating/deleting/moving files in your repo, you "check in" files from working directory into the repo; that results in a new commit object. In Git, a "check in" is a two-stage task. First, you must **add** the modified files to a *staging area* (originally called the *index* or *cache*):

```
git add .  # Adds all files in the CWD, recursively
```

When the staging area has the exact files you want to include in the commit, you make a commit like this:

```
git commit [ -m message ]
```

Git takes the files you added and the other information, and stores that as a new commit object.

In addition to commits, Git also holds references to them. These are handy text labels that refer to commits by ID, so you don't need to enter hex numbers to refer to a commit. There are two types of references: **tags** which always refer to the same commit, and **branches**, which automatically get adjusted whenever you make a new commit.

Since Git commit IDs are globally unique, you can even make references to commits in other repositories (called ***remote repos***), which may be anywhere on the Internet! For example, "`pollock-repo/some-branch`". (For remote repos, Git also needs to store the URL to access it.)

By default, Git creates two references from your first commit. "**`main`**" is the name of the branch (see below). (As of 6/2020, the original name of "**`master`**" is still used. I will use "main" here.) "**`HEAD`**" is the last checked-out commit on the current branch. (`HEAD` is really just a reference to some branch name, which in turn is a reference to the most recent commit on that branch.)

All work in Git is done in branches. Each time you make a new commit, HEAD and the current branch reference get reset to refer to the new commit.

> As you work on your project, you may want to rename, move, or delete files. The old renamed or deleted files cause problems with VCSs, including Git. When you later commit, Git thinks you merely didn't add

those files to the staging area. The result is the old (deleted/moved/renamed) files are marked as modified but not added. If you moved or renamed a file, the new name hasn't been added.

If a file is under version control, you really need to tell the VCS about the file rename/move/delete. You could just use your OS or IDE to move the file, then remove the old name from Git, then add the new name to Git. (If the contents haven't changed much before your next commit, Git will autodetect the file was just renamed so you don't lose any history. It cannot always tell however.)

Rather than use multiple commands, Git provides nice shortcuts that do all the steps needed as single commands. To move or rename a file, use the command "**`git mv`**". To delete a file, use "**`git rm`**".

If you deleted a file from your IDE or directly from your operating system, you still need to tell Git that the (no longer existing) file is no longer under its control:

```
git rm $(git ls-files --deleted)
```

Each commit except the first has a reference to the parent (previous) commit, forming a linked list (a branch). The default branch is named "main". You can easily create additional branches; usually each person working on the common project has one or more branches of their own. The repo also keeps track of the most recent comment object for every branch that has been created. To create and delete branches is simple:

```
git branch myNewBranch  # create but don't switch
git branch -d myNewBranch # delete a branch
git branch # show branches & which is checked out
```

You can easily checkout any branch:

```
git checkout someBranch
```

Checking out a branch means what it says; the working directory is altered to reflect the commit object's version of files. This checkout also resets the HEAD reference. (You can checkout individual files, in case you mess up one and want to restore just that file. In that case HEAD is not reset.) You can create and switch to a new branch in one step with "`git checkout -b myNewBranch`".

You can create branches from other branches. The result is a tree, not a linked list. You can view the tree with:

```
git log --graph --oneline --all --decorate=full
```

(ASCII art version) or "`gitk --all`" (GUI version).

When you switch between branches, Git completely replaces the working directory with the snapshot in that commit object, unless you have uncommitted changes. If there are un-committed changes in the working directory, Git will attempt to merge automatically the changes when you switch branches. If there is a conflict however, Git will abort the checkout so you don't lose the versions of files in your working directory. You need to resolve the issue before switching to a different branch. There are many ways to do that; you can use "git checkout -m *otherBranch*" and resolve the conflict. Then you can switch branches (which then has a new commit object at the tip). It is simpler to commit, then switch branches.

When the changes made on one branch are deemed worthy of adding to a different branch, they can be ***merged*** into another branch such as `main`. Git does this efficiently: the commit objects from the one branch are appended to the other, forming one longer linked list:

```
git checkout main
git merge someBranch
```

If the two branches you are merging have changes made to the same file (and same location in that file), Git gives up on automatically merging, and makes the committer deal with the conflicts by editing the file(s) with conflicting changes. The resulting snapshot becomes a new commit object (after you add), but one with two parents (for each of the two branches). System admins do not generally need to worry about merging and conflicts.

Commit objects can be given labels, called *tags*. A tag such as "version 1.3" makes it easier to locate a particular commit. The tags are maintained separately from the commit objects (but still in a file under `.git`).

Finally, you can have configuration settings. Git has system-wide settings, per-user settings, and pre-repo settings. The settings can include aliases for common but complex Git commands. Before you use Git for the first time, you should set your name and email at least in the per-user configuration (which Git calls "global" from some perverse reason):

```
git config --global user.name 'Wayne Pollock'
git config --global user.email 'pollock@acm.org'
```

Working with other repos

Very commonly, you need to fetch something from someone else's Git repo. The simplest way to do that is to make a complete copy of the remote repo locally. Note that once you've done so, you can refresh your copy with the latest updates from the remote repo. Making a copy of a repo is called *cloning*:

```
git clone someURL
```

The URL can use several different protocols. Some examples include:

```
    git clone
https://github.com/profwpollock/practice.git
    git clone
[ssh://]git@github.com:profwpollock/practice.git
    git clone http://git.wpollock.com/practice.git
    git clone git://git.wpollock.com/practice.git
    git clone file:///var/repos/practice.git
```

SSH is preferred, but you would need to set up SSH keys for that. (Note using "`ssh://`" is optional.) HTTPS also gives security and requires authentication, but you do not need to setup any keys in advance. The other protocols are for public repos that do not require authentication to clone. The `git://` protocol requires a Git server running at the remote end. (***Demo github.com.***)

The various URLs can be saved in a repo and given short simple names. When you clone a remote repo this way, the local repo automatically makes a reference to the remote one, using the name `origin`. These names can be used to refresh your local copy, like so:

```
git fetch origin
git merge
```

("`origin`" is the default name used by fetch, so just "`git fetch`" works.) You can easily add additional remote names. Also, with different options, you can fetch just a given branch or even one commit.

Notice how once you fetched the commit objects from the remote repo, you still had to merge them into your repo. There is a shortcut for these two steps:

```
git pull [remote]
```

(*`remote`* defaults to `origin`.). Similarly, if you have write access to the remote repo, you can upload then merge your changes there using:

```
git push [remote]
```

If you don't have write access, you can create a ***pull request*** instead, and email that to someone who does. This is often done anyway, so the changes can be reviewed by others before merging. (The command would be "`git request-pull`".)

> A datacenter or cluster (enterprise application or service) needs a better solution for tracking changes than individual host journals. Today we would use a ticketing system to track what needs to be done (and the status of such jobs), and then a configuration management solution such as Puppet, with the config file maintained under Git. A change to the web servers, for instance, would cause a new ticket to be generated, then the Puppet config to be updated (the log message should reference the ticket). The exact changes made to all hosts is preserved by Git, and notes about the change are recorded in the ticket. No per-host journal is needed.

Lecture 7 — Building Software: Compiling, Linking, and Loading C Software

Software development can be described (overly simplistically) as writing instructions for the computer, in a human-readable language, that can be translated into the binary ones and zeros that the CPU can understand.

One **difference between *scripts* and *compiled* (or *binary*) programs** is that with the latter, you translate the (human-readable) source to machine language once and save the result. The translator used for this task is called a ***compiler***. Scripts are interpreted as they run; the translator for scripts is called an ***interpreter***.

C programs are (almost always) compiled. For C programs, the translator used is often **`gcc`** or **`clang`**. (Demo **`c99 -c`** `hello.c`. "`c99`" was the POSIX required name, but the C11 version is called "`c11`". The previous version was `c89`, and the original K&R C compiler from 1970 was `cc`. As of 2024, POSIX only requires **`c17`**.) The result of the translation of a source code file is called an *object file* (dot-o file).

It is common to develop software in multiple, mostly independent files (or modules) of source code. Each is compiled independently, to an ***object file***. In a later step, these object files must be combined into a single executable (or occasionally, a library instead). That will be discussed below.

C souce code files have an extension of ".c". C programs usually ***include* "header" files** (`.h` a.k.a. *dot-h* files) that define the API of libraries. The location for standard header files is **`/usr/include`**. (See `suffixes(7)`.)

Not finding dot-h files is a common cause of build failure, one that a Sys Admin should be able to fix. Locate the "missing" header (or library) on your system, or install if missing. Adjust the options to the compiler to make it look in the correct locations:

- Use the C compiler option `-L`*`dir`* to have non-standard directories searched for libraries.
- Use `-I`*`dir`* to use search for header files in non-standard directories.

> If you did a minimal install it is unlikely you have the required development libraries and tools installed. You can either try to compile programs, check the error messages for what's missing, and repeat until you have install everything needed, or you can install all development tools and libraries, using the `dnf group install` feature. View available groups with "`dnf group list -v hidden`".
>
> Packages containing libraries used at compile time are often named "*foo*-devel".

APIs and Libraries

An ***API*** (application programming interface) is a set of functions and specifications that a program uses for communication, drawing graphics, printing, or anything else. Often, a kernel's API is the only way for an application to access hardware resources protected by the OS. The functions of the kernel's API are called *syscalls*. Modern systems provide about 300 of these.

> Linux provides another way for applications to access kernel functionality: *netlink sockets*. Depending on the kernel subsystem, access is through one or both of these mechanisms.

Libraries are the embodiments of (non-kernel) APIs. A library is the actual executable implementation of some API; it is an *archive* of object files. (Libraries are written and translated the same way as all software is.) A library is a collection of object files, each of which contains one or more functions that applications can load and use. An API describes how a library's functions should be used.

A commonly used archive tool for creating libraries of object files is **`ar`**, an ancient tool also used to create Debian ("`.deb`") packages.

Most programming languages include a "standard library" that implement efficiently and correctly many functions needed by programmers. Using the library means that the programmer need not re-implement those functions.

Many developers create their own libraries, used by different parts of their application or for use in other applications.

There are two types of libraries: static link libraries and dynamic link libraries (or "DLLs"). (The *nix term for DLLs is *shared object* file.) Which type you use depends on the type of application you are compiling. With *static* linking of executables, all the code used by the application is copied from the libraries and other object files into the executable. Such a statically linked application only needs the OS to run. (Note that if some function is needed from a library, the whole object file from that library must be copied, not just the one function needed. For this reason, many libraries contain many small object files.)

As the functions are appended to the executable file being built, their memory addresses are determined. The other code that invokes such functions is modified to use the correct addresses for the functions used. This process is called ***link-editing***, or simply *linking*, and is described below.

In contrast, dynamically linked executables don't actually copy the object code from libraries into executables. At *runtime*, the libraries used by the application must be loaded into memory separately. The executable must then be linked to the loaded library, so its functions can be invoked. (This requires some linking to occur at runtime.) This has the benefit of reducing the executable's size, allows sharing of libraries in memory, and allows the libraries to be updated

independently of the applications that use them. Note it becomes the end-user's responsibility to ensure the required libraries are installed and up to date. There are also some issues and admin tasks associated with dynamically linked executables; this is discussed in more detail, below.

Because of the different ways they are used and linked, static (link) libraries are compiled differently than dynamic ones. A good system admin should be able to build static or dynamic executables from source code, and to build static and dynamic link libraries.

> There's also a legal issue: as static linking copies code, any license restrictions (for example, from GPL code) apply to the executable. With dynamic linking, different (LGPL) or no license restrictions apply.

Compiling

Translating C source code files into object files is easy:

```
c99 -c file1.c file2.c ...
```

"`c99`" is the POSIX required C compiler currently (2022). On Linux, that is usually another name for "`gcc`". Soon, the 2011 or later version of standard C will make it into the SUS/POSIX standard. Until then, using `gcc` or `clang` should support the 2011 version of C, as well as the 1999 version. (For GCC, use the option "`-std=c11`".)

The result of this translation, assuming no errors, is a bunch of object files. Each contains the machine code equivalent of the instructions in each source code file. The object files are not complete programs; they need to be combined into an executable, or into a library.

> Developers often talk of activities occurring at either ***runtime*** or at ***compile time***.

Demo using `~wpollock/bin/src/hello.c`.

Linking

As mentioned earlier, one or more object files (including ones from used libraries) must be combined to make an executable, or *binary*. This process is called "***linking***" or "link-editing". You can invoke the linker directly using the **`ld`** command. (See the manual for Gnu `ld`, especially the section on linker scripts and versioning.)

All linking is controlled by a linker script. The main purpose of this script is to describe how the various parts ("sections") in the object files should be mapped into the output file, and to control the memory layout of the resulting process. You can view the default script using "`ld --verbose`".

With C or C++, you use the same tool but without the "`-c`" option to both compile and link in one step. You should specify "**`-o` *`name`***", or the resulting binary will be called "`a.out`" (or `a.exe` on Windows) for historical reasons.

If any of the listed files use functions listed in libraries, those libraries must be listed on the command line *after* the files that use them. To link with some library named "`libfoo.a`" or "`libfoo.so.`*X*", where *X* is a version number, you use the "`-l`*foo*" option. For example, if the program `foo` uses functions from ncurses library and the C math library, you would compile and link this way:

```
c99 -o foo foo.c -lncurses -lm
```

All libraries conventionally are named "lib*something.extension*". When specifying libraries to link with, you omit the "`lib`" prefix and any extensions.

So the command above looks for `libncurses.so` and `libm.so` (or "`libncurses.a`" and "`libm.a`", if no "`*.so`" files are found). Note the linker doesn't look for libraries with version numbers. **The standard location to search for libraries is `/lib` and `/usr/lib` (or `/lib64` and `/usr/lib64`) by default.** You generally don't have to list options to link with any "standard" (included in the language) library.

The compiler (the linker actually) looks for DLLs by name, with a "`lib`" prefix and a "`.so`" suffix. Once found, the resulting executable will contain a reference to the library's "**SONAME**", not its filename. However, this flexibility is never used and the SONAME is always in practice part of the filename.

The SONAME usually includes a major version number, something like "`libfoo.so.3`". If you install `libfoo` via `dnf`, you will likely have the file `libfoo.so.3` but not the file `libfoo.so`. Those are typically installed from a separate package with a name such as "`libfoo-devel`". What used to be generally installed was just a symlink to the latest version of `libfoo.so.`*X*; you could create those by hand if necessary. In more modern systems, the file is different, often a text file containing linker instructions.

At runtime, only the linked filename (set at compile time) is used.

As mentioned above, not all `libfoo.so` files are symlinks to an actual DLL. (All `libfoo.so.`*version are* symlinks to DLLs.) Some `libfoo.so` files are plain text, which contain additional linker commands. (The linker script language is described in the `ld` info page, not the man page; you'll probably never need to know it.)

Demo:

```
cd /lib
file lib*.so |grep $'text\nsymbolic link'
```

When linking, the order you list files and static libraries on the command line is significant. (DLLs resolve any undefined symbols automatically at runtime, so the order DLLs are listed doesn't matter unless you use the GCC linker option "`--as-needed`".)

So if file *A* uses some function from an object file *B* (which may be in a library), then when you statically link, you must list *A* before *B* on the compiler command line. (*Show "make use-hello-static-error" and "make clean; make use-hello-dll-error" from dll-demo*.)

Usually a C compiler such as `gcc` runs several different programs depending on which options are used:

- `cpp` (the preprocessor) is the only program that runs if "`-E`" is used,
- a C language to assembly language translator,
- `as` (the assembler) is used to generate the object files, and
- `ld` (unless "`-c`" is used) to link-edit the object files and libraries into an executable.

An executable file has a format that includes both data and machine language instructions. The modern format (for Linux and Unix) is **ELF**. Many tools understand ELF format and are useful when you need to examine some binary.

The historical use of the `tsort` utility was in a command substitution when compiling static executables to order the libraries on the command line for you. This was because the Unix V7 link editor used a single-pass design for resolving external symbols and only had static linking. `tsort` was used in a pipeline with `lorder` (library order) to determine an order for the object files in an archive library that would allow this single-pass resolution:

```
ar -cru libmylib.a `lorder $OBJFILES | tsort`
```

In actual practice, there were often irreducible cycles, and no one would have wanted the build to be interrupted by this; application developers were expected to deal by adding sufficient duplicate *"`-lmylib`"* flags to resolve the symbols in the cycle!

Archive libraries have had symbol tables for decades, so there is no longer a need for this hack: modern link editors do not search for symbols in a library by reading each object file sequentially.

You can use this to find statically linked executables on your system:

```
for f in /sbin/* /bin/* /usr/sbin/* /usr/bin/*
do if file "$f" |grep -q 'statically linked'
   then echo $f; fi
done
```

Loading

Loading is the process of running an executable. The OS must locate the executable file, read some information from a header within to determine how much memory is needed, what DLLs are needed, and other tasks. For dynamically linked executables, loading may involve some linking too. Developers generally talk about things that occur either at *compile-time* or at *run-time*. Loading occurs at run-time.

> DLLs in *nix systems can be used in a few different ways. The more commonly used way was described above. The other way involves using `libdl` ("`dlopen`", "`dlclose`", ...) to locate and load DLLs at runtime. With this method, the DLLs don't need to be available at compile time; this is useful for apps that take plug-ins, such as PAM. How they are used depends on which compiler options were used to create them.

Libraries are archive files containing some set of object files. Create *static* libraries ("`.a`" files) with `ar` and DLLs with `gcc`. For POSIX C18 and newer, you can generate DLLs with the **`-G`** (or `-B shared`) option. (Older versions of POSIX C had no standard way to do this; some compilers didn't have any option for this at all. As of 2019, there is no standard compiler options to force static or dynamic linking.)

> A static library is just a collection of dot-o files in an archive file. But without an index (table of contents), such archives make it difficult for a linker to know what things are defined within a library. That can causes problems when (for example) a program uses a function defined in `two.o`, and that in turn uses a function defined in `one.o`, and the archive lists the files in the order `one.o`, `two.o`. So generate an index.

An index can be generated with the non-POSIX `ranlib` utility or the `ar -s` option. With such an index, the order of dot-o files within a library won't matter. (*You can view the index; demo "`nm -s /lib64/libm.a 2>/dev/null | less`".*)

DLLs are similar but always contain a header with the index and other data.

> To compile a static binary, first install the `glibc-static` (and `libstdc++-static`) standard C (and C++) library. Then use the `gcc` option `-static`. (Static linking seems harder with `clang`.) To get rid of other DLL (*`xyz`*`.so`) dependencies, you need to link with the static

version of each of those libraries. However, few systems ship with static libraries anymore, and they may not be available from any repo either. In this case, you must build any needed static libraries from source using the `gcc` options including `-static -pie -fpie` flags. Then compile using those static libraries. Building and working with DLLs is described later.

Show dll-demo (unpack tar-ball, `cd dll-demo/; make`).

DLLs are discussed in more detail, below.

Java and other Virtual-Machine Languages

Since it was invented, Java has been among the most popular programming languages. Thus, many tools require Java to be present on a system. Additionally, Java is very popular middleware (software for running enterprise scale applications). This means Sys Admins should know a little about Java.

Java is a compiled language, like C. Unlike C executables, the resulting binary will not run on an Intel-compatible CPU or with *nix or Windows! The compiler makes binaries for a machine that does not exist, known as the ***Java virtual machine*** (JVM). Each real platform needs a JVM program that translates the binary (known as *bytecode*) into machine code for that platform. In essence, the JVM interprets bytecode. One difference with interpreted languages is that the JVM saves the result of translation in memory. This is called *just in time* (JIT) translation and makes long-lived Java applications as efficient as native code.

In addition to a JVM, a standard library provides access to commonly needed functions and services. In order to run Java programs, you need both the library and the JVM; together these are known as the ***Java Runtime Environment*** (JRE).

Java source code files contain one or more (usually one) "chunk" of code, known as a *class*. With Java, every class is saved in its own `.class` file no matter the name of the source file. The compiler may put these `.class` files in your CWD or in subdirectories. Any linking between these classes is done at runtime by the JRE. **The JRE searches for classes in specific locations, mostly specified from the environment variable `CLASSPATH`.** (If unset, this defaults to "`.`".)

Java source files must have a `.java` extension. To compile `Foo.java` into `Foo.class`, use the command "**`javac`** `Foo.java`".

Demo using `~wpollock/java/Hello.java`.

Although Java supports multi-threading very well, the JVM only runs a single Java application. To start a JVM to run `Foo.class`, use the command "`java Foo`".

Because the many files make it difficult to move an application about, Java includes a tool **`jar`** that works similarly to `tar` and creates Java archive (`.jar`) files. Jar files are actually zip archives and contain additional information ("meta-

data") such as author and support information, digital signatures, additional media files, and the default class. Setting the default class means you can execute the packaged application with the command "`java -jar myapp.jar`".

If you configure your Linux system correctly, you can also just double-click a jar file to execute it from the GUI, or just add execute permission to the jar to run it from the command line. (This is true for other types of executables and is discussed below.) (*Demo Hello.java and HelloGUI.java.*)

Microsoft's .NET platform is similar to Java: source programs are compiled into CIL (similar to bytecode; note Microsoft's virtual machine is called CLR), then assembled into portable executables (*assemblies*, similar to jar files). While these can be compiled into native machine code (so can Java), these CLR assemblies are portable as long as you have a CLR available. (A CLR is available for Linux, now blessed and supported by Microsoft, called **Mono**.) (*Demo Hello.cs*.)

Both the CLR and the JRE support multiple source code languages; the runtime doesn't care as long as the result is bytecode/CIL. Popular .NET languages include C# and VB; and Java, Scala, and Clojure for JRE.

Tool-Chains

In reality, several tools are needed to compile software. These are sometimes referred to as a ***tool chain*** (a set of development tools designed to work together). You usually need them all installed. A related term is IDE, or *integrated development environment*. (Other related terms include SDK, or *software developer kit*.) An IDE has a bunch of tools which all share a common interface, usually a GUI.

Automating the Build

Most of the time, a Sys Admin won't need to directly compile or link software. This is because most tool chains include tools to automate the process. (The details described previously are useful when the automatic build fails.)

The most commonly used tool for building C programs is **`make`** (discuss and demo **`makefile`**, Gnu project web resources; mention `make` *targets* and *rules*). A Sys Admin sometimes must "tweak" a makefile to fix some minor problem, such as changes to command options (when the software author used a different tool-chain) or file locations (header or libraries installed in different locations than the author assumed). Adding a "`-j` *num*" option (to use multiple cores to compile *num* files at once) is common.

Other tools in the Gnu C tool-chain include **`autoconf`**, which takes a list of what resources are needed and what software to build, and creates a shell script **`configure`** that examines the local system and creates custom makefiles with the **`automake`** tool. This is the famous "`./configure`" script needed.

While currently the most popular, the Gnu tool-chain isn't the only one available. The LLVM toolchain (including the Clang C/C++/Objective-C compiler) is being used by an increasing number of companies and projects, including Apple and FreeBSD. The collection is licensed under the BSD-like University of Illinois/NCSA Open Source License, a permissive license which allows commercial products to be derived from the LLVM project.

Increasingly popular (2021) is the Mason build system. Mason attempts to be a single, simple build system for all architectures (Linux, Mac, Windows), for most languages (C, C++, Java, Rust, ...). It is much simpler to use than autoconf but perhaps not quite as powerful (yet). Unlike some other language-specific build tools, Mason will check but not install (from the Internet) missing dependencies (Maven and Gradle for Java do that; autoconf for C/C++ do not).

Some tools in common tool-chains include: `make`, `cc` (`c99`), `cpp`, `as`, `ld`, `ar`, `ranlib`, and `strip`. (Mostly you need to use `make` and a compiler; the other tools are invoked internally by the compiler.) Other useful tools (not needed to build software) include `gprof`, `nm`, `objcopy`, `objdump`, `readelf`, `size`, and `strings`. Be sure to have these tools installed.

Reproducible Builds and Continuous Integration, Deployment, and Delivery

"Reproducible builds are a set of software development practices that create a verifiable path from human readable source code to the binary code used by computers." — reproducible-builds.org

This is important, since you may (will!) have to redeploy updated software someday. If the tools, versions of DLLs, the OS, the web, database, or other servers that the code depends on have been updated since then, rebuilding the code may (probably will!) fail. Thus, as part of the build process the build environment must be recorded in detail and kept. So a year or more from now, a virtual machine (or container) can be made with an identical environment, and the code built correctly. To ensure the build environment is correct, first recompile the old version and confirm it is identical with the deployed version. Then build the new version. One way to ensure this is to always build in a virtual machine, and keep the VM images in case you need to rebuild anything.

To have reproducible builds, you need a fully automated build system. Once the developers finish with code and it is time for you to deploy it, it should be a single command you run. With the technique known as ***continuous deployment*** ("CD"), you don't even run a command! A server such as Jenkins watches the software repositories for approved updated code, and

automatically builds and deploys it to some (RPM or other) repo. ***Continuous delivery*** (also "CD") goes one step further than continuous deployment: the human is taken out of the delivery to customer path. If the software makes it through all the tests and code review and packaging steps, it is automatically rolled out onto servers or shipped (or otherwise made available) to customers.

CD is related to ***continuous integration*** ("CI"), when much of the development is also automated: the developer creates the software, locally tests it, and when ready does a commit with Git (or other VCS). A CI server (such as Jenkins again) watches the Git repo, checks out the new commit, runs lots of tests and analysis, and if they all pass, sends the code to a code review server (such as Gerrit). After that, CD automation takes over.

> The software tools, configuration files, and scripts used for builds are also kept under version control, tested, and reproducibly deployed. So in the future one can rebuild the code using the exact same compiler, libraries, etc., as initially. This is part of *infrastructure as code*, related to reproducible builds.

Installing from Source Code

Using packages has become the standard way to install most software today, but it's not as good a method as using source code in many cases: Binary packages may not be installed where you want them (e.g., `/opt` versus `/usr/local`), whereas with source code you can chose where to put stuff.

Having the source code means having the ultimate documentation, and the ability to tweak the code or to apply patches. A package may not be available for your system (or may be an older version), or may be in *rpm* format when your system uses a *deb* package database. (Not all packages are written correctly, and thus may not be translatable from one format to another!) Some security and other tools and utilities need to be compiled as static binaries (example: `sln(1)`).

Even with source code, you generally do the building and compiling on a test machine. When correct, you typically build a package and add that to a local repo, then update the configuration management system (e.g., Puppet) to roll out the change to all applicable hosts.

> **Version numbers** for utilities, or any program built from source, usually follows standard package numbering guidelines:
>
> *major.minor.release-patch*
>
> Where *patch* generally refers to a distro/vendor's revision of a standard version. For example, Gnu "`coreutils 5.3.0-20`" is Gnu source version 5.3.0, with some vendor's patches 1 through 20 applied. Like the version number, sometimes the patch is two or three numbers.

tgz (tar-balls): (Demo `recode`, `bzip2`, `sudo`.) A compressed `tar` archive of source code is called a *tar-ball*. These are unpacked and built on your system.

`tar` review: **`tar -c|t|x -v -z -f file files...`**

To unpack a tar-ball, use **`tar -xf file`**. This will (usually) create a subdirectory with the same name as the tar-ball minus any extensions. After downloading and unpacking, `cd` into this subdirectory and look for the package documentation. This is typically one or more files with names such as `README`, `INSTALL`, ..., `doc/*`. (The README and/or INSTALL doc will often list prerequisite libraries you need to install for this software to build correctly.)

Most source packages today use the Gnu "autoconf" build system, which results in a simple installation procedure. All such source tarballs use the exact same 3 steps (5 if you count downloading then unpacking the tar-ball, 6 if you check integrity, MD5 or GPG, more if you count reading the docs):

```
./configure;   make;   su -c "make install".
```

Note: **only the last step requires super-user privileges**, and **only** if installing to system standard directories. On some systems, you should use `sudo` instead of `su -c`. (Also see the previous discussion (page 53) for PGP/MD5 integrity checking.)

The `configure` script usually takes arguments that modify the generated makefiles. Try **`./configure --help`** to see what options exist. If there are none, or there is no configure script, there will usually just be a `makefile`. You may wish to edit the `makefile` with `vi` to make changes such as where to install; you should at least look it over to see what comments and build targets it contains.

> The Gnu autoconf build system is not the only one, just the most popular. Python programmers generally prefer scons. Cmake is a cross-platform system; it generates makefiles on *nix systems, Visual Studio Solution ("`.sln`") files for Windows systems, etc. Note that Sys Admins don't care; the choice of tool-chain is up to the developer. You just need to have the correct tools installed.

Final steps include making sure the documentation (man pages, etc.) is installed, any new libraries are correctly installed and don't break existing applications (package installers generally determine this automatically), updating the configuration files (for system services) and boot scripts (say via `systemctl`), and notifying users of the system update.

Also the SA must **configure `syslogd`** and **log file rotation**, and **security** (PAM, firewall, `hosts.allow`) for newly added software. Also **update the journal**!

In larger organizations, such built software in put into a package, which is then uploaded to a local repo. This makes it easy to install the software on many systems.

To see what files are installed from a tar-ball, use this four-step procedure:

```
find / | grep -v -e ^/proc/ -e ^/tmp/ > ~/pre-list
   install software
find / | grep -v -e ^/proc/ -e ^/tmp/ > ~/post-list
diff pre-list post-list > pkg-installed-files-list
```

You can install a source package if you prefer to have the software built locally, or with custom options, and avoid the hassle of building and deploying tar-balls. To do this, create an *RPM build* environment (see below), typically `~/rpmbuild/*`. Then you download the source RPM (`*.src.rpm` or sometimes `*.srpm`) and install it. This will place the source code files (typically a tar-ball) in `SOURCES`, and the package spec file in `SPECS` (of your build environment). **The `spec` file contains detailed directions for the `rpmbuild` utility to compile and deploy the software.** Details are described below, but briefly, to build a binary package from a source package, use:

`rpmbuild -bb` *`packageName`*`.spec`

This will compile the source code, assemble a binary package, and put it in `RPMS/i386`. You can now install or deploy this binary package as normal. Note you can edit the spec file to change the configuration options before building the binary package. This is often easier than working with source code directly.

Note some tar-balls support "`make distclean`" or similar `make` targets to un-install.

Popular source code repositories include GitHub.com, sourceforge.net, ~~and code.google.com~~. You can browse these to find interesting software that you can download and build, for practice.

Trouble-shooting Compiles

Installing from a package will automatically install required dependent software, including tools, header files under `/usr/include`, and libraries in `/usr/lib` (or `/usr/lib64`) and `/lib` (or `/lib64`). But this doesn't happen when you install from source! **You must manually make sure the required development tools and other software is installed.** The easy way to do this, is to install everything, including all development packages (usually "*`name`*`-devel`"), on the system you use to build and test software. Otherwise you will get "*command not found*" or other errors when `make` tries to build and install software. If that happens go and install the missing tool packages. (**Use `dnf provides` *`file-name`* to find the package** you should install.)

If you have all the tools but don't have the required header files (or libraries), the compiler will show "not found" messages for them. Missing header files will cause a cascade of error messages, so **pay attention to the first one** that says what is missing! Here's an example of failing to install the PAM development package when building some software that needs PAM:

```
osdep.c:71:31: error: security/pam_appl.h: No such file or directory
```

To solve this sort of problem, first search your system to see if the `.h` file is there but possibly in a different location. (Note the default location for header files is "`/usr/include/`".) If not, use:

```
yum provides /usr/include/security/pam_appl.h
```

to get the package name ("`pam-devel`") and install it, or else find the source for that and install it. When you restart the compile, it will either work or report something else is missing, in which case you repeat the process to install that.

> When compiling C programs, tons of warning messages are normal. While it is recommended that developers clean up their code so that it compiles without warnings, few take the time to do so.
>
> Some of these warning might be about uninitialized or undeclared symbols. A *symbol* is just a name for a function or variable, and such warnings can usually be ignored.

Deployment: Occasionally an SA will need to distribute software or data. You can package your stuff as files in a ***tarball***, installed with:

```
./configure; make; su -c "make install"
```

or as a source or binary *package*.

Building packages is a requirement for various certifications. The Linux Standard Base (LSB) has standardized on RPM packages, which is use for all SuSE and Red Hat (and related) distributions. Other formats include: Solaris (*IPS,* or the older `.pkg`, really SysV packages), Debian (`.deb`), Ubuntu Snappy, and a few vendor-neutral ones (OSGi bundles, NIX, SMART, etc.).

Debian packages have *control* and *rules* files, not *spec*. Use `dpkg-buildpackage` or the simpler `debuild` command to create packages. (Snappy uses YAML *metadata* files to build snaps.) Unix SVR4 packages ("`*.pkg`") can be built on Solaris with `pkgproto` and `pkgmk` (or the newer FOSS tool `pkgbuild`, which takes a `spec` file).

> In today's cloud-based IT landscape (how's that for mixed metaphors?), you may need to apply a patch to some older code and redeploy it. The problem is the various libraries and build tools may have all been updated since then, so the build may not work at all.

A tool called Vagrant is sometimes used to help with this issue. Vagrant configuration files can specify exact versions of everything used to build the original software (in essence, specifying an image for a computer, virtual machine, or container). With these, it is easy to create a virtual machine/container that is identical to any past version. In addition, Vagrant can use configuration management software such as Puppet or Chef to ensure the resulting system is configured identically to the original one that worked. The vagrant config (and config manager's config) file is text and can be put into a file versioning system such as Git.

Building RPM Packages

The way to build RPMs has changed over the years. You need the files to install or the source code to be compiled, any required libraries, and the appropriate tool chain. In either case, you then need a "spec" file that provides all the information needed to compile and build the software to be included in the binary RPM, as well as information about the package itself (which is just copied into the binary RPM). The `spec` file controls the package building process.

LSB mandates the RPM package system. To avoid forcing all vendors use RH's tools, the LSB has created `makelsbpkg`, which does the same thing as `rpmbuild`. Since other vendors ignore that requirement, nobody cares about this.

To start, be sure to install the `rpmdevtools` (and `rpm-build`, `rpmlint`, `rpm-sign`, and possibly `mock`) packages. Then, from your non-root home directory (**never build packages as root**), run **`rpmdev-setuptree`**. This creates a hierarchy `~/rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS}`. It should also create the file `~/.rpmmacros`. In that file, you can set parameters for the build so you don't need to specify them every time. You can edit this file and add a line similar to this:

```
%packager Wayne Pollock <pollock@acm.org>
```

You can also add other lines. For example, to have your packages digitally signed, add the following lines (using *your* GPG key ID):

```
%_signature     gpg
%_gpg_name      779190A0
```

You need to copy the source (typically a tar-ball) into the `SOURCES` directory. Then copy or create a spec file in the `SPECS` directory. **To create a spec file**, use the command `rpmdev-newspec`. (Or use `vim` *`name-version`*`.spec`, e.g. `foo-1.0.spec`.) This creates a template spec file you can easily edit.

Spec files are text files with blank lines, comments, and RPM *directives* (some of which span multiple lines) in *sections*. Directives consist of a case-insensitive *tag name*, a colon, and a value (with optional whitespace around the colon). Sections are marked with "`%name-of-section`". Some sections are required and others optional. The order of sections matters.

(A common gotcha is using "*%something*" in a comment; `rpmbuild` will complain about duplicate or unknown sections. You can use "`%%`" in a comment if necessary.)

Directive values can use previously defined *macros* with the syntax "`%{name}`" ("`%name`" is legal too but rarely used.) The syntax "`%{?name}`" means to use the macro only if it exists; this prevents error messages with a directive such as "`Release: 1%{?dist}`". You can define your own macros with "`%global name value`"; ("`%define`" works too but is slightly different). The following macros are built into RPM and can help allow you to place your files in the right locations:

```
%_prefix          /usr
%_exec_prefix     %{_prefix}
%_bindir          %{_exec_prefix}/bin
%_sbindir         %{_exec_prefix}/sbin
%_libexecdir      %{_exec_prefix}/libexec
%_datadir         %{_prefix}/share
%_sysconfdir      %{_prefix}/etc
%_sharedstatedir  %{_prefix}/com
%_localstatedir   %{_prefix}/var
%_libdir          %{_exec_prefix}/lib
%_includedir      %{_prefix}/include
%_oldincludedir   /usr/include
%_infodir         %{_prefix}/info
%_mandir          %{_prefix}/man
```

More details can be found at Packaging:RPMMacros.

(*Show sample spec file resource.*) The spec file allows you to specify what your package depends on (requires) and what it provides. The "`Provides:`" tag is in addition to all the files installed; it permits you to install a *virtual package* such as "`mta`", when any one of several will do. Other packages can depend on `mta` instead of `sendmail` or `postfix` directly. (You can also specify a "`conflicts`" tag, so you don't install too many such packages.)

You can specify scripts to run before and after the package installs and un-installs (so, four scripts). Pre-install scripts often create users and groups,

and create missing directories. Post-install scripts will often set permissions of files and directories, create initial config files (possibly saving older ones first), run `alternatives`, enable services, and so on. The uninstall scripts run the same tasks in reverse, removing users, disabling services, etc.

Besides the obvious stuff to edit at the top, you need to add the complete pathname of each installed file at the bottom. There are some "RPM macros" which can be used to simplify this file. See the RPM guide or the shorter (but more readable) Creating [RPM] Packages How-To, both at `fedoraproject.org` for details.

RPM spec files can contain dependency information. If your package depends on package1 and package2, add the line:

```
Requires: package1, package2
```

If the "`%files`" section is empty (you still need the "`%files`"), you have a *meta-package*: installing this package only installs packages 1 and 2.

Note that while RPM can detect missing required packages (or package conflicts), it won't automatically install them; `yum` or `dnf` does that.

The spec files can also contain *triggers*, scripts that run when other packages are added, removed, or updated. So if package foo needs configuration depending on which MTA you install, a trigger can handle that. (The man-db package has this, to rebuild the man page database whenever packages change files under `/usr/share/man`. This is more efficfient then running `mandb` every day via cron.)

Create RPM packages with `rpmbuild` *options spec-file*. You can build source or binary packages, or both (`-bs`, `-bb`, `-ba`). You can specify "`--sign`" to have the packages digitally signed. (Note, the `--sign` option is supported by an `rpmbuild` add-on, from the `rpm-sign` package. If not installed, you don't get an error message, but you don't get a signed package either.) If there are errors, use `rpmlint` *`spec-file`* to trouble-shoot.

To ensure your packages will build properly on a vanilla (Fedora) system, and not just your system with all the extra packages you have probably installed, use the `mock` tool. This runs the build in a chroot environment with just the standard minimal system packages. Use it like this:

```
usermod -a -G mock auser  # one time only
... # create source RPM, foo-1.0-1.fc15.src.rpm
mock -r fedora-15-i386 --rebuild \
../SRPMS/foo-1.0-1.fc15.src.rpm
```

To sign an RPM, you first need a GPG key used to sign all your packages (and optionally, the yum repo meta-data). To create one, run `gpg --gen-key` and follow the instructions. Make sure to make a backup copy of your keys off-line! In general, there is one key used to sign all the packages in a given repo, and not one key per package. This implies the repo admins have successfully (re)built all the software in that repo.

> The comment for the key should be "*name-of-repo* package signer", or something similar, if you use your real name in the key. But I recommend using a key with a name set to the repo's name ("`myrepo`"). You can use that same key to sign the repo itself (actually, you sign the repo's XML data file).

In most cases, an RPM package is used to install new repos. Such a package will typically install the key used to sign packages from that repo.

Once you have the public and private keys created, you should be able to see them by running `gpg --list-[secret-]keys`. Let's assume the key is named "`Software Packager`". In order for `yum`, `dnf`, and `rpm` to use your key, you'll need to import it into the RPM database. First, export the public key to a file:

```
$ gpg --export -a 'name-of-repo' \
    > RPM-GPG-KEY-name-of-repo
```

This is the standard name of a yum repo's package-signing public key. Now, import it into the RPM database:

```
# rpm --import RPM-GPG-KEY-name-of-repo
```

> A little-known trick is that you can download a repo's key in a file into `/etc/pki/rpm-gpg/`, and yum/dnf will automatically import the key.

To view all keys imported into RPM, use "`rpm -qa gpg-pubkey\*`".

To tell `rpmbuild` to use this key, add the following lines to your `~/.rpmmacros` file:

```
%_signature gpg
%_gpg_name name-of-repo
```

When you run `rpmbuild` with the `--sign` option, it will generate a signed RPM package instead of an unsigned one. But if you have an unsigned RPM already built, say "`foo-1.0-noarch.rpm`", you can add a signature like this:

```
$ rpm --addsign foo-1.0-noarch.rpm
```

You can add the package to your own (local) repo. Copy the public key file to the root of the repo, so clients can easily fetch it; usually, such keys are put

in files named "`RPM-GPG-KEY-name-of-repo`". (Creating local yum repos was covered in the previous course; you can also see this guide to creating yum repos.)

Patch Files [*See also "Patch Management" from Admin I course.*]

A *patch* (or *patch set*) is a file that describes a change to one or more other files. It is often easier to distribute patches than complete new packages/bundles. In addition, patches can be used to add features or to support unusual configurations. Linux kernel features are patches to the kernel source. Third party enhancements and bug fixes (such as the Pine Maildir patch) often cannot be obtained any other way.

> Patch files, usually with annotations, can be sent to developers to suggest changes to their code. In this case, the patch may be called a ***pull request***.

Patches are created with `diff`, usually with Gnu `diff(1)` as:

`LC_ALL=POSIX TZ=UTC0 diff -Naur old-version new-version`

This type of patch updates text (usually source code) files. The options and settings shown ensure correct comparisons, and puts all timestamps in UTC. The patch file contains many "chunks" of `diff` output, for one or many files. Most other text in a patch file is ignored.

> It is often easier to distribute patches than complete new packages/bundles. A new RPM package type "**deltaRPM**" has been defined as a type of "patch" (*sparse package*) that depends on the original version, and updates only the files that are different. These *sparse packages* have the extension "`.drpm`". They contain binary patches (created with a variant of `bsdiff`) to the previous version of that package, which must be installed first.
>
> Once a deltaRPM has been downloaded, it is applied to (a copy of) the original package on your system, to create the new package. Some sort of checksum (a digital signature, or MD5 hash) is used to verify the package integrity. If the old package isn't found (or isn't valid), or the patched package isn't valid, `yum` goes ahead and downloads the full version. While downloading deltaRPMs can be fast, patching and validating the RPMs can be slow.

To apply such a patch, `cd` to the directory containing *old-version*, and run the command "**`LC_ALL=POSIX patch -Zp1 <patchfile`**". (The `-Z` says assume UTC for timestamps, and "`-p1`" says to strip one leading directory from pathnames used in the patch file. These options usually work best.)

A patch can be *incremental* or *cumulative*. Windows service packs (and Solaris kernel patches) are cumulative. Linux kernel patches generally are incremental, so you must apply them in the correct order to upgrade a very old version.

> The 2.6.36 version of the Linux kernel currently consists of approximately 13 million lines of code across over 33,000 files. Patches are approved at an average rate of 5 per hour.

Often, local changes often make the line numbers in the `diff` output a bit off, and modern `diff` can include a few lines of context before and after the line(s) to be changed. Doing so allows the `patch` utility to "look around" for the lines to change. Larry Wall (inventor of Perl) created the `patch` utility, which can read most types of `diff` output and find the correct lines to update. (The "unified" format is preferred today.)

The `patch` tool also has other options and features, including a "reverse" option to undo an applied patch, and the ability to have a single patch file update many different files. It can even automatically use RCS to `co`/`ci` changes.

> When an RPM spec file includes a patch directive, note that the patching is done before any compiling, so any messages about applying the patch will likely scroll away (due to all the compiler warnings) before you see them. Rest assured, if the patch failed, the build would stop at that point.

Patch File Format

Patch files contain one or more `diff` outputs ("chunks"), concatenated. They can contain other text ("junk") at the top, bottom, or between `diff` outputs; junk is ignored by `patch`. So you can just pipe in a saved email message with a patch in it, headers and all, and `patch` should still work.

The "junk" in front of each `diff` output section in the patch file can also have two lines with special meaning to `patch`: "**`Index:` *`filename`***" and/or "**`Prereq:` *`version`***". The first names the file the `diff` applies to (some non-Gnu `diff` output doesn't contain the filenames when using "`-c`" option, so you can add them in later this way). The second is usually added in front of the `diff` output for the special file `patchlevel.h`, which should just contain a version number on a single line. The `Prereq:` line makes `patch` check that the file contains the correct (current) version number, and prompts you (rather than reject that change) to abort or continue if you attempt to apply a patch to the wrong version.

The "`-p` *`num`*" argument to `patch` is the only tricky part. It strips away *num* levels from the pathnames in the patch file. This is useful when the patch file was created in (say) directory `/x/y/z` but your code is located in directory `/a/y/z` (use `-p1`), or if the patch was created for files in `/a/b/z` (use `-p2`), or `/a/b/c` (use `-p3`). If your code is also in `/x/y/z`, then use `-p0` to use the full pathnames of the files in the patch file. The value used here will depend on your current directory when you use the `patch`

command. (It is easier to understand than to explain; just play around with patch until -p makes sense.)

Even with the context lines, sometimes `patch` fails to find the old chunk of text in a file to patch. This can occur if the line numbers are too far off (possibly from other patches to the same file), or if comments or minor formatting changes make the source look too different from what `patch` expects. Any parts of the patch that failed to be applied are saved in `*.rej` files. A human often succeeds to locate the code to change when `patch` fails, so you need to try to apply the rejected bits manually. (**Show patch demo web resource.**)

On proprietary systems, you rarely get source code to patch. Instead, the patches are for non-text files (so `diff` isn't used to create them). Often special tools are used to read such patch files and apply them. Solaris < 11 used patches this way.

Today many applications are simply replaced in toto with a new version, rather than use patches. Solaris is an exception since versions of any standard application will not change except between Solaris versions, so any bug fixes use patches.

Binary (non-text) files can also be patched using tools to create and apply binary diff files, such as `rdiff`, which works well even on very large files:

```
rdiff signature file.old file.sig;
rdiff delta file.sig file.new file.delta;
rdiff patch file.old file.delta file.patched
```

You can also make binary patches with `bsdiff` (and apply them with `bspatch`) more easily. However, this tool requires memory up to 17 times the size of the original file! A variation of `bsdiff` is used to create deltaRPMs (`.drpm` files).

Managing and Creating Shared Libraries, a.k.a. Dynamic Link Libraries (DLLs), Shared Objects (`.so` files)

The old way to build software was to have a *static link library* (say `name.a`) available when compiling the code. This results in ***static-linked*** executables that don't depend on any other files to run, but they are much larger since each must include the same code (the standard C library, standard GUI toolkits, etc.) To use the newest version of some library all the statically linked software that uses it must be rebuilt. To build a static library, use **`ar`**: `ar -rcs mylib.a a.o b.o`

The alternative is to build software that looks for ***Dynamic Link Libraries*** (DLLs) at run-time. Such executables are small and run quickly as the bulk of the code (in DLLs) is already loaded in memory. (They do take longer to load however.)

DLLs on Unix/Linux are called *name*`.so` (***Shared Object***) files. Links from *name*`.so.`*major*`.`*minor*`[.`*release*`]` for *name*`.so.`*major* (and often *name*`.so` too) exist so a program can request a specific version only, the latest with a given major number, or just the latest without regard to versions. **DLLs with the same major number are *supposed* to be compatible.** Thus software often depends on *name.so.major* rather than *name.so*. (In many cases, *name.so* isn't even available.)

DLLs allow for much smaller programs (otherwise, each program must contain a copy of the common library code). Also, an update to a DLL instantly (and usually painlessly) updates all software that uses it. However, it takes longer to launch a program that uses DLLs, especially if those DLLs aren't already in memory.

(An important reason why so much software today is dynamically linked has to do with software licensing; more details in the next section.)

Issues with DLLs

> "DLL Hell," once a common event on Windows though now fortunately rare, occurs when programs tried to use a DLL, only to find that it is either not on a system at all, or that the version on the system does not match the version that the program expects. Today, it is common to use *containers* to avoid this problem. However, you then get back the problems of needing to update all copies of buggy DLLs, and the wasted space from having many copies.

The two biggest issues with DLLs for SAs are (1) correctly build and install DLLs *before* any application that may need it; and (2) update the system DLL cache and configuration, so the new DLLs can be found. (See *Managing DLLs* below.) If the needed library isn't available or has been updated since you last tested the application, that application could fail to run.

Another major issue is that when updating some DLL, every application and daemon that uses it must be retested before deployment on production servers. This sort of testing is known as ***regression testing***.

A regression test makes sure that something that used to work before the update, still works after the update.

As you update your system, you should add the tests of new stuff as additional regression tests. There are tools (including simple shell scripts) that and automate the running of such tests. This is important since manual testing may miss something, and in any case, a large test suite may take a long time to run.

> You need to run tests whenever anything is updated or changed, not just DLLs, including the compiler and other tools used for building software.

It is very hard to force static linking on modern systems; extra compiler options must be used and the `lib*.a` libraries are often not installed by default. **Static linking is useful for quick launch programs and for security (forensic) tools (where you don't trust the DLLs on the system).**

Your DLL may not be the one you think it is! Internally, DLLs have a ***SONAME*** (*shared object name*). Usually this is the same name as the filename including the major version number. It is important to realize that **executables generally have dependencies based on SONAME** and not the filename or package name. (This isn't always true; see below.) If you're having a dependency problem when installing a package and you think the stuff is there, check the SONAME of the library you actually have installed against the one used by the executable:

```
objdump -x somelibrary.so.version | grep SONAME
lddtree some_executable | grep somelibrary
```

(Note! Running the older `ldd executable` command may cause *executable* to run in order to see what DLLs it loads. So, **don't run `ldd` as root, or on an untrusted command**. There are other ways, not as convenient, to inspect an executable if necessary, but the best way is using `lddtree`, which does not run the executable.)

If some RPM doesn't install due to an unsatisfied dependency for a DLL and you are certain you have that DLL installed, you can use the "`rpm --nodeps`" option to make it install anyway.

Some software is not compiled to use a SONAME (and locate the DLL at runtime), but instead to use a specific file name. (***Not*** a good idea!) This usually shows with the error that "`libfoo.so`" is missing. This name is called the ***link name*** (the SONAME without any version numbers) and is used by the compiler to locate the DLL. Once found, the compiler records the SONAME of the DLL in the compiled binary ("`libfoo.so.`*`version`*") and the link name isn't needed to run the software.

> If you have the DLL `mylib.so.1.2.3` as part of a package `mylib`, you should have the following symlinks to that DLL by the package: `mylib.so.1` (the *soname*), and sometimes `mylib.so.1.2` as well. To get `mylib.so` (the *link name*), you usually must install the package `mylib-devel` or `mylib-dev` too (depends on packaging system used).
>
> It used to be the case that the "*-devel" package merely installed another symlink, but modern systems actually install a linker script for the link name file. Modern linking and loading are much more complex than in the past, due to changes made for security and performance.

But, if software was compiled to use a filename instead of the SONAME (bad idea!), then at runtime the loader looks for that file and doesn't try to locate a DLL by SONAME. In that case, you usually just **install the "`libfoo-devel`" package**. Or, you can add the missing symlink yourself: `ln -s libfoo.so libfoo.so.1`. (As mentioned previously, a symlink may not work for modern software.)

> It's not only the case that the developer foolishly used a filename of the link name (rather than the SONAME). Sometimes the developer uses compiler options to have the executable depend on a specific version such as `libfoo.so.1.2` or worse, `libfoo.so.1.2.3`. I have run into all these issues in the past. In such cases, since the DLLs should be compatible as long as they have the same major number, manually created symlinks solve the issue.

For example, the `unixODBC` package doesn't install a symlink for `libodbcinst.so` for some versions of Fedora, but some applications using a filename rather than an SONAME require that file. (This also happened with the Linux kernel 2.6.30.1, for `libXi.so.6`. The command "`make xconfig`" compiles a config program that tried to link to "`libXi.so`". Installing "`libXi-devel`" fixed the problem.)

> **`ldconfig`** doesn't automatically set up symlinks to DLLs for the link name, just the SONAME. This is because you might not want to compile code using the latest version of a library but might instead want to use a different (possibly incompatible) version of a library. So if you find that name is required to build some software, and installing the appropriate development package doesn't help, you should create that symlink manually.

Another problem may be that a DLL is available but **it was installed with incorrect permissions**. If root installed the DLL and the `umask` was set to (say) "027" then other users may not have permission to use the DLLs. This is easily fixable with `chmod`.

Since using an unloaded DLL is slower than a statically linked application, some OSes now uses "**prelink**" program which does most of the work ahead of time and speeds up the dynamic linking step later (at runtime). This step requires a detailed analysis of all the applications (installed on that host) that use some DLL, as well as all other DLLs they depend on. All addressing issues are resolved then, and the binaries are modified with the extra information.

> **Prelink can break security** checks that depend on checksums! Additionally, prelinked executables have predictable function addresses,

considered a bad security practice today. If possible (and performance isn't an issue for your system), you can disable prelinking.

Some packages run `prelink` as part of their post-install script, so may not validate immediately after installation. In that case, use the digital signature (or checksum) on the RPM package itself to verify that it was unmodified.

In order to have two different versions of some library available (so some applications can use one version while other applications use a different version), the symbols (function and data names) in the DLLs must be tagged with a version number (or name). This is called ***symbol versioning***. Not all systems support that. For example, Red Hat Linux and variants such as Fedora don't (2017), while FreeBSD and Debian Linux (and variants such as Ubuntu) do.

One reason not to use symbol versioning is that with versioned symbols, updating a DLL won't cause the programs to use the new version! Also, libraries may contain old versions of a function for compatibility, so knowing something depends on `glibc-2.1` may or may not mean anything.

If you want symbol versioning, you can rebuild the kernel and the DLL tools on a Red Hat type system and use multiple versions there too. (See this blog post for more details.)

Most DLLs are kept in `/usr/lib`. **If this is a separate partition from `/`, then when booting into single user mode, no DLLs are available and no utilities that depend on them can be run!** An example was `/bin/sh`, which on Solaris was static-linked, versus `/bin/bash`, which is dynamically linked.

For Solaris versions older than 10, *never* change `root`'s shell in `/etc/passwd`. (It's still not a good idea even on newer versions, but possible.) The shell listed is used in single user mode and was statically linked. If you change this to `bash`, your system will not be able to boot into single user mode! You would need a rescue CD to fix this, or you must re-install the system. (More recent Solaris versions have dynamically linked shells, but only depend on DLLs from `/lib`, which should be available.)

While **DLLs with the same major number are *supposed* to be compatible**, sometimes they aren't! DLLs are often installed from one package and used by programs in a different package. Thus, **updating one package may break programs from other packages**. You need to test your critical applications after any DLL updates (regression testing).

When preforming an update on a running system, some DLLs may be in use by running applications, or applications may be running, but expecting the older DLL version when using that functionality. This can lead to application crashes and other problems.

In Fedora 18, DLLs identified as "core" won't be installed at the time you run the update; instead, like some updates on Windows, they will get postponed until the next reboot. Note, `yum` doesn't include this functionality (it is in the GUI PackageKit tool only) and works as it always did. The `yum` replacement `dnf` should do this (and address some additional issues).

Some packages install different libraries with the same name, or different versions of the same library. The last one installed will over-write the earlier ones. (S*ymbol versioning* allows libraries with higher major numbers to remain compatible with previous versions.)

Most OSes included some static versions of vital programs for use in single user mode when the filesystem containing needed DLLs hasn't been yet mounted. The directory `/sbin` may contain static-linked versions of several utilities, useful when repairing the system from single user mode. **[*With large disks common today, there is no need to make `/usr` a separate partition, and you won't need statically links programs. Few OSes provide many of these anymore. Use statically linked apps for security.*]**

With so many DLLs and packages, vendors can only do limited compatibility testing. (Stable Debian and some Unixes do extensive testing before an update is released. Fedora does less, the price for leading-edge tech and frequent updates.) Keep in mind, no testing is ever 100% through.

Managing DLLs

The Linux link-loader **`/lib64/ld-linux.so`** (Solaris: **`ld.so`**, BSD: **`rtld`**) finds and loads DLLs into memory as needed by programs. `ld-linux.so` looks for the DLLS in:

1 `$LD_LIBRARY_PATH` directories,
2 a compiled-in location,
3 from the cached locations in `/etc/ld.so.cache` (a file of DLL paths that can be quickly searched), and
4 in the *trusted* directories `/lib` and `/usr/lib`.

Use **`ldconfig`** [`-v`] to rebuild `ld.so.cache` from the two standard locations plus the directories listed in `/etc/ld.so.conf`, and to create any missing links (e.g., *name.so.1* should be linked to *name.so.1.2*). Note this doesn't always create all needed symlinks (e.g., *name.so*). Use `-p` to show the cache. Use **`ldconfig -n ~/lib`** to processes `~/lib`. Note the cache does not survive a reboot.

BSD doesn't build a cache. Instead of `ld.so.conf` it uses `/usr/local/libdata/ldconfig/*` (one file per package, each listing directory pathnames). See the `ldconfig_*` settings in `rc.conf`. For Solaris, use instead the `crle(1)` command to configure `ld.so`.

A user can set environment variables to indicate DLL locations, and to over-ride system default DLLs with their own versions: **`$LD_LIBRARY_PATH`** (and for older types of executables, `$LD_AOUT_LIBRARY_PATH`) lists directories with `.so*` files. **This is not consulted for SUID programs, as that creates a big security hole.** Also, it is not reliable (it is used for *all* apps!), nor is it used for `gcc`.

In addition to `LD_LIBRARY_PATH`, a number of environment variables affect DLL loading. **`$LD_PRELOAD`** lists specific `.so` files to pre-load into memory when starting applications, for faster startup. This is ignored for SUID programs unless the DLL also is SUID. (These rules vary with the version of `ld.so` or `ld-linux.so` used. Consult the man pages for details.) For each DLL name *var*, Solaris also supports *var_32*, and *var_64*.

Use **`lddtree`** ***`file`*** to see what DLLs some *program* uses. Look especially for **`libwrap.so`**, which indicates that the *program* uses TCP Wrappers, and `libpam.so` if the program uses PAM. (Solaris has `ldd`, but use `pldd`.)

`lddtree` shows for each SONAME referenced by *file* the pathname of the DLL found for it. If missing, no pathname is shown (or you might see a "not found" message). To fix, add the configuration from SONAME to pathname, by editing `ls.so.conf` (or add a short file into `/etc/ld.so.conf.d/`), then rerun `ldconfig` to rebuild the cache. Alternatively, you could set `LD_LIBRARY_PATH`, or recompile, informing the compiler where the DLL file is. (That's not great solution, as it hard-codes the pathname into the executable.)

> **Qu: Why is `libc` *mounted* as a filesystem on Solaris, as shown by `df`?**
> **Ans**: On Solaris 10, different DLLs are optimized for different CPUs. The loader (`ld.so.1`) works out at load time which version of a library is best to use based on the capabilities (it is tagged with). Since this is a non-trivial task, the determination is optimized in the case of `libc` (which is used by most programs). The `moe(1)` program determines the best executable to use, and then the SMF filesystem `mount` service does a bind mount of the best `libc` onto `/lib64/libc.so.1`. `/usr/lib64/libc.so.1` is just a symlink to `/lib64/libc.so.1`.

Creating and Using DLLs

```
vi hello.h hello.c use-hello.c
```

```
gcc -fpic -c hello.c

gcc -shared -Wl,-soname,libhello.so.1 \
  -o libhello.so.1.0.0 hello.o

mv libhello.so.1.0.0 ~/lib

ldconfig -nv ~/lib

gcc -L~/lib -o use-hello use-hello.c -lhello

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/lib ./use-hello
```

(Show "dll-demo.tgz". *In class project idea: fix missing DLL symlink.*)

In 2020, SUS/POSIX agreed to a standard way to do this, for the next version of POSIX (issue 8). Note by then the C compiler may be renamed to c17 or c2x or something:

```
c99 -G -c foo.c
c99 -G -o /path/to/dir1/foo.so foo.o
c99 -G -c bar.c
c99 -G -o /path/to/dir2/bar.so bar.o
c99 -B dynamic -L /path/to/dir1 -L /path/to/dir2
\
    -R /path/to/dir1 -R /path/to/dir2 \
    -o foobar foobar.c -l foo -l bar
```

To create and use static libraries:

```
gcc -c hello.c
ar -rcs libhello.a hello.o
mv libhello.a ~/lib
gcc -L~/lib -o use-hello-static use-hello.c -lhello
```

Use `ar -tv libhello.a` to see the table of contents of an `ar` archive (output similar to `tar`).

Trouble-shooting Applications

When installed software fails to work as you expect, you can use debugging tools to determine the problem (the first step in fixing the problem). You first step should be to examine log data. If some issue repeats, you should adjust the log level to collect as much data as possible from the failing module. The log data just before and during the issue is often all you need to understand and solve some problem.

When the logs don't help (or you simply don't have detailed log data available), you should use *tracing* to find the issue. Tracing tools run software in a special way that reports everything done in detail (such as a service trying to open some file and failing). Use Linux tools **`strace`**, `ltrace`, eBPF, SystemTap, LTTng-UST, or `truss` or **`dtrace`** on Solaris, FreeBSD, and other Unixes. Ex:

```
strace -e open,close,read,write date
```

> `ltrace` has not been updated in many years (as of 2017), and no longer can trace most binaries; it depends on how they were linked. (See StackExchange for more info.)
>
> Oracle in 2017 open sourced dtrace. However, since it was not available for years, Linux developers went ahead and created something better: eBPF (*enhanced Berkley Packet Filter*). BPF started as the Berkeley Packet Filter, a language for declaring network packet filtering rules and eventually, a pretty good JIT (*just in time*) virtual machine runtime for applying these rules quickly. The rules compile into bytecode that in turn is run in the kernel safely, via this VM. Developers took the new BPF firewall code and expanded its abilities to run other things such as tracing probes. Using eBPF is difficult currently, as there is not a decent high-level language for it (such as for dtrace). There are several projects working to address that, including BCC and bpftrace.
>
> As of 2018, Systemtap can use eBPF behind the scenes, and Systemtab is much easier to use.

Tracing tools show you what system calls and library calls are made. **Mostly the stuff an SA can deal with is checking the open, read, and write system calls, to see which files the application is failing to access.** Often you can change the permissions of those files and fix the problem!

> On an old Fedora system, the PulseAudio server refused to play sound. It would work if I were root (sometimes, but that's a different bug). No matter how it was configured, the only error message it would emit was "Permission Denied". To find which file was the problem, I used "`strace -e files`"and found the config file missing `a+r` permissions. After using `chmod`, I had sound!

To see what currently running programs use some DLL, you can use `lsof`:

```
# lsof -nP +c 0 | grep libc-
```

That will display all running programs using `libc`. (The "`+c 0`" means not to truncate long lines, "`-n`" says don't do DNS lookup, "`-P`" says don't lookup port numbers.) Suppose you have updated some DLL and need to restart all daemons using the old version. You can generate a list is program names and/or PIDs using

this, and then use `xargs service restart` or some other command to restart all of them (and stop/kill the rest), without rebooting. (Rebooting may be simpler.)

Lecture 8 — Understanding Software Licensing and Open Source Software

Anytime you use software you did not write yourself you are subject to the terms of the license for that software. This includes applications, libraries, plug-ins, graphics and other media, database components, documentation. Some applications are bundled with libraries and thus there could be several licenses that apply.

A license provides you with the legal right to use some product, within certain specified constraints. This is known as the ***License Grant***. The grant clause of a license states when and how you can distribute the software. It also states the permissible uses of the software or service.

Permissible uses clauses involve limits on the site or location, equipment, network, third party data, outsourcing entities such as service bureaus, and parties such as successors and corporate subsidiaries or affiliates.

Software licensing is a complex topic that involves intellectual property, copyright, trademark, trade secrets, and contract law, including (in the U.S.) the *Uniform Commercial Code* (UCC), international law (e.g., if the Internet is used), *Uniform Computer Information Transaction Act* (UCITA), and the *Digital Millennium Copyright Act* (DMCA). Even patents!

Thanks to various copyright terms extensions over the last four decades, the U.S. is living in the midst of a public domain "draught" under which no important works will come out of copyright protection until 2019 (when works written in 1923 or earlier enter public domain).

Before the 1976 Copyright Act reforms, copyright in the U.S. lasted for 28 years, with another 28 if an extension was applied for. (But only around 15% of copyrighted worked had an extension applied for.)

Under the old regime, works from 1953 would have entered the public domain in 2010. However, the 1976 copyright reforms extended copyright, and 1998's so-called "Mickey Mouse" term extension went even further. Thanks to those laws, much pre-1976 material that is not already in the public domain was granted a 95-year period of copyright protection; books published since 1976 now have life-of-the-author + 70 years of protection.

The result is a gap in which no major works will enter the public domain in the US. For example, the song "Happy Birthday To You" will not hit public domain until 2030; Time Warner collected $2 million per year in royalties for that song for several years following 1998. (That copyright was deemed invalid in U.S. Fed. court in 2016; the EU copyright expired in 2017.)

Licenses that accompany commercial products are written by lawyers to provide maximum control and minimum liability to the company that makes the product. Often it takes your own lawyer to interpret the license terms for you. In many cases, software licenses are badly written or not legally enforceable (this may vary from state to state). Also, in many cases you can negotiate the terms with the license holder, although they usually don't make this easy. Software is generally licensed rather than sold, in order to better protect the vendor's intellectual property right. The license often prohibits reverse engineering, modifying, disclosing, and transferring the software.

Some licenses grant intrusive rights to the licensor, such as the right to examine your hard drive, install extra software (spyware), the right to monitor your use of the software or service, or the right to sell your personal information.

If you plan to depend on this product, it is essential to understand the license. Many companies don't allow you to make any backups of their software, forcing you to exclude that from your normal backup procedures. In some cases, the software is allowed on a single system, and if you upgrade your hardware you may not be allowed to use that software without purchasing a new license.

If the company has been sold or is no longer operational, it may be impossible to renew or purchase a license, forcing you to abandon that product. (This happened to me with a copy of backup software I had purchased on-line; I didn't even have any CDs to re-install from.) In some cases, this risk can be mitigated by the use of ***software escrow agents***, who (for a price) will keep a copy of the source code for some product and will release it under certain conditions (such as the company that produced the software is out of business).

Disaster recovery issues must be addressed in the context of the need to use the software in a remote location while protecting the vendor's intellectual property. Computers may be physically changed with software compatible upgrades or exchanges, due to upgrade needs or disasters.

Besides understanding licensing from the point of view of the purchaser, you may need to **license your own (or your company's) work**. While the license for any commercial product should be developed only with the advice of a competent attorney, there are some **standard licenses** you can use or adapt for your products. Most of these were developed for *open source* products. (Show web links to Open Source License comparisons and descriptions.)

Licenses for open source systems such as Red Hat (or worse, Fedora) can be a problem, compared to proprietary (or "commercial") Unixes. RHEL ships with hundreds of packages, each with its own license. RH has claimed they are all "compatible" (see License Incompatibility, below) and does spend a lot of resources evaluating licenses.

To see the license for any RH package, use `rpm -qi` *`packagename`*.

To use some distro for your business, you might worry that you will get sued (or be unable to sue) because you violated some license. It can take a team of lawyers weeks to sift through all the licenses, and make sure they can be used with your business, in the way you need for your organization's processes. The process is called a ***compliance audit***). For example, RHEL has its "core" software under GPLv2, but OpenOffice is GPLv3 (which is why RH doesn't distribute it). It is reported (Linux Journal 5/2010, p. 12) that RHEL contains 16 additional licenses! The one for MonoType (fonts) allows only a single backup copy (so only one level-0 dump!); the Adobe Reader license doesn't allow it to be installed on a cluster or "hot" standby server (for failover). And Adobe Reader's license links to dozens of other licenses for related software.

> In practice, license auditing is not that difficult. Red Hat takes a lot of care about that sort of thing, and that is why it does not ship with any software with a questionable license. That's also why so many businesses use RH Linux.

Types of Licenses and Restrictions

With a **Dual-use License,** the user can choose the least restrictive or otherwise most appropriate license. This is popular for software released in a commercial version and a freely available version (available "as-is").

You should decide whether or not your users must agree to the terms before being able to use the services offered or downloaded software. Such licenses are known as ***Click-through*** or ***click-wrap agreements***, but technically are called ***end user license agreements*** or ***EULAs***.

A ***Site License*** is sometimes called a *seat license*, and may limit software use to particular computers. The identification of these computers can be a tricky issue. It is a widely misused term; it is often misapplied to a licensing agreement that allows software to be run on a large number of CPUs. It actually refers to an unlimited number of software licenses purchased and sometimes distributed at no charge to affiliates. There are very few true site licenses these days, since they tend to be prohibitively expensive.

But what does a "seat" or site actually mean? Simultaneous users? Is it better for the vendor to have a sliding scale fee structure based on the size of a licensee? Concurrent use must also be considered regarding work shifts (several people using one computer), use on laptops. and home use (one person using several computers).

Volume Purchase Agreements or ***Software Volume License Agreements*** are formal agreements between you and a software manufacturer or vendor. A purchasing source, order process, and pricing limits are negotiated and put in a contract with specific terms and renewal conditions. Volume purchase agreements

are rarely mandatory (i.e., end users can buy elsewhere), but generally offer very advantageous pricing.

Exclusive versus ***Nonexclusive*** licenses: Mass-marketed or ***shrink-wrapped*** and ***click-wrapped*** (for downloaded software) software is generally marketed on nonexclusive basis. This means the vendor can sell the product or service to others besides you. However, with software developed on a custom or *exclusive* basis, the user/licensee wants to gain an exclusive license (to maintain an edge over competitors). Because the vendor may want to market the software more widely, these licenses can cost significantly more. (e.g., custom web portal, search engine, database). Either way the license granted limits the permissible uses.

> "Nothing seems to prevent [companies] from attempting to prohibit certain conduct in their documentation, even if the prohibition is unenforceable. By way of example, DVDs marked "for home use" or "non-commercial private use only" are not legally restricted as such. Unless there is a contract, those representations are not true.
>
> "A given use, e.g., classroom showing, is permitted or infringing depending on copyright law, not what is printed on the packaging—unless there is a contract including that restriction. Rights holders still print such claims on the packaging, however. They might as well print that infringers are obligated to forfeit their first-born child." — Matt Schruers

Network Uses: The site license must consider network issues since software/data on a network at a given site may be processed elsewhere. The implications of how using particular software over the internet must be considered. Software that runs on servers may well be utilized in an internet context since so-called "n-tier" (enterprise) computing simply runs software components on several servers (often including web and database servers) as part of a single application.

Transferability Restrictions: As well as use by subsidiaries and affiliates, the issue of what happens if the licensee, or even the licensor, is sold in some manner must be addressed.

Open Source Licenses

Open Source licenses vary by how restrictive they are. Some are incompatible with others. Such licenses are evaluated by the ***Open Source Initiative*** (OSI) at OpenSource.org, to determine if they are really open source or not.

> There is something like 100 licenses approved by the open source institute and there is an ongoing effort trying to reduce the number of these licenses. But you are not required to pick any of these for your FOSS. You can take any of these and make variations, giving rise to more than 1,000 known variations on licenses.

> Bruce Perens one of the founders of the OSI explained that the OSI has existed for 21 years and has been approving software licenses during that time. There are more than 100 such licenses, he said, and having that many is harmful to the community because when you combine software with multiple licenses, that creates a legal burden.
>
> "Most people who develop open source don't have access to lawyers," he said. "One of the goals for open source was you could use it without having to hire a lawyer. You could put [open source software] on your computer and run it and if you don't redistribute or modify it, you don't really have to read the license." ...
>
> "We've gone the wrong way with licensing," he said, citing the proliferation of software licenses. He believes just three are necessary, AGPLv3, the LGPLv3, and Apache v2. Recently (2020), Perens left the OSI over this issue (specifically the CAL).

The European Union has created their own open source license, the European Union Public License (EUPL). It is "compatible" with most OSI licenses (See the EUPL compatibility matrix).

Some of the most significant and common of these open source licenses are:

- **The MIT Consortium (X11) License**
  This just says you can use the software any way you wish, as long as you don't sue the authors over it. **This is the least restrictive license** (of these listed here).
- **The BSD License**
  Almost as liberal as the MIT license, this license says you can do what you want with the code, provided you don't remove the copyright notes or disclaimer, both with source code and with binaries. The copyright and disclaimer simply ask for credit of the work and disclaiming any responsibility for it. You can't use the BSD name to endorse your product.
- **The Apache Software Foundation License**
  Similar to the BSD license, except extra clauses are added to ensure you don't use *Apache* to endorse or promote your product without prior permission. This is probably the most commonly used open source license, and a good choice for your own work.
- **The Gnu Public License (GPL / LGPL)**
  This one is more complicated and restrictive than the others mentioned above and is currently the focus of on-going legal and political battles. This says you can use GPL'd coded in your products, only if you release your products under the GPL as well. This is

sometimes referred to (by distracters) as the *viral* property of the license. The LGPL (the ***Lesser GPL***) allows your products to link to GPL'd libraries, without forcing your code to be released under the GPL. Note there are two incompatible versions of this license, GPLv2 and GPLv3. The Linux kernel uses GPLv2.

> In 6/2012, Canonical announced that Ubuntu on new UEFI "secure boot" hardware would not use GRUB2, but switch to "efilinux". This is because GRUB2 has a GPLv3 license, which would apparently force Canonical to publish the private GPG key they use for signing the boot loader code.
>
> Update: According to the FSF's Executive Director John Sullivan, the fear that Canonical might have to release the private key used to sign its boot loader is "unfounded and based on a misunderstanding of GPLv3." In 9/2012, Ubuntu decided to use Grub2 after all.

The LGPL used to cover libraries or other modules used with other software, can also have a "viral" property depending on how it is incorporated into your (proprietary) software. **If statically linked, the whole software is covered by the GPL license. If dynamically linked instead, it is not covered.** This is one reason why Fedora and others no longer distribute static versions of libraries.

> In 11/2017, Red Hat, IBM, Facebook, and Google, all of whom sell GPL 2.0 covered software, announced a major change to their license. All four are adding a clause from the GLP v3 license (but not the whole thing).
>
> The clause is called Common Cure Rights Commitment, and it specifies how to fix non-compliance. Without that wording, under GPLv2 it is possible to immediately sue or take other legal action. Naturally some potential customers shied away from such software. The companies hope the new wording will assure users that they need not fear such actions, as long as they make a good-faith effort to become compliant.
>
> The Linux kernel, also under GPLv2, as added similar wording to the license for the kernel and related software, the Linux Kernel Enforcement Statement.

- **GNU Affero General Public License (AGPL)**

  AGPL license differs from the other GNU licenses in that it was built for network software. (Basically, web applications that are never distributed in a traditional way.) It provides the same restrictions and freedoms as the GPL but with an additional clause which makes it so that source code must be distributed (that is, made available) along with web publication.

- **The Common Public License (CPL)**
  Created by IBM, the CPL is considered business-friendly. It is approved for use by the FSF and the OSI. However, it isn't compatible with the GPL. The receiver of CPL-licensed programs can use, modify, copy and distribute the work and modified versions, in some cases being obligated to release their own changes. Microsoft has released several projects with this (including a Windows installer development tool).

- **The Common Development and Distribution License (CDDL)**
  This is an open source license used by Sun Microsystems for ZFS. It is not compatible with GPL, making it very risky to include CDDL software in a Linux distro.

- **The Eclipse Public License (EPL)**
  This is a revised version of the CPL (a section about patent litigation was removed). To reduce the number of open source licenses, IBM and Eclipse Foundation agreed upon using solely the Eclipse Public License in the future. The OSI now shows the CPL as deprecated and superseded by EPL.

- **The Microsoft Public License**
  Similar to the BSD license, this was approved by the OSI in 2007. Many of the projects hosted at the old MS site codeplex.com used this license.

  In 2009, Microsoft released a Windows 7 USB download tool under a Microsoft license. It turns out that the content that Microsoft believed was theirs wasn't; some developer had taken a piece of functionality from some open source content. That open source had a GPL licensing attribute. Once Microsoft put the Windows 7 software out, somebody in the market detected that it used a piece of GPL open source. Microsoft had to halt distribution until they decided what to do. Their options were either to open source the project, replace that functionality with another piece of software that had the right licensing attribute, or rewrite it from scratch. The

last two cases meant significant delays to the product launch, so they decided to open source that specific application.

- **The Microsoft Community Promise -** www.microsoft.com/.../community-promise (formerly at www.microsoft.com/interop/)
  This is a legally binding commitment, through which Microsoft pledges not to assert its patents against others who implement certain Microsoft standards and technologies. It is generally compatible with open source licenses and philosophies. In 2009, it was applied to ECMA 334 and 335, which define C#. This means Mono and other non-MS implementations of C# need never fear a patent infringement lawsuit.

License Incompatibility

Just because two licenses are both approved as open source licenses, that does not mean you can freely combine software licensed under them. The licenses in question must be *compatible*. The FSF defines this as:

"In order to combine two programs (or substantial parts of them) into a larger work, you need to have permission to use both programs in this way. If the two programs' licenses permit this, they are compatible. If there is no way to satisfy both licenses at once, they are incompatible."

As a concrete example, the ZFS filesystem is licensed under the CDDLv1 license, and Linux under the GPLv2 license. These licenses are *not* compatible, making it a violation to ship a compiled Linux kernel containing ZFS. Canonical thought they could work around that by keeping ZFS in a separate file from the Ubuntu main kernel, as a loadable kernel module ("zfs.ko"). But the software conservatory and others disagree and I don't think Canonical will risk it. It *is* permissible to ship source code, allowing end users to compile the code, but that is not a feasible option to most companies.

There are so many licenses out there, many incompatible, that is has become a real problem to track all the licenses that apply to a given software product (that uses many FOSS parts). Black Duck Software sells a product that can identify all the licenses that apply and determine any incompatibilities. (They were awarded a software patent on their method, but it is being contested.)

The differences between licenses accounts for why RPMfusion has two repos, one free and the other non-free. The non-free repo is for incompatible FOSS software.

(*Show graphics from IEEE Computer May 2006, comparing FOSS licenses*.)

ChooseALicense.com helps users select an appropriate FOSS license.

Violations of open source licenses are detected by competitors or any user. Typically, the violator is convinced to rectify the violation and obey the license terms. Organizations such as the Free Software Foundation, the Software Freedom Conservancy, and the Software Freedom Law Center will do that for you if you report the violation to them, turning to legal measures as a last resort.

Ethical Licenses

The JSHint software had this license since 2011: "The Software shall be used for Good, not Evil." It took seven years for the project lead to change this restrictive and vague license to a standard one, but other those years, they lost their place as the top JavaScript linter to arguably lesser programs, but which had better licenses.

There are several similar, so-called ethical licenses, such as the Hippocratic License. These licenses prohibit people from using the licensed software if they "actively and knowingly endanger, harm, or otherwise threaten the physical, mental, economic, or general well-being of underprivileged individuals or groups."

The problem with such licenses is ambiguity. The licensor might not know exactly what is meant by "evil". The term is not defined by most legal systems in the world, including in the U.S. Legally-conscious objectors to software with such a license are simply refusing to enter into an ambiguous contract.

Avoid these licenses. Whatever your political opinions or your ethics, the best possible chance for your open source project to become widely used is to use a standard open source license.

Understanding Software Patents and Copyright (and Copyleft)

A *copyright* is a government granted protection to the owner/creator of some content (written works, art, music, software, etc.) A copyright applies to the expression of an idea, not the idea itself. See the Copyright Clearance Center's educational resources for more information.

Often employees must agree that various inventions and ideas developed by them and related to the business of the company are deemed owned by the company (***Invention Assignment Agreements***), so you can't just develop some wonder product and then sell or even give it away; it belongs to the company.

> Starting around 2016, one Linux developer started suing GPL violators directly, apparently for financial gain. Patrick McHardy worked on the *netfilter* team and, as is common, claimed copyright for his contributions.
>
> He started suing for copyright violations. In 2016, he was suspended from the Linux developer's team as a result of his actions. In October 2017, Greg Kroah-Hartman, Linux kernel maintainer for the stable branch, summed up the Linux kernel developers' position. Kroah-Hartman wrote: "McHardy has sought to enforce his copyright claims in secret and for large sums of money by threatening or engaging in litigation. Some of his

> compliance claims are issues that should and could easily be resolved. However, he has also made claims based on ambiguities in the GPL-2.0 that no one in our community has ever considered part of compliance."
>
> In his first 38 lawsuits, each company paid rather than fight in court. However, in 2018 one company, Geinatech, decided to fight in German court. The judge found for Geinatech, forcing McHardy to pay legal costs for both sides. Hopefully, this will be the end of his activities.

A *copyleft* is a play on the word copyright, copyleft was first described by Richard Stallman (Free Software Foundation), in 1985. It describes the right of people to use, modify, and redistribute content. (Copyleft uses copyright law to enforce this.) This is the sort of terms included in FOSS licenses.

> Note that if there is no license stating otherwise, published work (including source code in a Github.com repo) is subject to copyright. By default, that means such work cannot be modified or incorporated into other (*derived*) work. That sort of defeats the point of publishing software in the first place. Always add a license. (Show how on a new GitHub repo.)

A *patent* is the exclusive rights granted by some government to a holder on any use of the ideas covered in the patent. A patent typically lasts for 20 years (since 1995, when the U.S. agreed to the WTO's TRIPS agreements), and may be extended under some circumstances. Companies can license their patents for *royalties* (money); IBM typically earns $2 billion a year from their portfolio of patents. (Some companies sell no products, and just hold a portfolio of patents.) Patents need to be filed for in every country; a U.S. patent won't stop someone in Europe from using the ideas covered by the patent.

Some companies attempt to make money by suing others for violating their patent rights. When done on software that should not be covered by the patent, the suers are derisively called ***patent trolls***. Unfortunately, it takes money to fight lawsuits; many people and organizations are forced to pay rather than risk losing the rights to some key software they need. This is what SCO tried, starting in 2003, suing IBM and others over their rights to Unix and Linux. (The courts eventually ruled against SCO.) Unisys tried this with GIF in the late 1990s, leading to the development of PNG.

> Some companies with a software patent will delay filing the patent for as long as possible. If others in the industry then develop the technology and it becomes popular, the patent holder files the patent and demands high royalties. Such patents are known as "*submarine patents*".

Because of this threat, some companies patent their own work even when they don't expect any royalties (they provide *royalty-free licenses*). Such patents are

called *defensive* patents. These prevent others from acquiring rights to the work and suing the inventors.

The OIN (Open Invention Network) was founded in 2005 by a group of companies that rely on Linux for key aspects of their business. In order to protect Linux from future patent litigation, they assembled a pool of extremely broad patents that cover a wide range of essential technologies. In the event that a patent lawsuit is launched against core pieces of the Linux kernel or platform stack, the OIN could theoretically retaliate by filing a patent suit of its own against the aggressor. The defensive advantage of this patent pool is that it can deter litigation and provide the Linux ecosystem with leverage to negotiate a favorable cross-licensing agreement should it become necessary. (This may not work against patent trolls, however.)

A similar effort to the ION was founded in 2014 to help stop trolls against any company, not just Linux using ones. The *License on Transfer (LOT) Network* (LOTnet), was founded by Google, Newegg, Canon, Dropbox, SAP, and others. LOT Network members put all their patents in a pool, which is immediately licensed to every other company in the network **if, and only if** they're ever sold (transferred) outside the network. That would include a sale to a patent troll or a hostile non-network competitor. As long as the patents aren't sold, they can be used both defensively (if needed to counter-sue a competitor) or offensively (if a company believes a competitor is infringing). However, the patents can't be used by trolls to sue any member companies.

> From creativecommons.org/weblog/entry/5709 (L. Lessig, 11/30/2005):
>
> Documentation is also covered under licenses. But there is a problem with these. You might be able — technically — to remix FOSS content from Wikipedia, Flickr, YouTube, etc. But can you remix it legally? Will the licenses for that "free" content permit that free content to be remixed?
>
> The surprising answer is probably not. Even if all the creative work you want to remix is licensed under some FOSS ("copyleft") license, because those are different licenses, you can't take creative work from one and remix it in another. Such a remix is defined as a *derivative work*, and few FOSS licenses allow derivative work to be under any other license (for fear of having FOSS works remade into proprietary works.)
>
> Wikipedia, for example, is licensed under the Gnu FDL. It requires derivatives be licensed under the FDL only. And the same is true of the Creative Commons Attribution-ShareAlike license that governs Opsound content, as well as much of the creativity within Flickr. All of these licenses were written without regard to the fundamental value of every significant advance in the digital age — interoperability.
>
> Work is in progress to address this, and it is expected that the various licenses will be updated to permit interoperability by early 2009.

Artistic works are also covered by licenses. As with software, these can vary. There are a number that have embraced FOSS principles, such as from creative commons or the "free art license" from artlibre.org/license/lal/en. Artistic works include fonts. These are often covered under the Open Font License (OFL) from scripts.sil.org/ofl.

Understanding and Evaluating Free/Libre Open Source Software (FOSS)

"*Free as in beer*" means you don't need to pay to use the software. "*Free as in freedom*" means there are few restrictions on your use of the software, e.g. include it with your commercial, proprietary product. To indicate something free in both senses, the term FLOSS was used; today, FOSS is preferred by most.

Market research firms reportedly estimate that for 2014-5, over 67% of all active services worldwide are Linux (or Unix) based, with some estimates over 95%. Netcraft (Internet research firm) reports the FOSS Apache web server at 48% of the market (with MS server at less than 30%, Nginx at 14%, and all others under 3% each). Other popular open source produces include Firefox (and other Mozilla.org projects), FreeBSD Unix, PostgreSQL RDBMS, SugerCMS, and thousands of others. Most cell phones, set-top boxes, gaming consoles, etc., all use open source for part of the system. Even proprietary software is often developed using OSS compilers, tools, and IDEs.

Before committing to a FOSS project, you need to understand that not all FOSS projects are successful. If the software you decide to depend upon is not supported you will not be happy with the additional expense of hiring programmers to re-write it, or to replace it.

Your organization is thinking of using (and depending on) some FOSS. If that FOSS fails your organization's products will fail. So how do you tell if a FOSS project is healthy or a bad risk?

Projects that have an element of intellectual engagement (i.e., are a hot topic) attract more and better talent than a project that offers a useful project or fulfills a community obligation.

Most healthy FOSS projects have one or two people that founded the project (say by creating a preliminary version then taking the project public), two to ten additional "core" programmers (those with access to commit changes to the repository). A few exceptional projects have many more core programmers (Debian, Linux, Apache), but in general too many core programmers means that a lot of time is wasted synchronizing their efforts.

Part of this core is the current project leader, who maintains a "vision" and thus can resolve disputes. The leader often also coordinates official releases (or versions) of the project. Initially the founder is the leader, but it is a healthy sign if

the leadership has successfully transitioned to a new project leader once the project matures.

Surrounding the core programmers are a group of ***active users*****. These are those who participate in early testing, who communicate ideas and report bugs directly with core programmers, but who don't have commit access to the repository**. Over time, some active users become core programmers, but FOSS projects rarely if ever have formal management that determines who gets commit rights. If you gain the respect of the project leader, founder, and other core programmers, you may be granted such access if you request it. Active users contribute to IRC, mailing list, and newsgroup (forum) discussions, write how-to documents and user guides, maintain a wiki site, etc.

> You can also check the mailing list for core developers, if available, and see how they all work together...or not.

Active users serve another vital role: they provide support to the vast hordes of ***passive users***, those who just download the software and need questions answered. It is not a good sign if the core users do this much, they quickly get burned-out and can leave a project stranded.

www.OpenHub.net (formerly ohloh.net) is a site that provides reviews and metrics for many/most open source projects (Demo Apache: select *compare projects*, show activity, contributors, and reviews). This is a good start to evaluation of some project! Note that GitHub and Sorceforge.net provide some of that data as well. Another interesting site is chaoss.community, which works on developing metrics and analytics for FOSS projects.

Note that if your organization chooses to make some software open source, you should examine several successful (and similar sized) FOSS projects, to see how they are organized and run.

It may not be obvious, but **whenever you use a dependency tool such as DNF, Docker, or Yum, you are automatically downloading software written by unknown parties.** Just because some package was uploaded to a popular repo is not a guarantee of quality! Worse, such 3rd party software often will depend on other packages (so 4th party software and beyond); these are called ***transitive dependencies***. It is unlikely you'll have sufficient time to do quality checks on all that. What can be done?

You can check the home website of each explicit dependency you have, to look for red flags (indicators of poor quality) and green flags (indicators of high quality).

Red flags include major but old bugs not fixed or no recent fixes. (Look at the project's issue tracker for these.) If the software doesn't include a test suite, that's a red flag. If the project seems to have a single developer/maintainer, that's a red flag (such projects are likely done in some person's spare time). Download and

compile their code with all important warnings and checks enabled. Run their test suite. Any issues are red flags.

Green flags include a large number of users (check the download/dependency statistics), a through test suite, an issue tracker showing important issues fixed quickly, and commit log showing recent activity (so the software is actively maintained).

Check the software and its transitive dependencies for security issues from the U.S. National vulnerability Database (NVD). An easy way to do this is with the OWASP dependency checker. The first time can take over 10 minutes, since the whole NVD must be downloaded. Later runs only download any updates and should take a few seconds if run regularly. Any serious security issues in the current versions are serious red flags.

All projects are bound to have some red flags and some green flags. You need to decide whether or not to use the software in your projects after evaluating all the flags. Considering your intended use, some flags may be more important than others for you.

Another green flag is a FOSS CII bage. The Linux Foundation (LF) Core Infrastructure Initiative (CII) Best Practices badge is a way for Free/Libre and Open Source Software (FOSS) projects to show that they follow best practices, especially for security.

> All software, not just FOSS, should be evaluated prior to adoption. One way to do this is simply using Google with the right keywords, to see what issues people have had in the past. Some good choices include (in addition to the name of the software): *crash, won't compile, won't build, won't start, slow*, etc. (Keep in mind there will be some hits for these searches, even for excellent quality software.)

Open Source versus Open Standards

FOSS (or just OSS) may or may not be the best choice for key technology for an organization. But without a doubt, *open standards* are the best choice. For example, you might choose some proprietary email client, but most likely, you would be upset if it didn't support SMTP or POP. Good sources of open standards are the RFCs for the Internet, ISO standards, ANSI standards (ANSI is part of ISO), W3C (web tech) standards, khronos.org standards (for media hardware), Oasis-approved standards, and others.

There are rare exceptions, of course. Sometimes a proprietary standard is just better than the open alternatives. (For example, some people argue that H.264 is superior to Google's VP9 media standard, even though it is not free. Another example is Microsoft's ActiveSync protocol.)

FOSS and License Compliance

Most FOSS software does require the corporate user, who often tweaks or patches the software to include in their own products and services, to obey certain restrictions on the distribution and sale of such products and services. A company must ensure compliance with the terms of all such software (and non-FOSS software too), usually by periodic audits as part of ***IT governance***.

> ***IT governance*** means putting structure around how organizations align IT strategy with business strategy, ensuring that companies stay on track to achieve their strategies and goals, and implementing good ways to measure IT's performance. It makes sure that all stakeholders' interests are taken into account and that processes provide measurable results. An IT governance framework should answer some key questions, such as how the IT department is functioning overall, what key metrics management needs and what return IT is giving back to the business from the investment it's making. (From CIO.com article.)
>
> ISO 38500 (released in 2008) is an international standard for IT governance. Resources such as COBIT, PMBOK, CMMI, ISO 27001, ISO 9000, ISO 20000, and ITIL to name just a few have provided invaluable guidance to companies of all shapes and sizes on how best to realize IT Governance. (From LongViewSystems.com.)

Interest in IT governance is partly due to compliance initiatives, for instance Sarbanes-Oxley in the USA and Basel II in Europe, as well as the acknowledgment that IT projects can easily get out of control and profoundly affect the performance of an organization. It allows an organization to make effective IT decisions, and reports and audits ensure compliance with licences, regulations, and best practices.

FOSS compliance means that users of FOSS must observe all the copyright notices and satisfy all the license obligations for the FOSS they use in commercial products. This is more complex than expected, since your organization may also want to protect their intellectual property from unintended disclosure. Note that every change to your source code for products and services may conflict with changed licensing terms, or new components with different licenses. It is suggested that the trouble-ticketing system include an item for license compliance checks.

An essential member of an **open source review board (OSRB)** is the organization's legal counsel. They provide numerous services to ensure a company's products comply with open source copyright and licenses. They provide approval of the use of FOSS in products, advice on licensing conflicts, advice on IP issues associated with the use of FOSS, and review and approve updates to end-user documentation. As product and service updates affect licensing compliance, the legal counsel should establish and

help maintain a FOSS policy and process across products and product teams. This should include directions for how to handle compliance inquiries sent to the company in relation to FOSS compliance. To make software development simpler/cheaper, the OSRB should have some resources kept up to date, including a pre-approved license list, and a license compatibility chart.

Some available resources for those responsible for FOSS compliance include some training course and license compliance checking tools sponsored by the Linux Foundation. They are also putting together a set of best practices (see Openchain).

The SPDX standard is an effort to help with license compliance. SPDX is an open standard for communicating software bill of material (SBoM) information, including components, licenses, copyrights, and security references. SPDX reduces redundant work by providing a common format for this information and various tools to work with it. They make a standard format for license information to be included in any package and support a variety of tools that can read that info and produce compliance reports. The list of licenses includes standard (short) identifiers and URLs for each; use in your POM.xml file or in a file such as "license.spdx" your Git repo. A short file describing your package's license is common, but you can include a lot more information such as all your dependencies' licenses too. SPDK will soon (2022) be an ISO standard.

Created in 2007 by HP, FOSSology is an open source FOSS license compliance tool. You can use it to run license, copyright and export control scans from the command line. FOSSology includes a database and GUI to make compliance checks simple. The results can be saved as a SPDX or text file, with the copyrights notices from your software.

> See Apache NetBeans Issues webpage for an example. As of 6/2018, release 9.0 had been held up for almost a year while the group resolved license compliance issues!

FOSS Business Models

How do you make money with free software? An early business model was to sell support services. (Red Hat uses this model.) Since companies will only pay for support for mission critical software, it is hard to make money this way. But one reason many companies go with proprietary software, is that there is someone to support it if it breaks, and someone to sue if it breaks badly. In the unlikely event that a PostgreSQL RDBMS fails, who you gonna call? RH sold over $500 million in support subscriptions in 2009 for software it (mostly) did not develop and is available for free elsewhere. Today (2014) Red Hat reportedly makes more than a billion dollars a year. Software subscription prices guarantee updates, patches, bug fixes, support,

training, compatibility with mission-critical applications, and legal protection from patent trolls that target open source users.

Another model is duel licensing: a limited version with no support for free and a commercial version. (SleepyCat, MySQL, and others use this.) But the free version requires users to provide their enhancements back to the community. Novell's strategy for monetizing Mono involved selling licenses for using the software in embedded environments. The company also offered commercial development frameworks that allowed application developers to build iOS and Android applications with C# and other .NET technologies.

> Attachmate, which bought Novell in late 2010, has discontinued the Mono project and fired those employees in 5/2011. The former developers have formed Xamarin, which intends to build a business around cross-platform Mono development by creating its own new Android and iOS frameworks. (The company also plans to port Moonlight, an open source Silverlight implementation, to both mobile operating systems.)
>
> Update 6/2012: Moonlight project discontinued, as MS "sort of" abandons Silverlight for HTML 5.

Another model is to provide a FOSS core product and sell proprietary (but useful) extensions (SugarCRM, Zimbra use this).

Mozilla's revenue is generated through search deals with Google and other popular website operators. Mozilla receives a kickback for integrating search services from various companies into the Firefox Web browser. Mozilla also generates revenue from investments and donations that are made by individuals to the non-profit foundation.

Finally, there is the cost savings model. IBM for example pays many thousands of dollars every year to the Apache Foundation, the Eclipse Foundation, and others; what do they get in return? If you check the Apache governance, you will find they employ nine senior executive managers (directors), who manage more than 3,000 employees (mostly programmers but some for PR, fund-raising, etc.) and volunteers, currently working on over 100 projects. (The Eclipse governance is similar.) How much would it cost IBM to do all that in-house (not forgetting tech support)? Even if you consider that IBM only uses a fraction of those products, you also must consider they are not the only funding source. (In addition, the get PR benefits and tax breaks as well!)

Lecture 9 — Kernel Concepts, Configuration, and Building

Qu: Why build a kernel? Ans: To remove unwanted features (saves memory and time; a stock RH kernel (2008) cannot process IP traffic faster than about 30MB/sec.!) or to add features (NFSv4). Also, to add *patches* (security, bug fixes, and sundry enhancements).

Changing a kernel, whether it is to the latest available from your vendor or a new custom kernel, is always a big deal. But using a custom kernel provides you with the option to include or omit any feature or patch. This can be a valuable option.

When building a custom kernel, you can reuse your configuration from the previous version, so it isn't that difficult to do this. (Just time consuming.) Either way, any new kernel, like any updated software, must be tested, and then rolled out during maintenance windows.

You can find information (change logs) for Linux kernels at KernelNewbies.org, linux.softpedia.com, LWN.net, and elsewhere. These discussions are often more readable then the official change log at kernel.org.

The kernel is a collection of named *subsystems* (a.k.a. *drivers* or *device drivers*). Each subsystem implements some feature of the kernel, or provides support for a particular type of hardware (the *device drivers*). The subsystems can either be compiled into the ***base kernel*** *image*, or be compiled separately as ***loadable kernel modules*** (**LKM**s). (Some subsystems must be compiled into the base kernel and won't work as LKMs.) For Linux, LKMs are found under `/lib/modules/`*`kernel-version/`*. (Or maybe `/lib64/...`)

The LKMs can be loaded into the kernel in RAM as needed, or unloaded (when no longer in use) to save RAM. Although it does take a small amount of time to load a module, once loaded they operate exactly the same as code in the base kernel. (Qu: What is the security implication of LKMs? Ans: Severe! To address that, since version 3.7, it is possible to sign kernel modules and configure the kernel only to load modules with verified signatures.)

Module (Subsystem) Names

There is no standard naming scheme for kernel subsystems; each kernel developer uses whatever name they think is best (or no name at all). Also, there is no maintained list of subsystem names for Linux, you just have to know (or guess) when you see a kernel message or try to configure some subsystem. This can be very confusing!

Each driver has a *specific* (*external*) *name* (used by the vendor who created the driver) which is used to locate LKMs, and a *type* (or *generic* or *internal*

name, used by the kernel). Although some types have obvious names (`eth0`, `ext3`, `snd`) many types have odd names such as `char-major-10-219`, which maps a specific driver to major device number 10 (and minor number 219). The command **`modprobe -c`** will produce a listed of used generic (type) names. (See `man modprobe`.)

In addition to a type (internal) name and specific (external) name, subsystems considered as device drivers are also assigned a major device number. When reading/writing device files (e.g., `/dev/tty0`), the major number is used to locate the device driver in question. (Major (and minor) device numbers were discussed in CTS-2301C.)

`modinfo` ***`module`*** shows options, other info.

(**Demo** `modinfo`, `nm -C --defined-only` *mod* for `snd`, `snd-cs4232`, and a sample **`modprobe.d/*conf`** file.) Also: `su -c "less /proc/kallsyms"`.

The kernel uses a table to map API syscalls and I/O attempts for device files (with a given major number), to specific driver-provided functions. If some syscall (say for playing a sound) has no specific driver module loaded for that *type* (i.e., the `snd` module), the kernel attempts to load the appropriate specific driver module (if it is a loadable kernel module, discussed below). The files **`/etc/modprobe.d/*.conf`**) on Linux includes ***alias*** directives that give the specific driver name for generic (type) names, and ***options*** directives to list important options for the device drivers (IRQ, I/O addr, DMA, ...). (For compiled in drivers, you can pass them options from LILO/GRUB; adding options for such drives with `modprobe` has no effect.)

Linux Kernel Names and Version Numbering

The Linux kernel *image* (the file that holds the base kernel program) is **`vmlinuz`**. (vm=virtual memory, z=compressed). Other names are vmlinux or **`bzimage`**, or unix, genunix, and kernel, on Unix systems. The actual name doesn't matter; you need to pass the name to the boot loader program and it will load any file you name.

> Linux kernels **had** a version-naming scheme of 2.X.Y. Even X values denote a stable kernel release; odd values were the development version.

Modern Linux versions are named 2.6.X.Y or M.X.Y, where all values of M.X denote (in theory) stable kernels. In either case, Y increments for bug fixes. The current (6/2020) "stable" version is 5.7.7. The major number (M) changes whenever Linus feels like it; it doesn't denote any technical changes or incompatibilities.

The 3.1.0 version of the Linux kernel was the next version after 2.6.39.2. But this major version number update doesn't denote major changes. It might just as well have been version 2.6.40.0 or 2.8.0.0. It was simply the preference of Linus Torvalds that the major version should be increased at this time (2011, the 20th anniversary of the Linux kernel). Under the new scheme, the major version is pinned at 3, the second digit will be used to indicate the actual stable release number, and the third digit will be used for patches to stable releases.

Certain kernel versions are considered "long term supported". Currently (6/2016) this is 3.18; some previous ones were 4.1.Y, 3.10.Y, 3.4.Y, 2.6.32.Y, 2.6.27.Y. (Show kernel.org.) The next one will be 4.4.Y.

> The Linux foundation has also decided to produce long-term stable kernel versions. The Long Term Stable Kernel Initiative (LTSI) will produce stable kernel releases that can be used reliably for the two- to three-year lifespan of a consumer electronics product. A new LTSI kernel release will be issued annually and will receive regular updates for a duration of two years. See the current versions at LTSI.LinuxFoundation.org.

On old Linux, the kernels were named `zImage` by default. These image files were compressed with `gzip`, limited to 520 KB, and included a boot loader in the first block. So you could just use `dd` to copy the kernel to a floppy and it would boot. The modern Linux kernel format is **`bzImage`**. The "`b`" means big (it still uses `gzip`, not `bzip`); this has no size limit and no built-in boot loader. You can use more modern compression such as zstd.

The Solaris kernel image name is usually **`unix`** or **`genunix`** (for a generic, platform-independent kernel). Unix kernels never have included a boot loader.

The image file and other required files such as **System.map** (kernel function addresses; used for debugging and error messages) are normally kept in **`/boot`**, but may be anywhere, say in **`/`**. It usually makes sense to have `/boot` set up as a "small" partition of about 500MB, of a filesystem type that can be accessed by your firmware and a boot loader. You can then use a more modern filesystem type (e.g., BtrFS under LVM) for the *root* volume.

Since even non-development systems may have several versions of the kernel on them, you usually give each one the name **vmlinuz-*version*** so you can tell them apart. (All the related files will use the same *version* string so you can tell which files go with which kernels.) Many vendors provide multiple kernels, with each one supporting different hardware. It is up to the SA to boot the proper kernel by properly configuring the boot loader.

> It *was* a good practice to make a symlink (or hard link) to the standard kernel to boot, with the name (for Linux) of *vmlinuz*, so that *vmlinuz* is the

default kernel to boot and *vmlinuz-old* is the previous (working) one. If you follow this guideline, the latest kernel will always be *vmlinuz* and if that fails to work, you can always boot from the old kernel *vmlinuz-old* and try to fix the problem. Most systems rarely need more than one current kernel and one fallback kernel.

With modern GUI bootloaders showing a menu of kernels and their versions, this scheme isn't useful on Linux and is no longer the default (when you install a RH/Fedora kernel RPM, `grub.conf` gets updated automatically.) It may still be useful on Unix systems with bootloaders lacking a menu that shows a list of kernels.

The Kernel Loading Process

When the power is turned on, the computer runs some software stored in NVRAM (or ROM). That *firmware* generally runs other firmware, which then runs a *boot loader*, which then loads a kernel into RAM. The kernel starts to run, does some initial tasks, and finishes the boot process by creating the init process.

The firmware that runs varies by system. All start with a POST, but after that it depends on the vendor and installed firmware. Some systems will have more firmware steps than others, such as LOM or Secure Boot. The various firmware programs that may be present are discussed next, followed by the kernel initialization steps.

POST When you power up a computer, ***firmware*** (a.k.a. the PROM) performs a *power on self-test*, detects and configures the RAM, keyboard, mouse, screen, various disk drives, and possibly stuff such as USB and NICs. In a sense, all such components are *Plug and Play* even before the concept was extended to other hardware. This works by having the main POST firmware look for additional firmware that may reside in NICs, disk, bus, or other controllers. Such *peripheral devices* may also run internal POST and initialization firmware. (**See man boot**.)

LOM/BMC Some servers (not often consumer-grade computers) have a low-level of firmware and special hardware (that often includes a network interface). ***Lights-out management*** or **LOM** (a.k.a. *out-of-band* management) allows a SA to monitor and manage servers and other network equipment remotely regardless of whether the machine is powered on. The hardware involved is usually called the ***baseboard management controller*** or **BMC**. The BMC is usually an ARM-based SoC, including NOR type flash memory for its firmware. The BMC has access to the PCIe bus, the CPU via a special bus (Intel called this LPC (low pin count) but now calls it eSPI (enhanced serial peripheral interface). The BMC is running as long as the motherboard is connected to power.

LOM/BMC functions include monitoring, logging, alerting, and basic control of the system. LOM is particularly useful for remotely managing a server in a typical "lights out" (loss of local power) environment, hence the name. (Note the UEFI standard provides many of the features of LOM; both can be used if desired.)

LOM uses some remote monitoring and management protocol, often an implementation of the open standard ***Intelligent Platform Management Interface*** **(IPMI)** developed by Intel. (That is, IPMI is the protocol for the outside world to communicate with the BMC, which generally runs proprietary firmware.)

> Intel replaced IPMI with *Active Management Technology* (AMT) on some systems, but IPMI is still widely used. The Intel BMC is called the Intel Management Engine, and the firmware that it runs was called Intel's *Management Engine BIOS Extension* (MEBx). In 2017, the name changed to *Converged Security and Manageability Engine* (CSME).
>
> Whatever it's called, you can usually get to that by powering up while holding down the F12 key. Note HP, Dell, AMD, and others also have their own propriety LOM firmware (but I didn't bother to learn what they named their stuff).

LOM provides for the monitoring and logging of dozens of sensors that measure voltage, temperature, fan speeds, and physical security of hosts. This provides the SA a good picture of the health of hosts.

In addition to monitoring, LOM can control other functions of a server such as power on/off and restart from a remote location, inventory, alerting, or even update the computer's NVRAM settings and boot loader configuration.

Even if the server is in an unresponsive state, an SA can gain access via LOM. The BMC controls the LOM process. It is accessed through a dedicated NIC, a special *wake-on-LAN* shared NIC (uses "stealth" ports), or a serial port. Dell calls this *chassis management* (and the hardware/firmware a *chassis management controller*).

> MEBx and newer Intel firmware use a BMC that only runs digitally-signed code from Intel, meaning end users have no control over the system. This system can run even when the main system is turned off and can invisibly use your NIC (if there isn't a separate one). It is strongly suggested you changed the default password for MEBx/CSME/whatever on your personal PC if it has that feature.
>
> IPMI has many known security vulnerabilities (some addressed in version 2.0 rev 1.1 (2013), but is propriety and secret, so is often neglected by vendors. But not by hackers! (See Pen Testers Guide to IPMI and BMCs.)

The DMTF (Distributed Management Task Force) together with Intel, HPE, and many others, started work on a successor to IPMI in 2016. Known as Redfish, new firmware implementing the Redfish protocol is already shipping on Dell and HPE servers (6/2017). The new system has a scalable interface based on HTTP RESTful API and JSON data using an open standard schema. Besides a much nicer, evolvable interface, Redfish include security enhancements as well.

> The Open Compute Project was founded by Facebook in 2011 to have open source designs and specifications for datacenter hardware and firmware. One outcome of this was OpenBMC, an open source firmware for BMCs. OpenBMC is based on Linux! There is also the newer u-bmc, written in Go by a Google developer as a minimal BMC system; it doesn't support IPMI or Redfish, but rather gRPC. This makes u-bmc highly secure but incompatible with management software that only talks IPMI or Redfish.
>
> Some folk at Dropbox developed open sourced BMC hardware specs known as RunBMC, which is not part of the Open Compute Project.

After the POST, the LOM (if any is present) is run. Typically, the LOM simply does nothing and the boot process continues with the final firmware stage, which is responsible for locating a loader program and running that.

Originally, all the system's firmware was simply known as BIOS, which had a collection of programs that ran POST, NVRAM editors ("*hit F2 to enter setup*"), diagnostics, and code that would locate and load a kernel loader (the *boot loader*).

> Originally, BIOS was designed to access the hardware; it was expected that the operating system would make BIOS calls to read/write a disk. Most (all) OSes ignore the services provided by the firmware, and accessed the hardware directly using their own drivers. The firmware still must provide that functionality; some boot loaders use it.

However, today's systems often include more modern firmware that replaces BIOS, but still do similar tasks: POST, initialization of basic hardware, etc. On any given system, you will only have one of these firmwares installed: BIOS, OpenBoot, or UEFI. (That is, if you have UEFI and not BIOS, your firmware still does a POST, runs LOM if present, then finds and runs a loader.) But that firmware is not BIOS; loaders or kernels that (for example) want to see the setting for NUMLOCK, must use one API for BIOS, another for OpenBoot, and a third API for EFI. Thus, the boot loader and kernel need to know which firmware is present.)

These three firmware systems are discussed next. Remember, a given computer will only have one of these present.

BIOS The *basic input/output software* for most systems in use today. After the POST (and possibly LOM if present), this firmware figures out a sequence of possible disks to boot from, and other configuration information, by examining the ***CMOS memory***, a.k.a. ***NVRAM*** (*non-volatile RAM*). With BIOS, the first bootable disk's MBR is examined for a **kernel loader program** (a.k.a. *boot loader*). If one isn't found, the firmware will look in the boot blocks of the partition marked as *active* in the MBR (that's the one to boot), then examine the next drive, until a boot loader is found. The firmware then copies the loader program to RAM, and runs it to continue the boot process.

At this point, BIOS firmware stops running. But it provides some functions that can be used by a boot loader or even the kernel later.

OpenBoot is an IEEE standard (IEEE 1275-1994) BIOS replacement for non-x86 hardware platforms and works similarly to BIOS, except it includes a boot loader. (BIOS doesn't, so you need additional boot loader software from the OS vendor, or some third-party boot loader such as GRUB.) Solaris uses this on Sparc; OpenBoot is also used by Apple (PPCs) and IBM. The OpenBoot firmware is sometimes called PROM, and has an interface so it can be examined and configured by user programs.

Because OpenBoot firmware includes a boot loader, it is more complex than BIOS. But it means you don't need any boot loader in MBR or in the boot blocks; OpenBoot will use its boot loader to find a kernel to load and run. (After the POST, of course.)

The firmware attempts to *autoboot* some kernel if the appropriate flag has been set in NVRAM. The name of the file to load and the device to load it from can also be manipulated. These flags and names can be set using the `eeprom(1M)` command (once the system is up and you can run commands), or by using PROM commands from the **`ok`** (OpenBoot's) prompt after the system has been halted. If your system doesn't auto-boot, you'll just see the `ok` prompt. Type `boot` to load the kernel, or type other commands. Note, OpenBoot doesn't assume the disk has an MBR the way BIOS does.

In 1998, Openboot was withdrawn as an official standard. However, it is still used on many non-x86 platforms. (There is a version called OpeniBoot that is designed to allow you to install Linux on an iPhone.)

EFI Today, the BIOS replacement is called UEFI (originally, and still commonly, just EFI). EFI was developed by Intel, but now is managed by the *Unified Extensible Firmware Interface Forum* (hence, the new UEFI name). Unlike OpenBoot, EFI doesn't necessarily include a boot loader (although it can), and BIOS-based boot loaders won't work. Instead, one of several EFI *boot managers* (notice the change of name from boot loader; probably a marketing decision!) can be configured to be used, such as elilo,

efilinux, or grub. (Non-Linux OSes typically include their own EFI boot manager.)

The EFI boot managers and other EFI software isn't stored as firmware. (Just a little bit of it is.) Most of it is stored in a FAT-32 partition on the disk. (The standard calls for FAT-32, but Fedora uses ext4; I guess any filesystem type will work as long as the EFI firmware includes a driver for that type.) This partition is called the ***EFI System Partition*** (***ESP***).

> Any boot manager must have the extension "`.efi`". Since versions 3.3, Linux has included an "EFI boot stub", so you can boot Linux simply by copying the kernel image ("`bzImage`") to the ESP, with the `.efi` extension. Assuming the kernel and the initial RAM disk are in the ESP directory "`Kernels`", you can then boot it from the EFI prompt this way:
>
> ```
> fs0:> cd Kernels
> fs0:\Kernels> bzImage.efi initrd=\Kernels\initrd.img
> console=ttyS0 root=/dev/sda4 …
> ```
>
> Notice how EFI uses DOS-style pathnames, I suppose since the ESP is mandated to be a FAT32 storage volume. Also note how the initrd pathname must be absolute, not relative. The "`root=`" parameter is not read by UEFI so uses normal *nix file names.

Calling UEFI "firmware" is a bit of a stretch. But this design makes UEFI very flexible. You can have dual-boot systems simply by having multiple UEFI boot managers installed. You then just pick one at boot time to run.

> Canonical announced (6/2012) that Ubuntu will switch to efilinux from grub at some point. This has to do with grub's GPLv3 license, which Canonical believes would require them to publish their private key used to digitally sign code modules. (Grub Legacy has patches to support UEFI, but nobody wants to maintain that old code.) Elilo is also GPL.

The new Intel-based Apple Mac installation DVDs only use EFI, which is why you can't install Mac OS X on older PCs. All OSes today (Windows, *nix) support EFI. Most x64 PCs only support EFI. Some systems use UEFI but in a BIOS emulation mode by default, so on such systems you don't get an EFI boot detected (by the OS installer software). In addition to DOS disks (those with an MBR), UEFI can boot disks formatted as GPT (*GUID Partition Table*).

The EFI interface consists of data tables that contain platform-related information (*EFI variables*), plus boot and runtime service calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system and running pre-boot applications. (EFI hosts have less need for LOM, but both can be used.)

The details of the boot manager are up to the firmware vendor, but all EFI boot managers are configured using well-known publicly defined and documented firmware configuration data. For instance, boot managers are required to inspect the `BootOrder` EFI variable for a list of EFI boot options, the device and path name of an EFI executable program to load and run, and a set of parameters to pass to that program. If no `BootOrder` variable is defined, removable drives are tried before fixed disks.

> To support older disks, EFI can include a ***BIOS emulation mode***. Currently, most motherboards have EFI firmware, but default to BIOS emulation mode. When an operating system installer, such as Anaconda probes the firmware, it may only detect BIOS. You may have to find some NVRAM setting to adjust that. (*Demo on classroom Dell.*)

Once this firmware stage is done (I don't think it has any standard name like the POST stage does), a boot loader and bootable kernel have been found. But there is an alternative available with modern versions of BIOS, OpenBoot, and UEFI: instead of running this firmware to locate a boot loader + kernel on a locally attached disk, the firmware can run one located somewhere on your network! This ability is especially useful for *diskless workstations*, and for remote installing new OSes on computers, without inserting a removable boot disk. (Which might be difficult on a blade server, with no removable drives!) This *network boot* feature is discussed next; keep in mind, if you configure your firmware for this, it won't look on a local disk for a boot loader.

PXE A system may also use a **PXE boot** (*Pre eXecution Environment*, pronounced "pixie"), also called a *network boot*. PXE is firmware used instead of the normal EFI, OpenBoot, or BIOS firmware to locate and run a boot loader (or directly load the kernel). PXE does that by initializing a NIC and loading software (a loader or a kernel) from across a network. Note, whether you have BIOS, OpenBoot, or EFI, you may or may not also have PXE. (Look at the BIOS setup options to see if you have this feature. *Show in class that the HCC classroom Dells do have PXE in the BIOS.*) To use PXE instead of the local disk option, you change the setting in the NVRAM.

PXE was developed by Intel for IA32, but may be found on other architectures too. This uses several standard protocols, most notably DHCP to initialize the host's NIC and to tell it the address of a TFTP server and name of a program available on that server to download and then run. The PXE client then uses TFTP to transfer the program (usually a network-capable boot loader, or a kernel image) across the network from a server.

So when powering up a server, you may see a LOM prompt, then either the OpenBoot, BIOS, EFI, or PXE prompt, then the boot loader prompt, and

finally the kernel loads. When powering off you may see the OpenBoot prompt (but never BIOS), and when that is exited, the LOM prompt (if used). For Solaris Sparc, either `inetboot` or `wanboot` is used when booting across the network, not PXE.

> If both UEFI and legacy BIOS clients exists on the subnet with the PXE server, it becomes a little more complex to setup the DHCP server, because it must provide a different bootloader to each PXE client depending on whether the system is using BIOS or UEFI to boot. One way to do this is by gathering the MAC address of each client, and putting these systems in groups in the `dhcpd.conf` file. Then, a different bootloader could be specified in each group.
>
> The UEFI specification says that client systems should send an "architecture" tag as part of the discover packet. The DHCP server can use this byte to determine dynamically whether the client is using UEFI or BIOS, just by putting something like this in the `dhcpd.conf` file:
>
> ```
> if option pxe-system-type = 00:07 {
>   # UEFI client
>   filename "uefi/bootx64.efi";
> } else {
>   # BIOS client
>   filename "bios/pxelinux.0";
> }
> ```
>
> (Not all EFI firmware correctly sends this tag as of 2012.)

Another step may be used as part of the firmware and/or boot loader. Designed for EFI, and probably applying to EFI's implementation of PXE, is ***secure boot***, which is discussed briefly next.

Secure Boot and the Trusted Platform Module (TPM)

Computers since 2012 probably include ***Secure Boot*** hardware and firmware. *Secure boot* is a technology described by recent revisions of the UEFI specification, but that is separate from any TPM. The TPM provides a hardware-verified, malware-free operating system bootstrap process that can improve the security of many system deployments. Sometimes, the TPM hardware is physically part of the BMC, but usually it is a separate SoC. When power is applied to the motherboard, the TPM runs first to verify the TPM and BMC firmware. After that, the BMC controls what happens next.

In essence, secure boot will use hardware to verify the digital signature of other firmware (EFI), and that firmware is responsible for validating the digital signatures of the bootloader, the kernel, and any other software modules loaded by EFI software. (Such as GRUB.) This requires the public key to be included in the hardware (in ROM, not NVRAM). The hardware

that does this verification is known as the *trusted platform module*, or TPM. Note that later software can be signed using different keys, as long as those keys are signed by (other keys, which in turn are signed by) the key stored in the hardware. This provides what is known as a ***chain of trust***.

Some hardware manufacturers have agreed to use Secure boot and not have any way to disable it (ARM, but not Intel), and to include Microsoft's public key in the TPM used. This would make it difficult to install alternative operating systems, or to include any patches. Since it isn't feasible for every OS vendor to have their keys included in all hardware (Canonical tried), Microsoft has agreed to sign other vendors' keys with their key, for a small fee ($100). This chain of trust may cause later boot loader modules to be slow to load (or fail to load), if their keys aren't available.

> Fedora signs only a "shim" boot loader with the key it has received from Microsoft and VeriSign. This shim boot loader is not licensed under the GPLv3. It just loads GRUB 2, which is signed with Fedora's key (or any copy of the GRUB 2 boot loader, if signed with any other key available in the firmware).
>
> Secure Boot requires that the hardware check a digital signature of the firmware used, and have that firmware check the digital signature of the kernel it loads. To support Linux, Red Hat has developed a "shim" boot loader, signed by Microsoft, which in turn loads grub. Ubuntu will use a modified version of that, which can load grub and kernel keys from a file on the disk; Canonical calls those *machine owner keys*, or MOKs. Unlike the RH scheme, you can update grub without rebuilding the shim boot loader. (RH said they like the idea and will likely use it in the future too.)
>
> The Linux foundation has developed a pre-loader loader that is similar to the shim loader, only it doesn't check the signature of the full loader (grub).

So now the boot loader is found, validated by secure boot, and run. Boot loaders may be third party software add-ons, such as lilo, grub, elilo, grub2, and others. There are some aspects of boot loaders an SA should know, and these are discussed next.

So the boot process might be this complex: Starts with the TPM-agumenting hardware (varies by vendor), which then starts the TPM, which then starts the BMC (for LOM), which then starts UEFI, which runs its own secure-boot firmware, which then loads a boot loader, which then loads a kernel. Whew!

The Boot Loader

A decent kernel loader program is too big and complex to store in the 446 bytes in the MBR (or boot block of a bootable partition), if using BIOS. This is because it usually allows interactive control of the kernel initialization

process: which kernel to load, what *run-level* to use, what parameters to pass to the kernel, etc. In practice, **loaders have two stages**. The stage one code simply locates and loads the more complex second stage code from a file on the boot media. It is the stage two code that can provide a GUI menu and other features, before loading the kernel.

The loader stage one code is very small, only a few hundred bytes long so if can fit into the MBR. It must be able to access the boot partition (the one containing the kernel) on the selected boot drive and therefore must understand IDE, SCSI, floppy, CD/DVD, or maybe USB *flash* drives, as well as the filesystem type used for that boot partition.

> OpenBoot includes a loader in firmware, so it doesn't have the same size restrictions as BIOS loaders do. EFI in essence includes a stage one loader in firmware. They call the stage two loader software a *boot manager*. There can be any number of those you want present, since they live in a regular (FAT-32) disk partition.

In practice, the stage one loader uses the BIOS/OpenBoot/EFI firmware to access the disk, but the filesystem type must be useable by the loader. This is why **some loaders can't handle boot partitions of newer types such as JFS or ReiserFS**. And, since BIOS won't understand software RAID or LVM, you can't use these features for the boot partition either. (**Show `ls /boot/grub`.**) Since those features are useful, most systems have a small boot partition of a simple type, and no LVM or software RAID, and the rest of the system with more sophisticated filesystems.

The stage 2 loader (or EFI boot manager) is copied into RAM and run. It may present a GUI menu and other functionality, but is mainly responsible for finding and loading the kernel, and passing the kernel some parameters.

> A given server may have several kernels installed, or different sets of options/parameters for a given kernel. A sys admin needs to know what boot loader is used on their system, and be able to change options or add additional menu choices (PXE may ask DHCP which kernel to boot, so you would need to configure that instead of a local boot loader).

The loader copies the kernel program (which is just a file) byte for byte into RAM and starts running this program, passing it any parameters that were set on the boot loader's command line.

While there are many loader programs available you can install, most OSes include one unique to their OS. For Solaris Sparc systems, the boot loader is called `ufsboot`. Such a loader may not be capable of dual-boot settings. For these reasons a vendor-neutral loader was developed called the ***grand unified boot-loader***, or GRUB. This loader can be used from the MBR, and each OS-specific loader can be installed in the boot blocks of the partition

holding that OS's boot files. (This technique, of one boot loader loading another, is called *chain loading*.)

Most *nix systems don't care what loader you use and don't come with some vendor-specific one. Today, GRUB is commonly used for Linux, Solaris, and can be used with Windows, FreeBSD, etc. (The older LILO or *Linux Loader* may still be found.) However, Grub legacy (version 1) can't deal with UEFI (unless patched), and the manual updating of the configuration file was seen as a drawback by some. So Grub 2 was produced; however, it is much harder to manage than Grub legacy. Both versions of Grub are described next.

Also popular today is an UEFI enhanced version of lilo, called elilo. (Elilo follows the UEFI specs very closely, and uses the configuration tool described there: efibootmgr.)

Grub Legacy (GRUB version 1)

The functional GRUB version 1 components include the `stage1` loader, the `stage2` loader, and **`menu.lst`**. The `stage1` loader is installed on the first sector of the bootable partition and can be optionally installed on the main boot record (MBR). The `stage2` loader is installed in a reserved area in the partition. The `menu.lst` file is located in `/boot/grub` and read by GRUB `stage2`. (A symlink to this called `/etc/`**`grub.conf`** is common.) (You may also find a "stage 1.5", which is really a stage 1 module specific for one filesystem type.)

You do not have to restart GRUB (as you do with LILO) for the changes to take effect. Simply save the file, and reboot. GRUB reads this file anew at every boot.

Hard disk are named in GRUB with `hd` and a number, where 0 (zero) maps to BIOS disk 0x80 (first disk enumerated by the BIOS), 1 maps to 0x81, and so on. Examples:

`(hd0)`	first bios found disk (also the BIOS boot disk 0x80)
`(hd1)`	second bios found disk (BIOS disk 0x81)
`(hd0,1)`	first bios found disk, second primary partition
`(hd0,a)`	first bios found disk, first Unix (BSD, Solaris) partition
`(hd0,1,a)`	first bios found disk, second *slice*, first Unix partition
`(fd0)`	first bios found floppy (rarely used anymore)

Partition numbers start at zero. 0 to 3 refer to physical partitions while 4 and higher refer to logical partitions (even if fewer than 3 physical partitions are used). Unix (not Linux) systems use *slices* rather than (or in addition to) partitions. In GRUB, the first Unix slice is called "`a`", the second "`b`", etc.

(The def. of *slice* and *partition* varies. Note, all this was discussed in the Admin I course.)

GRUB can also reference a single network interface called "`(nd)`" for network booting. This is almost always the interface the BIOS probed (and configured via DHCP or otherwise, for PXE booting-capable hosts). This is cool; even if your system doesn't support PXE, GRUB may be able to perform a network boot.

> GRUB names disks by the BIOS disk number; a change in BIOS boot device configuration may render GRUB menu entries invalid.

Unlike LILO, you cannot use `dd` to write `stage1` and/or `stage2` to some disk sector, because `stage1` must be told where `stage2` is located. The **`grub-install`** (and/or **`grubby`** command on Linux, and `installgrub(1M)` on Solaris) is used to install GRUB, or to update it if the disk has been repartitioned.

(Note that the Linux tool grubby can configure systems using elilo, lilo, or other boot systems, as well as GRUB. I recommend that tool if you have a choice.)

The `menu.lst` (`grub.conf`) file can contain blank lines, comment lines, general directives at the beginning, and one or more boot stanzas. (The first is number zero.) This file might look like this example (See the GRUB legacy manual for details):

```
default=0
fallback=1
timeout=5
splashimage=(hd0,1)/grub/splash.xpm.gz
hiddenmenu
title Fedora (2.6.17)
   root (hd0,1)
   kernel /vmlinuz-2.6.17-1.2142_FC4 ro
root=LABEL=/1
   initrd /initrd-2.6.17-1.2142_FC4.img
title Fedora (2.6.17) - RunLevel 3
   root (hd0,1)
   kernel /vmlinuz-2.6.17-1.2142_FC4 ro
root=LABEL=/1 3
   initrd /initrd-2.6.17-1.2142_FC4.img
title Fedora (2.6.16)
   root (hd0,1)
   kernel /vmlinuz-2.6.16-1.2141_FC4 ro
root=LABEL=/1
```

```
   initrd /initrd-2.6.16-1.2141_FC4.img
title Solaris 10
   root (hd0,2,a)
   makeactive
   kernel /platform/i86pc/multiboot -B console=ttya
   module /platform/i86pc/boot_archive
title Solaris failsafe
   root (hd0,2,a)
   kernel /boot/multiboot -B console=ttya -s
   module /boot/x86.miniroot.safe
title WindowsXP
   root (hd0,0)
   chainloader +1
```

If `timeout` is set to `-1`, user input is required. Some OSes may not boot unless their partition is set as the active one. GRUB allows you to do this by adding "`makeactive`" to some stanza (only for primary partitions). You can use the grub command `map` to change the BIOS names of disks (e.g., exchange `hd0` with `hd1`). The `chainloader` command means to load in the named file and run it, typically another boot loader. The "`+1`" syntax means the first block of the current partition, where a boot loader is usually located.

Grub 2 Configuration

Configuration for GRUB 2 (which supports both EFI and BIOS) is different from the original GRUB (*both described in Admin I course*). **The start of partition numbering has changed (to 1 from 0).** The `/boot/grub/*` files are now written by commands and scripts from config data in `/etc/grub` (or `/etc/default/grub`). The new grub configuration file `grub.cfg` replaces `menu.lst`. There are many other changes (see the GRUB2 manual for details):

The `grub.cfg` file is overwritten anytime there is an update, a kernel is added/removed, or the user runs **`update-grub`** (the Debian command) or **`grub2-mkconfig`** (the RH command).

The primary configuration file for changing menu display settings is **`/etc/default/grub`**. There are also the files in the `/etc/grub.d/` directory. While you can edit the existing files, to customize the kernel options for instance, these files get rewritten whenever GRUB2 is updated. If you want the same options used on all your menu items, you should edit the default options list in the `/etc/default/grub` file. The contents of this configuration file are used by the shell scripts in `/etc/grub.d`. However, the configuration items can vary between systems.

The files in `/etc/grub.d` should all be executable shell scripts. When run (by `update-grub`) they append data to the generated `grub.cfg` file under `/boot/grub`. The placement of the menu items in the grub boot menu is determined by the order in which these files are run.

After making any changes to any of the GRUB 2 configuration files, you must run **`update-grub`** or **`grub2-mkconfig`**. No changes made in the configuration files will take effect until that command is run. Also, automated searches for other operating systems, such as Windows runs whenever `update-grub` is executed.

Grub2 numbers partitions starting at 1 (one) instead of 0 (zero). Disks are still numbered from zero, so `(HD0,1)` would be the first partition on the first disk.

GRUB2 supports many interesting features, but they mostly require detailed knowledge and various "hacks" to use. One feature to know is how to set the default kernel to boot:

- Make sure your `/etc/default/grub` file contains this setting:
  `GRUB_DEFAULT=saved`
  That makes GRUB2 use the last booted kernel as the default for next time.
- Make sure all your changes are used by running the command:
  `grub2-mkconfig -o /boot/grub2/grub.cfg`
  If using UEFI, use this instead:
  `grub2-mkconfig -o /boot/efi/EFI/redhat/grub.cfg`
- Since the order of menu entries in GRUB2 can change, you need to use the *menuentry*'s title to set a default, not its position as with old GRUB. To see the available menu items:
  grep -E "submenu|^menuentry" /boot/grub2/grub.cfg | \
  cut -d "'" -f2 # that's double-quote, single-quote, double-quote
- To set the correct one as the default, run the command:
  `grub2-set-default "title"`
- To see the current default, run:
  `grub2-editenv list`

System (Kernel) Initialization

Now (finally!) the kernel starts to run. (It seems almost unbelievable that so much work must be done just to locate a file and load it into RAM, but powering up a computer is difficult.)

The first thing the kernel does once loaded, is to configure the various subsystems, using any parameters (**`man bootparam(7)`** and **`dracut.kernel(7)`**) passed in (from the boot loader) to over-ride default settings.

Next, the kernel runs hardware detection code and then initializes the hardware. (The firmware may have done this already, but the kernel doesn't assume that, so some hardware gets initialize twice. If you later start up X Window, it does that again to the video and keyboard.)

> In the previous course, we covered how udev (`udevd`) receives notices from the kernel as it detects hardware, and in turn creates and initializes the files under `/dev`. The problem with this is that udev is actually not part of the kernel; it runs as a regular process. However, neither `init` nor `udevd` processes can run correctly without some `/dev` entries — another catch-22!
>
> The Linux answer to this is to use a special RAM disk for `/dev`, of the type ***devtmpfs***. With devtmpfs, some `/dev` entries are created directly by the kernel, allowing `init` and udev to run correctly. (Udev will still receive notifications about that hardware, but since the `/dev` entries are already created, udev only needs to do whatever additional configuration is required.)

Modern kernels are *multi-threaded*. That allows one part of the kernel to be stuck waiting, while other parts continue to run. So one of the first things done is to start the kernel's thread daemon (subsystem); on Linux it is called `kthreadd`. The kernel starts the *swapper* (*pager*) kernel thread (`kswapd`) so virtual memory is available, and starts other kernel threads. Kernel threads show in square braces in `ps` listings.

Once the basic parts of the kernel are configured and running, the next step is to mount the root filesystem.

Once the kernel is loaded, configured, and initialized, and `/` is mounted (the root storage volume, which contains `/etc`, `/*bin`, and `/lib`), the kernel creates the **`init`** (systemd) process and the OS loading is considered completed. **From this point on, the kernel only executes in response to hardware events (*interrupts*) and *system calls* (the API).**

Some additional steps on modern systems include securing the system by setting the kernel *securitylevel* on BSD systems, *zone* setup on Solaris, and MAC setup on SELinux (or other) Linux systems. Only then is `init` started.

Qu: What if the code (the device driver) for some hardware device was compiled as a LKM? Ans: Since the code isn't loaded, the device can't be accessed. Modern kernels have an automatic module loader which can (usually, sometime not) determine which module is needed by some application, and load it. In Linux, this part of the kernel is called **`kmod`** (kernel/module.c). The (simplified) sequence is: app makes a system call (say to play a sound), kernel tries to look up the address of the function to invoke, determines that code is not loaded into memory, looks up the internal name of the module it needs, checks for `modprobe` alias names or options, locates the file on disk, loads it in, and finally runs that function.

Modules (and kernel configuration information) are stored in the ***root partition***. (Which partition is the root partition, is compiled into the kernel. It can be changed using the `root="/dev/xyz"` kernel parameter without re-building a custom kernel.)

Having many parts of the kernel as LKMs makes the system load faster, but can cause problems when the drivers needed to complete the boot process are not yet loaded. This problem and the solutions to it are discussed below.

dmesg and journalctl

As the kernel runs, it produces diagnostic messages describing the results of its hardware probing, subsystem initialization, LKM loading, etc. But at this point, there may not be a working console, let alone a disk log file to write to. Where can the messages go?

The kernel reserves a block of memory to hold such messages. The block is of fixed size; when full, the older messages are overwritten with the newer ones. This type of memory is called a *ring buffer*. You can use the command **`dmesg`** to view its contents. (*Show.*)

Note that if the system is up long enough, and has kernel activity, you can lose the oldest messages that were generated at boot time. Some systems run "`dmesg > /var/log/dmesg.log`" as part of the boot process (SysV `init` would do that), so the boot time messages are preserved. (Fedora doesn't do that anymore by default.)

Once `init` has started, such messages also are read by the system's logging daemon and those messages are saved in appropriate log files. But until `init` runs the logging process, the ring buffer is the only place to keep messages.

With systemd as the init system, all messages (kernel, daemon, and systemd) are stored in a journal (including any messages already in the ring buffer). Some of this journal is periodically saved to disk, so unlike the ring buffer, messages from previous boots are available. The command to

view this journal is **`journalctl`**. Useful options include `-b` (only since the last boot) and `-x` (add helpful comments). (*Show journalctl -bx, show the dmesg output in there.*)

Without `journalctl`, Sys Admins would use `grep` on the log files. `Journalctl` include filtering options and output control, making it more useful but slightly more complex to use. See the man page for some example usages.

You can configure `journald` to forward log files to your syslog daemon (usually rsyslog). If you enable that option in the config file, log data gets written to a socket `/run/systemd/journal/syslog`. You then need to configure rsyslog to also read log messages from that socket. Ultimately, this means you can have the binary journald logs as well as traditional text logs, assuming you want that (which doubles the disk space required). It might be a better idea to **learn how to use `journalctl`**.

Note that some daemons do not send log data to `journald` but directly to syslog instead. That means some log messages are in one place, some in the other. **When trouble-shooting your system, make sure you look in both places for log messages!**

(Hint: `journalctl` may output very long lines, and truncates those to the screen width by default. You can force it to use `less` (which wraps long lines) by "`journalctl` *`options`* `|less`".)

> The journal files are stored under `/var/log/journal`; the active journal is in `/run/systemd/journal`, a RAM disk, so data gets archived to disk every so often. You can control the disk space used from `journald.conf`, or manually using `journalctl`.

Initial RAM Disk

Since the kernel doesn't use the firmware to access any hardware, you can have a problem if your root partition is on a disk whose driver is compiled as a module. A common example is a SCSI disk when the base kernel only supports IDE. In order to load the SCSI disk driver, you need to access the root partition, which is on the SCSI disk! (A.k.a. *catch-22*.) You have similar booting issues with bootable USB thumb (flash) drives, or if the root partition uses LVM or software RAID, and the required drivers are LKMs.

One obvious answer is to build a kernel with the required code in the base kernel, rather than as LKMs. (LKMs are discussed in more detail, below.)

However, there is another solution. The base kernel does know how to access a RAM disk. You can make a copy of the required bits of the root partition and save it as an ***initial RAM disk image*** with the Linux **`dracut`**,

`mkinitrd`, or `mkinitramfs` command, depending on your version of Linux and distro—modern Fedora uses `dracut` (similar utilities exist for other Unixes). At boot time, the kernel can create a RAM disk by copying this image file into RAM, and pretending it is the root partition.

On newer Linux systems (those that use `systemd` and `dracut`), very small RAM disk images are made, containing only the absolute minimum software and modules. Using that image in single user mode may not work well, so a second, larger RAM disk image is made. That one is called the "rescue" image, and contains many tools and modules. (Note that a special type of RAM disk driver is used for this one purpose, `rootfs`.)

> The RAM disk image file is really a gziped `cpio` archive. Once the kernel builds the RAM disk, it just extracts all files from that archive into it. You can view the contents with `lsinitrd`.

Details vary by distro, but `initramfs` (sometimes called `initrd`) is a root filesystem with a copy of the kernel, some modules, and a few utilities. The special `init` program from `initramfs` runs a program (**`/linuxrc`** on Linux, typically a shell script) to load the required modules needed to mount the real root partition (e.g., SCSI, LVM, USB).

> `Initramfs` usually includes **`ash`** or **`dash`**, but not `bash` and no `stty` driver. So if you boot into *emergency mode* and get a shell prompt, ^C, ^Z, and other modern shell features aren't available. These shells have built-in versions of many file commands such as `ls`, and are used when space is at a premium.

Once all the required modules have been loaded from the RAM disk, the Linux **`pivot_root`**`(8)` command is run to switch the root partition to the real one. (See also **`dracut`**`(8)`.) The exact steps are complex and ugly; the kernel comes with a command usually called `switch_root` that does all the necessary work, including the call to `pivot_root`. When this command completes, the real `/sbin/init` program is started to complete the boot process. At some point after this, the `initramfs` is unmounted and its memory is freed.

> The steps taken are something of a dark art, and are changed from time to time. With the switch to dracut and initramfs, the init program launched from the RAM disk may not quit to start another instance (from the real `/sbin/init` or `/sbin/systemd`). The initial rootfs may or may not be unmounted, so there may or may not be two entries shown for `/` in `/proc/mounts`.

As mentioned above, the Linux `initramfs` image file is actually a gzip-ed `cpio` archive. However, the `cpio` format is not the default "bin" format normally used by `cpio`, but a format called "`newc`". If you make changes and recreate your `initrd`, you must use the new format. (`cpio` auto-detects the format on extraction; use "`-c`" when creating).

Here's an example of extracting, make changes, then repackaging some `initrd`:

```
mkdir ~/tmp;  cd ~/tmp
cp /boot/initrd-version.img ./initrd.gz
gunzip initrd.gz
cpio -itv <initrd |less  # to see a file list
mkdir tmp2;  cd tmp2
cpio -id < ../initrd
```

Now do the required changes to the files. Then pack the files back into the archive using the following commands:

```
cd ~/tmp/tmp2
find -depth | cpio -co  > ~/tmp/newinitrd
cd ~/tmp
gzip newinitrd
mv newinitrd.gz /boot/newinitrd.img
```

This is the new boot image, so edit `grub.conf` (or `lilo.conf` or whatever) to use it. (Make sure to have a GRUB entry for the old configuration too, in case this one has errors.) Note, you rarely need do more than configure your system, and run `dracut`.

> Once some students were editing some files in `/etc`, including `fstab`, while running `yum update`. When the file was not correct, `yum` ran `dracut` and copied the broken files to the RAM disk image. After fixing the file, the students tried to reboot. But the system wouldn't come up! They fixed the problem by booting the previous version, and replacing the bad `fstab` file in the new RAM disk image with the corrected version.
>
> It was hard for them to diagnose the problem, since `fstab` looked fine; they just weren't looking at the version that was actually used at boot time (in the RAM disk image).

Non-Linux systems do the same steps, but use different utilities and names. Solaris calls `initramfs` a ***boot archive***, which is loaded at boot time from "`platform/i86pc/boot_archive`", and is either an UFS or ISOFS image file. (Maybe ZFS is also supported by now.) The contents of the boot archive is listed in the file `/boot/solaris/filelist.ramdisk`.

At each shutdown, Solaris checks for updates to the root file system and updates the boot archive when necessary. You can manually update the boot archive by running the `bootadm(1M)` command.

> Linux doesn't do this (so it shuts down quicker); but if you rebuild some LKM or reconfigure your system, you will have to create an updated `initramfs` (with `dracut`). **If new hardware is added or other changes made that require a new RAM disk image**, use the command `dracut --regenerate-all --force` to rebuild and replace the old initramfs (and the "rescue" version of it too). (If you forget, you can run that command from "rescue" mode.)

The file **`/boot/x86.miniroot-safe`** contains a bootable, standalone Solaris image. This file can be loaded by choosing the `Solaris failsafe` entry from the GRUB menu. This image can be copied to a floppy, CD, or flash disk to create a bootable rescue disk.

The GRUB menu resides in `/boot/grub/menu.lst` (or `/stubboot/boot/grub/menu.lst` if a Solaris boot partition is used); for GRUB2, the file is `/boot/grub2/grub.cfg`.

Loadable Kernel Modules

On modern systems, most of the hardware device drivers are created as ***kernel loadable modules***, which are loaded automatically into RAM by *kmod* (or *kerneld*) on Linux as needed and unloaded after a period of non-use (usually 1 minute). Drivers may be compiled directly into the kernel instead, making the driver available at boot time. Note that kmod requires modules to be compiled for the specific version of the kernel you are using.

> An alternative to kmod managed modules is DKMS, discussed in detail later.)

Using the real (specific driver) name (say `sb.ko`), the kernel finds and loads the module from `/lib/modules/`*`kernel-version`*`/`*`type`*`/sb.ko`. On non-Linux systems, the location for modules (a.k.a. device drivers) may be different.

Solaris keeps loadable kernel modules in various places such as `/kernel/drv/` or `/usr/kernel/drv` or `/platform/$(`*`uname -i`*`)/`. (Note on Linux, module names can interchangeably use underscores or hyphens.)

Some modules depend on others; the system keeps a list (in built by **`depmod -a`**) in `modules.dep`. (Actually, that's a fallback text file; `depmod` also builds various binary files for quicker access.) However, the order of the modules in `modules.conf` is important; list the modules in dependency

order or they may not load! Use **`lsmod`** to see which modules are loaded, to add one use `insmod` or **`modprobe`** (better). Use **`rmmod`** to remove a module. (You can also look at **`/proc/modules`**.) Hints: `diff lsmod` output before/after `modprobe`, `grep /var/log/messages`.

On other systems, similar but different commands are used to manage modules. On BSD, use `kldload` *`module`*`.ko`. The Solaris commands are `modload`, `modunload`, `modinfo`, and `add_drv`, `update_drv` and `rem_drv`. The file **`/etc/driver_aliases`** maps the names much like `modprobe.conf` on Linux.

> **Solaris example**: Suppose you have an NVIDIA MCP51 Hi-Def Audio device with device ID pci10de,26c. You note from the Solaris for x86 *Device Support* list that the Solaris OS supports NVIDIA MCP55 Hi-Def Audio device with device ID pci10de,371 with the `audiohd` driver. If you're lucky, the same driver will support your card too.
>
> Search for the `audiohd` driver in the `/etc/driver_aliases` file to find the line "`audiohd "pci10de,371"`". You create a similar entry for your device: copy that line and change "`371`" to "`26e`". Next run "`update_drv audiohd`" to make the kernel scan that file for the change. Then run "`devfsadm -i audiohd`" to rebuilt the `/dev` entries. Finally load the driver with "`modload drv/audiohd`".

When a device driver loads (at boot time if compiled into the kernel or when the module attempts to load) it searches the system for a device it can drive. When found, the kernel table is updated to use this driver for a particular major device number (the one for the found hardware). Until this succeeds, the special files in `/dev` won't work. It may also register itself as the handler for the interrupt level that the device uses. It may also send setup commands to the device, so you may see lights blink or something like that.

A network device (interface) driver works similarly except it registers a device name of its choosing (e.g. `eth0`) rather than a major number. The currently registered network device names in `/proc/net/dev`.

A filesystem driver registers itself as the driver for a certain type of filesystem.

Linux LKMs are stored in **`/lib/modules/`*`kernel-version`*`/`**. (Show: `find /lib/modules/`*`version`*` -type d`, or `tree -d`.) Modules that come with the kernel may have strange names; modules provided by hardware manufacturers can be named anything at all. Module names have the extension "**`.ko`**" (for kernel module). (They used to be just "`.o`", but 2.6 changed the format of LKMs, and to avoid confusion, Linux adopted the new name.)

Modules can be loaded and passed parameters with **`insmod`** and removed (if allowed) with **`rmmod`**. The command **`modprobe`** can also do this better, and is generally used instead of the other commands.

You can see which modules are currently loaded with **`lsmod`** (which really just displays **`/proc/modules`**). (**Show**.)

```
ext3                  135304  3
lp                     13036  0 (unused)
parport_pc             29252  1
parport                42440  1 [lp parport_pc]
```

This shows the name and size, the number of "things" (open device files or mounted filesystems) currently using the code, whether the code has ever been used, and a list of other modules that depend on this one.

The daemon `kmod` controls the automatic loading of kernel modules. kmod will load a module with `modprobe` when needed, and possibly unload a module after a timeout period expires since its last use.

Modules are loaded into the kernel only if the **`modprobe.conf`** file is configured correctly. This is because the kernel internally uses (type) names such as "`snd`" or "`eth0`" for sound and NIC drivers, but the actual vendor's module is named something like "`CB4236.ko`" or "`3c509x.ko`". In this case, `modprobe` will fail to find the module the kernel is trying to load. (Check `dmesg` output for detected devices, and then search your LKMs or Google for the correct module name.)

As mentioned previously, the files in `/etc/modprobe.d/*.conf` can define ***aliases*** to map the actual names to the names the kernel expects. **These should be listed in the order the modules need to load.** Note, no alias is used for modules included with the Linux kernel distribution, as the internal and external module names are the same.

In addition, the `modprobe.d/*conf` (formerly called `modules.conf`) files can **list parameters** to be passed to LKMs whenever they are loaded, similar to passing parameters to base kernel subsystems in GRUB (discussed below). Finally, the file can **list arbitrary commands to be run either before or after a module is loaded or removed**. This can be used, for example to un-mute your sound system (when loaded, ALSA (Linux sound module) initializes to a muted state).

> Modern Linux kernels and `modprobe` can examine the boot parameters passed in when the kernel was booted, for options to use when loading LKMs. (Older systems couldn't do that; you had to pass options for LKMs via `modprobe.conf`.) Such options are specified using the naming scheme *module-name.option-name=value*.

If you need to load a module manually, the `insmod` command can be difficult to use. For one thing, some kernel code is dependent on other kernel code. In the case of LKMs, you may find that one module must be loaded before another. If you try to `insmod` some modules out of order, the load will fail with strange error messages. (Sometimes the error message is displayed, other times it is sent to the ring buffer, which you view with `dmesg`.)

This problem is solved with `modprobe`. (The Solaris equivalent command is `modload`.) This command uses the file `/lib/modules/`*`kernel-version`*`/`**`modules.dep`** to determine module dependencies, and will automatically load prerequisite modules first. If you load a module with the right option it will auto unload too.

Some example uses of `modprobe`:

`modprobe -c` # list all modules, about ~2500

`lsmod` # list all currently loaded modules

`modprobe crc8` # loads kernel/lib64/crc8.ko and its dependencies

`modprobe -r crc8` # remove the module and anything that depends on it

`modprobe --show-depends` *`module`* # unlike modinfo, this knows about aliases

The `modules.dep` file is built with the kernel utility program **`depmod`**. ("`depmod -a`" is usually run at boot time just in case, but only really needs to be run when you change the kernel.)

> The `depmod` utility produces several files used by the kernel: the list of dependencies, a list of symbols (names) provided by each module, and binary versions of those (makes it quicker to use). `depmod` will also a produce `modules.devname` file, for when installing a module means the kernel/udev should create a new entry in `/dev`.

To disable a subsystem from loading, you can compile a custom kernel without that feature. If the subsystem is compiled as an LKM, you don't have to rebuild the kernel to prevent it from loading. For example, to disable Firewire without rebuilding the kernel, you need to identify the Firewire LKMs:

```
$ cd /lib/modules/3.4.9-2.fc16.i686.PAE/
$ find -iname \*firewire\* -o -iname \*1394\*
./modules.ieee1394map
./kernel/drivers/firewire
./kernel/drivers/firewire/firewire-core.ko
```

./kernel/drivers/firewire/firewire-net.ko
./kernel/drivers/firewire/firewire-sbp2.ko
./kernel/drivers/firewire/firewire-ohci.ko
./kernel/drivers/media/dvb/firewire
./kernel/sound/firewire
./kernel/sound/firewire/snd-firewire-lib.ko
./kernel/sound/firewire/snd-firewire-speakers.ko

Now we need to identify which is the main module (that the others depend on); that's the one to disable. Let's examine one module to see what it depends on:

$ modinfo firewire-ohci
filename:
/lib64/modules/3.4.9-2.fc16.i686.PAE/kernel/drivers/firewire/firewire-ohci.ko
alias: ohci1394
license: GPL
description: Driver for PCI OHCI IEEE1394 controllers
author: Kristian Hoegsberg <krh@bitplanet.net>
alias: pci:v*d*sv*sd*bc0Csc00i10*
depends: firewire-core
intree: Y
vermagic: 3.4.9-2.fc16.i686.PAE SMP mod_unload 686
parm: quirks:Chip quirks (default = 0, nonatomic cycle timer = 1, reset packet generation = 2, AR/selfID endianess = 4, no 1394a enhancements = 8, disable MSI = 16, TI SLLZ059 erratum = 32) (int)
parm: debug:Verbose logging (default = 0, AT/AR events = 1, self-IDs = 2, IRQs = 4, busReset events = 8, or a combination, or all = -1) (int)

Is "firewire-core" the main module? Let's check if it depends on any other firewire modules:

$ modinfo firewire-core
filename:
/lib64/modules/3.4.9-2.fc16.i686.PAE/kernel/drivers/firewire/firewire-core.ko
license: GPL
description: Core IEEE1394 transaction logic
author: Kristian Hoegsberg <krh@bitplanet.net>
depends: crc-itu-t
intree: Y
vermagic: 3.4.9-2.fc16.i686.PAE SMP mod_unload 686

This seems to be the main module. Let's see what config file we already have:

$ ls /etc/modprobe.d/

anaconda.conf blacklist.conf dist-alsa.conf dist-oss.conf dist.conf openfwwf.conf udlfb.conf

To me, it makes sense to create a new config file (you can name these after the module, and have many files< or just one big file as shown here):

 # echo 'alias firewire-core off' >> /etc/modprobe.d/local.conf

(This works because there is no such module as "off", so any attempt to load `firewire-core` will fail.

> Note, the "blacklist" command doesn't work as expected; "`blacklist firewire-core`" won't prevent that module from loading as a dependency or manually. Red Hat recommends using the command "`install` *`module-name`* `/bin/true`" to prevent loading.

The format of files under `modprobe.d` (and `modprobe.conf`) is: one command per line, with blank lines and lines starting with "#" ignored (useful for adding comments). A backslash ("\") at the end of a line causes it to continue on the next line. Only files named `*.conf` in `modprobe.d` are examined. The man page for these files is `modprobe.d(5)`.

Sample Old `/etc/modprobe.conf`

```
alias eth0 e100
alias scsi_hostadapter1 aacraid
alias usb-controller ohci-hcd
alias char-major-116-* snd
alias sound-service-*-0 snd-mixer-oss
alias sound-service-*-1 snd-seq-oss
alias sound-service-*-3 snd-pcm-oss
alias sound-service-*-8 snd-seq-oss
alias sound-service-*-12 snd-pcm-oss

install sound-slot-* /sbin/modprobe
snd-card-${MODPROBE_MODULE##sound[_-]slot[_-]}

install snd-pcm /sbin/modprobe --ignore-install
snd-pcm && /sbin/modprobe snd-pcm-oss &&
/sbin/modprobe snd-seq-device && /sbin/modprobe
snd-seq-oss

options ip_conntrack_rpc_udp ports=7938
```

Note that unlike standard kernel modules, **third-party modules must be compiled under the kernel you expect them to run with**. Modules really are a part of the kernel, which is really one large program. If you don't

reboot before compiling third party modules, they will link to the wrong addresses when interfacing with the rest of the kernel!

Run `modinfo` *`modname`* (as root) to see information about a module, often including a description of its parameters.

> **NDISwrapper** is special software that can translate between Windows32 driver API and the Linux API. It was designed for NICs and Wi-Fi cards that lack Linux drivers. You install a (legal copy of a) Windows driver (a pair of files, *`foo`*`.sys` and *`foo`*`.inf`) and use `ndiswrapper` build a kernel module from it that you can load. Since Windows drivers are notorious for bypassing the Win32 API, this won't always work. But it is worth a try if you can't find a Linux/Unix driver.
>
> To Use NDISwrapper on Linux:
> `yum install kmod-ndiswrapper`
> `ndiswrapper -i` *`foo.inf`* # creates module
> `ndiswrapper -m` # adds conf info to /etc/modprobe.d/ndiswrapper

To build third party modules (such as NVIDIA drivers or VirtualBox Additions) requires an appropriate set of tools, development libraries, and kernel sources (or at least some of them). I would suggest you make sure you have the following installed (for Fedora): `kernel-doc`, `rpmdevtools`, `kernel-PAE-devel` (or `kernel-devel`), and `kernel-headers`, as well as the package groups "`Development Libraries`" and "`Development Tools`".

DKMS

Users generally need the latest version of device drivers. This is no problem if you rebuild the kernel using the latest source each time either the base kernel or any driver is updated, but what happens if you use a "stable" kernel such as Debian? If you don't update the kernel, you don't get the latest driver versions.

A related problem is with third party drivers. If you do update your kernel, some third party driver may not be updated yet and thus will break. This is common since many vendors need time to test the new kernels with their device drivers before releasing them.

An alternative to kmod managed modules for Linux can solve these issues, and is known as ***Dynamic Kernel Module Support***, or **DKMS**. This keeps the source code (or object files for proprietary code) for kernel modules (that are not part of the Linux core) on the system so they can be recompiled (or re-linked) as needed to match kernel upgrades, or to allow the source to be updated and recompiled for the current kernel version. (Recompiling can take an annoyingly long time.) System administrators can update kernels

and/or drivers independently. The DKMS config file for some module can list the kernel version it works with as *required*. This will block any kernel updates until the module has been updated for the new kernel. This is useful for critical modules, such as for OpenZFS if you're using that.

The drivers are kept under some form of version control systems too, so if the new driver doesn't work you can revert to the original working version.

DKMS was invented by Dell as a way to push updates to their Linux kernel modules, and is now widely adopted. See the DKMS man pages and `/etc/dkms.conf` files for more information.

Kernel Module Troubleshooting

The most common problems relate to device drivers not loading automatically. This is the time you may need to edit `modprobe.d/*conf`. First, examine the logs for the messages about which subsystem had the error. Suppose it is "`snd`". You need to figure out the actual name of the correct driver to load. Use `dmesg` to see what sound hardware is used. This often will reveal the driver name. If not you may need to resort to the vendor's website or Google.

Once you know the name, find and manually load that module (`modinfo` and `modprobe`). Now see if the system can use the hardware correctly. It may not since often multiple drivers are needed for a complex bit of hardware such as a modern sound card. So repeat, loading modules one at a time, until it works.

Now edit one of the `modprobe.d/*conf` files with the correct `alias` commands, in the correct order. The kernel should now be able to load the modules automatically.

> On Linux, each loaded module has a directory `/sys/module/`*`modname/`* that contains information about that LKM. You can also change parameters to a module with:
> `echo` *`value`* `> /sys/module/`*`modname`*`/parameters/`*`param`*

Discuss kernel *panic* (an un-recoverable error; the Unix version of the MS BSoD or Apple sad-mac.)

Daemons and `init` (*Discussed in detail in Admin I*)

The init process consults some configuration file(s) (such as **`/etc/inittab`** for Sys V init) and executes various startup scripts to *mount* filesystems, initialize the clock, the firewall, the NIC(s), etc., and to start *daemons* such as `sshd` and `httpd`.

Which daemons are started and which startup scripts are run depends on the *run-level* for Sys V init. This is often implemented by a program `rc` that consults various run-level directories (`/etc/rc#.d`) and runs the appropriate scripts found. (In practice, the run-level directories contain symlinks to the scripts in `/etc/init.d`.)

Modern systems start dozens of daemons (50–250); there are often dependencies from one service to the next. (Example: the mail server usually requires the network to be up before starting.) Controlling the order of these services is tricky, as is restarting services when their configuration changes, or if they crash.

Solaris 10 has a new scheme for starting services (SMF), as do most Linux systems (**upstart** or **systemd**). The new systems specify *events* that occur, and actions to take when those events occur.

SMF, Upstart, and Systemd don't use *run-levels* to determine which services to start; the SA simply configures the system to start or not start each installed daemon.

Currently, all init systems still support run-levels, so you can use the Sys V init system and the `rc` scripts. For Solaris they generally will invoke **`svcadm`** (part of Solaris10 *service management facility, or SMF*) utility, which maintains the dependency information in an XML configuration file. So starting a daemon will automatically start all the services the daemon depends on. The Linux upstart system doesn't do this.

Solaris Kernel Configuration

You can configure the kernel in some ways, and add modules (and patches) to it. (Solaris packages usually contain device drivers.) Solaris kernel files are found in:

`/kernel and /usr/kernel` — **standard modules**
`/platform``/`*`platform-name`*`/kernel` — platform specific modules
`/platform/`*`hardware-class`*`/kernel` — hardware specific modules

The main Solaris configuration file is `/etc/system`. To see the current entire configuration (including hardware), use the command **`prtconf`** or for more detailed information, use **`sysdef`**. To just see system version use the **`uname -i`** (the *platform-name*) and `uname -m` (the *hardware-class* name). Information about kernel modules can be found using the **`modinfo`** command. They can be added with **`modload`** and removed with **`modunload`**.

To add Solaris LKM device drivers, use **`pkgadd`**. (See also `pkginfo`, `pkgrm`, and `pkgtrans`.) If the driver (say `foo.o`) doesn't ship as a Solaris package, you need to copy it (and its configuration file `foo.conf`) to the directory `/platform/`*`hardware-class`*`/`*`kernel`*`/drv`, and then load it into the kernel with the command **`add_drv foo`**.

To add `foo.pkg`:

```
mkdir /usr/local/foo
pkgtrans foo.pkg /usr/local/foo all
pkgadd -d /usr/local/foo all
```

Like Linux, Solaris includes a `/proc` interface to the kernel where you can view and change tunable parameters.

Lecture 10 — Methods of Configuring the Kernel

The kernel rarely needs configuration changes. Modern systems monitor themselves and available resources, and adjust settings as needed. There are times however where it can be useful to change kernel settings. There are several ways to do this, discussed below. (Alternatively, you could modify kernel source code and build a custom kernel with the settings you want. That is rarely done anymore.)

When might you want to change some default kernel parameter? There are many reasons: When you use special hardware. If you run a busy database server, you might need to adjust the number of open files permitted per process. If you run a busy web server, you might need to configure some network settings for best performance. If you run a host in a hostile environment, such as in a DMZ or outside of a firewall, you may need additional security settings. And so on.

Initializing Kernel Subsystems

Each kernel subsystem may be initialized with ***parameters***. You can pass parameters to subsystem compiled in the base kernel on the **bootloader command line** (the `kernel` line in grub). You do this by naming the subsystem and passing a string with no spaces in it (the *subsystem=parameters* are separated by a space):

```
kernel ... subsystem=parameters
```

Such parameters (really a single string of text with no spaces) have no standard format. Each subsystem can define any parameters it wants (or none at all). Typically, device drivers are passed parameters identifying their IRQs, I/O addresses, DMA values, and other information, e.g., `foo=name1=val1,name2=val2,...`. See `bootparam(7)` and `dracut.kernel(7)` man pages on Linux as well as `/usr/share/doc/kernel-version/Documentation/admin-guide/kernel-parameters.txt`), and man `kernel` (Solaris). (Since Fedora 21 kernel docs are not installed by default. Either install the kernel source or view docs online at https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation or at https://www.kernel.org/doc/Documentation/.)

> Many parameters can be passed to the Linux kernel; the ones it doesn't recognize will be passed to the `init` (systemd) program. (This doesn't happen on Unix, but see the Solaris `eeprom` command for some additional parameters that can be passed to `init`.) The ones not recognized by the kernel or `init` are set in the environment of `init` (Linux only). Such settings can be used by various bootup RC scripts.

These additional parameters are not documented anywhere!
However, Linux collects all kernel parameters in `/proc/cmdline`. The various `init` scripts (and `init` itself) examine this pseudo-file to see if some parameter was set.

Because of this, you can find out what parameters have meaning on your Sys V init-based system, by:

```
grep "/proc/cmdline" /etc/rc.d/* /etc/init.d/*
```

(Hopefully the parameter name, or the script that uses it, will provide you a clue as to what the parameter is used for. (Try this: what kernel (really `init`) parameter(s) can you pass with GRUB, to force `fsck` on the next boot?)

Systemd supposedly documents these in various man pages, if you guess the right one to examine. Some may still be undocumented however. Try to guess the program responsible from those in `/lib/systemd`, then use the `strings` command on it; in the output, you should see `/proc/cmdline`, and usually after that is the parameters (and filenames) the systemd command uses.

LKMs can also be passed parameters when they are loaded. These are configured by setting the options in the `modprobe.d/*conf` file on Linux, or by using the syntax *module-name.parameter=value* on the kernel loader command line. See **modprobe.conf(5)**. (Note, passing parameters via the boot loader has no effect on LKMs.)

You can specify additional modules to be loaded at boot time by the `systemd-modules-load.service` daemon by creating a *`program`*`.conf` file in the `/etc/modules-load.d/` directory, where *program* is any descriptive name of your choice. The files in `/etc/modules-load.d/` are text files that list the modules to be loaded, one per line, along with blank and comment lines. See modules-load.d(5) for details.)

On Solaris, module options are set in `/etc/system`. Unlike `modprobe.conf` these options are read at kernel load time only, so changing the options and re-loading a module won't use the new settings!

Very old Linux used a boot loader as the first block of the kernel, so no additional loader was needed. To *tweak* an existing Linux kernel (to change where the *root* partition is, for instance when making a boot floppy or using a *RAM disk*), you would use the **`rdev`** command to adjust the parameters in that boot loader.

Changing Tunable Parameters

The kernel values that can be changed are called ***tunable parameters***. Such parameters may be used to initialize kernel subsystems as discussed above. (Early Unix systems kept these in a kernel source code file, and include kernel object files so you could rebuild the kernel to use your changes.) Most tunable parameters today can be changed at any time. Such changes take effect immediately but are not persistent; they affect the running kernel only (in RAM only) and changes are lost after a reboot.

> A POSIX system's required settings (*system parameters* or *variables*) that apply system wide are listed in the man pages for **`sysconf`** (e.g., `PATH`). Some system variables depend on the specific filesystem used. These are listed in the man page for **`fpathconf`** (e.g., `NAME_MAX`). Such settings are not tunable parameters, but read-only.
>
> The values for all POSIX mandated system variables (and some others) can be seen using **`getconf`**`(1)`. Because the variables defined in `fpathconf` depend on the filesystem used, they require a pathname argument too:
>
> ```
> getconf PATH;  getconf NAME_MAX /
> ```
>
> (On Linux (Gnu) you can see all settings with `getconf -a`.)

To modify kernel parameters in the live kernel, you can use the "fake" filesystem **`/proc`.** `/proc` is an interface to the kernel parameters and status information, in what appear to be ordinary files. Actually, they are a live view of some of the kernel's memory. You can examine these settings using `cat`.

Some of the files under the subdirectory `/proc/sys` are writable as well as readable (generally, only by root). To change any of those parameters, you merely overwrite the file using `echo`, for example:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 32768 > /proc/sys/fs/file-max
```

This will enable routing on your Linux system and will set the maximum number of files a process can open at once to 32,768.

> A description of all the information and settings available through the `/proc` interface can be found on the `proc` man page, or at kernel.org, or in the kernel documentation (e.g., "`sysctl/*`" and "`networking/ip-sysctl.txt`").
>
> On Linux, the fake files show as text. On Unix, they are often binary values.

Remember, such changes are lost when the system reboots. To make changes permanently, you can place such commands in the `rc.local` boot file or the equivalent for non-SysV init.

Most systems have a more convenient alternative, **`sysctl`**. You define the changes in **`sysctl.conf`** (or `/etc/sysctl.d/*.conf`) and at boot time all the settings defined there are applied.

To make these changes permanent, you must add the `echo` commands as shown above to the appropriate boot `rc` shell script or define the settings in `sysctl.conf` like this (show on YborStudent):

```
net.ipv4.ip_forward = 1
fs.file-max = 32768
```

Case Study: The Linux kernel is not setup for virtualization; running VMware or other virtualization that "hosts" guest OSes can easily cause *out of memory* errors even when there might be enough. That happened to me. Two poorly documented kernel parameters can be adjusted to help:

```
vm.lowmem_zone_ratio = "100 32 32"
```

(The first value is the one to change, defaults to 256) and

```
vm.dirty_ratio = 5
```

(Defaults to 10%.) The first change more aggressively protects the memory used by the hypervisor to prevent crashes. The second causes unused memory to be reclaimed more frequently (the kernel does it without waiting for the kernel swap daemon to kick in).

Note that some configuration information (such as the IP address to bind to some Ethernet NIC) is not stored in the kernel nor passed as subsystem parameters, but is kept in various configuration files under `/etc`.

On Solaris, you use the **`ndd`** command similarly to `sysctl`, and **`/etc/system`** similarly to `sysctl.conf`. Currently `ndd` only examines/changes TCP/IP kernel parameters; some are changed with **`routeadm`** instead. Other parameters are changed by editing `/etc/system` and rebooting, by editing `/etc/default/*` files (Linux has some of these too), or by using a debugger on kernel RAM (**`kmdb`** or `mdb`).

To edit `/etc/system`, modify/add lines like this: "`set` *`module:param=value`*". (Always make a copy of the `system` file first, so you can boot from it using `boot -a` if you goofed!)

To change the value of the integer parameter `maxusers` from 495 to 512, you must edit the in-memory kernel image. Do the following on Solaris (note: 512 decimal = 200 hex, '/D'=display, and '/W'=write):

```
# mdb -kw
Loading modules: [ unix krtld genunix ip
logindmux ptm
 nfs ipc lofs ]
> maxusers/D
maxusers:
maxusers: 495
> maxusers/W 200
maxusers: 0x1ef = 0x200
> $q
```

Solaris parameters are not set from a single place, but with many commands and files. In addition to the above, see the `cfgadm*` and **`sysdef`** commands.

Example: Configuring the Kernel to Support Additional Executable Formats

In addition to running machine code ("native") executables, the kernel supports a *she-bang* mechanism to be able to launch scripts of any sort. But some other types of executable files exist, such as java `.jar` files, .NET (mono) files, `.exe` (windows) files, and so on. These are called alternate *binary formats*, or "***binfmt***".

Normally, making such an executable executable (:-) would cause an error when you tried to execute it; the kernel by default only recognizes native executables or text files, which it assumes are shell scripts. To run such alternate executables, you normally need to launch some interpreter. For text file scripts, you can use the kernel's she-bang mechanism. But that isn't possible for non-text (binary) executables. You generally need to run them as "*`wrapper_script myapp`*", where *wrapper_script* runs the application correctly, for instance starting wine, mono, or the JRE, or starting some daemon first.

The Linux kernel can be configured to recognize such files and launch the appropriate command (or wrapper script) to execute them the same as native apps (just double-click or type the name at a prompt). There is a `/proc` interface where the sys admin can add configuration for "`binfmt_misc`", stating how to recognize such files, and the command line to use for them. `Binfmt_misc` recognizes the binary-type by matching some bytes at the beginning of the file with a magic byte sequence (masking out specified bits)

you have supplied, or can recognize a filename extension. To use `binfmt_misc`, make sure it is mounted:

```
# mount binfmt_misc -t binfmt_misc \
    /proc/sys/fs/binfmt_misc
```

(Systemd should handle this now.) Then, to register a new binary type, you have to set up a string looking like:

```
:name:M:offset:magic:mask:interpreter:flags
```

or to use extension recognition (case-sensitive, and no dot; e.g. "exe", "EXE", or "jar"):

```
:name:E::extension::interpreter:flags
```

(You can choose a different delimiter than "`:`" if needed). Just echo the string to `/proc/sys/fs/binfmt_misc/register`. Once registered, an additional entry is available in `/proc/sys/fs/binfmt_misc/status`, and a new "file" is made for the registered format as `/proc/sys/fs/binfmt_misc/`*`name`*. You can disable one (or all) registered formats by echoing "`-1`" to the `.../`*`name`* file (or all of them by using the `.../status` file). See bindfmt_misc.rst kernel documentation for details.

For example, **to enable support for running executable `.jar` (Java) files**:

```
# echo
':ExecutableJAR:E::jar::/usr/bin/jarwrapper:' \
  > /proc/sys/fs/binfmt_misc/register
```

Then create a shell script (or other executable type) called `jarwrapper`:

```
#!/bin/bash
# jarwrapper - the wrapper for binfmt_misc/jar
java -jar $1
```

Systemd also supports `binfmt_misc`, with similar config strings but the strings are put into files, in `/etc/binfmt.d/`. Even if not using systemd, you should set up such a directory, and have some boot time script re-configure your "binfmts".

An example file to run `.exe` (Windows or DOS) executables using the "`wine`" Windows emulator would be:

`/etc/binfmt.d/wine.conf`:

```
# Start WINE on Windows executables
:DOSWin:M::MZ::/usr/bin/wine:
```

Patching the Kernel

Modifying compiled in constants and rebuilding the kernel. (In Unix there is (or was) a `params.h` file with many ***kernel tunable parameters*** defined in it. All Unixes ship (or shipped) with sufficient source (and object files) to allow you to modify this file and rebuild the kernel.)

Modifying the actual kernel code. This is done by applying a ***patch file***. In the case of Linux, the patch contains the output of the `diff` program on the kernel source files. You apply the patch to the source, and then reconfigure and built the kernel. A few changes to the kernel image can also be made without rebuilding, using **`rdev`** on Linux.

To patch the Linux kernel, make sure you have the correct source code to which the patch applies. Kernel patches aren't cumulative (unlike MS service packs), **you need all the patches between the version you have (linux-2.6.x) and the version you want (linux-2.6.y), and must apply them all in order.**

```
# mv patch-2.6.X.bz2 /usr/src
# cd /usr/src/linux
# bzip2 -dc ../patch-2.6.X.bz2 | patch -p1
```

(Review *patch files* on page 96.)

The Linux script `patch-kernel` can be used to automate this process. It determines the current kernel version and applies any patches found. "`cd`" to the top of the source:

```
scripts/patch-kernel . [dir-containing-patch-files]
```

The first argument in the command above is the location of the kernel source. Patches are applied from the current directory, but an alternative directory can be specified as the second argument.

Linux kernels include *stable* kernels with names of linux-W.X.Y. To patch this to linux-W.X.Z, you must apply a single (cumulative) patch, not to linux-W.X.Y source, but to the linux-W.X.0 source. For example, to go from linux-2.6.16.2 to linux-2.6.16.5, you install the linux-2.6.16 source and apply a single linux-2.6.16.5 patch.

The kernel packages from some distro may not match the vanilla kernel source from `kernel.org`. The vendor usually adds some custom patches. You may obtain a list of any such patches by using the command on the Fedora kernel source package: `rpm -qpl kernel-`*`version`*`.src.rpm`

[*Reported by ACM Tech News 7/6/07*] Theoretically, the [Linux] kernel development process involves changes [(i.e., patches)] going from the original author, through a file driver maintainer, to the maintainer of a major subsystem such as PCI or SCSI, to Andrew Morton for testing, and finally to Linus Tovalds for [inclusion in] a kernel release.

The actual process, however, is far more complicated ... and a mess. The 2.6.22 release ... involved 920 developers, compared to 475 developers for the 2.6.11 release ... The "mess" is largely due to faster incorporation of new features, ..., the increasing number of kernel changes, and more rapid changes. Over the past two years, 3,200 developers have contributed at least one patch, with half that number contributing two or more, and one quarter contributing three or more.

Every patch has at least one author and one reviewer.

Patching Solaris

[*See "Solaris Patching" in CGS-2763*] For Solaris and other commercial Unixes, the patches are *binary* files that modify the kernel image file (and LKM files). To patch your kernel with a Solaris (binary) patch you use the "official" **`patchadd`** and **`patchrm`** commands. Use `patchadd -p` to see list of installed patches. Or use the other "official" tool `smpatch {analyze|download|add}`, download patches from (currently) `http://www.sun.com/sunsolve/patches/`.

In practice the most commonly used tool for patching Solaris (any version) is the third party "**`pca`**" Perl script. (This works post-Oracle too, but (a) you must log into SunSolve and accept the new Software License Agreement at least once, and (b) you must have a support contract to get any patches (before, critical/security patches did not require any contract.)

Solaris 10 has very bad performance of patching. It can take many hours, and you can't patch a running system! It is virtually impossible to apply any kind of large patch bundle to Solaris 10 while meeting a reasonable SLA.

One way of doing this for a system with mirrored (boot) disks would be to break the mirrors and use one of the sub-mirrors for ***Live Upgrade*** (LU). That works but the server has no redundancy until you have completed the upgrade, rebooted, and re-synced the mirrors. And the moment you rebuild the mirrors you no longer have a simple/quick way back to the old system. You would have to restore the boot disk from a backup, again taking a long time

You can use LU without breaking the mirrors by having an extra copy of your ***boot environment*** (BE) disk slice(s). Assume you used slices 1 (for root) and 3 (for `/var`). You just make the new boot environment (BE) from

slices 4 and 5, upgrade or patch onto it, then boot from it. The only outage is the reboot, and the back-out plan is simply another reboot from the original BE. (Not sure if this is needed with ZFS for the BE.)

Building a Linux M.X.Y Kernel

The steps haven't changed much since version 2.6, but may change at any time. You need to read the README file that comes with the source to see the steps you should take.

> If using UEFI secure boot, boot code and the kernel (and any LKMs) must be digitally signed. So if you build a custom kernel, you need to either sign it and its LKMs, or turn off secure boot. The process of signing varies between distros. You can either get your code signed my Microsoft for $99 (as of 2020), which will allow others to use your kernel easily. Or you can use a distro-specific way to sign your code. In general the process involves creating a new public-key pair, generating an X509 certificate, getting the UEFI boot code to recognize the certificate as trusted, and then signing the code with the private key.
>
> A good guide for Ubuntu can be found at ubuntu.com/blog/how-to-sign-things-for-secure-boot. For Fedora, see docs-ci-fedora-docs.apps.ci.centos.org or docs.fedoraproject.org/.../build-custom-kernel/. (A nicer but older guide is at jwboyer.livejournal.com.)

Below are the basic steps:

- Do any one-time prep, such as updates (and installing if needed) `gcc`, `make`, and other required tools. **Verify sufficient free space (quota)**, ~400MiB to unpack, ~2.1GiB to build. (You can build in a different location.) A useful set of items to install (not all of these are required in all cases):

```
sudo dnf install qt3-devel libXi-devel
(xconfig)
gtk+ gtk+-devel (for gconfig)
ncurses-devel   (for menu-config)
sudo dnf install elfutils-libelf-devel bison \
openssl openssl-devel flex ccache rpmdevtools \
rpm-sign grubby kernel-devel
sudo dnf group install c-development \
development-libs development-tools
# To sign the kernel and modules (not discussed further here):
sudo dnf install pesign mokutil keyctl perl
```

- Download kernel source (~60MB) and the `.sign` file. Verify the tar-ball with `gpg`: `gpg linux-`*`version`*`.tar.xy.sign`. If necessary

import the GPG key (the error message will tell you the key ID needed):
gpg --keyserver www.keyserver.net --recv-keys *Key_ID*

> Sometimes the whole compressed file is signed, but lately, only the tar archive is signed. You may need to uncompress the archive in order to verify it; you can also pipe the output of `unxz` through `gpg`.

- Unpack (`tar -xf linux-`*`version`*`.tar.xz`) in the correct location (say your home directory). If building as root (**not** recommended) under `/usr/src`, rename directories and create links if/as needed. Note that historically the location was `/usr/src/linux`, a symlink to `/usr/src/linux-`*`version`*. Today Linux may contain several kernels (for virtual systems) so a re-organization was needed. The new location is commonly `/usr/src/kernels/`*`version-arch`*`/`. Note the "PAE" versions of the kernel should be used especially if you have (or may get) more than 2GB of memory.

  Unpacking the tar archive should create a new directory `linux-`*`version`*. You should `cd` into that.

- Read the `README` file and other documentation as needed; also you can run **`make help`** to list available `make` targets.

- Initialize the kernel source (removes any old `*.o`, dependency, `.config`, and `makefiles`):
  `make mrproper` (Not needed with clean source, but it can't hurt!)
  (**`make clean`** only removes old `*.o` and dependency files.)

- Install any desired patches. OS specific patches may be required for some programs to work (e.g., Oracle requires RH patches). If you install the kernel source RPM (get with `yumdownloader`) it has RH patches already; you only need to configure it. See `fedoraproject.org/wiki/Docs/Customkernel` for more information.

- Configure the new kernel with the features you desire. You can use **`make *config`** will use an existing **`./.config`** file (which you should copy from your current kernel configuration to use that as defaults: `cp /boot/config-4.16-1.100_FC22 .config`). Now run **`make olddefconfig`** which reuses the working config and only prompts you for any new features and **should be run first** to update your `.config` file with the new items.

  Now is a good time to make a backup of the `.config` file.

  Then `make config` or `menuconfig` (TUI), or `xconfig` or `gconfig` (GUI), to make changes. These can also make a new `.config` from

scratch but will use an existing `.config` file if one is present. (I don't suggest trying to configure a kernel from scratch!)

- If you are building the same kernel version but with different configuration, you must set **`LOCALVERSION`** (previously `EXTRAVERSION`) to a unique name. Make a backup of the kernel modules used too. Ex: `-wp20180718`
- Determine which parts should be built as kernel modules. In the xconfig GUI, a check means "include in the base kernel image", a dot means "build as an LKM", and unchecked means to not include that at all.
- Be sure to configure a VGA Console, keyboard, and mouse, or you won't see anything or be able to type anything! Grep for `CONFIG_VGA` and `CONFIG_VT` in the `.config` file, make sure these are set to "`Y`". Also `grep` for `INPUT`, `KEYBOARD`, and `MOUSE` and make sure the relevant settings are "`Y`" as well.
- If you use ReiserFS, the Kernel Preemption setting may cause problems. However, you need that setting in 2.6 and later, or your desktop will be painfully slow! Until ReiserFS is fixed, you can work-around by mounting such partitions with the `nolargeio=1` option in `/etc/fstab`.
- Some hardware drivers have problems with ACPI and APIC. If you configure these options, you can turn them off with kernel parameters of `noapic acpi=off`.

- Save a copy of your final `.config` file somewhere; if you restart the config process or run "`make mrproper`", you will lose your config. You can also run `diff` on this and the version saved earlier.

Most people use the kernel shipped by distros. But some people like to compile their own kernels from kernel.org; configuring your own kernel, can be a difficult and tedious task. There are too many options, and sometimes userspace software will stop working if you don't enable some key option. You can use a standard distro `.config` file. However, it can take a very long time to compile all the options it enables.

To make kernel configuration easier, a new build target has been added: **`make localmodconfig`**. It runs `lsmod` to find all the modules loaded on the current running system. It will read all the `Makefiles` to determine which `CONFIG` options enable those modules. It then reads the `Kconfig` files to find the dependencies required. Finally, it reads the current `.config` file and removes any modules not needed to enable the currently loaded modules.

With this tool, you can strip a distro's `.config` of all the unused drivers that are not needed! This saves lots of time when building a kernel.

There's an additional "`make localyesconfig`" target, in case you don't want to use modules and/or `initrd`s.

- Build the base kernel with "**`make [V=1]`**" (`V=1` means verbose) and standard modules. (To build but not install, use `make modules`.) Remember any third-party modules must be built later. **Takes about 100-150 min.** Ignore the (scary) warning messages with `V=1`, but don't ignore error messages. "`make all`" is the same as "`make`".

If your system is multi-core, use "`make -j`*n*", *n* = # cores as shown by "`grep processor /proc/cpuinfo |wc -l`". Use that option on all the `make` commands you issue. On a system with 24 cores, the build took under 10 minutes!

- Install the kernel modules as root ("**`make modules_install`**") in the correct locations and with the correct names. (Names must match the bootloader configuration.) (You can use "`-j`*n*" here too.)
- Install the kernel itself. On Red Hat systems, you can use (as root) "`make install`", but on some distros, you need to manually copy the files from your local directory into `/boot`.
(`cp` **`arch/x86/boot/bzImage`** `/boot/vmlinuz-`*version*; and **`./System.map`** too.) Note, RH puts the `Documentation` directory into `/usr/share/doc/kernel*/Documentation`. Create any needed links.

You can list multiple targets to `make` and they run in order, so you can use (as root, although only the last two steps require that):
`make olddefconfig xconfig all modules_install install`

- Run additional steps if desired: `make mandocs installmandocs rpm` (As usual, any step that installs files into system locations requires root priviledge.)
- Create an initial RAM disk if needed (using `dracut` *file version*). The default filename is "`/boot/initramfs-`*version*`.img`"; use that.
- Update the bootloader to run the new kernel (and select which kernel to load by default). Note there is a **`make install`** target you can use to copy `bzimage`, `System.map`, `initrd`, create the required symlinks, and re-writes `lilo.conf` or `grub.conf` for you. It uses the `/sbin/`**`installkernel`** shell script to do the work. (If you don't

have this, it will default to attempting to update `lilo` only, not `grub`!) On Fedora this invokes other scripts and eventually a program called `grubby`. However, runing `make install` may not set your new kernel as the default one to boot, so you may need to run some additional command.

- Reboot **using new kernel**. If this fails, boot the old kernel. Then examine the error messages, re-configure and re-build the kernel, and try booting again until successful. Note you can verify which kernel is running using "`uname -r`".
- Adter rebooting, rebuild any third-party modules and install them. (Remember that non-kernel modules must be compiled under the kernel they are for! So you must reboot the system to the new kernel before building these.) For example, for NVIDIA:

```
# sh NVIDIA-Linux-x86-1.0-6629-pkg1.run --
extract-only
# cd NVIDIA-Linux-x86-1.0-6629-pkg1/usr/src/nv/
# make install
# modprobe NVIDIA
# telinit 5
```

- Have a party and brag to your friends about your kernel building skills.

Lecture 11 — Networking Concepts Overview

What is a network? Ans: Computers (a.k.a. ***hosts***) able to **share information and resources**. A network may be small (one geographical location) or may cover the globe. Different technologies are used in each case. For a ***LAN*** we use technology such as Ethernet, which broadcasts packets so every station sees them. For a ***WAN*** we use slower, error-prone serial links that connect one LAN to another (point-to-point connections).

Qu: What is needed physically? ***Media*** such as a cable between each pair of computers (called a *mesh*), or in a *ring*, or in a *bus*, or in a *star*. (***Network topologies***.) (Show bus/star with 4 hosts.)

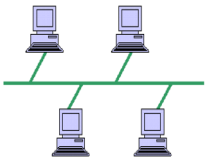
Computer cables can be tricky to work with. They don't work if too long, and they work poorly if kinked, if not terminated correctly, or if improperly grounded. When attaching a connector to the end of a cable, if you straighten out 0.5in at the end more than you should, the 100Mb/sec cable will only support about 30Mb/sec!

Besides being more delicate then commonly supposed, there are safety issues with standard cables. Such cables are clad in PVC, a strong, durable, flexible, and cheap insulator. However, in a fire the cables can get hot and then they give off deadly chlorine gas! In air spaces where people might be, you need to use more expensive (but safer) ***plenum cable***. There are various building and safety codes to consider as well. In the end, you should consider using a licensed cable installer.

Qu: What would happen if two or more hosts transmit at the same time? Ans: a ***collision***. To permit communications all parties must agree to a set of rules, or ***protocols***. Today the most common set of protocols for a LAN is called ***Ethernet***.

In addition to media and protocols, you need some hardware to connect the host to the media. Called a ***NIC*** (***Network Interface Card***). In Linux these have names such as "**`eth#`**", where "#"=0,1,2,... In Solaris, they have strange names such as "`elx#`" depending on the manufacturer/chipset of the NIC, but many modern NICs are simply known as "**`hme#`**".

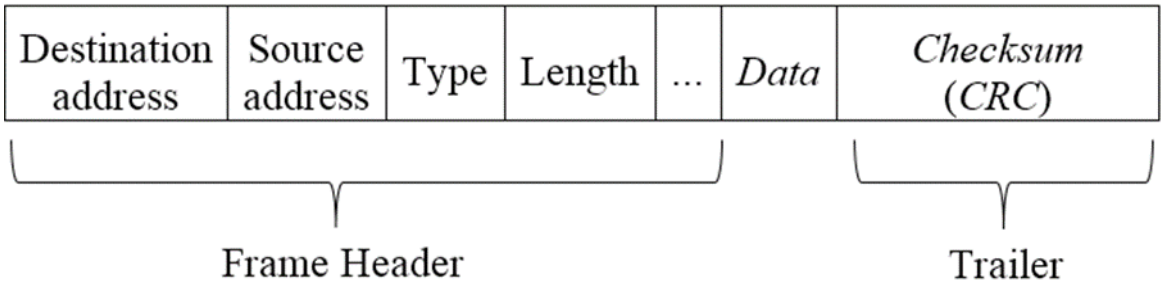
Your operating system needs the correct driver software to allow applications to send messages to the NIC. Then some application can send a message to another host by invoking the proper API function. The data will be passed to the NIC and then sent on its way.



Qu: (point to diagram) if *this* host wants to send a message to *that* host, how does it do that? Ans: each computer needs a unique ***address***, so when one computer sends data to another, the intended

recipient knows the data was meant for it. The other computers on the network are supposed (!) to ignore the message. In the old days, the administrator manually set each NIC with a unique number between 1 and 255. Today, NICs come configured with an address already, known as the ***MAC Address*** (or BIA, *data-link* address, ...). For Ethernet NICs, this address is 6 bytes (48 bits). The first three bytes uniquely identify the manufacturer, assigned by the IEEE. The last three bytes are a unique serial number.

For host A to send data to host B, host A must build a ***packet*** containing the data plus a ***packet header*** which contains the destination MAC address and other information (e.g., packet length, type of data, ...). Besides the header and data, a *checksum* (**FCS or *frame check sequence***) is appended to the end. This header and checksum are sometimes called ***framing***, and these packets are sometimes called ***frames***:



Overview of communication: Application invokes API function, passing it the address of the recipient and the data to send. API function builds the ***packet*** from this info and sends the packet to the NIC. The NIC sends out the packet onto the media, one bit at a time, according to the network protocol. If the data is very large, the API function will split it into several packets, which get reassembled at the destination. (Example: FTP a large file; Ethernet max packet size is 1522 bytes, including 22-bytes of header.)

Most NICs will examine only enough of the packet to see if it was intended for them or not. If not, they stop looking at it. The intended destination NIC will read in the whole packet, compute a checksum, and compare it with the checksum at the end of the packet. If they don't match, the packet is corrupted and must be sent again.

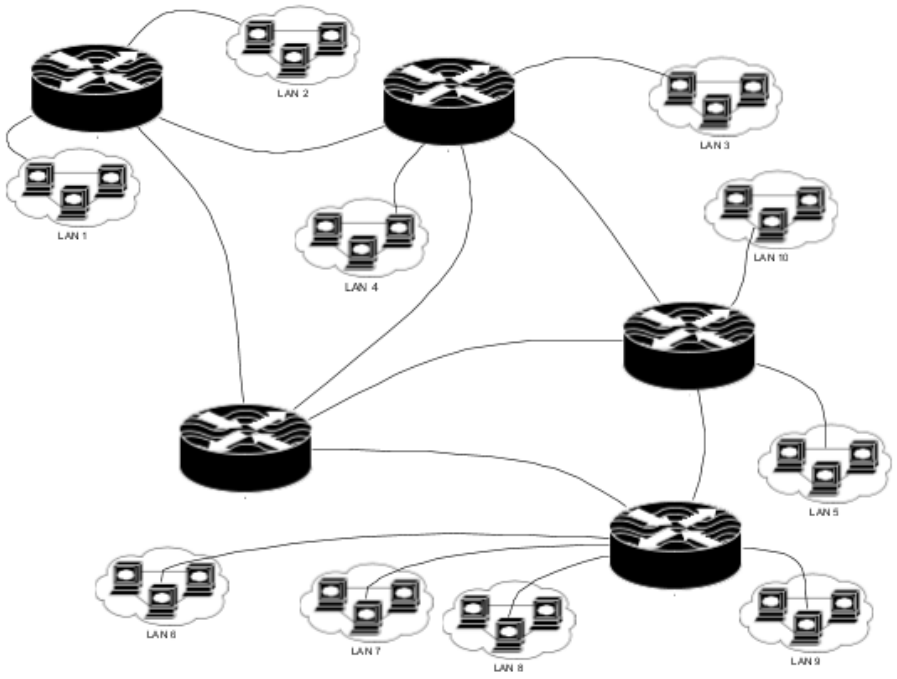
Internetworks

With media, NICs, protocols, addresses, and software, a network can function. But there are problems: What if one host on your network wants to send a message (a packet) to a host on a different network? Sending data between networks is called ***internetworking***, and a collection of networks that are connected are called an ***internet***. (The "Internet", with a capital "I",

refers to the global internet that connects nearly every network with every other. Lately, the popular press has stopped capitalizing it for some reason.)

The obvious solution of connecting both networks into one large network doesn't work. The technology for LANs has strict size (number of hosts) and distance (meters not kilometers) limitations.

Instead a more complicated solution is used. A device called a ***router*** (sometimes called a ***gateway***) with two or more NICs is used to connect the ***LANs***.



The sending host on the first LAN sends the packet to the router (to the NIC connected to its network). The router then resends the packet out a different NIC that connects to a second LAN. Now all hosts on that LAN, including the destination host, see the packet.

For this to work, every host must know the address of the router interface connected to its network. And when the packet is received by the router, it must somehow determine through which NIC to send the packet out. The router then needs a list of all addresses on all networks. The situation is made worse since not all networks connect to the same router. It is often necessary for the first router to forward the packet to another, and then another, etc., until the packet reaches the final network.

All hosts need to know the IP (*internet protocol*) address of the router. This can be set with the old `route` command or the newer `ip` command on Linux. This IP address can be stored in a file and used with the `route` command automatically when the network is brought up. With Fedora, use the file `/etc/default-route`, and in Solaris use

`/etc/defaultrouter`. (This information is often stored in different files, even on Solaris and Linux!)

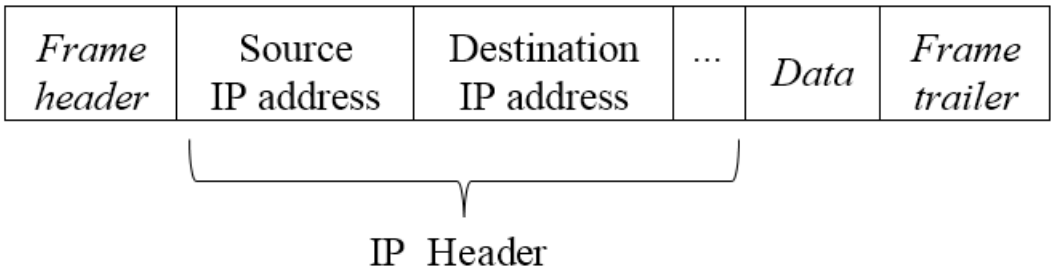
> Actually, there are other ways for a host to send packets to a router, that don't require knowing its address. They are not commonly used however.

The most common internet protocol suite in use today for this is ***IPv4***. A newer version called IPv6 is available, as are competitors such as IPX. The common name for the IP suite is ***TCP/IP***.

Addressing

For one host to send a packet to another, it must know the address of the destination host. This leads to a problem of finding out the addresses of all the hosts in the world. Why not use MAC (Ethernet) address? Huge unmanageable routing tables, administration problems (e.g., firewall configuration).

The most popular answer (there are others) is to assign an ***IP address*** to each NIC. **IP addresses have two parts: network number and a host number**. The idea is that routers only need to keep track of the various networks in the world, not all the host numbers. So, for one host to send a packet to another, it must have the IP address of the destination host. If the network number is the same for both hosts, the packet is just broadcast on the LAN as normal. If the destination is in a different network than the source, the sending host sends the packet to a router instead, trusting the router to forward the packet on its way. Once a packet is delivered to the correct network, Ethernet (MAC) addresses are used to deliver the packet to the correct host, as discussed previously.



IPv4 addresses are 32 bits long. (This isn't a lot of addresses! This changed to 128 bits for IPv6; see RFC-3513.) They are most commonly written in ***dotted-decimal*** notation: 10.3.200.42. The 32 bits are divided into two parts: the network number and the host number. Each LAN must have a unique network number, assigned by your local ***ISP***, who bought a block of them from a regional provider (ARIN), who in turn gets huge blocks of numbers from the IANA. ISPs lease them out to you and me.

A single host may have several NICs (so do routers). It is important to remember that **it is not the host that has an address, it is the NIC**. So a

host with two NICs has two addresses. Also, a single NIC may have multiple addresses. This is known as ***IP aliasing***. (e.g., `eth0:0`, `eth0:1`, ...)

> Modern Linux networking allows multiple IP addresses on each NIC without aliasing. However older tools such as `ifconfig` do not know about these newer features. Use new tools such as the `ip` command to work with networking.

The LANs still use Ethernet to sent packets locally. But hosts on a network only know the MAC address of NICs on that network. **So how does the sending host lookup the MAC address of some other host, given only its IP address?** One way is to keep a file of IP to MAC addresses on each host, and to update it regularly. For every host in the world.

A better way is to use ARP (*address resolution protocol*). The ***ARP*** protocol is used to map IPv4 addresses to MAC addresses, so sending hosts do not need to know the MAC address, only the destination IP address.

Illustrate this protocol: (1) *source* (local) host determines if *destination* (remote) host is on same network. If so, then (2) broadcast ARP request for destination host MAC address. If the destination host is on a different network, then broadcast an ARP request for the MAC address of the *gateway* (a router that connects one network to another). (3) wait for ARP reply. (4) now send packet to destination (or gateway).

Hosts maintain an ARP cache to save a lookup. To view the ARP cache, use the command **`arp -an`**.

> A (possibly) useful analogy: Suppose the teacher wants to ask a student a question. If they are in the same room (same LAN, a.k.a. same data link), the teacher just waits for a quiet moment in the room, and says "you there, in row 2 seat 4, ...". This is what Ethernet does. But this won't work when the teacher is in a different room at the time.
>
> Suppose the teacher is in their office (DTEC-404) and wants to ask the student "Hymie" in classroom DTEC-461 a question. In that case, the teacher asks a lab tech (a "router") to forward a message to a student "Hymie" in DTEC-461. The lab tech doesn't know which desk Hymie is at (Hymie's MAC address), but it doesn't matter; the lab tech simply goes to DTEC-461 and, in a quiet moment, shouts "Where is Hymie?". Hymie replies, "I'm in row 2 seat 4". The lab tech can now deliver the message to that student. Delivering messages between LANs (or classrooms) is what TCP/IP does. Analogy aside, when your computer sends a message to another, it must decide if the other computer is on the same network or not; if so just broadcast the message on the LAN; if not, send the message to a router (the "default gateway") for delivery to another LAN.

RARP (which is related to **BOOTP** and **DHCP** protocols) does the reverse: Given a host's MAC address (which is all the host typically knows when it boots up) it asks a server for its IP address. This is useful when you don't wish to configure each and every host in your organization individually. You can do this of course, by putting the host's IP address, the gateway router IP address, and other information in configuration files the host can use at boot time. But it is easier to have a single DHCP server on the LAN. When the host boots up it broadcasts a DHCP request packet containing its MAC address. The DHCP reply contains all the required network parameters.

Note your own computer has a virtual NIC with the address **127.0.0.1** (the ***loopback*** address, usually referred to by the name ***localhost***).

Port numbers and Sockets

Sending a packet to a host isn't enough. When the destination host gets the packet, what program should it send it to? (Web server? Email server? Telnet?) Part of the layer 4 header includes a ***port number*** to identify which program should receive the packet and which one sent the packet. These are 16-bit values. (Example: a web browser with two windows open. You click a line on one, switch to the other and click a different link. Each browser window's HTTP request packet will use a different source port number so the replies will be sent to the correct window.)

When a host receives a packet, the kernel will check the port number to see to which process to send it.

Show packets with Wireshark.

So how does a client (say a web browser) know which port number corresponds to a server? The servers listen for a particular port number that all agree on (**IANA**). The standard servers use ***well known*** **Port numbers** in the range 0–1023. Which service (and its application level protocol) uses which port number is documented in the **`/etc/services`** file. These Ports (both TCP and UDP) are reserved for public services such as FTP (20 and 21), telnet (23), SMTP (25), and HTTP (80), HTTPS (443). This makes it easy for clients; to contact your web server the client will send the request packet to your IP address and destination port 80. Note that on a Unix system root-privileges are needed to listen in on a well-known Port. (This prevents a user from crashing your web server and then starting their own, fooling people who visit your web site!)

The range 1024-49151 are **User (Registered) Ports**, used for other public services (such as Unix `rlogin` or the w3c `SSL` services). These are also registered by IANA (as a public service.)

The **Dynamic and/or Private Ports** are those from 49152 through 65535. Some Linux clients will use any available port number higher than 1024 or some other lower limit; the kernel keeps track of which are in use. (Note: you can use a telnet application to connect to any port: debugging.) Such ports are known as *ephemeral ports*.

A ***socket*** is the combination of an IP address and a port number. A pair of sockets will uniquely identify a network connection from a client application on one host to a server on another host.

Many servers are not started at boot time (ftp) although some are (httpd). (Q: Why?). Instead a "super-server" known as ***inetd*** or ***xinetd*** (or systemd on modern Linux systems) is started at boot time that listens for incoming packets with a variety of port numbers. Inetd (or whatever) then checks its configuration file to determine which service daemon should get that packet, starts the server, and hands off the packet to it. Such network servers are often referred to as network ***daemons***. Most ***spawn*** child processes for each incoming request. This important service is configured either by editing a file `/etc/inetd.conf`, editing files in a directory `/etc/xinetd.d`, or enabling and then starting a systemd socket.

> Analogy: Suppose you live in a fancy apartment (or condo) building with a doorman, and suppose you are expecting a package to be delivered. You can either wait by the door for the package yourself, or ask the doorman to accept the package when it arrives and deliver it to you. You then take a nap while waiting.
>
> Having many people wait for packages at once is inefficient; it works best to have a doorman wait for any package, and alert the correct person when a package arrives for that person. On the other hand, there is an extra delay for having a doorman wait while you wake up from your nap.
>
> (x)inetd/systemd is like the doorman; it can be told to expect packets for some daemon, and wake up that daemon when it arrives. But if that extra delay is not acceptable (some daemons take a long time to initialize), the daemon itself can be listening for arriving packets (stand-alone).

To see what is listening on a given port, use (as root) `fuser [-v] port/proto` (for example: `fuser ssh/tcp` or `fuser 22/tcp`). `lsof port/proto` works too. For all listening ports, use (as root) `lsof -i -sTCP:LISTEN` or `netstat -tl`.

Network sockets have proven so useful and easy to work with, that many types now exist. Besides the ones for TCP, UDP, and IP ("raw"), there are sockets that support other network protocols, sockets for kernel-to-process communications (*netlink* sockets, used for example by udev), and process-to-

process communications (*unix* sockets, similar to named pipes). Sockets have been references throughout this course; now you know what they are.

TCP/IP Transports: Connection and Connectionless

The IP addresses are sufficient to route a packet from one computer to the destination. However, two issues remain: How to identify the source process (client) and destination process (server)? The answer to this is to include another header (the ***transport layer header***) that includes the source and destination port numbers as described above.

The other issue is dealing with errors that can occur. One common approach is to have the sending process expect an acknowledgment from the receiving process. If no such reply is found after a *time-out* period then the sender can resend the packet. Implementing this correctly for every client and server gets old fast!

Another solution is to have the network itself guarantee delivery of the packets. In this scheme, the two hosts set up a session, send the data, and tear down the session when done. The TCP/IP system handles the time-outs and other issues.

The first scheme is known as ***datagram*** or ***connectionless*** service and is called the ***UDP*** in TCP/IP protocol suite. The second scheme is known as a *virtual circuit*, or more commonly a ***connection-oriented*** service and is called ***TCP***.

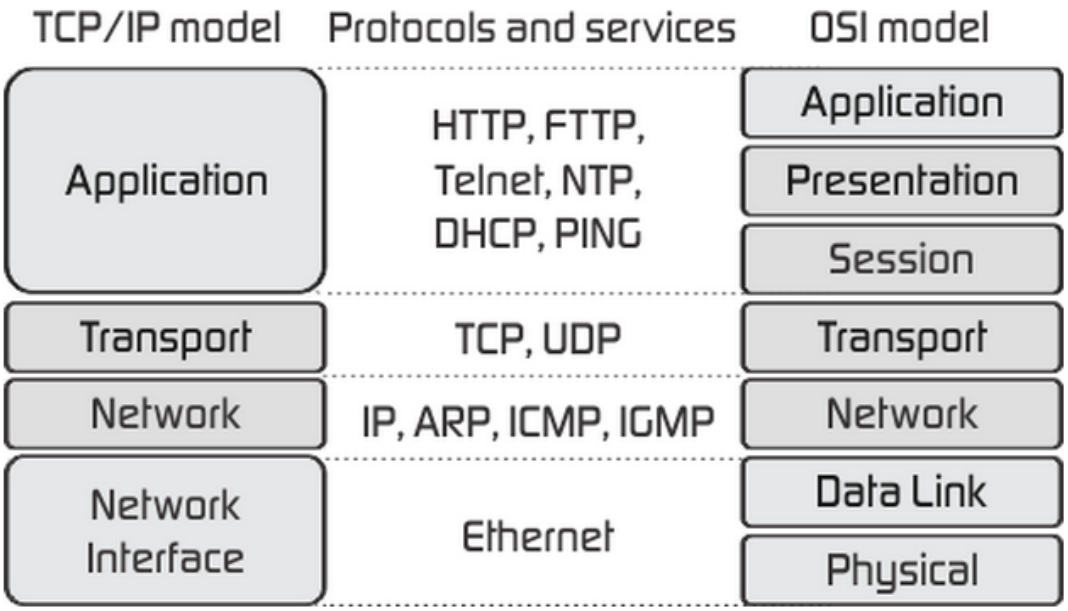
RPC

Sun developed a different scheme for connecting a server to a port number. Instead of using a well-known port number for each service, a single well-known port number (111) is used for the program **`portmapper`** (or **`portmap`**). This program assigns each **RPC** service a unique port number at each boot. A client wanting to use some RPC service sends a query to the portmapper, requesting the port number for that service. This scheme is full of security holes, and should be turned off on your server unless you are using RPC services. These include `NFS` (<v4) and `rlogin`.

> Another scheme is to use DNS "SRV" records to state an IP address and port number to use for each supported service.

The OSI Networking Model

Due to the complexity of networking, the various functions and terms have been standardized by the ISO. Known as the ***open systems interconnect (OSI)*** this model of networking is standard knowledge for all IT workers. Here's a picture showing how TCP/IP networking compares (from fiberbit.com.tw):



Configuring Basic Networking

The NIC must be configured at boot time with many *parameters*, such as an IP address and mask. In addition, your host will need a *default gateway* address to configure its routing table. To use DNS, your computer must be assigned a hostname, a default domain name, and must be configured with the IP address of a DNS server to use to translate names to IP addresses.

> An ISP's DNS is not always the best choice to use. Often these will never give you an error, but instead redirect a bad URL to a page of ads. They will sometimes also track a user's lookups for marketing purposes. You can use alternative public DNS servers, such as 4.2.2.1 or 8.8.8.8, or OpenDNS.
>
> (If you have Verizon as your residential ISP, you can currently (2015) opt-out of this, what Verizon calls "DNS Assistance".

The easiest way to configure TCP/IP networking is to let someone else do it. One way to achieve this is to configure your system to use **DHCP** (*dynamic host configuration protocol*) for each NIC. When the system brings up the NIC (usually at boot time), it will send a broadcast DHCP request packet. If there is a DHCP server listening on that LAN, it responds with all the required networking parameters. Your system uses that data to configure networking.

The other way to configure networking parameters is by manually editing various configuration files (or using some tool to edit those files). This is called *static* addressing.

For wireless laptops, Fedora Linux systems come with a newer networking system called `NetworkManager`. This software was poorly documented

and didn't work well for static, wired networking, but is much better now. You can use `c systemctl` to turn this daemon off and make sure it stays off, and then use those tools to turn on the older (legacy) `network` service (you may have to install that).

Red Hat has since modified NetworkManager to use the standard (for Red hat) config files; see `/etc/NetworkManager/NetworkManager.conf`. (Debian has done something similar.) **Thus, there is no real need to switch network services.** Even the GUI config utility, `nm-tool`, will use the config standard files. The command line tool is `nmcli`. but there is an easier to use ncurses-based tool `nmtui` you can install.

The configuration of NICs on Red Hat systems is controlled by the file **`/etc/sysconfig/network-scripts/ifcfg-`*`NameOfNIC`***. In many cases, *NameOfNIC* is `eth0`. On some systems, NICs are named differently (e.g., "`p7p1`").

By convention, the `ifcfg*` file's suffix is the same as the string given by the `DEVICE` directive in the configuration file itself. (Some versions of Fedora at least depend on that.) System-wide settings go in `/etc/sysconfig/network`. Note that a setting in the ifcfg file will override the same system-wide setting, for that interface.

> As with all RH config files under `/etc/sysconfig`, you can find documentation for each file in `/usr/share/doc/initscripts/sysconfig.txt`.

In the directions that follow, be sure to change `eth0` to your NIC's actual name. (You can use `dmesg` to see what name the kernel gave your NIC, or the "`ip link`" command.) To configure the system for DHCP, this file should look something like this (bold lines are the ones you might need to change):

```
# Intel Corporation 82557/8/9 [Ethernet Pro 100]
TYPE=Ethernet
NAME=enp0s3
UUID=5988518b-ab91-3a9f-a874-9d5c5d25c862
DEVICE=enp0s3
HWADDR=00:06:5B:3D:43:0F
...
BOOTPROTO=dhcp
ONBOOT=yes
```

This will cause the network system to configure everything using DHCP. To enable a normal user to set the interface up or down, add "`USERCTL=yes`".

If using DHCP, you can add additional entries to control the configuration: Add "**PEERDNS=no**" to prevent DHCP from updating `/etc/resolv.conf`.

For a static setup, this file should look like this (only the bold lines should be edited):

```
# Intel Corporation 82557/8/9 [Ethernet Pro 100]
TYPE=Ethernet
NAME=enp0s3
UUID=5988518b-ab91-3a9f-a874-9d5c5d25c862
DEVICE=enp0s3
HWADDR=00:06:5B:3D:43:0F
...
ONBOOT=yes
BOOTPROTO=none
IPADDR=192.168.0.7
PREFIX=24
GATEWAY=192.168.0.4
```

(The IPADDR and PREFIX can be followed by a number, as modern NICS and Linux support multiple addresses/prefixes per NIC.) That will configure the IP address and mask, and the default route. But not DNS. To configure the DNS system when not using DHCP, you can edit the file **`/etc/resolv.conf`**, which should look something like the following:

```
search hccfl.edu
nameserver 169.139.222.4
nameserver 169.139.222.15
```

NetworkManager added new entries to the config file to also configure DNS and other aspects of networking, see below. But I still just edit `/etc/resolv.conf` for that.

> You must first **turn off DHCP or changes to that file will be lost when DHCP does turn off.** It may also pay to make a copy of that file first, so you can see the IP addresses of the nameservers from the copy.

If using `PEERDNS=no` (or if not using DHCP), you can instead add entries such as "`DNS[1|2|3]=`*`ip-address`*" and "`SEARCH="gcaw.org hccfl.edu"`" to update `resolv.conf` with that information. Thus you don't need to edit resolv.conf.

Finally, you may need to set the hostname. The default of "`localhost.localdomain`" is fine for most purposes. If you do need to set a different hostname, use the **`hostname`** or **`hostnamectl`** command. (There is no standard file to edit on a modern Linux system to set

this, although many systems will pay attention to **`/etc/hostname`**. You should also add an entry to **`/etc/hosts`** with your static IP address and hostname.)

Configuring Service Daemons Review

Stand-alone services are simple: you start some systemd service unit or run some Sys-V init script. To have the service start at boot time, you enable it. With systemd, on-demand services create a new service unit file for each incoming request, stored on a RAM disk. These unit files are created from a template service unit, "*nameOfService@.service*". (The "@" identifies this as a template unit file.)

While all systemd managed daemons have sockets, for stand-alone services you can generally ignore them. However, for on-demand services you need to start the socket (since there is no service unit yet to start and starting the template does nothing). To enable on-demand services at boot time, you enable the socket.

(Do not confuse systemd socket unit files with the general networking concept of a socket.)

Network Security

At the host level, you have network security in several sub-systems: a **packet filtering firewall** (**`iptables`** or **`filewalld`** on Linux, and similar ones for other Unixes) is the first line of defense. This can be used to allow or deny incoming or outgoing packets, and to collect various statistics. **TCP Wrappers** can be used to examine incoming service requests and either allow or deny them (and log them); this is rarely used anymore. Note the packet filter can also be configured for this; however, TCP Wrappers can allow or deny access on information not in the packet (time of day, availability of some resource, the username making the request, etc.) Note that most computers have or can have firewalls enabled. That's a good thing to do! Also note that other network devices can act like firewalls, including routers, L3 switches, load balancers, and proxy servers.

The various services that listen for incoming requests can generally also be configured for security. Additionally, any network services that authenticate users will likely use **PAM** or other security systems; you must remember to configure those too.

Incoming data can such as FTP and WebDAV uploads, email, etc., can be scanned for viruses and other malware. A malware scanner such as clamAV is used for this.

You can run network monitoring tools that look for known attacks or any suspicious activity, and block access to attacking hosts (as well as log and alert the SA).

The last part of network security we will discuss is encryption and PKI. In the modern Internet, most connections are encrypted to prevent modification and eavesdropping. The "S" in "HTTPS" is for security, and likewise for POPS, IMAPS, and so on. In brief, a server obtains a *certificate* that is granted by a trusted certificate authority only after the owner of the server has proven their ownership. Clients can retrieve these certificates to check that the server in question is the one it claims to be. Then the client and server exchange some data that will secretly share a very long random number, known as a session key. They will encrypt all data exchanges using that key. When the session is done, the key is forgotten.

> It used to be possible to obtain PKI certificates with 10 year or longer valid lifetimes. This can cause security issues so the max valid age of server certificates has been getting shorter over the years. The new max will be lowered to just 47 days by 2028! Fortunately, obtaining fresh certificates can be automated so the system admin only needs to set this up once.

Perfect security is an illusion! Even with all this security, attackers can get in. By setting up file permissions carefully and using service isolation techniques, you can limit or prevent a successful penetration from doing (much) harm to your server.

Network Trouble-Shooting Tools and Techniques

Common tools are **`ping`**, **`traceroute`**, **`mtr`** (a modern Linux replacement for those two tools), **`top`**, `ipcalc` and **`ethtool`** or `mii-tool` (Linux), `host`, `dig`, `ip` (Linux; `ifconfig` for Unix), and various log files.

Before using any tools, verify the problem exists and is network related (and not a turned off server).

Test with `ping` (or `mtr`) first, which tests layers <3 (i.e., basic IP connectivity). ***Then if that works,*** try **`nc`** (netcat) or `telnet` next (to say port 80 on a web server, or port 25 on a mail server). If that fails, the problem is usually a firewall between client and server blocking access.

If `ping` fails, try **`traceroute`** to localize the fault. The problem is almost always a bad cable, connector, or NIC, or some joker unplugged something.

If the problem appears to be with a host, use the command **`ifconfig`** to examine the NIC parameters and **`route`** to examine the routing table.

Also, examine the resolver files to check for DNS errors. Use `nslookup` and/or `dig` to check for DNS server errors (discussed in more detail below).

To examine the firewall setup (including *masquerading*) you use **`iptables -L [-v];`** `iptables -t nat -L [-v]` to list all rules (if you use iptables and not firewalld). For firewalld, use **`firewalld-cmd --list-all`**.)

Often the best way to troubleshoot networking issues is to examine the IP packets. This is a useful technique for security monitoring as well. Various tools exist that can show and store in a file network packets. You can show all or, using a filter, only those packets of interest.

The tool **`wireshark`** is the most common and powerful tool for this. Out of the box `wireshark` knows hundreds of protocols and can dissect them for you. It has an easy GUI interface for basic tasks, although such a powerful tool can be challenging to master.

Sometime you need a command line tool in order to use it from a timer service (capture packets at a certain time of day) or for use in a shell script. **`tcpdump`** is a command line tool for this that is very mature. But if you don't want to learn it, you can use **`tshark`**, the command line version of `wireshark`.

Lecture 12 — System and Service Monitoring

If it isn't measured and monitored, you aren't managing it. A major part of the daily tasks of a Sys Admin is system and network monitoring. What are the goals for monitoring? They include: Early warning of problems that could cause **service outages** such as a filesystem filling up, or the load average getting too high, or an important daemon stopped accepting connections; Requests or completion rates drop significantly; trouble-shooting incidents. The results of monitoring are used to make decisions: e.g., CPU utilization is too high, so what to do? Adjusting the system in response to this data is called ***system tuning*** (networks too).

> **The purpose of monitoring is to provide useful and actionable information to a system administrator, in a timely fashion.**

Monitoring is traditionally performed by a framework that allows plug-ins, which are checks (often scripts) that test or measure something. The framework stores the results of such checks, called *events*, in a database. The framework also has rules configured by the sys admin for what events are worth alerting, who to alert, and how. Usually, the framework will have a console where you can enter in database queries, and a dashboard showing the status of your systems at a glance.

> Modern system admin is outsourced to the cloud, so skill with monitoring and management of the services is more important than skill in low level administration tasks.

Traditional monitoring focused on the health of systems and services. "Were they up?" was the question. Such monitoring didn't need much attention from system admins, as the number of services and servers rarely changed.

In contrast, modern systems, especially cloud-based ones, change all the time. Containers and/or virtual machines come and go so often, you use a special management service to keep track of them. In a larger cluster, instances may come and go within a few seconds. Also, the size of the cluster needs to be adjusted depending on current load and other factors. Because of the difference, modern monitoring focuses on collections of metrics and overall cluster health to determine current performance. The health of a service is derived from such data, rather than measured directly.

There are two basic types of traditional monitoring: *historical* and *service availability*:

- ***Service availability*** **(or *real-time* monitoring) show events of interest as they occur**, and is used to spot trouble (outages, resource exhaustion such as slow system or network). It must be tied into an ***alerting system*** to be useful, as the SAs must sleep sometimes and cannot spend all their time monitoring. A sophisticated alerting system is worth the investment for

medium to large organizations. Even without an alerting system **service availability monitoring is often turned on (or increased if already on) during trouble-shooting and service and system rollouts and cut-overs**.

- ***Historical monitoring*** **captures events that occur over a period of time**, in order to determine trends. This is used for ***baselining*** (Q: what is this?) a system to determine what is normal, to spot trends before you run out of resources, and for reporting (to management).

Historical monitoring consumes a lot of disk space. Service availability monitoring can consume a lot of network capacity. The benefits of monitoring must be weighed against this cost to determine exactly what should be monitored, to what level of detail, when, and how often. (For example, detailed performance data might be collected on an application, in order to determine how much time and/or memory is used per function. That data can help developers improve services and localize bugs.)

Service availability, performance data, and alerts are shown on dashboards, then forgotten. The same data can be saved instead, resulting in historical monitoring. Some software does exactly this, providing an *all-in-one* monitoring (and alerting) solution. However, you may wish to collect different data for each type of monitoring situation (e.g., performance data for some applications), in which case you can use two systems.

> The traditional service availability monitoring discussed above performs service and server checks at regular intervals. Each check is an *event* that is stored in a database, and also examined to see if it should trigger an alert (and/or if it should be sent to a dashboard display).
>
> Another type of monitoring collects and shows numbers, called *metrics*. These show performance data and can show trends over time. This is discussed below.
>
> Note that event monitoring is related to, but not the same as, monitoring log files (which are text, not numbers). Often a separate monitoring system is used for logs.

Any organization will have several servers and services. Medium and larger organizations may have clusters of hundreds or even thousands of servers. Even a small organization many have dozens. A system administrator must make sure these services are working correctly, by monitoring them.

> *Failure isn't an option. It's mandatory. — author unknown*

What is monitored and how depends on many factors. For example, consider a single web server; what should be monitored? For starters, make sure the server is "up"; loading a non-cached web page from that server periodically is a good way

to determine the service is available (and you can also see if the response time is acceptable). (See the example script below.) For a non-interactive, non-busy site such as wpollock.com, once an hour, or even once per day is sufficient. For a busier site, once every 5 minutes (or even more often) is appropriate.

But interactive sites must be more than "up". They need to be responsive, have working logins, working DB access, etc. For such sites, you would want to collect more data: number of connections per unit time (second, minute, or hour), and average delay (latency) of responses. Such data can tell you if the service is working well or not. But if not, this data won't provide many clues as to why a service is performing badly, nor will they provide early warning of impending problems. For that, you need to collect additional data: CPU and memory utilization, disk I/O rates and latency, network throughput and latency, and data about dependent services such as remote databases, DNS delays, etc. Finally, you would like to monitor the server error logs: the frequency of (some types of) errors can indicate a DoS attack or other problems.

It should be obvious that the data needed by a DNS server, or a DBMS server, is very different.

> The inputs to a monitoring system often come from scripts (also called "checks", that calculate performance of some service, "grep"-ing log files for alertable events, checking network service availability, and checking that a daemon process hasn't died directly.

You can write small scripts that get run from cron to check things, or use fancier tools. For example, to check process status of `httpd` the exit status of "`pgrep httpd >/dev/null`" will tell you if the web server processes are running (you could also count them with `wc -l` instead.) The result of this check can be sent to your monitoring system (such as Nagios, although that already has many plugins for this).

Example: Monitoring a Web Service (see Timing Details with Curl)

```
$ cat > curl.fmt <<\EOF
time_namelookup: %{time_namelookup}\n
time_connect: %{time_connect}\n
time_appconnect: %{time_appconnect}\n
time_pretransfer: %{time_pretransfer}\n
time_redirect: %{time_redirect}\n
time_starttransfer: %{time_starttransfer}\n
----------\n
http_response: %{http_code}\n
time_total: %{time_total}\n
EOF

$ cat >check-urls <<\EOF
```

```
while (($#)); do
   echo $1 @ $(date +%s):
   curl -s -w "@curl.fmt" -o /dev/null \
   -H "Pragma: no-cache" \
   -H "Cache-Control: no-cache" "$1"
   shift
done
EOF
$ check-urls http://wpollock.com/Hours.htm
http://wpollock.com/Hours.htm @ 1501040708:
time_namelookup: 5.515
time_connect: 5.565
time_appconnect: 0.000
time_pretransfer: 5.565
time_redirect: 0.000
time_starttransfer: 5.615
----------
http_response: 200
time_total: 5.615
```

The final line is the total round-trip response time in seconds. If it is easier, you can change the format to just that one number and the response code; see the (large) `curl(1)` man page for all the variables you can output. The extra details in the shown format can help to determine the *root causes* of any delays.

With such output, you can check the server was working (response codes of 2xx), and if the response time meets your SLA/SLO (if time_total < SLA_VALUE). In any case, an event can be sent into the monitoring system, to be stored in a database and so dashboards can be updated. (Note again, Nagios and other systems already support these checks, or you can use a tool such as http_load and send that data into your monitoring system. You will rarely need to write your own checking scripts as shown here.)

> An interesting tool that can be used in monitoring scripts is *osquery*. This tool has an OS-specific component that provides all sorts of information as a set of SQL tables. With this tool, you can use standard SQL queries to determine all sorts of data (open sockets, open ports, running processes, etc.) regardless of the OS type.

There are lists of **items to monitor** all over the Internet. Some examples include disk space, swapping activity (memory performance), CPU utilization, temperature of physical servers, disk I/O throughput, network throughput and latency, and so on. Specific applications may have additional items to monitor, for example query

performance, connections per minute, replica status and lag, and so on. **Always monitor *key performance indicators*** (KPIs; see below).

Some Monitoring Terms and Definitions

Most of the following terms do not have a universally-agreed upon definition. The terms and their definitions vary widely in the industry. Below are some useful definitions of some common terms (*taken from the Google SRE book, p. 55–ff*):

- **Monitoring** — Collections, processing, aggregating, and displaying real-time quantifiable data about systems, such as query counts and types, error counts and types, processing times, and server lifetimes.
- **White-box Monitoring** — Monitoring based on metrics exposed by the internals of the system, including logs, interfaces such as the Java Virtual Machine Profiling Interface, or an HTTP handler that emits internal statics.
- **Black-box Monitoring** — Testing externally visible behavior as a user would see it.
- **Dashboard** — An application (usually web-based) that provides a summary view of a service's core metrics on its main view and can display all metrics if asked. A dashboard may have filters, selectors, and so on, but is prebuilt to expose the metrics most important to its users. A dashboard may also display team information such as ticket queue length, a list of high-priority bugs, the current on-call system administrator, recent software development pushes (uploads of tested software ready for deployment), and current alerts.
- **Alert** — A notification intended to be read by a human and that is pushed to a system such as a bug list or ticketing system (these are called *tickets*), to an email alias (called an *email alert*), or to a pager or cell phone (called *pages*).
- **Push** — Any change to a service's running software or its configuration.
- **Root Cause** — A defect in a system that, if repaired, instills confidence that the event (incident) that exposed the defect won't happen again (at least, not in the same way). A given event might have multiple root causes: for example, perhaps it was caused by a combination of insufficient process automation, insufficient testing, an inadequate procedure, insufficient resources such as CPU, network bandwidth, memory, and so on.
- **Services** — The applications providing utility to its users, and that must be monitored to ensure they are running smoothly. (Other monitored information is collected to help determine root causes of any service outages, and possibly to provide historical monitoring to make capacity planning decisions.)

- **Node, Host, Machine** — used interchangeably to indicate a physical machine, a virtual machine, or a container. Each of these may be running one or multiple services. Such services may be related to each either (for example, a web server and a caching web server (or *reverse proxy*), or unrelated.

Some commonly monitored resources on production servers (some are discussed in detail below, under *system tuning*):

- Your services (Web, SSH, DNS, Print, file sharing ...)
- Free disk space
- The state of your mirrors
- Response time
- Failures of any sort (reliability)
- Memory (used, free, swap space used)
- Login events
- Mail events
- Unusual penetration events
- Network events
- Storage subsystem events
- Current, peak, and average load (and general trends) for anything of interest
- Hardware data (disk speed, failure rate, age; system temperature, power usage.)

Monitor events can look very different depending on the monitoring system (framework) used. For the popular Nagios monitoring system, each event is reported by a plugin that returns two or three items: a number (zero (all okay), one (warning), two (critical error), or four (unknown)), a single line (generally) of text, and optional performance data included. An example event from Nagios might look like this:

DISK OK - free space: / 3326 MB (56%);|/=2643MB;5948;5958;0;5968

The use of the performance data is up to the configuration you use.

Reimann events look very different. They are eight fields: host, service, status, time, description, tags, metric, and a time-out (ttl) for the event.

KPIs, SLIs, and SLOs

Monitor everything deemed important. Qu: What is important? Ans: A resource is important if you'll get into trouble with your PHB (*pointy haired boss*) if it (the resource) runs out. Memory, storage, and bandwidth are commonly monitored.

The most important things to monitor are the ***key performance indicators*** (KPIs). KPIs come from the business goals, not technical ones. Every system must have some KPIs, and these need to be monitored. A simple but common KPI is the

revenue generation rate for an ecommerce system. If that goes up, you will soon run out of resources. If it goes to zero, the service is likely unusable. If it drops, often something is now broken. (You also monitor basic information, so when some KPI indicates a problem, you have the information necessary to determine the ***root cause***.)

SLIs (***service level indicators***) include KPIs and other items you monitor. SLIs are chosen so you can tell if your system as a whole is working as expected. The goals for reliability and responsiveness are called **SLOs** (***service level objectives***). SLOs are jointly set by management and system admins.

SLIs should: relate to the customer experience and other business goals, be something that can be measured and related to SLOs, and provide sufficient information so that action can be taken.

For example, many parts of most systems query some database. What makes a good SLI for that? Not uptime, as replicas can fail without customers noticing. A better SLI metric would be database read query success rate. (Another SLI could be read response time.) What should the SLO be for this? After discussion with management, PR, marketing, and the help desk, you may discover customers notice a problem if fewer than 95% of database reads succeed in a 20-minute interval. To prevent this, you can set an SLO of 97% success over 20-minutes; if the rate drops below that, an alert should be sent to someone. A similar SLO could be determined for database read response times.

> Think about your own experiences with various websites. Did you ever have to click a button, and when nothing happens, click reload? How often must that happen to become annoying?

Finally, make sure you monitor enough details so when some SLO violation alert happens, you have sufficient information to determine where the problem lies.

> Metrics tell you when something isn't right but rarely can tell you why. Once your SLIs have determined which subsystem (e.g., the database) is the culprit, log files of those systems are then consulted to determine the problem details, so you can take action.

Monitoring Policy (including retention of data and personal identifiers)

A ***policy*** for monitoring must be decided. Because it costs time and money to perform monitoring, the SA (and management) must know what is expected. Do you want to log the status of the things you monitor, so you can go back after a crash and do a **postmortem (forensic) analysis** of how things were doing shortly before the crash? Do you want to be able to create **monthly charts and reports** of your servers' overall workload, so you can report to your boss and plan upgrades before the increasing loads cause trouble? How much will the monitoring cost (extra disk space, backups, security, and bandwidth)?

There are additional legal aspects of what to monitor. **Some things must be monitored** in a regulated industry, **others cannot be**. Some things should be monitored historically for audit/security purposes, or to collect trending data, and won't be used to generate alerts. **There are many legal ramifications to monitoring systems.** Only some collections made under specific circumstances are legal. On the other hand, you may have requirements to monitor some parts of your systems. Good books on auditing and incident response discuss these issues. You need to get and read one, such as "Incident Response & Computer Forensics" by Mandia, Prosise, and Pepe.

In all cases, you need a policy that **protects the data collected**, determines the ***data retention period***, states what to do with old data, and defines what personal identifying data is collected in accordance with the organization's data privacy policy. (Often posted on the corporate website in P3P format and monitored by organizations such as TRUSTe.) (**Show wpollock.com Privacy notice**.) It is easier to protect data if kept in a single, central location. This will be discussed later.

Old data that has been ***sanitized*** **by removing or replacing some sensitive information** (including *personal identifying information*, or PII) is often useful for training and reporting purposes.

> Using any auditing or monitoring tools require a carefully written AUP or the collection may be illegal. And guess who will be held responsible? (Hint: not the boss who ordered you to do this.) Consider if the auditing tools you enable capture command line arguments or network data also capture credit-card numbers, heath information, or any personally identifiable data. If so, you may need to sanitize or "blind" the logs.
>
> ***Blinding*** means replacing data with an MD5 checksum of that data, which preserves traceability and comparability. (This is discussed in more detail with logging, below.)
>
> ***Sanitizing*** has several meanings for IT: One is the examination of user-supplied data (say in a web form) and replacing any illegal or dangerous input (such as SQL queries embedded in the data, "`../`" in pathnames, etc.). Another definition is the secure destruction of data, either by securely removing sensitive data from existing records or from input data before storage (so sensitive data is never stored in the first place). Sanitizing also means securely destroying the storage media (e.g., shredding or melting disks).
>
> (Removing sensitive data that was stored, for publication without that data, is called ***redacting***.)

Consider the case of **email monitoring**: Some organizations require a Sys Admin to archive all email from all employees. Such archives (which are generally illegal

to make) should be secure and not readable except by those with the proper authority.

There are many legal and regulatory rules you need to follow to ensure privacy of data. For example, the GDPR in the UK. In the US, there is not a single privacy law, but many for specific types of data: Family Educational Rights and Privacy Act of 1974 (FERPA) for student education records, the Health Insurance Portability and Accountability Act of 1996 (HIPAA) for health-related data, the Children's Online Privacy Protection Act of 1998 (COPPA) for data related to children, and the Fair and Accurate Credit Transactions Act of 2003 (FACTA) for some financial data. The California Online Privacy Protection Act (OPPA) of 2003 and the California Consumer Privacy Act of 2018 (CCPA) which became effective in 2020, affects people and organizations outside of California too. (See also this guide to the CCPA.)

Be careful your network bandwidth isn't used up by your monitoring data. Aim for no more than 1% used, especially over slower links. (Ex: 100 mbps Ethernet provides ~50 mbps, and 1% is 500kbps.) Consider remote monitoring (RMON) that summarizes, or just sends alerts (called "traps") when needed.

Monitoring data is sensitive and needs security. Protect (encrypt) monitoring data. Use SSH for transport. Configure devices to respond only to data requests from specific *management consoles* or monitoring hosts (or a subnet). Change all default passwords! Use an independent host to monitor the monitoring system (called *meta-monitoring*).

Once you have setup a monitoring system for a few services, it becomes easier to monitor additional services with those monitoring tools. **Over time, an SA winds up monitoring more and more. This is usually a good thing** and with proper tools is not as time-consuming as one might think.

To get started it would probably be a good idea to write down everything that the server(s) do. Then for each service, ask *what are the most important items in the server environment for that particular set of tasks*? Then make the most important entry: Identify the meanest and nastiest corporate officer(s) and find out what applications are their hot buttons. (Your supervisor's buttons are important too, even if they're nice folk.)

A good (and short) intro to these topics can be found at Open Telemetry's Observability Primer.

TL;DR version — Monitoring

Metrics are simple measurements of something such as queue lengths, response times, completion rates, bandwidth used, free memory, disk I/O times, etc. A metric is just a number, a label (which says what it is), and a timestamp. Sometimes there is additional information (*tags*) included.

Metrics can tell you about the health of your services by reporting their performance. (Obviously, zero performance means the service is down.) Metrics can come from the OS or from instrumented applications that were written to provide them. Generally metrics are *pushed* to a collector or *pulled* by one that knows how to get them from various sources. From there, metrics are analyzed and stored. The analysis provides alerting and monitoring (using a dashboard) features.

Logs are collected events (or entries), typically a string of text (so log entries are usually bigger than metrics), a time-stamp, and other information (such as the source process and IP address). While metrics can tell you a system crashed, it won't help you determine the cause. Logs come from applications written to provided them (so *instrumented*, although that term is not used for logs often).

Log data can be controlled by the administrator, to reduce the amount of data produced to a reasonable level. You can control the log level (major incidents only through every function entry and exit) and location. Like metrics, log data is collected from many sources, stored in a database and analyzed for alerting and monitoring.

Tracing show details of the exact execution of some application or service. If properly designed and used by developers, an application's logs can provide tracing. Often however they cannot do that and external tracing tools can be used.

There is some overlap between all three. You can derive some metrics from log data and some log data from metrics, and tracing data from log data. But you cannot get all the metrics you need with logs along, nor all the log data you need from metrics alone, and very rarely get all the trace data from logs along. So a system administrator must set up and use all three types of monitoring data.

Open Telemetry is a vendor-neutral open-source observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, and logs. As an industry standard it is natively supported by a number of vendors and quite popular (2023). See OpenTelemetry.io for more details.

Metrics

Performance (and some other) data collected from servers and services are numbers called ***metrics***. These values are used to show trending of the metric over time. With cloud-based services, such trending data is much more valuable than the simple health and status checks (events) of traditional monitoring. Indeed, you can usually determine host and service health just from the performance metrics.

Monitoring can examine these metrics in (near) real-time, showing the status of services one way or another (for example, as scrolling line graphs of response times, or just a green=okay, yellow=warning, red=error indicator).

Visualizations of metrics are combined and displayed in ***dashboards***. Dashboards provide at-a-glance information on your services and server health, by providing both text and graphical ("visualization") views of service availability, health checks, and performance of services.

> Metrics can be used to calculate traditional monitoring events ("is the web server up?"), and those events can be used for traditional monitoring.
>
> To some extent, the reverse is also true. Nagios for example has a plug-in to collect metrics and record them as events in the system, and to visualize them in a dashboard. But that approach doesn't work for larger data centers or large cloud-based services.

Once collected, metrics are then stored in files or databases, for historical monitoring use. The saved metrics can be analyzed to provide trending data. Metrics can also be sent to other tools, such as dashboards and alerters.

> A metric has three components. The exact format depends on the application and protocol you use. Each metric has:
>
> - a value (a number, for example "17"),
> - a timestamp (usually with granularity of seconds or milliseconds, this is often a Unix timestamp (number of seconds since January 1, 1970, GMT) such as "1500619370", which translates to "Fri Jul 21 06:42:44 UTC 2017"), and
> - an indicator of what the value means. That might be a number that is defined in your system to be something such as "CPU load average for the past minute". Graphite uses a string they call a "metric_path", for example "`web.sessions.stats`" (you get to make these up; in general, a useful naming scheme would identify the general type, the host the metric came from, and the specifics, for example "server.example-com.cpu-1-minute").
>
> Other metric formats exist. For example, Prometheus and Influx store a metric name, time-stamp, and one or more name=value pairs. Often these are passed around using the JSON format. There was an effort to define a standard for metrics to allow easy interoperability, known as Metrics 2.0. However that has been superseded by OpenTelemetry.

As mentioned previously, collected metrics consume a lot of storage space. This can be expensive if using traditional RDBMS to hold the metrics. Additionally,

the calculations needed to show the trends of the metrics require many database queries and lots of CPU time.

> This topic is closely related to tracing. The kernel is *instrumented* and provides kprobes, tracepoints, and other data sources. There are tools that can get metrics out of the kernel's data sources, such as perf, ftrace and strace, LTTng, eBPF, Systemtap, and others. Many of these tools include front-ends to allow you to easily make a query. Tools with front-ends include perf, ftrace, bcc (works with eBPF), Systemtap, and others.
>
> The most recent and arguably best data collector is eBPF for Linux. However, the bcc front-end is not very useful. Systemtap since v3.2 (2017) supports eBPF as a "backend" data collector.

Modern monitoring systems designed for metrics and trending use a different sort of database, known as a ***time-series database*** to hold the metrics efficiently. Such databases make it quick to answer queries such as "on service X, how did metric Y change over interval Z"? One popular time-series DB is InfluxDB, although many others are included with specific monitoring tools (such as with Prometheus and with Graphite). One of the first of these, RRDtool, is still popular.

With a time-series database, older data can be *summarized*. For example, connections per second metrics over a month old could be replaced with connections per minute or per hour, resulting in a 60x or 3600x reduction of stored data. (It is more of an art than a science to determine how much raw data to keep, and what, how, and when to summarize it.)

> Monitoring service availability can alert you to (potential) problems, but not tell you the root cause of the problems. Once you determine there is an issue, you then need more data to determine the root causes, for example memory (swapping/paging), network bandwidth storage I/O performance, and CPU utilization, and perhaps other data as well, such as your database's performance. In addition, you often need to examine the affected service's log files.
>
> As a result of finding issues and their root causes, you may then decide to change the monitoring policy, collecting and monitoring more/different data, or change the frequency of collection.
>
> **Metrics have many sources: hardware, the kernel, and processes.** The hardware and kernel metrics are exposed (made available) in several ways, including various system calls, netlink sockets, `/proc`, `/sys`, and others. These metrics can sometimes be viewed directly (`/proc` on Linux, but on Solaris `/proc` is binary data), accessed directly by various tools with (hopefully) simple user interfaces, or through some framework.

Tools are used to collect system metrics, such as `collectd`. Such tools require little (but some) configuration, so they know what to collect, how often to collect it, and how to export the data to your monitoring framework. Other utilities can collect metrics from web servers and other daemons. Metrics are often generated by the daemons themselves, but that requires the developers to include that functionality. Such daemons are sometimes called ***instrumented***.

All the metrics from all sources then get fed into your monitoring tool to be stored, visualized in a dashboard, analyzed by some alerting system, and possibly used in other ways (e.g., triggering a firewall rule or a cluster expansion).

Metrics can also come from any program including shell scripts. For example, you can run a shell script with cron and have it report some metric each minute, say by analyzing some log file, and feed it to the Graphite (dashboard) system:

```
TS=$(date ...) # timestamp of log entries
cat log_file |awk 'extract entries for TS' \
| ...
METRIC=... # compute your metric
# Send metric to Graphite Carbon:
echo "my-metric-path $METRIC $(date +%s)" \
| nc graphite.example.com 2003
```

Measuring Performance Metrics with Percentiles

A common item to monitor is performance. However, that means different things depending on the application: throughput for batch jobs, response time for web user interfaces, queries per second for a database, etc. Let's talk about response times to make the discussion concrete, although the concepts are the same for other performance metrics.

Every measurement will be slightly different, even if the same request is done over and over. Thus, it is meaningless to say *the response time is X*. Instead, think about a plot or graph of the last 100 measurements. The result is a *distribution* of response times, some low, some high. The goal is to characterize the distribution in one or just a few values that can be graphed over time.

It is common to talk about the average response time, usually meaning the arithmetic mean (the sum of all values in some interval divided by the number of values). However, that is not a good measure. it won't tell you how many users had that response time or better.

Instead, use ***percentiles***. Rather than average the measurements in some interval, you sort them instead. The middle measurement, or 50th percentile, means that half the users had that level of performance or better. Percentiles are often denoted

as "P*number*", for example "P50" for the 50th percentile. P90, P95, P99, and P999 (for the 99.9th percentile, assuming you have 1,000 measurements in each interval), P9999, and so on, are commonly measured. Typical intervals are seconds to minutes (depending on how busy your service is).

For example, Amazon reportedly (in 2017) specifies an SLA with a response time for P999. This is because the slowest responses are seen by the customers with the most data, and those are the most valuable. Amazon also felt that trying to optimize for P9999 would be too expensive, as much of the latency is due to outside factors.

An SLA might be "the service must have P50 of less than 200 ms and a P99 of less than 1 s, 99.9% of the time". Thus, you need to monitor P50 and P99 and alert on violations (and add a log record too). You could calculate these values for (say) the previous 10-minute interval, every minute, and plot the two values on a moving line graph.

> In many cases, it is too time-consuming to collect all the measurements, sort them, and calculate the values. Instead there are well-known algorithms to approximate percentiles cheaply. Linux uses those to calculate CPU load averages.

Picking the Best Visualization — Often, the obvious display of metrics won't tell you much. There are two ways to determine the correct technique to use for a given metric. You can learn all about data visualization and statistical methods. Or you can determine this the way all sys admins do: trial and error.

The key point here is to try several ways to get useful graphs. Do not give up after the first pretty graph you produce, but instead try several to find the most useful.

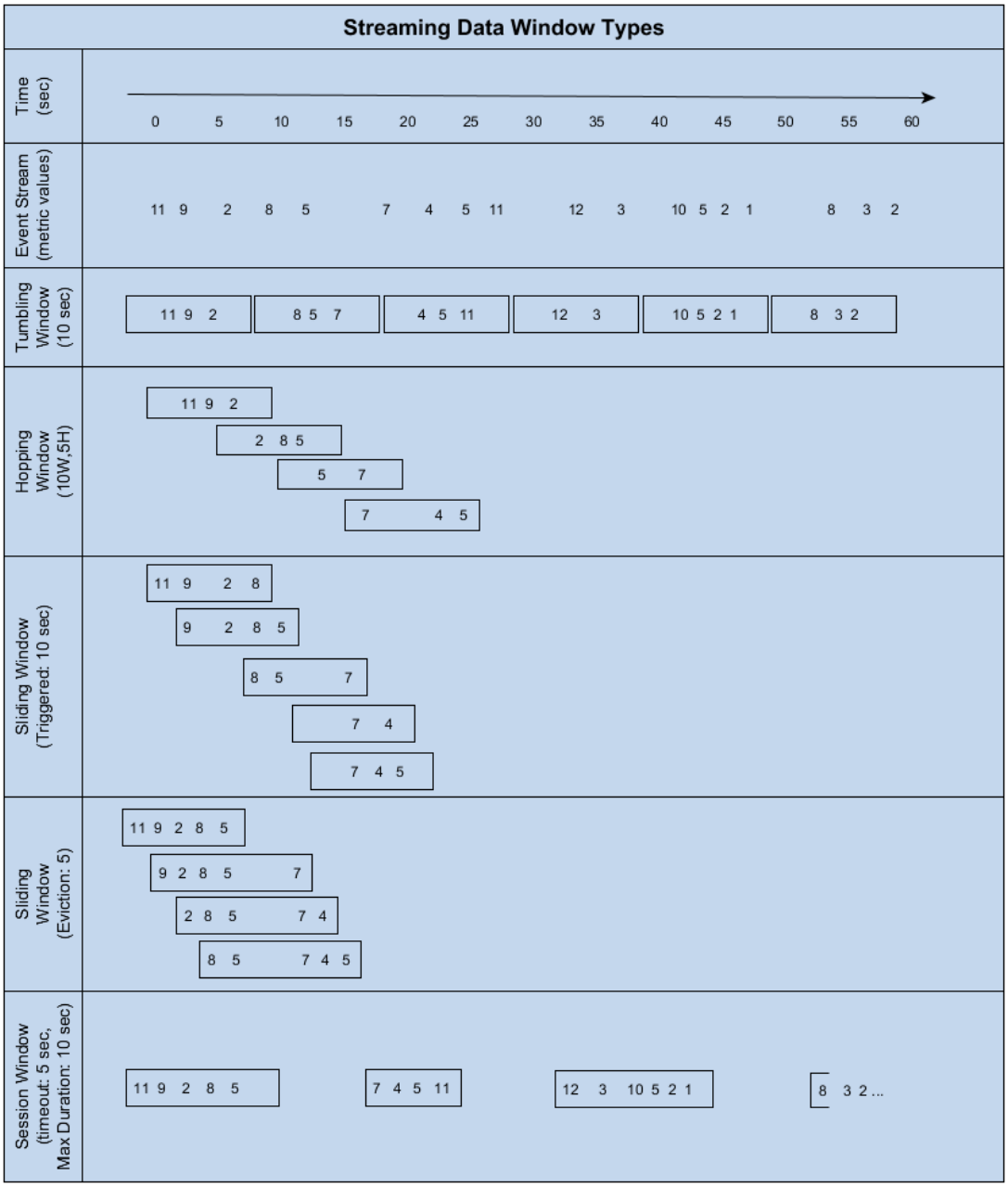
> An excellent book for this topic is Systems Performance by Brendan Gregg.

***Windows* —** When calculating percentiles, averages, or other statistics, as mentioned above you collect the measurements within some interval of time known as the *window*. This is actually a trickier concept than it seems. Generally, the intervals are based on the timestamps in the metrics, not the system clock. (This is because some remote metrics might be delayed due to network or other issues.) After waiting for all that interval's metrics, you then do the calculation.

Metrics that arrive after you stopped looking for them are known as ***stragglers*** and are mostly ignored. Usually you count the number of stragglers per interval (for example, 10 minutes for per-minute metrics) and report that as a metric; if the number of stragglers increases, you have a problem (besides the fact that the visualizations of your metrics will be wrong).

To measure some statistic for a 5-minute interval (for instance), there are many schemes possible to select which metrics to use for the calculation of the statistic:

- **Tumbling Window:** Collect all metrics with timestamps between 1:05:00 and 1:09:59, then calculate the statistic for that interval. The next window is 1:10:00 to 1:14:59, and so on. Each metric is used in one window, with no overlap.
- **Hopping Window:** Collect all metrics received within the past 5 minutes, every minute. (So windows will overlap, with some metrics used in several windows.)
- **Sliding Window (Triggered):** A window of fixed size is used. The current window is determined whenever a new metric is collected.
- **Sliding Window (Eviction):** A window of varying size is used, with a fixed number of metrics. For example, keep 5 metrics in the window and when a new one arrives, evict the oldest one.
- **Session Window:** Form a window for each "bunch" of metrics that arrive close in time. You need to specify a time-out (to close the curtrent window) and a max duration (max size of a window even if events keep arriving).



Metric Sampling Showing Different Windowing Strategies

Tracing

Tracing show low-level details of the exact execution of some application or service. If properly designed and used by developers, an application's logs can provide tracing. Often however they are not implemented natively in the software and external tracing tools are used. (These external tools do not work as well as properly instrumented software with appropriate log statements included.)

In monolithic software, trace data is usually turned off in logs. Enabling tracing in logs, when available, produces so much data that it can take a long time to read through it all to find events of interest. A well-designed logging system will enable trace logging in just some parts of the software.

In distributed software, there are often many small applications running on multiple hosts or containers. The individual software components communicate over a network. Each part is known as a *microservice*.

While you can often enable tracing on each microservice the same way as for monolithic software, a more useful tracing technique is used. Such software works by accepting requests (from other software) and passing the request along to each microservice until the request is handled. When the system produces an error, it is difficult to know which part is to blame.

Request tracing (also called *distributed* tracing) assigns a unique ID to each incoming request. All logging statements in the code will include this ID in their output. By using tracing tools (or even a simple grep) you can follow a faulty request result through the whole system and from that, figure out where the processing went wrong.

Tools for Monitoring: Alerting Tools

Log file monitoring tools are (usually simple) types of **alerting tools**. These should include ways of alerting SAs without using the monitored resource (e.g., a "network is down" email is useless). These tools tell the sys admin what is happening in near-real-time. Other tools are better for finding long-term trends, or for reporting.

> Another type of alerting tool monitors files and directories, then sends an alert if a watched file/directory has changed. The kernel must support this monitoring; Linux does with "inotify". You can use this without C programming by installing the `inotify-tools` package, which provides two useful tools: `inotifywait` that watches a specified file or directory and reports any changes, and `inotifywatch` that can watch a specified set of files for a given duration and report statistics. (See the man pages for examples.)

An important feature for such tools is ***automatic escalation*** (so if the SA first notified is on vacation, the alert goes to some manager.) A good alerting system can also notify groups of people using a variety of means, open trouble-tickets automatically for some alerts, summarize a burst of alerts with a single alert, notify the help desk of the potential problem, and contain other useful features.

Trending, Reporting, and Visualization Tools

A number of free monitoring tools that integrate many monitoring tasks for many hosts into a single web page are listed at linuxlinks.com, such as OpenNMS, Monit, and Zabbix. Fedora 16 shipped with a set of tools called Matahari. Fedora 21 shiped with an easy to use one called Monitorix. There is no shortage of "recommended" free monitoring tools. Some good ones are nmon (may be in your system's repositories), a command line tool (show on YborStudent) and `htop`.

The most popular traditional monitoring and alerting system for enterprises is Nagios. It has a web-based front end, making it easy to use. It can monitor nearly anything you can think of, via plug-ins. While Nagios can be used to monitor large clusters or hosts, it could consume too many resources when you scale it up. Also, Nagios has a steep learning curve, compared with Zabbix, Graphite, or some commercial systems.

Nagios Demo

```
# dnf install nagios nagios-plugins-all
# setenforce 0
# systemctl start nagios
# systemctl restart httpd
# firefox http://localhost/nagios/
  (user: nagiosadmin, password: nagiosadmin)
  (Try all menu items)
# cd; grep nagios /var/log/audit/audit.log \
  |audit2allow -M MyNagios # view MyNagios.te
# semodule -i MyNagios.pp
# setenforce 1
# systemctl restart nagios
# systemctl restart httpd
# firefox # nagios should be working
```

Show YborStudent Nagios demo. (Username: `nagiosadmin`)

Icinga is a Nagios fork that has become popular, possibly created due to friction between internal groups at Nagios.

If running containers (instead of virtual machines), Google's cAdvisor is highly useful. (Red Hat includes software for this, Cockpit, but currently (2016) it doesn't seem as useful.) If running virtual machines, VMware and its open source competitors include monitoring as a feature.

The Reimann Event Processor is an extendable monitoring solution that works on a different concept from Nagios. Its architecture doesn't periodically check things (polling or pull model). Instead, Reimann processes arbitrary event streams that are *pushed* to it from applications and external event collector daemons.

Reimann events contain eight attributes: data (a metric), hostname, service, state, and others. Events can be filtered by their attributes, summarized (combine several events into one new event), and be transformed into other events, such as taking raw metrics and converting into percentile or rolling average events, or a new event type such as *metric out of normal range*.

Monitoring up new machines is easy: Just have the new host start sending events (push model) to your Riemann server. As with all monitoring data, events can come from scripts, system metrics and health checks, or applications. For example, you can embed a Riemann client into a web service and "magically" start getting application level metrics. Reimann comes with some client agents (daemons running on each monitored host), but you can also use something like collectd to gather a machine's health metrics, use some simple tools or scripts to convert to Reimann event format, and send to your Riemann server using nc or other simple, scriptable tools.

Reimann is built for scale and can handle thousands of events per second. The cool part is all the filtering and transformation you can do on the stream of events. While it does support alerting and includes a pretty nice dashboard, Reimann can simply transform events into a format for a different monitoring system such as Nagios or Graphite, so you can take advantage of their strengths.

One big downside to Reimann is all the event rules need to be written in a special language. That makes Reimann harder to learn than Nagios.

One design is to send all metrics, exceptions (error reports), and even log messages to Reimann, which then in turn can send them to some metric data store (for long term storage and analysis) and to visualization and other systems (alerting, security monitoring, etc.)

While Nagios excels at monitoring hosts and services (and providing current status and escalating alerts), it isn't as good as some other tools for collecting and reporting historical (trending and baseline) data, nor as good as some others at graphing data, or dealing with performance metrics. (It can all that, just not as well as some other software.) In addition, it can't (easily) take corrective action when a problem is detected. (A tool such a Monit can do that.)

A common solution is to use Ganglia, Munin, Graphite, or some other tool to collect and digest (summarize) metrics and log data from hundreds/thousands of hosts, and store that data using RDDtool or some other time-series database, and also feed that summary data (or the raw data, if desired) into Nagios/Icinga for alerting and monitoring (although you could use any of these tools alone).

Graphite in particular does a good job with reporting (producing nice graphs). A good solution might be to use Icinga or Nagios, and Graphite. Icinga is for right now and alerting, while Graphite is for trends and reporting.

> Nagios is the "tried, tested and true" solution for traditional service and system monitoring and alerting. It has been around forever, is extremely modular and adaptable. Icinga is along the same lines as Nagios, just more modern (plus a more FOSS-oriented business practice).
>
> However, cloud-based systems that allow servers to be deployed and destroyed depending on system load, and container-based applications

that work the same way, require more than service availability monitoring from a relatively static bunch of servers. Service availability is available as a side-effect of service performance monitoring, and that is the modern trend.

Today, there are endless possibilities for building monitoring systems, and each requires some learning:

- Graphite (creates great graphs and okay dashboards from collected metric data; it is the modern replacement for Cacti) is a whole solution, although its components can be (and often are) used with other tools;
- Grafana creates great dashboards (many good dashboards can be imported from grafana.com/grafana/dashboards such as the Apache dashboard – show import) from Graphite Whisper or other time-series database queries, and is the current (2020) tool of choice (Grafana can be used with Elastic stack log monitoring system and some IDS, allowing a single dashboard for all uses);
- collectd, telegraf (newer with more features such as tag support), and similar tools collect host-level metrics from the kernel and other sources, but don't store or visualize them. You run them per (virtual) host and send the data elsewhere;
- Graphite's Whisper and Influxdb (newer and perhaps better, but only the commercial version scales up) are two of many time-series databases, many of which are not independent components (such as the one included with Prometheus);
- Graphite's Carbon collects performance metrics from other daemons, as do tools such as statsd (a daemon to collect data from applications and turn it into (aggregate) metrics) and jmxtrans or Jolokia (tools specifically for monitoring and metric collection from Java application servers).
- Open Telemetry provides standards and frameworks for metrics, monitoring and logging, including what items to observe. In includes a framework developers can use to instrument their server code.

For less pieces in a more traditional monitoring setup, you can use the plugin pnp4nagios, which adds the historical/trend graphing abilities of Cacti, as a Nagios plugin.

As of 2017, a new entry to the FOSS metric and monitoring arena has quickly become very popular. Prometheus uses a time-series database, has plug-ins for collecting metrics, includes alerting, and uses Grafana for dashboards and for a console. It is very fast and efficient, and includes its own query language, promql.

"Prometheus was launched as an internal tool by SoundCloud, the program and the additional components attached to it are now popular, but

power users complain: In many respects Prometheus is missing functions, and design decisions were made that are not a good match for many setups.

An example is Prometheus Node Exporter, which is designed to collect metrics from the systems in the environment, often not in a way that the administrator desires. Moreover, with a Prometheus server, you cannot store metrics redundantly and in a distributed storage system.

If your setup becomes too large for a single Prometheus instance, you have to split it, thus possibly canceling out one of the biggest advantages of a monitoring, alerting, and trending (MAT) system – namely, the single point of administration.

Additionally, Prometheus slows down as the volume of data increases. The program is fine for short and mid-term trending, but if you want to keep trending data safe for years, you will reach the limits of Prometheus as it loses much of its speed." [admin-magazine.com]

However, is actively maintained and is improving all the time. Meanwhile, a fork of Prometheus has been created. Cortex is Prometheus but with scalling, which is the major complaint.

An alternative to Prometheus is called the TICK stack, consisting of the tools Telegraf, InfluxDB, Chronograf, and Kapacitor. Telegraf is an agent that can collect, normalize, and export (to a time-series DB) metrics. InfluxDB is the time series database (can also store text strings!). Chronograf is a data visualizer (creates dashboards like Grafana); many users prefer Grafana (but TIGK is a terrible acronym). Kapacitor is an alerting engine, taking alerts from InfluxDB but also able to generate alerts from rules you define.

In addition to individual (and usually FOSS) tools, there are some all-in-one tools too. Generally, these are easier to configure but provide fewer capabilities. One very popular (2015) all-in-one tool is Zabbix. Another is OMD (Open Monitoring Distribution), a Nagios/Icinga system with many plugins already integrated and configured.

If you manage a very large and diverse network of several clusters of computers separated geographically, Ganglia is an obvious choice. Ganglia is used by Twitter, the San Diego Supercomputing Center, Industrial Light & Magic, Virginia Tech, Microsoft, NASA, National Institutes of Health, Harvard, and CERN, among others.

On the other hand, Munin is better suited for administrators of smaller networks, who need to be able to identify and fix problems by examining network history. Because Munin is written in Perl, its functionally can be

extended by using the huge archive of tested Perl code and documentation in the Comprehensive Perl Archive Network or CPAN library.

Nagios is an active system-availability monitor, along the lines of Openview, Tivoli, Unicenter, Patrol (which are all expensive), or OpenNMS. (There is also Groundwork Foundation, which competes well with the expensive ones, but is based on Nagios.)

Nagios is focused on service and host uptime monitoring and alerting. One of Ganglia, Munin, Cricket, Graphite, or MRTG (there are many) is useful as a data collector and trending tool, to generate graphs based on collected data. Some of these tools are better in some situations, but basically all do the same tasks. (Many of these tools use RDDtool internally to store the data and generate graphics.)

Most shops run both Nagios (to send alerts in the middle of the night when something breaks) and one or more of these tools: Cricket can be useful for long-term trending and data collection for non-Linux systems (routers, air handlers, etc.). Ganglia can provide high precision system statistics. (Most SNMP monitors only run every 5 minutes, whereas Ganglia provides statistics collected much more frequently.)

An alternative to Nagios is Zenoss. (See also sourceforge/zenoss.) It may be the better choice when you have a large (>100 networked devices), complex network: multiple locations, networked storage, virtual machines, clouds, mobile devices, and whatever else people are stuffing into their IT infrastructures these days. *Zenoss Service Dynamics* (not included in the free core) gives a complete, real-time picture of a complex dynamic network. It has *dependency and service mapping* that shows the relationships between all of your various network elements, useful when troubleshooting. Zenoss has a feature called *Automated Root Cause Analysis* for zeroing in on problem spots. (Both Nagios and Zenoss have free "community" editions, plus commercial editions with more features. Both allow third party plugins.)

There are many other tools out there, some of which are quite good. A Google search for "(Unix OR Linux) (System OR Network) Monitoring tools" will find many hits. Solaris comes with many excellent tools; the names usually start with `sm*`.

Commercial tools like **HP OpenView** and **IBM Tivoli** are very good but expensive, and generally not worth the investment for SOHO. These conform to ITIL standards.

The popular frameworks which probe the kernel and thus can be used to generate kernel-level metrics include Systemtap and eBPF for Linux, and Dtrace for Unix (with limited Linux support available due to licensing issues). These are programmable and are powerful. Indeed, many of the other monitoring tools available are built atop one of these frameworks.

Learning to program a framework is no harder than learning to script (also, no easier!). Once mastered, you probably will find you don't need other tools.

SystemTap is a tracing and probing tool from Red Hat that allows users to study and monitor the activities of the kernel in fine detail. It provides information similar to the output of tools like `netstat`, `ps`, `top`, and `iostat`; however, SystemTap is designed to provide more filtering and analysis options for collected information and is useful for performance monitoring. For Solaris and some other systems, use Dtrace.

A new project (2013) aims to standardize enterprise management. OpenLMI should provide a framework for remote daemons to respond to queries, and forward information, to a management tool. (See also SNMP, below.)

There are so many tools with similar capabilities, but strength in only one or two of those, which it is nearly impossible to keep up with the list. Indeed, often a system admin is not the person who designs the monitoring, performance collecting, log management, and alerting infrastructure and that skill takes specialized knowledge and a good understanding of your organization's KPIs.

The website github.com/monitoringsucks contains a wealth of information on various tools, common metrics and events to collect and monitor. (Sadly, not updated since 2017.)

Power (Electrical) Monitoring

Intel provides PowerTOP, a tool to monitor power consumption of Linux systems. PowerTOP gathers statistics from the system and presents you with a list of the components that are sending wake-ups to the CPU most frequently. PowerTOP also makes suggestions for tuning the system for lower power consumption. When it runs, PowerTOP gathers statistics from the system and presents you with several important lists of information as well as suggestions.

At the top is a list of how long your CPU cores have been in each of the available C and P states. The longer the CPU stays in the higher C or P stats the better (C4 being higher than C3) and is a good indicator of how well the system is tuned towards CPU utilization. Your goal should be residency of 90% or more in the highest C or P state while the system is idle.

The second piece of information is a summary of the actual wake-ups per second of the machine. The number of wake-ups per second gives you a good measure of how well the services or the devices and drivers of the kernel are performing with regard to power usage on your system. The more wake-ups per second you have, the more power is consumed, so lower is better here.

Any available estimates of power usage are followed by a detailed list of the components that send wake-ups to the CPU most frequently. (This can affect laptop batteries, not just servers.) At the top of the list are those components that you should investigate more closely to optimize your system to reduce power usage.

(See also the `sensors-detect` and `sensors` commands on Linux.)

SNMP/RMON: [*Discussed in networking course*] Overview: used to monitor any host or network device (routers, etc.) with a few simple commands: **`get`**, **`set`**, **`trap`**. (And `get-next` for fetching rows of a table of data. Also, version 2 added `getBulk` to make fetching data more efficient than a series of `get-next` commands, and an `inform` command for relaying traps between NMSes.) The info you can get varies with vendor and model (and firmware version), defined by **`MIBs`** (*management information base*). Each vendor/model/version has its own MIB. Even OSes such as Linux have MIBs!

A ***Network Monitoring Station*** (aka NMS, Monitor Console, or management console) is used to collect data and display/log it. A system admin must download the MIBs of all their managed devices into the monitor console.

When using SNMP always use at least version 2c, or preferably version 3 which features security and performance improvements. **Never trust the security of SNMP <3.** Beware of allowing `set` commands to Internet-exposed devices! SNMP is useful for monitoring network devices and hosts, but may consume lots of bandwidth. Command line versions of these tools are available for Linux (as well as a server to respond to requests), look for `snmp*` commands. Often such tools can be run via cron to produce a constantly updating web page.

> One of the oldest and still popular tools that uses SNMP is MRTG, which manages its data files so they never grow too large and produces beautiful web-based graphs.
>
> Show YborStudent MRTG Demo.

Sysstat and Process Accounting

`uptime`, `top`, `vmstat`, and other tools are great if you happen to be using them the moment something bad happens. What about when something bad happened last night? The **`sysstat`** package includes tools that continuously collect the same statistics and save them into files you can look at later.

Once sysstat is configured and enabled, it will collect statistics about your system every ten minutes and store them in a log file, under either `/var/log/sysstat` or `/var/log/sa` via a cron job in

`/etc/cron.d/sysstat`. Another (daily) cron job runs to rotate out the day's statistics. By default, the logfiles will be date-stamped with the current day of the month, so the logs will rotate automatically and overwrite the log from a month ago. So you get a month's worth of the same sort of statistics as from the "live" monitoring tools.

> Additionally, ***process accounting*** **is used to track system use to allocate system expenses (say by department**). (Demo if time permits.) Full accounting can take a lot of CPU time, RAM, and disk space, and is rarely used anymore. But it uses the same sysstat collection utilities, so you get it "for free" when you are monitoring your system. See the `acct` (or `psacct`) package for your system.

The `sar` utility is used to collect and record system activity, and to format that data to produce reports. `Sar` may be composed of several separate utilities, depending on your system. You can control them all with the `sar` command, but you may find the sub-commands are run directly from cron (e.g., Fedora). Also, **`ksar`** (sourceforge) produces graphic `sar` reports as PDFs.

Before you can examine your system, you must first collect the information. This is done by running the following command:

```
sar -o sar.dat [interval [count]] >/dev/null 2>&1 &
```

This makes snap-shots of the values of various activity counters in the kernel, to a file. If *interval* is zero, `sar` generates a report of the average statistics for the time since the system was started. Otherwise, *interval* is a number of seconds between taking samples (snap-shots). **The smaller this value the more *granularity* or accuracy you get, but the more data must be stored.** *Make sure you have sufficient disk space available for the data you collect!*

If *count* is zero, `sar` will generate data continuously until stopped. (Default *count* is one.) This type of collection is useful to characterize system usage over a period of time and to determine peak usage hours.

The collected data can also be saved in the file specified by the `-o` *`filename`* option, in addition to being displayed onto the screen. If filename is omitted, `sar` uses the standard system activity daily data file, `/var/log/sa/sa`*dd*, where *dd* indicates the current day of the month. So just "`sar`" shows today's report.

> The number of days of data to retain is set in the file (on RH) `/etc/sysconfig/sysstat`. The default is 7, but old files may not be deleted, so you could end up with up to a month's worth.

This command may be added to your boot-up scripts (`rc.local`) or run by cron. **Show `/etc/cron.d/sysstat`**. I have turned this on for YborStudent so you can practice and generate reports.

When using `sar` to generate a report you specify which data file to read with the `-f` *`filename`* option (if *filename* is omitted the daily system activity file is used). With no specific options, `sar` reports averages and totals for some system-wide statistics. You can select which items to view using various options. Use `-A` to report everything. (**Demo `sar -A`**.)

Use the default report to begin system activity investigation, since it monitors major system resources. If CPU utilization is near 100 percent (*user* + *nice* + *system*), then you can conclude the workload sampled is "CPU-bound". Try **`sar 2 3`**.

Review man page options to see what data can be reported.

In addition to this kernel-eye view of activity, you can use the command **`ac`** to view user's connection statistics (must enable with `touch /var/log/wtmp`). Use `lastcomm` to view statistics for recently run commands.

You can enable collection of user (process) activity with **`accton on`**. This saves info about every command executed in `/var/log/pacct`. Use **`sa -u`** to (or `dump-acct`) generate a report of this data. (Warning: the `pacct` file can grow very quickly.)

> You can get useful information about just one process, using the `time` utility. Try "`/usr/bin/time -v find |wc -l`" or "`/usr/bin/time -v links http://www.hccfl.edu/`". (Note, the bash built-in `time` doesn't support the Gnu "`-v`" option nor does it provide the same detail.)

System Auditing

Some process accounting can also be done with the system auditing feature of your OS. A *system auditing feature* is a part of the kernel, that can watch for and log every action (or just some of them, if you desire) of every process.

For Solaris, the audit subsystem is called the ***Basic Security Module*** (or *BSM*), and uses the **`audit`** command. When enabled, BSM can create an extremely detailed audit trail for all processes on the system. The level of auditing produced is at the level required by systems attempting to achieve the DoD "C2" level certification. The data collected is every system and library call for every process.

For Linux, auditing is handled by the **`auditctl`** command and the data can be examined with tools such as **`ausearch`** and **`aureport`**. Auditing can be enabled for particular resources and programs, by setting (and later removing) ***watches***. These can be set to take effect (loaded into the kernel) immediately with the `auditctl` command but such won't persist past a reboot. Instead, you can put the rules in a file *`something`*`.rules` in the directory `/etc/audit/rules.d`. (You can also edit the `audit.rules` file already in there.) **Do not edit the file `/etc/audit.rules`**; that file is automatically generated by intelligently merging the `rules.d/*.rules` files. To manually update the main file (and to load such rules into the kernel without a reboot), use the **`augenrules`** command.

When Linux kernel auditing was new, it was off by default and you needed to enable it by passing the kernel a boot time parameter "`audit=1`". You can use this to disable auditing ("`audit=0`") or to lock auditing on so it can't be turned off without a reboot ("`audit=2`"). Setting audit=1 (or =2) is a good idea to ensure the auditing is on as early in the boot process as possible. Use the `audit.conf` file (on Fedora, in `/etc/audit/`) to set these ("`-e 0|1|2`" and other parameters.

Audit data is stored automatically stored to `/var/log/audit/audit.log` by the audit daemon, `auditd`. However, another daemon named **`audispd`** watches the kernel for audit data and can dispatch it to other programs (including syslog) and sockets. This allows for real-time monitoring of audit data.

The audit data can be overwhelming. You can limit what you look at using **`ausearch`** (rather than trying to read the raw log file).

There are other audit system utilities you can use, but only one other is important to discuss here, **`aureport`**. This utility can produce summary reports of some or all events, depending on the options used. The reports include event numbers, so you can later lookup up a specific even using `ausearch`. Note that producing reports can take seconds to minutes. Here's a sample report from YborStudent:

```
# aureport

Summary Report
======================

Range of time in logs: 04/29/18 21:58:49.504 -
05/17/18 16:50:06.416
Selected time for report: 04/29/18 21:58:49 -
05/17/18 16:50:06.416
```

```
Number of changes in configuration: 250
Number of changes to accounts, groups, or roles:
577
Number of logins: 263
Number of failed logins: 7113
Number of authentications: 300
Number of failed authentications: 6665
Number of users: 36
Number of terminals: 30
Number of host names: 2010
Number of executables: 21
Number of commands: 8
Number of files: 7
Number of AVC's: 51
Number of MAC events: 483
Number of failed syscalls: 6
Number of anomaly events: 8
Number of responses to anomaly events: 0
Number of crypto events: 74731
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 14666
Number of events: 101762
```

Example Use of Auditing Watches

Auditing can be a valuable trouble-shooting tool. You can set watches for files then see what processes (and users) modified those files.

For an example, I got tired on Fedora of `yum` (this was an older Fedora that didn't yet use `dnf`) starting at boot time. But with systemd, there's no easy way to see what starts that. So I added the following audit watch in `audit.rules`:

```
-w /usr/bin/yum -p x -k yumwatch
```

"`-w /usr/bin/yum`" says to watch the path `/usr/bin/yum`; "`-p x`" says to log attempts to execute it; "`-k yumwatch`" adds a name (*key*) to all related audit log messages. Then I ran `augenrules --check && augenrules --load`.

To remove a watch set using `auditctl`, just repeat the exact command but use "`-W`" instead of "`-w`". When added by editing the `audit.rules` file, just comment out or remove that line from the file, or delete or rename the

file so it doesn't end with the "`.rules`" extension, then re-run "`augenrules --load`".

After rebooting (some time later), the command "`ausearch -k yumwatch`" produced this output:

```
time->Sat Sep  1 22:09:37 2012

type=PATH msg=audit(1346551777.419:89): item=2 name=(null)
inode=1701321 dev=fd:01 mode=0100755 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:ld_so_t:s0

type=PATH msg=audit(1346551777.419:89): item=1 name=(null)
inode=531656 dev=fd:01 mode=0100755 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:bin_t:s0

type=PATH msg=audit(1346551777.419:89): item=0 name="/usr/bin/yum"
inode=570439 dev=fd:01 mode=0100755 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:rpm_exec_t:s0

type=CWD msg=audit(1346551777.419:89):
cwd="/usr/libexec/webmin/webmincron"

type=EXECVE msg=audit(1346551777.419:89): argc=2
a0="/usr/bin/python" a1="/usr/bin/yum"

type=EXECVE msg=audit(1346551777.419:89): argc=3
a0="/usr/bin/python" a1="/usr/bin/yum" a2="check-update"

type=SYSCALL msg=audit(1346551777.419:89): arch=40000003
syscall=11 success=yes exit=0 a0=87d27d0 a1=87d29e8 a2=87d19d0
a3=87d19d0 items=3 ppid=2425 pid=2426 auid=4294967295 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none)
ses=4294967295 comm="yum" exe="/usr/bin/python2.7"
subj=system_u:system_r:rpm_t:s0 key="yumwatch"
```

From the **highlighted** line, it was obvious to me that `webmin` was running `yum`. After removing the watch, I ran Webmin and turned that off. (The setting "Collect available package updates?" was found under "Webmin configuration -->Background Status Collection"; but that setting can also be found in System-->Software Package Updates, called "Check for updates on schedule?".)

> A more sophisticated audit rule would have included the name of the executable that ran `yum` directly, and other data. Had the watch failed to pinpoint the problem, I would have had to figure out other `audit` rules to use. As an example, here's a rule that generates an audit event record whenever any file is deleted in the `/etc` directory (all one line):
> `-a always,exit -F dir=/etc -F arch=b64 -S unlink -S unlinkat -S rename -S renameat -S rmdir -k delete`

System Tuning: Turning Metrics into Actions

Besides turning your collected performance data into useful reports, if your monitoring shows issues, you should take corrective action. If things aren't as they should be (the *baseline* and system requirements determine this):

1 define the problem
2 determine the cause(s)
3 decide what to do about it (state specific performance goals)
4 formulate a plan (a change order, which gets scheduled)
5 monitor the result (measure the performance both before and after the changes to determine the effect of the parameter change)

Also remember, ***If it ain't broke, don't fix*!** Modern systems are largely self-tuning, so change settings reluctantly.

> How fast should your services be? A typical time from request to response is 200-300 mSec. Longer than this, businesses report lost revenues. This time is sometimes called a service's *latency budget*. Performance metrics can tell you where each delay comes from, and how long it is. Only then can a SA know how to improve performance.

Some things you can adjust (tune), and the metrics that show when you might want to adjust them, include:

- **CPU** — priorities, affinities, scheduling, scheduler and other kernel parameters (change only reluctantly!), number of CPUs. Tools: `free`, **vmstat** ***interval*** **[*count*]** columns: **r**=#of waiting processes that could be running, **ca**=# of context switches, **us**=%usertime, **sy**=%system time, **id**=%idle time, **`sar`** (system accounting), **`uptime`**: 1, 5, 15 minute *load averages*, also `top` and `tload` (displays ASCII graph). `mpstat`, **`procinfo [-a]`**. (May need to install additional packages for these commands, such as `sysstat`.) Use this info to recognize a CPU shortage (or a CPU "hog" process, found with **`top`**—e.g., if you see MySQL taking 93% of the CPU time, that is the problem). Also **`pidstat`** (like `top`, only non-ncurses). (Also `dstat`, `simon`, `htop`, `atop`, and others may be available depending on your OS.) A good way to visualize where your application spends its time is to use a flame graph.

> The **load averages** reported by `w`, `uptime`, `top`, `tload`, etc., come from `/proc/loadavg`. Every so often (5 seconds on Linux) the system counts the number of processes (actually *threads*, or on Linux, "tasks") that are runnable. This number is combined with the previous 3 load averages using some complex math ("exponential moving average") so that the three new numbers represent an estimate of the number of processes that are in the run

queue, for each 5 second interval, averaged over 1, 5, and 15 minutes.

So, a 1-minute load average of 8.5 on a host with 2 cores means that for the last one minute, about 6.5 processes had to wait. If the load average is less than the number of cores, then no process had to wait. For example, a 1-minute load average of 0.25 means that for the past minute, on average, the system has been 25% utilized.

Note that a load average of 2 on a single core system shows host about as busy as one with a load average of 8 on a four code system.

On Linux (not on Unix), processes waiting on I/O (and so using zero CPU time) add to the load average. Thus it is entirely possible to have a high load average with less than 100% CPU usage. This is an indication that you are I/O bound, not CPU bound.

The **percent utilization** shown is the sum of the utilization of each core. So, a four-core system might show a utilization of 400%. Per-core stats are available from the Gnu `top` command, by hitting "`1`" in interactive mode, or using the `mpstat` command.

Many processes heavily use the CPU when first starting up. Such a burst of high utilization does not indicate a problem; only if it lasts.

The bottom line: you can't blindly assume high load or utilization mean a problem. If your system is responsive, there is no problem.

- **Memory** — per process limits, swap space, some kernel mem. parameters.
  vmstat ***interval*** **[*count*]** columns: **`free`**=amount of avail RAM, **`so`**=# of pages swapped. **`free`** (check amount free, calculate %free, say with a script). **`swapon -s`** shows amount of swap space used. Use this information to recognize a RAM shortage. Also use **`top`**, **`sar -r`**, and **`/proc/meminfo`**. Two values are useful for a quick picture of whether your problem is with memory: `%memused` and `%swpused`. If not swapping, you don't have a problem even if memused is near 100%; it depends on how many processes are sleeping. If there is lots of these, then you might have a RAM shortage.

  The memory statistics can be confusing! When possible, the Linux kernel will use idle memory for disk buffers and page caches. But before doing any swapping, the kernel will shrink these (but not to zero). You get a better picture by mentally adding free + cache + buffers = estimate of free memory. (So sometimes the memory used will always seem close to total memory, even when you have plenty.)

Some commands automatically attempt to adjust the reported free memory for this, but that doesn't always work (and the man pages don't always tell you this is occurring). Use "`free -o`" to disable this adjustment, and do it mentally yourself.

Note the system always swaps some data preemptively, so that will never be 100% empty even when not swapping.

The total memory shown by these commands is less than the memory you actually have, because some memory reserved for the kernel is not considered "available". (Also, a left-over fractional page isn't usable.)

This shows no memory problems (note 512MiB = 524288KiB; see box above):

```
$ free
             total       used       free     shared  buffers     cached
Mem:        515164     503280      11884          0      77428     191856
-/+ buffers/cache:     233996     281168
Swap:      2096440      66216    2030224
$ vmstat
procs -----memory----------  -swap- --io--- -system- -----cpu------
 r b   swpd   free   buff  cache si   so   bi   bo   in   cs  us sy id wa st
 0 0  66216  11888  77380 191832  0    0   14    5    1    1   1  0 99  0  0
```

These commands show 11,884 1KiB of free memory, but the adjusted amount is 281,168 KiB. While there is 66,216 KiB of swap used, there are zero pages swapped in or swapped out shown, so no swapping is occurring. (The 66 MiB shown as swapped does not mean adding more RAM will prevent that! A lower `vm.swappiness` value (0-100, default=60) might; that is discussed later.)

The `top` utility reports memory used per process in the columns `VIRT` (total virtual memory used) and `RES` (physical memory used). But both of these columns include DLLs and in some cases, video RAM used. A better way to see how much memory is used per process is to hit '`f`' once top starts, and add the `DATA` column. That column only shows that process' stack and heap space used (virtual).

The effects of memory caches are not well understood. It takes time for a cache to "warm up" and become useful. A cache with no useful data in it is considered "cold". A loaded cache is "hot".

Linux keeps track of swapped-out pages that for one reason or another are also in memory. When Linux needs to swap a physical page out to swap space, it consults the *swap cache* and, if there is a valid entry for this page, it does not need to write the page out to the swap file. This is because the page in memory has not been modified since it was last read from the swap space.

The kernel won't allow its caches to grow too large. Even if there is plenty of RAM you might be short on cache space. It turns out on Linux, the total memory used for the swap cache and page cache combined is limited, meaning if your system starts doing lots of disk I/O, the page cache grows as the swap cache shrinks. This can mean that disk activity (such as backups) can indirectly impact CPU/memory intenisive processes, in some cases up to 40% slowdown! (Setting the `vm.swappiness` parameter to zero may help with this.)

> How long might it take a cache to warm up? Suppose your disks can perform 2,000 1-block (2 sectors, or 8KiB) reads per second, and you have a 600 GB SSD for a second-level cache, and 128 GB of RAM as a first-level cache. Since the caches fill at a rate of 16 MiB/sec, it takes 2+ hours for the RAM cache to warm up, and 10+ hours for the flash cache! Home users benefit more from faster disks then big caches.

- **Disks** — number of controllers, hardware RAID, LVM design, file placement, I/O parameters such as DMA, write cache, and number buffers. Monitor disk and controller stats: **`i2cdump`**, **`iostat`**, **`ioping`**, **`hdparm`**, **`sg*`**, **`smartctl`**. Monitor disk space: **`df`**, **`quota`**, **`repquota`**, **`du`** (spot huge websites, explosive log files). Use **`sar -dp`** for this. If SMART is supported, monitor the `Reallocated_Sector_Ct` (attribute #5, I think). This number should be low and rarely change. If you see it increasing over time, it indicates the disk is ready to fail.

  `Avgqu-sz` (average queue length of requests issued to the device) and `svctm` (average time in milliseconds that I/O requests for this device had to wait before being handled) are the two most useful values for determining if you have an I/O-bound machine. The longer the queue, the more requests are piling up before they get serviced. The longer each has to wait before being serviced, the slower everything gets.

  > The Linux 2.6.32 IO scheduler ("CFQ") got a new feature that improves the desktop experience by reducing the effect of background I/O activity. But it can cause noticeable performance issues. For servers without X, throughput is noticeably improved by turning it off:
  > echo 0 >/sys/class/block/<*device name*>/queue/iosched/low_latency

  The output of `iostat` can be used to see which storage volumes are getting the most traffic, and if that is mostly reads or writes. A more understandable tool is **`iotop`** (if available).

- **Network** — Memory config, # of NICs and their configs, traffic shaping and other network parameters, external network infrastructure upgrades.

To monitor your network interfaces: **`ip -s link show`**, or **`ifconfig`**. To monitor overall network activity: **`netstat -lep --inet`**, **`ping`**, **`nc`**, **`traceroute`**, **`lsof -i`** (show open network connections), **`+M`** (portmapper data), **`-n`** (don't translate addresses), **`-P`** (don't translate port numbers), **`-N`** (show NFS files), **`nmap`**. A good Linux tool is **`iptraf-ng`**, an ncurses `top`-like program showing lots of network statistics. Another is `nethogs`.

If you see values greater than zero for errors, dropped, overruns, and collisions, there is likely a network bottleneck problem. Check at a few different times and see if the problem is persistent. If it continues, it bears further investigation; if not, note it to see if a trend becomes apparent.

- **Applications (servers)** — Many have status options/commands, or examine their log and configuration files. Examples: httpd, DNS (bind), FTP, SSH, DB, mail, etc. *Load balancing* (clusters) may be needed.
- Monitor hardware sensors with (lm-sensor pkg) **`sensors*`**.

Change tunable parameters on the live system with **`/proc/sys/*`**, or with `rctladm -l` on Solaris < v8, `prctl` on Solaris >= v8, `sysctl` on Linux. Such changes are not permanent. To make changes permanent you either add the commands to some boot script or (preferred) edit an appropriate file. Locations for parameter files include `/etc/default/*` (Most Solaris params are found here), `/etc/system` (Solaris) and `/etc/sysctl.conf` (Linux), `/etc/rc.conf` (BSD and some Linux), `/etc/sysconfig/*` (Red Hat). Some params have special commands, e.g. for networking you have `ifconf`, `ip` (Linux), `ndd` and `routeadm` (Solaris), `route`, … Documentation for these parameters can be found in the `proc(5)` man page, or (for Linux) on `kernel.org`.

Diagnosing Problems: Checking the Usual Suspects

When there is a problem, or you suspect there might be one, where do you look? There are a number of steps you can do to try to spot the problem: the usual suspects. In no particular order, here are some:

- Check all log files, and `dmesg` output, for anything unusual. Of course, you should check these every so often, so you know what usual is on your systems.
- Check for any services that failed to start, or have crashed. On systemd systems such as Fedora, you can run `systemctl`. On Solaris, run `svcs -a`. On Ubuntu (the main system using Upstart), run `initctl list`.
- Check for full (or unusual changes in size of) storage volumes, using `df`.
- Check for network issues, using ifconfig -a. Make sure all expected interfaces are up. Check tx and rx error counts.

- If the system has an unknown problem after an update, check tunable parameters to see if any have values left over from the previous system. Make sure any that are not at the default value are what they should be.
- Check the load averages with top. If the number of threads ("tasks") is two or three times the number of cores, the system will be slow.
- Check `vmstat`, `iostat`, `mpstat`.

Resolving Issues and Fire-Fighting

System administrators check various logs, monitors, dashboards, and trouble-tickets/reports. Eventually, an issue comes along that required immediate attention, even if that causes disruption. Then the issue, whatever it is, must be resolved.

Often, issue resolution doesn't happen at first. Sometimes it is more important to restore service stability using temporary measures (such as a reboot to fix a memory leak) than to fully fix the problem at once. Such a *quick and dirty* fix won't last long, but it will provide extra time to understand and permanently resolve the issue.

On a bad day, you have multiple issues and must prioritize them: minor issues, performance problems, service outage, and all-hands-on-deck, 5-alarm fires (data loss, security breach). Borrowing from the medical field, this is sometimes called ***triage***.

After resolving an issue, there is a strong temptation to return immediately to the task that was interrupted. However, this is not a good idea. While the details are still fresh, is it important to conduct a ***post-mortem*** meeting. The purpose is to disseminate knowledge of the issue and the step(s) that resolved it, and see if it could have been prevented by a change to procedures, monitoring, etc. For any good ideas that come from the meeting, assign specific people to work on them, and they should have due-dates as well.

It is very important not to allow a post-mortem meeting (or any meeting) to become a "who's to blame?" session. Even if a human error caused the problem, know that humans can and will make mistakes. Review the procedures followed to see if the chance of such errors can be reduced, or more-quickly caught.

Lecture 13 — Logging and Log Management

To identify problems and trends, and to trouble-shoot them requires observing events over a period of time (**historical monitoring**). Since it is generally impossible to observe all events as they occur, most daemons record important events to files known as ***log files***. Log files are used for audits, for evidence in legal actions, for incident response, to reduce liability, for various legal and regulatory compliance reasons, and for debugging and trouble-shooting. Email logs can alert you to spam problems. Web logs may be useful for marketing and website design, and so on. When upgrading or deploying new (or newly configured) services, log data can be valuable in finding problems quickly.

> Logging and monitoring have a lot in common, and many of the issues for monitoring apply to logging. Today, logs are often considered as a stream of events. The logs can be analyzed and metrics extracted and/or derived, and sent into your monitoring and alerting system.
>
> However, log data is generally not a time-series of some metric, but actual text (or other) data produced by daemons, applications, or even the kernel. Such data cannot be summarized with a time-series database. Log data is saved in a different type of database.

Early services managed their own log files, each in a unique format and location. Today most (but not all!) rely on logging daemon originally named ***syslog*** to collect, identify (i.e., host, command name, and process ID), time-stamp, filter, store, alert, and forward logging data (almost always text) from many daemons. (That is, daemons such as named or MySQL can use a syslog API to send log data to syslog.) Syslog has the added benefit of partially standardizing log file formats, making it easier to examine log data with various tools. (Syslog was developed first for BSD, but there was no standard for it, resulting in many incompatible "syslogs".) However, some daemons may need to have their default logging configuration changed to have them take advantage of syslog (or indeed, to provide any log data at all).

> An important part of a system administrator's job is to check regularly various log files. Be sure to learn where your hosts keep their log files, as the directory is different for different flavors of *nix.
>
> As this task is actually impossible, a sys admin actually uses tools to monitor log files automatically, look for the more important/interesting events and summarize the rest, in a short report. Some log data should not wait however; the monitoring tools generally have a real-time alerting facility for more critical log messages.

An important use for logs is troubleshooting. In most cases, what goes into daemon logs are the only information you have to understand what went wrong in

the production application. This includes logged errors, warnings, caught exceptions, etc. In addition to such data, there are logs which relate to business metrics. Things like sales data, user behavior (click-traces), security events (session start and timeouts, login events, etc.), and application-specific data (e.g., removing items from your shopping cart). Analysis of such log data is used to produce *key performance indicators* (KPIs). Indeed, many metrics can be derived from log data.

Data and Network Requirements of Logs (and Metrics)

A typical log entry from Apache httpd (web server) access look might look like this (each log entry is typically one very long line, shown here as line-wrapped):

```
40.168.17.142 - - [26/Nov/2017:12:21:29 -0500] "GET
/favicon.ico HTTP/1.1" 200 1078
"http://yborstudent.hccfl.edu/wiki/index.php/Main_Page"
"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/62.0.3202.94 Safari/537.36"
```

The fields mean: the source IP, the user identity determined by identd ("-" means unknown), identity determined by HTTP authentication, the time the request was received, the actual request, the status of the request ("200" means success), the size of the response, and other data.

While some log entries are much shorter, some can be kilobytes in size. Log entries allow you to answer questions such as: what URL was requested? How long did it take? What was the return code? What was the source IP address?

For comparison, a typical metric looks like this:

```
http_request_total_per_minute 123 1511774466
```

(format: name, value, timestamp) or possibly like this:

```
http_request_total_per_minute
{method="post",code="200"} 123 1511774466
```

Suppose you have a mere 10 web servers in your cluster with 1,000 requests per second, and each request generates a log message of 1 KiB. That comes to nearly 100 MiB per second, filling up your network connection with just web access logs. And the storage required would be 36 GiB per hour! While you do get the benefit of lots of information about each event, the network and storage requirements of your organization (and budget) put a limit on how much log data you can save.

With metrics the bandwidth, disk I/O, and storage requirements are much, much less, but missing are the details that may be useful. In practice, you need both logging and metrics. (Note how these metrics could be derived from those log messages.) Often, the amount of log data (the level of detail) is adjustable by a

system admin, who will increase logging when deploying new services and troubleshooting. During normal operation, logging is decreased.

> Because of the differences between log data and metric data, log systems often include their own dashboard and a separate (from monitoring) console where queries can be entered. (Collecting log data from many sources is just as complex as collecting metrics from many sources.) It should be possible to add loging dashboards to Grafana however. But you will still need to learn two query languages, one for the metrics database and one for the log database.

Log Files

Log files need to be examined or they are useless. Too many sys admins never bother to configure logging, leaving everything at a default setting, and never bother to examine the log files (until it is too late.)

However, it would be foolish to try to read all log data, all the time. Since it is impossible to know in advance what log data will be useful, you end up collecting far more than any human (or SA) can possibly read and understand. This can be partially managed with log alerting and parsing tools. Such tools monitor log files automatically, look for the more important/interesting events, and summarize the rest, producing a short report. Some log data should not wait however; the monitoring tools generally have a real-time alerting facility for more critical log messages.

Log Summarization

Usually, summaries of the data are sufficient to alert you to potential problems, at which time you would then examine the relevant "raw" log entries. Some of these tools (also known as *data reduction* tools and *system audit* tools) include `logwatch`, `logcheck`, `swatch`, `logsurfer`, and SEC. Such tools are related to host-based intrusion detection systems (HIDS).

Mere humans can't manage to read megabytes of log files per day and expect to spot the important events. On most production environments ***log file monitoring tools*** (sometimes called *log file filtering* or *reduction tools*) are used to look for trends, security related events, etc. (**Show daily `logwatch` output from `wpollock.com`.**)

Some nicer tools can prepare graphs and reports (once setup) automatically. Use tools mentioned below along with `cron` to make reports. **Show saved webalizer web-site 2006 usage:** wpollock.com/usage/, also Feb 2015 usage. Show **awstats for Feb. 2015** for http://wpollock.com/ and for https://wpollock.com/. Show current wpollock.com stats from cPanel at GoDaddy.com (username: 34675237, password: *It's a secret!*). Show YborStudent usage demo.

Such tools usually require a lot of configuration to tell them what is of interest and what must be monitored. Commercial and free tools exist including some that can monitor whole networks of servers (and of course the networks themselves).

Unix and Linux systems generally come with a few basic tools such as **`logwatch`**, logcheck, (which are Perl scripts that use *regular expressions* to sift through log files), swatch, logsurfer, etc., but not the more comprehensive ones.

A good FOSS one is Simple Event Correlator (SEC); download from `simple-evcorr.sourceforge.net`. A good SEC overview is at http://ristov.users.sourceforge.net/publications/sec-issa2012.pdf.) SEC has a generalized rule engine that can correlate virtually any event source, including external syslogs, in real-time, looking for any kind of attack profile. It supports multi-line events spanning any time span, and its rule language is clean and powerful. And it's highly scriptable processing engine can trigger rules in external firewalls and routers, letting you extend protection to an entire network rather than individual hosts. [*TODO: Add SEC example.*]

The default `logwatch` setup on Fedora works reasonably well, but doesn't examine all logfiles. Which service log files are monitored is controlled by `/etc/logwatch/conf/*` files, which override the settings in the default files from `/usr/share/logwatch/default.conf`. All services to be monitored (other than the defaults) need `.conf` files in `/etc/logwatch/conf/services/`.

The default conf files are found in `/usr/share/logwatch/default.conf/services/*.conf`. By default, all these files are used. To modify one, copy it (for example, `sendmail.conf`) to `/etc/logwatch/conf/services/`, and edit that. To get more details, crank up sendmail's `LogLevel` setting from (the default of) `9` to `15` (see `/etc/mail/sendmail.{cf,mc}`). To get more detail in logwatch's report, change the value of "`detail`" in `sendmail.conf` file from (the default of) `3` to `4` (max is `10` for most detail possible).

(Most daemons allow the admin to adjust the amount/detail of log data produced; syslog and tools such as logwatch also have settings for this. In general, you can leave the log tools to report all available detail, and configure the daemons to produce the amount you want.)

To prevent some service from being monitored, edit the `/etc/logwatch/conf/logwatch.conf` file. For example, add this to suppress the "zz-fortune" pseudo-service from running:

```
Service = -zz-fortune
```

Syslog standards are over 20 years old (RFC-3164) and many issues have surfaced with them (see below for a discussion of these). New IETF standards (RFCs) for syslog are being developed (RFC-3164, replaced in 3/2009 with RFC-5424) to address security issues and other syslog shortcomings.

A sys admin must address the problems of the current syslog tools and format. Most hosts today ship with a basic syslog daemon, but a number of replacement versions (some are compatible) include many newer features. See *syslog-ng* (the most popular replacement currently), *module syslog*, *SDSC-syslog*, ***rsyslog*** (used by Red Hat since it is backward compatible with old syslog but with many new features available), and *nsyslog*. (There is also log aggregators, which can combine logs from Unix, Windows, Android, or whatever; see nxlog.)

Note that most network devices (routers, switches, firewalls, printers) today can produce syslog data. **Don't forget to collect logging data from all important sources**, including network devices, printers, workstations, and Windows servers. Use **SNMP/RMON** to monitor network devices that don't support syslog.

Common Linux Log Files and their Location

By default, *nix **systems put (syslog) log files in `/var/log`** (Solaris: `/var/adm`). A number of log file names are also standard (at least on a given distro). On RH systems, you should regularly check **`/var/log/{messages,secure,*}`**, use the Linux **`dmesg`** command (and **`/var/log/boot.log`** if present), and copy `/tmp/install.log` (first time only, as `/tmp` gets erased each boot (usually).) [*Show `/var/log`: `messages`, `boot.log`, `dmesg`, `secure`, ...*] The **`tail [-f]`** command is often used to examine these log files. (Gnu `tailf` is better if available.) (Use **`less -R boot.log`** to have color codes show up correctly.) Other log files include `audit/*`, for SELinux and related log messages. The names of standard log files can be found in the `*syslog.conf` file. For example, instead of `messages`, Debian and Ubuntu use `daemon.log`.

Tip: The command "`ls -lt /var/log |head`" will show the ten most recently modified log files.

Some *nix systems snap-shot the dmesg log to a file at the end of the boot process. This is because the dmesg log is a *ring* (or circular) buffer in memory, holding the most recent kernel log messages. If your system is up long enough, and sufficient messages are generated, old messages will be lost. (You can dump dmesg data to a log file as part of the boot up process.)

There are additional, non-syslog files maintained you should know about. **wtmp** is a binary log of who logged in and when (view with **last** cmd, manage with Linux `sessreg` or Solaris `fwadm`). **utmp** is a binary file of who is logged in now. Two related files: **btmp** (a log of failed login attempts, view with **lastb**) and **lastlog** (not a log file, but a *sparse file* (show with `ls -ls` and `du`) showing the last login per user id (view with `finger`, **lastlog** commands). You must manually create `btmp` and `lastlog` via `touch` if you want Linux to use them.

> The `wtmp` file also records a record for each shutdown, reboot, runlevel change, and system clock change. You can see those with the correct option to `last`. The `utmp` file on modern systems is kept in `/run` (a RAM disk), so is recreated at each boot. Thus to see system boot records with the `who` command, use "`who -b /var/log/wtmp`".
>
> With systemd, the journal is a binary file (or files), with each record tagged with a *boot ID*, a number randomly generated by the kernel at boot time (`/proc/sys/kernel/random/boot_id`). Running the command "`journalctl --list-boots`" reads the whole journal, looking for boot IDs and the earliest and last timestamp of each one.
>
> Every so often, `wtmp` is rotated, losing info. The same can happen with the journal. Thus, the results from `last`, `who`, and `journalctl` can differ. Indeed, since some of that data is written early in the boot process, the system clock may not have adjusted yet. I've even seen reboot times reported a day into the future!

Some data isn't logged by default. Usually, it isn't hard to configure some daemon and/or syslog to generate the data you want. In some cases, it is harder. For example, to log file transfers done via SFTP, you need to know that OpenSSH uses a subsystem for SFTP. You need to configure the subsystem to generate the data, not `sshd` itself. Add this to your `sshd_config` file:

```
Subsystem sftp internal-sftp -l VERBOSE
```

(Note you can also configure sshd to use a different FTP server, such as vsftpd. You then configure that FTP server as normal to generate transfer logs.)

Security Event Manager (SEM)

Other tools exist that attempt to interpret the logs, instead of just securely collecting the data. A ***Security Event Manager*** (SEM), or *Security Information and Event Manager* (SIEM), is a computerized tool used on enterprise data networks to centralize the storage and interpretation of logs, or events, generated by other software running on the network.

The difference between log management and SEM is that the former typically just provides data collection and storage, whereas SEM provides data analysis too.

It is beneficial to send all log events to a centralized SEM (or even just a centralized log) system. Some reasons include (*from Wikipedia SEM article*):

- Access to all logs can be provided through a consistent central interface.
- The SEM can provide secure, forensically sound storage and archival of event logs (this is also a classic Log Management function).
- Reporting tools can be run on the SEM to mine the logs for useful information.
- Events can be parsed as they hit the SEM for significance, and alerts and notifications can be immediately sent out to interested parties as warranted.
- Related events which occur on multiple systems can be detected which would be impossible to detect if each system had a separate log.
- Events which are sent from a system to a SEM remain on the SEM even if the sending system fails or the logs on it are accidentally or intentionally erased.

SEM is relatively new (the term was apparently coined by a vendor in the early 2000s) and SEMs are proprietary and thus non-interoperable. Two efforts to address this in the early 2000s include the OpenGroup's XDAS (See this nice PDF slide presentation on XDAS v2) project and MITRE's CEE. These and other efforts have been discontinued. The only active project I can find in 2020 is the DMTF's Cloud Auditing Data Federation.

Logging Issues

Keep in mind **you don't want to rotate binary log files!** (You may wish to back them up occasionally, and start fresh ones.) Log rotation is discussed, below.

HIDS and NIDS (intrusion detection systems, a.k.a. alerting systems) aren't enough! Don't rely on these tools exclusively. The alerts they generate are often meaningless without log data to see if the unusual event has resulted in a failure or a security breach. Only log data can tell you that. (Common IDSs include *file integrity checkers* such as *Tripwire*, *Osiris*, and *Samhain*. *Snort* is a common NIDS scanner.)

When managing a network, you will have many hosts and devices that generate log data. It is usually best to **funnel all the log data to a single host**. This ***loghost*** is often a dedicated host just for logging, and should be secured by removing all unused services and access, and generally hardened.

With loghosts listening for both local and remote syslog data, make sure you **open up the syslog port in all firewalls between you and the remote host: UDP/514**.

Note UDP is not secure and you should only allow selected hosts to use that port. (`syslog` must be started with the "`-r`" option or it won't listen to the network.)

A nifty trick to send log data via encrypted TCP to a central loghost is to configure old syslog (this trick not needed with modern syslog replacements) not to send any data to the loghost, but instead to localhost. You can then run netcat or a similar tool to listen for localhost UDP/514, pipe that into cryptcat (an encrypting version of netcat) to the loghost, something like this:

```
*.* @localhost # syslog rule
  nc -lu localhost 514 |cryptcat loghost port
```

On the loghost, pipe the data received by cryptcat into netcat, and have that send to localhost UDP/514 (where syslog is listening), something like this:

```
cryptcat -l port |nc -u localhost 514
```

(From *Mastering FreeBSD and OpenBSD Security* by Korff, Hope, and Potter.)

If you don't have or want `cryptcat`, you can use `openssl` instead, something like this (found at commandlinefu.com):

```
nc -lu localhost 515 \
  |openssl enc -aes-256-cbc -a -k secret |nc loghost port
```

And on the loghost:

```
nc -l port \
  | openssl enc -d -aes-256-cbc -a -k secret |nc -u localhost 514
```

Configure your services to use syslog. Many services ship with logging off by default, or attempt to use a custom log file. Most allow syslog configuration. Each service should send all non-debug messages to syslog. If this is too much data, you can configure syslog to filter it. This means you don't continually need to remember how to configure every service, when you need to change its log-level — just configure `syslog` for any service.

When initially installing a new server (or updating one), it will pay to **crank up the amount of log data collected** for that service until you are confident it is working correctly.)

Send all data to a single *central loghost*. If your network is spread out geographically, each location can have a local loghost and these in turn send data to the central loghost. These ***relay loghosts*** should also store the log data, giving you a backup in case the central loghost fails or if a network connection goes down. Storing logging data on a different machine is useful when some host is attacked or crashes; the log data is still safe.

Harvard U. collects syslog data from each switch, hub, router, firewall, and server. Reportedly (*;login: 4/2011*), they collect data from 3,500 devices and 400 servers. All the data goes to a central loghost, running data collection/indexing/monitoring/reporting software called Splunk. The loghost collects about 18 GB of log data per day, which is why Perl scripts (e.g., `logwatch`) and other "home-grown" solutions don't work.

The BSD `syslog` daemon only reports the last hop IP address in the log. So when using relay loghosts, the central loghost will lose the IP address of the original host that generated the message. (Some syslog replacements and the new syslog RFC address this issue.)

Loghosts are attacked by hackers so when using the old BSD syslog, it is sometimes worth the work to build a ***stealth loghost***. Such a host has one NIC connected to a private network, used for SSH connections by the administrator. A second NIC is on the LAN from which you are collecting the log data. This NIC is *unnumbered* and set in *promiscuous mode*.

You use various tools (*netcat* a.k.a. *nc*) to monitor the network for log data sent to a fictitious address on that LAN. The various hosts on the LAN will try to send their log data to that fictitious host. Log data is sent via UDP so the sending hosts won't know the difference. An attacker can't access such a stealth host. (Be sure to turn off ARP on that NIC as well or attackers can find it.) A stealth loghost doesn't prevent *log injection,* but since this makes it impossible to access remotely the loghost from the public LAN, it does prevent some attacks.

Syslog replacements use TCP instead of UDP (syslog is scheduled to use this in the future) and support IPSec, SSH, or SSL tunnels for the transport. **This makes stealth loghosts both impossible and unnecessary.** Digitally signing log entries can also reduce fake log entries (but not if the evil-doer has access to the remote system and can cause it to generate the correctly signed log entries!)

Newer syslog protocols may (or will) **use *syslog-sign* and *syslog-reliable*** for safer transport and sender authentication, and are already supported by some syslog replacements.

If you're not going to replace your current syslog, **consider using SSH or SSL tunnels (see `stunnel`) for transport of syslog data**. Set up a tunnel on each remote host so data sent to a specified unprivileged port on localhost (say port 9999) gets automatically forwarded to the loghost via the secure tunnel.

The problem is tunnels such as `stunnel` and `ssh` only transport TCP and syslog only uses UDP. You can use `netcat` to accept UDP, send as TCP, and reverse this on the loghost. The setup looks like this:

On client: `nc -lku localhost 9998|nc -lk loghost 9999`
On loghost: `nc -lk localhost 9999 |nc localhost -u syslog`

The client has the syslog daemon forward all log data to localhost:9998. ("`*.* @localhost:9998`"). This UDP data is then read by a second `netcat` and re-sent as TCP to localhost:9999. You have `stunnel` or `ssh` tunnel listening at that port, and forwarding the data to the loghost via a secure, encrypted tunnel. The loghost has `stunnel` or `ssh` listening for incoming data at port 9999. The (decrypted) output is then forwarded via `netcat` to `localhost` (the loghost) UDP/514. Using `stunnel` requires X.509 (PKI) certificates, so each end of the tunnel can authenticate the other end. Using `ssh` doesn't require (expensive) certificates.

Log files contain sensitive information. You should address these security issues:

- **Set log file permissions** accordingly! This includes the files and (sometimes) syslog's `/dev/log` device (and similar ones in `chroot` jails). Note, adding too much access to some log files will prevent them from being used!
- Consider **digitally signing the log file** to prevent tampering. If possible, digitally sign each log entry when added to the log file. Before a party may move for admission of a computer record or any other evidence in any court of law, the proponent must show that it is authentic. Some syslog replacements do this already.
  A different key should be used to digitally sign the whole log; either after every entry is added or after the log file is closed/rotated. This is an example of a *dual control* that prevents a single person working alone from falsifying data (e.g., hiding financial transactions to embezzle funds). Digital signatures can do this (along with a copy of your logging policy, often part of the security policy, that shows your data handling policies). See justice.gov/criminal/cybercrime/usamarch2001_4.htm (a PDF copy can be found in the class resources section).

> *Forward Secure Sealing* (FSS) is a method to cryptographically "seal" (sign) logs in regular time intervals. If some host is hacked, the attacker cannot alter log history (but can still entirely delete it or replace it with a fake one). FSS works by generating a key pair of "sealing key" and "verification key". The former stays on the machine whose logs are to be protected and is automatically

changed in regular intervals (and the previous one securely deleted). The latter should be written down on a piece of paper or stored on your phone or some other secure location (that means: not on the machine whose logs are to be protected). With the verification key alone, you can verify the journals on the machine and be sure that (if the verification is successful), the log history until the point where the machine was cracked has not been altered. The two FSS keys are generated using:

```
journalctl --setup-keys
```

For this to be useful, you must periodically check the log signature. (Systemd logs can use FSS since Fedora 18.) Using an external loghost is still more secure than FSS, but if you don't have a loghost, try FSS.

- When a log file is closed (when rotated for instance), consider **encrypting the log file** to prevent unauthorized access. This prevents attackers from *dumpster diving* for your old log files on discarded backup tapes. Especially if not encrypted, you should log all access to your log files!
  Without special hardware support, encryption and digital signatures can take a long time. This can cause a busy log server to fall behind, eventually losing log entries, or crashing altogether! (Some modern syslog replacements report such lost log entry statistics, so you can test your system under a simulated load to make sure you won't lose valuable data.)
- When possible, **don't collect data you don't need**: The best way to keep data private is not to store it at all.
- **Only keep data as long as you need to**. Most industries have standard limits on how long to keep different types of data. Follow those guidelines. As a rule of thumb, syslog data can be kept on-line up to a year; 3 months (or 6 months in some cases) are also commonly used policies. Note older data can be summarized for baselining purposes and only the summaries kept on-line. After that, you need to archive the old logs according to law and the data retention policy. In part, how much old data you keep on-line depends on how much will fit on one backup tape/CD-ROM/DVD. If 4 months nearly fills one DVD, then 4 months may be a better policy than 6 months.
- Of the data you do keep, decide which data can be ***blinded***, which data should be ***encrypted***, and which data can be safely left open:
  - ***Blinding data*** means that it is destroyed, but in such a way that makes it unique. A hash ("one-way") function is a good technique for this.

  - Blinding is a useful way to store personal data such as credit card numbers, phone numbers, addresses, customer ID numbers, etc. Note that blinded data can still be used as a primary key for a database table, as a hash of a unique value is still unique.
  - If possible, randomize the order of such data, to hide the sequence of data (not a good idea for system log files, but useful for, say, financial transaction logs).
  - Blinded data can't be recovered to its original form. So if there is a requirement for sensitive (private) data that must be recoverable, use encryption instead of data blinding.

- Always sanitize external data before logging it. (This applies mostly to developers, but also shell script writers.) You should for example replace/remove any newlines in usernames or filenames, or an attacker can generate fake log entries.
- PCI (the Payment Card Industry; really the banks and credit card companies) have mandated security standards for handling credit card data. Companies must certify each year (with an audit) that they comply with PCI-DSS or pay hefty fines (or lose the right to collect payments via credit cards). One point of PCI-DSS is that you can only store the first 6 and last 4 digits of a credit card number in an unencrypted file. Keep this in mind if you don't encrypt your log files.
- HIPPA, FISMA, and other standards require logging and specific log data handling procedures to be in compliance. Depending on your organization, one or more of these **regulations and laws will require compliance**.
- Syslog doesn't restrict which hosts can send log data, or what data is sent. This make all log data unreliable. Hackers can easily generate fake log data and send it to your loghost. This is called *log injection* and the possibility will reduce the evidentiary value of your data. This can be partially mitigated using a careful firewall configuration, but this issue is better addressed by using one of the modern syslog replacements.

> Advice from the Fedora `syslogd(8)` (version 1.3) man page, item 5 under the "Security Threats" section: "... if the problem persists ... get a 3.5 ft. (approx. 1 meter) length of sucker rod* and have a chat with the user in question. ... (Sucker rod [is a] 3/4, 7/8, or 1 in. hardened steel rod ...)"

Log data can grow to fill even large disks quickly. You must make sure large log files won't crash your system or result in some DOS (when sending log

information across your network). (See *log rotation*, below.) Consider various *data reduction* techniques: summarizing repetitive log entries, discarding low utility log messages, and data compression.

Old log data, at least some of it, has lasting value for *baselining, auditing*, and (if blinded) training. Consider archiving old log data, or filtering the logs and archiving important events and summary data (e.g., 208 SSH logins successful in past day). One useful design would be to have old log data sent to a database. Then you could easily get reports and make queries using SQL.

BSD Syslog time-stamps don't contain the year, so make sure the archived log files' names include the year.

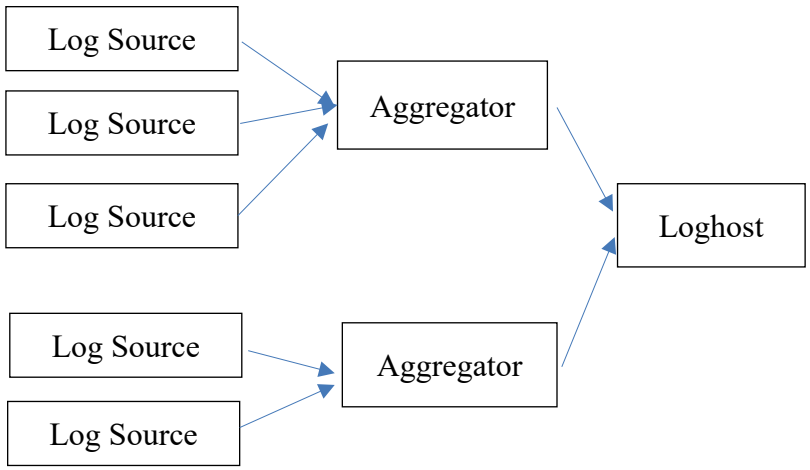
BSD Syslog time-stamps don't contain time zone information. When data is sent to a central loghost, it is important to know the time zone of the original host that generated the log message. Consider using one of the newer syslog replacements that does provide this information, or otherwise take care of this issue when performing data reduction and data parsing.

Accurate network-wide time is vital to correlating log events, and for preserving their value as an audit trail and evidence in court. Use NTP to synchronize your hosts and network devices to the same time. If an external timeserver is not feasible for some reason, pick one (well-connected) host to act as a timeserver. **For about $100, you can install a GPS or radio-controlled clock.**

Enterprise-wide Logging [*Adapted from "Enterprise Logging" by David Lang, ;login: Aug 2013, and from* https://qbox.io/blog]

Logs are a crucial part of any enterprise system because they give system admins, developers, and others insight into what a system is doing as well what happened. Nearly every process running on a system generates logs in some form or another. These logs are usually written to files on local disks. Some metrics can be generated by examining logs, which then must be fed into your monitoring system. Some log messages report critical events, which must be alerted somehow. When your system grows to multiple hosts, managing the logs and accessing them can get complicated.

It is essential to have a central loghost on a system with many hosts. One reason is that some data queries (e.g., security events) and metric generation require correlating data from several hosts. However, configuring every application on every host, and every hosts' logging daemons, to send only sanitized, secure (signed) log data is unrealistic. Some log data sources cannot be configured (such as some routers or other network devices, and some applications that weren't well designed for flexible logging). Instead, a hierarchical architecture should be used:



You can do the log data processing on log *aggregators*, which can be located on each LAN (or even each host) for easy TCP (or UDP if necessary) access by the log data generators, or *sources*.

> Note that even if something doesn't use syslog, you can use many tools to examine the generated data and forward that data to syslog. For example, to add memory statistics to the log data, you could use something like this at boot time:
>
> ```
> nohup vmstat 60 | logger -t vmstat \
>   2>&1 >/dev/null &
> ```
>
> (In most cases, you will have a separate metric collection and monitoring solution for such data, so there is little need to put that data into your logs as well.)

The processing includes fixing malformed log data, splitting long lines, escaping newlines, Unicode normalization, sanitizing data, adjusting timestamps, formatting numbers, signing (or generating a MAC), etc. Since the current syslog protocol doesn't include the source of log data, that should be added as well (the IP address). The log aggregators can queue log data, so when the network or loghost is busy no log data is lost.

The central loghost is so vital, it should be a pair of hosts configured for HA (*high availability*). The aggregators can securely send log data to each.

> Designing logging infrastructures (and monitoring solutions for that matter) are rarely done by system administrators. Instead, larger organizations hire system analysists to design their IT systems.

What happens to the log data once it reaches the loghost? It may get analyzed by several different applications, converted into events (which may then be fed into Reimann) and/or metrics (using logster for example). Also, the data needs to be

archived for long-term data retention (and be indexed for quick searches), the data may be watched by alerting tools (sec is one), the data may be fed to an IDS/IPS, (intrusion detection/prevention system), and finally, many different reports may be needed.

Basic Log Management Tools

The best log tools will store data from various logs, from many hosts, in a central database that can be examined and queried. The central host that collects the log data is called a *loghost*. This design has an advantage, in that if a host becomes corrupted and/or attacked, local log files will be lost. Also, a busy server won't need to waste CPU or memory analyzing log data locally.

Utilities include GUI tools to examine and mange log files, but since logs are text standard text processing tools are often used for single server or other small setups, such as `grep`, `tail`, `tail -f`, and `less` (for example: `grep` *`service logFile`* `|grep` *`date`* `|less`). (See also enterprise logging, below.)

Enterprise Log Management Tools

Searching for a particular error across many log files on many servers is difficult without good tools. Using shell tools such as `grep` are too painful to use once your systems grow to a moderate size. You need enterprise-grade tools that can store logs, normalize them, and allow sophisticated searches and analysis of them, quickly.

To effectively consolidate, manage, and analyze all the logs, one common solution is to use three tools together: Elasticsearch, Logstash, and Kibana, popularly known as ELK Stack (or today, Elastic Stack). **Elasticsearch** is a general data storage that permits full-text and other searches. **Logstash** is a log aggregator; it runs on the loghost to listen for incoming log data and store it into Elasticsearch (A popular alternative to logstash is Fluentd). **Kibana** is a web interface for searching, monitoring and visualizing the log data.

The other popular enterprise-grade logging system is Splunk. ELK is open source; Elastic Co. makes money selling support and consulting services (the "managed" solution.) Splunk is propriety; the cheapest managed plan (last time I looked) is $124/month for up to 1GiB of data, although they do have a free "trial" download to play and learn.

Both solutions provide on-premises and cloud-based products.

The ELK stack is fine for most situations, especially when you don't want to pay for a managed solution (at least, not at first). But it takes more effort for a system administrator to setup the logging infrastructure this way. Splunk may be much simpler to use, especially for complex situations. (For example, ELK currently (2017) doesn't support user management and access controls; anyone can read all

the data with no audit trail. Splunk does support that.) Splunk's dashboard supports more features than Kibana, but you may not need them.

> Naturally there are many solutions besides these two, including Loggly, Logscape, Logsene (uses ELK), Graylog, and others. Some products are actually services that you pay for (managed solutions), others are optimized for cloud-based clusters, and some (few) are designed for on-premises solutions. Over time products change and new ones are developed, so be sure to do your homework before deciding on solution.

Syslog Configuration

What is written to which file is controlled by ***`syslogd`***. The servers and other programs that produce loggable information (any status and error messages) should use `syslog` (a few servers don't use `syslog` by default, such as Apache). What is logged and where it goes is controlled by **`/etc/syslog.conf`**.

> What gets logged also depends on what a server (daemon) sends to syslog. Most services have configuration setting to increase or reduce the amount of log data then generate. There are two common setups: have syslog save everything and let the SA control the amount of log data by configuring each and every service (each service configuration file may use a different syntax), or have the services generate lots of logging data, and let the SA control what gets saved to log files by configuring only `syslog`. The first approach is less wasteful of CPU and RAM resources, but more demanding of the SA.

Here's how it works: A program uses the **`syslog` API function** (via a DLL; or uses the **`logger`** program for shell scripts) to send a log message to `syslogd`. `syslogd` will also read log messages from *sockets* (by default just **`/dev/log`**, but others can be used), and if started with the right option, also from the network.

The information passed to `syslogd` includes the source of the log message (called a ***facility***) and the ***priority*** of the log message. `Syslogd` then matches the source and priority against ***selectors*** (combinations of facilities and priorities) in its configuration file, and if a log message matches a selector(s), the message is sent to the corresponding destination(s). This is a primitive form of log message *filtering*, especially considering that syslog trusts programs to set the facility and priority accurately.

> Rsyslog, syslog-ng and other recent syslog replacements allow more sophisticated filtering. They use facilities, priorities, and arbitrary regular expressions, and provide sophisticated control over what data gets logged where, and the format of the log messages. The configuration files have therefore become much more complex. Although outside the scope of our

class, this is becoming important to learn and know. See www.rsyslog.com for a good guide and reference for rsyslog.

Note that **many PAM modules send log messages** to `syslogd`. Also, some systems use a separate log daemon for kernel messages, often called **`klogd`** that you may need to configure. (With Fedora, `klogd` just passes messages to `syslog` via the "`kern`" facility.)

Syslog.conf Syntax (also rsyslog.conf)

Aside from blank lines and comment lines, `syslog.conf` has rule lines, with two parts:

- The first part says what to log (that is, it is a filter called the *selector*)
- The second part says where it goes (called the ***action*** for some strange reason).

When a log message is processed by syslog, the message is compared with each selector in turn. If the selector matches the message, the action is done. So a given message may be handled by multiple actions, if multiple selectors match.

In many older syslog daemons, the selector and action ***must*** be separated with a TAB, and not just spaces!

Syslog Selectors: Facilities and Priorities

The source of a log message is referred to as a ***facility***. For example, any email related program that sends a log message uses (in theory) the `mail` facility, no matter what the name of the program actually is. When a daemon sends a log message to syslog, it includes the facility syslog should use. **Note that `syslog` trusts apps to use the correct facility when sending a log msg.** Since this is defined by the programmer, in some cases the facility may not be the one an SA would expect.

There is no way to define new facilities, but there are many predefined ones (up to 23 in all, depending on which syslog you use):

- **`auth`** (security events get logged with this)
- **`authpriv`** (user access messages use this)
- **`cron`** (for `cron`, `at`, and `anacron`, but not for the programs started by cron)
- **`daemon`** (other daemon programs without a facility of their own)
- **`kern`** (kernel messages)
- **`lpr`** (print system)
- **`mail`** (obvious)
- **`mark`** (used by `syslogd` to produce timestamps in log files)
- **`news`** (for netnews servers)

- **`syslog`**
- **`user`** (for user programs)
- **`uucp`** (obsolete form of networking)
- **`local0`** – **`local7`** (any system-defined use; RH uses `local7` for boot messages)
- (a wildcard, for all)

> Some *nix systems will have slightly different facilities, or use the same facilities for different purposes. FreeBSD has two additional facilities, `console` and `security`. They use `auth` and `authpriv` for the same purpose, but restrict security sensitive details to `authpriv` ("for messages like [those of] `auth` that should only be read by privileged individuals."

Due to the limited number of facilities available, it is inevitable that multiple services will wind up using the same facility for their log messages. Syslog allows programs to supply an identifying string, known as a ***tag***, that syslog will prepend to each line of the log messages. This tag usually identifies the daemon that created the log message. Doing this permits easy selection using `grep` or other tools to filter only the log messages of a particular program when several share the same facility.

The ***priority*** (or *level*) is one of the following eight levels, which are ranked in order from low to high priority:

- `debug` (or "`*`")
- `info`
- `notice`
- `warning`
- `err`
- `crit`
- `alert`
- `emerg`

When specifying a priority, that and all higher ones are selected too. (So "`debug`" is a wildcard meaning all priorities.)

The selector syntax is complex: One or more facilities (separated by commas), a dot, and then the priority. Some example selectors:

`mail.*`	mail facility, any priority
`mail.debug`	same as `mail.*`
`mail,news.*`	all messages from `mail` or `news`
`auth.err`	security messages with `err` or higher priority
`*.info`	all messages from any facility, except `debug` msgs
`*.=info`	any facility, `info` msgs only (and not higher)
`*.!err`	any facility, priority <= `err` only
`*.!=alert`	any facility, any priority except `alert`

`*.info;mail,news,authpriv.none`
all msgs with `info` or higher priority, except `mail`, `news`, `authpriv`

That last one is tricky. Using multiple selectors on a single line this way allows you to specify a general category first, then for the matching log messages you can specify exceptions. Always go from most general selector to most specific or your setup may not log what you think it should!

> A BSD security book I read has this warning: Using "`*`" with some systems means all facilities (or priorities), including "`none`"! On such hosts, the "`*`" priority would end up turning off all logs for the selected facilities.

Syslog Actions: files, users, pipes

Once a selector has filtered log messages, you still must decide what to do with them. Log messages don't only have to go to files, you can direct them to user terminals, run them through other programs (with a pipe to email, pager, or some IDS), or send them to another host running `syslogd`. (Handy if you have a network of computers you monitor.) Here's the syntax for the *actions*:

`/complete/path/of/some/file`
`/dev/console` This is the system console; can use any /dev device
`-/complete/path/of/some/file` (Don't flush file each time; better performance, risks loss of some log info.)
`username1[,username2 ...]`
`*` (all logged in users)
`@remotehost` (e.g., `@log.hcc.com`)
`|/path/to/named/pipe`

(The "`@host`" or "`@host:port`" format uses UDP. To use TCP, use "`@@`" instead (rsyslog feature). The default port is `514`. The remote `syslogd` must be

started with the "`-r`" to make it listen (UDP only); With `rsyslogd`, look for lines such as "`#$ModLoad imudp`" and "`#$ModLoad imtcp`", and uncomment those lines.)

To send output to a command, you must first create a *named pipe* (say `/foo/cmd.pipe`) with **`mkfifo`** command, then start the command with `cmd </foo/cmd.pipe`. Then in `syslog.conf`, specify `|/foo/cmd.pipe` as the action. Use this to trigger a pager, send an email, alert an IDS, etc.

(**Demo:** `mkfifo pipe;cat <pipe >fake.log&cal >pipe;rm pipe`) Using named pipes is handy with `chroot`, when not using syslog; you can configure a daemon to send log messages to the pipe inside the jail, while another process outside can read from that pipe to produce a log file. Using `syslog` is easier (put a `/dev/log` socket in the jail), but not all servers will support it.

To send log info to multiple destinations, use multiple lines having the same selector. For high security, attach a line printer to `/dev/lp0` and use that for important logs (such as security logs)—you know there is no chance of tampering with the log file this way!

Startup `syslogd` with the `-r` option to allow incoming log messages from remote `syslogd`s, making your host into a loghost. Use the `-m` *`mark-minutes`* option to have `syslog` produce `MARK` log messages every so often. Note that each log message has a timestamp in it, so on a busy system you may not need or want `MARK` log entries. Use a value of `0` in that case. Use `-a` *`socket`* to make `syslogd` listen to additional sockets other than **`/dev/log`**. This is needed when setting up services in a `chroot` environment, that don't use the syslog DLL.

> With modern BSD syslog, the rules can be grouped by hostname and/or by program name. This allows a loghost to handle logs from different hosts differently, or to handle log data from a given program with the same rule, regardless of which facility was used.

Rsyslog

Rsyslog is a popular, compatible replacement for syslog. In addition to accepting the same "rules" as syslog, rsyslog has many other features and advanced rules to address legacy syslog's shortcomings. The advanced features have default values, so normally you don't need to learn about them right away. However, there are some settings you may need to know about and change.

One setting to consider changing is rate-limiting. On Red Hat for example, rsyslog drops (or *rate-limits*) log messages if it sees too many in an interval of time: by default, messages are dropped if 200 are seen in 5 minutes. This is not a realistic setting in most cases, and can catch system admins unaware. (This happened to me once when trying to reconfigure `named`. Setting the log level to "`debug`"

meant `named` exceeded the rate-limit threshold, and the important log messages were dropped. This was difficult to diagnose.) You can tune this rate to your system, for example:

```
$SystemLogRateLimitInterval 10
$SystemLogRateLimitBurst 500
```

Alternatively, turn off rate limiting completely:

```
$SystemLogRateLimitInterval 0
```

`rsyslog` (and other syslog replacements) can send log messages to a DB. With rsyslog, you can create an appropriate DB "Syslog":
psql < /usr/share/doc/rsyslog-pgsql-3.21.11/createDB.sql

And create a user "`syslog`" with (say) password of "`secret`". Then add a rule to the `rsyslog.conf` file like this:

```
$ModLoad ompgsql
*.info  :ompgsql:127.0.0.1,Syslog,syslog,secret
```

Using logger

`logger [-p` *`facility.priority`*`] [-t` *`tag`*`]` *`message`*

The default selector is `user.info`, and the default tag is "`logger`".

You can also copy a file to the logs. Here's an example of copying `some-file` to the system logs:

```
logger -t "backup script" -f some-file #or <file, no -f
```

This will send all lines of *`some-file`* as individual log messages.

Boot-time Logging with Systemd's Journald

Since a number of tasks are done at boot time, before any system log daemon is started, such messages are traditionally printed to the console. But on modern hardware, there are so many messages, and they scroll by so quickly, it is easy to miss important information this way. Various *nix systems have created ad-hoc mechanisms for capturing these messages in a log file, with limited success. Instead, modern init system replacements include some logging support for this reason.

Systemd log messages are called the Systemd *journal*. You can view and manage that log using the Systemd command **`journalctl`**. A config file `journald.conf` determines what gets logged, and to which files, and many other things too. (For example, you can have `journald` forward log messages it gets to `syslog`, once `syslog` is started.) A good default is to save everything in a `boot.log` file, until the system is fully "up". (Of course, the Systemd folk

probably dream of replacing `syslog`, the audit and account logs, and any other logs completely, but that is unlikely.)

The daemon that handles the journal for Systemd is `journald`. That is started first by `systemd`, and even the early kernel messages can be handed by it. Daemons or other software can use a variety of APIs to send log data to `journald`: the kernel's `printk`, the standard `syslog()` function, or a new `journald` API.

Daemons pass data to `journald` as key/value pairs, and are stored internally in an indexed database. All messages are digitally signed and encrypted by `journald` upon receipt (I think). Currently, only root or members of the `adm` group can access (decrypted) system log messages; however any user can access non-system log messages (produced by some application program or daemon).

To view recent logs, run `journalctl`. To view only logs since the last boot, use the `-b` option. Due to the indexing done by `journald`, you can query the logs in powerful ways. For example:

```
journalctl -u httpd \
   --since -10m --until now
```

will show the boot entries generated by (or related to) the unit file `httpd.service`, in the last 10 minutes. (*Review the man page.*)

You can filter the journal by specifying matching rules for any field. For example:

```
journalctl -b -n 5 -p err _UID=48
```

will show the five most recent log entries, but only since the last boot, where the UID equals 48 (apache), and the message priority is err or higher. (See **`systemd.journal-fields(7)`** for a list of fields.) Bash completion should work for `journalctl`, including all legal field values. One more example: services such as cups ("`cups.service`") use the systemd journal instead of writing to their own files (or using syslog), since 2015 on Fedora. To view the cups log file, use "`journalctl _COMM=cupsd`". (Add "`-b`" to restrict that to messages since the last boot.)

`journalctl` also has options to export the journal in a variety of formats. It also has an **option ("`-x`") to display some help for each message**. (*See the tutorial link in the class resources.*)

Log File Rotation

One problem with log files is that over time they grow. When a system is experiencing problems the log files can grow very large, very quickly. Periodically trimming or removing log files is necessary. This is known as ***log file rotation*** and is a service usually run via cron.

There are several schemes possible. One is to use `tail` to save the last N lines of each log file, and discard the older log entries. Another scheme is to have several log files and use each file for some period of time, then switch to the next file in rotation. When the last file is used, start over with the first one again (discarding the old contents).

The most popular scheme is to rename a log file log as log.1 and to start a new log file. Next time, log.1 is renamed to log.2, log is renamed to log.1, and a new log file is started. This continues for *N* previous files. (After that, the oldest log files are deleted.) An even better scheme is similar, but use the date the file was rotated as the extension, rather than a simple number.

Instead of discarding old log entries, consider archiving them to some cheap backup media. You never can tell when old log records will come in handy. (But, be careful with privacy and security issues!)

Since dealing with log file rotation is a common problem most Unix systems have a standard way to deal with it. On Solaris 9, you have `/usr/sbin/logadm` (`/usr/lib64/newsyslog` on Solaris8). On Linux, you have the `logrotate` command. This command runs via the `cron` facility.

You can set your log rotation policy for any log file by editing the file **`logrotate.conf`**.

```
#Global                              /path/to/log/file {
daily                                  weekly
rotate 4                               create 0644 root root
errors root                            rotate 2
create                                 postrotate
#compress  # use zless to view        /usr/bin/killall -HUP syslogd
#per file setting:                     endscript
                                     }
```

The “rotate 4” directive means to keep four old (rotated) logs, in addition to the current one. You can configure `logrotate` to email to someone the old log files it would otherwise delete, handy for automatic archiving.

Important: When adding or enabling a new server, configure syslog and/or `logrotate` to manage its log messages! If possible, configure the service to use syslog (and not its own log files). Remember that `syslog` and `logrotate` are independent; even when not using `syslog`, you still need to configure log rotation for new daemons.

Consider always rotating logs on a weekly or monthly basis. This makes it much easier to guess which log file to examine when looking for an old event.

Note: With Debian systems, the `/etc/cron.daily/sysklogd` script reads the `syslog.conf` file and rotates any log files it finds configured there. This eliminates the need to use logrotate for the common system log files, but not for any daemon that doesn't use syslog.

> There is another important reason always to rotate your log files: the default syslog log file format timestamps do not include a year. If a system runs for longer than one year, tools such as `logwatch` will start reporting the old events again! **Always rotate all log files at least once per year.** If you can't easily configure your log monitoring tools to ignore old logs, they can be rotated into another directory that your log monitoring tools won't examine (such as `/var/archived-logs`), or archive them on a separate loghost. Another possibility is to encrypt old logs so that monitoring tools can't examine them (and neither can attackers).

Rotating Journald files

Journald stores the log files (the "journal") in files either in `/var/log/journal`, `/run/log` (a RAM disk), or both. The `journald` daemon keeps an eye on the amount of storage the logs take, and will trim them occasionally to prevent them from filling your filesystem (or running out of memory). The default is to grow them quite large, a cap of 4 GiB currently. You can control this by editing the `/etc/systemd/journald.conf` file, and restarting the journal daemon. To see space used, run "`journalctl --disk-usage`".

To manually rotate the logs, use the command "`journalctl --rotate`". After rotation, you can remove the old ("archived") journal log files with the command "`journalctl --vacuum*`"; the system will delete old rotated files according to your options. Frequent rotation provides a finer granularity for vacuuming.

Integrating Windows and Other Systems' Logs

Non-Unix/Linux systems also maintain log files, but usually not in syslog format. This may be a problem for the sys admin who must deal with a mix of Windows and *nix servers. Windows systems keep detailed *event logs*. Windows event log files are binary (not text like syslog). They are also fixed in size; when full, they erase themselves and start over, losing valuable data! (This policy can be changed from the control panel, and may not be the default in current Windows versions.) Although the logs are binary, the format is publicly available and a number of Perl and other tools exist to convert these to text.

Windows logs are consistent across all Windows versions and services (e.g., Event ID 529 always means a failed login). And since event logging is built into the OS, it is generally more secure than syslog.

Windows provides no mechanism to forward events to a central loghost. Instead, there are a number of third party tools for this, such as **Kiwi syslog for Windows**, **EventReporter**, **Snare for Windows**, and even roll-your-own with the Perl module `Win32::EventLog`.

The Windows event log is really 3 logs: the system log, the security log, and the application log. (Think of these as three syslog *facilities*.) Each log is stored in a separate file: `...\system32\conf\SysEvent.Evt`, `...\SecEvent.Evt`, and `...\AppEvent.Evt`. Applications must register themselves to be able to use the event log service (see registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\Application`).

System and service event logging is controlled by the *Windows Audit Policy* (Control Panel→Administrative Tools→Local Security Policy→Audit Policy).

Windows provides `logevent` (equivalent to the Unix/Linux `logger` command line tool) to create event log messages.

For older Macintosh systems (OS9 and earlier) you can use the syslog compatible `netlogger` tool. Modern Macintosh is built on BSD Unix, and thus supports `syslog` directly.

Using Logs

Because Unix/Linux logs are plain text, you can use text filters and scripts to process log files and produce custom reports. Most useful are `grep`, `awk`, `python`, and `perl`.

Example:

```
grep 'interesting' log-file |grep -v 'not-interesting'
```

By piping `grep` output through `cut` (or a similar filter), you can eliminate the timestamp and process ID columns; the result can then be sorted and/or counted:

```
grep 'Failed password' /var/log/secure \
 | cut -d' ' -f4,6-11 | sort | uniq -c | sort -n
```

Lecture 14 — DNS Overview

All hosts have IP addresses. These are hard to remember and type. IPv6 will extend the 4-byte IPv4 address to 16 bytes! A way is needed to assign easy to remember names. ***Every computer OS has a way to translate names to addresses, called the "resolver".*** The resolver is a library that any application can use. The resolver library found in all popular hosts is also called a "***stub resolver***", because it isn't capable of full ("recursive") DNS queries. Instead, hosts list the IP address of a DNS server that is capable of recursive queries. The stub resolver merely sends any DNS requests to the listed DNS server, waits for an answer, and sends that back to the application.

In the early days, you paid a small fee for your name (`foo.com`) and your application form included which IP address the name is for. Your name and number went into a large file **`HOSTS.TXT`** that every administrator was expected to download via FTP every month or so. By grepping for a name, the IP address could be easily found.

Soon there were too many names (and the list changed too fast) to make the `HOSTS.TXT` method practical. A global database used to translate names to numbers (and the reverse) was created. It is called the ***domain name system* (DNS)**. A server that provides the DNS service is called a ***name* (or *DNS*) *server***. (Often written as "NS".) DNS is a replicated and distributed database.

> Although DNS is by far the most popular way to translate names to addresses, other services can be used including LDAP and NIS+.

Small organizations can use a `HOSTS.TXT`-like scheme to maintain a list of local host names and their IP addresses, in a **`/etc/hosts`** file. Even if you use some DNS server, the `hosts` file is often used as well. For one thing, at boot time you may not be connected to the Internet. If you configured your mail or web servers to know their server name and that name isn't in the hosts file, the servers will waste about 4.5 minutes each waiting for a reply from a DNS server!

`/etc/nsswitch.conf`

This file configures how the resolver library (and other parts of the system) does lookups. The line that begins with `hosts` determines which mechanisms will be used for host name lookups, and in which order they are tried. For example, the line:

```
hosts:      files dns
```

Tells the resolver to first lookup names in the `/etc/hosts` file, then (if that fails) try DNS.

/etc/hosts

The `hosts` file format is not well documented but is easy to understand. Every line is one record and contains the following: **IP-Addr FQDN nickname ...** . The first column is the IP address. Note you can have several lines with the same IP address, if you want to provide several FQDN names to that same address. Here's an example: `127.0.0.1 hymie.piffl.com hymie hy`

Note that reverse lookups of a number to a FQDN vary by implementation: some return the first match, some the last, and some all.

> Some docs state you cannot have multiple lines with the same IP address. I do not believe that is correct, but you could try using "`127.0.0.2`" for your additional names if you are worried. Unfortunately, not all *nix system will resolve such addresses to the loopback device as they should.
>
> Some believe the second field should be the *canonical name* for that IP. In general, that is the same as the FQDN. But some docs state the canonical name for `127.0.0.1` should be "`localhost`" and not "`localhost.localdomain`". Because of this, you will find all sorts of different configurations out there. For example, Fedora reports "`localhost`" when using "`hostname -f`" and "`localhost.localdomain`" without the "`-f`" option. As system administrator, you can set that any way that makes sense to you; but you should have a good reason to change defaults of your distro (which will become confusing to others, and to yourself after some update has "restored" the defaults).

Hostnames

Before networking, you still wanted to name your servers. This name is called **the *host* (or *node*) *name***. Each name was associated with a server (which also has an unrelated and rarely used **`hostid`**).

With networking come complications: the host name was associated with a host's network address. No problem in the early days as each server had a single IP address assigned, so the DNS name still was used as the host or node name, rather than just a convenient way to refer to the IP address. Still it was possible to have a different host (node) name from the DNS name.

Soon servers started to get multiple NICs and thus multiple IP addresses. In the DNS system, each name corresponds to some address, not some server. A given server today can have many IP addresses assigned to it and each IP address can have many DNS names. So, which one is the host (node) name when you have a dozen names? When sending outgoing packets, which is the source name/address?

The situation is messy and there is no easy fix without breaking long-standing POSIX and Unix definitions. It is also worse because hosts can also have NIS and other domain names assigned as well, which have nothing to do with the DNS domain name.

Many systems default to having the administrator set the host (node) name manually (**`/etc/{hostname,nodename}`**). For others the official host (node) name is the first DNS name of the first IP address of the first NIC detected (that is, a reverse DNS lookup is made on `eth0`'s IP address to determine the hostname.)

DNS Domain names and FQDNs

The DNS host names have the form ***host.domain.tld***. The terms *host name* and *domain name* are often confused. The domain name doesn't refer to any specific host (e.g., `hccfl.edu`). The host name is the name for a specific host (e.g., `www`). A ***FQDN*** (Fully Qualified Domain Name) is the complete name for a host (e.g., `www.hccfl.edu`). (FQDNs technically end with a period: "`www.foo.com.`".)

Once you have a domain name you can add any host names to your domain at any time, or any *subdomains*. Common host names are `mail`, `ftp`, `firewall`, `ns`, `www`, etc. Note that it is allowed (and common) to have several names refer to the same IP address. This is called an ***alias name***. (Versus IP alias vs. IP masquerade)

The domain names are hierarchical (e.g., the domain `fl.us` has the parent domain `us`) with the subdomains separated by periods.

The rightmost domain has no parent (actually the real top is "`.`") and is called a ***top-level domain*** **or TLD**. The exact rules for domain names are spelled out in RFC1034. In brief, names are case-insensitive and must be composed of letters, digits, and hyphens. Each component must be 63 characters or less, and the FQDN must be 255 characters or less.

Your NS knows the IP addresses of the top-level name servers on the Internet (there are only 10-20 of these ***root servers***, see **`www.root-servers.org`**). These in turn know the IP addresses of all the **top-level domain (*TLD*)** name servers (e.g., the top server for the domain US). Those name servers knows the IP addresses of the next-level name servers (e.g., FL), and so on. Every domain has a primary name server that is the authoritative database for names and IP addresses for that domain.

Nameserver Lookups

The DNS system doesn't have a single machine containing all the records.

The typical lookup of a host's IP address given a name by the DNS is to see if it is a ***FQDN*** or just a partial name. If not a FQDN, the NS constructs one from the default or **`search`** domain listed in the config file (Qu: which file?). Next, it checks to see if it has cached this name previously. If so we are done.

If the FQDN is not in the cache, DNS checks the domain name to see if it is *authoritative* for this domain. If so, it looks up the name in its DB files. If not, it must ask some other DNS server on the Internet to look it up. If it has previously cached the IP address of a NS authoritative for any part of the domain, it can ask that NS to resolve the name for us. If not, the NS consults a *hints* file that gives the IP addresses of the dozen or so DNS servers that know the IP addresses of the NSs that are authoritative for the TLDs. (Say that three times fast.) (**Show www.root-servers.org.**)

Because of the way the lookup process works, a DNS server that does that is called a recursive resolver. Since these are DNS servers that also cache results, such servers are also called caching resolvers. (I prefer the slightly less confusing terms of: *resolver* for what you find on a host, *caching DNS server* for the server a host talks to, and *authoritative DNS server* for the servers that keep the official data records for some domain name.)

(**Demo resolving `www.wpollock.com`**: Not in cache, already a FQDN, so our NS finds the IP address of the NS for "`.`", then `com`, then asks that NS the IP of the NS for `wpollock.com`, than asks that NS for the IP of `www.wpollock.com`.)

If the lookup fails and additional search domains are listed, the next FQDN is tried.

Summary: Resolver-only: `hosts`, `resolv.conf`, `nsswitch.conf` (`host.conf` for legacy s/w). Some additional files are optional: `networks` , `ethers`, `hostname` (Debian), `nodename` (Solaris), `sysconfig/network` (Red Hat), `HOSTNAME` (old). See also `libbind-hostname(7)`.

WHOIS and RDAP

To see who owns what domains and who to contact to register names within those domains, you search the ***WHOIS* database**. The WHOIS database (and list of accredited registrars) is at **`www.internic.net`** (the official DNS website). Actually, the database is only informally called WHOIS, which is really just the name of the protocol used to access registration data.

The DNS database can be searched with various commands: **`nslookup www.redhat.com`**, or other commands:

```
dig [any] wpollock.com
```

`host [-a] wpollock.com`

All records = *zone xfer*: **`dig @ns1.hcc-online.com -t AXFR wpollock.com`**. Note you must tell `dig` which server to query as only authoritative servers will have the information. Also, that server must be configured to allow zone transfers.

Show **`whois`** command: **`whois hccfl.edu; whois wpollock.com.`**

(Demo `www.uwhois.com` search for `wpollock.com`.)

> `nslookup` is no longer maintained and doesn't fully work. Use `dig` or `host` instead.

WHOIS the protocol was meant for human use and is difficult for applications to use. Additionally, issues with WHOIS cause it to be abused frequently.

A new protocol called RDAP (Registration Data Access Protocol), is in use since 2017 and the WHOIS protocol no longer the official protocol that registrars must support (as of 2025). You can find RDAP client programs from icann-rdap GitHub repository and OpenRDAP. You can also find an online lookup tool at RDAQP.org. (You will find more RDAP info at these URLs too.)

DNS Records

Beside containing name (**A**) records, the DNS database contains *alias* (**CNAME**) records, number-to-name (**PTR**) records (for *reverse DNS* lookups), domain contact information (**SOA**) records contain the *hostmaster* email address with a dot used instead of the "@"), nameserver (**NS**) records, and mail exchanger (**MX**) records (used to direct email to a specific host: user@wpollock.com --> user@mail.wpollock.com). Also **TXT** (used with SPF) and **AAAA** (IPv6) records.

> It is not simple to find all DNS records of a given type for some domain's zone. The easiest solution is to do a *zone transfer* and use `grep` for the records of interest:
>
> ```
> dig @ns1.hcc-online.com wpollock.com AXFR |grep CNAME
> ```

In addition to the standard domain names, the special domain **`in-addr.arpa`** (and `ip6.arpa`) is used to support **reverse DNS lookups**, where you have the IP address and want the name. (MTAs use this to detect spam mail: If the host claims to be mail.foo.com but the reverse DNS lookup on their IP address is fake.bar.com then the email is dropped.) For example, if the host `www.foo.com` has the IP address of `200.32.40.6`, then the normal record is

```
www.foo.com. IN A 200.32.40.6
```

and the reverse record is

```
6.40.32.200.in-addr.arpa. IN PTR www.foo.com.
```

Nameservers

Each registered domain lists one or more ***authoritative*** **nameservers** that maintain all records for that domain. If any subdomains exist, each usually has its own authoritative nameserver(s). There is a record pointing to the authoritative nameserver for each *delegated* subdomain in that case. However, a single nameserver may contain records for many domains. The records for which a nameserver is authoritative is call its ***zone*** (a portion of the hierarchy, excluding any delegated domains).

Servers occasionally go down (even Unix servers). If you have only one authoritative nameserver and it fails, no one on the Internet can access your hosts by name. Most organizations use additional authoritative nameservers for redundancy and possibly load balancing. Thus, you may have one **primary nameserver** and one or more **secondary nameservers**. As the ***hostmaster,*** you manually maintain the records on the primary NS. A *zone transfer* will update the secondary NSs every so often.

In addition, you may have non-authoritative nameservers. These can be used for caching DNS data gathered from the Internet, for quick access by the hosts in your organization (and are called ***caching-only nameservers***.). You may also use a proxy DNS server to enhance security. Don't forget to setup your firewall rules to allow authorized DNS access. (This is UDP for lookups and TCP for zone transfers.) Don't forget that the outside world needs access to an authoritative DNS server.

The SA who manages the DNS related tasks is known as the ***hostmaster***. This is usually an email alias for `root`.

DNS has proven very effective and efficient in the world. Microsoft has a competing scheme called WINS that does the same task as DNS but for LANs (NetBIOS names) only. In addition, some servers now have dynamic IP addresses assigned, which requires that the DNS records be updated frequently. A system called *dDNS* (dynamic DNS) is available for this. It is possible to use multiple services; a common setup is to use both the hosts file and DNS.

Nameserver Configuration

This depends on what nameserver software you use. By far the most common is the BIND `named` server. BIND also supplies the `nslookup`, `dig`, `whois`, and other utilities.

If you set up an authoritative DNS server, you can verify its records using these services: **`www.dnsstuff.com`** or **`www.dnsreport.com`**. (**Show**.)

`nscd`

This is a caching name server, which is much simpler than a full DNS server is. It caches all resolver info (anything that goes through `nsswitch.conf`) to make lookups quick. However, any changes to name services aren't picked up automatically by `nscd`, so you have to restart this manually after any changes. Show: **`/etc/init.d/nscd start`**. Note this can be configured not to cache everything, and the cache retention time (TTL) can be adjusted as well.

BIND (named)

The most popular DNS server software by far is called **BIND**. You can use BIND to set up a primary, secondary, or caching-only nameserver (called **`named`**). The **`/etc/named.conf`** file configures `named` for its type, where to find its zone records for all domains for which it is authoritative, and other information. To run a DNS server, you create the records in the proper format and put in the proper files, edit the `named.conf` file, update the firewall rules, and finally arrange for named to run at boot time.

> BIND 10, released in 2013, was a complete re-write, with different utilities, daemons, and configuration files. The information here applies to BIND v9 (and earlier to some extent). Note that Fedora still ships with BIND 9.

The named server is controlled by **`rndc`** (*remote name daemon control*), which cannot only start/stop/restart/reload a server, but also flush caches, emit statistics, refresh zones, and more. (Run `rndc` with no arguments to see a list of commands.)

Using `rndc` to control multiple servers is handy but can be dangerous to allow since attackers can send fake commands. To support this utility safely `named` listens for `rndc` commands over TCP/953 and uses HMAC-MD5 to authenticate the commands (i.e., the commands plus a shared secret key are hashed with MD5 and send with the command; the remote `named` instance computes the same hash and compares the two).

The `rdnc.conf` file can be generated using `rdnc-confgen` utility. This puts the shared secret key in `rdnc.key`. That file must be (securely) copied to each DNS server, and included in `named.conf`.

The file **`/etc/named.conf`** is used to configure your nameserver. (Older versions of BIND used the file `/etc/named.boot`.) This file tells **`named`** to be a *primary*, *secondary*, or *caching-only* nameserver.

If caching-only, this file says to which nameserver to forward the DNS requests.

For a primary nameserver the names and locations of the DNS DB (the ***zone records***) files must be given.

For a secondary, the IP address of the primary nameserver must also be supplied.

Note that a given nameserver may be primary for some zones and secondary for others. (All nameservers will cache in RAM the results of any previous lookup. Every nameserver should be a primary NS for `localhost` (if DNS only and no files are used), and the `0.0.127.in-addr.arpa` domain (in any case).

The syntax for this file is similar to C programs: whitespace is not significant (put blanks and tabs and newlines almost anywhere), "`//`" and "`/* ... */`" comments are allowed, and the *directives* go in *sections* delimited by curly braces ("`{}`"). Each directive ends with a semicolon.

(Show `named.conf` samples. Explain setup for each type of nameserver.)

```
options {   directory "/var/named"; };
zone "." {  type hint; file "root.cache"; };
zone "hcc.com" { type master; file
"db.hcc.com";
};
zone "35.168.192.in-addr.arpa" {
        type master; file "db.192.168.35"; };
zone "0.0.127.in-addr.arpa" {
        type master; file "db.127.0.0"; };
```

The DNS records are kept in files (zone files) so named can reload the data after a restart. These are generally stored in `/var/named`.

The `named.conf` file has many options and directives, and different OSes ship with different defaults for this file. In Fedora for example **`named` only will serve requests from localhost by default**. Be sure to carefully check all options, logging (which must be configured separately if you use syslog), and `rndc.key` and `rndc.conf` files.

On Fedora 9, the bind package creates `/var/named` with incorrect permissions. Fix by running `chmod g+w /var/named`.

Logging with Bind v9

Log messages from named are generated in various *categories* and with various *severities*. (Similar to syslog facilities and priorities.) To control where log messages go, you define *channels* and state to which channels messages from a given category are sent. Some channels are pre-defined. In particular, use the channel *null* to discard messages of some category.

A channel also lists a severity. Only messages of that severity or higher will be sent through that channel. In addition, you can tweak the output of the messages a bit, to include a timestamp, the category, and the severity. Finally, you define where each channel sends its log messages: to a file or to some syslog facility.

In theory with bind v9, the categories of `default` and `general` should include all messages generated. However, you may need to list categories explicitly:

`client`, `config`, `database`, `dnssec`, `lame-servers`, `network`, `notify`, `queries`, `resolver`, `security`, `update`, `update-security`, `xfer-in`, and `xfer-out`.

Here's a sample:

```
logging {
        channel main {
                syslog local2;
                severity info;
                print-category yes;
                print-severity yes;
        };
        category default { main; };
        category general { main; };
//      category queries { main; };
};
```

DNS Research

It's often the case that a log file shows an IP address and you want more information. First you can run whois to find out the organization that owns the block containing that address. The whois record may show as range of IP addresses, but you need to double-check that as they are sometime incorrect or incomplete. The proper way to check is to get the organization's AS number, then look up all the IP ranges included in that.

For example, the daily log report showed a serious hacker attempt on YborStudent from IP `216.118.240.194`. Using `whois`, I see this is a Hong Kong company. The range of IP addresses in this block are shown as

`216.118.231.255 - 216.118.255.255`, which is equivalent to `216.118.231.255/19`. A double-check with `whois` on that shows no such range!

The next step was to get the AS number from this IP address. Going to https://ipinfo.io/ I searched for the original IP address and found the AS number: `AS45753`. Looking up that shows all IP address blocks assigned (494 of them!). Scrolling down, I find the block `216.118.224.0/19`. The specific IP address does not appear to be from any hosted server. (I just added the whole block to my firewall rules.)

Lecture 15 —Email Infrastructure and Web Services

Review Email Infrastructure resource. Need to know acronyms, concepts, and protocols. (Email projects will be given in the Networking class in the Fall term.)

A ***web server*** accepts requests from clients (known as *web browsers*) for specific documents, often called *web pages*. Usually these are text documents with **HTML** (*HyperText Markup Language*) formatting, but may be any type of document. In addition, a web server can generate documents dynamically, as the result of running separate (external) programs, or performing database lookups.

The requests include a **URL** (*Uniform Resource Locator*), which uniquely identifies a document on the Internet. A URL is actually a type of **URI** (*Uniform Resource Identifier*), which in turn is a type of IRI (*International Resource Identifier*; mention *punycode*).

A URL has several parts, including a *protocol* (such as `http://`, `https://`, `ftp://`, etc.; the two slashes indicate a hierarchical scheme such as a pathname, versus `mailto:` or `javascript:`), a *web server* (either an IP address or a DNS name such as `www.example.com`, `mail.example.com`, etc.), an optional *port number* (the default depends on the protocol used: "`:80`" for HTTP, "`:443`" for HTTPS), a *pathname*, and optional other data ("`#`*`fragment`*" and "`?`*`query`*"). A typical example might be `http://wpollock.example.com/somedocument.html`.

A URL can also point to a directory rather than a document, by ending in a slash. (If you forget the slash, many web browsers will retry the request adding a slash. In this case, it is up to the web server to determine what document to return to the client. Some possibilities include a (nicely formatted) directory listing, an error message, or some default document. For Apache, a default document is named "**`index.html`**" (or some variation such as "`index.htm`" or "`index.php`").

A default web page for the top directory of the web server is called the server's ***homepage***. This is the page you get with a URL similar to "`http://servername/`".) By default, Apache ships with a default "test" homepage. You will probably need to change that!

The requests and responses are sent via TCP using **HTTP (*HyperText Transmission Protocol*)**. The current version is HTTP/2 but HTTP/1.1 is still commonly used (and is the default for Apache web server). A new UDP-based version known as QUIC is in development as of 2021.

In addition to required headers, a request packet may include **form data** that the user entered on some web page that allows user input (e.g., a form that includes a submit button). The response from the web server will include some headers and the document's contents.

The most important header in the response is the **`Status`** header, which is a 3-digit number. **A value of 200 means no problems, 403 means permission denied (usually because you set the wrong permissions), 404 means document not found**, and so on.

> To troubleshoot HTTP and web server configuration problems, use the Firefox extensions called "Live HTTP Headers" or "Modify Headers".

A web server talks to web clients (browsers) using one of several protocols: HTTP (RFC-2616), HTTPS, or SPDY (see the box below). The server can serve documents, including dynamically generated content. The protocol allows seven commands, but the most commonly used ones are GET and POST. (The others are OPTIONS, HEAD, PUT, DELETE, TRACE, and CONNECT.)

> Google introduced SPDY in 2009, as an experiment to produce a faster alternative to HTTP. It is able to handle many concurrent streams over one connection, while maintaining HTTP-like semantics. SPDY runs over TCP and all communications are TLS-encrypted and gzip compressed, to ensure security and improve transfer speed. According to Google, in practical testing, SPDY can as much as double the effective speed of connections to web servers. By 2014, all major browsers and web servers supported SPDY.
>
> In 2015, Google announced it was removing SPDY from Google Chrome, and other browsers followed suit. Why? Because the IETF, using ideas learned from SPDY, created HTTP/2 which seems to have wide industry support. An HTTP/3 draft called QUIC is out too (2020), using UDP instead of TCP.

Besides the basic web service, you will often need PHP, some wiki or blogging software, and probably (depending on your site and users' abilities) content management software (CMS).

PHP is a popular and powerful server-side scripting language. The default PHP module uses CGI (*common gateway interface*) with Nginx, or runs inside of `httpd` as an Apache module. You can also try PHP-fpm, which uses the newer FastCGI interface. (There are other interfaces possible too.)

> **Note:** The default Linux build of Apache no longer uses the "*pre-fork*" MPM (Multi-Processing Module) but rather the better "*event*" MPM. This matters because `mod_php` only works with the pre-fork MPM.

> The default config is to use `mod_proxy_fcgi` to send PHP requests to PHP-fpm, which acts as a fastCGI server connecting to your actual PHP executable. PHP-fpm is a separate daemon and has a separate config file, `/etc/php-fpm.conf`. There is a dependency for PHP-fpm in the httpd unit file, so under systemd PHP-fpm is started automatically when httpd is. See Remi's blog for more info.

Wiki software allows easy sharing and collaboration. The most popular example is Wikipedia from Wikimedia. This software is FOSS and used for the class' UnixWiki site. Most wiki software requires PHP and some database. Docuwiki doesn't use a database; it stores everything in flat files. Blogging software is similar, such as WordPress.

A **CMS (*content management system*)** has a simple (or complex) web interface allowing users to add and create content to a web site. It allows management of content as well, including setting publication dates in the future, or expiration dates. Mostly the content is stored in some database.

> HCC's web server used the Novus CMS. This CMS "sanitizes" any content uploaded by faculty, stripping out any JavaScript, CSS, PHP, and most other "dangerous" content. So I couldn't use it, and faculty didn't get any superuser access. So HCC restored the older web server at `content.hccfl.edu`, and that continued to host just my web site for some years.

Apache Web Server

Apache httpd is the world's most popular web server for decades. (NginX has closed in, and there are other servers available these days.) Windows and other versions are available, but these may be limited in what features are provided (due to limitations of the operating system). The Apache foundation (apache.org) hosts many projects besides a web server (httpd.apache.org), but when we say "Apache", we generally mean the `httpd` web server.

Apache's httpd web server is the de facto choice. It is powerful, customizable, secure, fast, free, and very configurable. By default, it is robust, pre-forking processes instead of using multi-threading; apparently this is because some modules won't work well as threads. Either way, Apache is a memory hog.

The nature of httpd is changed by using different MPMs, or *Multi-Processing Modules*. There are three: prefork, worker, and event. Worker MPM generates multiple child processes similar to prefork. Each child process runs many threads. Each thread handles one connection at a time. Worker MPM is much faster and uses less resources than prefork. But some popular libraries

used with the server are non-thread safe, so the worker MPM won't, er, work. Each process under the event MPM also can contain multiple threads but, unlike worker MPM, each thread is capable of more than one task. Apache has the lowest resource requirements when used with the Event MPM. Like the worker MPM, some non-thread safe libraries won't work under this. This is way prefork MPM is the default.

> The Apache Foundation hosts many FOSS projects, not just httpd, at apache.org. By far the best known is **httpd**, and is usually what is meant by "Apache".

There are two major versions of Apache. Version 1 has a reputation for rock-solid performance and is still used for that reason. Version 2 is better, only it hasn't been used for as long. Version 2 is highly modular. This allows you to easily add or remove functionality as needed for your system, including modules for more recent protocols and security. Even the configuration file is modular.

Other Web Servers

Apache is not the only web server, however. Another popular choice is Nginx ("Engine-X"). This is much smaller (uses less RAM) and much faster, but much more limited web server than `httpd`. There are other web servers available, but none with the reputation of these two. Nginx is used by Hulu, SourceForge.net, and other big sites. According to Netcraft.com, as of 2012, Apache had 65% of the market, IIS has 14%, and Nginx has 10%. By 2016, Apache fell to 50%, IIS fell to 10%, and Nginx rose to 16%. (Active web sites only.) In 2020, NginX holds 37%, Apache 25%, and IIS 11%.

Apache Configuration

To set up the Apache web server is easy, as the default configuration just works.

Step one: install then turn on Apache. Do a "`ps -ef | grep [h]ttpd`". Qu: why so many processes? Ans: it takes time to start a process, so a "listener" `httpd` process starts up a bunch of spares ("workers"). When a HTTP request is received, the listener process hands it off to one of the spares.

View the server's *homepage*. Qu: what is the URL for that? Ans: `http://`*`hostname-or-IP/`*. If this works you should see a "test" page. If not, you need to examine the `/var/log/httpd/error.log` file to see the error messages.

Advanced Apache configuration is a whole course in itself. Apache include support for virtual web sites, address/URL rewriting, per directory access control, content and language negotiation with the web browser, and many

other features. Here we just cover basic Apache version 2 (Version 1 is still used in some places). You should probably bookmark the Apache manual.

Most of the configuration is listed in the file `/etc/httpd/conf/httpd.conf`. However that file has an "`Include conf.d/*.conf`" to include the snippets in those files, as if they were part of the main file. There are other Includes as well, depending on your distro.

To add new features, install modules (you can use **`apxs`** to compile new ones), and then *load* them by adding a line to a configuration file such as `httpd.conf`. Adding modules enables various *directives* to be used elsewhere in your configuration file(s). (These can be conditionally included by testing if the module is loaded or not.)

> Modern Fedora no longer loads modules in the main config file, `httpd.conf`. Instead, that file uses an `Include` directive to load modules from a directory `/etc/httpd/conf.modules.d`. This technique is common in Linux, and makes it easier for packages to add/remove modules without trying to edit any file such as `httpd.conf`.

The configuration consists of *directives*. These are applied to the whole web server (such as the **`User`** and **`Group`** to run `httpd` as), specific directories, filenames, or URLs. Ones that apply to the whole web server are called ***global directives***.

Containers

You can surround non-global directives with containers like this:

```
<Files "*.txt">
  directives that only apply to .txt files go here
</Files>
```

(Notices how similar that looks to HTML tags.) Such containers limit the effect of the directives within. In this example, only to files with a ".txt" extension.

The tags and directive names are generally not case-sensitive. Regular expression matching of filenames, directory names, and URLs ("location") is also supported.

Directives can also be applied to just one virtual web site. Directives in a virtual web site container will override global ones, and directives in file, directory, or location containers will override those.

To have a set of directives apply to a certain directory and sub-directories, use:

```
<Directory "path">
   directives go here
</Directory>
```

"*path*" can be absolute, relative (to the document root), or even a regular expression. Instead of "`Directory`", you can use "`Location`" to match URLs, usually relative to document root. But the URL doesn't have to match any real directory at all:

```
<Location "/help">
   directives go here
</Location>
```

You can use the "Files" container this way (a regular expression) to apply directives to all files that start with "`.ht`":

```
<Files "^\.ht">
   directives go here
</File>
```

Directives

Each Apache *module* provides some directives. For example, to use any of the SSL directives (to enable and configure HTTPS), you must load the `ssl_module` first. You can easily tune Apache by not loading modules you don't need. Apache includes many, many optional modules you can emable if you want to use those features, or leave out to speed up the server.

Apache has directives to: format logging, indexing, multi-language support, custom error messages, security, and a whole lot more.

> The documentation for Apache is well organized. You can lookup directives alphabetically, by type, or by module. There are plenty of examples, and the config file(s) include many comments to explain.

To start with, go through `httpd.conf` and examine the global directive defaults. Some of the ones I generally consider changing are:

```
ServerTokens Prod
MaxSpare{Servers,Threads}
ExtendedStatus On
ServerAdmin webmaster@localhost    set email alias for this
ServerName
DocumentRoot    /var/www/html is common
DirectoryIndex  add index.htm, index.php, ...
HostnameLookups Off
ServerSignature EMail
```

A few other directives worth noting:

```
ScriptAlias "/cgi-bin/" "/var/www/cgi-bin/"
```

```
Alias /other "/path/to/other/files"
```

```
DirectoryIndex index.html index.htm index.php
    index.rhtml index.cgi
```

```
LoadModule http2_module modules/mod_http2.so
```

```
Protocols h2 h2c HTTP/1.1
```
(*"h2c" is HTTP/2; "h2" is HTTPS/2*)

You can run a command to check the syntax of your configuration; on current versions, that is "`httpd -t`". If Apache fails to start, disable files in `conf.d` one at a time until it starts. (I often have a problem when a DB is not configured correctly, for say SquirrelMail, RT3, Wiki, etc.) Often the error messages in httpd's log, or the output of `journalctl` can provide a clue what caused the failure.

> There is a Red Hat GUI for configuring Apache, however it will overwrite any manual changes you make to the file(s).

Apache include special directives for security. For any file, directory, or location, **you can limit access in many ways**: limit HTTP methods, limit source IP addresses, even password protection. These (and some other) directives can also be placed in special per-directory files (default name is "**`.htaccess`**"). This allows webauthors to tweak access without providing them access to `httpd.conf` (that's how GoDaddy does it), however using this feature greatly slows Apache down. Also, not all directives are legal in `.htaccess` files.

Suppose you want to limit access to some URL, for example "`http://localhost/foo/`". To do so you need to add directives (of course) to a container for that URL (or the equivalent directory). You can use directives that limit access based on many things, such as the source IP address or type of request. In this example, we will restrict access to authorized users only, who will need to supply the correct username and password.

> This sort of access is dangerous unless using HTTPS! Setting up TLS security must wait until the Unix/Linux Security class. Additionally, beware of symlinks, or URLs containing "../", as those may be attempts to bypass your security. The directions shown here are over-simplified.

```
<Location "/foo">
   AuthType Basic
   AuthName "Foo"
   AuthUserFile "/var/www/passwords-foo"
```

```
    Require valid-user
</Location>
```

The details of these directives can be found in the Apache Auth How-to. In short: use passwords, name this security "realm" (different URLs can use the same realm) to "Foo", specify the location of the username and password file, and require any valid user to authenticate.

To create the password file in the right format, use **htpasswd(1)** command:

```
htpasswd -Bc /var/www/passwords-foo auser
htpasswd -B /var/www/passwords-foo buser
```

(The "-c" option creates a new file, deleting any existing one. The "-B" option says to use bcrypt rather than MD5, not that it matters for this demo.)

Note the resulting file is outside of the document root, but must still be readable by Apache.

Be careful of the security of your configuration. For example, enabling mod_status is useful, but should have restricted access. This module provides the /server-status page to show system load information. The output includes currently active connections showing URLs and IP addresses of users. To restrict to our classroom network (IPv4) only:

```
<Location "/server-status">
   Require ip 10.142.255
</Location>
```

Apache can be configured to serve up multiple ("virtual") websites. This can be done by name (only requires a single IP address, with many DNS names) or by IP address. Using SSL (HTTPS) requires using one IP address per virtual website.

Apache Virtual Hosts

Apache supports a concept of ***virtual hosts*****. A single physical host can appear to clients as multiple web servers.** This can be done with one of two methods:

Assign multiple IP addresses to the NIC(s) of your hosts. Then add:

```
<VirtualHost IP-Address>
    ServerAdmin webmaster@vhost1.example.com
    DocumentRoot /www/vhost1.example.com/html
    ServerName vhost1.example.com
    ErrorLog logs/vhost1.example.com-error_log
    CustomLog logs/vhost1.example.com-access_log
common
```

```
    ...
</VirtualHost>
```

For each host you want to support. You can use DNS names instead of IP addresses, but that could be a bad idea if Apache can't quickly resolve the names.

The HTTP 1.1 protocol added some new required headers to the request packet. Once of these contains the name (not IP address) of the web server the request is for. This allows ***named virtual hosts***, where you have multiple DNS names resolve to the same IP address (using `CNAME` DNS records to provide *IP Aliasing*). This economy of IP address space can be very useful when you need more virtual hosts than you have public IP addresses available, and simplifies firewall configuration.

To use named virtual hosts, first add the directive "**`NameVirtualHost *:80`**". Then add the `<VirtualHost` *`name`*`>` directives, using the DNS names rather than an IP address. Note, if you use wildcards in the VirtualHost directive, ties are broken by Apache using the ServerName defined within that virtual host.

> Named virtual hosts can't be used with HTTPS, which does a reverse DNS lookup to check the name and IP address match. In this case, you must use IP based virtual hosts.

Web Server Content

Add static content to the document root directory. If using virtual servers or different content for HTTPS and HTTP, you will have multiple document roots.

Make sure your content is readable, and directories are executable, by the apache user. If you wish FTP style index pages, directories must be readable as well.

It is common to set the document root as: owner `root`, group `www` (for your web authors), and set mode `rwxrwsr-x` (note the SetGID). Don't forget to create the `www` group, and add your web developers as members. (When installing Apache it generally creates a user and group named `apache`, but I prefer using the shorter "`www`" which can be useful if you switch web servers.)

Dynamic content comes from PHP or other scripts running on the server. Sometimes a web page will get content from a database or even another web server, to incorporate into the web page you see from a URL. For example, injecting ads. Other code in dynamic web pages collects user-tracking data (known as *real user monitoring*, or RUM) for marketing. One simple technique is to have a cron job update a GIF or other file(s) inside of the

document root, so when someone fetches that static file, they get updated information.

Red Hat versus Debian Apache Configuration

The default configuration for Apache depends on who wrote the package you install. For Red Hat, each virtual host (and each feature) has a config file in .../conf.d, named *something*.conf. An easy way to enable sites and features is just to rename the file to *something*.conf-OFF.

On Debian based distros, everything is different, starting with the package name: apache2. The main config file is /etc/papche2/apache2.conf, not httpd.conf. The available config files are found in /etc/apache2/conf-available/, not conf.d/. Rather than rename these to *something*.conf-OFF, a directory of symlinks is used, named /etc/apache2/conf-enabled/. To enable/disable some config, you create/remove a symlink.

Likewise, instead of conf.modules.d there are two directories, mods-available and mods-enabled, the latter holding symlinks to the former.

Finally, there are some other config files with no matching Red Hat file: envvars contains Apache environment variables, ports.conf (which ports to listen on), and the two directories for virtual servers: sites-available and sites-enabled. (The default document root, when not using virtual servers, is sites-available/000-default.conf.) The apache2 package includes some command line utilities to make this easier, such as a2ensite to add a new virtual server.

Nginx Configuration

While fast, Apache is designed for flexibility. On the other hand, Nginx is designed for speed. It does handle most standard web server tasks, including webmail.

Some sites use Nginx as a proxy (or front-end) for Apache. Using a cluster of Nginx webservers, outfitted with Memcached distributed memory cache, allows very short response times and many concurrent sessions. More complex queries (say for dynamic content) are forwarded to the Apache server(s).

On Fedora installing Nginx is simple: yum install nginx. Then start it. The default configuration serves content from /usr/share/nginx/html. You can edit the configuration file in /etc/nginx/nginx.conf. The syntax is:

```
section-name {
    directive;
    ...
}
...
```

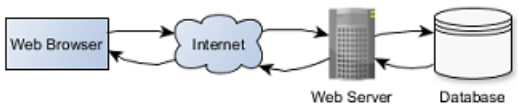
Sections can be nested. Any directives outside of any section apply universally (like Apache). Generally, you have one `http` section, containing one `server` section per virtual web server. Add directives to specify the document root in there:

```
user www www;
worker_processes 5;
# debug, info, notice, warn, error, or crit:
error_log logs/error.log info;
root /var/www/content;
http {
  include conf/mime.types;
  index index.html index.htm index.php;
  server {
    listen 80;
    server_name www.example.com;
    location /foo {
      ... other directives go here ...
    }
  }
}
```

Like Apache, the main Nginx conf file can `include` other files.

Web Service Architecture

In the past, a single web (HTTP and/or HTTPS) server was used for all requests. That one server needed to handle all sorts of tasks, such as inserting ads in the page, translations to other languages, etc. Soon a single server gets overwhelmed when there are too many requests. (Simple websites don't suffer as much and can handle a much larger load.) Here's a diagram:



Content Delivery Networks (CDNs)

To keep up with the growth in the number of clients, there has been a move towards architectures that scale better using replication, distribution, and caching. Also, content providers have also begun to deploy geographically diverse ***content delivery (or distribution) networks*** (**CDN**s) that bring origin-

servers closer to the "edge" of the network where clients are attached. (CDNs can better resist DoS attacks too.) The largest of these is Akamai, which serves most of the major ecommerce sites. It has been estimated (2012) that 70% of the most popular 1,000 websites use CDNs (NU study).

CDNs work by having ISPs' DNS servers return the IP address for some host name such as `google.com` to the geographically closest server. CDNs do have some issues with public DNS (since your location info is not used), and with TLS (since the name associated with the IP address used doesn't match the IP address of the "real" server. These problems have mostly been addressed.

On the content provider side, *replication* and *load-balancing* techniques allow the burden of client requests to be spread out over a myriad of servers. One way to implement this is to use a cluster of web servers, and use cookies to track sessions.

Another method is to use a *screening* (not a standard term) web server that accepts the single client request, and makes request to back-end servers to handle specific tasks. For example, it is common to configure Apache to forward requests for Java applications to Tomcat or Jetty application servers.

> A caching proxy server such as `squid` is a ***forward proxy***. It is installed on the client side of the network, caching data from many servers to (a few) clients.
>
> It is possible to install a caching proxy server at the server end too. This is called a ***reverse proxy***, which acts as a screening (and possibly load-balancing) web server. A reverse proxy caches data from a few servers to many clients. (Either static data, or dynamic data that is allowed to be some seconds or minutes old.) Such reverse proxies are often called *HTTPD accelerators*.
>
> Varnish is a modern reverse proxy server used as Facebook, Slashdot, and many other sites to increase access speed greatly (by clients), and to reduce load on DB and application servers by caching content. Apache Traffic is another.

No matter which solution you use, the result is "ad-hoc"; there is no standard way to implement complex web services... until recently (2015) that is. REC-3507 defines the ***Internet Content Adaptation Protocol (ICAP)***. ICAP is a lightweight protocol for executing a "remote procedure call" on HTTP messages. It allows ICAP clients to pass HTTP messages to ICAP servers for some sort of transformation or other processing ("adaptation"). The server executes its transformation service on messages and sends back responses to the client, usually with modified messages. The adapted

messages are either HTTP requests or HTTP responses. The ICAP client is typically a proxy web server such as Squid.

For example, a content provider might want to provide a popular web page with a different advertisement every time the page is viewed. Currently, content providers implement this policy by marking such pages as non-cacheable and tracking user cookies. Using an ICAP architecture, the page could be cached once near the edges of the network. These edge caches can then use an ICAP call to a nearby ad-insertion server every time the page is served to a client.

As another example, consider a user attempting to download an executable program. These requests are almost always handled via a caching proxy web server. The proxy server, acting as an ICAP client, can ask an external server to check the executable for viruses before accepting it into its cache.

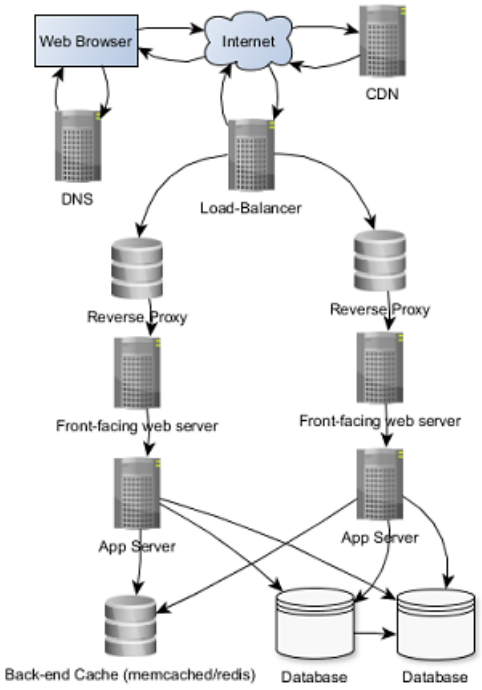
Firewalls or surrogates can act as ICAP clients and send outgoing requests to a service that checks to make sure the URI in the request is allowed.

> Comcast is using ICAP as part of a new malware alerting system. If the ISP detects signs of malware infections (e.g., an unusual spike in traffic from a customer's home, or if mass numbers of e-mails suddenly start going out of that user's account), then the ISP will modify returned web pages to include a warning notice to the customer (as opposed to sending them an email, I suppose). The design routes all HTTP requests to a Squid proxy, which (acting as an ICAP client) will check an ICAP server to see if a notice needs to be added or not. If so, the returned web page is modified with "a pop-up notice".
>
> [*arstechnica.com/security/news/2009/10/botnet-hosting-subscribers-soon-to-get-warnings-from-comcast.ars*]

Putting all these pieces together results in a complex web architecture. Not all web services require all these parts to be efficient, fortunately. Thus, every organization is likely to have a different architecture. In the extreme case, you have a web browser (with both memory and file caches), behind some organization's firewall and proxy (caching) server, then a CDN that can integrate some services into dynamic pages (such as ad injection) and cache static content. A CDN is reached via DNS geographical load-balancing to send your request to the nearest data center.

If the CDN doesn't have the data in its cache, or the data has aged-out, the request is routed instead to your organization's data center. Once there, the request may go through a reverse-proxy cache, then a load balancer to send to a web server (known as a front-facing web server). That web server in turn makes requests to databases, ad-injection, RUM, and other services, each of

which may have its own cache and/or load-balancer. The result is assembled by the front-facing server and returned:



Lecture 16 — Understand and Configure Reliable Time

Short answer: Use the default NTP daemon configuration, and forget about it.

[*See man time(7),* clock sources in linux, and Java API - Instant for details.] There are two main clocks in a computer. The ***hardware clock*** (a.k.a. the *real time* clock, the **RTC** (Solaris name on x86), TOD (*Time Of Day*; Solaris name on Sparc), TOY (time of year), the BIOS clock, or the CMOS clock) runs independently of any running OS, and runs even when the machine is powered off (it uses a battery). The OS can only set this clock to a whole second, but it can detect the edges of the 1-second clock ticks, so the clock actually has virtually infinite precision. On some systems, this clock can be accessed with the `/dev/rtc` driver.

> All x86 PCs, and ACPI based systems, have an RTC that is compatible with the Motorola MC146818 chip on the original PC/AT. Today such an RTC is usually integrated into the main-board's chipset (south bridge), and uses a replaceable coin-sized backup battery.
>
> All computer clocks are based on counting the oscillation of a piece of crystal. Crystals vibrate at a known frequency when electricity is applied. Unfortunately, that frequency varies based on environmental factors such as temperature: the hotter, the faster. The accuracy of such a clock depends on the stability of the crystal (its resistance to such skew), and is called *syntonization*.
>
> Most computer clocks, including for servers as well as commodity PCs, use a cheap bit of crystal that costs about 10 cents and can skew the frequency of your clock by seconds per hour. A higher quality crystal is possible, but not available today (2016). The next step up in quality is a PCI or external clock that costs about $1,000.

The main clock is called the ***system time*** clock, which is the time kept by the kernel and driven by **clock tick interrupts** (one per CPU).

The *system time* is the number of seconds since 00:00:00 January 1, 1970 GMT (i.e., the number of seconds since 1969). GMT time is also known today as UTC. It is just a number stored in RAM (or a CPU register). This special zero time is the start of the *epoch*, and often used (for network time, GPS, etc.) This time is set from the RTC at boot time. Thereafter, a "tick" counter is used to count time since then. The *Time Stamp Counter* (**TSC**) is a 64-bit register present on all x86 processors since the Pentium. It counts the number of cycles since reset (from the RTC at boot time).

> The TSC has been the preferred clock source since it is fast to query (if your CPU supports the RDTSCP instruction) and is very accurate (sometimes; keep reading). However, modern systems mess up TSC:

Each core may have a different TSC with a different value. Modern CPUs can work around that by providing a *constant TSC* for all cores.

Another modern issue is with power management. Such systems can increase or decrease the speed of the CPU. The TSC thus might be running much faster or slower (or occasionally, stopped altogether). Modern CPUs can account for that if they support *invariant* (or *nonstop*) *TSC*.

To see if your cores support the modern features needed to make TSC usable, try this command:

```
grep '^flags' /proc/cpuinfo |head -n 1 \
  |grep '[^ ]*tsc[^ ]*'
```

You are looking for the flags `tsc`, `rdtscp`, `constant_tsc`, and `nonstop_tsc`.

The system time is the time used by the kernel and other programs. The hardware clock's main purpose is to keep time when the OS isn't running; it is consulted at boot time and set at shutdown time.

Due to historical Unix definitions of time, the number is actually stored as a 32-bit number. The problem with that is, this will overflow after 2^{32} seconds. This will occur at 8:14:08 PM EST on Jan 19, 2038. Similar to the "Y2K problem", this is called the "Y2K38 problem".

On a network, it is very important that all your servers agree on the time, to within 1-2 seconds. This time period is known as the ***synchronization distance*** and is usually determined by 1/2 the round-trip time to send a packet to the time server, plus various other errors.

Linux provides `clockdiff` utility to measure the clock difference between your local host and some remote host.

The easy way today to synchronize time on many servers, to millisecond accuracy, is the Network Time Protocol, or NTP (RFC-5905). An NTP server takes an external time source and distributes it across a network ensuring all machines in that network are running to exact same time. The external time source is often the GPS time signals, the time signal sent out from various national laboratories (which often use atomic clocks), the average time from a pool of servers, or even just a host in your network.

Network latency is unpredictable and thus NTP time may not correlate across different hosts in a network. Practically speaking, the best you can hope for with NTP is **tens of milliseconds of accuracy**, most of the time. In addition, NTP may cause the system time to leap ahead or backward in

time to make the adjustment (although that can be prevented with proper configuration).

Networked hosts used for industrial control and sensors has a need to correlate events more accurately than NTP can handle, that is, sub-microsecond synchronization distances. Such systems generally don't use Internet time sources, don't cross routers or switches, and don't use NTP. Instead, PTP (*Precision Time Protocol* IEEE std. 1588-2008) is often used in such situations.

Why synchronized clocks are important

Many services won't work properly otherwise! Some SANs (storage area networks) require proper synchronization for filesystem usage and proper timestamp control. Some SANs (and some applications) can become confused when dealing with files that have timestamps that appear to be in the future. Central log entries will appear in the wrong order, making trouble-shooting and intrusion detection difficult or impossible. Kerberos and other security systems will deny access (so will Microsoft Active Directory, which affects us if using Samba). LDAP and other types of databases that use replication and synchronization, require accurate time or they just won't work.

It's not as important to keep accurate time on all hosts, as much as it is important to keep all the clocks in agreement. Of course, that's a lot easier when they all keep accurate time.

If you don't want to sync time to an internet time server for some reason (e.g., a closed, high security network) and don't want to invest in a radio clock or GPS solution, you use the localhost system time as a (*pseudo-)reference* clock.

A ***reference clock*** will generally (though not always) be a radio time-code receiver which is synchronized to a source of standard time such as an atomic clock. In many parts of the world (including the U.S.) time signals are available via radio broadcasts (or GPS time signals).

Local reference clocks are identified by a syntactically correct but invalid IP address, in order to distinguish them from normal NTP time servers. Reference clock IP addresses are of the form **`127.127.`*`t`*`.`*`u`***, where *t* is an integer denoting the clock type (=1 for "undisciplined local clock") and *u* indicates the unit number in the range 0–3.

If you can situate an antenna with a good view of the sky, GPS is a very good solution. You can purchase Garmin GPS18LVC for under $100 and build one. You will need a soldering iron, a DB9 or DB25 connector of appropriate gender and a 5 VDC power supply. For more money, you can get "plug it in and turn it on" solution, some from Microsemi (previously, Symmetricom) TimeServer can cost $6000 or more (<$1000 on eBay), but you could also get a TM1000A GPS

time server for around $300. (There are few if any radio-based time servers anymore, but there are some that don't use GPS exclusively, such as Clepsydra.)

If you don't have that kind of money, **remember that it isn't so important that all your computers have accurate time, but rather that they all agree on the time.** So, you can setup one of your (centrally well connected) hosts as a network time server (that is, a "fake" reference clock you manually update) and have all the rest synchronize to it.

> You might wonder why you cannot simply use the hardware clock built into most computers. After all, a $15 watch is accurate to a second or so per month of drift, at worst. Why are computer hardware clocks so much worse?
>
> It's because that watch has been calibrated to the exact frequency of the quartz crystal. The clock in your computer uses the time signal from your motherboard, and that wasn't calibrated. You may not care if you have a 2,800,000,000Hz CPU or a 2,800,010,000Hz CPU, but that affects your hardware clock a lot. Not only that, but the speed varies with temperature. Some PC firmware will even vary the CPU frequency for one reason or another (such as to save battery on laptops), and not bother to adjust the time for that change; the time appears to leap forward or backward.

To see what time sources are available on your system and which one is currently being used for system time, run the commands:

```
cat /sys/devices/system/clocksource/clocksource0/available_clocksource
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
```

You can force your system to use a particular time source at boot time with the kernel parameter of "clocksource=tsc". You can change it on a running system with "echo *some-time-source* > .../clocksource0/current_clocksource".

Clock Resolution

System time is more precise than seconds, but the exact precision varies from system to system. **The clock resolution is measured in *jiffies*.** (On Solaris, the term "jiffy" isn't used but the concept is.) The system CPU cycle is used to count a high-resolution timer in "clock ticks" or just "***ticks***" (or TSC). This is the highest resolution possible on your system. This value is scaled to provide a desired jiffy size regardless of the speed of your CPU. The size of a jiffy in Linux is determined by the value of the kernel constant **`HZ`**. On Solaris, a tick=jiffy is just set to 0.01 seconds. Note, the kernel uses jiffies for (for example) scheduling, but applications often use other time sources directly; Fedora 25 provides eight clocks! (See the man page for the POSIX clock_gettime(2) for a list and more details.)

For a long time, `HZ` was set at 100, giving a jiffy value of 0.01 seconds. On Linux 2.6, `HZ` is a configurable kernel parameter and can be 100, 250 (the default)

or 1000, yielding jiffies value of, respectively, 0.01, 0.004, or 0.001 seconds. Since kernel 2.6.20, a further frequency is available: 300, a number that divides evenly for the common video frame rates (PAL, 25 HZ; NTSC, 30 HZ) and gives a jiffy of 0.00333... seconds. (This can be useful for multi-media work.)

`HZ`=1000 is preferred for desktop systems, 300 for multi-media workstations, and 100 or 250 for servers. (Q: what is it set at for YborStudent? Ans: `grep CONFIG_HZ= /boot/config-*`.)

> POSIX systems provide the `gettimeofday` syscall, which returns the system time in seconds and microseconds. Higher resolution timers in *nix are non-standard. They are made available to programmers via system calls and device drivers (with `/dev` entries). Solaris calls these *cyclics*.

Clock Drift and Skew

A computer's software clock is not very accurate and can ***drift*** seconds a day due to temperature changes (***wander***) and processing delays (***jitter***). The hardware clock is more like an old pendulum clock—while not terribly accurate, the drift is much more systematic (and is known as ***skew***); the clock will likely gain or lose the same amount of time each day. The system clock has better precision of 1 μSec (0.001 seconds), but the hardware clock probably has better long-term stability. If you have a better clock available briefly, you can measure skew during some period of time, and then adjust for it.

> All sorts of schemes exist to adjust your software clock to keep it accurate, but the easiest way is to enable the **`ntpd`** (***network time protocol daemon***) to run at boot time. (On Solaris `xntpd`.) Then you don't have drift issues (it's the time source's problem then). See NTP below.

When not using NTP, drift/skew is controlled by first estimating it based on past performance, then configure your system to control it by periodically adjusting the clock. This is done by turning off NTP or other clock synchronization software, then (on Linux) manually setting the hardware clock (i.e., with `/etc/adjtime`) time with `hwclock --set`. Do this for a few times over a few days, and the drift/skew will be recorded in the `/etc/adjtime` file.

Once you have accurate drift data recorded, use `cron` to maintain an accurate hardware clock time by periodically setting the time with `hwclock --adjust` (on Linux). The adjustment is based on how far the system estimated the clock has drifted since the previous adjustment. Note that if the cumulative error is less than 1 second the clock won't be adjusted. It is also a good idea to use the command `hwclock --adjust` just before the `hwclock --hctosys` at boot time. The

Solaris `rtc` command manages drift similarly. On non-Linux systems, you use the `date` command to adjust both system time and clock time.

For a host not connected to a network, the system clock drift can be managed safely with **`adjtimex`** on any *nix system. Fedora doesn't seem to install this by default, use "`yum install adjtimex`". Note on Linux `hwclock` can also be used with its own drift file, and other OSes will have their own methods for this.

Setting System Time Manually

The kernel initializes system time from the hardware clock at boot time, and then (almost) never uses the hardware clock until the next boot. (Note that with early DOS, a human had to set the system time manually each boot.)

The **`date`** command may set both clocks on some systems (e.g., Solaris). On such systems user-level **commands may not be provided to access the hardware clock** directly. SAs use `date` to set both clocks at once. (Or write programs to use any system calls available to access the hardware clock. For example on Solaris Sparc, the `stime(2)` call is provided to set the TOD (*Time Of Day*) hardware clock.)

On other systems (e.g., Linux) `date` only sets the system clock. A different command sets the hardware clock. On Linux you can access the hardware clock with the **`hwclock`** (`--utc` or `--localtime`) command. Solaris x86 uses the **`rtc`** command to set the hardware clock's timezone and to adjust for DST, but not to set the time:

```
date
hwclock --show [--utc]
timedatectl
timedatectl list-timezones |grep whatever
date -s "31 OCT 2017 13:00:00" # day mon year
hh:mm:ss
date +%Y%m%d -s "20171031" # using arbitrary
format
date +%T -s "10:15:00" # the time in hh:mm:ss
format
hwclock --systohc # set hardware clock to system
time
timedatectl set-time YYYY-MM-DD [ HH:MM:SS]
timedatectl set-timezone 'America/New York'
```

The kernel reads the hardware clock at boot time. At shutdown the hardware clock is reset to the system clock time. (See the `init.d/halt` script.)

You can safely update the hardware clock while the system is running. (Doing so should(!) disable the automatic sync from the system clock that is done on some

systems.) So the next time the system boots up, it will do so with the adjusted time from the hardware clock.

Adjust Time Gradually, and Never Back!

It is important that the system time not have any *discontinuities* (abrupt changes forward or backward in time) such as what happens when you use the `date` command to set it. Any abrupt clock adjustments will cause problems in a networked environment. (Qu: Why? Ans: Mess up logs, billing, networking, and makes security holes.)

To avoid these problems, the system time should be ***monotonically non-decreasing***. This is best done by slowing down or speeding up the system clock until it is accurate. The next best approach (and more common) is to change gradually the time with a series of tiny adjustments, spread out over a long period. (Note that NTP doesn't guarantee this!) See the box above about Google's shmear.

When using NTP, **`adjtimex`** (see below), or some other means to keep system time accurate, you need to **sync the hardware clock from the system clock** (Q: why? Ans: the sync at shutdown may not occur, as Unix servers are rarely shutdown but may reboot unexpectedly, so in this case the system clock is assumed to be more accurate). The kernel may do this automatically from time to time (or not). For Linux, run `hwclock --systohc.`

Networked hosts (i.e., all hosts today) can set the system time from a remote host that is believed to have accurate time. Such a remote host is called a ***time server***. You can run cron shell scripts (e.g., run `ssh date` to get the remote time then use `date` to set the local system clock).

NPT isn't the only network time protocol; another is the old **`rdate`** which connects to an RFC-868 time server (port 37, TCP or UDP) running on a different host. But `rdate` has poor error handling and is rarely supported anymore. As mentioned earlier, control systems often use PTP. There are other protocols in use as well.

Today, NTP is used to set the system time from an internet time server. **Use `ntpdate -q` *`server`*** to display the time of *`server`*.

You can maintain a private time server for your network, but there are free alternatives. Publicly accessible time servers include **`time.nist.gov`** and `pool.ntp.org` (which queries 5 random servers which are averaged.) The *time server* must be running an NTP server. Note such public time servers are rarely secure.

Naturally Systemd uses its own command for this, **`timedatectl`**. This command can set the date/time, the timezone, control NTP, and more.

(Running the command with no arguments shows lots of interesting information.

The first advice I gave is probably the best: enable the NTP client daemon to keep your clocks accurate. Any errors are then the server's problem, not yours. NTP will be discussed in detail, below.

Linux "11 Minute" Mode [*From adjtimex(8) man page*]

On Linux, using `date` only sets the system time, not the hardware clock. Normally the hardware clock is set from the system clock at shutdown. If you are running NTP or some equivalent system that keeps the system time accurately synchronized to a (hopefully accurate) time server, the Linux kernel will also sync the hardware clock from the system clock every 11 minutes.

Running a server such as `[x]ntpd` turns this mode on. However, manually **setting the system clock with `date`, `hwclock`, or other means turns this mode off.** (Oddly, if you turn the mode off, restarting your NTP daemon will not re-enable the mode! The adjustment must be the first one made after the kernel boots or the mode is not changed to on.)

Run "`adjtimex --print`" to see if this mode is on, it is if the 7th (64) bit is zero. (See `11minmode.sh`.) Use `adjtimex -s 0` to turn it on and `adjtimex -Rs 0` to turn it off. (That may change other settings too; the number argument is a bit-mask and this mode is the 7th bit.)

If this mode is on, don't run `hwclock --adjust` or `hwclock --hctosys` (except at boot time before starting NTP, or else restart NTP afterwards). Remember if synchronizing to a time server, any drift is their problem, not yours.

In 11-minute mode, the command `hwclock --systohc` should still be run when the machine is shut down. To see why, suppose the machine runs for a week with `ntpd`, is shut down for a day, is restarted, and `hwclock --adjust` is run by a startup script. It should only correct for one day's worth of drift. However, it has no way of knowing that `ntpd` has been adjusting the hardware clock, so it bases its adjustment on the last time `hwclock` was run!

Also note that if 11-minute mode is on, the clock drift is not measured and the resulting adjustments may be quite far off.

Using NTP Remote Time Servers

Many schemes and protocols can be used for setting the system time from remote time servers, but most popular is **NTP** (**`www.ntp.org`**). Different OSes have different implementations but these should inter-operate using the protocol defined

in **RFC 1305**. Solaris calls its implementation `xntpd`. Even Window XP and newer supports NTP.

> When running a cluster, it is imperative that all hosts agree on the time as accurately as possible. This is a common reason to run an NTP server per datacenter.

An NTP daemon may run as a time client, time server, or both at once. As a time client, the daemon periodically will query one or more time servers for the current time. Then they will gradually adjust the local system time to synchronize with it.

On Fedora Linux, the default NTP daemon (as of 2014) is called Chrony ("`cronyd`"). It can be configured and controlled (say to make manual clock adjustments) with the `chronyc` interactive tool. Chrony is designed for intermittent Internet access and synchronizes the clock much more quickly than did `ntpd`.

> The config files are different from shown below (written for the older `ntpd` daemon in Fedora 15 and older); Chrony uses the file `/etc/chrony.conf`. However, the concepts and settings are quite similar. In addition to the Chrony website's documentation, see this Red Hat Crony Guide.

The common setting in `ntp.conf` for a "fake" reference clock is these two lines; the *stratum 15* part ensures this fake clock won't be used unless the real server is unavailable:

```
...
server 127.127.1.0
fudge 127.127.1.0 stratum 15
```

> In the Solaris `xnptd` implementation, you can instead run the server in *orphan mode* by setting XType to 1 for an "undisciplined local clock" in the config file `/etc/inet/ntp.server`.

There aren't enough reference clock time servers on the Internet for every host in the world to connect directly to them. The solution is to **use a hierarchical scheme**: a central authoritative time server, then a site time server at each location, and local computers sync to the local site time server. In theory, the clocks higher up in the hierarchy are more accurate than the ones lower down.

A stratum-*n* time server gets its time from a stratum-(*n-1*) server. Stratum-0 refers to reference time clocks. Often an *autonomous system* (AS) will have a one or two *peer* stratum-1 time servers that use radio clocks as a reference time source, or stratum-2 servers that use public Internet stratum-1 servers. Then each geographic location has a stratum-2 (or -3) server that syncs to the more authoritative one.

Really large enterprise networks may have stratum-3 (or -4) clocks at various sites. Avoid having more than three levels or ***strata*** or the time may not sync to within 1–2 seconds throughout the AS!

Various governments that own such clocks operate public stratum-1 servers, such as the services offered by the NRC in Canada and NIST and USNO in the US. See **`www.time.gov`** for radio stations WWV, WWVH, CHU, and other time servers on the Internet. ("Atomic" clocks/watches use this.)

NTP time servers are grouped according to their accuracy. ***Stratum-0*** refers to a *reference clock*, often an atomic clock or a radio-controlled clock, or even a GPS clock. *Stratus-1* refers to the host (the time server connected to such a reference clock). A stratum-2 time server gets its time not directly from a reference clock, but from a strtatum-1 time server.

The higher the number the more "hops" between you and a reference clock. The worst time server would be a stratum-15 server, although you rarely encounter any servers worse than stratum-2. Some servers report stratum-16 to indicate an "unsynchronized" clock.

Standard (Linux) NTP supports **`ntpd -q`**, used to sync the clock immediately with some pre-configured NTP time server. (Solaris `xntpd` uses `ntpdate` command for this.) On the other hand, in the daemon mode **`ntpd`** will sync the clock gradually in a series of small steps (if it is off by a lot). Thus `ntpd` runs on both time servers and clients.

A number of **public NTP servers** are available that sync to various governmental reference clocks and are thus public *stratum-1* clocks. There are also public *stratum-2* network time sources you can use. (For a list of these see `support.ntp.org/bin/view/Servers/WebHome`, and show NTP lists on `ntp.org` site.)

There is also a **pool of servers** available (a load balancing system where you get the time from several randomly selected time sources in the pool, and calculate the average, generally obtaining a highly accurate current time value.)

> An easy way to give back to the open source community is to participate in the NTP pool, to allow others to sync to your clock (if you have an accurate one!)

You will need to configure **`ntpd`** to point to some Internet time source (e.g., **`0.pool.ntp.org`**) and you will need to open incoming (and outgoing if not currently allowed) **port UDP/123**. (This may be done automatically on Fedora; read the `init.d/ntpd` script.)

> NTP rarely uses any security. That causes a huge problem since an attacker can set the clock back to a time before some PKI certificate was

revoked/expired, perform MitM attacks on HTTPS or SSH, and do other nasty things. Default NTP daemons rarely are configured for whatever security they do have available, and do not guarantee monotonically increasing time. You should definitely check your setup for maximum security, or use GPS or radio time servers instead.

NTP is subject to MitM attacks. A shared key is often used to prevent this, but a newer protocol, NTS uses TLS to exchange authentication keys (cookies).

NTP Client Time Adjustments

The NTP daemon is both a time server and a client. The configuration files say what other clients can use us as a time server. Meanwhile the local system time will also be adjusted by NTP.

If the system clock is off by a lot (e.g., more than 16 minutes), `ntpd` won't change the time in one step. That could lead to problems, especially if setting the time back more than a second. If the time is off by, a lot `ntpd` will adjust the clock in a series of small adjustments, spread out over some time.

If the system clock is off by less than some threshold (128 ms by default), NTP will speed up or slow down the clock until time is synchronized. This is called a ***slew*** adjustment. If the error is more than the clock is adjusted forward or backward(!) in a series of small ***step*** adjustments.

Slew adjustments are usually limited to 0.5 ms per second, so even a one-second error would take 2,000 seconds at least to correct with slew. An adjustment of 10 minutes will take almost 14 days to complete with slew!

If the clock is really off, `ntpd` will just log an error message and won't attempt the adjustment at all. (The threshold for this differs from one implementation to the next, but generally is between 3 seconds and several minutes. `ntpd` will skip a few bad time packets but if the difference persists the server will shut down for 15 minutes (until the clock frequency data in `ntp.drift` can be recalculated) or completely if the error is considered too large to adjust.)

To set the time manually, use `ntpd -nqN`. (This is the same behavior as the deprecated `ntpdate` program.)

Configure Linux NTP

Use the `ntp*` commands such as `ntpq` or `ntpdc` (if available). The configuration files are `/etc/ntp.conf`, `/etc/ntp/*`. Don't forget to make firewall holes (See below) and update security (e.g., SELinux) policies.

If you just want to keep your system clock accurately set to the true time, the configuration file `/etc/ntp.conf` (for the `ntpd` program from `ntp.org`) is really simple:

```
driftfile /var/lib/ntp/ntp.drift
server 0.pool.ntp.org
server 1.pool.ntp.org
server 2.pool.ntp.org
```

The `0`, `1` and `2.pool.ntp.org` names point to a random set of servers that will change every hour. **Make sure your computer's clock is initially set to within a few minutes of the correct time or NTP won't update it.** Use "`ntpd -qn`" or just use the "`date`" command to set the current time.

When the NTP daemon starts up, it looks for the `ntp.drift` file. This file contains system clock frequency data that tells NTP the rate of system clock error. **If the `ntp.drift` file isn't found `ntpd` will refuse to serve time for 15 minutes, while it measures the system clock drift.** Every hour while up, `ntpd` will write to that file the latest information.

The most basic `ntp.conf` file will simply list two servers, one that it wishes to synchronize with and localhost (`127.127.1.0`). This localhost IP indicates a "fake" local reference clock and is used in case of network problems or if the remote NTP server goes down; NTP will synchronize against itself until it can start synchronizing with the remote server again. Always list at least two remote servers that you can synchronize against. One will act as a primary server and the other as a backup. (`xntp` supports an *orphan mode* which will still serve time even if the server is not synchronized with an external source.)

You should also list a location for a drift file. Over time, NTP will learn the system clock's error rate and automatically adjust for it.

The `restrict` option can be used to provide control over who can do what to NTP, and who can use it as a time server. See the man page for details.

Once you have edited `ntp.conf`, start `ntpd` and after some time (this could take as long as half an hour), the system (and hardware) clock should be accurate. Run "**`ntpq -p[n]`**"; the output something like:

```
     remote           refid      st t when poll reach   delay   offset
jitter
==============================================================================
+81.6.42.224     193.5.216.14  2 u   68 1024  377  158.995 51.220
50.287
*217.162.232.173 130.149.17.8  2 u 191 1024  176   79.245   3.589
27.454
-129.132.57.95   131.188.3.222 3 u 766 1024  377   22.302 -2.928
0.508
```

The IP addresses will be different because you've been assigned random timeservers. The essential thing is that if one of the lines starts with an asterisk (*), then your host gets the time from the Internet.

You can query an NTP server with **`ntpdate -q`** or the **`ntpdc`** command. For example "`ntpdc -c loopinfo`" will show system clock *offset* from NTP time. Also try "kerninfo", "sysinfo", and "reslist". **`ntptime`** will show the current (local) time and estimated error. See other `ntp*` commands man pages for more.

As **`pool.ntp.org` will assign you timeservers from all over the world**, time quality will not be ideal. This is due to Internet *latency* which is worse the farther away the server is from you. You get a bit better result if you use the continental pools (`{europe,north-america,oceania,asia}.pool.ntp.org` currently exist), and even better time if you use the country pools (like `us`, or `ch.pool.ntp.org` in Switzerland). For all these pools, you can again use the `0`, `1` or `2` prefixes, like `0.us.pool.ntp.org` for different sets of servers.

Note that the country pool might not exist for your country, or might contain only one or two timeservers. If you know of timeservers that are really close to you (measured by network distance, with `traceroute` or `ping`), time probably will be more accurate if you use those servers instead (you'll need permission).

Check to see if your ISP has a timeserver, or if other local organizations using that ISP have accessible time servers. If so you should use that; you'll probably get better time and you'll use fewer network resources. If you know only one timeserver near you, you can of course use that and one or two from `pool.ntp.org`. **Configuring more than three time sources is an unnecessary waste of network resources.**

NTP Issues and Alternatives

There are some problems with NTP today:

- While lower stratum clocks are presumed to be more accurate than higher ones, there is no definition for that. In practice, most clocks are the same.
- NTP uses UDP to send time data. In order to avoid flooding a network with NTP traffic, NTP uses a default polling interval of once every 64 seconds, and only adjusts the local clock at that time. Thus it can take tens of minutes for the local clock to synchronize with the external sources. The default can be changed to a maximum rate of once per 16 seconds, but that's still not enough to get sub-millisecond synchronization.
- Since NTP traffic may pass through several routers and switches, the latency can change and be hard to measure.
- Using the default NTP server pools is like asking a stranger for the time. It may or may not be very accurate.

- A common problem is secure firewalls may block the NTP port and your server's clock never gets adjusted! Since the system won't crash (right away) as a result, many system admins don't notice the issue for months or longer.

Data centers, especially ones used for high-frequency financial trading, require better synchronization than NTP can provide. Increasingly common today is PTP (***precision time protocol***). PTP was originally defined by IEEE-1588 in 2002 and updated (incompatibly) in 2008. This standard is for use in factory automation, electrical grids, and cellular networks. (Consider a robotic arm moving at 30 MPH. That's 44 feet per second and a collision would not be pretty. Similar issues exist when one power grid hands off power to another.)

PTP was designed for use on a single LAN (no switches or routers), but can be deployed in an internet too. It uses a main clock (the grandmaster) to multicast time signals, one every second. Clients periodically send a sort of ping to the main clock, in order to measure the latency between them.

Leap Seconds

In 6/2012, a *leap second* was added. These are adjustments to UTC in order to keep standard time in alignment with the Earth's rotation, which wobbles a bit. Since 1972 (when the idea was invented), there have been 25 leap seconds (up to 2016). These are announced by the IERS in bulletin C.

The problem is applying the leap second to hosts in a cluster. Lots of transactions have microsecond timestamps. If transaction A happened after B, but A had a timestamp before B, lots of bad things will happen. As a result, a lot of websites and security software (SSL also uses timestamps) had outages after the most recent leap second adjustment: Reddit, Mozilla (with Java), Quantas airline, Yelp, and others. (The previous leap second was added in 2008). Because of this. another version of time, UTC1) is available but without the leap second adjustments.

Another leap second problem caused power consumption to jump in data centers using Linux. A kernel bug meant that the *hrtimer* (high resolution timer; Linux and Unix provide several clocks, not just one) code fails to set the system time when the leap second is added. The result is that the hrtimer representation of the time taken from the kernel is a second ahead of the system time. If an application then calls a kernel function with a timeout of less than a second, the kernel assumes that the timeout has elapsed immediately after setting the timer, and so returns to the program code immediately. In the event of a timeout, many programs simply repeat the requested operation and immediately set a new timer. This results in an endless loop, leading to 100% CPU utilization. That caused a megawatt of wasted power in at least one data center.

A patch was made available but resetting the system clock also solved the problem. Running "`date -s "$(LC_ALL=C date)"`" should cure Linux machines which are running at 100% CPU usage because of the bug.

> Google came up with a method to avoid the problem, they call leap shmear. They modified their internal NTP servers to gradually add a couple of milliseconds to every update, varying over a 1-day time window before the moment when the leap second actually happens. This meant that when it became time to add an extra second at midnight, our clocks had already taken this into account, by skewing the time over the course of the day. All of their servers were then able to continue as normal with the new year. The Java Virtual Machine clock use a similar scheme called UTC-SLS. Both schemes make accurate timing difficult since milliseconds may be longer or shorter than they should be during the adjustment.

Additional notes:

- When using NTP pools, it can rarely happen that you are assigned the same timeserver twice. Restarting the NTP server usually solves this problem. If you use a country pool note that it may be there is only one server known in the project; better to use a continental pool in that case. You can browse the pools to see how many servers there are in each.
- Make sure that the time zone configuration of your computer is correct. `ntpd` itself does not do anything about the time zones, it just uses UTC0 internally.
- Be friendly. Many servers are provided by volunteers, and almost all time servers are really file, mail, or web servers which just happen to also run `ntpd`. So don't use more than three public time servers in your configuration.
- If you use Windows you can also use the NTP client that is built into the system. Just execute:

  ```
  net time /setsntp:pool.ntp.org
  ```

  at the DOS command prompt. The same can be achieved by (as administrator) right-clicking the clock in the taskbar, selecting "Adjust Date/Time", and entering the server name in the "Internet Time" tab.
- If you have a static IP address and a reasonable Internet connection (bandwidth is not so important, but it should be stable and not too highly loaded), please consider donating your server to the server pool. It doesn't cost you more than a few hundred bytes per second traffic, but you help the project survive. Please read the joining page for more information.

- With systemd, you can control NTP with "`timedatectl set-ntp yes`".

BIOS Timers

One way to wake up (or shut down) your computer at a scheduled time is to enter your computer's BIOS, and set a wakeup alarm in the Power Management settings. This will be managed either by APM or ACPI settings, depending on the age of your BIOS. APM, *Advanced Power Management*, is an older power management standard. ACPI, *Advanced Configuration and Power Interface*, is the newer, more advanced standard. Most *nix systems provide access to this via an RTC (real-time clock, but that is a bad name) driver. (Try "`# grep -i rtc /var/log/messages`". That should show if the driver is present, and if it uses localtime or UTC time.)

To check if any wakeups are set, try "`cat /sys/class/rtc/rtc0/wakealarm`". If that file is empty, no alarms are currently set. To clear one, echo `0` (zero) to that file. To set one, echo a timestamp (seconds since midnight 1/1/1970 GMT). For example, to wake up your computer in three minutes from now, try this:

```
# echo $(date '+%s' -d '+ 3 minutes') \
     > /sys/class/rtc/rtc0/wakealarm
```

and then shutdown. Your computer should reboot itself in 3 minutes.

Summary

Don't set the time manually with `date` or other commands; instead rely on NTP. Don't forget to enable NTP on all your hosts *and* network devices (routers, switches, etc.), and don't forget the firewall holes needed. At first, check the system logs for NTP messages.

Lecture 17 — Timezones and Daylight Savings

Modern systems allow you to configure a system-wide **default *time zone*** for your users. Time zone information is created with *zone info* text files, that are compiled into timezone files via **`zic`** (zone info compiler). On Linux **`/etc/localtime`** is a link to such a file in **`/usr/share/zoneinfo`**. (Maybe **`/usr/share/zoneinfo/localtime`** too.) Changing this *may* affect running processes without any reboot.

> The Systemd `timedatectl` command can be used to view or change the system default time zone.

On Solaris, you must set a default value for the **`TZ`** environment variable usually in `/etc/profile`. If `TZ` isn't set then a default value compiled into the kernel is used.

Users can over-ride the default by setting the **`TZ`** environment variable to a properly formatted time zone string such as `PST8PDT` or `America/New_York` (a relative pathname). **See the `tzset(3)` man page for details.** (Demo **`tzselect`** command; this works on Solaris too.)

> *From* `opengroup.org/onlinepubs/009695399/basedefs/xbd_chap08.html#tag_08_03`
>
> The **format of `TZ`** is either "`:`*`pathname`*" (absolute or relative to the *system timezone directory*, usually `/usr/share/zoneinfo`), or (with no spaces):
>
> ```
> std offset[dst[offset][,start[/time],end[/time]]]
> ```
>
> Where `std` and `dst` indicate the designation for the standard or Daylight Savings Time) timezone. If `dst` is missing then the alternative time does not apply in this locale. `std` and `dst` are 3 or more characters each. Each of these fields maybe quoted with '<' and '>', which allows '-' and/or '+' in the names (but is a new feature and not supported on all systems yet).
>
> The `offset` indicates the value added to the local time to arrive at Coordinated Universal Time. The `offset` has the form "`hh[:mm[:ss]]`". The hour (`hh`) may be a single digit. If no `offset` follows `dst` the alternative time is assumed to be one hour ahead of standard time. If the `offset` is preceded by a '`-`', the timezone is east of the Prime Meridian; otherwise, it is west (which may be indicated by an optional preceding '+').

Since you can't change environment variables such as `TZ` in a running process, you can **only change the timezone by restarting daemons**. (This is an advantage of the Linux scheme.)

The hardware clock can hold UTC or local time. Some OSes interpret the hardware clock value as local time. Others assume UTC. (As mentioned

previously, Linux allows either by examining the last line in `/etc/adjtime`, which is set to whatever you used last with the `hwclock` command.) This can complicate dual-boot and virtualization. Early Windows used local time, but Vista and Win7 use UTC. Solaris x86 by default uses local time (in an attempt to permit dual-boot with Windows) and UTC on Sparc. Fortunately Linux and Solaris x86 use either, so you chose whatever the "other" OS forces you to use. **If a choice is possible, UTC is the way to go**.

> Note! Some modern BIOS will also attempt to adjust the hardware clock for daylight savings. On a dual-boot system each OS could try to adjust the hardware clock as well. If you're unlucky, your clock will gain or lose two or three hours, not one! Using UTC time in the hardware clock (for all OSes) avoids this problem. But be sure to check your BIOS settings too.

On Solaris you can change the time zone used with the **`rtc(1M)`** command. The choice is kept in **`/etc/rtc_config`** between boots. Use "`zone_info=UTC\nzone_lag=0`" to set UTC time. **Note this value must be set post-install!** Solaris doesn't allow you to set the hardware clock's timezone in the install procedure.

The Solaris file `/etc/timezone` is a symlink to **`/etc/default/init`** and that's where you set the default `TZ` and locale variables. See **`environ`**`(5)` man page for details, also `timezone(4)` and **`TIMEZONE(4)`**.

For the local host only, with Solaris the daylight savings time configuration of the hardware clock is done with the **`rtc`** command via a cron job by root. (If you set the timezone for the time server to UTC in the file `rtc_config`, you don't need to run that cron job.)

On Solaris the `date` and `rtc` commands has options that provide similar capabilities as the Linux `hwclock` (Solaris also has the `adjtimex` command).

The kernel has a timezone set too. This compiled-in default that is loaded into RAM can be changed with `hwclock` on Linux; when setting the system time from the hardware clock the time zone in effect (via `TZ` or `/usr/share/zoneinfo`) is used. This occurs at boot time.

> Some drivers may use the kernel's timezone (e.g., the DOS/VFAT drivers) and ignore `TZ`, and interpret the system time number according to the kernel's timezone. That can be confusing! If possible, make sure the kernel's time zone is the same as the system default time zone.

Note that changing the time zone only changes the value of time displayed, not the time-stamp of files etc.

To alter the format of displayed time you set the environment variable `LC_TIME`.

There is (and always has been) controversy concerning daylight savings time. Many people feel that DST is more harmful than useful and some states and countries have abolished DST.

Both `cron` and `at` take care of daylight savings time changes automatically, so jobs don't get skipped or run twice. However if your system is dual-booted or your BIOS tries to adjust this, you might have the correction applied twice!

To view the current timezone, use the command `date +%Z`. To view the current time in different timezones, use `zdump`.

Lecture 18 — Process Control and Signals *Optional Material — Skip*

Review these tools: **`ps`**, **`top`** (also `atop` and `htop`), **`ac`** (user connection times from wtmp), **`w`**, `/proc`, signals (`kill -l`, `man 7 signal`), **`kill`**, **`nice`**, **`renice`**. (Solaris: `preap`.) Also **`killall`** (different for Unix and Linux!), `pgrep`, and `pkill`. Schedule non-time sensitive jobs via `crontab`.

To see what process have files (or sockets) open, use **`lsof`**, **`fuser`**. `fuser` is standard in POSIX. `lsof` is more powerful but non-standard.

`fuser` *`filename...`* will show PIDs using any of the files listed. This can be used to see who is editing a shared file, or which processes are using some DLL. For example, `fuser /var/log/*`. The "`-u`" options will also list the user of that process. To see all the details, use "ps *PID*". For details on which processes are using the files, try:

```
ps -fl $(fuser -c /home 2>/dev/null)
```

(The "`-c`" option causes only a list of PIDs to be produced on stdout; all other information is sent to stderr.)

On Linux, use **`taskset`** to manage SMP, and **`chrt`** for real-time attributes. Process states: runnable (R), sleeping (S), stopped (T), zombie (Z).

Process resource limits can be controlled with the shell builtin command **`ulimit`** (`-a`: show all, `-H`: hard limits, -S (the default): soft limits), usually set from the system-wide login scripts (or from PAM, which a user can't modify). Soft limits can be changed by a process to any value that is less than or equal to the current hard limit. A process can (irreversibly) lower its hard limit to any value that is greater than or equal to the current soft limit. Only a process with appropriate privileges can raise a hard limit. In addition, cgroups can be used to change these limits on sets of processes. (**Show in `bash`; `ulimit -a; -Ha; -Hn 1000; -[S]n 900; -Hn 1000; -Sn 1010; -Sn 950`**.)

Process (and Thread) Management

An SA must understand how memory is used in order to monitor and manage it sensibly. Also an understanding of processes and how they use memory is important to understanding various security threats, such as buffer overflows.

"*A process is a running program*." Actually, this definition isn't correct anymore. Today all OSes support *threads*. Unix and Linux systems don't schedule processes, they schedule threads. (The Linux term is *tasks* but the industry standard term is *threads*.)

A *thread* can be thought of as a lightweight process. Threads make sense when you consider a typical server daemon (which is what Unix systems mostly run)

such as a web server: In the past every request caused the system to create a new process to handle it.

Creating a new process is a relatively slow operation on any OS. While you and I may never notice the time it takes, a busy server may have to handle hundreds (or thousands) of requests per minute. Each process also needs a lot of memory to hold the environment, security information, etc.

Finally, *inter-process communication* is difficult (say something changed and all web server processes must be notified).

There are different types of processes:

- *Interactive processes* are started by users from some UI.
- *Batch processes* are scheduled processes. These might be started automatically via `at` or `crontab`, or entered by a user. They run disconnected from any UI. Batch processes are often queued and run on a FIFO (*first in, first out*) basis.
- *Daemon processes* run continuously, disconnected from any UI. Often these are started at boot time. Most daemons are servers, but not all (some are client programs.)
- *Threads* are also known as *lightweight processes*. (Linux calls these *tasks.*) Every process is composed of one or more threads, which are generally scheduled independently.
- *Kernel threads* are asynchronous tasks of the kernel. They are generally always running. Users have little or no control over these.

Today we can **define a process as a container of threads and the data and resources they share**. Every process starts by creating a single thread. Later that thread can spawn new ones. Creating a new thread is relatively fast. Each thread only needs a little RAM, since all threads share the same global memory, environment, and code. Since threads do have shared access to memory, nothing special needs to be done to pass data between them. Even for non-server processes, using threads may make sense if your host has multiple CPU (cores), and you can divide the work between multiple threads. (E.g. the `make` program used to compile the Linux kernel takes a "`-j #`" argument to control how many threads to create. If you have two or more cores you can use "`-j2`" which will almost half the time it takes to compile.) **Qu: Suppose you download a file using the `ftp` utility on a host with two cores. Would it speed up the download to use two threads?** Ans: probably not.

Shared resources have issues of corruption, locking, deadlock, etc., and thus programming threads to share data is difficult. But worth it!

Although POSIX standardized threading ("*pthreads*"), they must have done a poor job because every kernel also implements an OS-specific thread system, which is often preferred by developers over the POSIX one.

On Linux at least, by default **every thread is issued its own unique PID and also a TID**. The PID is the same as the TID for the first thread in a process. For a multi-threaded process, each thread shares the same PID but has a unique TID.)

While most man pages discuss processes, they really refer to threads (e.g., the process priority is really a thread priority).

The Gnu `ps` command by default shows only the first thread's information, when a process has many. To see each thread's info, use "`-L`" (along with `-ef`). The `LWP` column is the TID, and the `NLWP` column is the count of threads in the process. (*Examine `mysqld`.*) To restrict the output to just processes of some users, use `ps -fu` *`user,user,...`*

Process Scheduling — The "Simplified" Linux Story

Processes (actually *threads* today) have complex priority calculations that vary by OS and OS version (and OS configuration). There are several controls available via the procfs ("`/proc/sys/`") system, but it is not recommended to tweak the default settings without a good understanding.

On Linux, real-time ("RT") processes (the Linux term for processes and threads is *tasks*, but I will stick with the more traditional term *process*) have assigned priorities from 0 to 99. (These are just extra high-priority; there is also a different concept of "hard real-time".) All "normal" processes have a priority from 100 to 139. That's 140 priorities possible.

Linux maintains a list of runnable processes for each priority level (conceptually anyway; in reality, there is one *runqueue* per CPU). When the scheduler needs to select some process to run next, it selects one from the lowest numbered priority queue; if there's more than one process with that priority, they are scheduled via round-robin.

The initial priority of normal (non-RT) processes is determined by their `nice` value. As the process runs, over time Linux will adjust that priority (the *static priority*) by plus or minus 5, resulting in the process' *dynamic priority*, which is used to select the next process to run. The adjustment is based on how long the process has been sleeping recently and how long it has been running recently. The idea is to give interactive jobs a priority boost.

In addition to the priority, each process gets a time-slice (traditionally called a *quantum*, since "time-slice" has a different meaning on other OSes; Linux CFS doesn't have traditional time-slices at all). The amount

depends on the `nice` value, and varies from 5mSec to 800mSec. With the default `nice` value of zero, the time-slice is 100mSec.

Once a process has exhausted its allocated time-slice, a new one is calculated for it. That process then has to wait until all other runnable processes have used up their time-slices. After the higher priority processes have no time left, the lower priority processes get some CPU time. When no processes have any time left, the whole thing starts again.

There are some extra steps in this. To prevent process starvation, if a process has been kept waiting too long, it gets a temporary priority boost, so it can run sooner. Processes identified as highly interactive don't have to wait for all other processes to use their time-slices; they get to run again right away.

Precautions are taken to prevent this from starving other processes. In essence, processes are assigned a proportion of the processor in addition to a time-slice. If one process uses more than its share of processor cycles, then another process is given a chance.

Each CPU has its own list of processes (and priorities). If one list is empty, Linux with attempt to shift some processes from another CPU's list (load-balancing). CPUs are grouped (for example, by NUMA nodes) and load-balancing occurs only with the CPUs in the same group.

Finally, the inclusion of *cgroups* (*control groups*) in Linux allows assigning priorities to groups of processes, not just individual ones. **Cgroups are a critical component of Linux containers**, able to enforce isolation and resource limits of all processes started within containers.

(As a result of all this complexity, the `ps` command can't really provide a single number for process priority. The value is just an approximation on some systems. I think that `ps` on Linux currently (2014) reports the static priority only.)

Processes (really, threads or tasks) have various ***states*** they can be in:

- *Runnable* — just waiting for an available CPU,
- *Running,*
- *Sleeping* or *waiting* or *blocked* — depending on what sort of event it is waiting for, such a process may be uninterruptible (waiting for hardware to respond) or interruptible (usually waiting for some signal),
- *Stopped* — either via a job control signal, or when tracing, and
- *Defunct* or *zombie* — terminated but not yet "reaped" by its parent.

The exact set of states and names for them vary by OS.

Using ulimit

The default for # of user processes per user varies by system from 3,000 to over 15,000. A *fork bomb* (type of *logic bomb*) can easily launch so many busy processes that the system crashes. While you need to see how many user processes are launched for various services you have running before adjusting this, for YborStudent 500 or even 100 is quite reasonable. Try with ulimit -S -u 500. (To set for all users, set in `/etc/security/limits.conf`.)

Memory Management

An understanding of how modern operating systems manage memory is useful for both software developers as well as system admins. Besides alerting a sys admin to insufficient RAM in a host, changes in memory use, page faults, swapping/paging, are all indicators of unusual activity (either faulty software or perhaps malware).

Process' Use of Memory

The physical RAM on a system is subdivided into parts or *zones* with specific uses, e.g., for the kernel versus processes, or for each processor/core (NUMA = *non-uniform memory access*) for shared memory, or for DMA (I/O) use. (Use `numactl` on Linux to tune this.) The details (the *memory model*) vary by OS and is not discussed here.

Each processes' virtual memory is divided into *sections*, each used differently. On all systems, the virtual memory has the following four sections at least (see graphic below):

- The ***text*** section holds the machine code (the program) for the process.
- The ***data*** section holds global data (initialized data, e.g. constants). Having different sections for text and data means you can set attributes on each, such as *sharable* or *read-only*. However, setting such attributes may impact system performance (e.g., each write to memory must check if the page is RO).
- Each thread in a program also contains a section historically called ***bss***, which is uninitialized memory used to hold the chunks of memory called *objects*, that the program may request from the kernel as more memory is needed. This section is often called the ***heap***.
- Finally, an execution ***stack*** section holds data local to functions in the program. As a function is invoked, a block of memory is reserved for its local variables, parameters passed to / from the function, and (unfortunately from a security standpoint) information needed to return to the previous function.

  Some OSes keep a second stack per thread, called the ***kernel* stack**. This is used when a thread invokes some kernel system call rather than a regular function.

The bss (heap) can be expanded over time, but the text, data, and stack sections are fixed in size when the process is created.

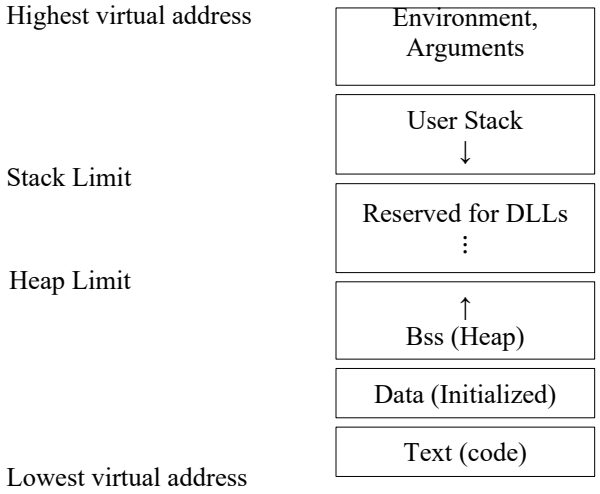
It's important to keep in mind that each 4 KiB of used actual memory (a page slot), is mapped to some virtual address (a page). A *page table* is maintained by the kernel for each process, stating which page slots contain which pages. The page table entry can contain additional info, such as whether the page is writable, executable, or marked for *copy on write*.

The virtual address space is very large and only sparsely used. Usually, the first page of virtual memory is unused (nothing maps to it), so any attempt to use `0` (*null*) as an address will be invalid. The text (code) pages start at the next slot, then the data, then the bss area. The bss (heap) has a max size, even though initially no memory is mapped there, all those virtual addresses are reserved. As memory is requested by the program, pages of zeroed physical memory are mapped to the virtual addresses reserved for the bss.

At the other end (the maximum virtual address, usually 4 GiB on 32-bit systems), pages are mapped that hold the environment and command line arguments. Below that are the virtual addresses reserved for the user mode stack. (On some systems, other virtual addresses are reserved for the kernel's use, and contain a kernel stack and other data. On older *nix systems, user mode virtual addresses run from 1 to 2 GiB, and the kernel mode addresses run from 2 GiB to 4 GiB.)

The vast number of virtual addresses between the bss (heap) and the user mode stack are reserved for mapping the text pages of DLLs.

While details may vary be OSes, the following diagram shows a typical process' use of virtual memory:



Use the command `pmap -x PID` to see the (virtual) memory usage of some process. On Linux, `pmap` show the heap as a bunch of "anon"

blocks, one for each chunk of memory requested by the process. You can also see memory details from `/proc/PID/status`.

The kernel's memory management was discussed in CTS-2301C. The key concepts and terms were: *physical and virtual memory*, *swapping* and *paging*, and *page faults*. Memory is first reserved for the kernel's use. Another chunk is reserved for shared memory (the text sections of DLLs for example). The rest is used for process' memory and for disk and I/O buffers. However, there is more to the story (isn't there always?).

A typical process can be quite large, using megabytes of memory. However, it won't need all of that data at the same time. Initially only the first page of instructions need be copied from disk into memory. As the program runs, other pages of instructions and data will be needed. Such a request will trigger a ***major page fault***, which means data must be copied from disk into memory before the process can proceed. These are the expensive page faults that you need to minimize. You can't eliminate all major page faults, since the pages must be copied from disk at least once. You want to avoid swapping pages out, and then needing to swap them back in again later. (See below.)

A ***minor page fault*** occurs when pages can be shared. If some DLL is already in memory, but not listed in your process' page table, your process can't access it. (Recall DLLs are assigned virtual addresses when the program using it was compiled. So it has a virtual address.) In this case a minor page fault occurs, which simply updates the page table. (The kernel also needs to know how many processes are using some page in memory, so it knows when that page slot becomes available for reuse.) Minor page faults don't involve storage, and have little impact on your system's performance.

An interesting performance issue is when loading DLLs from disk into RAM. If every process compiles the DLL at a different virtual address, the kernel can't simply copy the page table entry to satisfy the minor page fault. However, if all the applications using some DLL used the same virtual address for it, the minor page fault is much faster. This is the purpose of the `prelink` utility, which is often run on Linux when installing an RPM package of a DLL; many existing programs that use that DLL need to be prelinked again. (You sometimes get security alerts from that, saying some application has changed even though you didn't update that application directly.)

A virtual memory page can be in one of several states: not loaded into RAM, resident in RAM, unallocated, and COW. For instructions or data files on disk, the data is *mapped* into RAM. But a process can also simply ask the OS for (say) 8 MiB of memory. A modern OS will not actually reserve any RAM for such a request; it just tells the process "no problem", and creates page table entries

marked as uninitialized. Only when the process attempts to write data to those memory locations is a page slot actually allocated; attempts to read that memory are intercepted and zeroes are returned. COW is similar to unallocated. Consider the pages devoted to holding a processes initial data, or its environment variables. The chances are good that a page with that exact data is already in memory. Rather than load it again, or even make a copy of it, the page table entry will point to the already loaded page slot, but mark the page table entry as *copy on write*. So the first attempt to modify that memory will trigger a copy, which is then modified. Until that first write, the page is shared.

The kernel keeps information on every page, such as if that page is loaded already into some page slot, or if it was swapped, or if it is shared, read-only, COW, and so on.

> A *memory-mapped file* is RAM that the kernel will treat specially. It will load some file into RAM one page at a time as usual. However, every write to such RAM causes the underlying file to be updated as well.

No matter how much RAM you have, your system can use it ineffectively and negatively affect performance. The default values are not appropriate for virtualization and may even cause a *panic*!

An SA should monitor how much swap (page) space is used and how many major page faults occur. You can use `vmstat` and other tools (see system monitoring). *Demo:* "`ps -eo min_flt,maj_flt,cmd`". (*Also demo:* `/usr/bin/time -v links http://wpollock.com/`)

Swapping (paging) If a system is heavily loaded and a page fault requires a page to be swapped-in, there may not be enough free memory. At first the kernel will try to shrink the filesystem cache (discussed below) to free up some page slots. But that might not make sufficient page slots available. So to satisfy the swap-in, the kernel must first swap-out some other page. If the kernel guesses the wrong page to swap-out, it will need to be swapped-in soon, causing some other page to be swapped-out. A system that continuously swaps is *thrashing*, and will suffer a large performance loss. You can use the command `top` to see how much swap space is being used on your system, and `vmstat` to see the current numbers of swap-in ("si") and swap-out ("so") operations. Try: "`vmstat 1`".

If these values are high, it may mean some process is hogging the memory by growing its heap and causing the rest of the system to suffer. This can be controlled by adjusting the limits, or by increasing the kernel's `vm.swappiness` parameter (see below). However, this is not a big problem anymore since a process that attempts to grow the heap but not write to it, will not cause any page slots to be allocated at that time. (Remember those virtual pages are *uninitialized*.)

Another feature of memory management is the ***filesystem cache***. Performance is generally improved if recently read file data is kept in memory even after no process is using it. This is because some files are frequently used by different processes. If the data is already in RAM, to access it is only a minor page fault instead of a major one. All modern OSes will use some of the RAM for such a cache, but the question is, how much to use? If the cache is too small, performance will suffer. If too large, swapping may occur and that also hurts performance. Most modern OSes will use *all* memory not allocated to the kernel or processes for this cache, but will shrink the cache (to some minimum size) if a process needs some.

Look at the output of the `free -m` command (similar to `top` output):

```
      total  used  free  shared  buff/cache  available
Mem:   2000   319   149       1        1532       1600
Swap:  2047   225  1822
```

It may appear that this system has 2000MB of RAM and is only has 149 MiB available. Don't rush out to buy more RAM just yet! 1600MiB are used for the filesystem cache and for kernel buffers. If applications start using more memory, the OS will satisfy such requests by shrinking the size of the filesystem cache.

Thus, it is recommended to say that this machine is using 319MiB and has 1600MiB available. However, because of the importance of the filesystem cache to performance, the system may prefer to swap out rather than shrink the cache! There are a number of parameters that a SysAdmin can tweak to attempt to improve performance; see `proc(5)` and the Linux kernel vm.txt document.

Kernel's Use of Memory

The kernel keeps a large number of items cached in memory. Often this is needless but the SysAdmin sees less memory available. (Actually, all kernels will free up such cached items when an app requests additional memory.)

One example is the cached inodes and directory entries (*dentry*). Sometimes these don't get flushed quickly, without unmounting the storage volume. To free pagecache, dentries and inodes: `echo 3 > /proc/sys/vm/drop_caches`. This is a non-destructive operation. Since any "dirty" objects (those memory pages that have been modified but not yet saved to disk) are not freeable, dirty pages won't be dropped. To free up the maximum number of pages, a SysAdmin should run `sync` first. (Running `sync` first should also make `df` output more accurate, since its output includes data in the caches.)

When free memory is low, the Linux kernel will swap out some pages "just in case" in the background. (This makes the swap-ins twice as fast since no swap-out is needed first.) The Linux kernel's aggressiveness in preemptively swapping-out pages is governed by a kernel parameter called **`vm.swappiness`**. It can be set to a number from 0 to 100, where 0 means that more is kept in memory and 100 means that the kernel should try to swap-out as many pages as possible. The default value is 60.

> Kernel maintainer Andrew Morton has stated that he uses a swappiness of 100 on his desktop machines, "my point is that decreasing the tendency of the kernel to swap stuff out is wrong. You really don't want hundreds of megabytes of BloatyApp's untouched memory floating about in the machine. Get it out on the disk, use the memory for something useful." (See Ref)

To see which kernel subsystems are using memory on Linux, try `slabtop`.

Out of Memory (OOM) Killer

If memory is overcommitted and some more is actually used, the kernel will start killing off processes rather than crash, done by the *out of memory* "OOM" killer. By default, only a percentage of physical RAM is allowed to be overcommitted. So you may have spare gigabytes and still get the OOM killer running, even when it would be safe to overcommit more.

The OOM killer tries to kill "bad" processes first. Memory used is the primary number determining a process' badness score. This is somewhat configurable (you can designate preferred processes to kill first).

Summary

Use tools such as `free`, `vmstat`, and `top` to see data on your systems overall memory use (amount, swaping activity, cache sizes). Watch for unexpected changes. Due to caching by the kernel, the free memory shown is usually not the amount of available memory.

The largest impact on performance from your memory system will be the number of major page faults. Swapping causes these. Excessive swapping is called thrashing, which means some page must be swapped out before some needed page can be swapped in. These are the major page faults you should monitor and try to eliminate.

Minor page faults do not cause much of a performance issue, although prelinking executables may improve performance of these. There are also kernel VM parameters you can tweak. to try to improve performance.

Lecture 19 — Locales

Different countries and cultures have varying conventions for how to communicate. Internationalization of software means programming it to be able to adapt to the user's favorite conventions. With Unix, internationalization (or "**I18N**") works (mostly) by means of *locales*. (*Localization* is sometimes abbreviated as "**L10N**".)

A locale is a set of language, geographical, and cultural rules and conventions. These conventions include simple ones (such as the format for representing dates and times) and complex ones (such as the language used). They cover aspects such as the language used for messages, different character sets (the encoding, e.g., "UTF-8"), and lexicographic conventions such as the format of dates and time, currency, other numbers, names, phone numbers, and addresses. Related conventions are grouped into ***categories***.

> Time zones are not part of a locale even though the format of dates and time are. This may be due to the fact that a given region may have multiple time zones (the U.S includes 6 different ones).

The user chooses a set of conventions to use by specifying a locale (via environment variables) for each category. (You can specify different locales for different categories if you want; no need to be consistent!) POSIX systems include these six categories and eight environment variables (*from locale(7)*):

`LC_COLLATE`	This is used to change the behavior of the functions used to compare strings in the local alphabet. For example, the German sharp s ("ß") is sorted as "ss".
`LC_CTYPE`	This changes the behavior of the character handling and classification functions, such as `isupper()` and `toupper()`. (Regular expressions such as `[:upper:]` or `[:punct:]` are affected.)
`LC_MONETARY`	This changes monetary formats for currency display.
`LC_MESSAGES`	This changes the language system (and other) messages are displayed in (also man pages), and what an affirmative or negative answer looks like. The GNU gettext family of functions also obey the environment variable **`LANGUAGE`**. Message catalogs for various languages are found in the directories listed in **`$NLSPATH`**. (See below for more info.)

`LC_NUMERIC`	This changes the way non-monetary numbers are usually displayed, with details such as decimal point versus decimal comma and thousands grouping.
`LC_TIME`	This changes the display of the current time in a locally acceptable form; for example, most of Europe uses a 24-hour clock versus the 12-hour clock used in the United States. (Oddly there is no standard locale for ISO-8601 dates.)
`LANG`	Provides a default setting for any of the above, if unset.
`LC_ALL`	All of the above. Setting this environment variable overrides all other locale settings.

`LC_MESSAGES` is a bit different from the others. It specifies the format of text, really just the language to use). However setting this doesn't magically translate text from one language to another. I18N works by having I18N applications use *catalogs* for all text. **A *catalog* is a file of named text strings in some given language.** To support both French and English you would need two catalogs. The program then determines which catalog to use from the locale setting, finds the correct catalog, and looks up each bit of text by using the names. (Of course, not all programs are internationalized, and even when using such a program, you may not have the correct catalog installed even when you do have the correct locale installed.

The **`NLSPATH`** environment variable lists where to look for message catalogs. It is a format string (similar to `printf`). More than one directory can be specified (separating them by colons). The default value is:

```
prefix/share/locale/%L/%N: \
prefix/share/locale/%L/LC_MESSAGES/%N
```

where *prefix* is configured when installing system (usually either `/usr` or the empty string). Any "`%`*x*" in the string is substituted as follows:

`%N` The name of the catalog file.
`%L` The name of the currently selected locale for translating messages.
`%l` (Lowercase ell.) The language part of the currently selected locale.
`%t` The territory part of the currently selected locale.
`%c` The codeset part of the currently selected locale.
`%%` A literal percent sign.

Changing the locale will change how time and dates appear in command output, the sorting order used by `ls`, `sort`, etc., and even the language of

system error messages and man pages! ***(Demo: `LANG=fr_FR.utf-8 i18n_greet.sh`)***

Demo: `ls -a; LANG=en_US; ls -a; LANG=es_US man man`. Show **`locale`** command and "`set |egrep 'LC_|LANG'`". Note, only installed locales ("`locale -a`") will work, and only for the commands that support them.

The original six categories and rules for each were defined by POSIX. Later ISO-14652 built upon that and expanded it, adding categories for names, addresses, phone numbers, and others. (They also renamed "locale" to "FDCC-set".) The Gnu `glibc` locale support was (and is) designed for an early draft of ISO-14652 (that included `LC_PAPER`), but most *nix systems just use the POSIX categories (and POSIX utilities `locale` and `localdef`). Some of these non-POSIX categories (and corresponding environment variables) that you'll find on Linux systems include:

`LC_NAME`	display of a person's name: family, surname, use of middle names, ...
`LC_ADDRESS`	display of postal addresses
`LC_TELEPHONE`	
`LC_MEASUREMENT`	units of measurement, e.g. gallons or liters, inches or centimeters, ...
`LC_IDENTIFICATION`	has no effect on locale. This is information on a locale itself, including the name, the categories it addresses, author and contact information, etc.
`LC_PAPER`	standard paper size, e.g. A4 or US Letter
`LC_TOD`	Timezone information (IBM defines this one)

Any system includes a (often large) set of locales you can use. However if you attempt to set a locale not included on your system some commands won't work as intended, or at all.

> ICU (*International Components for Unicode*) is another set of popular localization standards that are similar to but not identical to POSIX locales. See userguide.icu-project.org/locale for information on ICU locales.
>
> In addition, some applications use internal locale definitions, not system-wide ones. This includes all Java applications.

The command "**`locale -a`**" will list all available locales. It is also possible to find additional locales for your system, or to create new ones using the **`localedef`** (POSIX) command. Note that installing a locale may not install the NLS support for that language (that is, installing the "`es`" locale

(for Spanish) won't install the Spanish version of man pages or system error messages).

All systems support a **POSIX** locale, also called the **C** locale, which was the only locale for most Unix systems until the late (mid?) 1990s.

> **Warning!** Range character classes used in regular expressions are only guaranteed to work for `LC_COLLATE=POSIX` locale! From Bash try "`ls [!A-Z]*`" with `en_US` locale. To avoid this error I've set `LC_COLLATE=C` (and `LANG=en_US.utf8`) on YborStudent.

Locales are identified by language tags (RFC-5646/BCP-47) and character set names (a.k.a. charmap or codeset). Most commonly a locale name has the form:

`language[_territory][.codeset][@modifier]`

where *language* is an ISO 639 language code (2 letter, or 3 letter if no 2 letter code), *territory* is an ISO 3166 2 letter country code, and *codeset* is a character set identifier like "`ISO-8859-1`" or "`utf8`" (a *partial* list is at `iana.org/assignments/character-sets`). A common example is "`en_US.utf8`". The modifier defines a variant, e.g. using `en_US.utf8@mil` for military date and time formats, or "`fr_FR@euro`" to use `fr_FR` but with euro currency. Note the "@" may need to be escaped in shell.

The codeset names must be correct. On Fedora for example there is no locale with the name "`en_US.UTF-8`". Oddly the **`locale -m`** command lists available charmap names, not the codeset names used in locales! **To see which codesets are available for some locale, say `en_US`, use "`locale -a |grep en_US`".**

A locale name containing any slashes is considered a pathname for a file defining the locale (e.g., "`LC_TIME=~/.locale/mylocale`"). Without slashes the locale is found in an implementation-specific way, usually a file in a standard directory such as `/usr/lib64/locale`, `/usr/share/i18n/locales`, or some such directory. Some systems will look for locale files in the directories listed in **`I18NPATH`** but that is not part of POSIX. Note these files generally don't include the codeset (or charmap) in the name. (On Linux glibc implements locale support, so look for glibc packages, then see the locale pathnames they use: rpm -qa |grep glib, and for each: rpm -ql *name* |grep locale)

On Fedora, the default locale for daemons is in **`/etc/sysconfig/i18n`**. (For systemd based systems, the file is **`/etc/locale.conf`**; the older file will be used if the newer one isn't found.) For interactive logins, the file

/etc/profile.d/lang.* is used (sourced) afterward. Note the various GUI systems (e.g., KDE) may also have their own locale settings, which often may not default to using English.

As a post-install task, the SA should set appropriate defaults for daemons, logins, and the GUI.

User can override locale settings by setting and exporting environment variables, or creating a file such as **~/.i18n** and sourcing that from a login script (`profile.d/lang.sh` does that on Fedora).

> The *charmap* (a.k.a. codeset or encoding) used needs to match the one expected by your terminal emulator. So if the locale is set to "`en_US.iso88591`" then with PuTTY, change Windows-->translation to "`ISO-8859-1`", or the non-ASCII characters won't display correctly.

Creating and Customizing Locales — *Only if time and interest allow; not on test*

To use a non-standard locale you must find or create the locale source file, edit that, rename the result, and compile it, and put the result where the `setlocale(3)` function can find it.

> The standard locations for locale files vary by system. For modern Fedora (and many other glibc systems), the standard locale data is put into a single archive file `/usr/lib64/local/local-archive`. Other compiled locale files go into subdirectories (one per locale, each containing one or more files, one for each category) of `/usr/share/locale`. The source files for locales and charmaps are usually found in `/usr/share/i18n/locale`.
>
> The `localedef` utility should show the system specific directories in the help output. In addition, you can define some environment variables that affects where the system will look: `I18NPATH` is used by `localedef` to find the locale source files, when you didn't list a pathname on the command line; `LOCPATH` is used by non-SUID programs on Linux to locate (compiled) locale files. You can use that to test your own locales before moving them into the system standard location.

It is preferable to save your locale data in UTF-8 and then encode it with the <UNNNN> format used in glibc locale files. Use the following script for this (lifted from the glibc locales mailing list):

$ echo ĐÑÑÑĐĐĐ |iconv -f UTF-8 -t ASCII --unicode-subst='<U%04X>'

<U0420><U0443><U0441><U0441><U043A><U0438><U0439>

All glibc locales should contain a comment like this at the top:

```
% Charset: UTF-8
```

(Just after the heading that defines the comment character.)

To have a custom setting you must define a new locale. Start with the source for a similar locale and create a *variant*, e.g., "`en_US.utf8@iso8601`". If you don't have the source you can use **`locale -k variable`** to see the current values; save that in a file for each locale variable you wish to change. Edit the file. Then make a directory to hold the resulting (up to six) files in someplace such as `~/.locale`. Compile your source file with:

```
localedef -c -f UTF-8 -i your-file en_US@iso8601
```

You need to have both locale and charmap sources. Put the locale sources in `/usr/share/i18n/locales/` and the charmap sources in `/usr/share/i18n/charmaps/`. Execute the `localedef` program to build the locale data files.

Move the resulting directory to the default location (see `localedef --help` and look for the "`locale path`" line) to allow others to use it by name: `en_US.utf8@iso8601`. The source file doesn't have to define all categories, but if it defines only one (say `LC_TIME`) it can only be used for that category. (So don't use it for `LANG=` or `LC_ALL=`.)

X11 does use system locales, but a number of files must be updated before any new or modified locales can be used. The location of these files varies by system; for Fedora look in `/usr/share/X11/locale/`.

`locale.alias` should list every variant (alias) of your new locales. (Alias names of locales should not be used, but this file is here to support older programs that used them.) There is the standard system `local.alias` file as well as one that X uses. Update both!

`compose.dir` and `locale.dir` also need updating.

Different desktop environments also define their own locale systems. Look in and `/etc/gdm/locale*` and KDE's `entry.desktop` files.

Brief Overview of Solaris Role Based Access Control (RBAC)

A ***role*** is a pseudo account that authorized users can use with `su` to assume some *profile*(s). These role accounts use special versions of the shell (`pfsh`, `pfcsh`, or `pfksh`). Example roles include password admin, mail admin, backup admin, login admin, mount, shutdown, ... admin, and service (start or stop) admin.

A ***profile*** is a collection of the Solaris ***authorizations*** (or *privileges*) which are listed in the file `/etc/security/auth_attr`. Profiles are defined in `.../prof_attr`. The actual commands usable in a profile and the GID and UID to use when running those commands are defined in `.../exec_attr`. The file `.../user_attr` associates user accounts with *roles*, and roles with profiles (although users can be assigned profiles or even individual authorizations, this is not as safe nor easy to administer).

You need to restart `nscd`, name service cache demon for changes to roles and profiles to take effect. Use sm* and role* commands to edit the files. (**Show RBAC.htm**.)

Demo: `roles`, `profiles`, **`auths|sed 's/,/ /g'|fold -sw 30|sort`**

RBAC provides similar functionality as `sudo`, except that it is supported by Sun (`sudo` is a third-party product provided as-is) and can use NIS, LDAP, ... to centralize the RBAC DB files. `sudo` is easier to configure (control is at the command line level) and probably a better solution, but not all organizations will allow third party non-supported software. RBAC is also available for Linux.