The Unix and Internet Fundamentals HOWTO

Eric Raymond

esr@thyrsus.com

Revision History

Revision 2.0	5 August 2000	Revised by: esr
First DocBook version. Detailed description of memory hierarchy.		
Revision 1.7	6 March 2000	Revised by: esr
Correct and expanded the section on file permissions.		
Revision 1.4	25 September 1999	Revised by: esr
Be more precise about what kernel does vs. what init does.		
Revision 1.3	27 June 1999	Revised by: esr
The sections `What happens when you log in?' and `File ownership, permissions and security'.		
Revision 1.2	26 December 1998	Revised by: esr
The section `How does my computer store things in memory?'.		
Revision 1.0	29 October 1998	Revised by: esr
Initial revision.		

This document describes the working basics of PC−class computers, Unix−like operating systems, and the Internet in non−technical language.

Table of Contents

1. Introduction

1.1. Purpose of this document

This document is intended to help Linux and Internet users who are learning by doing. While this is a great way to acquire specific skills, sometimes it leaves peculiar gaps in one's knowledge of the basics −− gaps which can make it hard to think creatively or troubleshoot effectively, from lack of a good mental model of what is really going on.

I'll try to describe in clear, simple language how it all works. The presentation will be tuned for people using Unix or Linux on PC−class hardware. Nevertheless I'll usually refer simply to `Unix' here, as most of what I will describe is constant across platforms and across Unix variants.

I'm going to assume you're using an Intel PC. The details differ slightly if you're running an Alpha or PowerPC or some other Unix box, but the basic concepts are the same.

I won't repeat things, so you'll have to pay attention, but that also means you'll learn from every word you read. It's a good idea to just skim when you first read this; you should come back and reread it a few times after you've digested what you have learned.

This is an evolving document. I intend to keep adding sections in response to user feedback, so you should come back and review it periodically.

1.2. Related resources

If you're reading this in order to learn how to hack, you should also read the How To Become A Hacker FAQ. It has links to some other useful resources.

1.3. New versions of this document

New versions of the Unix and Internet Fundamentals HOWTO will be periodically posted to comp.os.linux.help and news:comp.os.linux.announce and news.answers. They will also be uploaded to various Linux WWW and FTP sites, including the LDP home page.

You can view the latest version of this on the World Wide Web via the URL http://metalab.unc.edu/LDP/HOWTO/Unix−Internet−Fundamentals−HOWTO.html.

1.4. Feedback and corrections

If you have questions or comments about this document, please feel free to mail Eric S. Raymond, at esr@thyrsus.com. I welcome any suggestions or criticisms. I especially welcome hyperlinks to more detailed explanations of individual concepts. If you find a mistake with this document, please let me know so I can correct it in the next version. Thanks.

---

2. Basic anatomy of your computer

Your computer has a processor chip inside it that does the actual computing. It has internal memory (what DOS/Windows people call ``RAM'' and Unix people often call ``core''; the Unix term is a folk memory from when RAM consisted of ferrite–core donuts). The processor and memory live on the *motherboard* which is the heart of your computer.

Your computer has a screen and keyboard. It has hard drives and floppy disks. The screen and your disks have *controller cards* that plug into the motherboard and help the computer drive these outboard devices. (Your keyboard is too simple to need a separate card; the controller is built into the keyboard chassis itself.)

We'll go into some of the details of how these devices work later. For now, here are a few basic things to keep in mind about how they work together:

All the inboard parts of your computer are connected by a *bus*. Physically, the bus is what you plug your controller cards into (the video card, the disk controller, a sound card if you have one). The bus is the data highway between your processor, your screen, your disk, and everything else.

The processor, which makes everything else go, can't actually see any of the other pieces directly; it has to talk to them over the bus. The only other subsystem it has really fast, immediate access to is memory (the core). In order for programs to run, then, they have to be *in core* (in memory).

When your computer reads a program or data off the disk, what actually happens is that the processor uses the bus to send a disk read request to your disk controller. Some time later the disk controller uses the bus to signal the processor that it has read the data and put it in a certain location in memory. The processor can then use the bus to look at that data.

Your keyboard and screen also communicate with the processor via the bus, but in simpler ways. We'll discuss those later on. For now, you know enough to understand what happens when you turn on your computer.

3. What happens when you switch on a computer?

A computer without a program running is just an inert hunk of electronics. The first thing a computer has to do when it is turned on is start up a special program called an *operating system*. The operating system's job is to help other computer programs to work by handling the messy details of controlling the computer's hardware.

The process of bringing up the operating system is called *booting* (originally this was *bootstrapping* and alluded to the difficulty of pulling yourself up ``by your bootstraps''). Your computer knows how to boot because instructions for booting are built into one of its chips, the BIOS (or Basic Input/Output System) chip.

The BIOS chip tells it to look in a fixed place on the lowest−numbered hard disk (the *boot disk*) for a special program called a *boot loader* (under Linux the boot loader is called LILO). The boot loader is pulled into memory and started. The boot loader's job is to start the real operating system.

The loader does this by looking for a *kernel*, loading it into memory, and starting it. When you boot Linux and see "LILO" on the screen followed by a bunch of dots, it is loading the kernel. (Each dot means it has loaded another *disk block* of kernel code.)

(You may wonder why the BIOS doesn't load the kernel directly −− why the two−step process with the boot loader? Well, the BIOS isn't very smart. In fact it's very stupid, and Linux doesn't use it at all after boot time. It was originally written for primitive 8−bit PCs with tiny disks, and literally can't access enough of the disk to load the kernel directly. The boot loader step also lets you start one of several operating systems off different places on your disk, in the unlikely event that Unix isn't good enough for you.)

Once the kernel starts, it has to look around, find the rest of the hardware, and get ready to run programs. It does this by poking not at ordinary memory locations but rather at *I/O ports* −− special bus addresses that are likely to have device controller cards listening at them for commands. The kernel doesn't poke at random; it has a lot of built−in knowledge about what it's likely to find where, and how controllers will respond if they're present. This process is called *autoprobing*.

Most of the messages you see at boot time are the kernel autoprobing your hardware through the I/O ports, figuring out what it has available to it and adapting itself to your machine. The Linux kernel is extremely good at this, better than most other Unixes and *much* better than DOS or Windows. In fact, many Linux old−timers think the cleverness of Linux's boot−time probes (which made it relatively easy to install) was a major reason it broke out of the pack of free−Unix experiments to attract a critical mass of users.

But getting the kernel fully loaded and running isn't the end of the boot process; it's just the first stage (sometimes called *run level 1*). After this first stage, the kernel hands control to a special process called `init' which spawns several housekeeping processes.

The init process's first job is usually to check to make sure your disks are OK. Disk file systems are fragile things; if they've been damaged by a hardware failure or a sudden power outage, there are good reasons to take recovery steps before your Unix is all the way up. We'll go into some of this later on when we talk about how file systems can go wrong.

Init's next step is to start several *daemons*. A daemon is a program like a print spooler, a mail listener or a WWW server that lurks in the background, waiting for things to do. These special programs often have to coordinate several requests that could conflict. They are daemons because it's often easier to write one program that runs constantly and knows about all requests than it would be to try to make sure that a flock of copies (each processing one request and all running at the same time) don't step on each other. The particular collection of daemons your system starts may vary, but will almost always include a print spooler (a gatekeeper daemon for your printer).

The next step is to prepare for users. Init starts a copy of a program called **getty** to watch your console (and maybe more copies to watch dial−in serial ports). This program is what issues the **login** prompt to your console. Once all daemons and getty processes for each terminal are started, we're at *run level 2*. At this level, you can log in and run programs.

But we're not done yet. The next step is to start up various daemons that support networking and other services. Once that's done, we're at *run level 3* and the system is fully ready for use.

4. What happens when you log in?

When you log in (give a name to **getty**) you identify yourself to the computer. It then runs a program called (naturally enough) **login**, which takes your password and checks to see if you are authorized to be using the machine. If you aren't, your login attempt will be rejected. If you are, login does a few housekeeping things and then starts up a command interpreter, the *shell*. (Yes, **getty** and **login** could be one program. They're separate for historical reasons not worth going into here.)

Here's a bit more about what the system does before giving you a shell (you'll need to know this later when we talk about file permissions). You identify yourself with a login name and password. That login name is looked up in a file called /etc/passwd, which is a sequence of lines each describing a user account.

One of these fields is an encrypted version of the account password (sometimes the encrypted fields are actually kept in a second /etc/shadow file with tighter permissions; this makes password cracking harder). What you enter as an account password is encrypted in exactly the same way, and the **login** program checks to see if they match. The security of this method depends on the fact that, while it's easy to go from your clear password to the encrypted version, the reverse is very hard. Thus, even if someone can see the encrypted version of your password, they can't use your account. (It also means that if you forget your password, there's no way to recover it, only to change it to something else you choose.)

Once you have successfully logged in, you get all the privileges associated with the individual account you are using. You may also be recognized as part of a *group*. A group is a named collection of users set up by the system administrator. Groups can have privileges independently of their members' privileges. A user can be a member of multiple groups. (For details about how Unix privileges work, see the section below on permissions.)

(Note that although you will normally refer to users and groups by name, they are actually stored internally as numeric IDs. The password file maps your account name to a user ID; the `/etc/group` file maps group names to numeric group IDs. Commands that deal with accounts and groups do the translation automatically.)

Your account entry also contains your *home directory*, the place in the Unix file system where your personal files will live. Finally, your account entry also sets your *shell*, the command interpreter that **login** will start up to accept your commmands.

5. What happens when you run programs from the shell?

The shell is Unix's interpreter for the commands you type in; it's called a shell because it wraps around and hides the operating system kernel. The normal shell gives you the '$' prompt that you see after logging in (unless you've customized it to something else). We won't talk about shell syntax and the easy things you can see on the screen here; instead we'll take a look behind the scenes at what's happening from the computer's point of view.

After boot time and before you run a program, you can think of your computer of containing a zoo of processes that are all waiting for something to do. They're all waiting on *events*. An event can be you pressing a key or moving a mouse. Or, if your machine is hooked to a network, an event can be a data packet coming in over that network.

The kernel is one of these processes. It's a special one, because it controls when the other *user processes* can run, and it is normally the only process with direct access to the machine's hardware. In fact, user processes have to make requests to the kernel when they want to get keyboard input, write to your screen, read from or write to disk, or do just about anything other than crunching bits in memory. These requests are known as *system calls*.

Normally all I/O goes through the kernel so it can schedule the operations and prevent processes from stepping on each other. A few special user processes are allowed to slide around the kernel, usually by being given direct access to I/O ports. X servers (the programs that handle other programs' requests to do screen graphics on most Unix boxes) are the most common example of this. But we haven't gotten to an X server yet; you're looking at a shell prompt on a character console.

The shell is just a user process, and not a particularly special one. It waits on your keystrokes, listening (through the kernel) to the keyboard I/O port. As the kernel sees them, it echos them to your screen then passes them to the shell. When the kernel sees an `Enter' it passes your line of text to the shell. The shell tries to interpret those keystrokes as commands.

Let's say you type `ls' and Enter to invoke the Unix directory lister. The shell applies its built–in rules to figure out that you want to run the executable command in the file `/bin/ls'. It makes a system call asking the kernel to start /bin/ls as a new *child* process and give it access to the screen and keyboard through the kernel. Then the shell goes to sleep, waiting for ls to finish.

When **/bin/ls** is done, it tells the kernel it's finished by issuing an *exit* system call. The kernel then wakes up the shell and tells it it can continue running. The shell issues another prompt and waits for another line of input.

Other things may be going on while your `ls' is executing, however (we'll have to suppose that you're listing a very long directory). You might switch to another virtual console, log in there, and start a game of Quake, for example. Or, suppose you're hooked up to the Internet. Your machine might be sending or receiving mail while **/bin/ls** runs.

---

6. How do input devices and interrupts work?

Your keyboard is a very simple input device; simple because it generates small amounts of data very slowly (by a computer's standards). When you press or release a key, that event is signalled up the keyboard cable to raise a *hardware interrupt*.

It's the operating system's job to watch for such interrupts. For each possible kind of interrupt, there will be an *interrupt handler*, a part of the operating system that stashes away any data associated with them (like your keypress/keyrelease value) until it can be processed.

What the interrupt handler for your keyboard actually does is post the key value into a system area near the bottom of memory. There, it will be available for inspection when the operating system passes control to whichever program is currently supposed to be reading from the keyboard.

More complex input devices like disk or network cards work in a similar way. Above, we referred to a disk controller using the bus to signal that a disk request has been fulfilled. What actually happens is that the disk raises an interrupt. The disk interrupt handler then copies the retrieved data into memory, for later use by the program that made the request.

Every kind of interrupts has an associated *priority level*. Lower−priority interrupts (like keyboard events) have to wait on higher−priority interrupts (like clock ticks or disk events). Unix is designed to give high priority to the kinds of events that need to be processed rapidly in order to keep the machine's response smooth.

In your OS's boot−time messages, you may see references to *IRQ* numbers. You may be aware that one of the common ways to misconfigure hardware is to have two different devices try to use the same IRQ, without understanding exactly why.

Here's the answer. IRQ is short for "Interrupt Request". The operating system needs to know at startup time which numbered interrupts each hardware device will use, so it can associate the proper handlers with each one. If two different devices try use the same IRQ, interrupts will sometimes get dispatched to the wrong handler. This will usually at least lock up the device, and can sometimes confuse the OS badly enough that it will flake out or crash.

7. How does my computer do several things at once?

It doesn't, actually. Computers can only do one task (or *process*) at a time. But a computer can change tasks very rapidly, and fool slow human beings into thinking it's doing several things at once. This is called *timesharing*.

One of the kernel's jobs is to manage timesharing. It has a part called the *scheduler* which keeps information inside itself about all the other (non−kernel) processes in your zoo. Every 1/60th of a second, a timer goes off in the kernel, generating a clock interrupt. The scheduler stops whatever process is currently running, suspends it in place, and hands control to another process.

1/60th of a second may not sound like a lot of time. But on today's microprocessors it's enough to run tens of thousands of machine instructions, which can do a great deal of work. So even if you have many processes, each one can accomplish quite a bit in each of its timeslices.

In practice, a program may not get its entire timeslice. If an interrupt comes in from an I/O device, the kernel effectively stops the current task, runs the interrupt handler, and then returns to the current task. A storm of high−priority interrupts can squeeze out normal processing; this misbehavior is called *thrashing* and is fortunately very hard to induce under modern Unixes.

In fact, the speed of programs is only very seldom limited by the amount of machine time they can get (there are a few exceptions to this rule, such as sound or 3−D graphics generation). Much more often, delays are caused when the program has to wait on data from a disk drive or network connection.

An operating system that can routinely support many simultaneous processes is called "multitasking". The Unix family of operating systems was designed from the ground up for multitasking and is very good at it −− much more effective than Windows or the Mac OS, which have had multitasking bolted into it as an afterthought and do it rather poorly. Efficient, reliable multitasking is a large part of what makes Linux superior for networking, communications, and Web service.

8. How does my computer keep processes from stepping on each other?

The kernel's scheduler takes care of dividing processes in time. Your operating system also has to divide them in space, so that processes can't step on each others' working memory. Even if you assume that all programs are trying to be cooperative, you don't want a bug in one of them to be able to corrupt others. The things your operating system does to solve this problem are called *memory management*.

Each process in your zoo needs its own area of memory, as a place to run its code from and keep variables and results in. You can think of this set as consisting of a read−only *code segment* (containing the process's instructions) and a writeable *data segment* (containing all the process's variable storage). The data segment is truly unique to each process, but if two processes are running the same code Unix automatically arranges for them to share a single code segment as an efficiency measure.

8.1. Virtual memory: the simple version

Efficiency is important, because memory is expensive. Sometimes you don't have enough to hold the entirety of all the programs the machine is running, especially if you are using a large program like an X server. To get around this, Unix uses a technique called *virtual memory*. It doesn't try to hold all the code and data for a process in memory. Instead, it keeps around only a relatively small *working set*; the rest of the process's state is left in a special *swap space* area on your hard disk.

Note that in the past, that "Sometimes" last paragraph ago was "Almost always," −− the size of memory was typically small relative to the size of running programs, so swapping was frequent. Memory is far less expensive nowadays and even low−end machines have quite a lot of it. On modern single−user machines with 64MB of memory and up, it's possible to run X and a typical mix of jobs without ever swapping after they're initially loded into core.

8.2. Virtual memory: the detailed version

Actually, the last section oversimplified things a bit. Yes, programs see most of your memory as one big flat bank of addresses bigger than physical memory, and disk swapping is used to maintain that illusion. But your hardware actually has no fewer than five different kinds of memory in it, and the differences between them can matter a good deal when programs have to be tuned for maximum speed. To really understand what goes on in your machine, you should learn how all of them work,

The five kinds of memory are these: processor registers, internal (or on−chip) cache, external (or off−chip) cache, main memory, and disk. And the reason there are so many kinds is simple; speed costs money, I listed these kinds of memory in decreasing order of access time and cost; register memory is the fastest and most expensive and can be random−accessed about a billion times a second, while disk is the slowest and cheapest and can do about 100 random accesses a second.

Here's a full list reflecting early−2000 speeds and prices for a typical desktop machine. While speed and capacity will go up and prices will drop, you can expect these ratios to remain fairly constant −− and it's those ratios that shape the memory hierarchy.

Disk

Size: 13000MB Accesses: 100/sec

Main memory

Size: 256MB Accesses: 100M/sec

External cache

Size: 512KB Accesses: 250M/sec

Internal Cache

Size: 32KB Accesses: 500M/sec

Processor

Size: 28 bytes Accesses: 1000M/sec

We can't build everything out of the fastest kinds of memory. It would be way too expensive –– and even if it weren't, fast memory is volatile. That is, it loses its marbles when the power goes off. Thus, computers have to have hard disks or other kinds of non–volatile storage that retains data when the power goes off. And there's a huge mismatch between the speed of processors and the speed of disks. The middle three levels of the memory hierarchy (*internal cache*, *external cache*, and main memory) basically exist to bridge that gap.

Linux and other Unixes have a feature called *virtual memory*. What this means is that the operating system behaves as though it has much more main memory than it actually does. Your actual physical main memory behaves like a set of windows or caches on a much larger "virtual" memory space, most of which at any given time is actually stored on disk in a special zone called the *swap area*. Out of sight of user programs, the OS is moving blocks of data (called "pages") between memory and disk to maintain this illusion. The end result is that your virtual memory is much larger but not too much slower than real memory.

How much slower virtual memory is than physical depends on how well the operating system's swapping algorithms match the way your programs use virtual memory. Fortunately, memory reads and writes that are close together in time also tend to cluster in memory space. This tendency is called *locality*, or more formally *locality of reference* –– and it's a good thing. If memory references jumped around virtual space at random, you'd typically have to do a disk read and write for each new reference and virtual memory would be as slow as a disk. But because programs do actually exhibit strong locality, your operating system can do relatively few swaps per reference.

It's been found by experience that the most effective method for a broad class of memory–usage patterns is very simple; it's called LRU or the "least recently used" algorithm. The virtual–memory system grabs disk blocks into its *working set* as it needs them. When it runs out of physical memory for the working set, it dumps the least–recently–used block. All Unixes, and most other virtual–memory operating systems, use minor variations on LRU.

Virtual memory is the first link in the bridge between disk and processor speeds. It's explicitly managed by the OS. But there is still a major gap between the speed of physical main memory and the speed at which a processor can access its register memory. The external and internal caches address this, using a technique similar to virtual memory as we've described it.

Just as the physical main memory behaves like a set of windows or caches on the disk's swap area, the external cache acts as windows on main memory. External cache is faster (250M accesses per sec, rather than 100M) and smaller. The hardware (specifically, your computer's memory controller) does the LRU thing in the external cache on blocks of data fetched from the main memory. For historical regions, the unit of cache swapping is called a "line" rather than a page.

But we're not done. The internal cache gives us the final step−up in effective speed by caching portions of the external cache. It is faster and smaller yet −− in fact, it lives right on the processor chip.

If you want to make your programs really fast, it's useful to know these details. Your programs get faster when they have stronger locality, because that makes the caching work better. The easiest way to make programs fast is therefore to make them small. If a program isn't slowed down by lots of disk I/O or waits on network events, it will usually run at the speed of the largest cache that it will fit inside.

If you can't make your whole program small, some effort to tune the speed−critical portions so they have stronger locality can pay off. Details on techniques for doing such tuning are beyond the scope of this tutorial; by the time you need them, you'll be intimate enough with some compiler to figure out many of them yourself.

8.3. The Memory Management Unit

Even when you have enough physical core to avoid swapping, the part of the operating system called the *memory manager* still has important work to do. It has to make sure that programs can only alter their own data segments −− that is, prevent erroneous or malicious code in one program from garbaging the data in another. To do this, it keeps a table of data and code segments. The table is updated whenever a process either requests more memory or releases memory (the latter usually when it exits).

This table is used to pass commands to a specialized part of the underlying hardware called an *MMU* or *memory management unit*. Modern processor chips have MMUs built right onto them. The MMU has the special ability to put fences around areas of memory, so an out−of−bound reference will be refused and cause a special interrupt to be raised.

If you ever see a Unix message that says "Segmentation fault", "core dumped" or something similar, this is exactly what has happened; an attempt by the running program to access memory (core) outside its segment has raised a fatal interrupt. This indicates a bug in the program code; the *core dump* it leaves behind is diagnostic information intended to help a programmer track it down.

There is another aspect to protecting processes from each other besides segregating the memory they access. You also want to be able to control their file accesses so a buggy or malicious program can't corrupt critical pieces of the system. This is why Unix has file permissions which we'll discuss later.

9. How does my computer store things in memory?

You probably know that everything on a computer is stored as strings of bits (binary digits; you can think of them as lots of little on−off switches). Here we'll explain how those bits are used to represent the letters and numbers that your computer is crunching.

Before we can go into this, you need to understand about the the *word size* of your computer. The word size is the computer's preferred size for moving units of information around; technically it's the width of your processor's *registers*, which are the holding areas your processor uses to do arithmetic and logical calculations. When people write about computers having bit sizes (calling them, say, ``32−bit'' or ``64−bit'') computers, this is what they mean.

Most computers (including 386, 486, and Pentium PCs) have a word size of 32 bits. The old 286 machines had a word size of 16. Old−style mainframes often had 36−bit words. A few processors (like the Alpha from what used to be DEC and is now Compaq) have 64−bit words. The 64−bit word will become more common over the next five years; Intel is planning to replace the Pentium series with a 64−bit chip called the `Itanium'.

The computer views your memory as a sequence of words numbered from zero up to some large value dependent on your memory size. That value is limited by your word size, which is why older machines like 286s had to go through painful contortions to address large amounts of memory. I won't describe them here; they still give older programmers nightmares.

9.1. Numbers

Numbers are represented as either words or pairs of words, depending on your processor's word size. One 32−bit machine word is the most common size.

Integer arithmetic is close to but not actually mathematical base−two. The low−order bit is 1, next 2, then 4 and so forth as in pure binary. But signed numbers are represented in *twos−complement* notation. The highest−order bit is a *sign bit* which makes the quantity negative, and every negative number can be obtained from the corresponding positive value by inverting all the bits. This is why integers on a 32−bit machine have the range −2^31 + 1 to 2^31 − 1 (where ^ is the `power' operation, 2^3 = 8). That 32nd bit is being used for sign.

Some computer languages give you access to *unsigned arithmetic* which is straight base 2 with zero and positive numbers only.

Most processors and some languages can do in *floating−point* numbers (this capability is built into all recent processor chips). Floating−point numbers give you a much wider range of values than integers and let you express fractions. The ways this is done vary and are rather too complicated to discuss in detail here, but the general idea is much like so−called `scientific notation', where one might write (say) 1.234 * 10^23; the encoding of the number is split into a *mantissa* (1.234) and the exponent part (23) for the power−of−ten multiplier.

9.2. Characters

Characters are normally represented as strings of seven bits each in an encoding called ASCII (American Standard Code for Information Interchange). On modern machines, each of the 128 ASCII characters is the low seven bits of an 8−bit *octet*; octets are packed into memory words so that (for example) a six−character string only takes up two memory words. For an ASCII code chart, type `man 7 ascii' at your Unix prompt.

The preceding paragraph was misleading in two ways. The minor one is that the term `octet' is formally correct but seldom actually used; most people refer to an octet as *byte* and expect bytes to be eight bits long. Strictly speaking, the term `byte' is more general; there used to be, for example, 36−bit machines with 9−bit bytes (though there probably never will be again).

The major one is that not all the world uses ASCII. In fact, much of the world can't −− ASCII, while fine for American English, lacks many accented and other special characters needed by users of other languages. Even British English has trouble with the lack of a pound−currency sign.

There have been several attempts to fix this problem. All use the extra high bit that ASCII doesn't, making it the low half of a 256−character set. The most widely−used of these is the so−called `Latin−1' character set (more formally called ISO 8859−1). This is the default character set for Linux, HTML, and X. Microsoft Windows uses a mutant version of Latin−1 that adds a bunch of characters such as right and left double quotes in places proper Latin−1 leaves unassigned for historical reasons (for a scathing account of the trouble this causes, see the demoroniser page).

Latin−1 handles the major European languages, including English, French, German, Spanish, Italian, Dutch, Norwegian, Swedish, Danish. However, this isn't good enough either, and as a result there is a whole series of Latin−2 through −9 character sets to handle things like Greek, Arabic, Hebrew, Esperanto, and Serbo−Croatian. For details, see the ISO alphabet soup page.

The ultimate solution is a huge standard called Unicode (and its identical twin ISO/IEC 10646−1:1993). Unicode is identical to Latin−1 in its lowest 256 slots. Above these in 16−bit space it includes Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, the complete set of modern Korean Hangul, and a unified set of Chinese/Japanese/Korean (CJK) ideographs. For details, see the Unicode Home Page.

10. How does my computer store things on disk?

When you look at a hard disk under Unix, you see a tree of named directories and files. Normally you won't need to look any deeper than that, but it does become useful to know what's going on underneath if you have a disk crash and need to try to salvage files. Unfortunately, there's no good way to describe disk organization from the file level downwards, so I'll have to describe it from the hardware up.

10.1. Low−level disk and file system structure

The surface area of your disk, where it stores data, is divided up something like a dartboard −− into circular tracks which are then pie−sliced into sectors. Because tracks near the outer edge have more area than those close to the spindle at the center of the disk, the outer tracks have more sector slices in them than the inner ones. Each sector (or *disk block*) has the same size, which under modern Unixes is generally 1 binary K (1024 8−bit words). Each disk block has a unique address or *disk block number*.

Unix divides the disk into *disk partitions*. Each partition is a continuous span of blocks that's used separately from any other partition, either as a file system or as swap space. The original reasons for partitions had to do with crash recovery in a world of much slower and more error−prone disks; the boundaries between them reduce the fraction of your disk likely to become inaccessible or corrupted by a random bad spot on the disk. Nowadays, it's more important that partitions can be declared read−only (preventing an intruder from modifying critical system files) or shared over a network through various means we won't discuss here. The lowest−numbered partition on a disk is often treated specially, as a *boot partition* where you can put a kernel to be booted.

Each partition is either *swap space* (used to implement virtual memory or a *file system* used to hold files. Swap−space partitions are just treated as a linear sequence of blocks. File systems, on the other hand, need a way to map file names to sequences of disk blocks. Because files grow, shrink, and change over time, a file's data blocks will not be a linear sequence but may be scattered all over its partition (from wherever the operating system can find a free block when it needs one).

10.2. File names and directories

Within each file system, the mapping from names to blocks is handled through a structure called an *i−node*. There's a pool of these things near the ``bottom'' (lowest−numbered blocks) of each file system (the very lowest ones are used for housekeeping and labeling purposes we won't describe here). Each i−node describes one file. File data blocks live above the inodes (in higher−numbered blocks).

Every i−node contains a list of the disk block numbers in the file it describes. (Actually this is a half−truth, only correct for small files, but the rest of the details aren't important here.) Note that the i−node does *not* contain the name of the file.

Names of files live in *directory structures*. A directory structure just maps names to i−node numbers. This is why, in Unix, a file can have multiple true names (or *hard links*); they're just multiple directory entries that happen to point to the same inode.

11. Mount points

In the simplest case, your entire Unix file system lives in just one disk partition. While you'll see this arrangement on some small personal Unix systems, it's unusual. More typical is for it to be spread across several disk partitions, possibly on different physical disks. So, for example, your system may have one small partition where the kernel lives, a slightly larger one where OS utilities live, and a much bigger one where user home directories live.

The only partition you'll have access to immediately after system boot is your *root partition*, which is (almost always) the one you booted from. It holds the root directory of the file system, the top node from which everything else hangs.

The other partitions in the system have to be attached to this root in order for your entire, multiple−partition file system to be accessible. About midway through the boot process, your Unix will make these non−root partitions accessible. It will *mount* each one onto a directory on the root partition.

For example, if you have a Unix directory called \`/usr', it is probably a mount point to a partition that contains many programs installed with your Unix but not required during initial boot.

12. How a file gets looked up

Now we can look at the file system from the top down. When you open a file (such as, say, `/home/esr/WWW/ldp/fundamentals.sgml`) here is what happens:

Your kernel starts at the root of your Unix file system (in the root partition). It looks for a directory there called `home'. Usually `home' is a mount point to a large user partition elsewhere, so it will go there. In the top–level directory structure of that user partition, it will look for a entry called `esr' and extract an inode number. It will go to that i–node, notice it is a directory structure, and look up `WWW'. Extracting *that* i–node, it will go to the corresponding subdirectory and look up `ldp'. That will take it to yet another directory inode. Opening that one, it will find an i–node number for `fundamentals.sgml'. That inode is not a directory, but instead holds the list of disk blocks associated with the file.

12.1. File ownership, permissions and security

To keep programs from accidentally or maliciously stepping on data they shouldn't, Unix has *permission* features. These were originally designed to support timesharing by protecting multiple users on the same machine from each other, back in the days when Unix ran mainly on expensive shared minicomputers.

In order to understand file permissions, you need to recall our description of of users and groups in the section What happens when you log in?. Each file has an owning user and an owning group. These are initially those of the file's creator; they can be changed with the programs and .

The basic permissions that can be associated with a file are `read' (permission to read data from it), `write' (permission to modify it) and `execute' (permission to run it as a program). Each file has three sets of permissions; one for its owning user, one for any user in its owning group, and one for everyone else. The `privileges' you get when you log in are just the ability to do read, write, and execute on those files for which the permission bits match your user ID or one of the groups you are in.

To see how these may interact and how Unix displays them, let's look at some file listings on a hypothetical Unix system. Here's one:

```
snark:~$ ls -l notes
-rw-r--r--   1 esr      users         2993 Jun 17 11:00 notes
```

This is an ordinary data file. The listing tells us that it's owned by the user `esr' and was created with the owning group `users'. Probably the machine we're on puts every ordinary user in this group by default; other groups you commonly see on timesharing machines are `staff', `admin', or `wheel' (for obvious reasons, groups are not very important on single–user workstations or PCs). Your Unix may use a different default group, perhaps one named after your user ID.

The string `–rw–r––r––' represents the permission bits for the file. The very first dash is the position for the directory bit; it would show `d' if the file were a directory. After that, the first three places are user permissions, the second three group permissions, and the third are permissions for others (often called `world' permissions). On this file, the owning user `esr' may read or write the file, other people in the `users' group may read it, and everybody else in the world may read it. This is a pretty typical set of permissions for an ordinary data file.

Now let's look at a file with very different permissions. This file is GCC, the GNU C compiler.

```
snark:~$ ls -l /usr/bin/gcc
-rwxr-xr-x   3 root     bin         64796 Mar 21 16:41 /usr/bin/gcc
```

This file belongs to a user called `root' and a group called `bin'; it can be written (modified) only by root, but read or executed by anyone. This is a typical ownership and set of permissions for a pre–installed system command. The `bin' group exists on some Unixes to group together system commands (the name is a historical relic, short for `binary'). Your Unix might use a `root' group instead (not quite the same as the `root' user!).

The `root' user is the conventional name for numeric user ID 0, a special, privileged account that can override all privileges. Root access is useful but dangerous; a typing mistake while you're logged in as root can clobber critical system files that the same command executed from an ordinary user account could not touch.

Because the root account is so powerful, access to it should be guarded very carefully. Your root password is the single most critical piece of security information on your system, and it is what any crackers and intruders who ever come after you will be trying to get.

About passwords: Don't write them down –– and don't pick a passwords that can easily be guessed, like the first name of your girlfriend/boyfriend/spouse. This is an astonishingly common bad practice that helps crackers no end. In general, don't pick any word in the dictionary; there are programs called `dictionary crackers' that look for likely passwords by running through word lists of common choices. A good technique is to pick a combination consisting of a word, a digit, and another word, such as `shark6cider' or `jump3joy'; that will make the search space too large for a dictionary cracker. Don't use these examples, though –– crackers might expect that after reading this document and put them in their dictionaries.

Now let's look at a third case:

```
snark:~$ ls -ld ~
drwxr-xr-x  89 esr      users          9216 Jun 27 11:29 /home2/esr
snark:~$
```

This file is a directory (note the `d' in the first permissions slot). We see that it can be written only by esr, but read and executed by anybody else.

Read permission gives you the ability to list the directory –– that is, to see the names of files and directories it contains. Write permission gives you the ability to create and delete files in the directory. If you remember that the directory imcludes a list of the names of the files and subdirectories it contains, these rules will make sense.

Execute permission on a directory means you can get through the directory to open the files and directories below it. In effect, it gives you permission to access the inodes in thbe directory. A directory with execute completely turned off would be useless.

Occasionally you'll see a directory that is world–executable but not world–readable; this means a random user can get to files and directories beneath it, but only by knowing their exact names (the directory cannot be listed).

It's important to remember that read, write, or execute permission on a directory is independent of the permissions on the files and directories beneath. In particular, write access on a directory means you can create new files or delete existing files there, but ity does not automatically give you write access to existing files.

Finally, let's look at the permissions of the login program itself.

```
snark:~$ ls −l /bin/login
−rwsr−xr−x   1 root     bin         20164 Apr 17 12:57 /bin/login
```

This has the permissions we'd expect for a system command −− except for that 's' where the owner−execute bit ought to be. This is the visible manifestation of a special permission called the `set−user−id' or *setuid bit*.

The setuid bit is normally attached to programs that need to give ordinary users the privileges of root, but in a controlled way. When it is set on an executable program, you get the privileges of the owner of that program file while the program is running on your behalf, whether or not they match your own.

Like the root account itself, setuid programs are useful but dangerous. Anyone who can subvert or modify a setuid program owned by root can use it to spawn a shell with root privileges. For this reason, opening a file to write it automatically turns off its setuid bit on most Unixes. Many attacks on Unix security try to exploit bugs in setuid programs in order to subvert them. Security−conscious system administrators are therefore extra−careful about these programs and relucutant to install new ones.

There are a couple of important details we glossed over when discussing permissions above; namely, how the owning group and permissions are assigned when a file or directory is first created. The group is an issue because users can be members of multiple groups, but one of them (specified in the user's `/etc/passwd` entry) is the user's *default group* and will normally own files created by the user.

The story with initial permission bits is a little more complicated. A program that creates a file will normally specify the permissions it is to start with. But these will be modified by a variable in the user's environment called the *umask*. The umask specifies which permission bits to *turn off* when creating a file; the most common value, and the default on most systems, is −−−−−−−w− or 002, which turns off the world−write bit. See the documentation of the umask command on your shell's manual page for details.

Initial directory group is also a bit complicated. On some Unixes a new directory gets the default group of the creating user (this in the System V convention); on others, it gets the owning group of the parent directory in which it's created (this is the BSD convention). On some modern Unixes, including Linux, the latter behavior can be selected by setting the set−group−ID on the directory (chmod g+s).

There is a useful discussion of file permissions in Eric Goebelbecker's article Take Command.

12.2. How things can go wrong

Earlier we hinted that file systems can be fragile things. Now we know that to get to file you have to hopscotch through what may be an arbitrarily long chain of directory and i−node references. Now suppose your hard disk develops a bad spot?

If you're lucky, it will only trash some file data. If you're unlucky, it could corrupt a directory structure or i−node number and leave an entire subtree of your system hanging in limbo −− or, worse, result in a corrupted structure that points multiple ways at the same disk block or inode. Such corruption can be spread by normal file operations, trashing data that was not in the original bad spot.

Fortunately, this kind of contingency has become quite uncommon as disk hardware has become more reliable. Still, it means that your Unix will want to integrity−check the file system periodically to make sure nothing is amiss. Modern Unixes do a fast integrity check on each partition at boot time, just before mounting it. Every few reboots they'll do a much more thorough check that takes a few minutes longer.

If all of this sounds like Unix is terribly complex and failure−prone, it may be reassuring to know that these boot−time checks typically catch and correct normal problems *before* they become really disasterous. Other operating systems don't have these facilities, which speeds up booting a bit but can leave you much more seriously screwed when attempting to recover by hand (and that's assuming you have a copy of Norton Utilities or whatever in the first place...).

---

13. How do computer languages work?

We've already discussed how programs are run. Every program ultimately has to execute as a stream of bytes that are instructions in your computer's *machine language*. But human beings don't deal with machine language very well; doing so has become a rare, black art even among hackers.

Almost all Unix code except a small amount of direct hardware−interface support in the kernel itself is nowadays written in a *high−level language*. (The `high−level' in this term is a historical relic meant to distinguish these from `low−level' *assembler languages*, which are basically thin wrappers around machine code.)

There are several different kinds of high−level languages. In order to talk about these, you'll find it useful to bear in mind that the *source code* of a program (the human−created, editable version) has to go through some kind of translation into machine code that the machine can actually run.

13.1. Compiled languages

The most conventional kind of language is a *compiled language*. Compiled languages get translated into runnable files of binary machine code by a special program called (logically enough) a *compiler*. Once the binary has been generated, you can run it directly without looking at the source code again. (Most software is delivered as compiled binaries made from code you don't see.)

Compiled languages tend to give excellent performance and have the most complete access to the OS, but also to be difficult to program in.

C, the language in which Unix itself is written, is by far the most important of these (with its variant C++). FORTRAN is another compiled language still used among engineers and scientists but years older and much more primitive. In the Unix world no other compiled languages are in mainstream use. Outide it, COBOL is very widely used for financial and business software.

There used to be many other compiler languages, but most of them have either gone extinct or are strictly research tools. If you are a new Unix developer using a compiled language, it is overwhelmingly likely to be C or C++.

13.2. Interpreted languages

An *interpreted language* depends on an interpreter program that reads the source code and translates it on the fly into computations and system calls. The source has to be re−interpreted (and the interpreter present) each time the code is executed.

Interpreted languages tend to be slower than compiled languages, and often have limited access to the underlying operating system and hardware. On the other hand, they tend to be easier to program and more forgiving of coding errors than compiled languages.

Many Unix utilities, including the shell and bc(1) and sed(1) and awk(1), are effectively small interpreted languages. BASICs are usually interpreted. So is Tcl. Historically, the most important interpretive language has been LISP (a major improvement over most of its successors). Today Perl is very widely used and steadily growing more popular.

13.3. P−code languages

Since 1990 a kind of hybrid language that uses both compilation and interpretation has become increasingly important. P−code languages are like compiled languages in that the source is translated to a compact binary form which is what you actually execute, but that form is not machine code. Instead it's *pseudocode* (or *p−code*), which is usually a lot simpler but more powerful than a real machine language. When you run the program, you interpret the p−code.

P−code can run nearly as fast as a compiled binary (p−code interpreters can be made quite simple, small and speedy). But p−code languages can keep the flexibility and power of a good interpreter.

Important p−code languages include Python, Perl, and Java.

14. How does the Internet work?

To help you understand how the Internet works, we'll look at the things that happen when you do a typical Internet operation –– pointing a browser at the front page of this document at its home on the Web at the Linux Documentation Project. This document is

```
http://metalab.unc.edu/LDP/HOWTO/Fundamentals.html
```

which means it lives in the file LDP/HOWTO/Fundamentals.html under the World Wide Web export directory of the host metalab.unc.edu.

14.1. Names and locations

The first thing your browser has to do is to establish a network connection to the machine where the document lives. To do that, it first has to find the network location of the *host* metalab.unc.edu (`host' is short for `host machine' or `network host'; metalab.unc.edu is a typical *hostname*). The corresponding location is actually a number called an *IP address* (we'll explain the `IP' part of this term later).

To do this, your browser queries a program called a *name server*. The name server may live on your machine, but it's more likely to run on a service machine that yours talks to. When you sign up with an ISP, part of your setup procedure will almost certainly involve telling your Internet software the IP address of a nameserver on the ISP's network.

The name servers on different machines talk to each other, exchanging and keeping up to date all the information needed to resolve hostnames (map them to IP addresses). Your nameserver may query three or four different sites across the network in the process of resolving metalab.unc.edu, but this usually happens very quickly (as in less than a second).

The nameserver will tell your browser that Metalab's IP address is 152.2.22.81; knowing this, your machine will be able to exchange bits with metalab directly.

14.2. Packets and routers

What the browser wants to do is send a command to the Web server on Metalab that looks like this:

```
GET /LDP/HOWTO/Fundamentals.html HTTP/1.0
```

Here's how that happens. The command is made into a *packet*, a block of bits like a telegram that is wrapped with three important things; the *source address* (the IP address of your machine), the *destination address* (152.2.22.81), and a *service number* or *port number* (80, in this case) that indicates that it's a World Wide Web request.

Your machine then ships the packet down the wire (modem connection to your ISP, or local network) until it gets to a specialized machine called a *router*. The router has a map of the Internet in its memory –– not always a complete one, but one that completely describes your network neighborhood and knows how to get to the routers for other neighborhoods on the Internet.

Your packet may pass through several routers on the way to its destination. Routers are smart. They watch how long it takes for other routers to acknowledge having received a packet. They use that information to direct traffic over fast links. They use it to notice when another routers (or a cable) have dropped off the network, and compensate if possible by finding another route.

There's an urban legend that the Internet was designed to survive nuclear war. This is not true, but the Internet's design is extremely good at getting reliable performance out of flaky hardware in an uncertain world.. This is directly due to the fact that its intelligence is distributed through thousands of routers rather than a few massive switches (like the phone network). This means that failures tend to be well localized and the network can route around them.

Once your packet gets to its destination machine, that machine uses the service number to feed the packet to the web server. The web server can tell where to reply to by looking at the command packet's source IP address. When the web server returns this document, it will be broken up into a number of packets. The size of the packets will vary according to the transmission media in the network and the type of service.

14.3. TCP and IP

To understand how multiple−packet transmissions are handled, you need to know that the Internet actually uses two protocols, stacked one on top of the other.

The lower level, *IP* (Internet Protocol), knows how to get individual packets from a source address to a destination address (this is why these are called IP addresses). However, IP is not reliable; if a packet gets lost or dropped, the source and destination machines may never know it. In network jargon, IP is a *connectionless* protocol; the sender just fires a packet at the receiver and doesn't expect an acknowledgement.

IP is fast and cheap, though. Sometimes fast, cheap and unreliable is OK. When you play networked Doom or Quake, each bullet is represented by an IP packet. If a few of those get lost, that's OK.

The upper level, *TCP* (Transmission Control Protocol), gives you reliability. When two machines negotiate a TCP connection (which they do using IP), the receiver knows to send acknowledgements of the packets it sees back to the sender. If the sender doesn't see an acknowledgement for a packet within some timeout period, it resends that packet. Furthermore, the sender gives each TCP packet a sequence number, which the receiver can use you reassemble packets in case they show up out of order. (This can happen if network links go up or down during a connection.)

TCP/IP packets also contain a checksum to enable detection of data corrupted by bad links. So, from the point of view of anyone using TCP/IP and nameservers, it looks like a reliable way to pass streams of bytes between hostname/service−number pairs. People who write network protocols almost never have to think about all the packetizing, packet reassembly, error checking, checksumming, and retransmission that goes on below that level.

14.4. HTTP, an application protocol

Now let's get back to our example. Web browsers and servers speak an *application protocol* that runs on top of TCP/IP, using it simply as a way to pass strings of bytes back and forth. This protocol is called *HTTP* (Hyper−Text Transfer Protocol) and we've already seen one command in it −− the GET shown above.

When the GET command goes to metalab.unc.edu's webserver with service number 80, it will be dispatched to a *server daemon* listening on port 80. Most Internet services are implemented by server daemons that do nothing but wait on ports, watching for and executing incoming commands.

If the design of the Internet has one overall rule, it's that all the parts should be as simple and human−accessible as possible. HTTP, and its relatives (like the Simple Mail Transfer Protocol, *SMTP*, that is used to move electronic mail between hosts) tend to use simple printable−text commands that end with a carriage−return/line feed.

This is marginally inefficient; in some circumstances you could get more speed by using a tightly−coded binary protocol. But experience has shown that the benefits of having commands be easy for human beings to describe and understand outweigh any marginal gain in efficiency that you might get at the cost of making things tricky and opaque.

Therefore, what the server daemon ships back to you via TCP/IP is also text. The beginning of the response will look something like this (a few headers have been suppressed):

```
HTTP/1.1 200 OK
Date: Sat, 10 Oct 1998 18:43:35 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Thu, 27 Aug 1998 17:55:15 GMT
Content-Length: 2982
Content-Type: text/html
```

These headers will be followed by a blank line and the text of the web page (after which the connection is dropped). Your browser just displays that page. The headers tell it how (in particular, the Content−Type header tells it the returned data is really HTML).