



2. Unix

Unix was not developed with security, in any realistic sense, in mind.

D. Ritchie, *On the security of Unix*, 1975

In this book we will mostly use Linux. Linux is based on Unix, which is a completely different thing. It is important to set the record straight.

Unix was developed at AT&T Bell Labs in the late '60s by Ken Thompson, soon joined by Dennis Ritchie. It was a personal project that Thompson started on a PDP-7, when Bell Labs pulled out of the MULTICS project. It was then ported to a PDP-11 and enjoyed great success, first within the Bell Labs themselves and then at many universities and companies. The versions of Unix produced by Bell Labs are now called “Research Unix” and are numbered according to the version of their manual, starting with V1. The last well know Research Unix version is V7 from 1979; after that, and until the late '1990s, most people used a version of Unix that derived from either System V, a commercial version of Unix developed by AT&T, or BSD (Berkeley Software Distribution), a free version of Unix maintained by people at the University of California Berkeley (UC Berkeley). Both were based on research Unix code, extended in incompatible ways. The POSIX standard is (one of several) standards that try to define what every Unix system should look like.

A lawsuit was started in 1992 by AT&T against UC Berkeley, claiming that the latter had no right to use Unix code. Meanwhile, two important things happened: Richard Stallman had started the GNU (GNU's Not Unix) project in 1983, with the goal of rewriting Unix from the ground up to create a Unix-like system free of all legal problems. The GNU project produced many of the userspace commands, starting with the C compiler, but it never managed to materialize the kernel. This very important missing piece was finally created when Linus Torvalds announced his personal kernel project, which eventually became Linux, in 1991. By putting together Linux, the GNU software and a lot of other open source software (such as the Vim editor and the X Window System), we can now have several complete and free Unix-like systems.

The BSD people finally succeeded in rewriting their software and freeing it from any dependence on Unix. The BSD system survives today in FreeBSD, OpenBSD, NetBSD and some other variations.

Bell Labs did not stop at V7 Unix, producing other less well-know versions up to V10, and developing “Plan 9 from Bell Labs” (a reference to the trash sci-fi movie *Plan 9 from Outer Space*)

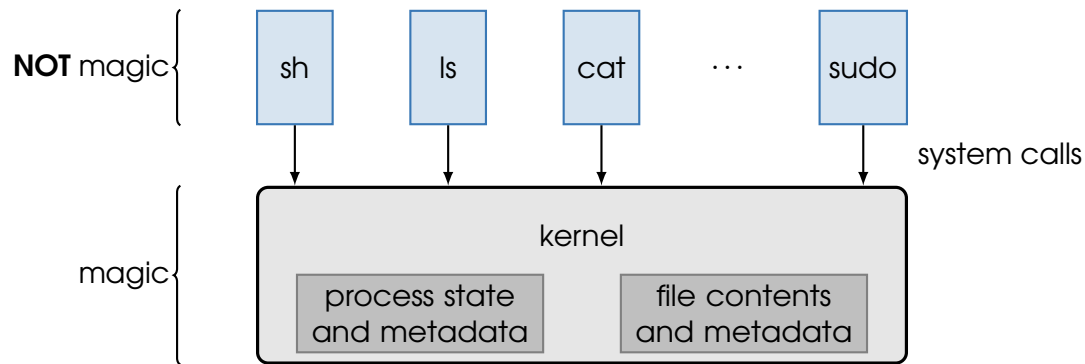


Figure 2.1 – The structure of Unix

and “Inferno”, where the Unix ideas were developed without the need to maintain compatibility with ancient decisions. Today, Linux also includes ideas from Plan 9, such as the `clone()` primitive, which replaces the old `fork()`.

2.1 The structure of Unix

You are probably already familiar with Unix, so we will move very quickly. For our purposes, one of the most important features to keep in mind is that no program in Unix is magic, except the kernel (see Figure 2.1). Only the kernel can do things that the other programs cannot. In particular, the shell is not magic: everything the shell does, you can do in your own programs, and you can even replace the default shell with another one. Not even `sudo` is magic: it only allows you to become root temporarily thanks to the set-uid feature. Moreover, this feature is not specific to `sudo`: any executable can have the set-uid bit set at the discretion of its owner (or the administrator).

The kernel implements a number of primitives in the form of system calls. Programs such as the shell, `ls`, `cat`, `sudo`, and so on, perform their tasks using these primitives. When you reason about what is possible and what is not, you only have to think about the primitives, not the programs: to read a file, `cat` must `open()` it, just like any other program that reads a file.

The original Unix was admired for its elegance, since the set of primitives was very small (a modern Linux system, unfortunately, implements hundreds of primitives). The system implements processes, which are the entities that execute programs, and files, which are automatically expanded sequences of bytes organized in a hierarchy of pathnames (the file system). Processes are handled by the `fork()`, `execve()`, `exit()` and `wait()` primitives. Files are handled with the `open()`, `read()`, `write()` and `close()` primitives.

The kernel keeps track of a number of properties for each process. Among them are:

- the process identifier (a small number which is reused when a process terminates);
- a *real* and an *effective* user identifier, and a real and effective group identifier.

R Later Unix-derived systems and modern Linux also implement a set of additional group identifiers, so a process can belong to many groups.

- a *current working directory*
- a table of open files.

The kernel also remembers, stored in the file system *inodes*, a set of attributes of each file and directory. Among them are:

- the file type (file, directory, symbolic link, device node)
- the ids of the file’s owner and group;

- the permission bits (read, write, execute permissions for the file’s owner, for the file’s group, and for other users)
- the set-uid and set-gid flags.

These are generically called process or file “metadata” in Figure 2.1. Only the kernel can access this state directly.

2.1.1 Process primitives

The `fork()` primitive is the only way to create a new process. The child process is a copy of its parent and, in particular, all of the process attributes mentioned above are copied to the child process (except, of course, the process identifier). In particular, the new process will have the same uid and gid as its parent, and will share the same open files. The executed program is also the same: the `fork()` primitive will return different values to the parent and the child, so the program can take different branches and let the two processes perform different actions.

The `execve()` primitive is the only way to execute a new program. The first argument of the primitive is the path of a file, which must be executable by the calling process (according to the file’s permissions and the process’s *effective* uid and gid). The process remains the same: same id, same open files, same current working directory, same real uid and gid (the effective ones may change because of the set-uid and set-gid feature, see below). However, all its memory is replaced by the contents of the file. Execution resumes at the file’s `_start` symbol. The `execve()` primitive takes two more arguments, which are two arrays of C strings, with each array terminated by `NULL`. The code in `_start` (which comes from the standard C library) will use the first array to create the `argc` and `argv` parameters of `main`, while the second array is used as the set of the *environment variables* of the process, pointed to by the global `environ` variable.

If the set-uid bit of the file is set, `execve()` changes the effective uid of the process to the uid of the owner of the file. The same goes for the set-gid bit and the effective gid. The real uid and gid do not change.

Processes terminate when they call the `exit()` primitive. A process can wait for any of its children to exit by using the `wait()` primitive. A small integer passed to `exit()`, can be received by `wait()` and used as a means for the child process to communicate errors or special conditions to its parent. By convention, a value of 0 means that all was well.

The uid and gid of a process can also be changed using the `setuid()` and `setgid()` primitives which behave differently when called by the root user (effective uid 0) and normal users (effective uid different from 0). If the effective uid of the calling process is 0, `setuid()` will accept any value and set both the real and effective uids to it (and similarly for `setgid()`). Normal users can only call these primitives when they are essentially no-ops (setting the ids to the value they already have), or (by set-uid/set-gid programs) to set the effective ids back to the real ones.

2.1.2 File primitives

We will cover only the most important features, assuming that you are already familiar with a Unix-like file system.

Files must be opened before they can be used. The `open()` primitive takes a path as the first argument and a mode (read and/or write) as the second. It checks if the calling process has the proper permissions to open the file in that mode (again, using the effective uid/gid and the file permissions). If successful, it finds the first free slot in the process open-file table, stores there a pointer to an in-kernel data structure describing the open file, and returns the index of the slot. This index (file descriptor) can be passed to the `read()` and `write()` system calls to sequentially read and write bytes to and from

the open file. The process should `close()` the files when it is done with them, mostly because the open-file table has a finite size, but the kernel will automatically close any open files when the process terminates.

If so instructed, `open()` can also create the file if it does not exist. The original design used a different primitive for this, `creat()`, which is still available today.

Directories must be created and deleted using the `mkdir()` and `rmdir()` primitives. Existing directories are opened and closed with `open()` and `close()`, like regular files, but you cannot use `write()`, and you can (no longer) use `read()` on them, since their content is system specific. The standard primitive to read directories is `getdents()`, but directories are best accessed using library functions (`opendir()`, `readdir()`, `closedir()`).

Finally, a process can change its own current working directory using the `chdir()` primitive, passing the path of the new directory as an argument.

2.1.3 Interpreting paths

All of `execve()`, `open()` and `chdir()` always interpret their first argument—a filesystem path—in the same way:

- if it *begins* with `/`, it is an absolute path starting from the root of the file system;
- if it *does not begin* with `/`, it is a relative path starting in the current working directory of the process.

Note, in particular, that no `PATH` variable is taken into account by `execve()`: `PATH` is handled in userspace before calling `execve()` (see Section 3.2).

Any character can be part of a file or directory name, except `0` and `/`. This includes whitespace, even newlines, and control characters like backspace.

2.2 A didactic reimplementation of some Unix programs

Unlike the original Unix code, the source code of modern libraries and commands, even the simplest ones, is huge and hard to read. Nevertheless, source code is very useful for understanding what is really going on in the system, how primitives are used and which program is responsible for which feature.

The `myunix` package contains a very simple rewrite of some standard Unix programs. The only purpose of the exercise is to show how things fit together without distracting details. The only thing that is assumed is the kernel—everything else is rewritten.

You can download the package here:

<https://lettieri.iet.unipi.it/hacking/myunix-3.1.zip>

Once unzipped, you obtain a `myunix-3.1` directory with the following subdirectories:

src contains the sources of all the commands

lib contains the sources of all the library functions

util contains a few scripts to build and run the programs

For many programs (`sh`, `ls`, `login`, ...) there are several sources numbered sequentially. Each version improves the previous one, typically by adding a single new feature.

The system should run on any sufficiently recent version of standard Linux distributions, such as Ubuntu or Kali. You can of course run Linux in a VM, but even WSL2 should work. Moreover, the sources include assembly for both Intel and ARM CPUs, so you should be able to run the system also in a Linux VM running on a Mac M1/2/...

The easiest way to compile everything is to run:

```
$ util/install
```

as a normal user. This creates a `root` directory containing a minimal Unix file system. Individual programs can be run as, e.g.:

```
$ root/bin/sh0
```

You can also run a “standalone” system which uses only the programs in `src`, but you need to be in `sudoers` and have a proper entry in `/etc/subuid` and `/etc/subgid`. Moreover, `xterm` and `uidmap` must be installed in your Linux distribution. If these conditions are met, you can run

```
$ util/start
```

as a normal user. You will be asked for your sudo password. The script will start a few `xterms`, each one emulating a terminal connected to the `mynix` system. Then, it will run the `init` program to spawn `getty` on the terminals. You should see the “`login:`” prompt on each one of them.

R You can follow the process described in Section 2.2.1 by reading `src/init.c`, then `src/getty.c` and finally `src/login1.c`.

The initial `passwd` file defines a couple of normal users (`user1` and `user2`) and the root user. Passwords are not set, so you can login by just typing `enter` at the password prompt. Once you have logged in, you can use `passwd` to set the passwords.

When several versions of the same program exist (e.g., `sh0`, `sh1`, etc.), the default one is the highest numbered one. So, for example, `/bin/sh` is actually a link to `/bin/shA`. You can select a different default one when you start the system, e.g.:

```
$ util/start sh=bad2sh
```

will start the system with `/bin/sh` pointing to `/bin/bad2sh`.

Type `ctrl+C` on the window where `utils/start` is running to stop the system.

The number of terminals can be tweaked by editing `src/ttys` and then running `util/install` again. Edit `src/passwd` to add/remove users, followed by `util/install`. Note that the install script does not delete the previous file system.

Finally, the programs can also be compiled using the system libraries, instead of the custom library in `lib`. Just enter the `src` directory and type `make` followed by the name of the program that you want to build (or just `make` to build them all). This should work even on a Mac M1/2/... without any VM, but some programs that use Linux specific features (such as `ps`) will not be compiled.

2.2.1 How things fit together

From a Unix access control point of view, you must always remember that what matters is the effective uid and gid of the processes, as checked by the kernel when they invoke primitives. The programs are not important: exactly the same `ls` program is run by normal users and by root, but the set of directories that can be listed depends on *who* is running `ls`. More specifically, it depends on the effective uid and gid of the process that `execve()`’d the `/bin/ls` file.

So, how do processes get the effective uids and gids of users? We can follow the boot process of `mynix` to have a clear view. It mimics, in a simplified way, the boot process of traditional Unix systems. When Unix boots, the kernel creates a process with id 1, both real and effective uid and gid set to 0 and no open files. The process executes the `init` command, which must exist and be executable. The Unix system is running on a large computer somewhere, and there are several terminals connected to it.

For example, in the picture at the beginning of this chapter, we see a pair of Teletype ASR 33 terminals connected to the PDP11 running Unix. An uncharacteristically shaved Ken Thompson is sitting at the first one, with Dennis Ritchie standing next to him.

The available terminals are listed, one per line, in a `/etc/ttys` file created by the administrator (ttys stands for teletypes). In this file, `init` forks and executes the `getty` command, passing the name of the teletype as an argument (via the second parameter of `execve()`). It then waits endlessly for one of its children to exit. Whenever a child exits, `init` wakes up and spawns a new `getty` for the corresponding terminal. You can take a look at `src/init1.c` to see how this can be done using the available system calls, and nothing else. However, instead of the expected `execve()` we see this in the function that spawns `getty`:

```
21     execl("/bin/getty", "getty", gettys[i].tty, NULL);
```

We are not calling the `execve()` system call directly. Instead, we use `execl()`, a C library function that is a little easier to use. You can find the sources in `lib/execl.c`. The function does some extra processing and then calls the system call (we know it has to call it: there is no other way to run a program). The first argument of `execl()` is exactly the same as `execve()`, and in fact `execl()` will pass it *as is*. The `execl` function is *variadic*, i.e. it takes a variable number of arguments, which in this case are C strings. You can take a look at `lib/execl.c` in `myunix` to see how this works. The function collects these strings, starting with the second argument until it finds `NULL`. It creates an array of pointers to these strings. This array then becomes the second argument of `execve()`, and then the `argv` of the new program. We follow the convention that the first element of `argv` must be the program name. As for the last `execve()` argument, the array pointing to the environment variables, `execl()` will simply pass the one from the calling process (this last part is actually done by `execv()`, see `lib/execv.c` in `myunix`). At this point the environment is still empty.

The `getty` process thus starts with `uid` and `gid` set to zero and no file open. It does whatever is necessary to put the teletype device into a usable state, then it opens the corresponding device file *three times*, once in read-only mode and twice in write-only mode. Here are the relevant lines in `src/getty.c`:

```
21     fd0 = open(buf, O_RDONLY);
22     fd1 = open(buf, O_WRONLY);
23     fd2 = open(buf, O_WRONLY);
```

(where `buf` is the path of the device received as an argument.) Since there were no open files before, and the `open()` primitive always uses the first free slot in the file descriptor table, the device goes into file descriptors 0, 1 and 2. By convention, these are the standard input, standard output and standard error files for programs started from this terminal: any descendant process will inherit these file descriptors, unless one of its ancestors explicitly `close()`s them. The `getty` program will then print “login:” to file descriptor 1:

```
26     write(fd1, "login: ", 7);
```

and starts reading from file descriptor 0:

```
27     n = read(fd0, buf, 9);
```

R Traditional usernames were limited to 8 characters, but we read 9 bytes to account for the newline. The lines after 27 (not shown) check that the newline is actually there and replace it with the string terminator, to leave a proper C string in `buf`.

When a user comes to the terminal and enters her name, `getty` will `execve()` the `login` program, passing the entered username as an argument:

```
34  execl("/bin/login", "login", buf, NULL);
```

Now the process (still the same one spawned by `init`) starts to run the `login` program. It still has its uids and gids set to zero, and its fds 0, 1, and 2 pointing to the terminal where the user has typed in her name. Let's take a look at the sources in `src/login1.c`. The program prints "password:" to file descriptor 1, then reads from file descriptor 0. It needs to do some `ioctl()` first, so that the terminal doesn't echo the password to the terminal printer, where it would remain there for everyone to see. This is done by the `getpass()` library function (in `lib/getpwnam.c`):

```
19  pass = getpass("Password: ");
```

Once it gets the password, it opens and reads the `/etc/passwd` file (prepared by the administrator), looking for a line starting with the username. This is traditionally done in another library function, `getpwnam()` (see `lib/getpwnam.c`).

```
22  if ( (pw = getpwnam(argv[1])) == NULL )
```

The function returns a pointer to a "struct passwd" structure, with one C field for each colon-separated field in the `/etc/passwd` file. If `getpwnam()` finds the user, `login` checks the password, and if they match, it calls

```
34  setgid(pw->pw_gid);
```

and

```
39  setuid(pw->pw_uid);
```

Since `login` is still running as root, these calls set both the real and effective gids and uids of the process those of the logged-in user. After that, it moves to the home directory of the user:

```
43  chdir(pw->pw_dir);
```

and finally executes the shell:

```
45  execl(pw->pw_shell, "-sh", NULL);
```

R The dash in the first argument (`argv[0]`) is a conventional of telling the shell that it is being called by `login`. The shell will use this fact to read some initialization script that should be only be run at login time.

`login` also sets some environment variables, which are then inherited by the shell (see `src/login2.c` for some examples).

When the shell runs, it will still have file descriptors 0, 1 and 2 pointing to the terminal opened by `getty` and the uids and gids set by `login`. From now on, all the programs started by the shell will inherit this setting, and, if the uid is greater than zero (a normal user has logged in) there will be no way to arbitrarily change the uid and gid since `setuid()` and `setgid()` are no longer available. Only by `execve()`ing set-uid/set-gid programs can the uids and gids be changed, but these programs will only perform safe actions (or so the administrator hopes).

When the shell exits because the user logs out, `init` will wake up and respawn `getty` on the same terminal.

- R** The shell is still running in the child process initially forked from `init`, since only `execve()` has been called since then. This is why `login` is able to understand that it needs to respawn `getty`.

Exercise 2.1 Can we swap `setgid()` with `setuid()` above? And `chdir()` with `setuid()`? Can we put the `chdir()` after the `execl()`? ■

2.2.2 The simplest UNIX program

- nano? Real programmers use emacs.
- Hey, real programmers use vim.
- Well, real programmers use ed.
- NO, real programmers use cat.

Rundall Monroe,
<https://xkcd.com/378/>

The `cat` program is probably the simplest Unix program that still manages to be useful in a variety of situations. When run without arguments, it copies its `stdin` to its `stdout`. The `src/cat1.c` version just does this:

```

9  int main()
10 {
11     int n;
12     char buf[4096];
13
14     while ( (n = read(0, buf, 4096)) > 0)
15         write(1, buf, n);
16
17     return 0;
18 }
```

Notice how the program uses file descriptors 0 and 1 without `open()`ing them: they are still the ones that were opened by `getty`.

If we run the program, it waits for a line of input, which is also echoed to the terminal as we type, prints it when we hit enter, and then starts over, until we type an EOT.

If we use shell redirection (see Section 3.5) we can send the output of `cat` to a file:

```
$ cat >newfile
```

This is about the simplest “editor” available in Unix (and the only one available in `mynix`).

2.2.2.1 On terminals

Note that, while `cat` is waiting for your line, you can edit what you have typed before hitting enter: you can delete the last character, the last word (Ctrl+W) or the whole line (Ctrl+U). Where is this feature implemented? Since we have only called `read()`, it must be implemented by something that is active during its execution. But `read()` is a system call, so line editing must be implemented either in the kernel, or in the terminal (emulator) itself. We cannot be more precise than this without adding a bit of history: old terminals were very simple and did not provide any line-editing functionality, so it had to be implemented in software. For a long chain of compatibility requirements, this is still true

today. In fact, this kind of line editing is implemented in the TTY module in the kernel, attached to the driver that talks to the (emulated) teletype device. The TTY module has a fairly complex set of options, which can be inspected and set using the `stty(1)` utility (or, more precisely, using a set of system calls also used by `stty`). You can print the list of the current options with `stty -a`. Many options are devoted to the details of the serial communication with the tty device and are mostly meaningless today; some options specify the action to be taken when special ASCII values are entered from the keyboard: some values cause a signal to be sent to the terminal's foreground process group, and others are used for line editing. The `erase` action removes the last input character and should be mapped to the “^?” ASCII value, which corresponds to DEL.

R The “^c” syntax is used to display unprintable, or *control*, ASCII values, i.e., the first 31 values and the last one (7f, DEL). The idea is to flip bit 0x40 to get a printable character, then prepend that character with “^”, which stands for $0x40 \oplus \dots$. Since the ASCII code of “?” is 3f, “^?” is $0x40 \oplus 3f = 7f$, i.e., DEL.

The ASCII code of DEL is 7f because ASCII was designed for paper tape: to erase any character, you could punch all seven holes.

Note that, for many keys, typing Ctrl+x will also toggle bit 0x40 of x so you can also enter DEL by typing Ctrl+? instead of the key mapped to DEL in your keyboard (probably backspace). This is also the way you can enter “^C” (ASCII ETX, usually mapped to the SIGINT signal) and the other values which are not otherwise available on the keyboard. For this reason the “^x” syntax is often pronounced “control x”.

This is reminiscent of how the Ctrl key was implemented in the Teletype ASR 33. However, in the ASR 33 the Ctrl key always *resets* bit 7 instead of toggling it. The Shift key instead toggled bit 5 (0x10), so if you shifted ‘1’ (ASCII 31) you got ‘!’ (ASCII 21) and so on. Many of the keytop pairs that we see on our modern keyboards come from old mechanical typewriters. The ASCII committee deliberately chose codes that differed by only one bit (bit 5) for these symbol pairs, precisely to allow for the implementation of the shift key as found on the ASR 33 device. Due to other constraints, however, the committee was only partially successful, and some typewriter pairs could not be preserved. The distinction between *bit-paired* and *typewriter-paired* keyboards has persisted to this day.

When terminals evolved more capabilities, like a bidimensional screen, they introduced *escape sequences* to embed control commands in the stream of ASCII characters exchanged with the computer. These sequences have been then standardized by ANSI^a. ANSI escape sequences start with ESC (ASCII 1B) and some variable sequence of printable characters, typically starting with “[”. For example, if the computer sends “ESC[A” to the terminal, it will move the cursor up one line.

New keys, like the arrow keys, also send escape sequences to the computer. For example, the up-arrow key sends “ESC[A”, i.e., the same sequence that moves the cursor up when received from the computer. Now, if we start `1-basic` and hit the up-arrow, we see “^[[A”. This is the echo (sent by the TTY module in the kernel) of the escape sequence sent by the terminal (emulator). By default, the TTY module echoes control characters in the same way explained above: “^[” stands for $0x40 \oplus 5B = 1B$, which is ESC. Note that the program reading from the terminal will receive the true sequence, not the one which is echoed. You can check this if you run `cat`, hit the up-arrow a few times and then press enter. Then `cat` will wake up and send the received characters back at the terminal and the cursor will move up.

^ahttps://en.wikipedia.org/wiki/ANSI_escape_code

2.2.3 Directories and inodes

The `src/ls[1-7].c` files contain increasingly complex implementations of the `ls` utility, designed to show various properties of directories and inodes. The simplest implementation, `src/ls1.c`, only prints the names of the entries in the current directory. Its code is as follows:

```

12 int main()
13 {
14     DIR *dir;
15     struct dirent* ent;
16
17     if ( !(dir = opendir(".")) ) {
18         perror(".");
19         return 1;
20     }
21
22     while ( (ent = readdir(dir)) ) {
23         printf("%s\n", ent->d_name);
24     }
25
26     return 0;
27 }
```

The implementation uses a pair of functions defined in `lib/opendir.c` and `lib/readdir.c`, rather than calling the system calls directly. Directories are opened with `open()`, just like normal files—`opendir()` just adds the allocation of the `DIR` object, which contains a buffer used by `readdir()`. However, the file descriptor returned by `open()` cannot be used with `read()` (and certainly not with `write()`): if we have read permission on a directory, we can call the `getdents()` system call. This system call can return one or more variable-length “`struct direct`” data structures. The `readdir()` library function used in line 23 is easier to use, since it internally buffers the entries obtained by `getdents()` and returns them one by one. The `dirent` structure contains at least the name (the one printed on line 23) and the inode number. On some systems, including Linux, it also contains a `d_type` field that encodes the type of entry (regular file, directory, device node, ...). If we run `ls1`, we may notice a few things:

- the entries are not returned in alphabetical order: this must be done by `ls` itself;
- *all* entries are returned, including those that start with “.”.

The “.” entries are hidden only in the sense that `ls` doesn’t show them by default: they are just regular entries for the kernel. The file `src/ls2.c` adds this behavior to our `ls`: we just need to skip entries whose `d_name` starts with a dot, unless the user has explicitly asked for them by passing the `-a` option.

File `src/ls3.c` adds the `-i` option, which prints the other information that is always available in `dirent`, i.e. the inode number. All other information about the entry, such as owner, group, mode, size and so on, is stored in its inode. File `src/ls4.c` adds a simplified version of the `-l` (long listing) option. To get the additional information we need to call `stat()` on each `d_name` found in the directory. Note that we need *search* (`x`) permission on the directory to pass the name of an entry to `stat()`.

The `ls4` binary prints the inode information *as-is*. The other task `ls` has to do is to make this information more readable. One thing that we may notice is that the inode only contains the numerical `uid` and `gid` of the file owner. This is generally true: the kernel *only* knows about these numerical

ids—all symbolic names are handled in userspace and are just conventions. The file `src/ls6.c` prints out the inode information using the conventions we are used to. Note in particular the following function:

```

169 char* pretty_print_owner(uid_t u)
170 {
171     static char owner[11];
172     struct passwd *pw;
173
174     if ( (pw = getpwuid(u)) != NULL )
175         return pw->pw_name;
176
177     snprintf(owner, 11, "%d", u);
178
179     return owner;
180 }

```

The function converts a numeric uid to the corresponding username, if found, otherwise it returns a string representing the uid itself. It uses the `getpwuid()` library function (`lib/getpwuid.c`) to look up the uid, similar to the `getpwnam()` function used by `login`. In both cases, the functions consult the `/etc/passwd`, which acts as a database mapping uids to usernames, and therefore needs to be readable by everyone.

R You can try removing read permissions from `/etc/passwd`: both `ls` and `ps` will lose their ability to display usernames and fall back to numerical ids. Group names are stored in `/etc/group`. If this file is readable, the tools will still be able to display symbolic group names.

2.3 Is Unix secure?

“On the Security of UNIX”¹ is a short paper by D. M. Ritchie, first published in the 6th edition of “The UNIX Programmer’s Manual” (1975) and then adapted for the 7th edition (1979). The main message of the paper is that

UNIX was not developed with security [. . .] in mind.

The paper lists several examples to support the above statement. Most of what Ritchie says is still true today, in some form.

2.3.1 Abuse of system resources

The first two examples deal with denial of service problems caused by overuse of system resources:

Here is a particularly ghastly sequence guaranteed to stop the system:

: loop	while : ; do
mkdir x	mkdir x
chdir x	chdir x
goto loop	done

(a) V6 script

(b) V7 script

¹<http://www.tom-yam.or.jp/2238/ref/secur.pdf>

This four-line script attempts to create an arbitrarily deep directory structure, and will only stop when either the i-node table is full or all disk blocks have been used. The `mkdir-chdir` sequence allows the paths passed to the commands to be kept short: trying to create `x/x/x/...` in one go would hit path size limit before doing any damage, while trying to create many files in the same directory would be stopped prematurely by the directory size limit.

The V6 script uses Thompson's shell syntax: the colon introduces a label and `goto` jumps to it. The implementation is interesting. Thompson's shell could only execute scripts by redirecting its standard input. The "`goto label`" statement was actually an external command that used `seek()` on its own standard input file pointer, looking for a line starting with a colon and the string `label`. Since the file pointer is shared between the shell and its children, after `goto` the shell would go back and read the command that followed the "`: label`" statement. The colon itself was another external command that did nothing and returned 0 (success). The `goto` command has disappeared, but the colon has remained as a built-in command in modern shells, where it is used either as a fast `true` (the V7 script shows an example of such usage), or when you are only interested in expanding its arguments for their side effects.

The second example exhausts system resources (either swap space or process table slots) by creating an excessive amount of background processes (the syntax used is the same as today):

For example, the sequence

```
command&
command&
command&
```

if continued long enough will use up all the slots in the system's process table [...]

V6 UNIX would either panic (i.e., crash) or become unusable in such circumstances. V7 UNIX mitigated the problems caused by the second example by introducing limits on the number of processes that a user can create; modern Unix and Unix-like systems do the same. Ritchie, however, duly notes that this is not sufficient and the system can still become essentially unusable, or even crash, if these processes use too many resources.

2.3.2 File permissions problems

Ritchie then goes on to explain file permissions.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically[.]

Ritchie explains the now-familiar rules for checking user/group/other permission, and the special cases for directories. He also introduces the set-UID/set-GID feature, using a motivating example that is much nicer than the ones commonly used today, typically based on `passwd`:

The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file [but] ordinary programs executed by others cannot access the score.

There are nonetheless a few problems he sees in this area:

1. The existence of the super user (aka root):

[...] there is a “super user” who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

2. Default permissions are too liberal:

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Unfortunately, UNIX software is exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone.


3. The permissions on directories are not intuitive:

[...] a writable file in a read-only directory may be changed, or even truncated, though not removed [...] it is possible to delete a file if one has write permission for its directory independently of any permissions for the file.

All of them are still true today, but they are not at the same level of importance. Problem 2 can be fixed in some cases by following Ritchie’s advice:

[...] if one wants to keep one’s files completely secret, it is possible to remove all permissions from the directory in which they live, which is easy and effective[.]

In the V7 version of the paper, Ritchie mentions the (then) recently introduced `umask` as a better solution for problem 2. The default value of `umask` set by `login` removes write permissions for others, and this also makes problem 3 much less visible.

 Many Linux distributions also remove group write permissions from the default `umask`. However, many still allow *read* permissions for group *and others*; this may be a problem in a multi-user system, if users don’t change the defaults.

Problem 1 is the most dangerous, and it has gotten worse: the number of “*privileged system calls*” that the root user may invoke has increased over the years and is now overwhelming. This is a problem when we are forced to run a program as root because the program needs one of these capabilities. For example, to give a web server the right to open port 80, we must also give it the right to wipe out the entire filesystem.

2.3.3 Passwords

Ritchie then moves on to the topic of passwords. He says that here UNIX behaves better than most other systems of the time, since

[p]asswords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood.

This allowed the `passwd` file to be world-readable, so that it could also be used as a database mapping usernames to uids (replacing an older `uids` file). This database is used, e.g., by `ls`, `ps`, `chown` and so on.

The encrypted-passwords idea was suggested by M. V. Wilkes in '68, but came to Unix via Multics, where it was developed in the aftermath of a funny incident in CTSS (the precursor of Multics). CTSS stored passwords in plain text, in a file readable only by administrators, like early versions of Unix. One day, two administrators at MIT were editing the message-of-the-day file and the password file, without knowing about each other. Since the editor program always used the same name for temporary files, the two files were accidentally mixed up. Until the system was stopped (by exploiting another bug to force it to crash) all users logging into the system were greeted with everyone’s passwords in clear text.

However, Ritchie points out that a world-readable `passwd` file opens the door to brute-force/dictionary attacks:

[...] 67 encrypted passwords were collected from 10 UNIX installations. These were tested against all five-letter combinations, all combinations of letters and digits of length four or less, and all words in Webster's Second unabridged dictionary; 60 of the 67 passwords were found.

So, users must choose strong passwords. This is even more true today, of course. However, to help prevent brute-force and dictionary attacks the encrypted passwords have now been removed from `passwd`, which must remain world-readable for compatibility, and are now stored in the `shadow` file, which is readable only by root.

Ritchie also mentions an old trick that works on UNIX too:

[...] write a program which types out "login: " on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

This is still true today, and it's not much of a problem only because we no longer log into our systems from public-access terminals. Windows NT is safer in this regard: the Ctrl+Alt+Del key combination cannot be intercepted by user programs and provides a "trusted path" from the user to Windows itself.

2.3.4 Set-UID and Set-GID programs

Then Ritchie's comes back to the very important topic of set-UID/set-GID programs (his own invention, and the only UNIX patent). First, he notes that set-UID/set-GID programs must not be writable by attackers (a problem related to how permissions work):

For example, if the super-user (`su`) command is writable, anyone can copy the shell onto it and get a password-free version of `su`.

This is a general note: the permissions of a file (including the set-UID and set-GID flags) are stored in its i-node: they don't change when you write into the file.

More importantly, he notes that set-UID/set-GID programs must be

[...] careful of what is fed into them.

This is still extremely important, and we will spend some time on the attacks that target this very feature of UNIX(-like) systems. Ritchie introduces an example that is no longer directly applicable, but is still instructive (Ritchie himself repeats it in the V7 version of the paper, even if it was already "*obsolete*" by then):

In some systems, for example, the mail command is set-UID and owned by the super-user [`setuid-root` in modern jargon, ndr]. The notion is that one should be able to send mail to anyone even if they want to protect their directories from writing.

This may be abused to violate privacy, since

[...] anyone can mail someone else's private file to himself[,]

and also integrity:

[...] make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say `/tmp`). Finally mail the bogus login line to `/tmp/.mail`; You can then login as the super-user, clean up the incriminating evidence, and have your will.

To understand the examples, we need a few explanations. The V5 `mail` command takes a *letter* and a list of *person* strings as arguments. To send “*someone else’s private file*” to himself, Ritchie could just type:

```
% mail path-to-the-file dmr
```

(“%” is the user prompt in the V5 and V6 shells.) For the second example, we need to read the V5 `mail(1)` man-page carefully:

A person is either a user name recognized by login, in which case the mail is sent to the default working directory of that user, or the path name of a directory, in which case mailbox in that directory is used.

So, Ritchie is suggesting to operate as follows (using `mailbox` instead of `.mail`, as the manual says):

```
% ln /etc/passwd /tmp/mailbox
% mail bogus /tmp
```

In this way the contents of `bogus` will be prepended to `/etc/passwd`. One can write the following into `bogus`, before “mailing” it:

```
dmr:encrypted-password:0:1::/usr/dmr:/bin/sh
```

It is the zero UID that grants super user powers, not the username.

- R** The `mail` command also prepended the sender’s name and a “postmark”, so the `passwd` file will contain an incorrectly formatted line after this attack—apparently `login` would not be confused by it and would simply skip the offending line.

As Ritchie implies in the V7 version of the paper, the V7 `mail` command is not vulnerable to either attack. He doesn’t say why this is the case, but we can note the following:

1. V7 `mail` doesn’t accept a *letter* argument, it can only read the letter from standard input;
2. in V7 `mail` all *person* arguments must refer to existing users, not directories.

Point 1 solves the privacy problem, since now you can only send files that you can already read (see also Exercise 3.10). However, V7 UNIX actually *exacerbated* the problem with `set-UID/set-GID` programs with the introduction of environment variables, as we will see. The V7 version of Ritchie’s paper doesn’t show any awareness of this fact yet.

Ritchie’s description of the attacks doesn’t exactly apply to any officially released version of `mail`, at least as described in the respective manuals. Only V6 uses `.mail` for the mailbox, but V5 was the last to have a *letter* argument.

The ability to read the letter from standard input was added to `mail` in V3, presumably to make it usable in pipelines. Until V5, this behavior was optional, selected if you passed only one argument, interpreted as *person*. This syntax, induced by the need to avoid ambiguity with the other use, was a bit limiting, so the decision to drop the *letter* argument in V6 may also have been motivated by usability considerations.

2.3.5 Untrusted filesystems

Finally, Ritchie mentions a problem with filesystem mounts:

Once a disk pack is mounted, the system believes what is on it. [...] a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of `su`; other files can be unprotected entries for special files.

The only thing that might need an explaining is the reference to “*unprotected entries for special files*”. Ritchie is referring to the special device nodes usually found in `/dev`, but which can actually be created anywhere. In Linux, a device like `/dev/sda` gives low-level access to the blocks of the first hard disk. Access to this device (and others like it) is usually restricted to root: a user who can read from this device can essentially see the entire contents of the file system (including blocks belonging to deleted files that have not yet been recycled), bypassing all other permission checks; a user who can write to `/dev/sda` can make arbitrary changes to the filesystem. Now, since the power of `/dev/sda` comes only from the major and minor numbers written in its i-node, a specially crafted disk could very well contain an i-node with the major/minor pair of `/dev/sda`, or of any other hard disk, with a permissive mode. A user who can mount this disk will then get unrestricted access to any other disk.

The V1 manual included the phrase “*This call should be restricted to the super-user*” in the BUGS sections of the `mount` and `umount` entries—another step towards the overpowered root user we see today. These calls are now privileged, but PC users still need a way to mount filesystems, if they want to use their pen-drives or virtual disks and so on, so the system may contain `suid-root` programs or root daemons that will call `mount()` on their behalf. Modern Linux kernels, however, allow you to specify “mount options” such as `nosuid` and `nodev` which are explicitly aimed at the types of threats hinted at by Ritchie: `nosuid` will not allow `set-UID/set-GID` binaries in the mounted filesystem, and `nodev` will not allow special device nodes.