

Command Level

2	Unix	15
2.1	The structure of Unix	16
2.2	How things fit together	21
2.3	Reading and writing files	24
2.4	Terminal I/O	25
2.5	Directories and inodes	27
2.6	Hard links	29
2.7	Set-uid and set-gid programs	31
3	How a Unix shell works	35
3.1	The simplest shell (tag: 1-basic)	36
3.2	Using <code>PATH</code> (tag: 2-path)	38
3.3	Splitting the command line (tag: 3-words)	39
3.4	Shell builtins (tag: 4-builtin)	42
3.5	I/O redirection (tag: 5-redir)	43
3.6	Scripting (tag: 6-intr)	44
3.7	Environment variables (tag: 7-env)	46
3.8	Quoting (tag: 8-quote)	50
4	Is Unix secure?	55
4.1	Abuse of system resources	55
4.2	File permissions problems	56
4.3	Passwords	58
4.4	Set-UID and Set-GID programs	58
4.5	Untrusted filesystems	60
5	Exploiting the environment	63
5.1	Exploiting <code>PATH</code>	63
5.2	Mitigation: Default value of <code>PATH</code>	65
5.3	Exploiting setuid programs that escape to a shell	66
5.4	Exploiting shell metacharacters	71
5.5	Prevention: don't use <code>system()</code>	73
5.6	Prevention: no set-uid scripts	73
5.7	Mitigation: Privilege drop	74
6	Exploiting links	77
6.1	Opening unexpected files	77
6.2	Exploiting races with <code>access()</code>	78
6.3	Exploiting set-uid shebang scripts	79
6.4	Symbolic links	81
6.5	Exploiting symlinks	83
6.6	Prevention: secure temporary file creation	84

In Part I we assume that the programs we use are memory safe and cannot crash or expose their internal implementation details. The attacks at this level abuse either misconfigurations or programming mistakes that don't depend on the language used to implement the commands: they would work the same, even if the entire system were rewritten in Rust. Nonetheless, we will use C throughout, since this is still the language of choice for this kind of programs. In the first two chapters we spend some time in understanding the actual logic of these commands, which can be surprising. We then study the most common ways by which users can exploit this logic to escalate their privilege.



2. Unix

The system had no name at that point, and (if memory serves) I suggested, based on Latin roots, that since Multics provided “many of everything” and the new system had at most one of everything, it should be called “UNICS” [...].

B. W. Kernighan, *UNIX: A History and a Memoir*, 2019

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 [25] 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 DEC PDP-7, when Bell Labs pulled out of the Multics project [3].

Nothing happens in a vacuum, and the roots of many Unix features can be traced clearly. It is well known that Multics (MULTIplexed Information and Computing Service) [11, 6] is the inspiration for many Unix ideas, but its predecessor CTSS (Compatible Time Sharing System) [10] is also important. CTSS was used to develop Multics, so all the Bell Labs people working on Multics were also CTSS users. Indeed, Ritchie wrote that Unix can be considered “a modern implementation of CTSS” [20]. The developers were also aware of other ongoing time-sharing research, such as Project GENIE at Berkeley [19], where Thompson did his Masters.

Many minutiae, such as syntactic choices and some technical terms, come from pre-existing non-Time-sharing systems, notably from IBM (CTSS ran on a modified IBM 7094) and, of course, from DEC. For example, the origins of what is now known as AT&T assembly syntax goes back to the assemblers of early versions of Unix, which were explicitly based on DEC assemblers.

Many other Bell Labs researchers started contributing to Unix. The system was 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 in 1971. The last well known 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 [7], a commercial version of Unix developed by AT&T, or BSD (Berkeley Software Distribution) [38], 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. Other incompatible versions of Unix came from companies such as IBM, HP and so on. These companies licensed AT&T code and then modified it to port the system to their own hardware, adding features in the process. In 1988 IEEE released the first version of the POSIX [37] family of standards, one of several attempts to define what every Unix system should look like.

A lawsuit was started in 1992 by USL (Unix System Laboratories), a subsidiary of AT&T, against the Regents of the University of California (Berkeley), claiming that the latter had breached their software licence contract by freely distributing the BSD code. Meanwhile, two important things happened: Richard Stallman had started the GNU (GNU's Not Unix) project [48] 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 [49], which eventually became Linux [50], 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 in the form of Linux distributions. Most of them are largely compliant with POSIX, even if they are not certified by IEEE.

The USL vs UC Berkeley suit was settled out of courts in 1994 (the University had countersued USL, since the latter had used the former's TCP/IP code without attribution). The BSD system survives today in FreeBSD [34, 5], NetBSD [43], OpenBSD [44] and some other variations.

Bell Labs did not stop at v7 Unix, producing other less well-known versions up to v10, and developing “Plan 9 from Bell Labs” [18] (a reference to the trash sci-fi movie *Plan 9 from Outer Space*) and “Inferno” [12], 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()`.

In the rest of Part I we will use Linux as if it were Unix, limiting ourselves to the core Unix features that are still clearly identifiable. Some of these features have evolved over the years, and we will note the differences where necessary. Some modern Linux features will be examined in Part III.

2.1 The structure of Unix

There is nothing special about system-provided commands except that they are kept in a directory where the Shell can find them.

K. Thompson, D. M. Ritchie, *Unix Programmer's Manual* (v3), 1973

You are probably already familiar with Unix, so we will move very quickly on the basics and then highlight some finer points that are often forgotten. 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 (we will write our own shell in Chapter 3). Not even `sudo` is magic: it only allows you to become root temporarily thanks to the `set-uid` feature, which is not specific to `sudo`: any executable can have the

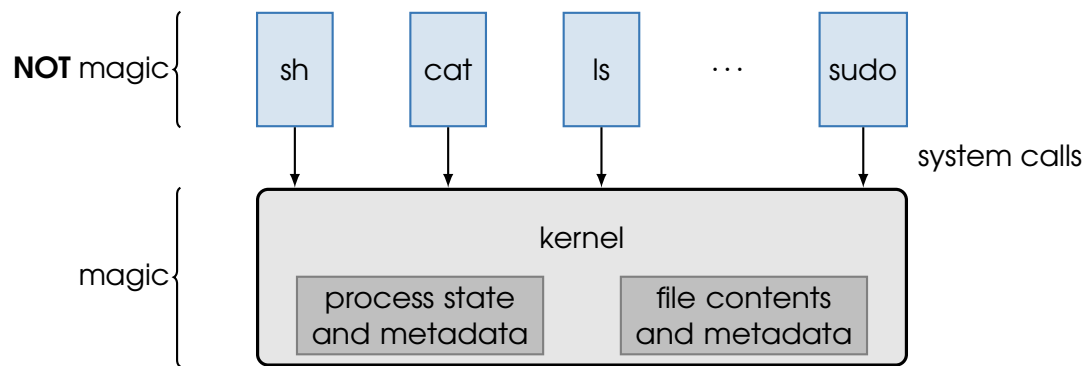


Figure 2.1 – The structure of Unix

set-uid bit set at the discretion of its owner or the administrator (see Section 2.7 below).

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

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

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. It is declared as follows:

```
pid_t fork(void);
```

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 (in particular, the child receives zero and the parent receives the pid of the child).

The `execve()` primitive is the only way to execute a new program and is declared as follows:

```
int execve(const char *path, char *argv[], char *envp[]);
```

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 pid, 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 (see also Section 3.7).

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, declared as follows:

```
void exit(int status);
```

R This is actually a userspace library function that first does some cleanup and then calls the `_exit()` kernel primitive. See also Exercise 3.11 on the difference between the two.

A process can wait for any of its children to exit by using the `wait()` primitive, declared as follows:

```
pid_t wait(int *status);
```

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); set-uid/set-gid programs can use them to set the effective ids back to the real ones, or to upgrade the real ids to the effective 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 is declared as follows:


```
int open(const char *path, int flags, ...);
```

The primitive takes a path as the first argument and some mode flags (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 `read()` and `write()` primitives are declared as follows:

```
ssize_t read (int fd,          void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);
```

Both return the number of bytes actually read or written, or `-1` in case of error. The number returned can be less than `count` for a number of reasons that depend on the type of the file being accessed. In particular, `read()` can return zero to signal an End Of File (EOF) condition. 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; in this case it takes a third argument “`mode_t mode`”, which sets the file permissions. The original design used a different primitive for file creation, `creat()`, which is still available today.

In modern Unix derivatives, 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()`, see Section 2.5).

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 the null byte and the slash (`/`). This includes whitespace, even newlines, and control characters like backspace.

2.1.4 A didactic reimplementation of some Unix programs

Use the source, Luke

anonymous

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 `mynix` 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/mynix-3.1.zip>

Once unzipped, you obtain a `mynix-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. The initial `/etc/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/sh9`. 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 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. Imagine that 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 PDP 11 running Unix. An uncharacteristically shaved Ken Thompson is sitting at the first one, with Dennis Ritchie standing next to him.

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 available terminals are listed, one per line, in a `/etc/ttys` file created by the administrator (ttys stands for teletypes). For each line 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.

■ **Example 2.1** Start `mynix` and login as root in `tty1`, then run “`ps jax`” to see the list of all processes. The important columns in the output are PPID (pid of the parent of the process), PID (pid of the process), UID (effective user id of the process) and COMMAND (the `argv` of the process). You should see a COMMAND `init` with PID 1 and UID 0. There is also a COMMAND “`getty tty2`” with UID 0 and PPID 1 (a child of `init`). It is the `getty` that is listening on the other 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);
```

```
init1.c
```

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. 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 `mylinux`). 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 (nothing, in our case), 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.

■ **Example 2.2** Continuing from example 2.1, assuming that the PID of “`getty tty2`” is 3, run

```
# ls -l /proc/3/fd
```

The `/proc` directory contains a pseudo filesystem that allows us to inspect the properties of all the active processes (it is where `ps` takes its data from). Each process has its own subdirectory in `/proc`, named from its PID. The `fd` subdirectory contains a link for each open file descriptor, pointing to the corresponding file. The above command should show three files, named 0, 1 and 2, all pointing to the same character device. ■

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.

■ **Example 2.3** Continuing from example 2.2, go to the `tty2` terminal, type “`user1`” and Enter, but do not answer yet to the “`password:`” prompt. Instead, go back to `tty1` (where `root` is logged in) and type “`ps jax`” again. You should see that process 3 is now running COMMAND “`init user1`”, but it is otherwise the same process that was running “`getty tty2`” before. In particular, it is still running with UID 0. ■

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: ");
```

login1.c

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 )
```

login1.c

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);
```

login1.c

and

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

login1.c

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

R The kernel doesn’t know which users belong to which groups, it only knows about the user ids and group ids of *processes*. The fact that the processes of a user also belong to the group of the user is entirely dependent on the fact that `login` calls `setgid()` with the proper value extracted from `/etc/passwd`.

After that, `login` moves to the home directory of the user:

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

login1.c

and finally executes the shell:

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

login1.c

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

■ **Example 2.4** Continuing from example 2.3, go back to `tty2` and enter `user1`’s password (empty if you have not changed it). Now `user1` is logged in. Go back to `tty1` and run “`ps jax`” again: process 3 is now running COMMAND “`-sh`”; most importantly, its UID is now 1000. ■

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 `init` is able to understand that it needs to respawn `getty` when it is notified of the termination of the shell.

■ **Example 2.5** Continuing from example 2.4, go back to `tty2` and logout `user1` (just enter Ctrl+D on an empty line). In `tty1`, type “`ps jax`” for the last time. Process 3 is no longer in the list: it terminated when the shell exited, and `init` has spawned a new process (with a new PID) to run “`getty tty2`”. The new process has printed “`login:`” on `tty2`. ■

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

2.3 Reading and writing files

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

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

The `cat` program, available since v1 and certainly older, is probably the simplest Unix program that still manages to be useful in a variety of situations (`echo` is another contender, but it came later, in v2). 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[512];
13
14     while ( (n = read(0, buf, 512)) > 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` and then inherited through `execve()` and `fork()`.

R For simplicity, the above program assumes that `write()` will actually write all `n` bytes, which is true for all normal cases. A more robust program should check the return value and retry until all bytes are written.

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.

Exercise 2.2 EOT (End Of Transmission, Ctrl+D) acts as an EOF (End Of File) only on *an empty line*. What happens if you type something and then type Ctrl+D before hitting Enter? ■

If we use I/O 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. It is also the only one available in `myunix`.

The `cat` command is supposed to concatenate files, but this simple version doesn’t even do that. A more complete implementation is in `src/cat2.c`. In `cat2`, `stdin` is used only if no arguments are passed. The final version, `src/cat3.c`, adds the special handling of the “-” option to refer to `stdin` together with other files.

According to McIlroy [see 16, page 6], the “-” syntax was added by Thompson in `v5 sort` and rapidly spread to other programs. However, since it is just a convention, it is not understood by all programs. The superior `/dev/fd` special files, also supported by Linux, were only added in `v8`.

2.4 Terminal I/O

When you type to UNIX, a gnome deep in the system is gathering your characters and saving them in a secret place.

K. Thompson, D. M. Ritchie, *Unix Programmer’s Manual* (v3), 1975

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. See Figure 2.2: the TTY module buffers all the bytes coming from the tty device and echoes them back as they come; the program reading from the device (in this case, `cat`) receives the bytes only when the buffer is flushed, for example, when the incoming byte is a newline (ASCII 0x0a), or EOT (ASCII 0x04, Ctrl+D). While the bytes are in the buffer, the TTY module allows us to edit them using the above control characters.

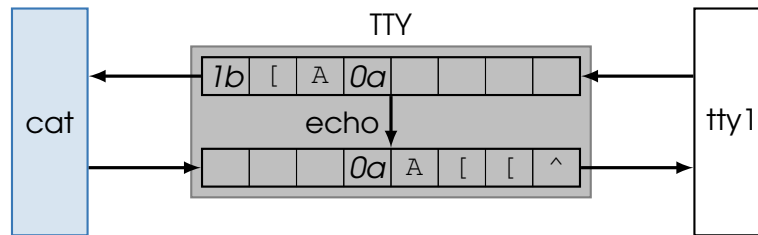


Figure 2.2 – The TTY module inside the kernel

The basic features of deleting the last character (“erase”) and the whole line (“kill”) were available since the first versions of Unix, using # for erase and @ for kill. The terminology and the choice of characters was the same as in CTSS.

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; there is an option to disable echo (the one used by `login` when it asks for your password)); 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 (0x7f, DEL). The idea is to flip bit 0x40 to get a printable character, then prepend that character with “^”, which stands for 0x40 ⊕ The ASCII code of “?” is 0x3f, and therefore “^?” stands for 0x40 ⊕ 0x3f = 0x7f, which is DEL.

The ASCII code of DEL is 0x7f 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 0x03, usually mapped to the `SIGINT` signal by the TTY module) 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 0x31) you got “!” (ASCII 0x21) 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 [4]. 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 ANSI¹. ANSI escape sequences start with ESC (ASCII 0x1B) 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.

■ **Example 2.6** Start `cat` and hit the up-arrow: you 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, so “`^[`” stands for $0x40 \oplus 0x5B = 0x1B$, which is ESC. Note that `cat` will receive the true “ESC[A” 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. ■

2.5 Directories and inodes

`cat .` doesn’t work anymore

R. Pike, *The Good, the Bad and the Ugly*, 2001

Files and directories are identified by their *inode number*, but are accessed using *links*, which pair a symbolic name with an inode number. Directories are special files that can only contain links.

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     closedir(dir);
27
28     return 0;
29 }
```

¹https://en.wikipedia.org/wiki/ANSI_escape_code

The implementation uses the functions defined in `opendir.c`, `readdir.c`, and `closedir.c` in the directory `lib` of `myunix`, 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 to get the directory entries. This system call can return one or more variable-length “`struct dirent`” 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 of the entry (the one printed on line 23) and its inode number.

If we run `ls1`, we may notice a few things:

- the entries are not returned in alphabetical order: any sorting 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 printed by the `-l` option of `ls`, such as owners, groups, modes, sizes and so on, is stored in the inodes. File `src/ls4.c` adds a simplified version of the `-l` option. To get the additional information we need to call `stat()` on each `d_name` found in the directory.



On some systems, including Linux, the `dirent` structure also contains a `d_type` field which encodes the type of entry (regular file, directory, device node, ...) and can therefore be obtained without calling `stat()`. This allows fast implementations of “`ls -F`” and similar features, such as using colours to distinguish different types of directory entries.

Exercise 2.3 The `src/ls5.c` version also accepts the name of a directory. Do the following:

```
$ mkdir onlyread onlysearch
$ touch onlyread/file onlysearch/file
$ chmod a=r onlyread
$ chmod a=x onlysearch
```

Try to predict what happens if you type the following commands:

1. `ls5 onlysearch`
2. `ls5 onlyread`
3. `ls5 -l onlysearch`
4. `ls5 -l onlyread`

The `ls4` and `ls5` binaries print 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 `src/ls6.c` version prints out the inode information using the conventions we are used to. Note in particular the following function:

```

171 char* pretty_print_owner(uid_t u)
172 {
173     static char owner[11];
174     struct passwd *pw;
175
176     snprintf(owner, 11, "%d", u);
177
178     if ( (pw = getpwuid(u)) != NULL )
179         return pw->pw_name;
180
181     return owner;
182 }

```

ls6.c

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` file, which acts as a database mapping uids to usernames, and therefore needs to be readable by everyone.



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.

2.6 Hard links

Links, which are now referred to as *hard links* to distinguish them from the *soft* links introduced later (see Section 6.4), are an integral part of the Unix file system design. Creating a file or a directory involves two distinct actions, performed indivisibly by either `creat()`, `open()`, or `mkdir()`:

- allocating and initializing an inode that describes the new file or directory;
- creating a new link that points to the inode (through its inode number).

Additional links to existing regular files can be created using the `link()` system call, declared as follows:

```
int link(const char *oldpath, const char *newpath);
```

This system call retrieves the inode number of `oldpath` and creates a new link to it from `newpath`. The `ln1.c` file in `mynix` implements a simplified version of the `ln` command that acts as a wrapper around this system call.

The process invoking this call must have search permission on all the directories in `oldpath` and `newpath`, and write permission on the final directory in `newpath`. However, it needs no permissions on the linked file itself. The idea is that the process does not gain any new permissions on the original file that it did not already have.

The new link is indistinguishable from the original. The kernel keeps track of all the hard links pointing to an inode and only frees the inode (and the disk space allocated for its data) if there are no hard links pointing to it.



The second column of the long listing output of `ls` shows the number of existing hard links to any file or directory. This number is stored in the file's inode.

There is no way to free an inode, other than removing all the hard links pointing to it. In particular,

the `rm` command uses the `unlink()` system call. For example, this is the code of `rm` in `myunix` (file `src/rm.c`):

```
6 int main(int argc, char* argv[])
7 {
8     int i, errors;
9
10    errors = 0;
11    for (i = 1; i < argc; i++)
12        if (unlink(argv[i]) < 0) {
13            perror(argv[i]);
14            errors++;
15        }
16    return errors;
17 }
```

If the file was open in any process, the behavior is peculiar: the file disappears from its directory and cannot be opened again, but the actual deallocation is delayed until the file is closed, so that the processes can continue to use it undisturbed.

Each directory also contains the special entries “.” and “..”, which are the only permitted hard links to directories. The “.” entry points to the directory itself, and the “..” entry points to the parent directory. The root directory has no parent directory, so its “..” link points to itself.

The original PDP-7 version of UNIX allowed unrestricted links to directories. The decision to disallow them came very early in Unix development, even before v1 [see 24, page 5]. However, the restriction was only meant for normal users: the super user retained this capability, to support directory creation, deletion and renaming. In fact, new directories were created using the generic `mknod()` system call, which didn't add the special “.” and “..” entries—the `mkdir` command was set-uid root and created them using privileged calls to `link()`. Directories were deleted with `unlink()`, but only the super user could do that; accordingly, `rmdir` was set-uid root too. It made sure that the directory to be deleted only contained the “.” and “..” entries, and unlinked them too. Even `mv` was set-uid root. It used its privilege when moving a directory: it created a hard link to the source directory in the destination path, adjusted the “..” entry if necessary, and then unlinked the source.

In modern Unix and Unix-like systems, the `mkdir()`, `rmdir()` and `rename()` system calls take care of all of the above link management inside the kernel, and the super user has lost the power to create and delete links to directories.

Note that the existence of links means that the Unix file system is a graph, not a tree. However, the limitations on the links to directories have two important consequences: the graph is *directed* (there can be no loops), and the directories form a tree.

2.7 Set-uid and set-gid programs

When this bit is set to one, the identification of the current user is changed to that of the owner of the executable file.

D. M. Ritchie, *Patent US4135240A*, 1979

The set-uid/set-gid idea can be seen as a generalisation of the system call mechanism. In both cases, there is a privilege switch coupled with a *protected transfer of control*. By this we mean a transfer of control to a destination defined by the callee: in the case of system calls, it is the kernel that defines the set of system call entry points, and the user cannot change them (e.g., she cannot jump in the middle of a syscall); in case of set-uid programs, the entry point is written inside the executable file, and unauthorised processes (i.e., processes that don't have write access to the file) cannot change it.

The mechanism is very subtle, and there are many misconceptions about it. The set-uid bit only changes the effective uid of the process that `execve()`s the program, and the set-gid bit only changes the effective gid. The two changes are independent: a uid change doesn't imply a gid change and *vice versa*: For example, if you run a set-uid program from your shell, the spawned process will run with the new uid, but it will still inherit your group(s). This is a necessary consequence of the fact that, as we noted in Section 2.2, the kernel doesn't know the relationship between users and groups: this relationship is written in regular files like `/etc/passwd` and `/etc/group`, and with the exception of executable files passed to `execve()`, the kernel never cares about the contents of regular files.

Most importantly, no other process is affected by the uid/gid change. The often-heard phrase that the effect lasts “only for the duration of the set-uid program” is a bit misleading, since “duration” has nothing to do with it: the shell always spawns a new process to run external programs, and when that program is set-uid (and/or set-gid), the child process undergoes the effective uid (and/or gid) change, but nothing happens to the process running the shell, or any other process.

■ **Example 2.7** Start `mynix`, login as `user1` and enter `passwd`. While `passwd` is still running, login as `user2` from the other terminal and run

```
$ ps jax
```

You should see the process that is running `user1`'s shell, in a sleep (S) state, and a child process that is running `passwd`. The latter has uid zero, but the former has uid 1000. ■

To see the difference, think of running a set-uid program in background: while the privileged program is still running, we can issue other commands to our shell, and these will run with our normal user id, not the upgraded one.

If the affected process `fork()`s other processes, those processes will inherit both the real and the effective uids and gids from their parent, so they will run in set-uid and/or set-gid mode too. If any of these processes `execve()` another program, the uids and gids will be preserved (unless the executed program is itself set-uid and/or set-gid, in which case the effective uid and/or gid of that process will be changed again). So, for example, a shell that is running in set-uid mode will execute all commands in set-uid mode too.

Any program can have these bits set: the owner of the program can set them with a simple call to `chmod`.

■ **Example 2.8** Start `mynix` and login as `user1`. Check that you cannot list the contents of the `/root` directory, nor can you `cd` into it:

```
$ ls -la /root
$ cd /root
```

Both commands should return a “Permission denied” error. Now login as root from the other terminal and run:

```
# chmod u+s /bin/ls
```

Go back to the first terminal (the one where `user1` is logged in) and try the `ls` and `cd` commands again: this time `ls` will succeed, but `cd` will still fail. ■

Only the superuser and the owner of the program can set these bits: this is not a new rule, just the normal rule limiting the use of the `chmod()` syscall.

- R The restriction applies also to the set-gid bit: non-owners have no right to set the set-git bit, even if they belong to the file’s group. On the other hand, a non-root owner cannot set the set-gid bit if she doesn’t also belong to the group of the file.

The effective uid and/or gid change gives the affected process the right to access files and directories using the new credentials. If the new effective uid is not zero, there are no other effects. It is only when the set-uid program is owned by root that the process gains the power to make privileged system calls (in particular, an executable that is only set-gid can never give root power, since only uid zero is special to the kernel and set-git programs can only change the gid, not the uid).

Something similar to `setuid` root programs can be found in CTSS. The system had a fixed table of “privileged” commands, stored in kernel memory. These programs were allowed to “make supervisor calls which users are forbidden to make” (one such command was `MAIL`, from which Unix `mail` derives: it needed the privilege to write to other users’ `MAIL BOX` files). The `setuid` root mechanism can be seen as a generalisation of this CTSS feature: instead of having a fixed table of privileged programs, any program can be made privileged by the superuser.

■ **Example 2.9** Let’s try something similar to example 2.8, but this time without involving root. If you still have the system in the state of example 2.8, first login as root and remove the set-uid flag from `/bin/ls` to restore its normal state:

```
# chmod u-s /bin/ls
```

Then logout from root and login as `user2`. Login as `user1` in the other terminal. In the terminal where `user1` is logged in, create a directory readable only by `user1`:

```
(user1)$ mkdir mydir
(user1)$ chmod go= mydir
```

Check that `user2` has no access to this directory:

```
(user2)$ ls -la /home/user1/mydir
```

This should return “Permission denied”. Now, let `user1` create a copy of `ls`:

```
(user1)$ cp /bin/ls /tmp/ls
```

The owner of this copy of `ls` is `user1`, so she can use `chmod` on it. In particular, she can set the `setuid`-flag:

```
(user1)$ chmod u+s /tmp/ls
```

Now go back to the terminal where `user2` is logged in and try:

```
(user2)$ /tmp/ls -la /home/user1/mydir
(user2)$ /tmp/ls -la /root
(user2)$ /bin/ls -la /home/user1/mydir
```

The first command succeeds and the other two fail: `/tmp/ls` grants you the ability to run `ls` as if you were `user1`, and nothing else. The effect depends on the `set-uid` flag on the `/tmp/ls` file: you gain nothing by running the original copy of `ls`. ■

2.7.1 The `sudo` command

The widespread use of `sudo` (Super User DO) has created many other misconceptions in this area. The `mynix` sources contain an extremely simplified version of `sudo` in the `src/sudo[1-3].c` files. The simplest version is as follows:

```
4 int main(int argc, char *argv[])
5 {
6     if (argc < 2) {
7         return 1;
8     }
9
10    execvp(argv[1], argv + 1);
11    perror(argv[1]);
12    return 1;
13 }
```

`sudo1.c`

The program simply `execvp()`s the program it received as the first argument, passing it all the other arguments. If you were expecting to see a system call that “grants privilege”, think again: there cannot be such a system call. If there were, *any* program could have called it, not just `sudo`. The `sudo` command only works because it is owned by `root` and its `set-uid` bit is set. It is this standard mechanism that grants privilege to `sudo`; its job is only to decide whether or not to use its privileges on behalf of the caller. The `src/sudo1.c` version always offers its privilege without objection, but the version in `src/sudo2.c` checks that the caller is listed in a `/etc/sudoers` file and is able to provide a password. Just like `/etc/passwd`, the `/etc/sudoers` file is just a userspace convention and has no meaning to the kernel (nor to any other program that doesn’t read that file).



The liberal `src/sudo1.c` version can still be useful, as you can still use the execute permissions on `sudo1` to determine who is authorised to use it. For example, the administrator could define a `sudoers` group, and then give execute permissions on `sudo1` only to the users of that group.


2.7.2 The need for real and effective ids

Why does the kernel remember both the real and effective uids and gids of a process, when only the effective ones are used for permission checks? There are at least a couple of reasons for this:

- to implement the `access()` system call;

- to allow set-uid and set-gid processes to drop their privilege.

Using the `access()` system call allows a set-uid/set-gid process to request that the kernel checks the permissions to open a file by using its real uid and gid instead of the effective ones. This should be used by set-uid/set-gid programs that accept file names as input and want to verify that their users had the necessary permissions to access those files.

 For an example of a program that could have benefited from this system call see the Ritchie's description of `mail` in Section 4.4. The system call was introduced in v7, probably motivated by this kind of abuses.

However, note that, even if the system call succeeds, the file must still be opened using `open()`, which uses the effective ids. This two-step process (`access()`, then `open()`) is vulnerable to Time-Of-Check-to-Time-Of-Use (TOCTOU) attacks, as we will discuss after talking about links. For this reason, the `access()` system call has been deprecated.

Privilege can be dropped by calling the `seteuid()` and/or `setegid()` system calls. Non-root users (i.e., processes with a non-zero effective id) can use these calls to reset their effective ids to the real ids. Set-uid/set-gid processes may wish to do this as a security measure, once they have completed the task that required the privilege. This is possible in Unix because permission checks are only performed during `open()`. Once a file descriptor has been successfully obtained, it can continue to be used even if the file's permissions or the process's credentials change. Therefore, a process that needs to be set-uid to open a sensitive file, may open it and then immediately call `seteuid(getuid())`, where `getuid()` returns the real uid of the process. This means that the process runs without privilege for the rest of its lifetime, making it less attractive to attackers (see also Section 5.7 for another use).



3. How a Unix shell works

[...] any blank gap is a delimiter of arguments; the first argument is specifically the name of the command; the other arguments are passed over to the command as actual values of formal parameters.

L. Pouzin, *The SHELL: A Global Tool for Calling and Chaining Procedures in the System*, 1965

The shell is used to run other programs and define their parameters, environment and open files. It is also a parser, interpreting and rewriting what we type at the command line in various complex ways.

We will try to understand what a shell does by incrementally building a simple one. A note of caution, however: while we will try to mimic the behavior of a real POSIX shell as closely as possible, we will not try to be complete, efficient, or even compatible with the standard, much less with any existing shell.

We will use the code that can be downloaded from here:

<https://lettieri.iet.unipi.it/hacking/esh-1.0.zip>

The code has been organized as a git repository, with each commit adding a single new feature. Once you have unzipped the file, you can enter the `esh-1.0` directory and run

```
$ git log --oneline
```

to see a list of all the commits. Each commit has been tagged for ease of reference. You can easily see the changes introduced by each commit using git. E.g., to show the changes introduced by 2-path, you can run

```
$ git show 2-path --
```

This shows the differences between the 2-path shell and its immediate predecessor. You can look at the full source of any version of the shell by adding `:esh.c` after the tag. E.g., to obtain the sources of the 7-env shell, just run:

```
1 #include <sys/wait.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <unistd.h>
5
6 #define MAX_LINE 1024
7
8 int main()
9 {
10     char buf[MAX_LINE];
11     int n;
12
13     while ( (n = read(0, buf, MAX_LINE)) > 0 ) {
14         buf[n - 1] = '\0';
15
16         if (fork()) {
17             wait(0);
18         } else {
19             execl(buf, buf, NULL);
20             perror(buf);
21             exit(1);
22         }
23     }
24     return 0;
25 }
```

Figure 3.1 – The simplest shell

```
$ git show 7-env:esh.c
```

If you run

```
$ make revisions
```

you will obtain the sources and the corresponding executable for each revision of the shell as separate files in the current directory.

3.1 The simplest shell (tag: 1-basic)

Figure 3.1 shows the simplest shell imaginable. It is a program that cyclically reads a line of text from its standard input (file descriptor 0) and tries to execute a program with that name. The program is executed in a new process, while the parent process, which is still running the shell, waits for its termination. It uses the process primitives described in Section 2.1.1.

If this is the login shell, file descriptor 0 will still point to the teletype device node opened by `getty` and inherited first by `login` and then by the shell (Section 2.2). Therefore, the user will be able to type commands on the terminal where she has logged into. The commands executed by the

shell will also inherit the files opened by `getty`, and will therefore accept input from, and write output and errors to, the same terminal.

The shell has also inherited its uids and gids, this time from `login`. The real ids will also be inherited by the shell's children. The effective ids will be inherited too, unless a child `execve()` s a program that has the set-uid and/or set-gid flags set. In the normal case these flags are not set, and therefore the command will be executed with the credentials of the current user. The same reasoning applies recursively to any other process that the command itself might create.

The `read()` at line 13 will also give us the newline character that ends the line typed by our user. We replace it with the null character, to get a C string (line 14).

R We did something similar in `login` in Section 2.2, but there we had a maximum length for valid login names. Here, the user could, in principle, type more than `MAX_LINE` characters, but we assume that she'll kindly refrain from doing so; we'll ignore this issue until Section 3.6.

We use `execl()` as we did in all the programs in Section 2.2. To follow the convention that the first element of `argv` must be the program name, we pass `buf` twice.

R Actually, the convention is a bit different: if `buf` contains any `"/"`, we should pass in `argv[0]` only the *basename*, i.e., the string that follows the last `"/`. We leave this out for simplicity—the convention is not followed by all programs anyway.

Remember that only the first argument is interpreted by the kernel as a path, while the second is just copied into the process memory and made available via `argv[0]`.

Now let's go to our home directory and start our minimal shell, then type `/bin/ls` and press enter. We should see the contents of the directory. We can type the full path of other commands, such as `/bin/ps`, or press Ctrl+D to send an EOF and cause our mini-shell to terminate (since `read()` will return 0).

Now remember how `execve()` interprets its first argument, the path of the new program to execute (reread Section 2.1.3 carefully if necessary), and try to guess what will happen if we start our minimal shell and then type `ls` (followed by Enter) assuming that the current directory is still our home.

Did you guess right? `execve()` fails and we get a “no such file” error. This is because the `PATH` variable is not used, `execve()` gets the string “`ls`” as its first argument, and this represents a relative path starting from the current directory (since it does not start with `/`). The current directory will (probably) not contain an executable file called `ls`, and therefore the kernel will not be able to find it. To confirm this, we can exit from our mini-shell (Ctrl-D), copy the `ls` program to our current directory (to avoid confusion, let's call it `myls`):

```
$ cp /bin/ls myls
```

Then run our mini-shell again. This time, typing `myls` will work.

Exercise 3.1 The `wc` command stands for Word Count: if run without arguments, it prints the number of lines, words and characters received from its standard input. What happens if you type `/usr/bin/wc` immediately followed by Ctrl+D? Why would this happen? ■

Exercise 3.2 Close `1-basic` and create a script text file containing any shell command, make it executable (`chmod +x script`), restart `1-basic` and try to run the script. What happens? Why? ■

Exercise 3.3 Do as in Exercise 3.2, but add a line with `#!/bin/sh` at the very top of the script file. What happens now? Why? ■

Exercise 3.4 If you try to use advanced editing keys such as the arrow keys, you get only strange characters in response. What are these characters? Why don't the keys work as expected? ■

3.2 Using PATH (tag: 2-path)

Now let's add the `PATH` variable into the picture. All we have to do is replace `execl` with `execvp` in Figure 3.1.

The `execvp()` function works much like `execl()`, and it must call `execve()` to ask the kernel to run a new program (there is no other way, remember). Therefore, it needs a path to the new executable, to pass it as the first argument to `execve()`. As a convenience, the function is able to *search for* a program in the set of directories listed in the `PATH` variable (separated by colons). This way the user is not forced to always type the full path of each command. More specifically, it works like this: to get the path of the executable, the function appends its first argument to each one of the paths listed in `PATH` in turn (adding a `/"` in between, if necessary) until it finds an existing executable file with the resulting path. If the list of directories is exhausted and no executable file is found, the function returns with an error¹.

However, there are occasions when the user wants to execute a program that is not in any of the directories listed in `PATH`. To accommodate for this use case, the function introduces a distinction between *commands* and *paths*: if its first argument contains *no slashes*, then it is a command; otherwise, it is a path. Only commands need to be converted to paths using `PATH`, while paths are passed directly to `execve()` without any further processing.

Now let's start our modified mini-shell in our home directory and let's type `/bin/ls` (a path) and then `ls` (a command) as before. This time both strings work: the first one is passed as-is to `execve()` and the kernel interprets it as an absolute path that successfully leads to the `ls` program. The latter one also works because the `PATH` variable most likely contains the `/bin` directory, and therefore `execvp()` will succeed in finding the `ls` program when it tries the `/bin/ls` path.

Now, assuming that `myls` is still in our current directory, let us type `myls` and, before hitting enter, let us try to guess whether it will work or not.

This time we get an error. And why is that? Because the `myls` string is classified as a command, since it contains no slashes, and therefore `execvp()` will look for it in the directories listed in `PATH`, and *only* there. The current directory is, most likely, not listed in `PATH`, and therefore `execvp()` will not be able to find our program.

Note that the above behaviour is caused by a corner case in the classification operated by `execvp()`: strings without slashes are legitimate paths according to the kernel (they are relative paths), but they are commands according to `execvp()`. Since `execvp()` runs first, its interpretation wins.

If we want to use `PATH` and still be able to run a program that lives in the current directory we have a few options:

- use a path that leads to our program and contains a `/"`;
- add the current directory to `PATH`.

The first method causes `execvp()` not to recognize the string as a command and pass it directly to the `execve()` system call. A common trick is to type `./myls`—a redundant path that has the required slash and is completely equivalent to `"myls"` as far as the kernel is concerned.

¹Check `lib/execvp.c` in `myunix` for the full story.

The latter can be done in many ways. Of course you can add the absolute path of any directory to `PATH`, but you can also add *relative* paths to it. Adding the “.” path will cause the `execlp()` function to also look for files in the current directory (essentially recreating the “./mys” path by itself).

Perhaps little known, but implicit in the above description, is the fact that `execlp()` will look in the current directory even if `PATH` contains an *empty* path: it will not concatenate anything to the input path, will not add a / since there is nothing to separate, and therefore will get the same path as before. If `PATH` contains an empty path and we type `mys`, `execlp()` will also try to pass `mys` to `execve()`, which will find it. An empty path exists if `PATH` starts or ends with a colon, or if it contains at least two colons with nothing in between.

Exercise 3.5 Assume that the current directory contains a subdirectory called `utils` that contains an executable called `exe`. Now we type

```
utils/exe
```

into `2-path`. Will it work? Does it depend on the contents of `PATH`? ■

Exercise 3.6 Do as in Exercise 3.2, but using `2-path` instead of `1-basic`. What happens now? Why? ■

3.3 Splitting the command line (tag: 3-words)

Now let us use `2-path` and try to type “`ls -l`”. Will it work?

No, the kernel will look for a program called “`ls -l`”, which most likely does not exist (but it could have existed: remember that spaces are allowed in file names).

The splitting of what we typed into the name of the command and its arguments will not happen by magic. Go through what we have said until now, and try to find the place where we said that something splits a string into words: you will not find it, because it does not exist.

Splitting the command line into words is one of the main tasks of the shell. The first word becomes the first argument to `execve()`, possibly after `PATH` processing. All the words (including the first one) are assembled into the array passed as the second argument. Figure 3.2 shows the `main` function of the `3-words` shell, which implements the processing we have just described. If you compare it with the shell in Figure 3.1 you will see that we have added a couple of function calls between the input of the command line and the creation of the new process. The first function is `getwords()`, that splits the line into words using whitespace characters as separators. For each word found, the function fills an element of the `words` array of `word_t` descriptors; then, it returns the number of words it found. If there are no words (the line consisted only of whitespace) we can continue with the next line of input; otherwise, we build the `c_argv` vector from the array of words, using the `buildargv()` function. The `getwords()` and `buildargv()` functions are defined at the bottom of the `esh.c` file in the repository, and just contain some strings and pointers manipulations.

■ **Example 3.1** See Figure 3.2. The figures in this chapter show the evolution of some data structures over time, using the following conventions: time runs from top to bottom; the horizontal dashed lines indicate the execution of the function mentioned at the beginning of the line; data structures that are modified by the function are replicated below the line, with a prime added to their name; data structures that do not change are not replicated.

The user has typed the string “`__ls__-l__`” followed by a newline. The top of the figure shows the contents of the `buf` array after line 14 in Figure 3.2, where the newline has been replaced by

```
1 #include <sys/wait.h>
2 #include <ctype.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <unistd.h>
6
7 #define MAX_LINE 1024
8 #define MAX_WORDS 10
9 #define MAX_ARGS 10
10
11 typedef struct {
12     char *w;
13 } word_t;
14
15 int getwords(char *buf, word_t words[]);
16 int buildargv(char *argv[], word_t words[], int nwords);
17
18 int main()
19 {
20     char buf[MAX_LINE];
21     int n;
22     word_t words[MAX_WORDS];
23     char *c_argv[MAX_ARGS + 1];
24     int nwords;
25
26     while ( (n = read(0, buf, MAX_LINE)) > 0 ) {
27         buf[n - 1] = '\0';
28
29         nwords = getwords(buf, words);
30         if (!nwords)
31             continue;
32         buildargv(c_argv, words, nwords);
33
34         if (fork()) {
35             wait(0);
36         } else {
37             execvp(c_argv[0], c_argv);
38             perror(c_argv[0]);
39             exit(1);
40         }
41     }
42     return 0;
43 }
```

3-words.c

Figure 3.2 – A shell that splits the command line into words

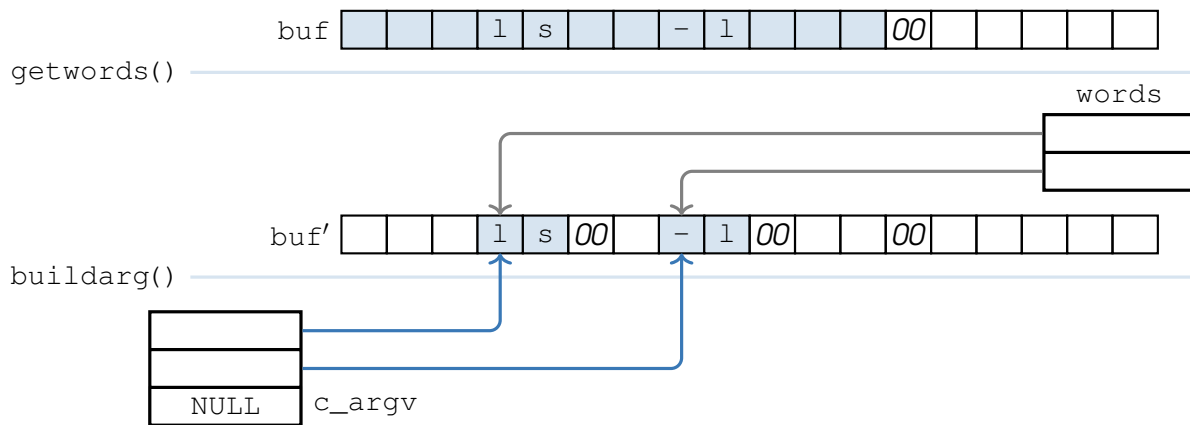


Figure 3.3 – Example execution of `getwords()` and `buildargv()`

00. The `getwords()` function skips the first three spaces and finds the first word (“ls”). It replaces the following space with 00 to terminate the string. It skips another space, finds the second word (“-l”), and terminates it too. Finally, it skips two more spaces without finding another word. The result is shown in the `buf'` and `words` arrays in the figure. The `buildargv()` function creates the `c_argv` vector by picking both words and adding the terminating NULL. ■

R Rather than having two distinct functions, one for word-splitting and the other to build `c_argv`, we could have done everything in a single step. However, we will see in a moment (Section 3.5) that some input words are not passed to the command, so it is cleaner to keep the two tasks apart.

The first element of `c_argv` (i.e., `c_argv[0]`) and a pointer to `c_argv` itself are then passed to `execvp()`. This is another “exec” variant that invokes the `execve()` system call under the hoods. Like `execlp()`, it uses `PATH` on its first argument and copies the parent environment to the child. Unlike `execlp()`, though, it does not build the `argv` vector by itself and just uses its second argument *as-is*.

Now, if we run our new shell and type “ls -l”, we will see the long listing of the current directory. We can pass any number of arguments (well, up to `MAX_ARGS`) to any command, and everything will (mostly) work.

Note how the shell only splits the line, but the meaning of the resulting words (besides the first one) is entirely up to the commands. They receive them in the `argv` parameter of their `main` function and they can do whatever they want with them. It is only by convention that many programs (but by no means all of them) understand arguments starting with “-” as an option, or a single “-” as the standard input file, and so on. One needs to check the documentation of each command to learn these details when needed. Since many commands were invented while the conventions had not yet been fixed, and others were invented by different groups of people in the long history of Unix, you will find a lot of inconsistencies.

Exercise 3.7 — catdash. Try to solve challenge *catdash*. This challenge uses 3-words as the system shell, only slightly modified to print a prompt. You have to print the “-” file and you only have `cat` (source in `/usr/src/cat.c`). ■

3.4 Shell builtins (tag: 4-builtin)

These peculiarities are inexorably imposed upon the shell by the basic structure of the UNIX process control system. It is a rewarding exercise to work out why.

K. L. Thompson, D. M. Ritchie, *Unix Programmer's Manual* (v2)

Now that we can pass arguments to commands, let's try to use `cd` to change the working directory. Assume that the current directory contains a `subdir` subdirectory (create it if it doesn't). Let us start `3-words` and type

```
cd subdir
```

Will it work?

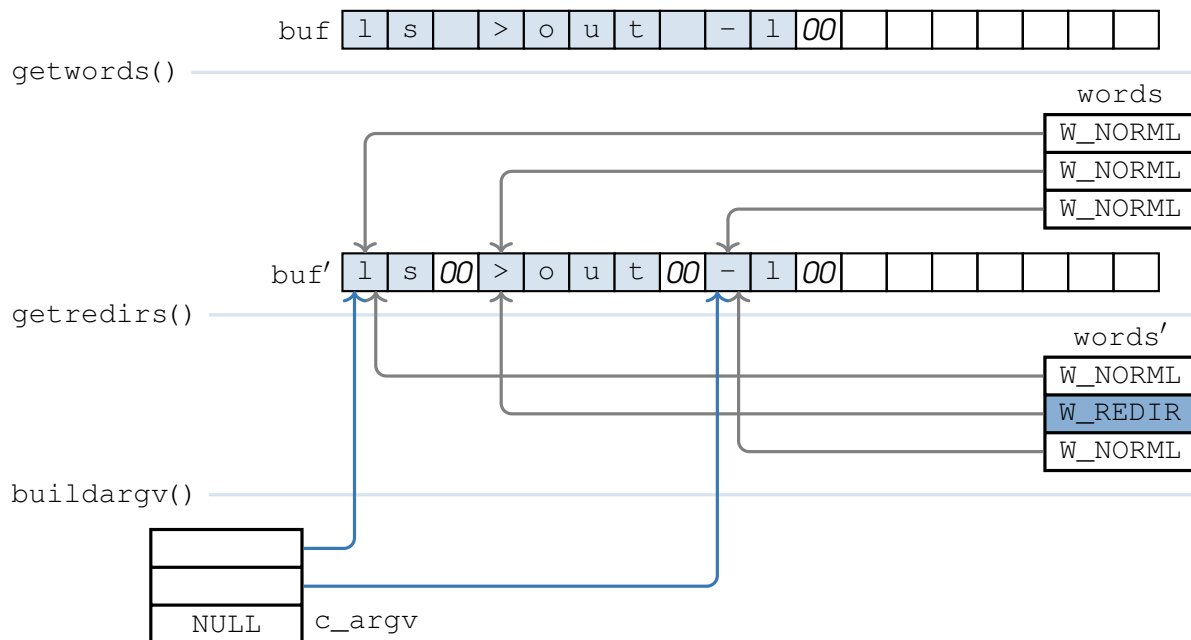
No, `3-words` will look for a `cd` binary to execute, but there is no such binary in the file system. Such a program would have to call the `chdir()` system call, but this system call only changes the current working directory *of the process calling it*. That is, it would change the current working directory of the *child* process that the shell would have created to run `cd`. The shell itself would continue to use its old working directory, and we would have accomplished nothing.

If you guessed wrong, it might make you feel better to know that Ritchie and Thompson made the same mistake when they added multiprocessing support to Unix. Before that, there *was* a `cd` command (it was actually called `chdir`). The command stopped working when they implemented `fork()`, and they were confused by it, at least for a while [see 23, page 6].

For `cd` to work, the shell must call the `chdir()` system call by itself. The `4-builtin` shell does it, and you can see the necessary differences with “`git diff 4-builtin^!`”. The only difference with `3-words` is that after the call to `buildargv()` the first element of `c_argv` is compared to the “`cd`” string. If the strings match, `4-builtin` calls `chdir()` without creating a new process; otherwise, it continues as before.

The commands that are executed directly by the shell are called “shell builtins”. Whenever a command needs to affect the environment of the shell itself, so that it can be inherited by all later commands, we need a shell builtin. Other common builtins are `umask` and `ulimit`. Over time, shells have acquired other builtins that were not necessarily needed, because it is more efficient to run something in the shell than to spawn a new process each time. For example, the simple `echo` utility is a builtin in most shells. For another example, the “`:`” `nop` become a shell builtin very early (the `v4` shell already had it). Another reason for adding a builtin is to take advantage of the greater knowledge that the shell may have about the current state. For example, `pwd` (print working directory) is also implemented as a builtin in many shells because, unlike the external command, the shell can remember when it has traversed a symbolic link and can display it in the path. If you want the external program instead, you can always invoke it by typing its full path (e.g., `/bin/echo` or `/bin/pwd`). More precisely, it is the presence of a slash in the string that disables the builtins, as well as `PATH` and aliases (which we will not discuss).

Exercise 3.8 Implement the `umask` builtin (`help umask`) and compare it with the one in `4-builtin.2`. You can limit yourself to the octal output and input syntax. ■

Figure 3.4 – Example execution of `getredirs()`

3.5 I/O redirection (tag: 5-redir)

Input and output redirection is one the first shell features implemented by Thompson: it had been available for many months even before `v1` come out. The design of the Unix basic system calls make it very simple to implement: assume you want, say, `ls` to write its output in a file, instead of printing it on the terminal; then, in the child process created by the shell, before calling `execvp()`, you `close(1)` and `open()` the file: the kernel will pick the first unused file descriptor, which will be the `1` we have just closed, essentially replacing the standard output file of the process. Since `execvp()` will then replace the program, but not the process, `ls` will write into the file, without even knowing about it. Note how the mechanism will work with all the programs that honor the standard input/standard output convention, and we have achieved this without adding any new system call.

We implement this idea in our next shell, `5-redir`. We adopt a simplified syntax, similar to the one used in the earliest versions of the Thompson shell: input redirection is obtained by writing a word like `<file` and output redirection with a word like `>file`. These must be *words*, i.e., they should not contain any space (not even between the `<` and `>` operators and the filename) and must be separated by whitespace from the other words. The implementation is then very simple. First, we add a field `type` to the word descriptor. Then, in the main process:

1. `getwords()` first assigns the “normal” type to each word it finds;
2. we then call a new `getredirs()` function that scans the words array and changes the type to “redirection” for the words that start with either `<` or `>`;
3. `buildargv()` now picks only the words that still have a “normal” type.

■ **Example 3.2** See Figure 3.4. The user has typed `ls > out -l` (note that redirection words can appear anywhere on the command line). The `getwords()` function has identified three words and has marked them all as “normal” (`W_NORML`). Then, the `getredirs()` function changed the type of the second word to “redirection” (`W_REDIR`) and `buildargv()` took only the first and the last words. ■

In the child process, a new function `redirect()` searches the array of words for those of type “redirection”, and performs the necessary `close()` and `open()` system calls before the call to `execvp()`.

Note that to implement the `>file` redirection when `file` doesn’t exist, the shell must be able to create a new file given only the path. This is only possible because of the radically simple notion of files introduced by Unix: all files are just sequences of bytes, so we don’t need to specify the type; the size grows automatically as we write to it, so we don’t need to specify it beforehand; owner and group are implicitly obtained from the creating process. However, there is still some information that the shell doesn’t have: in particular, the `open()` system call wants to know the initial read/write/execute permissions for the owner, group, and other users. This is a situation common to most Unix programs that need to create a file and are only given a path by the user; historically, they have always preferred ease of use over security and have simply given all relevant permissions to everyone. This is what we did in our shell—we just removed “x” permissions, since we expect files created in this way to contain data rather than code. This “liberal” default behaviour is the second problem with Unix file permissions that Ritchie mentioned in his “On the Security of UNIX” paper (see Section 4.2). The v7 version the paper mentions the introduction of `umask()` (see Exercise 3.8) as a partial mitigation to this problem.

Exercise 3.9 The `>` operator clears the file if it already exists (option `O_TRUNC`). The shell in v2 UNIX (1972) introduced the `>>` operator that appends output to the file if it exists. Try to implement this feature and then compare your solution with the one in `5-redir.2` ■

R Today we implement features like `>>` by passing flags to `open()`, but Research UNIX never introduced these flags: the UNIX shells just use `creat()`, `lseek()`, and `open()` as needed. However, the flags approach is not just for convenience: it ensures that these actions are performed atomically.

Exercise 3.10 Assume that you have to write something in a file `f` where you have no write access. You are listed among the sudoers, so you try

```
$ sudo echo something >f
```

but you still get a *permission denied* error. Why? How can you solve your problem? ■

3.6 Scripting (tag: 6-intr)

Buffered IO was, and still is, a necessary evil.

M. D. McIlroy, *A Research UNIX Reader*, 1987

Since the shell is a program like any other, we can call it recursively. If we redirect its standard input from a file in which we have placed some shell commands, the new shell will execute them one by one without further intervention. This gives us a (somewhat limited) scripting capability, without adding any new ad hoc mechanisms to the system. This was also the way scripting was implemented in the pre-v7, minimalist Thompson shell.

However, if we try this with the shells that we have built so far, it will not work. The reason is that we have assumed that each `read()` call will always return exactly one line. However, this is only true if we are reading from a terminal configured for line processing (and the line fits into the input buffer). This assumption fails especially if we call `read()` on a file: the system call will just try to fill the buffer, without stopping at newlines. The shells that we have written so far are not prepared for this.

■ **Example 3.3** Put a couple of commands in a script file, e.g. `ls` and `pwd`, each one on its own line. Then, run our redirection-capable shell, `5-redir`, and call it recursively with input redirected from the script:

```
$ ./5-redir
./5-redir <script
```

(The second line is the input to `5-redir`.) We should get the following error:

```
ls: pwd: No such file or directory
```

This comes from `ls` looking for a “`pwd`” file in the current directory. The `5-redir` shell received the entire file contents in a single `read()`, including the newline between `ls` and `pwd`. Since the newline counts as whitespace, the shell split the string in two words and then tried to run `ls` with `pwd` as an argument. ■

To always get exactly one line from standard input, whether it points to a terminal, a file, or something else, we can use the stdio library function `fgets()`, which is what we do in our next shell, `6-intr`. Unfortunately, there is a catch: the stdio library does its own buffering in userspace, to reduce the number of `read()` system calls and improve performance. This means that, if standard input is a file, our shell will get one line at a time from `fgets()`, but `fgets()` itself will still read *blocks* of bytes from the underlying file. This is a problem when the shell spawns another command that also needs to read from standard input: some of the bytes intended for the command may already have been eaten up by `fgets()` by the time the command runs (see Exercise 3.11). The simplest solution to this problem is to always read only one byte at a time from standard input, either by calling `read()` directly, or by disabling stdio buffering on standard input: this is the purpose of the “`setbuf(stdin, NULL)`” call in the `6-intr` sources. The call to `fgets()` is in the new `getcmd()` function. We take the opportunity to simplify things a bit and don’t remove the newline from the input buffer: since the newline is whitespace, `getwords()` already takes care of it.

3.6.1 Interactive vs non-interactive

Our shell can now be used either “interactively” or to run scripts. Shells actually try to understand when they are being used interactively, and change their behaviour slightly to be more human friendly. For example, shells print a prompt when they are waiting for input in interactive mode. A simple strategy for inferring that there might be a human being on the other end of `stdin` is to check if it is attached to a terminal: this can be detected because there are some `ioctl()` system calls that only apply to terminals, and will fail if used on anything else. The library function `isatty()` uses this trick to determine if a file descriptor points to a terminal. The `6-intr.2` shell revision uses this function to set an `interactive` global flag, which is then checked by `getcmd()` to decide whether a prompt is needed or not. Note that the prompt changes based on the effective user id of the process running the shell: `$` for normal users and `#` for root.

Interactive mode also differs from non-interactive mode in the way the shell handles input errors, such as nonexistent commands or syntax errors. In interactive mode, a diagnostic is usually printed, but the error is otherwise ignored. In non-interactive mode, however, the shell stops executing the script. The idea is that the human user can correct the error and retry, while the script cannot. The `6-intr.3` shell implements this behavior. Note that our shell doesn’t look at the value returned by the commands it spawns, and therefore doesn’t handle errors in *their* execution. This is also essentially true for real POSIX shells, unless the user has explicitly set a shell flag (“`set -e`”).

Another difference between interactive and non-interactive mode is in the way the shell handles terminal interrupts. If you type Ctrl+C in any of the shells that we have developed so far, the shell will terminate. This is not the expected behavior of an interactive shell: in fact, when running interactively, shells should ignore SIGINT (the signal the kernel sends by default when we type Ctrl+C) and SIGQUIT (sent by Ctrl+\). This behavior is implemented in 6-intr.3. Note that the shell restores the handlers for these signals in the child process, so we can abort a misbehaving command and return to the shell prompt.

In place of this, BSD introduced a very complex and un-Unix “job control” feature in the kernel and in the shell [39]; this was later adopted by POSIX and is now implemented in all modern shells.

Exercise 3.11 Remove the “`setbuf(stdin, NULL);`” line from the 6-intr sources and recompile the 6-intr shell. Write a file `script` containing the following lines:

```
cat
hello
```

Execute this first with a standard shell, e.g. by running “`bash <script>`”, then run “`./6-intr <script>`” and try to understand what is going on. If you are feeling adventurous, try to repeat the experiment with more “hello” lines at the bottom of the script. ■

3.7 Environment variables (tag: 7-env)

Now consider shell’s support for environment variables, such as `PATH` and `HOME`. Environment variables were added in v7, when Steve Bourne rewrote Thompson’s shell to make it more suitable for programming.

These variables can be used to personalize the user’s environment or to remember values across program executions. From the kernel’s point of view, environment variables are just another set of strings that `execve()` must copy into the process’s memory. Everything else about them is just convention, from their syntax to their meaning.

- Syntactically, these strings should be of the form *variable=value*, where *variable* should look like an identifier, starting with a letter or underscore and then containing only letters, numbers and underscores. However, the kernel does not check that any of these rules are actually followed: it just copies null-terminated strings, whatever they are. The C library assumes that these conventions are followed, and provides some functions to work with them: `getenv()` to get the value of a variable given its name, `setenv()` to create a new variable or, optionally, overwrite an existing one, and `unsetenv()`, to remove a variable completely.
- Semantically, some of these variables have meanings that are understood by most Unix programs. `PATH` is an obvious example, since its meaning is embedded in the C library functions that are used to run programs. Another example is `EDITOR`, which users can set to their preferred editor. Programs that need to spawn an editor should look at this variable and start the user’s preferred editor. However, nothing in the system enforces these rules, and each program is free to ignore any environment variable or assign a different meaning to it.

The most important thing to remember, if you really want to understand how environment variables work, is that they are *local to each process*, and that they originally come from the parent of the process. Unix users often forget this because environment variables look like a “global” thing. This illusion is

created by the fact that most programs simply pass to their children the environment that they have received from their parent. This behavior is encouraged by the C library, where most `exec*` variants (including the ones that we have used so far) copy the current environment under the hood (i.e., they pass the current value of the `environ` pointer to `execve()`). This illusion breaks if you want to change the value of a variable, or create a new one. This change is only visible in the current process and in its future children: you cannot change the environment of another process, since you cannot (normally) write to its memory.

Shell support for environment variables comes in two forms:

1. the shell maintains an environment that can be passed to its children, and provides some syntax for updating it;
2. the shell can “expand” the value of a variable as it parses the command line.

The latter is a form of macro processing: some text is replaced by other text, often without regard to the syntax of the resulting command line.

Supporting environment variables is the most complex addition to our shell. We add it a bit at a time.

3.7.1 Updating the environment

The shell `7-env` implements point 1 above, with the following syntax. To assign a value to an environment variable, use

variable=value

Again, the whole assignment must be a word: it must be delimited by whitespace and it cannot contain spaces (not even around the “=” operator)



Unlike the analogous limitations in Section 3.5, these ones apply also to modern shells, but see Section 3.8.

You can have more than one assignment on a single command line, separated by spaces. The assignments change the environment of the shell and of all subsequent commands. As a special case, if you write a command on the same line as the assignments, only the environment of that command is updated. To delete one or more variables use the new builtin `unset` followed by the names of the variables.

The implementation is simple: after `getredirs()`, and before `buildargv()`, we call a new function `getvars()`. Just like `getredirs()`, the new function scans the array of words looking for the ones that match the above syntax (an identifier immediately followed by =). It marks the matching words as “assignments”, so that `buildargv()` will skip them (since it only picks “normal” words). Unlike `getredir()`, it stops at the first non-matching word.

Bourne’s shell initially looked for assignments in all the words of the command line, but this conflicted with some commands (like `dd`) that use the assignment syntax for their own arguments.

■ **Example 3.4** See Figure 3.5. The user has typed “`CC=gcc_make_V=1`”. The `getwords()` function identifies three words and assigns the “normal” type to all of them. The `getredirs()` function doesn’t change anything because it doesn’t find any redirection word. Then, the `getvars()` function changes the type of the first word to “assignment” (`W_ASSGN`), because the word starts with a valid identifier immediately followed by an equal sign. Note that the third word is also syntactically a valid assignment, but it is not classified as such, because it comes after `make`, which is not an

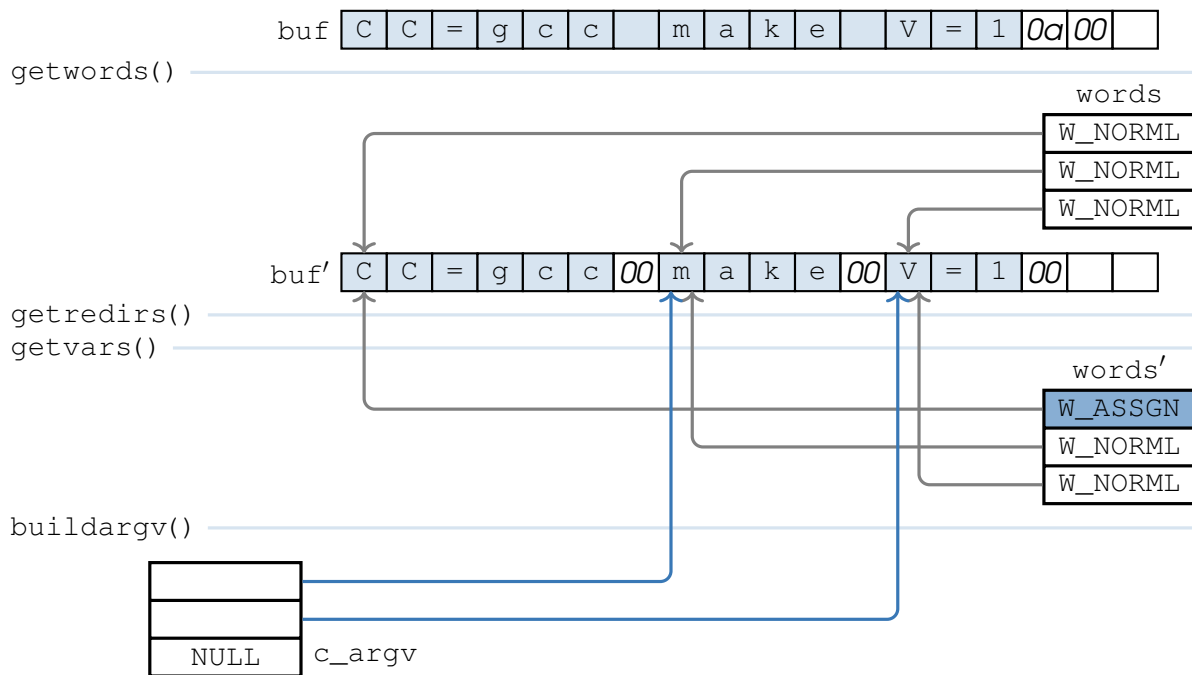


Figure 3.5 – Example execution of `getvars()`

assignment. Therefore, `buildargv()` takes both `make` and `V=1` as arguments. The `make` command will start with the variable `CC=gcc` in its environment, while `V=1` will be received in `argv[1]`. ■

The new function `assignvars()` rescans the array of words looking for those with type “assignment” and performs the assignments using `setenv()`. The `buildargv()` function now returns the number of elements it has put into `c_argv`. If the array is empty, `assignvars()` is called in the main process, otherwise it is called in the child process, to affect only the environment of the spawned command. Note that there is no need to change the call to `execvp()`, since this function will copy the current environment under the hoods.

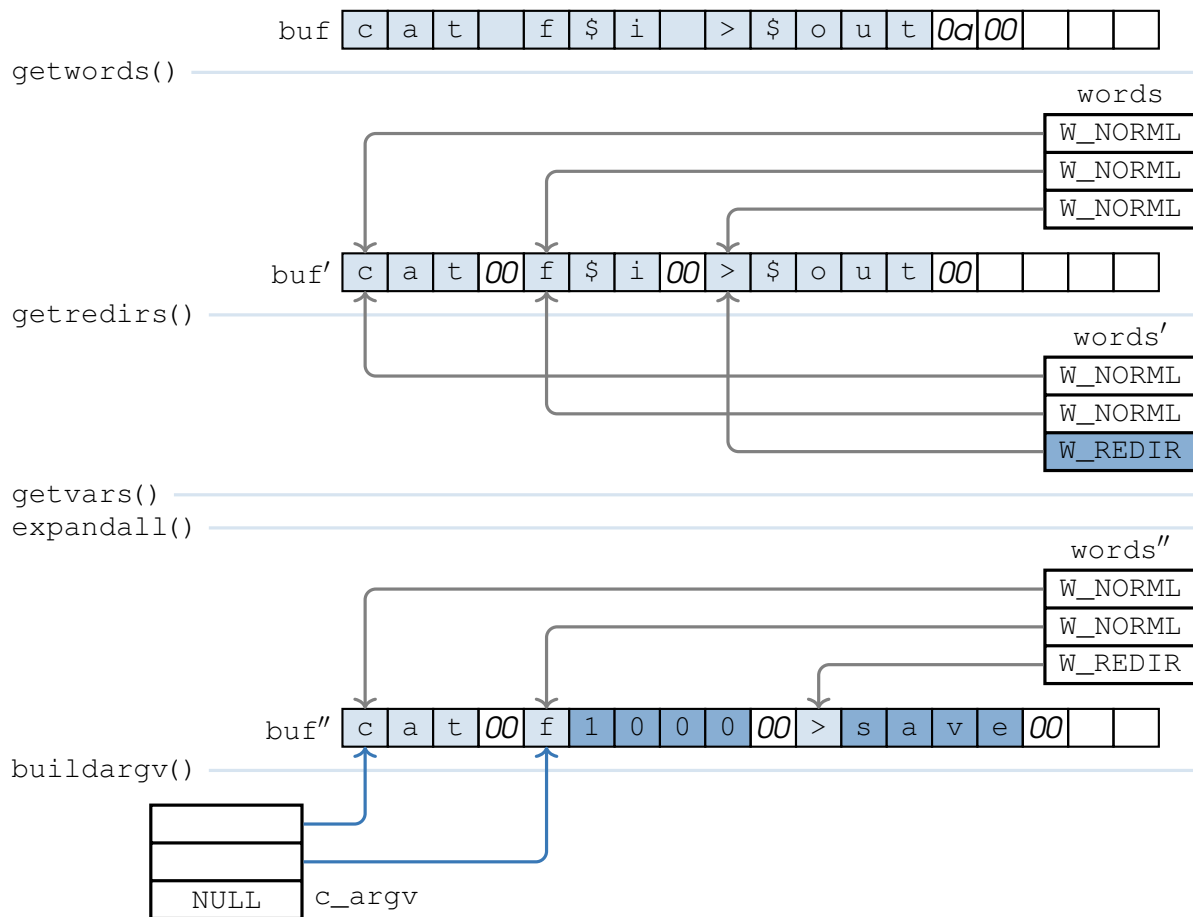
- R** POSIX shells behave a little differently than our own, distinguishing between *shell* and *environment* variables. Assignments to non-existent variables, for example, create only internal shell variables, which are not automatically copied to the child environments. To add a shell variable to the environment passed down to children, you have to `export` it, unless the `-a` flag is set, in which case all variables are exported automatically. Our shell behaves as if `-a` had been set.

In a real system the environment starts empty when `init` is run, but then other programs can add variables to it. For example, `login` adds the `HOME` variable set to the path of the home directory read from the `/etc/passwd` file (see `src/login2.c` in `mynix`). The shell inherits this variable and uses it as the default path for `chdir()` when you type `cd` without an argument (this is also implemented in our shell).

The shell itself can provide default values for the variables that are essential to its operation (e.g., `PATH`, `IFS`, `PS1`, `PS2`, ...).

3.7.2 Variable expansion

To expand the value of a variable, use `$variable`, followed by a space or a non-alphanumeric character anywhere on the command line. The expansion replaces the `$variable` string with the value of `variable` if it exists, and with nothing otherwise.

Figure 3.6 – Example execution of `expandall()`

The implementation is conceptually simple, but string manipulation in C is cumbersome and error-prone. Unlike all the manipulations that we have done so far, variable expansion can *increase* the number of characters we have to store in the input buffer, so we must be careful not to overflow it. To simplify the task we introduce a “struct `obuf`” that contains a pointer to a buffer and a counter of the available space in it. New characters must be added to the buffer only using the new `oput()` and `oputs()` functions, that turn into NOPs if there is not enough room. The `avail` field of `obuf` can be checked at any time: if we find it negative we can signal an error to the user.

The `7-env.2` shell calls a new function `expandall()` after `getvars()` and before calling `buildargv()`. The new function passes each word to `expandword()` for possible expansion. The function `expandword()` produces the expanded version of the word: it scans each character in the word string and copies it to a new buffer, unless the character is a `$` followed by an identifier: in that case, the function skips the identifier in the source string and places the (possibly empty) value of the corresponding variable in the new buffer (with the help of the function `expandvar()`). Note that `$` can be anywhere in the word, not just at the beginning, and that you can have more than one expansion in the same word. Expansion can also occur in words of the type “redirection” or “assignment”. Also note that the output of the expansion is not re-scanned for more expansions.

■ **Example 3.5** See Figure 3.6. The user has typed “`cat_f$i_>$out`”. Assume that the environment contains the definitions `i=1000` and `out=save`. The `getwords()` function identifies three words

(see `buf'` in the figure): note the presence of dollars in the middle of the second and third words. The `getredirs()` function classifies the third word as “redirection” because it starts with `>` (the presence of dollars in the word doesn’t affect the classification). The `getvars()` function doesn’t change anything because the first word is not an assignment. The `expandall()` function scans all the words, replaces the `$i` and `$out` substrings with `1000` and `save`, respectively, and adjusts the pointers in the `words` array (see `buf''` and `words''`).

R For simplicity, `expandall()` internally creates a new array of words (`tmpwrd`) and a new buffer `tmpbuf`, and copies them over the original ones using the utility function `updatewords()`.

Note that the replacement can affect the contents of words, but not their classification. In particular, the `redirect()` function will see the `>save` redirection word and will open the `save` file, without any knowledge of the `out` variable. Also the `cat` program will receive `f1000` in `argv[1]` and will never see the original `f$i` string. ■

The `expandall()` function can sometimes *reduce* the number of words. We have already seen that undefined or empty variables are expanded to nothing; if a word becomes empty after expansion, it is removed from the array of words. This is the purpose of the `if` after the call to `expandword()` in the `expandall()` function. For example, if the user types “`echo XY`” and both `X` and `Y` are undefined (or empty), `echo` will only get one argument (i.e. `argv[0]`).

Exercise 3.12 Assuming that `X` doesn’t exist yet, try to guess the output of

```
$ X=aaa echo $X
```

in either your normal shell, or in `7-env.2` (it should be the same). ■

3.8 Quoting (tag: 8-quote)

Twenty years after the publication of the first Posix shell standard, members of the standards working group are still debating the proper behavior of obscure quoting.

C. Ramey, *The Bourne-Again Shell*, in *The Architecture of Open Source Applications*, 2011

We have introduced a few characters that are used by the shell itself and are not passed to the commands: whitespace separates words, with newline terminating commands; “`<`” and “`>`” at the beginning of words are used for redirection; “`$`”, when followed by an identifier, is used for variable expansion; a “`=`” occurring in a word may turn it into a variable assignment. These are called *shell metacharacters*. We may know from experience that we can pass these characters to commands by *quoting* them, but where is the quoting implemented? Create a file with a space in its name using your normal shell, e.g.:

```
$ touch "a space"
```

Then start our latest shell, `7-env.2`, and try to run some command on the file you just created, e.g.:

```
cat "a space"
```

We get errors from `cat`, which cannot find the “`a`” and “`space`” files. This is because single and double quotes, as well as backslashes, are more metacharacters that are parsed *by the shell*, but our

shell doesn't know how to do it yet. When we have issued the `touch` command, your normal shell has recognized the quotes and has passed "a space", as a single string and with the quotes removed, to the `touch` program (in `argv[1]`). Our shell, on the other hand, has treated the "" characters as normal characters: it did not use them to protect the space between "a" and "space" and it did not remove them from the command line.

The quoting metacharacters are single (') and double (") quotes and backslash (\). Backslash removes the special meaning of the following character², while single quotes remove the special meaning of all the characters up to the first single quote. Double quotes are a bit more complex: they remove the special meaning of all the following characters up to the first double quote, except for backslash, but only if it is followed by one of a few special characters.

R The full set includes another backslash, double quotes, dollar and newline; in the first revision we will only consider other backslashes and double quotes.

Don't think of quotes as defining "strings", like in most programming languages: for the shell, everything is already a string by default. You need quotes only when you have to stop the shell from interpreting some of its metacharacters. Moreover, you can have quotes even in *in the middle* of words. For example, the word `a"$c` becomes the single word `a$c` after quote processing. Quoting characters that have no special meaning has no effect: the strings `"abc"`, `a"bc"`, `a"b"c` or even `" "abc` and `abc" "` are all equivalent to `abc`.

Why three metacharacters for quoting? The backslash is convenient when you need to protect just one metacharacter, while the quotes are convenient when you need to protect several of them in a row. Moreover, since the quoting characters are themselves metacharacters (interpreted by the shell and then removed from the input), you need a way to quote *them* when needed. You can quote the backslash with another backslash. In the earliest shells backslash had no special meaning inside quotes, so you could not use it to put quotes inside quotes. However, you could quote single quotes by putting them in double quotes and vice versa. Over time the double quotes have acquired some special behavior: now you can put a double quote within double quotes using backslash (`"\"`) but you cannot do the same with single quotes (see Exercise 3.15 for another special behaviour of double quotes).

If you *really* need to put a single quote in a single-quotes string, you can do as follows:

```
$ echo 'don'""'t do this at home'
```

Of course this is cheating, since you are actually just concatenating three strings (`'don'`, `""` and `'t do this at home'`), none of which contains single quotes within single quotes.

The 8-quote shell implements the quoting mechanism. Immediately after receiving a line of input from `getcmd()`, we pass the input buffer to the new function `quote()`. The function allocates a new buffer `tmp`, then scans each character of the input buffer, looking for quoting metacharacters. Unquoted characters are simply copied to the `tmp` buffer. Quoting metacharacters are used to copy a "quoted" version of one or more characters, according to the rules outlined above. To implement the "quoted" version of characters we steal the idea from the original Thompson shell: we mark the characters by setting their most significant bit (see the `QUOTE` constant that is or-ed to characters). The `quote()` function then replaces the input buffer with the `tmp` buffer. Now, quoted spaces, dollars, and so on, will be hidden from the eyes of `getwords()`, `getredirs()`, `getvars()`, `expandall()`, and any other string processing function that we may add later. Finally, before actually passing the

²Except when the character is newline, see Exercise 3.16.

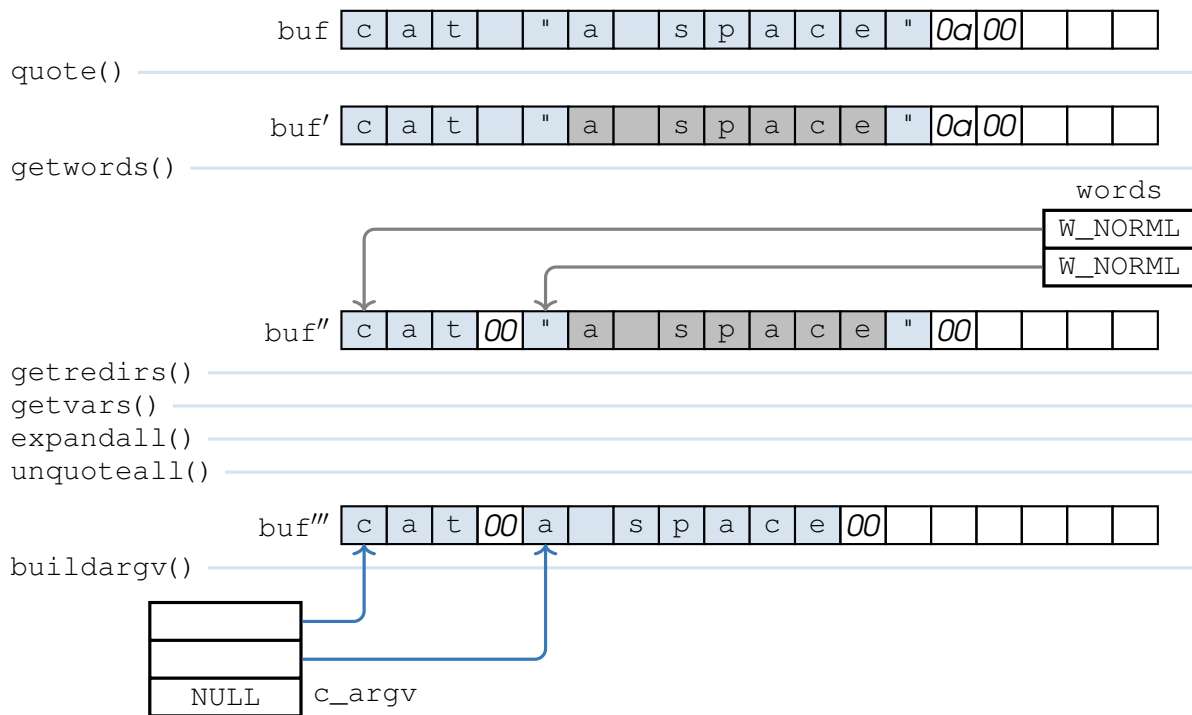


Figure 3.7 – Example execution of `quote()` and `unquoteall()`

argument vector to the commands (either builtin or external), we use the function `unquoteall()` to remove all the quote metacharacters and quote markings.

■ **Example 3.6** See Figure 3.7. The user has typed `cat "a space"`. The `quote()` function marks with `QUOTE` all the characters between the two double quotes (the quoted characters are shown in grey in the figure). Because of the quotes, the `getwords()` function does not recognise the space between “a” and “space”, and therefore treats `"a space"` as a single word. The other functions don’t do anything in this case, until we get to the `unquoteall()` function, which removes all `QUOTE` markings and the two double quotes. Finally, `buildargv()` takes the two resulting words. This way `cat` gets `a space` in `argv[1]`. ■

The trick is nice and should clarify what quoting means, but note that it only works in a pure ASCII world, since otherwise the character’s most significant bit would not be available (we say that our shell is not “8 bits clean”).

Note that strings like `" "` or `' '` become zero-length arguments because they are first recognised as non-empty words by `getwords()` and later converted to empty words by `unquoteall()`. This behaviour is mandated by POSIX, and is the reason why we let `quote()` preserve the quote metacharacters and only remove them in `unquoteall()` after `getwords()` has already identified all the words. However, this creates a problem with another POSIX requirement: quote metacharacters that were not in the original string should be preserved. For example, if the value of variable `X` is `"` (a double quote) and the user types `echo $X`, then `echo` should receive the string `"` in `argv[1]`. To meet this requirement, we introduce a new `oputexp()` function that we use to add new strings to our buffer. The `oputexp()` function marks all the new quote metacharacters with `QUOTE`, so that `unquoteall()` does not remove them from the string.

Exercise 3.13 Explain the differences (if any) among these commands:

```
$ echo "Hello World"
$ echo Hello World
$ echo "Hello   World"
$ echo Hello   World
```

Exercise 3.14 Now that we have quoting, can we solve the problem in Exercise 3.7 by typing, e.g., `cat '-'`? Explain.

Many groups at Bell Labs tried to extend the Thompson shell before Ritchie and Bourne decided that UNIX needed an official new one that could subsume them all [see 28, min. 12:25]. In particular, the PWB shell by John Mashey allowed for expressions like `$v` to be expanded within double quotes [42]. The expansion is disabled if the dollar is preceded by backslash. Note that the characters resulting from the expansion are still within the quotes, and therefore should be quoted. This feature was added to the Bourne shell, and is now mandated by POSIX.

Exercise 3.15 Implement this feature and compare it with the solution in `8-env.2`.

What if we hit enter before closing a single- or double-quotes string? A real shell will wait for a *continuation line*, printing its *secondary prompt* (“>” by default in the Bourne shell) if it is interactive. The same happens if we hit enter immediately after a backslash, either within double quotes or not, but with a difference: the backslash and the newline are removed from the input buffer.

Exercise 3.16 Implement these features and compare your solution with the one in `8-env.3`.

A real shell will have many other features. Some are easy to implement, while others are much more complex. The `9-fields*` shells implement “field splitting” and will be used in Chapter 5. Some other features are implemented in the `src/sh?.c` files in `myunix`. We will introduce some of them when the need arises.



4. Is Unix secure?

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

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

“On the Security of UNIX” [21] 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) [22]. 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.

4.1 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
mkdir x
chdir x
goto loop
```

(a) v6 script

```
while : ; do
    mkdir x
    cd x
done
```

(b) v7 script

These four-line scripts attempt 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 used `seek()` on the standard input file pointer, looking for a line starting with a colon and the string *label*. This caused the shell to go back and read the command that followed the "`: label`" statement. Since file pointers are shared between the shell and its children, Thompson could implement `goto` as an external command. The colon itself was another external command that did nothing and returned 0 (success). Bourne removed the `goto` command, but kept the colon (as a built-in command). In the Bourne and POSIX shells, the colon 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 `mynix` shell implements `goto` (as a built-in) to support some form of flow control without having to parse the much more complex `for` and `while` constructs of the Bourne shell.

According to Ritchie,

[e]ither a panic will occur because all the i-nodes on the device are used up or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

If we try the script on a modern Linux, it will (slowly) use all available inodes until `mkdir` fails with "no space left on device", but there will be no panic (unlike research UNIX, Linux tries its best not to panic). However, it is still true that any unprivileged user can do this and prevent anyone from creating new files (unless other files are removed first).

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 [...]

Since the shell doesn't call `wait()` when we terminate a command with `&` (see Section ??), the kernel cannot free the resources of all these processes. In this case too, v6 UNIX would either panic or become unusable. UNIX v7 mitigated the problem 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. A modern Linux system may not crash in such a scenario, but it can certainly become unusable.

4.2 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.

It's unclear how the super user got her "root" nickname. One hypothesis is that the name refers to the super user's home directory. This directory was originally the root of the file system and was changed to `/root` much later.

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.

Grampp & Morris [see 13, page 1655] expand on point 3: if a user can delete a file, she can also replace it with something else, "which for most purposes is equivalent to being able to modify [it]".

All of the above problems 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` (see Exercise 3.8) 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.

4.3 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 (see the remark at the end of Section 2.5).

According to Wilkes [see 8, pp. 147–148], R. M. Needham introduced encrypted passwords in Cambridge, likely in the mid 1960s. In contrast, CTSS stored passwords in plain text, in a file that could only be accessed by administrators, in a manner similar to that of early versions of Unix. At least a couple of incidents involved the theft of CTSS passwords [9, 26], and the second one is amusing. One day in 1966, two administrators at MIT were editing the message-of-the-day file and the password file, unaware of each other's actions. Since the editor program always used the same name for temporary files, the two files were accidentally mixed up. Consequently, until the system was stopped, all users logging into the system were greeted with everyone's passwords in clear text. Multics tried to obfuscate stored passwords [51], but this was a weak defence. Unix v6, on the other hand, improved on Needham's idea and provided its first widely available implementation [17].

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. Linux can also be configured to provide a trusted path, using the Secure Attention Key (SAK) mechanism.

4.4 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.)

R We have used the v5 version of mail because there is some inconsistency in Ritchie's description. The v6 mail command, at least as described in the released manual, is not vulnerable to this particular attack. See below for more details.

For the second example, we need to read the v6 mail(1) man-page carefully:

A person is either a user name recognized by login (I), 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 .mail in that directory is used.

So, Ritchie is suggesting to operate as follows:

```
% ln /etc/passwd /tmp/.mail
% 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.

The attack has many of the characteristics we will see throughout the rest of the book: ingenuity (which is what makes these attacks fun), in depth knowledge of obscure features ("mail to a directory"), using things for what they are rather than what they mean (both mail bodies and password entries are just text).

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 addresses the privacy problem. To see why, consider that now you can no longer write:

```
% mail letter person1 person2
```

Instead, you need to use input redirection (see Section 3.5):

```
% mail <letter person1 person2
```

Now it is your shell that has to open the file, *before* the set-UID flag of `mail` has had any effect.

- R** The v6 `mail` command already behaves in this way, and this is why we have used v5 `mail` in the example above. On the other hand, it is v6 `mail` that uses `.mail` for the mailbox: earlier versions used `mailbox`, and v7 moved to `/var/spool/mail/user`. Ritchie either mixed things up, or was referring to some interim version of `mail` that was never officially released.

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 previous use, was a bit limiting, so the decision to drop the *letter* argument in v6 may also have been motivated by usability considerations.

Both of these points solve the security problems nicely, because they work by simplifying the program and its interface, and by removing corner cases of dubious utility. Adding stuff is always more dangerous, because it can introduce new bugs and opportunities for exploitation. In fact, v7 UNIX actually *exacerbated* the problem of set-UID/set-GID programs with the introduction of environment variables, as we will see. The v7 version of Ritchie's paper doesn't yet show any awareness of this fact.

4.5 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 these privileged programs 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.



5. Exploiting the environment

Other subtle features of the modern *su*: the dot is excluded from the shell search path and the burglars' favorite shell variable `IFS` is reset.

M. D. McIlroy, *A Research UNIX Reader*, 1987

Now let us play the role of an attacker. We assume that we have a normal account on a Unix system, and we want to *escalate* our privileges—possibly “become” root. By this we mean that we want to be able to run programs of our choice in processes with an effective uid of 0. Our ultimate goal is to get a “root prompt”, the “# ” prompt that the Bourne shell prints out when running as root.

The `login` program will lookup our uid in the `/etc/passwd` file and then call `setuid(uid)` when we log into the system (Section 2.2, line 39). This uid will be inherited by our shell. Since our uid is greater than zero, we have lost the ability to call `setuid(uid)` again. Assuming that we cannot tamper with the system up to this point, and that the kernel is working correctly, we now have only two ways to run programs with an uid other than our own:

1. somehow make processes that have inherited a different uid run what we want;
2. somehow make available set-uid programs run what we want.

Both of these methods require that the legitimate owners and/or creators of such processes (case 1) or programs (case 2) make mistakes. In a properly configured system, we shouldn't be able to do any harm: processes owned by other users will run programs that we can't control, and set-uid programs will do what they are supposed to do without any interference from us. Unfortunately (for the victims, but fortunately for the attackers), mistakes are very easy to make, especially in a highly configurable system that was developed in an environment very different from our own: a small circle of people where everyone knows everyone else.

5.1 Exploiting `PATH`

Suppose we want user *u* to run a program of our own choice, let us call it *p*. When user *u* types a command like *c* at her shell prompt, the shell will search a set of directories, depending on the contents of *u*'s `PATH` variable, for a file called *c* (Section 3.2). If we can control either *u*'s `PATH` variable, or the contents of a directory that comes before the one containing *c* in the search list, we can make *u*

involuntarily execute our own *p* when she wants to execute *c*: just rename *p* to *c* and copy it into the controlled directory. The victim's shell will find our fake *c* before the legitimate one and execute it, using *u*'s credentials of course.

For this *attack vector* to work, however, we need *u* to make some mistake. We normally have no way of influencing the contents of *u*'s `PATH` variable, whose value is set in the chain of processes that leads from `init` to her shell, which we assume is out of our reach. Moreover, if *u*'s `PATH` variable only mentions directories that we cannot write to, such as `/bin`, `/usr/bin`, and so on, our attack options are zero.

However, if we boot our PDP-11, log in as root in our freshly installed Unix v7 system, and type “`echo $PATH`”, our Teletype ASR 33 will print out the following:

```
:/bin:/usr/bin
```

By default, any user—including root—has an empty path in her `PATH` (did you notice the first colon?) What's more, this empty path comes before the other directories. This means that whenever the system administrator types a command at her prompt, her shell will look for a matching executable file in her current directory before looking anywhere else.

The old `PATH=/bin:/usr/bin` default was a deliberate choice. In the earliest Unix implementations, Thompson's shell searched for executables first in the current directory, and then in the `/bin` directory: the idea was that users were expected to be programmers, and would want to run their own programs, created in their own directory; `/bin` was searched *after* the user's directory, to avoid accidentally hiding user programs with homonymous system programs. In v3, the shell also tried `/usr/bin` after `/bin`: this new directory was introduced when the available space in `/bin` was exhausted (in fact, the v3 manual calls the `/usr/bin` files “overflow” programs [see 2, entry for `sh`])—v4 hides this bit of history and tries to repurpose `/usr/bin` as a place for “lesser used” programs [see 1, entry for `sh`], presumably because it was searched for last). When environment variables were introduced in v7, the Bourne shell generalized the search for executables with the `PATH` variable, following the example of the PWB shell [15]; the default value was chosen to reproduce the old behavior.

If root has not changed the default, this is the mistake we need. Now all we have to do is put our *attack payload* (the program that we want root to run) into the directories where we have write access (hour home, or `/tmp`), give it the name of some common utility (`ls`, `find`, `cat` or whatever), and wait for root to `cd` there and execute the payload for us unnoticed.

A possible payload is the following:

```
cp /bin/sh /home/attacker/hello.c~  
chmod u+s /home/attacker/hello.c~
```

We make a copy of the shell with a filename that looks like something else (like the swap file of some editor) and set the set-uid flag on it. Remember that these commands will be run by root. Therefore, `hello.c~` is now a shell that gives root access to anyone who runs it.

5.1.1 Executable scripts

In order to solve the exercises, we need to learn something more about shell scripts. From Section 3.6.1, we know that we can execute a script by invoking the shell and redirecting its standard input. Since v1 (1971), the Thompson shell could also be called as follows:


```
$ sh script arg1 arg2 ...
```

The shell would still redirect its standard input from the `script`, which could now contain strings of the form `$n`, where $0 \leq n \leq 9$. While parsing `script`'s lines, the shell replaces every `$0` with `script`, `$1` with `arg1`, `$2` with `arg2`, and so on.

The Bourne and POSIX shells work similarly, except that they do not redirect their standard input: they simply `open()` the `script` and read from there. The `sh3` and later shells of `myntix` implement this method, including processing of `$n` special variables, and some additional related features. In particular, they also understand the special variables `*` and `@`. Both expand to all arguments (except argument 0), with a subtle difference when expanding within double quotes: `"$*"` always expands to a single quoted string (which might be empty), while `"$@"` expands to a quoted string for each argument or nothing if there are no arguments besides 0.

R The exact details are much more complex than this, and involve field splitting and the `IFS` variable (Section 5.3.2). We will not go into them here.

The same shell also implements the `“-c cmd”` option, which allows the shell to execute the commands contained in the `“cmd”` argument, e.g:

```
$ sh -c 'echo "hello world"'
```

Note that all of the above methods for running scripts require you to first invoke the shell. This means that executing a script is different from executing a binary program, where you can just type the name of the program itself. However, if you have completed Exercises 3.3 and 3.6, then you know that there are a couple of ways to hide this difference:

- if you make the script executable (using the command `“chmod +x”`), the `exec*p(path, args)` functions will first try to `execve()` it and then, as a fallback, will try to run

```
/bin/sh path args
```

i.e., they will try to let the shell interpret the file as a script, passing the arguments to it.

- if the script is executable and its first line is `“#!/bin/sh”`, the kernel itself will essentially do the same, in the `execve()` system call (this is the so-called “shebang” mechanism).

Either way, if the executable script is in the `PATH`, typing its name at the shell prompt will result in the script being interpreted by a new shell.

Exercise 5.1 — badpath. Try the `PATH` attack in the `badpath` challenge. Be careful: if something doesn't work as expected, root may find out what you are up to. ■

The attack is a bit risky, since an `“ls /tmp”` from a safe directory, an `“echo *”` from `/tmp`, an explicit call to `“/bin/ls”`, and so on, will easily make the administrator suspicious, and an `“ls -l /tmp/ls”` will also reveal our name as the owner of the script. However, it can be very effective as a step in a longer privilege escalation chain—we may have stolen the account of another regular user (e.g., by guessing their password) and created the script using that intermediate victim's credentials.

5.2 Mitigation: Default value of `PATH`

Default initialization scripts and programs no longer put the current directory in `PATH`, nor do libraries provide an unsafe `PATH` if the variable is not explicitly set, as they used to do. If users really want to keep the current directory in their `PATH`, they should very carefully `ls` directories like `/tmp` before `cd`-ing into them. Putting the current directory last in `PATH` can also help, but it's not foolproof either:

the attacker can put an `sl` in `/tmp` and wait for a user to mistype. Much better is not to put “.” or empty paths into `PATH` at all, and just use the “./” trick when we want to run a program that lives in the current directory.

Note, however, that the current directory may also enter `PATH` unintentionally. Empty paths can appear in `PATH` as a result of expanding undefined environment variables. Suppose you have installed a subsystem that puts its executables in a non-standard directory (a very common occurrence). You put the path to that directory in a variable, then expand that variable into your `PATH` in some of your shell initialization scripts:

```
mybin=/opt/mysubsys/v0.1/bin
# lots of other stuff
PATH=$mybin:/usr/local/bin:/usr/bin:/bin
```

Some time later you uninstall the subsystem, delete the line that creates `mybin`, but forget to remove `$mybin` from the assignment to `PATH`. Now you have an empty path in your `PATH`.

5.3 Exploiting setuid programs that escape to a shell

[...] designers must carefully consider the consequences of inherited files, signals, the shell's environment, and so on.

F. T. Gramp, R. H. Morris, *UNIX Operating System Security*, 1984

Now let us try to exploit the second possible attack vector: vulnerable set-uid programs. As Ritchie said (Section 4.4), these programs must be written very carefully and, as a rule, they should not trust anything coming from the outside: command line arguments, environment variables, open files, directories writable by untrusted users—the list is unfortunately very long.

Set-uid programs are the favorite targets of attackers with login access to a Unix system (also known as *local* attackers), and we will examine their possible vulnerabilities in several chapters of the book. For now we limit our discussion to programs that call a shell, as we are interested in bugs that manifest at the command level.

5.3.1 Exploiting `PATH` (again)

Suppose a novice programmer writes a set-uid program that uses the `system()` library function, such as:

```
BUG #include <stdlib.h>
int main()
{
    // stuff
    system("grep something somefile");
    // other stuff
}
```

The programmer needed a functionality similar to that provided by the `grep` utility, so she decided to reuse `grep` itself. The problem is not `grep`: it could have been anything. The problem is that the `system(cmd)` works by `fork()`ing a process and making it run “`/bin/sh -c cmd`”. See, for example, the implementation of `system()` in `mynix` (from `lib/system.c`):

```
17 int system(const char *cmd)
18 {
19     int status, w;
20     pid_t pid;
21
22     switch ( (pid = fork()) ) {
23     case 0:
24         execl("/bin/sh", "sh", "-c", cmd, NULL);
25         exit(127);
26         break;
27     case -1:
28         return -1;
29     default:
30         do {
31             w = wait(&status);
32         } while (w != pid && w != -1);
33         if (w == -1)
34             status = -1;
35         break;
36     }
37     return status;
38 }
```

lib/system.c



This is essentially the `system()` function as found in v7. UNIX v7 also ignored `SIGINT` and `SIGQUIT` while waiting, but we have omitted this code for simplicity. POSIX mandates more things: (i) we should also block `SIGCHLD` (a signal added later); (ii) if `cmd` is `NULL`, `system()` should just check that a shell is available; (iii) perhaps more importantly, we should use `waitpid(pid)` to wait only for the child created by `system()`. Our loop at lines 30–32, instead, may throw away the return status of processes created before `system()` was called.

The shell will parse `cmd` according to its usual rules, including using `PATH` to look for `grep`. Now, however, the attacker has a major advantage over the scenario in section 5.1: the shell created by `system()` will inherit the environment of the setuid program; unless the programmer explicitly cleans it up, the setuid program's environment will be the one inherited from the parent process, i.e., *the attacker's shell*.

Exercise 5.2 — bad0suid. Exploit the above idea to obtain a root shell in the *bad0suid* challenge.

Notice how vulnerabilities in set-uid programs are much better, from an attacker's point of view, than vulnerabilities like the one we examined in Section 5.1. In the “dot in `PATH`” vulnerability, there are many things that are not under the control of the attacker, who just has to wait for them to happen by accident: root (or another user) must have put the dot in her `PATH`, she has to `cd` into the directory where the attacker has planted the trap, she has to execute the fake command. Errors in the attack payload can also render the attack ineffective, and the attacker must wait for the entire sequence of events to occur again, which also increases the chances of getting caught. Vulnerable set-uid programs, on the other hand, are an attacker's dream: she can control essentially the entire execution environment, and she can run them at will.

Exercise 5.3 If we replace

```
system("grep something somefile");
```

in Exercise 5.2 with

```
execlp("grep", "grep", "something", "somefile", NULL);
```

(possibly in a child process), we are executing `grep` without going through the shell. Does this solve the problem with `PATH`? ■

5.3.2 Exploiting the `IFS` variable¹

Suppose that the inexperienced programmer tries to patch the above vulnerability in the following way:

```
#include <stdlib.h>
int main()
{
    // stuff
    system("/bin/grep something somefile");
    // other stuff
}
```

Since `/bin/grep` is a path, the shell will not use the `PATH` variable. Also, since the path is absolute and only traverses directories writable only by root, it must lead to the real `grep` utility.

Let's put on our attacker hat again. While thinking of ways to exploit the new program, we type the following into our v7 shell:

```
$ IFS=,
$ ls,-l
```

Perhaps surprisingly, our teletype starts printing the long listing of the current directory. What we have done is to change the value of the `IFS` variable, which contains the characters that the shell uses as field separators (`IFS` stands for Internal Fields Separator). After parsing the command line into words and operators, the shell examines each word for possible expansions (e.g., processing `$variable` expressions) followed by *field-splitting*. The latter processing uses `IFS` to split words into fields, which then become the actual arguments used to execute a command. The default value of `IFS` is `<space><tab><newline>`, but now we have changed it to a comma. This splits `ls,-l` into `ls` and `-l`, resulting in a normal call to the `ls` program with the `-l` option. The 9-fields revision of the elementary shell of Chapter 3 supports `IFS`. The processing is implemented in a new function `fieldsplit()` that is called by `expandall()` on every normal word, after `expandword()` has finished.

■ **Example 5.1** Figure 5.1 shows an example. The user has typed `ls,-l, a, $X`, where variable `X` is set to `b, c`. Assume that `IFS=, .` Independently of the value of `IFS`, the `getwords()` function always identifies the initial set of words using whitespace characters as separators. In this case, it finds two words. The `getredirs()` and `getvars()` functions don't do anything in this case, then `expandall()` expands the value of variable `X`. Now, `expandall()` calls `fieldsplit()` on every

¹The contents of this section are mostly of historical interest, but still very instructive.



Figure 5.1 – Field splitting example—Bourne shell style

`W_NORML` word (both of them in this case). The result is shown in `words'` and `buf'''`. All the characters in `IFS` are treated as whitespace: note in particular that the comma before the `a` is skipped. The final command line contains 5 words. ■

Exercise 5.4 — bad2suid. Abuse `IFS` to get a root shell from the vulnerable setuid program in the *bad2suid* challenge. This challenge uses a shell (*bad2sh*) that uses `IFS` the way the Bourne shell did: all the characters in `IFS` are equivalent to whitespace. You can see the code in the 9-fields revision of the elementary shell. ■

POSIX mandates a different behavior for `IFS`. In particular, point 3 of Section 2.6.5 of the standard [33] says:

[...] The term “`IFS` white space” is used to mean any sequence (zero or more instances) of white-space characters that are in the `IFS` value (for example, if `IFS` contains `<space><comma><tab>`, any sequence of `<space>` and `<tab>` characters is considered `IFS` white space).

- a. `IFS` white space shall be ignored at the beginning and end of the input.
- b. Each occurrence in the input of an `IFS` character that is not `IFS` white space, along with any adjacent `IFS` white space, shall delimit a field [...].
- c. Non-zero-length `IFS` white space shall delimit a field.

The 9-fields.2 revision of the elementary shell implements the necessary changes in the function `fieldsplit()`. With these rules, the attack you used in Exercise 5.4 will not work.

■ **Example 5.2** Figure 5.2 shows the processing of the same example of Figure 5.1, but using the POSIX rules for `IFS` interpretation. The difference, in this case, is in how the comma before the `a` is

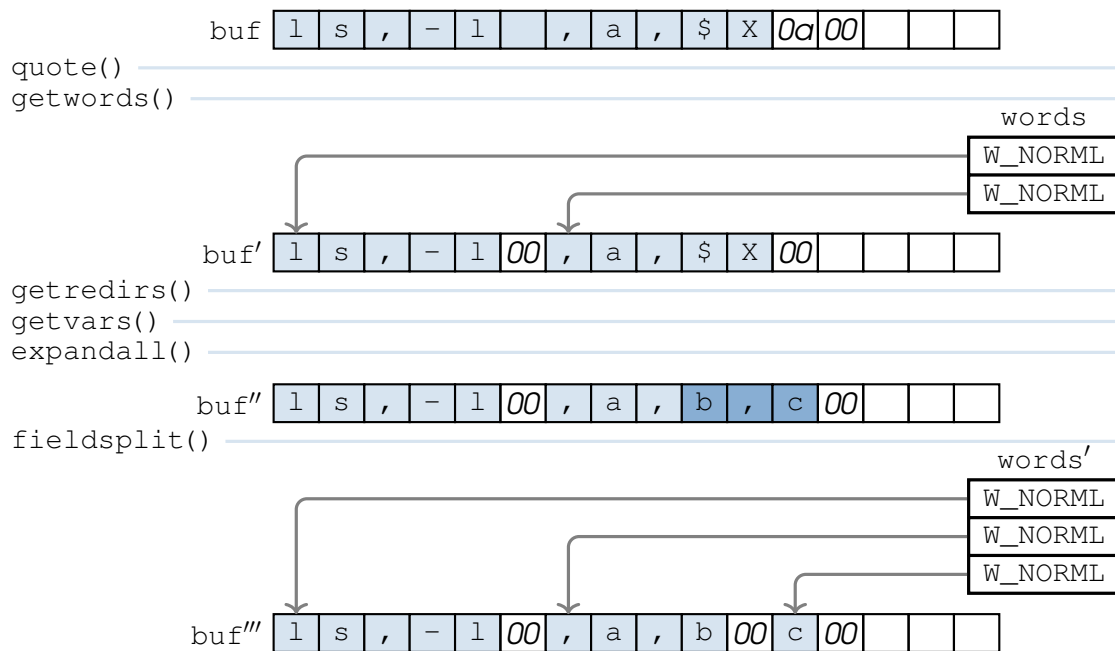


Figure 5.3 – Field splitting example—full POSIX style

`b`, `c` characters resulting from the expansion of variable `X`—all the other commas are not touched at all.

■

5.4 Exploiting shell metacharacters

Great care must be taken when feeding a shell with input obtained from untrusted sources. Consider the following lines from the source code of the buggy `setuid` program in challenge *bad1suid*:

BUG

```
if (snprintf(buf, MAX, "/bin/mkdir /etc/%s", argv[1]) >= MAX) {
    fprintf(stderr, "path too long\n");
    return 1;
}
return system(buf);
```

The program uses `argv[1]` to create a line that will be interpreted by the shell. It doesn't perform any kind of input validation. However, the `bad9suid` binary is installed with execution permissions for everyone, so its command-line arguments must be considered untrusted. A malicious user can pass literally any non-null character in `argv[]`, including all shell metacharacters.

The shell used in the challenge only understands the metacharacters explained in Sections 3.1 to 3.8. Nevertheless, they are sufficient for a successful exploit.

Exercise 5.7 — bad1suid. Obtain a root shell from the `bad1suid` binary.

■

The `bad7suid` challenge is enabled when you solve the one above. In it, the system programmer tries to be a bit smarter:

BUG

```

if (snprintf(buf, MAX, "/bin/mkdir \"/etc/%s\"", argv[1]) >= MAX) {
    fprintf(stderr, "path too long\n");
    return 1;
}
return system(buf);

```

The user’s input is now enclosed in double quotation marks, which prevents the shell from interpreting the metacharacters used to solve *bad1suid*.

R Don’t be confused by the backslashes. They are necessary for placing double quotes within a C string. The resulting string will contain the double quotation marks but not the backslashes. The shell will not receive the backslashes, which is what we want. This is a relatively benign example of the quoting chaos that results when a string is passed through several systems, each with its own quoting rules. In this case, the systems are the C compiler and the shell.

The system programmer still isn’t checking the contents of `argv[1]`, so the binary is still exploitable.

Exercise 5.8 — bad7suid. Obtain a root shell from the `bad7suid` binary. ■

By now, the programmer should realize that she needs to examine the input provided by the user. She should detect bad input and either:

- reject it with an error message;
- transform it into a benign form.

In the case of the shell, the latter strategy would entail removing metacharacters or transforming them into something innocuous, such as an underscore, or quoting them. All of these tactics have drawbacks, so the transformation strategy should be avoided unless it is necessary to solve the problem. Even then, it is much better to find a well-tested library function that performs the transformation rather than trying to implement it ourselves.

Exercise 5.9 — bad8suid. The programmer of the `bad8suid` binary failed to heed this advice. As one might expect, her `escape()` function does not adequately prevent exploitation. ■

Although rejecting bad input is preferable, we must still be careful. There are two approaches to detecting bad input: a *blacklist* approach, where input containing characters on the blacklist is rejected, and a *whitelist* approach, where input is accepted only if all its characters belong to the whitelist (e.g., alphanumeric characters and underscores). The whitelist approach is safer when applicable, since we may forget to include some bad characters in the blacklist.

5.4.1 Command substitution

Before trying the last exercise, let’s add another useful shell feature to our arsenal: command substitution using the `$()` syntax. This is another type of expansion that occurs at the same time as variable expansion. The shell lets a subshell interpret the command inside the parentheses and captures its output. Then, it places the output in the command line in place of the “`$ (. . .)`” string. For example,

```
$ vi $(grep -l TODO *.txt)
```

will open the `vi` editor on all the `.txt` files containing the `TODO` string.

Command substitution was introduced in the Bourne Shell in v7, but it used backquotes instead of `$()`. We don't implement backquotes: Steve Bourne himself admits that `$()` is a much better syntax [see 28, min. 32:00].

This is implemented in `sh8` in `mylinux`. Note that, like variable expansion (see Exercise 3.15), command substitution works also within double quotes.

Exercise 5.10 — `bad9suid`. The programmer of the `bad9suid` program has used the blacklist approach, but her shell understands some dangerous metacharacters that she missed. ■

5.5 Prevention: don't use `system()`

Set-uid programs should *never* call `system()`. Besides the problems outlined above, we should consider that shells, especially the modern ones, are large and complex programs with possibly many little-understood quirks and unexpected behavior. Consider `bash`, for example. Its manual runs for more than 100 pages, and its many features have repeatedly been abused in the past. Here are just a few examples.

1. You can define *shell functions* and inherit them through the environment. Up to version 4.2-208, `bash` function names allowed the “/” character in them; an attacker could therefore easily redirect a `system("/bin/cmd")` by defining a `/bin/cmd` function.
2. Up to version 4.4, `bash` “xtrace” feature would expand the `PS4` environment variable before executing any command; `xtrace` can be set in `SHELLOPTS` and `PS4` can execute the attacker's payload using command substitution.
3. In September 2014 the ShellShock [30] vulnerability was disclosed; it was caused by a bug in how `bash` parsed function definitions stored in environment variables; it allowed attackers to execute arbitrary code and was widely exploited before being fixed.

If an external program is really needed, it is better to use one of the `exec*()` functions without going through a shell (but recall exercise 5.3).

5.6 Prevention: no set-uid scripts

In Section 5.1.1, we examined a few mechanisms that enable scripts to behave similarly to executables, such as the fallback in the `exec*p()` functions and the “shebang” kernel feature. What would happen if we enabled the set-uid and/or set-gid flags in one of these executable scripts? Would the scripts be executed with their UID and/or GID changed?

It should be clear that the first mechanism (the `exec*p()` functions) cannot work. The kernel only looks at the set-uid and set-gid flags during `execve()`, and we are not running `execve()` on the script. Instead, we `execve()` the shell and pass it the script's path as an argument. The kernel has no chance to understand what is going on.

With the shebang mechanism, however, things are different. It is the kernel itself that spawns the shell during the `execve()` of the script. Therefore, the kernel can see the flags and arrange for the desired effect to be achieved—in particular, it can change the UID and/or GID of the process that runs the shell. This was one of the advantages Ritchie mentioned when introducing the mechanism sometime between v7 and v8 [45].

However, we should be convinced by now that set-uid/set-gid scripts are not a good idea. See [29] for many more examples of environment variables that affect the behavior of shell scripts. Beside these, there are actually more problems with the shebang mechanism itself, which we will discuss after

talking about links (Section 6.3). Due to these issues, many Unix and Unix-like systems, including Linux, made the extreme decision to completely disable them [41].

5.7 Mitigation: Privilege drop

Attempting the attacks in this chapter on a modern system will reveal that the resulting set-uid shell does not grant root access. This is true regardless of whether the “set-uidness” is achieved by setting the shell binary itself to set-uid (as in Section 5.1), or by another set-uid program spawning a shell (as in Section 5.3).

Consider, for example, the `PATH` attack in Section 5.1, where we tried to create a set-uid shell disguised as a normal `hello.c~` file. Let’s do this as root in a modern Linux system:

```
# cp /bin/sh hello.c~
# chmod u+s hello.c~
```

We can confirm that the set-uid flag is set on `hello.c~`, but when run by a normal user, the root prompt is not displayed. Running the `id` program reveals that our uid is still the unprivileged one. What’s going on? When they start, many shells (including `bash` and `dash`) call `getuid()` to retrieve the real uid of their process. They then ask the kernel to reset the effective uid back to the real one, using the `seteuid()` system call (see Section 2.7.2). To implement this mitigation in our `mynix` shell, we should add the following code to `main()`, before the program does anything else:

```
seteuid(getuid());
```

The same must be done for the real and effective group ids.

- R** Remember to make sure that the syscall actually succeeded, by checking that its return value is zero. In case of failure, the process would continue to run with elevated privilege.

We will see many mitigations like this in the book. They have all been retrofitted into systems that have been in use for decades and now face the difficult task of striking a balance between legacy behaviour and security. For this reason, they attempt to block a very specific use, while leaving all others untouched. In this case, the shell’s mitigation is intended for set-uid/set-gid programs that call `system()`. Since these programs should never call `system()` (see Section 5.5 above), the fact that a set-uid/set-gid process is executing a shell indicates that something malicious is occurring and should be blocked (most likely, an attack like those described in Part II is underway).

If a set-uid/set-gid program really wants to spawn a shell, it must make its intentions clearer. One such program is `sudo`, which can spawn a root shell with either “`sudo -i`” (login shell) or “`sudo -s`” (non-login shell), provided we are permitted by the `sudoers` file, of course. To do this, `sudo` simply calls `setuid(0)` before the `execve()` of the shell (see `sudo3.c` in `mynix` for an example). When run with an effective uid of zero, this call sets *both* the real and the effective uids, therefore rendering the `setuid(getuid())` statement in the shell ineffective.

Note that, if `sudo` can do something, then any other program running with sufficient privilege can do the same (no program is magic, remember?). For example, we can make the `PATH` attack described in Section 5.1 work on modern systems by operating as follows. First, we compile the following program:

```
#include <unistd.h>
int main()
{
```

```
setuid(0);  
execl("/bin/sh", "/bin/sh", NULL);  
}
```

We put the resulting binary somewhere, say in `/home/attacker/mysudo`. Then we use the following payload for the PATH attack:

```
chown root /home/attacker/mysudo  
chmod u+s /home/attacker/mysudo
```

If root is caught in the PATH trap, she will turn our `mysudo` program into a set-uid program that will give us a root prompt.

Since the (e)uid check is so easily defeated, some shells don't even attempt to protect themselves in the non-`system()` case. The `bash` and `dash` shells, for example, will skip the check and give you the root prompt if you use the `-p` option. However, the `system()` use case should still be safe, since it is `system()` that passes the options to the shell in that case, and the attacker should not be able to control them.



6. Exploiting links

Try this:

Make hard link of `/etc/passwd` to `/var/tmp/dead.letter`
Telnet to port 25, send mail from some bad email
address to some unreachable host.

Watch your message get appended to `passwd`.

C0WZ1LL4@NETSPACE.ORG, *Sendmail* 8.8.[34]
dead.letter exploit, 1997

Links can alter the meaning of file system paths in surprising ways. Programmers must be very careful when using paths whose components are not entirely trusted. However, this is easier said than done. Bugs in this area are very common. A search for the keyword “symlink” on <https://www.cve.org/> returns more than 1590 entries, starting in 1999 and ending in 2024. In this chapter we will examine some of the ways attackers can exploit these bugs.

6.1 Opening unexpected files

As discussed in Section 2.6, a user who wants to create a link to the file `/a/b/c` needs search permissions for the directories `/`, `a`, and `b`. However, she needs no permission for the file `c` itself. This is due to the way links are implemented. To create a link to `c`, you only need its inode number. This number can be obtained by searching for “`c`” in `b`—you don’t need to access `c` itself (or do you? We will reconsider this point in Section 6.6.1.2).

Creating a new link to a file does not *directly* grants you any new permissions on the target file. The owner, group and permissions of the linked file are stored in its inode, and these have not been changed. However, you may now be able to access the file *indirectly*. Programs that want to create a file usually pass the `O_CREAT` flag to `open()`. This flag instructs `open()` to create the file if it does not exist. If the file already exists, no error is returned and the existing file is opened. This is particularly the behavior of the `fopen()` function in the stdio C library, when using the “`w`”, “`w+`”, “`a`” or “`a+`” open modes (see, for example, `lib/fopen.c` in `mylinux`). This creates an attack vector when privileged programs create files in directories that are writable by less privileged users. A typical example is temporary files created in the `/tmp` directory, which is writable by everyone. If the privileged program creates temporary files with predictable names, without checking that they don’t

already exist, it may unintentionally open a link created by an attacker. Consequently, the privileged program will operate on the target of the link, which could be anything the attacker wants.

Exercise 6.1 — badlog. Challenge *badlog* contains a set-uid-root program that writes into a `log` file in the current directory. Try to exploit it to obtain root access. ■

6.2 Exploting races with `access()`

As we discussed in Section 4.4, `v5 mail` sends the file received as its first argument to the user specified as its second argument:

```
$ mail file user
```

The program is installed as set-uid root so that it can create a `mailbox` file in any user directory, even if the user has removed write permissions. However, this also enables `mail` to *read* any file. Therefore, any user can send themselves the private files of any other user (including root). In Section 2.7.2, we briefly introduced the `access()` system call, and suggested that it was probably intended for programs like `v5 mail`. The idea is that the `mail` program should do something like this:

BUG

```
if (access(argv[1], R_OK) < 0) {
    perror(argv[1]);
    exit(1);
}
if ( (src = open(argv[1], O_RDONLY)) < 0 ) {
    perror(argv[1]);
    exit(1);
}
```

Since `access()` uses the *real* uid and gid, the above code asks: “Can the user read this file?”. If the answer is negative, `mail` stops with an error; otherwise, it can continue and open the file.

Despite its good intentions, this system call is vulnerable to a Time-Of-Check-to-Time-Of-Use attack. The `access()` and `open()` system calls follow the same path twice, but there is no guarantee that they will reach the same file both times: an attacker can remove and create links to redirect the final part of the path, in the time between the calls to `access()` (Time-Of-Check) and the call to `open()` (Time-Of-Use). While the timing is not easy to get right, the attacker can retry as many times as they want (maybe in a script), until they succeed.

In the *badmail* challenge you will find a simplified version of the `mail` command that uses code similar to that shown above. In order to complete the exercise, it is necessary to understand another feature of our shell: background processes.

6.2.1 Background processes

The shell is capable of creating “background processes”, i.e., processes that run in the background while the shell accepts new commands. These are easily implemented by skipping the `wait()` if the user has terminated her command with a “&” (the implementation is in `src/sh2.c` in `mylinux`). For example,

```
$ yes 1 >/dev/null &
```


will spawn a process that will execute “yes 1”, and return to the shell prompt. We can check that the process is still running with `ps`. The “!” special variable contains the pid of the last background process. The `&` actually works as a command separator, e.g.:

```
$ yes 1 >/dev/null & yes 2 >/dev/null
```

will create two processes, one in the background (“yes 1”) and the other in the foreground. The `sh2.c` and later versions of the shell also implement the “;” separator, which can be used to run several commands in foreground, one after the other, on the same command line.

Exercise 6.2 — badmail. Read `/root/secret` in challenge *badmail*. Hint: it may be useful to look up the `nice` command. ■

6.2.2 Prevention: temporary privilege drop

There is no straightforward solution to the issue with `access()` and the system call has been deprecated. As the manual suggests, a much better approach for modern systems is to *temporarily* drop the credentials when opening the file:

FIX

```
uid_t euid = geteuid();
if (seteuid(getuid()) < 0) { exit(1); }
if ( (src = open(argv[1], O_RDONLY)) < 0 ) {
    perror(argv[1]);
    exit(1);
}
if (seteuid(euid) < 0) { exit(1); }
```

This is possible because modern kernels (including Linux) remember a *saved* uid alongside the real and the effective ones. When `execve()` changes the effective uid, it also sets the saved uid to the same value. After dropping the credentials, a process can use the `seteuid()` system call to restore the effective uid to the saved uid. Similar considerations can be made for the gids, as usual.

6.3 Exploiting set-uid shebang scripts

The “shebang” method of executing scripts (see Section 5.1.1) can hide many pitfalls when used in combination with the set-uid/set-gid flags.

6.3.1 Passing unexpected arguments

Let’s review the mechanism more carefully. Imagine a process calls

```
execve(path1, argv, env);
```

where

$$argv = [arg_0, arg_1, \dots, arg_n].$$

If the first line of the file reachable from `path1` is “#!`path2`”, the file is considered a *script* that must be passed to the *interpreter* reachable from `path2`. Therefore, the (Linux) kernel replaces the original call with

```
execve(path2, argv', env);
```

where

$$argv' = [path_2, path_1, arg_1, \dots, arg_n].$$

Note that the path of the script ($path_1$) is passed to the interpreter as the first argument ($argv[1]$), in exactly the same form that was passed to the original `execve()`. Using links enables an attacker to have complete control of the $path_1$ string, meaning they can choose the first interpreter argument at will.

As discussed in Section 5.6, Linux has completely disabled the support for set-uid shebang scripts. However, to demonstrate how easily this feature can be abused (on system where it is still available), we have patched the kernel to re-enable support for it. The *badscript1* challenge uses the patched kernel, and contains a set-uid root shebang script in `/bin/badscript1`. In order to solve the challenge, you may find it useful to take a look at the options supported by POSIX shells [see 47, OPTIONS section].

Exercise 6.3 — badscript1. Read `/root/secret` in challenge *badscript1*. ■

In systems where set-uid shebang scripts are supported, we can use the fact that the kernel accepts an optional argument on the “#!” line, separated by spaces from the interpreter path. This argument is passed to the interpreter before the script’s path. The programmer can then start the script as follows:

```
#!/bin/sh -
```

Note the dash: this marks the end of the option list for a POSIX shell. Any other arguments passed after the dash will be interpreted as the name of the script, followed by the arguments for the script itself, not the shell.

R It is useful to recall at this point that the shebang mechanism is not limited to shell scripts; the interpreter can be anything, and the way to prevent attackers from passing unwanted options must be found on an interpreter-by-interpreter basis.

6.3.2 Racing with the interpreter

The shebang mechanism contains a race. First, the kernel follows the script’s path during the initial `execve()`. Then, the interpreter follows the path again when it tries to `open()` the script. As in Section 6.2, an attacker can use links to redirect the path between these two accesses. This can be exploited in a similar way to challenge *badmail*, but it might be helpful to look up the `nice` utility.

Exercise 6.4 — badscript2. The script in *badscript2* uses the “#!/bin/sh -” trick to prevent the exploit used in *badscript1*. However, it is still exploitable. ■

There are ways to prevent this race condition. One option is to use the special files in the `/dev/fd` directory. For instance, when a process `open(s /dev/fd/4`, it reopens (duplicates) its own file descriptor number 4. This can be used in the shebang mechanism as follows: when the kernel opens the script in the initial `execve()`, it creates a file descriptor for the script in the process’s file descriptor table. Assuming the file descriptor is n , the kernel passes to the interpreter the special path `/dev/fd/n` instead of the script’s path. This ensures that the interpreter opens the exact same script that the kernel has already opened. Since it is impossible to change what an open file descriptor points to, this method works.

As we have noted, Linux prefers to disable set-uid scripts completely. However, the above `/dev/fd` mechanism is used in the “`binfmt_misc`” feature [40]. It is a Linux-specific alternative way to implement executable interpreted files that fully supports set-uid and set-gid flags.

6.4 Symbolic links

Symbolic links, or “soft” links, were introduced in the BSD system to overcome some of the limitations of classic Unix links, now renamed “hard” links to distinguish them from the new ones:

- hard links cannot point to directories
- hard links cannot point to a file on another device.

We know about the first limitation from Section 2.6. The second limitation comes from how hard links are implemented. Essentially, a link is just an entry in a directory, and therefore it is only a mapping between a name and an inode number, but inode numbers are only meaningful within a device.

Symbolic links (*symlinks* from now on) do not have these limitations. They are created by their own dedicated system call, `symlink()`. The user command `ln` uses either `link()` or `symlink()`, depending on the presence of the `-s` option in its arguments. For example, these are the sources of the `ln` command in `mylinux`:

```

10 int main(int argc, char* argv[])
11 {
12     if (argc < 3)
13         return 1;
14
15     if (strcmp(argv[1], "-s") == 0) {
16         if (argc < 4)
17             return 1;
18         if (symlink(argv[2], argv[3]) < 0) {
19             perror(argv[0]);
20             exit(1);
21         }
22     } else {
23         if (link(argv[1], argv[2]) < 0) {
24             perror(argv[0]);
25             exit(1);
26         }
27     }
28     return 0;
29 }
```

ln2.c

A symlink is not just an entry in a directory. Instead, it is a special kind of file with its own inode. The *contents* of the symlink are a filesystem path, pointing to a file or directory (or anything else) which is called the *target* of the symlink. This path can be obtained with the `readlink()` system call. This is used, for example, by “`ls -l`” to print “`-> target`” on the right of each symlink (see the `ls7.c` file in `mylinux`).

■ **Example 6.1** Either in `mylinux`, or in your own Linux distribution, create a file, an hard link to it, and

finally a symlink:

```
$ echo hello > new-file
$ ln new-file hardlink
$ ln -s new-file symlink
```

If you `cat` any of the files, you get the same “hello” output. Now look at the output of “`ls -li`”. You should notice several things.

- The `new-file` and `hardlink` entries have the same inode number and, accordingly, share all the other informations except the name.
- The `symlink` entry has a different inode, with type “l”.
- The size of `new-file` and `hardlink` is 5 (the characters in “hello”), but the size of `symlink` is 8 (the characters in “new-file”).
- The number of links of `new-file` and `hardlink` is 2, the `symlink` is not counted.

■

The permissions of symlinks are apparently all set, but they are actually ignored: when you open the symlink, the kernel will use the permissions of the target.

■ **Example 6.2** Continuing from the example above, remove the read permissions from the target and try to read the symlink:

```
$ chmod -r new-file
$ ls -l
$ cat symlink
```

Note, in the output of “`ls -l`”, that both `new-file` and `hardlink` have lost the read permissions (no surprise: they are the same file), but `symlink` has not. However, the last command will give you a permission denied error, since the only permissions that count are the ones of the symlink’s target. ■

There is no “hard” relationship between the symlink and its target, which is in no way affected by the creation of the symlink. In particular, symlinks to the same target are not reference-counted, and the target of a symlink does not even have to exist.

■ **Example 6.3** Continuing the above example, remove the symlink:

```
$ rm symlink
$ ls -l
```

You should see that nothing has changed in `new-file` (and in `hardlink`, which is the same file). Now create the symlink again and remove the hard links:

```
$ ln -s new-file symlink
$ rm new-file hardlink
$ ls -l
```

Nothing has changed in the symlink, but now it points to a non existing file. If you try to read it, you get an error:

```
$ cat symlink
```

The error should be “`symlink: no such file or directory`”, which may be confusing at first, since `symlink` exists and its only its target that does not exist. But `cat` is not aware of this: it has tried to open `symlink`, and its the kernel that has followed the link.

Now create a new `new-file`:

```
$ echo bye > new-file
$ cat symlink
```

The new `new-file` has no relation with the old `new-file`, and yet `symlink` now points to it. ■

To reason correctly about symlinks, you should think that a symlink just contains an uninterpreted string, which the kernel will interpret as a path only when the symlink is *followed*. Most of the system calls that take a path as an argument (e.g., `open()`, `execve()`, `chdir()` and so on) will “follow” a symlink when parsing the path. That is, if the last component of the path turns out to be a symlink, they will follow the path to the symlink target and then use that instead of the symlink. Exceptions are `link()` and `unlink()`, which do not follow symlinks in the last component of their paths. This is especially important in the case of `unlink()`, which will then remove the *symlink itself* instead of the symlink target.

■ **Example 6.4** We have seen this at work in the examples above: `cat`, which uses `open()`, always followed the symlink, but `rm`, which uses `unlink()`, only removed the symlink itself when we issued “`rm symlink`”. ■

The `stat()` system call follows the symlinks. There is a variant, called `lstat()`, that can be used to read the inode of a symlink instead of that of its target. The `ls7.c` version of `myaix ls` uses `lstat()` to obtain informations for the output of `-l`, since otherwise it would not be able to distinguish symlinks from other types of files.

Note that, since symlinks can also point to directories, they can be found also in the intermediate components of a path. All the systems calls will always follow these intermediate symlinks. This can be repeated recursively if they find more symlinks in the target path, up to a maximum “nesting level” which is a system constant.

6.5 Exploiting symlinks

Symlinks introduce a few new twists to the topic of links. Let’s discuss some examples of how these can be abused.

6.5.1 Bypassing directory restrictions

To create a symlink to a file, we don’t need any permission on any component of the file’s path. Compare this with hard links, where at least we need search permissions on all the directory components. Again, this is a consequence of how symlinks operate: the target path is not interpreted when the link is created, but only when the link is followed.

Consider, for example, Exercise 6.2 above. In the exploit, we need to create a link to `/root/secret`. In the proposed solution we used an hard link, because we hadn’t introduced symlinks yet. We were able to create the hard link only because the `/root` directory had search permissions for everyone. This was a simplification introduced for the exercise, since a realistic `/root` directory would be installed with `rwxr-x---` permissions, i.e., no permissions for others. This would be enough to prevent our exploit. However, with the introduction of symlinks, making directories inaccessible is not sufficient anymore. We just need to replace “`ln`” with “`ln -s`” and the exploit works again.

6.5.2 Attacker controlled file creation

The behavior of `open()` with `O_CREAT` on symlinks with non-existent targets may be surprising. The system call transforms the path, by following the symlink, before doing anything else. Once it has the transformed path, now pointing to the target, it checks whether the file exists and creates it if necessary. So the system call will create the missing target!

■ **Example 6.5** Either in mynux, or in your Linux distribution, type the following:

```
$ ln -s non-existent symlink
$ touch symlink
```

Check that you have created the `non-existent` file. ■

Let's consider CVE-1999-1187 [36] as an example of how this can be abused. The bug is in a version of PINE, an old mail reader.

PINE stands for Pine Is Not Elm, a play on an older mailreader called ELM, for ELectronic Mail. This is both a recursive acronym and a wordplay on the literal meaning of the two names.

The program creates a temporary file in `/tmp` with a random name, but always the same for the same user. It also creates the file with write permissions for everyone. The idea is to trick PINE into creating a strategic file that belongs to a victim user and is writable by the attacker.

The attacker can learn the temporary file names of the other users (by looking at `/tmp`), create a symlink with the same name while the victim is not using PINE, and wait. When the victim opens PINE again, the program will open the attacker's symlink with the victim's credentials.

In the proof of concept shown in the reference above, the attacker creates a symlink to a non-existent `.rhosts` file in the victim's home directory. This file was used in the old (pre `ssh`) system for remote access between networked Unix machines. It essentially played the role of the modern "authorized_keys", but without the keys: hosts listed in `.rhosts` are allowed access without passwords. When the victim opens PINE, a world-writable `.rhosts` file is created in the victim's home directory. The attacker can later write whatever she wants to the file and gain access to the victim's account.

Exercise 6.5 — bad4suid. Solve the *bad4suid* challenge. The `rsh` command will connect to a local `rshd` daemon. The daemon will ask for a login name, then check for an `.rhosts` file in the home directory of the user; if the `.rhosts` file doesn't exist or doesn't grant you passwordless access, the daemon will ask for a password. Also note that a `+` in the `.rhosts` file is a wildcard that grants access from any host. ■

6.6 Prevention: secure temporary file creation

Files created in directories writable by others should not have constant names, but this is not sufficient: names should also be unpredictable.

Exercise 6.6 — bad5suid. The set-uid binary in the *bad5suid* challenge doesn't use a constant name, but it is still easily exploitable. You can use `rsh` as in Ex. 6.5. ■

Moreover, programs should always check that the files they intend to create do not already exist. However, code like this is buggy:

BUG

```

if (stat(path, &sb) >= 0) {
    /* path exists: error */
} else {
    fd = open(path, O_CREAT|O_WRONLY, perms);
}

```

The problem is that `stat()` follows symlinks. Therefore, if an attacker has created a symlink to a non-existing file (as in the PINE attack above), `stat()` will fail and the above code will still call `open()`. The following code correctly uses `lstat()`, but is still not sufficient:

BUG

```

if (lstat(path, &sb) >= 0) {
    /* path exists: error */
} else {
    fd = open(path, O_CREAT|O_WRONLY, perms);
}

```

The problem with the above code is that there is a *race condition* between the `stat()` system call used to check that the file does not exist, and the subsequent `open()` used to create it. An attacker could create the symlink in the window between the two calls.

Exercise 6.7 — bad6suid. Solve the *bad6suid* challenge. You can use `rsh` as in Ex. 6.5 and 6.6. ■

The correct way to create a file is to use the `O_EXCL` flag in addition to the `O_CREAT` flag. That way, `open()` will atomically check that the file does not exist before creating it, and will return an error if it does. Note that it will also return an error if the path is actually a symlink, even if the target of the symlink does not exist. Thus, this technique closes all known attack vectors.

FIX

```

fd = open(path, O_CREAT|O_WRONLY|O_EXCL, perms);

```

If you are trying to create a temporary file, you should use the `mkstemp()` library function, which uses the above technique and also automatically chooses an hard-to-guess random name.

6.6.1 Linux-specific countermeasures

Linux implements a number of non-standard countermeasures against symlink attacks.

6.6.1.1 Protected symlinks

The first is a kind of “safety net” for buggy programs that create temporary files insecurely. It can be enabled by typing

```

# echo 1 > /proc/sys/fs/protected_symlinks

```

as root (your distribution may have already enabled this for you). It works like this: if the sticky bit is set on a world-writable directory (such as `/tmp`), Linux will only allow a process to follow a symlink if either the process owns the symlink, or the symlink and the directory have the same owner (which is usually root for `/tmp`). This way, even if the attacker has installed a symlink, the victim process will not be able to follow it and will receive an error. This turns a potential privilege escalation attack into a more benign denial of service for that particular process.

6.6.1.2 Protected hardlinks

There is also an option to protect hard links, but it works differently. If `protected_hardlinks` is set to 1, users can only create hard links to files they own or to files to which they have read and write access. One could argue that this is more sensible than the traditional rule, as it is not entirely accurate that creating a hard link does not require accessing the target file. In fact, the target's inode must be accessed to increment the link counter. This increment also has security implications since it can prevent files from being deleted. For example, a superuser trying to remove a vulnerable set-uid program could be thwarted by an attacker who creates a hard link to the program to keep a copy around.

Exercise 6.8 — bad4suid2. Solve the *bad4suid2* challenge. ■

Exercise 6.9 What happens if we set `protected_symlinks` in Exercises 6.5, 6.6, or 6.7? Are the exploits affected by the status of `protected_hardlinks`? ■

Modern kernels also have options to protect FIFOs and regular files in world-writable sticky directories.

6.6.1.3 O_TMPFILE

Another countermeasure Linux implements is the `O_TMPFILE` flag for the `open()` primitive. This flag causes `open()` to create a file with no links in the file system, i.e. it just allocates an inode without giving it a name. The primitive is guaranteed to always allocate a new inode, so symlink attacks are impossible. It also saves the programmer the trouble of inventing a unique name for the temporary file. Finally, since the inode has a link count of 0, the temporary file is automatically removed when all its open file descriptors are closed. In the normal case this happens automatically when the process exits, even if it does not exit cleanly (e.g., because it is killed). If you are writing an application that needs to run only on Linux, you should consider using this flag whenever you create a temporary file.