How a Unix shell works

G. Lettieri

26 September 2023

1 Introduction

Hackers know very well how things really work (the real ones, at least). We need to acquire a similar level of knowledge, and we start with the most important Unix tool: the shell.

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 behaviour 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/myUnix-1.6.3.tar.gz`

The names between the brackets in the section headings refer to the source file of the shell discussed in that section. The shells can be tested either within the myUnix environment, or by compiling and running them on your host machine (Linux or Mac) by going to the `src` directory and typing, for example,

```
make sh0
```

to build `sh0` from `sh0.c`. Then you can run it with `./sh0`.

2 The simplest shell [`sh0.c`]

Figure 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.

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. Therefore, the user will be able to type commands on the terminal where she

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h> /* for perror() */
#include <unistd.h>

#define MAX_LINE 1024

/* minimal shell.
 * shows fork/exec exit/wait
 */

int main()
{
        char buf[MAX_LINE];
        int n;

        while ( (n = read(0, buf, MAX_LINE)) > 1 ) {

                buf[n - 1] = '\0';

                if (fork()) {
                        wait(0);
                } else {
                        execl(buf, buf, NULL);
                        perror(buf);
                        exit(1);
                }
        }
        return 0;
}
```

Figure 1: The simplest shell.

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.

Note that `read()` will also give us the newline character that ends the line typed by our user[1]. We replace it with the null character, to get a C string.

Now let us focus on the function we use to execute the program. We are not calling the `execve()` system call directly. Instead, we use `execl()`, a C library function that is a little easier to use. The function does some additional 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 in `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. Take a look at `lib/execl.c` to see how this works. The function collects these strings, starting with the second argument until it finds `NULL`. It builds an array of pointers to these strings. This array will then become 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, so we pass `buf` twice. Remember, however, that only the first is interpreted by the kernel as a path, while the second is just copied into the process memory and made available via `argv[0]`.

As for the last `execve()` argument, the array pointing to the environment variables, `execl()` will simply pass the one from the calling process. The new process, therefore, will see the same environment variables as the shell (this latter part is actually done by `execv()`, see `lib/execv.c`).

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, 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

[1] Assuming she types less than `MAX_LINE` characters, but we'll ignore that for now.

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.

Exercises

2.1. Note that, while in `sh0`, you can edit the command line 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?

2.2. What happens if you type `/bin/ls` immediately followed by Ctrl+D? Why would this happen?

2.3. Close `sh0` and create a `script` text file containing any shell command, make it executable (`chmod +x script`), restart `sh0` and try to run the script. What happens? Why?

2.4. Do as in Exercise 2.3, but add a line with *`#!/bin/sh`* at the very top of the `script` file. What happens now? Why?

2.5. If you try to use more advanced editing keys than those mentioned in Exercise 2.1, 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 Using `PATH` [`sh1.c`]

Now let's add the `PATH` variable into the picture. All we have to do is replace `execl` with `execlp` in Figure 1.

The `execlp()` 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 error.

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 `execlp()` 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 `execlp()` 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 `execlp()` will not be able to find our program.

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

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 "`./myls`" 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 `myls`, `execlp()` will also try to pass `myls` 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.

Exercises

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

```
utils/exe
```

into `sh1`. Will it work? Does it depend on the contents of `PATH`?

3.2. Do as in Exercise 2.3, but using `sh1` instead of `sh0`. What happens now? Why?

4 Splitting the command line [`lsh.c`]

Now let us use `sh1` 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 2 shows a very simple implementation of the idea.

The new lines 22–30 scan the input buffer, looking for space characters. They then replace the spaces with string terminators (`'\0'`) and collect the starting addresses of the resulting strings in the `c_argv` array. When all the line has been scanned, the array is `NULL` terminated (line 30).

The first array element (`c_argv[0]`) and a pointer to `c_argv` are then passed to `execvp()` at line 35. 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.

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h> /* for perror() */
#include <unistd.h>

#define MAX_LINE 1024
#define MAX_ARGS 10

/* shell that splits lines into words */

int main()
{
        char buf[MAX_LINE];
        int i, n;
        int c_argc;
        char *c_argv[MAX_ARGS + 1];

        while ( (n = read(0, buf, MAX_LINE)) > 1 ) {

                buf[n - 1] = '\0';

                c_argv[0] = buf;
                c_argc = 1;
                for (i = 0; i < n && c_argc < MAX_ARGS; i++) {
                        if (buf[i] == ' ') {
                                buf[i] = '\0';
                                c_argv[c_argc++] = &buf[i + 1];
                        }
                }
                c_argv[c_argc] = NULL;

                if (fork()) {
                        wait(0);
                } else {
                        execvp(buf, c_argv);
                        perror(buf);
                        exit(1);
                }
        }
        return 0;
}
```

Figure 2: A shell that splits the command line into words.

Exercises

4.1. Exit from `lsh` and create a file called "–" (a single dash):

```
echo hello > -
```

Now restart `lsh` and try to run **`cat -`**. What happens? How can you invoke **`cat`** from `lsh` to see the contents of that file?

5 Parsing [`sh2.c`]

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.

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.

Another flaw of the shell in Figure 2 is that the parsing of the input line is far too simple: what happens if the line starts with spaces, or if there is more than one space between two words, or if the user wants to continue an input line with the standard "<backslash><newline>" sequence? None of this works as expected.

We can remedy these shortcomings by implementing a less naïve parser—one that examines input characters one at a time and builds *tokens* from them. Since programs are startting to get too big to include here, we will now refer directly to the files in `myUnix-1.6.3.tar.gz`. The `src/sh2.c` shell provides essentially the same functionality as the one shown in Figure 2 (i.e., splitting lines into words), but it does so using a slightly more sophisticated parser. The parser will also simplify the implementation of the other features that we will consider later.

The input processing is done in three stages:

1. the functions in the `input.h` file take care of reading the input one byte at a time, prompting the user if necessary, and processing the "⟨backslash⟩⟨newline⟩" sequence;

2. the `token()` function recognizes tokens in the resulting string and returns the type of the next token to process;

3. the main program repeatedly calls `token()` and decides what to do with the resulting tokens.

Stage 1 uses a **struct** `input` data structure, which must be initialized with the input file descriptor (0 in our case, for standard input). Then, `nextcmd()` must be called to start the processing of a new command, prompting the user and loading the first input character into the `ibuf` field of the structure. When the processing of `ibuf` is finished, the next character can be loaded by calling `inext()`. The data structure also provides some space to store the token values as they are found (the `obuf`). A new token value is started by calling `newtoken()`, then characters can be added to it by calling `oput()`. The last token value can be retrieved by calling `lasttoken()`.

Stage 2 is implemented in the `token()` function. It first skips any leading whitespace from the current position in the input stream, and then decides what to do based on the current input character. For now we only recognize two types of tokens separated by whitespace: *words* (type 1) and *newline* (type `'\n'`). Word tokens are any sequence of non-whitespace characters; the newline token terminates a command.

In stage 3, the main program accumulates words in the `c_argv` vector and terminates it with a `NULL` pointer when it receives the newline token. It then proceeds to execute the requested command exactly as in Figure 2.

This new shell behaves correctly if its standard input is redirected from a file, and it correctly parses sequences of whitespace characters, long lines and line continuations. It also prints a prompt when the input file is a terminal.

Exercises

5.1. Start `sh2`, type

`l`⟨backslash⟩⟨newline⟩`s`⟨newline⟩

and try to guess what is going to happen before you hit each ⟨newline⟩.

6 Quoting [**sh3.c**]

Now create a file with a space in its name using your normal shell, e.g.:

```
touch "a space"
```

Then start `sh2` an try to run some command on the file 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 normally parsed by the shell and not by the commands, but `sh2` doesn't know how to do it. 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 `sh2` shell, instead, has treated the "`"`" characters as normal characters: it has not used them to protect the space between "`a`" and "`space`" and has not removed them from the command line.

A shell that parses backslashes and single/double quotes can be found in `src/sh3.c`. With respect to `sh2` we have added a few functions and expanded the `word()` function. While examining the characters of a word in the `word()` function, we start special processing whenever we see a "`"`", "`'`" or "`\`". A backslash is removed and the next character is added to the current token value. The quotes (double or single) are removed and all characters are appended to the current token value, until the matching quote is found (see the `collect_sq()` and `collect_dq()` functions). While collecting the characters within double quotes (in `collect_dq()`) we need to take special care whenever we see a backslash, which may or may not quote the next character, depending on what the next character is. The detailed quoting rules can be read, for example, in the man page of the `dash` shell. Don't think of quotes as defining "strings", like in most programming languages: quotes in the shell can be found *in the middle* of words. For example, the sequence `a"b"c` becomes `abc` after quote processing. Think of them as markers that remove the special meaning from some or all of the enclosed characters.

To implement this "removal of special meaning" we steal the idea from the original Unix shell from Ken Thompson: we mark the quoted characters by setting their most significant bit (see the `QUOTE` constant that is or-ed to characters), so that their original code is no longer recognized. Then we remove all markings before passing the final strings to the commands (see the `remove_quotes()` function). Note that this is only possible in a pure ASCII world, since otherwise the character's most significant bit would not be available[2].

With this modifications in place, our new minishell correctly understands sequences like "`"a space"`" and our previous example works. This is because the quotes are removed and the space is added to the token value by the `collect_dq()` function. Note that the `QUOTE` markings don't play any special role yet, but they will become important when our shell will be able to recognized more token types.

Exercises

6.1. Explain the differences (if any) among these commands:

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

6.2. Now that we have quoting, can we solve the problem in Exercise 4.1 by typing, e.g., **`cat '-'`**? Explain.

6.3. Try to guess what happens when you enter

[2] In other words, our shell is not "8 bits clean".

```
echo '⟨backslash⟩⟨newline⟩'
```

and

```
echo "⟨backslash⟩⟨newline⟩"
```

7 Shell builtins [`sh4.c`]

Let us try something else. Assume that the current directory contains a `Movies` subdirectory (create it if it doesn't). Let us start `sh3` and type

```
cd Movies
```

Will it work?

No, `sh3` 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[3].

For **`cd`** to work, the shell must call the `chdir()` system call by itself. A shell that understands **`cd`** can be found in `src/sh4.c`. The only difference with `sh3` is that after the words are collected, the first word is compared to the "**`cd`**" string. If the strings match, `sh4` 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. 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

[3] See page 6 of the paper on Unix evolution on Dennis Ritchie's home page (`https://www.bell-labs.com/usr/dmr/www/`.)

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).

8 I/O redirection [`sh5.c`]

Standard input and output redirection is very easy to implement in Unix. You can take a look at `src/sh5.c` to see how it is done.

As far as the command line parsing is concerned, we now have a few more tokens: one for the input redirection operator (<), one for the output redirection operator (>) and a final one for the append operator (>>). We don't implement all the other redirection operators that most shells understand.

The redirection operators are only recognized if they are not quoted. Also, they add to the set of word delimiters other than spaces. When the main program scans the input tokens and finds one of these new types of tokens, it skips any adjacent whitespace and expects to find a word token, representing a pathname. Depending on the type of the redirection token, it remembers this path in either the `ifile` or the `ofile` pointer. Then, in the *child* process, before calling `execvp()`, if `ifile` was set, the shell closes file descriptor 0 and opens `ifile`. Since `open()` always uses the first unused file descriptor number, it will assign file descriptor 0 to the newly opened file. Something similar happens when `ofile` is set, but in this case the shell will also choose between the `O_TRUNC` or `O_APPEND` open flags, depending on whether the operator was > or >>. Now the program will start with its standard input and/or standard output redirected to these files. Note that the program is completely unaware of this, and this explains why the mechanism works for all programs (that follow the stdin, stdout and stderr convention). All this is achieved without adding any new system call.

Exercises

8.1. 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
```

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

9 Environment variables [`sh6.c`]

Now consider shell's support for environment variables, such as `PATH` and `HOME`.

The purpose of these variables is to personalize the user's environment or to remember values across between 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 containining 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 provided 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 behaviour 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:

- the shell maintains an environment that can be passed to its children, and provides some syntax for updating it;

- 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.

For our shell (`src/sh6.c`) we use the following syntax. To assign a value to an environment variable, use

variable=*value*

No spaces are allowed anywhere (but you can use single or double quotes to enclose spaces in the *value* part). 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 expand the value of a variable, use `$`*variable*, followed by a space or a non-alphanumeric character (including a backslash), anywhere on the command line. The expansion replaces the `$`*variable* sequence with the *value* of *variable* if it exists, and with nothing otherwise.

The implementation is conceptually simple, and is only made complex by the nuances of how POSIX shells work. When a `$`*variable* expression is found on the command line, the shell collects the *variable* name, looks for the variable in the environment, then replaces the `$`*variable* string with the value fo the variable (or with an empty string, if the variable is undefined). The resulting characters are treated differently depending on where the `$` was found: inside double quotes, or inside an unquoted word. In the former case, all the characters of the expansion are quoted, including the spaces; in the latter case, spaces are meaningful and can be used later to split the variable expansion into *fields*. Our implementation is in stages: in the first stage, the input is scanned, copying the (possibly quoted) characters into the `obuf` buffer, as in the previous shell. During this phase, variable expansions are recognized and specially marked in the `obuf`, so that the variable name is delimited and it is possible to tell whether it was seen inside double quotes or not. Then, the tokens are generated as before, but each word token is passed to the `expand()` function, which replaces each variable in the word with its value, quoting the resulting characters as needed (i.e., marking them with `QUOTE`). The word is then passed to `fields_split()`, which splits the word into one or more fields, using any unquoted whitespace as a delimiter[4]. Each resulting field is then passed to `remove_quotes()` to remove the `QUOTE` markers.

Assignments are recognized during the word token handling by looking for strings that start with an identifier immediately followed by an "=" character. The strings before and after the equal sign are then passed to `setenv()`. The new `unset` builtin can be used to clear variables.

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 shelli 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.

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`

[4]What is meant with "whitespace" here actually depends on the contents of the `IFS` variable, as we will see in another lecture.

file. 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`, ...).

Exercises

9.1. Assuming that `X` doesn't exist yet, try to guess the output of

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

10 Other features

A real shell will have many other features. Some are easy to implement, while others are much more complex. You can look at some of them in the other `src/sh?.c` files. For example, it is the shell that prints "Segmentation fault", by looking at the status value returned by `wait()` (`src/sh7.c`). "Background jobs", i.e., processes that run in the background while the shell accepts new commands, are easily implemented by skipping the `wait()` if the user has terminated her command with a "`&`" (`src/sh8.c`). A more general scripting facility is easily obtained by accepting a script name from the command line and then using it as input instead of stdin (`src/shA.c`). The same shell also implements the "`-c cmd`" option, which allows the shell to execute the commands contained in the "`cmd`" argument. It is the shell that expands filename wildcards such as "`*`" and "`?`" (`src/shB.c` and `src/shC.c`). It is also the shell that creates pipelines of commands, using essentially the same trick as in I/O redirection (`src/shD.c`). A few more extensions are collected in the `src/shE.c`, `src/shF.c` and `src/shG.c` shells.

A Solutions to the exercises

Ex. 2.1

Since we have only called `read()`, it must be implemented by something that is active during its execution. But `read()` is a system call, so line editing must be implemented either in the kernel, or in the terminal (emulator) itself.

We cannot be more precise than this without adding a bit of history: old terminals were very simple and did not provide any line-editing functionality, so it had to be implemented in software. For a long chain of compatibility requirements, this is still true today. In fact, this kind of line editing is implemented in the TTY module in the kernel, attached to the driver that talks to the (emulated) teletype device. The TTY module has a fairly complex set of options, which can be inspected and set using the `stty(1)` utility (or, more precisely, using a set of system calls also used by `stty`). You can print the list of the current options with `stty -a`. Many options are devoted to the details

of the serial communication with the tty device and are mostly meaningless today; some options specify the action to be taken when special ASCII values are entered from the keyboard: some values cause a signal to be sent to the terminal's foreground process group, and others are used for line editing. The `erase` action removes the last input character and should be mapped to the "`^?`" ASCII value. This syntax is used to display unprintable, or *control*, ASCII values, i.e., the first 31 values and the last one (0x7f, DEL[5]). The idea is to flip bit 0x40 to get a printable character, then prepend that character with "`^`", which stands for "0x40 XOR ...". Since the ASCII code of "`?`" is 0x3f, "`^?`" is 0x40 XOR 0x3f = 0x7f, i.e., DEL.

Note that, for many keys, typing Ctrl+x will also toggle bit 0x40 of x[6] so you can also enter DEL by typing Ctrl+? instead of the key mapped to DEL in your keyboard (probably backspace). This is also the way you can enter "`^C`" (ASCII ETX, usually mapped to the `SIGINT` signal) and the other values which are not otherwise available on the keyboard. For this reason the "`^`x" syntax is often pronounced "control x".

Ex. 2.2

If you try, you should see

```
/bin/ls/bin/l: No such file or directory
```

The inital `/bin/ls` is just the echo of what you typed. As soon as you entered Ctrl+D, `sh0` woke up and processed your characters. This is because Ctrl+D is actually used as an "end of input" by the TTY module: it is an order to pass the characters accumulated so far to reading process. Ctrl+D works as an end of file only when entered on an *empty* line: the process will receive 0 characters and will interpret this as an EOF condition (look at the **`while`** condition in `sh0.c`). In our case, `sh0` received "`/bin/ls`", i.e., 7 characters with no ending newline, it replaced the last one ("`s`") with the string terminator, and then tried to execute the resulting "`/bin/l`".

Ex. 2.3

You obtain an *Exec format error*: the kernel doesn't recognize your file as something than can be executed. Indeed, it is just a text file that needs an *interpreter*, and not something that can be simply loaded into memory and then executed directly by the CPU.

[5] DEL is 0x7f because ASCII was designed for paper tape: to erase any character, you could punch all seven holes.

[6] This is reminiscent of how the Ctrl key was implemented in the Teletype ASR 33. In the ASR 33, the Ctrl key always resets bit 7 instead of toggling it. The Shift key instead flipped bit 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.

Ex. 2.4

This time the script works: if the first two characters of the file are "*#!*" (sometimes called a *shebang*), then the rest of the line is the path of an interpreter for the file, optionally followed by whitespace and then a single argument for the interpreter. In this case, the kernel itself will actually load the interpreter, passing it the option (if present) and the full path of the original file. In our case, assuming that the `script` file is in the `/home/user1` directory, the kernel will run

```
/bin/sh /home/user1/script
```

The shell will then open and interpret the `script` file.

Note that the new shell will see the first line of the script, which was meant for the kernel. This works because "#" is the start-of-comment character for the shell, so the line will be ignored. Other interpreters (like **awk**, `perl`, `python` and many, many others) also use "#" as a comment character, so this feature automatically works for them too.

Ex. 2.5

When terminals evolved more capabilities, like a bidimensional screen, they introduced *escape sequences* to embed control commands in the stream of ASCII characters exchanged with the computer. These sequences have been then standardized by ANSI[7]. 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.

Now, if we start `sh0` and hit the up-arrow, we see "`^[[A`". This is the echo (sent by the TTY module in the kernel) of the escape sequence sent by the terminal (emulator). By default, the TTY module echoes control characters in the same way explained in the solution of Exercise 2.1: "`^[`" stands for 0x40 XOR 0x5B = 0x1B, which is ESC[8]. Note that the program reading from the terminal will receive the true sequence, not the one which is echoed. You can check this if you run **cat**, hit the up-arrow a few times and then press enter. Then **cat** will wake up and send the received characters back at the terminal and the cursor will move up.

The `libreadline` userspace library, used by `bash` and many other interactive programs, sets the terminal in *raw* mode, disabling echo and line editing in the kernel, then implements all the advanced eding functions (including com-

[7] `https://en.wikipedia.org/wiki/ANSI_escape_code`

[8] It should be clear, by now, that Ctrl+[is an alternative way to type ESC on the keyboard; this is very useful in, e.g., `vi`.

mand history) in userspace. We are not using `libreadline`, and this is why our shells lack these functions.

Ex. 3.1

Yes it works and it does *not* depend on the contents of `PATH`. Follow the rules: is there a slash in the string? *Yes*. So `PATH` is not used and the string is passed *as-is* to `execve()`. The kernel will then interpret it as a relative path, correctly leading to the `exe` file.

Many people seem convinced that the "`./`" prefix is necessary whenever you want to specifiy a relative path (e.g., some people may type something like **`cat`** `./file`). This is not the case: "`./`" is needed only to trick the shell (or similar programs) into not using `PATH`, and you only need it if you *don't already have a slash in your path*. Moreover, relative paths of files passed as arguments to other programs need "`./`" only if the program itself is using some `execlp()`-like logic to parse the paths, which is almost never the case. For example, **`cat`** will simply pass the "`file`" string to `open()` (but see Exercises 4.1 and 6.2).

Ex. 3.2

This time the `script` is executed. This is another feature of `execlp()` and the other `exec*()` functions with a `p` in their name: if the first `execve(path,...)` fails, they try a second time with `execve("/bin/sh",...)` with `path` as an argument. The shebang feature described in the solution of Exercise 2.4 can be seen as a generalization of this behavior, implemented directly by the kernel.

Ex. 4.1

The **`cat`** command, like many others, interprets "−" as standard input, i.e., it will start reading from file descriptor 0 instead of trying to `open("-"...)` (see `src/cat3.c`). The trick is that this behaviour is only triggered by the exact "−" string, so any other equivalent path will work:

```
cat ./-
```

Ex. 5.1

After the first newline the shell prints the secondary prompt (">" by default), waiting for more input. This is because the newline has been quoted and is not recognized as a command terminator, thus the shell is still waiting for the end of the command. The second newline ends the command and the shell tries to execute it. Perhaps surprisingly, the resulting command is `ls`: the ⟨backslash⟩⟨newline⟩ sequence is *removed* from the input, even if it occurs in the middle of a word! (The original Bourne Shell retained the backslash, but the behaviour implemented by `sh2` is mandated by POSIX).

Ex 6.1

The first two commands produce exactly the same output, even if the first one is psycologically nicer. The third one prints all the spaces between `Hello` and `World`, while the last one prints only one space between them. Note that **`echo`** never sees the unquoted spaces: they are parsed by the shell. The first and third **`echo`** instances receive a single argument while the second and third instances receive two arguments (not counting `argv[0]`).

Ex 6.2

No, this cannot solve the problem, since the quotes are only seen by the shell and **`cat`** will still receive the unadorned "`-`" string.

Ex 6.3

The first one prints the backslash, while the second one prints nothing. This is because the backslash is always quoted within single quotes, but it is not quoted within double quotes when followed by newline.

In our implementation, the ⟨backslash⟩⟨newline⟩ sequence is processed at the lowest level, in `input.c`, to simplify the implementation of the behaviour described in the solution of Exercise 5.1. The second argument to `inext()` is 1 when the processing of this sequence must be disabled (because we are collecting characters between single quotes).

Ex 8.1

The redirection is performed by the shell, in the forked process, before executing `sudo`. So, the process will try to `open()` the `f` file before the kernel has had any chance to change the user id, and will therefore fail.

One trick is to use `sudo` to run some command that opens the file by itself and then writes into it, so that the `open()` is performed after the `execve()`. The `tee(1)` command is good enough for this purpose: it copies its standard input to standard output and to all the files passed as arguments:

```
echo something | sudo tee f
```

This has the annoying side effect of writing `something` to standard output too. In a script we might prefer

```
echo something | sudo tee f > /dev/null
```

Ex 9.1

The "`$X`" is expanded by the shell before interpreting the line, and in particular before updating the environment with "`X=aaa`". Therefore, "`$X`" expands to nothing and **`echo`** prints an empty line. If the purpose is to print "`aaa`", the

assignment must be performed in a previous line with an empty command, so that the shell environment itself is updated:

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

In a normal shell this would also work:

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

It doesn't work in `sh6` because the semicolon is not recognized as a command separator. This syntax is recognized starting with `sh8`.