How a Unix shell works

G. Lettieri

26 September 2023

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/esh.zip`

The code has been organized as a git repository, with each commit adding a single new feature. Once you have unziped the file, you can enter the `esh` 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:

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

If you run

```
make revisions
```

```
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 1024

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

        while ( (n = read(0, buf, MAX_LINE)) > 0 ) {
                buf[n - 1] = '\0';

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

Figure 1: The simplest shell.

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

1 The simplest shell (tag: 1-basic)

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 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[2]. 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` in myUnix).

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 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
```

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

[2] You can take a look at `lib/execl.c` in myUnix to see how this works.

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

Exercises

1.1. Note that, while in `1-basic`, 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?

1.2. 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?

1.3. 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?

1.4. Do as in Exercise 1.3, but add a line with `#!/bin/sh` at the very top of the `script` file. What happens now? Why?

1.5. If you try to use more advanced editing keys than those mentioned in Exercise 1.1, such as the arrow keys, you get only strange characters in response. What are these characters? Why don't the keys work as expected?

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 `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[3].

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

[3] Check `lib/execlp.c` in myUnix for the full story.

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

2.1. 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`?

2.2. Do as in Exercise 1.3, but using `2-path` instead of `1-basic`. What happens now? Why?

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 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 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 boot of the `esh.c` file in the repository, and just contain some strings and pointers manipulations[4].

The first element of `c_argv` (`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

[4] 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 5) that some input words are not passed to the command, so it is cleaner to keep the two tasks apart.

```
#include <sys/wait.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 1024
#define MAX_WORDS 10
#define MAX_ARGS  10

typedef struct {
        char *w;
} word_t;

int getwords(char *buf, word_t words[]);
int buildargv(char *argv[], word_t words[], int nwords);

int main()
{
        char buf[MAX_LINE];
        int n;
        word_t words[MAX_WORDS];
        char *c_argv[MAX_ARGS + 1];
        int nwords;

        while ( (n = read(0, buf, MAX_LINE)) > 0 ) {
                buf[n - 1] = ’\0’;

                nwords = getwords(buf, words);
                if (!nwords)
                        continue;
                buildargv(c_argv, words, nwords);

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

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

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 where 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.

Exercises

3.1. Exit from 3-words and create a file called "-" (a single dash):

```
echo hello > -
```

Now restart 3-words and try to run cat -. What happens? How can you invoke cat from 3-words to see the contents of that file?

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. Thompson, D. Ritchie, *Unix 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[5].

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-builtin1̂. 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).

Exercises

4.1. 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.

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

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

on the terminal; then, in the child process created by the shell, before calling `execvp()`, you `close(0)` and `open()` the file: the kernel will pick the first unused file descriptor, which will be the 0 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 stardard 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 assignes 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.

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 third problem with Unix file permissions that Ritchie mentioned in his “On the Security of UNIX” paper (see Section **??**), and can be partially mitigated by setting the `umask` to filter out some permissions from all `open()` system calls.

Exercises

5.1. The > operator clears the file if it already exists (option `O_TRUNC`). The shell in V2 UNIX 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`[6].

5.2. 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?

6 Scripting (tag: 6-intr)

> Buffered IO was, and still is, a necessary evil.
>
> M. D. McIlroy, *A Research UNIX Reader*

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.

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

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

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 6.1). 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 buffering on standard input: this is the purpose of the `setbuf(stdin, NULL)` call in the 6-intr sources.

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[7].

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

Exercises

6.1. 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.

7 Environment variables (tag: 7-env)

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

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.

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)[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

[8] Unlike the analogous limitations in Section 5, these ones apply also to modern shells, but see Section 8.

"assignments", so that buildargv() will skip them (since it only picks "normal" words). Unlike getredir(), it stops at the first non-matching word[9]. 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.

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. 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, ...).

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.

The implementation is conceptually simple, but string manipulation in C is cumbersome and error-prone, so we implement it in two substeps (7-env.2 and 7-env.3). 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. In the first step we simply ignore this problem, and in the second step, we fix it.

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

[9] Bourne initially looked for assignments in all the words of the command line, but this conflicted with some commands (like dd) that use the assignemnt syntax for their own arguments.

(with the help of the functon `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.

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()`.

As unticipated, `expandword()` doesn't check for possible overflows in the `dst` buffer. The `6-env.3` shell fixes this by introducing 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 can be checked at any time: if we find it negative we can signal an error to the user.

Exercises

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

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

8 Quoting (tag: 8-quote)

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 ">" are used for redirection; "`$`", when followed by an identifier, is used for variable expansion; a "=" occurring in the name of a command may be mistaken for 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.3`, 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 character[10], 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[11].

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 metacharaters (interpreted by the shell and then removed from the input), you need a way to quote *them* when needed. You can quote the backslash whit 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[12].

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 metacharaters. Non-quoting characters are simply copied to the `tmp` buffer, while 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 eys of `getwords()`, `getredirs()`, `getvars()`, and `expandall()`, and will reach `buildargv()` untouched. Finally, before actually passing the argument vector to the commands (either builtin or external), we use the function

[10] Except when the character is newline, see Exercise 8.4.

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

[12] However, if you *really* need to put a single quote in a single-quotes string, you can do as follows: `'don'"'"'t do this at home'`.

`unquoteall()` to remove all the quote markings. 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[13].

With this modifications in place, our new minishell correctly understands strings like "`"a space"`" and our previous example works.

Exercises

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

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

8.2. Now that we have quoting, can we solve the problem in Exercise 3.1 by typing, e.g., `cat '-'`? Explain.

8.3. 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. In particular, the PWB shell by John Mashey allowed for expressions like `$`*v* to be expanded within double quotes. The expansion is disabled if the dollar is preceded by backslash. Try to implement this feature and compare it with the solution contained in `8-env.2`.

8.4. What if we hit enter before closing a single-quote 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. Try to implement these features and compare your solution with the one in `8-env.3`.

8.5. The POSIX shell mandates some peculiar behavior for words that expand to nothing, like `$X` when `X` is either undefined or null: the word should not be included in the argument vector passed to commands. However, if the original word contained any quoting character (in any position), then the (empty) word should be retained. E.g., if we type

```
    echo hello '' $NONEXISTENT $NONEXISTENT""
```

then `echo` should receive `hello` in `argv[1]`, an empty string in `argv[2]` (coming from the `''` word) and another empty string in `argv[3]` (coming from the last word). Try to implement this feature and compare your solution with the one in `8-quote.4`.

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

9 Other features

A real shell will have many other features. Some are easy to implement, while others are much more complex. The `9-fields*` shell implement some more string processing and will be used in Chapter **??**. Some other features are implemented in the `src/sh?.c` files in myUnix. For example, it is the shell that prints "Segmentation fault", by looking at the status value returned by `wait()` (`src/sh1.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/sh2.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/sh4.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/sh5.c` and `src/sh6.c`). It is also the shell that creates pipelines of commands, using essentially the same trick as in I/O redirection (`src/sh7.c`). A few more extensions are collected in `src/sh8.c` (`||` and `&&` operators), `src/sh9.c` (subshells) and `src/shA.c` (final touches).

Ex. 1.1

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[14]). 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.

[14] 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[15] 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. 1.2

If you try, you should see some strange output that mentions the uptime, login and such. As soon as you entered Ctrl+D, `sh0` woke up and processed the `/usr/bin/wc` characters that you entered. 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/usr/wc`", i.e., 11 characters with no ending newline, it replaced the last one ("`c`") with the string terminator, and then tried to execute the resulting "`/usr/bin/w`". This is an old command that is meant to show the currently logged-in users and the programs that they are running (it may not be able to actually show anything useful, depending on how your system is configured).

Ex. 1.3

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.

Ex. 1.4

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

[15] 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.

```
/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. 1.5

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[16]. 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 1.1: “`^[`” stands for 0x40 XOR 0x5B = 0x1B, which is ESC[17]. 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 command history) in userspace. We are not using `libreadline`, and this is why our shells lack these functions.

Ex. 2.1

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.

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

[17] 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`.

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 3.1 and 8.2).

Ex. 2.2

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 1.4 can be seen as a generalization of this behavior, implemented directly by the kernel.

Ex. 3.1

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 5.2

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 6.1

When we run "`bash <script`", we see the "`hello`" output and then get our prompt back[18]. To understand what is going on, it is important to remember that a parent process shares not only the open files with its children, but also the *read and write pointers* to those files. Therefore, `bash` reads the `cat` line from its standard input and then spawns the child process that executes `/bin/cat`; the child process continues reading where `bash` left off, consuming the `hello` line and printing it. Since the script is now over, `cat` sees an EOF and exits; `bash` wakes up and tries to read more lines from standard input, but since the shared read pointer has reached the end of the script, `bash` sees an EOF and also exists. This is the expected and correct behavior in these scenarios.

Now we try with "`6-intr <script`". This time we get an error:

```
hello: No such file or directory
```

The error comes from `6-intr`, and can be explained as follows: the modified `6-intr` did not disable stdio buffering, and therefore the `fgets()` in `getcmd()` internally read past the end of the first line, actually consuming the entire `script` and copying it into the stdio input buffer. The `6-intr` shell then spawned the `cat` child, which terminated immediately (since stdin was at EOF); `6-intr` woke up from the `wait()` and called `fgets()` again; the function extracted the `hello` line from the input buffer and finally `6-intr` tried to execute a non-exitent `hello` command.

If we add a second `hello` line to the script and repeat the experiment, we see something strange:

```
hello: No such file or directory
hello: No such file or directory
hello: No such file or directory
```

The `6-intr` shell has tried to execute `hello` *three times* instead of two. The problem now is with the cleanup functions that stdio runs on `exit()`. When we `fork()` a process, the child starts with a copy of the parent's memory, which contains the entire state of the stdio library, including its buffers and its registered cleanup functions. If the child `execve()`s another program, this state is wiped out; if it doesn't, the cleanup functions are called *in the child*: this is what happened when the first child failed to `execve()` the nonexistent `hello` command and therefore continued to use the copy of the `6-intr` shell's memory: at `exit()`, the stdio cleanup function saw that the input buffer contained a line that had not been consumed (since the `fork()` was done when `6-intr` had only extracted the first `hello` line) and issued an `lseek()` to move the read pointer back. Since the read pointer is shared with the parent process, the `6-intr` shell saw the second `hello` line again!

[18] Note that we explicitly use `bash` instead of `sh`. This is because most versions of `dash`, the shell used in Debian and its derivatives for `/bin/sh`, contain exactly the bug we are discussing here and will behave incorrectly, just like the modified `6-intr` shell.

If you add a third hello line to the script, "6-intr <script" will result in an infinite loop: the first fgets() will cache the whole script, and the shell will spawn a process for each hello line; at exit, the first child will move the read pointer back to the beginning of the second hello line; the second child will try to move it back to the beginning of the third hello line, but since it uses a relative offset and the read pointer has already been moved by the first child, it will move the pointer further back, to the beginning of the script. The shell is then forced to start over, and this will repeat forever.

Ex. 7.1

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 5-intr because the semicolon is not recognized as a command separator.

Ex 8.1

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 8.2

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