Lab 1: Write a UNIX shell

Magnus Johansson

April 13, 2005

1 Background

The following is not necessary to read in order to do the assignment, but it gives a historical context.

In 1969 the first version on UNIX was developed at Bell Labs. The name "UNIX" is not an acronym, but rather a pun on MULTICS (Multiplexed Information and Computing Service), its predecessor from 1965. Around 1977 UNIX spread to University of California-Berkeley which modified it and released its own version of UNIX. This version became known as Berkeley Software Distribution (BSD) 4.2. In 1984 AT&T (of which Bell Labs was a part) started selling a commercial version of UNIX they called System 5. If you use Linux you may have seen the term "System V init scripts". This is where it comes from.

Over time many companies and universities modified UNIX into their own versions. For a thorough overview see http://www.levenez.com/unix/. There you can see a very nice picture of pretty much all UNIX variants and how they relate to each other. The major players today include Solaris by Sun, HP-UX from Hewlett-Packard, AIX from IBM, and Tru64 from Compaq. There are also a bunch of free versions including FreeBSD, NetBSD, and Linux. For a very good, and more thorough, history of UNIX, see http://www.bell-labs.com/history/unix/. This short history serves just to illustrate that there are a myriad of versions of UNIX.

The different versions of UNIX diverged over time and the need for a standardization became obvious. In 1988 the first version of POSIX (Portable Operating System Interface) was released. POSIX is a standard that specifies the user and software interfaces that must be present on a compliant system. It consists of four parts (cut and pasted from http://www.localcolorart.com/search/encyclopedia/POSIX/):

Base Definitions - a list of definitions and conventions used in the specifications and a list of C header files which must be provided by compliant systems.

Shell and Utilities - a list of utilities and a description of the shell, sh.

System Interfaces - a list of available C system calls which must be provided.

Rationale - the explanation behind the standard.

Pretty much all UNIX versions are POSIX conformant at least to some extent. Even some versions of Windows (e.g. NT, 2000) includes POSIX support.

This assignment is all about using the POSIX interface to do process management.

2 The assignment

You are to program a UNIX shell similar to, for example, BASH, which probably is the command shell you normally use when you use a UNIX system. A shell is an interface between a user and the operating system. It lets us give commands to the system and start other programs. It uses several POSIX system calls, for example fork, execve, and wait. The final version of your shell should be able to fork at least two processes and connect them with pipes. It should work just like when you type `ls | wc` in BASH. This command redirects stdout of `ls` to stdin of `wc`. Together these commands will display the number of lines, words, and characters in the output of `ls`.

You may write the shell in any programming language you wish as long as you can do POSIX calls from it, but normally C is the preferred choice, since this is the language assumed in the POSIX specification. This assignment has been prepared for Java, however, since students nowadays usually don't know C. If you do master C, it is the recommended choice, but if you don't, please use a different language. Otherwise you will end up with a lot of C problems that are unrelated to the assignment. For the rest of this document it will be assumed that you do the assignment in Java.

As you most likely know, Java is not normally compiled to machine code, but rather to byte code. This makes it a bit harder to do low level programming in it, and by default Java has no support for POSIX calls since it is not portable to non-POSIX systems. There are a few external libraries you can use, however. The one that is described here is called jtux (http://www.basepath.com/aup/jtux/). You can find the documentation for jtux in the `docs` subdirectory of the assignment. The specific parts you need to use for the assignment are also briefly described in this document. It is also a good idea to take a look at the manual pages for the different POSIX calls, since the jtux documentation is a bit sparse. You do this by typing `man -s2 fork` in the shell, for the fork system call, etc.

You are given skeleton code that you need to complement with some code to get it to work as expected. The code will by default just display a prompt and echo whatever you type. Take a look at the code and make sure you understand most of it. It's well documented so things should be clearer after you've had a look. To make things easier for you, a suggested implementation order follows. Note that there are some questions you should hand in together with the program you write, in order to pass this assignment. You should start by running the shell script `build.sh` to compile the skeleton code into byte code. You can then run it by typing `./shell.sh`. Type stuff into the shell to get a feeling of how it works right now. You exit the skeleton shell by typing `exit`.

1. Make the shell fork a child whenever you type something at the prompt. Let the child die immediately. Open up another shell (bash, not your own) and type `ps -l -uyourloginname` to see your processes. You should be able to see a zombie appear each time you type something in your shell. Try to kill the zombies using the command `kill -9 pid`, where `pid` is the process id of the zombie. You can find this in the output from `ps`. What happens? Why? Now type `exit` in your shell to exit it. Now try to see the zombies again. They are gone. What happened to them?

   The only jtux method you need to use at this point is `long UProcess.fork()`. It works exactly like the POSIX call fork(), so type `man -s2 fork` to get the details.

2. Have the shell wait for the child it forks, so that you avoid zombies. After this step you should not be able to see any zombies when your shell is running.

   You now also have to use the jtux method `long UProcess.wait(UExitStatus status)`. See the man page for `wait` for more details. Note that there are more than one man page for wait. The one you want to see is in section 2 (`man -s2 wait`). The one in the first section is a shell command and not a system call. You need to declare a variable of type UExitStatus before you call wait. This variable will contain the exit status of the process you waited for. Use the method `status.get()` to access the actual status if you wish. `wait` returns the process id of the process you just waited for.

3. Make the forked child execute new code so that you can type e.g. `ls -lF` at your prompt and have it work.

   You now also need to use the jtux method `void UProcess.execvp(String file, String[] args)`. `file` is the command you want to run, e.g. `ls`, and `args` are all arguments. Note that the first argument should always be the name of the program that's being run. This means that for this step you should call execvp something like `UProcess.execvp(arguments[0][0], arguments[0])`, where `arguments` is the array given by the skeleton code. It contains everything you need. See the man page for execvp for more details.

   Note that if you try to run a program that expects to read from stdin, it will not work. This is because your shell reads from stdin. In order to make it work you would have to redirect stdin for the program, but that is not necessary for the assignment. If you are interested in how to do this, see the last step for details. If you have started such a program, don't type `Ctrl-C` to abort it, since this will mess up your console. Instead, use a different shell and kill the relevant process from it.

4. Make the shell fork and execute more than one child if more than one command is given. At this point you shouldn't try to connect the children with a pipe. Be careful here not to create any zombies. To make the next step a little bit easier you should first fork all children, and then wait for them. Don't create one and wait for it before you spawn the next one.

   No new POSIX calls are necessary for this step.

   For the assignment it is enough to handle just two processes here.

   If you try with `ls | wc` here, you will get weird behaviour. This is because `wc` tries to read from stdin, but will not get any input as described in the previous step. Try something else instead, for example `ls | ps`. Neither of those programs try to read from stdin.

5. Connect the children with pipes so that e.g. `ls | wc` works as expected.

   You will now have to create one or more pipes. This is done through the jtux method `void UFile.pipe(int[] pfd)`. `pfd` should be an integer array of size 2. See the documentation on pipes for more details. In addition you will also need to use `void UFile.close(int fd)` to close the unused ends of the pipe, and `int UFile.dup2(int fd, int fd2)` to do the redirection. See the documentation on pipes for more details, and/or read the man pages for pipe, close, and dup2.

At this point you may have discovered that you cannot do everything in your shell that you can do in BASH. There are at least two obvious differences. First, BASH has a few built in commands. Since they are built in BASH you cannot call them from your shell, unless you implement them in your shell yourself. Examples of built in commands are `cd` and `pwd`. Second, BASH has some built in magic that does wildcard expansion and some clever things with quoted strings. In your shell, for example, `ls *.java` will not work as expected since there is no file literarily called "*.java". BASH on the other hand will expand the expression and call ls with one argument for each file that ends with ".java". The wildcard expansion is built into BASH, and is not handled by ls. You don't have to handle these things in your shell.

3 Processes in UNIX

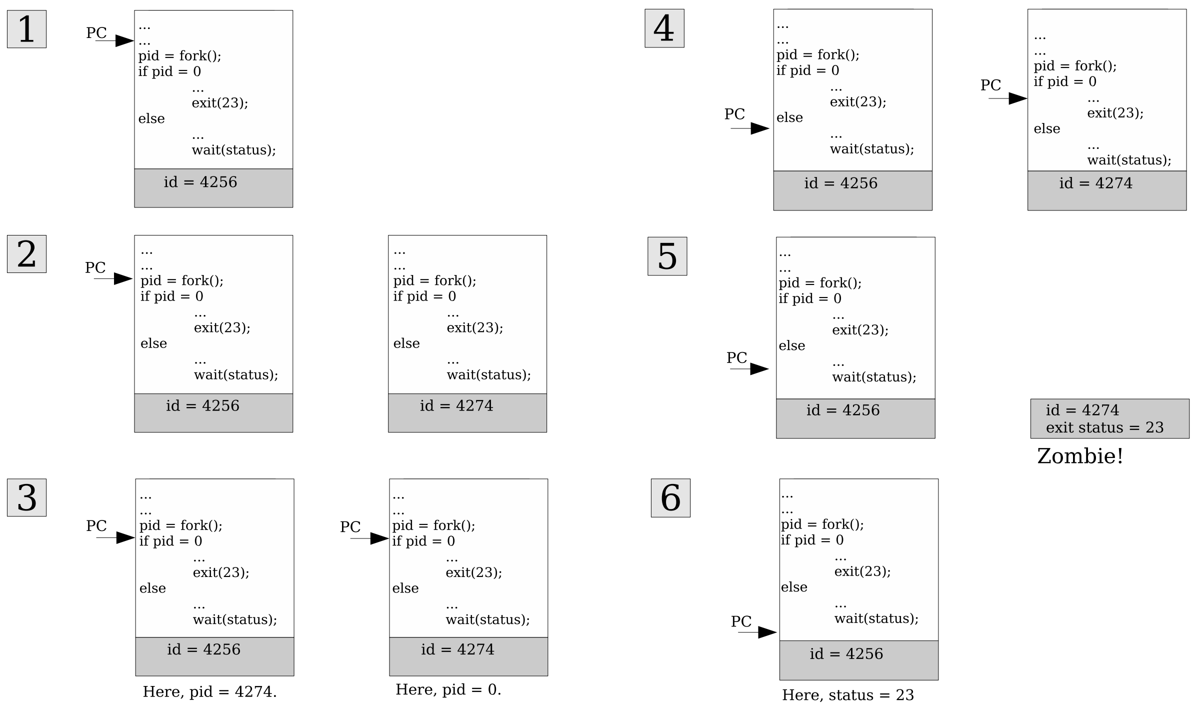
Several processes can run at the same time in a UNIX system. The operating system needs a way to keep track of each process. Information such as what state the process is in (running, sleeping, waiting, etc.), what user is running it, which files the process has open, and so on needs to be stored somewhere. All this information is stored in the process control block (PCB) of the process, which is located in kernel space. Each process also has a unique process id (pid) associated with it.

When a UNIX system starts, a single process is created, and it's called *init*. This process then forks into more processes which in turn executes new programs. These programs in turn can fork and execute new programs. This way a hierarchy of processes is created, with init as the ancestor of all processes. The init process always has a process id of 1.

The only way to create a new process in UNIX is to use the fork system call. This is what happens when a process calls fork:

1. The process calls fork.
2. The kernel creates a PCB for the new process.
3. The kernel allocates memory for the new process.
4. The kernel puts a copy of the process in the memory for the new process.
5. The kernel schedules both processes.

At this point both the parent and the child will run the same code, so to make them behave differently, the code will have to look at the return value of fork, and execute different branches depending on what the return value is. In the parent, fork will return the process id of the newly created child, and in the child, fork will return 0. The child will have copies of all variables in the parent, and of all descriptors (see next section) in the parent.



```
...
...
pid = fork();
if pid = 0
        ...
        exit(23);
else
        ...
        wait(status);
```

Figure 1: This is how fork works.

When the child exits, it will return a value. If exited normally, 0 will be returned, but if some error occurred, or it needs to return some value for some other reason, a value greater than 0 will be returned. But where will the return value be stored? When the process exits, the kernel will deallocate the memory it occupies, so there is no space to store the return value. The answer is that the return value is stored in the PCB. So when a process exits, another process may need to read the return value. For this reason, when a process exits, the memory it occupies is freed, but the PCB is not. The kernel can not release the PCB until someone has read the return value of the process. When a process is in this state, it is called a zombie. To read the return value of a process you use the system call `wait(status)`. It will store the return value in the variable `status`, and will return the pid of the process it waited for. See figure 1.

If you don't want to execute the same program in both the parent and the child, you will need to use another system call, called `exec`. It will load the contents of a given file into the process and start executing it. It will replace the old program.

4 I/O using pipes

A descriptor is an integer that identifies some kind of I/O-object in the kernel. It can for example be a pipe, a file, or a network socket. Depending on what kind of I/O-object you want to use, you create the descriptor using different system calls. For example you need a descriptor to read from a file. You get the descriptor by using the `open` system call. You then use the descriptor to access the file through other system calls such as `read` or `write`. One I/O-object may have many descriptors associated with it.

When a new process is created it will pretty much always have at least three open descriptors: 0, 1, and 2. These are more well known as `stdin`, `stdout`, and `stderr`, respectively. When you are using a shell, stdin is normally connected to the keyboard, and stdout and stdin are connected to the shell.

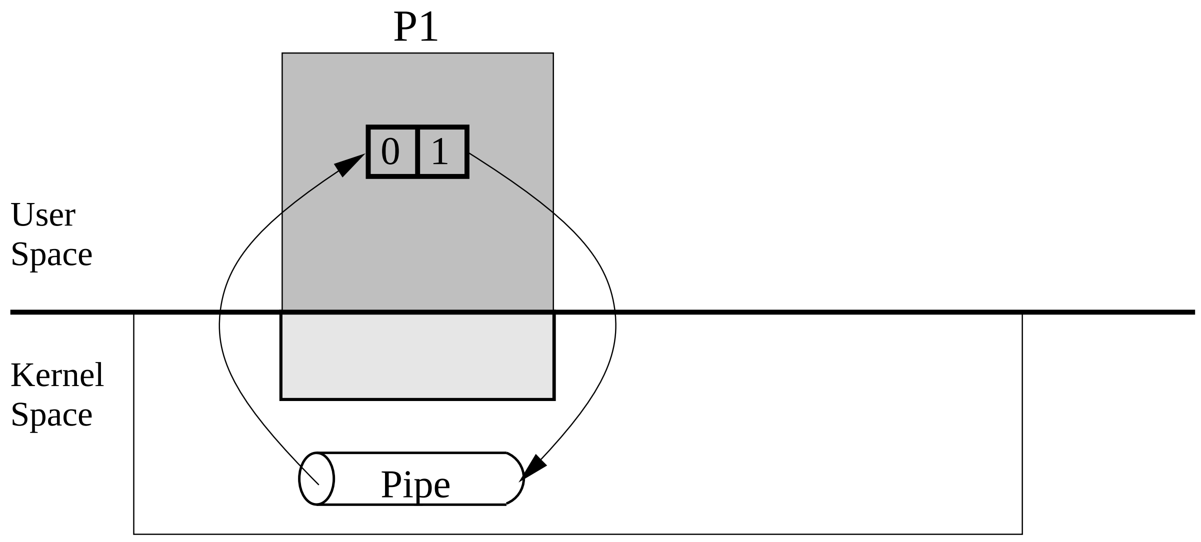


Figure 2: A pipe.

A pipe is a special kernel object. Basically it is just a pair of descriptors connected together. You write to one end and read from the other. A pipe is typically used in communication between a parent and a child, or between siblings. You create it by using the system call `pipe`. You give it an array of two integers as argument, and after the call these two integers will be descriptors for the ends of the pipe. If you then fork off a new process, this process will have copies of the descriptors, and you can then talk to the child through the pipe. If you fork off two children, these two can talk to each other through the pipe.

A common use of pipes is when you use a shell and type for example `ls | less`. This will create a pipe between the programs ls and less, and it will redirect stdout of ls to stdin of less. To do this redirection you use the system call `dup`. This system call will duplicate a descriptor so that the copy will refer to the same I/O-object as the original. For example, to redirect stderr to stdout you use a variant of dup, called `dup2`, which takes two arguments. If you call it like `dup2(stdout, stderr)` it will replace the contents of stderr with the contents of stdout. As a result, all output to stderr will now go to stdout instead.

5 More information

In these pages I've just presented what is necessary to complete the lab. For a more thorough discussion on all these concepts, you can visit
`http://www.cim.mcgill.ca/∼franco/OpSys-304-427/lecture-notes/lecture-notes.html`.

This is the first time this particular lab is used. I want your feedback!!! Please tell me what's good and what's not.