Lab 1: Write a UNIX shell

Created by Magnus Johansson (March 22, 2007)

Adapted from a version by John Hakansson (January 30, 2008)
April 13, 2008 & March 25, 2009 (Léon Mugwaneza)

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 experiment some process management operations.

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*, *getpid*, *getppid*, 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 do not know C. If you do master C, it is the recommended choice, but if you do not, 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 or C.

2.1 POSIX and Java : jtux

[if you will not use java, you may skip this].

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.

2.2 Writing a shell using POSIX

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 is well documented so things should be clearer after you have had a look.

Java :	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`.
C:	You should start by running the command `g++ -Wall –Werror  -o shell shell.c` to compile the skeleton code. You can run it by typing `./shell.`

Run the shell and type some stuff to get a feeling of how it works right now. You exit the skeleton shell by typing *`exit`*.

Step 1: Experiment the fork system call.

a. Read section 3 of this document about processes in Unix, and the manual page for the `fork` system call (`man –s2 fork`).

b. 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 `id` 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?

Java :	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.

☛ **Save your work in a file named step1 (.java or .c). You will have to hand in that program together with your answers to the questions in this step.**

<u>Step 2: Waiting for a child process, getting process and parent process identifications.</u>

a. 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.
Read 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.

Java :	You now also have to use the jtux method `long UProcess.wait(UExitStatus status)`. 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.

b. Modify your program to make:

- the child process display its id and the id of its parent process before it dies, and
- the parent process (your shell) displays the value returned by fork, the value returned by wait, and its own pid.

As the two processes use the same terminal for output, use the string "`child:`" before each information displayed by the child, and the string "`parent:`" before each information displayed by the parent process.

Make sure you understand that the only way for a process to get its id or its parent's id is to ask the operating system. You will need to use the `getpid` and `getppid` POSIX system call for that. Read the manual page for `getpid` (get process identification) and `getppid` (get parent process identification) for more details.

Java: You now have to use the jtux methods `long UProcess.getpid()` to get the process id and `long Uprocess.getppid()` to get the parent process id.

Why has a process to resort to the OS kernel to have its own identification?
What are the outputs of the 2 processes? Are they coherent one with the other?

Note: The process, and parent process identification are some of the information stored in the PCB. Among other information a process can get from the operating system through POSIX system calls are: the id of user who owns the process (`getuid`), the user group id (`getgid`), the current working directory (`getcwd`), the environment variable (`getenv`), information describing the resources utilized by the current process (`getrusage`), limits on the consumption of system resources (`getrlimit`), etc. ***You do not have to learn about these system calls in this assignment but it is good you know the operating system stores a lot of information on processes.***

☛ **Save your work in a file named step2 (.java or .c). You will have to hand in that program together with your answers to the questions in this step.**

Step 3: Experiment the exec call to replace the code of a process with a new program

a. Make the forked child execute new code so that you can type e.g. `ls -lF` at your prompt and have it work.
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, do not 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.

Java:	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 is being run. This means that for this step you should call `execvp` with 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.
C:	Use the POSIX call `int execvp(const char *file, char const *argv[])`, where `file` is the command you want to run, e.g. `ls`, and `argv` are all arguments. Note that the first argument should always be the name of the program that is being run. This means that for this step you should call `execvp` with something like `execvp(args[0], args)`, where `args` is the array given by the skeleton code. It contains everything you need. See the man page for `execvp` for more details.

b. Make the shell fork and execute more than one child if more than one command are given recall you separates commands typed on the prompt by the pipe character (|). At this point you should not 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. Do not 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 tries to read from `stdin`.

☛ **Save your work in a file named step3 (.java or .c). You will have to hand in that program.**

Step 4: Connecting sibling processes using pipes

a. Read section 4 of this document. It is about pipes and redirection.

b. 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 POSIX call `pipe`. See the documentation on pipes for more details. In addition you will also need to use the POSIX call `close` to close the unused ends on the pipe, and the POSIX call `dup2` to do the redirection. See the documentation on pipes for more details, and/or read the man pages for pipe, close, and dup2.

Java:	Use *`void UFile.pipe(int[] pfd)`* where `pfd` should be an integer array of size 2. Also use use *void UFile.close(int fd)* and *int UFile.dup2(int fd, int fd2)*.
C:	use POSIX calls *int pipe(int fildes[2]), int close(int fildes)*, and *int dup2(int fildes, int filedes2)*.

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 do not have to handle these things in your shell.

☛ **Save your work in a file named step4 (.java or .c). You will have to hand in that program.**

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 is called `init`. This process then forks into more processes which in turn execute 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 (see figure 1.a). In the parent, fork will return the process id of the newly created child, and in the child, fork will return 0 (a negative integer is returned if `fork` fails). The child will have copies of all variables in the parent, and of all file and other I/O descriptors (see next section) in the parent.

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. When a process exits, the kernel will reclaim the memory space the process occupies. The return value is stored in the PCB. As when a process exits, another process may need to read the return value, when a process exits, the memory it occupies is freed, but the PCB is not. The kernel cannot 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(pid_t wait(int *stat_loc);)*. It will store the value returned by at process termination in the variable referenced by `stat_loc`, and will return the `pid` of the process it waited for (see figure 1.c).

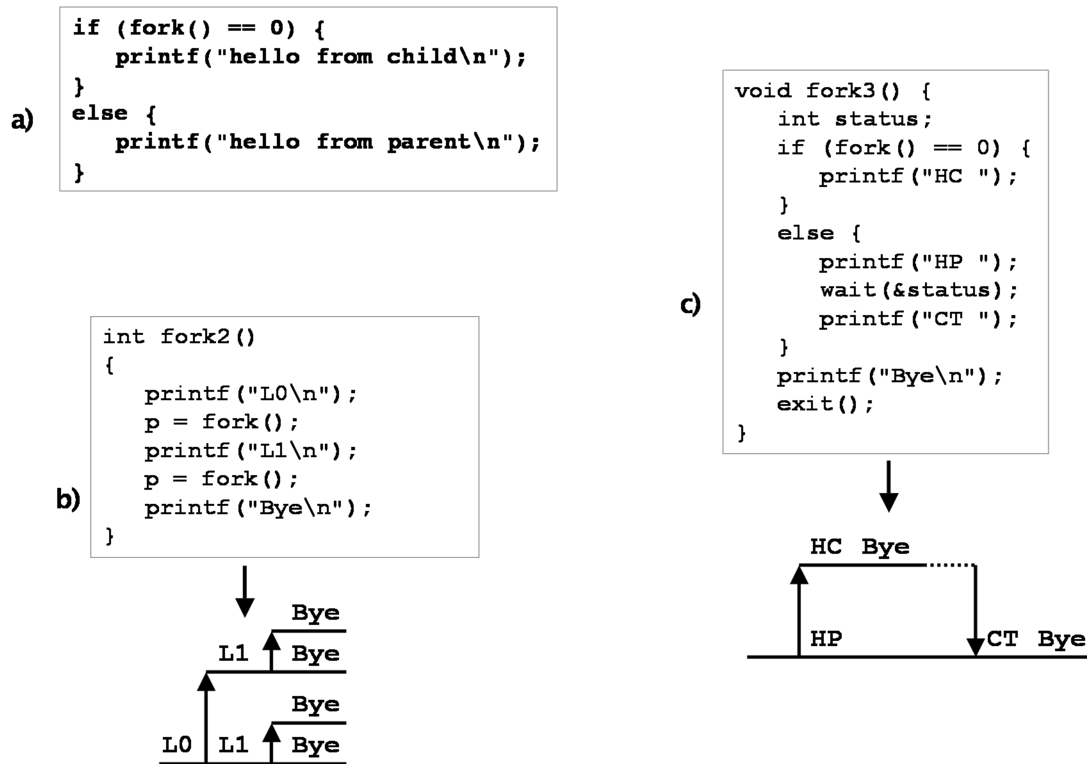

Figure 1: How fork and wait work. a) fork returns 0 in the child and the pid of the child in the parent, b) parent and child can create new processes, c) parent waits for child.

If you do not want to execute the same program in both the parent and the child, you will need to use another system call of the `exec` family `(exec, execve, execlp, execvp, …)`. This exec command will load the content of a given file into the process and start executing it. It will replace the old program.

4 I/O using pipes

I/O descriptors

A descriptor is an integer that identifies some kind of I/O-object in the kernel. The I/O object 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.

In fact we have an indirect naming: the integer we call a descriptor is an index of an entry in the process's table of descriptors where the real descriptors are stored. The table of descriptors of a process is stored in kernel memory in the PCB of the process.

When a new process is created it will pretty much always have at least three open descriptors: 0, 1, and 2. These are better known as stdin, stdout, and stderr, respectively. That is, the first three entries of the process's table of descriptors are used for stdin, stdout, and stderr (see figure 2 below). They contain pointers to buffers used for I/O. When you are using a shell, stdin is normally connected to the keyboard, and stdout and stdin are connected to the shell.

Pipes

A pipe is a special kernel object. Basically it is just a pair of descriptors connected together (pipe ends). 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 a pipe by using the system call `pipe` with an array of two integers as argument. After the call these two integers will be descriptors for the ends of the pipe (in fact indexes of entries in the table of descriptors where real descriptors are stored). When a process creates a pipe (using the system call pipe) two free entries in the process's table of descriptors are allocated to contain references to the read end and the write end of the newly created pipe. The indexes of these 2 entries are returned in the 2 integers array passed as argument to the pipe system call (read end in position 0, and write end in position 1). This is illustrated in figure 2 below.

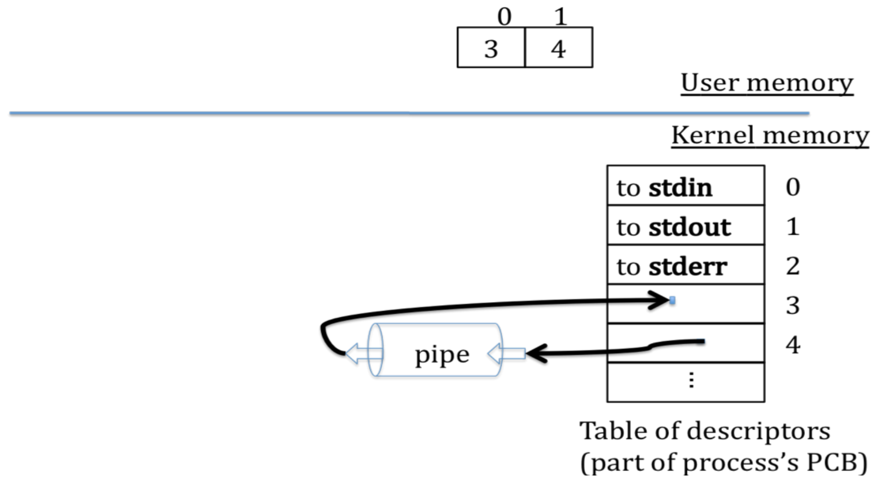

Figure 2: How a pipe is represented. Here the pipe read end is at entry 3 and the write end is at entry 4 in the process's table of descriptors. Therefore, in the 2 integers array used in user memory, entry 0 contains 3 and entry 1 contains 4.

Pipes are operating system objects that processes can use if they have properly acquired references to them (pipes created by the process or inherited from parent). When a new process is created using the unix *fork* system call, the child process will have a copy of the parent process's address space (a copy of all code and data), a copy of the CPU registers, and also a copy of most of the process's PCB (including the table of descriptors) stored in kernel memory (see figure 3). Note that the real pipe object is not duplicated by fork (only copies of the references to the pipe ends in the process's table of descriptors together with the indirect references in user space are maid).

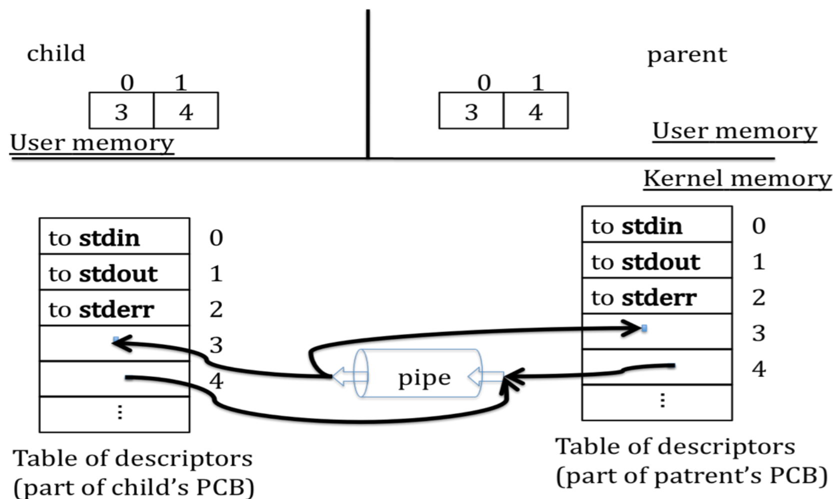

Figure 3: After the process in figure 2 creates a child using fork.

If the parent process had references to a pipe before calling fork (as in figure 3), it is then possible for the parent and child processes to communicate through the pipe (one writing to the write end of the pipe and the other reading from the read end of the pipe). Sibling processes created by a common parent process holding references to a pipe can also communicate through that pipe.

A pipe is a half-duplex communication object. For proper operation, more than one process should not use a pipe end at the same time. A process should first close the end it is not using (using the system call *close*).

I/O redirection using pipes

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* (*int dup(int x))*. The call *dup(i)* allocates a new entry in the process's table of descriptors, copies the content of entry i of the process's table of descriptors into the new entry, and returns the index of the new entry. We will use *dup2*, a variant of dup to do I/O redirection (*void dup2(int x, int y)). The call dup2(i, j)* copies the content of entry i in the process's table of descriptors into entry j (the 2 entries now contain the same real descriptor). For example:

- To redirect stderr to stdout one uses *dup2(stdout, stderr)* to replace the content of entry 2 (stderr) in the process's table of descriptors with the content of entry 1 (stdout). As a result, all output to stderr will now go to stdout instead ;
- To redirect stdin to the read end of a pipe p, one uses *dup2(p[0], 0)*. As a result, all input from stdin will now be from the read end of the pipe p (see figure 4). Now the pipe read end can be named using 2 entries in the table of descriptors, the process can release one entry using close. For example, *close(p[0])* will mark entry 3 as free (the process can still use the pipe read end through entry 0 of the table of descriptors);
- To redirect stdout to the write end of a pipe p, one uses *dup2(p[1], 1)*. As a result, all output to stdout will now go to the write end of the pipe p. We can release entry 4 of the table of descriptors (using *close(p[1]*).

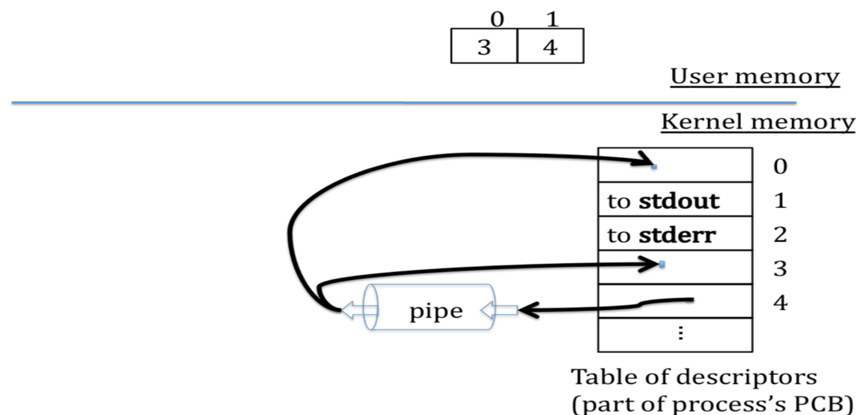

Figure 4: What dup2(p[0], 0) does (starting from figure 2 and assuming the 2 integers array in user mode has name p). Entry 3 (p[0], the pipe read end) of the process's table of descriptors is copied in entry 0.

5 More information

In these pages I have just presented what is necessary to complete the lab. For a more thorough discussion on all these concepts, see "Advanced UNIX Programming, 2nd Edition" by Marc J. Rochkind (Pearson, 2004). You can also see http://www.cim.mcgill.ca/~franco/OpSys-304-427/lecture-notes/.

I want your feedback on these labs!!! Please tell me what is good and what is not.