Programming Assignment #1: Process Management

Due: Check My Courses

Description of the Assignment

Let's start with a tiny shell that has minimal functionality. The tiny shell works like the following. We read a line of input, if the input is valid (longer than a single newline character), we run the command. Otherwise, the tiny shell waits for the next input line.

```
while (1) {
      line = get_a_line();
      if length(line) > 1
            my_system(line);
}
```

In the above pseudo code, we use the `my_system()` and `get_a_line()` functions. You need to implement these functions. The `my_system()` function takes a string and tries to execute it. The assumption is that your string is pointing towards a valid executable file that is available on the disk. To execute the file specified by the string, we need to create another process. In that newly created process, we load the specified file and execute it. You can use system calls to create a new process and load the executable from the file.

In the `get_a_line()` you need to keep reading the input until the end of line. That is, you keep reading until the RETURN key is pressed. It is necessary to allow editing while reading the line of input. That is, you should allow backspace to delete a character after it has been entered into the line. There is no need to support cursor movement-based edits, but it is desirable to support it (note: you won't be penalized if you are not supporting cursor-movement based edits).

Your tiny shell is an interactive program. It reads from the terminal. Such a program can be run non-interactively by redirecting the required input from an input file. For instance, suppose the tiny shell is named **`tshell`** and we have an input file name **`input.txt`**. The **`input.txt`** file has all the commands you want to run in the tiny shell. Then, issue the following command.

```
tshell < input.txt
```

You should be able to run the shell and it should terminate assuming your tiny shell is detecting the end-of-file condition (input lines with just the new line character) and terminating itself.

Your `my_system()` will use the `fork()` system call. In addition, reading input from the terminal and writing output to the terminal will also use system calls (at least in the underlying implementation). Use **`ltrace`** and **`strace`** to trace the program and detect all the library routines or system calls that are used by the program. For instance, the following command should give the library routines that are called by the tiny shell.

```
ltrace tshell < input.txt
```

Show a trace of the ltrace and strace output in the document you submit with the tiny shell implementation.

NOTE: The above is a suggested structure. It is acceptable to restructure the code to accommodate all the requirements.

A little bit more about **`my_system()`**.

`my_system()` is a function that spawns (i.e., creates) a child process and runs the command you passed as the argument to the call, assuming the command you requested to run is a valid one and is present in the machine. For example, if you specify **'`/bin/ls`'** it is going to execute the directory lister command. You could program so that absolute path names of the commands are not necessary. That is, you can just specify **'`ls`'** and it should do the same run as the previous one. It is important that your implementation makes the tiny shell fault tolerant. That is, if the executable crashes, the process (i.e., the tiny shell) that is calling the **`my_system()`** function **does not crash** as well.

An implementation that uses **`fork()`** reflects the semantics of the **`fork()`** system call or library function (note that fork() *is available as both*). Because you cannot do much to change the behavior of **`fork()`**, the **`my_system()`** implementation will exhibit a behavior that is dictated by the **`fork()`** operation.

```
#include <unistd.h>

pid_t fork(void);

                    In parent: returns process ID of child on success, or –1 on error;
                                          in successfully created child: always returns 0
```

With **`fork()`**, after a successful call, we have two processes and both continue the execution from the point where **`fork()`** returns in the program.

The implementation of **`my_system()`** function is completely up to you. However, you need to ensure that **`my_system()`** completes an ongoing execution before it launches the next command. That is, it does not put the command in the background. In order, to ensure that **`my_system()`** waits for an ongoing execution, you need to use the **`wait()`** system call. Here is a brief description of the call. You can always get more information by consulting the man page.

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
                         Returns process ID of child, 0 (see text), or –1 on error
```

The return value and *status* arguments of *waitpid()* are the same as for *wait()*. (See Section 26.1.3 for an explanation of the value returned in *status*.) The *pid* argument enables the selection of the child to be waited for, as follows:

- If *pid* is greater than 0, wait for the child whose *process ID* equals *pid*.
- If *pid* equals 0, wait for any child in the *same process group as the caller* (parent). We describe process groups in Section 34.2.
- If *pid* is less than –1, wait for any child whose *process group* identifier equals the absolute value of *pid*.
- If *pid* equals –1, wait for *any* child. The call *wait(&status)* is equivalent to the call *waitpid(–1, &status, 0)*.

In the above description of the tiny shell we were only concerned about "external" commands. That is commands that are supported by files on the disk. For example, when you type **`ls`**, we run the file **`/bin/ls`**. So, **`ls`** is an external command. There are *internal commands* in a shell. These commands modify the state of the shell. One of the important states of the shell (or the process that runs it) is the present (current) working directory. This is the directory that is searched by default by the shell process when it wants to open a file. The present working directory is an attribute of the process that is maintained inside the kernel. You need to use a system call to change that attribute.

You need to implement **`chdir`**, **`history`**, and **`limit`** as the internal commands. The **`chdir`** would change the current working directory of the process, **`history`** would list the last 100 commands that were executed in the tiny shell, and **`limit`** would set the upper limit for the allowed resource usage. Your implementation of deals only with setting upper limit for the memory size that could be consumed by a process (application).

The next part of the assignment is working with FIFOs. FIFOs are named pipes. We have already seen anonymous pipes in the lectures. They are one way to perform inter-process communication. FIFOs create the same, but the name of the pipe is left in the filesystem. This allow two arbitrary processes to communicate using a FIFO. Like the pipe, a FIFO has a reading end and writing end. You create a FIFO using a command like the following.

We can create a FIFO from the shell using the *mkfifo* command:

```
$ mkfifo [ -m mode ] pathname
```

After running this command, you should see a FIFO with the given name in the file system (i.e., in the current working directory).

You need to modify the tiny shell so that a FIFO can be used as the argument in the shell. If the command running in the tiny shell is outputting data to the terminal, it should go into the FIFO. Similarly, if the command is reading data from the terminal, it should come from the FIFO. Here we are considering commands that are not even written by us. For example, commands such as `ls` (directory listing) output the list of files to the terminal. When you run the command `ls` the tiny shell is responsible for redirecting the output to the FIFO.

Similarly, we can run `wc` to count the number of characters, words, or lines. By connecting the second tiny shell instance to the reading end of the FIFO we can make the `wc` running there to count the output from `ls`.

This command piping could be achieved in many different ways (including anonymous pipes). We want to do that piping using named FIFOs. You need to have command piping working for two commands only.

The last part is signal handling. If Ctrl + C is pressed SIGINT signal will be raised. You need to handle the signal, otherwise the shell would be terminated. When the Ctrl + C is pressed your shell would run a termination function and ask the user for confirmation. If the user confirms his/her intent to terminate the shell would terminate, otherwise it would ignore the signal. Similarly, you will handle Ctrl + Z as well. In this case, you will just ignore that signal. Ctrl + Z raised the SIGTSTP signal.

To implement the limit command, use the `getrlimit` and `setrlimit` system calls. You can consult the man pages to get to know more about them.

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

You need to use the correct resource to set the limits on. Read the man page and look at RLIMIT_AS, RLIMIT_DATA, etc. Determine the resource that is relevant to the maximum allowable memory use and set it. Run a test program that would allocate memory and show your shell would terminate that application when it reaches the specified limit.

Turn-in and Marking Scheme

The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable. **Your programming assignment should compile and run in Linux. Otherwise, the TAs can refuse to grade them**. Here is the mark distribution for the different components of the assignment.

What Should be Turned In?

1. C program for implementing the tiny shell.
2. Document showing the traces of libraries and system calls used by your shell. How did you isolate the system calls that are used by your tiny shell from the ones used by the programs running inside the tiny shell.

A marking scheme will be posted very soon. The marking scheme would grade the shell in stages. That is, if you don't implement FIFO and signal support you will still receive partial marks.