CSE 415 - Operating Systems
Project #2 - Fully-Featured Shell
Spring 2012 - Prof. Butler[1]

Due date: May 24, 2012

In this assignment, you will implement a more fully-featured shell. We'll call it *kinda-sh* for now, but you can come up with your favorite name if you so desire. Having some experience with building a shell, you'll now add more interesting tasks such as foreground and background processes, standard input and output file redirection, two-stage pipelines, and minimal job control. To accomplish this task you will only use the functions specified in this document. You may reuse code from the previous assignment, but only your own code. Note that you are allowed to work in groups of two for this project.

1 Specification

As described in the first project, a shell is at its heart a simple loop that prompts, executes, and waits. Upon each iteration, it prompts the user for a command, attempts to execute that command as a new process, waits for that process to finish, and re-prompts the user until `EOF`. Your shell will add some additional functionality: redirection, two stage pipelines, background/foreground processing, and minimal job control. You may use any code that you wrote from the previous assignment.

1.1 Differences From First Project

Before getting into the gritty details of the specification, here are some upfront differences from the previous project that you should keep in mind. First, there is no longer a time limit on program run time. In fact, you may not use the `alarm(2)` system call at all. Second, your shell must handle program command line arguments. To ease parsing, we have provided one for you (see below for more details). Finally, you should now use `execvp(3)` instead of `exeve(2)` so you will not need to search the path yourself (see below for more details).

1.2 Redirection

Each program has three standard file descriptors: standard output (`stdout`), standard input (`stdin`), and standard error (`stderr`). Normally, standard output and error are written to the terminal and standard input is read from terminal. One shell feature is the ability to redirect the standard file descriptors to (or from) files. A user requests a redirection by employing < and > symbols. Their usage is best demonstrated by example.

Consider the following command-line input:

```
kinda-sh>  head -c 1024 /dev/urandom > random.txt
```

The `head` program will read the first 1024 bytes from `/dev/urandom` which would normally be written to standard output. However, the > symbol indicates that standard output redirect should be redirected to the file `random.txt`. Ergo, a file named `random.txt` is created and the random 1024 bytes are written to it. Standard input can be redirected in a similar manner.

[1]Many thanks to Adam Aviv, Perry Metzger, Sandy Clark, Micah Sherr, Stefan Miltchev, and Eric Cronin for the design and refinement of this assignment.

```
kinda-sh> cat < /proc/cpuinfo
processor        : 0
vendor_id        : GenuineIntel
cpu family      : 6
(...)
```

It is also possible to combine redirections. The following command copies `/proc/cpuinfo` to a local file named `cpuinfo`:

```
kinda-sh> cat < /proc/cpuinfo > cpuinfo
```

The order of the redirection symbols does not matter. This command is equivalent to the previous one:

```
kinda-sh> cat > cpuinfo < /proc/cpuinfo
```

If the standard output redirection exists already, it should be opened and truncated, and if it does not exist, it should be created. Conversely, if the standard input file does not exist, an error should be reported.

1.2.1 `dup(2)` and `open(2)` system calls

To accomplish this task, you will use the `dup2(2)` and `open(2)` system calls. The `dup2(2)` system call will duplicate a file descriptor onto another, and the `open(2)` system call will open a new file descriptor. Here is a simple example of their usage (without error checking); more details can be found in the manual:

```
new_stout = open("new_stdout", O_WRONLY | O_CREAT, 0644);
dup2(new_stdout,STDOUT_FILENO);
write(STDOUT_FILENO, "Helloooo, World!!!!",9);
```

This code snippet will open a new file named `new_stdout`, creating it if it does not exist. The file descriptor mode is "write only," and if it is a new file, the mask is set to 0644 (this is octal notation). Next, the new file descriptor is duplicated on to the current standard output file descriptor, and all subsequent writes to standard output will actually be written to `new_stdout` instead.

1.2.2 Redirection Errors

If a user provides invalid redirections, your shell should report errors. This could be because there are contradictory redirections of the same standard file descriptor or because an open failed. Regardless of the error, your shell should **gracefully handle and report user input errors.**

1.3 Two-Stage Pipelines

Pipes are another form of redirection, but instead of redirecting to a file, a pipe connects the standard output of one program to the standard input of another. Again, this is best demonstrated via example:

```
kinda-sh> head -c 128 /dev/urandom | base64
```

Like before, the `head` program will read the specified number of random bytes from `/dev/urandom`, but instead of redirecting the output to a file, it is piped to the `base64` program. In many full feature shells, pipelines can be arbitrarily long. In this assignment you are only required to implement a two-stage pipeline, i.e., one process can pipe output to the input of another.

Extra Credit (5pt) For 5 points of extra credit, implement a pipeline that can be arbitrarily long. All the same requirements described above/below must still hold. If you attempt the extra credit and it affects your two-stage pipeline, it may influence your grade. The first priority should be to implement the two-stage pipeline.

1.3.1 `pipe(2)` system call

A pipe is a special unidirectional file descriptor with a single read end and a single write end. To generate a pipe, you will employ the `pipe(2)` system call which will return two file descriptors, one for reading and one for writing. You will then `dup2(2)` each end to the standard input and output of the respective programs in the pipeline. Be sure to close the file descriptor that is unused otherwise your pipe will not work. The manual page for `pipe(2)` is very helpful, so be sure to scroll all the way down.

1.3.2 Pipes, redirections, and errors

It is possible to mix redirection symbols with pipes, and this can naturally lead to invalid inputs. For example:

```
kinda-sh> cat /tmp/myfile > new_tmp | head > output
```

does not make sense, as there are two directives for redirecting standard output of `cat`. However, the command below is allowed because there are no overlapping redirections:

```
kinda-sh> cat < /tmp/myfile 2> err | head > output
```

If a user provides more than two pipeline stages, you should also report an error and not execute the pipeline. Again, your shell should gracefully handle and report all errors.

1.4 Jobs and Process Group Isolation

When executing a pipeline, there are multiple processes, since your shell will fork a new process for each part of the pipeline. Both processes are part of the same *job*; they are part of a single command line execution, be it a pipeline or not. Each job must be in its own **process group** that is different then the shells process group and other jobs process groups. Even a job containing a single process must be in its own process group. To set a process group you will use the `setpgid(2)` system call, and you should refer to the manual for more details.

Isolating jobs via process groups can affect terminal signaling. Before, all jobs were members of the same process group (the shell's process group) but now they will be in separate groups. You may need to block certain signals during critical sections. See the `signal.h` and `signal(7)` manuals and *APUE* for the reasons why certain signals are delivered, as well as which system calls are re-entrant and which are not. If a non re-entrant (or unsafe) function is used in context where it can be interrupted, the behavior of the program is undefined. This can cause many headaches and bugs, so be sure to consider what is critical and what is not. For signal blocking, refer to the `signal(2)`, `sigprocmask(2)` and `sigsuspend(2)` manuals.

1.5 Foreground and Background Processes

A key feature of your shell is the ability to start a job in either the foreground or the background. In the previous assignment, all jobs executed in the foreground. The shell executed the job and waited until it completed before prompting the user for more input. Alternatively, a job can be started in the background, and the shell will immediately prompt the user for the next command following executing the background job.

To background a job, the user places an ampersand (&) following the command. The & should only be accepted if it is the last input. Generally, it can appear within the command, but it has a different meaning. Here is an example:

```
kinda-sh> sleep 10&
Running: sleep 10
kinda-sh> echo "Hello World"
Hello World
```

`sleep` is started in the background, and the shell prompts after execution allowing the user to `echo` prior to the completion of `sleep`. Note that the user was notified of the background process, and that is described in more detail below.

1.5.1 Synchronous vs. Asynchronous Signal Handling

With multiple jobs executing at the same time, your shell will not always be able to simply call `wait(2)` and expect everything to work. For example, if there is a background job that completes while a foreground job is running, `wait(2)` will return due to the background completion and not the foreground one. Moreover, your shell must wait on all completing jobs in the background so they do not become zombies, and worse, a job may have two processes, each completing at different times, all of which must be waited on.

In this assignment you should implement **asynchronous signal handling**. You may implement the suboptimal synchronous signal handling for a *5 point penalty*. In both cases, your implementation should correctly wait on all process and jobs.

Synchronous Signal Handling In the synchronous approach, the shell will periodically poll for all child state changes by using `waitpid(2)` with the `WNOHANG` flag. That means, `waitpid(2)` will not block and return immediately if there are no signaling processes. An appropriate time to perform synchronous signal handling is just before a prompt to not interrupt user experience[2].

Asynchronous Signal Handling In the asynchronous approach, the shell sets up a signal handler for `SIGCHLD`. Remember, `SIGCHLD` is delivered whenever a child changes state (e.g., running to stopped, running to exited, etc.), and this invokes the signal handler. In the handler, the signaling process (or processes) are identified either via the `waitpid(2)` system call or the `siginfo_t` part of the `sigaction` structure. Remember you must wait on the child, otherwise you will create zombies.

There is still a bit more to consider. What happens when you are handling a `SIGCHLD` and another child changes state? You may find it useful to block certain signals in the handler.

[2] **Extra credit (2 pts):** What are the disadvantages of synchronous job control?

1.5.2 Terminal Control

With background and foreground jobs, your shell is now responsible for delegating the terminal control. This becomes particularly important when jobs are isolated via process groups. For example, if a process group that is not in control of the terminal attempts to read from it, a `SIGTIN` signal is delivered whose default action is to stop. That could be your shell or a job it is executing, so it is important to properly set the terminal foreground group.

This concept is best demonstrated with the `cat` program with no arguments whose default action is to read from standard input and write to standard output. If `cat` is executed in the foreground, then it should be in control of the terminal. Take this execution as an example:

```
kinda-sh> cat > file.txt
Hello World
^D
kinda-sh>
```

Here, the user used cat as a mini-text editor by redirecting input to the file `file.txt`. However, if this were executed in the background:

```
kinda-sh> cat > file.txt &
Running: cat > file.txt
Stopped: cat > file.txt
kinda-sh>
```

The `cat` command is stopped because it attempted to read from standard input when it was not in control of the terminal. A `SIGTIN` was generated and the default action is to stop, as represented in the status messages (see below for more info regarding status messages). Further, many programs rely on terminal signals, such as `SIGTIN` and `SIGTOU`, to function correctly.

To do this, you will use the library call `tcsetpgrp(3)`. This is perhaps the most misunderstood and most poorly documented function you will encounter. Dont fret. The general rule is this: Whatever process group that is in the foreground, be it the shell or another job, should be the terminal controlling process group.

1.5.3 Terminal Signal Relaying

Another side effect of having multiple process groups is handling the delivery of terminal signals. There are two that you should pay particular attention to: `SIGTERM`, the signal generated by `Ctrl-C`; and, `SIGTSTP`, the signal generated by `Ctrl-Z`. In the previous assignment, you may have noticed that `Ctrl-C` caused your shell and the running process to exit. That should no longer be the case.

Your shell is responsible for properly relaying those signals to the foreground process. If you have proper process group isolation and terminal control setup, then this should happen by default, but due to the asynchronous nature of signal deliver, be careful about your assumptions. Your shell should never stop or exit due to the delivery of `SIGTSTP` or `SIGTERM`. You should set up appropriate signal handlers or signal masks to either relay the signal to the right process group or block/delay the signal if necessary. See the `signal(2)`, `sigprocmask(2)`, and `sigsupsend(2)` system calls.

1.6 Minimal Job Control

Your shell must provide the ability to bring the most recent background process to the foreground and restart the most recent stopped process in the background. To do so, your shell will provide two built-in commands, `bg` and `fg`.

`bg` will deliver a `SIGCONT` signal to the most recently stopped background job. And similarly, `fg` will bring the most recently background job to the foreground and also deliver `SIGCONT` signal in case that job is stopped (e.g., because the user stopped it by `Ctrl-Z`). For example:

```
kinda-sh> sleep 100
^Z
Stopped: sleep 100
kinda-sh> bg
Running: sleep 100
kinda-sh>fg
sleep 100
^Z
kinda-sh>fg
sleep 100
kinda-sh>
```

The sleep command is stopped when the user delivers a `SIGTSTP` via `Ctrl-Z`, and next it is restarted in the background using the built-in `bg` command. It is brought to the foreground via `fg`, where it is stopped again, and restared. Finally it completes, and the user is prompted for more input.

To complete this task, your shell must keep track of the most recent job to deliver `SIGCONT` to the right process(es). Note, if there is a pipeline, then all processes in the pipeline should receive `SIGCONT`. You should refer to the `kill(2)` and `killpg(2)` system calls. If there is not a most recent job (e.g., it finished) then fg and bg should report an error to the user.

You are not required to keep a queue of the most recent jobs. You are only required to keep track of the most recent job. Once the most recent job has completed, even if there are uncompleted background jobs, you may report an error.

Extra Credit (3pts) Maintain a queue of the recent background jobs such that if one completes, the next time `bg` or `fg` is called, the next most recent job gets the `SIGCONT` signal. You may find a linked list useful for this task.

Extra Credit (5 pts) Add to your shell a built-in command `jobs` that will report on the current background processes. It should report their status (stopped or running) and their job id. Additionally, you should provide the ability to bring any job to the foreground or restart a job in the background by using its job id. For example: in this following code, no signal handler is registered:

```
kinda-sh> jobs
[1] sleep 100 (running)
[2] cat (stopped)
kinda-sh> fg %2
restarting: cat
^C
kinda-sh>jobs
[1] sleep 100 (running)
```

1.6.1 Reporting on Jobs Status

Your shell must notify the user when a background job changes state. You should try to not spam the user and interrupt him/her while entering input. Instead, you should queue up these messages and print them prior to printing the prompt. For example:

```
kinda-sh>sleep 5 &
Running: sleep 5
kinda-sh> cat
^C
Finished: sleep 5
kinda-sh> sleep 100
^Z
Stopped: sleep 100
kinda-sh>echo 10
10
kinda-sh>
```

The first notification is that the `sleep 5` job is running in the background, which finishes while executing `cat` in the foreground and the user is notified before the next prompt. The last notification occurs when the `sleep 100` job is stopped. Noticed that there was no notification for the `echo` command or `cat` command as they completed while in the foreground. The user does not need notification of this fact.

1.7 Data Structures

To complete many of the tasks in this document, we suggest that you implement your own data structures. You may find it particularly useful to implement a linked-list structure and a hash-table structure. Although neither is required, it will greatly help your development.

1.8 Code Design

This is a large, complex assignment that will likely require well over a thousand lines of code, and possibly more. You should **organize your code in a reasonable way**. It is not reasonable to have all your code in one file. Take the time to break your code into modules/libraries that you can easily reference and include in your build process. Poor code layout **may adversely affect your grade**.

1.9 Arguments to *kinda-sh*

kinda-sh does not require any arguments.

1.10 Command Line Input

Your shell does not need to handle arbitrarily long input on the command line. Instead, input may be truncated to 1024 bytes, but your shell should gracefully handle input that is longer than that.

Extra Credit (3 pts) Add a mechanism to allow your shell to handle arbitrarily long input.

1.11 Execution

In the previous assignment, you were only allowed to use `execve(2)`, but in this assignment, you will need to more easily access the programs on the system. As such, we will allow you to use `execvp(3)`. `execvp(3)` differs from `execve(2)` in two ways. First, `execvp(3)` will use the `PATH` environment variable to locate the program to be executed. That means the user of your shell will no longer need to provide a full path to the executable. Second, `execvp(3)` does not provide a means to set the environment.

1.12 Parsing

This assignment is not about writing a parser. As such, we have provided you with a parser on the course website, `token-shell.tgz`. In the gzipped tar-ball youll find a parser, `tokenizer.c`, a header file, `tokenizer.h`, and a sample program, `token-shell.c`. The tokenizer functions similar to `strtok(3)`, and you may use it freely. Although the tokenizer uses functions unspecified in this document, that does not imply you may use them in your shell. See the provided example program for more details on its functionality.

1.13 Terminal Signals

Your shell should never exit or stop because of the following terminal signals: `SIGTOU`, `SIGTIN`, `SIGTERM`, `SIGTSTP`, or `SIGINT`. As noted previously, you may need to relay some of these signals or use `tcsetpgrp(3)` so that signals are appropriately delivered. In a pinch, if youve wedged your shell, try sending a `SIGABRT` signal via `Ctrl-\` which should cause your shell to exit unconditionally.

2 Acceptable Library Functions

In this assignment, you may only use the following system calls and library functions**Extra Credit (1pt):** How can you tell the difference between system calls and library functions below without any additional information?:

- `execve(2)` or `execvp(3)`
- `fork(2)`
- `pause(2)`
- `wait(2)` or `waitpid(2)`
- `read(2)`, `write(2)` `printf(3)`, and `fprintf(3)`

- `signal(2)`, `sigaction(2)`, `sigprocmask (2)`, and `sigsuspend(2)`
- `kill(2)` and `killpg(2)`
- `exit(2)` or `exit(3)`
- `dup2(2)`
- `pipe(2)`
- `open(2)`
- `getpid(2)` and `getppid(2)`
- `setpgid(2)` and `getpgid(2)`
- `tcsetpgrp(3)`
- `malloc(3)` or `calloc(3)`
- `free(3)`
- `perror(3)`

Using any library function other than those specified above will adversely affect your mark on this assignment. In particular, if you use the `system(3)` library function, you will receive a **ZERO**.

3 Error Handling

All system call functions that you use will report errors via the return value. As a general rule, if the return value is less than zero, then an error has occurred and `errno` is set accordingly. You **must** check your error conditions and report errors. To expedite the error checking process, we will allow you to use the `perror(3)` library function. Although you are allowed to use `perror`, it does not mean that you should report errors with voluminous verbosity. Report fully but concisely.

4 Memory Errors

You are required to check your code for memory errors. This is a non-trivial task, but a very important one. Code that contains memory leaks and memory violations **will** have marks deducted. Fortunately, the `valgrind` tool can help you clean these issues up. It is installed inside the Linux VM distributed for the class. Remember that `valgrind`, while quite useful, is only a tool and not a solution. You must still find and fix any bugs that are located by `valgrind`, but there are no guarantees that it will find every memory error in your code: **especially** those that rely on user input.

5 Developing Your Code

The best way to develop your code is in Linux running on a virtual machine. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state. You may use the room 100 machines to run Linux within a Virtualbox environment, or run a VM on your own personal machine. The VM made available from the website should have all the tools that you need.

6 Submission Instructions

You **must** turn in the following:

1. A `README` file. The `README` will contain the following:
   - Your name and your parter's name if you work in a group of 2
   - A list of submitted source files
   - Extra credit answers (if any)
   - Compilation instructions
   - Overview of work accomplished
   - Description of code and code layout
   - General comments and anything that can help us grade your code
2. A `Makefile`. We should be able to compile your code by simply typing `make`. If you do not know what a `Makefile` is or how to write one, ask the admins, who covered it in a previous lab section, or consult one of the many online tutorials.
3. Your code. This is not a joke: people have been known to forget to include it.

Put all of your project files into their subdirectory called `hw2` and make sure that your `Makefile` has a "make clean" command to clear out any object files and other unneeded intermediate files. Run "make clean" on this subdirectory.

cd up a directory so that `hw1` is a subdirectory of your working directory, and run the following command to create your submission tarball, but replacing "UOEMAIL" with your cs.uoregon.edu email account name.

```
tar cvzf hw2_submission_UOEMAIL.tar.gz hw2
```

For example, since my UO CIS email account is "butler", I would run the command:

```
tar cvzf hw2_submission_butler.tar.gz hw2
```

Use the turnin script, located at `http://systems.cs.uoregon.edu/apps/turnin.cgi`, to submit your tarball. If you work in a group of 2, only one submission needs to be made for the group. Make sure both group members are identified in the README.

7 Grading Guidelines

This programming project will be graded as follows:

- 10% Documentation
- 10% Code design
- 20% Redirection

- 20% Two-Stage Pipeline
- 20% Foreground and Background Processes
- 20% Minimal Job Control

Note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, poor code organization, etc. Also, do not take the documentation lightly, as it provides those evaluating your project with a general roadmap of your code; without good documentation, it can be quite difficult to grade the implementation. Once again, these will be graded on machines running the Ubuntu 10 Linux distribution.

8 Attribution

This is a large and complex assignment, using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we do expect that you will be challenged with this assignment, and that you will learn about systems programming most effectively by *doing the work yourself/in your group*.

In order to avoid issues of plagiarism and cheating in this project, you must attribute any outside sources that you use. This includes both resources used to help you understand the concepts, and resources containing sample code that you may have borrowed as a template for your shell. The course textbook only needs to be cited in this latter case; all other sources must be attributed. You should use external code sparingly. For example, using most of an example from the man page for `pipe(2)` would be reasonable. Using a function such as `ParseInputandCreateJobandHandleSignals()` from a *Write Your Own Shell in Four Easy Steps* site would not be OK, either using text verbatim or closely following the structure for your own shell design. If you have technical concept questions, you should consider sending email to the mailing list. You should also email the admin list if you have questions.