# CS 439, Spring 2012

| | |
|---|---|
| Lab Assignment 1: | Fork/exec |
| Due: | Monday, Jan 30, 5:59PM |
| Lab Assignment 2a and 2b: | Signal handling, Unix Shell |
| Due: | Monday, Feb. 6, 5:59PM |

## Introduction

The purpose of this assignment is to become more familiar with the concepts of process control and signalling. You'll do this by writing a simple Unix shell program that supports job control.

## Logistics

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page or Piazza.

**Note:** This project will be graded on the UTCS public linux machines. Although you are welcome to do testing and development on any platform you like, we can not assist you in setting up other environments, and you must test and do final debugging on the UTCS public linux machines. The statement "It worked on my machine" will not be considered in the grading process.

## Hand Out Instructions

We provide a file shlab-handout.tar that contains a template for your program along with a number of useful helper functions. Get it from the class web page. E.g.,

```
unix> wget http://www.cs.utexas.edu/users/dahlin/Classes/439/labs/shlab-handout.tar
```

Put the file `shlab-handout.tar` to the protected directory (the *lab directory*) in which you plan to do your work. Then do the following:

- Type the command `tar xvf shlab-handout.tar` to expand the tarfile.

- Type the command `make` to compile and link some test routines.
- Type your team member names, ut eids at the top of the file `README`.

Looking at the `tsh.c` (*tiny shell*) file, you will see that it contains a functional skeleton of a simple Unix shell. To help you get started, we have already implemented the less interesting functions. Your assignment is to complete the remaining empty functions listed below. As a sanity check for you, we've listed the approximate number of lines of code for each of these functions in our reference solution (which includes lots of comments).

- `eval`: Main routine that parses and interprets the command line. [70 lines]
- `builtin_cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`. [25 lines]
- `do_bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
- `waitfg`: Waits for a foreground job to complete. [20 lines]
- `sigchld_handler`: Catches SIGCHILD signals. [80 lines]
- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [15 lines]
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [15 lines]

Each time you modify your `tsh.c` file, type `make` to recompile it. To run your shell, type `tsh` to the command line:

```
unix> ./tsh
tsh> [type commands to your shell here]
```

## General Overview of Unix Shells

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a *command line* on `stdin`, and then carries out some action, as directed by the contents of the command line.

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the child. The child processes created as a result of interpreting a single command line are known collectively as a *job*. In general, a job can consist of multiple child processes connected by Unix pipes.

If the command line ends with an ampersand ”&”, then the job runs in the *background*, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

For example, typing the command line

```
tsh> jobs
```

causes the shell to execute the built-in `jobs` command. Typing the command line

```
tsh> /bin/ls -l -d
```

runs the `ls` program in the foreground. By convention, the shell ensures that when the program begins executing its main routine

```
int main(int argc, char *argv[])
```

the `argc` and `argv` arguments have the following values:

- `argc == 3`,
- `argv[0] == ‘‘/bin/ls’’`,
- `argv[1]== ‘‘-l’’`,
- `argv[2]== ‘‘-d’’`.

Alternatively, typing the command line

```
tsh> /bin/ls -l -d &
```

runs the `ls` program in the background.

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Typing `ctrl-c` causes a SIGINT signal to be delivered to each process in the foreground job. The default action for SIGINT is to terminate the process. Similarly, typing `ctrl-z` causes a SIGTSTP signal to be delivered to each process in the foreground job. The default action for SIGTSTP is to place a process in the stopped state, where it remains until it is awakened by the receipt of a SIGCONT signal. Unix shells also provide various built-in commands that support job control. For example:

- `jobs`: List the running and stopped background jobs.
- `bg <job>`: Change a stopped background job to a running background job.
- `fg <job>`: Change a stopped or running background job to a running in the foreground.
- `kill <job>`: Terminate a job.

## Lab 1: Fork/exec

In this phase of the project, you will learn about the `fork` and `exec` system calls that you will use in the rest of the project.

### Part 1-1: Reading

Read every word of sections 3 and 4 of chapter 8 of Bryant and O'Hallaron.

Read every word of this handout before you write any code.

### Part 1-2: Fibonacci

Update `fib.c` so that if invoked on the command line with some integer argument $n$, it recursively computes the $n$th Fibonacci number ($n \leq 13$).

e.g.,

```
unix> fib 3
2
unix> fib 10
55
```

The trick is that each recursive call must be made by a new process, so you will call fork() and then have the new child process call `doFib()`.

The parent must wait for the child to complete and you need to figure out how to pass the result of the child's computation to its parent.

### Part 1-3: Fork/Exec

The fork system call creates a child process that is nearly identical to the parent. The exec call replaces the state of the currently running process with a new state to start running a new program in the current process.

Your job is to create a prototype for the shell you will be creating later. This prototype waits for a line of input. If the line is "quit", it exits. Otherwise, it parses the line and attempts to execute the program at the path specified by the first word with the arguments specified by the remaining words. It waits for that job to finish. Then it waits for the next line of input.

- The prompt should be the string "`psh> `".
- The command line typed by the user should consist of a `name` and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, then `psh` should handle it immediately and wait for the next command line. Otherwise, `psh` should assume that `name` is the path of an executable file, which it loads and runs in the context of a child process (In this context, the term *job*

refers to this child process). Your shell then waits for that job to finish. Then it waits for the next line of input.

All commands and jobs are executed in the foreground. In this phase you don't have to worry about background jobs. You also can assume that jobs execute until they exit; you don't need to worry about signal handling.

- Your shell should implement one built-in command: `quit`. If the user types `quit`, your shell should exit.

E.g.,

```
psh> /bin/ls -l -d
```

runs the `ls` program in the foreground.

We have provided psh.c, which provides framework for your shell, and util.h/util.c which provides some helper functions. Read these files.

Update the file psh.c by implementing the functions `eval()`, which the `main()` function calls to process 1 line of input, and `builtin_cmd()`, which your `eval()` function should call to parse and process the built-in `quit` command. (Later, you will extend the built-in command function to handle other built-in commands.)

## Lab 2a: Signal handling, Shell

### Part 2a-1: Reading

Read every word of section 5 of chapter 8 of Bryant and O'Hallaron. Examine the code for the `Signal()` function in `util.c`.

### Part 2a-2: Signal handling

Write a program in `handle.c` that first uses the `getpid()` system call to find its process ID, then prints that ID, and finally loops continuously, printing “Still here\n” once every second. Set up a signal handler so that if you hit ˆc (ctrl-c), the program prints “Nice try.\n” to the screen and continues to loop.

**Note:** The `printf()` function is technically unsafe to use in a signal handler. A safer way to print the message is to call

```
ssize_t bytes;
const int STDOUT = 1;
bytes = write(STDOUT, ‘‘Nice try.\ n’’, 10);
if(bytes != 10)
  exit(-999);
```

**Note:** You should use the `nanosleep()` library call rather than the `sleep()` call so that you can maintain your 1-second interval between "Still here" messages no matter how quickly the user hits ^c.

You can terminate this program using `kill -9`. For example, if the process ID is 4321

```
unix> kill -9 4321
```

### Part 2a-3: Signal sending

Update the program from Part 2-2 to catch the `SIGUSR1` signal, print "exiting", and exit with status equal to 1.

Now write a program `mykill.c` that takes a process ID as an argument and that sends the `SIGUSR1` signal to the specified process ID.

e.g.,

```
 unix> handle
 4321
 Still here
 Still here
 Still here                            unix> mykill 4321
 exiting                               unix>
 unix>
```

### Part 2a-4: Signal mechanics

If you compile a C program with the -S flag, the compiler produces the assembly language corresponding the the code it would generate for the program.

e.g.,

```
unix> gcc -S handle.c
unix> cat handle.S
...
```

Also, in the gdb debugger, you can see the assembly code for a function e.g.,

```
unix> gdb handle
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000100000970 <main+0>: push   %rbp
0x0000000100000971 <main+1>: mov    %rsp,%rbp
0x0000000100000974 <main+4>: push   %r12
0x0000000100000976 <main+6>: push   %rbx
...
```

In gdb, you can put a breakpoint for a function

```
(gdb) break main
Breakpoint 1 at 0x10000097b: file handle.c, line 30.
(gdb)
```

and you can `step` to the next C/C++ instruction or `stepi` to the next assembly instruction

```
(gdb) run
Starting program: /Users/dahlin/Classes/439/labs/shlab/src/handle

Breakpoint 1, main (argc=1, argv=0x7fff5fbff6e0) at handle.c:30
30   int pid = getpid();
(gdb) step
31   printf("%d\ n", pid);
(gdb) stepi
0x0000000100000989 31   printf("%d\ n", pid);
(gdb) stepi
0x000000010000098b 31   printf("%d\ n", pid);
(gdb)
```

Finally, you can tell GDB to pass a particular signal to your program

```
(gdb) handle SIGUSR1 pass
Signal        Stop Print Pass to program Description
SIGUSR1       Yes Yes Yes User defined signal 1
(gdb) handle SIGUSR1 nostop
Signal        Stop Print Pass to program Description
SIGUSR1       No Yes Yes User defined signal 1
(gdb)
```

In the file README, answer the following questions

1. What is the last assembly language instruction executed by the signal handler function that you write?
2. After the instruction just identified executes, what is the next assembly language instruction executed?
3. When the signal handler finishes running, it must restore all of the registers from the interrupted thread to exactly their values before the signal occurred. How is this done?

## Lab 2b: Shell

In this phase of the project, you will implement your simple shell, `tsh`.

Your `tsh` shell should have the following features:

- The prompt should be the string "`tsh> `".

- The command line typed by the user should consist of a `name` and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, then `tsh` should handle it immediately and wait for the next command line. Otherwise, `tsh` should assume that `name` is the path of an executable file, which it loads and runs in the context of an initial child process (In this context, the term *job* refers to this initial child process).
- `tsh` need not support pipes (`|`) or I/O redirection (`<` and `>`).
- Typing `ctrl-c` (`ctrl-z`) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.
- If the command line ends with an ampersand `&`, then `tsh` should run the job in the background. Otherwise, it should run the job in the foreground.
- Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer assigned by `tsh`. JIDs should be denoted on the command line by the prefix '`%`'. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.)
- `tsh` should support the following built-in commands:
  - The `quit` command terminates the shell.
  - The `jobs` command lists all background jobs.
  - The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
  - The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.
- `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `tsh` should recognize this event and print a message with the job's PID and a description of the offending signal.

## Checking Your Work

We have provided some tools to help you check your work.

**Reference solution.** The Linux executable `tshref` is the reference solution for the shell. Run this program to resolve any questions you have about how your shell should behave. *Your tsh shell should emit output that is identical to the reference solution* (except for PIDs, of course, which change from run to run).

**Shell driver.** The `sdriver.pl` program executes a shell as a child process, sends it commands and signals as directed by a *trace file*, and captures and displays the output from the shell.

Use the -h argument to find out the usage of `sdriver.pl`:

```
unix> ./sdriver.pl -h
```

```
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
Options:
  -h            Print this message
  -v            Be more verbose
  -t <trace>    Trace file
  -s <shell>    Shell program to test
  -a <args>     Shell arguments
  -g            Generate output for autograder
```

We have also provided 16 trace files (`trace{01-16}.txt`) that you will use in conjunction with the shell driver to test the correctness of your shell. The lower-numbered trace files do very simple tests, and the higher-numbered tests do more complicated tests.

You can run the shell driver on your shell using trace file `trace01.txt` (for instance) by typing:

```
unix> ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

(the `-a "-p"` argument tells your shell not to emit a prompt), or

```
unix> make test01
```

Similarly, to compare your result with the reference shell, you can run the trace driver on the reference shell by typing:

```
unix> ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
```

or

```
unix> make rtest01
```

For your reference, `tshref.out` gives the output of the reference solution on all races. This might be more convenient for you than manually running the shell driver on all trace files.

The neat thing about the trace files is that they generate the same output you would have gotten had you run your shell interactively (except for an initial comment that identifies the trace). For example:

```
bass> make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found.
tsh> ./myspin 10
Job (9721) terminated by signal 2
tsh> ./myspin 3 &
[1] (9723) ./myspin 3 &
tsh> ./myspin 4 &
```

```
[2] (9725) ./myspin 4 &
tsh> jobs
[1] (9723) Running    ./myspin 3 &
[2] (9725) Running    ./myspin 4 &
tsh> fg %1
Job [1] (9723) stopped by signal 20
tsh> jobs
[1] (9723) Stopped    ./myspin 3 &
[2] (9725) Running    ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (9723) ./myspin 3 &
tsh> jobs
[1] (9723) Running    ./myspin 3 &
[2] (9725) Running    ./myspin 4 &
tsh> fg %1
tsh> quit
bass>
```

Your solution shell will be tested for correctness on a Linux machine, using the same shell driver and trace files that were included in your lab directory. Your shell should produce **identical** output on these traces as the reference shell, with only two exceptions:

- The PIDs can (and will) be different.
- The output of the `/bin/ps` commands in `trace11.txt`, `trace12.txt`, and `trace13.txt` will be different from run to run. However, the running states of any `mysplit` processes in the output of the `/bin/ps` command should be identical.

### Hints

General hints

- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.
- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell. Stick with simple text-based programs such as `/bin/ls`, `/bin/ps`, and `/bin/echo`.

Hints for part 2b

- Use the trace files to guide the development of your shell. Starting with `trace01.txt`, make sure that your shell produces the *identical* output as the reference shell. Then move on to trace file `trace02.txt`, and so on.

- When you implement your signal handlers, be sure to send SIGINT and SIGTSTP signals to the entire foreground process group, using "-pid" instead of "pid" in the argument to the kill function. The sdriver.pl program tests for this error.

- One of the tricky parts of the assignment is deciding on the allocation of work between the waitfg and sigchld_handler functions. We recommend the following approach:

  - In waitfg, use a busy loop around the sleep function.
  - In sigchld_handler, use exactly one call to waitpid.

  While other solutions are possible, such as calling waitpid in both waitfg and sigchld_handler, these can be very confusing. It is simpler to do all reaping in the handler.

  Note that you probably can do something simpler for the prototype psh you build in part 1. Then, be ready to change how this works when you get to part 3.

- In eval, the parent must use sigprocmask to block SIGCHLD signals before it forks the child, and then unblock these signals, again using sigprocmask after it adds the child to the job list by calling addjob. Since children inherit the blocked vectors of their parents, the child must be sure to then unblock SIGCHLD signals before it execs the new program.

  The parent needs to block the SIGCHLD signals in this way in order to avoid the race condition where the child is reaped by sigchld_handler (and thus removed from the job list) *before* the parent calls addjob.

- When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the foreground process group. Since typing ctrl-c sends a SIGINT to every process in the foreground group, typing ctrl-c will send a SIGINT to your shell, as well as to every process that your shell created, which obviously isn't correct.

  Here is the workaround: After the fork, but before the execve, the child process should call setpgid(0, 0), which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type ctrl-c, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).

## Hand In Instructions

- Make sure you have included your names and UT-EIDs in the README.

- Create a tar file handin.tar that contains all of your source files, the README, and your Makefile.

- Use the turnin utility to submit your work

  E.g., for lab 1:

```
unix> turnin -submit [TA]  handin-439-shlab-1 handin.tar
```

for lab 2:

```
unix> turnin -submit [TA]  handin-439-shlab-2 handin.tar
```

- The makefile automates the handin process

  ```
  unix> make handin-1
  ```

  etc.

Good luck!