Home CS439

CS439: Principles of Computer Systems

Project 0

Due: 11:59p Friday, February 7, 2014

Projects will be submitted electronically.
Please refer to the turnin instructions at the end of this specification.

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.

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 and 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:

`msh> jobs` causes the shell to execute the built-in `jobs` command.

Typing the command line:

`msh> /bin/ls -l -d` runs the `ls` program in the foreground. By convention, the shell ensures that when this 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

`msh> /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. When a job is stopped or terminated, the user receives a message indicating the state of the process and then a command prompt.

Unix shells also provide various built-in commands that support job control. These commands include:

- `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.

Getting Started

We provide a file `shproj-handout.tar` that contains a template for your program along with a number of useful helper functions. Get it from the class web page. By either using the command:

```
unix> wget
http://www.cs.utexas.edu/~ans/classes/cs439/projects/shell_project/shproj-
handout.tar
```

or downloading in a browser.

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

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

   Once the command has finished expanding the archive, you should find the following files:

   ```
   Files:

   Makefile        # Compiles your shell program and runs the tests
   README          # This file
   msh.c           # A shell program that you will write and hand in
   mshref          # The reference shell binary.
   util.c/h        # Contains provided utilities
   jobs.c/h        # Contains job helper routines
   design_doc.txt  # Provide your answers to questions and explanations here

   #Files for Part 0
   fib.c           # Implement Fibonacci here
   psh.c           # Implement prototype shell here
   ```

```
#Files for Part 1
handle.c        # Implementation Needed
mykill.c        # Implementation Needed

# The remaining files are used to test your shell
sdriver.pl      # The trace-driven shell driver
trace*.txt      # The 16 trace files that control the shell driver
mshref.out      # Example output of the reference shell on all 16 traces

# Little C programs that are called by the trace files
myspin.c        # Takes argument  and spins for  seconds
mysplit.c       # Forks a child that spins for  seconds
mystop.c        # Spins for  seconds and sends SIGTSTP to itself
myint.c         # Spins for  seconds and sends SIGINT to itself
```

2. Type the command `make` to compile and link some test routines.
3. Type your team member names, UT EIDs, CS logins, and email addresses at the top of the file design_doc.txt.

Part 0: 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 0.1: Reading

Read every word of sections 3 and 4 of chapter 8 of Bryant and O'Hallaron. Also, read every word of this handout before you write any code.

Part 0.2: Fibonacci

Update fib.c so that if invoked on the command line with some integer argument `n`, where `n` is less than or equal to 13, it recursively computes the nth Fibonacci number. (The numbers are counted from 0.)

Example executions:

```
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 the child must pass the result of its computation to its parent.

You may modify `doFib()`, but you may not modify the number of parameters it accepts or its return value.

Part 0.3: 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 of the shell you will be creating in Part 2.

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

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. Your shell waits for that job to finish, and then it waits for the next line of input. (In this context, the term *job* refers to this child process.)

  For example,

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

  should run the `ls` program in the foreground.
- Your shell should implement one built-in command: `quit`. If the user types `quit`, your shell should exit.
- For now, all commands and jobs are executed in the foreground. You also can assume that jobs execute until they exit; you don’t need to worry about signal handling (again, for now).

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

Hints for Part 0: fork/exec

- The `waitpid`, `fork`, and `exec*` functions will come in very handy. Use `man` to learn about them. (Remember, you can use `man man` to learn about `man`!)
- The `WEXITSTATUS` macro described in the `waitpid` man page may also be useful.
- Recall that C does not have strings. Read more about string handling in C in these notes.

Part 1: Signal Handling

Part 1.1: 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 1.2: 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 `^c` (`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
```

Since `handle` has control of your current terminal window, you'll need to execute the `kill` command from another window on the same machine.

Part 1.3: Signal Sending

Update the program from Part 1.2 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.

For example:

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

The stub code is provided for you.

Part 1.4: Signal Mechanics

In this section, you will take a look at how signals are implemented by inspecting the assembly code associated with your program and with the signal call. There are questions in the design document for you to answer regarding your program's behavior in the presence of signals.

If you compile a C program with the `-S` flag, the compiler outputs the assembly language corresponding the the code it would generate for the program. You can then use `cat` to view the output.

For example:

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

In the gdb debugger, you can see the assembly code for a function. First, recompile your program using the following commands:

```
gcc -Wall -g   -c -o handle.o handle.c
gcc -Wall -g handle.o util.o -o handle
```

Then execute `gdb` as follows:

```
unix> gdb handle
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000100000970 : push %rbp
0x0000000100000971 : mov %rsp,%rbp
0x0000000100000974 : push %r12
```

```
0x0000000100000976 : push %rbx
...
```

In gdb, you can set a *breakpoint* for a function. A breakpoint indicates a place where execution should stop so that you can inspect its state.

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

You can then `step` to the next C/C++ instruction or `stepi` to the next assembly instruction:

```
(gdb) run
Starting program: /u/ans/classes/cs439/projects/shproj/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)
```

Now, use gdb to pass a particular signal to your program so that you can understand what happens. You can tell gdb to pass a particular signal to your program this way:

```
(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)
```

Note that `handle` here is a command to `gdb>` and not a reference to your program of the same name. Also note that it *passes* a signal sent to `gdb` on to the user process. You'll need to send a signal from outside `gdb`. You can read more about signal handling in `gdb` in this gdb documentation.

Use these techniques to answer the questions in the design document. relating to signal mechanics (Part 1).

Part 2: Shell

In this phase of the project, you will implement a simple shell, `msh`. Your msh shell should have the following features:

- The prompt should be the string “msh> ”.
- The command line typed by the user should consist of a path to an executable file, `name`, and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, then `msh` should handle it immediately and wait for the next command line. Otherwise, `msh` should assume that `name` is the path of an executable file, which it loads and runs in the context of an initial child process.
- Typing `ctrl-c` or `ctrl-z` should cause a SIGINT or SIGTSTP signal, whichever is appropriate, to be sent to the current foreground job, *as well as any descendants 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 `msh` 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 `msh`. 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.
- `msh` 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.
- `msh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `msh` should recognize this event and print a message with the job's PID and a description of the offending signal.
- msh need not support pipes (|) or I/O redirection (< and >).
- If a process is stopped or terminated by a signal, `msh` should print that information to the screen. Such as:

```
Job [1] (26961) terminated by signal 2
Job [1] (26963) stopped by signal 20
```

Looking at the msh.c (*mini 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 msh.c file, type `make` to recompile it. To run your shell, type msh to the command line:

```
unix> ./msh

msh> [type commands to your shell here]
```

Hints for Part 2: Shell

- 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.

- As in other portions of this project, the `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will be useful. 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`.

- 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 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 uses `addjob` to add the child to the job list. 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 the behavior we want.

    Here is the workaround: After the fork, but before the exec, 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).

Checking Your Work

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

Reference solution. The Linux executable `mshref`, which was included in the tarball you downloaded, is the reference solution for the shell. Run this program to resolve any questions you have about how your shell should behave. *Your msh 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 should 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 ./msh -a "-p"
```

The `-a "-p"` argument tells your shell not to emit a prompt. You could accomplish the same task with this command:

```
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 ./mshref -a "-p"
```

or

```
unix> make rtest01
```

For your reference, mshref.out gives the output of the reference solution for all traces.

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:

```
unix> make test15
./sdriver.pl -t trace15.txt -s ./msh -a "-p"
#
# trace15.txt - Putting it all together
# msh> ./bogus
./bogus: Command not found.
msh> ./myspin 10
Job (9721) terminated by signal 2
msh> ./myspin 3 &
[1] (9723) ./myspin 3 &
msh> ./myspin 4 &
[2] (9725) ./myspin 4 &
msh> jobs
[1] (9723) Running ./myspin 3 &
[2] (9725) Running ./myspin 4 &
msh> fg %1
Job [1] (9723) stopped by signal 20
msh> jobs
[1] (9723) Stopped
[2] (9725) Running
msh> bg %3
%3: No such job
msh> bg %1
[1] (9723) ./myspin 3 &
msh> jobs
[1] (9723) Running ./myspin 3 &
[2] (9725) Running ./myspin 4 &
```

```
msh> fg %1
msh> quit
unix>
```

For tests 11 through 13, your output does not need to (and very likely won't) match exactly. For your program to be correct, the state(s) of `mysplit` in your output must match exactly.

For tests 14 and 15, the PIDs listed in your output will likely differ from those in the provided output. As long as that is all that differs, your code will still be considered correct.

On Programming and Logistics

The following guidelines should help smooth the process of delivering your project. You can help us (and yourself!) a great deal by observing the following:

General

1. You **must** work in two-person teams on this project. Failure to do so will result in a 0 for the project. Once you have selected a partner, exchange first and last names, EIDs, and CS logins. Also, fill out the README distributed with the project. You must follow the pair programming guidelines set forth for this class.

   Please see the Grading Criteria to understand how failure to follow the pair programming guidelines OR fill out the README will affect your grade.

2. You must follow the guidelines laid out in the C Style Guide or your will lose points. This includes selecting reasonable names for your files and variables.
3. 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 cannot 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.
4. Your solution shell will be tested for correctness using the same shell driver and trace files that are included in your project 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.
5. Your code **must** compile without any additions or adjustments, or you will receive a 0 for the correctness portion.
6. You are encouraged to reuse *your own* code that you might have developed in previous courses to handle things such as queues, sorting, etc.
7. You are also encouraged to use code provided by a public library such as the GNU library.
8. You may not look at the written work of any student other than your partner. This includes, for example, looking at another student's screen to help them debug, looking at another student's print-out, working with another student to sketch a high-level design on a white-board. See the syllabus for additional details.
9. If you find that the problem is under specified, please make reasonable assumptions and document them in the README file. Any clarifications or revisions to the assignment will be posted to Piazza.

Submitting Your Work

1. After you finish your code, please use `make turnin` to submit your code. *Only one member of the pair should run make turnin.* Make sure you have included the necessary information in the README.
2. Once you have completed your design document (**each student should complete one individually**), please submit it using the following command:

```
turnin -submit ccoleman shell_design design_doc.txt
```

Make sure you have included your name and UT EID in the design document and be certain you have named it correctly.

Grading

Code will be evaluated based on its correctness, clarity, and elegance according to the criteria here. Strive for simplicity. Think before you code.

The most important factor in grading your code will be code inspection and evaluation of the descriptions in the write-ups. Remember, if your code does not follow the standards, it is wrong. If your code is not clear and easy to understand, it is wrong.

The second most important factor in grading your code will be an analysis its correctness.