CS350 Operating Systems Winter 2022

Midterm Exam

1 Introduction

There are two components to the CS350 Midterm Exam for Winter 2022. Read these instructions clearly.

Quiz 3 - Midterm Quiz - The quiz will be on Learn. It will be open for a 24 hour window. The quiz will have restrictions on the duration. The enforced time will be 40 minutes.The material covered in this quiz will be:

- Processes
- System Calls and Interrupts
- Threads
- Concurrency
- Synchronization
- Scheduling

The midterm quiz will be **available on March 11 starting at 12:01 am until March 11 11:59 pm**. You should see the availability of the quiz on Learn.

`userspace` Programming Assignment - The midterm porgramming assignment is to implement a shell. The progrmaming assignment will be in `C`. The shell you implement will be similar to the one you implemented in Assignment 1-`userspace`. The midterm programming assignment is **due on April 5th at 5pm, no further extensions**. The shell you implement will be similar to the one you implemented in Assignment 1 - `userspace`. In this midterm specification, we will provide:

- explicit implementation requirements for the midterm programming assignment
- starter code - with a `Makefile` and a stub of a `main` function
- instructions for submitting the midterm programming assignment

You are required to complete your implementation in the `main.c` file provided.

Some general advice for this assignment:

- **Start early.** The instructions are detailed, but even debugging simple mistakes are time consuming. This holds doubly true if you are not especially familiar with `C`.
- **Compile often.** By checking whether the code compiles after small modifications, you will be able to pinpoint problems very quickly.

2 Implementation Requirements

In this assignment, you will implement a shell. We will call its executable `myshell`. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

Your shell is an interactive loop that

- repeatedly prints a prompt "> ", note that the prompt is the ">" followed by a space
- parses the input
- executes the command specified on that line of input
- waits for the command to finish

This is repeated until the user types `exit`.

You should structure your shell such that it creates a new process for each new command **(there are a few exceptions to this, which we will discuss below).** There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously.

Your shell should be able to parse a command, and run the program corresponding to the command. For example, if the user types `ls -la /tmp` , your shell should run the program `ls` with all the given arguments and prints the output on the screen.

Note that the shell itself does not "implement" `ls` or really many other commands at all. All it does is find those executables and create a new process to run them. The maximum length of a line of input to the shell is 512 bytes (excluding the carriage return).

2.1 Built-in Commands

Whenever your shell accepts a command, it should **check whether the command is a built-in command or not**. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your `C` program.In this project, you must implement the shell with the following built-in commands:

`cd` - change directory. You do not have to support tilde `~`.

`pwd` - display the present working directory

`wait` - should only allow `myshell` to continue when all background jobs have completed, if any

`exit` - terminate the `myshell` program

`help` - list the built-in commands provided in `myshell`

The formats for these commands are as follows:

```
[optionalSpace]cd[optionalSpace]
[optionalSpace]cd[oneOrMoreSpace]dir[optionalSpace]
[optionalSpace]pwd[optionalSpace]
[optionalSpace]wait[optionalSpace]
[optionalSpace]exit[optionalSpace]
[optionalSpace]help[optionalSpace]
```

Hints: When you run `cd` (without arguments), your shell should change the working directory to the path stored in the `$HOME` environment variable. Use `getenv("HOME")` to obtain this.

You do not have to support tilde `~`.

Therefore, when a user types `pwd`, you simply call `getcwd()`. When a user changes the current working directory (e.g. "cd somepath"), you simply call `chdir()`. Hence, if you run your shell, and then run `pwd` it should look like this:

```
cs350-mt% ./myshell
> pwd
/cs350/w22/midterm/code
> cd ../
> pwd
/cs350/w22/midterm
> exit
cs350-mt%
```

2.2 Background Jobs

In Linux, a shell can run processes in the background. In this way, you can run multiple jobs concurrently. In most shells, this is implemented by letting you put a job in the "background". This is done as follows:

```
> ls &
```

By typing a trailing ampersand(`&`), the shell knows you just want to launch the job, but not to wait for it to complete. Thus, you can start more than one job by repeatedly using the trailing ampersand.

```
> ls &
> ps &
```

Of course, sometimes you will want to wait for jobs to complete. To do this, in your shell you will simply type `wait` . The built-in command `wait` should not return until **all** background jobs are completed. In this example, `wait` will not return until the both jobs (`ls`, `ps`) are finished.

```
mysh> ls &
mysh> ps &
mysh> wait
```

You are **not** required to implement the following:

- The commands `bg` or `fg` to restore background jobs to the foreground,
- the pipe (`|`) operation,or
- support for quotes and /in shell input.

2.3 Program Errors

The one and only error message. You should print this one and only error message whenever you encounter an error of any type:

```
char error_message[30] = "An error has occurred\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

The error message should be printed to stderr (standard error). Also, do not add whitespaces or tabs or extra error messages.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this assignment. If the syntax of the command looks perfect, you simply run the specified program. If there is any program-related errors (e.g. invalid arguments to `ls` when you run it, for example), let the program prints its specific error messages in any manner it desires (e.g. could be stdout or stderr).

2.4 Batch Mode

So far, you have run the shell in interactive mode. In the interactive mode, you display a prompt and the user of the shell will type in one or more commands at the prompt.

You are now required to support batch mode. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode.

In batch mode, you should not display a prompt. You should print each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). To print the command line, do not use `printf` because `printf` will buffer the string in the `C` library and will not work as expected when you perform automated testing. To print the command line, use `write(STDOUT_FILENO, ...)` this way:

```
write(STDOUT_FILENO, cmdline, strlen(cmdline));
```

In both interactive and batch mode, your shell should terminates when it sees the `exit` command on a line or reaches the end of the input stream (i.e., the end of the batch file).

To run the batch mode, your `C` program must be invoked exactly as follows:

```
> ./myshell [batchFile]
```

The command line arguments to your shell are to be interpreted as follows:

batchFile: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batchFile for commands to be executed. If not present or readable, you should print the one and only error message as specified above.

Implementing the batch mode should be very straightforward if your shell code is nicely structured. The batch file basically contains the same exact lines that you would have typed interactively in your shell. For example, if in the interactive mode, you test your program with these inputs:

```
% ./myshell
> ls
some output printed here
> ls > /tmp/ls-out
some output printed here
> notACommand
some error printed here
```

then you could cut your testing time by putting the same input lines to a batch file. For example myBatchFile could be:

```
ls
ls > /tmp/ls-out
notACommand
```

and run your shell in batch mode:

```
% ./myshell myBatchFile
```

In this example, the output of the batch mode should look like this:

```
ls
some output printed here
ls > /tmp/ls-out
some output printed here
notACommand
some error printed here
```

3 Submitting Your Work

To submit your work, you must use the `cs350_submit` program in the `linux.student.cs` computing environment.

Important! You must use `cs350_submit`, not `submit`, to submit your work for CS350.

Note the usage for `cs350_submit` command is as follows

```
% usage: cs350_submit <assign_dir> <assign_num_type>
```

The `assign_dir` is the path to the root directory that should contain the `C` file for your minishell. You are required to use the single `C` file with no headers, that is provided as `main.c` in the `mt_starter` code.

The `assign_num_type` for this midterm programming assignment is `MIDTERM`.

Therefore, to run the `cs350_submit` command for submitting the midterm programming assignment, the command will look like this:

```
% cs350_submit cs350-student/mt MIDTERM
```

The argument `assign_dir` in the `cs350_submit` command, packages up your `userspace` program and submits it to the course account using the regular `submit` command.

This assignment only briefly summarizes what `cs350_submit` does. Look carefully at the output from `cs350_submit`.

It is a good idea to run the `cs350_submit` command like this:

```
cs350_submit cs350-student/mt MIDTERM | tee submitlog.txt
```

This will run the `cs350_submit` command and also save a copy of all of the output into a file called `submitlog.txt`, which you can inspect if there are problems. This is handy when there is more than a screen full of output.

You may submit multiple times. Each submission completely replaces any previous submissions that you may have made for this assignment.