CST 334: Operating Systems
Dr. Glenn Bruns

Homework - Week 2

Important note! All homework is to be done on your own.

1. **Reading**. Read OSTEP chapter 4 and answer the following questions by editing file hw2.txt, which is attached to the homework assignment on iLearn.

○ What is the meaning of the 'Blocked' process state described in Chapter 4?
a) a process is blocked because other processes are running
b) a process is blocked while an I/O operation completes
c) a process is blocked because the memory it needs is not available

○ In Figure 4.4 of Chapter 4, the state of process$_1$ is changed by the operating system from Ready to Running after time = 3, when process0 initiates I/O. Why was process$_1$ changed from Ready to Running?
a) process0 had been running for a while
b) process1 wasn't ready to run until time = 4
c) process0 was waiting for an I/O operation to complete

○ While a program runs, its instructions are stored on disk (not memory), but its data is stored in memory.
a) true
b) false

○ Is the bash shell part of the Linux kernel?
a) yes
b) no

○ Fig. 4.5 shows a data structure that is a much-simplified version of the one that Linux uses to store info about each process. When does the OS use these register values?

a) when the process is running

b) when the process is blocked

c) when the process changes from the blocked state to the running state

2. **Bash puzzle.** I've created a puzzle for you. The following is output from a bash session.

```
$ ls
week2-test
$ cd week2-test
$ ls
a  foo.txt
$ ls .
```

```
a  foo.txt
$ cd a
$ ls ..
a  foo.txt
$ pwd
/cygdrive/e/Glenn/CSUMB/fall17/os/week2-test/a
$ cd a
$ ls
a  foo.txt
$ cat foo.txt
stuff
$ cat ../../foo.txt
some stuff
$ cd a
$ ls
$ cp ../foo.txt bar.txt
$ cp bar.txt ../../foo.txt
$ cd ../../..
$ pwd
/cygdrive/e/Glenn/CSUMB/fall17/os/week2-test
```

It's not easy to see what's happening because many files and directories have the same names.

I want you to look at the sequence of commands, and try to figure out the situation after all the commands have been executed.

The questions below give relative pathnames. They are relative to the current working directory after all the commands have been executed.

a. how many .txt files in the current working directory?
b. is it possible that a file a/bar.txt exists?
c. does file foo.txt exist?
d. does file a/a/bar.txt exist?
e. what is the content of file a/foo.txt?
f. was file a/foo.txt created during the execution of the commands?
g. what is the content of file a/a/foo.txt?
h. was file a/a/foo.txt created during the execution of the commands?
i. what is the content of the file a/a/a/bar.txt?
j. was file a/a/a/bar.txt created during the execution of the commands?

Provide your answers by editing hw2.txt.

3. **C Programming**.

In the next few homework assignments you will build a simplified version of the bash shell. Remember that bash is a command interpreter. The code for a command interpreter usually has this form:

```
initialize
```

```
while (true) {
    print prompt
    accept user input
    process input
}
```

This week, your code should do the following:

- ○ prompt for input with “msh> “ (‘msh’ stands for “my shell”)
- ○ read the user input
- ○ if the user input is ‘exit’ or ctrl-d, terminate
- ○ otherwise, simply echo the user input

You should be able to handle user input of at least 120 characters. You should print the error message "error: input too long" and re-prompt if more characters input than you can handle.

Here is an example of what your msh code should do: ($ is the bash prompt)

```
$ ./msh
msh> abc
abc
msh>

msh> 123
123
msh> 123 abc
123 abc
msh> ***
***
msh> exit
$
```

(In the second test, a space character was entered.) Your program should respond sensibly on other inputs, too.

If you’re not sure how to get started, you can follow these steps:

- ○ Create a subdirectory of your home directory named ‘msh’. (for “my shell”)
- ○ Copy /home/CLASSES/brunsglenn/cst334/hw/hw2/hello.c to your msh directory.
- ○ Compile and run hello.c.
    - ■ to compile: `$ gcc -o hello hello.c`
    - ■ to run: `$ ./hello`

○ Rename hello.c to msh.c (use the 'mv' command).

○ Modify msh.c so that, instead of "hello, world!", it prints "msh> ". Compile and run. (Note the space after '>'.)

○ Add code right after the printf statement so that you read in keyboard input from the user. Use function 'fgets' to get user input (not 'scanf'). Also, modify your program so that, after it gets the user input, the input is printed. Test your program by trying different kinds of user input.

○ Add code right after displaying the user input to check whether the user input is ctrl-d or string "exit". Compile and test again.

○ Put your code within a loop so that you:

  ■ prompt for input with "msh> "

  ■ read the input that the user types

  ■ if the user types "exit", or types ctrl-d, exit the program, but otherwise report what the user typed (as in part f), and then prompt for new input.

The hardest part about this homework is figuring out how to use function 'fgets'. Read the man page. Key points about fgets:

- The return value of fgets will allow you to know if ctrl-d was entered

- The end-of-line ('newline') character of the user input will be read by fgets.

- If the user gave too much input then the newline character won't be read by fgets.

- If the user gave too much input, then fgets will not read it all, and the remaining part will be read by the next fgets call. So if the user gave too much input, you need to take care of that before prompting again.

Testing your code. On mlc104, the directory `/home/CLASSES/brunsglenn/cst334/hw/hw2` contains five test files test1.sh, test2.sh, …, test5.sh. Copy these to the directory where you developed your file msh.c. Each test should give exit status 0, like this:

```
$ ./test1.sh
$ echo $?
0
```

You need to run test1.sh first, as it will compile your code and produce binary file 'msh' that is used by the other tests. The directory above also contains files 'Makefile' and 'run-all.sh', which you can look at and play with if you want.

4. **Peerwise** Spend about an hour on Peerwise. You are expected to contribute a question at least once every couple of weeks. Also, answer, rate, and comment on other students' questions.

Submission : Submit your homework on iLearn as two files: hw2.txt for parts 1 and 2, and msh.c for part 3.

Grading. The homework is worth 100 points: 20 points for part 1 (4 points/question), 30 points for part 2 (3 points/question), and 50 points for part 3. For part 3, 40 points for functionality and 10 points for tidy code. For functionality, I will run 4 automated tests on your code. For tidy code, I will look for proper commenting, indentation, and use of white space. Grading for your peerwise contributions is separate.