CS330-001: Assignment 2

Fall 2019 (201930)

#### Due Date and Time: Monday, September 30, 2019 at 4:20 PM

1. **(100 marks)** In this problem, you are required to modify your toyshell program from Assignment 1.

**PART 1**

Your toyshell program must provide services to support the following additional commands and features:

**Note**: The commands are shown in uppercase font for emphasis only, you don’t have to program them that way. Similarly, the parameters to the commands are enclosed between the symbols < and > for emphasis, but the symbols are not actually part of the parameters and should not be programmed that way.

1. *Comments*: A comment is any sequence of characters on the command line following a $ symbol. The examples given below show how comments might appear on the command line:

$

$ This command is just a comment.

ls $

ls $ The ls command followed by a comment.

cp file\_a file\_b $ Copy a to b.

Essentially, ignore the $ symbol and anything following it on the command line.

1. <*Linux\_command*> *re-implementation*: In Assignment 1, commands not recognized by toyshell were assumed to be Linux commands and executed using the Linux system function. Re-implement the execution of Linux commands using the fork, execve, and wait/waitpid system calls instead of the Linux system function. As done in Assignment 1, toyshell should wait for each command to complete before prompting the user for the next command (unless the command is followed by the - character to indicate background execution, as described below). Use the third argument to execve to provide the environment to a Linux command (see man 2 execve for a more complete explanation). The environment is available in a global variable called environ. To use this variable, you need to specify where it can be found. So, you should probably include the line extern char \*\*environ; somewhere in your main .h or .cpp file. To determine whether <*Linux\_command*> can be executed, use the access function to check each directory in the PATH environment variable for the existence of a file called <*Linux\_command*> (see man 2 access for a more complete explanation). However, you will first have to extract the value of the PATH environment variable from environ. Use the getenv function (see man 2 getenv for a more complete explanation). If the file is found, execute it using execve. If the file is not found, notify the user with an informative message.
2. *Background execution*: If a Linux command is followed by the - character, execute the command in background. This feature is similar to following a command by the & character when using tcsh. That is, do not wait for a command to complete executing before prompting the user for the next command. In addition, as each background job is created, add its process ID to a job table (i.e., a table of all jobs currently executing in background). The job table should also include other useful information: a job ID (i.e., a sequential number assigned to a new background job at the time it is created), the command entered at the command line, and the time the background job was created.
3. BACKJOBS: A command that prints a list of background jobs and their current status. This feature is similar to the jobs command when using tcsh. The current status (i.e., *in progress* or *completed*) of each job can be determined by using the waitpid system call (see man waitpid for more information). The calls that report an error will be those that have already completed, so their status will be “completed”. The calls that do not result in an error will have a status of “in progress”.
4. FRONTJOB <*job ID*>: A command for moving a background job to the foreground (this is similar to the fg command used in tcsh).
5. COND ( <*numeric expression*> ) <*command*>: A command that conditionally executes another command. COND executes the command <*command*> if the value of <*numeric expression*> evaluates to true. Otherwise, the command <*command*> is not executed. The value of <*numeric expression*> is the result of checking the status of a file. Valid status checks are CHECKE <*file\_name*>, CHECKD <*file\_name*>, CHECKR <*file\_name*>, CHECKW <*file\_name*>, and CHECKX <*file\_name*>, which correspond to checking whether the file <*file\_name*> exists, is a directory, is readable, is writable, and is executable, respectively. The command <*command*> is any command that toyshell can execute (including both built-in and Linux commands). For example, the command

COND ( CHECKR somefile1 ) cat somefile1

determines whether the file somefile1 is readable, and if it is, executes the command cat somefile1. Similarly, the command

COND ( CHECKE somefile2 ) cp somefile2 somefile1

determines whether the file somefile2 exists, and if it does, executes the command cp somefile2 somefile1. To determine a file’s status, use the stat system call (see man 2 stat for a more complete explanation).

1. NOTCOND ( <*numeric expression*> ) <*command*>: A command that conditionally executes another command. NOTCOND executes the command <*command*> if the value of <*numeric expression*> evaluates to false. Otherwise, the command <*command*> is not executed. See the COND command above for more on evaluating <*numeric expression*>. The command <*command*> is any command that toyshell can execute (including both built-in and Linux commands). For example, the command

NOTCOND ( CHECKE somefile1 ) cp somefile2 somefile1

determines whether the file somefile1 exists, and if it doesn’t, executes the command cp somefile2 somefile1. Similarly, the command

NOTCOND ( CHECKD somefile2 ) rm somefile2

determines whether the file somefile2 is a directory, and if it isn’t, executes the command rm somefile2. To determine a file’s status, use the stat system call (see man 2 stat for a more complete explanation).

1. *One other new feature*: Include one other feature of your choosing that you would find useful in your shell. Please discuss this with your instructor before proceeding to implementation.
2. *More error handling*: Your approach should effectively identify and recover from errors. For example, bad input and/or the inability to execute a command should not cause your toyshell to crash. To get credit for this component of the assignment, you must be able to demonstrate that you can handle at least 15 unique errors. The errors must be different from those you handled in Assignment 1.

**Note**: You must handle all the built-in commands with exactly the same syntax as shown above.

**PART 2 (45 marks)**

For programming problems, the Results are worth 70%. So, for this problem, the results are worth 70 marks out of the 100 marks available. Demonstrate that your toyshell works and that it can handle all of the requirements described in Part 1, except for the last two (i.e., your new feature and the error handling), as you will demonstrate these in Parts 3 and 4, respectively. So, this part will be worth 45 marks out of the 70 marks available. Each correctly executed command will be worth one mark. Your demonstration should be captured in a script file.

1: $ This is a comment – it is treated like a blank line.

2: $ This is also a comment. (*type* 10 *spaces before entering the command*)

3:  *press* Enter/Return *with a blank command line*

4: sleep 1000 - $ Start a background process.

5: sleep 15 - $ And another.

6: BACKJOBS $ Should show both of the above processes.

7: cp /home/venus/hilder/cs330/assignment2/datafiles/alpha file1

8: BACKJOBS $ Might still show a sleep process running.

9: $ Enter all comments as shown above. Enter all comments

10: $ shown below, as well.

11: cat file1 $ file1 should be there.

12: sleep 1000 -

13: BACKJOBS

14: BACKJOBS

15: sleep 5

16: sleep 1000 –

17: BACKJOBS

18: sleep 20 -

19: FRONTJOB <*the job number from the previous sleep command*>

20: who

21: NEWNAME mycopy cp

22: NEWNAME dog cat

23: SAVENEWNAMES myaliases

24: why

25: mycopy myaliases file2

26: NEWNAME mycopy rm

27: ls -

28: mycopy file2

29: ls

30: SAVENEWNAMES myaliases

31: when -

32: NEWNAME mycopy cp

33: READNEWNAMES myaliases

34: NEWNAMES

35: COND ( CHECKR myaliases ) dog myaliases -

36: mkdir junkdir

37: COND ( CHECKD junkdir ) rmdir junkdir

38: cp myaliases mysavedaliases

39: NOTCOND ( CHECKD myaliases ) mycopy myaliases

40: COND ( CHECKE mysavedaliases ) NOTCOND ( CHECKE myaliases ) cp mysavedaliases myaliases

41: BACKJOBS

42: *Kill any remaining background jobs before quitting*

43: BACKJOBS

44: BACKJOBS

45: STOP

**PART 3 (10 marks)**

Demonstrate your new feature. This will be worth ten marks out of the 70 marks available. Your demonstration should be captured in a script file.

**PART 4 (15 marks)**

Finally, demonstrate that your toyshell can handle more errors (i.e., the last item in Part 1). This will be worth 10 marks out of the 70 marks available. In order to receive all 15 marks, you must be able to handle at least 15 more unique errors. Each correctly handled error condition will be worth one mark. Your demonstration should be captured in a script file.

2. **(9 marks)** As main memory is allocated to, and de-allocated from, running processes, the sizes of the available “holes” (i.e., partitions) in main memory is constantly changing, leading to the dynamic storage allocation problem. Namely, how to best allocate space to processes in main memory from the available partitions. Given the initial state of three instances of main memory shown below, where processes have been allocated space as indicated, and the first-fit, best-fit, and worst-fit allocation schemes are utilized, respectively, show the final state of main memory on each instance after allocating space to processes requiring 17KB, 28KB, and 9KB (in the order listed).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| First-fit |  | Best-fit |  | Worst-fit | Partition Size |
| *allocated* |  |  |  |  | ← 20KB |
| *allocated* |  | *allocated* |  | *allocated* | ← 8KB |
|  |  | *allocated* |  |  | ← 12KB |
| *allocated* |  | *allocated* |  | *allocated* | ← 8KB |
|  |  |  |  |  | ← 40KB |
| *allocated* |  | *allocated* |  | *allocated* | ← 10KB |
|  |  |  |  |  | ← 30KB |

3. **(5 marks)** Assume a main memory with five fixed-size blocks of 16KB as shown below, where space is allocated to processes contiguously, and a process can span multiple blocks. Given these assumptions, the maximum size of a process is 80KB (i.e., limited by the size of main memory). Given the initial state of main memory shown below, where four processes have been allocated space as indicated:

|  |  |  |
| --- | --- | --- |
| Block | Initial State | Space Allocated |
| 1 | *allocated* | ← 10 of 16KB |
| 2 |  | ← 0 of 16KB |
| 3 | *allocated* | ← 16 of 16KB |
| 4 | *allocated* | ← 6 of 16KB |
| 5 | *allocated* | ← 6 of 16KB |

1. **(3 marks**) Are there any blocks in which fragmentation has occurred? If so, indicate the type of fragmentation that has occurred for each block.
2. **(2 marks)** Assume that another process is loaded and requires 20KB of space in main memory. Can the process be allocated any space given the initial state of the main memory? If so, show where it will be placed. If not, explain why not?

4. **(10 marks)** Consider an address translation scheme that utilizes a page table to map logical addresses in a process to physical addresses in main memory. The logical address space consists of 256 pages, where each page contains 4,096 words. The page table consists of 256 entries. The logical address space is mapped onto a physical address space of 1,024 frames. **You must justify your answers.**

1. **(2 marks)** How many bits are there in a logical address?
2. **(2 marks)** How many bits are there in a physical address?
3. **(2 marks)** How many bits are there in a page table entry?
4. **(4 marks)** If the CPU has just referenced the 13th word on the 54th page of the logical address space and the 54th page is currently in the 39th frame, what are the values of the logical address, the page table entry, and the physical address?

**WHAT TO SUBMIT**

*If you are working alone*, submit to UR Courses: (1) all your source code files (i.e., *only* the .cpp and .h files) zipped into a single file called cppandhfiles, (2) a single script file called part2script showing the compilation and execution of your toyshell from Part 2, (3) a single script file called part3script showing the compilation and execution of the new feature of your toyshell from Part 3, (4) a single script file called part4script showing the compilation and error handling capabilities of your toyshell from Part 4, and (5) your solutions for problems 2, 3, and 4 zipped into a single file called solutions.

*If you are working with a partner*, *one of the partners* should submit to UR Courses: (1) a file named partners that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%), (2) all your source code files (i.e., *only* the .cpp and .h files) zipped into a single file called cppandhfiles, (3) a single script file called part2script showing the compilation and execution of your toyshell from Part 2, (4) a single script file called part3script showing the compilation and execution of the new feature of your toyshell from Part 3, (4) a single script file called part4script showing the compilation and error handling capabilities of your toyshell from Part 4, and (6) your solutions for problems 2, 3, and 4 zipped into a single file called solutions. *The other partner* should submit to UR Courses: (1) a file named partners that provides the names and student numbers of the partners and the relative contributions of the partners (should total 100%). Note that *both* partners need to submit the partners file.