Computer Architecture and Technology Area

Universidad Carlos III de Madrid



OPERATING SYSTEMS

Programming Assignment 3. Multithread. Manufacturing process control.

**BACHELOR'S DEGREE IN COMPUTER SCIENCE AND**

**ENGINEERING**

Year 2016/2017

AUTHORS:

Ryan Cohane (100370386)

Carine Torres (100370405)

**Index**

1. Description of Code ………………………………………………………………………………… 3
   1. factory\_manager……….………………………………………………………… 3
   2. process\_manager ………...……...……………………………………………… 4
   3. queue …..…………………………………………..………………………………… 5
2. Description of Tests …...…………………………………………………………………………… 6
3. factory\_manager ……………………………………………………………...… 6
4. process\_manager ………………………………………..……..……………….. 7
5. queue ……………………………………………..……………….………...…......... 8
6. Conclusions …...……………………………………………………………..………………………. 15

**1. Description of Code**

**factory\_manager**

**process\_manager**

**queue**

**2. Description of Tests**

The program, **msh**, was intended to prompt a minishell that functions over the UNIX/Linux Operating System like any other shell, capable of executing commands like ls, cat, etc., as well as any sequence of commands (up to 3 commands).

**2.1.** *Simple commands with valid arguments*

Evidently, a command entered in the minishell should thus accomplish the same effect as that command entered in the built-in shell of any UNIX /Linux Operating System. Thus, our first test of **msh** included a comparison of the results of running the program with simple commands like **ls, rm file,** etc. to the results of running simple commands on the actual shell. We derived expectations for our minishell from running the simple commands on the actual shell. Then we restored the original state of all associated directories and files affected by those commands so that the minishell could run the same exact set of commands from the same exact directory / file state, and found that the actual results for our minishell matched, as documented below:

|  |  |  |
| --- | --- | --- |
| **Simple Command** | **Expected Result**  **(Actual Shell)** | **Actual Result**  **(Minishell)** |
| **ls** | 18 files/directories within ssoo\_p2\_msh directory listed | 18 files/directories within ssoo\_p2\_msh directory listed (in same order) |
| **rm** file | file deleted from ssoo\_p2\_msh directory | file deleted from ssoo\_p2\_msh directory |
| **cp** file newdir | file copied from ssoo\_2\_msh directory to newdir | file copied from ssoo\_2\_msh directory to newdir |
| **mv** file2 newdir | file2 moved from ssoo\_2\_msh directory to newdir | file2 moved from ssoo\_2\_msh directory to newdir |

**2.2.** *Command sequences with valid arguments*

Our second test of **msh** was whether it knew how to handle a command sequence of up to 3 commands (all paired with valid arguments if any). Our test of **msh** included a comparison of the results of running the program with command sequences like **ls | sort, ls | sort | wc -l,** etc. to the results of running those command sequences on the actual shell. We derived expectations for our minishell from running the command sequences on the actual shell. Then we restored the original state of all associated directories and files affected by those commands so that the minishell could run the same exact set of command sequences from the same exact directory / file state, and found that the actual results for our minishell matched, as documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result**  **(Actual Shell)** | **Actual Result**  **(Minishell)** |
| **ls | sort** | 18 files/directories within ssoo\_p2\_msh directory listed in ascending alphabetical order | 18 files/directories within ssoo\_p2\_msh directory listed in ascending alphabetical order |
| **ls | sort | wc -l** | The number of files / directories within ssoo\_p2\_msh directory (18 at runtime) | The number of files / directories within ssoo\_p2\_msh directory (18 at runtime) |

**2.3.** *Commands with invalid arguments*

One edge case to test is whether or not **msh** knows how to handle commands given invalid arguments. This is important because the minishell should not be incapable of handling invalid arguments, meaning, it should not crash, nor hang, but merely print the error statement for what is wrong with the arguments, and merely re-prompt msh> to take in the next command. This is because the attempted execution of the command will take care of handling the error by printing the command-specific error statement for what was wrong with the arguments given to the command. Running test commands with faulty arguments on the actual shell helped us derive our expectations for our minishell. The expected results and actual results are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **rm** imaginary  \*Goal: to confirm that a command-specific error arises when trying to manipulate a nonexistent file, though should not crash system either | rm error printed to standard output indicating the nonexistence of imaginary | Error printed to standard output:  rm: imaginary: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **ls** baddir | sort  \*Goal: to confirm that a command-specific error arises when trying to manipulate a file with no access permissions (set by **chmod 000 baddir**), though should not crash system either  \*Goal: to confirm error in a command sequence handled appropriately (i.e. following commands don’t then continue to execute if prior command had invalid arguments) | ls error printed to standard output indicating the lack of permissions with baddir | Error printed to standard output:  ls: baddir: Permission denied  …followed by msh> prompt for next command because shouldn’t crash minishell (neither should it print an additional error for sort, as expected, because sort should never have been reached if ls could not properly execute) |

**2.4.** *Commands with execution error*

Another edge case to test was whether or not **msh** could handle properly catching execution errors within the child process attempting to execute an invalid command. What should happen is that the execution should fail, meaning the child should return -1 to the parent upon the child exiting. In turn, perror would be triggered, thus outputting an “Error in exec:” statement followed by the actual execution error (set in errno). Because this execution error only happens in the child process, the parent should not crash (because it appropriately leaves the child responsible for execution and “waits” for the child’s response), and continue to run the minishell by prompting the next command with msh> even after an failed attempted execution. We caused an error in execution by trying to execute a nonexistent command. Running test commands with faulty commands on the actual shell helped us derive our expectations for our minishell. The expected results and actual results are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **lsa**  \*Goal: to confirm that an execution error arises for invalid simple commands (e.g. a simple command that doesn’t exist), though should not crash system either | perror triggers, indicated by “Exec in error:” statement printed to standard output, followed by error statement indicating nonexistence of the file path containing the command | Execution error printed to standard output:  Error in exec: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **lsa | sorty**  \*Goal: to confirm that execution errors will arise for invalid command sequences too, though should not crash system either | perror triggers per command in sequence that is invalid, indicated by “Exec in error:” followed by error statement indicating nonexistence of the file path containing the command (should be 2 statements for 2 invalid commands) | Execution error printed to standard output:  Error in exec: No such file or directory  Error in exec: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell |

**2.5.** *Commands in background*

Another test we did was to ensure that commands expected to run in the background (indicated by commands ending in &) did so, meaning they did not block the parent minishell process from continued execution even while these background commands were still running in their child processes. This is because if background command is indicated, our program changes behavior from blocking parent execution until child exits to waiting on the child to exit without hanging (a functionality obtained by passing the WNOHANG option as a parameter to the waitpid() function). Running test commands in the background on the actual shell helped us derive our expectations for our minishell. The expected results and actual results from our tests are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **ls &** | 18 files/directories within ssoo\_p2\_msh directory listed, though interspersed with msh> (the prompt for the next command) and a [PID] statement for the number of the child process running the background command | 18 files/directories within ssoo\_p2\_msh directory listed, though interspersed with msh> (the prompt for the next command) and a [PID] statement for the number of the child process running the background command |
| **ls | sort &**  \*Goal: to confirm that command sequences in the background also work, specifically, printing the PID of the last command of the sequence, not all of the child processes’ PIDs | 18 files/directories within ssoo\_p2\_msh directory listed in alphabetic order, though interspersed with msh> (the prompt for the next command) and a [PID] statement for the number of the child process running the last background command in the sequence | 18 files/directories within ssoo\_p2\_msh directory listed in alphabetic order, though interspersed with msh> (the prompt for the next command) and a [PID] statement for the number of the child process running the last background command in the sequence |

**2.6.** *Commands with valid redirection*

Another test was to determine whether **msh** could appropriately redirect standard input, standard output, and standard error to files specified in commands and command sequences. This should work correctly because in our program, we specify a behavior such that if we are at the first command in a sequence, if an input file is indeed specified, we close standard input—close(0)—and open the specified input file, assigning it to a file descriptor, fd0, which then gets assigned to be also referenced by the file descriptor for standard input via dup(fd0) because the current lowest available fd it can be copied to is 0. Similarly, we specify a behavior such that if we are at the last command in a sequence, if an output file is indeed specified, we close standard output—close(1)—and open the specified output file, assigning it to a file descriptor, fd1, which then gets assigned to be also referenced by the file descriptor for standard input via dup(fd1) because the current lowest available fd it can be copied to is 1. And again, for standard output, we specify a behavior such that for any command, if an error file is indeed specified, we close standard error—close(2)—and open the specified error file, assigning it to a file descriptor, fd2, which then gets assigned to be also referenced by the file descriptor for standard input via dup(fd2) because the current lowest available fd it can be copied to is 2. The expected results and actual results are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **cat < file**  \*Goal: to confirm that redirection from standard input to specified input file works | Writes content of file to standard output | Writes content of file to standard output |
| **cat < file | sort > file2**  \*Goal: to confirm that redirection from standard output to specified output file works  \*Goal: to confirm that redirection works with command sequences | Writes sorted content of file to file2 | Writes sorted content of file to file2 |
| **ls imaginary >& file3**  \*Goal: to confirm that redirection from standard error to specified error file works (will produce an error with this command because imaginary is not an existing directory) | Writes error from command (ls: imaginary: No such file or directory ) to file3, and does not write error to standard error (i.e. console) | Writes error from command (ls: imaginary: No such file or directory ) to file3, and does not write error to standard error (i.e. console) |

**2.7.** *Commands with invalid redirection*

Another edge case to test was to make sure **msh** catches redirection errors. It should not allow continuation of execution of commands that use files / directories it cannot access (because of permission rights or nonexistence). We would expect it to catch these redirection errors and halt execution of that command because we programmed **msh** to check for errors when opening any specified directories or files and if an error exits, trigger perror and exit the child process. The expected results and actual results from our tests are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **cat < imaginary**  \*Goal: to confirm that redirection from nonexistent file stops execution of command and displays error | Open file error regarding lack of existence of imaginary, immediately followed by msh> prompt for next command | Error written to standard output: Can't open file: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **ls | sort > badfile**  \*Goal: to confirm that redirection to file with no rwx permissions (set by **chmod 000 badfile**) stops execution of command and displays error  \*Goal: to confirm that failed redirection is handled with command sequences | Permission denied error regarding badfile, immediately followed by msh> prompt for next command | Error written to standard output: Can't open file: Permission denied  …followed by msh> prompt for next command because shouldn’t crash minishell |

**2.8.** *Internal commands: mycalc*

Another test for msh was whether or not it could handle internal commands, starting with mycalc. We designed our program to compare the name of all entered commands with mycalc and if so, then rather than go through the same process as built-in commands (where a child would be forked from the parent and execvp would be run), it would pass the arguments of the command to our mycalc function for execution. The mycalc function would then ensure that there were 2 operands and 1 operator (using our helper function numParameters to ensure that there were 3 arguments after mycalc), or else result in the structure error being output. If there were 2 operands, it would make sure that the operands were integers with an isInt helper function we wrote that uses isDigit() and checks for any negative signs. If there was an operator, it would check that the operator was only either “add” or “mod.” If these weren’t the case a structure error would be output. Otherwise, the command would execute by actually doing the mathematical calculations. The tests we ran included tests for functionality given valid operator and operands, functionality of accumulated sum, and functionality of error handling with invalid operator and/or operands. Expectations and results are documented below:

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **mycalc 3 add -7**  \*Goal: to confirm mycalc functionality with valid add statement | [OK] 3 + -7 = -4; Acc -4 | [OK] 3 + -7 = -4; Acc -4 |
| **mycalc 5 add 12**  \*Goal: to confirm mycalc functionality of accumulated sum with valid add statement (after the above) | [OK] 5 + 12 = 17; Acc 13 | [OK] 5 + 12 = 17; Acc 13 |
| **mycalc 10 mod 7**  \*Goal: to confirm mycalc functionality with valid mod statement | [OK] 10 % 7 = 7 \* 1 + 3 | [OK] 10 % 7 = 7 \* 1 + 3 |
| **mycalc 10 % 7**  \*Goal: to confirm mycalc functionality of error handling when operator is not of the type “add” or “mod” (i.e. error statement printed, and no crashing of the system) | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **mycalc 10**  \*Goal: to confirm mycalc functionality of error handling when incorrect number of arguments (i.e. error statement printed, and no crashing of the system) | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **mycalc s mod s**  \*Goal: to confirm mycalc functionality of error handling when non-integer operands (i.e. error statement printed, and no crashing of the system) | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] The structure of the command is <operand 1> <add/mod> <operand 2>  …followed by msh> prompt for next command because shouldn’t crash minishell |

**2.8.** *Internal commands: mybak*

Another test for msh was whether or not it could handle internal commands, starting with mybak. We designed our program to compare the name of all entered commands with mybak and if so, then rather than go through the same process as built-in commands (where a child would be forked from the parent and execvp would be run), it would pass the arguments of the command to our mycak function for execution. The mycak function would then ensure that there were 2 arguments passed after mybak (using our helper function numParameters), or else result in the structure error being output. If there were 2 arguments, it would make sure that the first was a file that could be opened, and if not output an error indicating the file to be copied was not openable. It would also make sure that the second argument was a directory that could be opened and modified, and if not output an error indicating that the copied file could not be opened (which would be the case if there was no directory that it could successfully be created / opened in). Otherwise, it would make the copy using read and write system calls and then indicate so with a success print statement. We ran tests for functionality of mybak with valid file and directory, functionality of error handling given bad file and/or directory, functionality given already existing file in the directory to copy to, and functionality given incorrect number of arguments.

|  |  |  |
| --- | --- | --- |
| **Command sequence** | **Expected Result** | **Actual Result**  **(Minishell)** |
| **mybak msh.c newdir**  \*Goal: to confirm mybak functionality with valid file and directory | [OK] Copy has been successful between msh.c to the directory newdir | [OK] Copy has been successful between msh.c to the directory newdir |
| **mybak imaginary newdir**  \*Goal: to confirm mybak functionality of error handling with invalid file—file that doesn’t exist (i.e. error statement printed, and no crashing of the system) | [ERROR] Error opening original file: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] Error opening original file: No such file or directory  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **mybak msh.c baddir**  \*Goal: to confirm mycalc functionality of error handling with invalid directory—directory that doesn’t have right rwx permissions (i.e. error statement printed, and no crashing of the system) | [ERROR] Error opening the copied file: Permission denied  …followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] Error opening the copied file: Permission denied  …followed by msh> prompt for next command because shouldn’t crash minishell |
| **mybak msh.c newdir**  \*Goal: to confirm mycalc functionality of error handling when file to copy already exists in directory to copy into (i.e. here, newdir already contains a file called msh.c) | [OK] Copy has been successful between msh.c to the directory newdir  …newdir/msh.c will have none of its original contents and copy of ./msh.c | [OK] Copy has been successful between msh.c to the directory newdir  …newdir/msh.c has none of its original contents and has copy of ./msh.c |
| **mybak origin.txt**  \*Goal: to confirm mycalc functionality of error handling when incorrect number of arguments (i.e. error statement printed, and no crashing of the system) | [ERROR] The structure of the command is mybak <original file> <output directory>…followed by msh> prompt for next command because shouldn’t crash minishell | [ERROR] The structure of the command is mybak <original file> <output directory>…followed by msh> prompt for next command because shouldn’t crash minishell |