**Programming Assignment 1**

***cseshell*,** not your ordinary shell

*50.005 Computer System Engineering*

**Due date: 14 March 23:59 (Week 7)**

[Outline](#_pckct1z1y98f)

[Getting Started](#_ayjir8e06gjo)

[Grading -- [100 points] (10% of total grade)](#_knltmoc6mpjv)

[Submission Procedure](#_rrx7joq8cqav)

[Download Materials, Check](#_h7g91urisbo0)

[Understanding shell.h](#_klfpbhi8nbkb)

[Understanding how shell works](#_9c5iun3cmsxt)

[Task 1: Implement shellReadLine function](#_uu8wqmr4ygqy)

[Task 2: Implement shellTokenizeInput](#_d8wzujavpt6i)

[Task 3: Implement shellExecuteInput](#_xpq7ypc7gl6h)

[Task 4: Complete the rest of the shell functions.](#_5lbka9h3x34l)

[Task 5: Looping the Shell](#_6vtar7fhizks)

[Task 6: Implement countline](#_c6byvljm90co)

[Task 7: Implement summond](#_9tw7s8lnu7ov)

[Task 8: Implement checkdaemon](#_mwv4zwclixbd)

[Summary](#_escref35rwls)

## Outline

In this assignment, you are tasked to create **a shell as well as a daemon process**, both of which are the common applications of fork(). **WARNING: please start as EARLY as possible. The length of this guide is 38 pages.**

The assignment is written entirely in C. At the end of this assignment, you should be able to:

* Create a shell and wait for user input
* Write several other custom programs that can be invoked by the shell
* Parse user input and invoke fork() with the appropriate program
* Create a program that results in a daemon process
* Use your shell to keep track the state of your daemon processes

## Getting Started

Download the starter code using git into your preferred working directory:

git clone https://github.com/natalieagus/ProgrammingAssignment1.git

If your distribution doesn’t come with git, [install](https://linuxize.com/post/how-to-install-git-on-ubuntu-18-04/) it first.

**Closely follow the instructions** given in this handout. You are only required to **modify** the starter code and header files. **DO NOT create your own script for submission.**

## Grading -- [100 points] (10% of total grade)

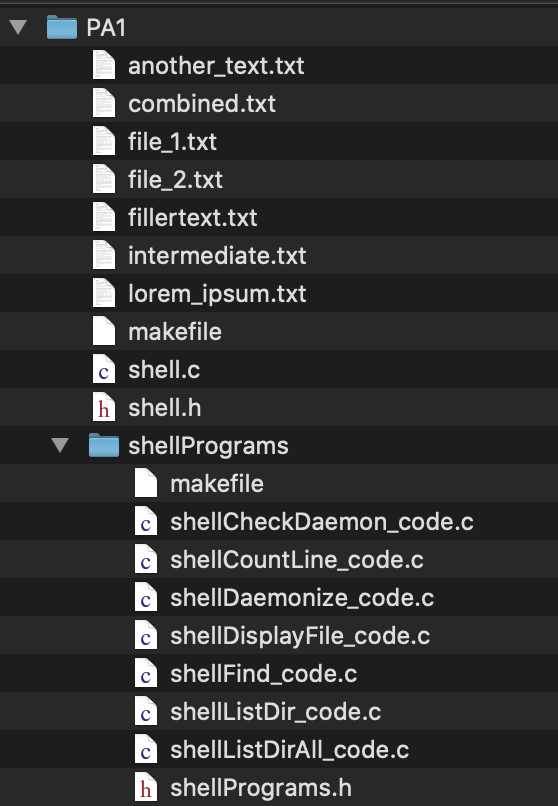
There are two main parts to this assignment, a **coding** part and a **quiz** part. The coding part is divided into 8 tasks:

1. Task 1 - 5: completing implementation of the shell **(40 pts)**
2. Task 6: implementing custom program countline **(10 pts)**
3. Task 7: implementing custom program summond **(20 pts)**
4. Task 8: implementing custom program checkdaemon **(15 pts)**

In the spirit of preventing us from copy-pasting ready-made code online, the quiz **(15 pts)** part aims to test you on your knowledge on each part of the coding task above. It consists of MCQ, short answers, matching, etc, and it will test your knowledge on the basic implementation of C functions that you have to use to complete the assignment. It will be an open **book**, administered online through edimension [**see course handout Week 9 row**].

## 

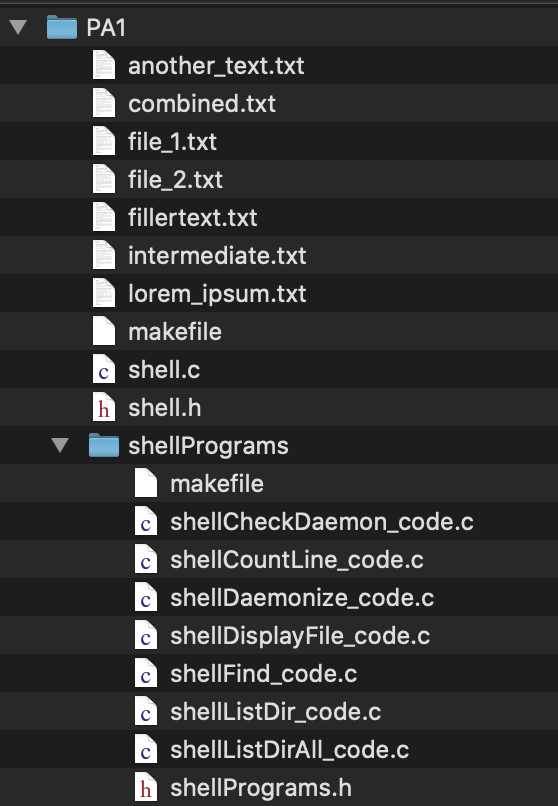
## Submission Procedure

1. **You may do this assignment in pairs,** but you can do it solo as well should you choose to take up the challenge alone. [Indicate your pair in this sheet](https://docs.google.com/spreadsheets/d/1AJKX7zpFAdiT7QVWt3jOqjrmz7rLfSnEmfXQ7Uick6s/edit?usp=sharing). **If you are doing it alone, fill up NIL on the Student 2 column.**
2. Zip back the folder containing the modified starter code with your answer and upload it to @csesubmitbot telegram bot using the command /submitpa1. **CHECK** your submission by using the command /checksubmission. **Any one person can submit.**
   1. **DO NOT modify ANY makefile**
   2. **DO NOT create more scripts as part of your answer. You are allowed to only modify the given scripts.** To speed up grading, your scripts will be run through an automatic grader, so whatever new file you are adding will not be considered by the grader. The screenshot on the left contains all the files that will be considered for grading. 
   3. **DO NOT modify ANY of the original functions in any header file shell.h and shellPrograms.h file: not the arguments, not the names, just don’t modify any of these.**
   4. **DO NOT import more c libraries. The imported libraries are all the libraries that you can use.** Let us know if you face any issue using the stated libraries.
3. **PERSONAL CHECKOFF REQUIRED:** Go to this checkoff sheet to book **your checkoff slot** **[TBC]** (you and your pair must both be present). You will be required to **demonstrate** the features of your shell. You **need to go for checkoff** to receive any grade for this assignment.

## Download Materials, Check

Download the starter code using git into your preferred working directory:

git clone https://github.com/natalieagus/ProgrammingAssignment1.git and you should see that you have the following files given to you:



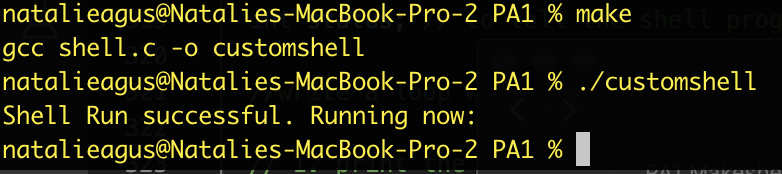
Notice that there’s another folder called shellPrograms inside the PA1 folder. This folder contains the scripts for custom programs of which the shell can run.

The shell script is called shell.c with header file shell.h, both are inside the PA1 folder.

There are *two* makefile scripts, one in each folder. The purpose of these makefiles are for you to conveniently recompile all the scripts within the folders whenever you make changes on them[[1]](#footnote-0).

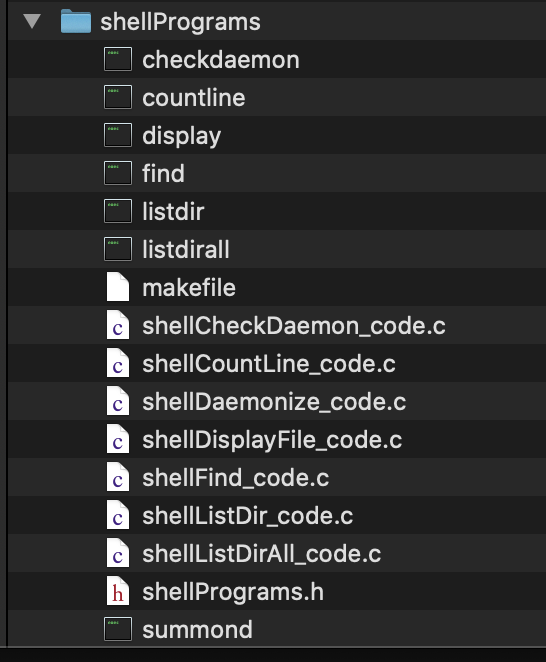
Now let’s **check** these programs first before we dive into the assignment. Follow the steps below:

1. Open terminal and change directory to this PA1 folder
2. Type make and you shall have a new binary called customshell
3. Run the binary by typing ./customshell, you should see the following output:



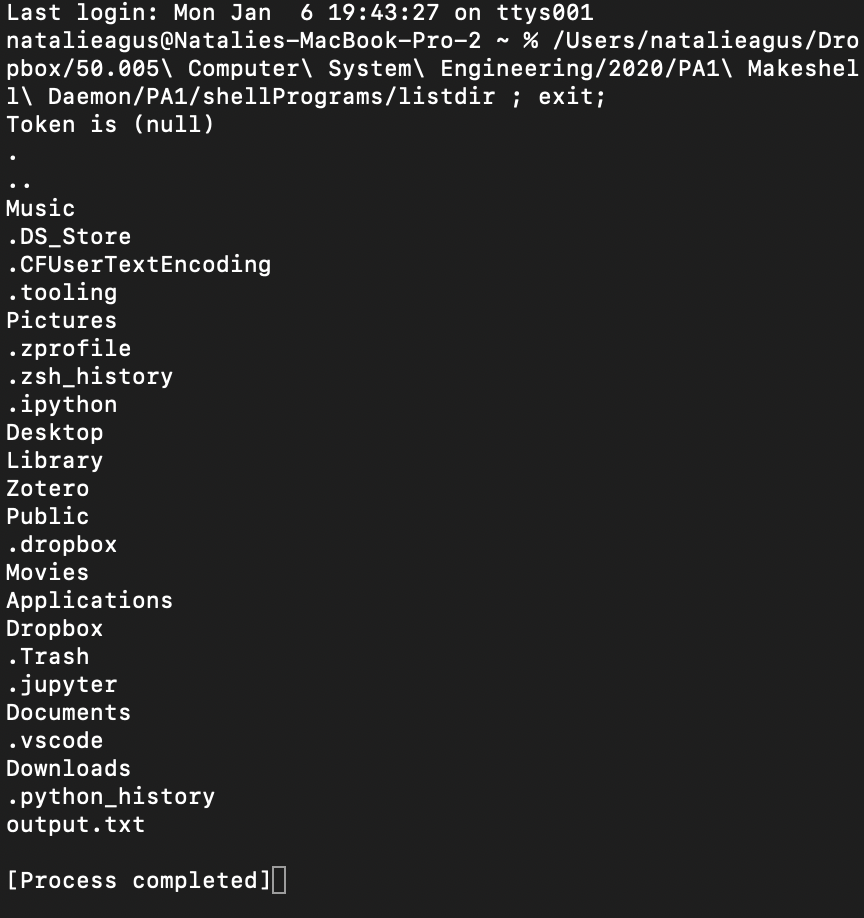
Of course there’s no *shell* yet. The print out is just to ensure that the program compiles and runs successfully before we begin the assignment.

1. Change directory to shellPrograms folder
2. Type make and you shall have several new binaries as follows:



These are all your “custom programs” that will be run with your shell given a particular input command. Right now: listdir, listdirall, find, and display are implemented for you. On top of completing the shell, you will implement the other three: checkdaemon, countline, and summond in this assignment.

1. Try executing listdir (command: ./listdir). You should find that it is a program whose task is to list all the contents of your home or root folder (in my case, it is the home folder). This is because listdir is a program that lists all the documents in the current directory, the location of current is depending on the process who calls it.



This is analogous to the command ls installed in your system that you can use to list files in the current directory.

**Recap**: ls is also the name of a system program that is invoked by the shell (bash / zsh) when you type the command ls. Typically, the system program ls is installed in /bin or /usr/bin depending on your machine.

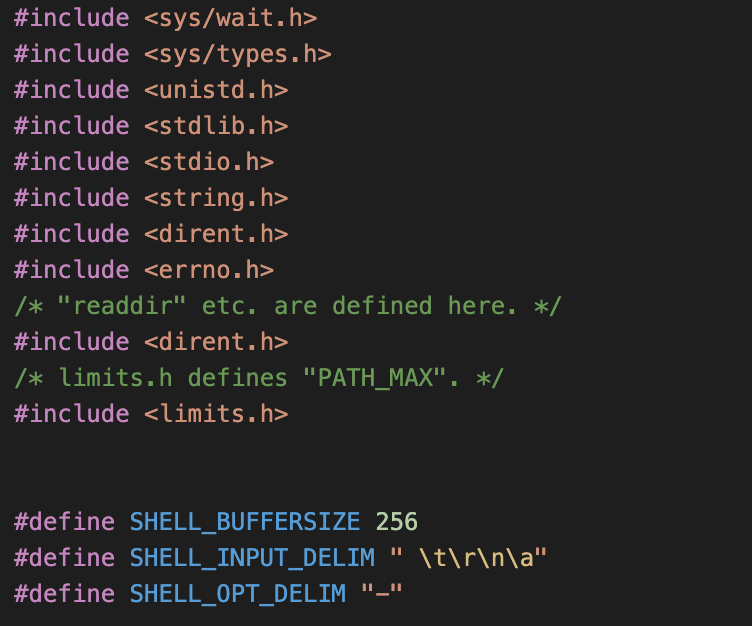
If all is well, congratulations you have finished setting up all the required starter code. You may proceed to Part 2. **DO NOT change any of these makefiles**. Your answer should be compilable with the original makefile. Programs with compilation error will **receive zero.**

## Understanding shell.h

Both files: shell.c and shell.h are the files containing the code for your shell program. The header file (shell.h) contains function declarations, imports, and macro definitions, while the .c file contains the implementation for these functions.

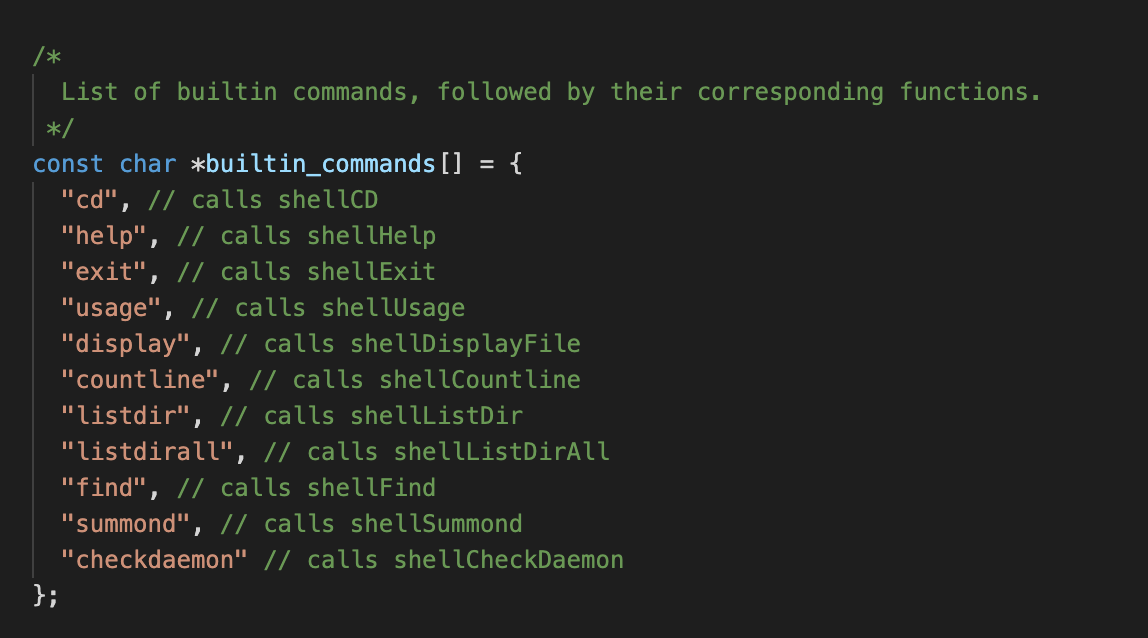
**Let's go through the header file first.**

1. Open shell.h and you shall see the following in the first few lines:



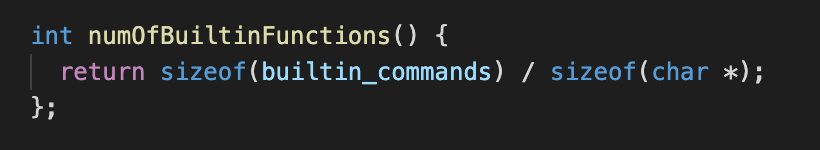
These are libraries and macro definitions. You may leave them as is.

1. Next, we have these ***array of pointers*** builtin\_commands global constant that stores the strings of commands that the user can type into the shell.

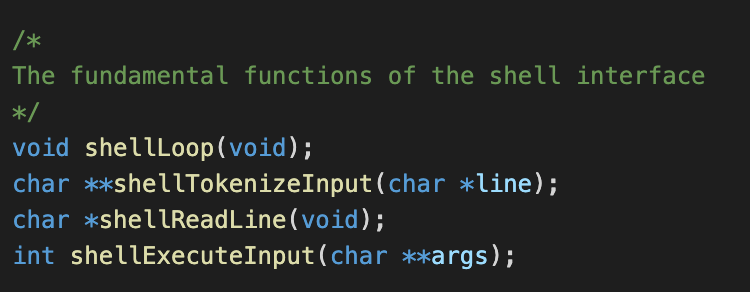


**Note**: builtin\_commands[0] will point to “cd”, builtin\_commands[1] will point to “help”, and so on. Formally, we can think of builtin\_commands as the array of addresses that points to the address of the first character in each command string.

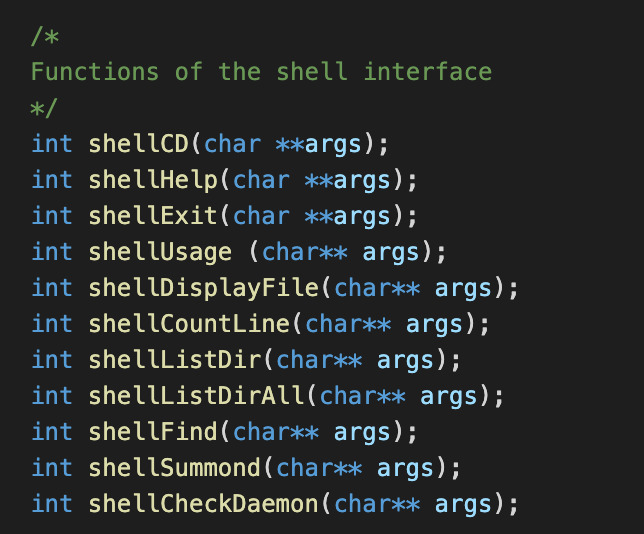
Below it there’s a convenient and very simple function made for you to return how many commands are there that the shell can accept. Right now, there’s 11 of them.



1. Next, is the function declarations of the shell that you will implement in shell.c:



1. Finally, the function declarations for the shell commands that you will also implement in shell.c:



**You are free to declare further functions in shell.h, but:**

1. **DO NOT modify ANY of these original functions: not the arguments, not the names.**
2. **DO NOT import more libraries. These are all the libraries that you can use.**
3. **DO NOT create more scripts as part of your answer. You are allowed to only modify the given scripts.**

## Understanding how shell works

The shell works very simply in these steps:

1. The main function in shell.c invokes shellLoop(void)
2. The function shellLoop(void) -- as the name suggests, never returns. It continuously loops to:
   * **Fetch one line of user input** from stdin using shellReadLIne function
   * **Parse** and **tokenize** user input using shellTokenizeInput function,
   * and then **execute** the appropriate system program inside shellPrograms folder depending on the command given using shellExecuteInput function
3. Of course you need to code a way to *terminate* the shell, i,e: jumps out of the loop when a user type exit onto the terminal

## Task 1: Implement shellReadLine function

Your first task is to implement shellReadLine function.



**Implementation notes:**

1. You should use malloc to allocate a **persistent memory space** (even after calling function exits) that contains the user command. malloc will return a pointer of the type that you allocate the memory space for, for example:

int \*buffer= malloc(sizeof(int) \* 10);

buffer[0] = 1;

buffer[1] = 9;

buffer[2] = 8;

// other work...

// after you are done with array

free(buffer)

allocates a memory that can contain 10 integers. Afterwards, we can start assigning values to the memory block. **The pointer array persists until you free them using free.**

2. In this function, you need to fetch user input using:

getline(&buffer, &size, stdin) where:

1. &buffer is the address of the pointer returned by malloc
2. &size is the address of constant that contains the size of the buffer

You can read more on how to use getline here: <http://man7.org/linux/man-pages/man3/getline.3.html>

You don’t need to actually worry if the input is larger than the size allocated by \*buffer. This is because getline will automatically realloc pointer if buffer overflows. If \*buffer was initially NULL, then getline will also automatically allocate a buffer before storing the line. Don’t forget to free this buffer after you are done with it.

**To test:**

1. Replace the main() function with the following:

**int main(int argc, char \*\*argv)**

**{**

**printf("Shell Run successful. Running now: \n");**

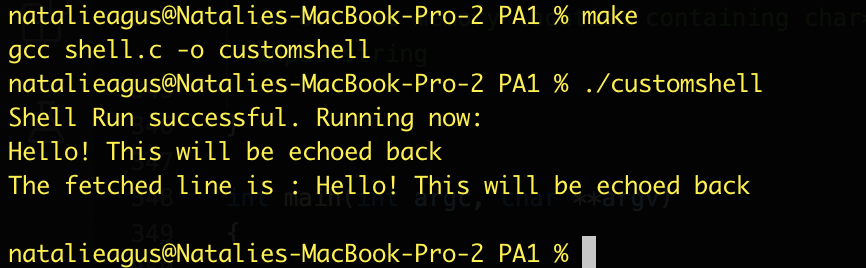
**char\* line = shellReadLine();**

**printf("The fetched line is : %s \n", line);**

**return 0;**

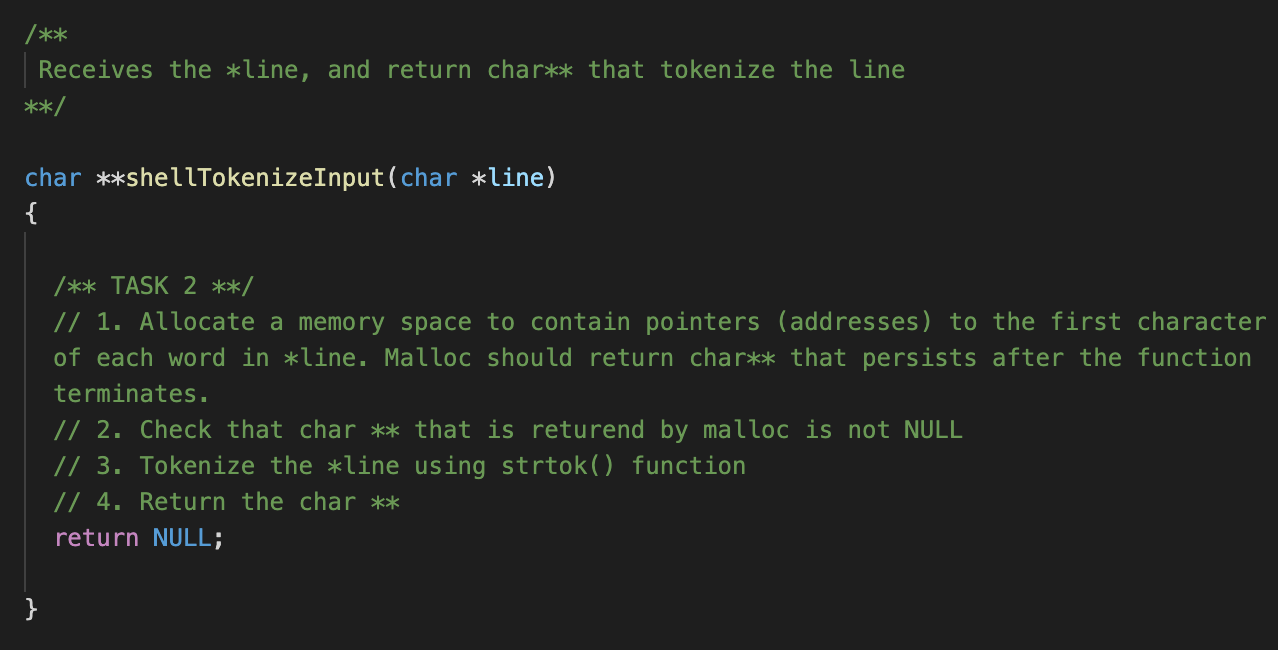
**}**

1. Recompile the code and you should see that what you typed in the console will be printed back after you pressed enter. You should see similar to the following in your terminal if you have implemented this task successfully:



## Task 2: Implement shellTokenizeInput

Next, you need to implement shellTokenizeInput function.



This function receives a pointer to the memory location that contains strings of character of the user input. It will return the pointers to addresses (char\*\*) that tokenize the input.

**Implementation notes:**

For example, let’s say the user types in the following input:

gcc shell.c -o customshell

This string of characters are stored within a persistent memory location. There are four tokens separated by spaces. These tokens are:

* gcc
* shell.c
* -o
* customshell

Now obviously we need to *remember* where these tokens are, therefore the function returning char\*\* type.

You must use strtok(char\* input, delimiter) to tokenize the input line. You can read on how to use it here: <http://man7.org/linux/man-pages/man3/strtok.3.html>

The following code snippet below should help you get started and learn how to use strtok():

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

int main()

{

char command[] = "gcc shell.c -o customshell";

char \*line = command; // pointer 'line' stores where the command starts

printf("Address if \*line is %0x\n", &line);

printf("Address of the letter 'g' is %0x\n", line);

printf("Address of the letter 's' is %0x\n", line + 4);

printf("Address of the letter '-' is %0x\n", line + 12);

printf("Address of the letter 'c' is %0x\n", line + 15);

char \*\*token\_positions = malloc(sizeof(char \*) \* 8);

char \*token = strtok(command, " ");

int index = 0;

token\_positions[index] = token;

index++;

while (token != NULL)

{

// Tokenize the rest of the command

token = strtok(NULL, " "); // continue finding the next token

token\_positions[index] = token; // store the position

index++;

}

token\_positions[index] = NULL; // dont forget to NULL terminate.

printf("First token is : %s, it is at address %0x \n", token\_positions[0], token\_positions[0]);

printf("Second token is : %s, it is at address %0x \n", token\_positions[1], token\_positions[1]);

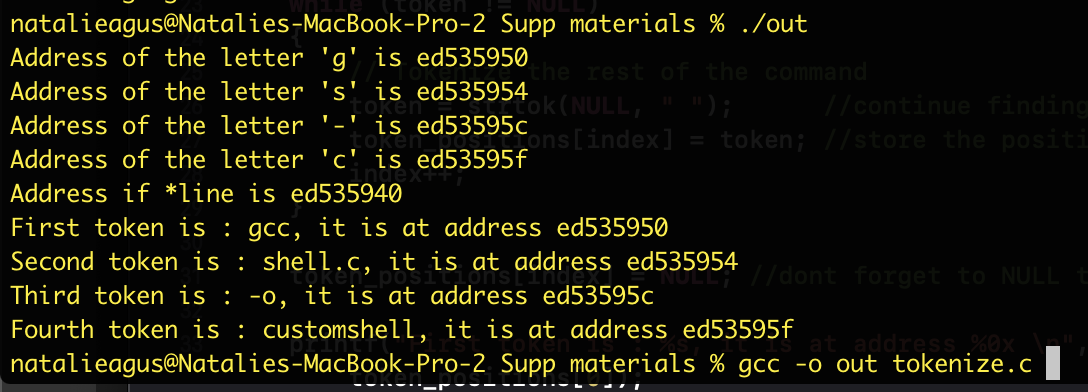
printf("Third token is : %s, it is at address %0x \n", token\_positions[2], token\_positions[2]);

printf("Fourth token is : %s, it is at address %0x \n", token\_positions[3], token\_positions[3]);

}

Note that for the delimiter field of strtok, remember to add in other delimiter as well such as \n and \t so that it will strip off the words from these trailing characters. This is what the macro SHELL\_INPUT\_DELIM for in shell.h.

If you compile and run the code above, you should have the printed output similar to follows (address will be different, depending on your machine):



You should observe that the address of the letter ‘g’ that was manually printed in the first place should correspond to the token address ‘gcc’, and so on.

**To test:**

1. Replace the main() function with the following:

int main(int argc, char \*\*argv)

{

printf("Shell Run successful. Running now: \n");

char\* line = shellReadLine();

printf("The fetched line is : %s \n", line);

char\*\* args = shellTokenizeInput(line);

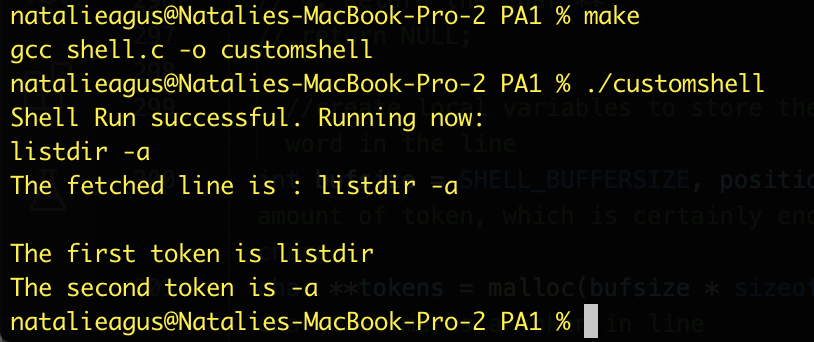
printf("The first token is %s \n", args[0]);

printf("The second token is %s \n", args[1]);

return 0;

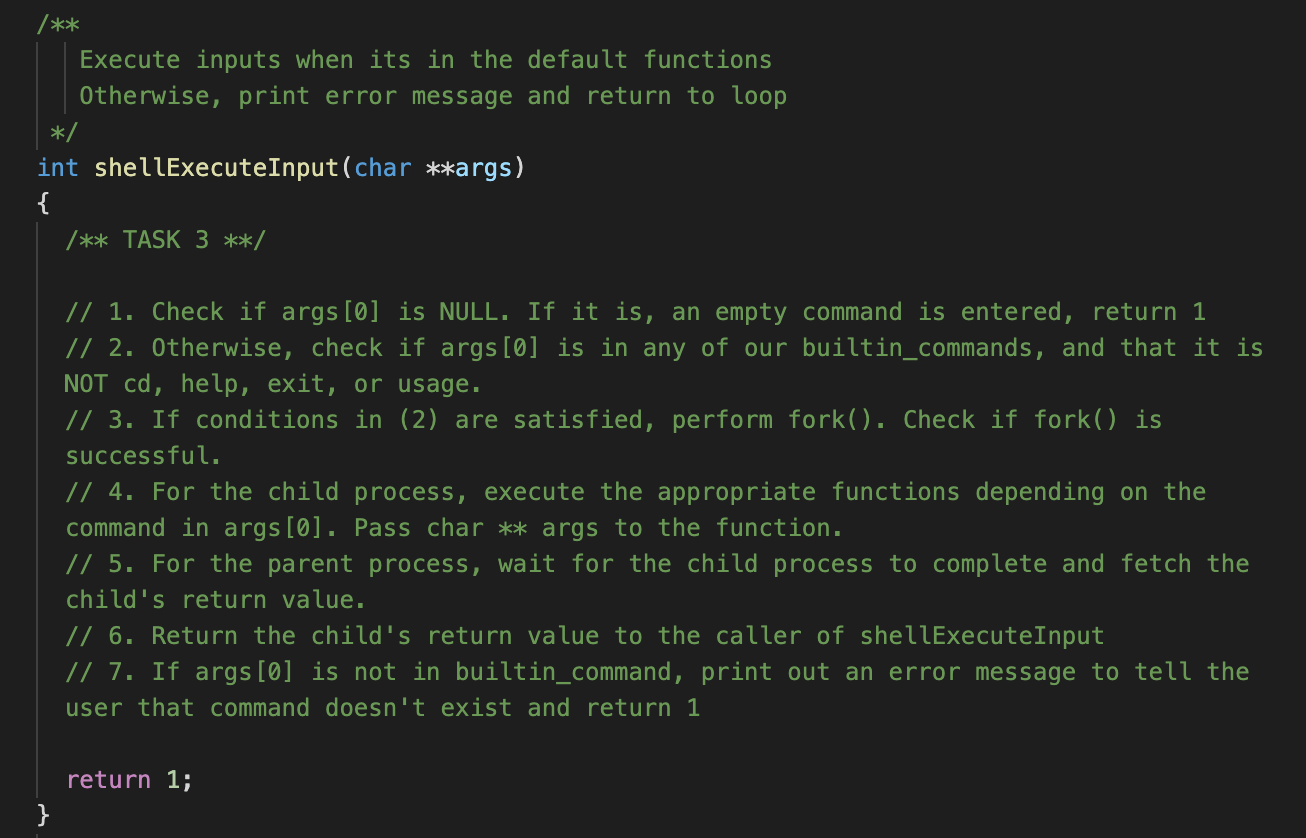
}

1. Compile and run it. Type the following input: listdir -a. You should see the following output if you have implemented the function successfully:



## Task 3: Implement shellExecuteInput

Next, you need to implement shellExecuteInput function.



**Implementation notes:**

1. Read how fork() works here: <http://man7.org/linux/man-pages/man2/fork.2.html>
2. The function fork() returns:
   * -1 upon unsuccessful fork
   * 0 in child process
   * > 0 (the pid of the child) in parent process
3. The **parent process** has to call:

**pid\_t waitpid(pid\_t *pid*, int \**stat\_loc*, WUNTRACED);**

to **wait for the child process** to finish **(waitpid with WUNTRACED blocks if child isn’t terminated yet, returns pid of child when child has terminated, returns -1 when there’s no more children)**, where pid is the pid of the child process returned by the fork(), and stat\_loc is a pointer to an integer which we can use to obtain the exit status of the child.

The parent process can read the exit status of the child process as such:

waitpid(pid, &status, WUNTRACED);

int exit\_status = 0;

// if child terminates properly, WIFEXITED(status) returns TRUE

if (WIFEXITED(status)){

exit\_status = WEXITSTATUS(status);

}

A little more info on what WEXITSTATUS do:

WEXITSTATUS(*stat\_val*)

If the value of WIFEXITED(*stat\_val*) is non-zero, this macro evaluates to the low-order 8 bits of the status argument that the child process passed to *\_exit()* or *exit()*, or the value the child process returned from *main()*.

Note: before you blatantly copy pasted the child exit status checker code above for the parent process to execute, think carefully on *whether you need them in the first place*. This all depends on your implementation.

1. The child process has to invoke the *right function* depending on args[0], and call exit(1) should the function invoked return. The *right function*, meaning any of the functions you see in [shell.h](#_klfpbhi8nbkb) you saw above depending on commands given in args[0]. The pseudocode is as follows:
   * if args[0] == “listdir”, means you need to call the function shellListDir(args)
   * if args[0] == “cd”, means you need to call the function shellCD(args)
   * And so on, checking args[0] whether it matches any of the builtinCommands.
2. It is implied that the child process has to **handle** the case whereby execvp fails, that is if it returns from the builtin functions, it has to ***exit***.
3. You can use strcmp() to compare between two strings, i.e: between args[0] and builtin\_commands[i], as you cannot perform == comparison with strings. Its documentation can be found here: <http://man7.org/linux/man-pages/man3/strcmp.3.html>
4. For built-in shell commands, check if **certain conditions apply** before calling them. Otherwise, print error messages. Your shell **SHOULD NOT** crash in any case. Although you are not allowed to change any default functions, **you are allowed to write helper functions / declare more global variables.**
5. You can use function pointers provided in shell.h if you are comfortable with using them. Otherwise, you can just use if-else to check between args[0] and builtin\_commands[i] and call the functions manually.

**To test:**

1. Replace the main() function with the following:

int main(int argc, char \*\*argv)

{

printf("Shell Run successful. Running now: \n");

char\* line = shellReadLine();

printf("The fetched line is : %s \n", line);

char\*\* args = shellTokenizeInput(line);

printf("The first token is %s \n", args[0]);

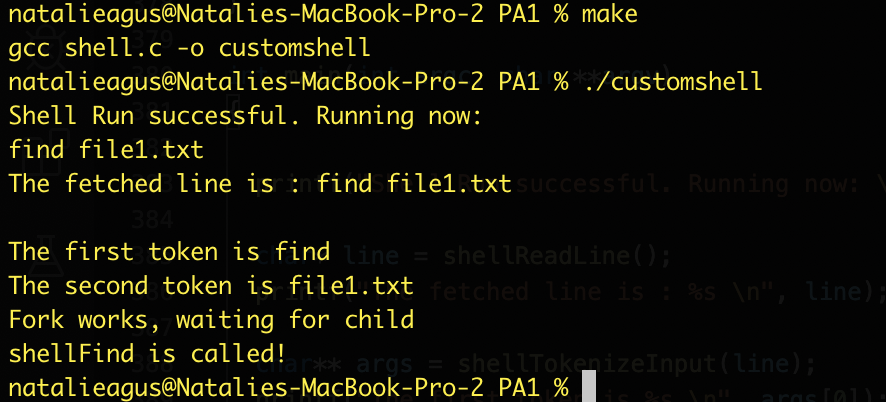
printf("The second token is %s \n", args[1]);

shellExecuteInput(args);

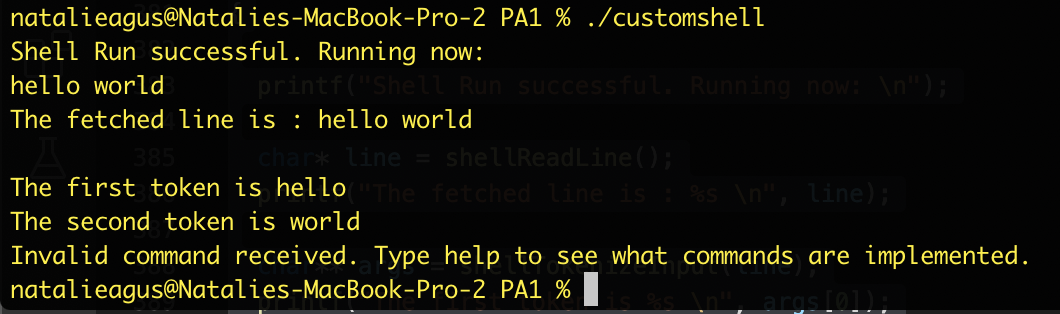
return 0;

}

1. Compile and run the code. Type find file1.txt to your terminal. You should see that the function shellFind is properly called. **You can print some kind of message to indicate that your fork() works properly.**



1. You may run the program again but this time round type in an illegal command, such as hello world. You should print out some kind of error message and return.



1. Finally, if you type commands such as cd, help, exit, or usage, your shell will NOT fork. Notice how the message “Fork works, waiting for child” no longer exists in this test case.

## 

1. You can easily test with help as well:

## 

## Note:

For checkoff, you need to make your shell ***complete***, meaning that it has a way to **exit** properly using the exit function given to you, it will give an error message if the command is not valid or incomplete, or if any system call fails, *etc*. In other words, it should implement ALL 11 commands as listed in the screenshot above. Your shell **should NOT** crash due to any input from the user.

## Task 4: Complete the rest of the shell functions.

These four functions in shell.c has been implemented for you:

int shellCD(char \*\*args);

int shellHelp(char \*\*args);

int shellExit(char \*\*args);

int shellUsage (char\*\* args);

Notice how they don’t involve any fork, as they must be performed in the shell’s address space.

As for the other functions fork() must be performed before calling them. Your job is to complete all their implementations using execvp():

int shellDisplayFile(char\*\* args);

int shellCountLine(char\*\* args);

int shellListDir(char\*\* args);

int shellListDirAll(char\*\* args);

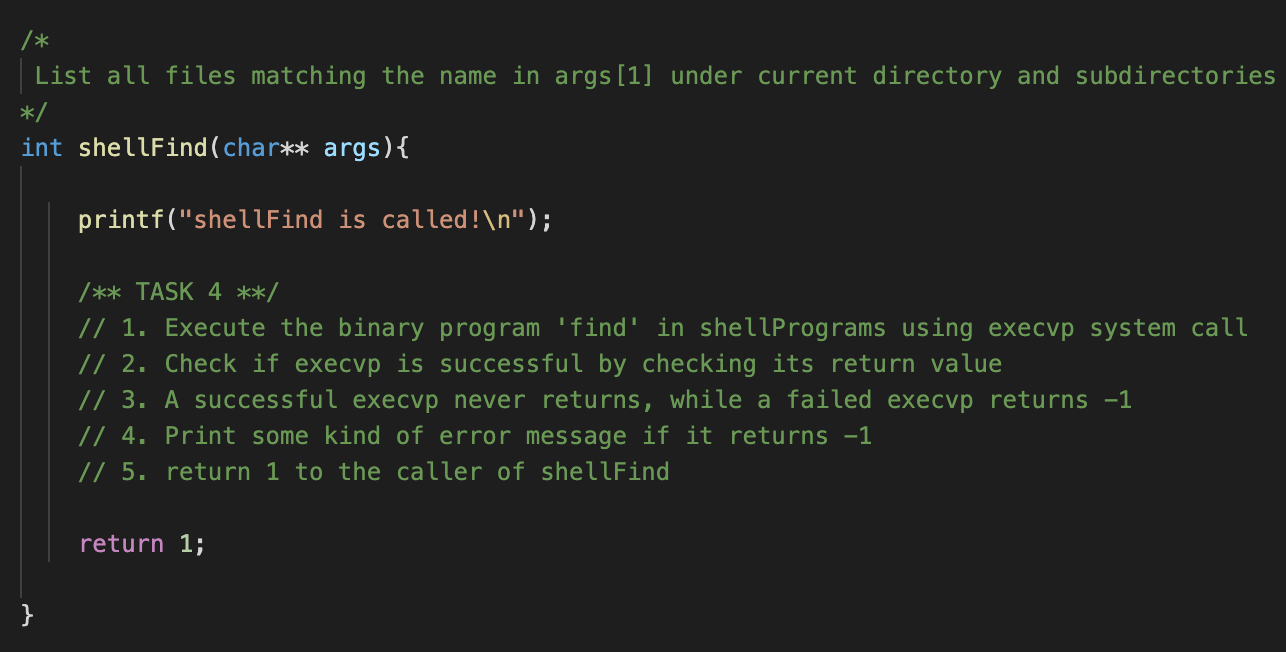
int shellFind(char\*\* args);

int shellSummond(char\*\* args);

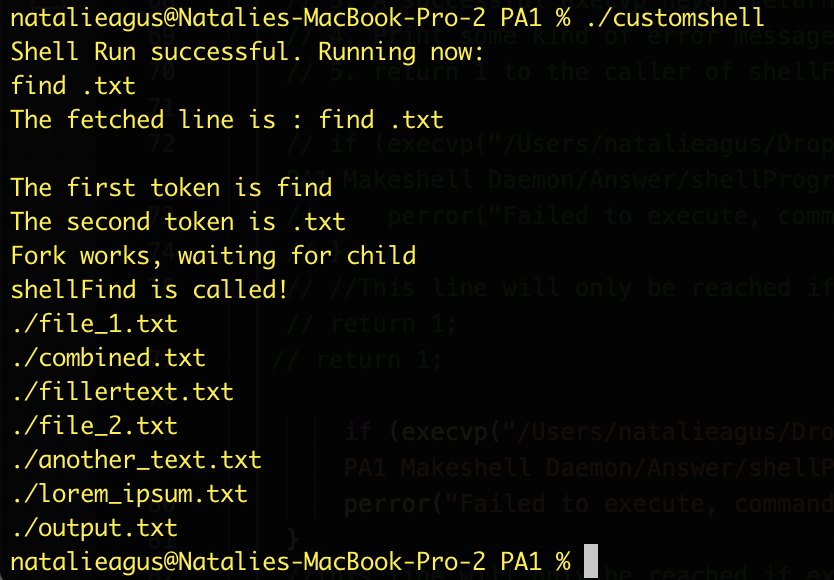
int shellCheckDaemon(char\*\* args);

This is because we will need to *execute* the appropriate custom programs in shellPrograms to do what the user commands the shell to do. The implementation of these commands **are not** implemented within shell.c itself.

For example, look at shellFind function. The goal of this program is to list all the matching files in the current directory and subdirectories.



If implemented properly, the output should be (keep the same main() function as given in Task 3):



**Implementation notes:**

1. Use execvp() to execute the appropriate binary file in shellPrograms folder. It is recommended that you write the *full path*. For example, using MacOS path, execvp can be used as follows:

execvp("/Users/natalieagus/Dropbox/50.005 Computer System Engineering/2020/PA1 Makeshell Daemon/Answer/shellPrograms/find", args)

**Debug tip:** Since this child process is created by customshell, it should inherit the current working directory of the customshell. Therefore, you can type a command like find .txt to find all the text file in the customshell working directory although the program find is in another directory (inside shellPrograms folder). If the programs are complaining that it cannot find the file you’re looking for, *it is useful to* ***do a sanity check*** *on the current working directory by inserting the code: system(“pwd”);* in the **child** **process** right after fork() OR in the shellPrograms script (e.g: inside shellFind\_code.c etc) temporarily *just to see* if its current working directory is as expected.

1. Read the manual on how to use execvp: <https://linux.die.net/man/3/execvp>

**To test:**

1. Replace the main() function with the following (this is the same as Task 3’s):

int main(int argc, char \*\*argv)

{

printf("Shell Run successful. Running now: \n");

char\* line = shellReadLine();

printf("The fetched line is : %s \n", line);

char\*\* args = shellTokenizeInput(line);

printf("The first token is %s \n", args[0]);

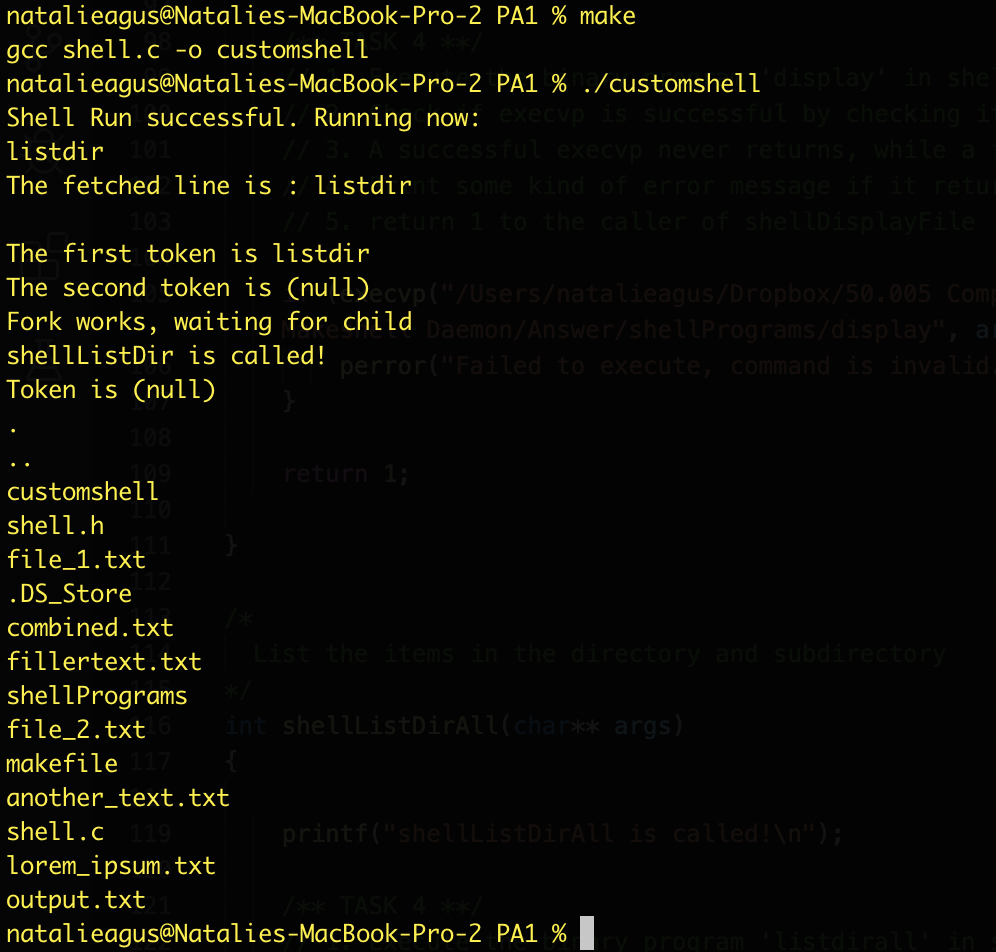
printf("The second token is %s \n", args[1]);

shellExecuteInput(args);

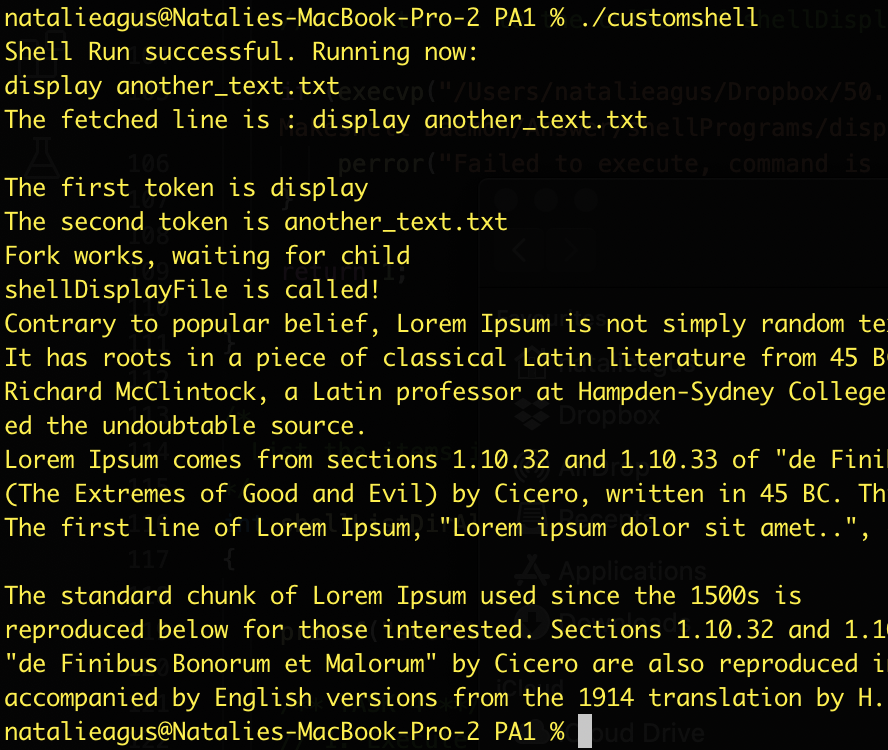
return 0;

}

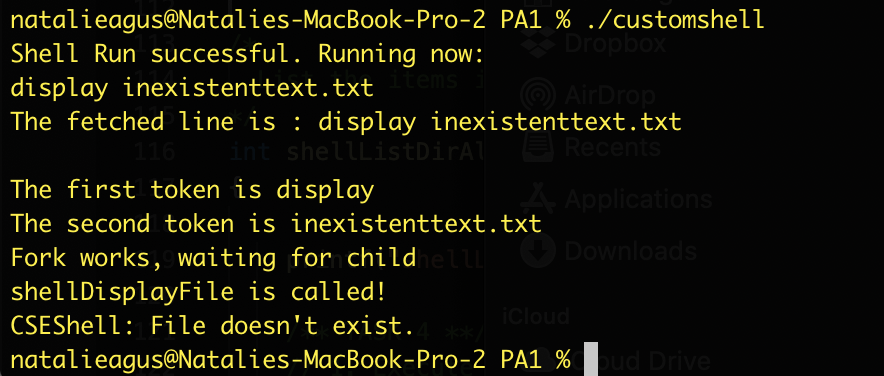
1. You can test with listdir and get the following output:



1. The command display another\_text.txt will display the content of the txt file:



1. You must print some kind of error message if the program cannot do what you asked it to do. For example, if we try to display a file that doesn’t exist with display inexsistenttext.txt:



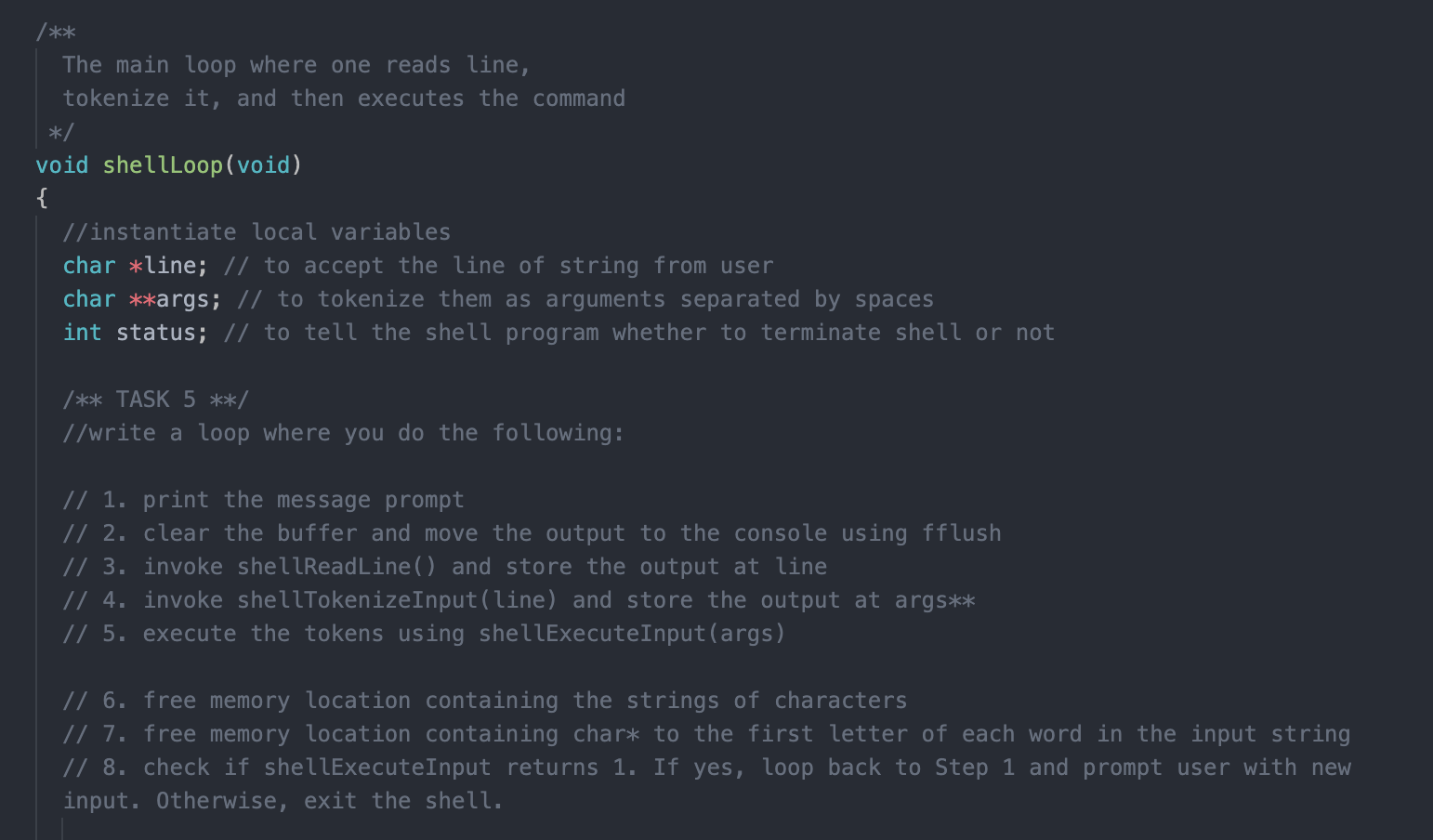
**Right now: listdir, listdirall, find, and display are also implemented for you. On top of completing the shell, you will implement the other three: checkdaemon, countline, and summond in this assignment.**

So don’t be surprised at this stage if you type countline <filename>, summond or checkdaemon and nothing else happens except the print message.

**Tips: the implementation of shellUsage *works* but buggy. Can you find out why?**

## Task 5: Looping the Shell

Of course a shell is not really a shell if it can only accept one command and exit. You would expect the shell to prompt you with a **new** command once it has executed (be it successfully or unsuccessfully, your previously entered command). We can do this by completing the shellLoop function. By now, you should know how to complete it by simply reading the pseudocode given.

*Update note: the old version had a step whereby you are asked to clear the buffer used to accept new string. You can ignore that, it is not necessary to do so.* 

**To test:**

1. Replace the main() function with the following:

int main(int argc, char \*\*argv)

{

printf("Shell Run successful. Running now: \n");

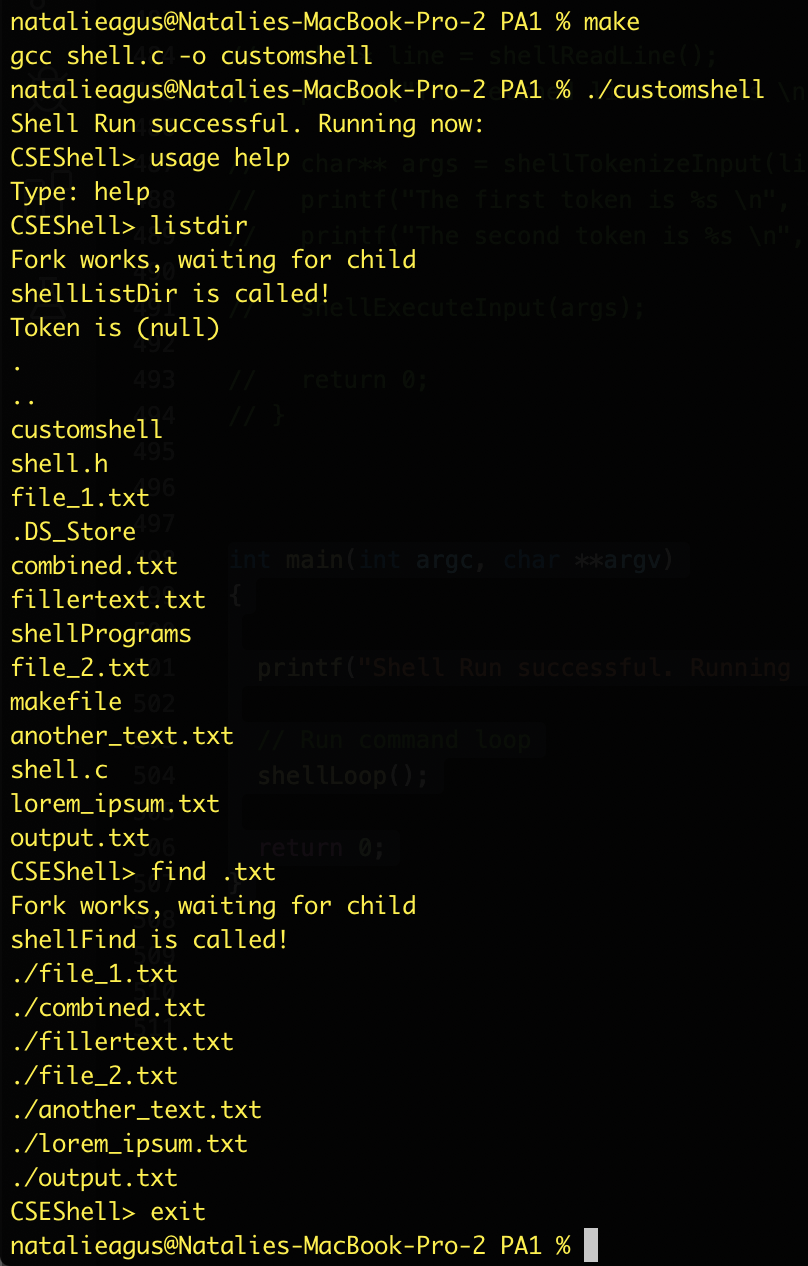
// Run command loop

shellLoop();

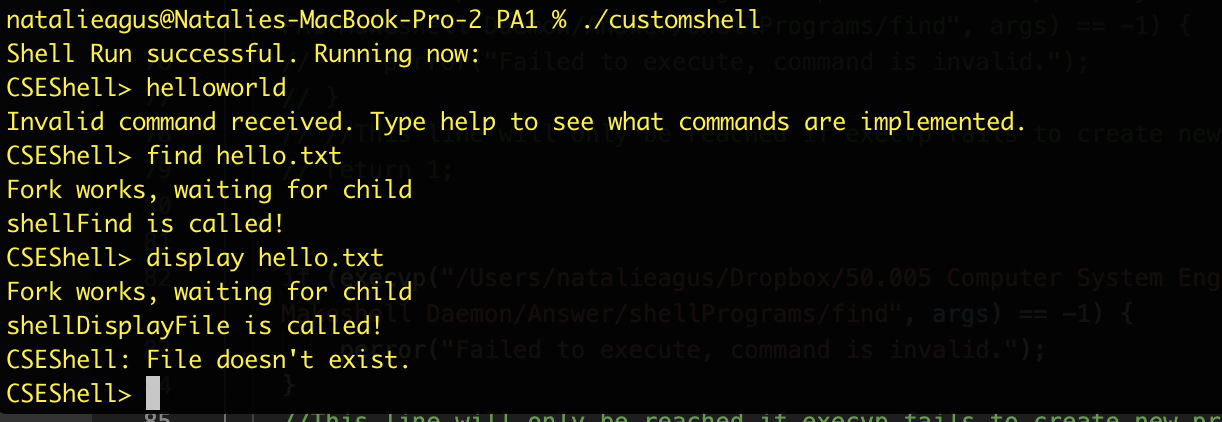
return 0;

}

1. Build and run the program. Your shell should prompt you with new line each time you type a command, and exit only when you type exit.

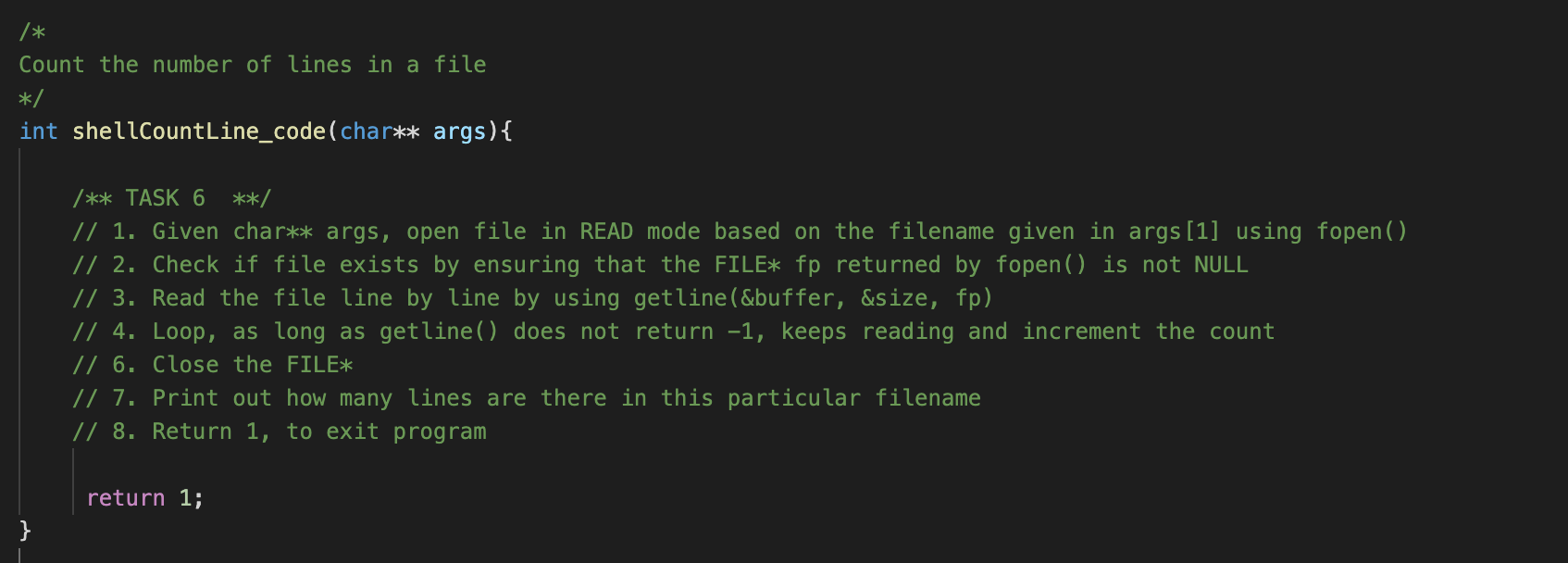


1. It also should be able to print some kind of error message upon encountering illegal command, and present the user with new prompt (it does not crash or exit):



## Task 6: Implement countline

Now that we have a shell that is up and running, it is time to make several custom programs by ourselves. Open shellCountLine\_code.c inside shellPrograms folder. As indicated in the file itself, your job is to implement the shellCountLine\_code function.



**You need to implement this program from scratch and are NOT allowed to use an existing program to help you, e.g: using system(“wc <filename>”).**

**Implementation notes:**

1. The filename given in args[1] is the name of the file in the **current** directory of the caller of countline binary, not the directory of the countline itself. Think about who calls execvp of countline.
2. To open a file, you need to use fopen. A typical usage is as follows for *reading* a file: FILE\* fp = fopen(args[1], "r");

Please read its documentation here: <http://man7.org/linux/man-pages/man3/fopen.3.html>

1. To fetch **1 line of characters** inside the opened file, you need to use getline (just like you did in Task 1 to read stdin). This function returns a whole line in a file (a line is defined as a string of characters terminated by ‘\n’).

getline returns -1 whenever we encounter the end of file, otherwise, it returns the number of characters in that line. An empty line “\n” will be returned as 1. A line “hello\n” will be returned as 6. Hence the terminating character is counted when getline returns.

You need to call it several times to traverse through the file line by line using a loop, where on each call of getline you check whether it returns -1. Each time it loops and does not return -1, you increment the line counter.

Please read its documentation here: <http://man7.org/linux/man-pages/man3/getline.3.html>

1. To simplify, you can count empty lines as 1 line as well (you don’t have to skip them).

**To test:**



1. Recompile the files you have in shellPrograms by typing make
2. Go one directory up to the PA1 folder using cd ..
3. Run ./customshell, and test countline with several text files given to you

## Task 7: Implement summond

This program summons a daemon, and terminates so that the shell may continue to print the next prompt. This is unlike other programs where the shell waits for it to finish before printing the next prompt.

**Daemons** are processes that are often started when the system is bootstrapped and terminate only when the system is shut down. **They don’t have a controlling terminal[[2]](#footnote-1)** and **they run in the background**. In other words, a Daemon is *a computer program that runs as a background process, rather than being under the direct control of an interactive user.* UNIX systems have numerous daemons that perform day-to-day activities.

Traditionally, the process names of a daemon end with the letter d, for clarification that the process is in fact a daemon, and for differentiation between a daemon and a normal computer program. For example: /sbin/launchd, /usr/sbin/syslogd, /usr/libexec/configd, etc (may vary from machine to machine). You can launch the command ps aux to identify these processes whose names end with a suffix ‘d’.



For the sake of our lab and our machine’s health, our daemon *terminates* after a certain period of time and violates the traditional daemon definition, but we’re sure you get the idea.

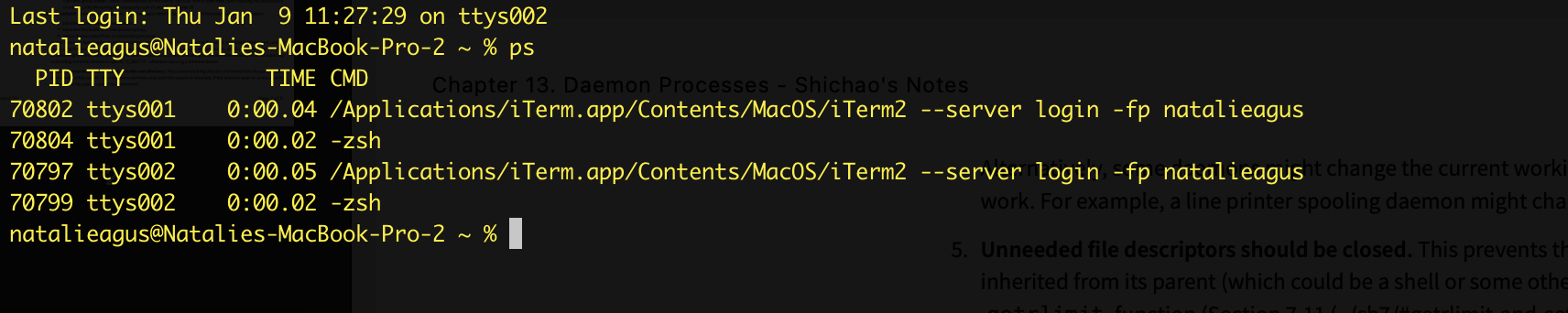
*Note: the basic information about daemons presented in this handout is sufficient. Do not over-Google about Daemons unless you are really interested in it. The concept of daemons alone is very complex and large, and is out of our scope.*

**A daemon process in essence is still a normal process, with certain characteristics which distinguish it from a normal process. The characteristics of a daemon process are as follows:**

1. It has no controlling terminal. By definition, a daemon process does not require direct user interaction and therefore must detach itself from any controlling terminal.

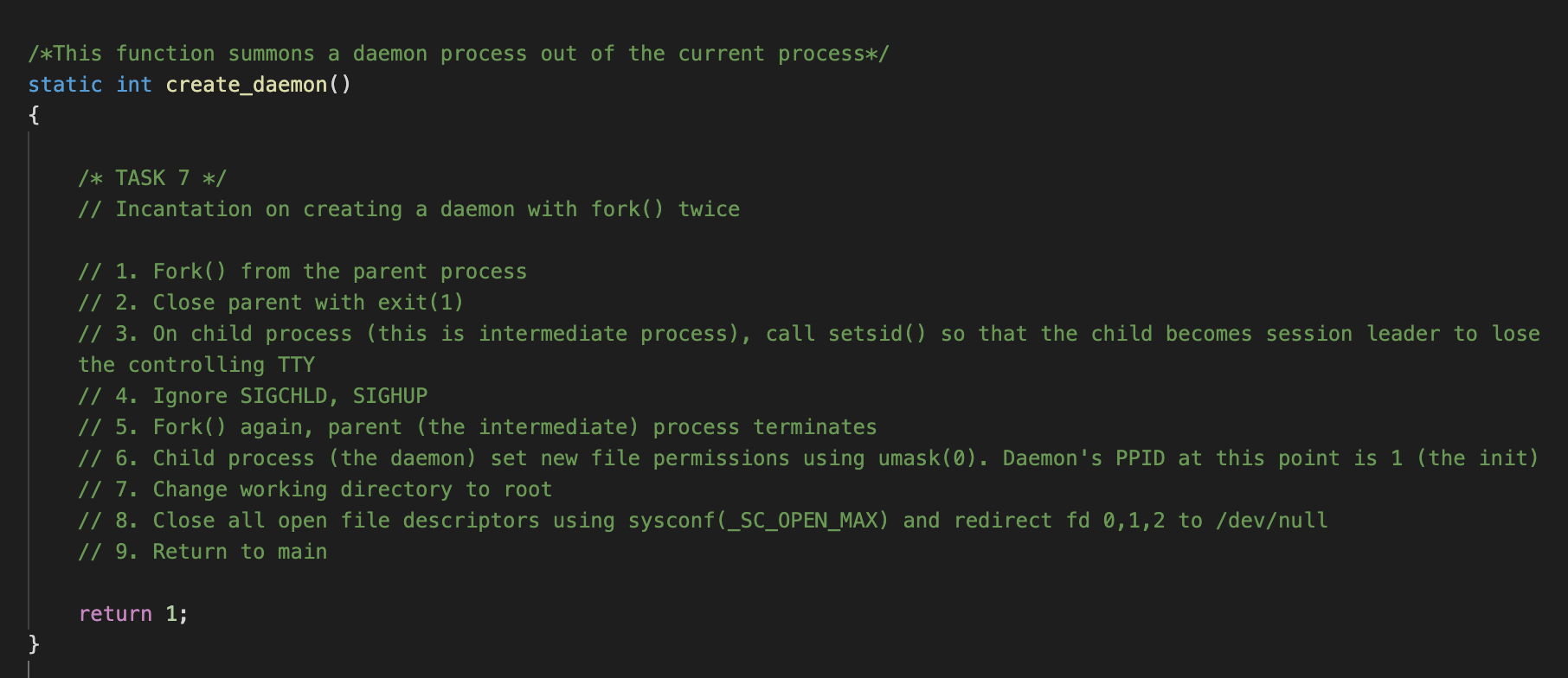
In the ps -efj output, if the TTY column is listed as a ? meaning it does not have a controlling terminal.

You can list all processes that have terminals instead, and it will look as the screenshot follows. This means two terminals are present (in laymen terms, two terminal windows are opened at the time this ps command is entered): ttys001 and ttys002, each controlling an instance of -zsh shell process.



1. The PPID of a daemon process is 1, meaning that whoever was creating the daemon process *must terminate* to let the daemon process be adopted by the *init* process.
2. The working directory of the daemon process is typically the root directory
3. It closes all unneeded file descriptors[[3]](#footnote-2)
4. Close and redirect fd 0, 1, and 2 to /dev/null
5. It logs messages through a central logging facilities: the BSD syslog

Write your answer in /shellPrograms/shellDaemonize\_code.c:



**Implementation notes:**

The steps written as a pseudocode above is a guide on how to create a daemon process that possesses the characteristics written previously. The section below explains why each steps are crucial:

1. The fork() splits this process into two: the parent (process group leader) and the child process (that we will call intermediate process here)
2. The reason for this fork() is so that the parent returns immediately and our shell ***does not wait*** for the daemon to exit (because usually, daemons are background processes that do not exit until the system is shut down. We don’t want our shell to hang forever).
3. The child (intermediate process) -- that is not a process group leader -- calls setsid() to be the session leader and loses controlling TTY (terminal)**. setsid() is only effective when called by a process that is not a process group leader**. The fork() in step 1 ensures this. The system call setsid() is used to create a new session containing a single (new) process group, with the current process as both the session leader and the process group leader of that single process group. You can read more about setsid() here <http://man7.org/linux/man-pages/man2/setsid.2.html>

Confused about process group leader and session leader? Read a short guide below.

Compile and tryout this code:

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

int main(){

pid\_t pid = fork();

if (pid == 0){

printf("Child process with pid %d, pgid %d, session id :%d\n", getpid(), getpgid(getpid()), getsid(getpid()));

setsid(); // child tries setsid

printf("Child process has setsid with pid %d, pgid %d, session id :%d\n", getpid(), getpgid(getpid()), getsid(getpid()));

}

else{

printf("Parent process with pid %d, pgid %d, session id :%d\n", getpid(), getpgid(getpid()), getsid(getpid()));

setsid(); // parent tries setsid

printf("Parent process has setsid with pid %d, pgid %d, session id :%d\n", getpid(), getpgid(getpid()), getsid(getpid()));

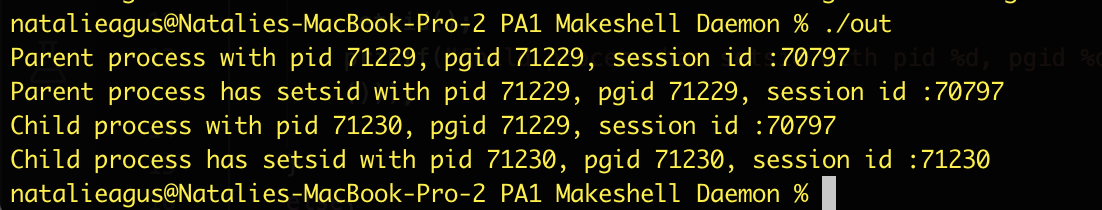
wait(NULL);

}

return 0;

}

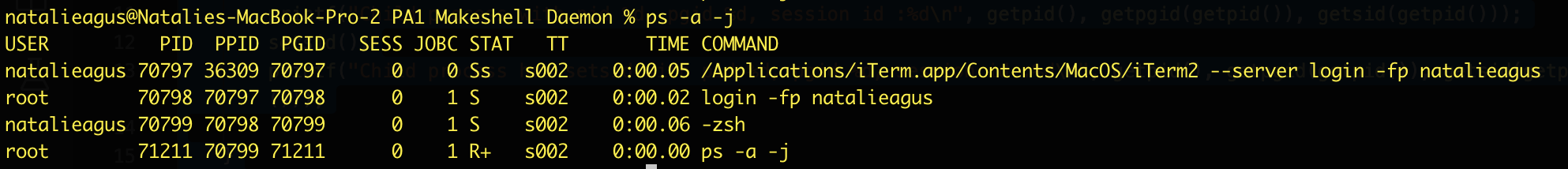
It results in such output:



So the parent process has pid == pgid, that is 71229. This tells us that this process ‘out’ is the process group leader, but not a session leader since the session id 70797 is not equal to the pid 71229.

When process 71229 forks, it has a child process with pid 71230. It is clear that since child pid != pgid, then the child process is *not* a session leader and is *not* a group leader either.

So who is 70797? We can type the command ps -a -j and find a process with pid 70797. Apparently, it's the iTerm, the terminal itself, connected to the controlling terminal s002.



Now both the child and parent process attempt to call setsid. In the child’s process, setsid effectively makes the pgid and session id to be equal to its pid, 71230. In the parent’s process, setsid has **no effect** on the session id, since the manual states that setsid only sets the process to be the session and process group leader if it is called by a process that is *not* a process group leader.

1. Ignore SIGCHLD and SIGHUP:

signal(SIGCHLD, SIG\_IGN);

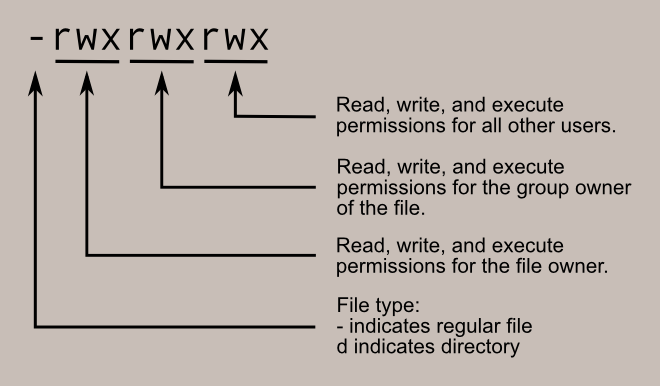
signal(SIGHUP, SIG\_IGN);

The reasons are as follows:

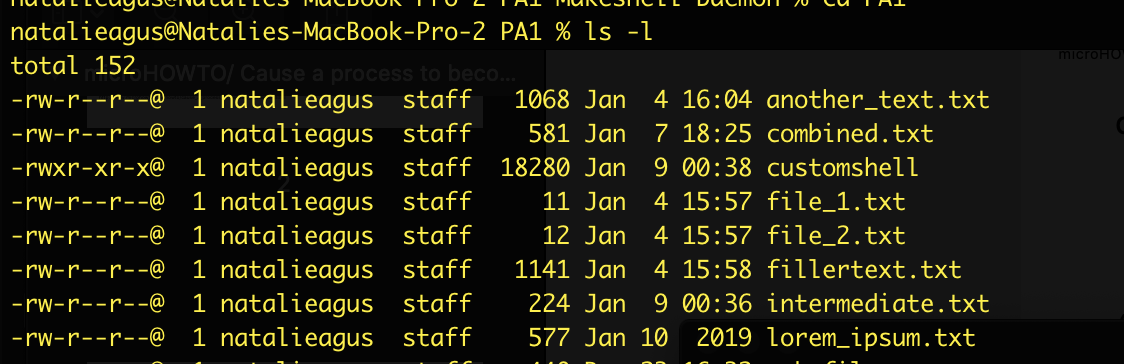
* Ignore SIGCHLD: This intermediate process is going to fork() one more time to create the daemon process. By ignoring SIGCHLD, our daemon process -- the child of this intermediate process *will not be a zombie process*. When the daemon exits, it is reaped immediately. However, the daemon will outlive the parent process anyway, so it does not really matter. This step is more like a belt and suspender approach.
* Ignore SIGHUP: We are going to fork() one more time for reasons stated in the next step 5, so the child of this intermediate process is the daemon we are creating. However, this intermediate process **is a session leader** (from step 3, since we need to lose the controlling terminal). If we terminate a session leader, a SIGHUP signal will be received and the children of the session leader will be killed. We do not want our daemon to be killed, therefore we need to call signal(SIGHUP, SIG\_IGN) first before forking, and terminating the intermediate process.

1. Perform the second fork(), of which the child of this intermediate process **is** the daemon process:
   * The second fork, is again useful for allowing the parent process to terminate. **This ensures that the child process is not a session leader.** Since a daemon has no controlling terminal, if a daemon **is a session leader**, an act of opening a terminal device will make that device the controlling terminal.
   * *We do not want this to happen with your daemon, so this second fork() handles this issue*. As mentioned above, before forking it is necessary to ignore SIGHUP. This prevents the child from being killed when the parent (which is the session leader) dies.
2. Set new file permissions using umask(0). Setting the umask to 0 means that newly created files or directories created will have all permissions set, so any file created by this daemon can be accessed by any other processes because we can’t directly control the daemon anymore. In other words, a umask of zero will cause all files to be created as permission 0777 or world-RW & executable (some system sets the permission as 0666 by default instead of 0777 for security reasons). The manual for umask can be found here: <http://man7.org/linux/man-pages/man2/umask.2.html>

If you want to know about file permission, read on. This is more for supplementary material. Unix file permission has the following format:



You can view them by typing ls -la.

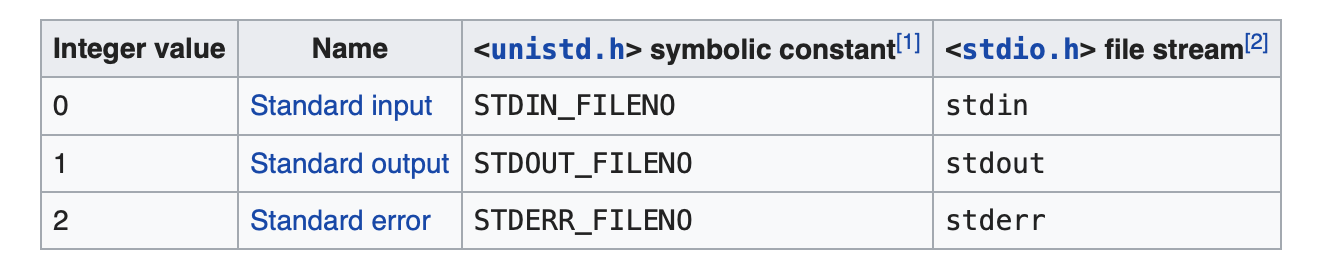


From the above example, it's obvious that .txt files are not executable, so it doesn’t have the ‘x’ field. Only the user ‘natalieagus’ can write onto these text files. Others can only read. Also, none of these are directories.

Setting a file umask(0) (basically umask(000)) is equivalent to having the permission 0777. 0777 actually stands for **octal** 777 (in C, the first 0 indicates octal notation), and you can translate it binary: 111 111 111, which means we will have - rwx rwx rwx, equivalent to *global RW and executable*. If we want to restrict permission of write to only the owner, we can set umask(022) -- equivalent to having permission 755, with the binary: 0 111 101 101, which means we will have - rwx r-x r-x, similar to the screenshot of the textfiles (customshell row) above. Note that for files that are not executable, the permission for executable is simply nonexistent - (so the permission set with umask(0) is 0666).

1. Change the current working directory to root using chdir("/"). If a daemon were to leave its current working directory unchanged then this would prevent the filesystem containing that directory from being unmounted while the daemon was running. It is therefore good practice for daemons to change their working directory to a safe location.
2. Close all open file descriptors and redirect stdin, stdout, and stderr (fd 0, 1, and 2 by default in UNIX systems) to /dev/null so that it won’t reacquire them again if you mistakenly attempt to output to stdout or read from stdin.

Quick note about file descriptors, each UNIX process (except a daemon) should have at least three standard file descriptors corresponding to each streams (input, output, error):



Once it is running a daemon should not read from or write to the terminal from which it was launched. The simplest and most effective way to ensure this is to close the file descriptors corresponding to stdin, stdout and stderr. These should then be reopened, either to /dev/null, or if preferred to some other location.

There are two reasons for not leaving them closed:

1. To prevent code that refers to these file descriptors from failing
2. To prevent the descriptors from being reused when we call open() from the daemon’s code.

To close all opened file descriptors, you need to loop through existing file descriptors, and re-attach the first 3 fd’s using dup(0). Note that open() and dup() will assign the smallest available file descriptor, in this case that is 0, 1, and 2 in sequence.

/\* Close all open file descriptors \*/

int x;

for (x = sysconf(\_SC\_OPEN\_MAX); x>=0; x--)

{

close (x);

}

/\*

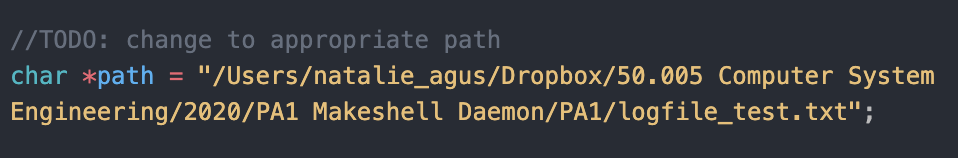
\* Attach file descriptors 0, 1, and 2 to /dev/null. \*/

fd0 = open("/dev/null", O\_RDWR);

fd1 = dup(0);

fd2 = dup(0);

1. Finally, return to the main function, of which the next function daemon\_work() is called. Please edit the path variable to point to the the right working directory in your machine. **You can set it wherever you want it to be, logfile\_test.txt will be created there.**

****

The daemon\_work() function basically contains a very simple code. The daemon “simulates” logging behavior by periodically writing to a text file, stating its PID. In practice, logging is a very complicated matter.

It cannot (simply) write to:

1. **stderr**: it shouldn't have a controlling terminal.
2. **Console device: o**n many workstations the console device runs a windowing system.
3. **Separate files** (like what we do here): it's a headache to keep up which daemon writes to which log file and to check these files on a regular basis, not to mention the space it keeps.

In practice, there exist *daemon error-logging facilities.* The BSD syslog facility has been widely used since 4.2BSD. You can print these logs using syslog(). To try, replace the main code with the following:

int main(int argc, char\*\* args)

{

create\_daemon();

/\* Open the log file \*/

openlog ("customdaemon", LOG\_PID, LOG\_DAEMON);

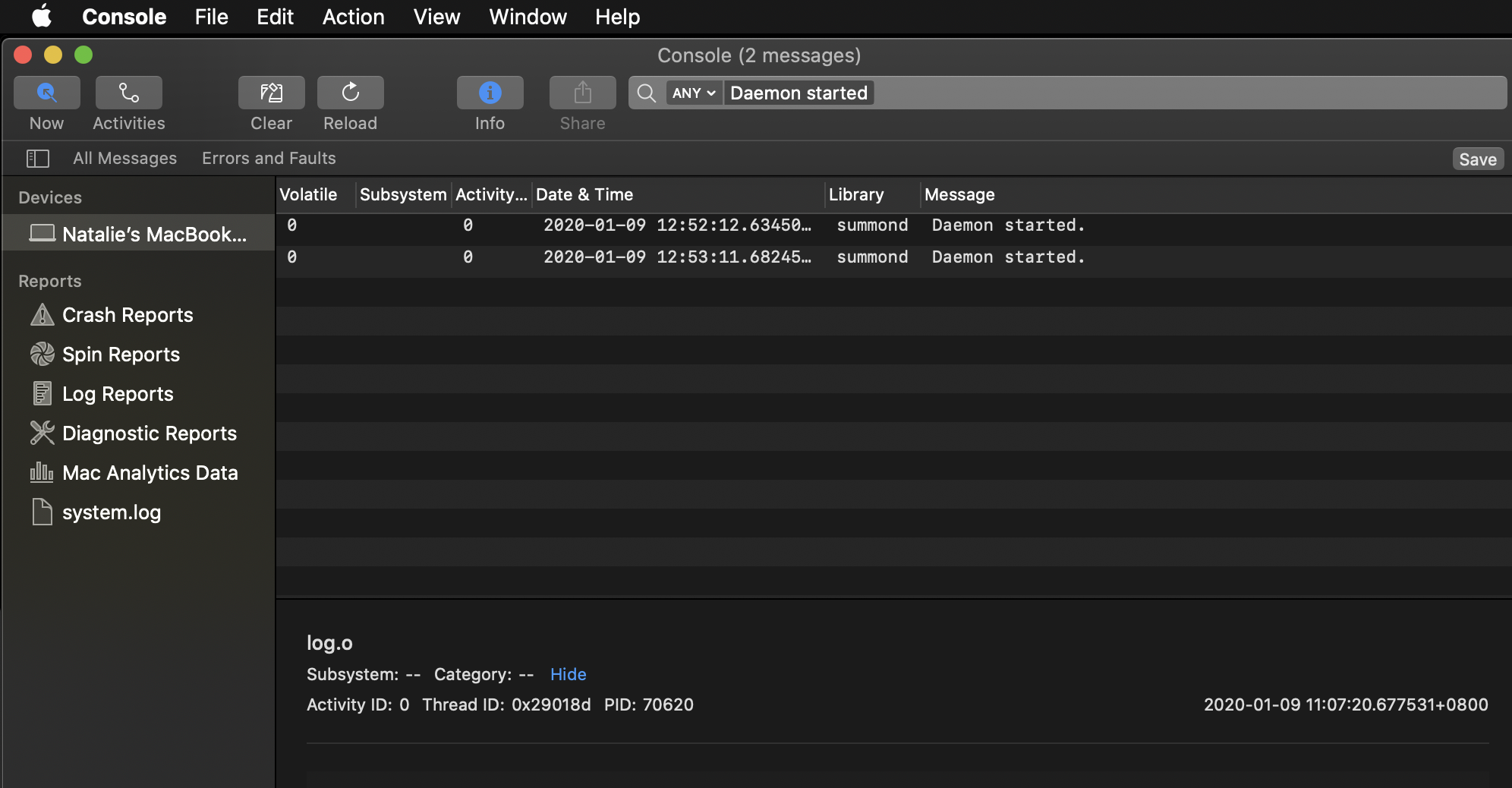
syslog (LOG\_NOTICE, "Daemon started.");

closelog();

return daemon\_work();

}

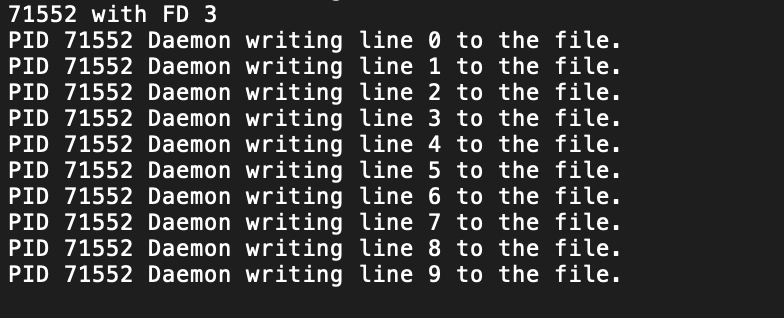
Depending on your machine, your syslog facility may vary. In MacOS, the syslog facility is the Console.app. You can simply search the message or the process name that is summond. In Ubuntu, you can use the Logs and search the message.



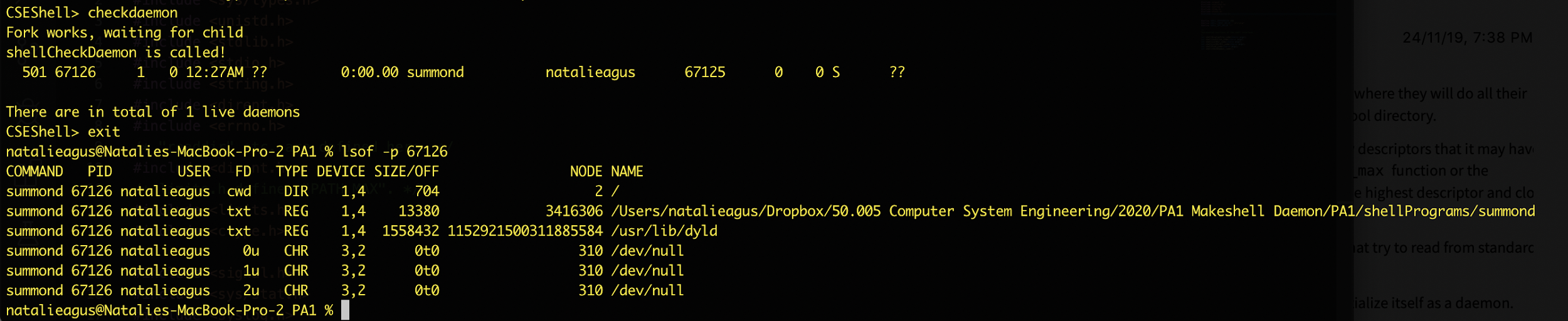
In practice, once you direct the log message here, you can tell the facility on how to forward it to your own logfile. This process is rather specific and out of our scope, therefore for the sake of the lab, we just assume that our daemon can directly write to our own logfile.

**To test:**

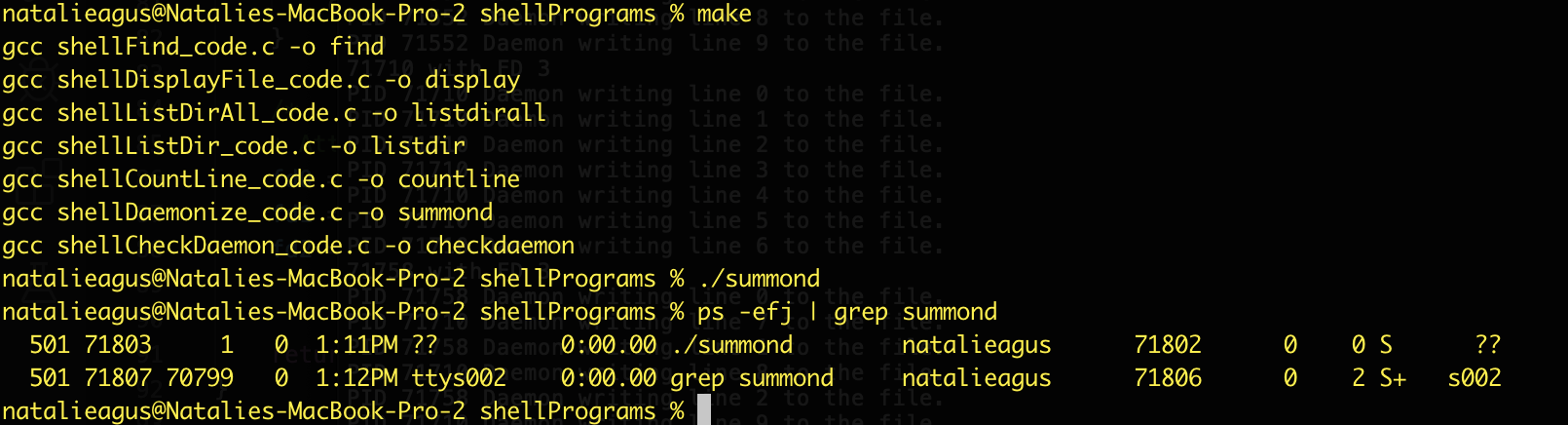
1. It is difficult to test a daemon easily. Compile the shellPrograms using make, and run your daemon process manually using ./summond to test. Finally, when you want to use your shell, you can of course type “summond” to invoke the daemon process. One way to check if it runs properly is to check if the logtest\_file.txt is indeed written by the daemon. You should see the content of the logfile\_test.txt resembles something like this:



1. After getting the PID, you can manually execute lsof -p <pid> to observe that its first 3 file descriptors are attached to /dev/null.



1. Observe its PPID by executing ps -efj | grep summond



The third field, indicates 1 for ./summond, which means that the parent (PPID) of the daemon is the init process. Notice also that **it is neither a session leader nor a group leader since its PGID 71802 that is not equal to its PPID 71803.**

## Task 8: Implement checkdaemon

The checkdaemon program checks and prints out the number of daemons that are currently alive right now. The skeleton code is more or less given to you already. Fill up the two parts labeled as TODO.



**Implementation notes:**

1. For the first TODO, the key idea is to use system(command) to help you execute standard system programs. It is very simple to use it. Don’t overthink your answer. Basically you just need to call system(command) and check its return value and continue if it is not -1. The documentation can be used here: <http://man7.org/linux/man-pages/man3/system.3.html>
2. The command that we are executing is:

ps -efj | grep summond | grep -v tty > output.txt

Basically, it lists the list of processes using the -efj option (you can google online on what this format means), and pipe | the output to grep program. The grep program filters the output of ps -efj to those that includes the word ‘summond’. Then, we pass the output to another grep program that excludes any with tty. Depending on your machine, you may need to **modify** this command to exclude basically the grep summond process from your results.

For example, in some machine grep summond process has a tty, so the sample command has the grep -v tty to exclude the line with the word tty on it (where the daemons dont have a tty). Therefore we effectively only obtain the results of the daemon process only.

The screenshot below should make it obvious to you why we need to call grep *twice,* chaining one with another.

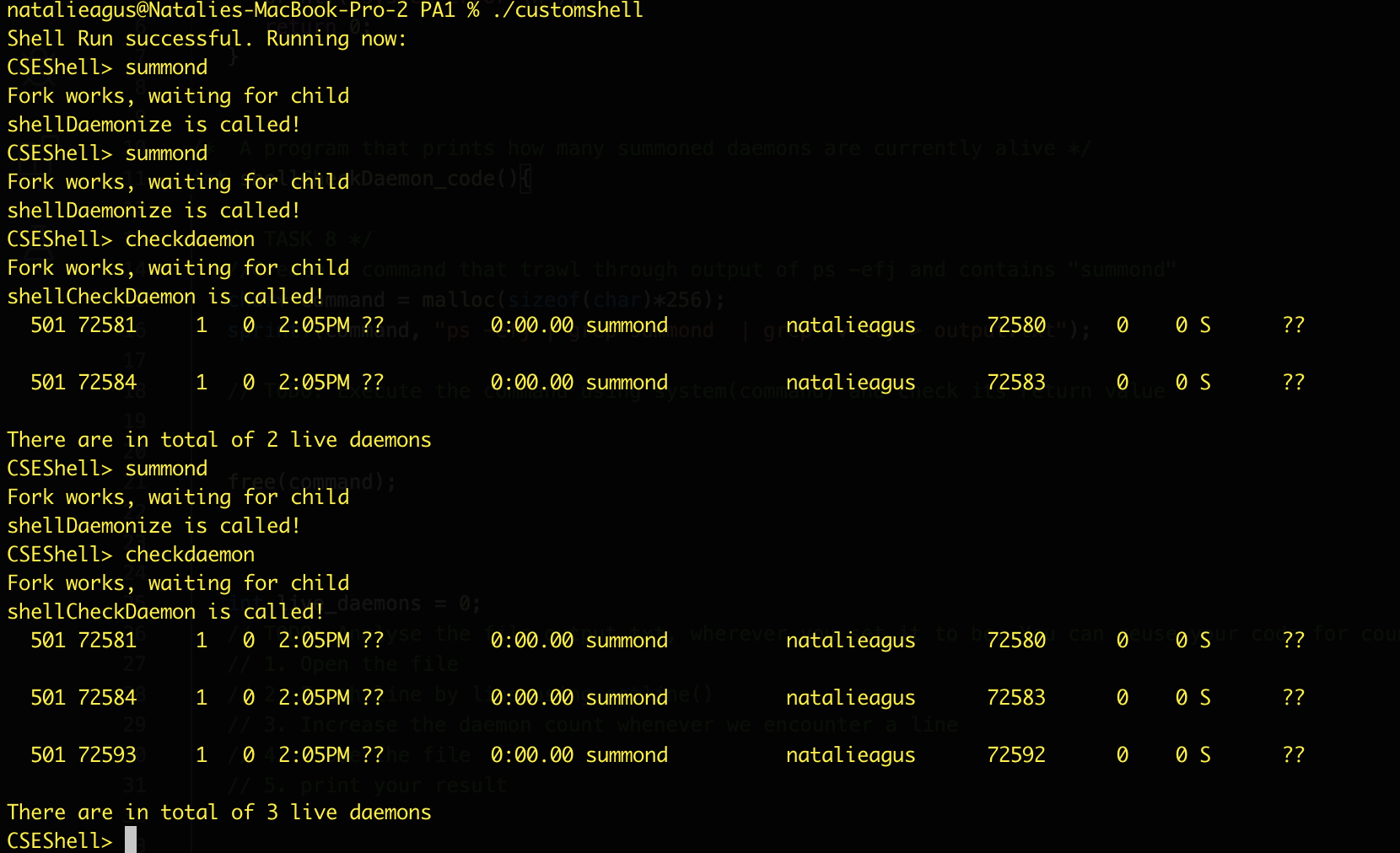


Finally the > command directs the stdout to output.txt file because we need to analyse it.

1. For the second TODO, we simply count how many lines are there in output.txt, since one entry represents each daemon that’s alive. You can reuse your code in countline.
2. The third TODO reminds you to close any file pointer you created and free any dynamically allocated memory.

**To test:**

1. Run your shell, summond some daemons and type checkdaemon. Your output should resemble the following.



1. Like what is done above, although **not necessary,** you can try printing each line of output.txt while looping through it using getline() in checkdaemon for easy debugging.

## Summary

**If you have reached this stage, congratulations for completing the programming assignment.** We hope that you have appreciated some valuable things regarding:

1. Purposes of fork() in shell programs
2. Incantation steps of creating daemon processes, and the reason why these steps must be done
3. A few of C standard functions: system, getline, malloc, free, signal, chdir, among many others and amp up your experience in reading documentations
4. Compiling using make and using the terminal
5. Basic knowledge about file descriptors, and file permissions

Refer back to [this section](#_wmq4adjsqujd) for submission procedure.

1. Actually, we can just have one makefile inside PA1 folder that will compile both programs in PA1 and shellPrograms folder. However, there might be some problems in path naming between different machines, and to simplify things we give you one makefile in each folder. [↑](#footnote-ref-0)
2. A Terminal is your interface to the underlying operating system via a shell, usually bash. Most programs you invoke using your shell [↑](#footnote-ref-1)
3. In Unix and related computer operating systems, a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket. [↑](#footnote-ref-2)