Programming Assignment 1

50.005 Computer System Engineering

Due date: 15 March 23:59

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 36 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 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)**

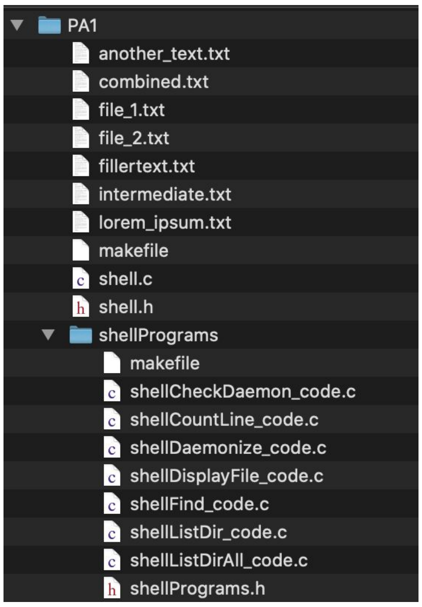
In the spirit of preventing us from copy-pasting ready-made code online, the quiz **(15 pts)** part aims to test you on your knowledge on each part of the coding task above. It is comprised of MCQ, short answers, matching, etc, and it will test your knowledge on the basic implementation of C functions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

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. **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 eDimension. **Only one person should submit.**

a. DO NOT modify ANY makefile



```
▼ PA1
    another_text.txt
    combined.txt
    file_1.txt
    file_2.txt
    fillertext.txt
    intermediate.txt
    lorem_ipsum.txt
    makefile
    shell.c
    shell.h
  ▼ shellPrograms
      makefile
      shellCheckDaemon_code.c
      shellCountLine_code.c
      shellDaemonize_code.c
      shellDisplayFile_code.c
      shellFind_code.c
      shellListDir_code.c
      shellListDirAll_code.c
      shellPrograms.h
```

b. 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.

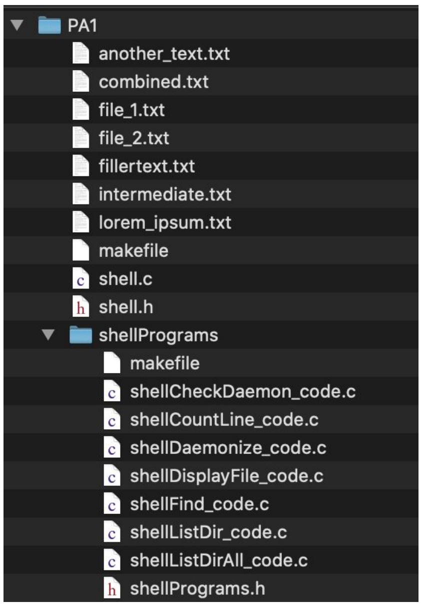
c. DO NOT modify ANY of the original functions in any header file `shell.h` and `shellPrograms.h` file: not the arguments, not the names, just don't modify any of these.

d. 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 sheet to book **your checkoff slot** (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:



```
PA1
    another_text.txt
    combined.txt
    file_1.txt
    file_2.txt
    fillertext.txt
    intermediate.txt
    lorem_ipsum.txt
    makefile
    shell.c
    shell.h
    shellPrograms
        makefile
        shellCheckDaemon_code.c
        shellCountLine_code.c
        shellDaemonize_code.c
        shellDisplayFile_code.c
        shellFind_code.c
        shellListDir_code.c
        shellListDirAll_code.c
        shellPrograms.h
```

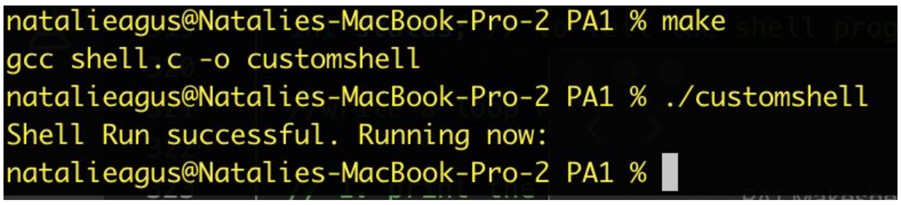
Notice that there's another folder called `shellPrograms` inside `PA1` folder. This folder contains the scripts for custom programs of which the shell can run.

The shell script is called `shell.c` with header file `shell.h`, both are inside `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].

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:

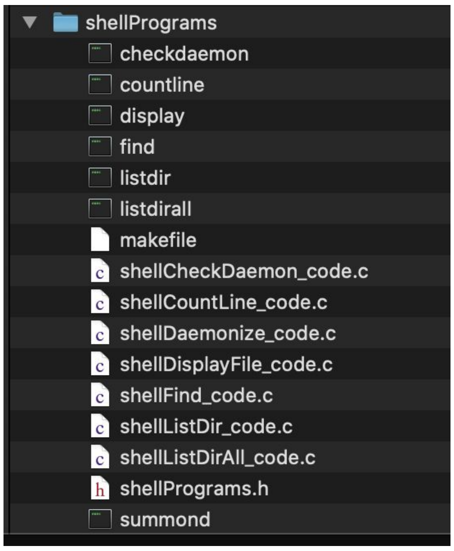


```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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] 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.

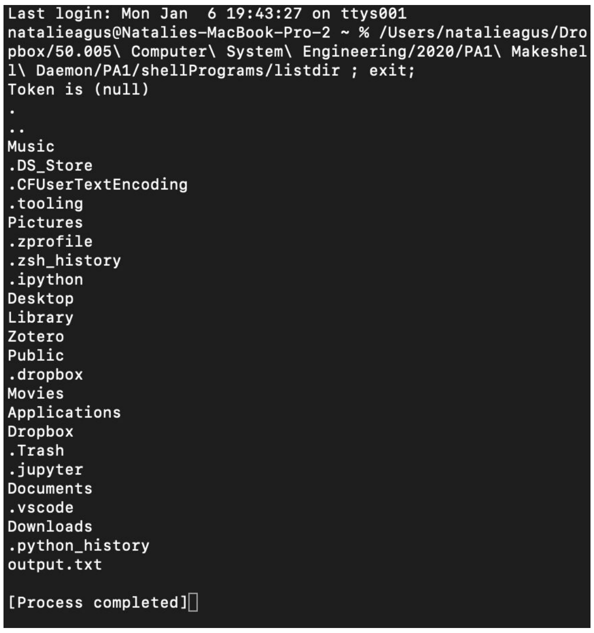
4. Change directory to `shellPrograms` folder
5. Type `make` and you shall have several new binaries as follows:



```
shellPrograms
    checkdaemon
    countline
    display
    find
    listdir
    listdirall
    makefile
    shellCheckDaemon_code.c
    shellCountLine_code.c
    shellDaemonize_code.c
    shellDisplayFile_code.c
    shellFind_code.c
    shellListDir_code.c
    shellListDirAll_code.c
    shellPrograms.h
    summond
```

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.

6. Try double clicking `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.



```
Last login: Mon Jan  6 19:43:27 on ttys001
natalieagus@Natalies-MacBook-Pro-2 ~ % /Users/natalieagus/Dro
pbox/50.005\ Computer\ System\ Engineering/2020/PA1\ Makeshel
l\ Daemon/PA1/shellPrograms/listdir ; exit;
Token is (null)
.
..
Music
.DS_Store
.CFUserTextEncoding
.tooling
Pictures
.zprofile
.zsh_history
.ipython
Desktop
Library
Zotero
Public
.dropbox
Movies
Applications
Dropbox
.Trash
.jupyter
Documents
.vscode
Downloads
.python_history
output.txt

[Process completed]
```

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.

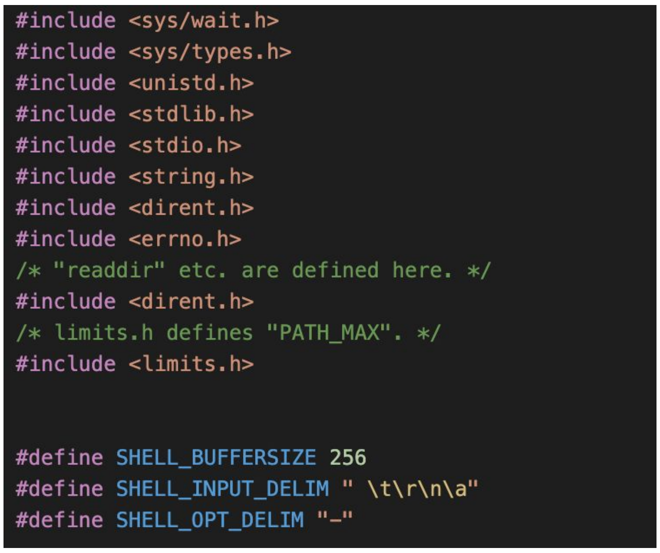
If all is well, congratulations you have finished setting up all the required starter code. You may proceed to Part 2. **DO NOT change the any of these makefiles.** Your answer should be compilable with the original makefile. Programs with compilation error will **receive zero.**

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.

Lets go through the header file first.

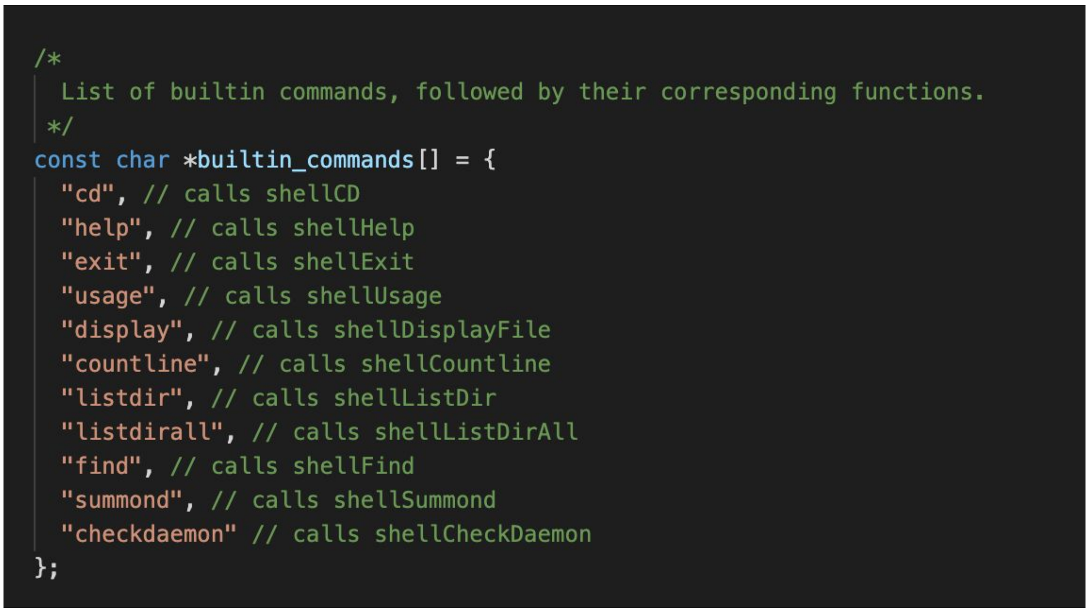
1. Open shell.h and you shall see the following in the first few lines:



```
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>
/* "readdir" etc. are defined here. */
#include <dirent.h>
/* limits.h defines "PATH_MAX". */
#include <limits.h>


#define SHELL_BUFFERSIZE 256
#define SHELL_INPUT_DELIM " \t\r\n\a"
#define SHELL_OPT_DELIM "-"
```

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

2. 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.



```
/*
  List of builtin commands, followed by their corresponding functions.
 */
const char *builtin_commands[] = {
  "cd", // calls shellCD
  "help", // calls shellHelp
  "exit", // calls shellExit
  "usage", // calls shellUsage
  "display", // calls shellDisplayFile
  "countline", // calls shellCountline
  "listdir", // calls shellListDir
  "listdirall", // calls shellListDirAll
  "find", // calls shellFind
  "summond", // calls shellSummond
  "checkdaemon" // calls shellCheckDaemon
};
```

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.

```
int numOfBuiltinFunctions() {
    return sizeof(builtin_commands) / sizeof(char *);
};
```

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

```
/*
The fundamental functions of the shell interface
*/
void shellLoop(void);
char **shellTokenizeInput(char *line);
char *shellReadLine(void);
int shellExecuteInput(char **args);
```

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

```
/*
Functions of the shell interface
*/
int shellCD(char **args);
int shellHelp(char **args);
int shellExit(char **args);
int shellUsage (char** args);
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);
```

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.

```
/**
  Read line from stdin, return it to the Loop function to tokenize it
 */
char *shellReadLine(void)
{
  /** TASK 1 **/
  // read one line from stdin using getline()

  // 1. Allocate a memory space to contain the string of input from stdin using malloc.
  Malloc should return a char* that persists even after this function terminates.
  // 2. Check that the char* returned by malloc is not NULL
  // 3. Fetch an entire line from input stream stdin using getline() function. getline()
   will store user input onto the memory location allocated in (1)
  // 4. Return the char*


   return NULL;

}
```

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

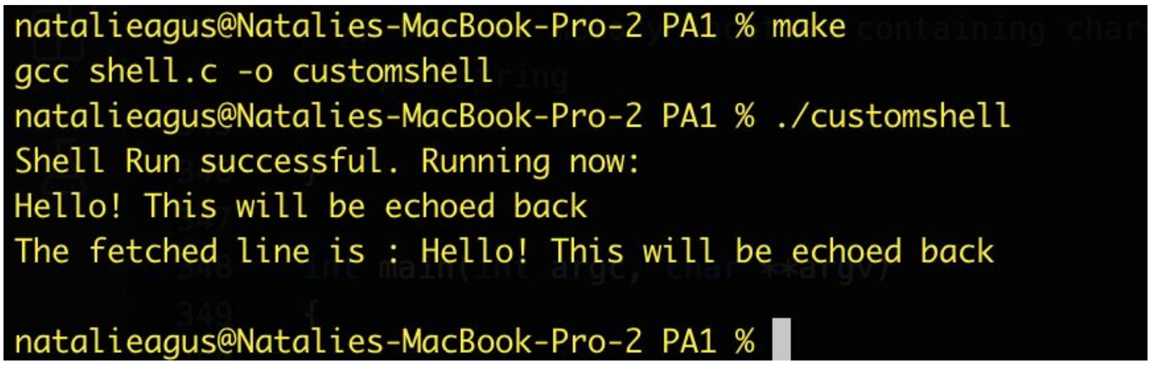
You don't need to actually worry if the input is larger than the size allocated by `*buffer`. This is because `getline` will automatically realloc pointer if buffer overflows. If `*buffer` was initially NULL, then `getline` will also automatically allocate a buffer before storing the line. Don't forget 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;
}
```

2. 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:



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
Hello! This will be echoed back
The fetched line is : Hello! This will be echoed back

natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

Task 2: Implement `shellTokenizeInput`

Next, you need to implement `shellTokenizeInput` function.

```c
/**
 Receives the *line, and return char** that tokenize the line
**/

char **shellTokenizeInput(char *line)
{

  /** TASK 2 **/
  // 1. Allocate a memory space to contain pointers (addresses) to the first character
  of each word in *line. Malloc should return char** that persists after the function
  terminates.
  // 2. Check that char ** that is returend by malloc is not NULL
  // 3. Tokenize the *line using strtok() function
  // 4. Return the char **
  return NULL;

}
```

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.

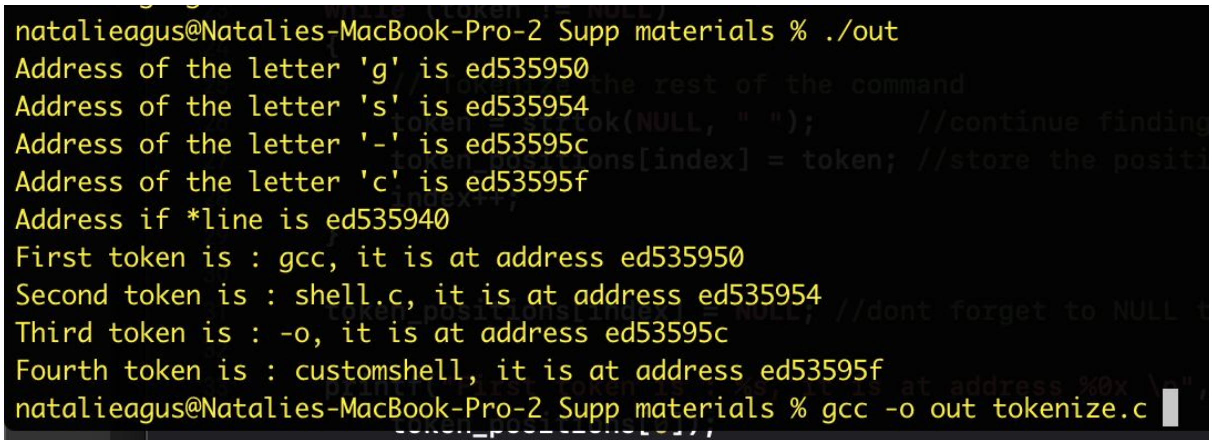
You must use `strtok()` to tokenize the input line. You can read on how to use it here: http://man7.org/linux/man-pages/man3/strtok.3.html

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

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```c
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]);
}
```

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):



```
natalieagus@Natalies-MacBook-Pro-2 Supp materials % ./out
Address of the letter 'g' is ed535950
Address of the letter 's' is ed535954
Address of the letter '-' is ed53595c
Address of the letter 'c' is ed53595f
Address if *line is ed535940
First token is : gcc, it is at address ed535950
Second token is : shell.c, it is at address ed535954
Third token is : -o, it is at address ed53595c
Fourth token is : customshell, it is at address ed53595f
natalieagus@Natalies-MacBook-Pro-2 Supp materials % gcc -o out tokenize.c
```

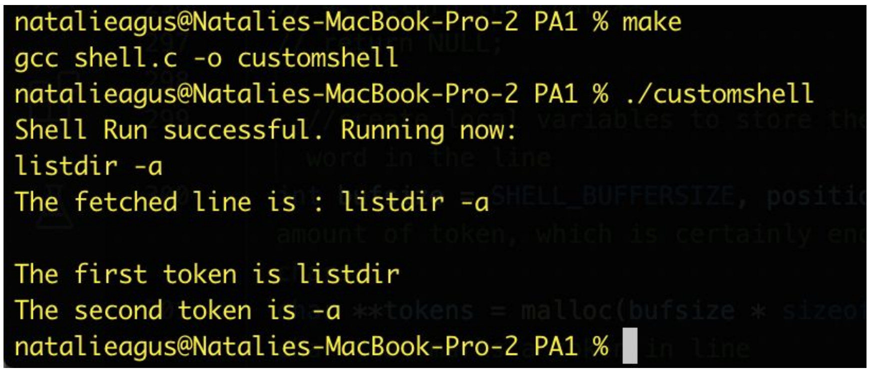
You should observe that the address of the letter ‘g’ that was manually printed in the first place should correspond to the token address ‘gcc’, and so on.

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;
}
```

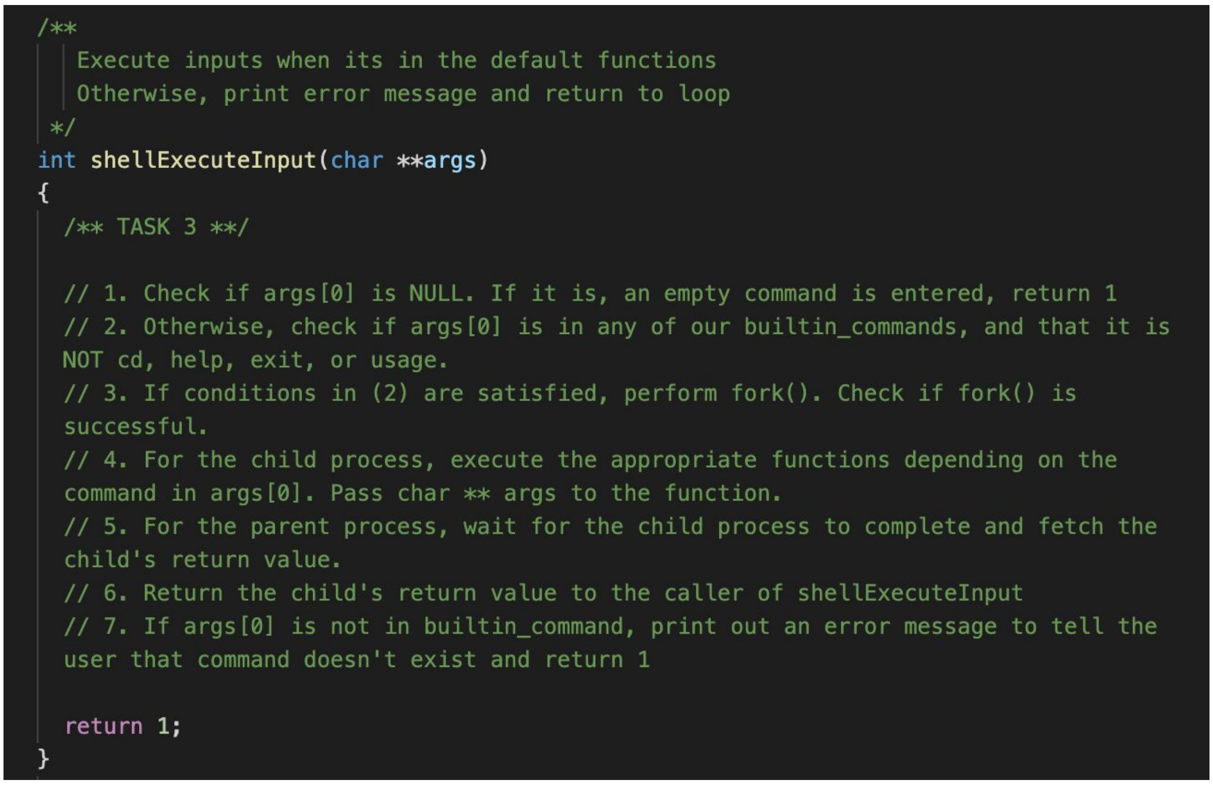
2. Compile and run it. Type the following input: `listdir -a`. You should see the following output if you have implemented the function successfully:



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
listdir -a
The fetched line is : listdir -a

The first token is listdir
The second token is -a
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

Task 3: Implement `shellExecuteInput`

Next, you need to implement `shellExecuteInput` function.



```c
/**
   Execute inputs when its in the default functions
   Otherwise, print error message and return to loop
 */
int shellExecuteInput(char **args)
{
  /** TASK 3 **/

  // 1. Check if args[0] is NULL. If it is, an empty command is entered, return 1
  // 2. Otherwise, check if args[0] is in any of our builtin_commands, and that it is
  NOT cd, help, exit, or usage.
  // 3. If conditions in (2) are satisfied, perform fork(). Check if fork() is
  successful.
  // 4. For the child process, execute the appropriate functions depending on the
  command in args[0]. Pass char ** args to the function.
  // 5. For the parent process, wait for the child process to complete and fetch the
  child's return value.
  // 6. Return the child's return value to the caller of shellExecuteInput
  // 7. If args[0] is not in builtin_command, print out an error message to tell the
  user that command doesn't exist and return 1

  return 1;
}
```

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, where pid is the pid of the child process returned by the fork(), stat_loc is a pointer to an integer used store the exit status of the child.

4. The child process has to invoke the right function depending on `args[0]`, and call `exit(1)` should the function invoked returns.

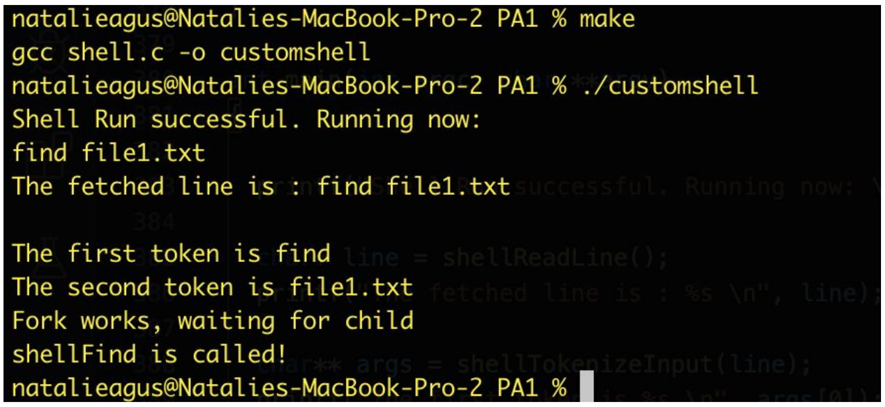
5. You can use `strcmp()` to compare between two strings, i.e: between `args[0]` and `builtin_commands[i]`. Its documentation can be found here: http://man7.org/linux/man-pages/man3/strcmp.3.html

To test:

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

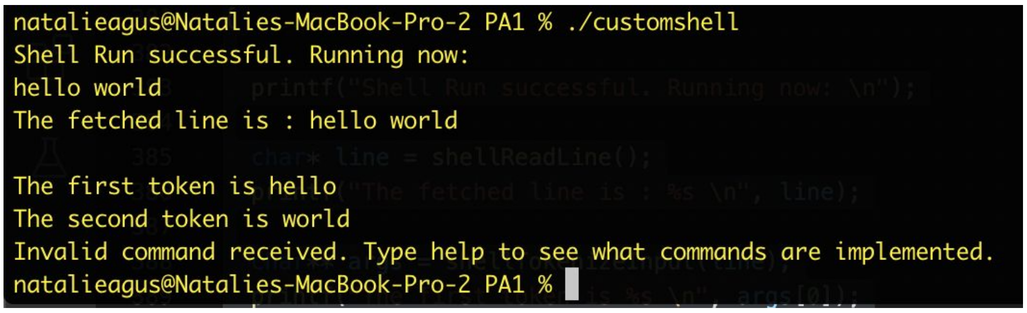
```c
int main(int argc, char **argv)
{

 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;
}
```

2. 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.**



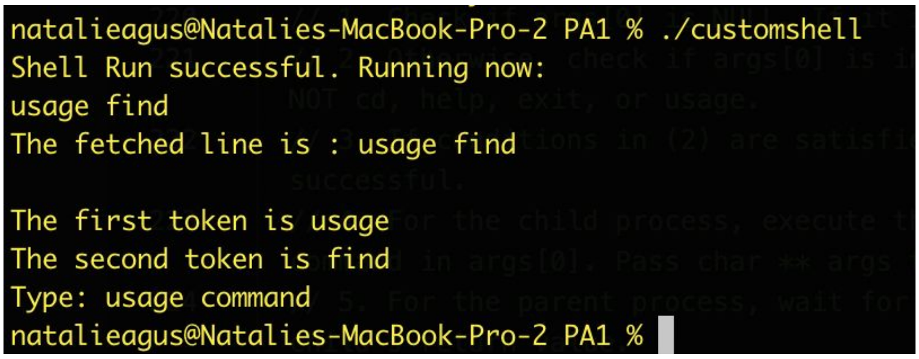
```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
find file1.txt
The fetched line is : find file1.txt

The first token is find
The second token is file1.txt
Fork works, waiting for child
shellFind is called!
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

3. 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.



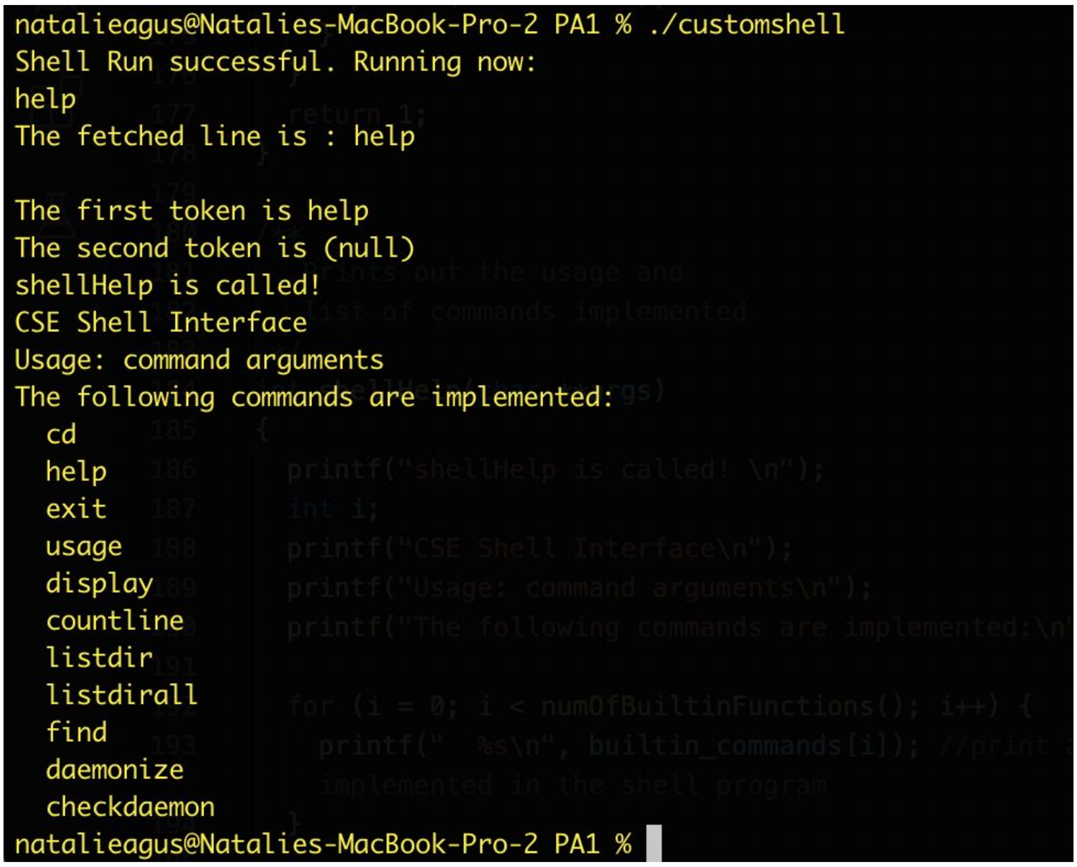
```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
hello world
The fetched line is : hello world

The first token is hello
The second token is world
Invalid command received. Type help to see what commands are implemented.
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

4. 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 exist in this test case.



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
usage find
The fetched line is : usage find

The first token is usage
The second token is find
Type: usage command
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

5. You can easily test with help as well:



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
help
The fetched line is : help

The first token is help
The second token is (null)
shellHelp is called!
CSE Shell Interface
Usage: command arguments
The following commands are implemented:
  cd
  help
  exit
  usage
  display
  countline
  listdir
  listdirall
  find
  daemonize
  checkdaemon
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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);
```

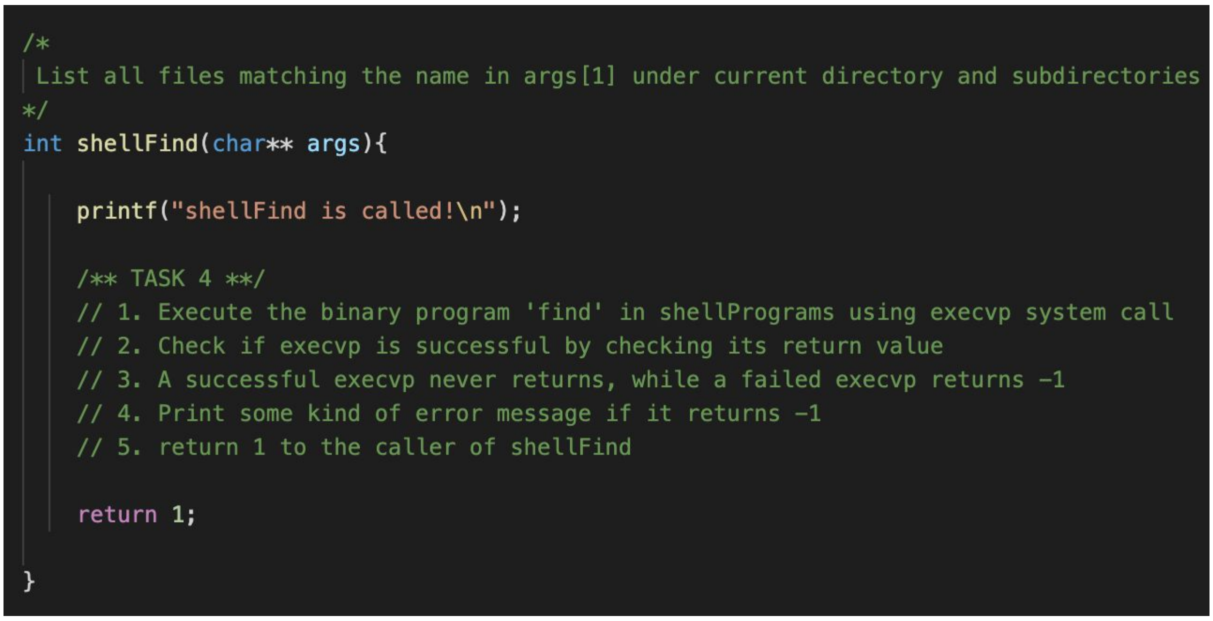
Notice how they don't involve any fork, as they must be performed in the shell's address space.

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);
```

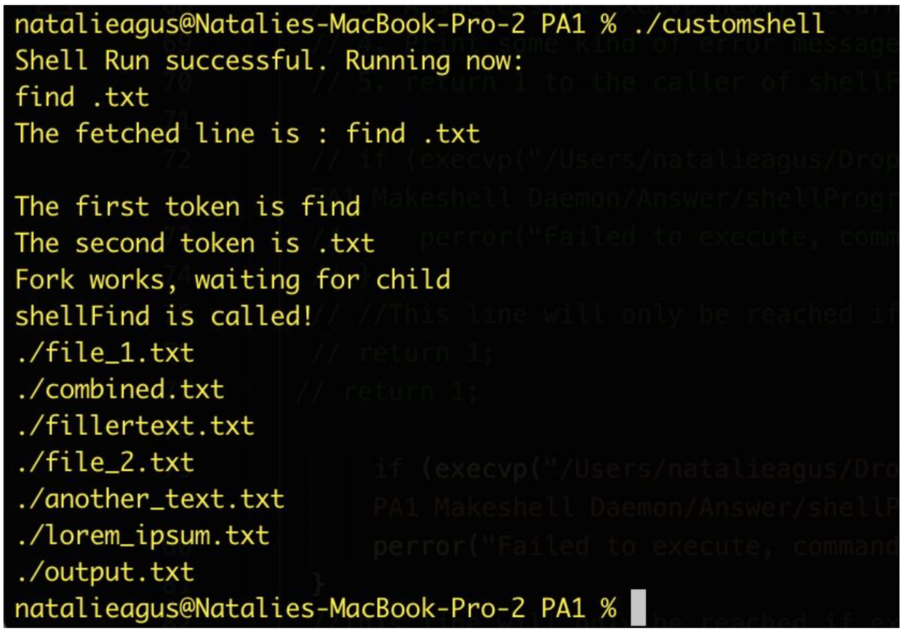
This is because we will need to *execute* the appropriate custom programs in shellPrograms to do what the user commands the shell to do. The implementation of these commands **are not** implemented within shell.c itself.

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



```
/*
 List all files matching the name in args[1] under current directory and subdirectories
*/
int shellFind(char** args){

    printf("shellFind is called!\n");

    /** TASK 4 **/
    // 1. Execute the binary program 'find' in shellPrograms using execvp system call
    // 2. Check if execvp is successful by checking its return value
    // 3. A successful execvp never returns, while a failed execvp returns -1
    // 4. Print some kind of error message if it returns -1
    // 5. return 1 to the caller of shellFind

    return 1;

}
```

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



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
find .txt
The fetched line is : find .txt

The first token is find
The second token is .txt
Fork works, waiting for child
shellFind is called!
./file_1.txt
./combined.txt
./fillertext.txt
./file_2.txt
./another_text.txt
./lorem_ipsum.txt
./output.txt
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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)
```

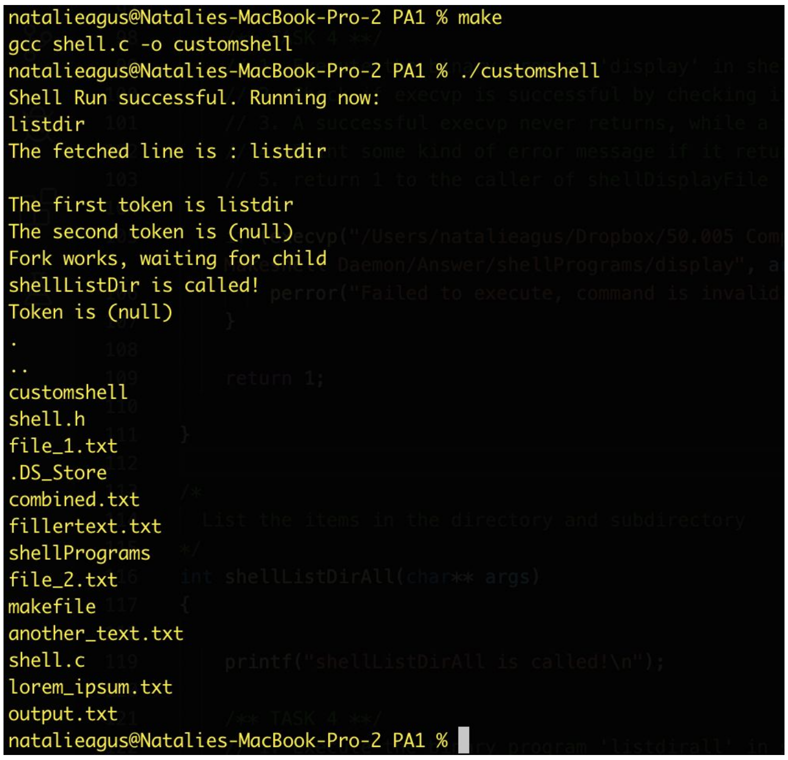
2. Read the manual on how to use execvp: https://linux.die.net/man/3/execvp

To test:

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

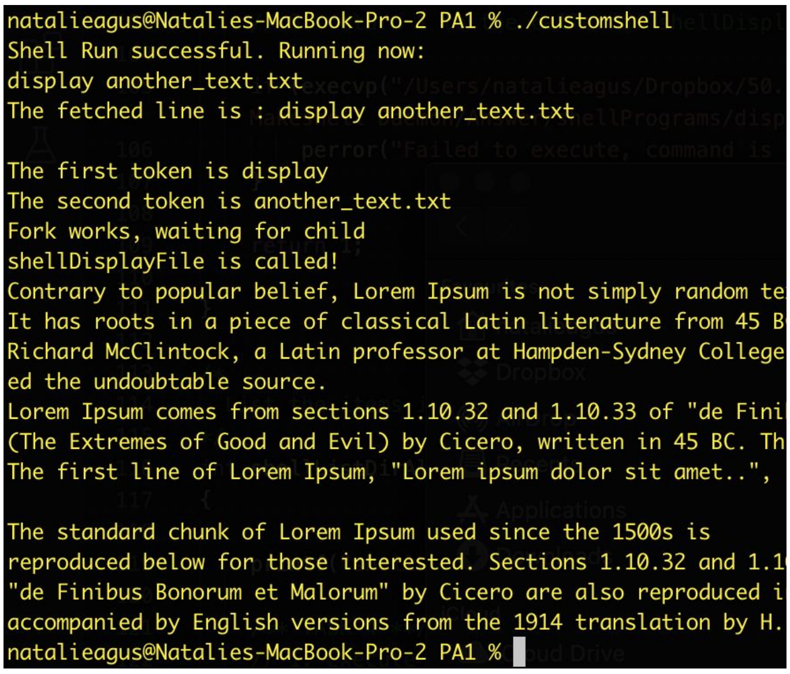
```
int main(int argc, char **argv)
{

 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;
}
```

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



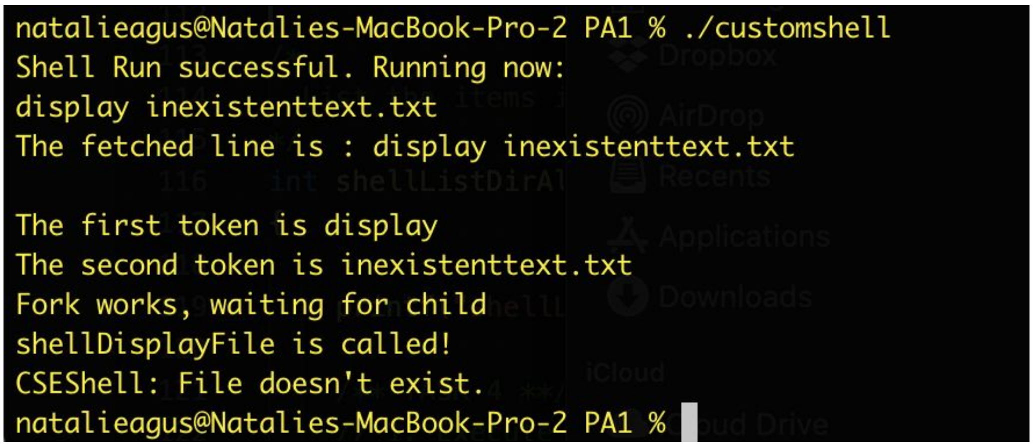
```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
listdir
The fetched line is : listdir

The first token is listdir
The second token is (null)
Fork works, waiting for child
shellListDir is called!
Token is (null)
.
..
customshell
shell.h
file_1.txt
.DS_Store
combined.txt
fillertext.txt
shellPrograms
file_2.txt
makefile
another_text.txt
shell.c
lorem_ipsum.txt
output.txt
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

3. The command `display another_text.txt` will display the content of the txt file:



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
display another_text.txt
The fetched line is : display another_text.txt

The first token is display
The second token is another_text.txt
Fork works, waiting for child
shellDisplayFile is called!
Contrary to popular belief, Lorem Ipsum is not simply random te
It has roots in a piece of classical Latin literature from 45 B
Richard McClintock, a Latin professor at Hampden-Sydney College
ed the undoubtable source.
Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Fini
(The Extremes of Good and Evil) by Cicero, written in 45 BC. Th
The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..",

The standard chunk of Lorem Ipsum used since the 1500s is
reproduced below for those interested. Sections 1.10.32 and 1.1
"de Finibus Bonorum et Malorum" by Cicero are also reproduced i
accompanied by English versions from the 1914 translation by H.
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

4. You must print some kind of error message if the execvp fails. For example, if we try to display a file that doesn't exist with `display inexsistenttext.txt`:



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
display inexistenttext.txt
The fetched line is : display inexistenttext.txt

The first token is display
The second token is inexistenttext.txt
Fork works, waiting for child
shellDisplayFile is called!
CSEShell: File doesn't exist.
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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.

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

Task 5: Looping the Shell

Of course a shell is not really a shell if it can only accepts one command and exit. You would expect the shell to prompt you with 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.

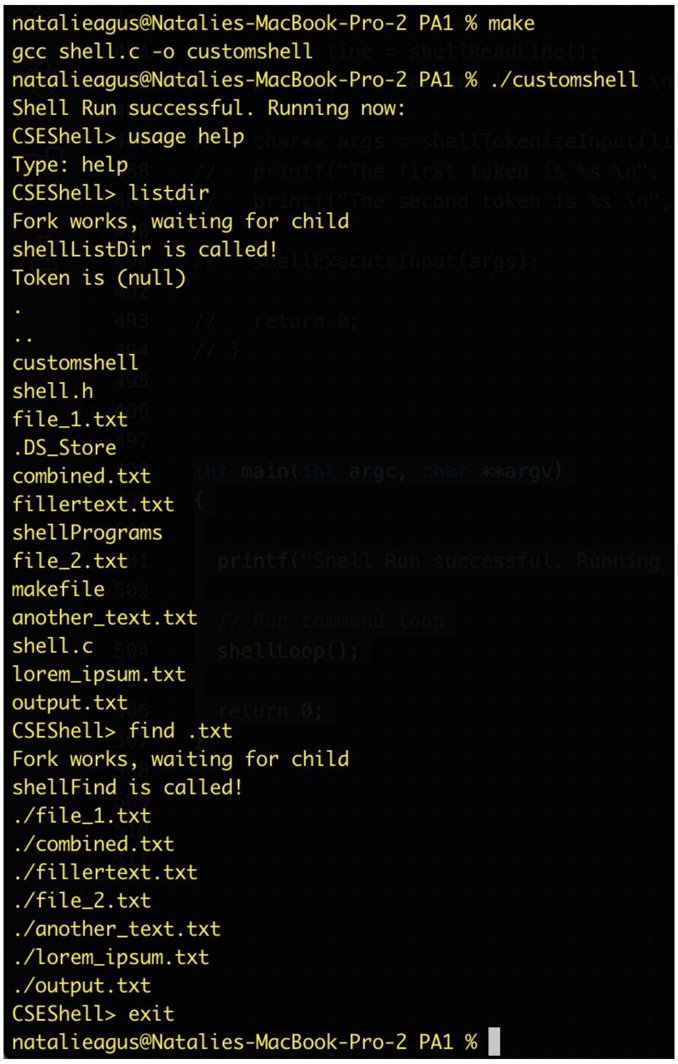
```
/**
  The main loop where one reads line,
  tokenize it, and then executes the command
 */
void shellLoop(void)
{
  //instantiate local variables
  char *line;  // to accept the line of string from user
  char **args; // to tokenize them as arguments separated by spaces
  int status;  // to tell the shell program whether to terminate shell or not

  /** TASK 4 **/
  //write a loop where you do the following:

  // 1. print the message prompt
  // 2. clear the buffer and move the output to the console using fflush
  // 3. clear the buffer to accept a new string in readLine()
  // 4. invoke shellReadLine() and store the output at line
  // 5. invoke shellTokenizeInput(line) and store the output at args**
  // 6. execute the tokens using shellExecuteInput(args)

  // 7. free memory location containing the strings of characters
  // 8. free memory location containing char* to the first letter of each word in the input
  string
  // 9. check return value of shellExecuteInput. If 1, continue the loop (point 1) again and
  prompt for another input. Else, exit shell.


}
```

To test:

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

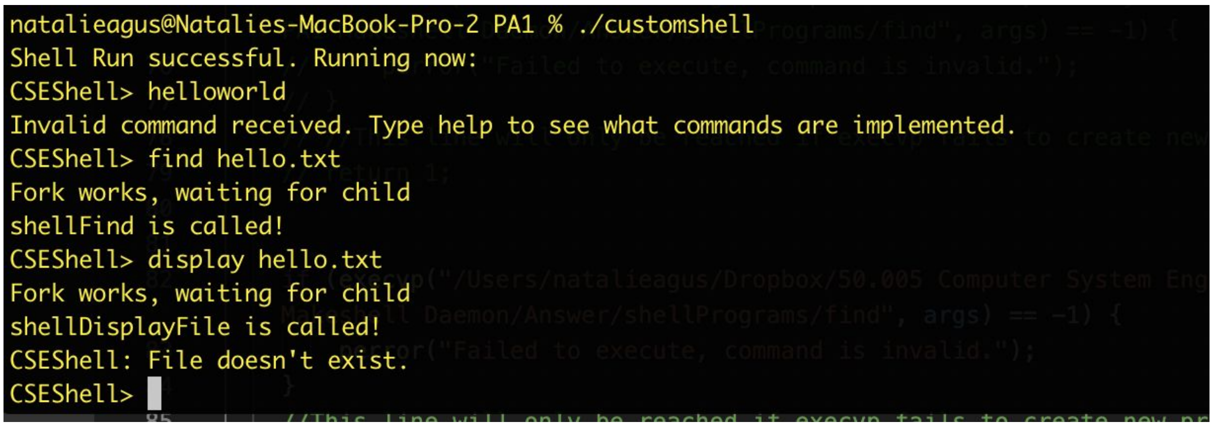
```
int main(int argc, char **argv)
{

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

 // Run command loop
 shellLoop();
 return 0;
}
```

2. 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.



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % make
gcc shell.c -o customshell
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
CSEShell> usage help
Type: help
CSEShell> listdir
Fork works, waiting for child
shellListDir is called!
Token is (null)
.
..
customshell
shell.h
file_1.txt
.DS_Store
combined.txt
fillertext.txt
shellPrograms
file_2.txt
makefile
another_text.txt
shell.c
lorem_ipsum.txt
output.txt
CSEShell> find .txt
Fork works, waiting for child
shellFind is called!
./file_1.txt
./combined.txt
./fillertext.txt
./file_2.txt
./another_text.txt
./lorem_ipsum.txt
./output.txt
CSEShell> exit
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

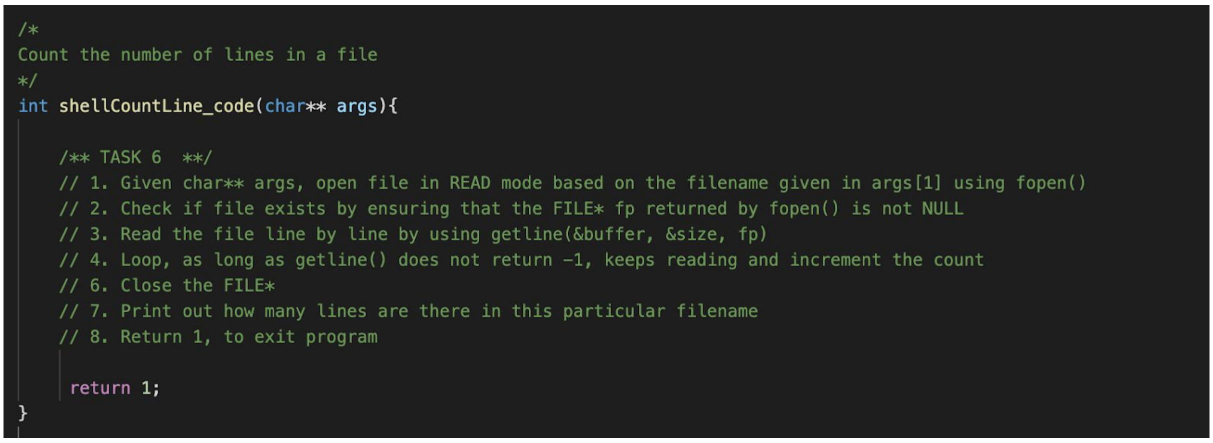
3. It also should be able to print some kind of error message upon encountering illegal command, and present the user with new prompt (it does not crash or exit):



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
CSEShell> helloworld
Invalid command received. Type help to see what commands are implemented.
CSEShell> find hello.txt
Fork works, waiting for child
shellFind is called!
CSEShell> display hello.txt
Fork works, waiting for child
shellDisplayFile is called!
CSEShell: File doesn't exist.
CSEShell>
```

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.



```c
/*
Count the number of lines in a file
*/
int shellCountLine_code(char** args){

    /** TASK 6  **/
    // 1. Given char** args, open file in READ mode based on the filename given in args[1] using fopen()
    // 2. Check if file exists by ensuring that the FILE* fp returned by fopen() is not NULL
    // 3. Read the file line by line by using getline(&buffer, &size, fp)
    // 4. Loop, as long as getline() does not return -1, keeps reading and increment the count
    // 6. Close the FILE*
    // 7. Print out how many lines are there in this particular filename
    // 8. Return 1, to exit program

     return 1;
}
```

You need to implement this program from scratch and are NOT allowed to use 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

3. 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 character terminated by '`\n`').

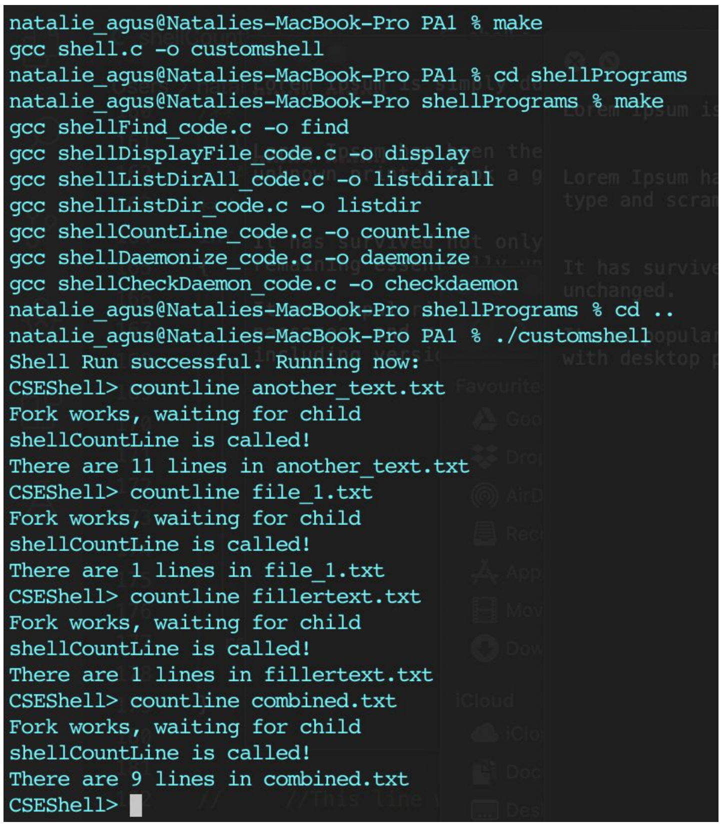
   `getline` returns -1 whenever we encounter the end of file, otherwise, it returns the number of characters in that line. An empty line "`\n`" will be returned as 1. A line "`hello\n`" will be returned as 6. Hence the terminating character is counted when `getline` returns.

   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

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

To test:



```
natalie_agus@Natalies-MacBook-Pro PA1 % make
gcc shell.c -o customshell
natalie_agus@Natalies-MacBook-Pro PA1 % cd shellPrograms
natalie_agus@Natalies-MacBook-Pro shellPrograms % make
gcc shellFind_code.c -o find
gcc shellDisplayFile_code.c -o display
gcc shellListDirAll_code.c -o listdirall
gcc shellListDir_code.c -o listdir
gcc shellCountLine_code.c -o countline
gcc shellDaemonize_code.c -o daemonize
gcc shellCheckDaemon_code.c -o checkdaemon
natalie_agus@Natalies-MacBook-Pro shellPrograms % cd ..
natalie_agus@Natalies-MacBook-Pro PA1 % ./customshell
Shell Run successful. Running now:
CSEShell> countline another_text.txt
Fork works, waiting for child
shellCountLine is called!
There are 11 lines in another_text.txt
CSEShell> countline file_1.txt
Fork works, waiting for child
shellCountLine is called!
There are 1 lines in file_1.txt
CSEShell> countline fillertext.txt
Fork works, waiting for child
shellCountLine is called!
There are 1 lines in fillertext.txt
CSEShell> countline combined.txt
Fork works, waiting for child
shellCountLine is called!
There are 9 lines in combined.txt
CSEShell> 
```

1. Recompile the files you have in `shellPrograms` by typing `make`

2. Go one directory up to PA1 folder using `cd ..`

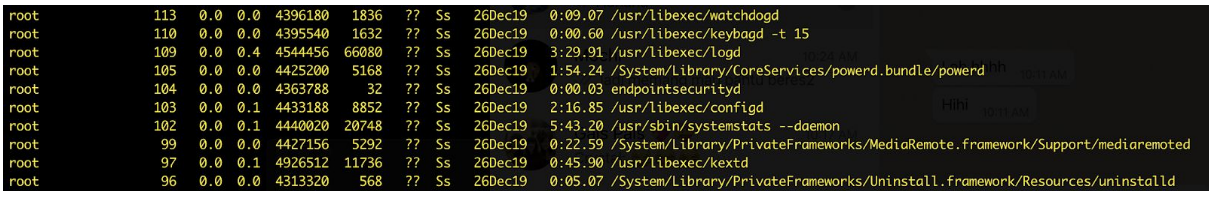
3. Run `./customshell`, and test countline with several text files given to you

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] 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'.



```
root              113   0.0  0.0  4396180   1836   ??  Ss    26Dec19    0:09.07 /usr/libexec/watchdogd
root              110   0.0  0.0  4395540   1632   ??  Ss    26Dec19    0:00.60 /usr/libexec/keybagd -t 15
root              109   0.0  0.4  4544456  66080   ??  Ss    26Dec19    3:29.91 /usr/libexec/logd
root              105   0.0  0.0  4425200   5168   ??  Ss    26Dec19    1:54.24 /System/Library/CoreServices/powerd.bundle/powerd
root              104   0.0  0.0  4363788     32   ??  Ss    26Dec19    0:00.03 endpointsecurityd
root              103   0.0  0.1  4433188   8852   ??  Ss    26Dec19    2:16.85 /usr/libexec/configd
root              102   0.0  0.1  4440020  20748   ??  Ss    26Dec19    5:43.20 /usr/sbin/systemstats --daemon
root               99   0.0  0.0  4427156   5292   ??  Ss    26Dec19    0:22.59 /System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremoted
root               97   0.0  0.1  4926512  11736   ??  Ss    26Dec19    0:45.90 /usr/libexec/kextd
root               96   0.0  0.0  4313320    568   ??  Ss    26Dec19    0:05.07 /System/Library/PrivateFrameworks/Uninstall.framework/Resources/uninstalld
```

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:

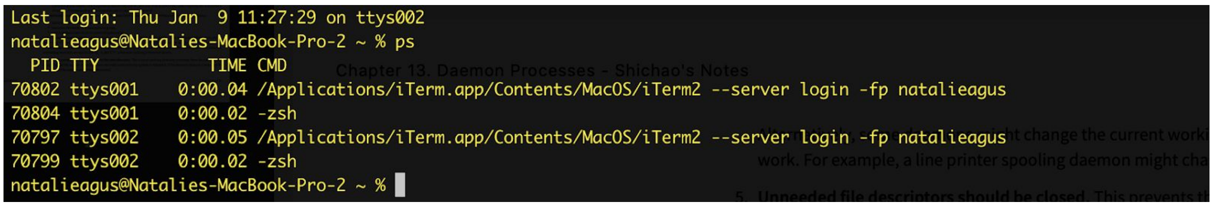
1. It as no controlling terminal. By definition, a daemon process does not require direct user interaction and therefore must detach itself from any controlling terminal.

   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

[2] A Terminal is your interface to the underlying operating system via a shell, usually bash. Most programs you invoke using your shell

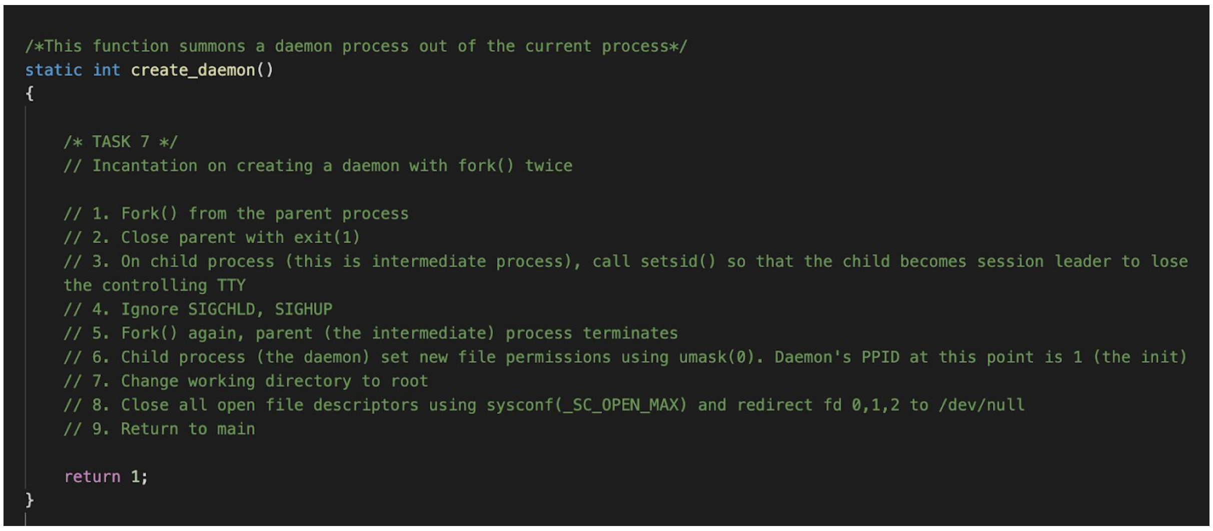
terminal windows are opened at the time this ps command is entered): ttys001 and ttys002, each controlling an instance of -zsh shell process.



```
Last login: Thu Jan  9 11:27:29 on ttys002
natalieagus@Natalies-MacBook-Pro-2 ~ % ps
  PID TTY           TIME CMD
70802 ttys001    0:00.04 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp natalieagus
70804 ttys001    0:00.02 -zsh
70797 ttys002    0:00.05 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp natalieagus
70799 ttys002    0:00.02 -zsh
natalieagus@Natalies-MacBook-Pro-2 ~ %
```

2. 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.
3. The working directory of the daemon process is typically the root directory
4. It closes all unneeded file descriptors[3]
5. Close and redirect fd 0, 1, and 2 to /dev/null
6. It logs messages through a central logging facilities: the `BSD syslog`

Write your answer in `/shellPrograms/shellDaemonize_code.c`:



```c
/*This function summons a daemon process out of the current process*/
static int create_daemon()
{

    /* TASK 7 */
    // Incantation on creating a daemon with fork() twice

    // 1. Fork() from the parent process
    // 2. Close parent with exit(1)
    // 3. On child process (this is intermediate process), call setsid() so that the child becomes session leader to lose
    the controlling TTY
    // 4. Ignore SIGCHLD, SIGHUP
    // 5. Fork() again, parent (the intermediate) process terminates
    // 6. Child process (the daemon) set new file permissions using umask(0). Daemon's PPID at this point is 1 (the init)
    // 7. Change working directory to root
    // 8. Close all open file descriptors using sysconf(_SC_OPEN_MAX) and redirect fd 0,1,2 to /dev/null
    // 9. Return to main

    return 1;
}
```

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] 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.

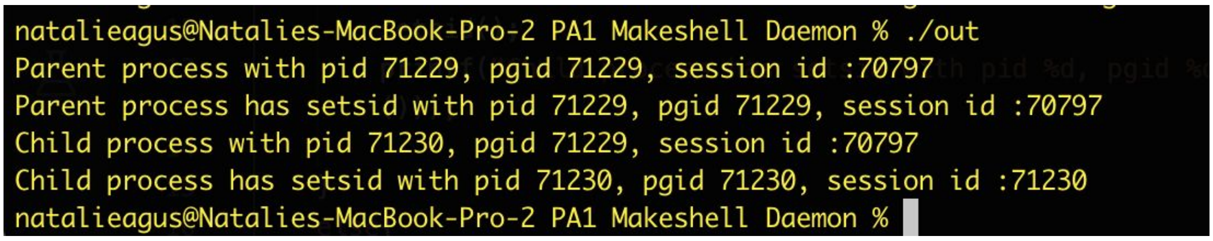
3. The child (intermediate process) -- that is not a process group leader -- calls setsid() to be the session leader and lose controlling TTY (terminal). setsid() is only effective when called by a process that is not a process group leader. The fork() in step 1 ensures this. The system call setsid() 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
4. Don't forget to change the path in order for the program properly.

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:

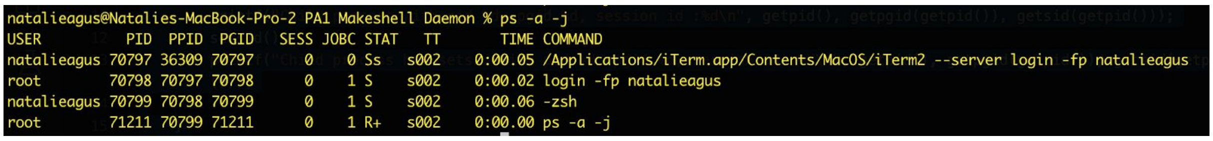


```
natalieagus@Natalies-MacBook-Pro-2 PA1 Makeshell Daemon % ./out
Parent process with pid 71229, pgid 71229, session id :70797
Parent process has setsid with pid 71229, pgid 71229, session id :70797
Child process with pid 71230, pgid 71229, session id :70797
Child process has setsid with pid 71230, pgid 71230, session id :71230
natalieagus@Natalies-MacBook-Pro-2 PA1 Makeshell Daemon %
```

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.

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



```
natalieagus@Natalies-MacBook-Pro-2 PA1 Makeshell Daemon % ps -a -j
USER           PID  PPID  PGID   SESS JOBC STAT   TT       TIME COMMAND
natalieagus 70797 36309 70797      0    0 Ss    s002    0:00.05 /Applications/iTerm.app/Contents/MacOS/iTerm2 --server login -fp natalieagus
root        70798 70797 70798      0    1 S     s002    0:00.02 login -fp natalieagus
natalieagus 70799 70798 70799      0    1 S     s002    0:00.06 -zsh
root        71211 70799 71211      0    1 R+    s002    0:00.00 ps -a -j
```

Now both 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.

5. Ignore `SIGCHLD` and `SIGHUP`:

```
signal(SIGCHLD, SIG_IGN);
signal(SIGHUP, SIG_IGN);
```

The reasons are as follows:

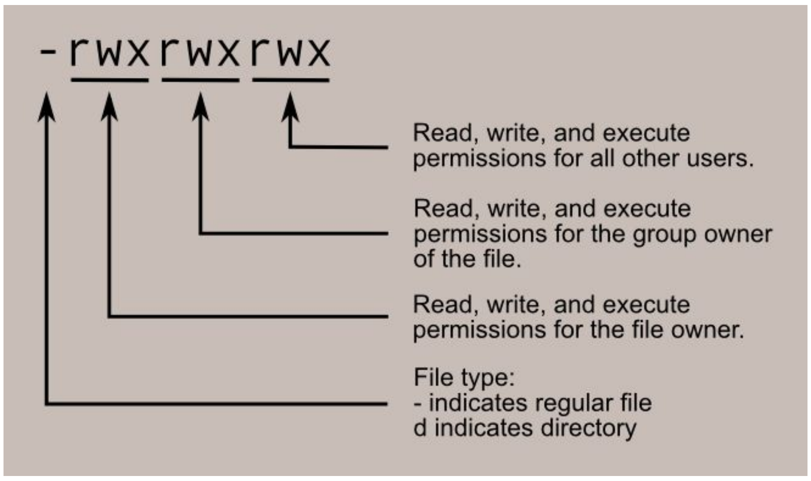
- Ignore `SIGCHLD`: This intermediate process is going to `fork()` one more time to create the daemon process. By ignoring `SIGCHLD`, our daemon process -- the child of this intermediate process *will not be a zombie process*. When the daemon exits, it is reaped immediately. However, the daemon will 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.

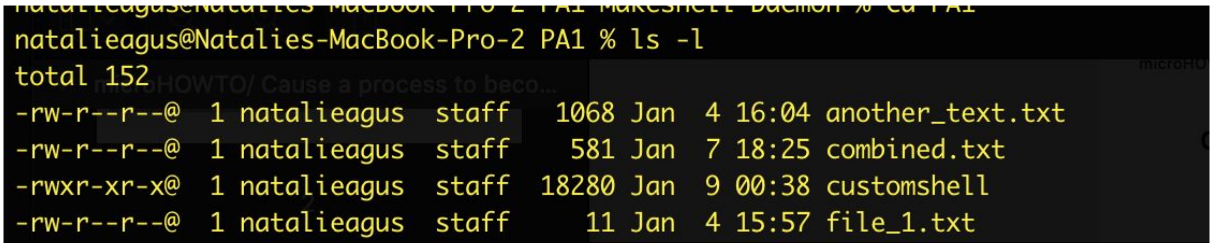
6. Perform the second `fork(),` of which the child of this intermediate process **is** the daemon process:
   - The second `fork`, is again useful for allowing the parent process to terminate. **This ensures that the child process is not a session leader.** Since a daemon has no controlling terminal, 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.

7. Set new file permissions using `umask(0).` Setting the umask to 0 means that newly created files or directories created will have no privileges initially revoked. In other words, a umask of zero will cause all files to be created as 0666 or world-writable. 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 a supplementary material. Unix file permssion has the following format:


-rwxrwxrwx
Read, write, and execute permissions for all other users.
Read, write, and execute permissions for the group owner of the file.
Read, write, and execute permissions for the file owner.
File type:
- indicates regular file
d indicates directory

You can view them by typing ls -l.



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ls -l
total 152
-rw-r--r--@  1 natalieagus  staff   1068 Jan  4 16:04 another_text.txt
-rw-r--r--@  1 natalieagus  staff    581 Jan  7 18:25 combined.txt
-rwxr-xr-x@  1 natalieagus  staff  18280 Jan  9 00:38 customshell
-rw-r--r--@  1 natalieagus  staff     11 Jan  4 15:57 file_1.txt
```

From the above example, its 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 a mask of `0666` (umask is kind of the "opposite"). `0666` is actually translated into mask binary: `0 110 110 110`, which means we will have `- rw- rw- rw-`, equivalent to *global writeable*. If we want to restrict permission of write to only the owner, we can set `umask(022)` -- equivalent to having a mask of `0644`, with the binary: `0 110 100 100`, which means we will have `- rw- r-- r--,` similar to the screenshot of the textfiles above.

8. 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.

9. Close all open file descriptors and redirect stdin, stdout, and stderr (fd 0, 1, and 2 by default in UNIX systems) to /dev/null.

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):

Integer value	Name	<`unistd.h`> symbolic constant[1]	<`stdio.h`> file stream[2]
0	Standard input	`STDIN_FILENO`	`stdin`
1	Standard output	`STDOUT_FILENO`	`stdout`
2	Standard error	`STDERR_FILENO`	`stderr`

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.

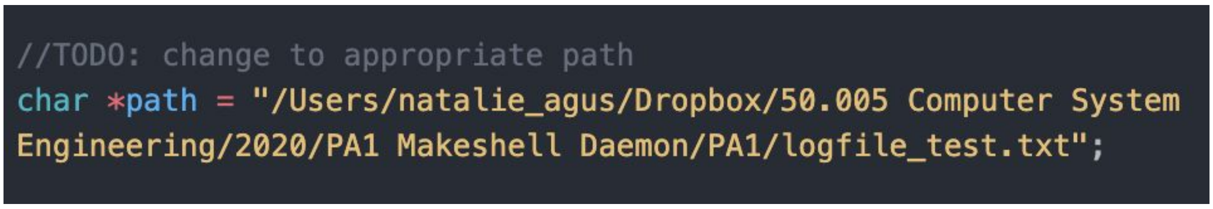
There are two reasons for not leaving them closed:
1. To prevent code that refers to these file descriptors from failing
2. To prevent the descriptors from being reused when we call `open()` from the daemon's code.

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);
```

10. 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.**



```
//TODO: change to appropriate path
char *path = "/Users/natalie_agus/Dropbox/50.005 Computer System
Engineering/2020/PA1 Makeshell Daemon/PA1/logfile_test.txt";
```

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.

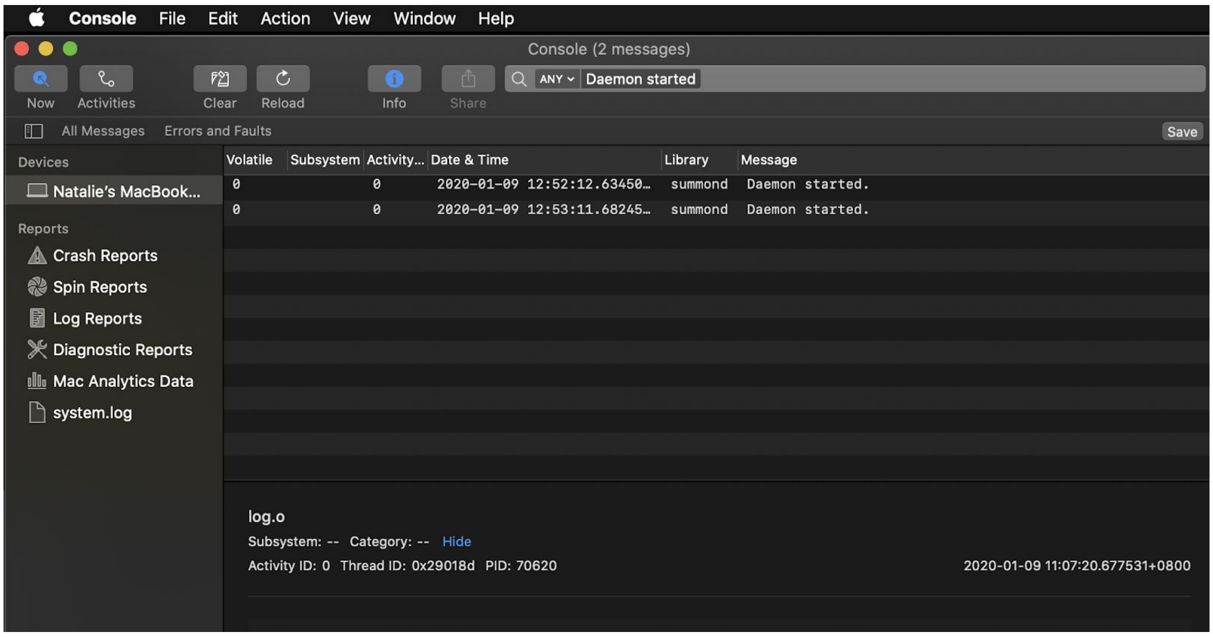
It cannot (simply) write to:

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 facility.* 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.



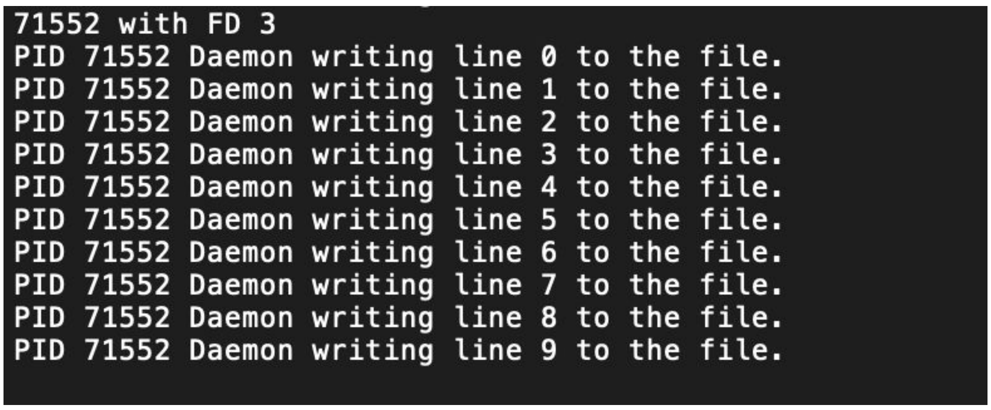
```
Volatile  Subsystem  Activity...  Date & Time                   Library   Message
0                    0            2020-01-09 12:52:12.63450…    summond   Daemon started.
0                    0            2020-01-09 12:53:11.68245…    summond   Daemon started.
```

log.o
Subsystem: -- Category: -- Hide
Activity ID: 0 Thread ID: 0x29018d PID: 70620 2020-01-09 11:07:20.677531+0800

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.

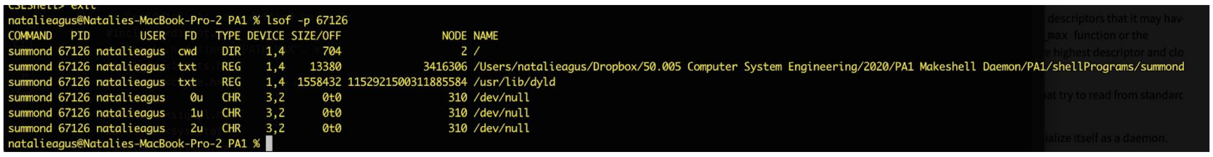
To test:

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:



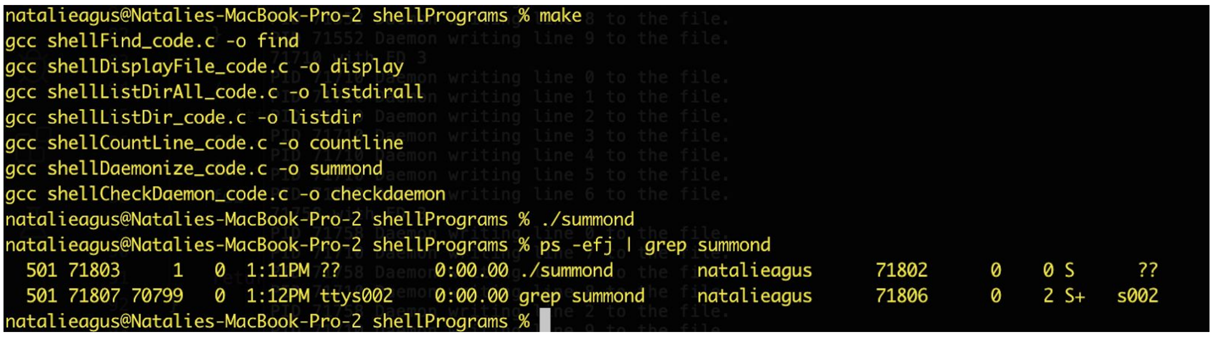
```
71552 with FD 3
PID 71552 Daemon writing line 0 to the file.
PID 71552 Daemon writing line 1 to the file.
PID 71552 Daemon writing line 2 to the file.
PID 71552 Daemon writing line 3 to the file.
PID 71552 Daemon writing line 4 to the file.
PID 71552 Daemon writing line 5 to the file.
PID 71552 Daemon writing line 6 to the file.
PID 71552 Daemon writing line 7 to the file.
PID 71552 Daemon writing line 8 to the file.
PID 71552 Daemon writing line 9 to the file.
```

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



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % lsof -p 67126
COMMAND   PID        USER   FD   TYPE DEVICE SIZE/OFF                NODE NAME
summond 67126 natalieagus  cwd    DIR    1,4      704                   2 /
summond 67126 natalieagus  txt    REG    1,4    13380             3416306 /Users/natalieagus/Dropbox/50.005 Computer System Engineering/2020/PA1 Makeshell Daemon/PA1/shellPrograms/summond
summond 67126 natalieagus  txt    REG    1,4  1558432 1152921500311885584 /usr/lib/dyld
summond 67126 natalieagus   0u    CHR    3,2      0t0                 310 /dev/null
summond 67126 natalieagus   1u    CHR    3,2      0t0                 310 /dev/null
summond 67126 natalieagus   2u    CHR    3,2      0t0                 310 /dev/null
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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

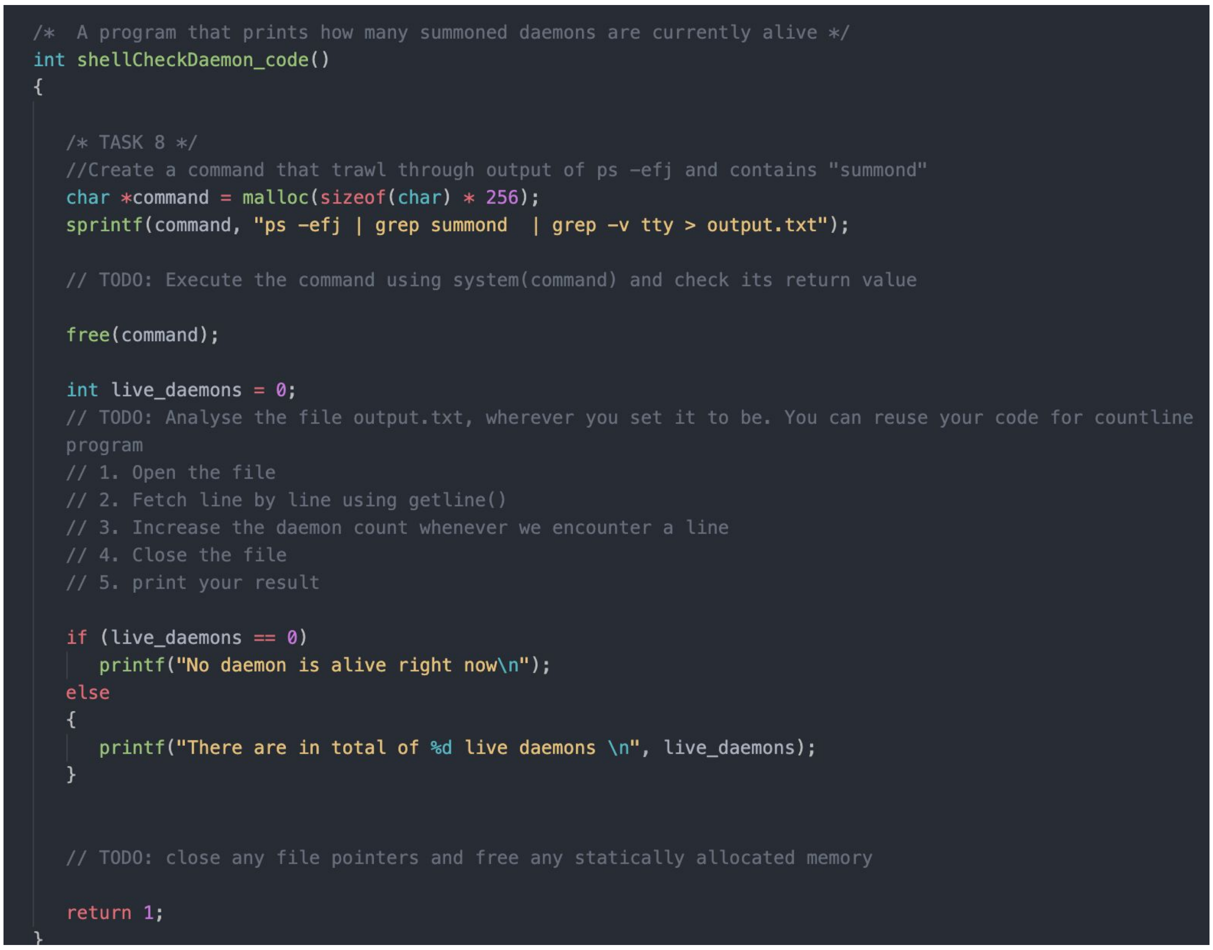


```
natalieagus@Natalies-MacBook-Pro-2 shellPrograms % make
gcc shellFind_code.c -o find
gcc shellDisplayFile_code.c -o display
gcc shellListDirAll_code.c -o listdirall
gcc shellListDir_code.c -o listdir
gcc shellCountLine_code.c -o countline
gcc shellDaemonize_code.c -o summond
gcc shellCheckDaemon_code.c -o checkdaemon
natalieagus@Natalies-MacBook-Pro-2 shellPrograms % ./summond
natalieagus@Natalies-MacBook-Pro-2 shellPrograms % ps -efj | grep summond
  501 71803     1   0  1:11PM ??         0:00.00 ./summond      natalieagus      71802      0    0 S      ??
  501 71807 70799   0  1:12PM ttys002    0:00.00 grep summond   natalieagus      71806      0    2 S+   s002
natalieagus@Natalies-MacBook-Pro-2 shellPrograms %
```

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`.



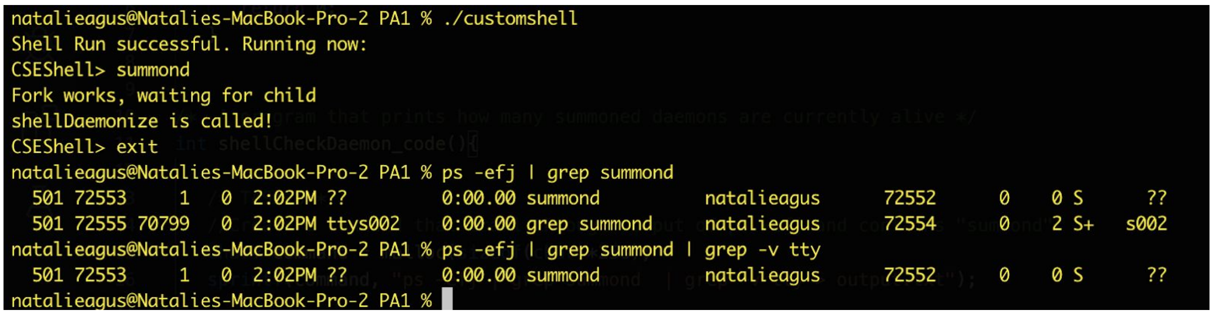
```c
/*  A program that prints how many summoned daemons are currently alive */
int shellCheckDaemon_code()
{

    /* TASK 8 */
    //Create a command that trawl through output of ps -efj and contains "summond"
    char *command = malloc(sizeof(char) * 256);
    sprintf(command, "ps -efj | grep summond  | grep -v tty > output.txt");

    // TODO: Execute the command using system(command) and check its return value

    free(command);

    int live_daemons = 0;
    // TODO: Analyse the file output.txt, wherever you set it to be. You can reuse your code for countline
    program
    // 1. Open the file
    // 2. Fetch line by line using getline()
    // 3. Increase the daemon count whenever we encounter a line
    // 4. Close the file
    // 5. print your result

    if (live_daemons == 0)
        printf("No daemon is alive right now\n");
    else
    {
        printf("There are in total of %d live daemons \n", live_daemons);
    }


    // TODO: close any file pointers and free any statically allocated memory

    return 1;
}
```

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`.

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



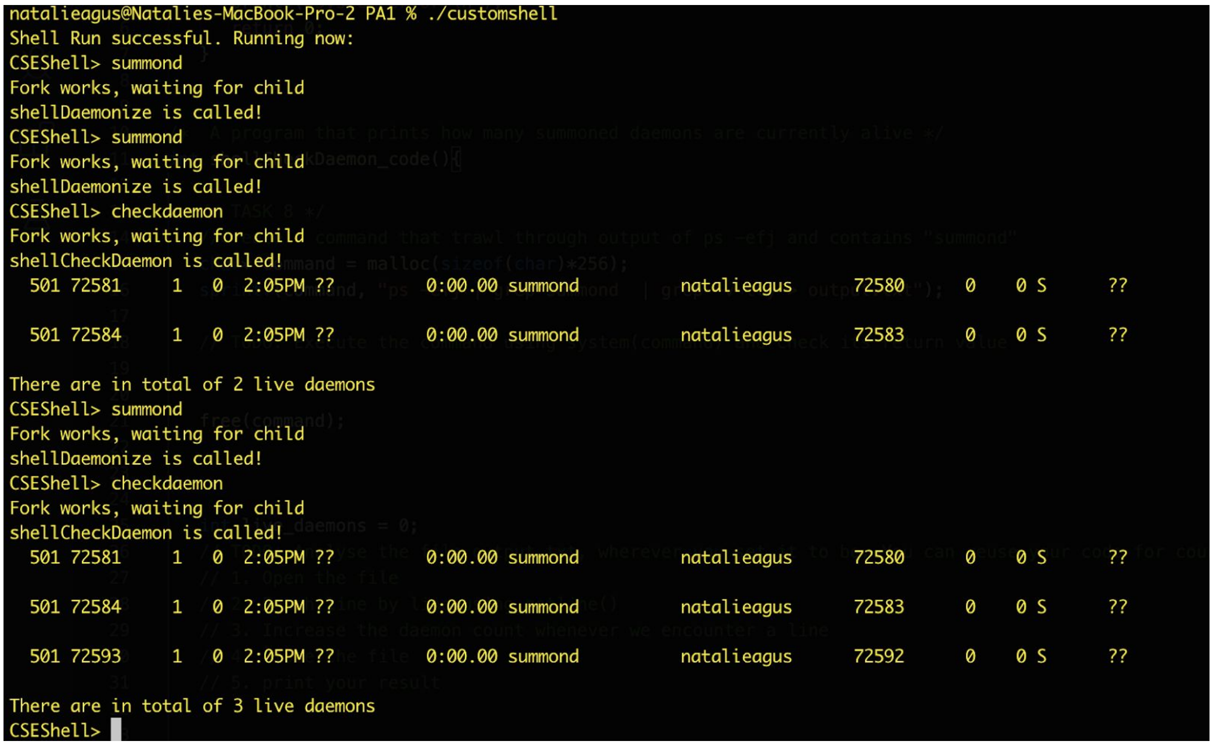
```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
CSEShell> summond
Fork works, waiting for child
shellDaemonize is called!
CSEShell> exit
natalieagus@Natalies-MacBook-Pro-2 PA1 % ps -efj | grep summond
  501 72553     1   0  2:02PM ??         0:00.00 summond          natalieagus      72552      0    0 S      ??
  501 72555 70799   0  2:02PM ttys002    0:00.00 grep summond     natalieagus      72554      0    2 S+    s002
natalieagus@Natalies-MacBook-Pro-2 PA1 % ps -efj | grep summond | grep -v tty
  501 72553     1   0  2:02PM ??         0:00.00 summond          natalieagus      72552      0    0 S      ??
natalieagus@Natalies-MacBook-Pro-2 PA1 %
```

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

3. 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`.
4. 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.



```
natalieagus@Natalies-MacBook-Pro-2 PA1 % ./customshell
Shell Run successful. Running now:
CSEShell> summond
Fork works, waiting for child
shellDaemonize is called!
CSEShell> summond
Fork works, waiting for child
shellDaemonize is called!
CSEShell> checkdaemon
Fork works, waiting for child
shellCheckDaemon is called!
  501 72581     1   0  2:05PM ??         0:00.00 summond          natalieagus      72580      0    0 S      ??

  501 72584     1   0  2:05PM ??         0:00.00 summond          natalieagus      72583      0    0 S      ??

There are in total of 2 live daemons
CSEShell> summond
Fork works, waiting for child
shellDaemonize is called!
CSEShell> checkdaemon
Fork works, waiting for child
shellCheckDaemon is called!
  501 72581     1   0  2:05PM ??         0:00.00 summond          natalieagus      72580      0    0 S      ??

  501 72584     1   0  2:05PM ??         0:00.00 summond          natalieagus      72583      0    0 S      ??

  501 72593     1   0  2:05PM ??         0:00.00 summond          natalieagus      72592      0    0 S      ??

There are in total of 3 live daemons
CSEShell>
```

2. 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 for submission procedure.