Discipline of Information Technology,

Media and Communications

College of Arts, Business, Law and Social Sciences

ICT374 ASSIGNMENT ONE

CHECK LIST

Surname: Lim

Given Names: Wen Chao

Student Number: CT0360379/34368872

Tutor's Name: Leng Kang

Assignment Due Date: 18/6/2022 **Date Submitted:** 15/6/2022

Your assignment should meet the following requirements. Please confirm this (by ticking boxes) before submitting your assignment.

- [✓] I have read and understood the Documentation Requirements
- [✓] **This assignment submission is compliant to the Documentation Requirements.**
- [✓] I have included all relevant Linux source code, executables and test files in the tar archive. The file names are chosen according to the assignment specification.
- [✓] I have kept another copy of this assignment and associated programs and files in a safe place.

The unit coordinator may choose to use your submission as sample solutions to be viewed by other students, but only with your permission. Please indicate whether you give permission for this to be done.

- ☐ Yes, I am willing to have my submission without change be made public as a sample solution.
- [✓] Yes, I am willing to have my submission be made public as a sample solution, as long as my submission is edited to remove all mentions of my identity.
- ☐ No, I am not willing to have my submission made public.

Table of Contents

List of files

File Name	Purpose
Assignment1.pdf	Documentation and answers for assignment 1
q2	Folder containing all file related to question 2 of assignment 1
q3	Folder containing all file related to question 3 of assignment 1
main.c	Source code for question 2
executor	Linux executable for question 2
myls.c	Source code for question 3
myls	Linux executable for question 3

Question 1: Comparing the Cost of Output with Three I/O Models

1) what are the main differences between the three I/O models and what are their strength and weakness?

Programmed I/O	**Interrupt-driven I/O**	**Direct Memory Access (DMA)**
I/O module do not interrupt Processor	I/O module will interrupt Processor	I/O module will interrupt Processor
Processor responsible for reading/writing data to/from main memory	Processor responsible for reading/writing data to/from main memory	*I/O module* responsible for reading/writing data to/from main memory
Required constant status checking of I/O module	Eliminates constant status check of I/O	Eliminates constant status check of I/O module and
Require Processor to process every data that passes between it and the I/O module	Require Processor to process every data that passes between it and the I/O module	Delegates the task of reading and writing data to I/O or DMA module
Inefficient because processor is kept needlessly busy with the constant status check	More efficient than Programmed I/O without the constant status check	Most efficient when transferring large amount of data. Slightly inefficient when transferring small amount of data as the processor would have to wait for DMA to read/write data to/from the main memory.

The three I/O models are Programmed I/O, Interrupt-driven I/O and Direct Memory Access (DMA).

The main differences between the three I/O models are how much the processor is responsible for.

For the programmed I/O, the processor is responsible for reading and writing to/from main memory. The processor would also have to constantly check the I/O module for task completion as the I/O module would not interrupt the processor once task is completed. This model makes programmed I/O inefficient.

For Interrupt-driven I/O, the processor is responsible for reading and writing to/from main memory, but the processor would not be checking the I/O module for task completion. The I/O module would instead interrupt the processor once the task is done. This model makes Interrupt-driven I/O more efficient than the programmed I/O model.

For Direct Memory Access (DMA), the processor is not responsible for both the reading and writing to/from the main memory and checking for task completion. The processor would simply send the task information to the I/O module and allow it permission to read or write to/from the main memory. Once the task is done, the I/O module would interrupt the processor like in Interrupt-driven I/O.

DMA model is very efficient when handling large amount of data but is slightly less efficient when it must handle many small data. This is because the processor allowed the I/O module read write permission to the main memory but there is only one bus shared between the processor and the I/O module, this forces them to compete for the bus usage and the processor might have to wait for the I/O module.

2) Assuming that an application needs to output 1000 words from the internal memory to the hard disk, calculate the following values for for each I/O model:

- **How many times the processor is interrupted?**
- **How many times the internal memory is read by the processor for those 1000 words?**
- **How many times the disk controller is read by the processor?**
- **How many times the disk controller is written to by the processor?**

Draw a table to contain your answers. You need to justify the numbers you put in the table.

	Programmed I/O	**Interrupt-driven I/O**	**Direct Memory Access (DMA)**
Times processor is Interrupted	0	1000	1
Times internal memory is read by processor	1000	1000	0
Times disk controller is read by processor	0	0	0
Times disk controller is written by processor	1000	1000	1

Programmed I/O

The processor does not get interrupted as it is the processor's responsibility to check on the I/O module, so the processor gets interrupted 0 times. The internal memory is read by the processor 1000 times as words are read one at a time as the data bus could only transfer 1 word worth of data at a time, the word is then passed to the I/O module. The disk controller is read 0 times by the processor because there is no need for the processor to read the disk controller when it is trying to write to hard disk. The disk controller is written 1000 times by the processor because only 1 word is read at a time from the internal memory.

Interrupt-driven I/O

The processor will be interrupted 1000 times for each word that the processor gives to the I/O module to process. The internal memory is read 1000 times by the processor as only 1 word can be transferred at a time through the data bus. The disk controller is not read by the processor as there is no need for the processor to read the disk controller when writing to disk. The disk controller is written by the processor 1000 times for each word that is read by the processor from the internal memory.

DMA

The processor is interrupted only 1 time for the entire 1000 words as the processor would be passing the instructions for writing the 1000 words to hard disk to the I/O module and allowing the I/O module to read from the internal memory and writing to the hard disk by itself. The internal memory is not read by the processor because the I/O module would be the one that is reading the 1000-word data based on the instructions given by the processor. The disk controller is not read by the processor as we are not doing a read of the hard disk. The disk controller is written only 1 time to pass the instructions for the writing the 1000 words to hard disk.

Question 2: Executing Commands in Child Processes

Write a C program that takes a list of command line arguments, each of which is the full path of a command (such as /bin/ls, /bin/ps, /bin/date, /bin/who, /bin/uname etc). Assume the number of such commands is *N*, your program would then create *N* direct child processes (ie, the parent of these child processes is the same original process), each of which executing one of the *N* commands. You should make sure that these *N* commands are executed concurrently, not sequentially one after the other. The parent process should be waiting for each child process to terminate. When a child process terminates, the parent process should print one line on the standard output stating that the relevant command has completed successfully or not successfully (such as "Command /bin/who has completed successfully", or "Command /bin/who has not completed successfully"). Once all of its child processes have terminated, the parent process should print "All done, bye-bye!" before it itself terminates.

Note: do not use function system, or any other function that will invoke a shell program in this question.

Self-diagnosis and evaluation

The program can:

- Take multiple command line arguments as full path of a command and execute each of them in a direct child process.
- Executed each command concurrently and will print success or failure correspondingly.
- Print all done only when all child processes are terminated from the parent process.
- Warn and exit if no argument is provided.

The program do not use any function that will invoke a shell program.

The program do not have any unfulfilled requirements.

Test Evidence

1) Check that for each arguments have a direct child process created and are ran simultaneously

   Purpose:
   This test case is to ensure the program creates a direct child process for each command provided in argument and are ran simultaneously. In addition, this test case will ensure the program can take multiple arguments.
   Note that for this test case each child is made to sleep for 30 seconds to ensure the test result can be obtained.

   Output from program:

```
$ ./executor /bin/ls /bin/ps /bin/date /bin/who /bin/uname
    PID TTY          TIME CMD
   2387 pts/0    00:00:00 bash
Tue 14 Jun 2022 01:47:26 AM +08
   2500 pts/0    00:00:00 executor
   2501 pts/0    00:00:00 executor
   2502 pts/0    00:00:00 ps
   2504 pts/0    00:00:00 executor
Command /bin/date has been completed successfully
   2505 pts/0    00:00:00 executor
Command /bin/ps has been completed successfully
ct0360379 tty2         2022-06-14 00:53 (tty2)
Linux
Command /bin/uname has been completed successfully
Command /bin/who has been completed successfully
executor  main.c  temp.txt
Command /bin/ls has been completed successfully
All done, bye-bye
```

Output from ps:

```
$ ps -efH | grep executor
ct03603+    2500    2387  0 01:46 pts/0    00:00:00         ./executor
/bin/ls /bin/ps /bin/date /bin/who /bin/uname
ct03603+    2501    2500  0 01:46
pts/0    00:00:00           ./executor /bin/ls /bin/ps /bin/date
/bin/who /bin/uname
ct03603+    2502    2500  0 01:46
pts/0    00:00:00           ./executor /bin/ls /bin/ps /bin/date
/bin/who /bin/uname
ct03603+    2503    2500  0 01:46
pts/0    00:00:00           ./executor /bin/ls /bin/ps /bin/date
/bin/who /bin/uname
ct03603+    2504    2500  0 01:46
pts/0    00:00:00           ./executor /bin/ls /bin/ps /bin/date
/bin/who /bin/uname
ct03603+    2505    2500  0 01:46
pts/0    00:00:00           ./executor /bin/ls /bin/ps /bin/date
/bin/who /bin/uname
ct03603+    2508    2482  0 01:46 pts/1    00:00:00         grep --
color=auto executor
```

Result:
From the test result, we can see that there are 5 processes created that has the same parent process ID 2500 from the output from ps. This shows that the program created 5 direct child processes for the 5 command from argument and are ran simultaneously. The program is also proven to be able to take in multiple arguments.

2) Check that each process will run the command from the argument and will print success or failure accordingly

   Purpose:
   This test case will ensure that program can print success or failure of each command that is executed by child processes.

   Output by program:

```
$ ./executor /bin/ls /bin/ps /bin/dat
Command /bin/dat has error executing
Command /bin/dat has not been completed successfully
executor  main.c  temp.txt
Command /bin/ls has been completed successfully
    PID TTY          TIME CMD
   2387 pts/0    00:00:00 bash
   2658 pts/0    00:00:00 executor
   2660 pts/0    00:00:00 ps
Command /bin/ps has been completed successfully
All done, bye-bye
```

   Result:
   The test results shows that the program can print error and command not completed for command that did not executed successfully and can print completed successfully for command is executed successfully.

3) Parent process will print all done only when all child process is terminated

   Purpose:

   This test case is to ensure that the program will only print it is all done after all other processes are completed.

   Output from program:

```
$ ./executor /bin/ls /bin/ps /bin/date /bin/who /bin/uname
    PID TTY          TIME CMD
   2387 pts/0    00:00:00 bash
Tue 14 Jun 2022 01:47:26 AM +08
   2500 pts/0    00:00:00 executor
   2501 pts/0    00:00:00 executor
   2502 pts/0    00:00:00 ps
   2504 pts/0    00:00:00 executor
Command /bin/date has been completed successfully
   2505 pts/0    00:00:00 executor
Command /bin/ps has been completed successfully
ct0360379 tty2         2022-06-14 00:53 (tty2)
Linux
Command /bin/uname has been completed successfully
Command /bin/who has been completed successfully
executor  main.c  temp.txt
Command /bin/ls has been completed successfully
All done, bye-bye
```

Result:

The test result shows that only after all 5 commands was completed did the program print all done and exits.

4) Program will warn user if no argument is provided

Purpose:
This test case is to ensure the program will warn user and exit gracefully if no arguments are provided.

Output:

```
$ ./executor
No argument is provided
All done, bye-bye
```

Result:
The result shows that the program can recognise there was no argument given and exited after warning.

Source Code Listing

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
    const char* filename = "temp.txt";
    int i,count = argc;
    pid_t pid[argc];
    pid_t waitStatus = 0, wpid;
    FILE* file;
    char* line=NULL;
    size_t len=0;

        //Delete the temp file for storing the pid and command
        //temp file is used so that pid could be used to identify which
command is ran for a particular process
    remove(filename);
        //Check if argument was provided
    if(argc < 2)
    {
        printf("No argument is provided\n");
    }

        //Read command line from command line arguments Lab2 e8
    for (i = 1;i < argc;++i)
    {
            //For each line create a child process and run the command line
        pid[i-1] = fork();
        if (pid[i-1] == 0) // child
        {
                //create/open the temp file that will store child pid and
command
            FILE* file = fopen(filename, "a+");
            fprintf(file, "%d\n%s\n", getpid(), argv[i]);
            fclose(file);

            if(execl(argv[i],argv[i], (char *) 0) == -1)
            {
                printf("Command %s has error executing\n", argv[i]);
                exit(1);
            }
        }

        if(pid[i-1] < 0)   //error
        {
            printf("Command %s has encountered error while forking\n",
argv[i]);
        }

    }

    while(count > 1)
    {
        wpid = wait(&waitStatus);
            //Check if Process exited normally
            //Once command is completed print 'completed successfully' before
exiting
            //If command is not successful print 'not completed succuessfully'
            //Loops through tempfile to compare wpid
            //Once pid matches the next line is the command
        file = fopen(filename, "r");
        while(getline(&line, &len, file) != -1)
        {
            if (WIFEXITED(waitStatus))
            {
                if(WEXITSTATUS(waitStatus) == 0)
                {
                        //
                    if(atoi(line) == wpid)
                    {
                        getline(&line, &len, file);
                        line[strlen(line)-1] = '\0';
                        printf("Command %s has been completed successfully\n",
line);
                    }
                }
                else
                {
                    if(atoi(line) == wpid)
                    {
                        getline(&line, &len, file);
                        line[strlen(line)-1] = '\0';
                        printf("Command %s has not been completed
successfully\n", line);
                    }
                }
            }
            else
            {
                if(atoi(line) == wpid)
                {
                    getline(&line, &len, file);
                    line[strlen(line)-1] = '\0';
                    printf("Command %s has not been completed successfully\n",
line);
                }
            }

        }
        fclose(file);
        count--;
    }

        //print 'All done, bye-bye' once all child terminated before
terminating parent
    printf("All done, bye-bye\n");
    exit(0);
}
```

Question 3: Reporting Information of Files

Write a C program, myls.c, that is similar to the standard Unix utility ls -l (but with much less functionality). Specifically, it takes a list of command line arguments, treating each command line argument as a file name. It then reports the following information for each file:

1. **user name of the user owner (*hints: Stevens & Rago, 6.2.*);**
2. **group name of the group owner; (*hints: Stevens & Rago, 6.4.*);**
3. **the type of file;**
4. **full access permissions, reported in the format used by the ls program;**
5. **the size of the file;**
6. **i-node number;**
7. **the device number of the device in which the file is stored, including both major number and minor number (*hints: Stevens & Rago, 4.23.*);**
8. **the number of links;**
9. **last access time, converted to the format used by the ls program (*hint: Stevens & Rago, 6.10.*);**
10. **last modification time, converted to the format used by the ls program;**
11. **last time file status changed, converted to the format used by the ls program;**

Like ls -l command, if no command line argument is provided, the program simply reports the above information about the files in the current directory.

Of course, your program cannot use programs, such as /bin/ls, and /usr/bin/stat, that is able to report the file status in this question.

Self-diagnosis and evaluation

myls program can

- Take in multiple command line argument as file names, myls will get the files' information provided
- Take in no command line arguments, in which case, myls will instead get all files' information in current directory
- Take in invalid file names, in which case, myls will report the error
- Report the username of the user owner for each file
- Report the group name of the group owner for each file
- Report the type for each file
- Report the full access permissions in the same format used by ls program for each file
- Report the size for each file
- Report the i-node number for each file
- Report the major and minor device number for each file

- Report the number of links for each file
- Report the last access time in the same format used by ls program for each file
- Report the last modified time in the same format used by ls program for each file
- Report the last status change time in the same format used by ls program for each file

myls program do not have any requirements that it does not meet.

Test Evidence

1) Check that program can run both with or without arguments. Program will output information for all file in current directory if ran without arguments

   Purpose:
   This test case is to ensure the program can run with argument provided. Which will be read as a filename that it will get the information of. If no arguments are provided, the program will instead output the information of all files found in current directory. This test case will also ensure all output information are correct and formatted as intended.

   Output from myls when given single argument:

```
$ ./myls myls.c
Filename: myls.c
Username of user owner: root
Group name of group owner: vboxsf
File type: Regular file
File permission: rwxrwx---
File size: 7816
File I-node number: 40
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:06
Last modified time: Jun 13 01:12
Last status changed time: Jun 13 17:06
```

   Output from stat for the same file:

```
$ stat myls.c

File: myls.c
Size: 7816        Blocks: 16         IO Block: 4096   regular file
Device: 31h/49d Inode: 40          Links: 1
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:06:46.870418600 +0800
Modify: 2022-06-13 01:12:43.797201400 +0800
Change: 2022-06-13 17:06:46.870418600 +0800
```

```
Birth: -
```

Output from myls program without argument:

```
$ ./myls

Directory path: /media/sf_ICT374_OS_SP/Assignment 1/q3

FileName: .
Username of user owner: root
Group name of group owner: vboxsf
File type: Directory
File permission: rwxrwx---
File size: 0
File I-node number: 38
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:17
Last modified time: Jun 13 17:17
Last status changed time: Jun 13 17:17

FileName: ..
Username of user owner: root
Group name of group owner: vboxsf
File type: Directory
File permission: rwxrwx---
File size: 4096
File I-node number: 1
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:16
Last modified time: Jun 13 17:15
Last status changed time: Jun 13 17:15

FileName: myls
Username of user owner: root
Group name of group owner: vboxsf
File type: Regular file
File permission: rwxrwx---
File size: 17200
File I-node number: 45
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:17
Last modified time: Jun 13 17:17
Last status changed time: Jun 13 17:17

FileName: myls.c
Username of user owner: root
Group name of group owner: vboxsf
```

```
File type: Regular file
File permission: rwxrwx---
File size: 7818
File I-node number: 40
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:17
Last modified time: Jun 13 17:16
Last status changed time: Jun 13 17:16
```

Output from stat:

```
$ stat .

File: .
Size: 0             Blocks: 0          IO Block: 4096   directory
Device: 31h/49d Inode: 38          Links: 1
Access: (0770/drwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:17:41.331315700 +0800
Modify: 2022-06-13 17:17:21.530806800 +0800
Change: 2022-06-13 17:17:21.530806800 +0800
Birth: -

$ stat ..
File: ..
Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 31h/49d Inode: 1           Links: 1
Access: (0770/drwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:19:59.171177100 +0800
Modify: 2022-06-13 17:19:59.171177100 +0800
Change: 2022-06-13 17:19:59.171177100 +0800
Birth: -

$ stat myls
File: myls
Size: 17200         Blocks: 40         IO Block: 4096   regular file
Device: 31h/49d Inode: 45          Links: 1
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:17:24.680528900 +0800
Modify: 2022-06-13 17:17:21.567639200 +0800
Change: 2022-06-13 17:17:27.804942800 +0800
Birth: -

$ stat myls.c
File: myls.c
Size: 7818          Blocks: 16         IO Block: 4096   regular file
Device: 31h/49d Inode: 40          Links: 1
```

```
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid: 
(  999/  vboxsf)
Access: 2022-06-13 17:17:39.201231700 +0800
Modify: 2022-06-13 17:16:57.188707700 +0800
Change: 2022-06-13 17:16:59.132970600 +0800
Birth: -
```

Result:
From the test result, we can see that the all file information displayed by myls program is correct and matches what was shown from stat command.
The Date format are also changed to match what one would expect from a ls command as well.
myls program was also able to obtain and display the file information of all file in current directory when not given any arguments.

2) Program can take multiple arguments with each argument being a filename in current directory.

Purpose:
This test case is to ensure program is able to take in multiple arguments as filenames and obtain the file information all of them correctly

Output from myls when given 2 arguments:

```
$ ./myls myls myls.c
Filename: myls
Username of user owner: root
Group name of group owner: vboxsf
File type: Regular file
File permission: rwxrwx---
File size: 17200
File I-node number: 45
Device number(major/minor): 0/49
Number of links: 1
Last access time: Jun 13 17:17
Last modified time: Jun 13 17:17
Last status changed time: Jun 13 17:17

Filename: myls.c
Username of user owner: root
Group name of group owner: vboxsf
File type: Regular file
File permission: rwxrwx---
File size: 7818
File I-node number: 40
Device number(major/minor): 0/49
Number of links: 1
```

```
Last access time: Jun 13 17:17
Last modified time: Jun 13 17:16
Last status changed time: Jun 13 17:16
```

Output from stat:

```
$ stat myls
File: myls
Size: 17200         Blocks: 40         IO Block: 4096   regular file
Device: 31h/49d Inode: 45          Links: 1
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:17:24.680528900 +0800
Modify: 2022-06-13 17:17:21.567639200 +0800
Change: 2022-06-13 17:17:27.804942800 +0800
Birth: -

$ stat myls.c
File: myls.c
Size: 7818          Blocks: 16         IO Block: 4096   regular file
Device: 31h/49d Inode: 40          Links: 1
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid:
(  999/  vboxsf)
Access: 2022-06-13 17:17:39.201231700 +0800
Modify: 2022-06-13 17:16:57.188707700 +0800
Change: 2022-06-13 17:16:59.132970600 +0800
Birth: -
```

Result:
From the test result, we can see that myls was able to successfully and correctly obtain multiple file information based on provided filename.

3) Check that program will print error when the file name provided is incorrect

Purpose:
This test case will make sure the program prints error and gracefully exits when given an invalid file name argument.

Output from myls:

```
$ ./myls mels
lstat error: No such file or directory
```

Result:
From the test result we can confirm myls is able to know when a file name is invalid.

Source Code Listing

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdbool.h>
#include <sys/sysmacros.h>
#include <dirent.h>
#include <limits.h>


//Compare time to determine if they are recent.
//Time is recent if less than 6 months apart
bool IsTimeRecent(time_t t)
{
    time_t curTime = time(0);
    double timeDiff = difftime(t, curTime);

    if(timeDiff > 15778800 || timeDiff < -15778800)
    {
        return false;
    }
    else
    {
        return true;
    }
}

//Format Time
//If recent MMM DD HH:mm
//Not recent MMM DD YYYY
void FormatTime(time_t time, bool isRecent, char* timeBuf)
{
    //char* timeBuf;
    struct tm *timeinfo;

    timeinfo = localtime(&time);

    if(IsTimeRecent(time))
    {
        strftime(timeBuf, 13, "%b %d %H:%M", timeinfo);
        return;
    }
    else
    {
        strftime(timeBuf, 12, "%b %d %Y", timeinfo);
        return;
    }
}

// user name of the user owner (hints: Stevens & Rago, 6.2.);
void GetOwnerName(struct stat *buf)
{
    struct passwd *pwd;

    pwd = getpwuid(buf->st_uid);
    printf("Username of user owner: %s\n", pwd->pw_name);
    return;
}
// group name of the group owner; (hints: Stevens & Rago, 6.4.);
void GetGroupName(struct stat *buf)
{
    struct group * grp;

    grp = getgrgid(buf->st_gid);
    printf("Group name of group owner: %s\n", grp->gr_name);
    return;
}
// the type of file;
void GetFileType(struct stat *buf)
{
    if(S_ISREG(buf->st_mode))
    {
        printf("File type: Regular file");
    }
    else if(S_ISDIR(buf->st_mode))
    {
        printf("File type: Directory");
    }
    else if(S_ISCHR(buf->st_mode))
    {
        printf("File type: Character special");
    }
    else if(S_ISBLK(buf->st_mode))
    {
        printf("File type: Block special");
    }
    else if(S_ISFIFO(buf->st_mode))
    {
        printf("File type: FIFO");
    }
    else if(S_ISLNK(buf->st_mode))
    {
        printf("File type: Symbolic link");
    }
    else if(S_ISSOCK(buf->st_mode))
    {
        printf("File type: Socket");
    }
    else
    {
        printf("File type: **unknown file type");
    }

    printf("\n");
    return;
}
// full access permissions, reported in the format used by the ls program;
void GetPermission(struct stat *buf)
{
    printf("File permission: ");
        //User read write execute permission
    if(buf->st_mode & S_IRUSR)
    {
        printf("r");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IWUSR)
    {
        printf("w");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IXUSR)
    {
        printf("x");
    }
    else
    {
        printf("-");
    }

        //Group read write execute permission
    if(buf->st_mode & S_IRGRP)
    {
        printf("r");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IWGRP)
    {
        printf("w");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IXGRP)
    {
        printf("x");
    }
    else
    {
        printf("-");
    }

        //Others read write execute permission
    if(buf->st_mode & S_IROTH)
    {
        printf("r");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IWOTH)
    {
        printf("w");
    }
    else
    {
        printf("-");
    }
    if(buf->st_mode & S_IXOTH)
    {
        printf("x");
    }
    else
    {
        printf("-");
    }

    printf("\n");
    return;
}

// the size of the file;
void GetFileSize(struct stat *buf)
{
    printf("File size: %ld\n", buf->st_size);
    return;
}
// i-node number;
void GetFileINode(struct stat *buf)
{
    printf("File I-node number: %ld\n", buf->st_ino);
    return;
}
// the device number of the device in which the file is stored, including both
major number and minor number (hints: Stevens & Rago, 4.23.);
void GetDeviceNumber(struct stat *buf)
{
    printf("Device number(major/minor): %d/%d\n", major(buf->st_dev),
minor(buf->st_dev));
    return;
}

// the number of links;
void GetNumOfLinks(struct stat *buf)
{
    printf("Number of links: %ld\n", buf->st_nlink);
    return;
}
// last access time, converted to the format used by the ls program (hint:
Stevens & Rago, 6.10.);
void GetLastAccessTime(struct stat *buf)
{
    time_t accessTime = buf->st_atime;
    bool isRecent = IsTimeRecent(accessTime);
    char formattedTime[13];
    FormatTime(accessTime, isRecent, formattedTime);

    printf("Last access time: %s\n", formattedTime);
    return;
}
// last modification time, converted to the format used by the ls program;
void GetLastModifiedTime(struct stat *buf)
{
    time_t modifiedTime = buf->st_mtime;
    bool isRecent = IsTimeRecent(modifiedTime);
    char formattedTime[13];
    FormatTime(modifiedTime, isRecent, formattedTime);

    printf("Last modified time: %s\n", formattedTime);
    return;
}
// last time file status changed, converted to the format used by the ls
program;
void GetLastStatusChange(struct stat *buf)
{
    time_t statusChangeTime = buf->st_ctime;
    bool isRecent = IsTimeRecent(statusChangeTime);
    char formattedTime[13];
    FormatTime(statusChangeTime, isRecent, formattedTime);

    printf("Last status changed time: %s\n", formattedTime);
    return;
}


int main(int argc, char *argv[])
{
    struct stat buf;
    int i;
    char cwd[PATH_MAX];


    //Read argv
    //if argc == 1
    if(argc == 1)
    {
        //Get list of files in current directory
        if(getcwd(cwd, sizeof(cwd))!= NULL)
        {
            DIR *dp;
            struct dirent *dirp;

            dp = opendir(cwd);

                printf("Directory path: %s\n", cwd);

            while((dirp = readdir(dp)) != NULL)
            {
                //Get file information for each file
                if(lstat(dirp->d_name, &buf) < 0)
                {
                    perror("lstat error");
                    continue;
                }
                //Print file information
                printf("FileName: %s\n", dirp->d_name);
                GetOwnerName(&buf);
                GetGroupName(&buf);
                GetFileType(&buf);
                GetPermission(&buf);
                GetFileSize(&buf);
                GetFileINode(&buf);
                GetDeviceNumber(&buf);
                GetNumOfLinks(&buf);
                GetLastAccessTime(&buf);
                GetLastModifiedTime(&buf);
                GetLastStatusChange(&buf);
                printf("\n");
            }
            closedir(dp);

        }
        else
        {
            perror("getcwd() error");
            exit(1);
        }
    }
    //If argc > 1
    if(argc > 1)
    {
        for(i = 1; i < argc; i++)
        {
            //For each argv
            //Get file information
            if(lstat(argv[i], &buf) < 0)
            {
                perror("lstat error");
                continue;
            }
            //Print file information
            printf("Filename: %s\n", argv[i]);
            GetOwnerName(&buf);
            GetGroupName(&buf);
            GetFileType(&buf);
            GetPermission(&buf);
            GetFileSize(&buf);
            GetFileINode(&buf);
            GetDeviceNumber(&buf);
            GetNumOfLinks(&buf);
            GetLastAccessTime(&buf);
            GetLastModifiedTime(&buf);
            GetLastStatusChange(&buf);
            printf("\n");
        }
    }

    exit(0);
}
```