COMP356 Operating Systems Project

Introduction

The objective of the following project is to construct a rudimentary operating system with basic multiprogramming and time-sharing utility. The operating system runs on a 1.44MB floppy disk image, which is loaded with programs that make up the core of the operating system. Initially, the disk image contains a bootloader, that simply boots the system to run the kernel program. The kernel contains implementation of several system calls and runs a shell program that receives user input to handle and execute instructions specified by the user like a command line interface. The operating system implements a simple file system structure and offers process, and memory management support to allow processes to run in a secured environment. It responds to services requested by programs and perform basic tasks. The following report touches on the specific implementation of system calls the operating system handles and raises future improvements that can be made to enhance the program to include more convenient features.

Implementation

The kernel contains various system calls written by the group and calls on existing BIOS routines to handle software generated interrupts when necessary. A description of each system call function along with its specific implementation will be discussed in the following pages.

<u>PutChar() / PutStr()</u>

The first two functions implemented handle system calls to display either a character or an array of characters in the form of a string to the screen. This is achieved by calling the PutInMemory() function from the existing kernel.asm file that directly connects to the video memory and stores the desired output to the screen. The functions take in arguments that specifies the color, text, and the desired location of the output on the screen.

BIOS Routines

The next set of functions invoke BIOS routines by making system calls to generate interrupt service routines through the interrupt() function from the existing kernel.asm file. This function takes in parameters that specify the type of interrupt generated and other necessary parameters for calling the specified interrupt.

<u>PrintInt()/ PrintString()</u>

These two functions achieve the same effect to the putChar()/putStr() functions above and prints a specific string or integer to the screen at the cursor location. The functions generate interrupt0x10 and prints to the current cursor location and automatically updates the cursor location upon each operation. This is a lot more practical than the functions above, as it does not require the user to keep track of cursor locations and calls on interrupt routine services to handle iterative work.

<u>readChar()/readString()</u>
These functions read from the keyboard and invoke interrupt0x16 to store user inputs from the keyboard to a desired buffer location specified by the parameter. These functions along with the printString() function allow printing to the screen through keyboard inputs and handle other keyboard utilities such as delete and arrow keys for backspace.

<u>readSector()/writeSector()</u>
ReadSector() reads from a specified disk sector and stores the data in the buffer specified by invoking interrupt0x13. The parameters passed to the interrupt() function specifies a track, head, and relative sector addresses, which are calculated from the absolute sector number presented by the parameter. WriteSector(), similarly, writes to a disk sector specified by the function parameter.

Disk File System
The following functions read and write files from and onto the disk. Thus, this requires the presence of a defined file system where files and user programs can be tracked and stored correctly onto the disk. This was achieved by implementing a disk map, which keeps records of free and used sectors, and a disk directory that records file names on the disk and the sectors each file uses. Existing disk map and disk directory image files were provided and placed to the disk during the launch of the operating system.

<u>readFile()</u>
This function reads a file, name specified by function parameter, from the sector and places the data into a buffer, also specified by the parameter. It searches through the disk directory and retrieves the desired file. When the file is not found, it simply returns -1.

<u>deleteFile()</u>
The function deletes a file, name specified by the function parameter, by first, marking all sectors the file uses free on the disk map, then sets the first character of the file name in the disk directory as 0. The content is, in fact, not deleted and erased from the disk but rather indicated as empty.

<u>writeFile()</u>
This function loads data from buffer, a function parameter, and writes it into a file, also specified by the function parameter. If no Disk Directory entry is free for writing the file, the function returns -1.

Also, if the Disk Map contains fewer free sectors than the file requires, the function writes as many sectors as possible and return -2.

System Call Interface

For other user programs besides the kernel to generate system calls and call for interrupts, a system call interface is implemented that creates an interrupt service routine for interrupt0x21, an existing user generated function called from kernel.asm file. Additional assembly routines from kernel.asm file is used to place entries into 0x21 of the interrupt vector to setup the interrupt service routine. To call interrupt0x21, makeInterrupt21() routine is executed to initialize the setup for that.

<u>handleInterrupt21()</u>

This function takes in parameters and calls specific interrupts specified by the user. Upon calling makeInterrupt21(), each time an interrupt0x21 is called, interrupt21ServiceRoutine will be invoked to call handleInterrupt21() function. This allows other user programs to invoke interrupt0x21 and grants access to all system calls in the kernel to other programs.

Multiprogramming

Process and memory management is essential to multiprogramming as we must perform context switching between processes and store all process control blocks in different memory location to retrieve and store processes. The structure we used to implement this is outlined in proc.c file, which follows an implementation from the provided proc.h file.

<u>executeProgram()</u>

This function executes a program, specified by the function parameter, by obtaining a process control block of the process and inserting it into the ready queue, ready for execution.

Time Sharing

<u>handleTimerInterrupt()</u>

Time sharing is enabled through function as well as existing functions in the kernel.asm file. Firstly, makeTimerInterrupt() is called in the main function of the kernel, which prompts the interrupt timer to generate interrupts and sets the interrupt vector to point to timer_ISR function. Then, timer_ISR function calls the handleTimerInterrupt(), that we implemented, which saves the stack pointer of the current running process and chooses the next process to run.

Setup Process

The operating system can be launched with bochs, an x86 computer simulator, which is ran on the X11 server (xQuartz) in MacOS. Once, the server is started, change the directory to COMP354Projects and find the project5 folder containing the compileOS.bash file. Run this shell script and follow the onscreen instructions to use this basic operating system.

User Commands Supported

The operating system contains a shell program that facilitates a basic command line interface. Following is the list of user command the shell supports.

type <file>

This user command returns the content of the file specified. It displays an error message if the file is not found.

execute <file>

This command runs the program specified by the file name by loading the program into memory and executing it.

delete <file>

This command deletes the file specified. Just like the deleteFile() function mentioned above, it does not actually remove the content from disk but simply mark the sectors the file uses empty, and change the first character of the file name to 0x00.

copy <src> <dest>

This command copies the content of the file in src into a new file dest. If the dest file already exists, it overwrites it.

dir

This command returns a list of existing files currently stored on the disk.

Bugs & Limitations

Calling execute<file> from shell on files that are not programs will display byte code of the text message and output unreadable characters.

Conclusion

This technical report entails a description of the operating system.