Build an Operating System from Scratch Project

Project C - Loading Files and Executing Programs

suggested time: two weeks

Objective

In this project you will write routines to read files into memory and execute programs. You will then write a basic shell program that will execute other programs and print out ASCII text files.

What you will need

You will need the same utilities you used in the last project, and you will also need to have completed the previous project successfully. Additionally, you will need to create some additional files. These are included at the end of the assignment.

kernel.asm same as before but with a new routine *launchProgram*
loadFile.c a utility for transfering files onto your disk image
tstpr1.c a test program that prints a message and hangs up
tstpr2.c a second test program that prints a message and then calls your terminate system call
userlib.asm this has a single function *syscall* that will allow you to call interrupts from your shell
message.txt a test message for your readfile function

The File System

The main purpose of a file system is to keep a record of the names and sectors of files on the disk. The file system in this operating system is managed by two sectors at the beginning of the disk: the *Map* at sector 1, and the *Directory* at sector 2. (This is why you put your kernel at sector 3.)

The Map tells which sectors are available and which sectors are currently used by files. This makes it easy to find a free sector when writing a file. Each sector on the disk is represented by one byte in the Map. A byte entry of 0xFF means that the sector is used. A byte entry of 0x00 means that the sector is free. You will not need to read or modify the Map in this project as you are only reading files for now.

The Directory lists the names and locations of the files. There are 16 file entries in the Directory and each entry contains 32 bytes (32 times 16 = 512, which is the storage capacity of a sector). The first six bytes of each directory entry is the file name. The remaining 26 bytes are sector numbers, which tell where the file is on the disk. If the first byte of the entry is 0, then there is no file at that entry.

For example, a file entry of:

```
4B 45 52 4E 45 4C 03 04 05 06 00 00 00 00 00 00 00 00 00...
 K  E  R  N  E  L
```

means that there is a valid file, with name “KERNEL”, located at sectors 3, 4, 5, 6. (00 is not a valid sector number but a filler since every entry must be 32 bytes). If a file name is less than 6 bytes, the remainder of the 6 bytes should be padded out with 00s.

You should note, by the way, that this file system is very limiting. Since one byte represents a sector, there can be no more than 256 sectors used on the disk (128kB of storage). Additionally, since a file can have no more than 26 sectors, file sizes are limited to 13kB. For this project, this is adequate storage, but for a modern operating system, this would be grossly inadequate.

LoadFile

You are provided with a utility *loadFile.c,* which you should compile with gcc (*gcc -o loadFile loadFile.c*). LoadFile reads a file and writes it to diskc.img, modifying the Map and Directory appropriately. For example, copy *message.txt* to the file system by typing

./loadFile message.txt

This saves you the trouble of modifying the map and directory yourself. Note that "message.txt" is more than six letters. LoadFile will truncate the name to six letters on loading it. The file name will become *messag*

Important note

You should put that loadFile message.txt line at the end of your compile script. Beforehand, however, remove your *dd if=kernel...* line from the script and replace it with *./loadFile kernel*

Step 1 - Read a file and print it

You should create a new function *readFile* that takes a character array containing a file name and reads the file into a buffer. After you complete your function, you should make it into an interrupt 0x21 call:

Read File:
AX = 3
BX = address of character array containing the file name
CX = address of a buffer to hold the file
DX = address of an integer to hold the number of sectors you read

Your readFile function should work as follows:

1. Load the directory sector into a 512 byte character array using readSector

2. Go through the directory trying to match the file name. If you do not find it, set the number of sectors read to 0 and return.

3. Using the sector numbers in the directory, load the file, sector by sector, into the buffer array. You should add 512 to the buffer address every time you call readSector. Make sure to increment the number of sectors read as you go.

Hints

This part is challenging. Try using a for loop with an iterator *fileentry* to step through the directory in increments of 32. Then for each file entry, look at the next six characters and compare them to your file name parameter. For example, does filename[0]==dir[fileentry+0], filename[1]==dir[fileentry+1], and so on? If not, move on to the next entry. You might use your *printChar* function to see how many times the for-loop is running.

When you find your file, use another for-loop to load it. Call readSector using dir[fileentry+6] as the sector number. Add 512 to the buffer address. Call readSector with dir[fileentry+7]. Repeat this until you reach a sector number 0.

Testing

Use your readFile function to read in a text file and print it out. You can use the message.txt file you just loaded. In main:

```
char buffer[13312];   /*this is the maximum size of a file*/
int sectorsRead;
makeInterrupt21();
interrupt(0x21, 3, "messag", buffer, &sectorsRead);   /*read the file into buffer*/
if (sectorsRead>0)
        interrupt(0x21, 0, buffer, 0, 0);   /*print out the file*/
else
        interrupt(0x21, 0, "messag not found\r\n", 0, 0);  /*no sectors read? then print an error*/
while(1);   /*hang up*/
```

Step 2 - Load a Program and Execute it

The next step is to load a program into memory and execute it. This really consists of four steps:

1. Load the program into a buffer (a big character array) using readFile
2. Transfer the program into the bottom of the segment where you want it to run
3. Set the segment registers to that segment and setting the stack pointer to the program's stack
4. Jump to the program

You should write a new function

void executeProgram(char* name)

that takes as a parameter the name of the program you want to run. Your function should do the following:

1. Call readFile to load the file into a buffer.

2. In a loop, transfer the file from the buffer into memory at segment 0x2000. You should use putInMemory to do this.

3. Call the assembly function void launchProgram(int segment), and pass segment 0x2000 as the parameter. This is because setting the registers cannot be done in C. The assembly function will set up the registers and jump to the program. The computer will never return from this function.

4. Finally, make your function a new interrupt 0x21 call, using the following:

Load program and execute it:
AX = 4
BX = address of character array holding the name of the program

Testing

To try this out, you are provided with a test program *tstpr1.c* You should compile it as follows:

```
bcc -ansi -c -o tstpr1.o tstpr1.c
as86 -o userlib.o userlib.asm
ld86 -d -o tstpr1 tstpr1.o userlib.o
./loadFile tstpr1
```

Call your function to load *tstpr1*and start it running. In main(), write the following:

```
makeInterrupt21();
interrupt(0x21, 4, "tstpr1", 0, 0);
while(1);
```

If your interrupt works and tstpr1 runs, your kernel will never make it to the while(1); Instead, the test program will print out a message and hang up.

Step 3 - Terminate program system call

This step is simple but essential. When a user program finishes, it should make an interupt 0x21 call to return to the operating system. This call terminates the program. For now, you can just have a terminate call hang up the computer, though you will soon change it to make the system reload the shell.

You should first make a function *void terminate()*. terminate for now should contain an infinite while loop to hang up the computer.

You should then make an interrupt 0x21 to terminate a program. It should be defined as:

Terminate program:
AX = 5

You can verify this with the program *tstpr2*. Unlike tstpr1, tstpr2 does not hang up at the end but calls the terminate program interrupt. Compile and load tstpr2 the same way you did tstpr1.

Step 4 - The Shell - making your own user program

You now are ready to make the shell! Your shell will make use of the file *userlib.asm* that contains a single assembly language function *syscall*. *syscall* is used to make an interrupt 21 call and call your kernel functions. For example, whereas in kernel you called your readSector function by writing something like *interrupt(0x21, 3, buffer, 30)* now in shell you would write *syscall(3, buffer, 30)*

Your shell should be called shell.c and should be compiled the same way that the kernel is compiled. However, in this case, you will need to assemble userlib.asm intead of kernel.asm and link userlib.o instead of kernel_asm.o. Your final file should be called *shell*.

After you compile the shell, you should use loadFile to load the shell onto diskc.img. You should add all of these commands to your compileOS.sh script.

Your initial shell should run in an infinite loop. On each iteration, it should print a prompt ("SHELL> " or "A:> " or something like that). It should then read in a line and try to match that line to a command. If it is not a valid command, it should print an error message ("Bad Command!" or something similar) and prompt again. Since you do not have any shell commands yet, anything typed in should cause Bad Command to be printed.

All input/output in your shell should be implemented using *syscall* calls. You should not rewrite or reuse any of the kernel functions. This makes the OS modular: if you want to change the way things are printed to the screen, you only need to change the kernel, not the shell or any other user program.

Kernel adjustments

In your kernel, main() should now simply set up the interrupt 0x21 using makeInterrupt21, and call an interrupt 0x21 to load and execute *shell*. Also in your kernel, you should change terminate to no longer hang up. Instead it should use interrupt 0x21 to reload and execute *shell*.

Important note:

In terminate, do not write executeProgram("shell") because strings are stored in data memory, which is still associated with the program that just ended (you'll learn about this in project E). Instead make a character array:

```
char shellname[6];
```

copy the letters in one at a time:

```
shellname[0]='s';
shellname[1]='h';
...
shellname[5]='\0';
```

and then pass that array to executeProgram

```
        executeProgram(shellname);
```

Step 5 - Shell Command “type”

You should now modify your shell to recognize the command “type filename”. If the user types, at the shell prompt, *type messag*, the shell should load messag into memory and print it out the contents. You should implement this using *syscall*s to call read file and print string.

For full credit, you should make your command robust. If the user enters a file that doesn't exist, it should print out a "file not found" error.

Step 6 - Shell command “exec”

Now modify the shell to recognize the command “exec filename”. If the user types, at the shell prompt, *exec tstpr2*, the shell should call the kernel to load and execute tstpr2, overwriting the shell. Test this. If you are successful, tstpr2 should run, print out its message, and then you should get a shell prompt again. Again, for full credit, print out an error message if the program cannot be found.

Submission

You should submit a .tar file containing all your files on Blackboard. Be sure that all files have your name in comments at the top. Your .tar file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it. Please also create a *github* account for your work and submit a link to the project.

kernel.asm

```
;kernel.asm
;Margaret Black, 2007

;kernel.asm contains assembly functions that you can use in your kernel

        .global _putInMemory
        .global _interrupt
        .global _makeInterrupt21
        .global _launchProgram
        .extern _handleInterrupt21

;void putInMemory (int segment, int address, char character)
_putInMemory:
        push bp
        mov bp,sp
        push ds
        mov ax,[bp+4]
        mov si,[bp+6]
        mov cl,[bp+8]
        mov ds,ax
        mov [si],cl
        pop ds
        pop bp
        ret

;int interrupt (int number, int AX, int BX, int CX, int DX)
_interrupt:
        push bp
        mov bp,sp
        mov ax,[bp+4]       ;get the interrupt number in AL
        push ds             ;use self-modifying code to call the right interrupt
        mov bx,cs
        mov ds,bx
        mov si,#intr
        mov [si+1],al ;change the 00 below to the contents of AL
        pop ds
        mov ax,[bp+6]       ;get the other parameters AX, BX, CX, and DX
        mov bx,[bp+8]
        mov cx,[bp+10]
        mov dx,[bp+12]

intr:   int #0x00     ;call the interrupt (00 will be changed above)

        mov ah,#0     ;we only want AL returned
        pop bp
        ret

;void makeInterrupt21()
;this sets up the interrupt 0x21 vector
;when an interrupt 0x21 is called in the future,
;_interrupt21ServiceRoutine will run

_makeInterrupt21:
        ;get the address of the service routine
        mov dx,#_interrupt21ServiceRoutine
        push ds
        mov ax, #0      ;interrupts are in lowest memory
        mov ds,ax
        mov si,#0x84    ;interrupt 0x21 vector (21 * 4 = 84)
        mov ax,cs       ;have interrupt go to the current segment
        mov [si+2],ax
        mov [si],dx     ;set up our vector
        pop ds
        ret

;this is called when interrupt 21 happens
;it will call your function:
;void handleInterrupt21 (int AX, int BX, int CX, int DX)
_interrupt21ServiceRoutine:
        push dx
        push cx
        push bx
        push ax
        call _handleInterrupt21
        pop ax
        pop bx
        pop cx
        pop dx

        iret

;this is called to start a program that is loaded into memory
;void launchProgram(int segment)
_launchProgram:
        mov bp,sp
        mov bx,[bp+2]           ;get the segment into bx

        mov ax,cs       ;modify the jmp below to jump to our segment
        mov ds,ax       ;this is self-modifying code
        mov si,#jump
        mov [si+3],bx           ;change the first 0000 to the segment

        mov ds,bx       ;set up the segment registers
        mov ss,bx
        mov es,bx

        mov sp,#0xfff0          ;set up the stack pointer
        mov bp,#0xfff0

jump:   jmp #0x0000:0x0000              ;and start running (the first 0000 is changed above)
```

userlib.asm

```
;userlib.asm
;Margaret Black, 2019

;userlib.asm contains assembly functions for user programs such as shell

        .global _syscall

;void syscall (int AX, int BX, int CX, int DX)
_syscall:
        push bp
        mov bp,sp
        mov ax,[bp+4]          ;get the parameters AX, BX, CX, and DX
        mov bx,[bp+6]
        mov cx,[bp+8]
        mov dx,[bp+10]
        int #0x21
        pop bp
        ret
```

loadFile.c

```
//loadFile.c
//Margaret Black, 2007, modified 2019
//
//Loads a file into the file system
//This should be compiled with gcc and run outside of the OS

#include <stdio.h>

int main(int argc, char* argv[])
{
	int i;

	if (argc<2)
	{
		printf("Specify file name to load\n");
		return 0;
	}

	//open the source file
	FILE* loadFil;
	loadFil=fopen(argv[1],"r");
	if (loadFil==0)
	{
		printf("File not found\n");
		return 0;
	}

	//open the floppy image
	FILE* floppy;
	floppy=fopen("diskc.img","r+");
	if (floppy==0)
	{
		printf("diskc.img not found\n");
		return 0;
	}

	//load the disk map
	char map[512];
	fseek(floppy,512,SEEK_SET);
	for(i=0; i<512; i++)
		map[i]=fgetc(floppy);

	//load the directory
	char dir[512];
	fseek(floppy,512*2,SEEK_SET);
	for (i=0; i<512; i++)
		dir[i]=fgetc(floppy);

	//find a free entry in the directory
	for (i=0; i<512; i=i+0x20)
		if (dir[i]==0)
			break;
	if (i==512)
	{
		printf("Not enough room in directory\n");
		return 0;
```

```
}
int dirindex=i;

//fill the name field with 00s first
for (i=0; i<6; i++)
	dir[dirindex+i]=0x00;
//copy the name over
for (i=0; i<6; i++)
{
	if(argv[1][i]==0)
		break;
	dir[dirindex+i]=argv[1][i];
}

dirindex=dirindex+6;

//find free sectors and add them to the file
int sectcount=0;
while(!feof(loadFil))
{
	if (sectcount==26)
	{
		printf("Not enough space in directory entry for file\n");
		return 0;
	}

	//find a free map entry
	for (i=3; i<256; i++)
		if (map[i]==0)
			break;
	if (i==256)
	{
		printf("Not enough room for file\n");
		return 0;
	}

	//mark the map entry as taken
	map[i]=0xFF;

	//mark the sector in the directory entry
	dir[dirindex]=i;
	dirindex++;
	sectcount++;

	//move to the sector and write to it
	fseek(floppy,i*512,SEEK_SET);
	for (i=0; i<512; i++)
	{
		if (feof(loadFil))
		{
			fputc(0x0, floppy);
			break;
		}
		else
		{
			char c = fgetc(loadFil);
			fputc(c, floppy);
		}
```

```
			}
		}

		//write the map and directory back to the floppy image
	fseek(floppy,512,SEEK_SET);
	for(i=0; i<512; i++)
			fputc(map[i],floppy);

	fseek(floppy,512*2,SEEK_SET);
	for (i=0; i<512; i++)
			fputc(dir[i],floppy);

		fclose(floppy);
		fclose(loadFil);

		return 0;
}
```

tstpr1.c

```
main()
{
	syscall(0,"tstpr1 is working!\r\n");
	while(1);
}
```

tstpr2.c

```
main()
{
	syscall(0,"tstpr2 is working!\r\n");
	syscall(5);
	while(1);
}
```

message.txt

```
If this message prints out, then your readFile function is working correctly!
```