Universidad Tecnológica Centroamericana

Sistemas Operativos I

Period 2016/2

File System and Single Process Management

1 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.

2 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 projects successfully. Additionally, you will need to download the new kernel.asm, map.img and dir.img for this project.

3 The File System

See the document "Using FUSE SmallFS Linux driver"

4 Tasks to do

4.1 Load 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. When you complete your function, you should make it a system call:

Syscall Read File:
AX = 3
BX = *address of character array containing the file name*
CX = *address of a buffer to hold the file*

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, 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
4. Return

4.1.1 Testing

Use your readFile function to read in a text file and print it out. You can use your message.txt file from the previous project. Create a small test program with the following code:

```
char buffer[13312];        /*this is the maximum size of a file*/
syscall_readFile("messag", buffer);/*read the file into buffer*/
syscall_printString(buffer);          /*print out the file*/
while(1);          /*hang up*/
```

Then, compile the test program, remember to add the function syscall_readFile to the os-api.asm file. After compiling the test program copy it to sector 11 (this sector is part of the kernel, but we are using it to save the test program because the kernel it's very small and it's not using this sector). Finally be sure to load the file messag using the SmallFS FUSE Linux driver.

4.2 Load a Program and Execute it

The next step is to load a program into memory and execute it. Remember we have done this before by loading a program from sector 11, but in this project you'll be loading a program from the file system. This really consists of four steps:

1. Loading the program into a buffer (a big character array)
2. Transferring the program into the bottom of the segment where you want it to run
3. Setting the segment registers to that segment and setting the stack pointer to the program's stack
4. Jumping to the program. To jump to the program you have to do what x86 called a far jump, but you cannot do this using the jump instruction (actually you can, but it's a little bit complicated), because the segment is not a constant. So to do this you'll be using the retf instruction, which is different from the ret instruction. retf pops two words from the stack, the first is the value of the instruction pointer register and the second is the value of the code segment segment, in contrast ret pops only one word from the stack, which is the value assigned to the instruction pointer register.

To try this out, you can use any of the test programs you have written so far. You should write your function to load the test program from the file system into memory and start it running. Remember to load the test program into the file system of the floppy image using the FUSE Linux driver, DON'T copy the test program to sector 11, like in previous tasks.

You should write a new function void executeProgram(char* name, int segment) that takes as a parameter the name of the program you want to run (as a character array) and the segment where you want it to run.

The segment should be a multiple of 0x1000 (remember that a segment of 0x1000 means a base memory location of 0x10000). 0x0000 should not be used because it is reserved for interrupt vectors. 0x1000 also should not be used because your kernel lives there and you do not want to overwrite it. Segments above 0xA000 are unavailable because the original IBM-PC was limited to 640k of memory. (Memory, incidentally, begins again at address 0x100000, but you cannot address this in 16-bit real mode.
This is why all modern operating systems run in 32-bit protected mode).

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 the bottom (0000) of memory at the segment in the parameter. You should do this in assembly. The x86 movsb or movsw instructions will be very useful to do this.
3. Implement the function void launchProgram(int segment) in assembly, which takes the segment number as a 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. To implement this function you just have to change the function void loadProgram(void), which you used in the previous project to load a program from sector 11 of the disk. Remember that the segment will be a parameter of the function, also remove the code that reads the sector 11 from the disk.

Finally, make your function a new syscall, using the following:

Load program and execute it:
AX = 4
BX = *address of character array holding the name of the program*
CX = *segment in memory to put the program*

4.2.1 Testing

In the kernel main funtion, write the following:

```
makeInterrupt21(); executeProgram("tstprg",
0x2000); while(1);
```

If your interrupt works and tstprg runs, your kernel will never make it to the while(1);. Instead, the tstprg will carry out it's task and hang up.

4.3 Terminate program system call

This step is simple but essential. When a user program finishes, it should invoke a syscall to return to the operating system. This call terminates the program. For now, you should 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 a syscall to terminate a program. It should be defined as:
AX = 5

You can verify this using a test program the invoke syscall_terminate at the end.

4.4 The Shell - making your own user interface

You now are ready to make a shell. A shell is basically a command line interface usually called CLI. To implement your shell you'll be using the syscalls you implemented in the file os-api.asm, in this task you'll add one more syscall to clear the screen, that's the only assembly functions you'll need to implement your shell, the rest will be done in C.

Your shell should be called shell.c and should be compiled the same way the test programs were compiled. However, don't forget to assemble os-api.asm instead of kernel.asm and link os-api.o instead of kernel_asm.o. Your final executable file should be called shell. After compiling the shell, you should load it into floppy image file system using the FUSE Linux driver. Remember to change the Makefile to include the commands neccesary to build the shell.

4.4.1 Shell operation

On each iteration the shell will 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.
All input/output in your shell should be implemented using syscalls. 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.

The shell should implement the following commands:

Command	Description
clear	Clear the screen. To implement this you'll have to add a new syscall to clear the screen, use the number 0xa for this syscall.
type *<file>*	Show the content of the file *<file>*
execute *<prg>*	Load the program *<prg>* into segment 0x2000 and then execute it.

4.4.2 Kernel adjustments

In your kernel main function you should simply set up the interrupt 0x21 using makeInterrupt21, and call executeProgram to load and execute the "shell" at segment 0x2000.

Also in your kernel, you should change terminate to no longer hang up. Instead it should reload and execute the shell at segment 0x2000.

Original document by Michael Black, 3/9/2009

Edited by Ivan de Jesús Deras, May 10, 2015