O.S. Lab 4: Loading and Executing Programs

Overview to Working with Executable Programs

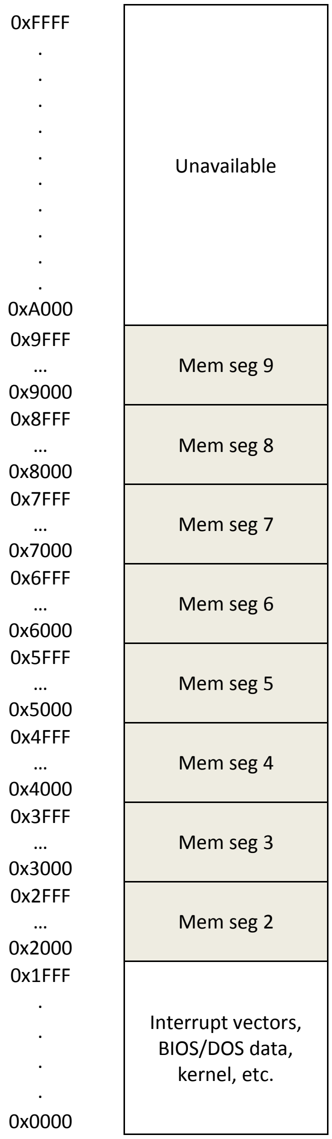
Now that we can load entire files into memory, we should be able to load executable files into memory and run them. To simplify this we will have to begin with a consideration of the memory layout in the original IBM-PC architecture.

Background: Bochs RAM Layout

The original IBM-PC was limited to 640K of useable memory, which required 20-bit registers to access it. Since the engineers had only 16-bit registers they developed a *segment-offset memory scheme*. To derive a real 20-bit address the CPU would left-shift the 16-bit segment address by one byte, then add the 16-bit offset. There were numerous problems with such real-mode operations (e.g. redundant representations of the same address corrupting multiple segments), so eventually the IBM chip set had to be upgraded to the modern 32- (now 64-) bit address *protected mode*.

Now, if you're limited to 640K of memory, the highest segment number possible is 0x9C40, limiting the range of possible addresses. Furthermore some memory is used by the system (for interrupt vectors, BIOS, DOS, device drivers, …) and unavailable to user programs. To make most efficient and flexible use of the memory left the IBM architecture assigns *segment registers* to keep track of the starting addresses of code (CS), data (DS) and stack (SS) segments, plus index registers to hold data and instruction offsets relative to the segment bases (notably the stack pointer SP for the top of the stack/offset from SS). Thus segments vary in number and size over the computer's execution cycle.

So why the history lesson? *BIOS interrupts are only available in real mode*. This means that all IBM-compatible operating systems boot in 16-bit real mode so that they can use BIOS interrupts, then transition to protected mode for most user functions. If the o.s. needs BIOS functionality it traps to the kernel, enters real mode, does its business and returns.



Based on this discussion it should be clear that our simulation has been running in real mode. Therefore, as we now consider loading programs to memory and running them, we will implement our own primitive segment-offset addressing, depicted at right above. We will have fixed-sized segments with base addresses that are multiples of 4096 (0x1000) for a total of nine useable 4K memory segments. Note that if a program needs more than 4K it may occupy multiple adjacent segments.

What You Will Need

There are two new functions that you will use contained in the **kernel.asm/kasm.o** file:

- **putInMemory(int baseLocation, int offset, char c)**
  Add *baseLocation* and *offset*, then write *c* to this computed address.
- **launchProgram(int baseLocation)**
  Set up registers, jump to *baseLocation* and commence execution.

Loading and Executing a Program

Before doing anything else add the line

```
dd if=config of=floppya.img bs=512 count=1 seek=258 conv=notrunc
```

to your **compileOS.sh** file so that your results are visible on screen.

The process of loading a program into memory and executing it 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

Write a new function **void runProgram(int start, int size, int segment)** that takes as parameters the starting sector of the program you want to run from the disk, the number of sectors to read and the segment where you want it to run (2 through 9 inclusive).

Your function should do the following:

1. Call **readSectors** to load the file from disk into a local buffer. Assume a maximum program size of 13312 bytes (26 disk sectors).
2. Multiply *segment* by 0x1000 (4096) to derive the base location of the indicated segment.
3. In a loop, transfer the loaded file from the local buffer into the memory based at the computed segment location, starting from offset 0, one byte at a time. You should use **putInMemory** to do this. Transfer all 13312 bytes even if most of them aren’t relevant to your program.
4. Call **launchProgram** which takes the base segment address from (2) as a parameter.

Remember to include a **runProgram** prototype at the beginning of the kernel code. After adding this function rewrite **kernel.c** to become this:

```
void main()
{
  char buffer[512];
  makeInterrupt21();
  interrupt(33,2,buffer,258,1);
  interrupt(33,12,buffer[0]+1,buffer[1]+1,0);
  printLogo();
  runProgram(30, 1, 2);
  interrupt(33,0,”Error if this executes.\r\n\0”,0,0);
  while (1) ;
}

/* more stuff follows */
```

To test recompile your program, rebuild the floppy disk and include the runnable program **kitty1** at sector 30:

```
dd if=kitty1 of=floppya.img bs=512 count=1 seek=30 conv=notrunc
```

Run **bochs** and see which message prints out. (If this works correctly it should look familiar.)

Terminate Program System Call

This step is simple but essential. When a user program finishes, it should make an interrupt 33 call 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.

Two steps:

1. Add the code **void stop() { while (1) ; }** to your kernel prior to the interrupt handler.
2. Tie interrupt 33/5 to this function.

You can verify this all works with the **kitty2** program which does not hang up at the end (as **kitty1** does) but calls the terminate program interrupt. Rewrite your **main()** function appropriately and add **kitty2** to your disk image at sector 30 as before.

Executing Multi-Sector Programs

To date our sample executables (**kitty1** and **kitty2**) have been simple single-sector programs. As a more complex test download, add to **floppya.img** and run the 4-sector **Stenv** program. (Repeat the steps above, with **count=4** in the **dd** call and **runProgram(30,4,2)** in your code.) This utility reads your configuration options (i.e. sector 258) from the disk image, queries you to change them, and writes them back to disk. You can see your changes by clicking on “Reset” on the top bar of the BlackDOS window.

Homemade Test Cases

Part of our justification for tying everything to interrupt 33 was so that programs other than our kernel could use o.s. functions. Let us now study a simple example.

From the lab page download the files **blackdos.asm**, **blackdos.h** and **fib.c**. A quick look at the listing of **fib.c** (seen at right) reveals what's happening. The program itself simply prints out a requested number of terms in the Fibonacci sequence. What is different are the invented commands for I/O. In the local header file we have created C-like macros for inputting and outputting strings (SCANS/PRINTS/LPRINTS) and integers (SCANN/PRINTN/LPRINTN), as well as for program termination (END). The preprocessor will substitute the correct interrupt calls for us, permitting us to hack code at a high level and ignore the underlying details. (It's still a little clumsy but to do better will require much more work.)

We compile the program in the same manner as the kernel but compile and link to **blackdos.asm** instead of **kernel.asm**, since we only need the single assembly function *interrupt()*. Any other low-level functions will be provided by our o.s. kernel.

```
#include "blackdos.h"
void main()
{
  int i, a = 1, b = 1, c, n;
  PRINTS("How many terms? \0");
  SCANN(n);
  if (n < 3) n = 3;
  PRINTN(n);
  PRINTS(" terms: \0")
  PRINTN(a);
  PRINTS(" \0");
  PRINTN(b);
  PRINTS(" \0");
  for (i = 0; i < n - 2; i++)
  {
    c = a + b;
    PRINTN(c);
    PRINTS(" \0");
    a = b;
    b = c;
  }
  PRINTS("\r\n\0");
  END;
}
```

As review, we now compile our program via the instruction sequence

```
bcc -ansi -c -o fib.o fib.c
as86 blackdos.asm -o basm.o
ld86 -o fib -d fib.o basm.o
```

then use **dd** to add the **fib** executable to our BlackDOS disk at sector 30. (This is only a single-sector file so adjust your code and script accordingly.) Running the kernel to execute **fib** will now produce the desired result. For practice you should verify that this does indeed work. You should also create test cases of your own to experiment with going forward.

Note that fib is another one-sector program. In similar fashion feel free to download, compile and add other sample programs to your disk image to further test your code.

Conclusion

When finished, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **bdos.txt**, **fib**, **kasm.o**, **compileOS.sh**, **kernel.c**, **config**, and **README**) to the drop box. Make sure that your disk image includes the **fib** file with the kernel running **fib** by default. **README** should explain what you did and how the t.a. can verify it. Your .zip/.tar file name should be your name.

Last updated 10.1.2019 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 7.2.2019, 1.9.2019, 1.26.2018, 2.2.2017, 1.25.2016, 1.27.2015, 2.21.2014, 2.22.2014, 11.3.2014.