CSC 565 - Operating Systems

Spring, 2008

Project E - Processes and Multitasking

Objective

In the previous projects you wrote a single process operating system where only one program could be executed at once. In this project you will make it possible for up to eight programs to be executing simultaneously.

Multitasking

There are two basic requirements for multitasking. First, you have to have a preemptive scheduler that can interrupt a process and save its state, find a new process, and start it executing. Second, you need to have memory management so that the processes do not get in the way of each other. Ideally, no process ever knows that it is interrupted and every process thinks it has the whole computer to itself.

Memory management

Nearly all modern operating systems use virtual memory, which completely isolates one process from another. Although all processors since the 386 support virtual memory, you will not use it in this project. Instead, you will take advantage of the earlier method of segmentation.

As you observed in previous projects, all addresses in real mode are 20 bits long, while all registers are 16 bits. To address 20 bits, the processor has six segment registers (of which only three are really important): CS (code segment), SS (stack segment), and DS (data segment). All instruction fetches implicitly use the CS register, stack operations (including local variables) the SS register, and data operations (global variables and strings) the DS register. The actual address used by any instruction consist of the 16 bit value the program thinks it is using, plus the appropriate segment register times 0x10 (shifted left by 4). For example, if your program states: JMP 0x147, and the CS register contains 0x3000, then the computer actually jumps to 0x3000 * 0x10 + 0x147, or 0x30147.

An interesting note is that 16-bit C programs (those compiled with bcc) never touch the segment registers. The pleasant thing about this is if we set the registers beforehand ourselves, the same program can run in two different areas of memory without knowing. For example, if we set the CS register to 0x2000 and ran the above program, it would have jumped to 0x20147 instead of 0x30147. If all of our segment register values are 0x1000 away from each other, the program's memory spaces will never overlap.

The catch to all this is that programs must be limited to 64k in size and never touch the segment registers themselves (unlikely in an era of 10 to 100 meg programs). They also must never crash and accidentally take out the rest of memory. That is why this approach has not been used in practical operating systems since the 1980s. However, 64k will be enough for our operating system.

Scheduling

All x86 computers (and most others) come equipped with a timer chip that can be programmed to produce interrupts periodically every few milliseconds. The interrupt produced by this timer is interrupt 8. If the timer is enabled, this means that the program is interrupted and the interrupt 8 service routine is run every few milliseconds. Your scheduler should be called by the interrupt 8 service routine.

The scheduler's task is to choose another process and start it running. Consequently it needs to know who all the active processes are and where they are located in memory. It does this using a process table.

Every interrupt 8, the scheduler should back up all the registers onto the process's own stack. It should then save the stack pointer (the key to where all this information is stored) in the process's process table entry. It should select a new process from the table, restore its stack pointer, and restore its registers. When the timer returns from interrupt, this new process will start running.

Loading and terminating

Creating a new process is a matter of finding a free segment for it, putting it in the process table, and giving it a stack. If a program, such as the shell, wants to execute a program, it creates a new process, copies the program into the new process's memory, and just waits around until the timer preempts it. Eventually the scheduler will start the new process. Terminating a program is done by removing the program from the process table and busy waiting until the timer goes off. Since the program no longer has a process table entry, the computer will never go back to it.

An interesting thing now is that the a program does not have to terminate when it executes another program. Instead it can start the new process running and move on to another task. The shell, for example, can start another process running and not end. This is known as making a background process; the user can still use the shell to do other things and the new program runs in the background.

What you will need

You will need the same utilities and support files that you used in the last project, and you will need to have completed all the previous projects successfully. You will need to download a new *kernel.asm* that has routines for handling the timer, a new *lib.asm*, and test program *phello*.

Step 1 - Timer Interrupt

Making a timer interrupt service routine is almost identical to the interrupt 0x21 service routine. The only difference is that the service routine must back up all the registers and reinitialize the timer.

Since nearly everything related to making this service routine involves handling the registers, this step is mostly already done for you in assembly. You are provided with three new assembly functions. The first, *void makeTimerInterrupt()* sets up the timer interrupt vector and initializes the timer. You need simply call it at the end of main() in kernel.c before launching the shell. The second assembly function is the interrupt 8 service routine, which will call a function you need to write: *void handleTimerInterrupt(int segment, int sp)*. The third is a routine you will call at the end of *handleTimerInterrupt*: *void returnFromTimer(int segment, int sp)*.

For now, your handleTimerInterrupt routine should call printString to print out a message (such as “Tic”) and call *returnFromTimer* with the same two parameters that were passed into handleTimerInterrupt.

Compile your operating system and run it. If it is successful, you will still be able to use the shell, yet you will see the screen fill up with “Tic”.

Step 2 - Process Table

Be sure to comment out printString(“Tic”) before proceeding.

A process table entry should store two pieces of information:

1. Whether or not the process is active
   (stored as an int: 1=active, 0=inactive)
2. The process's stack pointer
   (stored as an int)

The process table itself should be a global array in kernel.c. Note that you do not have to store the segment, since that is what you will use to index the table. You may choose to make an entry a *struct* (the C rough equivalent of a class), or just make two parallel int arrays.

Your table should contain eight entries. Segment 0x2000 should index entry 0, and segment 0x9000 should index entry 7. Thus, to find the correct table entry, divide the segment by 0x1000 and subtract 2.

To keep track of the current process, you should have a global variable *int currentProcess*, that points to the process table entry currently executing.

You should initialize the process table in main() before calling makeTimerInterrupt. Go through the table and set active to 0 on all entries, and set the stack pointer values to 0xff00 (this will be where the stacks will start in each segment). Set *currentProcess* to 0 too.

Now revise your interrupt 0x21 execute program function. Previously it launched all programs at a segment given as a parameter. Now it should launch programs in a free segment. In *executeProgram*, search through the process table for a free entry. Set that entry's active number to 1, and call *launchProgram* on that entry's segment.

You should change *executeProgram* so that it takes no parameters.

Make sure your operating system compiles correctly. Hopefully it will not run any differently from before.

Step 3 - Load Program

initializeProgram

The problem with *launchProgram* is that it never returns the program that called it. With multitasking, you want the caller program to be able to continue running while the new program runs.

You are provided with a new assembly function *void initializeProgram(int segment)*. This function sets up a stack frame and registers for the new program, but does not actually start running it. Change your *executeProgram* function to call initializeProgram instead of launchProgram.

terminate

Additionally, *terminate* needs to be changed. Previously you had it reload the shell, but now the shell never stops running. *terminate*'s new task is to set the process that called it (*currentProcess*) to inactive (0) and start an infinite while loop. Eventually the timer will go off and the scheduler will choose a new process.

setKernelDataSegment

There is a problem in the code you just created. The process table is a global variable and is stored in the kernel's segment. However, when interrupt 0x21 is called, the DS register still points to the caller's segment. When you try reading the process table in *executeProgram* and *terminate*, you are actually accessing garbage in the calling process's segment instead.

To solve this, you are provided with two more assembly functions. *void setKernelDataSegment()* and *void restoreDataSegment()*. You must call setKernelDataSegment before accessing the process table in both executeProgram and terminate. In executeProgram, you should then call restoreDataSegment right after accessing the process table.

enableInterrupts

By default, programs are loaded with interrupts disabled. This means that the programs cannot currently be preempted. You are provided with a new function in lib.asm: *void enableInterrupts()*. You should call that function at the very beginning of shell and all other user programs.

Step 4 - Scheduling

Now you should write the scheduler (handleTimerInterrupt) so that it chooses a program using round robin. The scheduler should first save the stack pointer (sp) in the current process table entry. Then it should look through the process table starting at the next entry after *currentProcess* and choose another process to run (it should loop around if it has to). Finally it should set *currentProcess* to that entry, and call returnFromTimer with that entry's segment and stack pointer.

If there are no processes at all that are active, the scheduler should just call returnFromTimer with the segment and stack pointer it was called with.

Testing

First compile your operating system and check that the shell loads correctly.

You are provided on Blackboard with another test program: *phello*. This program prints out “Hello World” ten thousand times. In the shell, execute phello. It should start printing. While it is printing, you should still be able to issue shell commands: try typing “dir”.

Step 5 - Kill Process

You should create a new interrupt and shell command that terminates a process (similar to the *kill* command in Unix). A user should be able to type “kill 3” as a shell command. Process 3 should be forcibly terminated.

You will need to do three things: create a killProcess function in kernel.c, create a new interrupt 0x21 call to terminate a process, and add the command to the shell.

Killing a process is simply a matter of setting that process's process table entry to inactive (0). Once that is done, the process will never be scheduled again.

Your interrupt 0x21 call should take the form:

Kill process
AX = 9
BX = process id (from 0 to 7)

Testing

From the shell, start phello executing. Then quickly type kill 1. If you immediately stop seeing Hello World and are able to type shell commands normally, then your kill function works.

Step 6 - Process Blocking - Extra Credit

You will notice that all calls to execute run simultaneously with the shell. It is often convenient for the shell to stop executing temporarily while the new process is running, and then resume when the new process terminates. This is essential if the new process wants keyboard input; otherwise it has to compete with the shell.

This can be done by marking processes in the process table as “waiting”. There should also be an field in the process table entry stating what process the process is waiting on. A process marked as waiting is never executed by the scheduler but is also never overwritten.

When a process is killed or terminates, any processes waiting on it should be set back to active.

You should make another execute interrupt 0x21 call that causes the caller process to “wait.” Then make a new shell command “execforeground filename” that causes the shell to block until the program “filename” terminates.

Submission

You should submit a .zip or .tar file (no .rar files please) containing all your files and a shell script for compiling on Blackboard Digital Dropbox. Be sure that all files have your name in comments at the top. Your .tar/.zip file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it.