German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

CSEN 602-Operating Systems, Spring 2017
Milestone 4: Multitasking
Due on Tuesday 25/4/2017 by 11:59 pm

Milestone Objective

In the previous milestones you wrote a single-process operating system where only one program could be executed at any point of time. The objective of this milestone is to extend your operating system to handle the multitasking of up to 8 processes. You will implement a preemptive scheduler to create the illusion of parallelism for the 8 processes.

Before you start

You need to make sure that you completed Milestone 3 successfully before you start this milestone. Create a copy of your Milestone 3 folder and continue writing in `kernel.c` and `shell.c`. You will need to download the updated `kernel.asm` and `lib.asm` from the MET website. Make sure you use the new assembly files in this milestone.

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.

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, to make things simple, you will take advantage of the earlier method of segmentation.

The Memory Structure

As you might have observed, all addresses in real mode in our system 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 time interrupt 8 is issued, 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.

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 (or a daemon); the user can still use the shell to do other things and the new program runs in the background.

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 in the `main()` function of `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 routine is `void returnFromTimer(int segment, int sp)` which you will call at the end of `handleTimerInterrupt`. 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, after launching the shell, 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); and (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 the process table will be indexed by the segment. 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 since we will handle only up to 8 processes. Segment `0x2000` should index entry 0, and segment `0x9000` should index entry 7. Thus, to find the correct table entry index, 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 executeProgram` 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 only the name of the program to be executed. Make sure your operating system compiles correctly. Hopefully it will not run any differently from before.

Step 3: Loading Programs

The problem with `launchProgram` is that it never returns the calling program. With multitasking, you want the caller program to be able to continue running until its time quantum is up. 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`.

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

Now, 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.

Step 4: Scheduling

Now you should write the scheduler code in `handleTimerInterrupt` so that it chooses a program using round robin. The time quantum for each program will be 100 time units. That is, the scheduler should pick another process to run after `handleTimerInterrupt` is called 100 times. When the scheduler decides to switch after 100 units, it 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 other processes that are active, the scheduler should just call `returnFromTimer` with the segment and stack pointer it was called with.

To test this step, you are provided in M4 resources on the MET website with two test programs: `hello1` and `hello2`. The program `hello1` prints out "Hello from program 1" in an infinite loop, and `hello2` prints out "Hello from program 2" in an infinite loop too. Now, modify your main method to execute both programs by adding the following two lines.

```
interrupt(0x21, 4, "hello1\0", 0, 0);
interrupt(0x21, 4, "hello2\0", 0, 0);
```

Do not forget to load both files to floppya.img in `compileOS.sh`. If your scheduler is working properly, you should see output from `hello1` for 100 time units, then output from `hello2` for another 100 units, then back again to `hello1` and so on.

Now to test multitasking with the shell, change the main method to execute the shell. You are provided with another program `phello` that prints "Hello World" 10000 times. Try to execute `phello` from inside the shell, it should start printing. While it is printing, you should still be able to issue shell commands: try typing "dir".

Step 5: Killing Processes

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. Once that is done, the process will never be scheduled again.

To test this step, start `hello1` executing from the shell. Then quickly type `kill 1`. If you immediately stop seeing the output from `hello1` and are able to type shell commands normally, then your kill function works.

Submission

For this milestone you are required to submit a zip containing all of your files. You should use this webform `https://podio.com/webforms/18295207/1229819` to submit your project.

Congratulations!! You are finally done with the course project! We hope you had fun developing your own OS ;)