Universidad Tecnológica Centroamericana

Sistemas Operativos I

Period 2/2016

Processes and Multitasking

Original document by Michael Black, 3/9/2009

Edited by Ivan de Jesús Deras, May 25, 2015
Edited by Elvin Deras, Jun-2016

1 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.

2 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.

2.1 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.

2.2 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.

2.3 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 operating system will never go back to it.

An interesting thing now is that 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.

3 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 a few new functions irqInstallHandler, setTimerPhase, setKernelDataSegment, and restoreDataSegment which are available on Appendix A: Assembly functions

4 Tasks to do

4.1 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 should be done in assembly. You are provided with a new function void irqInstallHandler(int irq_number, void (*fn)()). You'll use this function to set up the timer interrupt vector, irq_number will be 8 in this case and fn is the timer interrupt handler function, this is the function that will be called every time the timer produces an interrupt. There's another function void setTimerPhase(int hz) you'll use this function to initializes the timer, hz is the frequency of the timer in Hertz.

For now, you can test the timer by doing the following:

1. Call irqInstallHandler(0x8, timerISR), timerISR is the interrupt handler, you'll write that function in assembly.
2. Call setTimerPhase(100), this will set the timer frequency to 100hz. The handler will be executed 100 times per second.
3. Code the timerISR function in assembly:
    (a) Disable interrupts cli
    (b) Push all the registers on the stack bx, cx, dx, si, di, bp, ax, ds, es

(c) Reset the interrupt controller, this is very important, if you don't do this the timer won't generate more interrupts. This is also called acknowledgement. Use the following assembly code:

```
mov al, #0x20
out #0x20, al
```

(d) Switch to kernel data segment using the following code:

```
mov ax, #0x1000
mov ds, ax
```

Remember this is neccesary because you'll use a global variable in the kernel C code. So to access this variable you'll need to ensure the DS register is pointing to the kernel data segment 0x1000

(e) Call a C function handleTimerInterrupt()

(f) Restore the registers. Remember to pop es first, then ds and so on.

(g) Return from interrupt using iret instruction

4. Create the function void handleTimerInterrupt() in C, for now you can just print a message like "Tic" every 100 times the handler is called, that means 1 message per second. To do this you'll have a counter global variable in the kernel, which will be incremented every time the handler is called, when the counter reach the value of 100 one second has passed.

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

4.2 Process Queue

A process queue entry which we will call it PCB (Process Control Block) should store the following information:

1. Process state (1=Ready, 2=Waiting, 3=Running, 4=Dead)

2. Process Segment

3. Stack Pointer

4. A reference to a process waiting on this process, this will used to allow one process to wait for other to finish. The PCB will be defined as follows in C:

```
    struct PCB {
      unsigned int status;         // Process status
     unsigned  int sp;             // Stack pointer
     unsigned  int segment;        // Process segment
        struct PCB *waiter;        // Process waiting on this process
}
```

Then you can define you process queue as struct PCB process_queue[8]; an array of 8 PCBs. 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 struct PCB *currentProcess, that points to the process table entry currently executing.

You should initialize the process table in main() before initializing the timer. Go through the table and set state to 4 (Dead) on all entries, and set the stack pointer values to 0xff00 (this will be where the stacks will start in each segment), also set the segment and the reference to the waiting process accordingly. Set currentProcess to 0 too.

Now revise your syscall execute program. 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 status to 1 (Ready), and call launchProgram on that entry's segment.

You should change executeProgram so that it takes only one parameter, the program name to execute.

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

4.3 Load Program

4.3.1 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'll have to write a new C 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 loadProgram.

To implement initializeProgram you'll use the following struct (which represents the process context) to set the initial value for all the registers.

```
struct regs {
        unsigned int es;
        unsigned int ds;
        unsigned int ax;
        unsigned int bp;
        unsigned int di;
        unsigned int si;
        unsigned int dx;
        unsigned int cx;
        unsigned int bx;

        unsigned int ip;
        unsigned int cs;
        unsigned int flags;
};
```

The value for the segment registers ds, es, cs will be the segment value passed as parameter to the function. The flags register should have the value 0x0200, this will ensure that the process start with interrupts enabled by default (Bit 9 is IF, interrupt flag). All other registers will be 0, including ip, which is the instruction pointer. Finally, we have to copy this initial register values to the top of the process stack, at the address segment:0xff00, use your copyToSegment function to do this.

4.3.2 Process 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 4 (Dead) and start an infinite while loop. Eventually the timer will go off and the scheduler will choose a new process.

4.3.3 Kernel Data Segment and the Process Queue

The process queue is a global variable and is stored in the kernel's data segment. However, when a syscall is executed, the DS register still points to the caller's segment. When you try reading the process queue, you are actually accessing garbage in the calling process's segment instead.

To solve this, you are provided with two additional assembly functions on Appendix A: Assembly functions: • void

setKernelDataSegment() and

• void restoreDataSegment().

You must call setKernelDataSegment before accessing the process queue, then call restoreDataSegment right after accessing the process queue.

4.4 Scheduling

Now you should write the scheduler, to do this first you have to write the timer interrupt service routine (timerISR function) in assembly. Remember this function will be called every time the timer generate an interrupt (100 times per second).

First we disable the interrupts temporarily (cli), then we have to save the context of the interrupted process, that is all the registers, look at the struct regs defined previuosly. The last 3 registers (ip, cs and flags) are saved by the CPU in the process stack, so we have to save the rest, in reverse order, so we push bx first then cx and so on. We need also to save the stack pointer (sp register) in the current PCB (currentProcess variable).

After saving the process context we will call a C function void scheduleProcess(), this function should look through the process queue starting at the next entry after currentProcess and choose another process to run (it should loop around if it has to), so that it chooses a program using round robin. Finally it should set currentProcess to that entry, change the new process state to 3 (Running) and the state of the previous process to 1 (Ready) and return.

After scheduleProcess return, we need to activate the process pointed by currentProcess, to do that we have to set the *stack pointer* and the *stack segment* to the one in the current context, after that we should pop the registers, in reverse order, that means the last register we pop should be bx. If there are no processes at all that are active, the timerISR function should just return (iret).

4.4.1 Testing

First compile your operating system and check that the shell loads correctly. Then create a test program that prints a message many times, insert a delay after printing the message, so the program last for some time in execution. Compile and load this program into your floppy image, then start your operating system. Once the shell is loaded execute the test program, you should see the messages printed on screen and still be able to access the shell, try to execute the clear command.

4.5 Kill Process

You should create a syscall 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:

1. Create a killProcess function in kernel.c
2. Create a new syscall to terminate a process,
3. Add the command to the shell.

Killing a process is simply a matter of setting that process's process table entry to 4 (Dead). Once that is done, the process will never be scheduled again.

Your syscall should take the form:

Kill process
AX = 9
BX = *process id (from 1 to 8)*

4.5.1 Testing

From the shell, start the test you created previously. Then quickly type kill 2. If you immediately stop seeing the message printed by the program and are able to type shell commands normally, then your kill function works.

4.6 Process Blocking

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 setting the process state to 2 (Waiting). There should also be an field in the process control block 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 1 (Ready).

You should make a syscall that causes the caller process to “wait”. Then make a new shell command “executew filename” that causes the shell to block until the program “filename” terminates.

4.7 Process List

Finally you’ll add a new syscall and command to the shell “ps”, this is also a Unix/Linux command. You’ll add a new syscall as follows:

List Process
AX = 11
BX = *pointer to array of structs*

The syscall should fill an array of structs containing at least the process id, the segment and the process state. You shall not return a pointer to the kernel process queue. Additionally the syscall should return the number of process in the system, then use this syscall to implement the shell ps command.

5 Appendix A: Assembly functions

```
; void irqInstallHandler(int irq_number, void (*fn)())
; Install an IRQ handler
_irqInstallHandler:
        cli

        push bp
        mov bp, sp
        push si push
        ds

        mov dx, [bp+6] ; Function pointer
        xor ax, ax mov ds, ax  ; Interrupt vector is at lowest memory
        mov si, [bp+4]
        shl si, 2                        ; ax = irq_handler * 4

        mov ax, cs
        mov [si + 2], ax
        mov [si], dx

        pop ds
        pop si
        pop bp

        sti
        ret

; void setTimerPhase(int hz) ; Set the timer frequency in Hertz
_setTimerPhase:

        push bp
        mov bp, sp
        mov dx, #0x0012 ; Default frequency of the timer is 1,193,180 Hz
        mov ax, #0x34DC
        mov bx, [bp+4]
        div bx

        mov bx, ax ; Save quotient

        mov dx, #0x43 mov
        al, #0x36
        out dx, al                       ; Set our command byte 0x36

        mov dx, #0x40
        mov al, bl
        out dx, al      ; Set low byte of divisor mov al, bh out
        dx, al          ; Set high byte of divisor
```

```
	pop bp
	ret

; void setKernelDataSegment()
; Sets the data segment to the kernel, saving the current ds on the stack
_setKernelDataSegment:
	pop bx
	push ds
	push bx
	mov ax, #0x1000
	mov ds, ax
	ret

; void restoreDataSegment() ;
Restores the data segment
_restoreDataSegment:
	 pop bx
	 pop ds
	 push bx
	 ret
```