Build an Operating System from Scratch Project

Project E - Processes and Multitasking
suggested time: two weeks

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 update your *kernel.asm* with new routines for handling the timer. You should also update *userlib.asm*, and compile two test programs *number* and *letter*. These are all included at the end of this document.

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 printChar several times to print out a message, such as 'T' 'i' 'c', 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 the "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. Note that you do not have to store the segment, since that is what you will use to index the table. I recommend making the table with two parallel int arrays: *int processActive[8]* and *int processStackPointer[8]*.

Your table should contain eight entries. Segment 0x2000 should index entry 0, and segment 0x9000 should index entry 7. Thus, given a table entry, you can find the segment by adding two and multiplying by 0x1000.

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 kernel.c's main() before calling makeTimerInterrupt. Go through the table and set processActive to 0 on all entries, and set the processStackPointer values to 0xff00 (this will be where the stacks will start in each segment). Set *currentProcess* to -1, since there are no user processes currently.

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

Step 3 - Execute Program, revisited

Your execute program is going to work differently. Previously it loaded a program at segment 0x2000 and never returned. With multitasking, you want the calling program to keep running while the new program runs. Your new executeProgram will find a free segment, load the program into it, mark it as active, and return. Later, the timer interrupt will start the program running.

There are a few new assembly functions you will need to use:

void initializeProgram(int segment)

This function sets up a stack frame and registers for the new program, but does not actually start running it. Your *executeProgram* will now call this instead of launchProgram.

int setKernelDataSegment()
void restoreDataSegment(int segment)

There is a problem with accessing the process table variables in executeProgram. These variables are global variables and are 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 you are actually accessing garbage in the calling process's segment instead.

To solve this, you are provided with two more assembly functions: *setKernelDataSegment* and *restoreDataSegment()*. Surround any access of *processActive, processStackPointer,* or *currentProcess* with the following:

```
dataseg = setKernelDataSegment();
currentProcess=...
restoreDataSegment(dataseg);
```

executeProgram steps

Your new executeProgram will do the following:

1. read the file into the buffer

2. step through the processActive array looking for a free entry (0).

3. determine the segment: take the entry number, add 2, and multiply by 0x1000.

4. copy the buffer into the segment with putInMemory

5. call initializeProgram

6. set the processActive for that entry to 1. also set the entry's processStackPointer to 0xff00.

Again, make sure you surround steps 2 and 6 with setKernelDataSegment and restoreDataSegment since they access global variables.

You can't test your work until you implement the scheduler, so continue on to step 4.

Step 4 - Scheduling

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

You can implement this as follows:

1. call dataseg=setKernelDataSegment so you can access the process table

2. save the stack pointer (*sp* parameter) into the processStackPointer entry as indexed by the currentProcess. Only do this, however, if the currentProcess is not -1.

3. Find an active process to run. Add one to the currentProcess, looping back to 0 if the currentProcess reaches 8. Then check if the processActive entry for currentProcess is 1. Repeat this and break when you reach an active process.

4. set the *segment* to (currentProcess+2)*0x1000 and *sp* to processStackPointer[currentProcess]

5. call restoreDataSegment(dataseg) and finally returnFromTimer.

Step 5 - some additional adjustments, and testing

Before you're ready to test you'll need to make a few additional changes.

makeTimerInterrupt

In kernel.c's main() you should only call makeTimerInterrupt *after* you call executeProgram to run the shell. This will ensure that the shell is loaded before the scheduler goes looking for it. Be sure you still call *while(1);* at the end of main.

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 userlib.asm: *void enableInterrupts()*. Call that function at the very beginning of shell, tstpr2, and all other user programs.

terminate

Additionally, *terminate* needs to be changed. Previously you had it reload the shell, but now the shell kept running and doesn't need reloading.. *terminate*'s new task is to set the process that called it (*currentProcess*) to inactive (set its processActive entry to 0) and start an infinite while loop. Eventually the timer will go off and the scheduler will choose a new process. Be sure to call setKernelDataSegment first.

Testing

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

You are provided on Blackboard with another test program: *number*. This program prints out a message a hundred times. In the shell, execute *number*. It should start counting. While it prints you should still be able to issue shell commands: try typing “dir”.

Next try executing *tstpr2* from the last project. Make sure you recompile it with *enableInterrupts* called first. It should print its message and terminate without hanging up the shell.

Optionally, you can place the following code in your handleTimerInterrupt function just after calling

setKernelDataSegment. It will draw the active process numbers at the top right of the screen.

```
for(i=0; i<8; i++)
{
	putInMemory(0xb800,60*2+i*4,i+0x30);
	if(processActive[i]==1)
		putInMemory(0xb800,60*2+i*4+1,0x20);
	else
		putInMemory(0xb800,60*2+i*4+1,0);
}
```

Step 6 - 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 number executing. Then quickly type kill 1. If you immediately stop seeing the message and are able to type shell commands normally, then your kill function works.

Step 7 - Process Blocking (Challenge)

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". Make another array *int processWaitingOn[8]* that holds the id of the process that the entry waiting on. If the processActive entry is 2, the process is considered waiting. 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" and takes

as a parameter the id of the process it wants to wait on.

Wait on process
AX = 10
BX = process id (from 0 to 7)

Then modify the shell command “exec filename” to cause the shell to block until the program “filename” terminates. You will need to modify the executeProgram system call so that it returns the process id of the process it launches. This will allow you to pass that id into the wait command when you call it.

Submission

You should submit a .tar file containing all your files on Blackboard. Be sure that all files have your name in comments at the top. Your .tar file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it. Please also create a *github* account for your work and submit a link to the project.

updated kernel.asm

```
;kernel.asm
;Margaret Black, 2007

;kernel.asm contains assembly functions that you can use in your kernel

        .global _putInMemory
        .global _interrupt
        .global _makeInterrupt21
        .global _launchProgram
        .extern _handleInterrupt21
        .global _makeTimerInterrupt
        .extern _handleTimerInterrupt
        .global _returnFromTimer
        .global _initializeProgram
        .global _setKernelDataSegment
        .global _restoreDataSegment

;void putInMemory (int segment, int address, char character)
_putInMemory:
        push bp
        mov bp,sp
        push ds
        mov ax,[bp+4]
        mov si,[bp+6]
        mov cl,[bp+8]
        mov ds,ax
        mov [si],cl
        pop ds
        pop bp
        ret

;int interrupt (int number, int AX, int BX, int CX, int DX)
_interrupt:
        push bp
        mov bp,sp
        mov ax,[bp+4]  ;get the interrupt number in AL
        push ds        ;use self-modifying code to call the right interrupt
        mov bx,cs
        mov ds,bx
        mov si,#intr
        mov [si+1],al  ;change the 00 below to the contents of AL
        pop ds
        mov ax,[bp+6]  ;get the other parameters AX, BX, CX, and DX
        mov bx,[bp+8]
        mov cx,[bp+10]
        mov dx,[bp+12]

intr:   int #0x00      ;call the interrupt (00 will be changed above)

        mov ah,#0      ;we only want AL returned
        pop bp
        ret

;void makeInterrupt21()
;this sets up the interrupt 0x21 vector
;when an interrupt 0x21 is called in the future,
;_interrupt21ServiceRoutine will run

_makeInterrupt21:
        ;get the address of the service routine
        mov dx,#_interrupt21ServiceRoutine
        push ds
        mov ax, #0     ;interrupts are in lowest memory
        mov ds,ax
        mov si,#0x84   ;interrupt 0x21 vector (21 * 4 = 84)
        mov ax,cs      ;have interrupt go to the current segment
        mov [si+2],ax
        mov [si],dx    ;set up our vector
        pop ds
        ret

;this is called when interrupt 21 happens
```

```
;it will call your function:
;void handleInterrupt21 (int AX, int BX, int CX, int DX)
_interrupt21ServiceRoutine:
        push dx
        push cx
        push bx
        push ax
        sti
        call _handleInterrupt21
        pop ax
        pop bx
        pop cx
        pop dx

        iret

;this is called to start a program that is loaded into memory
;void launchProgram(int segment)
_launchProgram:
        mov bp,sp
        mov bx,[bp+2]  ;get the segment into bx

        mov ax,cs      ;modify the jmp below to jump to our segment
        mov ds,ax      ;this is self-modifying code
        mov si,#jump
        mov [si+3],bx  ;change the first 0000 to the segment

        mov ds,bx      ;set up the segment registers
        mov ss,bx
        mov es,bx

        mov sp,#0xfff0 ;set up the stack pointer
        mov bp,#0xfff0

jump:   jmp #0x0000:0x0000    ;and start running (the first 0000 is changed above)


;void makeTimerInterrupt()
;sets up the timer's interrupt service routine
_makeTimerInterrupt:
        cli
        mov dx,#timer_ISR ;get address of timerISR in dx

        push ds
        mov ax,#0       ;interrupts are at lowest memory
        mov ds,ax
        mov si,#0x20    ;timer interrupt vector (8 * 4)
        mov ax,cs       ;have interrupt go to the current segment
        mov [si+2],ax
        mov [si],dx     ;address of our vector
        pop ds

        ;start the timer
        mov al,#0x36
        out #0x43,al
        mov ax,#0xFF
        out #0x40,al
        mov ax,#0xFF
        out #0x40,al

       sti
        ret

;this routine runs on timer interrupts
timer_ISR:

        ;disable interrupts
        cli

        ;save all regs for the old process on the old process's stack
        push bx
        push cx
        push dx
        push si
```

```
        push di
        push bp
        push ax
        push ds
        push es

        ;reset interrupt controller so it performs more interrupts
        mov al,#0x20
        out #0x20,al

        ;get the segment (ss) and the stack pointer (sp) - we need to keep these
        mov bx,ss
        mov cx,sp

        ;set all segments to the kernel
        mov ax,#0x1000
        mov ds,ax
        mov es,ax
        mov ss,ax
        ;set the kernel's stack
        mov ax,#0xdff0
        mov sp,ax
        mov bp,ax

        ;call handle interrupt with 2 parameters: the segment, the stack pointer.
       mov ax,#0
        push cx
        push bx
        call _handleTimerInterrupt

;void returnFromTimer(int segment, int sp)
;returns from a timer interrupt to a different process
_returnFromTimer:
        ;pop off the local return address - don't need it
        pop ax
        ;get the segment and stack pointer
        pop bx
        pop cx

        ;get rid of the junk from the two calls and no returns
        pop ax
        pop ax
        pop ax
        pop ax
        pop ax
        pop ax
        pop ax

        ;set up the stack
        mov sp,cx
        ;set up the stack segment
        mov ss,bx

        ;now we're back to the program's area
        ;reload the registers (if this is it's first time running, these will be zeros)
        pop es
        pop ds
        pop ax
        pop bp
        pop di
        pop si
        pop dx
        pop cx
        pop bx

        ;enable interrupts and return
        sti
        iret

;void initializeProgram(int segment)
;this initializes a new program but doesn't start it running
;the scheduler will take care of that
;the program will be located at the beginning of the segment at [sp+2]
_initializeProgram:
```

```
        ;bx=new segment
       push   bp
        mov     bp,sp
        mov     bx,[bp+4]

;make a stack image so that the timer interrupt can start this program

        ;save the caller's stack pointer and segment
        mov     cx,sp
        mov     dx,ss
        mov     ax,#0xff18      ;this allows an initial sp of 0xff00
        mov     sp,ax
        mov     ss,bx

        mov     ax,#0   ;IP
        push    ax
        mov     ax,bx   ;CS
        push    ax
        mov     ax,#0x0         ;a normal flag setting
        push    ax
        mov     ax,#0           ;set all the general registers to 0
        push    ax      ;bx
        push    ax      ;cx
        push    ax      ;dx
        push    ax      ;si
        push    ax      ;di
        push    ax      ;bp
        push    ax      ;ax
        mov     ax,bx
        push    ax      ;ds
        push    ax      ;es

        ;restore the stack to the caller
        mov     sp,cx
        mov     ss,dx
       pop    bp
        ret

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

;void restoreDataSegment(int segment)
;sets the data segment to segment
_restoreDataSegment:
       push bp
       mov bp,sp
       mov ax,[bp+4]  ;get new data segment into ax
       mov ds,ax
       pop bp
        ret
```

updated userlib.asm

```
;userlib.asm
;Margaret Black, 2019

;userlib.asm contains assembly functions for user programs such as shell

	.global _syscall
	.global _enableInterrupts

;void syscall (int AX, int BX, int CX, int DX)
_syscall:
	push bp
	mov bp,sp
	mov ax,[bp+4]  ;get the parameters AX, BX, CX, and DX
	mov bx,[bp+6]
	mov cx,[bp+8]
	mov dx,[bp+10]
	int #0x21
	pop bp
	ret

;void enableInterrupts()
;call at the beginning of programs.  allows timer preemption
_enableInterrupts:
	sti
	ret
```

number.c

```
main()
{
	int i,j,k=1,l;
	char* msg="Number is 0\r\n";

	enableInterrupts();
	for(i=0; i<100; i++)
	{
		msg[10]++;
		if(msg[10]==0x3a)
			msg[10]='0';
		syscall(0,msg);
		for(j=0; j<10000; j++)
		{
			for(l=0; l<10; l++)
				k=k*2;
		}
	}
	syscall(5);
	while(1);
}
```

letter.c

```
main()
{
	int i,j,k=1,l;
	char* msg="Letter is A\r\n";

	enableInterrupts();
	for(i=0; i<26; i++)
	{
		syscall(0,msg);
		for(j=0; j<300; j++)
		{
			for(l=0; l<10; l++)
				k=k*2;
		}
		msg[10]++;
	}
	syscall(5);
	while(1);
}
```