Operating Systems

Project #5: Processes and Multiprogramming

Objective

In projects 1-4 you created a fairly functional operating system that could run one process at a time. In this project you will extend your operating system so that it can run multiple processes concurrently.

Background

There are three key pieces required to implement multiprogramming.

1. The operating system must perform ***memory management***. Memory management includes keeping track of which segments of memory currently contain processes and which are available to load new programs. It also includes the ability to load multiple programs into different memory segments so that the operating system can switch rapidly between them.

2. The operating system must implement ***time-sharing***. Time-sharing requires that the operating system periodically regains control from the executing process, allowing it to suspend the executing process and start a different one.
3. The operating system must perform ***process management***. Process management includes keeping track of which processes are ready to run, which processes are blocked and why, where each process is located in memory and the context information necessary to suspend and resume each process.

Memory Management

Our operating system will manage memory using fixed-size segmentation. With fixed-size segmentation, the available memory is divided into a number of fixed-size segments and each executing process is allocated one of these segments. Earlier, we viewed memory as logically divided into ten segments (0x0000, 0x1000, …, 0x9000). Segment 0x0000 was reserved for the interrupt vector and segment 0x1000 was reserved for the kernel. The kernel routines that you wrote for loading and executing programs allowed programs to be loaded into a specified segment between 0x2000 and 0x9000. You will modify your kernel so that it keeps track of which segments are free and which are currently occupied by executing programs. By keeping track of the free and occupied memory segments, the operating system will be able to automatically load multiple programs into memory.

We will use a memory segment map to keep track of which segments are used and which are free. A memory segment map is conceptually similar to a disk map. It will have one entry for each memory segment. The segments that are used will be indicated by one value and the segments that are free will be indicated by a different value. Thus, finding free segments and releasing segments that are no longer being used will be fairly straightforward.

You may notice that this memory management scheme suffers from two significant problems. First, there is internal fragmentation of the memory. Small programs will have a full segment allocated to them, even though they use only a small portion of it. Thus, there is memory allocated that is not being used and is unavailable for use by other processes. Second, the size of a program, including its data and stack segments, is limited to the size of a segment (0x1000 bytes = 65536 bytes = 64kB). Most modern operating systems get around these types of issues by using paged virtual memory. While we will be discussing paged virtual memory in class, we will not be implementing it in this project. All Intel processors since the 80386 provide hardware support for paged virtual memory, so implementing it would make a great--though ambitious--final project.

Time-Sharing

With time sharing, the operating system allows each process to run for a specified amount of time, called its time slice. When a process' time slice has expired, the operating system will select another process to run for a time slice. To implement time-sharing, the OS needs a mechanism by which it can regain control of the machine when a process' time slice expires. The mechanism that is used on Intel x86 based machines (e.g., 80386, 80486, Pentium, Itanium etc…) is a programmable interrupt timer. Basically the programmable interrupt timer can be set to generate an interrupt after a set amount of time. On x86 machines, the programmable interrupt timer generates interrupt 0x08.

Thus, if our OS installs an interrupt service routine for interrupt `0x08` and sets the timer appropriately, it can gain control of the machine each time the timer goes off. The details of interacting with the programmable interrupt timer will be handled by code that you are given in `kernel.asm`. The one detail that you'll need to know is that the given code programs the interrupt timer to generate an interrupt `0x08` approximately 12 times per second. If you are curious about the details check out the page: http://www.brokenthorn.com/Resources/OSDevPit.html.

Process Management

In a multiprogramming operating system, three of the main responsibilities related to process management are starting new processes, handling timer interrupts and terminating processes that have completed. The memory management system described earlier as well as two process management data structures, the process control block (PCB) and the ready queue, are central to process management. Each process has an associated PCB that stores information about it such as where it is located in memory and what its current state is (running, waiting, blocked, etc…). The ready queue is a list of the PCBs of the processes that are ready to run.

When a new process is started, the process management system must consult with the memory management system to find a free memory segment in which to load the program. A PCB is then obtained for the process and the process is loaded into memory, and its PCB is inserted into the ready queue. When a timer interrupt occurs, the interrupt service routine for interrupt 0x08, which is part of the kernel, must save the context of the currently running process, select a new process from the ready queue to run, restore its context and start it running. This is what we described as the process or short-term scheduler in class. When a process terminates, the memory segment that was used by the process and its PCB must both be released.

Getting Started

Make a complete copy of your `project4` directory as `project5`.

Drew University:

Download & unzip the files for project 5, then move them into your project 5 folder.

Washington and Lee University:

See project assignment web page for instructions.

You should have a directory called `project5NewFiles` containing the following:

- `kernel.asm` – assembly language routines you will use in your kernel.
- `lib.asm` – assembly language routine for invoking interrupts (i.e. making system calls) from user programs.
- `proc.h` – defines the data structures and declares the prototypes used to manage memory and processes.
- `proc.c` – defines some of the functions used to manage memory and processes.
- `testproc.c` – the set of tests that test the implementations of the data structures and functions defined in `proc.h`.
- `bootload.asm`, `map.img`, `dir.img` – a version of the bootloader, disk map and disk directory that allow 20 sectors for the kernel.

Note that `kernel.asm` and `lib.asm` contain some new functions that were not included in earlier versions. Replace the old versions that were there.

Timer Interrupts

The programmable interrupt timer periodically generates an interrupt. The new `kernel.asm` provided with this project contains three new functions that will allow your OS to handle timer interrupts: `makeTimerInterrupt`, `timer_ISR` and `returnFromTimer`.

- **`makeTimerInterrupt`** programs the interrupt timer to generate approximately 12 interrupts per second and sets entry 0x08 in the interrupt vector to point to the `timer_ISR` function. Thus, each time the timer generates an interrupt, `timer_ISR` will be invoked.

- **`timer_ISR`** saves the context of the interrupted process by pushing all of the register contents onto its stack. After saving the interrupted process' context, `timer_ISR` invokes a function named `handleTimerInterrupt` that you will define in your kernel. `timer_ISR` will pass the memory segment (e.g., 0x3000 or 0x5000) and stack pointer of the interrupted process to `handleTimerInterrupt`. Eventually, you will implement that function so that it saves the stack pointer of the interrupted process and then selects a new process to be run. When `handleTimerInterrupt` has finished its work, it will call `returnFromTimer`, passing it the segment and stack pointer of the process that you wish to run next.

- **`returnFromTimer`** will restore the context of the process by popping all of the register values that were pushed by the `timer_ISR` routine and then resume the process.

For now, we just want to setup and test the timer interrupts to be sure they are working. Add a call to `makeTimerInterrupt()` to the `main` function in your kernel after the call to `makeInterrupt21` and before you execute the shell. Add the `handleTimerInterrupt` function to your kernel with the prototype:

```
  void handleTimerInterrupt(int segment, int stackPointer);
```

For testing purposes, `handleTimerInterrupt` should print a message (e.g., “tic”) to the screen and then invoke `returnFromTimer` defined in `kernel.asm`. `returnFromTimer` has the prototype:

```
     void returnFromTimer(int segment, int stackPointer);
```

Thus, when you invoke `returnFromTimer`, you will need to provide arguments for the segment and stack pointer. For now, you should pass it the same segment and stackPointer that were passed to your `handleTimerInterrupt` function, which means that you will be resuming the same process that was interrupted (i.e., your shell in this case). Later you’ll change this to allow a different process to be resumed after each timer interrupt.

When you compile and run your kernel now, your shell should start and then the screen should fill with the message you printed in `handleTimerInterrupt`. Once you’ve tested that this works, feel free to comment out the “tic”.

Structures and Functions for Managing Memory and Processes

To manage memory and processes, your kernel will need several data structures. While there are many different possibilities for these structures, you are provided with one possible definition of the structures in `proc.h`. Study the comments in this file. Once you understand the role that will be played by each of the structures and functions, create a file named `proc.c`, include `proc.h` at the top, and provide implementations for each of the defined functions.

Because testing and debugging the functions in your `proc.c` file would be very difficult within the kernel, you will test and debug them as a stand-alone C program running not on `bochs` but on your machine.

The file `testproc.c` contains a `main` function and functions that are unit tests for the functions defined by `proc.h`. You can compile `testproc.c` using the `gcc` compiler and run it using the following commands:

```
gcc testproc.c proc.c
./a.out
```

Since `initializeProcStructures` has been implemented for you in `proc.c`, you should see the output:

```
Testing initializeProcStructures
done
```

Add the rest of the functionality to `proc.c` and run the unit tests in the `testproc.c` program to check all of the functionality in your `proc.c` file.

Implementing Multiprogramming

To implement multiprogramming in your kernel, you will need to complete a number of tasks:

Set Up

To create and initialize the data structures that you will use to manage the processes, modify `kernel.c` so that it defines the label `MAIN` (see `testproc.c` for example), includes the `proc.h` file, and invokes `initializeProcStructures` in `main`. Also modify `run.sh` so that it compiles `proc.c` (using `bcc` now) and links it with your kernel. Note: make sure you **do not** put //comments before the `#define` or `#include`.

Starting Programs

To start a new program in a multiprogramming system,

1. find a free memory segment for the process
2. obtain and setup a PCB for the process
3. load the program into the free memory segment
4. place the process' PCB into the ready queue.

The process then waits in the ready queue until it is selected to run by the scheduler.

Currently programs are loaded and run by the `executeProgram` function that you wrote in project 3. This function has the prototype:

```
int executeProgram(char *fname, int segment);
```

This function loaded the program `fname` into the specified memory segment and then invoked the `launchProgram` function provided in `kernel.asm` to jump to the first instruction in that segment, starting the program. To implement multiprogramming you will need to modify this function.

Change the prototype of the `executeProgram` function to

```
int executeProgram(char *fname);
```

Change all calls to `executeProgram` (e.g. in `handleInterrupt21`) so that they no longer provide an argument for the segment parameter that has been removed. Now modify `executeProgram` so that it finds an empty memory segment (using a function from `proc.c`) and then loads the desired program into that segment and jumps to its first instruction using the `launchProgram` function. You don't need to check that the memory segment is valid any more.

If you compile and run your kernel at this point, it should execute exactly as it did in project 4.

To setup for multiprogramming as described above, you should further modify `executeProgram` so that it obtains a PCB for the process, initializes the PCB, and places it into the ready queue. The PCB should be initialized by

1. setting the name of the process to its filename
2. the state of the process to STARTING
3. the segment to the memory segment where the process is loaded.
4. The stack pointer should be set to 0xFF00, which will make the top of the process' stack begin at offset 0xFF00 within its segment.

Finally, we no longer want to jump to the first instruction of the new process at this point. Instead we simply want to return to the process that made the call to `executeProgram`. Eventually, a timer interrupt will occur and your scheduler will select a PCB (possibly the new process) from the ready queue to be run. Replace the call to `launchProgram` with a call to `initializeProgram`. `initializeProgram` is provided by the new `kernel.asm` file and has the prototype:

```
    void initializeProgram(int segment);
```

`initializeProgram` creates an initial context for the process and pushes it onto the process' stack. This is a clever way of making a new process look exactly as if its context was saved by the `timer_ISR` following timer interrupt. This has the advantage that when the scheduler later wants to start this process, it can treat it the same as any other process (i.e., by calling `returnFromTimer` to restore the process' context by popping it off of its stack.)

With the call to `launchProgram` replaced with a call to `initializeProgram`, `executeProgram` will now actually return to its caller. `executeProgram` should return

- -1 if the program file cannot be found
- -2 if the memory is full
- 1 if the program loads and initializes successfully.

Handling Timer Interrupts (i.e. Scheduling)

You will now modify `handleTimerInterrupt` so that your OS schedules processes using round-robin scheduling. As we saw earlier, `handleTimerInterrupt` is invoked each time a timer interrupt occurs. When `handleTimerInterrupt` is invoked, it is passed the segment and the stack pointer of the process that was interrupted (i.e., the running process). You should

1. save the segment and stack pointer into the PCB of the running process
2. mark that process as READY
3. add it to the tail of the ready queue.
4. remove the PCB from the head of the ready queue, mark it as RUNNING
5. set the running variable to point to it
6. invoke `returnFromTimer` with the segment and stack pointer of the new running process

If the ready queue is empty, then complete the above steps using the idle process instead.

Note that the first time `handleTimerInterrupt` is invoked, the running variable should be pointing to the PCB of the idle process (see `initializeProcStructures` function in `proc.h`). Thus, your code will save the stack pointer that was passed into the idle process' PCB. When the very first timer interrupt occurs, the infinite while loop at the end of the kernel's main function will be executing. Thus, the idle process becomes that while loop! Thus, anytime there are no processes in the ready queue, the OS will run that while loop for a time slice and then check the ready queue again.

Terminating Programs

In project 3, you implemented the `terminate` function so that anytime a process terminated the shell was reloaded. With multiprogramming, the shell will still be running concurrently with other processes. Thus, there will be no need to reload it. Instead, when a process terminates, you will need to free the memory segment that it is using, free the PCB that it is using, set its state to DEFUNCT and enter an infinite while loop. Eventually a timer interrupt will occur, and the scheduler will pick a new process from the ready queue and start it. **Note** that you will need to modify `handleTimerInterrupt` so that it deals appropriately with a running process that is DEFUNCT.

Oops… That's Not My Data

There are a few problems with the code that you have written in the `executeProgram` and `terminate` functions. These functions are usually invoked via system calls (i.e., interrupt 0x21). When that happens, the data segment (DS) register points to the data segment of the program that made the system call. However, the global variables being used to store the memory and process management data structures are stored in the kernel's data segment. Thus, we need to set the DS register to point to the kernel's data segment before we access those structures and then restore the DS register to the calling program when we are finished. Note that this was not a problem with the timer interrupts earlier because the `timer_ISR` routine you were given in `kernel.asm` resets the DS register to point to the kernel's data segment for you.

`kernel.asm` provides two functions that will help, their prototypes are:

```
     void setKernelDataSegment();
     void restoreDataSegment();
```

You will need to invoke `setKernelDataSegment` before accessing any of the global data structures and `restoreDataSegment` after accessing them. For example, to find a free memory segment you might write:

```
     setKernelDataSegment();
     freeSeg = getFreeMemorySegment();
     restoreDataSegment();
```

When you copied the filename from the parameter to `executeProgram` into a PCB, you were attempting to copy data that is addressed relative to the start of one segment (the shell's stack segment) to a space that is addressed relative to the start of a different segment (the kernel's data segment). Specifically, the filename was stored in an array of characters in the shell's stack segment and the array of characters in the PCB is in the kernel's data segment. The problem arises because the `MOV` machine language instruction that ultimately copies the data assumes all addresses are relative to the data segment (DS) register. Thus, we need a way to copy data from one segment to another. We can do this using the `putInMemory` function from project 1 as follows:

```
/* kStrCopy(char *src, char *dest, int len) copy at most
len
 * characters from src which is addressed relative to
 * the current data segment into dest,
 * which is addressed relative to the kernel's data segment
 * (0x1000).
 */
void kStrCopy(char *src, char *dest, int len) {
     int i=0;
     for (i=0; i<len; i++) {
          putInMemory(0x1000, dest+i, src[i]);
          if (src[i] == 0x00) {
               return;
          }
     }
}
```

This `kStrCopy` method can be used to copy the filename into the PCB as long as it is used when the data segment register is set to the interrupted program's segment (i.e., not between calls to `setKernelDataSegment` and `restoreDataSegment`).

Modify your executeProgram, `handleTimerInterrupt` and `terminate` functions (and possibly other locations depending upon your implementation) so that any code that accesses the global data structures (or calls a function that does) is surrounded by calls to `setKernelDataSegment` and `restoreDataSegment`. Also add the `kStrCopy` function to your kernel and make appropriate modifications to the code that copies the filenames into PCBs.

Enabling Interrupts

You will also need to make a small change to each of your user programs to run them concurrently. It turns out that 16-bit real mode programs are started with hardware interrupts (e.g. the timer) disabled by default. Thus, you need to enable interrupts. The new `lib.asm` file provides a function with the prototype:

```
void enableInterrupts();
```

You should place a call to this function as the first line of main in each of your user programs.

Testing

If you have implemented all of the above functionality correctly, your kernel should run exactly as before. When you start it the shell should be launched. Of course, how that happened is different than it was before. Now, the shell was loaded into a free memory segment and its PCB was put in the ready queue. The kernel's main method entered the infinite while loop. Eventually a timer interrupt occurred, and the scheduler selected the shell from the ready queue and started it executing.

To more fully test your implementation of multiprogramming, you'll need a program (or two) that run for a while. The following user program will print out Hello 1000 times, pausing for a time between each output.

```
main() {
    int i=0;
    int j=0;
    int k=0;

    enableInterrupts();

    for(i=0; i<1000; i++) {
```

```
                print("Hello\n\r\0");
                for(j=0; j<10000; j++) {
                     for(k=0; k<1000; k++) {

                     }
                }
           }
           exit();
      }
```

Note: `print` and `exit` are the names of functions in my user library. You'll need to replace them with your own.

You should be able to run this program from your shell and then, while it is running, enter a command like `dir`. If all is working correctly you will see the directory listing intermingled with the output of the running program.

Improvements

1. Add a `yield` function to your kernel with the signature:

```
     void yield();
```

This function causes the executing process to give up the remainder of its time slice and be put back into the ready queue. Hint: You can simulate a timer interrupt with the interrupt function.

In addition, you should add a system call and user library function for yielding. The `handleInterrupt21` function should provide yield as follows:

yield:	give up the remainder of the time slice.	
	AX:	0x09
	BX:	Unused
	CX:	Unused
	DX:	Unused
	Return:	1

2. Add a `showProcesses` function to your kernel with the signature:

```
void showProcesses();
```

This function should display a list of the names and memory segment indices of all of the currently executing processes. Note the index of a memory segment is its index in the memory map (e.g. 0x2000®0, 0x3000®1, etc.).

In addition, you should add a user library function and a system call for showing the processes. Your `handleInterrupt21` function should now provide the following service:

showProcesses:	list the currently executing processes
AX:	0x0A
BX:	Unused
CX:	Unused
DX:	Unused
Return:	1

Finally, extend your shell so that it recognizes the command `ps`, which will display the names and memory segments of all of the running processes.

3. Add a `kill` function to your kernel with the signature:

```
int kill(int segment);
```

This function should kill the process that is executing in the segment with the specified index. When the process is killed it no longer executes and its memory and PCB are freed. This function returns

- 1 if the process is successfully killed
- -1 if there is no process currently running in the segment with the specified index

In addition, you should add a user library function and a system call for killing a process. Modify your `handleInterrupt21` function so that it provides the following kill service:

`kill`: kill the process executing in the segment with index indicated by BX

AX:	0x0B
BX:	the segment index
CX:	Unused
DX:	Unused
Return:	1 if the process is successfully killed
	-1 if there is no process executing in segment BX

Extend your shell so that it recognizes the command `kill <seg>`, which will kill the process currently executing in the segment with the specified segment index. The shell should print a message indicating if the process was successfully killed or not.

Bonus Features

1. Add a sleep function to your kernel with the signature:

```
void sleep(int seconds);
```

This function should cause the invoking process to sleep for the specified number of seconds. Sleeping the process should be removed from the ready queue until they are done sleeping at which point they should be returned to the ready queue. In addition, you should add a user library function and a system call for sleeping a process. Modify your `handleInterrupt21` function so that it provides the following sleep service:

`sleep`: cause the process to sleep for the number of seconds indicated by BX

AX:	0xA1
BX:	the number of seconds to sleep
CX:	Unused
DX:	Unused
Return:	1

2. At this point all programs executed in your shell are executed concurrently with the shell. In most modern OS shells, the default behavior when program is executed is to have the shell wait for the process to complete before continuing. Make the appropriate modifications to your shell and operating system so that it supports the following two forms of the execute command:

- `execute <prog>` the shell is suspended while prog executes. When prog is complete the shell begins executing again.
- `execute <prog> &` the shell and prog are executed concurrently. This is the current behavior of your shell.

3. Implement a static-priority scheduling algorithm. Make the appropriate modifications to your shell and operating system so that its supports the following form of the execute command:

`execute <prog> <priority> &`
the program is executed concurrently with the shell with the specified priority. The priorities should be integer values with higher numbers indicating higher priority. The highest priority process in the ready queue should always be run. If no priority is specified the process should run with a default priority.

Submission

Your project will be graded on its correctness as well as its style. Your source code files and your `run.sh` script should be nicely formatted and well documented.

Drew University:

Create a zip archive of your `project5` directory and submit to google classroom before the deadline. Your source code files and your `run.sh` script should be nicely formatted and well documented.

Washington and Lee University:

Copy your `project5` directory into your `turnin` directory.

Acknowledgement

This assignment as well as the accompanying files and source code have been adopted with minor adaptations from those developed by Michael Black at American University. His paper "Build an operating system from scratch: a project for an introductory operating systems course" can be found in the ACM Digital Library at: http://portal.acm.org/citation.cfm?id=1509022.