Operating Systems
CSCI-UA.0202 Fall 2013

Programming Assignment 1

Due Monday, October 7

Note: This is not a group project. Each student must write his or her own code and not use any code from other students.

Your assignment is to implement in C some interrupt handlers and a round-robin scheduler within a simulated hardware framework.

I have implemented a simple simulator that simulates hardware support for interrupts and a trap instruction. It uses an interrupt table, called INTERRUPT_TABLE, to point to the interrupt handlers (procedures) to be called when a trap or interrupt occurs. Specifically, the trap/interrupts are numbered as follows:

TRAP: 0 (occurs when a program makes a system call)

CLOCK_INTERRUPT: 1 (generated by the hardware on a regular basis)

DISK_INTERRUPT: 2 (generated when a disk read request has been completed)

KEYBOARD_INTERRUPT: 3 (generated when a keyboard read request is completed)

When a system call is made or an interrupt occurs, the procedure pointed to by the corresponding element of INTERRUPT_TABLE is called. For example, when a DISK_INTERRUPT occurs, the call INTERRUPT_TABLE[1]() is made automatically.

The processes that the simulated hardware runs are read in from the file "processes.dat". Each process consists of a sequence of the following instructions:

pid **`run`** *time*

pid **`diskread`** *size*

pid **`diskwrite`**

pid **`keyboardread`**

pid **`fork`** *newpid*

where *pid* is an integer giving the process id of the process containing the instruction. The **`run`** instruction, containing an integer *time* parameter, indicates that the process will run without performing I/O for *time* milliseconds of CPU time. The **`diskread`** instruction specifies that the process ask for data from the disk, with the *size* parameter indicating the amount of data being requested. The time that the disk read takes, during which the process is blocked, is a function of the *size*. The **`diskwrite`** instruction is a request to write data to the disk. Although it requires a system call, it is non-blocking (i.e. the program can continue to run while the data is being written to the disk). The **`keyboardread`** instruction specifies that the process perform a

blocking read from the keyboard (which the simulator assumes to take a fixed time of 100ms). Finally, the **`fork`** instruction creates a new process with the process id of *newpid*. The instructions for the new process also has to be given in processes.dat .

After the simulator has read in the process definitions from processes.dat, it assumes that the only active process is process 0 (i.e. the process with *pid* = 0). Thus, for any other process to execute, a fork instruction must be executed specifying the pid of that process.

A simple example of a sequence of instructions for several processes is as follows:

```
0 fork 1
0 fork 2
1 run 100
1 diskwrite
1 run 10
1 keyboardread
1 run 50
2 run 100
2 diskread 10
2 run 60
```

In this example, process 0 simply forks process 1 and 2 and then exits. Process 1 (after it is created by process 0 executing the fork) runs for 100 ms of CPU time, then performs a disk write, then runs for another 10 ms, then performs a keyboard read, and finally runs for 50ms more. Process 2 runs for 10 ms, performs a disk read, and then runs for 60ms. Other than the run instruction, the other instructions are assumed (for our purposes) to take 0ms.

Important note: You do not need to write any code that interacts with the information in processes.dat. That is handled by my hardware code. The above description is just so you can see what is happening.

I have also provided drivers that can be used to request a disk read or disk write from the simulated disk controller and a driver to request input from the keyboard. These drivers are:

disk_read_req(PID_type pid, int size);

keyboard_read_req(PID_type pid);

disk_write_req(PID_type pid);

The handler for the trap (pointed to by INTERRUPT_TABLE[0]), depending on the service being requested by the system call, will call one of the above drivers. The first two, disk_read_req() and keyboard_read_req() should blocking calls. After the trap handler calls either of these, the trap handler should put the process in the blocked state. The hardware will subsequently generate the necessary interrupts, DISK_INTERRUPT or KEYBOARD_INTERRUPT, when the (simulated) disk or keyboard operation completes.

When your trap handler causes a process to block, it will need to choose another process to run. Thus, you'll have to implement a round-robin scheduler that manages a queue of ready processes. Similarly, when a process has used it's quantum of CPU time, it will need to be put back on the ready queue and another process will need to be selected from the ready queue. The clock interrupt is generated by the hardware every 10ms. However, the scheduler should use a quantum

of 40ms, thus a clock interrupt does not mean that it is necessarily time to switch to another process.

What you need to do

You will need to have the gcc C compiler installed on your computer. On Windows, this means using the Cygwin environment, which can be downloading using the link on the course web page. For a Mac, gcc can be also downloaded using the link on the course web page. If you are running Linux, gcc should already be installed on your computer.

The first thing you need to do for this assignment is download a zip file containing the code I'm providing for the machine you are using. Follow the link on the course web page corresponding to the operating system (Windows, Mac OS X, or Linux) that your computer runs.

The good news is that most of the work in the simulation environment has been done for you. For example, I have written the code that actually reads in the processes from processes.dat, "executes" the processes, and generates the appropriate interrupts and traps. Your only tasks are (1) to implement the interrupt handlers for each of the interrupts and trap instruction listed above, (2) to install the interrupt handlers in the INTERRUPT_TABLE, and (3) implement the round-robin scheduler that is invoked by the interrupt handlers to choose the next process to run.

Be sure that your scheduler keeps a count of active processes, so it can terminate the simulator it (using the exit(0) system call, for example) when there are no more active processes.

What you are provided with

I have implemented the simulated hardware in a C file that I compiled using gcc into a file named hardware.o . The header file (which you will need to #include) is hardware.h . The drivers are compiled into a file called drivers.o, with a header file drivers.h . Your code should be placed in a file called kernel.c and compiled and linked with hardware.o and drivers.o . That is, using gcc, you would compile and link your file by:

gcc –m32 –o system kernel.c hardware.o drivers.o

This will generate an executable file called system.exe (if you are using cygwin on a PC) or system (on a Mac or Linux machine).

To make compilation easier, I have provided a Makefile for you. A Makefile is a configuration file that streamlines the build process (i.e. compiling and linking). If you simply type "make" in the shell, gcc will run and generate the executable described above.

I have provided some possibly helpful code to you in kernel.h and kernel.c . You are welcome to use the code if you wish, but you don't have to.

I have also provided a fully working compiled version of the entire system (including interrupt handlers and scheduler) in the file ben_system.exe (for the PC) or ben_system (for the Mac or Linux). You can compare the output of my program to the output of yours, to see if you are on the right track.

Important: Be sure to read carefully the code and comments in hardware.h, drivers.h, and kernel.h . These will provide you with the best guide for implementing your interrupt handlers and scheduler.

I have also provided several other files, process1.dat, process2.dat, and process3.dat, containing the process instructions. To get the hardware to execute one of these files, just copy it to processes.dat.

What the output should look like

Every time one of your interrupt handlers is called by the hardware, it should print out a message giving the current time (see the **`clock`** variable described in hardware.h) and information about the interrupt or trap that occurred. Every time the schedule chooses to run a different process, it should print out a message giving the current time and the process that is about to run. When a process finishes (as indicated by the trap specifying END_PROGRAM, see hardware.h), the current time, the pid of the process, and the total CPU time of the process should be printed out.

For example, given the sequence of instructions above, the output might look something like what is shown below.

```
Time 0: Creating process entry for pid 1
Time 0: Creating process entry for pid 2
Time 0: Process 0 exits. Total CPU time = 0
Time 0: Process 1 runs
Time 40: Process 2 runs
Time 80: Process 1 runs
Time 120: Process 2 runs
Time 160: Process 1 runs
Time 180: Process 1 issues disk write
Time 190: Process 1 issues keyboard read
Time 190: Process 2 runs
Time 210: Process 2 issues disk read
Time 210: Processor is idle
Time 270: Handled DISK_INTERRUPT for pid 2
Time 270: Process 2 runs
Time 290: Handled KEYBOARD_INTERRUPT for pid 1
Time 310: Process 1 runs
Time 350: Process 2 runs
Time 370: Process 2 exits. Total CPU time = 160
Time 370: Process 1 runs
Time 380: Process 1 exits. Total CPU time = 160
-- No more processes to execute --
```

Your output doesn't have to look exactly like this. However, the output of your program should reflect the same series of events as the output of my executable (again, ben.exe or ben.out). Be sure to run my executable on your computer and compare its output to that of your program.

When you have completed the assignment and are sure that your code works, email me your kernel.c file (to goldberg@cs.nyu.edu).

Please let me know if you have any questions.