CS 354 - FALL 1994
Assignment 12

Grader: Vaishnavi Anjur

Due before noon on Wednesday, Dec 14

1 Purpose

The purpose of this assignment is to give you experience implementing an exception handler. You will learn how a kernel manages the interface between a user program and the hardware, and gain some experience in dealing with interrupts.

2 Overview

You will use the program you wrote for Assignment 11, that reads in a user-entered string. However instead of printing the string just once in the forward and backward direction, your program must enter a loop that continuously prints the string forward, then backward, then forward , then backward and so on, until the user enters any character when you exit from the loop and print the number of seconds the program spent continuously printing the string.

Completing this assignment will involve modifying and extending the interrupt-driven kernel provided in **~cs354-1/public/kernel.12** (and printed in chapter 12) as well as writing your own code.

3 Details

3.1 Getting Started

This assignment involves modifying the interrupt-driven kernel provided to you to support the **clock** interrupt.

The unmodified code can be found in the file **~cs354-1/public/kernel.12**. There are a few minor differences between the on-line and text versions of this code, but they will not affect your program. The on-line version also contains comments giving you hints on where to make your changes. Make a copy of this file in your own directory, and use it as the basis of your assignment. Add your user-level (non-kernel) code at the end of your copy of the file.

3.2 Kernel Modifications

You will need to add support for a real-time clock to the kernel. Once per second, the simulator delivers a clock interrupt to the simulated CPU. Write the necessary code to handle this interrupt. You will need a counter to keep track of the number of seconds that have elapsed.

You will need to modify **HandleInt** and a few other routines to call your handler.

You will also need a syscall to determine if a key has been pressed indicating you should exit the loop.

3.3 User Program

After printing a welcome message, your user-level program will use getc to read a line character by character from the keyboard. When a return is pressed on the keyboard, your program will enter a loop displaying the string that was entered forwards and backwards repeatedly. All communication with the kernel must be done via the **syscall** instruction, not with MAL instructions such as done, getc, putc, or puts. Your program should be able to buffer input lines of 80 characters (not including the newline.) It may ignore characters beyond the 80th, but should not crash.

4 Example

An example of what the program execution might look like:

```
Please enter a string : abcdEYURSFMopertLK
abcdEYURSFMopertLK KLtrepoMFSRUYEdcba
abcdEYURSFMopertLK KLtrepoMFSRUYEdcba
abcdEYURSFMopertLK KLtrepoMFSRUYEdcba
abcdEYURSFMopertLK KLtrepoMFSRUYEdcba
Q                     <---- User entered character
Time spent = 50 secs  <---- This is just a sample value
Bye!
```

5 Notes About the Kernel Code

The kernel provided for this assignment is fairly complex, and it is not always apparent why it does things the way it does. This section of the assignment will attempt to explain some of its mysteries. You will also find it extremely helpful to read chapter 12 and the comments in the kernel code itself.

5.1 Overview of the Kernel

When an exception occurs, the simulator jumps to the kernel. The kernel is always entered at memory location **0x80000080**, at the beginning of the kernel code. After saving registers (the code labeled OK) the kernel uses a jump table to branch to the handler routine for the particular type of exception that occurred. If you use additional registers, you will need to add code to save and restore them. The two types of exception you need to worry about for this assignment are system calls and interrupts. The handlers for system calls and interrupts work together to provide asynchronous I/O for the user program. When an exception handler routine is finished, it jumps to the label **Return**. This code restores saved registers, then returns to the user program with an **rfe** (return from exception) instruction. Note that since all saved registers are restored, any exception handler routine which wishes to modify the user program's registers (such as the **getc** system call routine) must modify their saved values.

The interrupt handler is labeled **HandleInt**, and is located shortly after the jump table. HandleInt does not try to determine which interrupt occurred. It could find out by examining the **Cause register**, but simply calling the handlers for each type of interrupt in turn is faster when there are only a few types of interrupts. You will add a call to your clock interrupt handler here.

The system call handler is labeled **HandleSys**, and is found shortly after **HandleInt**. The syscall instruction causes an exception before the PC is incremented. Therefore, HandleSys must increment the return PC or the system call would be re-executed as soon as it returned. HandleSys tests the value passed in register \$2 (=\$v0) and jumps to the appropriate system call handler routine (labeled **Putc**, **Puts**, **Getc**, or **Exit**.) You need two syscall numbers one to determine if the user has entered a character indicating the program should exit from the loop and another to get the current value of the timer. You can use number **8** in register \$2 for the first syscall and number **9** for the second.

5.2 Interrupts and I/O

Interrupts can occur at any time, and signal your program that an event has occurred. Bit(s) are set in the Status register indicating which device(s) caused the interrupt, and a flag in the memory-mapped status location for the device(s) is set. In this kernel, the interrupt handlers use the memory-mapped status locations to determine which device(s) need service.

Once per second, the most significant bit of the memory-mapped I/O location labeled **ClockStatus** (defined with a .eq directive at the beginning of the kernel) is set and the simulator delivers a clock interrupt to the simulated CPU. You will write an interrupt handler which checks the value of ClockStatus to keep track of the number of seconds which have passed. Checking the value of the ClockStatus resets it so that the next clock interrupt can set it again.

The routine which prints a character to the screen is called **DP_Handler**, and is found near the end of the kernel. This routine is called from various places, so it does not know ahead of time if the screen is actually ready to accept another character. Therefore, the first thing it does is check the display's status by loading from the location DisplayStatus. If the value is nonnegative, the display is not ready to accept a character and the handler branches to **PutRet**. Otherwise, the display is ready and DP_Handler checks if there is any data to print. Characters to be printed are kept in a queue. Characters are placed in this queue by the **putc** and **puts** system call handlers and by the keyboard interrupt handler (for echoing typed characters.) If there is a character to print (the queue head and tail pointers are not the same), DP_Handler increments the queue tail pointer and prints the character by copying it to **DisplayData**.

The putc and puts system call handlers put the character to be printed in register 4, then call the routine **PutR4**. PutR4 checks if there is room in the output queue for the character. If there is, it puts it in the queue and calls DP_Handler, which will print the character if the screen is ready to accept it. Only one interrupt is raised each time the screen is ready to print. If there is no data to print, DP_Handler simply returns. Since only one interrupt is raised when the screen is ready for display, PutR4 must call DP_Handler directly, instead of simply placing the character in the queue and letting it be printed on the next interrupt. If the output queue is full, PutR4 jumps to **Full**, which calls DP_Handler to try to print a character. It then jumps back to PutR4, which will check the queue again. In this way the kernel will spin, calling DP_Handler until it's able to print the character.

Keyboard input is handled similarly to screen output. The keyboard interrupt handler **KB_Handler** (found at the end of the kernel code) tests the KeyboardStatus location to see if a character has been typed. If it has, it tries to put it into the input queue. If there is no room, the character will be lost. In this case KB_Handler jumps to **GetQueueFull**, which turns off keyboard interrupts (see the table of interrupt masks in the comments at the beginning of the kernel.) Interrupts will be enabled the next time getc is called.

Characters are removed from the input queue by the routine **GetR4**, called from **Getc**. The first thing GetR4 does is turn on keyboard interrupts, in case GetQueueFull had disabled them. It then checks if there is a character in the queue. If there is, it returns it in register $2 (=$v0). Getc then stores register $2 into tmpr2, so Return will pass the character back to the user program. If the queue is empty, GetR4 jumps to **GotNone**, which calls the display interrupt handler (to keep output going while waiting for input) and the keyboard interrupt handler (to check for input) before jumping back to GetR4. In this way, the kernel spins until a character is typed.

5.3 Programming and Debugging in the Kernel

- Writing kernel code is much like writing user-level code, but there are a few differences. First, kernel text is in the .ktext segment instead of the .text segment, and kernel data is in the .kdata segment instead of the .data segment. Be sure to use the proper assembler directives.

- There is no kernel stack (at least not in the kernel used for this assignment.) This means that procedure linkage is handled differently in the kernel. Instead of saving registers on the stack, registers must be saved in the .kdata segment. It must call DP_Handler, but it needs to preserve the values in registers $4 and $31. Therefore, it saves $4 and $31 to temporary locations and restores them after the call.

- This kernel uses the convention that all user registers modified by the kernel are saved when the kernel is entered and restored just before it exits (in Return.) Therefore, whenever a routine needs to pass a value back to the user program in a register, it does so by putting the value in that register's storage location. See the routine Getc for an example. If your version of the kernel modifies a register not already saved (such as \$3 or \$v1) you must add code to save and restore it in this way.
- Debugging in the presence of interrupts is difficult. The simulator does not suspend interrupts while the program is stopped, so when you step or continue the program, display and clock interrupts usually occur immediately. You can avoid this problem by manually modifying the Status register to disable interrupts. Use the "**Set Value**" button in the simulator to do this. Clearing the least significant bit in the Status register disables all interrupts. In addition, clearing any of bits 8 through 15 disables interrupts from specific devices (see the comments near the beginning of the kernel for which bits correspond to which devices.) Be sure to re-enable interrupts eventually!

6 Turn In

Turn in your solution by using the handin12 program by typing:

```
~cs354-1/bin/handin12 filename
```

where

```
<filename>
```

is the file containing the source code of your program (both kernel and user code.) As always, hand in your solution (using the handin12 program) only once.

7 Quote

"We're gonna work round the clock, Saturday, Sunday," he said. *"We're gonna do it all, and bathe in glory, guys."* Late on Saturday afternoon, however, while running one of the last, advanced Eagle diagnostics, Gollum came down with a bad-looking bug.

The failure occured when the diagnostic program called for an *"interrupt,"* an order that the computer pause, store all unfinished work, and start in on another job. A computer must deal with interrupts smoothly and regularly if it is going to handle many terminals at once.

Guyer, Holberger and Rasala took on the bug. They studied the printed listing of the steps in the diagnostic program, and they saw that in addition to everything else – as if interrupts weren't tricky enough – *the program was trying to test Eagle's ability to move from one block of memory to another.*

–**Tracy Kidder, The Soul of a New Machine, Avon Books, 1981.**