# For More Practice

### Writing Code to Benchmark I/O Performance

**8.25** [1 day–1 week] <§§8.2–8.5> Take your favorite computer and write programs that achieve the following:

1. Maximum bandwidth from and to a single disk
2. Maximum bandwidth from and to multiple disks
3. The maximum number of 512-byte transactions from and to a single disk
4. The maximum number of 512-byte transactions from and to multiple disks

What is the percentage of the bandwidth that you achieve compared to what the I/O device manufacturer claims? Also, record processor utilization in each case for the programs that are running separately. Next, run all four together and see what percentage of the maximum rates you can achieve. From this, can you determine where the system bottlenecks lie?

### Finding I/O Bandwidth Bottlenecks

**8.35** [20] <8.3–8.6> Assume all accesses in the I/O system described in Exercise 8.26 are 4 KB block reads. If there are a total of two I/O buses, two DMA controllers, and eight disks, find the maximum number of I/Os the system can sustain in steady states assuming that the reads are uniformly distributed to the disk. What is the sustained I/O bandwidth?

**8.36** [15] <§§8.3–8.6> With the organization in Exercise 8.31, clearly it is possible to saturate the I/O buses because you have two of them at 100 MB/sec and eight disks at 40 MB/sec. Compute the minimum block size (which should be a power of two) that will saturate the I/O buses. For this block size, how many I/O operations per second can the system perform and what is the I/O bandwidth?

### I/O System Operation

**8.37** [15] <§§7.3, 7.5, 8.4, 8.5> Consider a write-back cache used for a processor with a bus and memory system as described in Exercise 8.18 (assume that writes require the same amount of time as reads). The following performance measurements have been made:

- The cache miss rate is 0.05 misses per instruction for block sizes of 8 words.
- The cache miss rate is 0.03 misses per instruction for block sizes of 16 words.
- For either block size, 40% of the misses require a write-back operation, while the other 60% require only a read.

Assuming that the processor is stalled for the duration of a miss (including the write-back time if a write-back is needed), find the number of cycles per instruc-

tion that are spent handling cache misses for each block size. (Hint: First compute the miss penalty.)

### Using SPIM to Explore I/O

**8.41** [2 days–1 week] <§8.5, Appendix A> This assignment uses SPIM to build a simple set of I/O routines that will perform I/O to the terminal using polling. First, you need to build two I/O routines, whose C declarations and descriptions are shown below:

```
void print (char *string);
```

The procedure `print` takes a single argument, which is the address of a null-terminated ASCII string. All of the characters of the string except the null-terminating character should be output by `print`. It should print the characters one at a time, waiting for each character to be output before sending the next one. It should not return until all the characters have been output. The procedure `print` should work for strings of any length. This version of `print` should not use interrupts; just test the ready bit of the transmitter control register continuously until the device is ready.

```
char getchar();
```

The procedure `getchar` takes no arguments and returns a character result. If `getchar` waits until a character has been typed on the terminal, then it should return the character's value in `$v0` (the result register). Do not use interrupts; simply test the ready bit continuously until a character has arrived.

Write a main program that uses these two procedures to read a line from the terminal, which will be terminated by a carriage return. Then print the entire line to the terminal, including a carriage return and line feed. All your code should obey the conventions in Appendix A for procedure calling, stack usage, and register usage.

### Writing Code to Perform I/O

**8.42** [3 days–1 week] <§8.6, Appendix A> Your assignment is to build an interrupt-driven mechanism for buffered I/O to and from the terminal. (This exercise handles output only; Exercise 8.43 handles input.)

For the output-only portion, there are three parts to the program:

1. A main program, which repeatedly calls procedure `print` to print the string "`I know what I am doing.`"

2. The procedure `print`, which stores the output characters in a buffer shared by it and the interrupt routine
3. The interrupt routine, which copies characters from the output buffer to the transmitter

You need to write all three routines. The routine `print` and the interrupt routine should communicate by using a shared circular buffer with space for 32 characters. The `print` procedure should take a string as argument and add the characters of the string to the output buffer one at a time, advancing as soon as there is space in the buffer. Keep in mind that `print` should not manipulate the terminal device registers directly, except to make sure that transmitter interrupts are enabled. Furthermore, `print` should contain additional code to deal with a full output buffer. The main program generates characters much faster than they can be output, so the buffer will quickly fill up. In a real system, if the output buffer fills up, the operating system will stop running the current user's process and switch to a different process. Your program doesn't need to support multiple users, so `print` can take a simpler approach: it just checks the buffer over and over again until eventually it isn't full anymore. The buffer is full when the next position in which `print` wants to insert a character has not been emptied by the interrupt routine.

After writing print, write the interrupt routine called by `print`. Here is a list of things the interrupt routine must do:

1. If the transmitter is not ready, then the interrupt routine should not do anything. (You shouldn't have received an interrupt in the first place if the transmitter isn't ready, but it's a good idea to check anyway.)
2. If the output buffer isn't empty, copy the next character from the output buffer to the Transmitter data register and adjust the buffer pointers.
3. If the output buffer is empty, turn off the interrupt-enable bit in the Transmitter control register. Otherwise, continuous interrupts will occur. Each time it deposits a character in the buffer, `print` will need to turn this bit on.
4. Don't forget that you must save and restore any registers that you use in the interrupt routine, even temporary registers such as register `$t0` and register `$t1`. This is necessary because interrupts can occur at any time and those registers may be in use at the time of the interrupt. You must save the registers on the stack. The only exceptions to this rule are registers `$k0` and `$k1`, which are reserved for use by interrupt routines; these registers need not be saved and restored. One of these registers can be used to return from the interrupt routine back to the code that was interrupted.

Test your code by writing the main routine that calls `print` to print the string. It should output lines continuously, with each line containing the characters "`I know what I am doing.`"

**8.43** [3 days–1 week] <§8.6, Appendix A> Extend the code you've already written to be able to handle interrupt-driven input. This program should do input in the same way as the previous program did output: by using a buffer to communicate between the routine `getchar` and the interrupt routine. Be aware that `getchar` returns a character from the buffer, waiting in a loop if no characters are present. Similarly, the interrupt routine will add characters as they are typed, discarding characters if the buffer is full when they arrive. For this, an eight-entry buffer should work well.

Use these two routines to read characters from the terminal and to output them to the terminal. Try typing characters rapidly to make sure your program can handle the output or the input buffer filling up. For example, if you type two or three characters rapidly, the output buffer may fill up. However, no output should be lost: the print procedure will simply have to spin for a bit, during which time additional input characters will be buffered in the input buffer. If you type eight or ten characters very rapidly, then the input buffer will probably fill up. When this happens, your interrupt routine will have to discard characters: the program should continue to function, but there won't be any output of the discarded input characters you typed. Once the output catches up with the input, your program should accept input again just as if the input buffer had never filled up.