CPSC 213: Assignment 7

Update Nov 4, 3:45pm. Clarification of ".address" in A7-1.s and "NYI" cases in A7-2.c. Updated Oct 31, 6:15pm. Updated code.zip.

Due: Thursday, November 6, 2014 at 11:59pm

After an eight-hour grace period, no late assignments accepted.

Goal

This assignment has two goals.

The first is to explore the use of jump tables to implement switch statements. You will examine snippet B, read a mystery assembly file, and transform C code that uses switch statements into code that uses jump tables and gotos.

The second goal is to examine asynchronous I/O programming. You are given a C file that models a simplified disk with an asynchronous read operation, simulated DMA, and interrupts. You are also given two programs that perform a sequence of disk reads in different ways. The first of these programs is completely implemented. It access the disk sequentially, waiting for each request to finish before moving on. The other programs is partly implemented, with the rest left to you. It improves on the sequential version using event-driven programming. You will gain some experience with this style of programming and then compare the runtime performance of the two alternatives. You'll continue to with this example in Assignment 8, where you will implement a third alternative that uses threads.

Groups

You may do this assignment in groups of two if you like. If you do, be sure that you both contribute equally to the assignment and that you each work on every part of the assignment. Do ***not*** split up the assignment in such a way that one of you does one part and the other does another part. Keep in mind that the core benefit to you of doing the assignment is the learning that happens while you do it. Each assignment is worth only around 1.8% of your grade, but what you learn while doing the assignment goes a long way to determining the other 85%.

If you choose to do the entire assignment with a partner, submit one solution via **`handin`** and list both of your names, student numbers, and computer-science login ids in the **`README`** file.

Alternatively, you can also choose to collaborate with another student for a portion of the assignment and have your work marked individually. Do do this, simply submit your own version via **`handin`** and list the other student as a collaborator in your **`README`** file. Just don't do this if you and your partner are turning in identical solutions, as we would like to realize marking efficiency where possible. You may have at most one collaborator for the assignment; i.e., you

can not collaborate with one student for one part of the assignment and another student for another part.

Code Provided this Week

The file www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a7/code.zip contains the following files that you will use with this assignment

- **SB-switch.**{**c**,**s**} and **SB_switch.java**
- **A7-1.s**
- **A7-2.c**
- **max.s**
- **test.s**
- **Makefile**
- **disk.**{**c**,**h**}, **queue.**{**c**,**h**}, **uthread.{ch}**, and **uthread_util.h**
- **sRead.c** and **aRead.c**

Part 1: Switch Statements

Part 1a: Snippet SB

Examine the code for **SB-switch**. Carefully read the three different implementations in the C file. Run the assembly version in the simulator and observe what happens.

Unlike previous assignments you are not required to document anything here. Doing this will help you with Parts 1b and 1c. You are not required to turn in anything for Part 1a.

Part 1b: Reading Assembly Code

The file **A7-1.s** contains uncommented assembly code that corresponds to a single C procedure starting at address **0x300** and another procedure that sets up the stack and calls the first procedure. Read this code and run it in the simulator. Comment every line and write an equivalent C program that is a likely candidate for the program the compiler used to generate this assembly file (with the exception of the stack-setup code, which is only for the **.s** file). Call this program **A7-1.c**. Note that **A7-1.s** uses the directive **.address** instead of **.pos**; they are equivalent.

Part 1c: Implement Switch Statements with Jump Tables

The file **A7-2.c** is a C implementation of CPU.java; i.e., it is a sm213 virtual machine.

It has been setup to load and execute two assembly programs include in code.zip: **max.s** and **test.s**. It does this by loading the machine representation of the instructions since this program does not include an assembler. It is easy to get the translation from assembly to machine code from the Java simulator (its just to the left of the assembly code, line by line).

Take a look at this so you understand what is going on, but don't worry about the details of the instructions. They are stored in **`A7-2.c`** as constants (i.e., **`loadMax()`** and **`loadTest()`**).

The difference between **`max.s`** and **`test.s`** is that **`max`** does something interesting (it computes the maximum value of an array of integers) and **`test`** simply executes each instruction in the ISA once (in order of their op code). For **`max`**, the virtual machine prints the value stored in memory at address 0 after **`max`** finishes; this is the maximum value of the list that starts at address 4. Notice that it works. Modify the integer array in **`loadMax`** so that it computes a different maximum.

Your task is to write an equivalent C program called **`A7-2-jumptable.c`** that replaces the two switch statements in **`exec()`** with gotos using a jump table and label pointers as shown in class. Note that label pointers are a gnu C extension and so some compilers may not support them (all versions of gcc and llvm (the Mac compiler) do support this extension). So, you may need to move to a lab computer to test this your program. Leave cases marked "NYI" empty; i.e., they are cases with an empty body so just implement them that way.

To test your program it is sufficient to show either that it computes the correct value for max or that **`loadTest`** runs okay. Start with **`loadMax`**. If it works; you are done. If not, then switch to **`loadTest`** and change every case arm to include a print statement that prints the value of the case label (e.g., "**`case 2: printf ("2\n"); …`**"). This will simplify debugging greatly. Then if you show that you print the case labels in the correct order, you are done.

Part 2: Asynchronous Programming

Background

The Simulated Disk

The simulated disk is implemented in the file **`disk.c`** and its public interface is in **`disk.h`**. A disk contains a collection of disk blocks named by a block number. This disk simulates a fixed, per-access read latency of 10 ms (1 ms is 10^{-3} seconds), which is about the average access time of real disks. This means that it takes the disk 10 ms to process a single read request. However, the disk can process multiple requests in parallel. When it completes a request, it does what a real disk does, it uses a direct-memory transfer (DMA) to copy the disk data into main memory and it delivers an interrupt to the CPU. In this case, of course, the DMA is just a memory-to-memory copy between parts of your program's memory. And the interrupt is delivered by calling a specified handling procedure you register when you initialize the disk.

To initialize the disk using an interrupt handler called **`interruptServiceRoutine`**, you call the following procedure at the beginning of your program (all of the provided files already do this).

```
disk_start (interruptServiceRoutine)
```

To request that the first **nbytes** of the content of the disk block numbered **blockno** be transferred into **buf** you call.

```
disk_scheduleRead (buf, nbytes, blockno)
```

This procedure returns immediately. Then 10ms later, the target data is copied into **buf** and an interrupted is delivered to you by calling **interruptServiceRoutine**. The disk completes reads in request order, and calls **interruptServiceRoutine** for each completion.

Now, since the simulated disk doesn't really store data, it does not transfer anything interesting into **buf** other than one thing. It write the integer **blockno** into the buffer so that you can test your code to be sure you have the right disk block. For example if you make the call

```
char buf [100];
disk_scheduleRead (buf, 100, 37);
```

And wait the 10ms for the interrupt, and then examine **buf**, you will see it unchanged except for the beginning, which will store the number **37**. Thus the following expression evaluates to true.

```
*((int*) buf) == 37
```

Note — **and this is important** — the interrupt operates just like a regular interrupt. It will interrupt your program at some arbitrary point to run the handler, continuing your program when the handler returns. There are some potentially difficult (and I mean horrible) issues that arise if your program and the handler access any data structures in common or if the handle calls any *non-reentrant* procedure (e.g., **malloc** or **free**). You are provided with the implementation of a reentrant, thread-safe queue that is safe to access from the handler and your program. It is also okay to access the target data buffer (**buf)** in the handler. Do not access any other data structures in the handler and do not call **free** from the handler (its okay to call **dequeue**). You will learn more about this issue in a couple of lectures, but it is not a concern for this assignment (other than to avoid the problem).

The disk provides one addition operation that you can call to wait until all pending reads have completed.

```
disk_waitForReads()
```

The Queue

The files **queue.c** and **queue.h** provide you with a thread-safe, re-entrant implementation of a queue. If you need to enqueue information in your program that is dequeued in the interrupt handler, use this queue.

To create a queue named something like **prq**, for example, you declare the variable like this.

```
queue_t prq;
```

Then before you use the queue you need to initialize it, in **main**, for example like this.

```
queue_init (&prq);
```

To add an item pointed to by the variable **item** (of any type) you do this:

```
queue_enqueue (&prq, item);
```

To get an item (e.g., of type **struct Item***), you do this:

```
struct Item* item = queue_dequeue (&prq);
```

Makefiles

The provided code contains a file called **Makefile** that describes how this week's program should be built. In general, most C projects have a makefile like this. To compile the program **sRead**, for example, type the following at the command line:

```
make sRead
```

To compile every program just type this:

```
make
```

To remove executables, object files and other derived files type this:

```
make clean
```

Timing the Execution of a Program

The UNIX command **time** can be prepended to any command to time its execution. When the command finishes you get a report of three times: the total elapsed clock time, the time spent in user-mode (i.e., your program code) and the time spent system-mode (i.e., the operating system). The format is otherwise a bit different on different platforms.

For example if you type this:

```
time ./sRead 100
```

You will get something like this on Mac OS when **sRead** completes:

```
1.074u 0.010s 0:01.08 100.0%   0+0k 0+0io 0pf+0w
```

And on Linux:

```
real  0m1.100s
user  0m1.084s
sys   0m0.000s
```

Ignore the user (u) and sys (s) times; they really are approximations. Pay attention only to the real, elapsed time, which is 1.08 s in the Mac example and 1.1 s in the Linux example.

You will use this command to assess and compare the runtime performance of the three alternative programs.

Part 2a: Synchronous Reads

Examine the program **sRead.c**, compile it by typing **make sRead** and run it.

You run it from the command line with one argument, the number of reads to perform. For example, to perform 100 reads, you type this:

```
./sRead 100
```

Temporarily modify **handleRead** to assert that the value at the beginning of **buf** is something other than **blockno** to confirm that this is really working. For example

```
assert (*((int*) buf) == blockno-1)
```

Now, restore the **assert** statement and temporarily add a **printf** statement to **handleRead** to print number at the beginning of every the buffer something this:

```
printf ("buf = %d, blockno = %d\n", *((int*) buf), blockno);
```

Again, this is just to convince yourself that this works. Now, remove the **printf**.

Finally, time the execution of the program for executions that read various numbers of blocks. For example

```
time ./sRead 10
time ./sRead 100
time ./sRead 1000
```

Knowing how the simulated disk and this program perform you should be able to explain why you see the runtime performance you do. Do so, in **README.txt**; this is **QUESTION 1**.

Part 2b: Asynchronous Reads

Now open **aRead.c** in an editor and examine it carefully. This version of the read program each read should not wait for the disk to complete, as the synchronous version does. Instead, it should schedule each read in **run** and call **handleRead** for each of them when their interrupt is delivered. You will receive interrupts in the same order as calls to **disk_scheduleRead**. So, for example, if you schedule reads for **blockno** values of 0, 1, and 2, in that order, the first time **interruptServiceRoutine** runs it will be to tell you that the read for block 0 has completed. The second call will be for block 1 and so on. But note, that the interrupt is just an interrupt. It does not transmit any values; the interrupt service routine has no parameters. So, you will need to remember these somehow. Remember the queue?

Compile and test your program the same way that you did for **sRead**.

Measure the runtime performance of **aRead** for executions that read different numbers of disk blocks as you did with **sRead** and so that you can directly compare their performance. There is some experimental error and so you should run each case at least three times and take the **minimum**. The reason for taking the minimum is that this is the most reliable way to factor out

extraneous events that you see as noise in your numbers. Obviously, this is not a good way to determine the *expected* behaviour of the programs; its just a good way to compare them.

Now you know more and have more data. Record the **`time`** values you observe for both **`aRead`** and **`sRead`** for a few different numbers of reads. Be sure to choose meaningful values for this parameter; e.g., at least a small one of around 10 and a large one of around 1000 (though if your system is too slow to run either of these for 1000, choose a smaller number). Explain what you observe: both *what* and *why*. The why part is important: carefully explain *why* one of these is faster than the other. Record your observations and data in **`README.txt`**. This is **`QUESTION 2.`**

Requirements

Here is a summary of the requirements for each part of this assignment.

1. Examine the execution of Snippet B. There is nothing to hand in for this step.
2. Add comments to **`A7-1.s`** and write **`A7-1.c`**.
3. Write **`A7-2-jumptable.c`**.
4. Run **`sRead`** as described in Part 2a and then measure its runtime performance. Answer **`QUESTION 1`**.
5. Complete the implementation of **`aRead.c`**. Run it as described in Part 2b and then measure its runtime performance. Answer **`QUESTION 2`**.

What to Hand In

Use the **`handin`** program. The assignment directory is **a7**.

1. A single file called "**`README.txt`**" that includes your name, student number, four- or five-digit cs-department undergraduate id (e.g., the one that's something like **`a0b1`**), and all written material required by the assignment as listed below.
2. If you had a partner for the entire assignment, turn in only one copy of your joint solution under one of your ids and list both student's complete information in the **`README.txt`** file and include the text "**`PARTNER - MARK JOINTLY`**".
3. If, on the other hand, you collaborated with another student for a portion of the assignment, but your solutions are not identical and you want them marked individually, each of you should include the other student's complete information in your **`README.tx`**t file, include the text "**`COLLABORATOR - MARK SEPARATELY`**", and turn in copies separately via **`handin`**.
4. The files **`A7-1.s`** and **`A7-1.c`**.
5. The file **`A7-2-jumptable.c`**.
6. Your answer to **`QUESTION 1`** in the **`README.txt`** file.
7. The file **`aRead.c`**.
8. Your answer to **`QUESTION 2`** in the **`README.txt`** file.