CPSC 213 – Assignment 9

IO, Asynchrony and Threads

Due: Friday, March 24, 2017, at 11:59pm
After 12 hour, no-penalty, grace period, no late assignments accepted after that.

Goal

The goal of this assignment is to examine asynchronous I/O programming and threads. You are given a C file that models a simplified disk with an asynchronous read operation, simulated DMA, and interrupts. You are also given three 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 two programs are partly implemented, with the rest left to you. The first 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. The second uses threads to turn the asynchronous operations into synchronous ones, allowing you to write code with the first program and get performance like the second.

Background

The Simulated Disk

A simulated disk is implemented in the file `disk.c` and its public interface is in `disk.h`.

Recall that a disk contains a collection of disk blocks named by a block number. Applications running on the CPU request disk blocks (usually 4-KB or so at a time) using a combination of PIO, DMA and interrupts. They use PIO to tell the disk controller which blocks they want and where in memory they want the disk to place the data (i.e., the content of the blocks). The disk controller uses DMA to transfer this data into memory and then sends an interrupt to the CPU to inform it that the transfer has completed. The total elapsed time for this operation is referred to as the *latency* of the read.

Our simulated disk models 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 (interrupt_service_routine);
```

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

```
disk_schedule_read (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 `interrupt_service_routine`. The disk completes reads in request order, and calls `interrupt_service_routine` for each completion.

Like a real disk, multiple calls to `disk_schedule_read` will be handled in parallel; this fact is helpful to know when comparing the performance of the different versions of the read programs you will be modifying and running.

Now, since the simulated disk doesn't really store data, it does not transfer anything interesting into `buf` other than two things. It writes two integers to the beginning of the buffer. The first is the requested block number that you will use to test your code to be sure you have the right disk block. For example if you make the call

```
char buf [8];
disk_schedule_read (buf, 8, 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
```

The second is the value associated with the block. Your program must sum as specified in the code all of these values and print this sum when the program terminates. So you will declare a global variable `sum` like this.

```
long sum = 0;
```

Then do this for each read as it finishes.

```
sum = sum * 1.1 + *(((int*) buf) + 1)
```

And then print `sum` at the end of the program like this.

```
printf ("%ld\n", sum);
```

The implementation of the disk that you are provided sets this value field in each block to the number 1 and so the sum you print should equal the total number of blocks you read. We will change the value the disk returns when we mark you assignment, so be sure to really read the value.

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 truly mean horrible) issues that arise if your program and the handler access any data structures in common or if the handler 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`).

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

```
disk_wait_for_reads()
```

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.

Hints: Creating and Joining with Threads in Run

You should create a separate thread for each call to `read` using:

```
uthread_create (void* (*start_proc)(void*), void* start_arg)
```

Also note that it is necessary to joint with a thread (i.e, `uthread_join(t)`) if you want to wait until the thread completes. You will need to do this in `run`, because when `main` returns the program will terminate, even if there are other threads running.

Hints: Blocking and Unblocking Threads

A thread can block itself at any time by calling `uthread_block`. Another thread can wakeup a blocked thread (`t`) by calling `uthread_unblock(t)`. Recall that you will need to block threads after they call `disk_scheduleRead` and before they call `handleRead`. And that this blocked thread should be awoken when the disk read on which it is waiting has completed. Also recall that a thread can obtain its own identity (e.g., for unblocking) by calling `uthread_self()`.

What to Do

Download the Provided Code

The file *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a9/code.zip* contains the code files you will use for this assignment this includes the implementation of uthreads, spinlocks, a simulated disk, and other files used in each of the questions below.

Question 1: Synchronous Disk Read by Wasting CPU Time

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.

Record your data, observations, and explanation in the file `Q1.txt`.

Question 2: Implement Asynchronous Read — aRead.c

Now open `aRead.c` in an editor and examine it carefully. This version of the read program will use event-driven style programming to handle the disk reads asynchronously. That is, each read request registers a completion routine and returns immediately. The completion routines are then called by the disk interrupt handler when each read completes.

To keep things simple, the disk delivers interrupts in request order (i.e. the same order as calls to `disk_schedule_read`). So, for example, if you schedule reads for `blockno` values of 0, 1, and 2, in that order, the first time `interrupt_service_routine` 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 the details of each pending read using a request queue. You must use the queue; this is the only data structure that `handle_read` is allowed to accesses.

Compile and test your program the same way that you did for `sRead`. But sure to accumulate the block values and print their sums as `sRead` does and as described in the Background section and use `valgrind` to ensure that it is free of memory leaks.

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 data, observations, and explanation in the file `Q2.txt`.

Question 3: Using Threads to Hide Asynchrony

Now open the new file `tRead.c` in an editor and examine it carefully. Compare this version to both of the other versions on worked on in Assignment 7 (i.e., `sRead.c` and `aRead.c`). The goal in this new version is to read and handle disk blocks sequentially in a manner similar to `sRead` but to do so using threads to get performance similar to `aRead`.

To do this, it will be necessary to create multiple threads so that threads can stop to wait for disk reads to complete without the negative performance consequences seen in `sRead.c`.

See the Background section above for some notes that help with the implementation.

Compile and test your program. Again, be sure to accumulate and print the sum of the block values and ensure that it is free of memory leaks.

Evaluate this version as you did the other two.

Compare their performance and record your data. Compare both the elapsed time (as you have previous) and the system time of `aRead` and `tRead`, which measures the approximate amount of time the program spends running in the operating system. If there is a significant difference note it.

Record your data, observations, and explanation in the file `Q3.txt`.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/a9`, it should contain the following ***plain-text*** files.

1. `README.txt` that contains the name and student number of you and your partner
2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything.
3. For Question 1: `Q1.txt`.
4. For Question 2: `aRead.c` and `Q2.txt`.
5. For Question 3: `tRead.c` and `Q3.txt`.