CPSC 213 – Assignment 9

IO, Asynchrony and Threads

Due: Friday, June 15, 2018 at 11:59pm
After a 24-hour grace period, late assignments will only be accepted in extenuating circumstances with documentation.

Goal

The goal of this assignment is to examine asynchronous programming and threads. You are given a C file that models a simplified disk with asynchronous read operations, simulated DMA, and interrupts. You are also given four 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 next two programs are partly implemented, with the rest left to you. The first improves on the sequential version using asynchronous, 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 like the synchronous program and get performance like the asynchronous one. The final program goes back to the asynchronous model for a more interesting challenge.

Background

The Simulated Disk

The simulated disk is implemented in the file `disk.c` and its public interface is in `disk.h`. You can completely ignore the implementation; you'll just use it.

Recall that a disk contains a collection of blocks named by a block number. Programs 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 to use an interrupt handler called `interrupt_service_routine`, 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 content of the disk block numbered `blockno` be transferred memory at the address stored in `result`, you do this.

```
disk_schedule_read (result, blockno);
```

This procedure returns immediately. Then 10ms later, the target data is copied into memory at the address in `result` 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.

Unlike a real disk, each block of this disk is just an integer. So, for example to schedule a read of the value of block `1234` into the variable `result`, you could do something like this.

```
int result;
disk_schedule_read (&result, 1234);
```

But, be careful, because this read will happen later and so be sure that (a) `result` isn't a local variable that is deallocated before the read completes and (b) if you have multiple outstanding disk reads, they don't all use the same `result` variable.

Note — **and this is important** — the interrupt operates just like a real interrupt. It will interrupt your program at some arbitrary point to run your interrupt service routine (isr) and then continue, your program when the isr completes. There are some potentially difficult (and I truly mean horrible) issues that arise if your program and the isr access any data structures in common or if the isr 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 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.

```
prq = queue_create();
```

To add an the tuple (val, arg, callback) to the queue do this (note that *arg* isn't needed until the last question, so you an use NULL for this parameter in the meantime).

```
queue_enqueue (prq, val, arg, callback);
```

To get a tuple from the queue, declare variables to store the results and then pass them by reference to dequeue:

```
void *val, *arg;
void (*callback) (void*,void*);
queue_dequeue (prq, &val, &arg, &callback);
```

Or if you aren't using `arg`:

```
void *val;
void (*callback) (void*,void*);
queue_dequeue (prq, &val, NULL, &callback);
```

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
```

You might want to add some `printf`'s to get a better idea of what is happening. Be sure that you understand what `is_read_pending` is and how it is used. Before moving to the next step be sure that you remove any `printf`'s that you added.

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

Step 1

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 service routine when each read completes.

The `interrupt_service_routine` and some other code is provided. You must implement the `handle_read` procedure and the code that schedules the reads in `main`. Note that there is an infinite loop at the end of `main`. This is needed because the program will terminate when main returns even if there are some reads that haven't completed. And, of course, NONE of them will have completed since all that will have happened by the time you reach the end of `main` is that you will have *scheduled* all of the reads; none of them will have completed yet. The program will terminate when `exit()` is called, as you can see in `handle_read_and_exit`.

Compile and test your program and be sure that it produces the same output as `sRead`.

Step 2

To get a better understanding of how the interrupt interleaves with your program, run your programming in the debugger and set a breakpoint in `interrupt_service_routine`. When it stops, print out a stack trace. You will see something like this, for example.

```
* frame #0: 0x00000001000007db aRead`interrupt_service_routine at aRead.c:23
  frame #1: 0x0000000100000d58 aRead`deliverInterrupt at disk.c:105
  frame #2: 0x0000000100000e1a aRead`handleTimerInterrupt(signo=14, info=0x00007ffeefbff620,
                               uap=0x00007ffeefbff688) at disk.c:118
  frame #3: 0x00007fff76328f5a libsystem_platform.dylib`_sigtramp + 26
  frame #4: 0x0000000100000a00 aRead`main(argc=2, argv=0x00007ffeefbff7d8) at aRead.c:65
  frame #5: 0x00007fff760a7115 libdyld.dylib`start + 1
  frame #6: 0x00007fff760a7115 libdyld.dylib`start + 1
```

Look at activation frame #4 in this example (your stack trace will be a bit different). It shows the program running at `aRead.c` at line 65, which is the infinite while loop at the end of `main`. Frame #3 is the "interrupt" (actually a timer signal delivered by the operating system to `disk.c`, which is how we simulate interrupts). Frames #2 and #1 are code in the simulated disk that handle this interrupt and translate it into a call your isr, which is the top frame, stopped at the breakpoint.

Step 3

Now, 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. 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.

This time you must implement the `interrupt_service_routine` yourself.

Compile and test your program.

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`.

Question 4: More Fun with Asynchronous Programming

In this final step you will implement a program that goes on a treasure hunt through the disk. The program's command-line argument is the block number where the hunt will start. In the first step of the hunt, you must read the value of this block. This value will tell you two things: (1) the number of additional steps to perform and (2) the address of the next block to read. In each subsequent step the block value that is read gives you the block number of the next block to read. Print out the value of the last block you read.

For example if you type this

```
./treasureHunt 21
```

You should read block 21 first. Lets assume that the value of block 21 is 7. If so, you should read 7 more blocks, starting with block number 7. If the value of block 7 is 12, then that is the third block you read, an so on. Note that in total you will have read 8 blocks.

The code pack includes the starter file for `treasureHunt.c`. Work incrementally. You can use `aRead` as a guide to get started. A good first step would be to read the value of the initial block and print it. Then a good second step would be to read the second block and print its value. Now think about adding the counter argument to the callback (that's what `arg` is for) and counting down. Finally, be sure that you only print the value of the last block and that your program terminates when its done.

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`.
6. For Question 4: `treasureHunt.c`.