COMP3430 **Assignment 2** Due: March 2nd, 2018 @11:59PM

Assignment Objectives:

- *To learn some more about threads and synchronization.*
- *To gain some experience in identifying the synchronization requirements in an unfamiliar concurrency problem and use standard synchronization primitives to solve the problem.*
- *To get a taste of pthreads programming with synchronization using a relatively simple, multi-threaded application running on a multicore computer system.*
- *To introduce you to a new synchronization primitive from a different language.*

Assignment Instructions:

- *You must hand in your assignment electronically via UMLearn using the "Assignments" functionality under the "Assessments" drop-down menu. A folder for Assignment 2 has been created there. You can hand in multiple files if so desired but we would ask that you be kind to the markers and hand in everything in one batch at the end if at all possible. Recall that you must agree to the online honesty document before the dropbox will become visible. You can read back in the announcements on UMLearn for details on how to do this though you should have already done this to hand in you first lab.*
- *In general, for written questions PDF (.pdf), Word (.docx), PowerPoint (.pptx), Rich Text Format (.rtf) and Text (.txt) format files are acceptable. Handwritten then scanned PDF files are **not** acceptable. For any programming questions, please hand in all **source** files and any requested results as well as a Makefile to generate the executable file(s) for the target platform. You **must** also include a text file named 'README' for each programming question that briefly describes how to use your program.*
- *Your programs must run on the Linux lab machines, but you can try to develop your code on other platforms if you like though this is **not** recommended. In some cases, this may simply not be possible since your running programs will be making Linux system calls. Should you choose to use different platforms for development, you **alone** will be responsible for dealing with any differences in the compilers, tools, etc. that you may encounter. In such a case, be sure to leave plenty of time to port your code to Linux. Normally this takes less than 30 minutes but sometimes it may take several hours if your code is still a little buggy. (For example, the C compiler on some systems sets all uninitialized memory to zeroes. Hence, bad pointers are treated as NULLs and thus, sometimes incorrect code works there but not with other compilers and operating systems such as gcc on Linux.)*
- *Note that assignments are to be done **entirely independently** unless otherwise explicitly stated and that inclusion of materials from online sites is entirely forbidden.*

Assignment Questions:

[3] 1. This question builds on your prior experience with the concurrent linked list you coded in Lab 2. Having threads access entirely **in**dependent parts of a linked list **concurrently** is clearly safe (since there is no concurrent access to any given list node). Briefly discuss why implementing synchronization using this ability based on pthreads mutex variables would likely be difficult. (Hint: think about what each mutex variable would have to be guarding, and thus associated with, and how such mutexes would have to be used when doing **insert** and **delete** operations.)

[4] 2. Recall that pthreads implementations may be either user-level or kernel-level. In the latter case, the OS kernel is aware that the threads exist and can deal with them directly. Based on the material in the notes comparing the behaviour of kernel and user threads, write a **short** pthreads program (just 2 threads should do) that demonstrates whether or not the threads running in the pthreads implementation you are using in the Lab are visible to the kernel. Submit your code, the output from a run of your program (in a file named q2output.txt) and a brief justification of why your program output correctly determines the thread type used.

[12] 3. Suppose you have a program where one thread collects data from an experimental instrument as it operates (e.g. from a detector in a particle accelerator) producing one new file each minute. Another thread pre-processes those files to produce new ones in a form that is more convenient for later processing (e.g. recognizing and filtering noise from the sensor data). A final thread uploads the pre-processed files to a web site where they can later be accessed by scientists around the world who will download and mine the data to discover any scientifically relevant patterns. All three threads must execute concurrently. Obviously, we cannot pre-process a file that does not yet exist, nor can we upload one that has not yet been pre-processed. In general, the speed of the three processes is comparable but sometimes processes may be delayed due to other activity on the machine(s) on which they are running. Using your knowledge of synchronization from the lectures, you are to design a synchronization solution for this problem and then provide **pseudo**-code that synchronizes the concurrent operation of the three processes using (a) semaphores (with types of your choosing) and, separately, (b) monitors with whatever data structures you decide to use. You are free to expand on and re-use techniques discussed in the lectures. How might an implementation of your solution change if one of the latter two processes was periodically delayed for a longer time? Why do you think it is necessary to deal with this issue? (Hint: think about what happens to the first process.)

[22] 4. Using pthreads, you are to write a C program that will implement the key functionality of a multi-threaded print manager. Your program should accept two integer command line arguments: `NumPrintClients` and `NumPrinters` (via argv[]). It should begin by first creating `NumPrinters` instances of a `PrintServer` thread and then creating `NumPrintClients` instances of a `PrintClient` thread (both described below). Your main thread should then simply exit (remembering to dutifully call `pthread_exit()` before doing so). The `PrintClient` threads (that play the role of applications requesting printing) and the `PrintServer` threads (that play the role of the OS code managing each printer) should communicate with each other using a bounded buffer containing up to 4 outstanding `PrintRequest` entries. Each `PrintRequest` structure contains three things: a `long` client id, a `char *` (i.e. string) file name, and an `int` file size. A suitable C language struct definition for a `PrintRequest` is the following:

```
typedef struct PR {
   long   clientID;
   char   *fileName; //ptr to a dynamically alloc'd string
   int    fileSize;
} PrintRequest,*PrintRequestPTR;
```

A declaration for the bounded buffer itself would then be the following:

```
PrintRequest BoundedBuffer[4];
```

You will, of course, also need to add some variables to keep track of the head and tail in the buffer and maybe the number of elements it currently contains. (It's time to dig out your notes from your data structures class to refresh your memory on queue implementation.) It might make sense to group these variables and the buffer into a `struct` as well but the choice is yours to make.

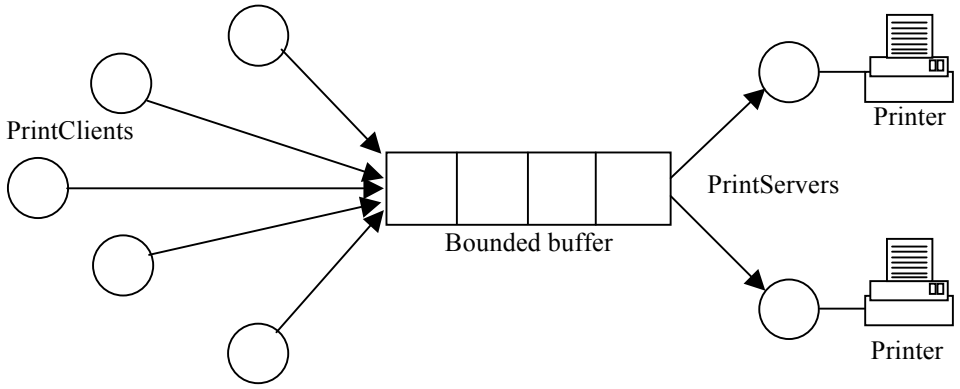
Your `PrintClient` code should simply consist of a loop that runs six (6) times. In each iteration of the loop your code should insert a new `PrintRequest` entry into the bounded buffer. Each entry should contain the client's pthread-id (i.e. obtained by calling `pthread_self()`), a "filename" consisting of the string '`File`' with the client id and loop iteration appended to it separated using hyphens (e.g. the filename would be `File-24876529-5` if the thread had id 24876529 and was executing in the 5th iteration of its loop), and a random number between 500 and 40,000 representing the "size" (in bytes) of the corresponding file. Your client should print a message identifying itself (using its thread id) together with the relevant filename immediately after it inserts an entry into the bounded buffer. At the end of each iteration, your `PrintClient` code should sleep for a random number of seconds (between 0 and 3). See `man 3 rand` and `man 3 sleep` for information on how to do this.

Your `PrintServer` code should run an infinite loop attempting to remove entries from the bounded buffer. When an entry is removed, the thread code should print a message specifying its own id (again via `pthread_self()`) together with all the information from the `PrintRequest` and the current time (using the `time` and `ctime` functions described in `<time.h>`and described in the *nix man pages, sections 2 and 3, respectively) indicating that it is starting to "print" the job. It should then sleep for a number of seconds determined by the size of the "file" to be printed. You should assume that printers print 4000 bytes per second. When it awakens, your thread code should print a message containing the same information (with updated time) indicating it has completed the print job.

Your code should include separate routines called by the `PrintClient` and `PrintServer` threads (in the obvious way) to insert into and remove from the bounded buffer. These routines should manipulate a mutex variable to ensure mutually exclusive access to the buffer as well as two condition variables used to allow threads attempting to insert into a full buffer or remove from an empty one to wait and be signaled when the condition they are waiting on has been satisfied. (i.e. You will be doing a pseudo-monitor implementation of the producer consumer problem described in class.)

Test your program with five (5) print clients and two (2) "printers". Note that since your PrintServer threads are running infinite loops, you will need to use CTRL-C to terminate your program once it is finished.

The following diagram illustrates the high-level structure of the system:



[4] 5. Most modern languages provide support for threading and for a variety of different synchronization primitives to be used by the threads they provide. Python is no exception and, in fact, provides a plethora of such features. Do some Internet research on Python's `RLock` objects. Briefly describe what they are, what operations they support and give an example of where they might be more useful than Python's basic `Lock` objects.

Total: 45 marks

Want another Challenge?

No marks for this, just as there were no marks for the challenge problem in Assignment 1, but an interesting opportunity to explore data sharing **without** the use of Threads. You are to re-implement Question 4 (the print server) using *nix processes (instead of pthreads) and a Posix message queue (see man 7 mq_overview on the lab machines as a starting point) for connecting the print clients and servers. As in Assignment 1, solving this challenge problem will require a fair bit of "Googling". Also, be aware that there are at least two different kinds of message queues out there. Don't get information on "SysV" queues and "Posix" queues confused. No marks for this challenge so, obviously, once again, no handin is required.