COMP3430 **Assignment 3** Due: March 21st, 2018 @11:59PM

Assignment Objectives:

- *To reinforce your understanding of some basic scheduling techniques.*
- *To introduce you to the sharing of memory between Unix **processes**.*
- *To introduce you to the use of Unix signals.*

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 3 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:

[12] 1. Consider a system with 5 processes as described in the following table.

Process	**Arrival Time**	**Duration**	**Priority**
P_1	1	5	3
P_2	3	6	1
P_3	5	5	4
P_4	5	7	3
P_5	7	6	2

Illustrate the execution of the processes under the PRIORITY scheduling algorithm (assuming 4 is high priority) for time slices of 2 and 3 time units and then again under the round robin scheduling algorithm (where priorities are irrelevant), also for 2 and 3 time units. Use GANTT charts (plots of activity by processes vs. time) to show when the various processes execute. You will have a total of four GANTT charts, one for each combination of algorithm and time slice size. Be sure to factor in arrival time in your work.

[22] 2. Using Linux **processes** (not p-threads), SysV shared memory segments (as in the last lab) and SysV semaphores, you are to write a C program that will re-implement the key functionality of the multi-threaded print manager from assignment 2. You will have to change your code to work with processes, shared memory segments and semaphores instead of threads with mutexes and condition variables. To simplify the problem, you need only support a **single** `PrintClient` and `PrintServer`. A modified description of the problem from the last assignment follows:

Your program should begin by first creating **single** instances of a `PrintServer` **process** and a `PrintClient` **process**. The `PrintClient` **process** (which plays the role of an application requesting printing) and the `PrintServer` **process** (which plays the role of the OS code managing a printer) should communicate with each other using a bounded buffer containing up to **4** outstanding `PrintRequest` entries **stored in a shared memory segment (created using your student number as a key)**. A `PrintRequest` structure contains three things: a `long integer` client id, a string file name, and an `integer` file size. A suitable C language struct definition for a `PrintRequest` is the following:

```
typedef struct PR {
   long   clientID;
   char   filename[64]; //ptr unusable in shared memory
   int    fileSize;
} PrintRequest;
```

A declaration for the bounded buffer **to be stored in the shared segment** would then be:

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

Note that you will also need to store the various queue-related variables (indexes to start and end elements, count of elements in the queue) **in the shared segment**.

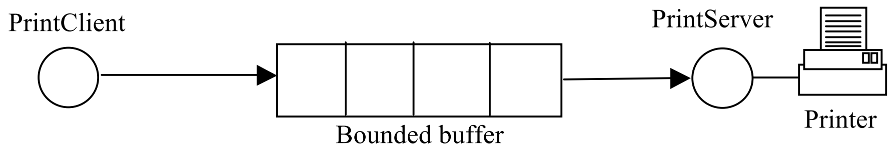
Your `PrintClient` code should simply consist of a loop that runs **six** (**6**) times. In each iteration of the loop your code should insert a new `PrintRequest` entry into the bounded buffer. Each entry should contain the client's **process id** (obtained by calling **`getpid()`**), a "filename" consisting of the string '`File`' with the client's **process** id and loop iteration appended to it separated with underscores (e.g. the filename would be `File_24876529_3` if the **process** had id 24876529 and was executing in the 3rd iteration of its loop), and a random number between 500 and 40,000 representing the "size" (in characters) of the corresponding file. Your client should print a message identifying itself (using its **process** id) together with the relevant filename immediately after it inserts an entry into the bounded buffer. At the end of each iteration, your code should sleep for a random number of seconds (between 0 and 3).

Your `PrintServer` code should run an infinite loop attempting to remove entries from the bounded buffer. When an entry is successfully removed, the code should print a message specifying its own id (via **getpid()** again) together with all the information from the PrintRequest 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 can print 4000 characters per second. When it awakens, your **process** code should print a message containing the same information indicating that it has completed the print job.

Your code should include separate routines called by the PrintClient and PrintServer **processes** (in the obvious way) to insert into and remove from the bounded buffer. These routines should manipulate a **semaphore** to ensure mutually exclusive access to the buffer as well as two other **semaphores** used to allow a **process** 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.

Note that since your PrintServer **process** is running an infinite loop, you will need to use CTRL-C to terminate your program once it is finished.

The following diagram illustrates the modified high-level architecture of the system:



An example of the creation of shared memory segments was provided to you in the code used in Lab 3. You will need to consult the Linux man pages to learn about SysV semaphores. Be careful since there are three different semaphore implementations provided on Linux and you don't want to mix calls from the three systems. See man 3p sem_open (and also sem_close, sem_unlink, sem_post, sem_wait) for the relevant information. You will need to link with the "real-time" library (specifying -lrt to gcc) when compiling your code.

[6] 3. Having servers run infinite loops means that they must be terminated externally (e.g. by having a human type CTRL-C). Using CTRL-C, however, is not a reasonable approach since there will be many servers typically started by the OS at boot time that will need to be terminated when the OS later shuts down and requiring human intervention at this point is undesirable. To support the ability to shutdown servers under **program** control, *nix provides the kill command (as seen in Lab 3) and its system call equivalent kill(). The kill() system call does not just kill a process (though it is commonly used for that purpose). In general, it sends a "signal" to the process, one of which always results in killing the process. Most signals, however, can be "caught" and "handled" by the process (in a fashion similar to interrupts and interrupt service routines) as discussed in class. Thus, any process can set up a handler routine that will be invoked when a particular signal is received that will result in the corresponding handler code being executed. To solve the problem with terminating servers we can send the server a "Please Shutdown" signal which it can catch and then shut itself down cleanly via the corresponding handler code. You are to add a handler routine to your PrintServer process from question 2 that will catch a SIGUSR1 signal and then print a message stating that the server is shutting down and clean up the running server (e.g. detaching the shared segment) before exiting, via exit(0), the server. You are also to write a program shutdown.c that, when invoked by the user, will send a SIGUSR1 signal to the server. To be able to use the kill system call to send the signal, you must know the process id

of the process to be signaled. Your program should get this from the command line using `atoi(argv[1])`. The `signal()` and `kill()` system calls are documented in section 3 of the man pages (so can be researched using `man 3 signal` and `man 3 kill`, respectively). The hardest part of dealing with this facility for most students is in the requirement to use function pointers to specify the handler to be executed when a signal is received. To help you with this, some simple example code for `signal()` and `kill()` as well as a few slides describing Unix signals from an old course are available along with the assignment. Note that you will need to include the header files related to use of signals as in:

```
#include <signal.h>
#include <sys/types.h>
```

Total: 40 marks

No Challenge Questions this time – you are better off focusing on core material in this and other courses you are taking!