C502: Operating Systems
Assessed Coursework

Due date: **11th December 2015**
(Submission on CATe by 17:59)

The purpose of this exercise is to use the producer-consumer problem to gain experience into using system calls for semaphores and shared memory. In this exercise, the producer-consumer problem is modelled using a shared circular queue, which has the following structure:

```
typedef struct jobtype
{
  int id;
  int duration;
} JOBTYPE;
typedef struct queue
{
  int size;
  int front;
  int end;
  JOBTYPE job[500]; // Can assume this to be maximum queue size
} QUEUE;
```

The sequence of steps for the producer and consumer are provided below:

1. Producer
   (a) Read in two command line arguments - id of producer, number of jobs to generate.
   (b) Associate with the shared memory segment, if available.
   (c) Add the required number of jobs to the circular queue, with each job being added once every 2 to 4 seconds. The job id of the created job is one plus the location that they occupy in the circular queue. If a job is taken (and deleted) by the consumer, then another job can be produced which has the same id. Duration for each job should be in the range 2 to 7 seconds. If circular queue is full, block while waiting for an empty slot.
   (d) Keep track of the time from the beginning (starting at time = 0).
   (e) Print the status (format given in *example_output1.txt* and *example_output2.txt*).
   (f) Quit when there are no more jobs left to produce.
2. Consumer
   (a) Take in one command line argument - id of consumer.
   (b) Associate with the shared memory segment, if available.
   (c) Take a job from the circular queue and 'sleep' for the duration specified. If circular queue is empty, block while waiting for jobs and quit if no jobs arrive within *10 seconds*.
   (d) Keep track of the time from the beginning (starting at time = 0).
   (e) Print the status (format given in *example_output1.txt* and *example_output2.txt*).
   (f) If there are no jobs left to consume, wait for *10 seconds* to check if any new jobs are added, and if not, quit. Before quitting, ensure that the shared memory and/or any semaphores used are appropriately deleted.

In order to ensure there are no ‘race conditions’ with respect to creating the semaphores and shared memory, you can create and set these up (along with other parameters such as queue size) in a different program (called *start.cc*) and run that first before running the producer and consumer programs. Also, there is an undesirable side-effect of using System V semaphores, in that the semaphore values incremented/decremented within a process are automatically undone when that process terminates. An easy option to ensure this doesn’t happen is to make the producers/consumers that have finished executing their main code sleep for a long time (so that their semaphore values don’t get reset), while other producers/consumers are still running. You are however welcome to come up with more challenging/-complicated/interesting solutions ☺.

Your solution must be able to handle multiple producers and consumers concurrently and use semaphores to restrict access to the shared resource (in this case, the circular queue). The solution should also deal with input errors, system call errors, and any other errors, appropriately. Useful command-line tools for debugging by checking the status of semaphores and shared memory are *ipcs* and *ipcrm*.

You should hand in the following on CATe:

1. An archive (**code.tar.gz**) containing **only** the C++ files (*start.cc, producer.cc, consumer.cc, helper.cc, helper.h*), the *Makefile*, and any *scripts* used for running and testing your programs.

Notes: Download outline source code and scripts from CATe. You are not obliged to use these; however they should simplify the process of achieving working solutions. You can choose to change these files as necessary. A few notes on the layout and support files follow:

- These support files have been tested on Linux in the DoC labs. You can however choose to use your own programming environment.
- The file *helper.h* includes the required header files, defines the structures used, as well as some of the pre-defined variables. Please ensure that you change the value of the key used for the semaphore and shared memory in-order for it to be unique amongst students.
- The file *helper.cc* contains a few helper functions, including functions for creating, using and destroying semaphores.
- The files *run1.sh* and *run2.sh* contain example scripts that allow you to test your program. *run1.sh* has one producer and one consumer, while *run2.sh* has three producers and two consumers. Corresponding example outputs from running the above scripts are given in *example_output1.txt* and *example_output2.txt*. You are expected to write your own tests beyond this.
- The *Makefile* allows users to use the `make` command on Linux to compile the various parts of the exercise. It can also be used to help configure your preferred development environment with the correct commands, flags and parameters.

Tutorials

Please be aware that the basic functionalities of these may slightly differ from the lectures.

Semaphores: http://beej.us/guide/bgipc/output/html/multipage/semaphores.html
Shared Memory: http://beej.us/guide/bgipc/output/html/multipage/shm.html

Useful Links

Workstations in DoC: https://www.doc.ic.ac.uk/csg/facilities/lab/workstations
Remote Login in DoC: https://www.doc.ic.ac.uk/csg/services/linux
Creating and Unpacking .tar.gz files: http://arxiv.org/help/tar
ipcs - provide information on ipc facilities: http://linux.die.net/man/1/ipcs
ipcrm - remove a message queue, semaphore set or shared memory id:
http://linux.die.net/man/1/ipcrm