McGill

Programming Assignment #2: Printer Spooler

Due date: Check My Courses`

1. Why this assignment?

Synchronization is an important part of modern computer applications that have multiple threads or processes executing within them. For example, an Internet browser such as Chrome has many processes executing within it. Most operating systems and programming languages provide some primitives for synchronization. Java, for instance, provides several high-level constructs for synchronization. C, on the other hand, does not provide any constructs by itself. However, there are various library-based or OS-based solutions for synchronization that can be invoked from a C program.

In this assignment, you are going to develop a printer spooler application. You will create this application with multiple processes. The processes should be independently launchable from the shell. You will use shared memory for creating the inter-process communication. Because processes do not share memory, you will create shared memory among the processes and setup the appropriate synchronization to do the printer spooler simulation.

Pthreads or any other threading library should NOT be used as part of this assignment.

2. What is required as part of this assignment?

As part of this assignment, you are expected to implement a printer spooler (server) application. We have one process that mimics the printer server. It has a bounded buffer (a buffer of fixed size), the clients insert the jobs to print into this buffer. The printer server takes a job from the buffer (it could treat the buffer as a FIFO queue) and prints it. Each job has a duration (time taken to print) as a defining parameter.

The print server is a process that runs forever. It does the following:

```
setup_shared_mem();          // create a shared memory segment
attach_shared_mem();         // attach the shared memory segment
init_semaphore();            // initialize the semaphore and put it in
                                shared memory
while (true) {
      take_a_job(&job);      // this is blocking on a semaphore if no job
      print_a_msg(&job);     // duration of job, job ID, source of
                                job are printed
      go_sleep(&job);        // sleep for job duration
}
```

The job itself is in a buffer (array) that is placed in a shared memory. The take_a_job() function is like the consumer function in the producer-consumer problem we discussed in the class. You can see the slides on you could implement it. It is important to note that we assumed the existence of shared memory in the class. However, in this assignment you don't have shared memory. So you need to explicitly setup shared memory.

The print server creates a shared memory segment and places the buffer and semaphore there. The shared memory setup is split between the first two functions shown in the above pseudo code.

The print server does not generate the jobs that get put into the job queue. Those jobs actually come from the print clients. A print client looks like the following:

```
attach_share_mem();      // use the same key as the server so that the
                            client can connect to the same memory segment
place_params();          // this is to place the parameters in the shared
                            memory – in particular the job queue
                            and semaphore
get_job_params();        // read the terminal and get the job params
job = create_job();      // create the job record
put_a_job(&job);         // put the job record into the shared buffer
release_share_mem();     // release the shared memory
```

The print client attaches to the shared memory segment that was generated by the print server. To access the same segment it needs to use the same key value. One the shared memory segment is attached, you place the queue and semaphore in the shared memory. Remember the server already created them, you need to access them by using a pointer casting. The suggested approach is to put the buffer and semaphore and any other shared variable you want into a C struct and place it there in the server and then access it from the client using pointer casting.

You get a handle to the buffer and semaphore and use in the put_a_job() function. See the slides discussing the producer-consumer problem. This is the producer routine there. You are just producing one item unlike the infinite number that was produced in that example. Also, you need to find out how to handle the semaphores in Linux.

If there is no client running to put jobs, the server should wait. It will be waiting for jobs to arrive in the buffer. You won't see any messages from the server. When you run a client, it will create a job and put it in the job queue and terminate assuming it finds a buffer space to put the job. If does not find the buffer space, the client will wait until the buffer space becomes free. Remember the buffer was created with finite number of slots and a slot it taken by each job while it is in the queue.

You start the print server. It should request some arguments such as the number of slots in the shared memory. If you find that other parameter values are needed to create a print server (for the purposes of this assignment), you can read them as well.

You start the client with may be with the client ID, job duration, etc. as arguments. You could pass the arguments to the client and server as command line parameters (*this is much better approach than reading it from the terminal*). Reading from the terminal is still acceptable.

You should not use sockets or any other mechanism besides shared memory for communicating information between the client and server. Through the shared memory the client (could have many clients at a given time) and server are manipulating shared variables and semaphore values. The shared variables need protection using locks for mutual exclusion. You could implement the locks using binary semaphores.

Your program could provide an event trace something like the following to show how it is operating.

```
---
Client 2 has 6 pages to print, puts request in Buffer[2]
Client 5 has 3 pages to print, puts request in Buffer[0]
```

```
Printer 0 starts printing 6 pages from Buffer[1]
Client 1 has 8 pages to print, puts request in Buffer[1]
Client 3 has 5 pages to print, puts request in Buffer[2]
Client 4 has 2 pages to print, buffer full, sleeps
Printer 1 starts   printing 3 pages from Buffer[0]
Client 4 wakes up, puts request in Buffer[0]
Printer 0 finishes printing 6 pages from Buffer[2]
Printer 0 starts   printing 8 pages from Buffer[1]
Printer 1 finishes printing 3 pages from Buffer[0]
Printer 1 starts   printing 5 pages from Buffer[2]
Printer 0 finishes printing 8 pages from Buffer[1]
Printer 0 starts   printing 2 pages from Buffer[0]
Printer 1 finishes printing 5 pages from Buffer[2]
No request in buffer, Printer 1 sleeps
Printer 0 finishes printing 2 pages from Buffer[0]
No request in buffer, Printer 0 sleeps
---
```

You need to ensure your programs (client and server) are implementing the following synchronization requirements.

- Clients should check for buffer overflow conditions. If a buffer overflow condition is detected, the client that is trying to enqueue a print job should wait. The wait should be implemented by a shared semaphore variable.
- Printers should check for buffer underflow conditions. If a buffer underflow condition is detected, the printer daemon should wait. Similar to the client, the printer daemon should wait on a semaphore variable.
- Manipulation of shared variables such as the buffer should protected using a semaphore.

3. What should be handed in?

1. You should submit the C program
2. A trace of your program running for example input values
3. A brief documentation if your program is incomplete that shows what you have implemented and tested in your submission. If nothing works, you need to outline your design and a path towards complete implementation – partial credit will be assigned in this case.