McGill

Programming Assignment #2: Printer Spooler

Due date: Check My Courses`

This assignment should be completed individually. You can collaborate in the design phase but the implementation should be an individual effort.

This assignment should be completed in a Linux machine. If you don't have Linux running on your personal machine, please use the ENGTR 3rd Floor Linux Lab machines.

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. You should be able to launch the processes independently from the shell. You will use shared memory for carrying out the inter-process communications needed for the cooperative activity (i.e., printer spooler simulation). 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. Lets call this application the **`print_server`**. It is responsible for creating and maintaining a bounded buffer and associated semaphores in a shared memory area. When a job is available in the buffer, which is treated as a FIFO, the server wakes up and "prints" the job. Otherwise, the print_server is sleeping waiting for a job. That is, you need to implement the underflow synchronization we discussed in the class and make the print_server go to sleep when there are no jobs in the buffer (queue). Also, the buffer has finite capacity. Therefore, we need to implement the overflow synchronization, which makes the print_client go to sleep until there is space in the buffer. Each job has a duration. For that amount of time, the print_server is "busy" (i.e., does not do any activity to simulate actual printing). So it is necessary to generate the durations in seconds to make the simulation complete in a reasonable amount of time.

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
```

inside initialize_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, and source of
                               job are printed
      go_sleep(&job);       // sleep for job duration
}
```

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 to get an idea for implementing it. In the class, we assumed that everything was in a shared memory area. In this assignment, you are explicitly setting up the shared memory to realize that assumption. In addition to the buffer, the **print_server** can put other elements such as the semaphores in the shared memory. The shared memory setup is split between the first two functions shown in the above pseudo code. You don't need follow the pseudo code strictly. It is a guideline to structure your code. Anything better (well structured) than what is shown in the pseudo code is completely acceptable!

The print server does not generate the jobs that are 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
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 created by the print server by using the same key that the print_server used while creating the segment. Once the shared memory segment is attached, you can access the queue and semaphore in the shared memory. Remember the server already created them, you need to access them by using pointer casting. You can modify them. That is, you can add jobs to the queue from the client side. The suggested approach is to put the buffer and semaphore and any other shared variable you want into a C struct and place it in the shared memory using the server and then access it from the client using pointer casting.

Once you get an handle to the shared memory, you can use 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.

When there are no jobs in the buffer, the print server waits. There will be no message from the print server. When a client is run, it will create a job and put it in the queue. If the client is unable to put the job in the queue because there is no space, the client will wait for space. That is the program hangs waiting for space to show up. May be the printer is offline and that makes the printer queue to build up and reach its limit.

When the print_server starts, it should request for some arguments like the number of slots from the user. The user may pass additional parameters if your design require them.

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. The print_server should maintain a persistent counter variable that gives the print job ID. Your printer could crash, but the job ID counter does not lose its value.

You start the print_client with at least one argument: job duration (in seconds). Again, your design could mandate other additional parameters.

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 clients (could have many clients at a given time) and servers are manipulating shared variables and semaphore values. The shared variables need protection using locks for mutual exclusion.

Your program could provide an event trace something like the following (actual trace of your program could differ from the one shown) to show how it is operating.

```
..
---
Client 2 has 6 pages to print, puts request in Buffer
Client 5 has 3 pages to print, puts request in Buffer
Printer 0 finishes printing 8 pages
Printer 0 starts printing 6 pages
Client 1 has 8 pages to print, puts request in Buffer
Client 3 has 5 pages to print, puts request in Buffer
Printer 1 finishes printing 12 pages
Printer 1 starts printing 3 pages
---
..
No request in buffer, Printer 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.

Your program should allow multiple clients and servers. That is, there is a single printer queue. Multiple clients are putting jobs into it. Multiple printers (print_servers) are taking jobs out of it. Some print_servers could go offline and come back. You don't need to handle the situation where a print_server takes a jobs and crashes. When a job is taken out of the queue, you assume it is done after the given

duration. In the case the print_server crashes and is restarted, it could go ahead and take the next job. So effectively the previous job got tossed in no time. We don't worry about this problem!

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.

4. Grading scheme

1. Basic print server and client implemented: 60 points
2. Basic configuration tested to work correctly (synchronizations handled properly): 75 points
3. Multiple servers (synchronization working with multiple servers): 90 points
4. Well documented and structured code: 100 points