McGill

Programming Assignment #2: Printer Spooler

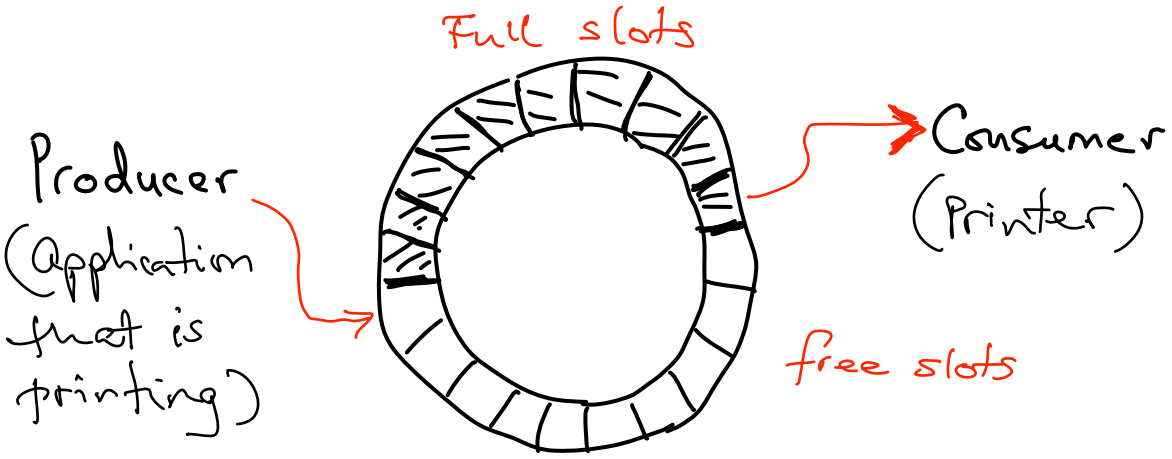
Due date: Check My Courses`

1. Why this assignment?

Synchronization is an important part of modern computer applications that have multiple threads of execution within them. For example, an Internet browser tool such as Mozilla Firefox would have multiple threads to manage various user and system tasks simultaneously. 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 learn how to build high-level constructs using some basic primitives supported by Pthreads/UNIX within C.

2. What is required as part of this assignment?

As part of this assignment, you are expected to solve a simple bounded buffer problem. In this problem, we will use a *simulated* print spooler as example. There are *C* number of clients, each independently requesting to print a certain number of pages. The spooler has *P* number of printers to serve these concurrent requests. The requests are served in a FCFS manner.



Obviously, the requests should be queued in an FCFS buffer of a finite size *B,* in the spooler. You need to implement the buffer as a circular array as shown in the figure, for better use of memory. In this *bounded-buffer* or *producer-consumer* problem, you can think of the clients as request producers and the printers as consumers.

Design and implement a C/Linux program to simulate the above print spooling scenario using the threading and synchronization primitives of Pthreads library. Each printer is simulated by a thread. The printing time will be simulated by **`sleep(n)`** function, where *n* is the number of pages to print. A thread simulates each client as well. A client keeps generating requests at a predefined interval T_C(=5sec, say). Each request consists of *n* number of pages, where *n* is a random number between 1 and 10.

You should write the codes for both printer and client threads. Any client, if it finds the spooler buffer full, sleeps until space becomes available in the buffer (space in the buffer is measured in slots –

can hold a request of any number of pages). If a printer finds that there is nothing more to print, it goes to sleep. The printer should be woken up when something becomes available in the print spooler. Similarly, a sleeping client should be signaled if space becomes available in a previously full buffer.

Your simulator program should take *C* (number of clients), *P* (number of printers) and *B* (buffer size) as command line parameter or user input. Test your program with different combination of the values for *C, P* and *B* (obviously *C,P,B* > 0). As output your simulator should print the event trace of the spooler. Each of the clients (as well as the printers) should have an id (say 0 to *C* or 0 to *P*) to identify them in the trace. Buffer slots are also numbered from 0 to *B*. A sample snapshot of the event-trace output might look like the following –

```
---
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
---
```

Implement your simulator in C using Pthread primitives. While designing a solution for this problem you should adhere to the following requirement.

- 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 using a semaphore. You **should not busy wait the client** until a buffer slot becomes available. *Optionally, you could use a condition variable as well. The pthread library supports condition variables – although we did not cover those in the class, yet.*
- 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 condition variable.
- Manipulation of shared variables such as the circular buffer should be protected using a semaphore.

To complete this assignment, you need to review the pthread lifecycle management functions discussed in the class and tutorials. In addition, you need to understand the synchronization primitives that

ECSE427/COMP310 Page 2 of 3

are covered in the class. Using semaphores from the synchronization primitives is one of the ways of implementing this assignment. The simple C program shown below illustrates how semaphores can be used. You can compile the program using gcc filename.c –lpthread.

```
#include <stdio.h>
#include <semaphore.h>

sem_t sem;
int count = 1;
int pcount = 1;

void main()
{
  sem_init(&sem, 0, 2);

  printf("Wait semaphore %d \n", count++);
  sem_wait(&sem);
  printf("Wait semaphore %d \n", count++);
  sem_wait(&sem);


  printf("Post semaphore %d \n", pcount++);
  sem_post(&sem);

  printf("Wait semaphore %d \n", count++);
  sem_wait(&sem);
}
```

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.

ECSE427/COMP310 Page 3 of 3