

## CMSC 312 Assignment 3: Multi-process and multi-threaded print server

Project due date: 11:59 pm EST, 4/8/20

### What is a print server?

- Consider a single globally shared printer queue that all processes and threads can read from or write into. The print queue has a finite size of 30 print jobs.
- Producer-Consumer problem: user processes versus consumer threads.
- The user processes act as producers and add print requests into the queue. Each print request is characterized by its size (in bytes). Each user can submit up to 30 print jobs; use a random number generator to determine the number of jobs for each user. For each print job, use another random number generator to find the size (in bytes) of that print request (between 100-1000 bytes).
- The different printers serve as consumers and run on different threads. Each such printer can process a print job and removes it from the global print queue.
- The main process will read the command line to determine the parameters, start the producer processes and consumer threads, and acts like an interactive command console. It also initializes the print queue but cannot modify it after initialization.
- The main process will read the command line to determine the parameters, start the producer processes and consumer threads, and acts like an interactive command console.
- The command line parameters include: (i) number of producer (user) processes, and (ii) number of printer (consumer) threads. Use `nanosleep()` or `sleep()` utilities to simulate a random delay (between 0.1 – 1sec) between two successive print jobs submitted by the same user. The user process inserts the print request into the global print queue; additional book-keeping parameters are allowed to be inserted. Hence, use a struct for the producer print request and the global queue will be an array or linked list of structs; similarly, use a struct for the consumer to process a print request. The producer processes cannot remove anything or update a currently existing print request from the queue.
- Each of the printer threads will process one print request at a time depending on their availability for as long as the queue is not empty. You need three semaphores (i) to check if the queue is full, (ii) to check if the queue is empty and (iii) for reading/writing the same queue element. You need to initialize these semaphores before starting the processes/threads. Since the semaphores need to be shared between the processes and threads, set them up on shared memory. Use your implementation of counting semaphores from Assignment-2 for the counting semaphores needed in this exercise.
- The main process needs to figure out when all print requests have completed and then deallocate the global print queue; also deallocate the semaphores. The printer threads can only exit after all print requests have completed. You may have to use messaging between threads to signal them to complete or this can be simply handled by semaphores.
- Print out the results after all threads have completed. Specifically, we want how many print jobs and of what size each were sent to the print server by the user processes. Then for each printer (i.e., consumer thread), we want to check how many print jobs (and bytes) were actually processed.
- Use a signal handler in your code to ensure graceful termination of the threads on receiving a ^C from the console.

- Finally, turn in a report to show the execution times of your code as a function of the number of users, printers and queue size. Also, plot the average waiting times for all the print jobs as a function of these metrics. We expect to see the SJF scheduling to work better than the FCFS scheduling at least in terms of average wait times; however, implementing the SJF scheduler will have its overheads. Additionally mention which portions of your code are not working in the report as we cannot do in person demos anymore.

**The global print queue: two implementations are needed.**

- First use the global count concept for producer/consumer problems from Assignment-2 (maintained by the variable `buffer_index`). This is essentially an FCFS queue.
- Implement a priority queue based on shortest job first. In this scenario, the in/out pointers will not work. If the queue is not full, the next print request gets inserted at the right position in the queue that is sorted on job size. The printer threads will always read from the one end of the queue (depending on your implementation). For this reason, it's better to implement a linked list of structs for the queue; note that you will prepopulate the entire linked list right at the beginning as it is a fixed size queue. The items consumed can be marked as such with an additional field in the struct. Any type of implementation is fine for this component though; so be creative!

**Required semaphore utilities.**

`sem_init()`  
`sem_destroy()`  
`sem_wait()`  
`sem_post()`  
`sem_getvalue()`

**Required pthread utilities.**

`pthread_self()`  
`pthread_exit()`  
`pthread_create()`  
`pthread_join()`

**Required process utilities.**

`fork()`  
`wait()`

---

Note: Late assignments will lose 5 points per day upto a maximum of 3 days. Code must compile and execute on the class Linux server.

---