Assignment #3 - Programming Rubric

- Total Possible Points: 100 per student: (90+10)

- Necessary items for submission
    - Source files - LIFO & FIFO Queue Implementation Source Code
    - Report detailing assignment experiments

This next section details grading criteria for the assignment submission.

Task Coding Requirements (90 pts)

LIFO (45 pts) & FIFO (45 pts)

Listed criteria applies to each implementation to obtain 90 pts total. i.e. 45 + 45 = 90 pts

- Producer-Consumer Synchronization (18 pts)
    - Reads number of producers and consumers from command-line. (2 pts)
    - Implements producers as processes and consumers as threads. (2 pts)
    - Incorporates LIFO/FIFO insert/dequeue algorithm (2 pts)
    - Incorporates printer job buffer as array or linked-list of structs. (2 pts)
    - Global print buffer implemented using shared memory. (2 pts)
        - Buffer initialized to 30 elements. (Size is immutable)
    - Utilizes three semaphores to limit the number of producers/consumers from inserting/dequeuing jobs to/from the printer buffer to 1 process/thread at any given time. (2 pts)
    - Utilizes implementation of the (correct) counting semaphore from assignment #2. (2 pts)
    - Utilizes shared memory / shared semaphores between producers & consumers i.e. processes and threads. (2 pts)
        - Printer buffer synchronization across processes & threads.
            - No buffer overflow or under-run issues
    - Elapsed program times are reported. (2 pts)

- Print Job Preparation, Submission and Job Handling (11 pts)
    - Print job structure contains at minimum: (2 pts)
        - PID of sending process
        - Job size in bytes ranging from 100-1000
    - Print job size generated randomly in producer function (1.5 pts)
        - i.e. JOB_SIZE = (rand() % (1000 - 100 + 1)) + 100;
    - Each process sends a random number of jobs to the printer buffer (up to 30). (1.5 pts)

        - i.e. PROCESSOR_LOOPS = rand() % 30 + 1;
    - Producer includes a slight delay between two consecutive jobs sent to the printer buffer. (0.1 - 1.0 sec) (2 pts)
        - Use nanosleep(), usleep() or sleep() for this.
    - Producer/Consumer output is similar to the following: (2 pts)

      Producer <process-id> added <job_size> to buffer

      Consumer <consumer-id> dequeue <process-id, job_size> from buffer
    - Upon producer/consumer completion (termination) report the following information: (2 pts)
        - Number of jobs processed
        - Total size of processed jobs
        - Output example:

          Producers submitted <total_number_of_producer_jobs> jobs totaling <total_number_of_producer_bytes>

          Consumers processed <total_number_of_consumer_jobs> jobs totaling <total_number_of_consumer_bytes> bytes.

- Termination Detection & Termination Signal Handling (16 pts)
    - Termination Detection i.e. Normal program exit condition
        - Waits until all processes have finished sending jobs to the printer buffer. (2 pts)
        - Waits until all consumer threads consumed and processed all jobs in the buffer (2 pts)
        - Ensures all producer processes terminate gracefully. (2 pts)
            - No lingering processes after main program termination.
            - All processes should detach from shared memory upon completion.
        - Joins, stops or terminates all consumer threads. (2 pts)
        - Deallocates shared memory / Shared semaphores (2 pts)
            - Global printer buffer and semaphores
    - Termination Signal Handling i.e. SIGINT signal sent to program
        - Signals all processes to terminate (2 pts)
            - kill(pid) or use shared global termination flag
        - Terminates consumer threads. (pthread_cancel) (2 pts)
        - Deallocates shared memory / Shared semaphores (2 pts)
            - Global printer buffer and semaphores

Report Requirements

- Give an outline of the working portions of your code. For each part not functioning, list them out.
- Show the following:

    - Logic used to identify the terminating condition
    - How were the semaphores and other book-keeping variables shared between processes and threads?
    - What was done in the signal handler for graceful termination?
    - How did the LIFO and FIFO implementations differ in terms of your usage of either the ‘buffer_index’ variable or ‘in/out’ pointers (or some other method that you may use for the FIFO queue)?
- Show two sample runs of your code i.e. Copy & paste the output from the server:
    - LIFO
    - FIFO
- Finally, show the execution time and average waiting plots for LIFO and FIFO for different values of number of producer processes and number of consumer threads. Plots will be good to show here or you can simply use a table format; vary both number of producers and number of consumers.
- Two options:
    - 3-D graph: you will just have a single 3-D graph for LIFO, and another 3-D graph for FIFO to report execution times.
    - 2-D graphs: 5 different graphs are needed for the number of producers = 2, 4, 6, 8, 10 respectively.
        - In each graph, plot execution time VS number of consumers (varied as 2, 4, 6, 8, 10) for LIFO and for FIFO. Each 2-D graph will have two lines, one for LIFO and another one for FIFO.
- Other statistics to include in your report:
    - Report total number of produced jobs (producers) and total size in bytes.
    - Report total number of consumed jobs (consumers) and total size in bytes.

Bonus Points (10 pts)

- Implement the extension where each queue element is controlled by a separate binary semaphore. (You need 30 additional binary semaphores to do this).
    - You may choose either implementation (LIFO or FIFO) for this.
- Show the execution time improvements (if any) for this implementation.

Miscellaneous Items

- Your code must compile and run on the class server.

- If the assignment has been submitted late, a penalty is deducted from the total score per Dr. Ghosh’s programming specifications sheet.

- Discretion will be utilized when calculating scores so that you have the maximum number of points possible.

Helpful Hints

- Start early and pace yourself! The earlier you start the assignment, the more you can focus on the individual components of the assignment and be sure everything is working as expected.
- Do not try to complete more than a single task at any given time. Handling one task at a time is the best approach to finishing the assignment with minimal issues. Debugging can be a nightmare if you try to implement the global queue in shared memory while implementing your FIFO solution using your correct semaphore implementation at the same time.
- Try to consider each task in isolation from one another. (However, some tasks depend on the completion of others).
- When implementing your counting semaphore in shared memory
    - This Stack overflow page may become useful to you: Stack Overflow Link
- There are many ways to implement the correct solution, but many more to produce the incorrect result. Implementation details are up to you. Use your best judgment given what you have learned.
- Use the knowledge you have gained from the previous assignments when completing this one.
    - If possible, refer back to 'Assignment 1 - Question 2' for a refresher on using shared memory between processes and forking to create these processes.
    - Be sure you know how the buffer-bounded producer-consumer code works and what's happening behind-the-scenes with the 'full_sem', 'empty_sem' semaphores and mutex variables.
- The base 'prod_con-2.c' code is useful to you when completing this assignment. There are many similarities between the base code and this assignment. Consider this assignment an extension of the base code (or Assignment 2).
- Ensure you have a thorough understanding of the differences between mutexes, binary semaphores and counting semaphores. Know when each applies given the task at hand.
    - Mutexes should not be used when protecting critical sections from other processes. Semaphores should be used for this.
    - Binary semaphores and mutexes perform 'similar' operations. However, binary semaphores can also protect critical sections among processes, not just threads.
- When implementing code to keep track of the total number of jobs produced/consumed and their total size in bytes:
    - Sharing two global variables among consumers (threads) is fine to accomplish this. But be aware of data-race conditions.
    - This same approach will not work for the producer (processes). But you have other ways of 'sharing' information among processes.
- Do not try to complete the bonus implementation first. Only attempt this when both LIFO and FIFO implementations are 100% complete!
- Use office hours and any time provided to ask questions in class to your advantage!