CS 341: Operating Systems, Spring 2017
Programming Assignment 2: A Producer-Consumer Problem

1 Introduction

Your assignment is to write a multithreaded program incorporating a producer-consumer algorithm. You will write a producer-consumer program with a single producer and multiple consumers. Your producer and your consumers will run as separate threads, so you will use mutexes to protect and manage shared data structures.

2 Producer: Acquisition of Categorized Data

Your program will spawn a single data input thread (the producer). The input thread will read in lines of data where each line specifies a color and a number. The producer will sort (i.e. categorize) the different input lines by color and feed the data for a single color to a single consumer. There will be multiple colors, so there will be multiple consumers. For example, the producer would feed red data to one consumer, the green data to a second consumer and the blue data to a third consumer.

You have (at least) two choices for where the producer thread puts the categorized data. First, you may opt to put the data into separate queues, one for each category. Each consumer thread will have exclusive access to a specialized queue. Second, you may choose to put all data into a single shared buffer that will be used by all the consumer threads. The data queues (or buffer) are initially empty.

3 Consumers: Data Extraction Threads

Your program will also spawn multiple data extraction threads (the consumers), one consumer thread for each category. The data extraction threads will extract all data for their category from the data structure shared with the producer and print the individual data items.

You should write a `red`, `green` and `blue` consumer. The red consumer will extract prime numbers, the green consumer will extract Fibonacci numbers and the blue server will extract multiples of 3 and 5 (in memory of an old drinking game). Each server extracts and prints the required numbers and ignores the others. The individual consumers would not get all of the primes or Fibonacci or multiples, but it would produce interesting output nonetheless.

Requirement 1: there should be a minimum of BUSYWAITING in your code.

Requirement 2: there should be NO DEADLOCKS and NO RACE CONDITIONS in your code.

4 Program Start-up

The program reads in data from stdin.

5 Implementation

This program is an implementation of the Producer-Consumer problem where the producer may create or acquire large amounts of input (i.e. any amount of input data), but have only a small, fixed space to communicate the data to the consumers. Your program must be able to handle any amount of input data. Therefore, it is not acceptable for the producer to simply read in all the data into a large buffer and quit, and then have the consumers simply read the data left behind by the now terminated producer.

To distinguish the output of one thread from another, each thread should indent its output by a distinct number of tab stops for each consumer.

6 Program Termination

After all in input has been processed, the producer and all the consumer threads should shut down and the program should print out the final report as described above.

7 Extra Credit

You could write color-specific consumers that would redirect selected numbers to another downstream consumer, making the upstream consumer a producer as well. The downstream consumer would sort the data from the upstream consumers and print out sorted output retaining the different colors from the upstream consumers. Given randomness of thread scheduler, the downstream consumer could sort multiple input streams using a minheap to hold out-of-sequence values until they become in-sequence values.

8 What to turn in

- A writeup documenting your design **paying particular attention to the thread synchronization requirements of your application**.
- All source code including both implementation (.c) and interface(.h) files.
- A makefile for producing an executable program file.

Your grade will be based on:

- Correctness (how well your code is working, including avoidance of deadlocks.
- Efficiency (avoidance of recomputation of the same results, avoidance of busy-waits).

- Good design (how well written your design document and code are, including modularity and comments).