CSE 5441 Spring'22

Assignment #2 – Pthreads Programming

The producer - consumer problem is a classic programming exercise in computer science. It has both high practical benefit as well as a straight-forward implementation. This problem is composed of two parts. First, a "producer" creates some sort of workload, storing instructions or intermediate work in a queue. Second, a "consumer" reads this queue, performs the desired work, and produces appropriate output.

The program takes the number of consumers as an argument (defaulting to 1) and a sequence of numbers from stdin. We give you a couple of test sequences: shortlist and longlist. For more explanation of how this works, see the comment at the top of prod_consumer.c

The producer thread reads the sequence of numbers and feeds that to the consumers. Consumers pick up a number, do some "work" with the number, then go back for another number.

The program as provided includes output from the producer and consumers. For reference, a working version of the code with a bounded buffer of size 10 running on shortlist with four consumers produces this output (the comments on the right are added): (NOTE: Your output may not match what is shown identically due to the randomness of thread scheduling. However, your output should show all entries being produced in the correct order and consumed in the correct order).

```
[subramon@haswell1 Homework]$ ./a.out 5 < shortlist
main: nconsumers = 5
buffer_init called
  consumer 0: starting
  consumer 1: starting
  consumer 2: starting
  consumer 3: starting
  producer: starting
  consumer 4: starting
producer inserting 1 at location 0
producer inserting 2 at location 1
  consumer 2 extract 2 from 1
  consumer 4 extract 1 from 0
producer inserting 3 at location 0
producer inserting 4 at location 1
producer inserting 5 at location 2
producer inserting 6 at location 3
producer inserting 7 at location 4
```

```
  consumer 3 extract 7 from 4
  consumer 0 extract 6 from 3
  consumer 4 extract 5 from 2
  consumer 2 extract 4 from 1
producer inserting 8 at location 1
producer inserting 9 at location 2
producer inserting 10 at location 3
producer inserting 9 at location 4
producer inserting 8 at location 5
producer inserting 7 at location 6
producer inserting 6 at location 7
producer inserting 5 at location 8
producer inserting 4 at location 9
  consumer 1 extract 4 from 9
  consumer 3 extract 5 from 8
  consumer 4 extract 6 from 7
producer inserting 3 at location 7
producer inserting 2 at location 8
producer inserting 1 at location 9
  consumer 2 extract 1 from 9
producer: read EOF, sending 5 '-1' numbers
  consumer 0 extract 2 from 8
  consumer 1 extract 3 from 7
  consumer 3 extract 7 from 6
  consumer 2 extract 8 from 5
  consumer 4 extract 9 from 4
  consumer 0 extract 10 from 3
  consumer 1 extract 9 from 2
  consumer 3 extract 8 from 1
  consumer 2 extract 3 from 0
producer inserting -1 at location 0
  consumer 4 extract -1 from 0
  consumer 4: exiting
producer inserting -1 at location 0
  consumer 1 extract -1 from 0
  consumer 1: exiting
producer inserting -1 at location 0
  consumer 2 extract -1 from 0
  consumer 2: exiting
producer inserting -1 at location 0
  consumer 3 extract -1 from 0
  consumer 3: exiting
producer inserting -1 at location 0
producer: exiting
```

```
  consumer 0 extract -1 from 0
  consumer 0: exiting
buffer_clean called
```

Finish the bounded-buffer code in prod_consumer.c, adding synchronization so that the multiple threads can access the buffer simultaneously. **Limit the maximum size of the bounded buffer to 10.**

There are really two problems here: managing the bounded buffer and synchronizing it. I suggest writing and test your bounded buffer implementation first before implementing synchronization.

There are several possible synchronization strategies. As part of this assignment, please propose solutions with both strategies mentioned below and measure the performance and correctness of each solution.

A. [40 points] Use a mutex to wait when the buffer is empty or full.
B. [40 points] Use mutexes and condition variables i.e. using pthread_cond_wait() to wait when the buffer is empty or full.
C. [10 points] Try measuring the execution time with /bin/time. When running with longlist, doubling the number of consumers should roughly halve the execution time. What is the minimum possible execution time?
D. [10 points]: Cleanup the resources allocated by your program.

You should be able to reproduce the output above.

Report Requirements

Run your program against all the test files provided. Your report should contain

- The first 50 and last 50 lines of your program output.
- The 'real' and 'user' time reported by time(1). The time reported **MUST BE** from the Owens supercomputing system at OSC. Include the output of the salloc/sbatch command you used.
- The total runtime of your producer and consumer modules as reported by time(2).
- A brief explanation of the result and program including how the synchronization mechanisms you introduced ensure mutual exclusion.

Testing & Submission Instructions

When compiling and making short (less than a couple minutes) test runs, these jobs can be run on the login nodes from the command line. Larger runs should be performed using salloc/sbatch, which causes your program to run on the supercomputing cluster.

If you are running a job on the cluster, but are not performing your final benchmarks, you can set ppn=1 for this lab. This will cause your program to run in a multiuser environment, so it might take more wallclock time for your job to run, but your job will likely start sooner.

When performing your final “benchmark” runs, where you are measuring the runtime of your program for your report, always set ppn to the max for regular nodes the specific cluster you are running on (ppn=28 for Owens).

You are responsible for completely testing your program. Your program must compile and run correctly on all evaluation data.