CMPUT 379 Fall 2023

Assignment 2

Producer-Consumer Problem

Description

> In computing, the **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
> [Wikipedia]

This assignment is for you to implement a solution to the producer-consumer problem. Think of the producer as a server, accepting incoming transactions and adding them to a queue of work to be done. The consumers are each a thread looking for work from the server to execute. The assignment has several important goals, including exposing you to thread programming, shared memory, and synchronization.

Specifications

You will write a program called `prodcon` that accepts one or two command line arguments:

```
prodcon nthreads <id>
```

where `nthreads` is the number of consumer threads to be created and `id` is optional and represents an integer to be used to differentiate the output files. If `id` is not given, then it defaults to 0. More specifically, the command:

```
prodcon 8 12
```

will spawn 8 consumer threads, and all activity will be logged in the file `prodcon.12.log`.

```
prodcon 6
```

will spawn 6 consumer threads, and with an `id` value of 0 (explicitly or implicitly) all activity will be logged in the file `prodcon.log`.

prodcon reads in input (either from the keyboard or redirected from a file) containing two commands:

`T<n>` Execute a transaction with integer parameter `n`, with `n` > 0. You will have a routine that gets called for each transaction: `void Trans(int n)`. The parameter n is used by Trans to determine how long to simulate the execution of the transaction.

`S<n>` The sleep command simulates having the producer wait between receiving new transactions to process. The producer sleeps for a time determined by integer

parameter `n`, with `n` taking on values 1..100. You will have a routine that gets called for each sleep request: `void Sleep(int n)`.

We will guarantee that the input files used are in the correct format. The code for `Trans` and `Sleep` will be available on *eClass*. Do not change these functions!

When the producer reaches the end of file, then the program will end once all the consumers have completed their work.

Sample Output

Comments (not part of the program's output) are added at the end of the line. Note that:

- Times in the log file are given in seconds, to three decimal places. They represent the amount of real time passed since the program started running.
- The Producer has `ID=0` and all Consumers have `ID`s > `0`.
- Q is the number of transactions received and waiting to be consumed.
- `Ask`/`Receive`/`Work`/`Sleep`/`Complete`/`End` indicate high-level events that occur in the computation, and that are recorded in the log file.
- The last number in a line is the parameter that was given to the `T` or S command.
- Be warned: transactions may appear in your log file in an order you do not expect. For example, lines 3 and 5 in the output below are odd – it seems like the consumer starts doing work (`n=4`) before the producer receives it!

```
~/Assign2> cat inputexample
T4
T2
T1
T1
S9
T5
S1
T1
T1
T1
~/Assign2> prodcon 3 1 < inputexample // Run 3 consumers using id 1
~/Assign2> cat prodcon.1.log
   0.000 ID= 1      Ask               // Thread 1 asks for work
   0.000 ID= 3      Ask               // Thread 3 asks for work
   0.000 ID= 3 Q= 0 Receive      4    // Thread 3 takes work, n=4
   0.000 ID= 2      Ask               // Thread 2 asks for work
   0.000 ID= 0 Q= 1 Work         4    // Parent receives work with n=4
   0.000 ID= 0 Q= 1 Work         2    // Parent receives work with n=2
   0.000 ID= 0 Q= 2 Work         1    // Parent receives work with n=1
   0.000 ID= 1 Q= 1 Receive      2    // Thread 1 takes work, n=2
   0.000 ID= 2 Q= 1 Receive      1    // Thread 2 takes work, n=1
   0.000 ID= 0 Q= 2 Work         1    // Parent receives work with n=1
   0.000 ID= 0      Sleep        9    // Parent sleeps, n=9
   0.002 ID= 2      Complete     1    // Thread 2 completes task, n=1
   0.002 ID= 2      Ask               // Thread 2 asks for work
   0.002 ID= 2 Q= 0 Receive      1    // Thread 2 takes work, n=1
```

```
   0.003 ID= 1      Complete     2    // Thread 1 completes task, n=2
   0.003 ID= 1      Ask               // Thread 1 asks for work
   0.003 ID= 2      Complete     1    // Thread 2 completes task, n=1
   0.003 ID= 2      Ask               // Thread 2 asks for work
   0.007 ID= 3      Complete     4    // Thread 3 completes task, n=4
   0.007 ID= 3      Ask               // Thread 3 asks for work
                                      // Notice nothing is happening…
                                      // because there is no work
   0.090 ID= 0 Q= 1 Work         5    // Parent receives work with n=5
   0.090 ID= 0      Sleep        1    // Parent sleeps, n=1
   0.090 ID= 1 Q= 0 Receive      5    // Thread 1 takes work, n=5
   0.099 ID= 1      Complete     5    // Thread 1 completes task, n=5
   0.099 ID= 1      Ask               // Thread 1 asks for work
   0.100 ID= 0 Q= 1 Work         1    // Parent receives work with n=1
   0.100 ID= 0 Q= 2 Work         1    // Parent receives work with n=1
   0.100 ID= 2 Q= 1 Receive      1    // Thread 2 takes work, n=1
   0.100 ID= 0 Q= 2 Work         1    // Parent receives work with n=1
   0.100 ID= 3 Q= 1 Receive      1    // Thread 3 takes work, n=1
   0.100 ID= 1 Q= 0 Receive      1    // Thread 1 takes work, n=1
   0.100 ID= 0      End               // End of input for producer
   0.101 ID= 2      Complete     1    // Thread 2 completes task, n=1
   0.101 ID= 2      Ask               // Thread 2 asks for work
   0.102 ID= 1      Complete     1    // Thread 1 completes task, n=1
   0.102 ID= 1      Ask               // Thread 1 asks for work
   0.102 ID= 3      Complete     1    // Thread 3 completes task, n=1
   0.102 ID= 3      Ask               // Thread 1 asks for work
Summary:                              // All work complete
    Work          8                   // Producer: # of 'T' commands
    Ask          11                   // Consumer: # of asks for work
    Receive       8                   // Consumer: # work assignments
    Complete      8                   // Consumer: # completed tasks
    Sleep         2                   // Producer: # of 'S' commands
    Thread  1     3                   // Number of 'T's completed by 1
    Thread  2     3                   // Number of 'T's completed by 2
    Thread  3     2                   // Number of 'T's completed by 3
Transactions per second: 78.13        // 8 pieces of work in .102 secs
```

Implementation

Your program will maintain a queue, with the producer adding work to the queue and consumers removing work from it. The queue must be able to hold 2 x #consumers amount of work. For example, if there are 10 consumers, the queue must be able to hold 20 pieces of work. When the queue gets full, the producer has to wait until a piece of work is removed by a consumer before continuing. When the work queue is empty, consumers have to wait until work gets added to the queue by the producer.

There are many ways you can implement synchronization between the producer and consumers, and between the consumers. It is up to you to choose an appropriate method. Two criteria to consider: simplicity (save programming time) and efficiency (there are good software solutions, but something as extreme as using files will not be viewed favorably). Explore the `man` pages to find possible options. Explain your decision in your comments.

The consumer threads should be running concurrently to each other and to the producer. Be careful in that there is only one log file, and you will have multiple producer/consumers wanting to write to that file. Make sure the I/O does not get garbled!

Your program must be implemented in C or C++. Create a `makefile` to compile your program and produce an executable called `prodcon`. Your program must consist of at least three source code files and at least one header file. Your code should be logically organized between these files.

You will be provided with a single file called `tands.c` that contains only the functions `Trans()` and `Sleep()`. Do not change these functions.

Important: make sure you provide a way in your `makefile` to allow us to compile your program with "`-O`" (optimization) instead of "`-g`" (debugging).

Considerations

Here are some important things to watch out for:

- The producer and consumers must all execute concurrently.
- All consumers should be treated roughly equally (e.g., one thread should not be receiving the bulk of the work).
- The log file must not contain garbled output (lines out of order is okay).
- Be careful about boundary conditions – full or empty conditions.
- Do not "over synchronize" your solution (i.e., synchronize beyond the minimum needed for correctness).
- Your `makefile` should do the minimum amount of work required to produce an executable.
- Do not leave anything running after the producer exits. Students who log out and leave processes running – any time before the assignment deadline – will be penalized.

Submission

Submit the following as a single ZIP or TAR file:

- Source code files (C, C++, headers),
- `Makefile`, and
- (optionally) a file called README containing anything special that the TA needs to know about your submission (e.g., acknowledgements, design decisions).

The zip/tar file contains only the above files with no directory structure, and no additional files.

The assignment is due no later than 10:00 PM on Sunday, October 29.