TCSS 422: Operating Systems
Winter 2017
http://faculty.washington.edu/wlloyd/courses/tcss422

Institute of Technology
University of Washington – Tacoma
Instructor: Wes Lloyd

Assignment 3

Multithreaded Producer Consumer

Due Date: Friday March 3rd, 2016 @ 11:59 pm, tentative

Objective

The purpose of this assignment is to implement a multi-threaded producer / consumer bounded buffer based on the Chapter 30 signal.c example described in class. Producer thread(s) will generate R x C matrices, while Consumer thread(s) consume them. Add a SumMatrix() matrix helper routine to add all of the elements of the matrix. The producer thread(s) will sum all matrices produced to compute a total value of all elements, for all *produced* matrices. The consumer thread(s) will sum all consumed matrices to compute a total element value for all *consumed* matrices. Ultimately this leads to the program reporting identical sums for both produced and consumed matrix values. To validate correctness of synchronization used, test that produced and consumed matrix sums are equal using both *random element values* and *fixed element values = 1*. **In all cases, the sum of all matrix elements produced should equal the sum of all matrix elements consumed.** Additionally, the program should not deadlock.

Assignment #3 should be implemented as a modular C program. Each C module file should "encapsulate" related functionality for a piece of the program, similar to how a Java class file encapsulates the elements (data and methods) of a class. Modules in C predate classes in object oriented languages. To facilitate a multiple module programs in C, it is helpful to create header files ".h". Header files declare function prototypes, required data structures, and data for the module. The objective of a modular design is to decouple aspects of the program so that "modules" could be "-in theory-" reused in other unrelated programs. This enables code reuse, and is the basis for development of APIs, shared libraries, etc. To make modular C implementation easier for assignment 3, a tar gzip archive provides the building blocks for a modular program. The tar archive includes separate source files, header files, and a makefile. Most of the matrix module already implemented. This project file is available here:

http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/pcmatrix.tar.gz

Use of the modular code has been provided as an example only. Students are free to use it, or develop their own structure.

As a starting point for assignment 3, download the signal.c example from chapter 30. This provides a working matrix generator which uses locks and conditions to synchronize generation of 1 matrix at a time. There is a producer thread already provided, but the consumer code is implemented inside of int main().

http://faculty.washington.edu/wlloyd/courses/tcss422/examples/Chapter30/

Here are the suggested modules of the pcMatrix program:

Source file	Header file	Purpose
matrix.c	matrix.h	common location of matrix routines
prodcons.c	prodcons.h	common location of producer consumer routines
counter.c	counter.h	common location of shared counter data structure
pcmatrix.c	pcmatrix.h	location of int main(), controls execution of program

New modules could be added beyond these four, if they help encapsulate related sections of functionality as a new module.

prodcons module

This module can be used to provide an adaption of the Producer and Consumer example in the textbook from Chapter 30, section 2 combined with code from the signal.c class example to implement a bounded buffer of 2-dimensional matrices. Put() and get() will add and remove pointers to and from the bounded buffer. The bounded buffer (array) should have a MAX size. This module can also define producer and consumer thread routines. The producer will produce #LOOPS (defined in pcmatrix.h) matrices. Instead of a for loop, as in the book, a shared counter variable should be used. Use of a shared counter allows multiple producer and consumer threads to work in parallel at producing and consuming matrices. Producer and consumer routines should return the sum of all elements, of all matrices they produce to int main() as the output of the thread. This data should be returned via pthread_join. This requires creation of an integer on the heap, and returning a pointer to it.

counter module

This module provides a shared synchronized counter data structure. The counter is used to track whether the producer and consumer threads have produced and consumed up to #LOOPS (defined in pcmatrix.h) matrices.

matrix module

This module provides the shared matrix routines for generating new matrices and performing operations on them. Implement a SumMatrix() routine, which adds every element of the matrix, and returns a value. Hint: this is very similar to AvgElement(). For testing, try both random elements and fix element values of "1". With fixed element values it is easy to determine the correct value for the matrix producer and consumer operations. This helps in proving a solution's correctness. A program could avoid deadlock, yet be incorrect.

pcmatrix module

This module provides flow control for the entire program. It is where int main() lives. Producer and consumer threads are launched here, and the sum of all matrices produced and consumed is determined by adding results provided from the threads, and reporting this to the console using printf.

To watch multiple threads in action, try monitoring active threads using top:

```
top –H –d .5
```

The pcMatrix program should produce output as below. Most of the output code has already been written in pcmatrix.c. The major task will be in implementing the producer and consumer with a bounded buffer to store generated matrices, and to obtain equal matrix sums, for produced and consumed matrices.

```
PROCESS REPORT:
$ ./pcMatrix
Producing 1200000 5x5 matrices.
Using a shared buffer of size=200
With 3 producer and consumer threads.
Produced = Consumed --> 134987753 = 134987753
```

Phases

As with other programs, it is recommended to implement the code in phases to build on successes. Below is a description of suggested incremental phases.

(60%) Phase 1: Implement the bounded buffer producer consumer with one statically defined producer thread, and one statically defined consumer thread.

(80%) Phase 2: Extend the implementation to support two or more statically defined producer threads, with one statically defined consumer thread. Int main aggregates sums of all produced and consumed threads.

(100%) Phase 3: Extend the implementation to support two or more statically defined producer and consumer threads.

(120%) Phase 4: Extend the implementation to support a dynamic number of producer and consumer threads. The number is specified using the NUMWORK constant in pcmatrix.h or as the first command line argument. The program will produce #LOOPS R x C arrays in parallel using #NUMWORK producer threads and #NUMWORK consumer threads. This implementation should enable the programmer to experiment with the optimal number of threads for performance. To receive extra credit programs must: (1) never deadlock, (2) produce correct matrix sums, and (3) correctly support functionality for phases 1-3.

Requirements

Key requirements

1. (R1) Int main will aggregate output from producer and consumer threads. Each thread will return the sum of arrays it produced. With multiple producers and multiple consumers, int main will aggregate the sums for all producers, and all consumers, to provide the output shown in the example.
2. (R2) A concurrent shared data structure will support tracking the number of arrays produced and consumed. The producer and consumer will have unique instances of the counter datatype. The counter will count matrices produced, and matrices consumed. Producers and consumers will work together to produce #LOOPS (defined in pcmatrix.h) matrices. A shared counter is required if there is more than 1 producer or 1 consumer thread. The for loop approach shown in the producer consumer example code will not support multiple producers and/or consumers.
3. (R3) Put() will add a matrix to the end of the bounded buffer array. Get() retrieves a matrix from the other end. When moving to multiple producers and multiple consumers, access to the shared bounded buffer will require synchronization.
4. (R4) This program will require the use of both locks (mutexes) and condition variables.

Grading

This assignment will be scored out of 100* points, while 120 p+oints are available. (120/100)=120%
Any points over 100% will be applied to other program assignments (0, 1, 2, or 4) as extra credit.
* *If necessary the total points scored from may be lowered, while the total available points remains 120.*

Rubric:
120 possible points: (Currently 20 extra credit points are available)

Functionality Total: 65 points	

25 points	Program working as described with 1 producer and 1 consumer thread (Phase 1)
	>>> *10 points, produced sum equals consumed sum*
	>>> *10 points, there is evidence that R3 is supported*
	>>> *5 points, there is evidence that R4 is followed*

20 points	Program working as described with 2 or more statically defined producers, and 1 consumer thread (Phase 2)
	>>> *10 points, produced sum equals consumed sum*
	>>> *10 points, there is evidence that R1 & R2 is supported*

20 points	Program working as described with 2 or more statically defined producers, and 2 or more statically defined consumer threads (Phase 3)
	>>> *10 points, produced sum equals consumed sum*
	>>> *10 points, there is R1 & R2 are supported and working well*

20 points	Program working as described with a dynamic number of producer and consumer threads (Phase 4)
	>>> *15 points, first command line argument supports specification of #NUMWORK, to define a dynamic number of producer and consumer threads. The program will then execute while supporting all requirements (R1-R4) with all functionality working in this mode.*
	>>> *5 points, produced sum equals consumed sum with dynamic number of threads*

Miscellaneous Total: 35 points	
5 points	Program compiles without errors, makefile working with all and clean targets
5 points	Coding style, formatting, and comments
5 points	Matching the described Output example (even when output is incomplete)
10 points	Program is modular. Multiple modules have been used which separate core pieces of the program's functionality
10 points	Global data is only used where necessary. Where possible functions are decoupled by passing data back from routines. In particular in pthread_join.

WARNING!	
10 points	Automatic deduction if final program is not called "pcMatrix"

What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas.

Tar archive files can be created by going back one directory from the source directory with "`cd ..`", then issue the command "`tar cf <lastname>_<firstname>.tar pcMatrix/`". Name the file with your lastname underscore first name dot tar. Then gzip it: `gzip <lastname)_<firstname>.tar`. Upload this file to Canvas.

Pair Programming (optional)

O*ptionally*, this programming assignment can be completed with two person teams.

If choosing to work in pairs, ***only one*** person should submit the team's tar gzip archive to Canvas.

Additionally, ***EACH*** member of a pair programming team must write a summary of their contributions and their teammate's contributions to the overall project. This write up must be written and submitted INDEPENDENTLY by each team member and describe each person's perspective of the teamwork and outcome of the programming assignment. Summaries must describe group contributions in providing functionality for phases 1-4, and requirements 1-4. **Summaries should be a minimum of 200 words.** Contribution summaries should be submitted in confidence to Canvas as a PDF file named: "pair_contributions.pdf". Google docs and modern versions of MS Word provide the ability to save or export a document in PDF format.

Here is an example write up:

1. Jane Smith

   Jane contributed by writing requirement R1 and requirement R2 in the program. Jane also researched several APIs on the web, and was crucial at designing the algorithm to sum matrices. Jane's ideas supported the design of the program output. Jane also discovered and helped code process API routines. Jane was instrumental in testing for deadlock and fixing bugs related to program correctness.

2. John Doe

   John contributed by helping Jane correct a few errors with the implementation of requirement R1 and requirement R2. Additionally, John wrote most of requirement R3. John researched how the Linux system PThread API works for synchronizing threads and exchanging data. John investigated the use of 3 key routines in the API and deciphered how to use them.

Team members may not share their write ups, but should submit them independently in Canvas as a PDF file. Failure of one or both members to submit the contributions report of the required length will result in both members receiving NO GRADE on the assignment… (*considered late until both are submitted*)

<u>Disclaimer regarding pair programming:</u>
The purpose of TCSS 422 is for everyone to gain experience programming in C while working with on projects related to operating systems and parallel programming. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer simply watching the champion! The tasks and challenges should be shared as equally as possible.