TCSS 422: Operating Systems
Winter 2019
http://faculty.washington.edu/wlloyd/courses/tcss422

School of Engineering and Technology
University of Washington – Tacoma
Instructor: Wes Lloyd

Assignment 2

Parallel Matrix Multiplier

Due Date: Sunday March 3rd, 2019 @ 11:59 pm, tentative
Version: 0.10

Objective

The purpose of this assignment is to implement a multi-threaded C program that uses a shared bounded buffer to coordinate the production of NxN matrices for consumption in matrix multiplication. For two matrices M1 and M2 to be multiplied, the number of columns of M1 must equal the number of rows of M2. The program will perform parallel work using multiple threads to: (1) produce NxN matrices and place them into a shared buffer, and (2) consume NxN matrices from the bounder buffer for pairing with another matrix for matrix multiplication having a valid number of rows and columns. Matrices consumed from the bounded buffer with an invalid number of elements for multiplication are discarded and the buffer is queried again to obtain a new candidate matrix for multiplication.

Producer algorithm:

One or more producer threads work together to produce "**LOOPS**" # of matrices and place them in the shared bounded buffer. The producer should call `Matrix * GenMatrixRandom()` to generate a NxN matrix where the number of rows and columns is random between 1 and 4.

Consumer algorithm:

One or more consumer threads work together to perform matrix multiplication. Each consumer thread gets a matrix from the bounded buffer (M1). Then the consumer thread gets a second matrix from the bounded buffer (M2). Calling the matrix.c routine `Matrix * MatrixMultiply(Matrix * m1, Matrix * m2)` will return a pointer with a result of the matrix multiplication (M3), or a `NULL` if matrix multiplication fails due to a mismatch of the number of elements. If a NULL is received, then the consumer thread discards the matrix and memory is free'd by calling `void FreeMatrix(Matrix * mat)`. The consumer thread then grabs the next available matrix from the bounded buffer as M2. When a valid matrix M2 is found that pairs with M1, the matrix multiplication operation is performed and the result in M3 is printed using the `void DisplayMatrix(Matrix * mat, FILE *stream)` routine.

Starter code is provided to help jumpstart implementing the parallel matrix multiplier with the synchronized bounded buffer. The goal of the project is to focus on synchronization and pthreads, not implementing matrix functions and operations as this code is already provided.

Starter code is online at:
http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/pcmultiply.tar.gz

The following modules are provided:

Module	Header file	Source File	Description
Counter	counter.h	counter.c	Synchronized counter data structure
Matrix	matrix.h	matrix.c	Matrix helper routines
Prodcons	prodcons.h	prodcons.c	Producer Consumer worker thread module
Pcmatrix	pcmatrix.h	pcmatrix.c	Program main module with int main()

A Makefile is provided to compile the modules into a pcMatrix binary.

An initial demonstration of the random matrix generation routine, matrix multiplication, and matrix display is provided in pcmatrix.c int main(). The matrix multiplication output format should be followed for the actual program implementation.

The following constant parameters and global values are defined in pcmatrix.h:

DEFAULTS

NUMWORK	DEFAULT number of producer and consumer worker threads.
OUTPUT	Integer true (1) / false (0) to enable or disable debug output. See matrix.c for example use of #if OUTPUT / #endif.
MAX	DEFAULT size of the bounded buffer defined as an array of Matrix struct ptrs.
LOOPS	DEFAULT number of matrices to produce/consume.
DEFAULT_MATRIX_MODE	DEFAULT type of matrices to produce

GLOBALS

BOUNDED_BUFFER_SIZE	Global variable defining the bounded buffer size.
NUMBER_OF_MATRICIES	Global variable defining the number of matrices to produce/consume.
MATRIX_MODE	Defines the type of matrices to produce. (0=random, 1-n=fixed row/col)

Note that the number of worker threads is handled using a local variable called numw in pcmatrix.c.

Code is included in pcmatrix.c to load command line arguments into the global variables for the user. Dynamically sizing the bounded buffer is already done. The matrix_mode is also already implemented in matrix.c. If no command line arguments are provided, the default values are used. A message is displayed indicating the parameterization:

```
$ ./pcMatrix
USING DEFAULTS: worker_threads=1 bounded_buffer_size=200 matricies=1200 matrix_mode=0

$ ./pcMatrix 1 1 2 2
USING: worker_threads=1 bounded_buffer_size=1 matricies=2 matrix_mode=2
```

Otherwise the command line arguments
You are responsible to read and implement NUMBER_OF_MATRICIES to control the number of matrices produced and consumed.

It should be possible to pass different values for these parameters as command line arguments to invoke your program differently for testing purposes.

The following data types are provided:

Struct	Defined in File	Description
`counter_t`	`counter.h`	Synchronized shared counter
`counters_t`	`counter.h`	Shared structure with a producer and consumer counter.
`Matrix`	`matrix.h`	Matrix structure that tracks the number of rows and cols and includes a pointer to an NxN integer matrix.
`ProdConsStats`	`prodcons.h`	Structure that tracks the number of matrices produced, consumed, as well as the sum of all matrices produced and consumed, and the number of matrices multiplied.

The program uses the ProdConsStats struct in prodcons.h to track:

sumtotal	The sum of all elements of matrices produced and consumed.
multtotal	The total number of matrices multiplied by consumer threads.
matrixtotal	The total number of matrices produced by producer threads, and consumed by consumer threads.

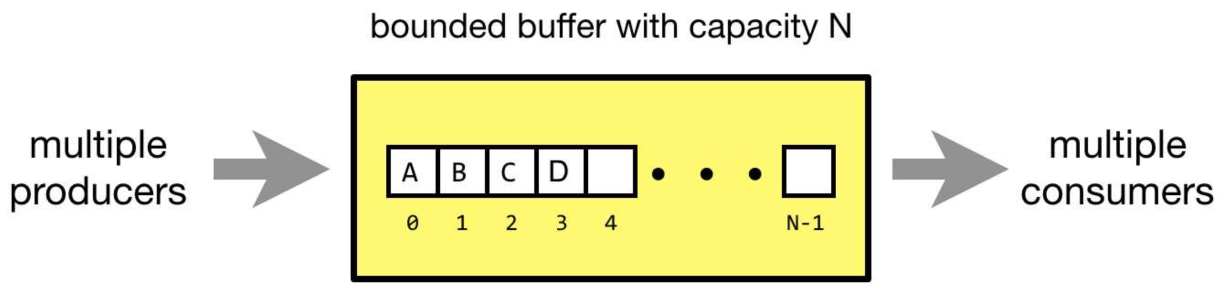
This struct is passed to each consumer and producer thread and used to track the work of the thread. The parent is then responsible for adding up the cumulative work to print out a summary of the total work. The total number of matrices produced and consumed must equal. The sum of all elements produced and consumed must equal. This self-accounting ensures correctness of the program as the number of producer and consumer threads is scaled from 1 to N.
A possible counter design is to have two separate synchronized counters, one to count the number of matrices produced, the other to count the number of matrices consumed. Inside the producer or consumer worker method, having access to both counters is helpful.

Lock and Condition Variable Recommendations

Consider using more than one lock variable in your program implementation. Having multiple locks may allow better parallelism. For example, one lock might only protect adding and removing data from the shared bounded buffer inside the get() and put() routines. Other lock(s) can be combined with condition variables to signal when the bounded buffer is full, or when the bounded buffer is empty. It is one thing to ensure correctness of synchronization (e.g. no threads deadlock). It is another challenge to have an optimal implementation where the maximum number of operations can proceed in parallel to achieve the highest possible Thread Level Parallelism (TLP) for the program. On a multicore machine, when monitoring load with "`top -d .1`", the max percent CPU utilization demonstrates the highest degree of parallelism achieved. On an 8-hyperthead computer, 800% is possible. On a 4-hyperthread computer, 400% is possible.

Program Testing Recommendations

For testing correctness of concurrent programming, try out different sizes of the bounded buffer (MAX). If the bounded buffer is too large, this could minimize errors, and hide possible concurrency problems. The graders will reduce MAX to a low setting to quickly expose flaws. Similarly, only producing and consuming a very small number of matrices (LOOPS) will hide concurrency problems. Testing your program with a large number for matrices (LOOPS) also can help expose concurrency problems.



Sample Output

```
$ ./pcMatrix
Producing 12 matrices in mode 0.
Using a shared buffer of size=5
With 1 producer and consumer threads.

MULTIPLY (1 x 3) BY (3 x 3):
|  5   6   5|
    X
|  2   3   5|
|  4   9   8|
|  7   1  10|
    =
| 69  74 123|

MULTIPLY (2 x 2) BY (2 x 1):
| 10  10|
| 10   8|
    X
|  2|
|  5|
    =
| 70|
| 60|

MULTIPLY (2 x 3) BY (3 x 4):
|  4   1   2|
|  7   5   5|
    X
|  1   9   3   5|
|  3   6   4  10|
|  4   6   1   4|
    =
| 15  54  18  38|
| 42 123  46 105|

MULTIPLY (3 x 4) BY (4 x 2):
|  3   7   9  10|
|  1   3   4  10|
|  3   7  10   6|
    X
|  6   5|
|  6  10|
|  3   6|
|  9   5|
    =
|177 189|
|126 109|
|144 175|

Sum of Matrix elements --> Produced=421 = Consumed=421
Matrices produced=12 consumed=12 multiplied=4
```

Starting Out

As a starting point for assignment 2, inspect 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 to a shared bounded buffer of 1 defined as **`int ** bigmatrix;`**. A producer thread example is provided as the worker routine **`void *worker(void *arg)`**, and the consumer thread code is implemented inside of int main(). It has not been refactored into a separate method- this would be a logical next step. The signal.c example program stores matrices in a bounded buffer of 1. For assignment #2, the bounded buffer becomes an array of Matrix struct pointers: **`Matrix * bigmatrix[MAX];`**.

The signal.c example is here:
http://faculty.washington.edu/wlloyd/courses/tcss422/examples/Chapter30/

Development Tasks

The following is a list of development tasks for assignment #2.

Task 1- Implement a bounded buffer. This will be a buffer of pointers to Matrix structs (records). The datatype should be "Matrix * bigmatrix[MAX]", and the bounded buffer will be limited to MAX size.

Task 2 – Implement get() and put() routines for the bounded buffer.

Task 3 – Call put() from within prod_worker() and add all necessary uses of mutex locks, condition variables, and signals. Integrate the counters.

Task 4 – Call get() from within cons_worker() and all necessary uses of mutex locks, condition variables, and signals. Integrate the counters. Implement the matrix multiplication by consuming matrices from the bounded buffer as described above.

Task 5 – Create one producer pthread and one consumer pthread in pcmatrix.c to launch the parallel matrix production and multiplication.

Tasks 6- Once a 1 producer and 1 consumer version of the program is working correctly without deadlock, refactor pcmatrix.c to use an array of producer threads, and an array of consumer threads. The array size is NUMWORK. (Extra credit for correct implementation of 3 or more producer/consumer pthreads).

Points to consider:

1. A concurrent shared bounded buffer will store matrices for potential multiplication. The use of signals is required to inform consumer threads when there are matrices available to consume, and to signal the producer when there is available space in the bounded buffer to add more matrices. For testing, we might change the size of the bounded buffer (MAX) to a low number, for example 2, to ensure your program still works.

2. Put() will add a matrix to the end of the bounded buffer. Get() retrieves a matrix from the other end. With multiple producers and consumers, multiple matrices can be added and removed for multiplication from the shared bounded buffer simultaneously. You'll need to ensure that no two consumers consume the same matrix.
3. This program will require the use of both locks (mutexes) and condition variables.
4. Memory for matrices should be freed once a matrix is consumed to prevent a memory leak. Without releasing memory, generating millions of matrices will place severe demands on the program's memory heap.

Grading

This assignment will be scored out of 100* points, while 110 points are available.
Any points over 100% are extra credit.

Rubric:
110 possible points: (10 extra credit points available)

Functionality Total: 90 points

15 points	Matrix multiplication support
	>>> *5 points, correctly identify M1 and M2 and production of M3*
	>>> *5 points, discard M2 when incompatible with M1 for multiplication*
	>>> *5 points, free (garbage collect) M1, M2, and M3 after multiplication*
15 points	Display Requirements and command-line arguments
	>>> *5 points, properly show matrices multiplied as in the demonstration code*
	>>> *5 points, display the total number of matrices multiplied*
	>>> *5 points, properly support command line arguments*
40 points	Program working correctly with 1 producer thread to produce matrices and 1 consumer thread to consume matrices for matrix multiplication
	>>> *10 points, put() and get() correctly implement bounded buffer*
	>>> *10 points, synchronization working correctly with mutexes, conditions, signals*
	>>> *10 points, matrices produced equal matrices consumed and displayed*
	>>> *10 points, sum of elements of matrices produced equals sum of elements of matrices consumed and displayed*
20 points	Program is working with multiple producer and consumer threads to provide thread level parallelism > 2 (CPU Utilization > 200%)
	>>> *10 points, 2 producer threads, 2 consumer threads*
	>>> *10 points, 3+ producer threads, 3+ consumer threads*

Miscellaneous Total: 20 points

5 points	Program compiles without errors, makefile working with all and clean targets
5 points	Coding style, formatting, and comments
5 points	Program is modular. Multiple modules have been used which separate core pieces of the program's functionality.
5 points	Global data is only used where necessary. Where possible functions are decoupled by passing data back from routines.

WARNING!

10 points	Automatic deduction if executable binary file 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 pcmatrix.tar pcMatrix/`**". Then gzip it: **`gzip pcmatrix.tar`**. Upload this file to Canvas. Canvas automatically adds student names to uploaded files.

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 provide an **effort report** of team members to quantify team contributions for the overall project. **Effort reports** must be submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member to capture each person's overall view of the teamwork and outcome of the programming assignment. Effort reports are not used to directly numerically weight assignment grades.

Effort reports should be submitted in confidence to Canvas as a PDF file named: "effort_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format.

Distribute 100 points for the categories to reflect each teammate's contribution for: research, design, coding, testing. Effort scores should add up to 100 for each category. Even effort 50%-50% is reported as 50 and 50. **Please do not submit 50-50 scores for all categories. This is not necessarily realistic or possible.** Ratings should reflect an honest confidential assessment of team member contributions. ***50-50 ratings and non-confidential scorings run the risk of an honor code violation.***

Here is an **effort report** for a pair programming team (written from the point of view of Jane Smith):

1. John Doe

Research	24
Design	33
Coding	71
Testing	29

2. Jane Smith

Research	76
Design	67
Coding	29
Testing	71

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

Disclaimer regarding pair programming:

The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. 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.

Change History

Version	Date	Change
0.1	2/12/2019	Original Version