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

Assignment 2

Multi-threaded Parallel Matrix Multiplier

Due Date: Thursday May 31, 2024 @ 11:59 pm, tentative
Version: 0.11

Objective

The purpose of this assignment is to implement a multi-threaded C program that uses a shared bounded buffer to coordinate production of NxN matrices for use in matrix multiplication. To multiply two matrices, the number of columns of M1 must equal the number of rows of M2.

The program will perform parallel work using multiple types of threads:
Producer threads: will add NxN matrices and place them into a shared bounded buffer.
Consumer threads: will remove NxN matrices from the bounder buffer to pair with another matrix for multiplication when there are a valid number of rows and columns.

The consumer threads discard matrices removed from the bounded buffer that have an invalid number of elements for matrix multiplication.

Producer algorithm:

One or more producer threads work together to produce a specified number of matrices described by the "**LOOPS"** compiler directive in pcmatrix.h and place them in the shared bounded buffer. The producer calls **`Matrix * GenMatrixRandom()`** to generate an NxN matrix where the number of rows and columns is random between 1 and 3.

Consumer algorithm:

One or more consumer threads 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 retains M1, and 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. Much of this starter code is based on Chapter 30 of the Three Easy Pieces

textbook. 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

Optionally this program can be completed in Java.
Java starter code is online at:
http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/pcmultiply_java.tar.gz

To encourage program implementations in C, 5 points bonus are awarded to C implementations.

Notes regarding Java implementation:
If coding in Java, solutions are required to use the Java equivalent of C's condition variables.
In Java, the java.lang.Object parent class integrates the equivalent of condition variables with the methods: wait(), notify(), and notifyAll(). Like in C, conditions need to be associated with a lock, which is also called a "monitor" in Java. In Java, locks (monitors) are implicit. Similar to how pointers are hidden in Java, locks are hidden through incorporation with synchronized methods and code blocks. Any thread that executes a synchronized method or a synchronized block of code first acquires an "implicit" hidden lock from Java that is not concretely declared. This may impact the readability of the code. Java also offers an explicit lock similar to pthread_mutex_lock called ReentrantLock. Use of ReentrantLock is not required. However we ask that solutions be coded with either synchronized methods, synchronized blocks, or ReentrantLock so they are reasonably similar to an implementation in C. **Use of any other synchronization mechanism (e.g. Semaphore, Java atomic data types, Google Guava monitor, or other 3rd party locking APIs) is strictly prohibited.** Solutions using these constructs will be returned ungraded and the program will receive 0 points until the program is resubmitted using the required locking constructs.

The following modules are provided:

C Module	Header file	Source File	Description
Counter	counter.h	counter.c	Synchronized counter data structure from Ch. 30
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()

In C, a Makefile is provided to compile the modules into a pcMatrix binary.

Java Classes	Source File	Description
Counter	Counter.java	counter data structure (synchronization must be added)
Matrix	Matrix.java	Matrix helper routines
Producer	Producer.java	Empty Producer class which will implement producer thread
Consumer	Consumer.java	Empty Consumer class which will implement consumer thread
Buffer	Buffer.java	Empty Buffer class which will implement the bounded buffer (put/get)
PCMatrix	PCMatrix.java	Program main module with public static void main()

In Java, a pom.xml file is provided to build a Maven project. Use of maven is required for a Java solution. Java solutions with a non-working maven build will be returned without review and will receive 0 points.

Maven projects are supported by most major Java IDEs. Netbeans is the recommended IDE for working with Java maven projects. Projects must operate using Java 11 (default for Ubuntu 22.04).
To build the project from the command line:

```
# cleans existing build files
$mvn clean

# builds a new Java jar file
$mvn install

# run the sample code
$cd target
$java -jar pcmultiply-v1.jar
```

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

Your program should accept command line arguments.

Code is included in pcmatrix.c and PCMatrix.java to obtain command line arguments and load them into global variables for the user.

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

$ java -jar pcmultiply-v1.jar --help
Unrecognized option: --help
usage: PCMatrix
 -b,--bounded-buffer-size <arg>   the size of the bounded buffer
 -m,--matrices <arg>              the number of matrices
 -o,--matrix-mode <arg>           the matrix mode
 -t,--worker-threads <arg>        the number of worker threads
```

Starter code is provided to load command line arguments into global variables for use throughout the program to control these settings. Implementation of the global variables is required for program testing.

Global variables include in pcmatrix.h and PCMultiply.java:

```
int BOUNDED_BUFFER_SIZE;    // Size of the buffer ARRAY  (see ch. 30, section 2, producer/consumer)
int NUMBER_OF_MATRICES;     // Specifies the number of matrices to produce/consume
int MATRIX_MODE;            // MATRIX MODE FLAG: mode 0: generate random matrices,
                            // 1-n: fixed # of rows/cols with elements of 1
```

In Java, there is a 4th global variable, “WORKER_THREADS” which is the number of producer and consumer threads the program should create and use. In C, this variable is defined as the local variable “numw” in pcmatrix.c.

You are responsible to implement the use of the command line arguments with these variables throughout the program to manage the number of matrices produced and consumed, the number of producer and consumer threads, the matrix generation mode, and the bounded buffer size.

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 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

In C, 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 implementation is to use 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. (see Chapter 30)

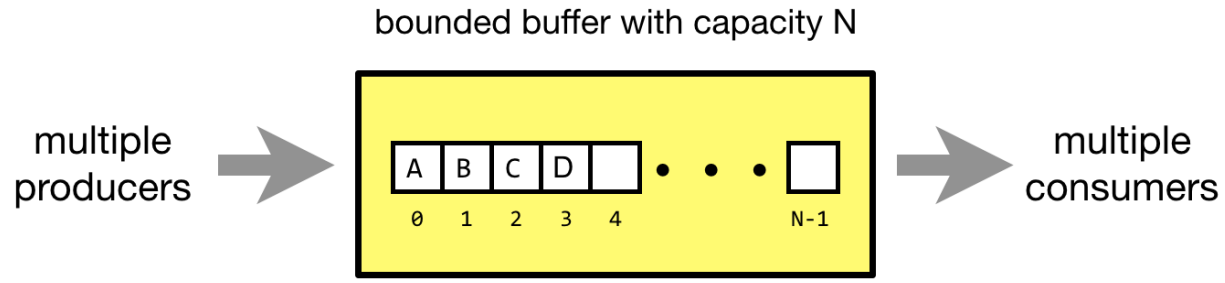
Lock and Condition Variable Recommendations

Consider using more than one lock variable in your program implementation. For example, one lock might 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 will minimize errors, and hide possible concurrency problems. The 422 grader will reduce MAX to a low setting to test for 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. (i.e. millions or more) To increase the synchronization challenge, direct your program's output to "/dev/null". By canceling standard output, this will increase the speed of your program potentially exposing additional concurrency issues.



Sample Output

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

MULTIPLY (3 x 1) BY (1 x 3):
|  5|
|  5|
| 10|
    X
|  7   1   8|
    =
| 35   5  40|
| 35   5  40|
| 70  10  80|

MULTIPLY (1 x 1) BY (1 x 1):
|  9|
    X
|  7|
    =
| 63|
```

```
MULTIPLY (1 x 3) BY (3 x 2):
|  9   4   6|
    X
|  6   7|
|  2   8|
|  4   2|
    =
| 86 107|

MULTIPLY (1 x 3) BY (3 x 2):
|  7   6   7|
    X
|  3   1|
|  7   9|
|  1   7|
    =
| 70 110|

Sum of Matrix elements --> Produced=190 = Consumed=190
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. The signal.c example is here:
http://faculty.washington.edu/wlloyd/courses/tcss422/examples/Chapter30/

In assignment #2, the bounded buffer is defined in the C starter code file: **prodcons.h**.

```
Matrix ** bigmatrix;
```

The buffer can then be initialized as follows where MAX is the size.
The bounded buffer is a NULL terminated array of struct Matrix pointers of size "BOUNDED_BUFFER_SIZE":

```
bigmatrix = (Matrix **) malloc(sizeof(Matrix *) * BOUNDED_BUFFER_SIZE);
```

Development Tasks

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

Task 1- Implement the "bigmatrix" bounded buffer as described above. The "bigmatrix" bounded buffer is a buffer of pointers to Matrix structs (records). The buffer should be limited to BOUNDED_BUFFER_SIZE 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. Calculate running total for produced matrices.

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. Calculate running total for consumed matrices.

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.

Java Resources

https://www.baeldung.com/java-mutex
https://www.baeldung.com/java-wait-notify
https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

Grading

Rubric:
Scored out of 100 points. 120 possible points: (20 extra credit points available)
EC – indicates EXTRA CREDIT

Functionality Total: 100 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, Display total number of matrices produced and consumed. They should be equal.*

>>> *10 points, Display sum of elements of matrices produced, and sum of elements of matrices consumed. They should be equal.*

25 points Program is working with multiple producer and consumer threads to provide thread level parallelism > 2 (CPU Utilization > 200%)

>>> *10 points, Implementation of 2 producer threads, 2 consumer threads*

>>> ***EC-1:** 5 points, Implementation of 3 producer threads, 3 consumer threads*

>>> ***EC-2:** 5 points, Implementation of 4+ producer threads, 4+ consumer threads*

>>> ***EC-3:** 5 points, program does not deadlock under any condition with at least 3+ producer and consumer threads implemented.*

Miscellaneous Total: 25 points

5 points Program compiles without errors, makefile working with all and clean targets, or mvn build working with install 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.

5 points Program implementation is in C.

WARNING!

10 points Automatic deduction if executable binary file is not called "pcMatrix" in C
Or pcmultiply-v1.jar in Java.

- EXTRA CREDIT- COMMENTS ARE REQUIRED:

Comments must be included at the top of the pcmatrix.c file (or in Java PCMatrix.java) to indicate which extra credit features (e.g. EC1, EC2,and EC3) have been implemented to receive credit. If there is no indication that extra credit features are implemented, no extra credit will be awarded.

*Example of **required** comment:*

```
// EXTRA CREDIT FEATURES: EC2, EC3 implemented
```

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*, it is encouraged to complete this programming assignment 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 category 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.

It is highly unlikely that effort is truly equal for everything. Ratings must 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

TCSS 422 effort reports should include a **short description** of how pair programming was conducted. The description should LIST tools that were used and how they facilitated pair programming. Some recommended tools include: Zoom, Discord, Canvas, Slack, etc.

Team members may not share their **effort reports**, and should submit them independently 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 that only passively participates! Tasks and challenges should be shared as equally as possible to maximize learning opportunities.

Change History

Version	Date	Change
0.1	5/14/2024	Original Version
0.11	5/30/2024	Clarification of task 1 regarding bigmatrix