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

Assignment 3

Matrix Task Processor

Due Date: Friday March 9th, 2018 @ 11:59 pm, tentative
Version: 0.1

Objective

The purpose of this assignment is to implement a multi-threaded task processor, similar to a server, which uses pthreads to constantly scan and complete newly arriving task requests submitted as files into a task directory. Upon task arrival, the Matrix Task Processor program will use a producer thread to add tasks to a bounded buffer. One or more consumer threads will then remove "queued" tasks from the bounded buffer and perform them. The tasks involve matrix operations. This program provides an experience in developing a server-style application where files are used to submit task requests and generate task output. This is an alternative to a TCP/IP server where requests arrive as packets for processing.

For assignment #3, the task processing framework has been provided. The assignment requires implementing the threaded portion of the application. This requires adding a bounded buffer to collect and disseminate tasks to the consumer threads.

"Producer" Thread: For assignment #3, there will be just 1 producer thread. The producer thread will be started when the program is launched and will run the "readtasks()" routine. This thread, once launched, constantly scans the "tasks_input" directory. Any files in this directory are processed. Each file will consist of one or more commands. The "readtasks()" routine captures these commands into the "buffer" string. Each individual command should be added to the bounded buffer.

"Consumer" Threads: One or more consumer threads will remove commands from the bounded buffer and execute them. The Matrix Task Processor upon startup should launch one or more "Consumer" threads to consume and work on tasks. These threads will run the "dotasks()" routine.

The crux of assignment #3 is to correctly add pthreading, mutexes, conditions, and the bounded buffer so the task processor operates as a server.

Directories:

Your program will operate on a "tasks_input" and "tasks_output" directory which **must** exist in the same directory as your program's executable file. The "tasks_input" directory will contain command files which describe one or more matrix tasks. All output files are sent to "tasks_output". This module allows us (the professor and graders) to simply drop commands into the "tasks_input" directory. We can then inspect how the program is working by examining the "tasks_output" directory.

The matrix processor should continuously read and process all files in tasks_input until receiving the exit ('x') command. In the absence of the exit command ('x'), this implies reprocessing the same files over again continuously.

Structure of the command files

There are six total commands. Five of the commands have been implemented already. You are responsible for implementing the "average matrix" command.

Here is the command syntax. Each command is described with one line of text in a command file. Command files can have any name. Every file in "tasks_input" will automatically be processed using the code which is already implemented in readtasks().

Create matrix command:

```
c name row col element_type
```

Create matrix will create a matrix and save it to a file using the name specified by "name" with a ".mat" extension to the "tasks_output" directory. The matrix will be created to be row x col size. The element_type is used to specify the type of elements to generate.

Element types:

1	Use 1 for every element
2	Use the column number of each matrix element
3-100	Will pick a random number from 3 to 100 for each matrix element

Display matrix command:

```
d name row col element_type
```

Display matrix will create and display a matrix with the specified configuration and display the matrix on the terminal. No matrix is saved to disk. This command is largely for testing purposes.

Sum matrix command:

```
s name row col element_type
```

Sum matrix will create a matrix with the specified configuration and write the sum of its elements to a file specified by "name" with a ".sum" extension to the "tasks_output" directory. Here is example output of a ".sum" file:

```
sum=25
```

Average matrix command:

```
a name row col element_type
```

*** You are to implement the average matrix command ***
Average matrix will create a matrix with the specified configuration and write the average of its elements to a file specified by "name" with a ".avg" extension to the "tasks_output" directory. Here is example output of a ".avg" file:

```
avg element=4
```

Remove matrix command:

```
r name row col element_type
```

This command simply deletes the matrix file specified by "name" with a ".mat" extension from the "tasks_output" directory. The following parameters are ignored for this command: row col element_type.

Exit command:

```
x
```

The exit command will immediately exit the matrix task processor program.

For help, see the bounded buffer producer / consumer in Chapter 30. Also, check out the signal.c example described in class. (see examples on the course website) To validate correctness of synchronization used, your program will need to complete tasks correctly provided into the "tasks_input" directory. 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 use 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 implementation. (My initial organization could be better!) The tar archive includes separate source files, header files, and a makefile. Most of the matrix module is already implemented. This project file is available here:

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

Sample input files are available online. These can be placed into tasks_input for testing:
http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/sample_tasks
http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/a1

Use of the modular code has been provided as an example only. Students are free to use the provided code, or recode everything from scratch to meet the assignment specifications.

As a starting point for assignment 3, 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. There is a producer thread already provided, but the consumer code is implemented inside of int main(). For signal.c the notion is that matrices are stored in a bounded buffer. For assignment #3, instead of storing matrices in the bounded buffer, ***we are storing commands***. Consumer threads consume and perform commands. The processing engine has essentially been written for you, minus the average matrix command.

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

Here are the suggested modules of the Matric Task Processor program:

Source file	Header file	Purpose
matrix.c	matrix.h	common location of matrix routines
tasks.c	tasks.h	common location of matrix task processor routines
pcmatrix.c	pcmatrix.h	location of int main(), controls execution of program
*taskbuffer.c	*taskbuffer.h	suggested location of bounded buffer of tasks

** - does not yet exist*

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

Taskbuffer module

This module can be used to implement the bounded buffer of matrix tasks. Creation of this module is optional and extra credit is provided if completed and if your program is working.

matrix module

This module provides the shared matrix routines for generating new matrices and performing operations on them. Implement the AvgElement() routine, which averages every element of the matrix, and returns a value. Hint: this is very similar to SumElement(). For testing, try both random elements and fix element values of "1".

pcmatrix module

This module provides flow control for the entire program. It is where int main() lives. Producer and consumer threads are created and launched here. You will need to grab a command line argument from the user, which is the time to sleep in miliseconds between scans of the "tasks_input" directory. By default, a 1 second sleep statement has been used. You are to replace the 1 second sleep with a millisecond sleep function, and grab the value of the sleep from the user as the first and only command line argument to your pcMatrix program.

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

```
top –H –d .2
```

The output of your program will vary greatly depending on requested tasks. Sample tasks have been defined in dotasks() which can be run without any threading or bounded buffer. Try it out to see what the different commands do.

Development Tasks

The following is a list of development tasks for assignment #3. It is probably easiest to accomplish them separately.

Task 1 – Implement AvgElement() in matrix.c. This will calculate the average element of a matrix.

Task 2- Implement task processing for averaging matricies in the dotasks() method in tasks.c.

Task 3- Implement the sleepms() function in tasks.c.

Task 4- Replace sleep() with sleepms(). The sleepms() function should be called when the "in_dir" is closed for reading. The delay will be between subsequent scans of the "tasks_input" directory.

Task 5- Read the sleep interval in ms from the user as the first and only optional command line argument. If the user doesn't provide a duration, 500ms should be used.

Task 6- Implement a bounded buffer. This will be a buffer of variable length strings. The datatype should be "char **" and the bounded buffer will be limited to MAX size.

Task 7 – Implement get() and put() routines for the bounded buffer.

Task 8 – Call put() from within readtasks() and add all necessary uses of mutex locks, condition variables, and signals.

Task 9 – Call get() from within dotasks() and all necessary uses of mutex locks, condition variables, and signals.

Task 10 – Create one pthread in pcmatrix.c and invoke readtasks(). The sleep duration should be passed to the pthread as an argument. **Do not use a global variable.**

Tasks 11- Create one or more pthreads in pcmatrix.c and have each one invoke dotasks(). (Extra credit for implementation of 2 or more consumer pthreads)

UPDATE: It is **recommended** that you call pthread_join() on your threads in int main(). Though, please note, technically they should never join/return…

Requirements

Key requirements

1. (R1) A concurrent shared bounded buffer will support tracking tasks to perform. The use of signals is required to inform consumer threads when there are tasks to consume, and to signal the producer when there is available space in the bounded buffer to add more commands. For testing, we might change MAX to a low number, for example 2, to ensure your program still works.
2. (R2) Put() will add a command to the end of the bounded buffer array of strings. Get() retrieves a command from the other end. With multiple consumers they will both pull commands off the shared bounded buffer from processing at the same time. You'll need to ensure they both don't process the same command.
3. (R3) This program will require the use of both locks (mutexes) and condition variables.

Grading

This assignment will be scored out of 100* points, while 105 points are available.
Any points over 100% are extra credit.
Final program submissions made by Friday March 9th will receive an additional 5% extra credit.
This enables a maximum of (110/100)=110%

Rubric:

105 + 5 possible points: (5 extra credit points available, plus 5 points for final submission by March 9th)

Functionality Total: 85 points

20 points	Ability to average matrices
	>>> 10 points, implementation of routine in matrix.c
	>>> 10 points, command processing in dotasks() in tasks.c
20 points	Ability to sleep in milliseconds between tasks_input directory scans
	>>> 5 points, implementation of sleep routine
	>>> 5 points, retrieving sleep interval as a command line argument
	>>> 10 points, passing sleep interval to thread as an integer (global variable is not used !)
30 points	Program working as described with 1 producer thread to readtasks() and 1 consumer thread to dotasks()…
	>>> 10 points, bounded buffer defined and working
	>>> 10 points, put() and get() work correctly
	>>> 20 points, synchronization working correctly with mutexes, conditions, signals
15 points	Program is working with multiple consumer threads to perform tasks
	>>> 10 points, 2 total working consumer threads
	>>> 5 points, 3+ total working 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 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 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.** 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 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 simply watching the champion! The tasks and challenges should be shared as equally as possible.

Change History

Version	Date	Change
0.1	02/21/2018	Original Version