SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431: Operating System
Winter 2022
Project 2-Multi threading and Synchronization

Date out: Feb 28, 2022
Due on: March 30, 2022 (11:59 PM)

Instructions:

- Final submission should include the code which is compilable and runnable, a 2 page report describing the approach (including the pseudo-code), results and discussion, any innovative features added, reasons for failure (if any) and References (important).
- Cite in the report if you have adapted your algorithm from any web resources/textbooks/papers. As the solutions to some of these problems are already available in the web, it is perfectly fine to refer to them and understand the context and their approach. However, I strongly encourage you to write your own code from the scratch. That is the only way you can practice and get more insights into the problem and the OS in general.
- Final submissions should be a zip file (code and report) and to be uploaded to the MS Class Teams before 11:59 pm on the due date.
- Name your file following this convention: CSCI3431-<*Lastname*>.
- During the evaluation, the students are expected to download the zipped file from the MS Class Teams and show the results on either their laptop or the lab machine using screen sharing. The evaluation will be done in the following two/three recitations or office hours.
- **You may work in groups of two, if you wish to**. If you plan to work in a group, you must inform the instructor about the team details latest by March 11. Team members should divide the tasks meaningfully. During the evaluation, each team member will be asked to explain your role and contribution to the project.

Background:

In Lecture 7, we discussed about threads (light weight processes). You practiced a few minimal examples of multi threading programs in the last two recitations (assignments). In this assignment, you will learn how to design and develop a multi-threading application which involves synchronization. High level constructs for synchronization such as semaphores or mutexes from **pthreads** library will be used in the application. Please start your project as early as possible, there will not be any extension! Consult with the instructor if you have difficulties with the design or prototyping.

Examples:

Multiple readers-writers problem using semaphore. Example Illustrates thread management functions and semaphore functions such as sem_t, sem_wait(), sem_post(), sem_init(), sem_destroy().

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t readCountLock;
sem_t dbLock;
int readCount = 0;

void *Reader(void *args) {

	sleep(1);
	int id = (int)args;
	printf("Reader %d is trying to enter the db\n", id);

	//Enter the db
	sem_wait(&readCountLock);
	readCount++;
	if (readCount == 1) {
		sem_wait(&dbLock);
		printf("Reader %d is reading the db\n", id);
	}

	sem_post(&readCountLock);

	//leave the db
	sem_wait(&readCountLock);

	readCount--;
	if (readCount == 0) {

		printf("Reader %d is leaving the db\n", id);
		sem_post(&dbLock);

	}

	sem_post(&readCountLock);
```

```
}

void *Writer(void *args) {

	sleep(1);
	int id = (int)args;
	printf("Writer %d is trying to enter the db for modifying the data\n", id);
	sem_wait(&dbLock);
	printf("Writer %d is writing into the database\n", id);
	printf("Writer %d is leaving the database\n", id);
	sem_post(&dbLock);

}

int main(int argc, char *argv[]) {

	int i = 0, NumberOfReaderThreads = 0, NumberofWriterThreads;
	sem_init(&readCountLock, 0, 1);
	sem_init(&dbLock, 0, 1);

	pthread_t *WriterThreads, *ReaderThreads;
	printf("Enter the Number of Reader Threads\n");
	scanf("%d", &NumberOfReaderThreads);
	printf("Enter the Number of Writer Threads\n");
	scanf("%d", &NumberofWriterThreads);

	ReaderThreads = (pthread_t *)malloc(sizeof(pthread_t) * NumberOfReaderThreads);
	WriterThreads = (pthread_t *)malloc(sizeof(pthread_t) * NumberofWriterThreads);

	for (i = 0; i < NumberOfReaderThreads; i++) {
		pthread_create((ReaderThreads + i), NULL, Reader, (void *)i);
	}

	for (i = 0; i < NumberofWriterThreads; i++) {
		pthread_create((WriterThreads + i), NULL, Writer, (void *)i);
	}

	for (i = 0; i < NumberofWriterThreads; i++) {
		pthread_join(*(WriterThreads + i), NULL);
	}

	for (i = 0; i < NumberOfReaderThreads; i++) {
		pthread_join(*(ReaderThreads + i), NULL);
	}

	sem_destroy(&dbLock);
	sem_destroy(&readCountLock);

	return 0;

}
```

Sample output:

```
jiju@os:~$ ./multi_RW
Enter the Number of Reader Threads
2
Enter the Number of Writer Threads
2
Writer 1 is trying to enter into database for modifying the data
Writer 1 is writing into the database
Writer 1 is leaving the database
Writer 0 is trying to enter into database for modifying the data
Writer 0 is writing into the database
Writer 0 is leaving the database
Reader 1 is trying to enter into database for reading the data
Reader 1 is reading the database
Reader 1 is leaving the database
Reader 0 is trying to enter into database for reading the data
Reader 0 is reading the database
Reader 0 is leaving the database
```

Your Task:

In project 2, you will implement Floyd-Warshall (FW) All-Pairs-Shortest-Path algorithm in a multi-threaded fashion along with enforcing the readers-writers problem. The single-threaded version of FW algorithm is given below:

Algorithm: Floyd-Warshall (FW) algorithm

```
1 for k = 1 to n do
2      for i = 1 to n do
3             for j = 1 to n do
4                    if (dist[i][k] + dist[k][j] < dist[i][j])
5                    dist[i][j] = dist[i][k] + dist[k][j]
6             end for
7      end for
8  end for
```

In the pseudo-code, the outer *k* loop denotes that we are including *k* as an intermediate vertex in this iteration. The 2D array *dist* represents the distance matrix (*dist[i][j]* = minimum distance between nodes *i* and *j* in the graph till the current iteration). We need to maintain only 1 copy of the matrix. In iteration number *k*, we only need to read values from the k^{th} column and the k^{th} row

of the matrix and we can update values (*i, j*) in the same copy of the matrix without any inconsistency.

Multi threaded Implementation:

In the multi threaded implementation of FW, the *k* loop remains same. Instead of the *i* loop, we create *n* threads that represent each iteration of the *i* loop. Each thread runs the *j* loop from *1* to *n*. There is a global matrix *Graph* that stores the adjacency matrix for the graph. All the threads can read it simultaneously. There is another global matrix *dist* that is used for updating the minimum distances between the vertices. Only 1 copy of *dist* will suffice. At the start of every iteration inside the *k* loop, you need to create *n* threads. Each thread will run its own *j* loop. You have to enforce readers-writers problem in the manner of how the *dist* matrix is accessed by the different threads. Any number of threads can read from the *dist* matrix provided that no other thread is writing to it. Only 1 thread can write to the *dist* matrix at a time. If you observe closely, line 4 has only read statements and line 5 is the only write statement present in the algorithm. **Hence, these two lines need appropriate synchronization.** At the end of every iteration of the *k* loop, you need to wait for all the threads to finish. Then only can you start another iteration of the *k* loop. You have to join the threads at the end of every iteration of the *k* loop.

Analysis:

Implement the single threaded and multi-threaded versions of Floyd-Warshall algorithm. Run the algorithm for graphs of different sizes, e.g., 10, 100, 1000, 10000 and observe the speed up improvement. Do you find any limit on the number of threads that can be created on your system? Compute the speed up (time taken by single threaded version/time taken by multi threaded version) obtained through multi threading and include the **graph plot in your final report with relevant inferences**. Note that to get the real speed up, you have to perform this experiment on a multi core system. Most of the modern computers are multi-core today and can actually run several threads concurrently. Report the system architecture on which you performed the experiment, e.g., processor type and clock rate, # cores, RAM etc...

Input Format:

Graph structure: The first line contains 2 integers *N* and *M*. *N* is the number of nodes. *M* is the number of undirected edges. Then each entry contains the link information (*M* entries). Line *i+1* contains 3 integers u_i, v_i and w_i which represents an undirected edge between nodes u_i, v_i and w_i is the weight of that edge. Use positive edge weights. Your program should inform the user if he/she unknowingly provide a negative edge weight.

Output Format:

Print the final *dist* matrix. In case node *j* is not reachable from node *i*, then print INF as the distance between *i* and *j*.

Sample input and output:

Input

4 4

1 2 1

2 3 1

3 4 1

4 1 1

Output:

0 1 2 1

1 0 1 2

2 1 0 1

1 2 1 0

Suggestions

- Start early!! If you are not comfortable with the language/concepts, it may take you a bit longer to implement.
- Backup your work frequently. It's possible (and most likely) you go try a new feature and your program crashes!
- Document your work properly

Grading Scheme:

Program compiles, runs	25
Shows the expected output	15
Code quality	15
Report	30
Viva	15

References:

1. Design and Analysis of Algorithms, Coreman et al.