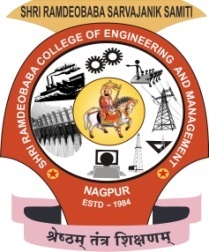
****

**SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT**

**Department of Computer Science & Engineering**

**Session: 2018-19**

**OPERATING SYSTEM**

**ASSIGNMENT**

**IV Semester, B.E.**

**Shift II**

**Group Members**

**Praja Bobde**

**Arpit Kidile**

**Sankalp Jain**

**Course Co-ordinator**

**Prof.Heena Agrawal**

**Statement ID**-**8:A**

Write a program that uses POSIX threads to perform a couple operations on Boolean matrices. All shared data should be declared as global variables. Mutual exclusion and thread synchronization must be achieved by pthread library calls only (mutexes and condition variables). You are not allowed to use shared memory, semaphores or any other inter-process communication mechanism (like pipes).

**Part 1: Generating a Boolean matrix**

The master thread creates an M × M Boolean matrix A with random entries. Each entry of the matrix is zero or one with probability half. The matrix is to be stored in a global two-dimensional array, and may be statically or dynamically allocated (your choice).

**Part 2: Creating the worker threads**

The master thread creates N worker threads among which the following Boolean matrix operations will be equitably shared. Notice that the same threads will take part in all of the following operations. That is, you must not create a fresh set of threads for each individual matrix operation or in each iteration. The master thread must create all necessary resources (like mutexes and condition variables to be used later) before the creation of any worker thread. The worker threads must be joinable.

**Part 3: Counting the number of ones in A**

Each of the N worker threads computes the number of ones in an (M / N) × M submatrix, and adds this count to a shared (that is, global) counter variable. The master thread initializes the counter to zero before any worker thread accummulates its count in the counter. The updating of the counter by the N threads must be mutually exclusive. After all the worker threads accummulate their respective counts, the master thread prints the value of the counter.

**Part 4: Computing the transitive closure of A**

For two Boolean matrices U and V, define the product UV as the Boolean matrix W such that the (i,j)-th entry of W is one if and only if there is a k for which the (i,k)-th entry of U and the (k,j)-th entry of V are both one. This multiplication is not the standard modulo-two product of U and V. The worker threads compute A, A2, A4, A8, ..., A2 e for some e satisfying 2e ≥ m. In each squaring step, the current matrix stored in A is multiplied with itself, and the result is temporarily stored in a second global matrix B. Each of the N worker threads computes M / N rows of the product. After all the worker threads complete their respective parts in the computation of B, they collectively copy back the product stored in B to the matrix A. Each worker thread copies the portion of B computed by it back to A.

Since the locations of A and B modified by the worker threads are disjoint from one another, no mutual exclusion is necessary during squaring or copying back. However, the copying of B to A must start after all computations involving the old A are over. Moreover, the next squaring step is allowed to start only after the entire copy of B to A is over. Use condition variable(s) to synchronize the worker threads.

After A^2^ e is computed (and stored in the global array A), the master thread prints the matrix A.

**Part 5: Winding up Each worker thread individually terminates at this point.**

The master thread waits for all the worker threads to join. After that, the master thread cleans up thread resources, and exits. Submit a single C source file with the name BoolMat.c. Take M = 20 (dimension of the matrix A) and N = 4 (number of worker threads) in your submission.

**Introduction :**

The below program deals with various operations on boolean matrices. The program uses POSIX threads to perform these operations.

**POSIX Threads**

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system. A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. POSIXThreads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API. POSIX Threads is an API defined by the standard POSIX.c

**Global variables**

Globalvariables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A globalvariable can be accessed by any function. That is, a globalvariable is available for use throughout your entire program after its declaration.For Example

#include <stdio.h>

/\* global variable declaration \*/

int g = 20;

int main () {

/\* local variable declaration \*/

int g = 10;

printf ("value of g = %d\n", g);

return 0;

}

**Mutual** **exclusion**

Mutual exclusion means that only a single thread should be able to access the shared resource at any given point of time. This avoids the race conditions between threads acquireing the resource. Monitors and Locks provide the functionality to do so.mutualexclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enters its critical section at the same time that another concurrent thread of execution enters its own critical section Mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy. A Mutex is a mutually exclusive flag. It acts as a gate keeper to a section of code allowing one thread in and blocking access to all others. This ensures that the code being controled will only be hit by a single thread at a time. A mutex is a lockableobject that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations.For Example

#include <synch.h>

mutex\_t Lock;

int mutex\_init(

mutex\_t \*Lock, /\* pointer to a lock \*/

USYNC\_THREAD, /\* use this in CS270 \*/

(void \*) NULL); /\* always use this \*/

int mutex\_lock(

mutex\_t \*Lock); /\* pointer to a lock \*/

int mutex\_unlock(

mutex\_t \*Lock); /\* pointer to a lock \*/

**Thread synchronization**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

**P\_thread library**

The Posix standard defines a number of thread system calls. The posix function to create a new thread within the same process has the following rather ugly function prototype.For Example

#include <pthread.h>

The *<*pthread.h*>* header defines the following symbols:

PTHREAD\_CANCEL\_ASYNCHRONOUS

PTHREAD\_CANCEL\_ENABLE

PTHREAD\_CANCEL\_DEFERRED

PTHREAD\_CANCEL\_DISABLE

PTHREAD\_CANCELED

PTHREAD\_COND\_INITIALIZER

PTHREAD\_CREATE\_DETACHED

PTHREAD\_CREATE\_JOINABLE

PTHREAD\_EXPLICIT\_SCHED

PTHREAD\_INHERIT\_SCHED

PTHREAD\_MUTEX\_DEFAULT

PTHREAD\_MUTEX\_ERRORCHECK

PTHREAD\_MUTEX\_NORMAL

PTHREAD\_MUTEX\_INITIALIZER

PTHREAD\_MUTEX\_RECURSIVE

PTHREAD\_ONCE\_INIT

PTHREAD\_PRIO\_INHERIT

PTHREAD\_PRIO\_NONE

PTHREAD\_PRIO\_PROTECT

PTHREAD\_PROCESS\_SHARED

PTHREAD\_PROCESS\_PRIVATE

PTHREAD\_RWLOCK\_INITIALIZER

PTHREAD\_SCOPE\_PROCESS

PTHREAD\_SCOPE\_SYSTEM

**Condition variables**

Condition variables are variables that represent certain conditions and can only be used in monitors. Associated with each condition variable, there is a queue of threads and two operations: conditionsignal and conditionwait. When a thread calls condition wait, the caller is put into the queue of that condition variable. When a thread calls condition signal, if there are threads waiting in that condition variable's queue, one of them is released. Otherwise, the condition signal is lost.For Example

|  |  |
| --- | --- |
| Initialize a condition variable | [pthread\_cond\_init Syntax](https://docs.oracle.com/cd/E18752_01/html/816-5137/sync-21067.html#sync-59145) |
| Block on a condition variable | [pthread\_cond\_wait Syntax](https://docs.oracle.com/cd/E18752_01/html/816-5137/sync-21067.html#sync-44265) |
| Unblock a specific thread | [pthread\_cond\_signal Syntax](https://docs.oracle.com/cd/E18752_01/html/816-5137/sync-21067.html#sync-53686) |
| Block until a specified time | [pthread\_cond\_timedwait Syntax](https://docs.oracle.com/cd/E18752_01/html/816-5137/sync-21067.html#sync-46756) |
| Block for a specified interval | [pthread\_cond\_reltimedwait\_np Syntax](https://docs.oracle.com/cd/E18752_01/html/816-5137/sync-21067.html#sync-120) |

#include <pthread.h>

pthread\_cond\_t cv;

pthread\_condattr\_t cattr;

int ret;

/\* initialize a condition variable to its default value \*/

ret = pthread\_cond\_init(&cv, NULL);

/\* initialize a condition variable \*/

ret = pthread\_cond\_init(&cv, &cattr);

**Multithreading**

A thread is a path which is followed during a program's execution.These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously. Multitasking is of two types: Processor based and thread based.For Example

#include <stdio.h>

#include <pthread.h>

/\* This is our thread function. It is like main(), but for a thread \*/

void \*threadFunc(void \*arg)

{

char \*str;

int i = 0;

str=(char\*)arg;

while(i < 10 )

{

usleep(1);

printf("threadFunc says: %s\n",str);

++i;

}

return NULL;

}

int main(void)

{

pthread\_t pth; // this is our thread identifier

int i = 0;

/\* Create worker thread \*/

pthread\_create(&pth,NULL,threadFunc,"processing...");

/\* wait for our thread to finish before continuing \*/

pthread\_join(pth, NULL /\* void \*\* return value could go here \*/);

while(i < 10 )

{

usleep(1);

printf("main() is running...\n");

++i;

}

return 0;

}

**Boolean matrix**

Booleanmatrix is a matrix with entries from a Boolean algebra. When the Boolean algebra has just two elements {0,1} the Booleanmatrix is called a logicalmatrix.

**Matrix multiplication**

Matrixmultiplication or matrixproduct is a binary operation that produces a matrixfrom two matrices with entries in a field, or, more generally, in a ring or even a semiring. The matrixproduct is designed for representing the composition of linear maps that are represented by matrices.

**Formula**

Given two n×n matrices A, B over {0, 1}, we define Boolean Matrix Multiplication (BMM) as the following: (AB)[i, j] = Vk (A(i, k) ∧ B(k, j))

**Implementation Description:**

The below program deals with various operations on boolean matrices. The program uses POSIX threads to perform these operations.

The first part of the program creates an M\*M boolean matrix. This boolean matrix is created by the master thread having random entries of 0's and 1's with probability of 50%. Here, rand() function is used to generate 0's and 1's with equal probability.

The second part of the program creates N worker threads by the master thread. The master thread creates N worker threads among which the Boolean matrix operations that are to be performed will be equitably shared. The worker threads used here are joinable. The worker threads are created using pthread\_create function. To make one thread stop and wait for another thread to finish ,pthread\_join function is used. The master thread creates all the necessary resources (like mutexes and condition variables to be used later) before the creation of any worker thread.The mutexes and condition variables are created using pthread\_mutex\_t and pthread\_cond\_t functions respectively.

The third part of the program performs the counting of number of 1's in A. A function named "" is created to perform this operation.Worker threads computes the number of ones in an (M/N)\*M submatrix. The master thread initializes the counter to zero before any worker thread accummulates its count in the counter. The updating of the counter by the N threads is mutually exclusive(done by using pthread\_mutex\_lock and pthread\_mutex\_unlock). After all the worker threads accummulate their respective counts, the master thread prints the value of the counter.

The fourth part of the program computes the transitive closure of the matrix. The worker threads compute A, A2, A4, A8, ..., A2 e for some e satisfying 2e ≥ M. In each squaring step, the current matrix stored in A is multiplied with itself, and the result is temporarily stored in a second global matrix B. A function named “” is created to perform this operation. Each of the N worker threads computes M / N rows of the product. After all the worker threads complete their respective parts in the computation of B, they collectively copy back the product stored in B to the matrix A. Each worker thread copies the portion of B computed by it back to A. To synchronize worker threads, a condition variable is used.After A2 e is computed (and stored in the global array A), the master thread prints the matrix A.

The fifth and the last part of the program is the exciting part. Each worker thread individually terminates at this point. This is done by using the function pthread\_mutex\_destroy The master thread waits for all the worker threads to join. After that, the master thread cleans up thread resources, and exits.

**Program :**-

/\*prg.c\*/

/\* header file inclusion \*/

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<math.h>

/\* matrix size and number of threads \*/

#define M 8

#define N 4

/\* array declaration \*/

int A[M][M], B[M][M];

/\* shared data \*/

/\* global variables \*/

long unsigned int count=0;

int Z[M][M];

/\* mutex and condition variable intialization \*/

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cond1 = PTHREAD\_COND\_INITIALIZER;

/\* boolean number generator with 50% probability of 0 and 1\*/

int rand50 ()

{

/\* rand() function will generate odd or even number with equal probability. If rand() generates odd number, the function willreturn 1 else it will return 0. \*/

return rand () & 1;

}

/\* number of 1 counter \*/

long unsigned int NoOf1 (int x, int y)

{

int i, j;

for (i = x; i < y; i++)

{

for (j = 0; j < M; j++)

{

if (A[i][j] == 1)

{

count = count + 1;

}

}

}

return count;

}

/\* thread function \*/

void \*threadsearch (void \*args)

{

pthread\_mutex\_lock (&lock);

int i, j;

long int number;

static int x = 0;

static int y = M / N;

static int e = 2;

printf ("worker threads\n");

number = NoOf1 (x, y);

printf ("the no of 1's : %lu\n", number);

x = x + M / N;

y = y + M / N;

pthread\_mutex\_unlock (&lock);

}

/\* multiplication function \*/

void multiplication (int x, int y, int e)

{

int i, j, k, t,l,base=2;

int power=1;

for(l=1; l<=e; l++)

{

power = power \* base;

}

for (t = 0; t <power; t++)

{

for (i = x; i < y; i++)

{

for (j = 0; j < M; j++)

{

B[i][j] = 0;

for (k = 0; k < M; k++)

{

B[i][j] = B[i][j] + A[i][k] \* A[k][j];

}

if (B[i][j] > 0)

{

B[i][j] = 1;

}

else

{

B[i][j] = 0;

}

A[i][j] = B[i][j];

Z[i][j] = B[i][j];

}

}

}

for (i = x; i < y; i++)

{

for (j = 0; j < M; j++)

{

printf ("%d\t", B[i][j]);

}

printf ("\n");

}

}

/\* thread function \*/

void \*transitiveClosure(void \*input)

{

pthread\_mutex\_lock (&lock);

int i, j;

int a = \*(int \*) input;

static int x = 0;

static int y = M / N;

printf ("worker threads2\n");

multiplication (x, y, a);

x = x + M / N;

y = y + M / N;

printf ("x=%d y=%d\n", x, y);

pthread\_mutex\_unlock (&lock);

}

/\* main function \*/

int main ()

{

int i, j, done, ch, e;

pthread\_t thread[N], thread2[N];

printf ("\nBOOLEAN MATRIX\n");

for (i = 0; i < M; i++)

{

for (j = 0; j < M; j++)

{

A[i][j] = rand50 ();

}

}

for (i = 0; i < M; i++)

{

for (j = 0; j < M; j++)

{

printf ("%d\t", A[i][j]);

}

printf ("\n");

}

printf ("\n");

do

{

printf ("\nEnter the operation to be performed\n1.No of 1's\t 2.Transitive closure\t0.Terminate\n");

scanf ("%d", &ch);

printf ("\n");

switch (ch)

{

case 1:

{

for (i = 0; i < N; i++)

{

pthread\_create (&thread[i], NULL, threadsearch,(void \*) NULL);

}

for (i = 0; i < N; i++)

{

pthread\_join (thread[i], NULL);

}

printf("\nTOTAL NO. OF 1=%lu\n",count);

break;

}

case 2:

{

printf ("Enter the value of e\n");

scanf ("%d", &e);

for (i = 0; i < N; i++)

{

pthread\_create (&thread[i], NULL, transitiveClosure, (void \*) &e);

}

for (i = 0; i < N; i++)

{

pthread\_join (thread[i], NULL);

}

printf("\nThe transitive closure of matrix is\n");

for (i = 0; i < M; i++)

{

for (j = 0; j < M; j++)

{

printf ("%d\t", B[i][j]);

}

printf ("\n");

}

break;

}

}

}while (ch != 0);

pthread\_mutex\_destroy (&lock);

return 0;

}

**Input & Output :**

