Lab 1 part 1:

dot1m.c compilation and output

Graphical user interface, text

Description automatically generated

dot1s.c compilation and output

Text

Description automatically generated

Lab 1 part 2:

dot1rw.c compilation and output:

Graphical user interface, text, application

Description automatically generated

dot1rw.c listing:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: dotprod\_mutex.c

\* DESCRIPTION:

\* This example program illustrates the use of mutex variables

\* in a threads program. This version was obtained by modifying the

\* serial version of the program (dotprod\_serial.c) which performs a

\* dot product. The main data is made available to all threads through

\* a globally accessible structure. Each thread works on a different

\* part of the data. The main thread waits for all the threads to complete

\* their computations, and then it prints the resulting sum.

\* SOURCE: Vijay Sonnad, IBM

\* LAST REVISED: 01/29/09 Blaise Barney

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

/\*

The following structure contains the necessary information

to allow the function "dotprod" to access its input data and

place its output into the structure. This structure is

unchanged from the sequential version.

\*/

typedef struct {

double \*a;

double \*b;

double sum;

int veclen;

} DOTDATA;

/\* Define globally accessible variables and a reader-writer lock \*/

/\* now moved to main \*/

// #define NUMTHRDS 10

// #define VECLEN 100000

// pthread\_t callThd[NUMTHRDS];

pthread\_rwlock\_t rw\_lock\_sum;

DOTDATA dotstr;

/\*

The function dotprod is activated when the thread is created.

As before, all input to this routine is obtained from a structure

of type DOTDATA and all output from this function is written into

this structure. The benefit of this approach is apparent for the

multi-threaded program: when a thread is created we pass a single

argument to the activated function - typically this argument

is a thread number. All the other information required by the

function is accessed from the globally accessible structure.

\*/

void \*dotprod(void \*arg) {

/\* Define and use local variables for convenience \*/

int i, start, end, len ;

long offset;

double mysum, \*x, \*y;

offset = (long)arg;

len = dotstr.veclen;

start = offset\*len;

end = start + len;

x = dotstr.a;

y = dotstr.b;

/\*

Perform the dot product and assign result

to the appropriate variable in the structure.

\*/

mysum = 0;

for (i=start; i<end ; i++) {

mysum += (x[i] \* y[i]);

}

/\*

Lock a reader-writer lock prior to updating the value in the shared

structure, and unlock it upon updating.

\*/

pthread\_rwlock\_wrlock (&rw\_lock\_sum);

dotstr.sum += mysum;

printf("Thread %ld did %d to %d: mysum=%f global sum=%f\n",

offset,start,end,mysum,dotstr.sum);

pthread\_rwlock\_unlock (&rw\_lock\_sum);

pthread\_exit((void\*) 0);

}

/\*

The main program creates threads which do all the work and then

print out result upon completion. Before creating the threads,

The input data is created. Since all threads update a shared structure, we

need a reader-writer lock for mutual exclusion. The main thread needs to wait for all threads to complete, it waits for each one of the threads. We specify

a thread attribute value that allow the main thread to join with the

threads it creates. Note also that we free up handles when they are

no longer needed.

\*/

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

// modify the program to get MUNTHRDS as the argument1

// and VECLEN as argument2

// that is, run the program: ./dotm 10 100000

//

int NUMTHRDS = 10;

int VECLEN = 100000;

pthread\_t callThd[NUMTHRDS];

long i;

double \*a, \*b;

void \*status;

pthread\_attr\_t attr;

/\* Assign storage and initialize values \*/

a = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));

b = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));

for (i=0; i<VECLEN\*NUMTHRDS; i++) {

a[i]=1;

b[i]=a[i];

}

dotstr.veclen = VECLEN;

dotstr.a = a;

dotstr.b = b;

dotstr.sum=0;

pthread\_rwlock\_init(&rw\_lock\_sum, NULL);

/\* Create threads to perform the dotproduct \*/

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

for(i=0;i<NUMTHRDS;i++) {

/\* Each thread works on a different set of data.

\* The offset is specified by 'i'. The size of

\* the data for each thread is indicated by VECLEN.

\*/

pthread\_create(&callThd[i], &attr, dotprod, (void \*)i);

}

pthread\_attr\_destroy(&attr);

/\* Wait on the other threads \*/

for(i=0;i<NUMTHRDS;i++) {

pthread\_join(callThd[i], &status);

}

/\* After joining, print out the results and cleanup \*/

printf ("Sum = %f \n", dotstr.sum);

free (a);

free (b);

pthread\_rwlock\_destroy(&rw\_lock\_sum);

pthread\_exit(NULL);

}