Lab 3: Shared Memory Programming with PThreads
CMSC 395
Due: Tuesday, Mar. 25 by 11:59:59pm

So far, we have looked at ways to measure the performance of sequential algorithms and learned how to parallelize software using "message passing" libraries such as MPI. The advantage of message passing is that messages can be sent either over a local bus or over a network. This allows us to harness the power of many computers in a large supercomputing cluster. However, it also comes at a huge cost. Communication over a network is very expensive (slow) compared to the nanosecond speeds of modern memory and processors.

This means that while MPI is excellent for algorithms which are largely computational and do not need to exchange data often, it performs less well for problems which require a large degree of information sharing between nodes.

For these applications, shared memory parallelism is a powerful alternative. We will be learning soon about some very powerful shared memory frameworks, but in this lab we will focus on a very low-level library that is supported on nearly every Unix compatible system: the POSIX threads (pthreads) library.

The advantage of shared memory parallelism is that communication happens rapidly (at the speed of the memory bus). The disadvantage is that we now need to protect shared variables against synchronization issues such as race conditions, deadlocks, and starvation.

One common use of pthreads is to implement multi-threaded network servers. Each thread of the server can handle a separate incoming request. This is not a class on computer networking, but I would like you to see how useful pthreads can be in this context.

To that end, I am going to start this lab by having you implement a simple "summation" server using sockets and pthreads. I will give you most of the code, but you will need to add some of the threading functions.

Then, I am going to have you implement the "matrix multiply" algorithm we looked at in Lab 1 using pthreads and measure its performance. As in the previous labs, I will ask you to time your algorithm and create a graph of its performance using Libreoffice.

Finally, I am going to give you a simple "game" I wrote and ask you to use synchronization structures such as "mutex" and "barrier" to prevent it from crashing or misbehaving.

Step 0. Getting Started

Log in to one of the Unix systems in the lab and download the file Lab3.tar.gz from the course web site:

wget http://marmorstein.org/~robert/Spring2021/cs395/Lab3.tar.gz

Then extract it:

tar xvf Lab3.tar.gz

This will create a folder named Lab3 which contains the following files:

ezsocket.h: Interface for a “socket” library that makes writing clients and servers a bit easier
ezsocket.c: Implementation of the “socket” library
running_total.c: Server that uses the “ezsocket” library to

matrix_mult.c: The sequential matrix multiply code from Lab 1.
test1024.txt: A file containing two 1024 matrices to use for testing the matrix multiply algorithm.
output.txt: A file with the result of running a correct matrix multiply on the input file (for testing).
generate_matrix.py: A python script that writes a matrix of size N to a file named “input.txt”

lair.c, player.h, player.c, offer.h, and offer.c: files for the game we are going to implement at the end of this project.

Step 1. Network Server

Compile then running total server with:

```
gcc running_total.c ezsocket.c -pthread -o Total -g
```

Open up a new terminal window and run the server with:

```
./Total 8889
```

The 8889 is a port number – if port 8889 is taken, feel free to pick any other port number over 2000.

This will start a server that waits for 50 incoming connections. Each connection should send it a number to add to the running total and then disconnect. When all 50 connections have completed, the server will print out the total of the numbers.

Here is a short bash script that you can use to test the server:

```
#!/bin/bash
count=0
while [ $count -lt 25 ]; do
      sleep 1;
      echo 5 | nc localhost 8889;
      count=$((count + 1));
done
```

Type this in and save it as “script5.sh”, then make it executable:

```
chmod +x script5.sh
```

Run it with:

```
./script5.sh
```

You will have to run it twice for the server to finish.

Now make a copy of the script:

```
cp script5.sh script7.sh
```

Edit script7.sh so that it sends a 7 to the “nc” command instead of a 5. Open up a third terminal window. Try running the script5.sh and script7.sh scripts simultaneously. Does your server handle the simultaneous connections properly?

Step 2. Matrix Multiply

In your Lab 3 folder is a file matrix_mult.c that contains a sequential implementation of matrix multiplication. Right now, it loops through every row and every column of the output matrix and calculates what value should be there by summing the product of each element in the corresponding row of input matrix A with the corresponding elements of the appropriate column of matrix B.

We can parallelize this by splitting the output matrix up into roughly equal pieces of size SxS where S is the number of rows in the matrix divided by square root of the number of threads.

Let’s start by adding a few constants and variables. After the line that defines N to be 1024, add:

```
#define CHILDREN 16
#define sqrt_p 4
const int stride = N / sqrt_p;

pthread_t threads[CHILDREN];
```

Then change the “mult” function so that instead of taking an int and not returning anything, it is a correct “pthread start function:

```
void * mult (void * pid)
```

Be careful not to accidentally delete the curly braces.

In the loops, replace “size” with N, since we are no longer going to be passing the size in as a separate parameter. Then add this code before the for loops:

```
long int id = (long int) pid;
long int p_row = id / sqrt_p;
long int p_col = id % sqrt_p;
long int row_start = p_row * stride;
long int col_start = p_col * stride;
long int row_end = (p_row+1) * stride;
long int col_end = (p_col+1) * stride;
```

Now change the “row” and “col” loops so that instead of starting at 0 and ending at N, the row loop starts at row_start and ends at row_end, while the col loop starts at col_start and ends at col_end.

In the main function, replace the line mult(N) with these loops:

```
for (long int i=0; i < CHILDREN; ++i) {
      pthread_create(&threads[i], NULL, mult, (void *)i);
}

for (long int i=0; i < CHILDREN; ++i) {
      pthread_join(threads[i], NULL);
}
```

Compile your code with:

```
gcc -pthread matrix_mult.c -g -o Mult
```

and run it with:

```
./Mult test1024.txt > test.out
```

You can then compare it to the correct output using diff:

```
diff test.out output.txt
```

This should not print anything other than the Time and Seconds lines of the output.

Step 3. Performance Measurement

Measure the performance of the Matrix Multiply algorithm as you did for Lab 1. Place your Libreoffice graphs in a file named “Graphs.ods”.

To do this, use the “generate_matrix.py” script to generate ten different input matrices. The generate_matrix script puts its output in a file named “input.txt”, so you will need to move this to ten separate files.

python3 generate_matrix.py 1024
mv input.txt trial1.txt

Use this to create ten files “trial1.txt”, “trial2.txt”, and so forth. Then time your Mult program on all ten trials for 16, 4, and 1 children. To do this, you will need to adjust both CHILDREN and sqrt_p in the code, recompile, and then time it on the ten test cases.

Graph your results against each other using Libreoffice. Does having more children result in faster execution times?

Step 4. A multi-thread game: Lair

It’s now your turn to show me what you’ve learned about POSIX threads. Hasbro has a famous trading game in which players trade cards in an attempt to get complete sets of resources. We are going to implement a simplified version of this and use pthreads to implement a simple real-time game AI.

In “Lair”, players will be dealt a hand of N cards (where N is the number of players). Each card is in one of N “suits” represented by the integers 0 through N-1. For example, in a three player game, the hands might look like this:

Player 1: 0 0 1
Player 2: 1 2 2
Player 3: 0 1 2

Notice that because there are three players, there are three 0 cards, three 1 cards, and three 2 cards distributed randomly across the three hands.

The objective of the game is to be the first to have only one type of card in your hand. This is obtained by offering trades to other players (which they may decline or accept). Trading happens in "real time" (it is not turn-based), so a player might make two trades before another player makes any.

The basic logic for a player's turn is as follows:

Each turn the player will make an offer to trade a certain number of cards from their hand
They will then sleep for a random amount of time
After they wake up, they will decide whether to accept any of the existing offers (including their own)

Once a player successfully claims victory, the game is over and no player can make any further moves.

Trades can only contain one type of card (such as all 0's or all 1's). Also, a card should never be in more than one player's hand at a time.

A player can only make one offer at a time (this minimizes the problem of trading the same card to different players in separate offers).
Here is what an example game might look like:

```
Thread 0 starting
Thread 1 starting
Thread 2 starting

Thread 1 making offer 0[2]
Hand[0]: 1 2 2
Hand[1]: 0 0 1
Hand[2]: 0 1 2

Thread 0 making offer 1[1]
Thread 0 chooses offer 0[2] from 1

Hand[0]: 1 0 0
Hand[1]: 2 2 1
Hand[2]: 0 1 2

Thread 2 chooses offer 1[1] from 0
Thread 1 making offer 2[2]
Hand[0]: 2 0 0
Hand[1]: 2 2 1
Hand[2]: 0 1 1

Thread 2 chooses offer 1[2] from 2
Thread 0 making offer 0[1]
Thread 2 making offer 1[1]
Hand[0]: 2 0 0
Hand[1]: 2 2 1
Hand[2]: 0 1 1

Thread 1 chooses offer 1[1] from 2
Hand[0]: 2 0 0
Hand[1]: 2 1 1
Hand[2]: 0 1 2
Thread 2 making offer 1[1]
Thread 1 chooses offer 1[1] from 2
Game over!
Hand[0]: 2 0 0
Hand[1]: 1 1 1
Hand[2]: 0 2 2
```

After the three "thread starting" messages, thread 1 offers a trade of two "0" cards. Thread 0 then makes an offer of one "0" card. It then accepts thread one's offer of the two "0" cards, trading them for

its two “2” cards. The game continues until thread finally wins by accepting an offer of one “1” card from thread 2 in exchange for its last “2” card. Since thread 1 now holds only “1” cards, it wins.

I have provided code implementing most of this game. It consists of five files: player.h, player.c, offer.h, offer.c, and lair.c.

Here is a detailed explanation of the contents of each file:

lair.c: Contains the main game logic, including the main function, thread management functions such as “start_threads()” and “join_threads()”, the “play_lair” function that implements a single player’s turn, and several helper functions.

player.h: Contains a “struct” representing one player. This contains:
- thread: a pthread_t identifier for the thread controlling that player.
- rank: a number uniquely identifying the player
- hand: an array of N numbers representing the cards the player currently has
- hand_size: the number of cards in the players hand
- offers: the number of offers the player has outstanding (should always be 0 or 1)

After dealing, the hand_size will always be equal to the number of threads.

The player.h file also contains functions for managing the players’ hands.

player.c: Contains implementations of functions for dealing the cards to each player, printing out a player’s hand, determining if a player has won, and freeing memory used by the player object.

offer.h: Contains a “struct” representing a trade offer. This contains:
- rank: An integer indicating which player who made the offer
- size: The number of cards (of the same type) to trade
- value: The type of card to trade
- next: A pointer for use in building a linked list of current offers

The offer.h file also contains the interface for functions for managing the list of current offers.

offer.c: Contains the implementation of functions for checking if an offer is valid (a player has the requisite cards to make an specific offer), making an offer, and choosing an offer.

This is done by managing a linked list of offer “structs” named “offers”. Making an offer adds a new offer to the list, which choosing an offer swaps the offered cards with randomly selected ones from the accepting player’s hand and then removes the offer from the list.

Your task is in some ways simple: I have removed the synchronization structures that protected this code from race conditions. You need to put them back. To do this, you must add three things:

1. Create a pthread_mutex named “offers_mutex”. Lock the mutex at the beginning of the choose_offer function (before we enter the loop) and unlock it at the end of the function. Then use the same mutex to protect the offers list from race conditions in the make_offer function.

The two lines in make_offer that can cause problems are the ones that set the new offer's "next" pointer to the "offers" pointer and the one that sets "offers" to point to the new "offer" struct.

2. Create a pthread_mutex named "output_mutex". Use it to be sure that none of the "printf" statements in play_lair, make_offer or choose_offer will run at the same time as the print_hands function.

3. Use a pthread_barrier named "start_barrier" to ensure that no thread begins actively trading and thae main thread does not begin printing hands until every thread has printed its "thread <rank> starting" message.

Submitting

When you have finished tar up your Lab3 folder:

```
cd ..
tar czvf Lab3.tar.gz Lab3/
```

And upload it to the course web site: http://marmorstein.org/~robert/submit/