50.005 CSE

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Lab 2: Multithreading with POSIX

Natalie Agus

1 Overview: POSIX Threads

In class, we have learned that threads can be used to implement parallelism *if the architecture supports multiprocessors*. Otherwise, we will achieve only *concurrency*, where programs *seem* to be executed in parallel. In essence, a thread is a smaller portion of your program that can be scheduled independently of the main thread (the larger program). So, you can divide the execution of a single, possibly huge process into sub-parts that can each be scheduled by the OS.

In the past, threading is provided via hardware and OS support. However, there was great variance across hardware and OS so portability was a concern for software developers. Hence, a unifying setting was used for UNIX systems: a standardised C language threads programming interface specified by the IEEE POSIX 1003.1c standard. POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming. Implementations that adhere to this standard are referred to as POSIX (Portable Operating System Interface) threads, or Pthreads.

In this lab you're going to learn basics of Pthreads, such as how to initialise and use them to solve simple problems, as well as analyse its performance as compared to sequential programs.

2 Pthreads Basic

2.1 Pthread Creation [3m]

You need to `#include <pthreads.h>` at your code header to link your program with pthread library. Threads are *created* by the starting function (also known as the main thread). Have a look at the file `pthreadbasics.c`, compile, and run it. To refresh your memory on how to compile the C-file from the command line:

1. Navigate to your directory using `cd`
2. Type `gcc -o pthreadbasics.o pthreadbasics.c`
3. Run the output by typing `./pthreadbasics.o`
4. Inspect the output and the code

Notice the function that creates the thread is:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
```

Answer the following questions:

1. **[1m]** In `pthread_create`, what does the first argument mean?
2. **[2m]** What about the third and the fourth? (ignore the second argument for now).

Note: the third and the fourth argument type is `void*`. It just simply means that you are passing the *pointer* (address) of the function for the thread to run, and the address of argument for that thread. The notion of passing *pointer* is apparent in C because it doesn't require further memory allocation to contain a copy of the code for the function or parameter anymore. **Do not worry so much about pointers if you aren't familiar with the idea**.

2.2 Pthread Termination [2m]

In `void * hello_fun(void * args)`, notice there's the function:

```
void pthread_exit(void *value_ptr);
```

Answer the following questions:

1. **[1m]** What is this function for?
2. **[1m]** What does the argument mean?

2.3 Pthread Join [4m]

In the `main()` function, theres the function:

```
int pthread_join(pthread_t thread, void **retval);
```

Answer the following questions:

1. **[1m]** What is this function for?
2. **[2m]** What are the reasons on why do we need to do this?
3. **[1m]** Does it block the main thread if the target thread has yet to terminate?

2.4 Further Observation [2m]

Compile and run `pthreadbasics.c` several times and answer the following questions:

1. **[1m]** Do you get the same output each time you run the program?
2. **[1m]** Why do you think so?

3 Speeding Up Processes with Pthreads: Matrix Multiplication Task [9m]

Although there is overhead in thread creation (each thread requires a thread control block), we can still use the multi-threading paradigm to speed up certain computing processes. In this section, we are going to code a simple matrix multiplication program using multi-threading paradigm.

Have a look at the starter code in `matrixMul.c`. The function `void initMatrices(void)` generates two random matrices (type `int`) M_1: with dimension $M \times N$ and M_2: with dimension $N \times K$ respectively. The function `void multiplyMatricesSequentially()` multiplies the two matrices together sequentially and stores the result in M_3. The function `void printMatrix(matrixName m)` is provided for you to print one of the three matrices to debug your program more easily.

Complete the following tasks:

1. Given M, N, and K, what is the dimension of the output matrix $M_3 = (M_1)(M_2)$?
2. Write the function `void multiplyMatricesWithTThreads(int T);` in the starter code to perform matrix multiplication in parallel using T different threads, where T is a variable. The answer should be stored in M_3.

You may follow these guidelines:

1. You don't have to use mutex, so you can set the smallest task of a thread to be $M_1[i][all] \cdot M_2[all][j]$, which is the dot product between a row of M_1 and a column of M_2.
2. This means that a thread can still compute multiple cells in M_3 but each cell in M_3 is strictly handled by at *most* 1 thread.
3. Hence the maximum value of T is simply $M \times K$.

4 Analysis [5m]

Suggest the optimum number of threads to perform this matrix multiplication task. Thread creation causes overhead, so creating too many threads do not necessarily

speed up the execution of your program. To simplify the process, you may want to set all dimensions of the matrices into one value, e.g: N, and express your answer in terms of % of N. **Provide sufficient plots (time vs thread vs matrix size N) to support your answer**. You can begin by varying the value of T while setting N into some fixed value and perform the matrix multiplication task. Repeat this for different N values.

5 Scheduling Policies [5m]

The second argument of `pthread_create` sets the attribute of the thread you are going to create. There is extensive documentations on attribute setting that is beyond the scope of this class. You can usually get by with providing `NULL` instead, of which the thread is going to be created with default attributes.

One of the attributes that we can set is the scheduling policy of the thread. There are a few standard scheduling policies: `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`, and also `SCHED_DEADLINE`.

1. **[4m]** What does each scheduling type mean?
2. **[1m]** Which of these are the confirming to ”real-time” policies specified by POSIX standard?

6 Pthread Priority [5m]

We can also set the priority level of each thread, so the OS scheduler will know which thread is more ”important”. To do this, add the following codes to `pthreadbasics.c`:

1. Add the following code before the `for` loop in the `main()` function:

```
pthread_attr_t tattr[NUM_THREADS];
struct sched_param param[NUM_THREADS];
```

2. Add the following code after initializations of the thread parameters:

```
pthread_attr_init (&tattr[i]);
pthread_attr_getschedparam (&tattr[i], &param[i]);
param[i].sched_priority = i;
pthread_attr_setschedparam (&tattr[i], &param[i]);
pthread_attr_setschedpolicy(&tattr[i], SCHED_FIFO);
```

3. Under `pthread_create` method, add `&tattr[i]` as its second argument to set the priority levels and scheduling policy of the threads.
4. Compile and run the code, answer the questions below:

(a) **[1m]** Does higher number represent higher priority? You may want to Google this.

(b) **[1m]** Is thread scheduling pre-emptive? You may need to do some googling to answer this.

(c) **[1m]** Is there any difference in the order of output after we set the priority of the threads, as compared to when we did not set any priority?

(d) **[2m]** Now set the priority level for each thread to be the same. Do you notice any difference in the order of output? Why do you think so?

7 Submission Details

The total points you can get from this lab is **[30m]**. Type out your answer and submit it as .pdf together with your modified code `matrixMul.c` and `matrixMul.h` to e-Dimension.