1 Threading using Pthreads

In this lab, the topic of how to create threads, manage threads, and write programs that use threads will be explored. Again, sections 2 and 3 of the man pages will be your best friend for this lab.

2 Creating Threads in a Program

Recall that to create a new thread in a c program, the following lines of code are needed:

```
#include <pthread.h>
.
.
.
pthread_t thread0;
.
.
.
error = pthread_create(&thread0, NULL, (void *)startRoutine, NULL);
.
.
.
```

These lines would create a variable named `thread0` which will hold the ID of a newly created thread, courtesy of the function call `pthread_create`, and will be used to perform various functions on the thread in subsequent pthread calls. The function call `pthread_create` takes in as arguments a buffer to a `pthread_t` variable(`&thread0` in this case), a pointer to a thread attribute structure(the first `NULL`), a function pointer to a function that the thread is to perform(`(void *)startRoutine` here), and finally, a pointer to the arguments to the function that the thread is to perform(the final `NULL` in the example). Do note that leaving the thread attribute structure as NULL will set the attribute of the newly created thread to be the default values.

As an example, here is a piece of code that will create two threads that print out a message special to each thread:

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

void thread1Print(void){
    printf("I am thread 1\n");
}

void thread2Print(void){
    printf("I am thread 2\n");
}

int main(int argc, char **argv){
    int err = 0;
    pthread_t t1;
    pthread_t t2;

    err = pthread_create(&t1, NULL, (void *)thread1Print, NULL);
```

```
    if(err != 0){
        perror("pthread_create encountered an error");
        exit(1);
    }else{
        err = 0;
    }

    err = pthread_create(&t2, NULL, (void *)thread2Print, NULL);
    if(err != 0){
        perror("pthread_create encountered an error");
        exit(1);
    }else{
        err = 0;
    }

    printf("I am thread 0\n");
    return 0;
}
```

Save this code into a file called `pthread_test.c`, and to compile the code, run gcc on the source file, but with a few extra arguments as follows:

```
gcc pthread_test.c -pthread -o pthread_test
```

Note the `-pthread` at the end of the line. This is needed to ensure that the pthread library is linked during the linking phase of gcc's execution.

Run the program produced by compiling `pthread_test.c` and answer the following questions:

- What is one expected output of running this program?
- What is the actual output of the program?

3 One function call to join them all

The behavior of the code in the above example is the result of the main thread(the thread that prints "I am thread 0") not waiting on the other two threads to complete their routine before returning from main. As explained in the man page for `pthread_create`, a thread that is created would terminate if the main thread returns from main, even when the created thread has not completed its task yet. To ensure that the main thread waits until the created threads complete before it continues, the function call `pthread_join` is used. An example of how to use it is as follows:

```
...
pthread_t thread0;
...
err = pthread_create(&thread0, NULL, (void *)startRoutine, NULL);
...
err = pthread_join(thread0, ret);
...
```

The above example will ensure that `thread0` is joined to the main thread, and the main thread will not terminate before `thread0` has finished execution of `startRoutine`.

Add `pthread_join` calls to pthread_test.c after the creation and checking of the second thread, recompile, and run the program:

- What is the output of the program?
- Does it match with the expected output of the program?

4 Tasks for this lab

Now that thread creation and joining have been introduced, it is time to do something fun with it. The student will be tasked with writing a program that would do matrix multiplication for matrices of size 64x64 using pthreads. The specification for the program is as follows:

- The program is expecting three arguments:
  1. A file containing the numbers for the first matrix
  2. A file containing the numbers for the second matrix
  3. A path name for the numbers of the product matrix

- All of the files will be ASCII .dat files.
- If there are insufficient arguments, or one or more arguments cannot be opened, the program should print usage information before exiting.
- If any errors are encountered during program execution, the program should print out error information and exit.

- No segmentation faults should be allowed to occur.
- The output file should be formatted in a similar fashion to the input files.

Example input files will be available in the lab repository.

5 Hint

To make sure that the matrix multiplication is performed as fast as possible, each entry in the output matrix should have its own thread. However, this is not possible due to the fact that there is a limit of 1024 threads that can be allocated using pthreads at any given time, and that there are 4096 entries in the output matrix(assuming that the size of the input matrices are 64x64). Due to this, one thread per entry in the output matrix is not possible. However, there are ways to get around it, and it is up to the student how best to circumvent this limitation.

6 Extra Credit

Extra credit will be given for this lab if the program is capable of:

- handling matrices of any size.
  - Note that in the example input files, the first line is `#N = 64`. This line should tell the program the size of the matrix of the input file.
- The program can perform matrix multiplication in the most optimal way. This means that it should be on par with, or beat the program written to calculate the solution to the grading matrices.

7 License