Threading using Pthreads

February 9, 2016

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 a great resource for this lab.

1 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");
}
```

```c
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?

2 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 one of the expected outputs of the program?

3 Example program

Now that some code examples have been presented, it is time to examine a concrete code example. Examine the `parallel_merge.c` file in the folder. The program is a simple program that generates a large set of random numbers that will populate an array, which it then prints out to a file for reference. Then the program will perform a merge sort on the set using pthreads, as the array is just too large to do with recursion, and it would be too time consuming to perform using a for loop. Once the merge sort is complete, then the sorted array will be printed out into a different file for comparison. To compile the program run the command:

```
$ gcc -o parallel_merge parallel_merge.c -pthread
```

This will compile the program in `parallel_merge.c` into the program `parallel_merge`. Note that the program can take multiple arguments, or none at all. For the purpose of this lab, run the program as follows:

```
$ ./parallel_merge -v -n 4096 > out.log
```

The `-v` option makes this program become verbose, the `-n` sets the array size to be the number following it, and the `-s` option(not shown here) will set the seed of the random number generator to be the number following it. Note that running the program in this fashion produces a lot of output to `stdout`, thus, it is redirected to `out.log`. The original array is printed out to `in_array.dat` and the sorted array is printed out to `out_array.dat`.

The student is encouraged to play around with the parameters passed to the program to better understand how it works. To see how long it takes the program to do the merge, run:

```
$ time ./parallel_merge -n 4096
```

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 solve an extension to the eight queens puzzle, the N queens puzzle, using pthreads. For further information regarding the eight queens puzzle, you can read more about it at https://en.wikipedia.org/wiki/Eight_queens_puzzle. Skeleton code for this lab's task is provided in the `nqueens` folder. This code is based on the algorithm found at https://en.wikibooks.org/wiki/Algorithm_Implementation/Miscellaneous/N-Queens. Please read through it and understand how it runs before beginning work on the task. Note the following:

- Currently, the skeleton code can be compiled and executed, though it uses one thread to find all solutions.
- The single thread solution takes exponentially more time to compute as N increases; this can be greatly reduced if parallelized.
- A library, `libresuse` is used in this lab exercise. This is a library to help keep track of, and to easily display process and thread resource usage while executing any program it is used in.

The task left for the student is as follows:

- Modify the skeleton code in `nqueens.c` to be able to solve the N queens problem in parallel using the `pthread` library.

Before any work should start on the program, please perform the following instructions to acquire all the necessary elements to build and run the skeleton code:

1. Navigate to the `nqueens` directory
2. In the directory, clone the `libresuse` library into the directory by executing:

```
$ git clone git@github.com:jvens/libresuse.git
```

3. Once the cloning of the repository is complete, navigate into the `libresuse` directory and execute:

```
$ make
```

4. Once `make` is done executing, navigate back to the `nqueens` directory and execute:

```
$ make
```

5. Once `make` completes execution, an executable of `nqueens.c` will be available to execute. To test the program run the following command:

```
$./nqueens -n 10 -v -d
```

The programm reccognizes the following flags:

```
- ‘n‘: The number of queens to use
- ‘t‘: Uses multiple threads to solve
- ‘v‘: Display verbose output
- ‘d‘: Display the chess boards with the arrangement of queens
```

From here on, to recompile `nqueens.c` should be performed using the `make` command.

4.1 What to submit

You should submit your working code which can find the number of solutions to the N Queen Problem both using N threads and 1 thread depending on the `-t` flag. You should study how threading effects the time for different numbers of threads. Why is it not linear as the number of threads increases? Is there solutions for which one thread is faster than multiple threads? Please answer all of these questions and give some thoughtful analysis in your lab report.

5 Hint

The threadded solution should complete considerably faster than the single threaded solution for most N. If you are not seeing this there is a good chance you are only spawning one thread at a time instead of all at once. The `-v` flag is useful to see if this is happening.

There is not one correct solution to this problem, but there is one that is easiest to implement. The idea is to use N threads to solve the N queens. Think about what the `for` loop in main is doing, and how each itteration could be done at the same time.

6 Extra Credit

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

- Solving the N queens problem with M threads, where M not equal to N.
- Solving for other chess peices such as knights or bishops.

7 License