Threading Tutorial

Multithreading is a powerful tool that can be used to split a large task into smaller pieces. If your processor has multiple cores, you can run multiple programs, or even parts of a larger program and combine the result. So why don't we do this all the time? Although multithreading is common-place today, it was a rather long road to end up in the multiprocessing revolution we find ourselves in. For one it's just hard. Designing a program to be easily divided into discrete and separate tasks is a difficult ask, and often not possible. There's also the added problems of thread synchronization, latency, and the potential for race conditions. This is where threading libraries, such as pthreads come in. They aim to simplify the process of thread creation, synchronization, and management.

In this tutorial, we will be discussing the C implementation of threading, known as pthreads. Note that C++ is capable of using pthreads, but also has its own threading library: threads. Threads have the advantage of being purely designed for C++ and simpler to implement. Threads in fact wraps itself around a pthread call, so they are rather similar under the hood. However, it does introduce more overhead. We want to be able to not only minimize overhead, but also remove abstractions to better understand how to optimize a multithreaded application, so we will be moving forward using pthreads.

Including The Library

Let's start by including the library:

```
#include <pthread.h>
```

Creating Threads

First you want to create an array of pthreads. The array should be sized to the number of threads you want to create. This number is often based on the number of cores you have. If you have 4 cores for example, you should create 4 worker threads. Likely you will need a synchronization thread to manage the data flows, but this thread can sleep while the workers are running. Creating more worker threads will not make your program faster unless you have more cores to accommodate them. The added time to switch threads will actually slow down your program.

```
int numworkers = 5; //a #define would be good to use here;
```

```
pthread_t *worker_threads = new pthread_t[numworkers];
//C++'s new keyword is used, but malloc or stack allocation can be used in C
```

It's a good idea to store the thread ids in an array. If you want to pass data to the threads (you probably do), pthreads take a void* as there 1 argument. If you are making a synchronization thread, a good way to manage this is to create a simple struct containing both:

```
typedef struct thread_data {
   void *data;
   int threadid;
} thread_data;
```

With this struct, you can create an array of these structs for easier management.

```
thread_data *thread_table = new thread_data[numworkers];
//C++'s new keyword is used, but malloc or stack allocation can be used in C
```

Now we need to create our pthreads. This is done with the pthread_create function:

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

A for loop is a concise way to create all your threads:

```
for (int i = 0; i < numworkers; i++) {
      threadID[i].threadid = i; //set thread id
      thread_table[i].data = &data; //add data here if necessary
      pthread_create(&worker_threads[i], NULL, thread_function,
&thread_table[i]); //pass the information to the          thread
}
```

Thread Synchronization - Barriers

Let's say you start these 4 threads to run some calculations on a dataset. How do you pass more data to them so they can run more calculations? These threads might run at slightly different rates due to uncontrollable factors, so how do we ensure that one thread doesn't outrun the other? We use barriers. Barriers function similarly to how they do in real life: they block threads from continuing forward until a certain number reach it. This is a key tool for synchronizing worker threads.

To create a simple barrier, you use the pthread_barrierr_init function:

int pthread_barrier_init(pthread_barrier_t *restrict *barrier***,**
const pthread_barrierattr_t *restrict *attr***, unsigned** *count***);**

//creates a barrier at the pointer barrier with attributes determined by attr which will lift when count threads reach it

If you'd like to erect a barrier during your thread function you can use the pthread_barrier_wait function:

/* Pre-Barrier Function Code*/

pthread_barrier_wait(barrier) //will wait until the number of threads specified in pointer barrier hit it

/*Post-Barrier Function Code*/

Taking down a barrier is easy too. Just use the pthread_barrier_destroy function.
pthread_barrier_destroy(barrier);

Mutexes:

Sometimes two different threads need to write to the same piece of data. Although simultaneous reads are safe, writes are not. If you have two threads write to the same piece of memory, you will experience race conditions, where each thread attempts to write to memory simultaneously, resulting in corruption and undefined behavior. Luckily, the pthread library includes a useful tool to mitigate this problem: mutexes. A mutex is an object that can be locked by specific threads. When locked, a mutex prevents threads from running critical sections of a program.

To create a mutex, you use the pthread_mutex_init function
pthread_mutex_init(mutex, NULL);
//creates a mutex at pointer mutex

The mutex pointer can easily be passed around to different threads to control program flow.

A thread program's critical section can be preceded by a pthread_mutex_lock(mutex) or pthread_mutex_try_lock(mutex) to ensure that it doesn't pass into a section while another thread has the mutex locked. The standard lock function will attempt to lock mutex and block the calling thread until the mutex is unlocked. The try_lock will return 0 if it locks or a non_zero if it fails.

pthread_mutex_lock(*mutex***);**
pthread_mutex_trylock(*mutex***);**

Once done with the mutex, make sure to lock it before leaving:

pthread_mutex_unlock(mutex);