Tutorial 3: Multithreading and Synchronization

Justin Nguyen 10/11/2019

This tutorial assumes basic knowledge of C++, Makefiles, and Linux.

1. Many different types of functions can be parallelized using threads. This tutorial will cover the use of POSIX pthreads.

In the pthreads API, we will be covering the following objects:

pthread: The main thread object attr: Attributes that can be applied to pthreads mutex: Resource locking mechanism barrier: Synchronization barrier

This pthreads tutorial from LLNL contains more informations about the pthreads library with some good examples

2. To compile a program with pthreads the following are required:

- Include file (source/header): `#include <pthread.h>`
- Linker library (Makefile/cli): `-lpthread`

3. This is the signature for the `pthread_create` function:

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

The following example should show how to initialize threads. Pay close attention to the types of the arguments.

```
#include <pthreads.h>

pthread_t thread[2];

struct threadArgs {
    int a;
    int b;
};

void *fnForThread1();
void *fnForThread2(void *threadArgs);
void *thread1Status;
void *thread2Status;

int main(int argc, char *argv[]) {
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    struct threadArgs = {.a = 0, .b = 1};
    int retVal1 = pthread_create(&thread[0], NULL, fnForThread1, NULL);
    int retVal2 = pthread_create(&thread[1], NULL, fnForThread2, (void *)&threadArgs);
    // Parent thread will continue
    // until we reach a point when we wait for the child threads to finish
    int retVal3 = pthread_join(thread[0], &thread1Status)
    int retVal4 = pthread_join(thread[1], &thread2Status)
    // Parent collects child thread's result
}
```

This snippet of code initializes the attributes for new threads to be joinable. We then create the new threads running the fnForThread1 and fnForThread2 functions. When the parent thread is ready for the child threads result, the pthread_join function will force the calling thread to wait for the child thread to finish.

Note the pthread attr is not always required as some implementations as the default attribute for new threads is joinable.

When actually implementing a multithreaded program, the simplest method of sharing memory between the threads are global variables.

4. To prevent race conditions and meet resource access rules, synchronization is required. The most basic synchronization techniques are mutexes, semaphores, barriers, and monitors. The pthread library includes implementations for this in addition to condition variables which can also be used for thread synchronization. This tutorial will cover mutexes and barriers. The other synchronization techniques are explained in the LLNL tutorial or on many pages online.

Mutex: Named for mutual exclusion, this technique of thread synchronization is used to enforce access to a resource or a serialized section of code. It works like so:

1. A thread locks the mutex
2. Other threads who try to lock the mutex will either wait (`pthread_mutex_lock`) or continue (`pthread_mutex_trylock`) with a busy signal
3. The thread with the mutex lock will do its work in the "critical section"
4. Once the work in the "critical section" is finished, the thread will unlock the mutex allowing waiting threads to continue

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Inside some threaded function
    pthread_mutex_lock(&mutex);
    // Do work in critical section
    pthread_mutex_unlock(&mutex);
    // End critical section
```

Barrier: A synchronization technique that acts like a barrier in that a predetermined number of threads must pause at the barrier before passing. This can be used to control access to a critical section by ensuring some (or all threads) are at the barrier.

```
#include <pthread.h>

pthread_barrier_t barrier;

void setupFunction(int numThreads) {
    pthread_barrier_init(&barier, NULL, numThreads);
}

// Inside some threaded function with multiple threads about to reach the barrier:
    pthread_barrier_wait(&barrier);  // Wait for numThreads
    // Work will commence when enough threads reach barrier
// End of some threaded function

int main(int argc, char *argv[]) {
    int numThreads;
    setupFunction(numThreads);
    // Start threaded work
    // ...
}
```

5. Use the information in this guide to write some threaded programs! GDB can be used to debug multithreaded programs, but it is also very helpful to debug by print statement.