An Application of Pthreads and Mutexes
Ethan Y. Myers
Cedar Brook Academy

An Application of Pthreads and Mutexes

1.0 Introduction

The paper *Multiprocessing Using Pthreads and Mutexes* [1] describes the basics of Pthreads and mutexes and how to effectively create and code them. Creating Pthreads is straightforward. The more complicated part is the application of Pthreads and mutexes in a program. The example program will help explain an application of Pthreads and mutexes.

2.0 Importance of Organization

Documentation is very important in programming, but just as necessary as documentation is organization, especially with Pthreads. When examining the code in this paper, note that there are many different sections in the code. There are three sections for defining, initializing, and locking the mutexes. There is one section for creating, initializing, and setting the attributes for the Pthreads. Additionally, there are three sections for defining and creating Pthreads. Finally, there is a section for the main program. Without this type of organization, all these different functions would be used in apparently random places and it would be hard to know what was created and where. The programmer would have a difficult time changing the code without structural organization. With this arrangement, it is easy and quick to find what is created and where it is used. Note, for mere information on defining and creating threads, refer to the paper *Multiprocessing Using Pthreads and Mutexes*.

3.0 Example Implementation of Pthreads

Following is an example program using Pthreads and mutexes. This code is an abridged example of the actual Cedar Brook Academy (CBA) code. Omitted are the threads that control the functioning of the claw at the same time that the arm is functioning. In total, the full system has seven threads. The program is an example of what the CBA code does in the first few seconds of the match. It grabs Botguy with a claw that is swung out by an arm. The idea behind the code is to be time efficient and as quick as possible. Using Pthreads and mutexes helped us accomplish this goal. As one will see in the program, there will be four different threads executing at the same time, including the main thread. More detailed explanations of the code will follow.

```
#include <pthread.h>

////////////////////////////////////////////////////////////////////////////////////////////////////
/* Define the mutexes */
/* Mutexes must be global so the other functions can see it */
pthread_mutex_t swing_arm_forward_m;
pthread_mutex_t grab_botguy_m;
pthread_mutex_t swing_arm_back_m;
////////////////////////////////////////////////////////////////////////////////////////////////////

int main()
{

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Initiliaze all the mutexes */
    pthread_mutex_init(&swing_arm_forward_m, NULL);
    pthread_mutex_init(&grab_botguy_m, NULL);
    pthread_mutex_init(&swing_arm_back_m, NULL);
    ////////////////////////////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Lock all the mutexes */
    pthread_mutex_lock(&swing_arm_forward_m);
    pthread_mutex_lock(&grab_botguy_m);
    pthread_mutex_lock(&swing_arm_back_m);
    ////////////////////////////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Create, Initialize and Set the attributes of the Pthreads */
    pthread_attr_t CBA_pthread_attr;
    pthread_attr_init(&CBA_pthread_attr);
    pthread_attr_setdetachstate(&CBA_pthread_attr, PTHREAD_CREATE_DETACHED);

    pthread_attr_t CBA_pthread2_attr;
    pthread_attr_init(&CBA_pthread2_attr);
    pthread_attr_setdetachstate(&CBA_pthread2_attr, PTHREAD_CREATE_JOINABLE);
    ////////////////////////////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Define the threads */
    pthread_t swing_arm_forward_t;
    pthread_t grab_botguy_t;
    pthread_t swing_arm_back_t;
    ////////////////////////////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Create all the threads */
    /* attr2 is joinable, attr is detached */
    pthread_create(&swing_arm_forward_t, &CBA_pthread_attr2, swing_arm_forward, NULL);
    pthread_create(&grab_botguy_t, &CBA_pthread_attr2, swing_arm_forward, NULL);
    pthread_create(&swing_arm_back_t, &CBA_pthread_attr, swing_arm_forward, NULL);
    ////////////////////////////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////////////////////////////
    /* Main Program */

    wait_for_light(7);                                  // Wait for the game to start

    pthread_mutex_unlock(&swing_arm_forward_m);         // Swing the arm to prepare to grab botguy
    go_straight(distance_to_botguy, top_speed);         // Move to botguy
    pthread_join(&swing_arm_forward_t, NULL);           // Wait till the arm is ready to grab botguy

    pthread_mutex_unlock(&grab_botguy_m);               // Grab Botguy
    pthread_join(&grab_botguy_t, NULL);                 // Wait till it is done grabbing botguy

    pthread_mutex_unlock(&swing_arm_back_m);            // Swing the arm back
    go_straight(cup_distance, top_speed);               // Move forward

    // Rest of code...

    ////////////////////////////////////////////////////////////////////////////////////////////////
}

```

```
74 void swing_arm_forward()
75 {
76    pthread_mutex_lock(&swing_arm_forward_m);
77    do_stuff_to_swing_arm_forward();
78 }
79
80 void grab_botguy()
81 {
82    pthread_mutex_lock(&grab_botguy_m);
83    do_stuff_to_grab_botguy();
84 }
85
86 void swing_arm_back()
87 {
88    pthread_mutex_lock(&swing_arm_back_m);
89    do_stuff_to_swing_arm_back();
90 }
91
```

4.0 Explanation of Code

Notice that the '`pthread.h`' header file is included in the program. Without it, the program cannot use Pthreads. Also note that everything is passed by reference. This means that a pointer to the object is passed to a function as opposed to a copy of the object being passed. This is needed because the functions using the threads and mutexes need to be able to change the objects that represent a thread or mutex. In object oriented terminology, `pthread_t`, `pthread_mutex_t`, and other Pthread objects are opaque objects, which means that they have to be updated indirectly by using a function call.

4.1 Overview of Code

In the code there are four threads, the main thread, a thread for swinging the arm forward, a thread for grabbing Botguy, and a thread for swinging the arm back. Each thread has an associated mutex and Pthread. With each mutex, it must be defined, initialized, and locked. Similar with the Pthreads, each mutex must be defined and created. There is only one attribute of the Pthread that is used, its *detach state*. The *detach state* has two settings, *joinable* and *detached*. For each different state, the program creates an attribute setting associated with it. Each attribute must be defined, initialized, and set. The explanation of the main code will come later.

4.2 Mutex Sections of the Code

The very first section in the code (lines 3-9) defines the mutexes globally, which means that they are in the scope of all functions. The mutexes are defined globally so that the three functions, `swing_arm_forward()`, `grab_botguy()`, and `swing_arm_back()`, can lock the mutexes. Without the mutexes being defined globally, the functions would not be able to lock the mutexes because they would be non-existent in the function's scope.

The second section (lines 14-19) initializes the mutexes. When initializing the mutexes, passing in `NULL` tells the program to set the attributes of the mutex to the default settings.

The third section (lines 21-26) locks the mutexes. The mutexes are initially locked so that when the Pthreads are created, they do not start the function at that time. This will be described later in more detail.

4.3 Pthread Sections of the Code

The fourth section (lines 28-37) creates, initializes, and sets the attributes that will define the threads properties. The attribute `CBA_pthread_attr` is set as detached, and the attribute

`CBA_pthread_attr2` is set as joinable. These will later be passed to the `pthread_create()` command when the program creates the Pthreads.

The third section (lines 39-44) defines the threads. The fourth section (lines 46-52) creates the threads. Examining the first thread, `pthread_create(&swing_arm_forward_t, &CBA_pthread_attr2,swing_arm_forward,NULL)`, the first argument passes in the thread `swing_arm_forward_t`, which tells the program the thread ID of the thread it is creating. The second argument, `CBA_pthread_attr2`, tells the program that the thread is joinable. The third argument, `swing_arm_forward`, is the function that will be run in the thread. The fourth argument, `NULL`, tells the program to not pass anything to the function. Note that in the fourth section, the second thread created is joinable, and the third thread created is detached.

4.4 The Three Functions

There are only three functions in this program, `swing_arm_forward()`, `grab_botguy()`, and `swing_arm_back()`. Each function does as described by its name. The crucial part to these functions is the mutex locking that each one does. As can be seen in the code (lines 74-91), each function calls a lock on its associated mutex. Each mutex is already locked before the thread associated with each function is created. As described more thoroughly in the paper *Multiprocessing Using Pthreads and Mutexes*, if a mutex is already locked, and another function tries to lock it, that function will be blocked. Therefore, the function must wait until the mutex is unlocked before it can continue to execute the remainder of its code. This allows for instant execution of the functions by the main program by simply unlocking the mutexes.

4.5 Main Program

Prior to this section, the initializing and setting of the mutexes and Pthreads has not caused the robot to move. The primary section of the program (lines 54-71) actually starts to move the robot. The sections starts out by unlocking the mutex holding the `swing_arm_forward` thread with the command, `pthread_mutex_unlock(&swing_arm_forward)`. As described in section 4.4, this will allow the rest of the code in the `swing_arm_forward()` function to execute. In essence, this kick starts the function. While the robot is swinging its arm forward, it is also moving forward to Botguy using the `go_straight()` command. This is an example of two threads executing at the same time.

The next command, `pthread_join(&swing_arm_forward_t,NULL)`, in line 58, causes the robot to wait till the arm is swung forward before the claw will grab Botguy. Botguy cannot be grabbed without the arm fully forward. This line prevents this from happening. Similarly, the mutex holding the `grab_botguy()` function is unlocked, allowing the function to execute. Again, a `pthread_join()` command is utilized so that the robot won't swing the arm back before completely having a hold on Botguy. Lastly, in line 63, the `swing_arm_back()` function is started, and in the next line, the robot moves forward. Thus, again having two threads execute at the same time.

5.0 Saving Time

In reality, all four threads are running at the same time. However, the three threads holding the functions appear to be dormant because they are waiting for their associated mutexes to be unlocked.

The idea behind this method is to save time. The time it takes for the program to define, create, and initialize mutexes and Pthreads is considerably long. In reality it may take almost a second to run these startup routines. Although one second may not sound like a lot, it is very significant. Botguy can be grabbed or knocked down in less than three seconds. If one second is lost, thirty-three percent of the time it can take to get Botguy is wasted. One second could make the difference in a match. This is why before `wait_for_light()` is called, the creation of all the Pthreads and mutexes are done.

6.0 Naming Convention

As the program shows, CBA created a naming convention to keep the Pthreads and mutexes organized. Each Pthread created has a 't' following its name, and each mutex created has a 'm' following its name. For each thread and mutex created, the name of the associated function is used in naming the thread and mutex. For example, the function `grab_botguy()` uses the Pthread `grab_botguy_t` and the mutex `grab_botguy_m`. This convention helps the programmer analyze the code quickly without getting confused with which thread or mutex is doing what.

7.0 Summary

Pthreads are an effective way to implement parallelism within a program. It allows a program to do two or more functions at any given time. With the example presented, it can be concluded that organization is very important. Without clear organization, a programmer can get lost trying to understand the code. Being time efficient is also key, which is why everything is initialized and created up front before the robot is ready to start. Another key point is to lock the mutexes first in the main thread and have each function lock its related mutex. This allows for easy immediate execution of the function in the main thread by simply unlocking the mutex. Lastly, create a naming convention to easily link each mutex and thread to the function. CBA has done this effectively and it is very important in our programming.

References

[1] Myers, Ethan Y. "Multiprocessing Using Pthreads and Mutexes." Submitted to Proceedings of 2009 Global Conference on Educational Robotics.