CS 374 Lab 4: Threads and Race Conditions

Today’s goals:

* Understand the syntax of pthreads, and how they utilize functions within the code file
* Understand the syntax for implementing a simple semaphore (in this case, a pthread mutex)
* Conceptualize conditions which lead to race conditions

The program at the end of this lab is a draft example of how you might create a threaded application with a mutex used to protect a shared counter variable. Note that this code operates a bit differently than fork() did in the last assignment. While fork() combined with exec() creates a separate **process** to run a separate program, pthread\_create() builds a new path of execution that is scheduled like a process to run a **function** declared within this program. The new thread itself is part of the same process, so it can see global variables for the program.

There may be variant differences between threads on our system and this older piece of code. Debug it first, then work on the assignment part. You will probably also want to consult the POSIX threads (Chapter 12) portion of the Matthew and Stones text.

Note that this piece of code uses a mutex to ensure mutual exclusion. **Your job** is to remove this and tinker with the code to ensure that it *generates many race conditions on the counter variable*. This may not be as easy as you think! For example, you may have to copy the counter variable to a local function variable, increment it, sleep a tiny bit, and then copy it back to counter in order to ensure that changes from another thread are overwritten. You might vary the sleep time slightly with a random number (remember not to sleep too long, you want your program to run quickly enough for test runs!). You may have to run many more iterations of the function call, or put the counter increment into a loop, or generate more threads (don’t go overboard with this). You will need to know exactly how many increments are supposed to occur and be able to quickly show where they are in your code. The key to success here is to clearly model in your mind how race conditions occur.

Compilation of threaded code which utilizes the pthread library requires using a library loading option on the end of the compilation line in addition to the #include, much as with the math library:

* gcc –o runout myprog.c -lpthread

To complete the lab, show your instructor a run illustrating that the wrong number of increments occurred! This means you will show me both how many increments should occur, and what the final value from the run was. You should be able to get half of the increments to not occur with relative ease once your group is clear on what you are doing.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int counter = 0; /\* Note: global variable! \*/

void \*functionC();

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

int main()

{

int rc1, rc2;

pthread\_t thread1, thread2;

/\* Create independent threads each to execute functionC \*/

if( (rc1=pthread\_create( &thread1, NULL, functionC, NULL)) )

{

printf("Thread creation failed: %d\n", rc1);

}

if( (rc2=pthread\_create( &thread2, NULL, functionC, NULL)) )

{

printf("Thread creation failed: %d\n", rc2);

}

/\* Wait till threads are complete before main continues. Unless \*/

/\* we wait we risk executing an exit( ) which will terminate \*/

/\* the process and all threads before the threads have completed. \*/

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

exit(0);

}

void \*functionC()

{

pthread\_mutex\_lock( &mutex1 );

counter++;

printf("Counter value: %d\n",counter);

pthread\_mutex\_unlock( &mutex1 );

}