CS 0449 – pthreads Lab

Make a program with two threads. *Note*: Remember to compile your program with the -pthread option.

First, define a global variable `int x`.

In the first thread, increment x in a loop and print its value in the end:

```
int i;
for(i=0; i < 100000; i++) {
   x++;
}
printf("Thread 1: %d\n", x);
```

In the second thread, do the same thing except to print Thread 2 in the end:

```
int i;
for(i=0; i < 100000; i++) {
   x++;
}
printf("Thread 2: %d\n", x);
```

Join the two threads in the end by using `pthread_join()` so that each thread has a chance to print its output before exiting the program.

Run the program, and observe the output.

1.) What value do you expect for `x` at the end of the program?

2.) What value actually does get printed for `x` at the end of the program?

3.) Is the value that gets printed at the end deterministic (always the same)?

The above seemingly inexplicable result is due to a phenomenon called *data races*. A data race, loosely defined, is a simultaneous access to a shared (global) variable by two threads where at least one is a write. In the example, the seemingly atomic increment of x (x++) is actually compiled to a sequence of *read*, *add*, and *store* operations by the compiler. These operations can interleave in unexpected ways between two threads causing increments to seemingly disappear. The C standard says the behavior of any program with a data race is *undefined*. It is the responsibility of the programmer to write a program with no data races.

Try using valgrind to detect the data races. You have to give the –g option when compiling (for debug symbols) and also give the –tool=helgrind option to valgrind (which enables the data race detection component of valgrind) as such:

```
gcc -pthread -g ./main.c
valgrind --tool=helgrind ./a.out
```

Valgrind should output the exact source code location where the data race occurred for each thread.

Modify the program you have written to remove the data race using a *lock*. The section of code that is protected by a lock is called a *critical section*. Refer to the following Wikipedia entry for an explanation of critical sections:

http://en.wikipedia.org/wiki/Critical_section

In the above link, there is a code example on how to use pthread locks using pthread_mutex_lock() and pthread_mutex_unlock() functions. Use this method to correctly lock the increment of x and also the printf of x. The example uses the macro PTHREAD_RECURSIVE_MUTEX_INITIALIZER to initialize the lock which is not POSIX compliant and therefore not portable. You will need to change it to the POSIX compliant PTHREAD_MUTEX_INITIALIZER for it to work on thot.

Now your program should correctly print the expected value at the end of execution. Also, running with valgrind should output no errors.

Submission

Show the TA the answers that you've discovered and your modified program correctly executing.