CS 0449 – Pthreads Lab

Introduction

A thread is similar to a process in that it is a container for a task that can be concurrently run on a CPU with other tasks. As such, it is often used for a similar purpose: to speed up program execution. However, there is a key difference: whereas a process *clones* all of its state on a fork() call, a thread *shares* most of its state with other threads. When a thread is created, it is given a function to execute in parallel with its parent thread. The new thread is given separate stack space to store the function frame of the given function (and its callee functions), but otherwise it shares the same (virtual) memory space as the parent thread. In fact, the parent and child threads become part of the same process. That means all static variables in the data segment are shared and also all objects in heap memory are shared. We will study the implications of threads and sharing in this lab using Pthreads (POSIX threads), a standard API for threading.

Part 1: Basic Thread Creation

Copy `~wahn/public/cs449/pthread.c` to your local directory. Right now the program is a single threaded program, and when compiled and run, displays the following output:

```
local=10000000, global=50000
local=10000000, global=100000
```

We are going to modify the above program such that when we compile it with the –pthread GCC option like the following, we get a multi-threaded version of the same program:

```
gcc -pthread ./pthread.c -o ./pthread_multi
```

The –pthread option tells GCC that this program uses the Pthreads API. One of the things it does is implicitly define the _REENTRANT macro, which is used to enable / disable certain code in the standard header files pertaining to multi-threading. We are going to use the same macro to enable / disable multi-threading inside pthread.c. Your job is to fill in the space between `#ifdef _REENTRANT` and `#endif` to enable multi-threading when _REENTRANT is defined. The goal is to have the two calls to function foo() run in parallel in two threads instead of sequentially. To achieve this, you are going to do the following:

1.) Define two variables of type `pthread_t` to hold two thread IDs.
2.) Create two threads that execute function foo() using `pthread_create(),` storing the two thread IDs in the above two variables.
3.) Wait for the two threads to complete by calling `pthread_join()` on the two thread IDs before returning from main(). Note that if you do not call `pthread_join()`, then the main thread may return from the `main()` function and terminate the program before the two child threads have had a chance to complete foo() and print the output.

Refer to the slide example code on hints on how to do this. Also try doing 'man pthread_create' and 'man pthread_join' to read the usage of these functions.

If you are successful, you should get output that looks (approximately) like the following:

```
local=10000000, global=94414
local=10000000, global=94433
```

Note that the value of global is different from the sequential version and it even changes from run to run! This is due to a nondeterministic bug caused by data sharing called a *data race*. This bug will be the subject of Part 2. For now, let's see whether multi-threading has actually improved the performance of the program by using the UNIX `time` utility. First, let's try measuring the runtime of the default sequential version of the program:

```
/usr/bin/time -v ./pthread_single
```

This command should give an output similar to the following:

```
...
User time (seconds): 0.11
System time (seconds): 0.00
Percent of CPU this job got: 99%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.11
...
```

This shows that this program ran for a total of 0.11 seconds on the CPU(s) of this machine. Also, 0.11 seconds have elapsed in wall clock time to run this program. During this elapsed the program got 99% of one CPU.

Now let's try measuring the runtime of the multi-threaded version:

```
/usr/bin/time -v ./pthread_multi
```

This command should give an output similar to the following:

```
...
User time (seconds): 0.11
System time (seconds): 0.00
Percent of CPU this job got: 198%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.06
...
```

This shows that the program ran for a total of 0.11 seconds on the CPU(s), similar to the sequential version. The big difference is in the wall clock time that has been almost halved to 0.06 seconds. This was possible because the program got 198% of one CPU (or two CPUs) during that elapsed time. **Show the TA the results of running `pthread_multi` with `/usr/bin/time`.**

Part 2: Data Races and Synchronization

The seemingly inexplicable value of `global` for the multi-threaded version is due to a phenomenon called *data race*. Data races affect only shared memory locations and since the variable `local` is an automatic variable and hence resides on the stack of the child thread (not shared), `local` is unaffected. A data race occurs because of an illegal interleaving of operations between two threads that would never occur in a sequential implementation. So how could the operation `global++` interleave in such a way to produce this result?

The key to understanding the problem is that the seemingly monolithic `global++` increment operation is actually compiled to a sequence of 3 machine instructions by the compiler: *read from memory*, *add*, and *write to memory*. These operations can interleave in unexpected ways between two threads causing increments to seemingly disappear. Try thinking of an interleaving like the following (sequence of events numbered from ① to ⑥):

Initially, global == 0	
[Thread 1]	[Thread 2]
① $EAX 🡐 global ② $EAX 🡐 $EAX + 1 ⑤ global 🡐 $EAX	③ $EAX 🡐 global ④ $EAX 🡐 $EAX + 1 ⑥ global 🡐 $EAX
Now, global == 1	

In the example, Thread 2 reads the stale value of `global` before Thread 1 has had a chance to update its value. So, even when the increment happened twice logically, `global` is still 1 after the sequence has finished. This may or may not happen depending on the speed of Thread 1 and Thread 2. Hence, the C standard says the behavior of any program with a data race is *undefined*. It is this nondeterministic interleaving that caused the wrong value of `global`.

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

```
gcc -pthread ./pthread.c -o ./pthread_multi -g
valgrind --tool=helgrind ./pthread_multi
```

Valgrind should output the exact source code locations where data races occurred for each thread. When interpreting the output, consider the following.:

1. A data race can occur when there is a read memory access and write memory access that can 'race' with each other. Depending on who finishes first, the read memory access will return a different value, leading to a non-deterministic result, which is usually a sign of a bug.
2. By the same token, a data race can also occur when there is a pair of racing write memory accesses. Depending on who finishes first, the memory location will end up with a different value.
3. A data race cannot occur for a pair of racing read memory accesses since no matter who finishes first, both read accesses will return the same value.

Note: valgrind will find one more data race besides the one described above.

Read the following URL if you are more interested in how valgrind detects data races:
http://valgrind.org/docs/manual/hg-manual.html#hg-manual.data-races

Modify the program you have written to remove the data race using a *lock* or *mutex*. The section of code that is protected by a lock is called a *critical section*. A mutex guarantees mutual exclusion between two critical sections that use the same mutex, meaning no interleaving between the two critical sections is allowed. The two critical sections will always execute one after another and never in parallel. Refer to the slides for an example. As in the slides, use a `pthread_mutex_t` mutex to correctly place a critical section around the increment of `global` so that no races can occur. You also need to put a critical section around the `printf` since it reads `global`.

Note that you have a choice between placing a critical section around the entire loop or just the increment of `global`. Both will remove the race and make your program deterministic and correct. As a general rule, making critical sections as small as possible while preserving correctness (usually) results in a more concurrent and faster program. Hence, expert programmers expend great effort into minimizing critical sections while keeping them correct. If you are curious, you can try creating both versions and measure the runtime of each using `/usr/bin/time`. Locking the entire loop does not improve the wall clock time compared to the original sequential version, as expected.

Now your program should correctly print the expected value at the end of execution. Also, running with valgrind should output no errors. Valgrind may take a while to finish verifying so have patience.

Submission

Remove all files except pthread.c. Assuming you are working under the directory lab6/,

```
cd ..
tar zcvf USERNAME_lab6.tar.gz lab6
cp USERNAME_lab6.tar.gz ~wahn/submit/449/RECITATION_CLASS_NUMBER
```