Lab 8: Threads (8.3.16)

Do Exercise II(a) during the lab. Your solutions to the remaining exercises will be checked off in the week following the mid-semester break, i.e. week of March 28.

In pthreads, new threads are created by calling the function `pthread_create`. The actual definition of `pthread_create` is quite verbose:

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

Let's start at the beginning. `pthread_create` returns an `int`, which is used to determine if a thread was successfully created (0 for yes, non-zero for no).

- The first argument is a pointer to a `pthread_t` structure. This structure holds all kinds of useful information about a thread (mostly used by the threading internals). So, for each thread you create, you'll need to allocate one of these structures to store all of its information.
- The second argument is a pointer to a `pthread_attr_t struct`. This structure contains the attributes of the thread. This has even more information about the thread. The only really interesting thing, for us at least, is that the `pthread_attr_t` struct tells the system how much memory to allocate for the thread's stack. By default, this is usually 512K. This is actually quite large, and if you run with lots of threads you can run out of memory just from allocating thread stacks.
- The third argument to `pthread_create` is even more mysterious. Lets look at it again: `void *(*start_routine)(void *)`. This is actually the function that the thread should start executing. This argument is an instance of what C programmers call a function pointer. The name of the argument is actually `start_routine`, and it is a function that returns `void *` and takes a `void *` as an argument.
- The last argument to `pthread_create` is a `void *` which is the argument to be passed to `start_routine` when the thread gets going. So, why a `void *`? Because in C, `void *` basically means anything. Remember you can cast pointers to `void *`, but you can also cast `int`s, `char`s, etc. to a `void *`. Which means that it is really the only way to specify a generic argument in C.

Part I. Race conditions

You need to complete the implementation of `ptcount.c`. When you are finished, the main process of `ptcount` should create three pthreads and wait for these pthreads to complete execution. Each thread should increment (in a loop) a shared variable named count as well as a local counter. When the child threads are finished executing, the main thread should print out the value of count. The value reported by the main process should be consistent with what you would expect with the given loop bound and increment values. Your program should match the following output:

```
bash$ ./ptcount 100000 1
Thread: 0 finished. Counted: 100000
Thread: 1 finished. Counted: 100000
Thread: 2 finished. Counted: 100000
Main(): Waited on 3 threads. Final value of count = 300000.
Done.
```

After you have finished your implementation, answer the following questions:

1. What accounts for the inconsistency of the final value of the `count` variable compared to the sum of the local counts for each thread in the version of your program that has no lock/unlock calls?
2. If you test the version of your program that has no lock/unlock operations with a smaller loop bound, there is often no inconsistency in the value of `count` compared to when you use a larger loop bound. Why?
3. Why are the local variables that are printed out always consistent?
4. How does your solution ensure the final value of `count` will always be consistent (with any loop bound and increment values)?
5. Consider the two versions of your `ptcount.c` code. One with the lock and unlock operations, and one without. Run both with a loop count of 1 million, using the time time command: "`bash> time ./ptcount 1000000 1`". Real time is total time. User time is time spent in User Mode. SYS time is time spent in OS mode. User and SYS time will not add up to Real time for various reasons that need not concern you at this time. Why do you think the times for the two version of the program are so different?

Part II. Worker threads

a. Write a program that will take a list of filenames on the command line. If the list is empty, print an error message and exit. The goal is to compute how many lines are in each file using a thread for each file.

   Create a thread for each argument given on the command line. Each thread should open its filename. If the open succeeds, the thread should read the file and count the number of newlines. When EOF is reached, the thread should print its filename with the number of newlines in the file, and terminate the message with a newline. The thread can then exit.

b. Write a program that will take a list of filenames on the command line. If the list is empty, print an error message and exit. As in part a) assign a thread to each filename and have it count the number of newlines in its file. But in this part have the threads coordinate to report a single global sum of how many newlines there are in total in all files. Your solution should use a mutex to protect threads adding their local count to the global count. Also have the main thread wait for the global count to be complete, then have the main thread print the global count.