Recitation 6: Pthreads API

So far, you have been introduced to Cilk as a way to parallelize programs. Cilk is high-level in the sense that the Cilk runtime takes care of scheduling parallel work and work stealing. You just have to give it hints (e.g., cilk_for) as to where parallelism can occur. This recitation provides an overview of programming with Pthreads, a “low-level” thread framework that gives you finer control over thread behavior. Although you are unlikely to need this material for Project 2, you might find these tools useful in coming weeks.

1 Getting started

We recommend that you work on the course machines.

$ ssh <username>@cloud

Use Git to clone a local copy of the repository for your work:

$ git clone /afs/csail.mit.edu/proj/courses/6.172/student-repos/fa14/recitations/\> recitation6/<username>.git recitation6

2 Overview of Pthreads

The IEEE POSIX 1003.1c standard (1995) offers a standardized programming interface for threads on UNIX systems. Thread implementations following this standard are referred to as POSIX threads or Pthreads. Whereas Cilk offers high-level linguistics to expose parallelism in programs, Pthreads offer a raw interface to threads. Although this interface can be powerful, this interface also requires the programmer to explicitly delegate work to threads, making Pthreads complex to code.

As we shall see, Pthread libraries offer a set of C language types and procedure calls. All threads within a process share the same address space and have access to the global shared memory. Threads can also have their own private data. The program needs to synchronize thread access to shared global data to avoid creating race conditions. The ability of a function or procedure to let multiple threads execute it without creating races is called “thread safety.”

3 Pthreads API

Interfaces in the Pthreads API can be grouped into four categories:

1. Thread management: Routines to create, detach or join threads.

2. Mutex (Section 5): Routines to create, destroy, lock and unlock mutexes.
3. Condition variables (Section 6): Routines to handle communication among threads that share a mutex. These routines include routines to create, destroy, wait and signal based on specified variable values.

4. Synchronization: Routines to manage read/write locks and barriers.

All identifiers in the Pthreads library start with the prefix `pthread_`. Programs that use Pthreads must include `pthread.h` header file and must be linked with the Pthreads library when the compiler is invoked (e.g. using the `-pthread` flag with GCC).

## 4 Creating Pthreads

The `pthread_create()` call creates a new thread and starts it running. This function is analogous to `cilk_spawn`. The signature of the function is as follows:

```c
int pthread_create(pthread_t *restrict newthread, const pthread_attr_t *restrict attr,
                   void *(*start_function)(void *), void *restrict arg);
```

The arguments are:

- **newthread**: An opaque, unique identifier for the new thread returned by the subroutine.
- **attr**: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- **start_function**: A pointer to the C function that the thread will execute once it is created.
- **arg**: A single argument (which could be a struct) that may be passed to start_function. It must be passed as a pointer cast of type void. NULL may be used if no argument is to be passed.

If successful, the `pthread_create()` function returns zero; otherwise, it returns an number indicating the error.

The function `pthread_exit()` has the declaration

```c
void pthread_exit(void *value_ptr);
```

The `pthread_exit()` function shall terminate the calling thread and make the value `value_ptr` available to any successful join with the terminating thread. The `main()` function also calls `pthread_exit()` so that it blocks until all the threads that were spawned by the main function complete their execution.

**Exercise 1**: Compile and run `sample.c`. This code is a simple program that demonstrates thread creation and execution. You can make and run this code as follows (you don’t need to use Lanka):

```
$ cd sample
$ make sample
$ ./sample
```
5 Mutexes

Mutexes are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. A mutex (also called a lock) is a variable that is held by one thread at a time. A thread can call a special function `pthread_mutex_lock()` that tries to acquire the mutex. If no other thread owns the mutex when `pthread_mutex_lock()` is called, then the calling thread gets the mutex and continues execution. Otherwise, the thread pauses at the `pthread_mutex_lock()` call and only resumes when the mutex is released by another thread.\(^1\) This allows threads to “take turns” running the code after the `pthread_mutex_lock()` call. If a shared global variable gets modified after the lock call, the mutex will prevent a data race between threads.

**Exercise 2:** Compile and run `race.c`:

```bash
$ cd race
$ make test
```

This program demonstrates thread races. Identify the race in the code.

Let’s use a mutex to fix the race in `race.c`. A typical sequence in the use of a mutex is as follows:

- The program creates and initializes a mutex variable.
- Several threads attempt to lock the mutex.
- Only one thread succeeds, and that thread owns the mutex.
- The owner thread performs some set of actions.
- The owner unlocks the mutex.
- Another thread acquires the mutex and repeats the process.
- Finally, the mutex is destroyed.

The functions to create/destroy a mutex and lock/unlock a mutex are:

```c
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                     const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutex variables have the type `pthread_mutex_t`. The `pthread_mutex_init()` function initializes the mutex with attributes specified by `attr`. If `attr` is `NULL`, then the default mutex attributes are used. If successful, all the function listed above return zero; otherwise, they return a number indicating the error.

**Exercise 3:** Using a mutex, fix the race in `race.c` and verify you no longer have data races.

\(^1\) A buggy program might have a thread acquire a mutex without releasing it. This is a type of deadlock and can cause the program to hang. Likewise, if a Cilk strand never completes, execution will get stuck at the `cilk_sync`. 
6 Condition Variables

Condition variables provide another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon actual data values. Without condition variables, the programmer would need to have threads continually polling the value to check if the condition is met. This can consume a lot of resources, since the thread would be continuously busy performing this activity. A condition variable is a way to achieve the same goal without polling. A condition variable is always used in conjunction with a mutex lock.

The following functions operate on condition variables:

```c
int pthread_cond_wait(pthread_cond_t * restrict cond,
                      pthread_mutex_t * restrict mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

The `pthread_cond_wait()` function blocks the calling thread until the specified condition is signaled. This function should be called while the mutex is locked, and will automatically release the mutex while it waits. After a signal is received, the thread is awakened and `pthread_cond_wait()` returns, the mutex will be automatically locked for use by the thread and must be unlocked before another thread may proceed.

The `pthread_cond_signal()` function is used to signal (or wake up) another thread which is waiting on the condition variable. It behaves most predictably when the signaling thread holds the mutex and immediately unlocks the mutex in order for a woken process’ `pthread_cond_wait()` function to complete. As a note, `pthread_cond_signal()` wakes up at least one thread, but possibly can wake up more than one thread, causing the woken threads to contend for the mutex. If multiple wakeup is desired, `pthread_cond_broadcast()` wakes up all threads waiting on the condition variable.

Exercise 4: The `cond.c` program demonstrates code that would benefit from use of condition variables. In `cond.c`, thread 1 running `watch_count` keeps polling until count reaches `COUNT_LIMIT` and then increments count by 125. However it doesn’t get to increment count always. Fix this bug by adding `pthread_cond_wait()` and `pthread_cond_signal()` calls at the appropriate places so the thread that executes `watch_count` doesn’t need to keep polling and always gets its chance to increment count by 125 once count reaches `COUNT_LIMIT`. You can make and run this program as follows:

```
$ cd cond
$ make test
```

Does your fix solve the problem?

7 Barriers

Barriers are a method of synchronizing a set of threads at some point in time by having all participating threads in the barrier wait until all threads have called the said barrier function.
This is analogous to \texttt{cilk\_sync} and, in essence, blocks all threads participating in the barrier until the slowest participating thread reaches the barrier call. For example, say we have a group of threads that need to execute some task X and then, only after every thread finishes X, execute some task Y. This can be accomplished by adding a barrier in between tasks X and Y.

A barrier can be initialized with:

\begin{verbatim}
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
const pthread_barrierattr_t *restrict attr,
unsigned int count);
\end{verbatim}

The function that performs the synchronization is:

\begin{verbatim}
int pthread_barrier_wait(pthread_barrier_t *barrier);
\end{verbatim}

which causes the calling thread to block until a total of \texttt{count} threads (including itself) have reached the barrier point. Finally, the barrier must be cleaned up with:

\begin{verbatim}
int pthread_barrier_destroy(pthread_barrier_t *barrier);
\end{verbatim}

8 The \texttt{__thread} storage class specifier

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per thread. This means that, in a multithreaded application, a unique instance of the variable is created for each thread that uses it and destroyed when the thread terminates. The \texttt{__thread} storage class specifier can provide a convenient way of assuring thread-safety; declaring an object as per-thread allows multiple threads to access the object without the concern of race conditions, while avoiding the need for low-level programming of thread synchronization or significant program restructuring.

Here is an example of how thread-local variables can be declared:

\begin{verbatim}
__thread int i;
extern __thread struct state s;
static __thread char *p;
\end{verbatim}

The \texttt{__thread} specifier may be used alone or with the \texttt{extern} or \texttt{static} specifiers, but not with any other storage class specifier. When used with \texttt{extern} or \texttt{static}, \texttt{__thread} must appear immediately after the other storage class specifier.

(In C++, the \texttt{__thread} specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data members.)

Exercise 5: In \texttt{cond.c}, change \texttt{mycount} to be a global variable. Verify that your program is now buggy. Change \texttt{mycount} to be a thread-local variable using \texttt{__thread} and verify that the bug disappears.