Recitation 6: Programming with Pthreads

This recitation provides an overview of programming with Pthreads.

1 Getting started

We recommend that you work on course machines. You can access the machines using ssh

```
% ssh username@cloudN.csail.mit.edu
```

To get a local copy of the repository for your work, you need to use git to clone it.

```
% git clone /afs/csail.mit.edu/proj/courses/6.172/
student-repos/recitation6/username recitation6
```

2 Overview of Pthreads

IEEE POSIX 1003.1c standard (1995) offers a standardized programming interface for threads on UNIX systems. Thread Implementations that follow this standard are called POSIX threads or Pthreads. Pthread libraries offer a set of C language types and procedure calls. Pthreads offer a raw interface to threads and hence can be powerful and complex to code. All threads within a process share the same address space and have access to the global shared memory. Besides, threads can also have their own private data. The program needs to synchronize thread access to the 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

Pthreads API can be grouped into four categories.

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

2. Mutex: Routines to create, destroy, lock and unlock mutexes.

3. Condition variables: Routines to handle communication among threads that share a mutex. This includes 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.

- Exercise: Compile and run a simple program sample.cpp that demonstrates thread creation and execution. To make and run, use the following command.

```
% make sample
% ./sample
```
4 Creating Pthreads

The routine `pthread_create` creates a new thread and makes it executable. The declaration for the routine looks as follows:

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

The arguments are:

- `thread`: 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_routine`: the C routine that the thread will execute once it is created.
- `arg`: A single argument that may be passed to `start_routine`. It must be passed by reference as a pointer cast of type `void`. NULL may be used if no argument is to be passed.

If successful, the `pthread_create` function shall return zero; otherwise, an error number shall be returned to indicate the error. The routine `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. Note that 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.

5 Mutex

Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. Only one thread can lock (or own) a mutex variable at any given time. Other threads must wait for the owner to relinquish the lock and then compete to acquire the lock. Threads thus “take turns” accessing protected data.

- **Exercise**: Compile and run a simple program `race.cpp` that demonstrates thread races. Identify the race in the code. To make and run, use the following command.

  ```bash
  % make test
  ```

Let us use Mutex to fix the race in the above program. A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one 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 routines 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 shall initialize the mutex with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used. If successful, all the routines listed above shall return zero; otherwise, an error number shall be returned to indicate the error.

• **Exercise:** Using mutex, fix the race in `race.cpp`. To make and run, use the following command.

```
% make test
```

### 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 the actual value of data. Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in 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 `pthread_cond_wait` routine blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread.

The `pthread_cond_signal` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait` routine to complete.

The signature of these functions is as follows:

```c
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```
• Exercise: cond.cpp demonstrates the use of condition variables. In the program, 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 such that thread 1 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. Add pthread_cond_wait and pthread_cond_signal calls at the appropriate places in the code. To make and run, use the following command.

% make test

Does your fix solve the problem?

7 Barriers

Barriers are a method to synchronize 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, in essence, blocks all threads participating in the barrier until the slowest participating thread reaches the barrier call. For example, if a group of threads need to execute some task X, wait until all threads complete task X and then execute task Y, barrier can be added in between tasks X and Y.

8 The __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 extant thread. This means that, in a multi-threaded application, a unique instance of the variable is created for each thread that uses it, and destroyed when the thread terminates. The __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. For example, thread-local variables can be declared as follows:

__thread int i;
extern __thread struct state s;
static __thread char *p;

The __thread specifier may be used alone, with the extern or static specifiers, but with no other storage class specifier. When used with extern or static, __thread must appear immediately after the other storage class specifier.

The __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 member.