User Level Thread Library

Operating Systems Theory: CS 519
October 30, 2011
Due by Midnight, Sunday, Nov 13, 2011

Problem Statement

Part-A: Multithreaded sorting problem

Write a multithreaded program using the standard thread library to implement the following sorting algorithm. This is to help you understand multithreaded program before you go ahead and design the thread library. Consider an algorithm for sorting an array of n+1 elements using n threads. As shown in the figure, each thread is responsible for two adjacent elements of the array. For sorting, each thread must compare the two elements it is responsible for, and swap them if required. For example for thread Ti, if a[i + 1] is larger than a[i], they must be swapped.

Threads execute the sorting function in parallel. Since elements (1, 2, 3,..., n-1) are shared between two threads, there must be mutual exclusion when the array is updated. For this, assume an array of mutexes of size n+1. The thread must have exclusive access to both elements (a[i] and a[i+1]) to perform the swapping operation.

Another thread should perform the job of checking all the elements in the array at some specified intervals of time to check if the array is sorted. If the array is sorted, the program should terminate.

Part-B: Basic Thread Library

1. Write a non-preemptive user-level thread library similar to the Linux pthreads library implementing the following calls. Please prefix your function names with "my_" to avoid conflicts with the standard library

   - pthread_create
In this case, the thread yields control of the processor when one of the following conditions is true:

- thread exits by calling `pthread_exit`
- thread explicitly yields by calling `pthread_yield`
- thread waits for another thread to terminate by calling `pthread_join`

2. **Write a preemptive thread library that implements the above calls and uses round robin scheduling.**

Preemption can be achieved by using timer and signals. Threads now relinquish control of the processor based on all of the above conditions mentioned for the non-preemptive library as well when the timer expires. Use the `makecontext`, `getcontext`, `setcontext` and the `setitimer` calls to implement the scheduler. To avoid race conditions, do preemption only when the control is in the user application.

**Part-C: Synchronization and Threads**

Implement the following synchronization primitives in your thread library. Please refer to the section below for a brief summary of each of these calls.

- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutex_destroy`
- `pthread_cond_init`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- `pthread_cond_wait`
- `pthread_cond_destroy`

**Part-D: Implement non-blocking calls (extra credit)**

Avoid blocking calls in your thread library by using asynchronous read operations. Replace the read calls with `my_read` function that performs asynchronous I/O. Whenever the thread tries to read a file from disk, it should not block on the read operation. Instead, the thread makes a call to the asynchronous read function and sleeps. The thread will be woken up on receiving the signal for I/O completion.
Summary of the pthreads API

For more details on the API, refer to the pthread library API.

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

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function start_routine passing it arg as first argument. The new thread terminates either explicitly, by calling pthread_exit(), or implicitly by returning from the start_routine function. The latter case is equivalent to calling pthread_exit() with the result returned by start_routine as exit code. You can ignore that attribute values in this call.

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

The pthread_exit() function terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread. Remember that a thread can terminate implicitly without calling pthread_exit() function.

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

The pthread_join() function suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated.

```c
pthread_yield();
```

This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads.

```c
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

The pthread_mutex_init() function initializes the mutex referenced by mutex. You can ignore the attributes here in your implementation.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

A mutex is a mutual Exclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors. A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first. pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and pthread_mutex_lock returns immediately. If the mutex is unlocked, the calling thread blocks.
int pthread_mutex_unlock(pthread_mutex_t *mutex);

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to pthread_mutex_unlock.

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance.

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

A condition (short for “condition variable”) is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition. A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. pthread_cond_init initializes the condition variable cond. You can ignore the attributes in the call above.

int pthread_cond_signal(pthread_cond_t *cond);

pthread_cond_signal restarts one of the threads that are waiting on the condition variable cond. If no threads are waiting on cond, nothing happens. If several threads are waiting on cond, exactly one is restarted, but it is not specified which.

int pthread_cond_broadcast(pthread_cond_t *cond);

pthread_cond_broadcast restarts all the threads that are waiting on the condition variable cond. Nothing happens if no threads are waiting on the time it waits on the condition variable.

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

The pthread_cond_wait() function atomically unlocks the mutex argument and waits on the cond argument.

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_destroy destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to pthread_cond_destroy.

**Posting Instructions**
The file should contain the following information in this order:

Your name
Parts of the project you have done in the following format:
Part A: Done
Part B:
  1. Done
  2. Not Done
  3. Done
And so on.

Explain the working of your code on a function by function basis. Please keep the explanation short and meaningful.

Code:

- pthread.c: Code for your thread library
- sorting.c: Program for Part-A, the sorting problem.

All other header files used by your code, if any have to be included.

Create an archive of all the above files and upload it on Sakai.

Creating the static or shared library

This link is a tutorial for creating a static or shared library. For your purpose, you can only create the static library.