Assignment 4

A Preemptive User-Level Thread Library (v 1.1)

Max Points: 35/45 (see below)

Due date: Thursday, March 7 (11.59pm).

Overview:

In this last project, you will start with a skeleton of a preemptive user-level thread library under the SOLARIS operating system. The skeleton library contains a set of basic thread management functions. You will add several functions to the existing skeleton to complete a preemptive user-level thread.

The library will consist of a set of functions that allow management of threads in user space (i.e., the operating system is not aware of your threads) according to a many-to-one model. These functions will allow the creation, termination, and synchronization of threads, as well as setting the scheduling policy.

Description of the Thread Library Skeleton:

You will be provided with 3 headstart files: `threadlib.c`, `test.c`, and a `Makefile`. The `threadlib.c` file contains the skeleton management functions. The design of this library is based on a user-defined timer that periodically sends signals to the process, and a signal handler that implements a simple thread scheduler.

Here is a description of the core library functions:

- Creation of a user-level thread:

```
thread * create_thread(void (* thread_function_t )(void *),
                       void* parameter,
                       int priority)
```

  This routine creates a user-level thread that will start running a function (specified as a function pointer). The function pointer is passed as a single parameter. The thread creation function also requires a thread priority scheduling parameter (see below). The library maintains a table of data structures called thread control block (TCB), one for each thread in your application. Whenever a user-level thread is created, you will need to add an entry in this table. The `create_thread` function returns a pointer to the TCB structure of this thread (which can be used as an identifier). Note that this exposes internal thread library data structures to the application and is an example of a VERY BAD API design.

- Termination of a user-level thread:

```
void thread_exit()
```

Currently, all threads are created as "joinable:, i.e., several threads may be executing a (potentially) blocking `thread_join` call (see below) to wait for another thread to exit. Therefore, each TCB entry will contain a list of threads that are blocked waiting to join a thread. When a thread exits, all the "joining" threads on this list will be unblocked. After exiting, the thread will change its state to "Zombie". Note that you have to be careful about removing joinable threads from the TCB! To illustrate this, suppose a thread 1 wants to join another thread 2. If thread 1 executes the `thread_join` call after thread 2 has bee removed, thread 1 could be erroneously waiting for the wrong thread (or, worse, the thread * pointer could point to invalid memory!). It is possible that a thread tries to join another thread after it has changed its state to "Zombie" (in which case the join call would be non-blocking!). Note that, unlike Posix threads, this function does not allow to return an exit status to the waiting thread.

- Waiting for a thread:

```
int thread_join(thread* thread_ptr)
```

Blocks the calling thread until the thread identified by the argument exits. The function returns immediately if either no such thread exits or the thread was already terminated.

- Yield the processor:

```
void thread_yield()
```

This function causes the current thread to yield the processor. It results in a call to the scheduler to schedule another thread (if any).

- Set the priority of the calling thread:

```
int thread_set_priority( thread* t, int pri)
```

The library currently implements a weighted-priority round robin scheduler. This scheduler works as follows. A time-quantum is the maximum time period a thread is allowed to run before the scheduler has a chance to preempt the thread and switch to another one. This interval is determined by the length of the interval for the periodic invocation of your timer (see `setitimer` function in your `threadlib.c` headstart file). Each thread in your process is associated with a default priority ranging from 1 to 10. This priority indicates the number of successive time-slices that a thread may run before it will be forcibly removed from the CPU. For example, if your timer interval is 0.5 seconds and the static priority of a thread is 5 then the thread is allowed to run at most 2.5 seconds (=5 time-slices) before the scheduler has a chance to remove it from the CPU and switch to another ready thread (Note that the thread could deliberately relinquish control earlier using '`thread_yield()`'). .

The `thread_set_priority` function allows changing the priority of a thread (initially set in the `thread_create()` call).

- Synchronization primitives:

```
void lock()    and   void unlock()
```

These functions are provided in the headstart files. They are used to ensure that a certain sequence of instructions (i.e., a sequence of code that accesses a shared data structure) can be executed atomically. You will also need these routines to ensure mutual exclusion inside your library. Essentially, lock() is implemented by "turning off" and unlock() by "turning on" the alarm signal (see headstart file).

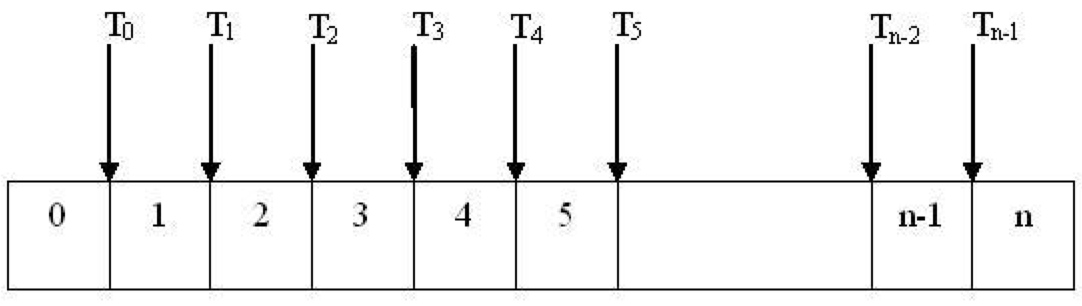
The test.c file demonstrates the use of the skeleton library functions.

Requirements:

1. Write a multithreaded program using the POSIX 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.

   For additional information on the POSIX thread API, refer, e.g., to

   - Lecture slides
   - POSIX thread library man pages
   - http://www.llnl.gov/computing/tutorials/pthreads/



2. Use the headstart files describe above. Feel free to reuse or remove any of the functions and data structures contained in the skeleton library. Using the exising code, extend the thread library in the following ways:

    a. Implement the following synchronization functions:

    ```
    int mutex_lock(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. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `mutex_lock` returns immediately. If the mutex is already locked, the calling thread blocks. Note that the skeleton library contains two functions, `lock()` and `unlock()`, that are based on blocking all current signals. These functions are used to deactivate the timer during critical sections inside the thread library. Using them to provide mutual exclusion for user-level threads would ***not be a good design choice!*** (a user thread might monopolize the CPU indefinitely) You can use the `lock()` and `unlock()` function as a basis, but you will need to find a more effective way to implement locks.

    ```
    int mutex_unlock(mutex_t *mutex);
    ```

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

    ```
    int mutex_destroy(mutex_t *mutex);
    ```

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

    Depending on your mutex implementation, you will also need a function (or a macro) that allows to statically (or dynamically) initialize a mutex.

    ```
    int mutex_initialize(mutex_t *mutexptr);
    ```

    Initializes the `mutex` represented by `mutexptr`.

    Example: Two threads (thread_a and thread_b) perform the parallel sorting on an array with 3 elements:

```
int global_array[3]={7,8,3};
mutex_t mutex_1; // Lock to protect global_array[1]

/* thread a swaps elements 0 and 1, if necessary */
thread_a(void * a)
{
   int temp;
   mutex_lock(mutex_1); //will block if already acquired
   if(global_array[0]>global_array[1]){
         temp = global_array[1];
         gloabal_array[1] = global_array[0];
         global_array[0] = temp;

   }
   mutex_unlock(mutex_1);
}
/* thread a swaps elements 1 and 2, if necessary */
thread_b(void * a)
{
   int temp;
   mutex_lock(mutex_1); //will block if already acquired
   if(global_array[0]>global_array[1]){
         temp = global_array[1];
         gloabal_array[1] = global_array[0];
         global_array[0] = temp;

   }
   mutex_unlock(mutex_1);
}

main()
{
      thread * ta, *tb;
      mutex_initizlize(mutex_1);

/* Create two threads: pass no parameter, time quantum is 1
timer tick */
      create_thread( thread_a, NULL, 1 );
      create_thread( thread_b, NULL, 1 );
/* Block main thread until thread_b and thread_a complete */
      join_thread(thread_b);
      join_thread(thread_a);
      mutex_destroy(mutex_1);
}
```

3. User your own thread library to re-implement the multithreaded sorting algorithm developed under 1 above. Compare the performance of both implementations by conducting an analysis similar to the one on slide 8 in section 4 (slides_4.pdf).
4. Make the following changes to the current scheduler. Your new scheduler should support two scheduling classes: Interactive and Background threads. Interactive tasks are scheduled according to the weighted round robin algorithm that is already implemented in the library. Background threads are lower-priority threads that should only be executed if no interactive thread is ready to be executed. Extend the create_thread function so that you can specify whether a new thread is

an interactive or a background thread. Threads of both types can create interactive and background threads. Additionally, implement a `thread_sleep()` function that allows a thread to sleep for a given number of milliseconds. Provide a **meaningful** test program to demonstrate that your scheduler works. This test program should create a number of interactive and background threads each of which should run for a random period of time and sleep for a random period of time.

5. **Grading / Tips:**

- This is not a group project. You have to work on the project on your own.
- Your multithreaded sorting algorithm needs to run the sorting threads in parallel. I will deduct points if the threads execute sequentially!
- You need to study and understand the provided thread library functions before you start coding.
- Turn in your program using 442submit 4.
- The grades are assigned as follows:
    - [10 points] Implementation of the multithreaded sorting using POSIX Pthreads
    - [20 points] Implementation of synchronization primitives (mutex_lock, mutex_unlock, etc.)
    - [5 points] Re-implementation of multithreaded sorting using your own thread library
    - [10 points] New scheduling algorithm (This part is mandatory for grad students and extra credit for undergraduate students).

: