1. Cooperative User-Level Threading

User-level threading packages have become somewhat of a lost art. Their inherent disad
dvantages prevent them from being used in many contexts in which application-level concurrency is required. They do not support the use of multiple CPUs or cores, they lack the ability to preempt an uncooperative thread’s access to the CPU, and they can make it difficult to prevent the OS from blocking an entire process if a blocking system call is made by a thread.

Their advantages include their low cost of context switching, the lack of need for any OS support, and full control over scheduling. Occasionally, they are used in high-performance server designs \cite{[1,2]} to increase scalability and push the boundaries of how many concurrent tasks can be managed on a single CPU or core.

In this exercise, you will be implementing mutexes and semaphores for a small user-level threading package. The need for such synchronization devices exists even in an environment that does not support preemption or allows the use of multiple CPUs, because context switches will still occur if threads block to wait for a resource or event.

The directory \cs3214/public_html/spring2014/exercises/uthreads contains an incomplete implementation of a user-level threading package called \textit{uthreads}. Your task is to study its implementation and complete the methods associated with the types \textit{uthreads_mutex_t} and \textit{uthreads_sem_t}, as shown below.

```c
/** *
 * Support for mutexes.
 */
typedef struct uthreads_mutex {
    uthreads_t holder; // if NULL, mutex is available;
                        // else denotes thread holding mutex
    struct list waiters; // waiting threads, if any.
} *uthreads_mutex_t;

/** *
 * Initialize this mutex. *
 */
void uthreads_mutex_init(uthreads_mutex_t m);

/** *
 * Acquire this mutex. *
 */
void uthreads_mutex_lock(uthreads_mutex_t m);

/** *
 * Release this mutex. *
 */
void uthreads_mutex_unlock(uthreads_mutex_t m);

/** *
 * Support for semaphores.
 */
typedef struct uthreads_sem {
    uthreads_t holder; // if NULL, sem is available;
                        // else denotes thread that owns sem
    struct list waiters; // waiting threads, if any.
} *uthreads_sem_t;

/** *
 * Initialize this semaphore. *
 */
void uthreads_sem_init(uthreads_sem_t m);

/** *
 * Acquire this semaphore. *
 */
void uthreads_sem_acquire(uthreads_sem_t m);

/** *
 * Release this semaphore. *
 */
void uthreads_sem_release(uthreads_sem_t m);
```
typedef struct uthreads_sem {
    int count;  // value of semaphore. Always non-negative.
    struct list waiters;  // waiting threads, if any.
} * uthreads_sem_t;

/* Initialize this semaphore. */
void uthreads_sem_init(uthreads_sem_t s, int initial);

/* Wait on this semaphore. */
void uthreads_sem_wait(uthreads_sem_t s);

/* Post ('signal') this semaphore. */
void uthreads_sem_post(uthreads_sem_t s);

You should fill in the skeletons uthreads_mutex.c and uthreads_semaphore.c. You will need to invoke the function uthreads_block and uthreads_unblock as appropriate to communicate with the scheduler. Since uthreads are not preemptive and do not support multiple cores, no use of atomic instructions is required. If your implementation is correct, running make check should print Ok 3 times. A correct solution can be implemented in 13 lines (uthreads_mutex.c) and 9 lines (uthreads_semaphore.c) of code.

2. Are Linux Mutexes Fair?

A mutex is said to be contended if it is held by another thread at the time when a thread attempts to acquire it. When a mutex is frequently contended, fairness becomes important. As different threads attempt to acquire the mutex, they should have the same chance of acquiring the mutex, and no thread should have to wait indefinitely for it.

In this exercise, you are asked to design and implement a test program that starts N threads that contend for the same mutex in a tight loop. Your program should count how often each thread was able to acquire the lock within a given timespan. To ensure that all threads have started when you start counting you may wish to reset the counters to zero some time (say 1 second) after the threads have started. The main thread should then sleep for some time (say 3 seconds) while the threads contend for the shared lock.

Make the number of threads a command line argument to your program and run it with 1, 2, 3, 4, 5, or more threads. You may wish to repeat each run a number of times.

1. How many times per second was a single thread able to acquire/release the mutex?
2. If you ran 2 threads, how many times per second was the mutex acquired/released?
3. In the case of 2 threads, were both threads able to acquire the lock roughly the same number of times?
4. In the case of 3 or more threads, were those threads able to acquire the lock roughly the same number of times?
5. The fairness of a lock implementation is also affected by whether the contending threads execute on different CPUs or cores or the same CPU. By default, when you start a multi-threaded process on Linux, Linux attempts to distribute the threads belonging to the process evenly across all available cores. The `taskset(1)` command can be used to restrict the set of cores on which a process’s threads may be run. For instance, if you ran `taskset -c 3 ./linux_fairness_test 4` then all threads created by the process started to execute the command `./linux_fairness_test` would be allowed to execute only on core 3. (On our rlogin cluster, each node contains 16 cores numbered 0-15).

Repeat the previous cases (2 or more threads) while restricting the number of cores on which those threads can run to one core - choose one of the available 15. How does this restriction affect fairness?

---

\(^1\)Another way to accomplish the same is using the `pthread_setaffinity_np()` call.
Bibliography
