Project 3

CS6456 (F2016) Operating Systems

William Young

25 October 2016

1 Introduction

This paper describes the implementation of a user-level thread library supporting pre-emptive and priority scheduling and synchronization primitives, wherein multiple threads are created and managed within a single UNIX process. Section 2 includes background information regarding some details surrounding the thread library implementation. The interface routines provided in the library, summarized in sections 3-6, allow a user to initialize and interact with the user-level threads.

2 Library Routines

The user-level thread library implemented in this project contains twelve predefined methods:

- `int uthread_init(void)`
- `int uthread_create(void (*func)(int), int val, int pri)`
- `int uthread_yield(void)`
- `void uthread_exit(void *retval)`
- `int uthread_mutex_init(uthread_mutex_t *mutex)`
- `int uthread_mutex_lock(uthread_mutex_t *mutex)`
- `int uthread_mutex_unlock(uthread_mutex_t *mutex)`
- `int uthread_join(uthread_tid_t tid, void *retval)`
- `int usem_init(usem_t *sem, int pshared, unsigned value)`
- `int usem_destroy(use_t *sem)`
- `int usem_wait(usem_t *sem)`
- `int usem_post(usem_t *sem)`

2.1 Structs

This implementation utilizes custom-defined structures to enable simpler management of individuals threads and the thread library. In addition to the thread and thread library structures implemented in the previous project, this work incorporates a *semaphore* struct, a *mutex* struct, and ten separate *queue* structs (to support priorities). The structs consist of a series of flags and pointers that allow conditions to be easily set, tested, and modified at runtime. For example, the the semaphore and mutex structs each include identification fields so that they only interact with those threads that have made prior invocations to their respective action calls (lock/unlock and wait/post).

2.2 Ready Queues

The ready queues are implemented as linked lists in which newly created or yielding threads are appended to the tail of the list of appropriate priority. Head and tail pointers allow for easy queue management. When *swapcontext* is invoked, the outgoing thread is saved to the tail of its respective priority queue, whereas the incoming thread is selected from the highest priority ready queue that has active threads available.

2.3 Blocking

Rather than remove a thread from a ready queue upon blocking, the project chooses to merely set a *block* flag that is saved with the thread's context. Every block action in the program is followed by a call to the scheduler, meaning that the blocked context will be immediately saved to its appropriate queue. The scheduler is designed to skip blocked threads upon encountering them in a ready queue according to the following:

```
// inside uthread_yield (scheduler)
n = next_thread;
if(n.exit == 1 || n.lck_block > 0 || n.sem_block > 0 || n.j_block > -1)
    uthread_yield();
```

3 Thread Library Initialization

The library call `uthread_init()` functions mostly the same as in project 2, initializing the thread library. The following new additions extend its functionality:

- Allocate memory for ten priority queues via the *malloc()* system call.
- Set *uthread_yield* as the SIGVTALRM signal handler via *sigaction()* system call
- Set the timer interval to 1ms via *setitimer()* system call

4 Thread Creation

The library call `uthread_create(func, val, pri)` creates a new user-level thread with priority *pri* that starts execution with function *func* whose argument is *val* and adds it to the tail of the appropriate ready queue based on *pri*. This library call functions similar here as it did in project 2, save for a single addition:

- Initialize flags for blocked states (lock, semaphore, and join) and return value storage (join)

```
new_thread.j_blocked = -1;
new_thread.lck_blocked = 0;
new_thread.sem_blocked = 0;
threadlib->rets[new_thread.id] = -3091995; //updated on exit()
```

5 Thread Yielding (Context Switching)

The library call `uthread_yield()` saves the context of the yielding thread to the tail of its appropriate priority queue and context switches to the thread located at the head of the highest priority queue with available threads. This library call serves as the signal handler for SIGVALRM and is invoked every time the timer expires or a thread sets a *block* flag.

- Select the highest priority queue that still has available, non-exited threads
    - Invokes the *select_highest_priority_queue* helper method, which returns an integer
- Check to see if a mutex or semaphore

- Check to see if the thread at the head of the selected queue is blocked
    - If blocked on a semaphore, check if semaphore is available
        - If available, unblock the thread
    - If blocked on a mutex lock, check if mutex is available
        - If available, unblock the thread
    - If blocked on a join, check if target thread has exited
        - If it has, unblock the thread (return value now available to thread)
    - If none of these flags are set, make a call to the scheduler (will select a new thread)
- Designate the tail of the outgoing queue as the location where the current context will be saved on *swapcontext*
- Save thread metadata (exit status, flags, ID, etc) to the tail of the outgoing queue
- Update the global metadata to reflect the state of the next thread to schedule
- *swapcontext()* is invoked to perform two actions:
    - Save the context of the yielding thread
    - Load the context of the head of the ready queue

6 Thread Termination

The state of the current thread (set in *uthread_ yield*) is saved as a global entry in the *threadlib* struct and examined in each instance of *uthread_ yield.* As a result, when a thread invokes *uthread_ exit*, by setting the current thread's *exit* flag to 1, we can ensure that it will *not* be scheduled in a subsequent invocation of *uthread_ yield.*

- Set the current thread's *exit* flag to 1.
- Designate a return value (random number between 1 and 256), save it to the return value array, and decrement the active thread counter
- Check to see if another thread is waiting to join with us.
    - If so, set a flag that will be visible when the scheduler attempts to determine if the next thread is blocked on a join
- If more threads remain queued in any of the priority queues, call *uthread_ yield.*
- If the thread calling *uthread_ exit* is the last remaining thread, deallocates the helper arrays, the ready queues, and the *threadlib* object.

7 Mutex and Sempahore Initialization/Destruction

Initialization

These methods are simple—in each case, initialize the object in question with default values and set *init* to 1 to indicate the the object in question has been initialized

Semaphore Destruction

In *sem_ destroy*, set *init* of the semaphore object to 0, preventing other threads from accessing it.

8 Mutex Lock/Unlock

Lock

To lock the mutex, first determine whether it's already locked. If so, set the current thread's mutex lock-blocked flag to 1 and invoke the scheduler. If the mutex is available, set lock = 1 and set a global flag indicating that the mutex is now unavailable (lock value and global flag set within *sigprocmask* that blocks SIGVTALRM).

Unlock

To unlock the mutex, set lock = 0 and set a global flag indicating that the mutex is now available (lock value and global flag set within *sigprocmask* that blocks SIGVTALRM). Now, when a thread blocked on the mutex is selected by the scheduler, it will see that the mutex has been unlocked and is therefore free to assume ownership of the mutex.

9 Semaphore Wait/Post

Wait

- If the semaphore value is 0, set a global flag indicating that the semaphore is unavailable, set the current thread's semaphore block-flag to 1, and invoke the scheduler.
- If the semaphore value is greater than 0, decrement the value (while in possession of a mutex lock)

Post

- Increment the semaphore value (while in possession of a mutex lock)
- If the value is now greater than zero, set a global flag indicating that the semaphore is available. Now, when a thread blocked on *sem_wait* is selected by the scheduler, it will see that the semaphore value is greater than 0 and is therefore free to decrement the value.

10 Join

If the thread ID specified to join with is invalid or another thread is already waiting on it, exit with error. If the thread in question has already exited, send its return value back to the caller. If the thread has not yet exited, set the current thread's join block-flag to the ID of the thread it's waiting for and invoke the scheduler.

```
int uthread_join(uthread_tid_t tid, void *retval) {
     if(tid < 0 || tid > threadlib->num_nodes) { return -1; }
     if(threadlib->joins[tid] > 0) { return -1; }

     // thread has not terminated yet
     if(threadlib->rets[tid] == -3091995) {
       threadlib->global_curr.j_blocked = tid;
       threadlib->joins[tid] = threadlib->global_curr.id;
       uthread_yield(); }

     // thread terminated; return value
     int* ret = &(threadlib->rets[tid]);
     *((int**)retval) = (void*)ret;

     return 0;
}
```