Operating Systems User Level Thread Library Part 2

Submitted by: Bryan Walsh (bpw7xx) and Jared Culp (jjc4fb)
Total effort expended: ~30 hours

Data Structures

In our implementation of the scheduling program we used three main data structures: a uthread struct to hold thread information about created threads, a priority_level struct to emulate a queue for scheduling threads, and a uthread_mutex struct to hold information about whether or not the mutex lock is held and a queue of threads waiting on the mutex lock to be released. Each structure is listed below, along with a description of its contents.

```
typedef struct uthread {
      ucontext_t context;
      void *stack;
      struct uthread *next;
      int priority;
      int t_id;
      int holds_lock; // prevent others from unlocking
} uthread_t;
```

Each uthread_t contains a context, a pointer to its stack, a pointer to the next thread to run, and a thread id (t_id). Things that we added to assist our implementation are a priority integer, which can range from 0 to 9, and an integer "holds_lock" that indicates whether or not a thread currently holds the mutex lock.

```
typedef struct priority_level {
      uthread_t *head;
      uthread_t *tail;
      unsigned int size;
} priority_level_t;
```

Each priority_level_t acts as a queue; it contains a pointer to the head and the tail of the queue, and also has a notion of the size (i.e. the number of threads in that priority level)

```
typedef struct uthread_mutex {
      int lock_held;
```

```
    struct priority_level queue[PRIORITY_LEVELS];
} uthread_mutex_t;
```

Each uthread_mutex_t contains an integer “lock_held” that indicates whether or not the mutex is currently locked by a thread. Each also contains a queue of priority_level_t’s which serves as a way to keep an ordered queue of threads waiting on the mutex to be unlocked. i.e. the threads in the mutex’s queue are threads that have been put to sleep while they are waiting for the lock to be released.

Design Justifications

Our implementation depends on a data structure that we created, an array of priority_level structs which, in essence, serves as a “queue of queues”, meaning that the array itself is a queue, containing “mini” queues that are queues for threads of equal priorities. This provides an easy way to find the next runnable thread, since we can then just loop through the queue of queues and retrieve the head of the highest priority_level that contains threads. When a thread is put to sleep while waiting for a mutex lock, it becomes as simple as moving it from the global queue of queues to the mutex’s waiting queue (which identical in structure to the global queue of queues).

Our implementation also utilizes a timer that decrements from 1ms to 0ms. Every time the timer reaches 0ms, a signal is raised and a call to yield is made to handle switching between threads. This is our way of implementing the round robin scheduler using 1ms time slices.

Implementation Details

Our thread library now supports the following API calls:

- `void uthread_init(void);`

In uthread_init we create main’s context, with highest priority, and add it to the global schedule_queue, which is the global list of threads to run. We also set up the cleanup context which is used to free memory associated with threads when they make a call to exit(). Additionally, an important action taken in uthread_init is that we fill a global sigset_t called all_signals with all available signals (by making the call: sigfillset(&all_signals)). We do this so that we can block and unblock all signals during uthread_mutex_lock and uthread_mutex_unlock.

- `int uthread_create(uthread_func_t func,int val,int pri);`

In uthread_create we create a new thread by malloc’ing for a new uthread_t. We then set up the thread by setting it’s stack pointer, stack size, and uc_link. We also set it’s priority and thread id. Lastly, we put the new thread into the global list of threads to run (schedule_queue).

- `void uthread_yield(void);`

In uthread_yield, we update the next thread to run within the yielding thread's priority_level, and then we find the next thread that needs to run by looping through the queue of threads to run and picking the highest priority thread that is at the beginning of its priority_level. We then make a call to swapcontext() to swap from the yielding thread's context to the context of the thread to which we are switching.

- `void uthread_exit(void);`

In uthread_exit, we update the next thread to run within the exiting thread's priority_level, and then we make a call to setcontext() to call our cleanup_context which cleans up allocated memory associated with the exiting thread.

- `void uthread_mutex_init(uthread_mutex_t *lockVar);`

In uthread_mutex_init we simply just set the lockVar's lock_held variable to zero, meaning that upon the mutex's creation it is in an unlocked state.

- `void uthread_mutex_lock(uthread_mutex_t *lockVar);`

In uthread_mutex_lock, first we disable interrupts. Next, we lock the mutex if it is available, or, if it is unavailable, we remove the calling thread from the global queue of threads to run (schedule_queue) and add it to the mutex's queue in the appropriate priority level queue, where it waits until the mutex is unlocked before returning to the list of threads to run. We then must swap our context to the next thread waiting to be scheduled in the schedule_queue.

If a lock is not held then we simply set lockVar's lock_held variable to be 1 and then set the calling thread's holds_lock variable to 1 as well, meaning that the mutex knows that it is locked and the thread knows that it has the mutex lock.

At the end of the lock function we then re-enable interrupts so that they are unblocked for the rest of the program execution.

- `void uthread_mutex_unlock(uthread_mutex_t *lockVar);`

In uthread_mutex_unlock, first we disable interrupts. Then we set lockVar's lock_held variable to 0 and the calling thread's holds_lock variable to 0, meaning that the mutex knows it is unlocked and the calling thread knows that it no longer holds the lock.

Next, we check to see if there is any thread that has been waiting for the mutex to be unlocked. If there is a thread waiting, then we must move the waiting thread (of highest priority, of course) from the mutex's queue (where it has been sleeping) back into the global list of threads to run (where it can then be scheduled when appropriate). I.e. we "wake" the thread up from sleep.

- `void uthread_mutex_destroy(uthread_mutex_t *lockVar);`

For our implementation of this scheduling, we did not dynamically allocate any memory for the mutex structure, so this function serves no purpose.

Difficulties Encountered

Some of our original difficulties came from indecisiveness about how to store threads waiting to run. Originally we tried to maintain just a linked-list structure that was kept in a specific order, but that ultimately proved to be too difficult. We then decided to implement an array of priority_level structures, essentially a "queue of queues". Each array index corresponds to a priority_level which is a queue of threads waiting to run within that priority level.

Another difficulty encountered was with swapping threads between the schedule_queue and the mutex's queue. Extracting a thread from the schedule_queue and rearranging the extracted thread's pointers (as well as the pointers in schedule_queue pointing to the extracted thread) proved to be difficult and required multiple attempts before being able to successfully put threads to sleep and wake them up.

One last difficulty comes from lack of experience in dealing with interrupt handlers and signals in general in a C context. My partner and I had little-to-no idea about how to write interrupt handlers and how to deal with a timer and the interrupt generated by the timer, so the last part of the lab, the implementation of the 1 ms time slices, was challenging as well.