CS 415
Operating Systems Practicum

Project 2: Preemptive Threading

Oliver Kennedy
okennedy@cs.cornell.edu

Goals

- Activate the clock
- Add an alarm
- Add Sleep()
- Improve your scheduler

Part 0: Setting up

- Get the source from CMS
- You will need to merge your changes to project 1 with project 2
  - Replace queue.c, synch.c
  - 3 new functions in minithreads.c
  - All other source files should be from project 2.

Part 1: Advanced Threading

- Your code should already be threadsafe
  - Now we turn on the clock
- Change minithread_init()
  - minithread_clock_start(&clock_handler);
  - set_interrupt_level(ENABLED);
- clock_handler != minithread_yield

Part 1: Advanced Threading

- Implement _stop() and _unlock_and_stop()
  - Update semaphore_p() and _v()
- stop() should switch to another thread without putting the current thread back on the run queue
- unlock_and_stop() should clear a lock and atomically switch to another thread
  - Implement this by disabling interrupts

Part 2: Alarms

- Fill in the blanks: alarm.c
- register_alarm(delay, func, arg);
  - in [delay] seconds, call [func]([arg])
  - return a unique id;
- deregister_alarm(id);
  - prevent alarm [id] from triggering
    - (if it hasn't already been triggered)

Part 2: Alarms

- How to store alarms
  - Use a queue
  - Use a custom datastructure
- Run alarms in clock_handler
  - use time(NULL)
  - Don’t block.
- Run alarms in their own thread.

Part 2: Alarms

- Representing an alarm
  - Trigger time (time(NULL) + delay)
  - func + arg
- When to run an alarm?
  - compare time(NULL) > trigger

Part 2: Alarms

- Queue
  - queue_iterate() to find an alarm to run (and then delete it)
- Custom Datastructure
  - Implement a sorted queue.
  - Insert alarms into the queue so that the next alarm is always at the front of the queue.

Part 3: Sleep()

- Fill in the blank: ..._sleep_with_timeout()
- Implement using alarms
  - Register an alarm for minithread_start()
  - Call minithread_stop();

Part 4.1: Advanced Queues

- Fill in the blanks: multilevel_queue.c
- A datastructure that stores an arbitrary number of queues
  - Remember, arrays are just chunks of memory.
  - malloc(num * sizeof(int)) creates an array of num integers
  - Allocate an array of queues.

Part 4.2: Advanced Scheduling

- Not so much fill in the blanks as tweak minithread.c
- You had 1 run queue before, now you have 4.
  - Run the thread at the front of the highest priority queue.
  - If the queue is empty, go to the next
- After 10 cycles of waiting at the head of a queue, a thread graduates to the next priority.

Part 4.2: Advanced Scheduling

- Priorities
  - Each thread is assigned a priority when it is created (Start in the middle)
  - When a thread yield()s (or hits a clock interrupt), it should enter the queue for the priority level below the one it was running at. (or stay if it's at the bottom)

Part 4.2: Advanced Scheduling

- Quanta
  - clock_handler() should only call yield() every X times it gets called.
- X is defined by the priority level.
  - For the highest priority threads, X should be 1.
  - X doubles at every step down the priority ladder.

Good Luck