Assignment 2
“This time, it’s preemptive”

Ari Rabkin

Preemption

- Need to preempt threads for scheduling fairness.
- Use interrupts to regain control
- Call minithread_clock_init to get a stream of interrupts; specify interrupt handler.
- Return from handler ends interrupt

Interrupts

- Recall that interrupts are delivered on stack of current thread.
- Can call yield to switch threads
- Interrupts arrive every PERIOD.
- Should update counter (*ticks*) on each interrupt

Scheduling

- We’re going to do strict priority scheduling.
- Keep k queues, one for each priority.
- Next thread should come from highest non-empty queue
- Hide complexity inside multilevel_queue.c

Strict priorities

- We’re going to have two kinds of threads:
- User threads (low priority)
- System threads (high priority)
- “Run system threads if you can, user threads if no system threads to run”
- Pick new thread every tick.

Hide details

- Hide the details of this in multilevel queue module. You might add more levels later.
- Note that this is **not** a feedback queue.

Alarms

- Want stuff to happen at fixed time in future.
- Alarms let a thread say “run this code in 200 milliseconds.”
- To create alarm, specify function to be called, argument, and time offset.
- Identify alarms with alarm IDs.
- See alarm.c and alarm.h

Alarm operations

- Two operations on alarms: register and deregister.
- Third operation (invocation) is implicit.

Design choices

- Can run alarm procedure in special thread, or at interrupt time, or...?
- Common case is short and small alarms.
- Note that if you call at interrupt time, alarms must be small.
- Specify which implementation you chose in your design doc!

More design choices

- Lots of clever things you can do.
- But think them through in advance, and check with us.
- This is your chance to be creative and practice your engineering.

Sleep_with_timeout

- Sometimes want to have a thread sleep for a fixed number of milliseconds.
- For instance, network timeouts.
- You should add a function to do this.
- Can implement timeouts cleanly with alarms and semaphores.

Measurement

- Often want to know how fast code runs.
- Add support for per-thread timers
- Read them with read_counters() or whatever you name it.
- For each thread, track several quantities.
- For each quantity, read clock at start and end of timeslice, and add difference to counter.

What to measure

- Want to measure three things per thread: wall time, process CPU time, and minithread time.
- Get wall time from GetTickCount in windows.h
- Get CPU time from clock() in time.h
- Get minithread time from your tick counter.

What this measures

- clock() gives you ticks of CPU time given to the process (CLOCKS_PER_SEC ticks per second)
- GetTickCount() returns milliseconds since system boot.
- Your counter PERIOD miliseconds. (See interrupts.h)

Why are these different?

- Q: Why can CPU time not match minithread time?

  A: libraries, interrupts off...

- Q: Why doesn’t CPU time match wall time?

  A: Other processes!

Reading atomically

- Want to read all three counters “atomically”
- Easiest way is to define a minithread_stats struct, and have your read_counters function return it.
- Be sure to make read_counters threadsafe.

Part 5: Prime numbers

- A prime number is an integer divisible only by itself and one.
- You’re going to write a random-prime-generator.
  - Operates on “small” numbers (order of 10^8), can use trial division

Prime numbers

- Going to also use threads.
- One thread to get random, one thread to do trial division, and a third to print the results
- Threads communicate using semaphore-protected queues.
- Need two semaphores; one as mutex on queue, one to put bound on queue length.

The generator

- Generator thread needs to pick ints in range from 2 to MAXPRIME, where there's some #define MAXPRIME xx statement.
- xx should be 10^8 or so.
- Use genintrand(int x) to get random numbers between 0 and x
  - Declared in random.h

Getting random numbers

- Problem: computers are deterministic; where can random numbers come from?
- Solution: seed generator with some "random" data from world.
- Call *sgenrand(long seed)* with some random seed. Using *time()* is common.

Prime testing

- Testing is easy. Just divide candidate prime p by all integers between 2 and sqrt(p).
- Remainder zero implies not prime.
- Can use % operator to check if a number divides another:
  - $a\%b$ = remainder when a is divided by b.
- Pass primes on to printing thread

The printing thread

- Display some of the primes. (Use printf.)
- Don’t have to print them all. Tune the fraction printed so you print a few per second.
- Should periodically (every 10 seconds?) update with total found per second.
- Use time(0) to find seconds since epoch.
- Subtract to find elapsed time.

Statistics

- Use your stats feature!
- Have program display fraction of time spent in each thread.
- Time spent in libraries (per thread)
- etc...

Design process

- This time, we'd like you to sign up in advance for a design doc review.
- No more than two groups per TA per hour.

What to put in

- Same story as last time.
- Specify functions — particularly those you define. Interface, behavior, assumptions, pseudocode if it's not obvious.
- Invariants on data structures. What goes in them? When are they defined?
- Test strategy, bugs you want to catch.

Further doc. help

- We’re especially interested in data structure invariants.
- We’ll have a template and an example from P1 available soon.

Scheduling

- Sign up in advance for a time slot.
- Signup sheet is up front.
- Can do it now!