Project 2
Supplemental Lecture

Sean Ogden
Cornell CS 4411, October 4, 2013

Today's Lecture

- Administrative Information
- Common mistakes on Project 1
- Project 2 FAQ
- Discussion

Administrative Information

- Project 1 is still being graded
- Project 2 deadline is October 10, 11:59:59

Project 1 Queue Errors

- Mallocing sizeof(queue_t)
- Not checking return value of malloc
- Memory leaks in dequeue and delete
- Not checking return value of function in iterate
- Lack of thorough testing
  - Many errors can be caught by simple unit tests

Project 1 Minithread Errors

- Idle thread runs when there are other threads to run
- minithread_yield() and minithread_stop() switch to idle thread
- Final proc context switches directly to clean up thread
- Cleanup thread not using semaphore
- Final proc can reach end of function

Project 2 FAQ

- All library calls are safe: interrupts are automatically disabled upon calling
  - interrupts will be restored to its original state (enabled/disabled) after the call.
- Units of time
  - PERIOD is defined as 50 ms, which is 50000 as a constant.
  - Alarm and wakeup delays are specified in milliseconds.
  - You have to convert units; don’t blindly subtract PERIOD.
- Irregular/random clock interrupts
  - This is normal
  - Be careful of introducing heisenbugs because of your debug statements.

Disabling Interrupts

- When you need to do something that must be done atomically.
- Typically manipulations on shared data structures.
  - Data structures that can be accessed by multiple threads ‘simultaneously’.
  - Modifying the cleanup queue, ready queue, alarm list.
- Trivial way of achieving correctness: disable interrupts for everything.
  - Why is this a bad idea?

Interrupt Handler - Reminder

- Entry point when a clock interrupt occurs.
- Are there problems if the interrupt handler is interrupted?
  - Yes – accessing shared data structures
  - Solution – disable interrupt in the interrupt handler

CANNOT BLOCK

Semaphore Revisited

- Typical sem_P code:

```
while (TAS(&lock) == 1) yield();

sem->counter--;
if (sem->counter < 0)
{
    append thread to blocked queue
    atomically unlock and stop
}
else
{
   atomic_clear(&lock);
}
```

Semaphore Revisited

- Typical sem_V code:

```
while (TAS(&lock) == 1) yield();

sem->counter++;
if (sem->counter <= 0)
{
    take one thread from blocked queue
    start the thread
}

atomic_clear(&lock);
```

Semaphore in User Space

- Interrupts can arrive at any time.
- If interrupts arrive while a TAS lock is held:
  - Another thread that tries to acquire the TAS lock will yield.
  - Eventually the holder of the TAS lock will regain control and clear it.

semaphore_P

```
while (TAS(&lock) == 1) yield();

sem->counter--;
if (sem->counter < 0)
{
    append thread to blocked queue
    atomically unlock and stop
}
else
   atomic_clear(&lock);
```

semaphore_V

```
while (TAS(&lock) == 1) yield();

sem->counter++;
if (sem->counter <= 0)
{
    take one thread from blocked queue
    start the thread
}

atomic_clear(&lock);
```

Semaphore In Kernel Space

- Typically used to block some thread and wake it up on some condition
  - minithread_sleep_with_timeout()
  - wake up the thread after the elapsed time
- Waking up requires calling sem_V on that sleep semaphore
- Where is this done?
  - Done in kernel space with interrupts disabled.

Unfortunate Interleaving

- What if user calls sleep_with_timeout(0) ?
  - sem_P is called, and thread blocks itself.
- What if sem_P was interrupted just after placing thread on blocked queue but before clearing TAS lock?

user calls sleep_with_timeout(0)...

```
while (TAS(&lock) == 1) yield();

sem->counter--;
if (sem->counter < 0)
{
    append thread to blocked queue
    [redacted] stop
}
else
   atomic_clear(&lock);
```

clock interrupt!

...clock handler tries to wake that thread up

```
while (TAS(&lock) == 1) yield();

sem->counter++;
if (sem->counter <= [redacted]
{
    take one thread from blocked queue
    start the thread
}

atomic_clear(&lock);
```

interrupts are no longer disabled!

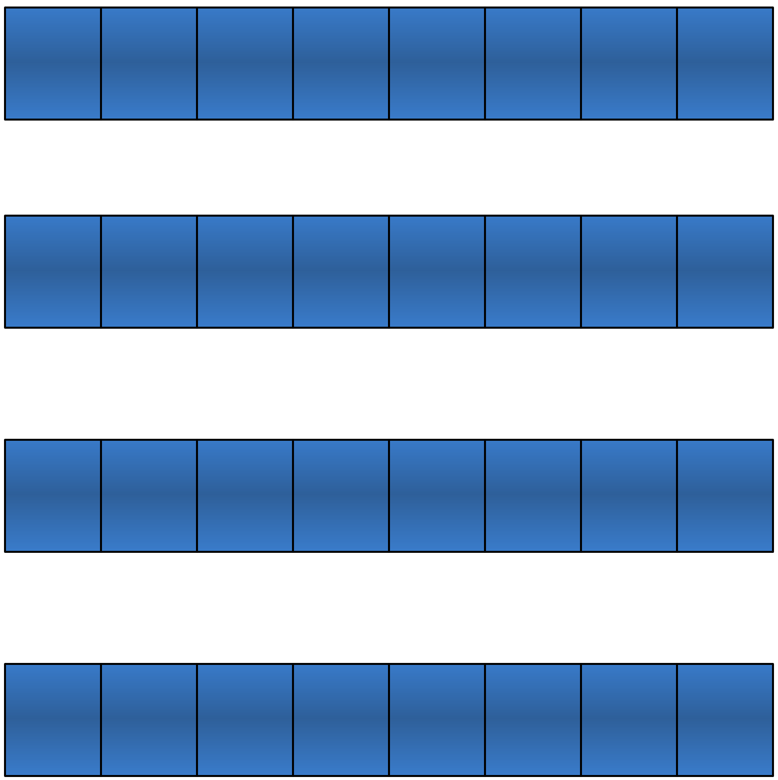
Solution

- Disable interrupts for sem_P and sem_V for minithread_sleep
  - Atomicity: sem_P will be done with everything before an interrupt can possibly arrive.
  - If you **always** access the semaphore with interrupts disabled, acquisition of TAS is guaranteed
- What about sem_V?
  - sem_V is called from interrupt handler.
  - Interrupts are already disabled in the handler.

When is this applicable?

- If semaphore will be used in portions of your kernel where interrupts are disabled.
  - Right now: only the sleep semaphore.
- What about cleanup semaphore?
  - Cleanup semaphore is not signaled from any place where interrupts are disabled.
  - Cleanup code should only disable interrupts while accessing the cleanup queue, not for semaphore signaling.

Scheduling



0 Schedule 80 quanta – 1 quanta per thread

1 Schedule 40 quanta – 2 quanta per thread

2 Schedule 24 quanta – 4 quanta per thread

3 Schedule 16 quanta – 8 quanta per thread

What is the maximum number of unique threads that can run per level?

Scheduling

- Completes 1 sweep over the queue in approximately 160 ticks
- If there are no threads in a given level, schedule threads from the next available level
- Thread level starts at 0 and can only increase throughout its lifetime

Priority Changing

- Threads are scheduled to run for the max duration of the current level
  - For example, in level 1, each thread will be scheduled to run for 2 quanta
- A thread is demoted if it uses up the entire quanta
  - What if the thread is from a different level?

Alarms

- Useful construct for scheduling a thread for future execution
  - Can be used for minithread_sleep()
- Each alarm requires a call back function
  - Call back functions might not be executed by the thread that registered the alarm!
- How to keep track of alarms?
  - Add functionality to existing queue.
  - Insert should be O(n), remove min should be O(1).

Alarm Firing

- Where should the alarm be fired?
  - Interrupt handler
- When should an alarm be fired?
  - Tick == alarm expiration time
  - Can this be missed?
- How should be alarm be fired?
  - Context switch to alarm thread?
  - Should fire in the context of the currently executing thread

Testing

- There are a lot of parts to this project
  - Multi-level queue
  - Interrupts
  - Alarms
  - Thread levels

- Common pitfalls
  - Unnecessarily disabling interrupts
  - Not disabling interrupts when necessary
  - Multi-level queue corner cases

Questions

Questions?