Project 2

Adding Preemption

Zhiyuan Teo

Slide heritage: Previous TAs → Robert Escriva

Cornell CS 4411, September 23, 2011

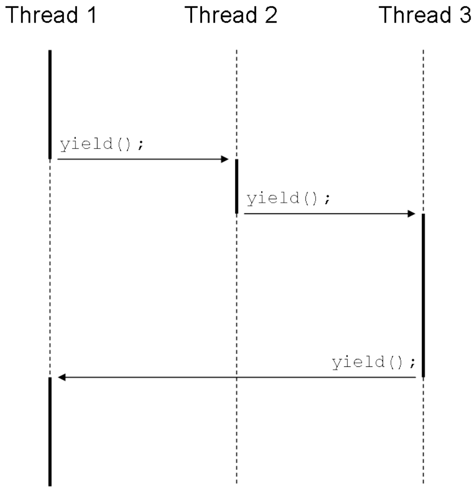
Announcements

- Project 2 is out, due 11.59pm Friday, 30 Sep.
- For CS4411-only students: no need to do CS4410 miniprojects.
- Some partners will be swapped; affected groups will be informed by e-mail.
- This project builds upon project 1. If you had serious problems with project 1, see the course staff immediately.

Building an operating system

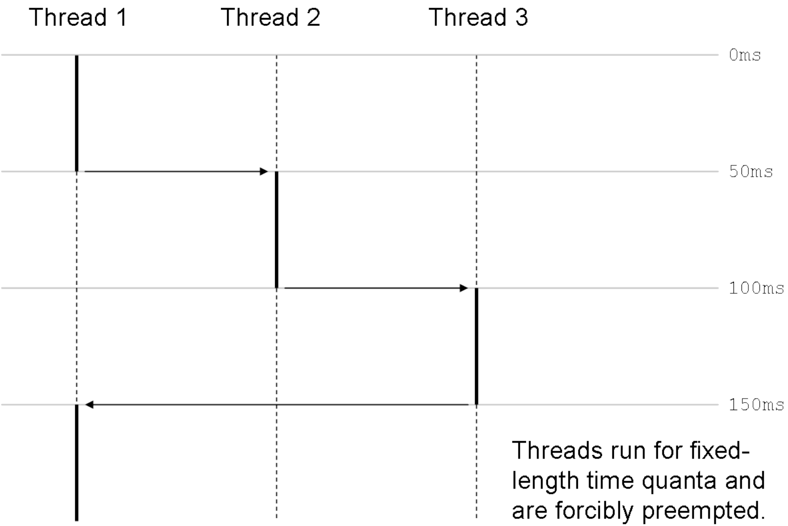
- Theory is neat and great, but engineering is dirty work (sometimes).
- Issues arise in real systems design that don't fit nicely into what theory says.
- Make your own assumptions where specs are unclear.
- Reminder: this isn't CS100! We are not looking for exacting solutions most of the time.
- But be reasonable about your assumptions.

Lessons learnt: non-preemptive scheduler



Threads may run for arbitrary periods; they have to give up the CPU voluntarily.

Preemptive scheduler (simple view)



Threads run for fixed-length time quanta and are forcibly preempted.

Goals of this project

- Design an interrupt handler.
- Learn to reason about and write thread-safe code.
- Upgrade your OS to an advanced multilevel scheduler.

Deliverables

- Add preemption to your scheduler.
  - You will use clock interrupts for preemption.
  - All code you wrote before must be made (mini)thread-safe.
- Alarms; sleeping with a timeout.
- Multilevel feedback scheduling policy.
  - Assign priorities to threads.
  - Round-robin between threads of the same priority.
  - Scheduler will change thread priority based on feedback from thread behavior.

What does adding preemption involve?

1. Identify portions of code where race conditions can occur.
2. Make your code thread-safe by introducing synchronization.
3. Install the interrupt handler and enable clock interrupts.

Implementation plan

1. Start receiving clock interrupts.
   - Register interrupt handler.
   - Start measuring time in ticks.
2. Add preemption.
   - Synchronize access to global structures.
     - Interrupts may arrive at any time.
     - Our synchronization method of choice: disabling interrupts. *
   - Switch threads in the interrupt handler (if required).

---

*But user threads should not rely on disabling interrupts to achieve synchronization.

Implementation plan

3. Add alarms.
   - Create software structure(s) to track pending alarms.
   - Use the software clock to measure elapsed time.
   - Register/deregister alarm callbacks.
   - Start firing alarms from the clock interrupt handler.
4. Add sleeping.

```
minithread_sleep_with_timeout(int delay);
```

   - Block threads until it is time to wake them up.

Implementation plan

5. Add multilevel feedback scheduling.
   - Implement multilevel feedback queues.
     - Use a regular queue as the underlying structure.
     - Add a cyclic search for multilevel dequeue operation.
   - Extend your scheduler to use the new policy.
     - Switch to the new data structure.
     - Cycle through all four levels (to avoid starvation).
     - Add feedback and move threads between levels.

Interacting with Interrupts

- Definitions:

```
typedef void (*interrupt_handler_t)
                    (void *);
void minithread_clock_init(
    interrupt_handler_t clock_handler);
```

- Sample clock handler: †

```
void clock_handler(void* arg) { }
```

† although the clock handler accepts an argument, it is not used.

Writing an Interrupt Handler

- The interrupt handler is interruptible!
  You should disable interrupts (temporarily) while in the handler.
- Interrupt handlers should be fast:
  - System functions, `printf`, etc. are all too expensive.
  - You definitely
    CANNOT BLOCK!

Enabling/Disabling Interrupts

- Definitions for changing interrupts:

```
typedef int interrupt_level_t;
#define ENABLED 1;
#define DISABLED 0;
interrupt_level_t set_interrupt_level(
    interrupt_level_t newlevel);
```

- **Strongly recommended** usage:

```
interrupt_level_t oldlevel =
    set_interrupt_level(DISABLED)
do_something();
set_interrupt_level(oldlevel);
```

Keeping Time

- Change the `PERIOD` in `interrupts.h`:

```c
#define SECOND 1000000
#define MILLISECOND 1000
#define PERIOD (100*MILLISECOND)
```

- Measuring elapsed time
  - System functions to read current time are way too slow.
  - Software clock: deduce current time by counting interrupts.

```c
extern long ticks;
```

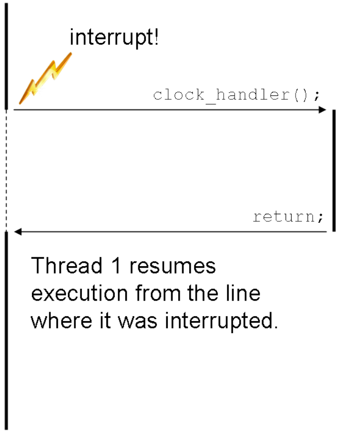
How are interrupts processed?

- Interrupts are not threads!
  - They execute in the context of the thread that happened to be running when the interrupt was triggered.
- The process of an interrupt:
  - Thread is running when interrupt arrives.
  - Current state is saved on the stack of the running thread.
  - Interrupt handler is called. ‡
  - After the handler completes, the saved state is restored.
  - Thread continues to run.

‡What happens if a context switch takes place in here?

Interrupt processing without a context switch

Thread 1



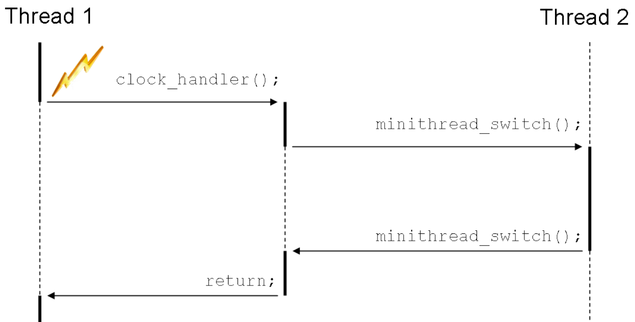
An interrupt is like an unscheduled function call.

```
int body_proc(arg_t arg) {

   counter = 0;
   counter += 10;
          interrupt arrives
   clock_handler();
          exit interrupt handler

   counter += 100;
   printf("counter %d",
      counter);
}
```

Interrupt processing with a context switch



After switching back from thread 2 to thread 1, execution resumes from the instruction after `minithread_switch()` in the interrupt handler.

Interrupts are enabled at this point (because our context switch function automatically enables interrupts).

Interrupts and System Calls

- Windows' system libraries are not (mini)thread-safe... ... so interrupts are automatically disabled while the process is inside system calls.
- Caveat: If an interrupt arrives when interrupts are disabled, it is lost.
- What happens if e.g. a thread spends a lot of time making system calls (eg. printing to the screen)?
  - Most interrupts are missed.
  - Scheduler cannot promptly switch between processes.
  - Software clock drifts; alarms don't fire on time.

Why the need to synchronize?

- Clock interrupts may arrive at any (unprotected) place in your code.
- Any thread may be preempted while reading/writing the kernel's data-structures.
- The clock handler needs to access the same global structures (so that it may preempt threads).
- If shared data structures are in an inconsistent state when an interrupt arrives, the kernel could crash when another thread accesses them.

Synchronization Strategies for kernel data structures

- What <u>not</u> to use: spin locks
  - Cannot use with interrupts disabled.
    - Without preemption, how will the lock being spun on ever change anyway?
- What to use: disabling interrupts
  - Works well on uniprocessors.
  - Critical sections must be short (interrupts should not be disabled for long).
  - Disabling interrupts unnecessarily will be penalized.
  - Follow the recommended pattern of usage.
- These strategies do not apply for semaphores.

Information so important that it has its own section

- Avoid unmatched enabling/disabling.
  - Your function could be called with interrupts already disabled (enabling them would compromise your system's safety).
  - Application code should *never* run with interrupts disabled.
- Do not disable interrupts unnecessarily.
  - User threads should not manipulate interrupts directly to synchronize their code.
  - Use better synchronization primitives that our OS provides.
- Do not disable interrupts for too long.
- Context switches automatically re-enable interrupts.
  - Reason about where control could be after switching back - subsequent code is no longer protected from interrupts.

Semaphores

- Follow the implementation taught in CS4410.
- If a thread is unable to acquire the TAS lock, it should spin, not yield!
- You should disable interrupts first before making calls to semaphore_P from within your kernel code.

Implementing Alarms

- What you need to implement:

```
int register_alarm(
    int delay,
    void (*func)(void *),
    void* arg);
void deregister_alarm(int alarmid);
```

- What you need behind the scenes:
  - Some structure to keep information about registered alarms.
  - Code in the interrupt handler to fire alarms.
    - Use `ticks` to calculate elapsed time.

Using Alarms

- Alarms are fired in the interrupt handler.
  - Alarm callbacks are not threads! They are just functions called from the interrupt handler (which also isn't a thread).
  - Interrupts should already be disabled in the interrupt handler.
  - You cannot spend much time in the alarm callback.
  - You cannot block.
  - Assume alarm callbacks are cooperative.

  - Alarm handler is called in the context of the **currently executing thread**...

    ... which is likely to be **different from the thread that registered the alarm**.

Implementing thread sleeping

- What you need to implement:

```
void minithread_sleep_with_timeout(
    int delay);
```

- Expected behavior:
  - Block the caller (and relinquish the CPU).
    The caller should not be on the ready queue.
  - Wake up the thread after the timeout expires.
    Make the thread runnable (put it on the ready queue); a context switch is unnecessary.

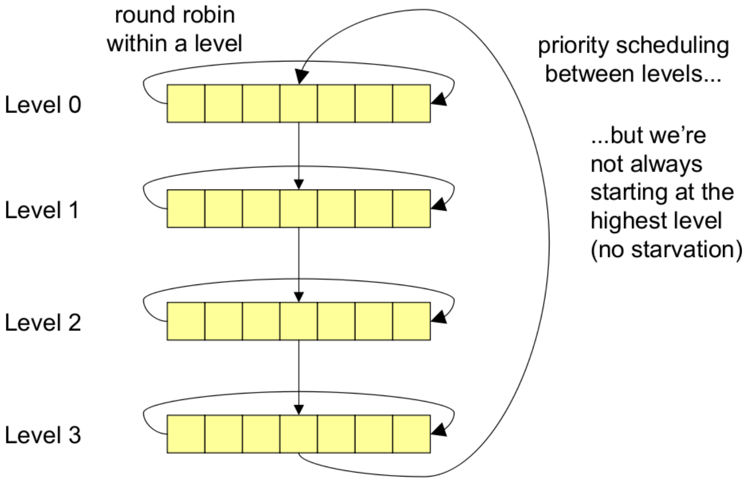
Implementing thread sleeping

- Use the alarm callback functionality you developed earlier to implement thread sleeping.
- You should use semaphores instead of `minithread_start()` and `minithread_stop()`
  - This is more modular structure.

Multilevel Queue Prototypes

```
typedef struct multilevel_queue*
    multilevel_queue_t;
multilevel_queue_t multilevel_queue_new(
    int number_of_levels);
int multilevel_queue_enqueue(
    multilevel_queue_t queue,
    int level, any_t item);
int multilevel_queue_dequeue(
    multilevel_queue_t queue,
    int level, any_t *item);
int multilevel_queue_free(
    multilevel_queue_t queue);
```

Multilevel Queue Structure



Scheduling Policy

- Cycle through all four levels (moving the starting point for a dequeue).
- After a given number of quanta, move to the next level.
- Spend 80 / 40 / 24 / 16 quanta in levels 0 to 3, respectively.
- Assign 1 / 2 / 4 / 8 quanta at a time to levels 0 to 3, respectively.
- If there are no threads to schedule for a level, look in the following levels.
- Schedule in round-robin fashion within a level.

Edge cases in multilevel scheduling

- A thread scheduled outside of its level should be awarded quanta based on the current level, not the thread's level.
  - Example: current level is 1, but next available thread is in level 3. Schedule the thread for 2 consecutive quanta.
- Yielding, stopping or blocking a thread exhausts all its allocated time quanta, even if they have not fully utilized their allocation.
  - Example: a thread is allocated 8 quanta, but it completes its work and yields in 2.5 quanta. The rest of the 5.5 quanta are forfeited.

Edge cases in multilevel scheduling

- Partial quanta are allocated as if they were a full quantum.
  - Example: a thread is allocated 1 quantum, but it yields in 0.9 quantum. The rest of the 0.1 quantum is allocated as if it were a full quantum.
  - Raises the question of fairness, but this is engineering![§] (Linux does the same thing).
- If there are insufficient level quanta to allocate to the next thread, allocate it anyway.
  - Example: a thread on level 3 is promised 8 quanta, but there are only 3 quanta remaining for the level. Run the thread anyway and treat it as if all 8 quanta were allocated.

[§] Can you do better?

Thread Priorities

- Extend the TCB to keep a thread's priority.
- A thread's priority determines which queue (0-3) a thread goes into.
  - A thread's queue determines the size/frequency of a thread's allocated run time.
- A thread starts at the highest priority.
- Priorities decrease over time.
  - A thread receives lower priority when it outruns its quanta.

Changing priorities

- Change the thread's priority (in the TCB).
- Re-evaluate priority on context switch.
  - Leave the priority unchanged
    - When a thread is blocking (stop/semaphores).
    - When a thread is yielding.
  - Lower the priority (until it hits bottom)
    - When a thread is preempted.
- Priorities are never raised.

Implementation Hints

- Inspect your code: any variable or data structure that could be concurrently accessed should be protected.
- Use semaphores and alarm callbacks to implement sleeping/waking.
- TCBs will need two more extensions: priority and a semaphore to control sleeping/waking.
- Semaphore will need one more extension: a TAS lock variable.

Grading

- Correctness
  - Avoid race conditions.
  - Preemption and interrupt handler must work correctly.
  - Do not leak memory.
- Efficiency
  - Interrupts should be disabled for short periods of time.
  - Don't disable interrupts unnecessarily.
  - Interrupt handler processing should be fast.
  - Schedule the idle thread only when there is nothing more to schedule.
- Elegance
  - Your code should be modular and easy to understand.
- Test suite
  - You should write a test suite for your code.
- A clean compile will be worth 10%.

Concluding Advice

- Start early.
- Work incrementally.
- Test thoroughly.
- E-mail course staff if you have any questions, or come for office hours.