Project 2
Adding Preemption

Robert Escriva

Slide heritage: Previous TAs → Krzysztof Ostrowski

Cornell CS 4411, September 20, 2010
Project 1 due Wednesday at 11:59PM.

Project 2 no sooner than one week from today at 11:59PM.

Web page is being updated frequently; check for updates.

Email cs4410staff@systems.cs.cornell.edu for help.
1 Project Scope

2 Implementation details
   - Interrupts
   - Adding synchronization
   - More on interrupts
   - Alarms
   - Sleeping with timeout
   - Multilevel Scheduling

3 Concluding Thoughts (Grading)
What are does adding preemption involve?

1. Make your code threadsafe.
2. Install the interrupt handler.
3. ???
4. Profit!*

*Profit will come in the form of grades
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.
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 come at any time.
     - Our synchronization method of choice: disabling interrupts.†
   - Switch threads in the interrupt handler.

†You only really need to disable interrupts in minithread.c
Implementation plan

3 Add alarms.
   - Create software structure(s) to track pending alarms.
   - Use the software clock to measure elapsed time.
   - Start firing alarms from the clock interrupt handler.

4 Add sleeping.

   minithread_sleep_with_timeout(int delay);

   - Register alarms, block/unblock threads.
5  Add multi-level feedback scheduling.
   - Implement multilevel feedback queues.
     - Use a regular queue as the underlying structure.
     - Add a cyclic search for dequeue.
   - 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.
Definitions:

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

Sample clock handler:

```c
void clock_handler(void* arg) {
}
```
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!**
Definitions for changing interrupts:

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

Strongly recommended usage:

```c
interrupt_level_t intlevel =
    set_interrupt_level(DISABLED)
    do_something();
set_interrupt_level(intlevel);
```
Keeping Time

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

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

- **Measuring elapsed time**
  - System functions are way too slow.
  - Software clock: just count interrupts.

    ```c
    extern long ticks;
    ```
How are interrupts processed?

- Always execute in the context of a thread...
  ... that happened to be running when the interrupt was triggered.
- The process of an interrupt:
  - Current state is saved on the stack of the running thread.
  - Handler is called.
  - After the handler completes, the saved state is restored.
Interrupts and System Calls

- Windows’ system libraries are not (mini)thread-safe...
  ... so interrupts are disabled (underneath, not by you) while the process is inside system calls.

- What happens if e.g. a thread spends a lot of time 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 scheduler’s data-structures.
- Multiple threads could concurrently try accessing the same structures.
- The clock handler needs to access the same global structures (so that it may preempt threads).
Synchronization Strategies

- **What not to use: spin locks**
  - Cannot use with interrupts disabled.
    - Active waiting is time consuming.
    - If we’re consuming processor time, who will unlock the lock?

- **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.
Unmatched enabling/disabling.
- Your function could be called with interrupts disabled (enabling them would compromise your system’s safety).
- Application code should *never* run with interrupts disabled.

Disabling interrupts unnecessarily.
- You should use better synchronization methods outside *minithreads.c*

Disabling interrupts for too long.
Implementing Alarms

**What you need to implement:**

```c
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.
  - Interrupts are disabled in the interrupt handler.
  - You cannot spend much time in your callback.
  - You cannot block.

- 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:

```c
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 (on the ready queue); a context switch is unnecessary.
You should use the alarm functions.
You should use semaphores instead of `minithread_start()` and `minithread_stop()`
- This is more-modular structure.
Avoid race conditions‡:
- Side effects of this function should be atomic.

‡It’s good practice to spot the race condition
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);
MLQ Structure

- **Level 1**: Round robin within a level
- **Level 2**
- **Level 3**
- **Level 4**: Priority scheduling between levels...

...but we’re not always starting at the highest level (no starvation)
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.
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.
- Any other reasonable policies?
Grading

- **Correctness**
  - Avoid race conditions.
  - User interrupts 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.
  - Use semaphores where possible.

- **Elegance**
  - Your code should be modular and easy to understand.
Advice

- Start early.
- Work incrementally.
- Test thoroughly.