Project 1
Non-Preemptive Multitasking (with minithreads)

Robert Escriva

Slide heritage: Previous TAs → Owen Arden

Cornell CS 4411, September 13, 2010
Announcements

- Tentative due dates published.
- Tentative lecture schedule published.
- Web page is being updated frequently; check for updates.
- Project 2 will be released before Saturday.
- Email cs4410staff@systems.cs.cornell.edu for help.
1. Project Scope

2. Implementation details
   - Queues
   - Minithread structure
   - Semaphores

3. Concluding Advice
What are minithreads?

- User-level threads for Windows NT/2000/XP
  - Windows only provides kernel-level threads; user-level threads can perform better in some cases.
Goals of this project

- Learn how threading and scheduling work.
- Actually implement said processes.*

*“In theory, there is no difference between theory and practice. But, in practice, there is.” Jan L. A. van de Snepscheut
Deliverables

- A working implementation of minithreads.
- Required pieces (we recommend this order for implementation)
  - FIFO Queue with “O(1)” append/prepend/dequeue.
  - Non-preemptive threads and FCFS scheduling.
  - Semaphore implementation.
  - A simple "food services" application.
- Optional (for those itching to start part II):
  - Add preemption.
  - Optional material is not graded (yet); focus on getting Part 1 right.
Interfaces for the queue (queue.h), minithreads (minithread.h), and semaphores (synch.h).

Machine specific parts (machineprimitives.h).
  - Context switching, stack initialization, etc.

Simple (non-exhaustive) test applications.
  - Statistically, there are a large number of untested potential bugs.
  - Write some tests of your own (be abusive to minithreads; it can take it).
Project 1

Implementation details

Minithreads structure

- minithread.h
- minithread.c*
- machineprimitives.h
- machineprimitives.c
- synch.h
- synch.c*
- queue.h
- queue.c*
- interrupts.h
- interrupts.c

* files to finish implementing
Singly- or doubly-linked lists can both satisfy $O(1)$.

- `any_t` is really `void *`
  - Allows the queue to hold arbitrary data (that is the size of a pointer).

- `queue_dequeue` takes `any_t *`.
Examples of `queue_dequeue`

Usage:

```c
any_t  datum = NULL;
queue_dequeue(run_queue, &datum);
/* check return value */
```

Internals:

```c
int queue_dequeue(queue_t queue,
                  any_t* item) {
    *item = queue->head->datum;
}
```
Need to create a Thread Control Block (TCB) for each thread.

The TCB must have:

- Stack top pointer (saved esp).
- Stack base pointer (given to us by \texttt{minithread\_allocate\_stack}).
- Thread identifier.
- Anything else you find useful.
Operations to implement *(minithread.c)*

```c
minithread_t minithread_fork(proc, arg);
    Create a thread and make it runnable.

minithread_t minithread_create(proc, arg);
    Create a thread and but don’t make it runnable.

void minithread_yield();  Let another thread in the run queue run.

void minithread_start(minithread_t t);  Start another thread.

void minithread_stop();  Stop yourself.
```
Creating minithreads

Two methods

- `minithread_t minithread_create(proc, arg);`
- `minithread_t minithread_fork(proc, arg);`

`proc` is a `proc_t` (a function pointer)

```c
/* the definition of arg_t */
typedef int* arg_t;
/* the definition of proc_t */
typedef int (*proc_t) (arg_t);
/* how you declare a proc_t */
int run_this_proc (arg_t arg);
```
We give you functions to allocate and initialize the stack. Here’s how they are defined:

```c
void minithread_allocate_stack
    (stack_pointer_t *stackbase,
     stack_pointer_t *stacktop);

extern void minithread_initialize_stack
    (stack_pointer_t *stacktop,
     proc_t body_proc,
     arg_t body_arg,
     proc_t final_proc,
     arg_t final_arg);
```
Sets up your stack to look as though a context switch occurred.

<table>
<thead>
<tr>
<th>stack_base</th>
<th>0xff0</th>
<th>final_proc addr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xfec</td>
<td>final_arg</td>
</tr>
<tr>
<td></td>
<td>0xfe8</td>
<td>body_proc addr</td>
</tr>
<tr>
<td></td>
<td>0xfe4</td>
<td>body_arg</td>
</tr>
<tr>
<td>stack_top</td>
<td>0xfe0</td>
<td>root_proc addr</td>
</tr>
</tbody>
</table>
**final_proc**

- **final_proc** is responsible for cleaning up the TCB, and stack after your thread terminates.
- It’s not safe for a thread to free its own stack or TCB.
- **Solution:** Dedicated cleanup thread.
  - It should **wait** for threads to be ready for cleanup; otherwise it should be blocked.
Context switching

- Swap the currently executing thread with one from the run queue.
- State to save:
  - Registers
  - Program counter
  - Stack pointer

```c
void minithread_switch
  (stack_pointer_t *old_thread_sp,
   stack_pointer_t *new_thread_sp);
```
Before starting a context switch

old_thread_sp  esp  new_thread_sp

state
Push old context

old_thread_sp  esp  new_thread_sp

state

state

state
Change stack pointers

old_thread_sp → esp → new_thread_sp

state

state
Pop off new context

old_thread_sp  esp  new_thread_sp

state
We haven’t specified any preemption. We need a way to voluntarily switch between threads.

```c
void minithread_yield();
```

Use `minithread_switch` to implement `minithread_yield`.

What happens to the yielding thread?
Initializing minithreads

```c
void minithread_system_initialize
  (proc_t mainproc,
   arg_t mainarg);
```

- Starts up the system, and initializes global datastructures.
- Creates a thread to run `mainproc(mainarg)`
- This should be where all queues, global semaphores, etc. are initialized.
What about our Windows thread?

- We have a kernel thread used to call `minithread_system_initialize`. What should I do with it?
  - Re-use this thread as one of your behind-the-scenes threads.
  - Be careful not to cleanup or exit this thread.

- The program should never really exit, so it is a good idea to use the Windows thread (which never should be terminated) as the idle thread.
Semaphores

semaphore_t semaphore_create(); Create a semaphore (and allocate its resources).

void semaphore_destroy(semaphore_t); Destroy a semaphore (and free its resources).

void semaphore_initialize(semaphore_t, int); Set the initial value of a semaphore (how many semaphore_P functions may be called without blocking).

void semaphore_P(semaphore_t); Decrements a semaphore; (block value \( \leq 0 \)).

void semaphore_V(semaphore_t); Increments a semaphore, unblocking a thread that is blocked on it.
Semaphores

Semaphore Properties

- Calling `semaphore_P` will decrement `value` and MUST block when `value ≤ 0`.
- Calling `semaphore_V` will increment `value` and MUST unblock one thread that is waiting on the semaphore.
Submitting your work

- Include a README file with your names and net IDs.
- Write SHORT notes about anything you think we should know (e.g. broken code).
- This README should be nearly empty as all of your code should work and be well-tested.
Concluding Advice

- Manage your memory and pointer, for they are the key to bug-free code.
- Write clean and understandable code.
  - Variables should have proper names (e.g. `stack_pointer` not `lol`)
  - Provide meaningful comments (but do not comment in excess).
- Make your intentions clear. Do not make us make assumptions about what you wrote. This is a simple project, and we should be able to understand what you are doing with minimal effort.
- Do not terminate when program threads are done.
  - Idle threads never terminate.