Project 1
Non-Preemptive Multitasking (with minithreads)

Sean Ogden

Slide heritage: Previous TAs → Robert Escriva → Zhiyuan Teo → Ayush Dubey

Cornell CS 4411, September 6, 2013
Announcements

- Project 1 will be published today, due September 22, 2013.
- Make sure you are added on CMS for CS 4411, and that you have been assigned a group partner.
- Students without partners who haven’t contacted us will be purged from CMS soon (see course webpage for list).
- No formal lecture next week; instead an FAQ session with some tips.
- Email cs4410staff@systems.cs.cornell.edu for help.
Outline

1 Project Scope

2 Implementation details
   - Queues
   - Minithread structure
   - Semaphores

3 Concluding Advice
Goals of this project

- Learn how threading and scheduling work.
- Learn simple synchronization primitives.
- 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 "retail shop" application.
- Optional (for those itching to start part II):
  - Add preemption.
  - Optional material is not graded (yet); focus on getting Part 1 right.
What are minithreads?

- User-level threads for Windows/OSX/Linux
  - User-level threads can perform better in some cases.
  - User-level threads can also be useful in OSes that do not provide kernel level threads.
Kernel threads
User threads
Starting point

- Interfaces for the queue \((\text{queue.h})\), minithreads \((\text{minithread.h})\), and semaphores \((\text{synch.h})\).
- Machine specific parts \((\text{machineprimitives.h})\).
  - 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).
Outline

1. Project Scope

2. Implementation details
   - Queues
   - Minithread structure
   - Semaphores

3. Concluding Advice
Minithread structure

```
minithread.h
minithread.c*
machineprimitives.h
machineprimitives.c
synch.h
synch.c*
queue.h
queue.c*
```

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

- Data in queue is stored as `void *` allowing the queue to hold arbitrary data (that is the size of a pointer).

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

Usage:

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

Internals:

```c
int queue_dequeue(queue_t queue,
                   void **item) {
    *item = queue->head->datum;
}
```
Minithread structure

- 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 `minithread_allocate_stack`).
  - Thread identifier.
  - Anything else you find useful.
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(); Voluntarily give up CPU; let another thread in the run queue run.
void minithread_start(minithread_t t);

Makes a thread runnable by putting it onto the ready queue. Useful in semaphore operations, or to start a thread after it has been created through minithread_create().

void minithread_stop();

Stops running a thread immediately (ie blocks the thread); the next scheduled thread on the ready queue should run. Also useful in semaphore operations.
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>final_proc addr</th>
<th>final_arg</th>
<th>body_proc addr</th>
<th>body_arg</th>
<th>root_proc addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfee8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfee4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xfe0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Context switching

- Swap the currently executing thread with one from the run queue.
- State to save:
  - Registers
  - Program counter
  - Stack pointer
- We give you a function for this:

```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

Diagram showing the structure before context switch.
Push old context

old_thread_sp  esp  new_thread_sp

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
Yielding a thread

- 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?
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.
Summary of minithread lifecycle

1. \texttt{minithread\_fork(body\_proc, args)}
   - thread starts, \texttt{body\_proc(args)} is called.

2. \texttt{minithread\_yield()}
   - thread gives up control of CPU

3. another thread yields, and this thread resumes execution again
   - control resumes from instruction after \texttt{minithread\_yield()}

4. thread terminates by executing return from within \texttt{body\_proc}.

5. when \texttt{body\_proc} returns, epilogue code \texttt{final\_proc(args)} is immediately called.
   - This code should wake up the cleanup thread to free the stack and TCB.
   - A context switch should be made to the next runnable thread.
**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.
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.
How to reuse the original stack for the idle thread

- Create a TCB for the idle thread in `minithread_system_initialize`.
- In the TCB, set `stacktop` and `stackbase` to `NULL`.
  - Don’t need `stacktop` because the stack is already initialized.
  - Don’t need `stackbase` because the stack will never be freed.
- What code should the idle thread execute?
A quick primer on concurrency

- Race condition: result of computation depends on the relative running speed of threads.
  - Multiple concurrent threads reading from/writing to the same memory location.
  - E.g. two threads manipulating a linked list.

- Atomic operation: either the operation goes to completion, or fails altogether.
Solution: synchronization

- We want critical section of code to run without other threads interfering.

```c
queue process_queue;
lock process_queue_lock;
void manipulate_queue {
    lock_acquire (process_queue_lock);
    /* critical section begins */
    queue_dequeue (process_queue);
    queue_append (minithread_self);
    /* critical section ends */
    lock_release (process_queue_lock);
}
```

- Beware: deadlock and starvation!
Semaphores

- A synchronization primitive used to limit the number of threads accessing a shared resource.
- You decide how many threads can concurrently hold the semaphore when initializing it.
- Semaphore value is manipulated atomically:
  - `semaphore_P`: decrements the value by 1, if value becomes $\leq 0$ blocks the thread (wait)
  - `semaphore_V`: increments the value by 1, if value was $\leq 0$ then unblocks one waiting thread (signal)
- Special case: binary semaphore is a lock.
Semaphore operations

```c
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 if value ≤ 0 before decrementing).
void semaphore_V(semaphore_t);  // Increments a semaphore, unblocking a thread that is blocked on it.
```
Outline

1. Project Scope

2. Implementation details
   - Queues
   - Minithread structure
   - Semaphores

3. Concluding Advice
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.
  - Good luck!
Project 1
Non-Preemptive Multitasking (with minithreads)

Sean Ogden
sean @ cs
September 6, 2013