Pintos Task 1: Threads

Paul Marinescu, Mark Rutland

Imperial College London

February 3, 2011

Introduction

Goal

Make Pintos more amenable for multithreading

Tasks

i. Non-busy sleep
ii. Priority scheduling
iii. Priority donation
iv. BSD-style scheduler

Getting Started

Where do I work?

- All work should be done under src/threads/ & src/devices/
- Compilation should occur under src/threads
- Any new files should be added to src/Makefile.build

Testing

- Tests can be run with: `make check`
- Single tests can be run with
  `make build/tests/threads/test.result`
- Some tests (mlfqs-*) take much longer to run than others

Important Files

Files to (possibly) modify:

Most of your work will be done in the 'threads' directory.

i. threads/thread.c
   - Handles creation, modification, and destruction of threads
   - Scheduling code goes here

ii. threads/synch.c
   - Contains basic synchronisation constructs for multithreading

iii. devices/timer.c
   - Handles busy sleeping
   - Calls `thread_tick()` in threads/thread.c once every timer tick

Important Files

Files with useful functions

i. `<debug.h>`
   - ASSERT() macro
   - UNUSED qualifier macro

ii. `<list.h>`
   - Generic linked-list implementation

iii. `<stdio.h>`
   - In-kernel implementation of `printf()`

iv. `"threads/interrupt.h"`
   - `intr_yield_on_return()` makes the current thread yield after returning from an interrupt

v. `"threads/thread.c"`
   - `thread_tick()` called every timer tick

Pintos Initialisation

Initial thread

- Initial thread (threads/init.c: `main()`) is started by the boot loader
- Initial thread starts threads subsystem and 'promotes' itself to a standard Pintos thread in `thread_init()`
- Initialises other subsystems
- Parses command-line arguments
- Starts other threads, via `thread_create()`

Warning

The initial thread is not created by `thread_create()`. You may need to add initialisation code for any structures you add to `thread_init()` also.

Threads in Pintos

How are threads implemented in Pintos?

- Threads defines as `struct thread` in src/threads/thread.h
- Each thread is stored at the start of a 4KB page
- The rest of the page is used as stack space for the thread

Scheduling

- Preemptive
- Next thread chosen by `next_thread_to_run()`
- Context switch is handled via assembly code, you should not modify it

Threads in Pintos

Thread states

Threads exist in 1 of 4 possible states at any given time:

- THREAD_RUNNING
  only the case for the current running thread (thread_current())
- THREAD_READY
  ready to be scheduled, but not currently running (in ready_list)
- THREAD_BLOCKED
  unable to be run, waiting on some event (e.g. semaphore up)
- THREAD_DYING
  finished with execution, waiting to be destroyed by the kernel

Synchronisation

Pintos synchronisation constructs

Pintos offers several general-purpose synchronisation mechanisms (in threads/synch.c):

- Semaphores
- Locks
- Monitors

Interrupt handler synchronisation

- Disabling interrupts

Warning

Do not disable interrupts unless you need exclusive access to data modified by an interrupt handler. Disabling interrupts is not a general-purpose synchronisation mechanism. You **will** lose marks for using it as such.

Project Requirements

- Non-busy sleep
- Priority scheduling
  - Allow processes to modify & query their priority
- Priority donation for locks
- BSD-style scheduler
  - Fixed-point math routines

Non-busy sleep

Idea

While a thread is sleeping, it must not consume any CPU.

`timer_sleep(int64_t ticks)`

- Blocks the calling thread
- The thread must be unblocked after `ticks` ticks have occurred
- A call with a negative value for `ticks` should not block
- Allows other threads to run whilst calling thread is blocked
- Multiple threads may call `timer_sleep()` simultaneously

Warning

Part of your code will be in an interrupt handler. Think about shared data that must be protected from concurrent access.

Priority scheduling

Idea

Make sure that at any point in time, the highest priority ready thread is running.

Yielding the processor

At certain points, another thread may have the highest priority. This could occur when:

- A new thread is created
- A thread is unblocked from a synchronisation construct
- A thread is woken from `timer_sleep`

If this happens the current thread should yield to it **immediately**

Warning

Sleeping threads should not be woken early, regardless of priority.

Priority donation

Idea

Prevent priority inversion by allowing high-priority threads to 'donate' their priority to other threads holding contended resources.

Donation model

- Multiple threads may donate to a single thread
- A thread can only donate to one thread (which holds the resource)
- The donation is revoked when the resource is freed
- Donations may nest (A donates to B donates to C)

Warning

Blocked threads may have their priorities modified! May want to consider for semaphore re-design.

Allow processes to modify & query their priority

Why?

To make effective use of priority-based scheduling. Tests rely on processes being able to set their priority, and querying the (effective) priority of a thread is useful for debugging.

Functions to implement

- `thread_set_priority(int new_priority)`
  sets the thread's priority to new_priority, if valid
- `thread_get_priority(void)`
  returns the current thread's (effective) priority

Interaction with donation

A thread's returned priority should always be the highest from its set priority and donations.

BSD-style scheduler

Idea

Another way of preventing priority inversion, using pre-emptive scheduling.

- Measure CPU usage of each thread every tick
- Decay CPU usage for all threads once per second
- Calculate system load average once per second
- Update priority every 4th tick
- Run thread with highest priority

Warning

Do not disable the priority scheduler with `#ifdefs`. Instead, switch between schedulers at runtime based on the `thread_mlfqs` boolean variable.

Fixed-point math routines

Idea

Simulate real arithmetic without using the FPU.

Correctness

- Abstract out the actual arithmetic to make testing easier
- Small differences in implementation (round to 0 vs round up) can make huge differences in output

Efficiency

- This code will be run many times a second, being optimal is essential
- Functions vs macros may give different performance characteristics

Very specific examples are given are given in section B.6 (Fixed-Point Real Arithmetic) of the Pintos-IC document.

Suggested order of implementation

Following the suggested order may save you time in debugging.

i. Non-busy sleep
ii. Initial `thread_get_priority()` & `thread_set_priority()`
iii. Prioritised unblocking on synchonisation primitives
iv. Priority scheduling
v. Priority donation
vi. Fixed-point math routines
vii. BSD-style scheduler

Tips

1. Make the design first
2. Try to keep it simple (e.g., heap vs list)
3. Avoid code duplication
4. Verify error codes
5. Keep in mind the context in which functions expect to be called