Project 1: Threads

Jin-Soo Kim (jinsoookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Pintos Kernel (1)

- The current Pintos kernel
  - There is only one address space
  - There can be a number of threads running in the kernel mode
  - All the kernel threads share the same address space

<table>
<thead>
<tr>
<th># threads per addr space:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS Early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>The current Pintos</td>
<td>Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX</td>
<td></td>
</tr>
</tbody>
</table>
Pintos Kernel (2)

- **Address space**
  - Up to 64MB of physical memory
  - The kernel maps the physical memory at PHYS_BASE (0xc000 0000)

```
static inline void *ptov (uintptr_t addr) {
    return (void *) (paddr + PHYS_BASE);
}
static inline uintptr_t vtop (void *addr) {
    return (uintptr_t) vaddr - (uintptr_t) PHYS_BASE;
}
```
Kernel thread

- The kernel maintains a TCB (Thread Control Block) for each thread (struct thread)
- Created using thread_create()

```
tid_t thread_create (const char *name, int priority, thread_func *function, void *aux);
```

- Allocate a page (4KB) for thread stack
- Initialize TCB
- Add TCB to the run queue
- Return the corresponding tid

- The function running_thread() returns the pointer to the TCB of the current thread
Pintos Kernel (4)

- TCB (Thread Control Block)

```c
struct thread *running_thread() {
    Get %esp;
    return (%esp & 0xffffffff00);
}
```
Thread states

- Refer to Appendix A.2: Threads
**Pintos Kernel (6)**

- **Ready queue**

```
all_list

- tid = 1
  THREAD_RUNNING
  allelem
  elem
  ...

- tid = 2
  THREAD_BLOCKED
  allelem
  elem
  ...

- tid = 3
  THREAD_RUNNING
  allelem
  elem
  ...

- tid = 4
  THREAD_RUNNING
  allelem
  elem
  ...
```

ready_list
Pintos Kernel (7)

- List management in Pintos
  - `#include <list.h> /* src/lib/kernel/list.h */`
  - A type oblivious, easy-to-use, circularly-linked list
List management in Pintos (cont’d)

- `list_init (struct list *list);`
  - Initializes list as an empty list
- `list_push_front (struct list *list, struct list_elem *elem);`
  - `list_push_back (struct list *list, struct list_elem *elem);`
  - Inserts elem at the beginning (end) of list
- `list_remove (struct list_elem *elem);`
  - Removes elem from its list
- `list_pop_front (struct list *list);`
  - `list_pop_back (struct list *list);`
  - Removes the front (back) element from list and returns it
- `list_entry (LIST_ELEM, STRUCT, MEMBER);`
  - Converts pointer to list element LIST_ELEM into a pointer to the structure that LIST_ELEM is embedded inside.
List management example

- Display thread list (tid & name)

```c
void list_thread ()
{
    struct list_elem *e;

    for (e = list_begin(&all_list);
         e != list_end(&all_list);
         e = list_next(e))
    {
        struct thread *t =
            list_entry (e, struct thread, allelem);
        printf ("%d: %s\n", t->tid, t->name);
    }
}
```

((cf.) http://isis.poly.edu/kulesh/stuff/src/klist/)
Project 1: Threads

- **Requirements**
  - Alarm clock
  - Priority scheduling
  - Priority donation
  - Advanced scheduler

- **Test cases to pass (total 27 tests)**
Alarm Clock (1)

- **Reimplement timer_sleep()**

  ```c
  void timer_sleep (int64 x);
  ```

  - Suspends execution of the calling thread until time has advanced at least \( x \) timer ticks
  - The current version simply "busy waits."
    - The thread spins in a loop checking the current time and calling `thread_yield()` until enough time has gone by.
  - Reimplement it to avoid busy waiting
  - You don’t have to worry about the overflow of timer values.
Alarm Clock (2)

- Time management in Pintos
  - On every timer interrupt, the global variable ticks is increased by one
    - The variable ticks represent the number of timer ticks since the Pintos booted
    - Timer frequency: TIMER_FREQ (= 100) ticks per second (defined in <src/devices/timer.h>)
  - The time slice is set to TIME_SLICE (= 4) ticks for each thread (defined in <src/threads/thread.c>)
  - timer_interrupt(): Timer interrupt handler
    - Increase the ticks variable
    - If the current thread has exhausted its time slice, call thread_yield().
The current timer_sleep() implementation

- In <src/devices/timer.c>
- timer_ticks() returns the current value of ticks

```c
int64_t timer_elapsed (int64_t then)
{
    return timer_ticks () - then;
}

void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);

    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```
### Hints

- Make a new list of threads ("waiting_list")
- Remove the calling thread from the ready list and insert it into the "waiting_list" changing its status to THREAD_BLOCKED
- The thread waits in the "waiting_list" until the timer expires
- When a timer interrupt occurs, move the thread back to the ready list if its timer has expired.
- Use `<list.h>` for list manipulation
Priority Scheduling (1)

- Scheduling
  - The scheduling policy decides which thread to run next, given a set of runnable threads

- The current Pintos scheduling policy: Round-robin (RR) scheduling
  - The ready queue is treated as a circular FIFO queue
  - Each thread is given a time slice (or time quantum)
    - \( \text{TIME\_SLICE} (= 4) \) ticks by default
  - If the time slice expires, the current thread is moved to the end of the ready queue
  - The next thread in the ready queue is scheduled
  - No priority: All the threads are treated equally
Priority Scheduling (2)

- The current Pintos scheduling

```c
/* Yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();
    if (cur != idle_thread)
        list_push_back (&ready_list, &cur->elem);
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```
The current Pintos scheduling (cont’d)

/* Schedules a new process. At entry, interrupts must be off and the running process's state must have been changed from running to some other state. This function finds another thread to run and switches to it.

It's not safe to call printf() until schedule_tail() has completed. */

static void
schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
        prev = switch_threads (cur, next);
    schedule_tail (prev);
}
Priority Scheduling (4)

The current Pintos scheduling (cont’d)

/* Chooses and returns the next thread to be scheduled. Should return a thread from the run queue, unless the run queue is empty. (If the running thread can continue running, then it will be in the run queue.) If the run queue is empty, return idle_thread. */

static struct thread *
next_thread_to_run (void)
{
    if (list_empty (&ready_list))
        return idle_thread;
    else
        return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
Priority Scheduling (5)

- **Priority scheduling**
  - Each thread is given a scheduling priority
  - The scheduler chooses the thread with the highest priority in the ready queue to run next

- **Thread priorities in Pintos**
  - 64 priority levels (default = 31)
  - Lower numbers correspond to lower priorities
    - Max priority = 63
    - Min priority = 0
  - The initial priority is passed as an argument to `thread_create()`
Priority Scheduling (6)

- **Note**
  - When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread.
  
  - A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.
  
  - When threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first.
Synchronization (1)

- **Synchronization problem**
  - Accessing a shared resource by two concurrent threads creates a situation called **race condition**
    - The result is non-deterministic and depends on timing
  - We need "**synchronization**" mechanisms for controlling access to shared resources
  - **Critical sections** are parts of the program that access shared resources
  - We want to provide **mutual exclusion** in critical sections
    - Only one thread at a time can execute in the critical section
    - All other threads are forced to wait on entry
    - When a thread leaves a critical section, another can enter
Synchronization (2)

- Synchronization mechanisms in Pintos
  
  • Locks
    - void lock_init (struct lock *lock);
    - void lock_acquire (struct lock *lock);
    - void lock_release (struct lock *lock);
  
  • Semaphores
    - void sema_init (struct semaphore *sema, unsigned value);
    - void sema_up (struct semaphore *sema);
    - void sema_down (struct semaphore *sema);
  
  • Condition variables
    - void cond_init (struct condition *cond);
    - void cond_wait (struct condition *cond, struct lock *lock);
    - void cond_signal (struct condition *cond, struct lock *lock);
    - void cond_broadcast (struct condition *cond, struct lock *lock);
  
  • Refer to Appendix A.3: Synchronization
Synchronization (3)

- **Locks**
  - A lock is initially free
  - Call `lock_acquire()` before entering a critical section, and call `lock_release()` after leaving it
  - Between `lock_acquire()` and `lock_release()`, the thread holds the lock
  - `lock_acquire()` does not return until the caller holds the lock
  - At most one thread can hold a lock at a time
  - After `lock_release()`, one of the waiting threads should be able to hold the lock
Synchronization (4)

- **Semaphores**
  - A semaphore is a nonnegative integer with two operators that manipulate it atomically
  - `sema_down()` waits for the value to become positive, then decrement it
  - `sema_up()` increments the value and wakes up one waiting thread, if any
  - A semaphore initialized to 1 is similar to a lock
  - A semaphore initialized to N (> 1) represents a resource with many units available
    - Up to N threads can enter the critical section
Synchronization (5)

- **Condition variables**
  - Condition variables allow a thread in the critical section to wait for an event to occur.
  - Condition variables are used with locks.
  - `cond_wait()` atomically releases lock and waits for an event to be signaled by another thread.
    - Lock must be held before calling `cond_wait()`.
    - After condition is signaled, reacquires lock before returning.
  - `cond_signal()` wakes up one of threads that are waiting on condition.
  - `cond_broadcast()` wakes up all threads, if any, waiting on condition.
Priority Donation (1)

- Priority inversion problem
  - A situation where a higher-priority thread is unable to run because a lower-priority thread is holding a resource it needs, such as a lock.
  - *What really happened on Mars?*

```
lock_acquire()
```

```
lock_release()
```

Bus management task

communications task

meteorological data gathering task

priority inversion
Priority Donation (2)

- Priority donation (or priority inheritance)
  - The higher-priority thread (donor) can donate its priority to the lower-priority thread (donee) holding the resource it requires.
  - The donee will get scheduled sooner since its priority is boosted due to donation.
  - When the donee finishes its job and releases the resource, its priority is returned to the original priority.
Priority Donation (3)

- Before priority donation

 Priority Donation

- After priority donation

Priority Donation
Priority Donation (4)

- **Multiple donations**
  - Multiple priorities are donated to a single thread

```
Thread H (P = 33)
create
acquire(a)
release(a)
exit

Thread M (P = 32)
create
acquire(a)
release(a)
exit

Thread L (P = 31)
create
acquire(a) acquire(b)
release(b)
release(a)
exit
```
Priority Donation (5)

Multiple donations example

```c
void
test_priority_donate_multiple (void)
{
    struct lock a, b;

    /* This test does not work with the MLFQS */
    ASSERT (!thread_mlfqs);

    /* Make sure our priority is the default. */
    ASSERT (thread_get_priority () == PRI_DEFAULT);

    lock_init (&a);
    lock_init (&b);

    lock_acquire (&a);
    lock_acquire (&b);

    thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);
    msg ("Main thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 1, thread_get_priority ());

    thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);
    msg ("Main thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 2, thread_get_priority ());

    lock_release (&b);
    msg ("Thread b should have just finished.");
    msg ("Main thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 1, thread_get_priority ());

    lock_release (&a);
    msg ("Thread a should have just finished.");
    msg ("Main thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT, thread_get_priority ());
}
```

```c
static void
a_thread_func (void *lock_)
{
    struct lock *lock = lock_;

    lock_acquire (lock);
    msg ("Thread a acquired lock a.");
    lock_release (lock);
    msg ("Thread a finished.");
}
```

```c
static void
b_thread_func (void *lock_)
{
    struct lock *lock = lock_;

    lock_acquire (lock);
    msg ("Thread b acquired lock b.");
    lock_release (lock);
    msg ("Thread b finished.");
}
```

src/tests/threads/priority-donate-multiple.c
Priority Donation (6)

- **Nested donation**
  - If H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H’s priority.
Priority Donation (7)

- Nested donation example

```c
void
test_priority_donate_nest (void)
{
    struct lock a, b;
    struct locks locks;

    /* This test does not work with the MLFQS. */
    ASSERT (!thread_mlfqs);

    /* Make sure our priority is the default. */
    ASSERT (thread_get_priority () == PRI_DEFAULT);

    lock_init (&a);
    lock_init (&b);

    lock_acquire (&a);

    locks.a = &a;
    locks.b = &b;
    thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
    thread_yield ();
    msg ("Low thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 1, thread_get_priority ());

    thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
    thread_yield ();
    msg ("Low thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 2, thread_get_priority ());

    lock_release (&a);
    thread_yield ();
    msg ("Medium thread should just have finished.");
    msg ("Low thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT, thread_get_priority ());
}

static void
medium_thread_func (void *locks_)
{
    struct locks *locks = locks_;

    lock_acquire (locks->b);
    lock_acquire (locks->a);

    msg ("Medium thread should have priority %d. Actual priority: %d.",
         PRI_DEFAULT + 2, thread_get_priority ());

    msg ("Medium thread got the lock.");

    lock_release (locks->a);
    thread_yield ();

    lock_release (locks->b);
    thread_yield ();

    msg ("High thread should have just finished.");
    msg ("Middle thread finished.");
}

static void
high_thread_func (void *lock_)
{
    struct lock *lock = lock_;

    lock_acquire (lock);
    msg ("High thread got the lock.");
    lock_release (lock);
    msg ("High thread finished.");
}
```

src/tests/threads/priority-donate-nest.c
Priority Scheduling/Donation

- **Hints**
  - Remember each thread’s base priority
  - When you schedule a new thread, find the thread with the highest priority among candidates
  - The “effective” priority of a thread can be greater than the base priority due to priority donation
  - The “effective” priority should be adjusted properly on `lock_acquire()` and `lock_release()`
  - You don’t have to implement priority donation for semaphores or condition variables
Advanced Scheduler

Overview

• Using Fixed-Point Number.
  – Real Number can be represented as two ways.
    » IEEE Floating pointer format.
    » Fixed-Point Number.

• 4.4 BSD Scheduler (Old Unix-Style)
  – About “NICE” Concept.
  – Priority QUEUE – Basically it is more likely with Hash.

• Those are well described on 91p. PintOS documentation.
Real Number representation

- **Floating point number (IEEE)**

  - S : Sign-bit (31)
  - F : Fraction-bit (30:23)
  - N : Number-bit (22:0)

  \[ (-1)^S \times M \times 2^E \]

  » You will learn this formula on System Programming
Real Number representation

- **Fraction Number**
  - What is it? The point position is fixed.
  - For example...

\[ b_{31}b_{30}b_{29}b_{28}b_{27}b_{26}...b_{10}.b_{9}b_{8}...b_{0} \]

Assume that this is the point.

- We will use this number representation.
4.4 BSD Scheduler

- Why BSD Scheduler?
  - It is based on mathematical.
  - Prevent starve → Guaranty fairness.
  - If you developed this project without this concept, the low priority thread may starve.

- Concept of “NICE”
  - BSD Scheduler has two concept of “priority”.
    - Relative Priority : Which is current priority
    - Absolute Priority : This is so called “NICE”
  - How “nice” the thread should be to other threads.
4.4BSD Scheduler

What is “Recent CPU” and “Load Average”?
- Recent CPU: Measure how much CPU time each process has received recently.
- Load average: load average is much like recent CPU but system-wide, not thread-specific.

Formulas for calculating Priority, Recent CPU, Load average. (91p ~ 93p)
- Priority = PRI_MAX − (recent_cpu / 4) − (nice * 2)
- Recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice
- Load_avg = (59/60)*load_avg + (1/60)*ready_threads
Hints

• Question! Which one is good data structure to build this scheduler?
  – Use PRI_MAX size of array linked list header.

• Before start implementation core area, implement fixed point arithmetic function.
  – It is well defined on 94p.

• Now you must implement calculating function.

• And now, think about it. How can we reduce the operation?
  – It is restriction. Your program must not charge many CPU usage for running scheduler. – It will increase load average.
  » It means, you must find out real good algorithm.
Applying jitter value

Overview

- Set timer interrupt in bochs at random intervals
  - Apply ‘-j seed’ option at pintos running
    $ pintos -j 10 -- run alarm-multiple

- In make operation
  - See line 56 in src/tests/Make.tests
    » TESTCMD += $(PINTOSOPTS)
  - When you run ‘make check’ or ‘make grade’, use this option
    $ make check PINTOSOPTS=’-j seed’
    $ make grade PINTOSOPTS=’-j seed’

- Those are well described on 4~5p. PintOS documentation.
Submission (1)

- **Due**
  - April 3, 11:59PM
  - Fill out the design document and save it with PDF format (GDHong_2012345678.pdf)
  - Tar and gzip your Pintos source codes
    - $ cd pintos
    - $ (cd src/threads; make clean)
    - $ tar cvzf GDHong_2012345678.tar.gz src
  - Upload them at sys.skku.edu
Submission (2)

- Submitting your report
  - Hand in the printed version of your design document (DESIGNDOC file) in the following class on April 3.
  - Good luck!