LAB 1: PINTOS THREADS
Overview

- Background: Pintos kernel
  - Address space
  - Kernel thread
  - Internal list structure

- Project 1: Threads
  - Alarm clock
  - Priority scheduling
  - Priority donation
  - Advanced scheduler
  - Tips
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
Address Space (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 (1)

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

```c
#include <stdlib.h>

int 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 thread_current() returns the pointer to the TCB of the current thread
Kernel Thread (2)

- TCB (Thread Control Block)

struct thread *t →

1 page = 4KB

%esp →

```
struct thread *running_thread() {
  Get %esp;
  return (%esp & 0xffffffff000);
}
```
Kernel Thread (3)

- Thread states
  - Refer to Appendix A.2: Threads

```
<table>
<thead>
<tr>
<th>Thread state</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREAD_READY</td>
<td>thread_create()</td>
</tr>
<tr>
<td>THREAD_RUNNING</td>
<td>schedule()</td>
</tr>
<tr>
<td>THREAD_BLOCKED</td>
<td>thread_block()</td>
</tr>
<tr>
<td>THREAD_DYING</td>
<td>thread_exit()</td>
</tr>
<tr>
<td></td>
<td>thread_unblock()</td>
</tr>
</tbody>
</table>
```

- thread_yield()
List management in Pintos

- `#include <list.h>` /* src/lib/kernel/list.h */
- Circularly-linked list

```c
struct list
{
    struct list *prev;
    struct list *next;
};
```

```c
struct list_elem
{
    struct list *prev;
    struct list *next;
};
```
Internal List Structure (2)

- List management example
  - Display thread list (tid & name)

```c
struct list all_list;

struct thread {
    tid_t tid;
    char name[16];
    ...
    struct list_elem allelem;
    ...
};

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/](http://isis.poly.edu/kulesh/stuff/src/klist/)
Internal List Structure (3)

- Ready queue

- Struct thread:
  ```
  struct thread{
    ......
    ....
    struct list_elem elem
    .....}
  ```

- All list

- Ready list

```

all_list

```
PROJECT 1: THREADS
Project 1: Threads

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

- Test cases to pass (total 27 tests)

- Official document:
  - [http://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15](http://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15)
Reimplement `timer_sleep()`

- 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.

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

```
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
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).
  - 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);
}
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)
  - Max priority = 63
  - Min priority = 0
  - The initial priority is passed as an argument to thread_create()
  - The change to highest priority should preempts other thread immediately
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.
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.
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

  - Thread H (P = 35)
  - Thread M (P = 33)
  - Thread L (P = 31)

- After priority donation

  - Thread H (P = 35)
  - Thread M (P = 33)
  - Thread L (P = 31)
Multiple donations

Multiple priorities are donated to a single thread
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.
Hints

- When you schedule a new thread, find the thread with the highest priority among candidates.
- You don’t have to implement priority donation for semaphores or condition variables.
- Remember each thread’s base priority:
  - Base priority is used for return value.
- Threads in nested donation should be on the same list.
Advanced Scheduler (1)

- 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
Why Advanced 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
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** = \((\frac{2*load\_avg}{2*load\_avg + 1}) \times recent\_cpu + nice\)
- **Load_avg** = \((\frac{59}{60})\times load\_avg + \frac{1}{60}\times ready\_threads\)
Real Number representation (1)

- Floating point number (IEEE)

- S : Sign-bit (31)
- F : Fraction-bit (30:23)
- N : Number-bit (22:0)
Real Number representation (2)

- Fraction Number
  - What is it? The point position is fixed
  - \( b_{31}b_{30}b_{29}b_{28}b_{27}b_{26}...b_{10}.b_9b_8...b_0 \)

  Assume that this is the point.

- We will use this number representation
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

- How can we reduce the operation?
  - Your program must not charge many CPU usage for running scheduler
    - It will increase load average » It means, you must find out pretty good algorithm
Tips

- **Applying jitter value**
  - 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

- **Due**
  - April 13, 11:59PM
  - Upload your source code and documentation at iCampus

- **Source code**
  - `tar cvzf [Name]_[ID]_source.tar.gz src`

- **Document**
  - File format – PDF format (NOT .hwp or .doc)
  - File name – `[Name]_[ID].pdf`

- **Tar and gzip your source code and document**

- **Good luck!**