Project 1: Threads

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Project 1-1 Review

- **Some groups did not add comments**
  - Each name to commission
  - You should comment Korean comments to a P 1-2

- **Some groups use tricks to pass the test cases**
  - Those groups will receive points of test server
  - P 1-2 test cases will operate abnormally
  - Some test cases can be modified or added

- **Some groups use strange skills for P 1-1**
  - Very similar to a last year project

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)

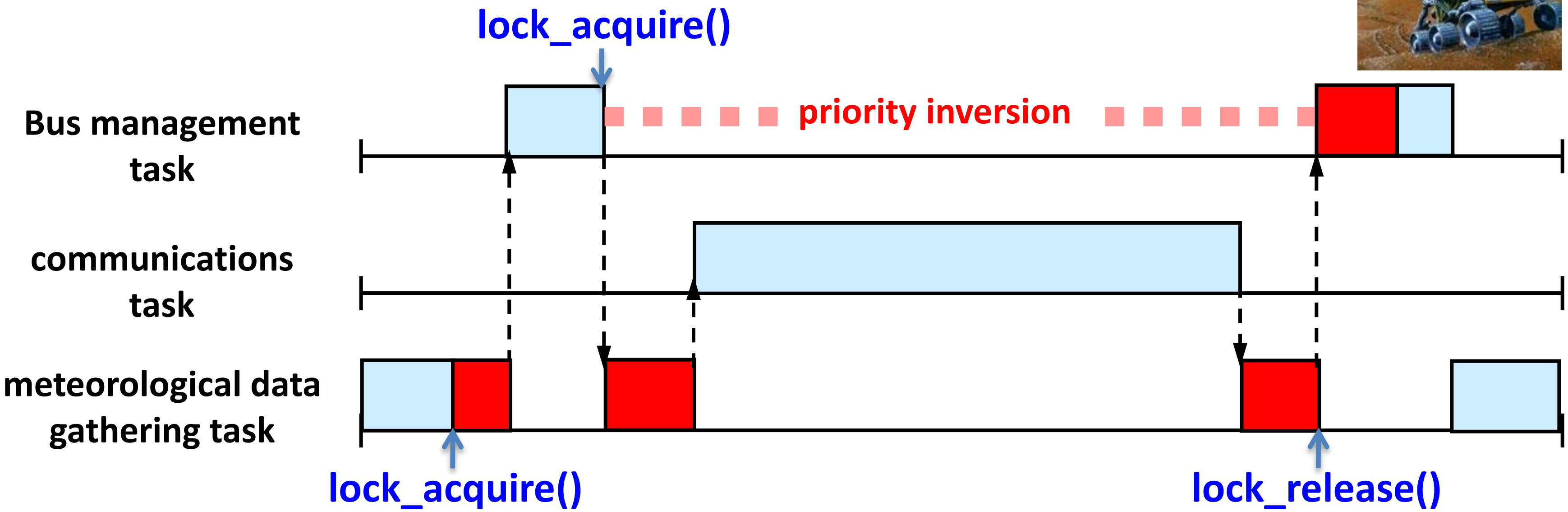
- **Semaphores**
  - A semaphore is a nonnegative integer with two operators that manipulate it atomically
  - `sema_down()` waits for the value to become positive, then decrement it
  - `sema_up()` increments the value and wakes up one waiting thread, if any
  - A semaphore initialized to 1 is similar 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?*

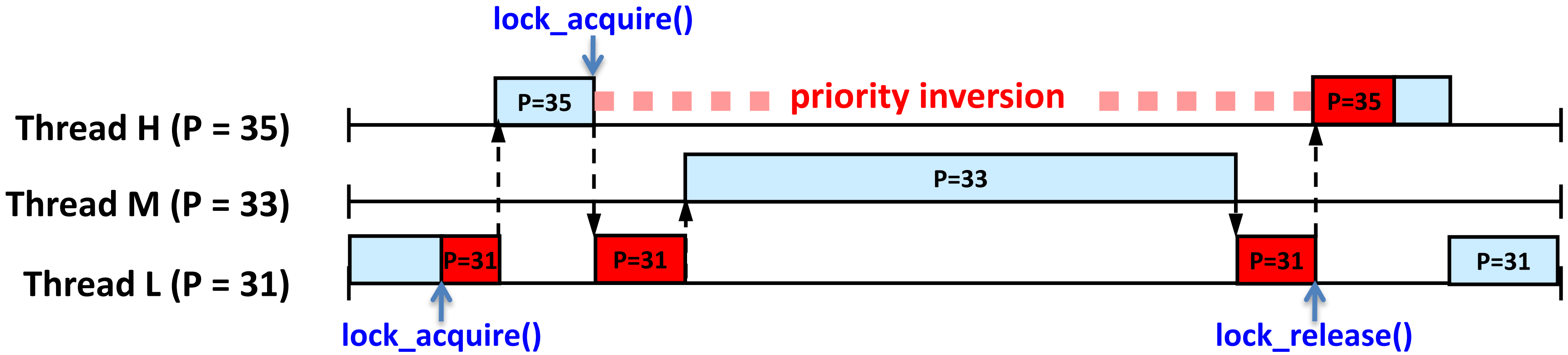


Priority Donation (2)

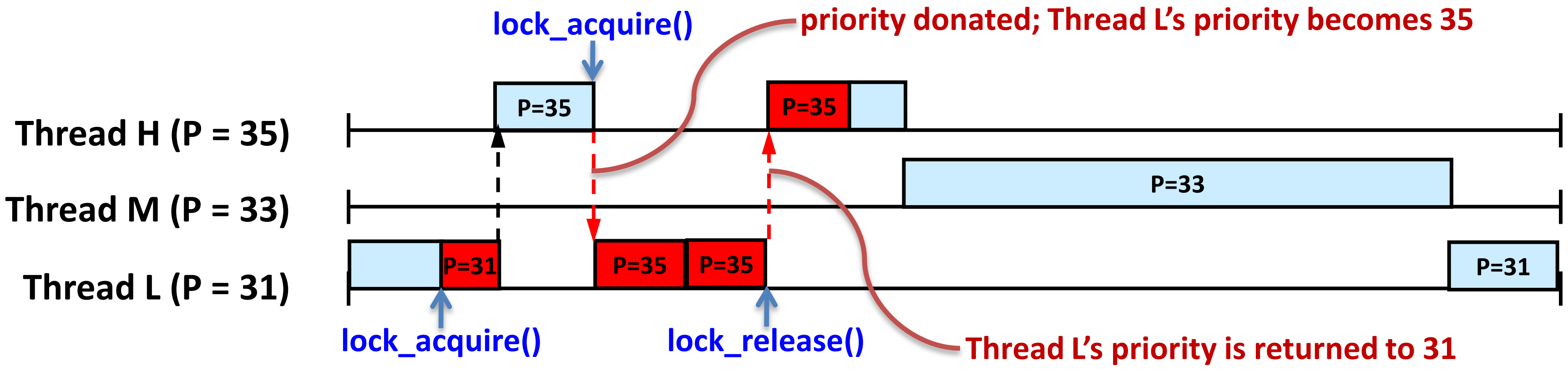
- **Priority donation (or priority inheritance)**
  - The higher-priority thread (donor) can donate its priority to the lower-priority thread (donee) holding the resource it requires.
  - The donee will get scheduled sooner since its priority is boosted due to donation
  - When the donee finishes its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

Priority Donation (3)

- Before priority donation

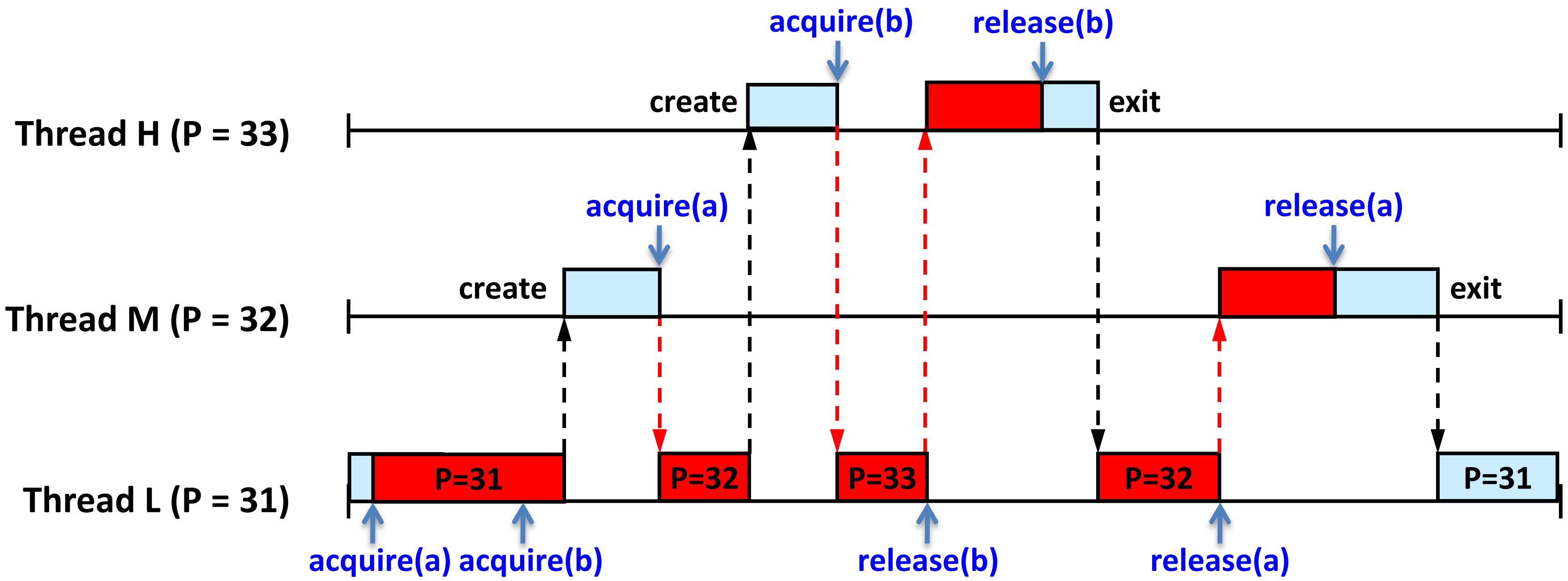


- After priority donation



Priority Donation (4)

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



Priority Donation (5)

- **Multiple donations example**

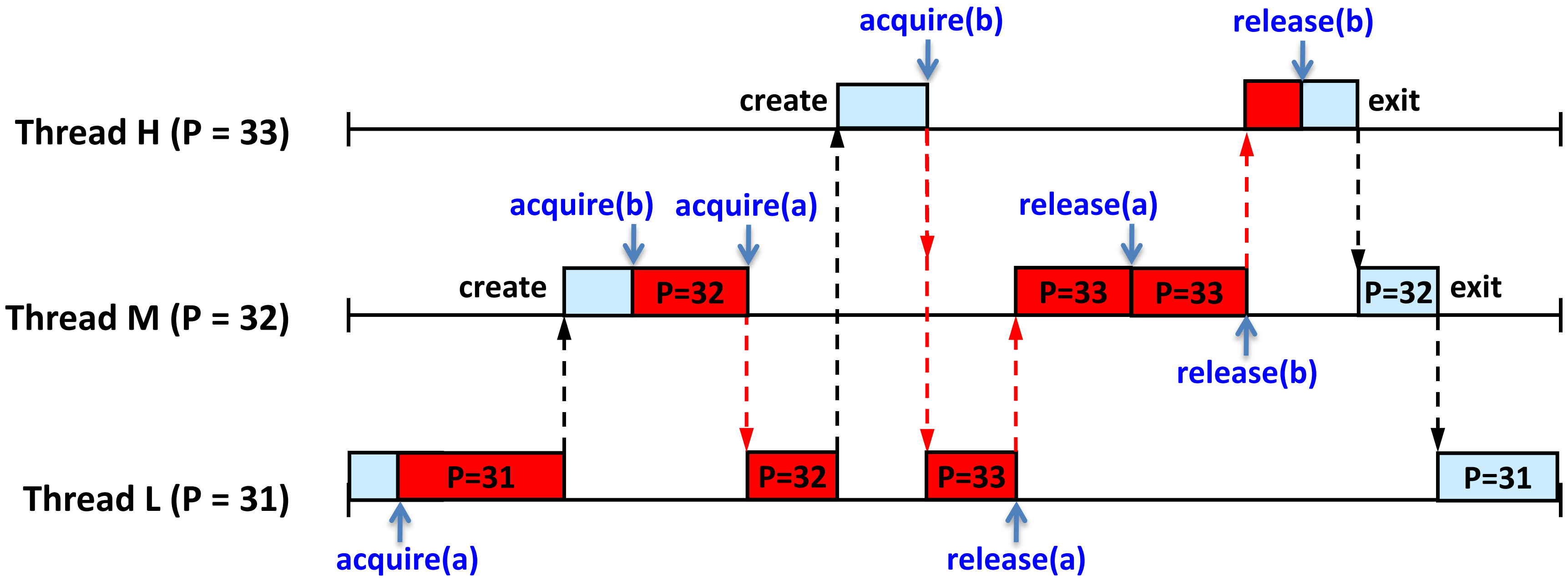
```
void
test_priority_donate_multiple (void)
{
  struct lock a, b;

  /* 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 ());
}
```

```
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.");
}

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`

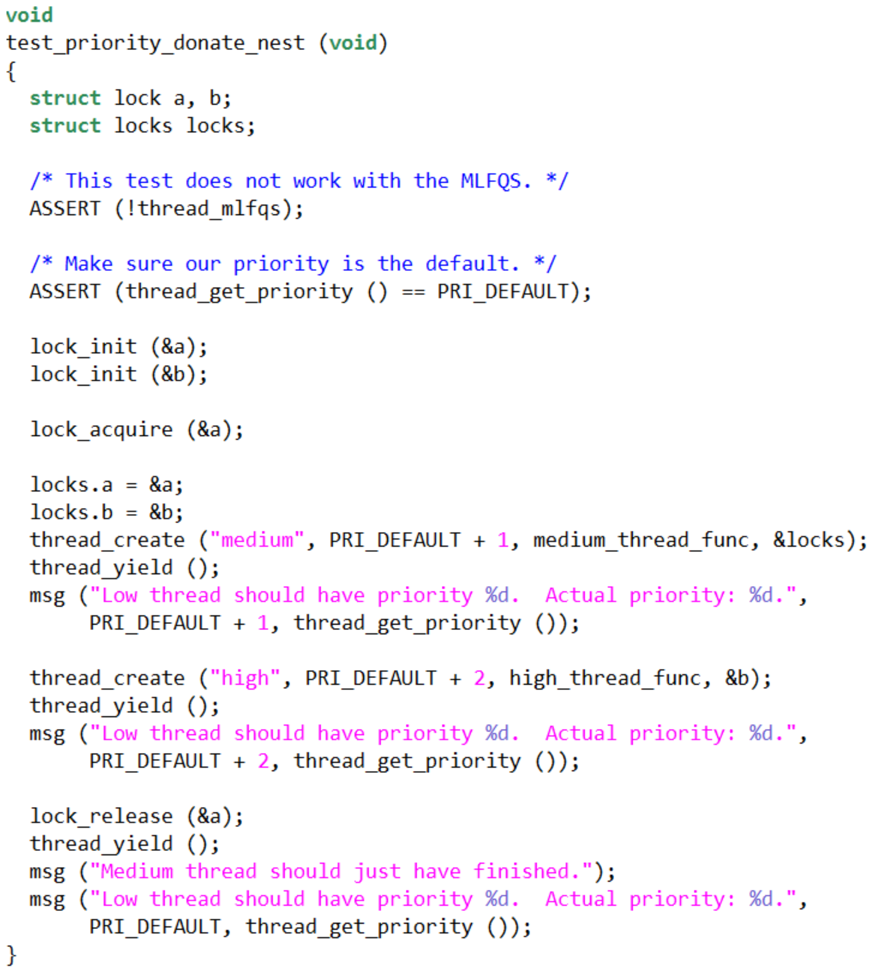
Priority Donation (6)

- **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

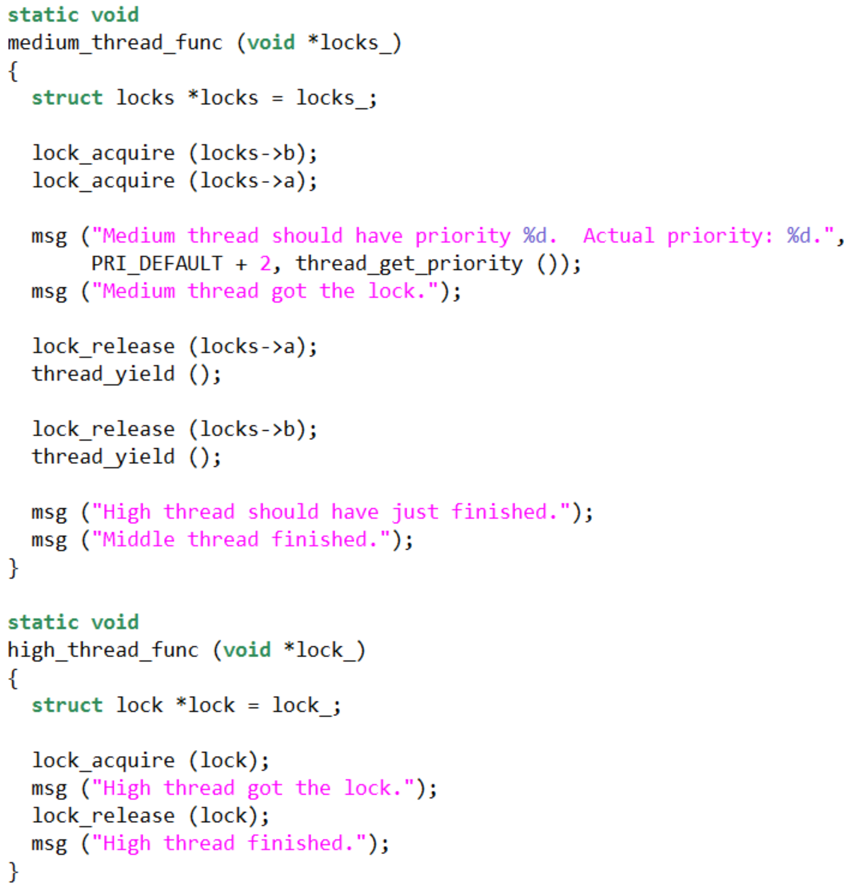


Priority Donation (7)

- 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 ());
}
```



```c
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

Hint

- **You must release the highest priority thread of waiting threads**
- **You don’t have to implement priority donation for semaphores or condition variables.**
  - Mutex lock is a semaphore whose initial value is one
- **Remember each thread’s base priority**
  - Base priority is used for return value
- **Threads in nested donation should be on the list**

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.

Advanced Scheduler

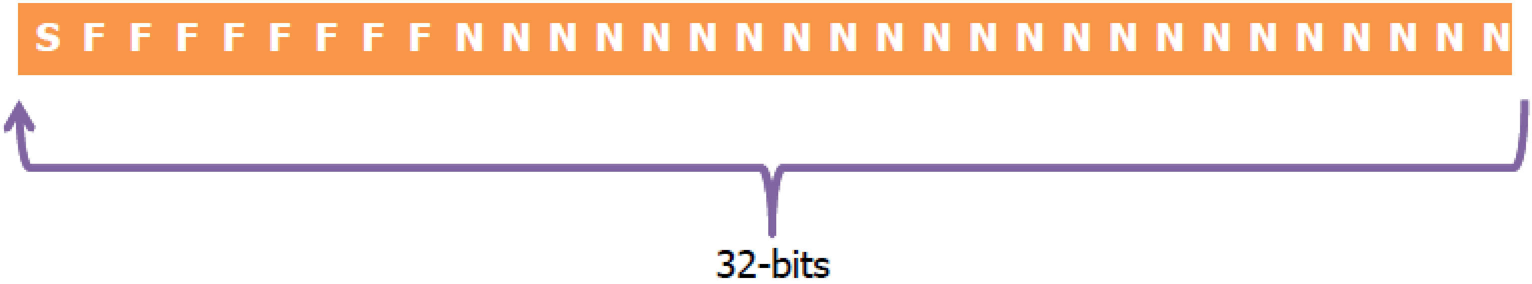
- **Why Advanced Scheduler?**
  - It is based on mathematical.
  - Prevent starve → Guaranty fairness.
  - If you developed this project without this concept, the low priority thread may starve.
- **Concept of "NICE"**
  - BSD Scheduler has two concept of "priority".
    - Relative Priority : Which is current priority
    - Absolute Priority : This is so called "NICE"
  - How "nice" the thread should be to other threads.

Advanced 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

Real Number representation

- **Floating point number (IEEE)**



S F F F F F F F F N N N N N N N N N N N N N N N N N N N N N N N

32-bits

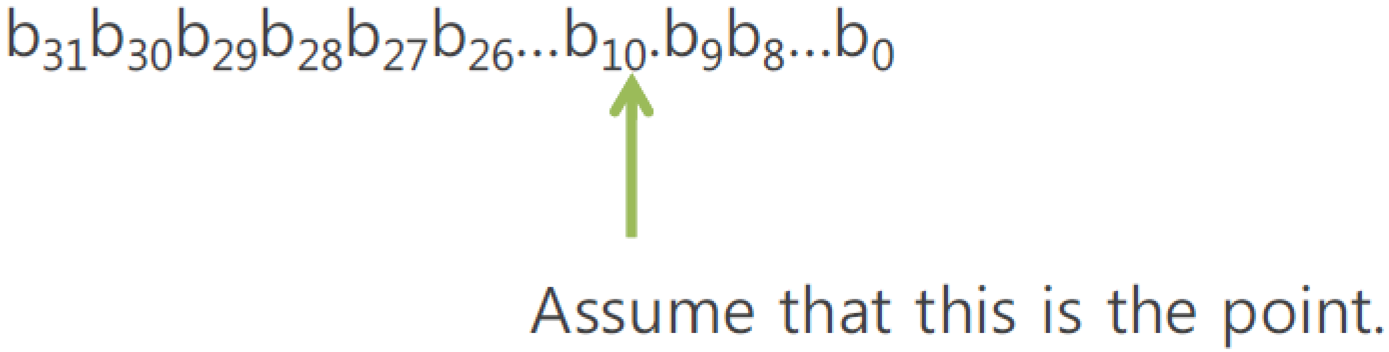
  - S : Sign-bit(31)
  - F : Fraction-bit (30:23)
  - N : Number-bit (22:0)

$(-1)^S \times M \times 2^E$: » **You will learn this formula on System Programming**

Real Number representation

Fraction Number

- What is it? The point position is fixed.
- For example…



$b_{31}b_{30}b_{29}b_{28}b_{27}b_{26}\ldots b_{10}.b_9b_8\ldots b_0$

Assume that this is the point.

- We will use this number representation.

Advanced Scheduler

- **Hints**
  - Before start implementation core area, implement fixed point arithmetic function.
    - It is well defined on 95p.
  - Load average value is global, recent cpus and priorities are per thread structure
  - Initial value
    - Load avg : zero when “init thread” starts
    - Recent cpu : parent value when a thread creates
    - Priority : inital value when a thread creates
  - 64bits Integer casting for multiplying
  - Multiplying before division

Advanced Scheduler

- **Addition your code with "thread_mlfqs"**
  - if (thread_mlfqs)
  - thread_mlfqs is true when –mlfqs option turns on

Submission

- **Due**
  - October 12, 11:59PM
  - Fill out the design for the priority donation and save it with PDF format ([Group_Number]_project1.pdf)
    - Ex) 1_project1.pdf
  - Submit a contribution of each member by percent and signature
  - Upload it at sys.skku.edu
- **Source code**
  - [GroupNumber]_Project1-2.tar.gz
  - Ex) tar czf 1_project1-2.tar.gz src
  - Upload it at sys.skku.edu