Operating Systems Lab

Part 1: Threads

Youjip Won

Overview

- Three topics
  - Alarm clock
  - Priority scheduling
  - Advanced scheduler

Alarm clock

Overview

- Main goal

  `timer_alarm(int ticks)`

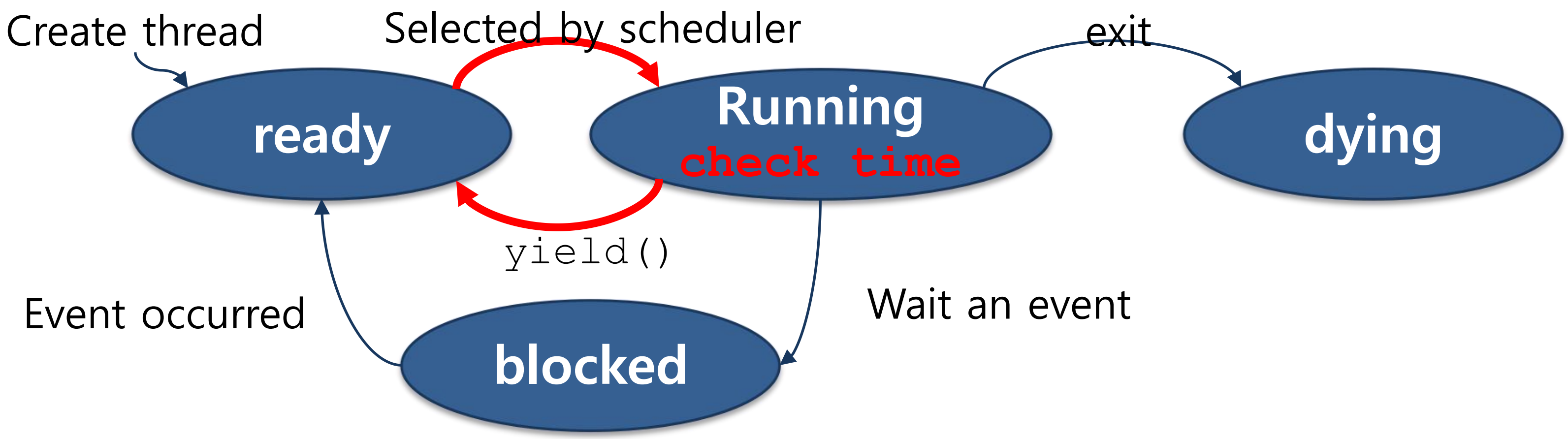
  system call that wakes up a process in `ticks` amount of time.

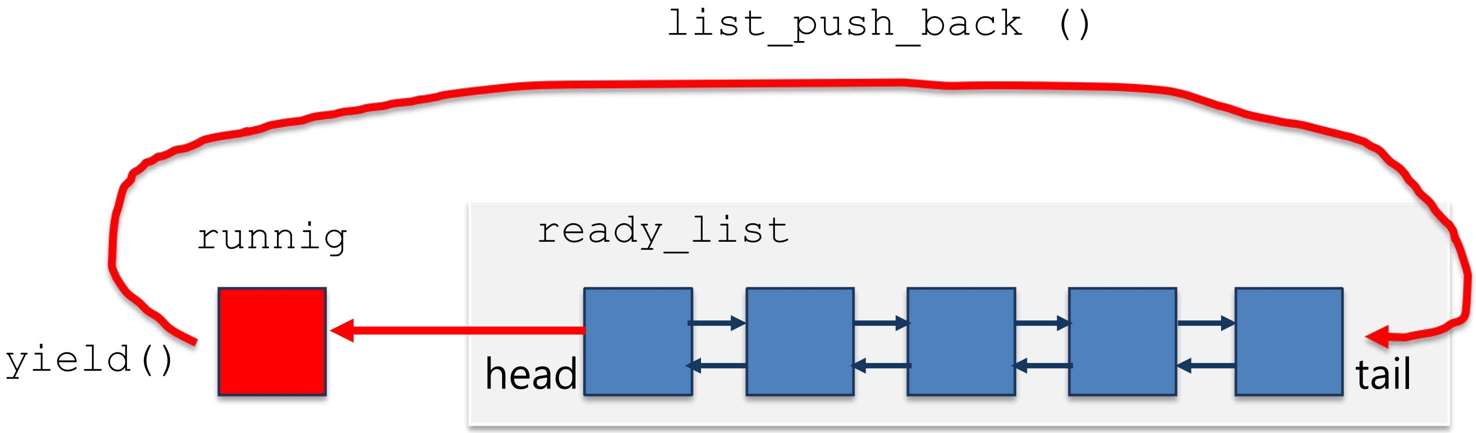
  - Pintos uses busy waiting for alarm.
  - Modify PintOS to use sleep/wakeup for alarm.

- Files to modify
  - threads/thread.*
  - devices/timer.*

`timer_sleep()` in current Pintos

- Keeps consuming CPU cycle





Sleep in original pintos

- Loop-based waiting up to the given tick.
- The thread that called this function is inserted to `ready_list` after the given tick.

pintos/src/device/timer.c

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

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

- `thread_yield`() : yield the cpu and insert thread to `ready_list`.
- `timer_ticks`() : return the value of the current tick.
- `timer_elased`() : return how many ticks have passed since the `start`.

thread_yield()

- Yield the cpu and insert the thread to `ready_list`.

pintos/src/device/timer.c

```
void thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    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);
}
```

functions in `thread_yield()`

- Description of `thread_yield()`

  `thread_current()`

  - Return the current thread.

  `intr_disable()`

  - Disable the interrupt and return previous interrupt state.

  `intr_set_level(old_level)`

  - Set a state of interrupt to the state passed to parameter and return previous interrupt state.

  `list_push_back(&ready_list, &cur->elem)`

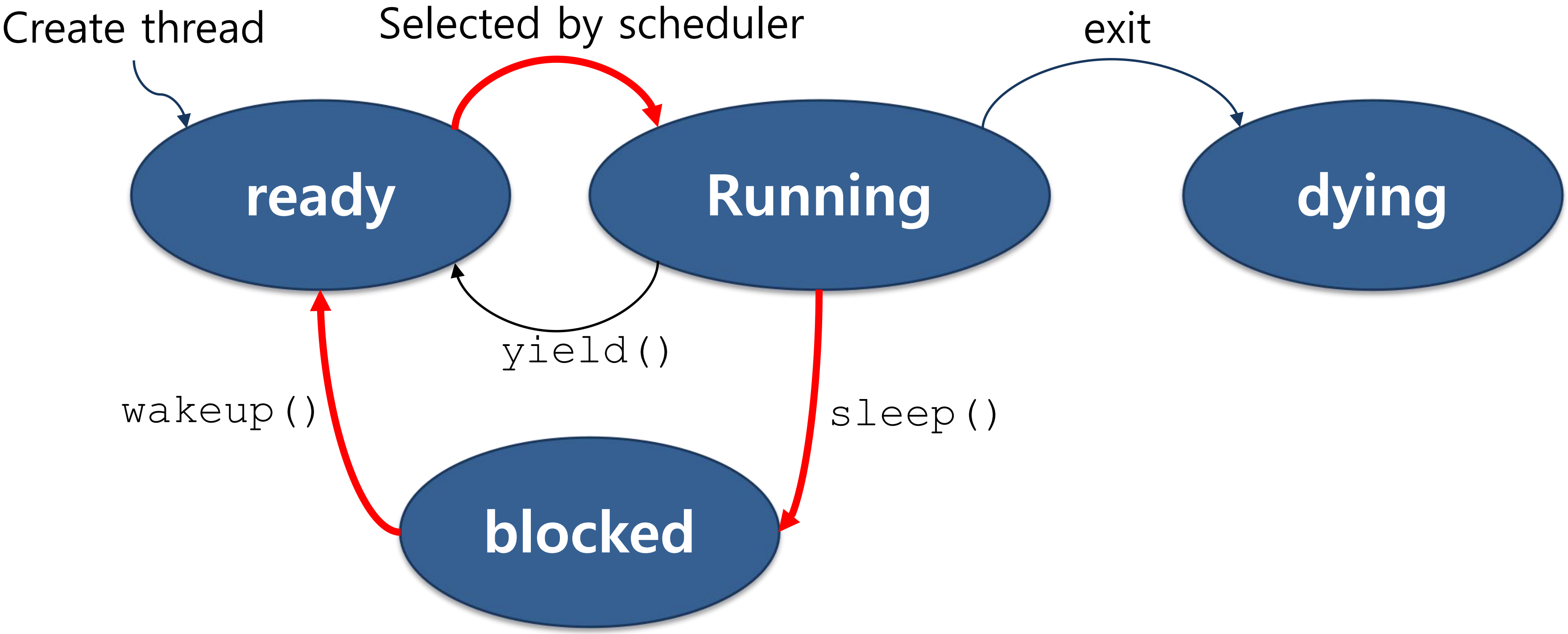
  - Insert the given entry to the last of list.

  `schedule()`

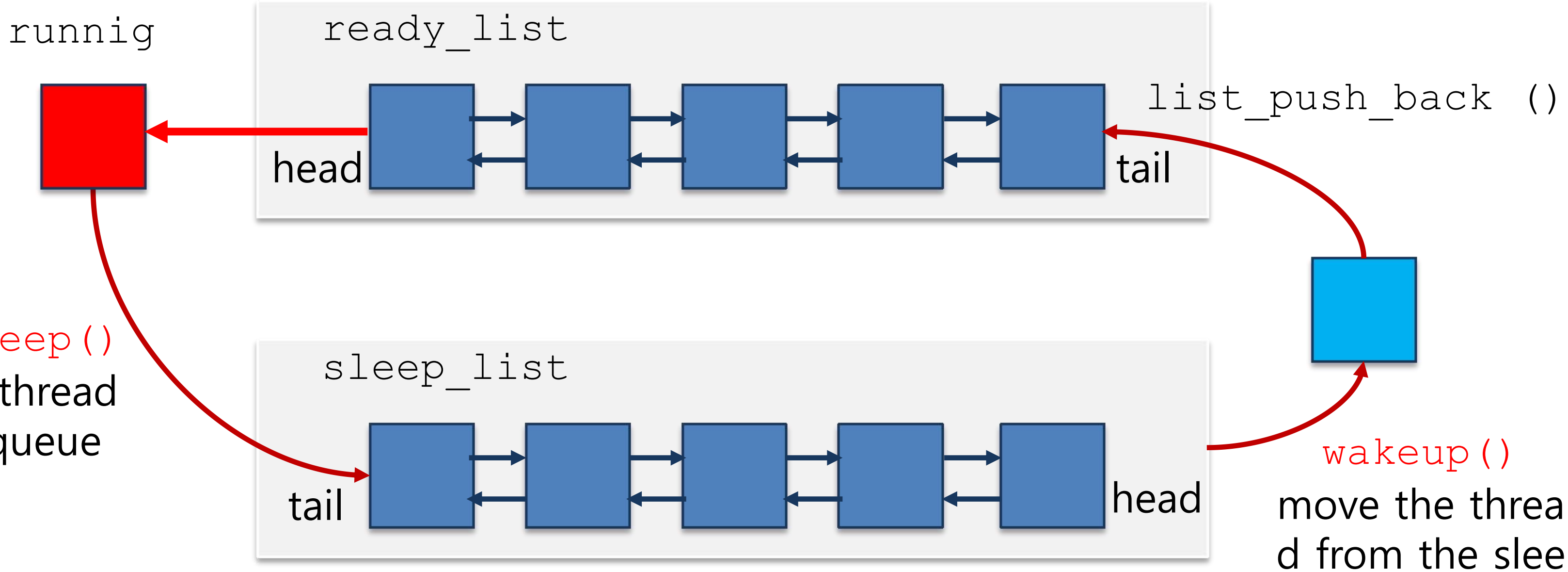
  - Do context switch

Design: use ‘blocked’ state for new `timer_sleep()`

- Save CPU cycle and power consumption.



Design: Sleep/wakeup-based alarm clock



timer_sleep()
Insert the thread to sleep queue

wakeup()
move the thread from the sleep list to the read list

Implementation of Alarm Clock

- Define Sleep Queue.

```
static struct list sleep_list;
```

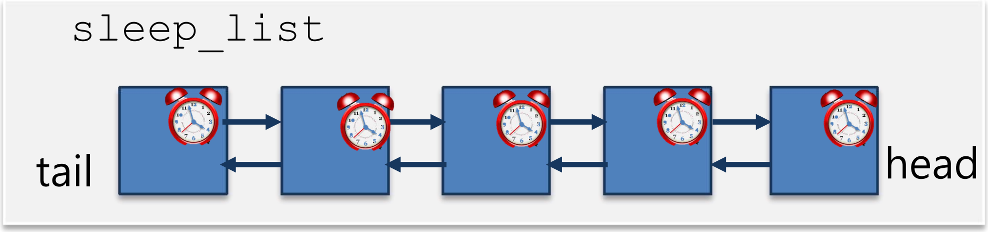
- and initialize it.

- Point to think:

where to declare the list and when to initialize it.

Global tick vs. local tick

- Kernel (timer interrupt handler) needs to check which threads to wake up.
- local tick
  - Each thread needs to maintain the time to wakeup.
  - Modify the `thread` structure: store the time to wake up.
- “tick”, the global variable
  - the minimum value of local `tick` of the threads
  - Save the time to scan the sleep list
  - Don’t forget to initialize it.



Modify thread structure

- Add new field for local tick, e.g. `wakeup_tick`
- Use `int64` type.

pintos/src/thread/thread.h

```
struct thread
{
        …

        /* tick till wake up */

        …
}
```

Implementation of Alarm Clock

- Thread: Move thread (itself) to the sleep queue.
  - When `timer_sleep()` is called, check the tick.
  - If there is time left till the wakeup, remove the caller thread from `ready_list` and insert it to sleep queue.

Sample implementation: pintos/src/devices/timer.c

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

~~    while (timer_elapsed (start) < ticks)~~
~~        thread_yield ();~~

    if(timer_elapsed (start) < ticks)
        thread_sleep(start + ticks); //implement by yourself
}
```

Implementation of Alarm Clock

- Value of ‘`start`’ may become invalid at (2).
- Let’s forget it for now.
  - Challenge: Think about how to fix it.

pintos/src/devices/timer.c

```
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks (); -------(1)

~~    while (timer_elapsed (start) < ticks)~~
~~        thread_yield ();~~

    if(timer_elapsed (start) < ticks).-------(2)
        thread_sleep(start + ticks);
}
```

thread_sleep()

- Change the state of the caller thread to ‘blocked’ and put it to the sleep queue

.

pintos/src/threads/thread.c

```
void thread_sleep(int64_t ticks){

  /* if the current thread is not idle thread,
        change the state of the caller thread to BLOCKED,
        store the local tick to wake up,
        update the global tick if necessary,
        and call schedule() */
  /* When you manipulate thread list, disable interrupt! */

}
```

Implementation of Alarm Clock

- In the timer interrupt,
  - Timer interrupt is heart of everything!.
  - Determine which threads to wake up everytime when timer interrupt occurs.
  - For the threads to wake up, remove them from the sleep queue and insert it t o the `ready_list.`(Don’t forget to change the state of the thread from sleep to ready!!!)

timer_interrupt()

pintos/src/devices/timer.c

```c
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick (); // update the cpu usage for running process

    /* code to add:
       check sleep list and the global tick.
       find any threads to wake up,
       move them to the ready list if necessary.
       update the global tick.
       */
}
```

Summary

- Functions to modify
  - `thread_init()`
    - Add the code to initialize the sleep queue data structure.
  - `timer_sleep()`
    - Call the function that insert thread to the sleep queue.
  - `timer_interrupt()`
    - At every tick, check whether some thread must wake up from sleep queue and call wake up function.

Design tip for modularization

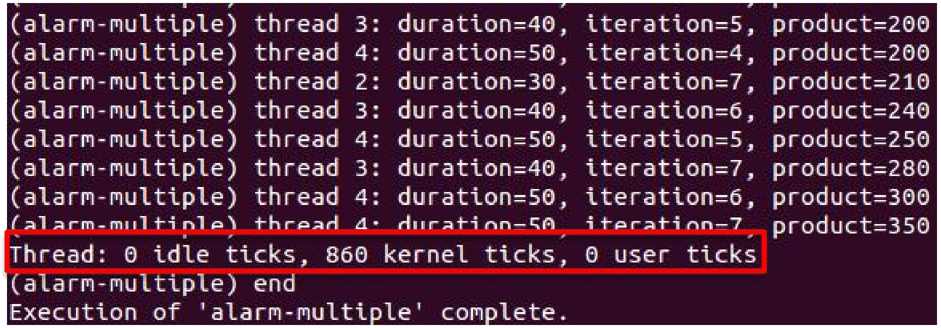
- Functions to add

1. The function that sets thread state to blocked and wait after insert it to sleep queue.
2. The function that find the thread to wake up from sleep queue and wake up it .
3. The function that save the minimum value of tick that threads have.
4. The function that return the minimum value of tick.

Result

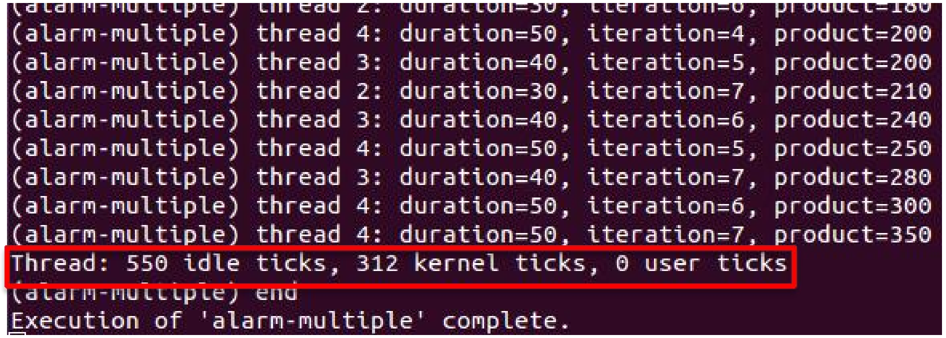
```
$ pintos -- -q run alarm-multiple
```

- Busy waiting



```
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
Thread: 0 idle ticks, 860 kernel ticks, 0 user ticks
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

- After removing the busy waiting (using sleep queue)



```
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
Thread: 550 idle ticks, 312 kernel ticks, 0 user ticks
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

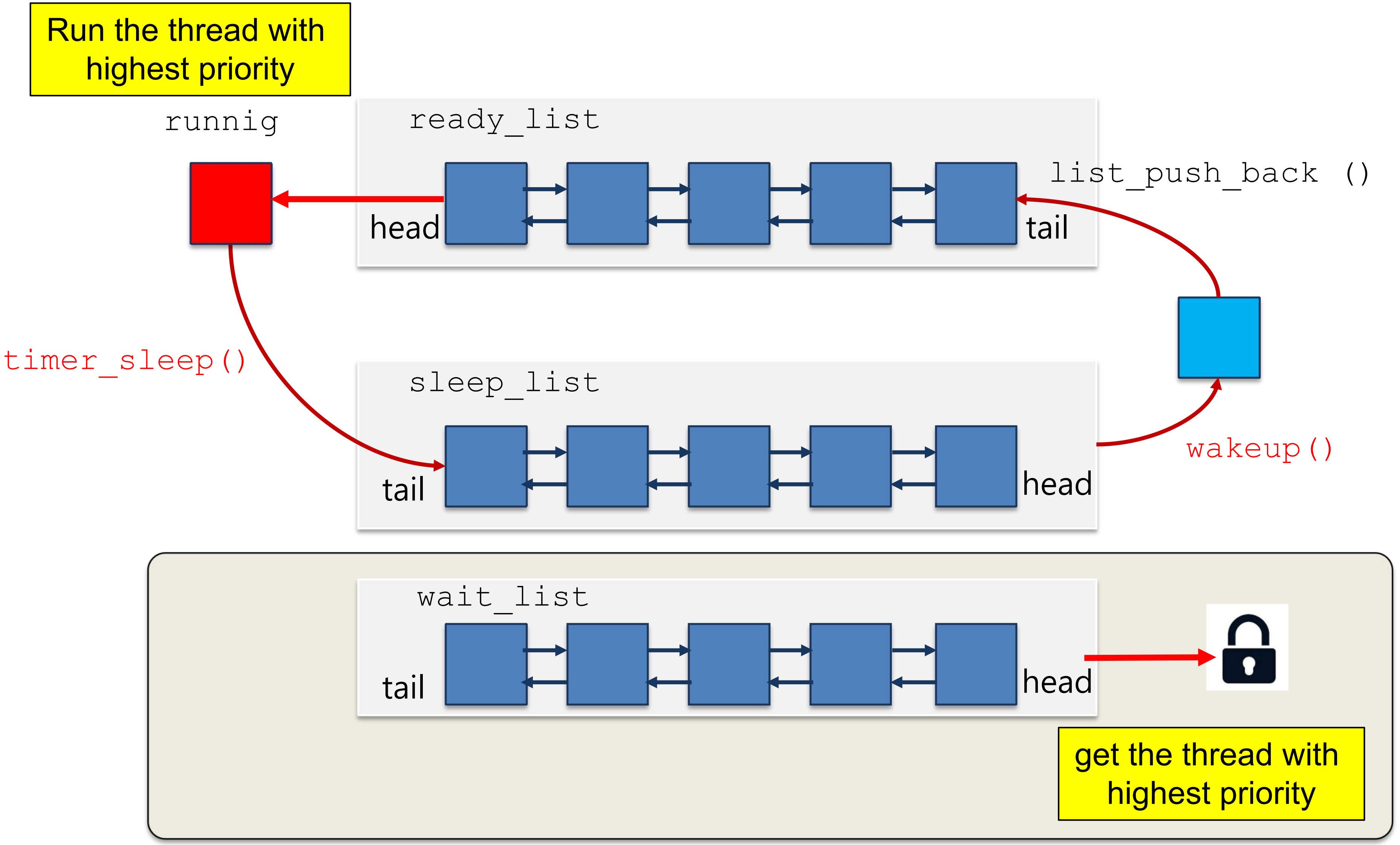
- The idle tick was zero because it occupied the CPU even in the sleep state, but the idle tick increased after removing the busy waiting.

Priority Scheduling

Outline

- Main goal
  - Pintos uses FIFO scheduling.
  - Modify PintOS scheduler for priority scheduling
    - Sort the ready list by the thread priority.
    - Sort the wait list for synchronization primitives(semaphore, condition variable).
    - Implement the preemption.
    - Preemption point: when the thread is put into the ready list (not everytime when the timer interrupt is called).
- Files to modify
  - threads/thread.*
  - threads/synch.*

Design: Sleep/wakeup-based alarm clock



Three things to consider

- When selecting a thread to run in the ready list, select the one with the highest priority.
- Preemption
  - When inserting the new thread to the ready list, compare the priority with the running thread.
  - Schedule the newly inserted thread if it has the higher priority with the currently running thread.
- Lock: semaphore, condition variable,
  - When selecting a thread from the set of threads waiting for a lock (or condition variable), select the one with the highest priority.

Priority in pintos

- Priority ranges from `PRI_MIN(=0)` to `PRI_MAX(=63)`.
  - The larger the number, the higher priority .
  - Default is `PRI_DEFAULT(=31)`
- PintOS sets the initial priority when the thread is created by `thread_create()`
- Existing functions
  - `void thread_set_priority (int new_priority)`
    - Change priority of the current thread to `new_priority`
  - `int thread_get_priority (void)`
    - Return priority of the current thread.

Implementation of Priority Scheduling

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

- Point of updates
  - Insert thread in `ready_list` in the order of priority. (note that it is not scalable)
  - When the thread is added to the `ready_list,` compare priority of new thread and priority of the current thread.
  - If the priority of the new thread is higher, call `schedule()` (the current thread yi elds CPU).

thread_create()

- When inserting a thread to `ready_list` , compare the priority with the currently running thread.
- If the newly arriving thread has higher priority, preempt the currently running thread and execute the new one.

pintos/src/threads/thread.c

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

    /* compare the priorities of the currently running t
hread and the newly inserted one. Yield the CPU if the n
ewly arriving thread has higher priority*/
    return tid;
}
```

Others to modify

- `void thread_unblock(struct thread *t)`
  - When the thread is unblocked, it is inserted to `ready_list` in the priority order.
- `void thread_yield(void)`
  - The current thread yields CPU and it is inserted to `ready_list` in priority order.
- `void thread_set_priority(int new_priority)`
  - Set priority of the current thread.
  - Reorder the `ready_list`

Hint: thread_unblock (happy holiday~!^^)

- `thread_unblock()`
  - When unblocking a thread, use list_inert_ordered instead of list_push_back.

pintos/src/threads/thread.c

```
void thread_unblock (struct thread *t)
{
    …

      //list_push_back (&ready_list, &t->elem);     delete

      list_insert_ordered(& ready_list, & t-> elem, add
                                   cmp_priority, NULL);

      t->status = THREAD_READY;
      intr_set_level (old_level);
}
```

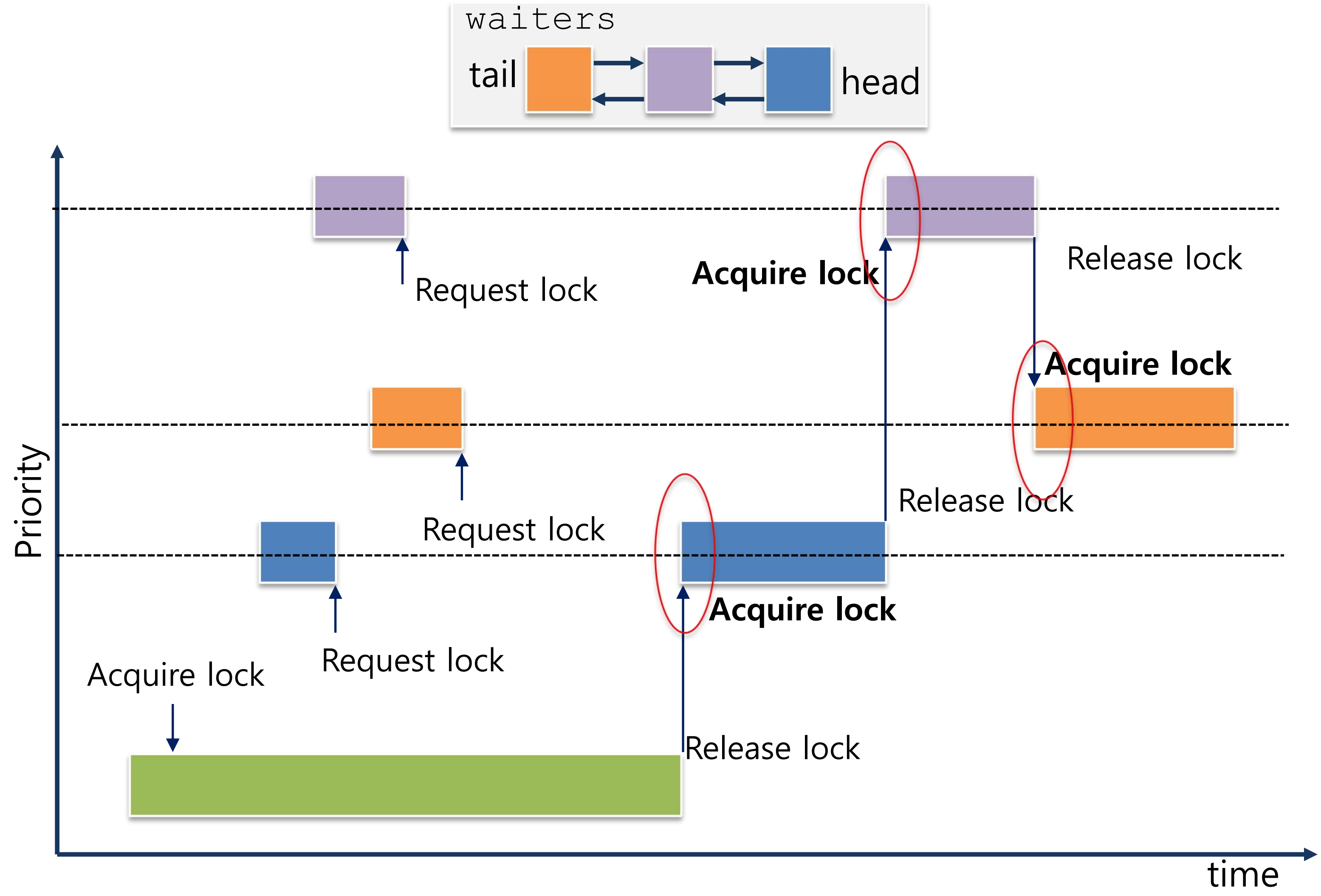
Change the synchronization primitives

- Lock
- Semaphore
- Condition variables

Wake up the waiting thread with respect to the thread’s priority.

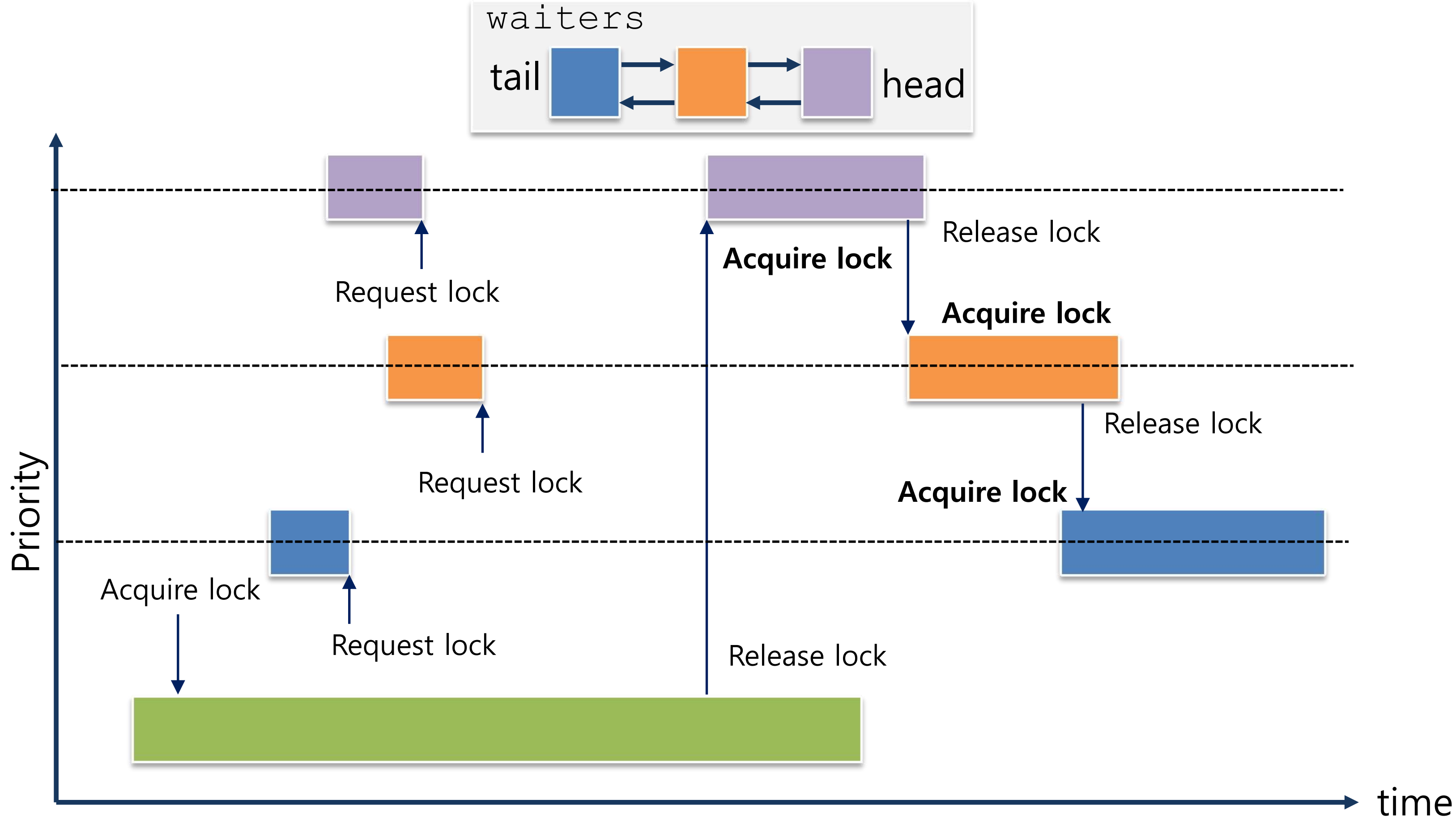
FIFO lock/unlock in priority-less Pintos

- Lock is acquired by FIFO order in `waiters` list, ignoring priority.



Priority-based lock/unlock

- When threads try to acquire semaphore, sort `waiters` list in order of priority.
  - Modify `sema_down() / cond_wait()`



Semaphore in pintos

pintos/src/threads/synch.h

```
struct semaphore   {
    unsigned value;               /* Current value. */
    struct list waiters;          /* List of waiting
                                        threads. */
};
```

```
void sema_init(struct semaphore *sema, unsigned value)
```

- Initialize `semaphore` to the given `value`

```
void sema_down(struct semaphore *sema)
```

- Request the `semaphore`. If it acquired the `semaphore`, decrease the `value` by 1

```
void sema_up(struct semaphore *sema)
```

- Release the `semaphore` and increase the `value` by 1

Lock in pintos

pintos/src/threads/synch.h

```
struct lock
{
     struct thread *holder;          /* Thread holding lock */
     struct semaphore semaphore;     /* Binary semaphore
                                        controlling access. */
};
```

```
void lock_init (struct lock *lock)
```

- Initialize the `lock` data structure.

```
void lock_acquire (struct lock *lock)
```

- Request the `lock`.

```
void lock_release (struct lock *lock)
```

- Release the `lock`.

Condition variable in pintos

pintos/src/threads/synch.h

```
struct condition {
    struct list waiters;      /* List of waiting threads. */
};
```

```
void cond_init(struct condition *cond)
```

- Initialize the condition variable data structure.

```
void cond_wait(struct condition *cond, struct lock *lock)
```

- Wait for signal by the condition variable.

```
void cond_signal(struct condition *cond,
                 struct lock *lock UNUSED)
```

- Send a signal to thread of the highest priority waiting in the condition variable.

```
void cond_broadcast(struct condition *cond, struct lock *lock
)
```

- Send a signal to all threads waiting in the condition variable.

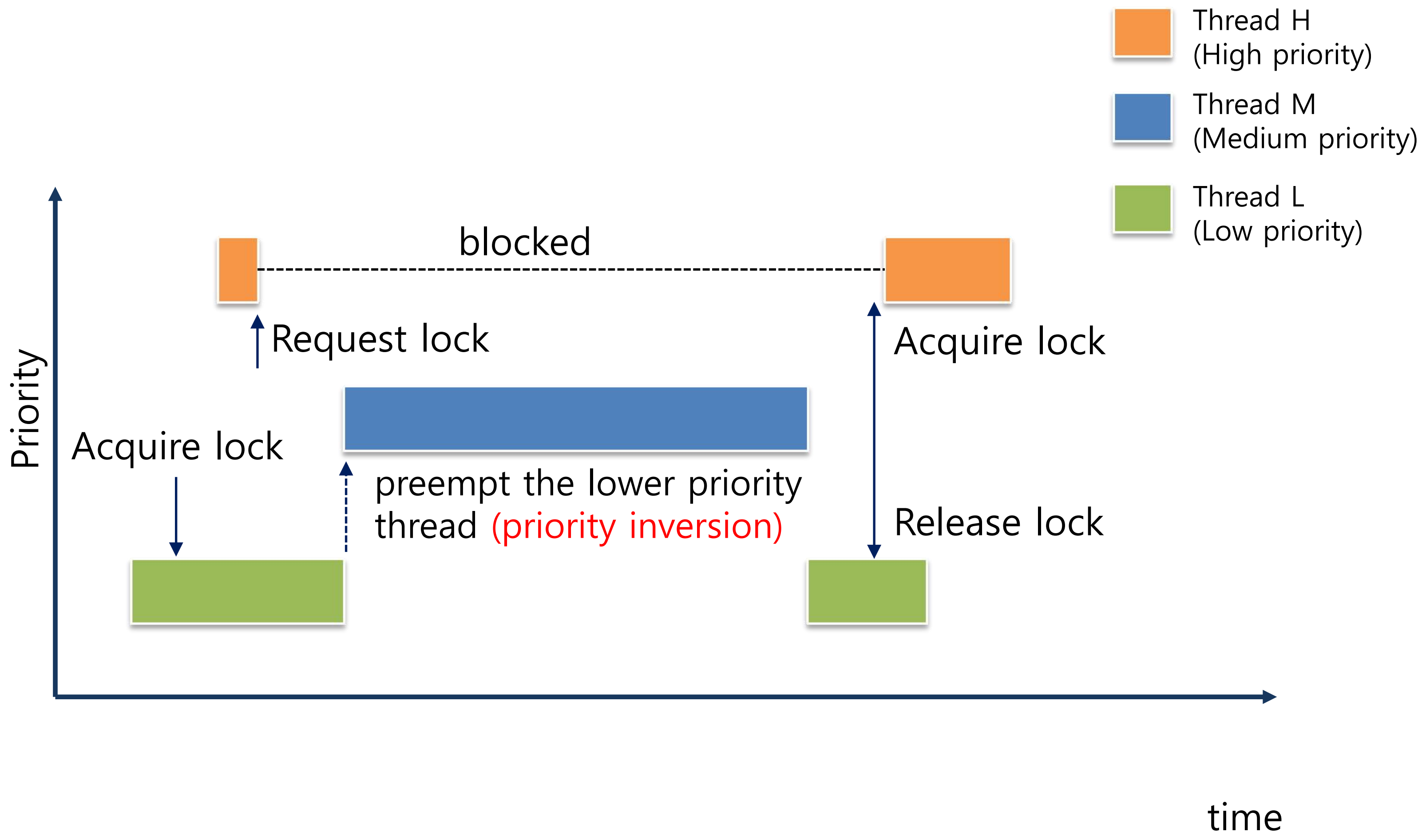
Implementation of Priority Scheduling-Synchronization

- Functions to modify.
  - Modify to insert thread at `waiters` list in order of priority
    - `void sema_down(struct semaphore *sema)`
    - `void cond_wait(struct condition *cond, struct lock *lock)`
  - Sort the waiters list in order of priority
    - It is to consider the case of changing priority of threads in `waiters` list.
    - `void sema_up(struct semaphore *sema)`

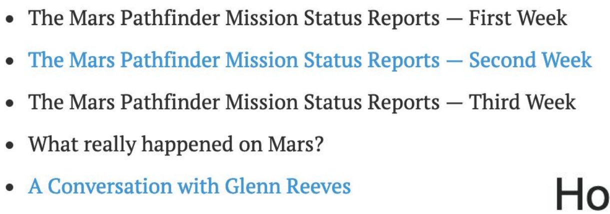
    -
      ```
      void cond_signal(struct condition *cond,
                       struct lock *lock UNUSED)
      ```

Priority Inversion

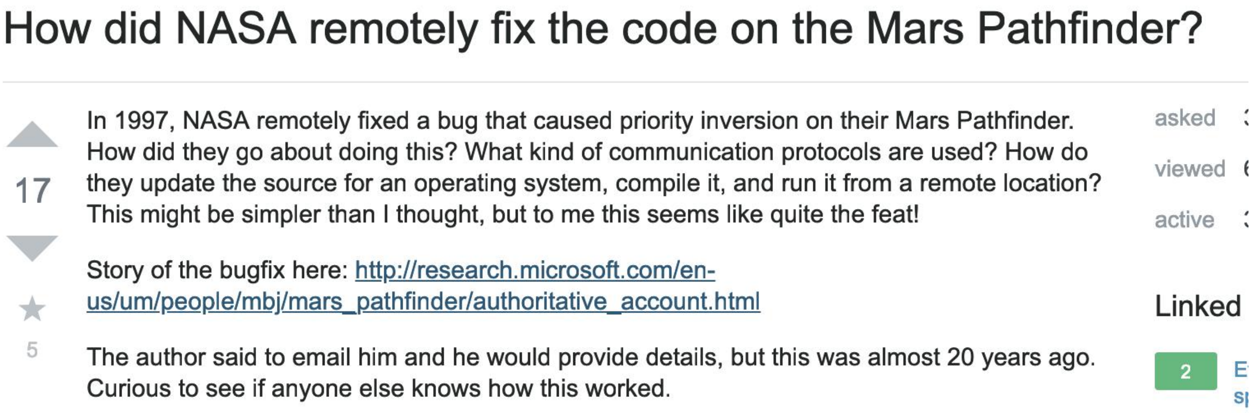
- The situation where thread of the higher priority waits thread of the lower priority.



In 1997, Pathfinder on Mars has stopped. OS has crashed due to the priority inversion.



- The Mars Pathfinder Mission Status Reports — First Week
- The Mars Pathfinder Mission Status Reports — Second Week
- The Mars Pathfinder Mission Status Reports — Third Week
- What really happened on Mars?
- A Conversation with Glenn Reeves



How did NASA remotely fix the code on the Mars Pathfinder?

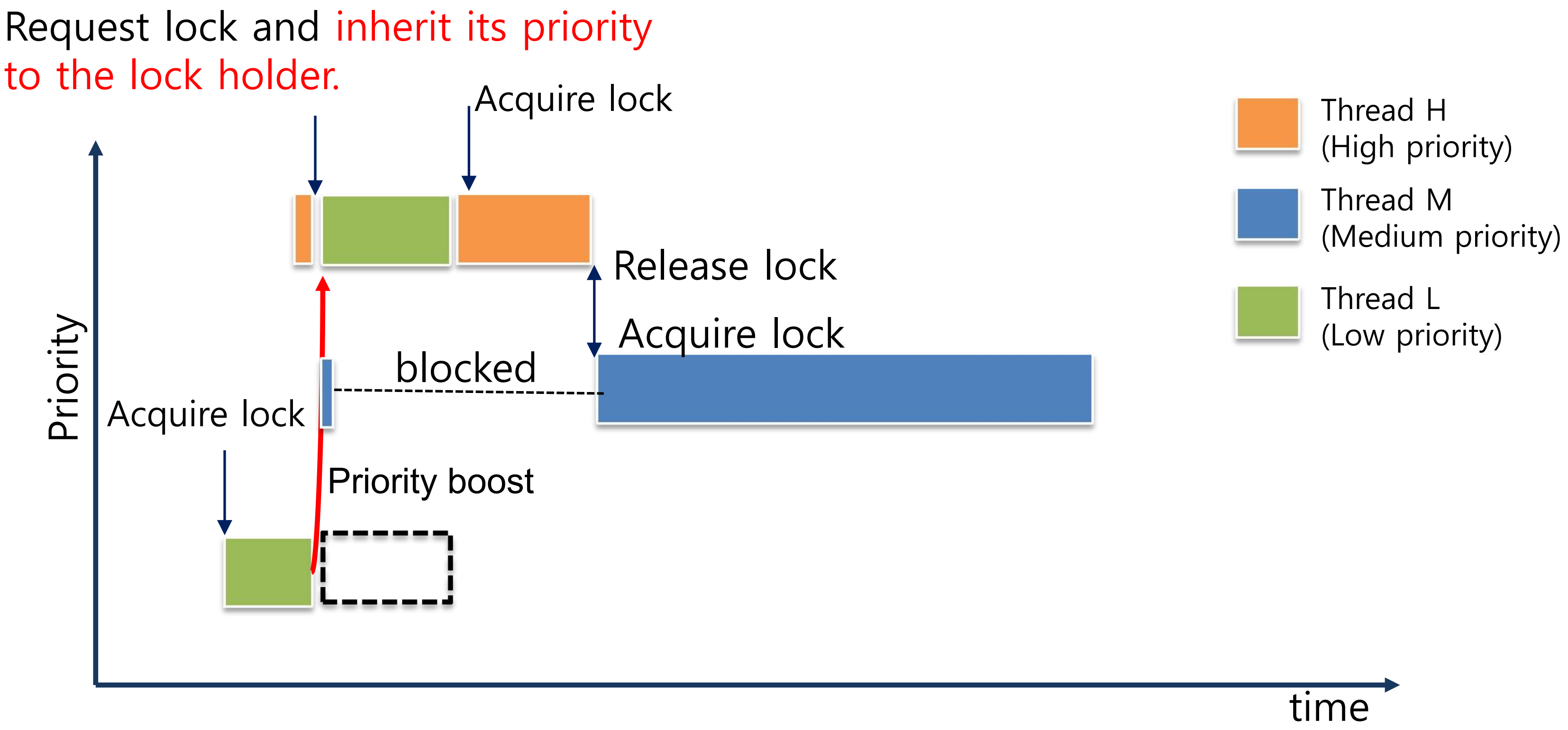
In 1997, NASA remotely fixed a bug that caused priority inversion on their Mars Pathfinder. How did they go about doing this? What kind of communication protocols are used? How do they update the source for an operating system, compile it, and run it from a remote location? This might be simpler than I thought, but to me this seems like quite the feat!

Story of the bugfix here: http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

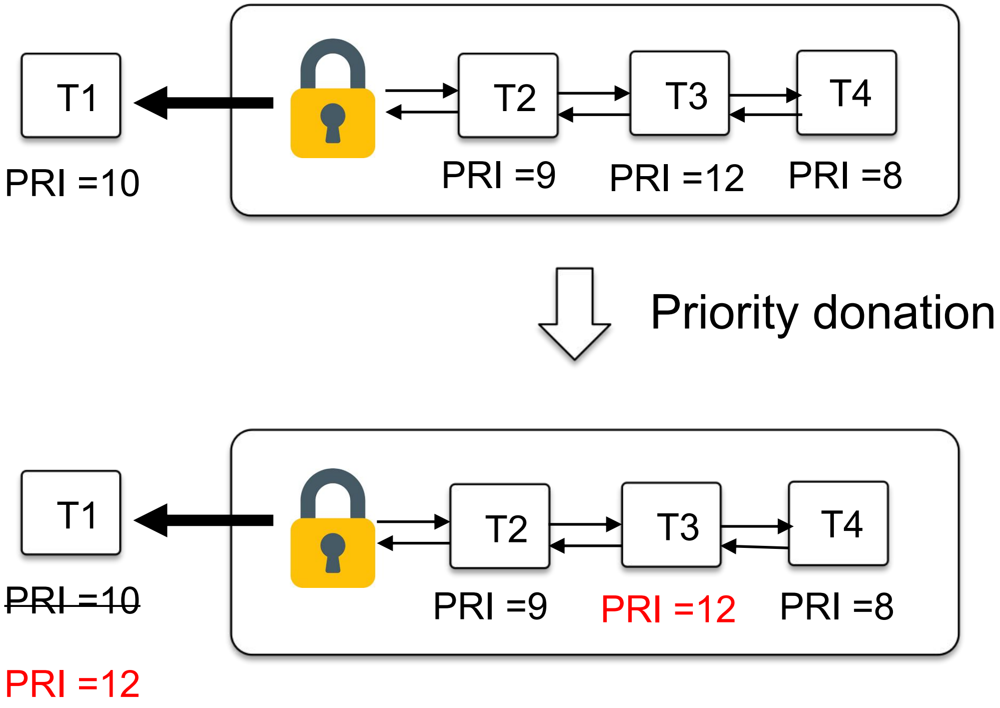
The author said to email him and he would provide details, but this was almost 20 years ago. Curious to see if anyone else knows how this worked.

Priority Donation

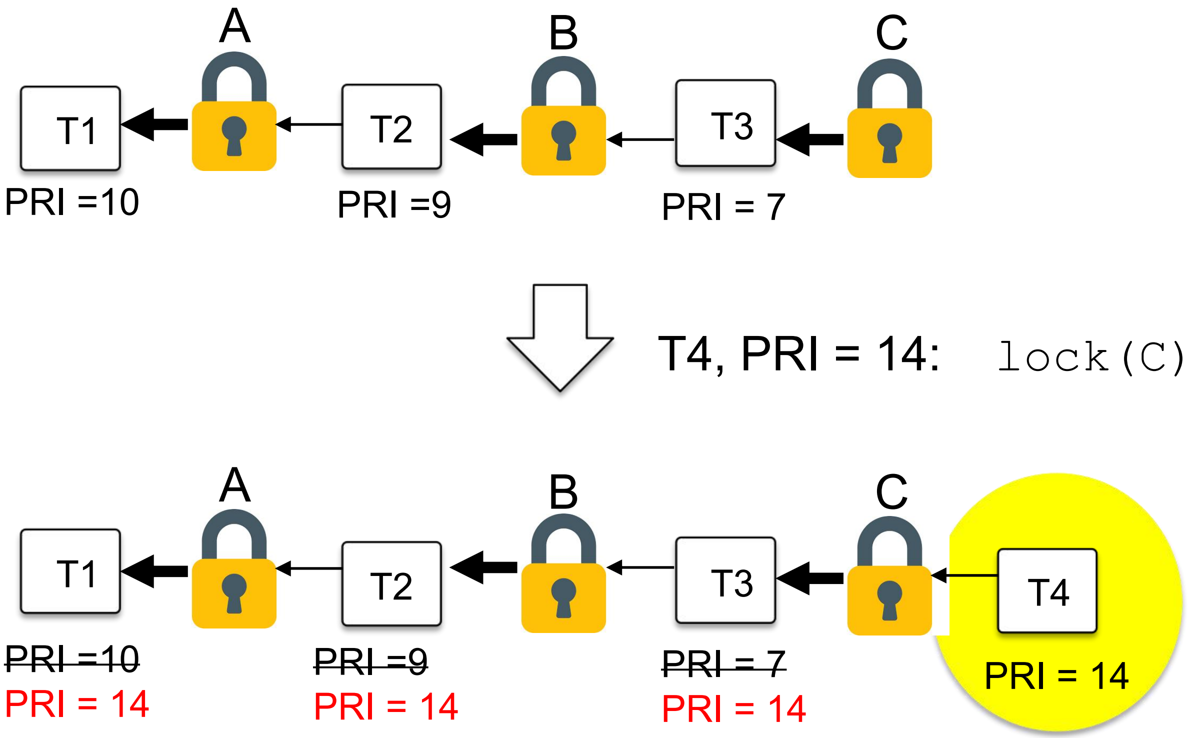
- Inherit its priority to the lock holder.



Priority Donation

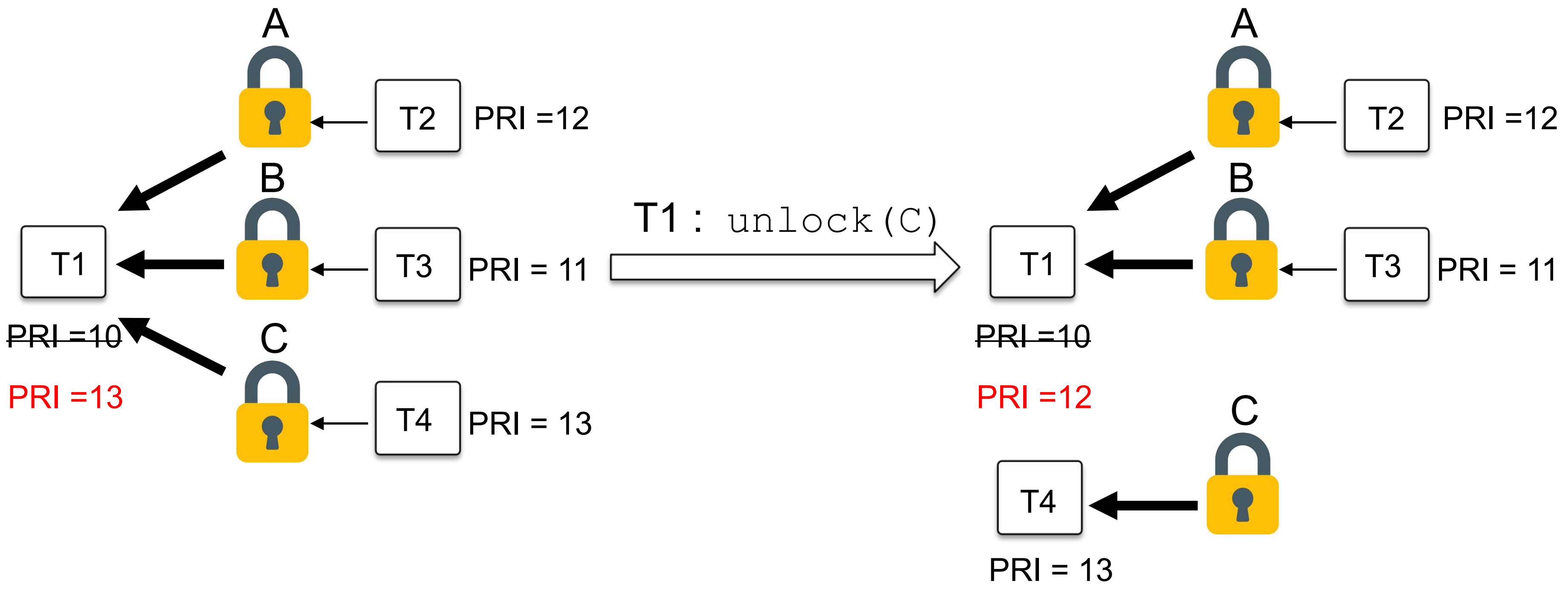


Nested Donation



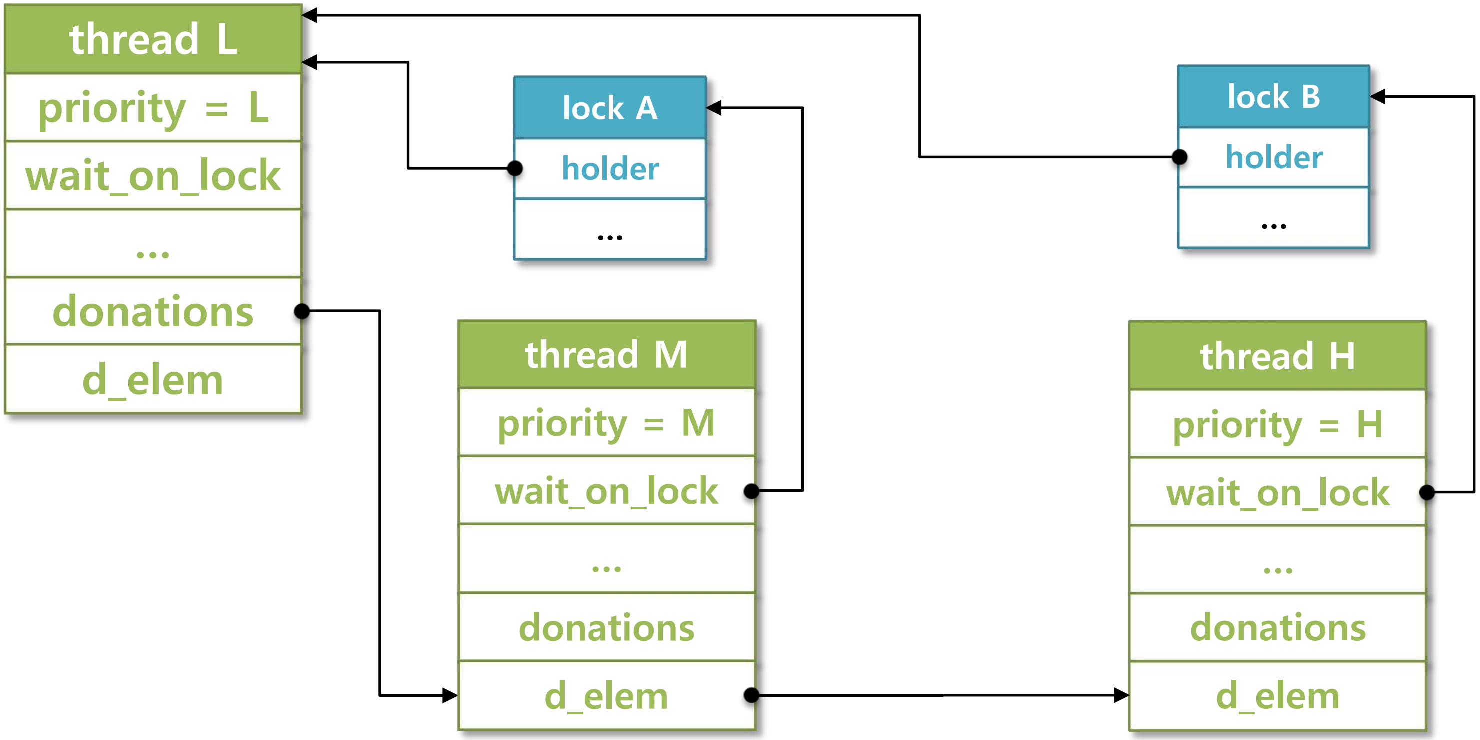
T4, PRI = 14: `lock(C)`

Multiple Donation



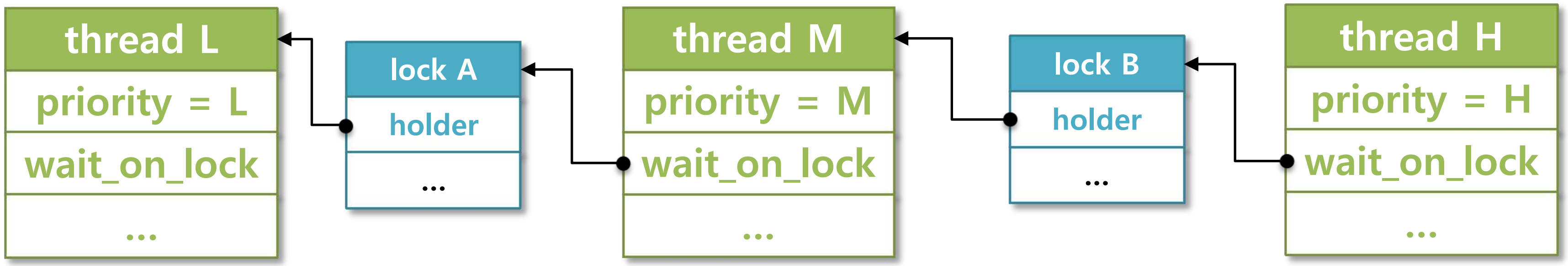
Data Structure for Multiple Donation

- `Donations`: list of Donors



Data Structure for nested donataion

- `wait_on_lock`: lock that it waits for

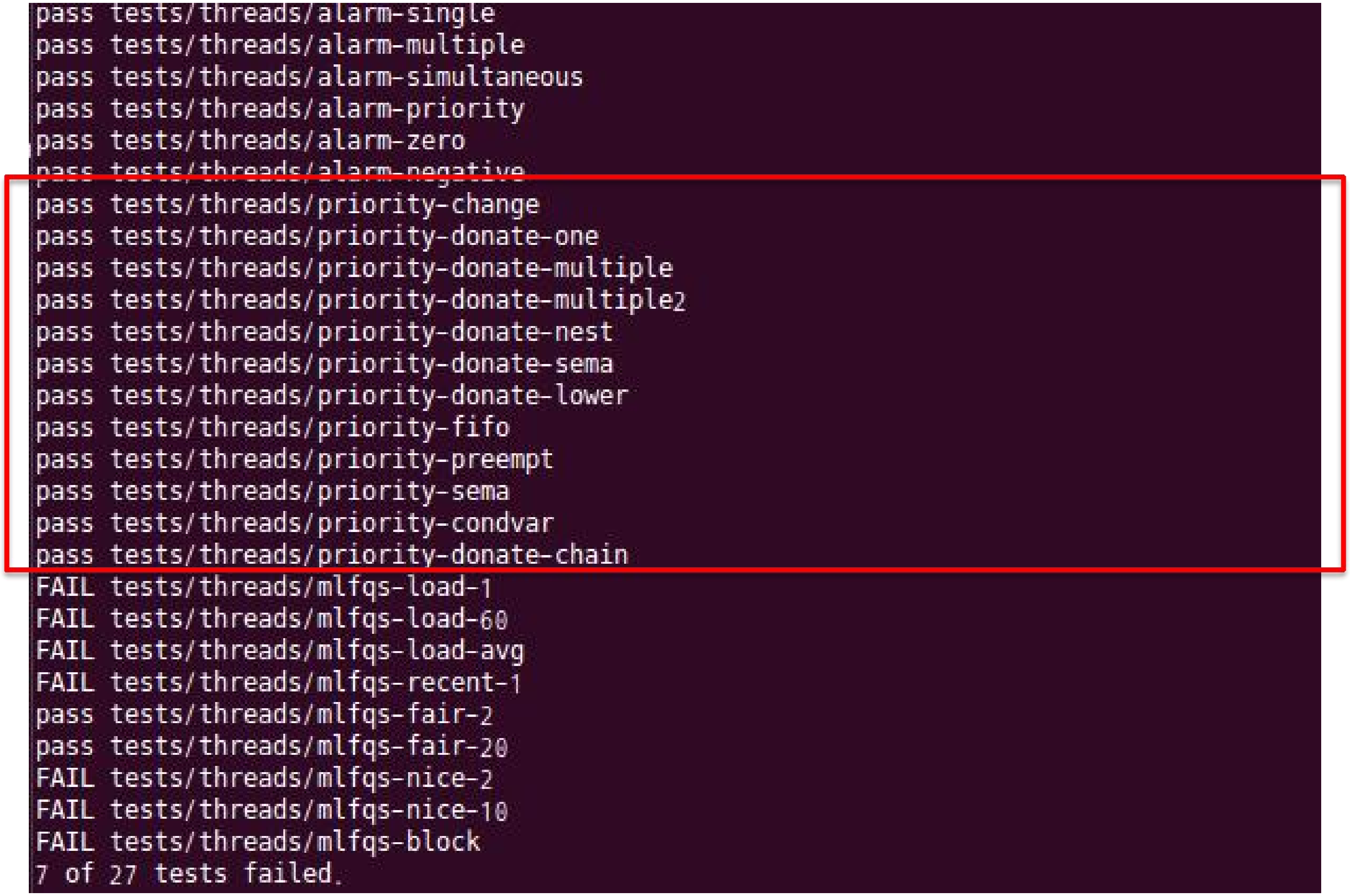


Implementation of Priority Donation

- Functions to modify
  - `static void init_thread(struct thread *t,`
    `const char *name, int priority)`
    - Initializes data structure for priority donation.
  - `void lock_acquire(struct lock *lock)`
    - If the lock is not available, store address of the lock.
    - Store the current priority and maintain donated threads on list (multiple donation).
    - Donate priority.
  - `void lock_release(struct lock *lock)`
    - When the lock is released, remove the thread that holds the lock on donation list and set prior ity properly.
  - `void thread_set_priority(int new_priority)`
    - Set priority considering the donation.

Result

```
$ make check
```



```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
7 of 27 tests failed.
```