Operating Systems

Project #6

Seong-Yeop Jeong

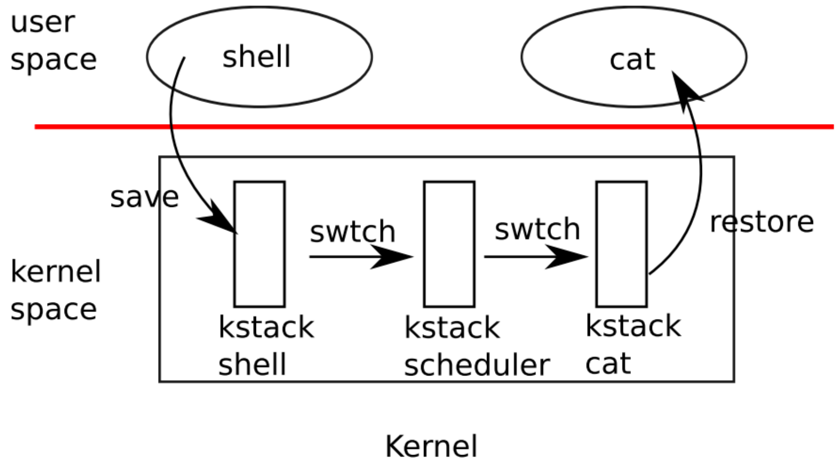
2020.06.04.

Project#6: Kernel threads

- Kernel threads (45 points)
- Preemptive priority scheduling (15 points)
- Priority donation for sleeplock (30 points)

Context switching in xv6 – 1

- Switching from one user process to another.
- 1. a user-kernel transition (system call or interrupt)
  2. to the old process's kernel thread,
  3. a context switch to the current CPU's scheduler thread,
  4. a context switch to a new process's kernel thread,
  5. and a trap return to the user-level process



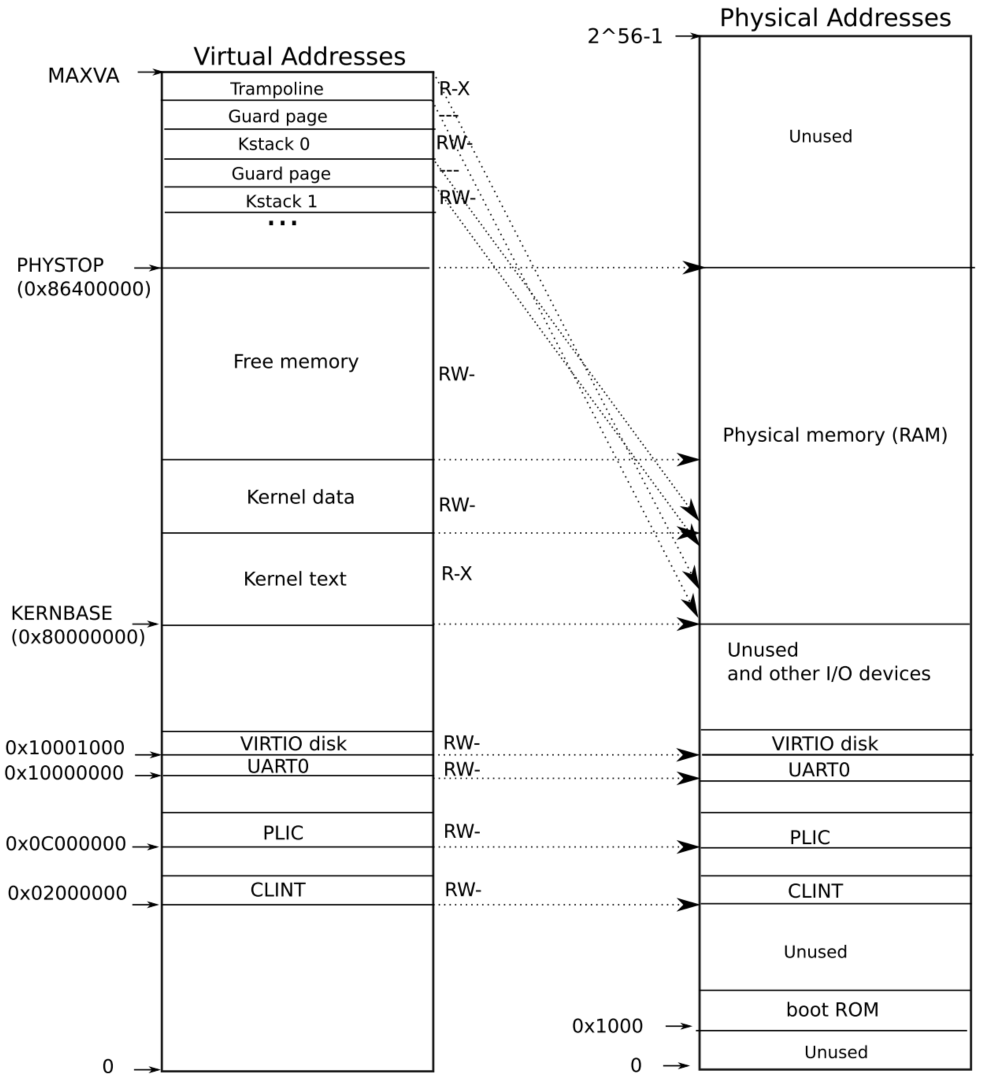
Context switching in xv6 – 2

- The function **swtch** performs the saves and restores for a kernel thread switch.
- **swtch** doesn't directly know about threads; it just saves and restores re gister sets, called contexts.
    - Save current registers in old. Load from new (in /kernel/swtch.S)
- When **swtch** returns, it returns to the instructions pointed to by the restored ra register, that is, the instruction from which the new thread previously called swtch. (context.ra)
- In addition, it returns on the new thread's stack. (context.sp)

Kernel threads in xv6

- You can reuse most of the existing data structures to implement kernel threads.
  - allocate stacks for kernel thread by using 'already' allocated kernel stack.
  - use allocpid() for allocate tid(thread id) for kernel thread.
- When a kernel thread is created, allocate an entry in the proc structure
- initialize its address space to kernel_pagetable.
- Because our kernel thread never runs user-space code, you can simplify its implementation
  - you don't have to allocate trapframe for kernel threads.
  - Just allocate context for swtch.

Kernel address space in xv6



kthread_create()

- int kthread_create(const char *name, int prio, void (*fn)(void *), void *arg);
  - The name can be assigned the same as the existing process creation.
  - pior becomes the base priority of the new kernel thread.
  - The fn is a function that begins when a thread occurs, and arg is a argument that is passed on to it.
  - If you understand the above context, you will know that thread executes a function.
  - And if the new thread's priority is higher than the existing thread, it should yield.

kthread_yield()

- You just need to make it work almost like an existing yield() function.

kthread_exit()

- When a kernel thread is terminated, **all the resources** allocated for the kernel thread should be free.
- And at the very end, run the sched() so that you can move to the scheduler.

kthread_setprio()

- The kthread_set_prio() function sets the calling kernel thread's *base* priority value to newprio

- In order to implement the priority donation, yield must be called accord ing to effective priority not base priority.

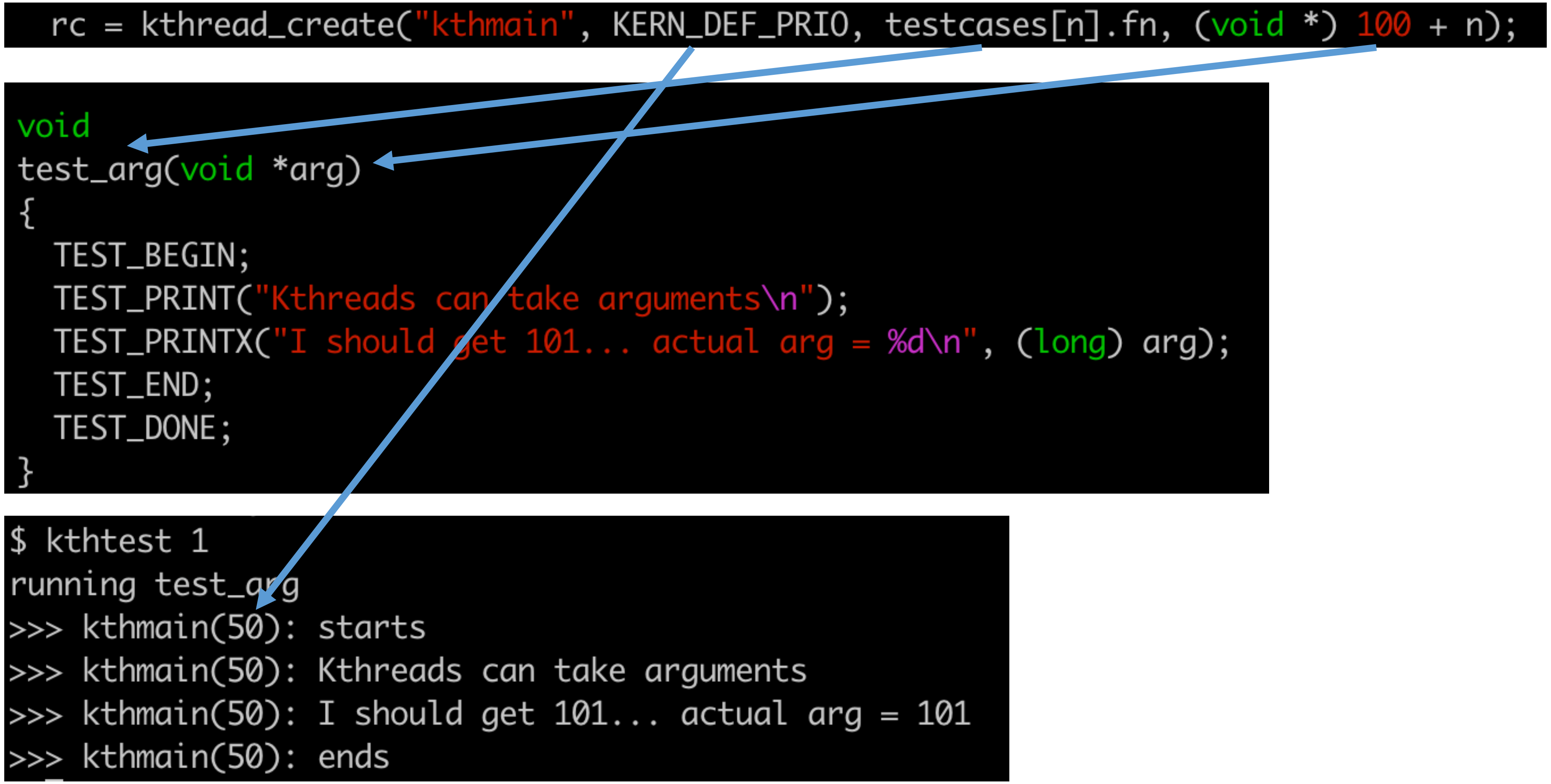
- Effective priority may be lower than base priority due to donation, At th is time, if it is lower than newprio, you should change the effective priori ty to newprio and cancel the priority change.

kthread_getprio()

- The kthread_get_prio() function returns the calling kernel thread's *effective* priority value

kthread | example

- n = 1, testcases[n].fn = test_arg



```
  rc = kthread_create("kthmain", KERN_DEF_PRIO, testcases[n].fn, (void *) 100 + n);
```

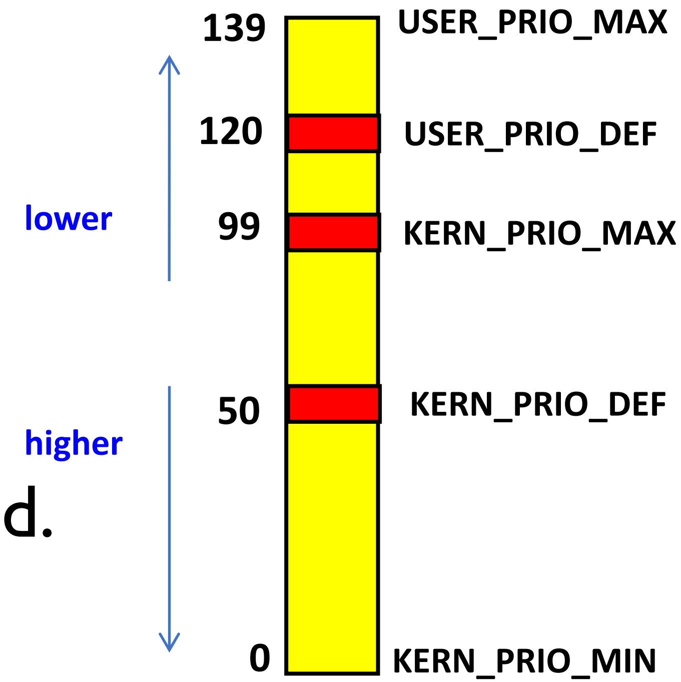
```
void
test_arg(void *arg)
{
   TEST_BEGIN;
   TEST_PRINT("Kthreads can take arguments\n");
   TEST_PRINTX("I should get 101... actual arg = %d\n", (long) arg);
   TEST_END;
   TEST_DONE;
}
```

```
$ kthtest 1
running test_arg
>>> kthmain(50): starts
>>> kthmain(50): Kthreads can take arguments
>>> kthmain(50): I should get 101... actual arg = 101
>>> kthmain(50): ends
```

Preemptive priority scheduling

- preemptive priority scheduler basically chooses a task with the highest priority among the runnable tasks.

- When a thread is added to the runnable that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread.



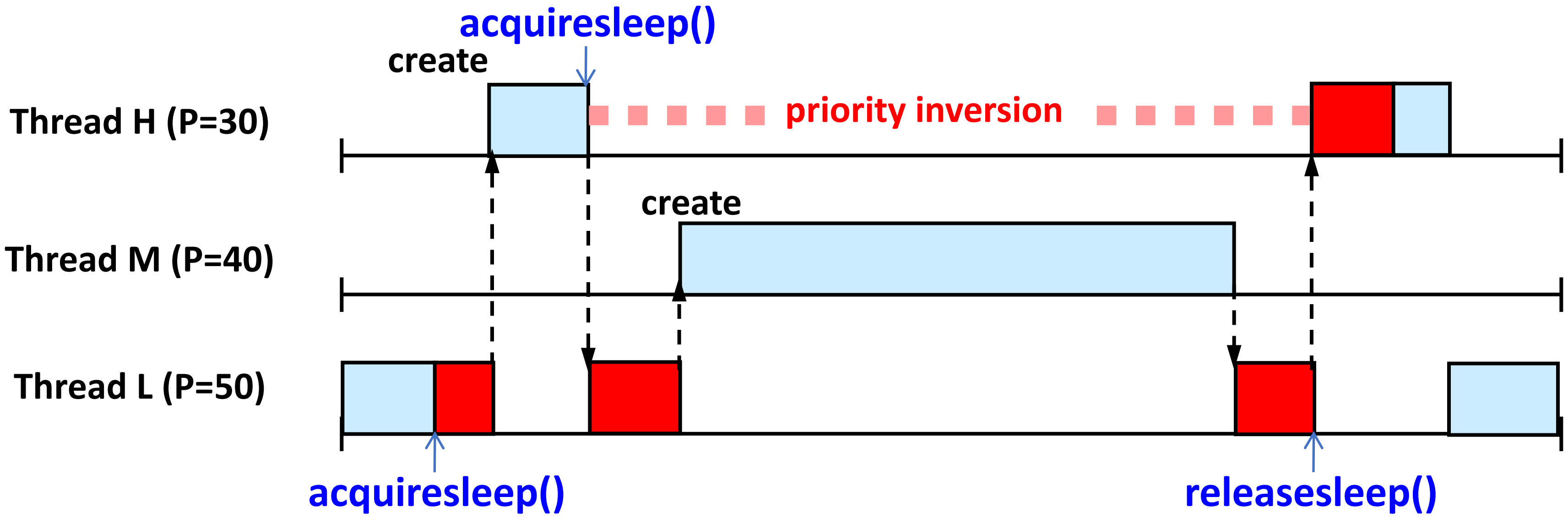
- If there are multiple highest priority tasks with the same priority, those tasks should be run in a round-robin fashion.

Preemptive priority scheduling

- Scheduling should be done according to effective priority.
- The important part is that when it is the same priority, it should be implemented in the round robin fashion.
- As for how to run round robin at the same priority, cycling from the beginning of the proc always selects the process ahead.
  - You have to change this part a little.

Priority inversion

- Sleeplock is the same concept as mutex in xv6.
- 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.

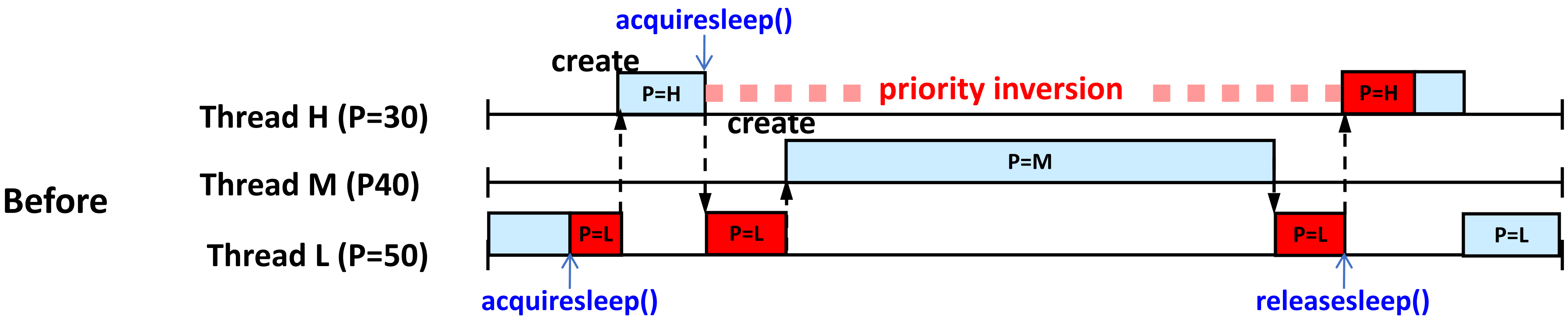


Priority donation (or priority inheritance)

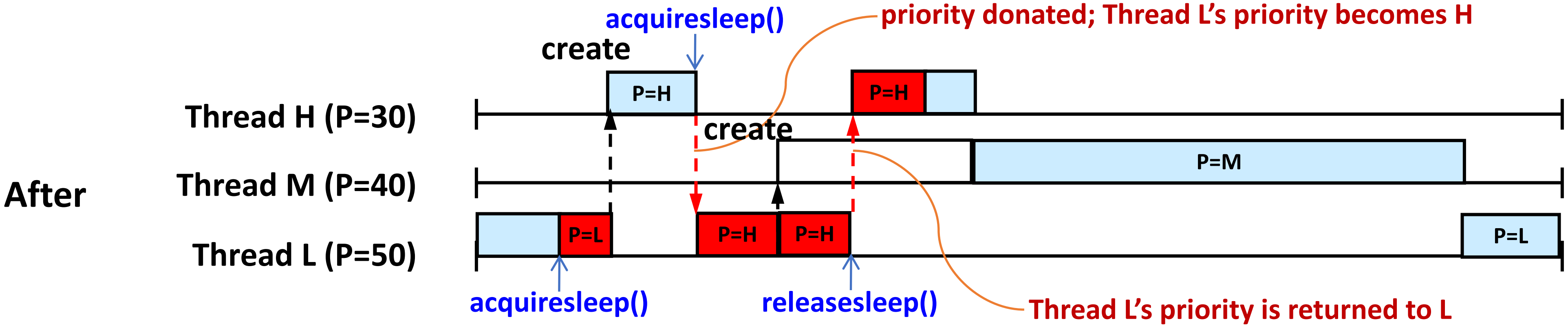
- The higher-priority thread (donor) can donate its priority to the lower-priority thread (donee) holding the resource it requires.
- The donee will get scheduled sooner since its priority is boosted due to donation
- When the donee finishes its job and releases the resource, its priority is returned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

Priority donation

Before

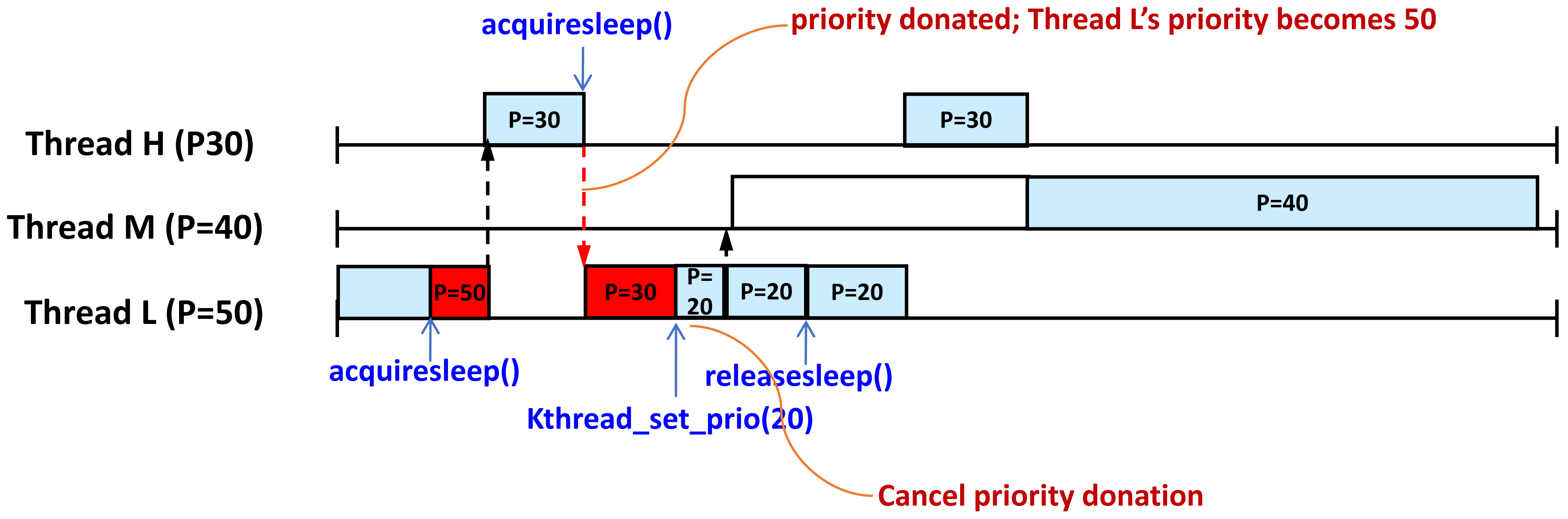


After

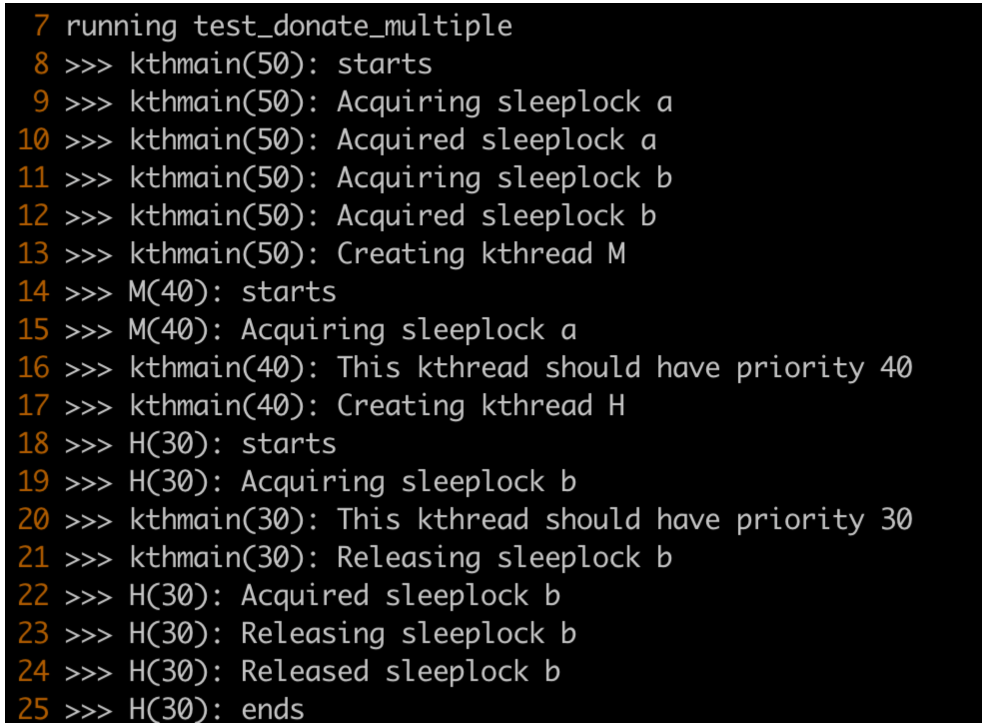


Cancel donation

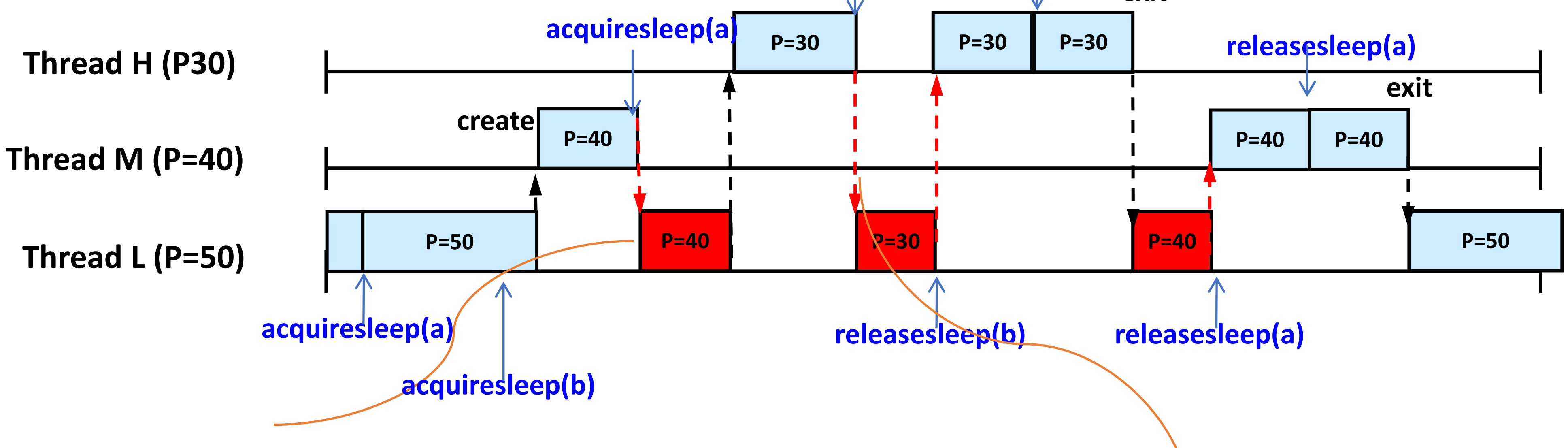
- kthread_setprio() example



Multiple donation



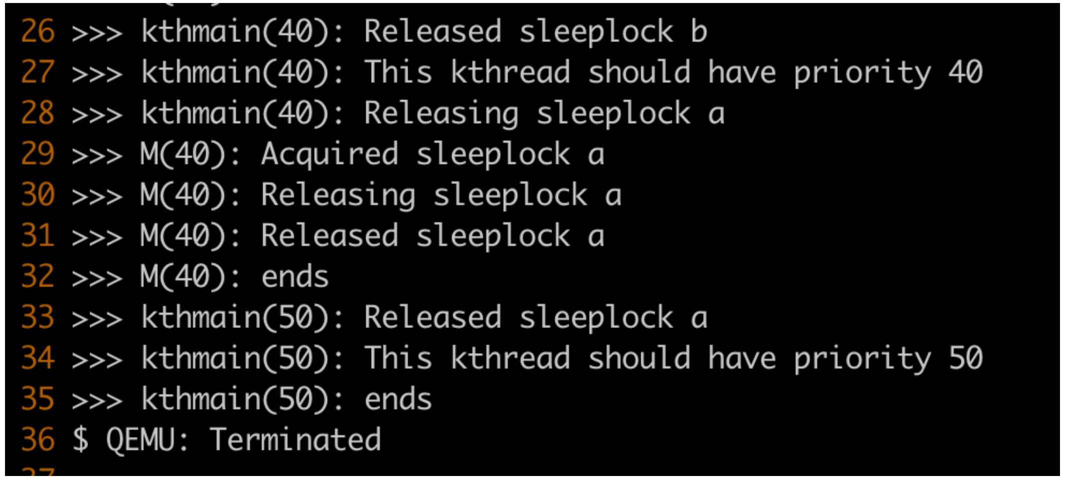
```
running test_donate_multiple
>>> kthmain(50): starts
>>> kthmain(50): Acquiring sleeplock a
>>> kthmain(50): Acquired sleeplock a
>>> kthmain(50): Acquiring sleeplock b
>>> kthmain(50): Acquired sleeplock b
>>> kthmain(50): Creating kthread M
>>> M(40): starts
>>> M(40): Acquiring sleeplock a
>>> kthmain(40): This kthread should have priority 40
>>> kthmain(40): Creating kthread H
>>> H(30): starts
>>> H(30): Acquiring sleeplock b
>>> kthmain(30): This kthread should have priority 30
>>> kthmain(30): Releasing sleeplock b
>>> H(30): Acquired sleeplock b
>>> H(30): Releasing sleeplock b
>>> H(30): Released sleeplock b
>>> H(30): ends
```



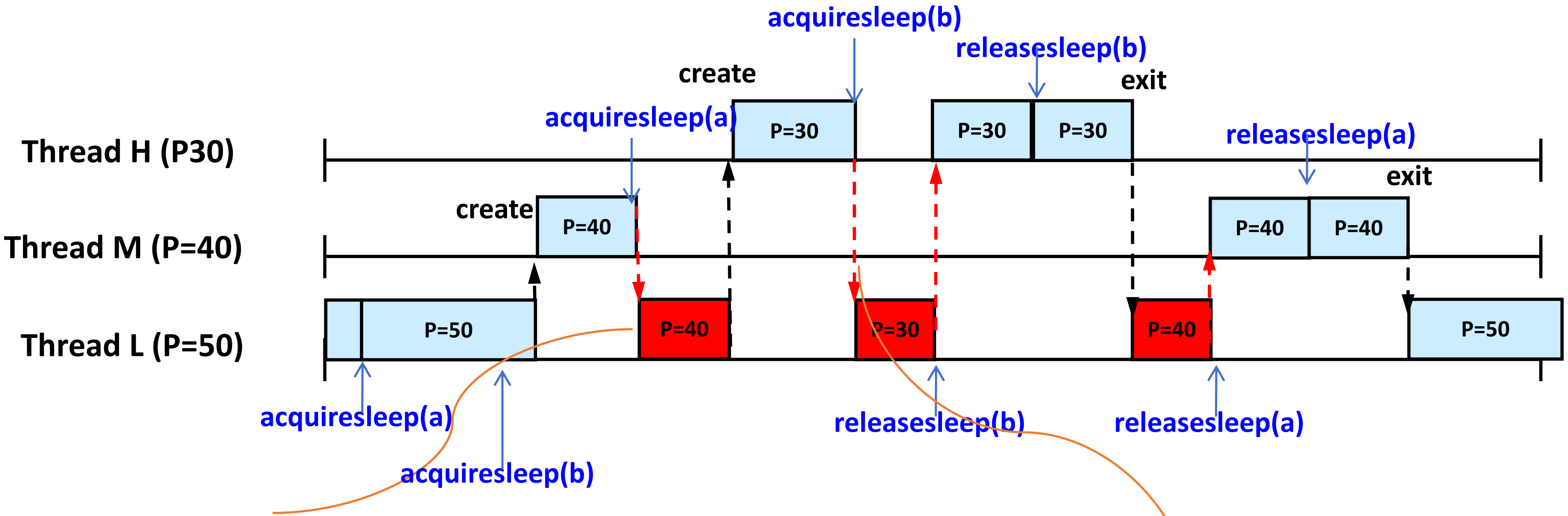
priority donated; Thread L’s priority becomes 40

priority donated; Thread L’s priority becomes 30

Multiple donation



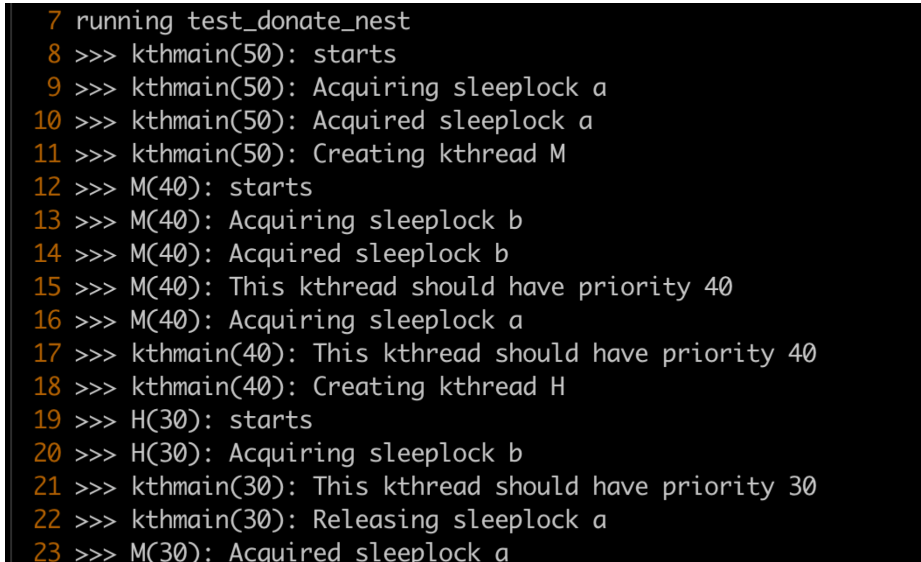
```
>>> kthmain(40): Released sleeplock b
>>> kthmain(40): This kthread should have priority 40
>>> kthmain(40): Releasing sleeplock a
>>> M(40): Acquired sleeplock a
>>> M(40): Releasing sleeplock a
>>> M(40): Released sleeplock a
>>> M(40): ends
>>> kthmain(50): Released sleeplock a
>>> kthmain(50): This kthread should have priority 50
>>> kthmain(50): ends
$ QEMU: Terminated
```



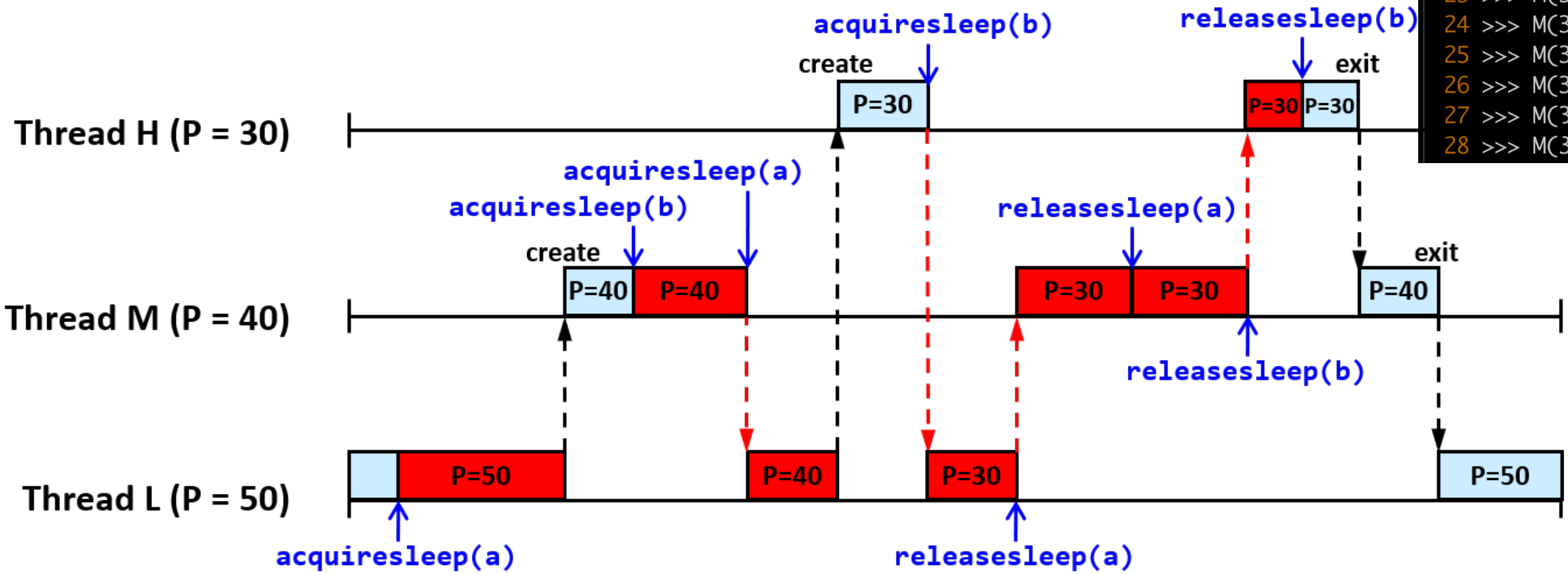
priority donated; Thread L's priority becomes 40

priority donated; Thread L's priority becomes 30

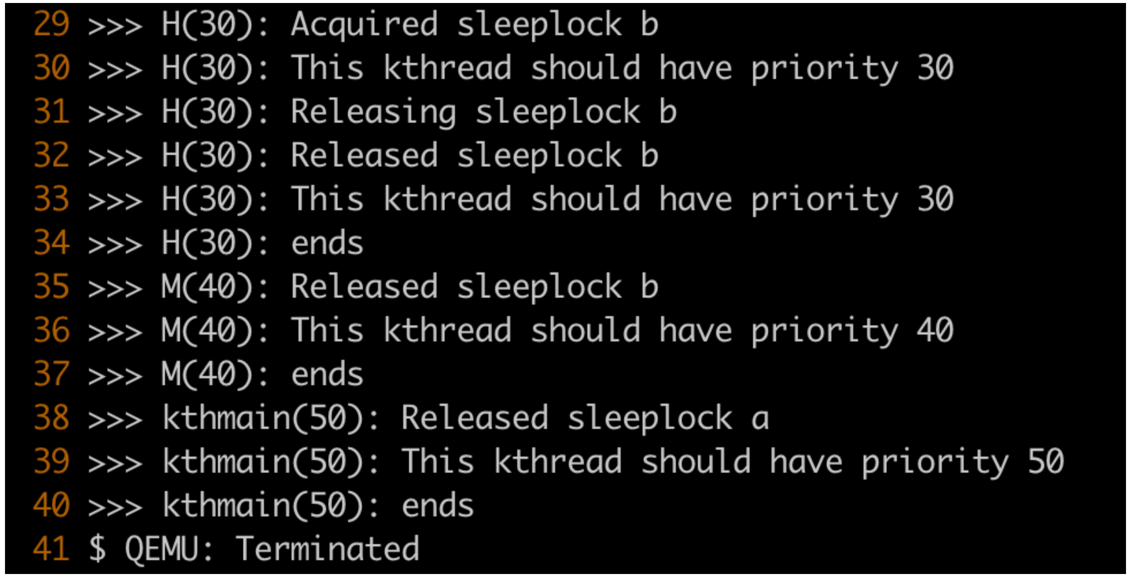
Nested donation



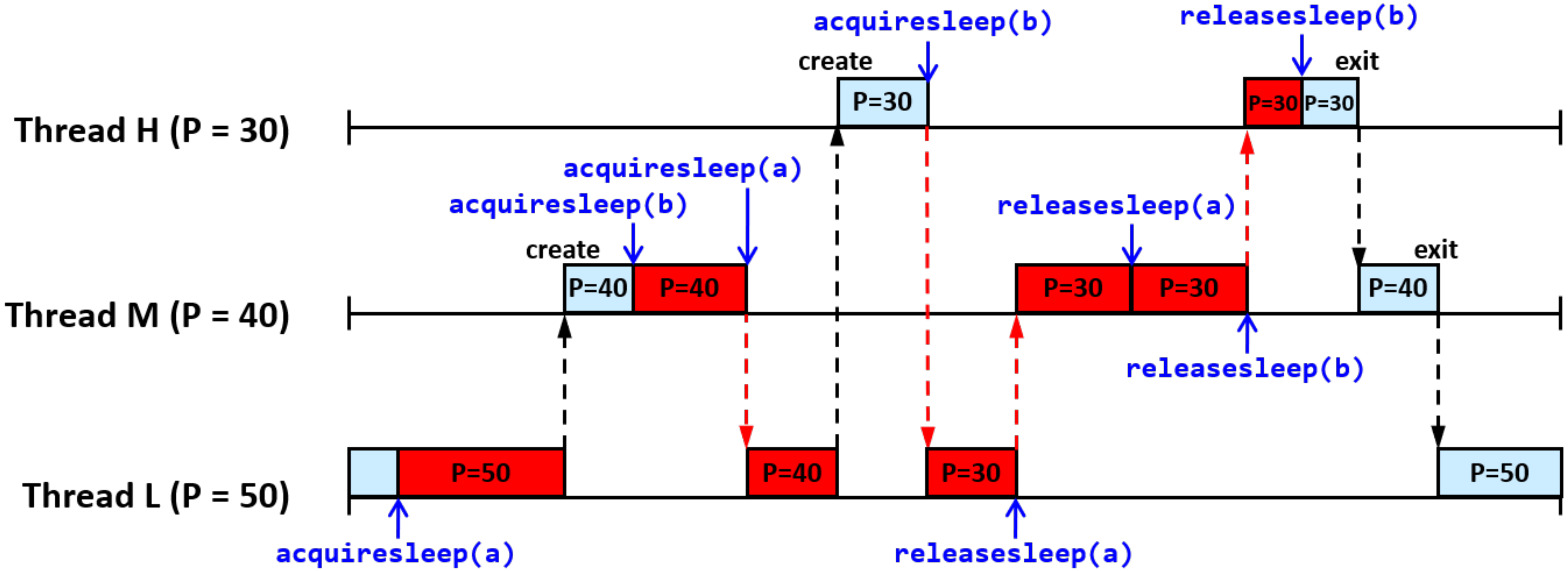
```
running test_donate_nest
>>> kthmain(50): starts
>>> kthmain(50): Acquiring sleeplock a
>>> kthmain(50): Acquired sleeplock a
>>> kthmain(50): Creating kthread M
>>> M(40): starts
>>> M(40): Acquiring sleeplock b
>>> M(40): Acquired sleeplock b
>>> M(40): This kthread should have priority 40
>>> M(40): Acquiring sleeplock a
>>> kthmain(40): This kthread should have priority 40
>>> kthmain(40): Creating kthread H
>>> H(30): starts
>>> H(30): Acquiring sleeplock b
>>> kthmain(30): This kthread should have priority 30
>>> kthmain(30): Releasing sleeplock a
>>> M(30): Acquired sleeplock a
>>> M(30): This kthread should have priority 30
>>> M(30): Releasing sleeplock a
>>> M(30): Released sleeplock a
>>> M(30): This kthread should have priority 30
>>> M(30): Releasing sleeplock b
```



Nested donation



```
>>> H(30): Acquired sleeplock b
>>> H(30): This kthread should have priority 30
>>> H(30): Releasing sleeplock b
>>> H(30): Released sleeplock b
>>> H(30): This kthread should have priority 30
>>> H(30): ends
>>> M(40): Released sleeplock b
>>> M(40): This kthread should have priority 40
>>> M(40): ends
>>> kthmain(50): Released sleeplock a
>>> kthmain(50): This kthread should have priority 50
>>> kthmain(50): ends
$ QEMU: Terminated
```



Priority donation

- You need to modify acquiresleep() and releasesleep().
- If you do acquirelock, you have to change the effective priority accordin g to the Prioirty Donation.
- If you do releasesleep, change effective priority to the base priority and switch the context using kthread_yield().

Design documents

- New or changed data structures for this project
- How to make the newly created kernel thread start from the given function
- How to deliver an argument to the new kernel thread
- How to implement preemptive priority schduler
- How to ensure the highest priority kernel thread wating for a sleeplock wakes up fir st
- Overall flow of acquiresleep() and releasesleep()
- How to support multiple priority donation and nested priority donation

Late Submission policy

- You can use up to 5 slip days for this semester
  - You should explicitly declare the number of slip days to use in the Q&A board on the submission server
  - https://sys.snu.ac.kr/main.php?classIdx=1&menu=Board
- 25% penalty per day after slip day

Thank you!

- This is the last assignment.
- If you have all the slip days, you can submit them by June 21+5.
- If your implementation operate in multi-processor environment, you will get bonus points.

- Any questions?