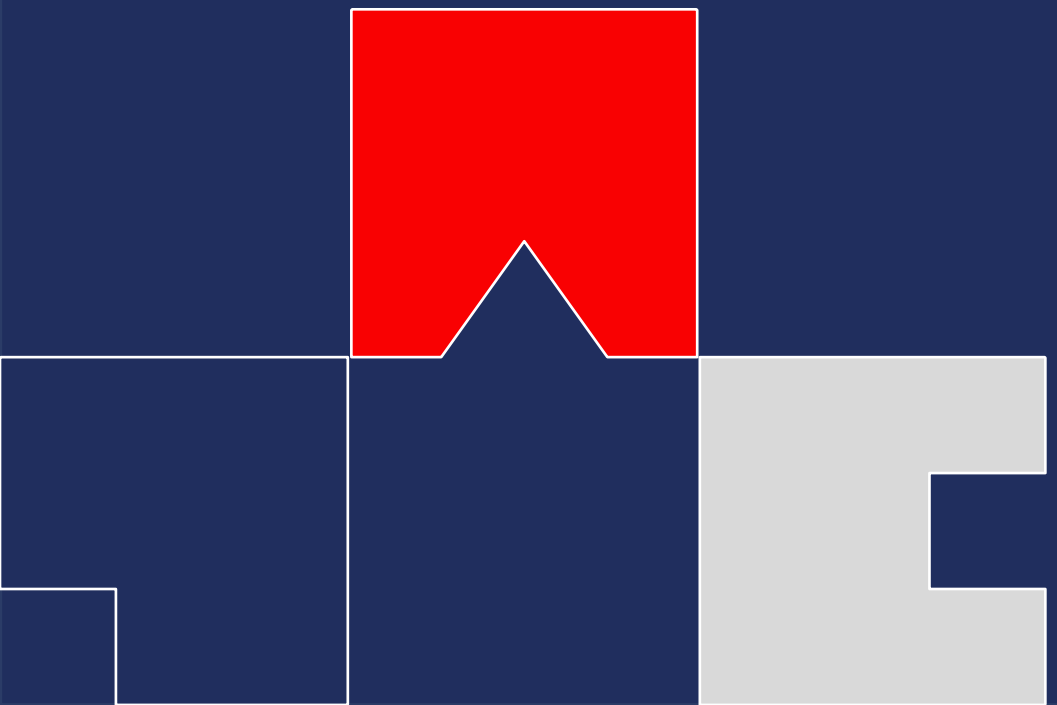
Operating Systems Design
PintOS Part1 : Thread



CONTENTS

Notice

- All implementations on the slides are merely illustrative examples, and alternative approaches are also feasible.

- Additionally, feel free to incorporate any necessary functions or features required for the implementation..

1

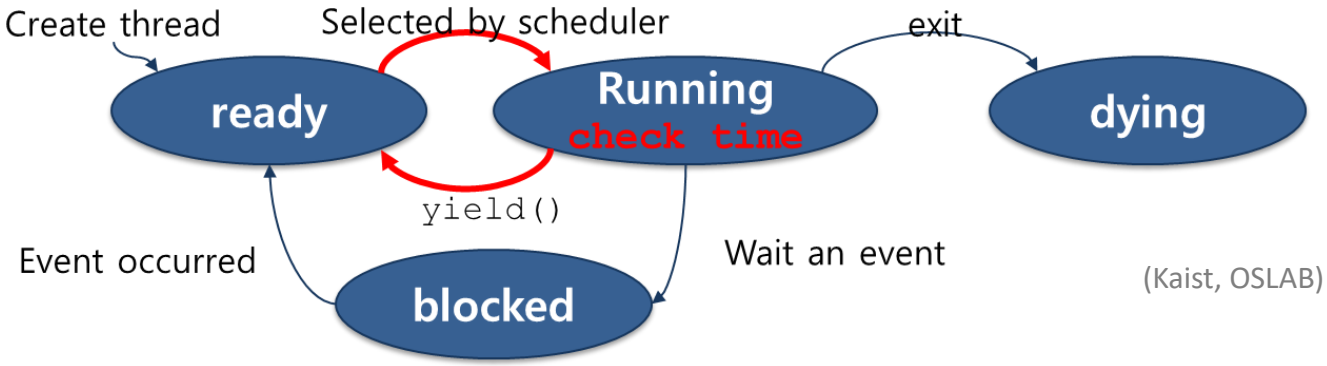
Alarm clock

Alarm clock

- **What is Alarm?**
  - "Alarm" is a kernel internal function that restarts a designated process after a specified time.

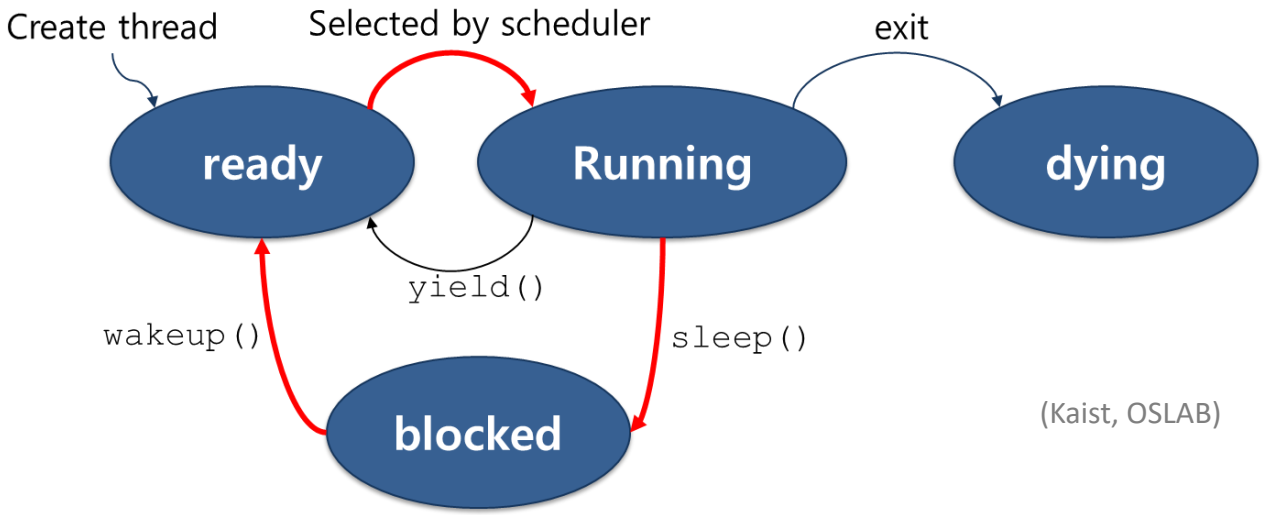
- **Main goal**
  - Current Pintos uses busy-waiting for alarm clock, so we try to modify PintOS to use sleep/wakeup instead of busy-waiting.

- **What is busy-waiting**
  - Current Pintos uses busy-waiting for alarm clock.
  - When a thread occupies the CPU while waiting, it causes wastage of CPU resources.
  - Process
    1. When a thread in the running state receives a sleep command(timer_sleep()), and save the starting time
    2. The thread yields CPU resources, transitions to the ready state and gets added to the ready queue (thread_yield()).
    3. Threads in the ready queue transition to the running state regardless of whether it's their scheduled wake-up time.
    4. Once in the running state, a thread checks if it's time to wake up(Measure the elapsed time using the starting time and the current time in the timer_tick()); if not, it transitions back to the ready state.



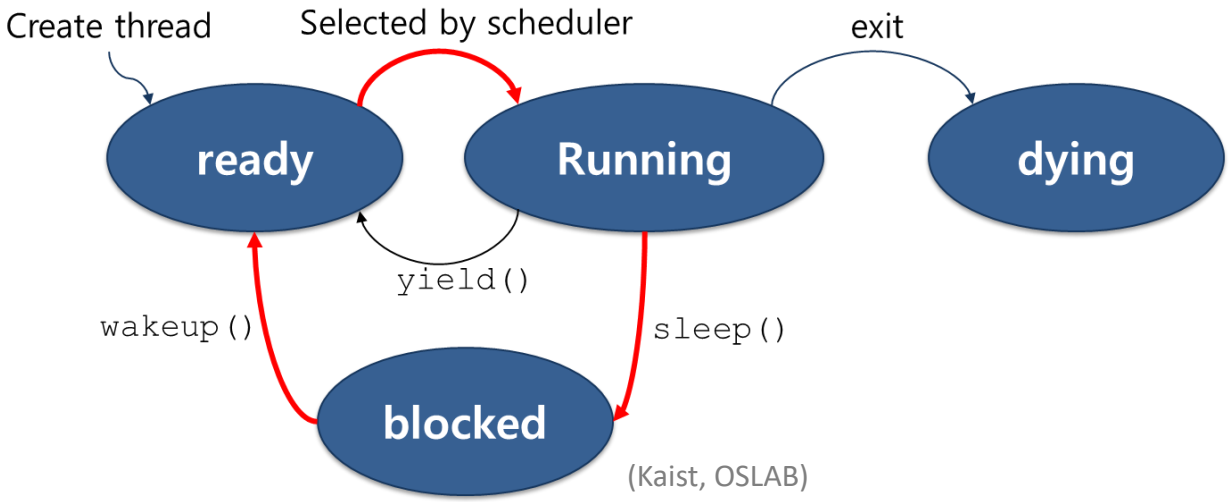
(Kaist, OSLAB)

- **How to improve busy-waiting? → sleep/awake**
  - Placing a sleeping thread into the block state rather than the ready state ensures that it is excluded from scheduling until its scheduled wake-up time. When it's time to wake up, transitioning it to the ready state suffices.
  - This approach conserves CPU cycles and reduces power consumption.



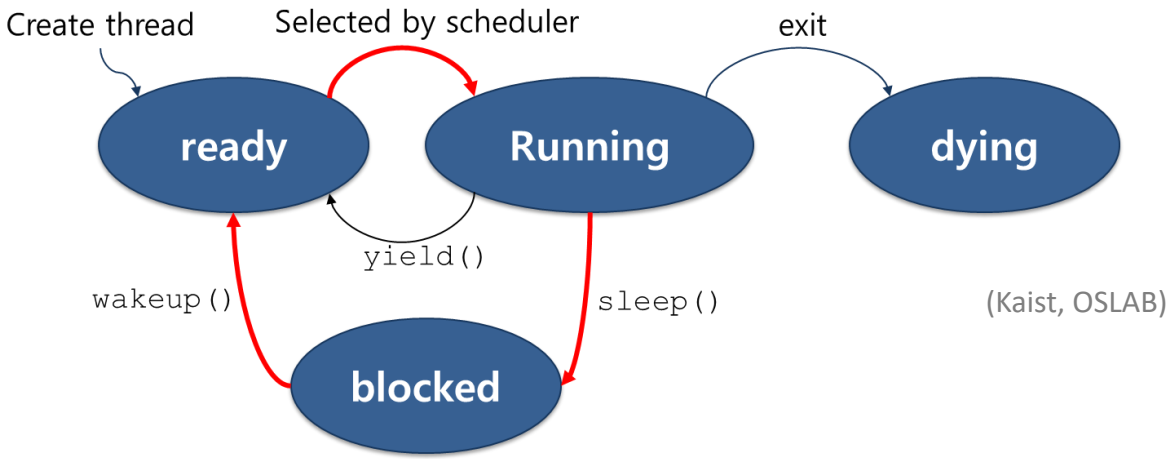
(Kaist, OSLAB)

- **How to implement sleep/awake?**
  - **thread_init**(`void`)
    - The existing Pintos system utilizes the ready_list to manage threads awaiting execution and the all_list to track all threads within the system.
    - In the sleep/awake paradigm, an additional sleep_list is required to store threads in the blocked state.
  - **timer_sleep**(`int64_t ticks`)
    - interrupt is disabled, and Thread_sleep(wake_up_time) is called (wake_up_time : the time at which the thread should wake up)
      - The current thread's local tick stores the wake_up_time.
      - The current thread is stored in the sleep list, and a context switch is invoked to change the thread's state to blocked (thread_block()).



(Kaist, OSLAB)

- **How to implement sleep/awake?**
  - **timer_interrupt**(`struct intr_frame *args UNUSED`)
    - The timer interrupt automatically sends an interrupt signal to the CPU based on a pre-set interval of the hardware timer (in this case, 1 millisecond), without relying on operating system commands or software requests.
    - ticks += 1 (ticks : internal value within Pintos used to measure time)
    - **thread_wakeup**(`int64_t ticks`) (in timer_interrupt)
      - sequentially checks all threads within the sleep_list, which are currently in a blocked stat
      - if the thread's local tick matches the current time, it is placed into the ready_list and its state is changed to ready using thread_unblock().
        - Local tick : the time when the respective thread is scheduled to wake up



Alarm clock

- **How to implement sleep/awake?**

Threads/thread.*

- **Modify thread structure. (threads/thread.h)**
  : Add local tick
- **Modify thread_init(`void`) function. (threads/thread.c)**
  : Add the code to initialize the sleep queue data structure.
- **Add thread_sleep(`int64_t ticks`) function. (threads/thread.c)**
  : Set thread state to blocked and wait after insert it to sleep queue.
- **Add thread_wakeup(`int64_t ticks`) function. (threads/thread.c)**
  : Find the thread to wake up from sleep queue and wake up it.

Devices/timer.*

- **Modify timer_sleep(`int64_t ticks`) function. (devices/timer.c)**
  : Call the function that insert thread to the sleep queue.
- **Modify timer_interrupt(`struct intr_frame *args UNUSED`) function. (devices/timer.c)**
  : At every tick, check sleep list find any threads to wake up, move them to the ready_list, if necessary,For the threads to wake up, remove them from the sleep queue and insert it to the ready_list.

- **Result Check**
  - **$ pintos -- -q run alarm-multiple**
  - **Busy waiting**

```
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 577 ticks
Thread: 0 idle ticks, 577 kernel ticks, 0 user ticks
Console: 2954 characters output
Keyboard: 0 keys pressed
Powering off...
```

  - **Sleep/awake**

```
(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
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 584 ticks
Thread: 550 idle ticks, 35 kernel ticks, 0 user ticks
Console: 2955 characters output
Keyboard: 0 keys pressed
Powering off...
```

    - Previously, when the busy-waiting method was implemented, the output showed "Thread: 0 idle ticks." However, it changed to "550 idle ticks." This indicates that the idle thread ran for 550 ticks. The idle thread runs when no other thread is executing, signifying a period of rest lasting 550 ticks. This demonstrates a reduction in system resource wastage with the sleep/awake approach.

2

Priority scheduling

Priority Scheduling

- **Main goal**
  - The current scheduler in Pintos is implemented as FIFO (First-In-First-Out), so we try to modify PintOS to 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 every time when the timer interrupt is called).
  - We will consider 1. priority scheduling, 2.priority synchronization, 3. priority donation

1. Priority scheduling

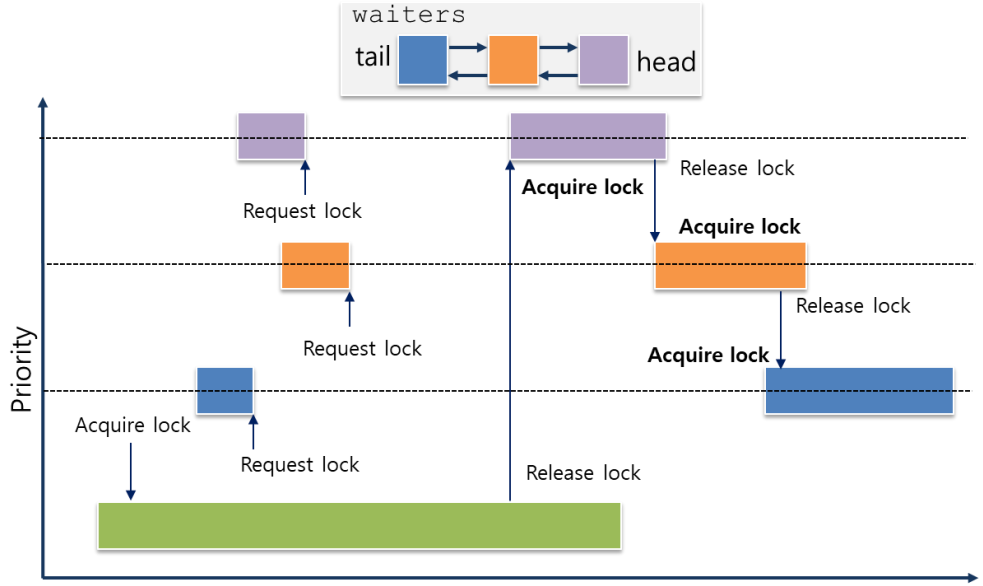
- **You need to consider 3 things**

1. The priority in the current PintOS has the following characteristics:
   - 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.

2. When selecting a thread to run in the ready list, select the one with the highest priority.
3. 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.
   - When selecting a thread from the set of threads waiting for a lock (or condition variable), select the one with the highest priority.

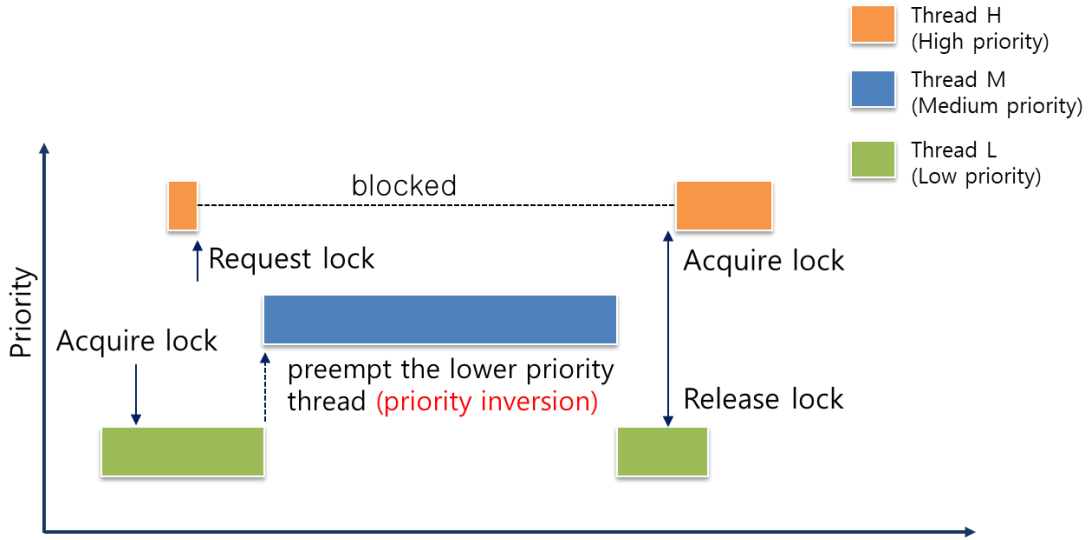
2. Priority synchronization

- You also need to change the synchronization primitives (semaphore, lock, condition variable)
- Implement using semaphores to prevent issues when using shared resources..
  - When threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first
  - waiters : a list for threads waiting when all shared resources are fully occupied



3. Priority donation

- **When you implement priority synchronization, you have to consider 'priority inversion' issue.**
  - Priority inversion : the situation where thread of the higher priority waits thread of the lower priority



Above picture,

1. If a thread(L) with a lower priority is holding the lock, and the running thread(H) requests the lock, the thread(H) enters the Blocked state until the lock is returned.
2. After that, the next priority thread(M) is executed according to the schedule.
3. As a result, even though threads H and M are unrelated, the higher-priority thread H waits for the lower-priority thread M to finish execution. (priority inversion)

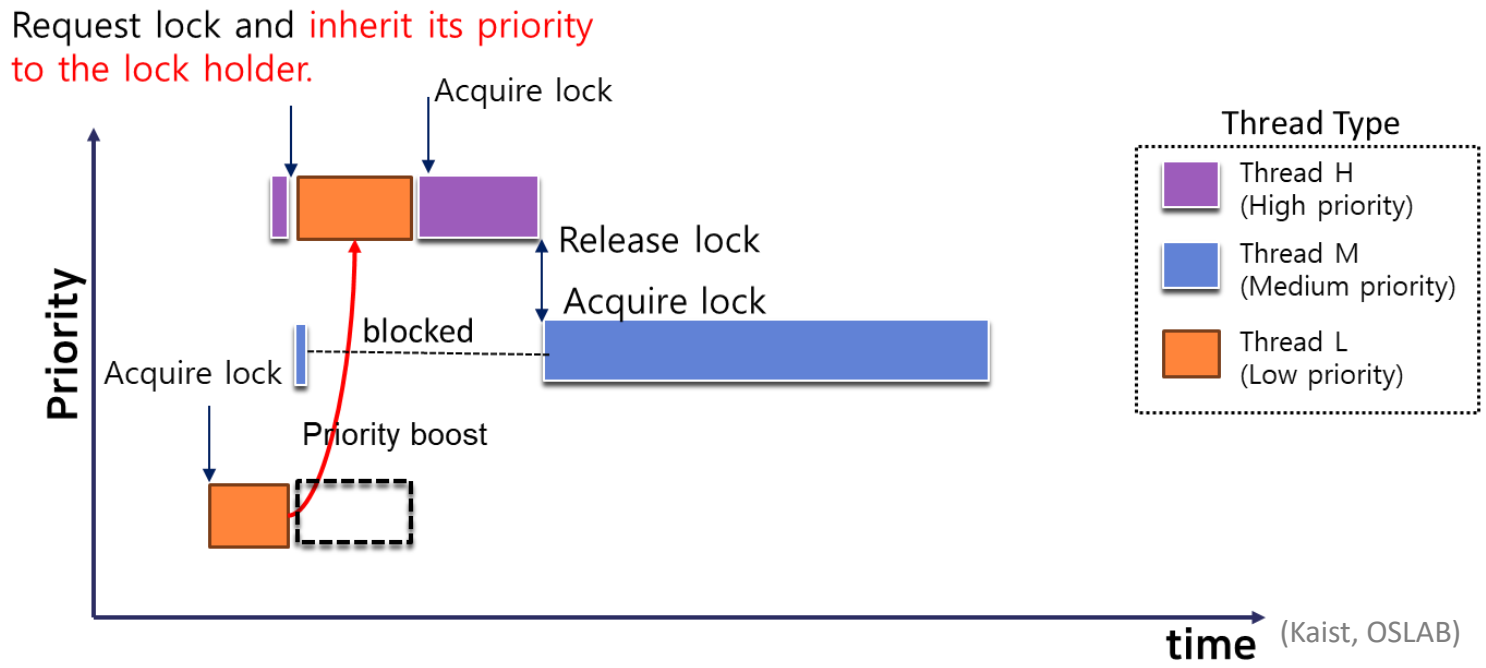
3. Priority donation

- To resolve the priority inversion problem, three things need to be implemented.
  1. Priority donation
  2. Multiple donation
  3. Nested donation

3. Priority donation

1. Priority donation

- To a thread that holds a lock and has a lower priority, the thread that requested the lock will donate its priority.
- Allows execution until the donated thread executes and releases the lock.
- If the lock holder thread is running at the donated priority and the lock is released, the thread will revert to its original priority.

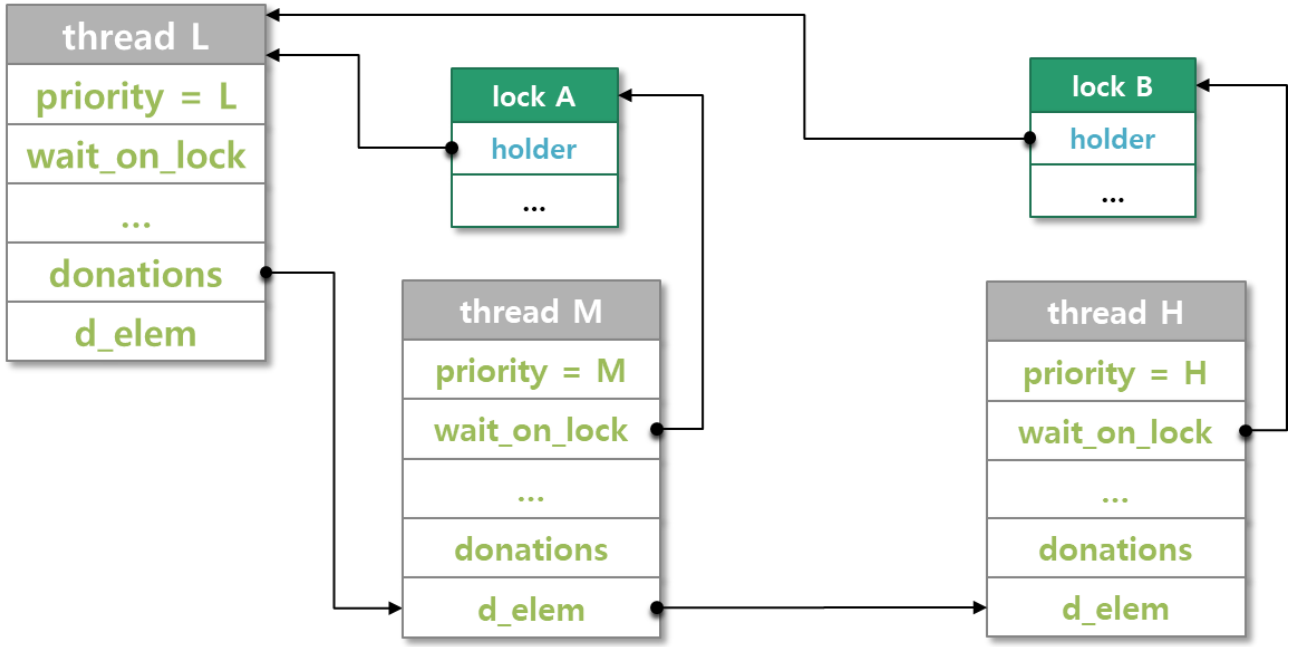


(Kaist, OSLAB)

3. Priority donation

2. Multiple donation

- When multiple priorities are donated to a single thread, upon releasing a lock, the priority used is determined by selecting the highest value between the initial priority and the priorities in the donation list.



(Kaist, OSLAB)

3. Priority donation

3. Nested donation

- In the figure below, suppose that thread L holds lock A.

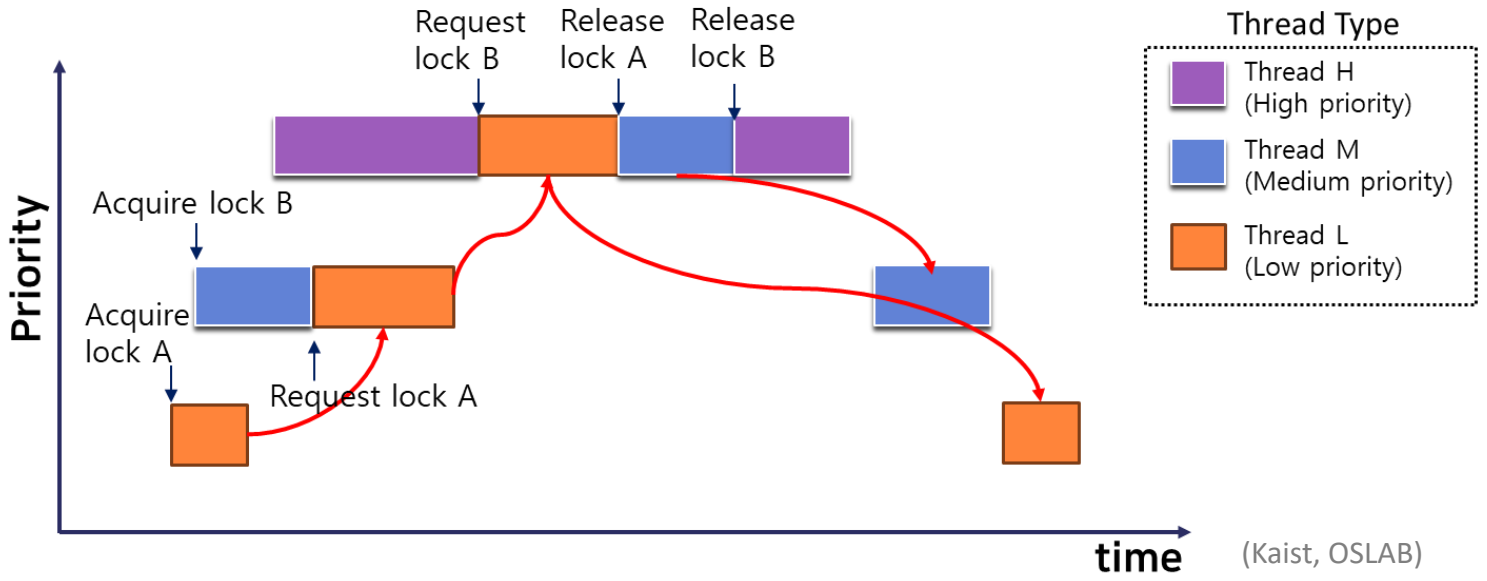
1) Thread M has lock B, and requests lock A. Thread M gives its priority to thread L (priority donation).

2) Thread H requests lock B. Thread H gives its priority to thread M, which holds lock B (priority donation)

3) Thread L has the same priority as thread H because it was donated thread M's prioirty.

4) When thread L releases lock A, it reverts back to its original priority (init_priority).

5) Repeats this behavior.



(Kaist, OSLAB)

Priority Scheduling

- **How to implement priority scheduling?**
  - **thread_create(**`const char *name, int priority, thread_func *function, void *aux`**) function**
    - 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 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
thread and the newly inserted one. Yield the CPU if the
newly arriving thread has higher priority*/
    return tid;
}
```

Priority Scheduling

- **How to implement priority scheduling?**
  - **thread_set_priority(`int new_priority`)**
    - Priority : priority value that is changed by donations
    - Init_priority : the initial priority value that the thread has
    - This function changes Init_priority to new_priority.
    - Thread must have the higher of the donated priority value and init_prioirty as its priority value.
    - The thread's initial priority (init_priority) and the changing priority (priority) due to donation should be dual managed. (The priority of a thread may change during lock acquisition and release.)
    - If any of the threads in the ready state have a higher priority than the current thread, then call thread_yield()

Priority Scheduling

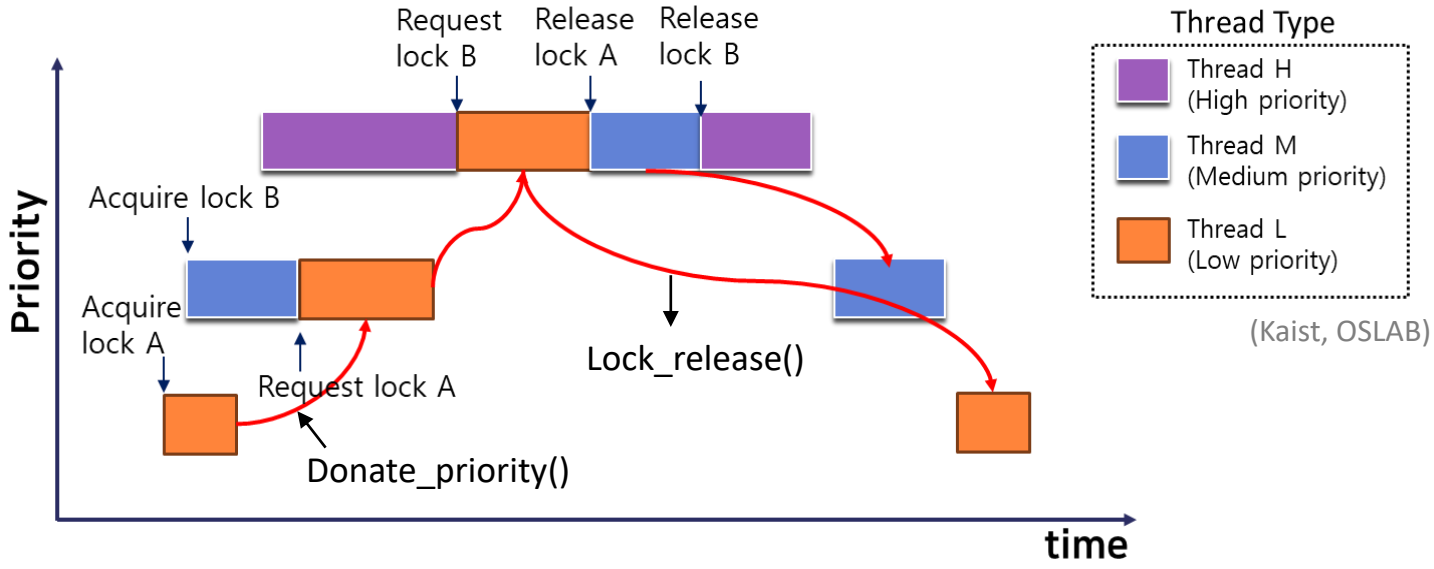
- **How to implement priority scheduling?**
  - **thread_wakeup(int64_t ticks)**
    - If local tick is less than tick (when it's time to get up), call thread_unblock(). (blocked -> ready)
    - At this point, the ready_list should be reordered by the priority of the thread.
  - **thread_yield(void)**
    - Change the currently running thread to the ready state. (running -> ready)
    - This requires reordering the ready_list by priority.

- **How to implement priority scheduling?**
  - **sema_down(`struct semaphore *sema`)**
    - When the semaphore value is 0, put the thread in the waiters list, sort the list by priority, and change the thread to block state (thread_block()) (running -> block)
      - Waiters : a list of threads waiting for a resource. This waiters list is used to hold threads until the resource becomes available again
    - When the semaphore value is non-zero, decrease the value of the semaphore by one and occupy a resource.
  - **sema_up(`struct semaphore *sema`)**
    - Increases the semaphore value by one (freeing a resource, increasing the number of available resources by one).
    - If waiters list is not empty, the thread with the highest priority waiting in waiters_list is changed to the ready state. (thread_unblock()) (blocked -> ready)
    - When interrupts are disabled, preemptive actions are not possible, allowing the code to execute atomically.
    - Change the unlocked thread to ready_state (thread_yield()), only if the current code is not executing in interrupt context.
    - The purpose of this logic is to allow the current thread to yield the CPU when resources become available by releasing the semaphore (sema_up), if a higher priority thread is in the ready state.

- **How to implement priority scheduling?**
  - **lock_acquire(`struct lock *lock`)**
    - Before a thread accesses a shared resource, it acquires the associated lock to gain exclusive access to that resource.
    - If there is no holder thread occupying the lock requested by the currently executing thread, decrement the semaphore value by one and assign the lock to the current thread.
    - If there is a holder thread that currently occupies the lock requested by the running thread,
      - update the running thread's 'wait_on_lock' value with that lock.
      - maintain donated threads on donations list of lock holder, by adding the current thread to the holder's donation list
    - **donate_priority(`void`)** (in lock_acquire())
      - If holder's priority is lower than the current thread's priority, donate the current priority to holder.

- **How to implement priority scheduling?**
  - Donate priority example
    1. Initial state:
       - Thread A acquires and holds lock X.
       - Thread B acquires and holds lock Y.
       - Thread C acquires and holds lock Z.
    2. Priority donation situation:
       - Thread A acquires an additional lock Y, but this lock is already held by Thread B.
       - In this case, Thread A must wait for Thread B's work to complete in order to satisfy its request.
       - If Thread A's priority is higher than Thread B's, then Thread A donates its priority to Thread B. This causes Thread B to have the high priority that it received from Thread A
    3. Cascading priority donations:
       - If Thread B also acquires lock Z, and this lock is held by Thread C, then Thread B must wait for lock Z, which is also held by Thread C.
       - Having received a priority donation from Thread A, Thread B now donates this higher priority to Thread C. Eventually, Thread C will inherit priority from both Threads A and B.
    4. Result:
       - Through this cascading priority donation, the thread that holds the lock C has a higher priority, making it more likely to execute before the threads A, B that don't hold the lock
       - Priority donation is repeated, and this process continues until there is no more priority to inherit. This helps prevent deadlocks between threads and minimizes the latency of high-priority tasks.

- **How to implement priority scheduling?**
  - **lock_release(`struct lock *lock`)**
    - Releasing a lock after removing the holder thread that had the lock.
    - **refresh_priority(`void`) (in lock_release())**
      - If no priority donation has been received, the currently running thread will revert to its initially set priority (init_priority).
      - On the other hand, if any threads have donated priority to the current thread, it looks for the thread with the highest priority in the priority donation list. If this priority is higher than the current thread's priority, it updates the current thread's priority.
    - And call sema_up()



- **How to implement priority scheduling?**
  - Condition variable
    - Variables used by monitors to synchronize access to shared resources.
    - Condition variables are typically used in conjunction with locks to cause a thread to wait for a specific condition to be met.
    - When a thread acquires a lock, it checks the condition on the shared resource. The condition refers to a specific state that must be met for the thread to perform its work.
    - If the conditions are met, the waiting thread reacquires the lock and continues to execute.
      - Each process must execute wait before entering the critical region and signal when exiting the critical region. This ordering is important because if it is not followed correctly, both processes can be in the critical zone at the same time.
    - A thread 'waits' (cond_wait()) for a condition variable, and is unlocked and wake up until another thread 'signals' (cond_signal()) that the condition has been met.

- **How to implement priority scheduling?**
  - **cond_wait(**`struct condition *cond, struct lock *lock`**)**
    - When a thread calls the cond_wait function, the function first prepares to transition that thread to the blocked state. It does this by creating a new semaphore with a value of 0, allowing the thread to wait on it.
    - To allow other threads to access the shared resource and perform the necessary condition changes
    - Call the sema_down function to wait on the created semaphore. Calling sema_down puts the thread in the block state, which causes the thread to stop executing and wait until the condition is met.
    - When another thread satisfies the condition and calls cond_signal or cond_broadcast, one or all of the waiting threads are woken up. The woken threads will again attempt to acquire the associated lock. If the lock is successfully acquired, the thread assumes that the condition it was waiting for has been satisfied and resumes execution.
    - **cond_signal (**`struct condition *cond, struct lock *lock UNUSED`**)** (in cond_wait())
      - Wake up the highest priority thread when the wait list for a conditional variable is non-empty..
  - **cond_broadcast(**`struct condition *cond, struct lock *lock`**)**
    - Sends a signal to all threads on the waitlist for in conditional variable.

Priority Scheduling

- **How to implement priority scheduling?**

threads/thread.*

- Implement a function that compare the priority between two thread.
- Implement a function that compare the priority and run the void thread_yield (void) depending on the condition.
- **Modify tid_t thread_create (const char *name, int priority, thread_func *function, void *aux) function. (threads/thread.c)**
  - 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 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
thread and the newly inserted one. Yield the CPU if the
newly arriving thread has higher priority*/
    return tid;
}
```

How to implement priority scheduling?

threads/thread.*

- **Modify void thread_set_priority (int new_priority) function. (threads/thread.c)**
  - Set priority of the current thread, and reorder the ready_list
- **Modify void thread_unblock (struct thread *t) function. (threads/thread.c)**
  - When the thread is unblocked, it is inserted to ready_list in the priority order
  - Use list_insert_ordered instead of list_push_back

pintos/src/threads/thread.c

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

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

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

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

(Kaist, OSLAB)

- **Modify void thread_yield (void) function. (threads/thread.c)**
  - The current thread yields CPU and it is inserted to ready_list in priority order

Priority Scheduling

- **How to implement priority synchronization?**

threads/synch.*

- **Modify void sema_down (struct semaphore *sema) function. (threads/synch.c)**
- **Modify void cond_wait (struct condition *cond, struct lock *lock) function. (threads/synch.c)**
- **Modify void sema_up (struct semaphore *sema) function. (threads/synch.c)**
- **Modify void cond_signal (struct condition *cond, struct lock *lock UNUSED) function. (threads/synch.c)**
- **Implement a function that compare the priority of sema. (threads/synch.*)**

Priority Scheduling

- **How to implement priority donation?**
  - **Modify thread structure (threads/thread.h)**
  - **Implement a function that update priority (threads/thread.c)**
  - **Add void donate_priority (void) function (threads/thread.c)**
  - **Modify init_thread (struct thread *t, const char *name, int priority) function (threads/thread.c)**
  - **Modify void thread_set_priority (int new_priority)function (threads/thread.c)**
  - **Modify void lock_acquire (struct lock *lock)function (threads/synch.c)**
  - **Modify void lock_release (struct lock *lock)function (threads/synch.c)**

Priority Scheduling

- **Result**
  - $ cd ~pintos/src/threads
  - $ make clean
  - $ make
  - $ make check

```
root@af5c69f752e8:/pr1/home/islab/pintos/src/threads# make check
cd build && make check
make[1]: Entering directory `/pr1/home/islab/pintos/src/threads/build'
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
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

- Check if the test case corresponding to this section(priority-scheduling) has passed.

- **Result**
  - If a timeout error occurs when entering "make check," try changing the code in pintos/src/devices/shutdown.c to the code provided in the attached link and then try again.
  - https://github.com/t3rm1n4l/pintos/blob/master/devices/shutdown.c : Fix broken pintos shutdown with QEMU

3

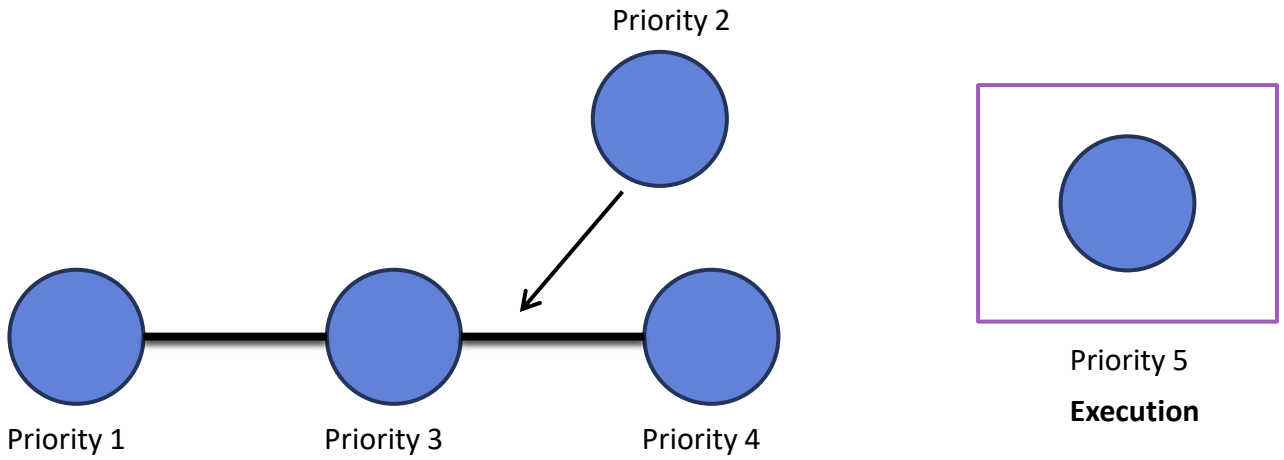
MLFQ:4.4 BSD scheduler

MLFQ :4.4 BSD scheduler

- **Main goal**
  - Implement 4.4 BSD scheduler MLFQ like scheduler
    - Give priority to the processes with interactive nature.
    - Priority based scheduler
    - Prevent starvation by updating the priority
  - Use nice, recent_cpu ,load_avg, for update the priority
  - The Pintos kernel can only perform integer arithmetic
  - Need to implement the fixed-point arithmetic for calculate the priority

MLFQ :4.4 BSD scheduler

- **Starving**
  - If new process with a high priority keeps comming then a process with priority 1 will be starved(never executed)



  - 4.4 BSD scheduler prevent this problem with updating the priority
  - 4.4 BSD scheduler Use nice, recent_cpu ,load_avg, for update the priority

MLFQ :4.4 BSD scheduler

- **Nice value**
  - Represents the ‘niceness’ of a thread.
  - If a thread is nicer, it is willing to give up some of its CPU time

- **Value (from -20 to 20)**
  - Nice (0) : not influence on priority. (initial value)
  - Nice (positive) : decrease priority.
  - Nice (negative) : increase priority.

MLFQ :4.4 BSD scheduler

- **Update the cpu usage(Lower the priority of a process if it has recently used a lot of CPU)**
  - Recent_cpu : A value indicating how much CPU that thread has used recently.
  - Increase the `recent_cpu` of the currently running process by 1 in every timer interrupt.
  - Decay `recent_cpu` by decay factor in every hundred tick

$$\texttt{recent\_cpu} = \texttt{decay} * \texttt{recent\_cpu}$$

  - Adjust `recent_cpu` by `nice` in every hundred tick

$$\texttt{recent\_cpu} = \texttt{recent\_cpu} + \texttt{nice}$$

  - Putting them together,

$$\texttt{recent\_cpu} = \texttt{decay} * \texttt{recent\_cpu} + \texttt{nice}$$

MLFQ :4.4 BSD scheduler

- **Decay factor In 4.4BSD**
  - In heavy load, decay is nearly 1.
  - In light load, decay is 0.

    decay = (2*load_average)/ (2*load_average + 1)

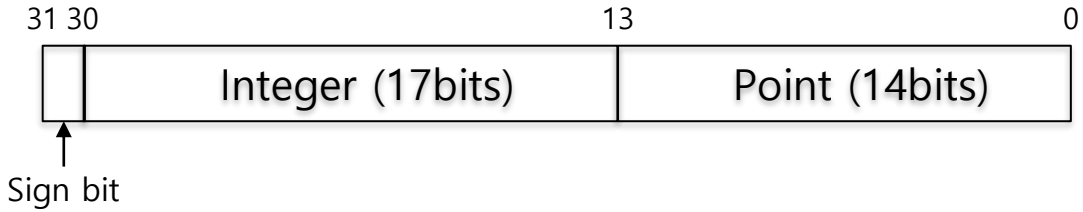
  - load_average
    - load_avg = (59/60)*load_avg + (1/60)*ready_threads
      - At booting, load_avg is initially set to 0.
      - ready_threads: the number of threads in ready_list and threads in executing at the time of update. (except idle thread)
      - Load_avg : The average number of threads recently performed. Indicates how busy the system is.
      - The calculation of load_avg influences how priorities are determined: a lower load_avg relies on the nice value for priority calculations, indicating less system load. Conversely, a higher load_avg suggests a greater need for aging, leading to an increased decay factor in the formula. This effectively lowers the priority of threads that have recently consumed a significant amount of CPU time, ensuring fair resource distribution.

MLFQ :4.4 BSD scheduler

- **Summary**
  - In every fourth tick, recompute the priority of all threads
    - priority = PRI_MAX – (recent_cpu / 4) – (nice * 2)
  - In every hundred tick, increase the running thread's recent_cpu by one.
  - In every hundred tick, update every thread's recent_cpu
    - recent_cpu = decay * recent_cpu + nice, where
    - decay = (2*load_average)/ (2*load_average + 1)
    - load_avg = (59/60)*load_avg + (1/60)*ready_threads

MLFQ :4.4 BSD scheduler

- **Fixed point arithmetic**
  - Inside kernel, you can do only integer arithmetic.
    - Kernel does not save floating point register when switching the context.
  - We need to implement fixed point arithmetic using integer arithmetic.
    - `priority`, `nice`, `ready_threads` value is integer, and `recent_cpu`, `load_avg` value is real number.
  - Implement the fixed-pint arithmetic using 17.14 fixed-point number representation.
    - Decimal point is 14 right-most bits.
    - Integer is 17 next bits to the left.
    - Last of left 1bit is sign bit.
  - You need to implement fixed point arithmetic



- **How to implement 4.4 BSD scheduler**
  - During the execution of MLFQ tests, the value of **thread_mlfqs is set to true automatically.**
  - When considering the 4.4 BSD scheduler in the context of the Multi-Level Feedback Queue (MLFQ), priority donation is not taken into account. Thereby **disabling priority donation** when using the 4.4 BSD scheduler.
  - Modify void thread_start (void)
    - initiates the process by initializing the load_avg value, which represents the system's average load.
  - Modify void thread_tick (void) (in thread\thread.c)
    - Every 100 ticks, it recalculates load_avg and recent_cpu to reflect current system conditions.
    - Every 4 ticks, it recalculates the priority of all threads in the all_list

How to implement 4.4 BSD scheduler

Threads/thread.*

- **Modify thread structure.** (threads/thread.h)
  - Add nice,recent_cpu to thread structure
- **Modify init_thread**(struct thread *t, const char *name, int priority) (in thread/thread.c)
  - initialize nice, recent_cpu
- **Modify thread_set_priority**(int new_priority) (in thread/thread.c)
  - Disable the priority setting when using the 4.4 BSD scheduler
- **Modify timer_interrupt**(struct intr frame *args UNUSED) (in device/timer.c)
  - Recalculate load_avg, recent_cpu of all threads, priority every 100th tick.
  - Recalculate priority of all threads every 4th tick.
- **Modify lock_acquire**(struct lock *lock) (in thread/synch.c)
  - When thread_mlfqs is true, allocate the lock to the current thread and perform semaphore down
- **Modify lock_release**(struct lock *lock) (in thread/synch.c)
  - When thread_mlfqs is true perform sema_up

MLFQ :4.4 BSD scheduler

- **How to implement 4.4 BSD scheduler**
  - **Modify int thread_get_load_avg**(void) (in thread/thread.c)
    - timer_ticks() % TIMER_FREQ == 0
    - Return load_avg multiplied by 100
  - **Modify int thread_get_recent_cpu**(void) (in thread/thread.c)
    - Rerturn recent_cpu multiplied by 100
  - **Modify thread_tick** (void)
    - increase recent_cpu by 1 every 1 tick, recalculate load_avg every 100 ticks, and recalculate priority of all threads every 4 ticks
  - **Modify Int thread_get_nice** (void)
  - **Modify void thread_set_nice (int nice)**

MLFQ :4.4 BSD scheduler

- Result check
  - $ cd ~pintos/src/threads
  - $ make clean
  - $ make
  - $ make check

```
root@af5c69f752e8:/pr1/home/islab/pintos/src/threads# make check
cd build && make check
make[1]: Entering directory `/pr1/home/islab/pintos/src/threads/build'
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
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
All 27 tests passed.
```

- Check this section for MLFQ
- When successfully implementing, all 27 test cases pass

Reference

- Kaist Pintos Project1-1 Thread
  - https://www.youtube.com/watch?v=myO2bs5LMak
- Stanford
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15
- Debugging tools
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos_10.html

Thank you