Pintos Project 1. Pintos Scheduler

(설계 프로젝트 수행 결과)

과목명: [CSE4070-01] 운영체제
담당교수: 서강대학교 컴퓨터공학과 이혁준
조원: 54조 고창영

개발기간: 2014. 11. 09. - 2014. 11. 12.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 1. Pintos Scheduler
제출일: 2014. 11. 27.
참여조원: 54조 고창영

1 개발목표

최소한의 기능을 갖는 교육용 OS Pintos를 만들어나가는 과정에서 OS의 다양한 Scheduling 방법들의 구현과 지속적인 개선을 통해 scheduling 전반 및 priority(및 priority scheduling), 더불어 starvation과 aging및 multi-level feedback queue scheduling에 대한 개념들을 직접 실습함으로써 이들에 대한 이해도를 높인다.

2 개발 범위 및 내용

2.1 개발 범위

1. Alarm Clock
2. Priority Scheduling
3. Advanced(4.4BSD) Scheduler

2.2 개발 내용

2.2.1 Alarm Clock

Thread를 block상태로 만들어 두고, 일정 시간(보통 tick: 1/100초 단위)이 지난 뒤 다시 ready상태로 만들어준다. 단, 이때 busy waiting을 이용해서는 안 된다.

2.2.2 Priority Scheduling

Ready 상태인 thread들 중 priority가 가장 높은 thread가 우선적으로 수행되도록 schedule하고, 그리고 각 lock/semaphore등의 synchronization construct들의 waiting queue에서 우선적으로 wake up되도록한다.
단, 높은 priority의 thread가 낮은 priority의 thread를 lock때문에 기다릴 때에는 **낮은 priority의 thread가 lock을 release할 때까지** 높은 priority의 thread가 낮은 priority의 thread에게 **priority를 기부**함으로써 해당 lock과 관련된 모든 thread가 동작하지 않는 상황을 방지한다.

2.2.3 Advanced(4.4BSD) Scheduler

Pintos에서 thread의 priority는 PRI_MIN(0)부터 PRI_MAX(63)까지 총64개로 이루어져 있다. 이를 이용해서 64개의 reqdy queue를 갖는 multi-level feedback queue를 생각해볼 수 있다. 이를 이용, 4.4BSD-Scheduler와 비슷한 Scheduler를 구현한다.
Advanced Scheduler는 시스템의 Average response time을 줄일 수 있고, priority donation을 필요로 하지 않는다.
이러한 advanced scheduler의 priority 계산은 다음과 같다:

$$\texttt{priority} = \texttt{PRI\_MAX} - (\texttt{recent\_cpu}/4) - (\texttt{nice} \times 2)$$

여기서 recent_cpu는 각 thread가 최근에 사용한 cpu time의 추정값으로, 최근에 cpu time을 많이 소모한 thread일수록 우선순위가 낮아지도록 하여 starvation을 방지하고, nice는 해당 thread가 얼마나 nice한지, 즉 다른 thread에게 우선순위를 어느정도 양보할지를 나타낸다.
한편 recent_cpu의 값은 다음과 같이 추정된다(단, 이는 **1초**마다 각 thread에 대해 계산되는 식이며, running thread의 경우엔 추가적으로 1 timer tick마다 recent_cpu=recent_cpu+1이 수행된다):

$$\begin{aligned}&\texttt{recent\_cpu}(0) = 0,\\ &\texttt{recent\_cpu}(\mathrm{t}) = \mathrm{a}\times\texttt{recent\_cpu}(t-1) + \texttt{nice},\\ &\mathrm{a} = \texttt{load\_avg}/(\texttt{load\_avg}+1)\end{aligned}$$

이는 이전에 해당 thread가 사용한 cpu time일수록 영향을 적게 끼치고, 최근에 사용한 cpu time일수록 큰 영향을 주도록 하기 위해 일종의 Exponential Moving Average의 공식을 이용해서 계산된다.
한편 load_avg는 시스템 전역의 average load를 나타내는 값으로, 다음과 같이 추정된다(단, 이는 **1초**마다 계산되는 식이다):

$$\begin{aligned}&\texttt{load\_avg}(0) = 0,\\ &\texttt{load\_avg}(\mathrm{t}) = \mathrm{a}\times\texttt{load\_avg}(t-1) + (1-a) \times \texttt{number\_of\_ready\_threads},\\ &\mathrm{a} = 59/60\end{aligned}$$

3 추진 일정 및 개발 방법

3.1 추진 일정

- 11.09 - 11.09: Pintos 매뉴얼 정독 및 devices/timer.c, threads/thread.c/h, 그리고 threads/synch.c/h정독 후 priority donation 구상.
- 11.10 - 11.10: Alarm Clock 및 Priority Scheduling 구현.
- 11.11 - 11.11: priority-lifo와 같은 추가 test들의 분석과 -aging의 구현.
- 11.11 - 11.12: Advanced Scheduler 구현.

3.2 개발 방법

개발은 다음과 같은 원칙을 바탕으로 진행했다:

1. Pintos Manual의 내용을 숙지한다.
2. 이미 작성되어있는 Pintos 내부의 코드(thread.c 등)를 숙지한다.
3. 구현해야할 각 기능의 목표와 작동, interrupt disable/enable의 필요성을 숙지한다.
4. 구현시 한 함수를 작성하면 해당 함수의 작동을 완벽히 분석한 다음 다른 함수를 작성한다.
5. 각 기능 구현의 기한을 정하고, 이에 맞춰 작성할 수 있도록 시간을 잘 안배한다.

또한, 다음과 같이 도표를 그려서 각 함수의 연결과 그 흐름을 확인했다.

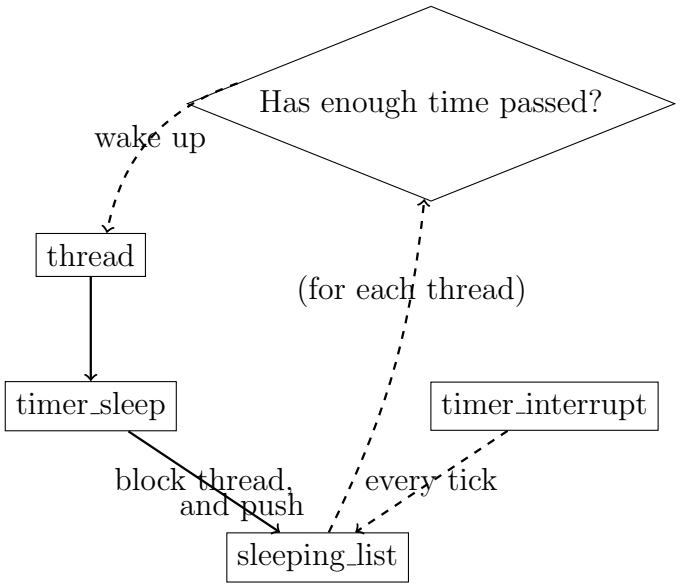

Figure 1: Alarm clock 구현 흐름도

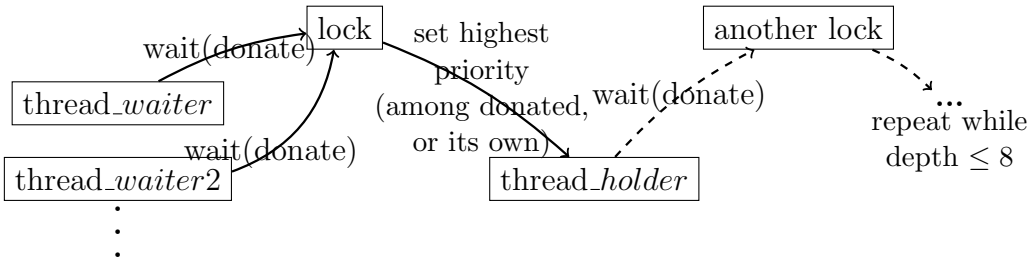

Figure 2: Priority Donation의 구현 흐름도

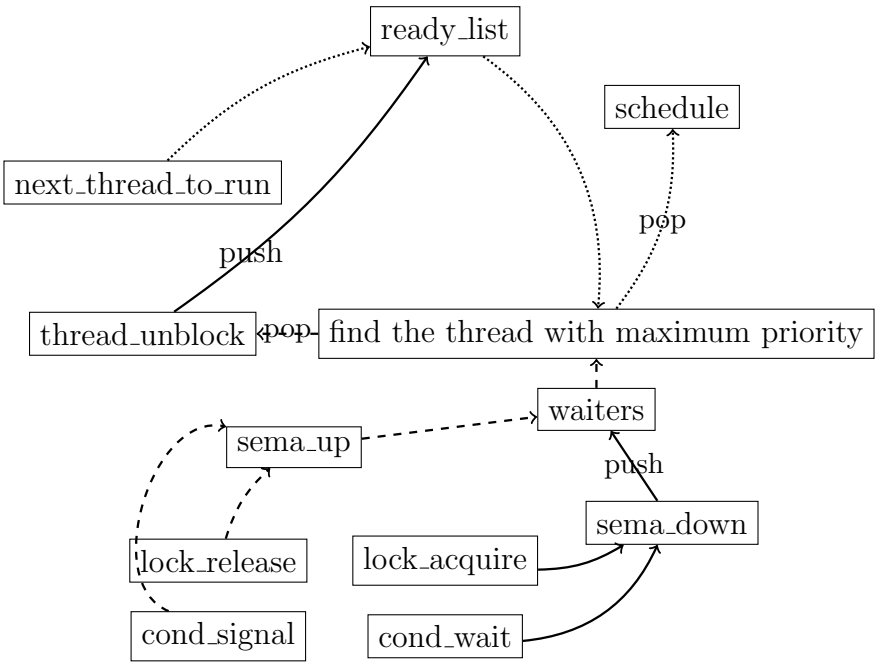

Figure 3: Priority Scheduling의 구현 흐름도

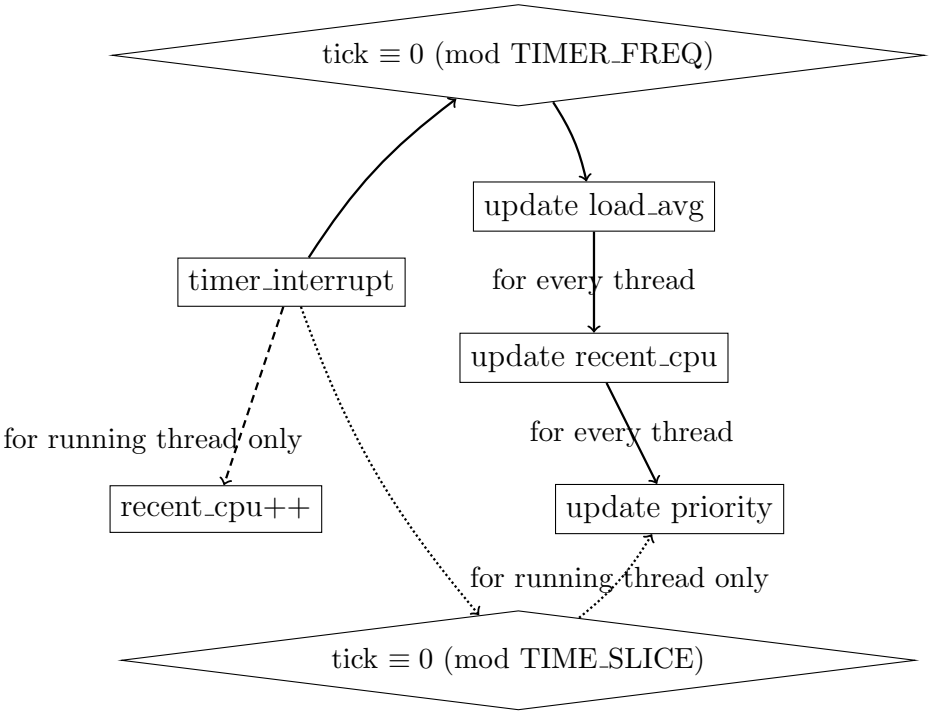

Figure 4: Advanced Scheduler의 구현 흐름도

3.3 연구원 역할 분담

혼자서 하는 프로젝트였으므로 역할을 분담하기보다는 위에서 적은 것과 같이 날짜를 기준으로 작업을 분배하였다.

4 연구 결과

결과적으로 모든 테스트에 대해 pass를 받을 수 있었다(*: 서강대학교 추가 테스트):

```
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-aging*
pass tests/threads/priority-change
pass tests/threads/priority-change-2*
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-lifo*
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 30 tests passed.
```

4.1 합성 내용

4.1.1 Alarm Clock

timer_sleep은 Figure 1과 같이 Thread를 block시켜두고, wakeup할 tick과 함께 sorted list(wakeup할 tick이 작은 순서대로 정렬된 리스트)에 넣어둔다. 그리고 매 timer interrupt마다 리스트의 첫 번째 원소부터 wakeup해야 할 thread가 있다면 unblock하고 리스트에서 제거한다. sorted list를 유지함으로써 매timer interrupt마다 이 작업을 수행하더라도 딱 필요한 만큼(wakeup할 thread의 개수만큼)만 iteration이 이루어지도록 부하를 최소화했다.

4.1.2 Priority Scheduling

Figure 3과 같이, ready queue에 thread가 추가될 때에는 무조건 순서대로 추가되도록(push_back) 했다. 이렇게 할 경우 ready queue에서 다음에 수행될 thread를 찾을 때 ready queue에서 가장 높은 priority를 갖는 thread를 찾아야 해서 비효율적으로 보일 수 있지만, priority aging과 같이 thread가 ready queue 안에 있는 상황에서 priority가 변경되는 경우에 효율적으로 대처할 수 있다. ready queue를 sorted list로 유지하는 것은 이렇게 priority가 변경될 경우 sort를 필요로 하므로 priority의 변경이 잦은 경우 굉장히 느려질 수 있다.
synchronization construct들에 대해서도 역시 Figure 3처럼 ready queue와 같은 방식을 사용했는데, 모든 synchronization construct는 내부적으로 semaphore를 이용하므로 sema_up에서는 가장 높은 priority를 가진 thread를 찾아서 unblock했고, sema_down에서는 무조건 순서대로 추가되도록(push_back)했다.
한편 Priority Donation의 경우에는 Figure 2와 같이 구현했는데, lock의 holder인 thread는 donated priority의 list를 가지고 있어서, holder 자신의 priority와 이 lock을 기다리는 thread들의 priority들 중 가장 큰 priority를 자신의 priority로 가진다. 그리고 priority가 lock을 release할 때에는 waiter중 가장 큰 priority를 갖는 thread, 즉 다음 holder가 될 thread에게 donated priority list를 그대로 넘겨줘서 donation을 유지한다. 여기서 가장 문제가 되는 부분은 Figure 2와 같이 여러 단계로 lock을 기다리는 때인데(lock1의 holder는 lock2를 기다리고, ...), 이럴 때엔 holder의 priority가 donation에 의해 변경될 경우, DFS를 이용해서 최대 8단계까지(max_depth=8) priority를 수정해준다.
마지막으로, priority-aging의 경우에는 ready queue에 있는 thread들의 priority를 일정 tick마다 증가시키도록 하여 구현했다.

4.1.3 Advanced(4.4BSD) Scheduler

64개의 ready queue를 이용하지 않고, priority만을 이용해서 reqdy queue에서 가장 높은 priority를 갖는 thread부터 수행되도록 작성했다. priority의 업데이트는 Figure 4와 같이 일정 tick마다(timer interrupt에서 수행) load_average와 각 thread의 recent_cpu, 그리고 priority를 업데이트함으로써 해결했다. 다만 구현상에 있어서 다음과 같은 식들을 이용하므로 계산 과정과 계산 결과에서 정수가 아닌 유리수가 등장하게 되는데, 이를 표현하기 위해 17,14-fixed point arithmetic을 구현하여 계산했다.

$$\texttt{priority} = \texttt{PRI\_MAX} - (\texttt{recent\_cpu}/4) - (\texttt{nice} \times 2)$$
$$\texttt{recent\_cpu}(t) = \texttt{load\_avg}/(\texttt{load\_avg} + 1) \times \texttt{recent\_cpu}(t-1) + \texttt{nice}$$
$$\texttt{load\_avg}(t) = (59/60) \times \texttt{load\_avg}(t-1) + (1/60) \times \texttt{number\_of\_ready\_threads}$$

4.2 제작 내용

4.2.1 Alarm Clock

timer_sleep은 Figure 1과 같이 Thread를 block시켜두고, wakeup할 tick과 함께 sorted list(wakeup할 tick이 작은 순서대로 정렬된 리스트)에 넣어둔다. 그리고 매 timer interrupt마다 리스트의 첫 번째 원소부터 wakeup해야 할 thread가 있다면 unblock하고 리스트에서 제거한다. sorted list를 유지함으로써 매timer interrupt마다 이 작업을 수행하더라도 딱 필요한 만큼(wakeup할 thread의 개수만큼)만 iteration이 이루어지도록 부하를 최소화했다.

```
/* For sleeping threads.. */
static struct sleeping {
  int64_t wakeup;
  struct semaphore lock;
  struct list_elem elem;
};
static struct list sleeping_list;
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  /* Create and initialize new sleeping entry for wakeup. */
  struct sleeping *sleepy = malloc(sizeof *sleepy);
  sleepy -> wakeup = start + ticks;
  sema_init(&sleepy -> lock, 0);
  list_insert_ordered(&sleeping_list, &sleepy -> elem, sleeping_less, NULL);

  /* Block this thread until the time.. */
  sema_down(&sleepy -> lock);
  free(sleepy);

  /* Interrupt must be on for concurrency. */
  ASSERT (intr_get_level () == INTR_ON);
}
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  /* ... */

  /* Wake up threads. */
  while (!list_empty(&sleeping_list)) {
    struct sleeping *s = list_entry(list_front(&sleeping_list), struct sleeping, elem);
    if (s -> wakeup <= ticks) {
     list_pop_front(&sleeping_list);
     sema_up(&s -> lock);
```

```
    } else break;
  }
}
```

4.2.2 Priority Scheduling

Figure 3과 같이, ready queue에 thread가 추가될 때에는 무조건 순서대로 추가되도록(push_back) 했다. 이렇게 할 경우 ready queue에서 다음에 수행될 thread를 찾을 때 ready queue에서 가장 높은 priority를 갖는 thread를 찾아야 해서 비효율적으로 보일 수 있지만, priority aging과 같이 thread가 ready queue 안에 있는 상황에서 priority가 변경되는 경우에 효율적으로 대처할 수 있다. ready queue를 sorted list로 유지하는 것은 이렇게 priority가 변경될 경우 sort를 필요로 하므로 priority의 변경이 잦은 경우 굉장히 느려질 수 있다.
synchronization construct들에 대해서도 역시 Figure 3처럼 ready queue와 같은 방식을 사용했는데, 모든 synchronization construct는 내부적으로 semaphore를 이용하므로 sema_up에서는 가장 높은 priority를 가진 thread를 찾아서 unblock했고, sema_down에서는 무조건 순서대로 추가되도록(push_back)했다.

```
bool
thread_priority_less(const struct list_elem *a, const struct list_elem *b, void *aux)
{
    struct thread *lhs = list_entry(a, struct thread, elem);
    struct thread *rhs = list_entry(b, struct thread, elem);

    return (lhs -> priority) > (rhs -> priority);
}

static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else {
      struct list_elem *e = list_min(&ready_list, thread_priority_less, NULL);
      struct thread *nxt = list_entry(e, struct thread, elem);
      list_remove(e);
      return nxt;
  }
}

void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;
  int pri_nxt = -1000;

  ASSERT (sema != NULL);
```

```
  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    struct list_elem *e = list_min(&sema->waiters, thread_priority_less, NULL);
    struct thread *t = list_entry (e, struct thread, elem);
    list_remove(e);
    pri_nxt = t -> priority;
    thread_unblock (t);
  }
  sema->value++;
  intr_set_level (old_level);
  if (!intr_context() && pri_nxt != -1000) thread_yield();
}
```

한편 Priority Donation의 경우에는 Figure 2와 같이 구현했는데, lock의 holder인 thread는 donated priority의 list를 가지고 있어서, holder 자신의 priority와 이 lock을 기다리는 thread들의 priority들 중 가장 큰 priority를 자신의 priority로 가진다. 그리고 priority가 lock을 release할 때에는 waiter중 가장 큰 priority를 갖는 thread, 즉 다음 holder가 될 thread에게 donated priority list를 그대로 넘겨줘서 donation을 유지한다. 여기서 가장 문제가 되는 부분은 Figure 2와 같이 여러 단계로 lock을 기다리는 때인데(lock1의 holder는 lock2를 기다리고, ...), 이럴 때엔 holder의 priority가 donation에 의해 변경될 경우, DFS를 이용해서 최대 8단계까지(max_depth=8) priority를 수정해준다.

```
#define MAX_DEPTH 8
static void update_donation_tree (int depth, struct lock *lock, struct thread *root);
static void
update_donation_tree (int depth, struct lock *lock, struct thread *root)
{
    if (depth > MAX_DEPTH || lock -> holder == NULL) return;

    struct donation *d;
    struct list_elem *e;
    for (e = list_begin(&lock -> holder -> donated);
    e != list_end(&lock -> holder -> donated);
         e = list_next(e)) {
        d = list_entry(e, struct donation, elem);
        if (d -> tid == root -> tid)
            d -> priority = root -> priority;
    }

    /* should recurse more.. */
    if (lock -> holder -> priority < root -> priority) {
        lock -> holder -> priority = root -> priority;
        list_sort(&lock -> holder -> donated, donation_less, NULL);
        if (lock -> holder -> blocked_for == NULL) return;
        update_donation_tree(depth + 1, lock -> holder -> blocked_for, lock -> holder);
    }
```

```
}
#undef MAX_DEPTH

void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);

  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));


  if (thread_mlfqs != true) {
      /* donate my priority to holder. */
      if (lock -> holder != NULL) {
          struct donation *giving = malloc(sizeof *giving);
          giving -> priority = thread_current() -> priority;
          giving -> tid = thread_current() -> tid;
          giving -> lock = lock;
          list_insert_ordered(&lock->holder->donated, &giving->elem, donation_less, NULL);
          thread_current() -> blocked_for = lock;
          if (thread_current() && thread_current() -> blocked_for)
              update_donation_tree(1, thread_current() -> blocked_for, thread_current());

          if (list_empty(&lock -> holder -> donated) ||
              giving -> priority > lock -> holder -> priority) {
              lock -> holder -> priority = giving -> priority;
          }
      }
  }

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
  if (thread_mlfqs != true) {
      if (!intr_context() && thread_current() -> blocked_for == lock) {
          thread_current() -> blocked_for = NULL;
      }
  }
}
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  int pri_nxt = -1000;
```

```
if (thread_mlfqs != true) {
    if (!intr_context()) {
        int tid_nxt = 0, found = 1;
        struct thread *p;
        struct list_elem *e;
        struct donation *d;
        while (!list_empty(&thread_current() -> donated) && found) {
            for (e = list_begin(&thread_current() -> donated);
                    e != list_end(&thread_current() -> donated); e = list_next(e)) {
                found = 0;
                d = list_entry(e, struct donation, elem);
                p = list_entry(list_min(&lock->semaphore.waiters,
                           thread_priority_less, NULL), struct thread, elem);
                tid_nxt = p -> tid;
                pri_nxt = p -> priority;
                if (d -> lock == lock) {
                    if (p -> tid == d -> tid) {
                        list_remove(e);
                        free(d);
                    } else {
                        list_insert_ordered(&p->donated, list_remove(e),
                                            donation_less, NULL);
                        p -> priority = (p -> priority < d -> priority ?
                                         d -> priority : p -> priority);
                    }
                    found = 1;
                    if (list_empty(&thread_current() -> donated)) {
                        thread_current() -> priority = thread_current() -> own_priority;
                    } else {
                        d = list_entry(list_begin(&thread_current() -> donated),
                  struct donation, elem);
                        if (d -> priority > thread_current() -> own_priority)
                            thread_current() -> priority = (d -> priority);
                        else thread_current() -> priority =
                             (thread_current() -> own_priority);
                    }
                    break;
                }
            }
        }
    }
}

lock->holder = NULL;
sema_up (&lock->semaphore);
if (thread_mlfqs != true) {
```

```
      if (!intr_context() && pri_nxt > thread_get_priority()) thread_yield();
  }
}
```

마지막으로, priority-aging의 경우에는 ready queue에 있는 thread들의 priority를 일정 tick마다 증가시키도록 하여 구현했다.

```
/* Aging threads to prevent starvation. */
#define AGE_CYCLE 100
static void
thread_aging (void)
{
    struct list_elem *e;
    struct thread *t;
    for (e = list_begin(&ready_list); e != list_end(&ready_list); e = list_next(e)) {
        t = list_entry(e, struct thread, elem);
        if (++(t -> age) % AGE_CYCLE == 0) {
            ++(t -> own_priority);
            if (t -> priority < t -> own_priority)
                t -> priority = t -> own_priority;
        }
    }
}
#undef AGE_CYCLE
```

4.2.3 Advanced(4.4BSD) Scheduler

64개의 ready queue를 이용하지 않고, priority만을 이용해서 reqdy queue에서 가장 높은 priority를 갖는 thread부터 수행되도록 작성했다. priority의 업데이트는 Figure 4와 같이 일정 tick마다(timer interrupt에서 수행) load_average와 각 thread의 recent_cpu, 그리고 priority를 업데이트함으로써 해결했다. 다만 구현상에 있어서 다음과 같은 식들을 이용하므로 계산 과정과 계산 결과에서 정수가 아닌 유리수가 등장하게 되는데, 이를 표현하기 위해 17,14-fixed point arithmetic을 구현하여 계산했다.

$$\texttt{priority} = \texttt{PRI\_MAX} - (\texttt{recent\_cpu}/4) - (\texttt{nice} \times 2)$$
$$\texttt{recent\_cpu}(t) = \texttt{load\_avg}/(\texttt{load\_avg} + 1) \times \texttt{recent\_cpu}(t-1) + \texttt{nice}$$
$$\texttt{load\_avg}(t) = (59/60) \times \texttt{load\_avg}(t-1) + (1/60) \times \texttt{number\_of\_ready\_threads}$$

```
#define FIXED_BASE 14
#define FIXED_F (1LL << FIXED_BASE)
#define to_int_strict(x) ((x) / FIXED_F)
#define to_int(x) (((x) >= 0 ? ((x) + (FIXED_F/2)) : ((x) - (FIXED_F/2))) / FIXED_F)
#define to_fixed(n) ((n) * FIXED_F)
#define add_fixed(x, y) ((x) + (y))
#define sub_fixed(x, y) ((x) - (y))
#define mul_fixed(x, y) ((((int64_t)(x)) * (y)) / FIXED_F)
#define div_fixed(x, y) ((((int64_t)(x)) * FIXED_F) / (y))
```

```
void
thread_mlfqs_update(bool update_priority_only)
{
    int old_level = intr_get_level();
    intr_disable();

    int64_t coeff = to_fixed(59) / 60,
            ready = list_size(&ready_list) + (thread_current() != idle_thread),
            load_avg = mul_fixed(coeff, load_average) + to_fixed(1) / 60 * ready;

    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);
        if (t == idle_thread) continue;
        int64_t coeff = div_fixed(load_average * 2, load_average * 2 + to_fixed(1)),
                recent_cpu = mul_fixed(coeff, t -> recent_cpu) + to_fixed(t -> nice);
        t -> recent_cpu = recent_cpu;
        t -> priority = PRI_MAX - to_int_strict(t -> recent_cpu / 4) - (t -> nice) * 2;
        if (t -> priority < PRI_MIN) t -> priority = PRI_MIN;
        if (t -> priority > PRI_MAX) t -> priority = PRI_MAX;
    }

    load_average = load_avg;
    intr_set_level(old_level);
}
#define TIME_SLICE 4            /* # of timer ticks to give each thread. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  if (thread_mlfqs) {
      thread_current() -> recent_cpu += to_fixed(1);
      if (ticks % TIMER_FREQ == 0) // every second.
          thread_mlfqs_update(false);
      if (ticks % TIME_SLICE == 0) // update priority of current thread.
          thread_set_nice(thread_current() -> nice);
  }
  /* ... */
}
#undef TIME_SLICE
void
thread_set_nice (int nice)
{
    thread_current() -> nice = nice;
    thread_current() -> priority = to_int_strict(to_fixed(PRI_MAX) - (thread_current() -> r
```

```
    if (thread_current() -> priority < PRI_MIN) thread_current() -> priority = PRI_MIN;
    if (thread_current() -> priority > PRI_MAX) thread_current() -> priority = PRI_MAX;
}
int
thread_get_nice (void) {
    return thread_current() -> nice;
}
int
thread_get_load_avg (void) {
    return to_int(load_average * 100);
}
int
thread_get_recent_cpu (void) {
    return to_int(thread_current() -> recent_cpu * 100);
}
```

4.3 시험 및 평가내용

평가는 기본적으로 make check의 pass / fail로 평가했으며, 특정 문제를 해결하기 힘든 경우에는 특정 프로그램을 특정 option와 함께 시험해보도록 다음과 같은 커맨드를 이용했다.
`$ make clean && make && cd build && pintos -v -k -T 480 --bochs -- -q -mlfqs run mlfqs-recent-1` pass / fail에 대한 결과는 다음과 같이 모든 테스트에 대해 pass이다. 이 평가 내용들은 각각 개발 목표의 세 부분, 그리고 생산성과 내구성으로 나눌 수 있다.

```
1a. Alarm Clock(생산성 - 프로그램이 제대로 기능하는가):
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority

1b. Alarm Clock(내구성 - 프로그램이 예외상황에 잘 대처할 수 있는가):
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative

2a. Priority Scheduling(생산성 - 프로그램이 제대로 기능하는가):
pass tests/threads/priority-aging*
pass tests/threads/priority-change
pass tests/threads/priority-change-2*
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-lifo*
pass tests/threads/priority-preempt
```

```
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain

3a. Advanced Scheduler(생산성 - 프로그램이 제대로 기능하는가):
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
```

4.3.1 보건 및 안정

fixed point arithmetic에서 underflow나 overflow가 일어나지 않도록 int 대신 int64_t를 이용했고, 이로 인해 느려질 수 있는 부분들은 최대한 최적화를 해서 수행 속도에는 영향을 주지 않게 하기 위해 노력했다. 또한, priority scheduling에서 sorted list를 유지하지 않음으로써 어떤 경우에 priority가 수정되더라도 항상 가장 높은 priority를 가진 thread부터 수행될 수 있도록, 즉 priority의 update를 놓치는 경우가 없도록 작성했다.

4.3.2 priority-lifo test 분석

```
$ diff -u priority-fifo.c priority-lifo.c

--- priority-fifo.c 2014-11-12 11:31:21.023594392 +0900
+++ priority-lifo.c 2014-11-12 11:31:21.023594392 +0900
@@ -29,7 +29,7 @@
 static thread_func simple_thread_func;

 void
-test_priority_fifo (void)
+test_priority_lifo (void)
 {
   struct simple_thread_data data[THREAD_CNT];
   struct lock lock;
@@ -50,7 +50,7 @@
   ASSERT (output != NULL);
   lock_init (&lock);

-  thread_set_priority (PRI_DEFAULT + 2);
+  thread_set_priority (PRI_DEFAULT + THREAD_CNT + 1);
   for (i = 0; i < THREAD_CNT; i++)
     {
       char name[16];
```

```
@@ -60,7 +60,7 @@
       d->iterations = 0;
       d->lock = &lock;
       d->op = &op;
-      thread_create (name, PRI_DEFAULT + 1, simple_thread_func, d);
+      thread_create (name, PRI_DEFAULT + 1 + i, simple_thread_func, d);
     }

   thread_set_priority (PRI_DEFAULT);
@@ -75,7 +75,7 @@
       ASSERT (*output >= 0 && *output < THREAD_CNT);
       d = data + *output;
       if (cnt % THREAD_CNT == 0)
-        printf ("(priority-fifo) iteration:");
+        printf ("(priority-lifo) iteration:");
       printf (" %d", d->id);
       if (++cnt % THREAD_CNT == 0)
         printf ("\n");
```

priority-lifo.c는 위의 diff 결과에서 알 수 있듯이 priority-fifo.c를 아주 조금 수정한 테스트 파일이다.
차이점을 보면 딱 네 줄이 다른데, 그중 두 줄은 테스트 이름이 다르고, 나머지 두 줄은 thread를 만들 때, priority-fifo의 경우 모든 thread의 priority를 동일하게 만들지만, priority-lifo의 경우 나중에 만들어진 thread일수록 높은 priority를 갖도록 작성되어있다. 따라서 이 테스트의 정상 동작은 다음과 같이 나중에 만들어진 thread부터 순서대로, 각자 자신이 종료할 때까지 계속해서 수행되는 것임이 자명하다(다음 테스트에서 출력되는 수가 의미하는 것은 만들어진 순서이다 - 즉, 첫 번째 만들어진 thread는 0을 출력하고, 두 번째 만들어진 thread는 1을 출력하고, ... 이런 식이다):

```
...
Executing 'priority-lifo':
(priority-lifo) begin
(priority-lifo) 16 threads will iterate 16 times in the same order each time.
(priority-lifo) If the order varies then there is a bug.
(priority-lifo) iteration: 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
(priority-lifo) iteration: 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
(priority-lifo) iteration: 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
(priority-lifo) iteration: 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
(priority-lifo) iteration: 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
(priority-lifo) iteration: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
(priority-lifo) iteration: 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
(priority-lifo) iteration: 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
(priority-lifo) iteration: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
(priority-lifo) iteration: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
(priority-lifo) iteration: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
(priority-lifo) iteration: 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
(priority-lifo) iteration: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
(priority-lifo) iteration: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
(priority-lifo) iteration: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(priority-lifo) iteration: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(priority-lifo) end
Execution of 'priority-lifo' complete.
...
```

5 기타

5.1 Priority Inversion

5.1.1 의미

Priority Inversion이란, Priority scheduling을 사용하는 컴퓨터에 (priority순서대로) L, M, H라는 thread가 있을 때, L이 Mutex를 가지고 있고, H가 해당 Mutex를 기다리면 L은 M보다 priority가 낮으므로 영원히 실행되지 않고 M만 지속적으로 수행되거나, M의 수행으로 인해 H의 수행이 지연되는 현상을 가리킨다. 즉, 우리가 Priority Donation을 이용하는 이유가 바로 이러한 Priority Inversion을 방지하기 위함이다.
Inversion이란 이름은 이유는 중간 priority를 가진 thread(M)가 높은 priority를 가진 thread(H)보다 우선적으로 수행되는 상황 때문에 '실제 유효한 priority는 inverted되었다(역전되었다)'라는 것을 의미한다.

5.1.2 해결책

1. 모든 interrupt를 disable한다. 이는 system의 concurrency를 없애는 행위이므로 실용성이 없다.

2. Priority Celling: Mutex를 가지고 있는 동안 해당 thread의 priority를 매우 높게(HH) 설정한다. 이는 보통 그럭저럭 괜찮은 결과를 보여주는데, 이 Mutex를 기다리는 thread의 priority가 HH보다 높지 않아야 한다.

3. Priority Inheritance(Priority Donation): 현재 Pintos에서 사용한 방식으로, H가 L이 가지고있는 Mutex를 기다리는 동안, L이 H의 priority를 갖고 수행되는 것이다.

4. Mutex를 가지고 있는 thread가 critical section을 벗어날 때까지 랜덤하게 priority를 높인다. 이 방식은 Microsoft Windows가 사용중이다.

5. 이외의 방식으로는 애초에 L이 H를 block하지 않도록 하는 방법이 있다(물론 이게 어려우니 문제가 된다).

5.2 연구 조원 기여도

고창영 : 설계 및 코드 작성, 그리고 보고서(100%).

5.3 느낀점

Scheduling의 원리와 이론들을 직접 구현하면서 실습할 수 있는 기화가 되어 유익했다.
특히 priority scheduling의 구현에서 최적화 방안 등 많은 것을 배울 수 있었고 priority donation과 같은 새로운 개념도 생각해볼 수 있어서 좋은 기회였다.
하지만 test가 획일적이고 적은 것과, priority donation을 구현할 때 printf가 되지 않고 gdb로도 값을 볼 수 없어서(optimized-out) 오류를 찾기 힘들었던 점이 아쉽다.