Pintos Project 3: Threads

I. 개발 목표

Busy waiting을 하던 기존 코드를 수정하여 busy waiting을 하지 않도록 Alarm clock을 구현하고, scheduling algorithm을 Round-Robin에서 priority scheduling으로 수정하며, BSD scheduler를 구현한다.

II. 개발 범위 및 내용

A. 개발 범위

1) Alarm Clock

devices/timer.c의 timer_sleep() 함수는 busy waiting을 하도록 작동한다. thread를 sleep시키고 wake-up시키는 것으로 수정하면 busy waiting을 피할 수 있다.

2) Priority Scheduling

Priority scheduling을 구현해서 더 높은 priority를 가지는 thread가 먼저 수행되도록 한다.

3) Advanced Scheduler

Multi-Level Feedback Queue (MLFQ)를 구현하여, thread마다 서로 다른 scheduling need의 균형을 맞춘다. General purpose scheduler인 4.4 BSD scheduler와 유사하게 구현해 가장 priority가 높은 ready queue에서 thread가 먼저 수행되도록 한다.

B. 개발 내용

1) Blocked 상태의 스레드를 어떻게 깨울 수 있는가

   sleep queue에 BLOCKED 상태의 thread가 있는지 확인하고, 깨어날 thread가 있다면 thread_unblock() 함수를 호출해 깨워준다. thread_unblock() 함수는 interrupt를 disable하고, 인자로 전달받은 thread를 ready list에 삽입한 뒤, ready state로 바꾸고 interrupt를 set한다.

2) Priority scheduling: Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우

   현재 running thread보다 priority가 높은 thread가 ready list에 추가되었다면, 현재 thread는 즉시 그 thread에게 CPU를 yield해야 한다.

3) Advanced Scheduler에서 priority 계산에 필요한 각 요소

Priority는 thread가 생성될 때 31로 초기화되며, 다음과 같이 계산하여 시간이 흐름에 따라 업데이트된다.

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$$

여기서 recent_cpu는 thread가 최근에 사용한 CPU time을 의미하며, nice는 -20~20 사이의 값을 가지는 정수이다. 해당 thread가 다른 thread에게 얼마나 친절(nice)할지를 나타내주는 값으로, 양수일 경우 priority를 낮추고 음수일 경우 priority를 높이게 된다. recent_cpu는 실수 값을 가지며, 다음과 같이 계산된다.

$$recent\_cpu = (2 * load\_avg) / (2 * load\_avg + 1) * recent\_cpu + nice$$

recent_cpu 계산에 필요한 load_avg는 ready state에 있는 thread의 평균 개수를 의미하며, 0으로 초기화되었다가 다음의 식으로 계산되며 업데이트된다. ready_threads는 running state, ready to run state에 있는 thread의 수를 나타낸다.

$$load\_avg = (59/60) * load\_avg + (1/60) * ready\_threads$$

III. 추진 일정 및 개발 방법

A. 추진 일정

11/15-11/21: 프로젝트 명세서와 pintos manual을 읽으며 요구사항을 파악한다.

11/22-11/25: Alarm clock을 구현한다.

11/26-11/29: Priority scheduling을 구현한다.

11/30-12/03: Advanced scheduler를 구현한다.

12/04: 보고서 작성

B. 개발 방법

1) Alarm Clock

src/devices/timer.c

: timer_sleep() 함수에서 busy waiting을 하는 코드를 삭제하고, thread를 sleep시키는 함수를 호출한다. 그리고 timer_interrupt() 함수에서 thread를 깨우는 함수를 호출한다.

src/threads/thread.*

: thread 구조체에 thread가 깨어나야 할 tick을 저장할 변수 wakeup_tick을 추가한다. 그리고 sleep 상태의 thread들을 관리하기 위한 리스트를 전역 변수로 선언하고, thread_init() 함수에서 이 리스트를 초기화해준다. timer_sleep() 함수에서 호출할 thread_sleep() 함수를 추가하고, interrupt를 disable한 상태에서 wakeup_tick을 set해주고, sleep 리스트에 추가해준다. 그리고 timer_interrupt() 함수에서 호출할 thread_awake() 함수를 추가하고, sleep 리스트를 순회하며 깨워주어야 할 thread를 깨워준다.

2) Priority Scheduling

src/threads/thread.*

: thread가 생성되고 현재 thread보다 priority가 더 높다면 yield하도록 thread_create() 함수에 코드를 추가한다. 그리고 thread_unblock() 함수, thread_yield() 함수는 priority 순으로 ready list에 thread가 삽입되도록 수정하고, 이 과정에 필요한 priority_cmp() 함수를 추가해 priority 비교에 사용한다. thread_set_priority() 함수에서 thread의 priority를 변경 후 더 이상 가장 높은 priority가 아닌 경우에 yield할 수 있도록 코드를 작성한다. 또한 waiters 중 priority가 가장 높은 thread를 먼저 깨울 수 있도록 sema_up() 함수를 수정한다. Starvation이 발생할 수 있으므로 thread_prior_aging 변수를 추가한다.

src/threads/init.c

: -aging 옵션을 받으면 thread_prior_aging을 true로 set해준다.

3) Advanced Scheduler

src/threads/fixed-point.h

: recent_cpu, load_avg 값의 계산에 필요한 fixed point 연산 함수를 추가한다.

src/threads/thread.*

: fixed-point.h를 포함시키고, thread 구조체에 recent_cpu와 nice 변수를 추가한다. thread_init() 함수에서 load_avg, nice, recent_cpu를 초기화하고, init_thread() 함수에서 recent_cpu, nice를 running thread의 값으로 set해준다. update_priority, update_load_avg(), update_recent_cpu() 함수를 작성해 각각의 계산식에 따라 priority, load_avg, recent_cpu를 계산하여 업데이트해주는 코드를 작성한다. 그리고 recent_cpu 값을 늘려주는 increment_recent_cpu() 함수를 추가한다. thread_set_priority() 함수에 thread_mlfqs가 true일 경우 priority를 변경할 수 없도록 조건을 코드를 추가하고, thread_set_nice() 함수에서 nice 값을 변경하고 nice 값이 변경되었으므로 priority를 재계산하여 그에 맞게 scheduling해준다. thread_get_nice(), thread_get_load_avg(), thread_get_recent_cpu() 함수 내부에 코드를 작성해 현재 thread의 nice, 현재 load_avg 값, 현재 thread의 recent_cpu 값을 요구사항에 맞게 반환하도록 한다.

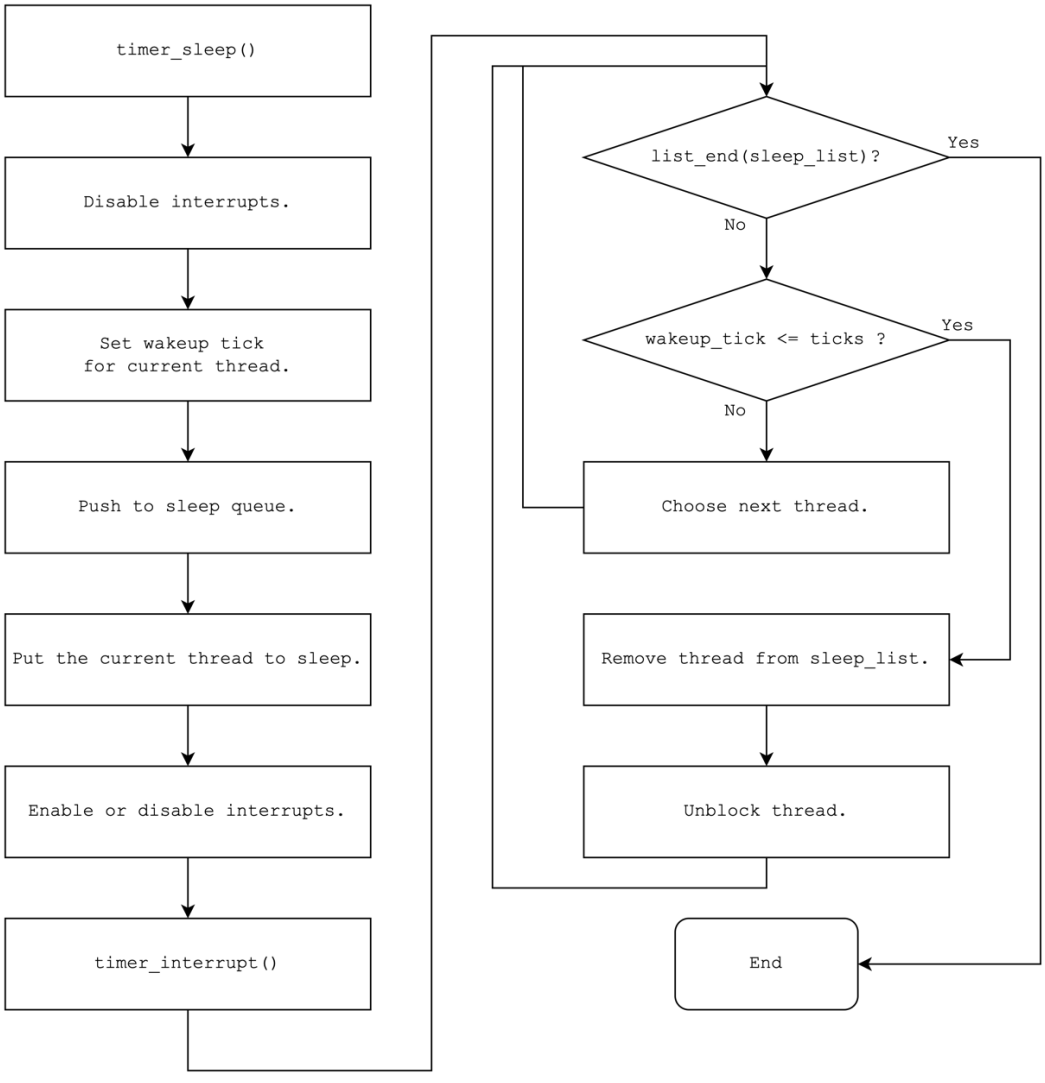
src/devices/timer.c

: timer_interrupt() 함수에 코드를 추가해, thread_prior_aging이 true이거나 thread_mlfqs가 true일 경우, increment_recent_cpu()를 호출해 recent_cpu를 증가시키고, 시간이 지남에 따라 조건을 확인하여 load_avg, recent_cpu, priority를 update하도록 update_load_avg(), update_recent_cpu(), update_priority()를 호출한다.

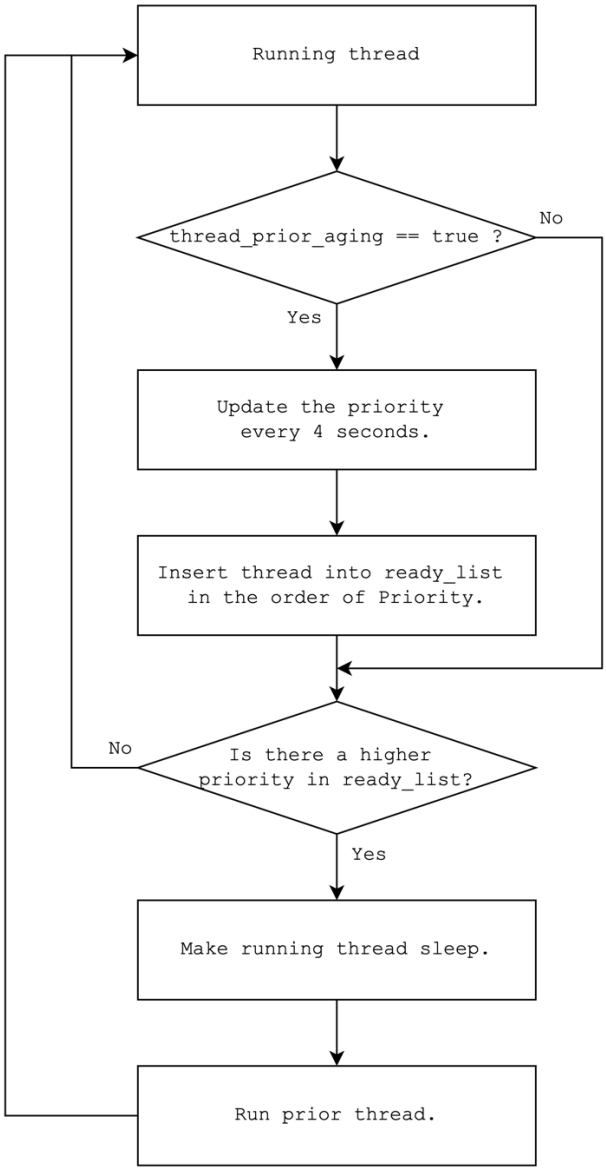
IV. 연구 결과

A. Flow Chart

1) Alarm Clock



2) Priority Scheduling



B. 제작 내용

1) Alarm Clock

먼저 wakeup_tick 변수를 thread 구조체에 추가하여, 깨어나야 할 tick을 저장할 수 있도록 했다.

src/threads/thread.h

```
struct thread
  {
    . . .
      /* ====== Project3 ====== */
    int64_t wakeup_tick;
    int recent_cpu;
    int nice;
      /* ====================== */
    . . .
};
```

그리고 THREAD_BLOCKED 상태의 thread들을 관리할 수 있도록 sleep_list 리스트를 전역 변수로 추가하고, thread_init() 함수에서 초기화해주었다.

src/threads/thread.c

```
/* ====== Project3 ====== */
static struct list sleep_list;
/* ====================== */
```

```
void
thread_init (void)
{
  . . .
  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);
  list_init (&sleep_list); /* Project 3 */
  . . .
}
```

timer_sleep() 함수에서 busy waiting을 하던 코드를 지우고, thread_sleep() 함수를 호출하도록 수정했다.

src/devices/timer.c

```
void
timer_sleep (int64_t ticks)
{
```

```
  /* ====== Project3 ====== */
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  thread_sleep (start + ticks);
  /* ====================== */
}
```

추가된 thread_sleep() 함수에서는 실행 중인 thread를 sleep하도록 했다. 이 과정 중에는 interrupt를 받아들이지 않도록 intr_disable() 함수를 호출했다. 그리고 thread가 깨어날 wakeup_tick을 저장해주고, sleep_list에 삽입해주고, thread를 sleep시켰다. intr_disable(), intr_set_level() 함수는 src/threads/interrupt.*를 참고했으며 thread_block() 함수는 src/threads/thread.*를 참고했다.

src/threads/thread.c

```
void
thread_sleep (int64_t ticks)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  /* Disable interrupts. */
  old_level = intr_disable ();

  /* Set wakeup tick for current thread. */
  cur->wakeup_tick = ticks;

  /* Push to sleep queue. */
  list_push_back (&sleep_list, &cur->elem);
  /* Put the current thread to sleep. */
  thread_block ();

  /* Enable or disable interrupts as specified by old_level. */
  intr_set_level (old_level);
}
```

timer_interrupt() 함수에서는 thread_awake() 함수를 호출했다.

src/devices/timer.c

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  /* ====== Project3 ====== */
  /* Alarm clock */
  thread_awake (ticks);
  . . .
  thread_tick ();
}
```

thread_awake() 함수를 추가 구현해 sleep_list에서 thread를 찾아 깨웠다. sleep_list를 순회하며 wakeup_tick 값이 ticks보다 작거나 같은 것은 충분히 기다렸다는 의미이므로 깨워주는 코드를 작성했다.

src/threads/thread.c

```
void
thread_awake (int64_t ticks)
{
  struct list_elem *e = list_begin (&sleep_list);
  struct thread *t;

  while (e != list_end (&sleep_list)) {
    t = list_entry (e, struct thread, elem);
    if (t->wakeup_tick <= ticks) {
      e = list_remove (e);
      thread_unblock (t); /* Awake */
    }
    else {
      e = list_next (e);
    }
  }
}
```

2) Priority Scheduling

thread_create() 함수를 수정해 현재 thread의 priority와 비교해서 생성된 thread의 priority가 더 높다면 즉시 yield하도록 thread_yield()를 호출했다.

src/threads/thread.c

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  . . .
  /* Add to run queue. */
  thread_unblock (t);

  /* ====== Project 3 ====== */
  if (priority > thread_get_priority ())
    thread_yield();
  /* ======================= */

  return tid;
}
```

unblock되는 thread가 priority 순으로 ready_list에 thread가 삽입되도록 thread_unblock() 함수를 수정했다.

src/threads/thread.c

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_insert_ordered (&ready_list, &t->elem, priority_cmp, NULL); /* Project 3 */
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

thread_yield() 함수도 마찬가지로, 현재 thread가 ready_list에 삽입될 때 priority 순으로 삽입되도록 수정했다.

src/threads/thread.c

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

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_insert_ordered (&ready_list, &cur->elem, priority_cmp, NULL); /* Project 3 */
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

thread_unblock(), thread_yield() 내부에서 list_insert_ordered() 함수의 인자로 사용된 priority_cmp() 함수는 다음과 같이 구현했다. 첫 번째 인자와 두 번째 인자의 priority를 비교해서 전자가 더 높으면 1, 후자가 더 높으면 0을 반환한다.

src/threads/thread.c

```
bool
priority_cmp (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  int a_priority = list_entry (a, struct thread, elem)->priority;
  int b_priority = list_entry (b, struct thread, elem)->priority;

  return a_priority > b_priority;
}
```

thread_set_priority() 함수에서는 thread의 priority가 변경되었을 때, priority 순으로 선점될 수 있도록 더 이상 가장 높은 priority를 가지지 않는다면 양보할 수 있도록 thread_yield()를 호출하는 코드를 추가 수정했다.

src/threads/thread.c

```
void
thread_set_priority (int new_priority)
{
```

```
  /* ====== Project 3 ====== */
  int cur_priority = thread_get_priority ();

  if (thread_mlfqs) return;

  /* Sets the current thread's priority to NEW_PRIORITY. */
  thread_current ()->priority = new_priority;

  /* If the current thread no longer has the highest priority, yields. */
  if (new_priority < cur_priority)
    thread_yield();
  /* ====================== */
}
```

priority가 높은 thread를 먼저 깨워주어야 하므로 sema_up() 함수를 수정해주었다. if문 내부를 수정해, priority가 가장 높은 element를 리스트에서 제거하고, thread를 깨워준다.

src/threads/synch.c

```
void
sema_up (struct semaphore *sema)
{
  /* ====== Project 3 ====== */
  enum intr_level old_level;
  struct thread *t, *max_thread;
  struct list_elem *e, *max_elem;

  ASSERT (sema != NULL);

  old_level = intr_disable ();

  if (!list_empty (&sema->waiters)) {
    e = max_elem = list_begin (&sema->waiters);
    max_thread = list_entry (e, struct thread, elem);

    for (e = list_next(e); e != list_end (&sema->waiters); e = list_next (e)) {
      t = list_entry (e, struct thread, elem);
      if (t->priority > max_thread->priority) {
        max_thread = t;
        max_elem = e;
      }
    }

    list_remove(max_elem);
```

```
    thread_unblock(max_thread);
  }

  sema->value++;
  intr_set_level (old_level);

  thread_yield();
  /* ======================== */
}
```

- Aging

아래와 같이 thread.h, thread.c 코드에 aging 관련 변수 thread_prior_aging을 추가하고, init.c 코드에서 -aging 옵션을 받으면 이 변수를 true로 set해주었다. thread_prior_aging을 활용한 코드는 3) Advanced Scheduler 항목에서 후술했다.

src/threads/thread.h

```
. . .
#include "threads/synch.h"
. . .

/* Porject 3 */
#ifndef USERPROG
extern bool thread_prior_aging;
#endif
. . .
```

src/threads/thread.c

```
/* Project 3 */
#ifndef USERPROG
bool thread_prior_aging;
#endif
```

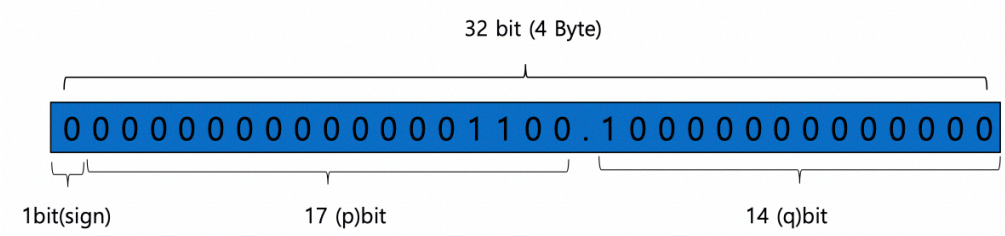
src/threads/init.c

```
static char **
parse_options (char **argv)
{
```

```
    . . .
#ifndef USERPROG
     /* Project 3 */
     else if (!strcmp (name, "-aging"))
       thread_prior_aging = true;
#endif
     . . .
}
```

3) Advanced Scheduler

Pintos는 부동 소수점 연산을 지원하지 않으므로, 실수 값인 recent_cpu와 load_avg의 계산에 fixed point 연산을 사용하기 위해 fixed-point.h 코드를 생성하고 thread.c에서 포함시켜주었다. 14 bit의 소수 부분을 위해 FRACTION 변수를 선언했다.



src/threads/fixed-point.h

```
#define FRACTION (1 << 14)

int int_to_float (int);
int float_to_int (int);
int float_add_float (int, int);
int float_sub_float (int, int);
int float_mul_float (int, int);
int float_div_float (int, int);
int float_add_int (int, int);
int int_sub_float (int, int);
int float_mul_int (int, int);
int float_div_int (int, int);

/* int -> float */
int
int_to_float (int i)
{
    return i * FRACTION;
}
```

```c
/* float -> int */
int
float_to_int (int f)
{
   if (f >= 0)
   {
      return (f + FRACTION / 2) / FRACTION;
   }
   else
   {
      return (f - FRACTION / 2) / FRACTION;
   }
}

/* float + float */
int
float_add_float (int f1, int f2)
{
  return f1 + f2;
}

/* float - float */
int
float_sub_float (int f1, int f2)
{
  return f1 - f2;
}

/* float * float */
int
float_mul_float (int f1, int f2)
{
  int64_t temp = f1;
  temp = temp * f2 / FRACTION;
  return (int) temp;
}

/* float / float */
int
float_div_float (int f1, int f2)
{
  int64_t temp = f1;
  temp = temp * FRACTION / f2;
  return (int) temp;
}
```

```
/* float + int */
int
float_add_int (int f, int i)
{
  return f + i * FRACTION;
}

/* int - float */
int
int_sub_float (int i, int f)
{
  return i * FRACTION - f;
}

/* float * int */
int
float_mul_int (int f, int i)
{
    return f * i;
}

/* float / int */
int
float_div_int (int f, int i)
{
    return f / i;
}
```

src/threads/thread.c

```
. . .
#include "threads/fixed-point.h"
. . .
```

thread 구조체에 recent_cpu, nice 변수를 추가해주고, load_avg를 전역 변수로 선언해주었다. 그리고 thread_init() 함수에서 이 변수들을 0으로 초기화해주었다.

src/threads/thread.h

```
struct thread
  {
    . . .
```

```
    /* ====== Project3 ====== */
    int64_t wakeup_tick;
    int recent_cpu;
    int nice;
    /* ====================== */
    . . .
};
```

src/threads/thread.c

```
static int load_avg;
```

```
void
thread_init (void)
{
  load_avg = 0; /* Project 3 */
  . . .
  initial_thread->nice = 0; /* Project 3 */
  initial_thread->recent_cpu = 0; /* Project 3 */
}
```

init_thread() 함수에서는 running thread의 recent_cpu, nice로 set해주었다.

src/threads/thread.c

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  . . .
  /* Project 3 */
  t->recent_cpu = running_thread ()->recent_cpu;
  t->nice = running_thread ()->nice;
  . . .
}
```

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$$

$$recent\_cpu = (2 * load\_avg) / (2 * load\_avg + 1) * recent\_cpu + nice$$

load_avg = (59/60) * load_avg + (1/60) * ready_threads

위 식을 이용해 priority, load_avg, recent_cpu를 계산하는 update_priority(), update_load_avg(), update_recent_cpu() 함수를 작성했다.

update_priority() 함수에서는 모든 thread에 대해 priority 값을 계산해주고, 범위를 벗어나는 경우 최댓값 혹은 최솟값으로 설정해주었다. 그리고 현재 thread의 priority가 가장 높지 않다면 reschedule해주기 위해 intr_yield_on_return()을 호출했다.

src/threads/thread.c

```
void
update_priority (void)
{
  struct thread *t;
  struct list_elem *e;

  for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e))
  {
    t = list_entry(e, struct thread, allelem);
    t->priority   =   float_to_int(float_sub_float(float_sub_float(int_to_float(PRI_MAX),
float_div_int(t->recent_cpu, 4)), float_mul_int(int_to_float(t->nice), 2)));
    if (t->priority < PRI_MIN)
      t->priority = PRI_MIN;
    if (t->priority > PRI_MAX)
      t->priority = PRI_MAX;
  }

  if (thread_current()->priority < get_max_priority())
  {
    intr_yield_on_return();
  }
}
```

update_load_avg() 함수에서 load_avg 값을 계산해주었다. 식을 다음과 같이 변형해 계산했다.

load_avg = (59/60) * load_avg + (1/60) * ready_threads
= (59 * load_avg + ready_threads) / 60

ready_threads는 idle_thread를 포함하지 않은 수이므로, 만약 현재 thread가 idle_thread가 아니라면 ready_thread를 증가시켜주었다.

src/threads/thread.c

```
void
update_load_avg (void)
{
  int ready_threads = list_size(&ready_list);

  if (thread_current() != idle_thread)
    ready_threads++;

  load_avg = float_div_int(float_add_int(float_mul_int(load_avg, 59), ready_threads),
60);
}
```

update_recent_cpu() 함수에서는 recent_cpu를 계산해주었다.

src/threads/thread.c

```
void
update_recent_cpu (void)
{
  struct list_elem *e;
  struct thread *t;

  for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e))
  {
    t = list_entry(e, struct thread, allelem);
    if (t != idle_thread)
    {
      t->recent_cpu                                                    =
float_add_int(float_mul_float(float_div_float(float_mul_int(load_avg,              2),
float_add_int(float_mul_int(load_avg, 2), 1)), t->recent_cpu), t->nice);
    }
  }
}
```

추가로 현재 thread의 recent_cpu 값을 늘려주는 increment_recent_cpu() 함수를 구현했다.

src/threads/thread.c

```
void
increment_recent_cpu (void)
{
```

```
  struct thread *t = thread_current();
  t->recent_cpu = float_add_int (t->recent_cpu, 1);
}
```

thread_set_priority() 함수 내부에 thread_mlfqs가 true일 때 priority를 변경할 수 없도록 조건문을 추가했다.

src/threads/thread.c

```
void
thread_set_priority (int new_priority)
{
  . . .

  if (thread_mlfqs) return;

   . . .
}
```

thread_set_nice() 함수에서는 nice를 변경하고, 바뀐 nice 값을 이용해 priority를 재계산하여 scheduling해주었다. 즉, 더 이상 가장 높은 priority가 아니라면 yield해주었다.

src/threads/thread.c

```
void
thread_set_nice (int nice UNUSED)
{
  /* Project 3 */
  struct thread *t = thread_current ();

  t->nice = nice;
  t->priority    =    float_to_int(float_sub_float(float_sub_float(int_to_float(PRI_MAX),
float_div_int(t->recent_cpu, 4)), float_mul_int(int_to_float(t->nice), 2)));

  if (t->priority < PRI_MIN)
    t->priority = PRI_MIN;
  if (t->priority > PRI_MAX)
    t->priority = PRI_MAX;
  if (t->priority < get_max_priority())
    thread_yield();
  /* ========= */
}
```

get_max_priority()를 아래처럼 최대 priority를 반환하도록 구현하여 update_priority() 함수와 thread_set_nice() 함수에서 사용했다.

src/threads/thread.c

```
int
get_max_priority (void)
{
  int priority = -1;
  struct thread *t;

  if (!list_empty(&ready_list))
  {
    t = list_entry(list_front(&ready_list), struct thread, elem);
    priority = t->priority;
  }

  return priority;
}
```

비어있던 thread_get_nice() 함수를 수정해 현재 thread의 nice 값을 반환하도록 했다.

src/threads/thread.c

```
int
thread_get_nice (void)
{
  /* Project 3 */
  return thread_current ()->nice;
}
```

비어있던 thread_get_load_avg() 함수를 수정해 현재 load_avg 값에 100을 곱해서 정수형으로 변환해 반환하도록 했다.

src/threads/thread.c

```
int
thread_get_load_avg (void)
{
  /* Project 3 */
  return float_to_int(float_mul_int (load_avg, 100));
```

```
}
```

비어있던 thread_get_recent_cpu() 함수를 수정해 현재 thread의 recent_cpu 값에 100을 곱해서 정수형으로 변환해 반환하도록 했다.

src/threads/thread.c

```
int
thread_get_recent_cpu (void)
{
  /* Project 3 */
  return float_to_int(float_mul_int (thread_current ()->recent_cpu, 100));
}
```

thread_prior_aging이 true 또는 thread_mlfqs가 true인 경우, timer_interrupt가 발생할 때마다 increment_recent_cpu()를 호출해 recent_cpu를 1 증가시키고, timer_ticks() 값이 TIMER_FREQ (현재 100)의 배수라면 update_load_avg(), update_recent_cpu()를 호출해 load_avg, recent_cpu 값을 계산해서 업데이트했다. 그리고 4 tick마다 update_priority()를 호출해 priority를 업데이트했다.

src/devices/timer.c

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  /* ====== Project3 ====== */
  . . .

  /* Update */
  if (thread_prior_aging || thread_mlfqs) {
    increment_recent_cpu ();

    if (timer_ticks () % TIMER_FREQ == 0) {
      update_load_avg ();
      update_recent_cpu ();
    }
    if (timer_ticks () % 4 == 0) {
      update_priority ();
    }
  }
```

```
  /* ====================== */
  thread_tick ();
}
```

C. 시험 및 평가 내용

- priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

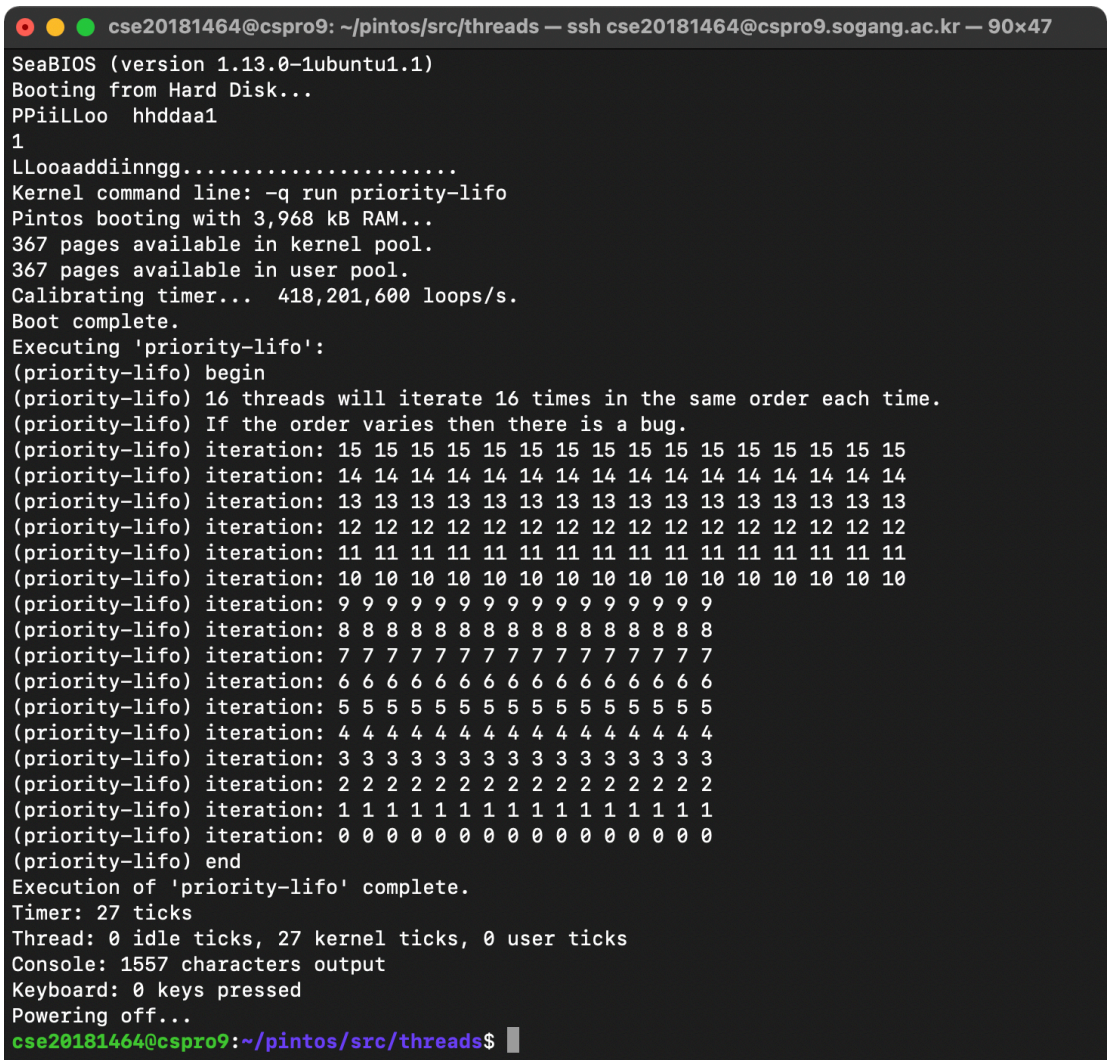
priority-lifo.c에서는 PRI_DEFAULT + THEAD_CNT + 1로 현재 thread의 priority를 set하고, for loop을 돌면서 PRI_DEFAULT + 1 + i의 priority를 가지는 thread를 생성한다. 즉 나중에 생성된 thread일수록 priority가 높다.

이후 PRI_DEFAULT로 현재 thread의 priority를 set하면, 생성된 thread들이 priority가 높은 순서대로 수행되어 THREAD_CNT번 id를 출력한다.

```
26 #define THREAD_CNT 16
   . . .
31 void
32 test_priority_lifo (void)
33 {
    . . .
52
53   thread_set_priority (PRI_DEFAULT + THREAD_CNT + 1);
54   for (i = 0; i < THREAD_CNT; i++)
55     {
56       char name[16];
57       struct simple_thread_data *d = data + i;
58       snprintf (name, sizeof name, "%d", i);
59       d->id = i;
60       d->iterations = 0;
61       d->lock = &lock;
62       d->op = &op;
63       thread_create (name, PRI_DEFAULT + 1 + i, simple_thread_func, d);
64     }
65
66   thread_set_priority (PRI_DEFAULT);
    . . .
69
70   cnt = 0;
71   for (; output < op; output++)
72     {
73       struct simple_thread_data *d;
74
75       ASSERT (*output >= 0 && *output < THREAD_CNT);
76       d = data + *output;
77       if (cnt % THREAD_CNT == 0)
78         printf ("(priority-lifo) iteration:");
79       printf (" %d", d->id);
80       if (++cnt % THREAD_CNT == 0)
81         printf ("\n");
82       d->iterations++;
83     }
84 }
```

테스트 결과, 나중에 생성되어 priority가 높은 thread부터 id를 THREAD_CNT(현재 16)번 출력

하는 것을 확인할 수 있다.


cse20181464@cspro9: ~/pintos/src/threads — ssh cse20181464@cspro9.sogang.ac.kr — 90×47
SeaBIOS (version 1.13.0-1ubuntu1.1)
Booting from Hard Disk...
PPiiLLoo hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -q run priority-lifo
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 418,201,600 loops/s.
Boot complete.
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.
Timer: 27 ticks
Thread: 0 idle ticks, 27 kernel ticks, 0 user ticks
Console: 1557 characters output
Keyboard: 0 keys pressed
Powering off...
cse20181464@cspro9:~/pintos/src/threads$

- make check 수행 결과