운영체제

Pintos Project - Threads

이동석, 임현, 정창현

서문

- 정창현 소감문

대략 한달동안 핀토스 프로젝트를 진행하면서 지난 2 년동안 이렇게 크고 아름다운 코드의 집합을 본 적이 없었습니다. 처음 일주동안은 핀토스의 전체적인 코드의 구조를 이해하면서 괴로운 시간을 보냈습니다. 하지만, 일주일정도 지나면서부터 코드들이 어떻게 움직이는지 보이기 시작했고, 어떻게 문제를 풀어나갈지 조금씩 보이기 시작했습니다. 교수님께서 올려 주신 핀토스 보충자료와 핀토스 매뉴얼을 보면서 차근차근 한 단계씩 밟아보았습니다. 물론 바로 문제를 해결하게 된 것은 아니지만, 이런 코드, 저런 코드를 써 보면서 조금씩 더 완성의 길이 보였습니다. 이런 방대한 프로젝트를 통해 코딩 실력이 한 단계 더 향상되었고, 앞으로도 큰 프로젝트를 통해 더욱 실력있는 개발자가 되고 싶습니다. 그리고 이번 기회를 통해 간접적으로 개발자의 삶을 살아볼 수 있었습니다.

- 이동석 소감문

12 월 4 일 기준으로 약 2-3 주 전부터, 교수님께서 주신 핀토스에 대하여 이해하려고 노력하고 또 고민하였습니다. 이렇게 노력한 것에 비하여, 결과는 부족하지만, 좋은 훌륭한 경험이었습니다. 팀원 2 명과 같이 하면서, 팀웍이 정말 중요하다는 것을 깨달았습니다. 그리고 교수님의 alarm clock 보충자료는 정말 도움이 많이 되었습니다. 감사합니다. 이렇게 큰 코드를 다루어 본 경험이 저의 프로그래밍 실력에 도움이 되었습니다. 저는 만약, 교수님께서 기회가 되셨다면, 교수님께서 핀토스 코드에 대해서 추후에 설명해 주셨으면 좋겠습니다. 저희가 배운 운영체제를 바탕으로, pintos 라는 운영체제의 소스코드 파일을 어떻게 해석하고 수정하면 되는 지에 대해서 더 알고 싶습니다. 물론 과제 제출이 다 끝난 시기를 의미합니다. 그래도 프로그래머라는 높은 벽에 좀더 가까이 다가간 기분입니다. 교수님 그동안 정말로 감사했습니다.

- 임현 소감문

핀토스에 대한 평판은 운영체제라는 과목을 수강 신청하기 전부터 들어왔었습니다. 하지만 수업을 열심히 듣고, 노력하면 쉽게 해낼 수 있을 것이라 생각했습니다. 그래서 저는 운영체제라는 과목에 애정을 가지고 열심히 들었으나, 제가 생각했던 것보다 핀토스의 소스 코드는 많았고, 그것을 이해하는 것만 해도 어려운 과정이었습니다. 하지만 교수님께서 주신 핀토스 보충자료를 토대로 기초를 쌓고, 소스 코드의 주석으로 이해를 돕고, 차근차근 알고리즘을 생각하며, 코드를 짜고, 해결할 수 있었습니다. 그리고 평소 학교를 다니며 양질의 프로젝트를 하고 싶다는 생각이 많았었는데, 이번 핀토스 프로젝트를 하며 학부생 시절에 그 꿈을 이루게 된 것 같아 매우 기쁩니다.

2016 년 12 월 4 일 임현 정창현 이동석

들어가기에 앞서

✓ make check 에 대하여

핀토스 매뉴얼 1.2.1 절의 Testing 부분을 보면 자신이 고친 코드가 잘 변경된 것인지 한 번에 확인해주기 위해 모든 테스트 파일을 make check 라는 하나의 명령어로 확인하는 방법이 나와있습니다. pintos/src/tests 디렉토리의 Make.test 파일에서 verbose 속성을 1 로 set 해주면 make check 라는 명령어로 test 파일의 성공 여부를 pass 나 fail 로 나타내어 줍니다.

✓ Pintos 우선순위에 대하여

핀토스 매뉴얼 2.2.3 절의 Priority Scheduling 에서는 priority 0 이 가장 낮은 우선순위, priority 63 이 가장 높은 우선순위를 가진다고 명시 되어있습니다. 즉, 높은 숫자의 우선순위 값을 가지는 스레드가 높은 우선순위를 갖습니다.

✓ pintos -v -- -q run 명령어에 대하여

위 명령어를 설명하자면 '-v'는 가상 머신을 키지 않고 실행하는 명령어 이고, '-q'는 테스트를 종료하고 핀토스 운영체제를 끝내는 명령어 입니다. 그리고 그 중간에 있는 '--'는 boch 와 pintos 명령어를 구분해주는 명령어 입니다. 저희는 모든 테스트를 저렇게 해줌으로써 핀토스 운영체제가 종료되면서 'idle ticks', 'kernel ticks', 'user ticks'을 확인이 가능해져서 Alarm clock 가 제대로 동작하는지 확인 할 수 있었습니다.

✓ Alarm clock 을 'thread.c' 위주로 개선해준 것에 대하여

많은 이유가 있었지만, 그 중에서도 'tests/threads/mlfqs-load-avg.c'의 10 번째 줄의 주석을 확인하면 timer_interrupt 에서 많은 일을 할 경우 주 스레드가 자체 작업을 수행 할 충분한 시간을 갖지 못 하므로 'therad.c' 위주로 개선을 해주었습니다.

✓ 부록에 대하여

Alarm clock 의 개선을 'thread.c'와 'thread.h'를 사용하지 않고 'timer.c' 만으로 구현할 수도 있다는 것을 보여드리기 위해, 부록으로 추가 구성하여 작성하였습니다.

✓ Multi-level feedback queue scheduler 구현에 대하여

저희는 MLFQS 의 구현을 위해 핀토스 매뉴얼 'Appendix B 4.4BSD Scheduler' 부분을 단계별로 밟아가며 구현하고, 보고서를 작성하였습니다.

✓ 우선순위 스케줄러 소스 코드 부분의 노란색 줄에 대하여

노란색 줄은 기존 코드와 바뀐 코드 부분을 한 눈에 알아보기 쉽게 하기 위해 삽입한 것으로, 이해를 돕기 위해 사용하였습니다.

✓ 코드 공유 방법에 대하여

저희는 바뀐 코드 뒷 부분에는 '///'을 삽입하여 팀원 간 무슨 코드가 삽입되고 수정되었는지 알아보기 쉽게 하였습니다. 또한 제출하는 코드에도 '///'이 포함될 예정이니, 참고하여 주셨으면 감사하겠습니다.

✓ trial_and_error 폴더에 대하여

제출 코드한 폴더 안에 있는 'pintos/tiral_and_error' 폴더는 핀토스 프로젝트를 시작하고부터 시행착오를 겪었던 우리의 소중한 소스파일들 입니다.

차례

0. 팀 구성

0.0. 팀 구성 :

학번	학과	이름	티켓
201511061	컴퓨터과학과	정창현	4
201511041	컴퓨터과학과	이동석	3
201511054	컴퓨터과학과	임현	2

0.1. 역할 분담 :

✓ 정창현 :

Alarm clock 구현, 우선순위 스케줄러 구현 및 보고서 작성, MLFQS 시도 및 보고서 작성

✓ 이동석 :

Alarm clock 시도 및 보고서 감수, 우선순위 스케줄러 구현 및 시도 (priority-fifo 구현, preempt 시도, donation 시도)

✓ 임현 :

Alarm clock 구현 및 보고서 작성, 우선순위 스케줄러 분석, MLFQS 시도, 부록 작성

1. Alarm clock의 개선

[문제정의] 'devices/timer.c'에 정의되어 있는 timer_sleep() 함수를 개선한다. 현재의 timer_sleep()은 기능적으로는 정상 동작하나, 정해진 시간이 경과할 때까지 thread_yield()를 반복 호출하며 busy waiting 하도록 구현되어 있다. 이를 busy waiting 없이 수행하도록 수정하여 개선한다..

[해결] 우선 오른쪽에 있는 현재의 'timer_sleep()' 함수의 개선해야할 점을 설명하자면, ticks 이 경과할 때 까지 'thread_yield()'를 반복 호출 (busy waiting) 하고 있는 부분을, 실행중인 스레드를 리스트로 옮긴 후 'thread_block()'을 호출하는 식으로 바꿔줄 예정입니다.

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

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

thread_yield(); 대신에 thread_block()을 호출하는 식으로 바꿔주는 이유는 running → ready가 되는 상태전이가 running → waiting 이 되게 해주기 위해서 입니다.

그리고 본격적인 시작 전에 현재의 alarm clock 에서는 어떻게 동작하는 지를 알아보기 위해서, 오른쪽 그림은 'pintos -v -- -q alarm-multiple' 이라는 명령어를 실행했을 때의 모습입니다. 우선 간략하게 명령어에 대해 설명해 드리자면 '-v'는 가상 머신을 키지 않고 실행하는 명령어 이고, '-q'는 테스트를 종료하고 핀토스 운영체제를 끝내는 명령어 입니다.

그리고 결과를 보시면 맨 아래와 같이 아직까진 idle ticks가 하나도 활성화되지 않은 모습입니다.

```
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 867 ticks
Thread: 0 idle ticks, 870 kernel ticks, 0 user ticks
```

그래서 이를 해결하기 위해서 저희의 알고리즘에 대해 간략하게 설명해드리자면 (timer_sleep 함수 호출) → (만료되는 시점 계산) → (thread를 연결 리스트에 삽입) → (현재 스레드를 thread_block();) 입니다.

그리고 코드 분석에 앞서 수정되거나 추가된 함수와, 추가된 자료구조 입니다.

수정된 함수
- 'threads/thread.c'의 'thread_init' 함수 - 'devices/timer.c'의 'timer_sleep' 함수 - 'devices/timer.c'의 'timer_interrupt' 함수

추가된 함수
- 'threads/thread.c'의 'thread_sleep' 함수 - 'threads/thread.c'의 'exp_less' 함수 - 'threads/thread.c'의 'thread_unsleep' 함수

추가된 자료구조
- 'threads/thread.c'에 alarm 구조체 - 'threads/thread.c'에 alarms 더블 링크 리스트

1.2.1. 자료구조

- 스레드들이 요청한 알람을 유지하는 linked list
- 알람을 위한 구조체의 예

```
static struct list alarms;
```

```
struct alarm
{
    int64_t expiration;
    struct thread * th;
    struct list_elem elem;
};
```

```
list_init (&alarms);
```

그래서 저희는 우선 'alarms' 라는 스레드들이 요청한 알람을 유지하는 더블 링크 리스트와 'alarm' 이라는 구조체를 'threads/thread.c'에 선언 해주었습니다. 'thread.c'에 선언해준 이유는 그 쪽에 많은 list들이 있어서 그 쪽에 선언해주었습니다. 또한 'thread_init()' 함수에서 alarms 리스트를 초기화 해주었습니다.

1.2.2. timer_sleep() 에서는

- 알람을 해당 스레드와 함께 위 리스트에 매단 후,
- thread_block()을 호출, 해당 스레드를 블록시킴

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

  ASSERT (intr_get_level () == INTR_ON);
  thread_sleep(start + ticks);
}
```

그리고 'timer_sleep()' 함수 안에 'thread_sleep()' 함수를 넣어주었습니다. 이 'thread_sleep()' 함수는 'thread/thread.c'에 선언 해주었는데, 그 이유는 'timer.c'에서 선언했던 alarms를 컴파일러가 인식을 하지 못 했습니다. 그래서 어쩔 수 없이 'thread.c'에서 'thread_sleep()'라는 이름의 'thread_yield()'와 비슷한 함수를 하나 만들고, 그 곳에서 waiting로 상태로 바꿔주는 작업을 진행했습니다. 또한 이 'thread_sleep()'이라는 함수의 매개변수에 'start + ticks'을 넣어주는 이유는 ticks만 넣어줄 경우, 시작하는 시점을 알 수 없는 문제로 끝나는 시간을 알 수 없기 때문입니다.

그리고 본격적인 'thread_sleep()' 함수를 분석하자면, 아래와 같습니다.

```
void
thread_sleep (int64_t ticks)
{
  struct thread *cur;

  enum intr_level old_level;
  old_level = intr_disable ();

  cur = thread_current ();
  ASSERT (cur != idle_thread);

  struct alarm a;
  a.expiration = ticks;
  a.th = cur;

  list_insert_ordered (&alarms, &a.elem, exp_less, NULL);

  thread_block ();

  intr_set_level (old_level);
}
```

우선 기본 틀은 'thread_yield()'를 참고했습니다. 그 후 '*cur'라는 스레드를 뜻 하는 'thread' 구조체를 선언 해주었습니다. 그 후 'cur'라는 변수가 현재 실행 중인 스레드라고 지정해주었습니다. 그 후 만들었었던 alarm 구조체를 a라는 변수로 선언해주었습니다. 그 후 a의 expriration (알람 만료 시간)에 ticks을 넣어주고, a의 th (알림 요청한 스레드)에 앞서 선언해주었던 cur를 넣어주었습니다.

그리고 만료시간을 기준으로 정렬해서 나중에 'unblock()' 시켜주기 위해서, 'lib/kernel/list.c'에 있는 list에 매달면서 정렬해주는 함수인 'list_insert_ordered (struct list *list, struct list_elem *elem, list_less_func *less, void *aux)'에다가 'alarms'라는 list의 주소, 'a'이라는 구조체의 elem의 주소, 구조체의 무엇과 비교할지 명시해주는 반환 형이 부울인 함수인 'exp_less', 그리고 'exp_less' 함수의 매개 변수가 필요하지 않기 때문에 'NULL'을 집어 넣었습니다. 그리고 마지막으로 현재 스레드를 block시키는 'thread_block' 함수를 사용하면서, 스레드를 running → waiting으로 전환해주었습니다.

그리고 앞서 나왔던 'thread.c'에 있는 'exp_less()' 함수를 분석하자면 아래와 같습니다.

```
static bool
exp_less (const struct list_elem *a,const struct list_elem *b, void *aux)
{
  struct alarm *A = list_entry (a, struct alarm, elem);
  struct alarm *B = list_entry (b, struct alarm, elem);

  if(A->expiration < B->expiration) return true;
  else return false;
}
```

우선 'exp_less()' 함수는 매개변수로는 두 개의 list와 현재는 쓰이지 않는 aux라는 변수입니다. 그러면서 alarm *A, *B라는 구조체에 list_entry(LIST_ELEM, STRUCT, MEMBER)라는 매크로를 사용하여, 구조체의 elem 부분에 접근해주었습니다. 그리고 그 후 조건문을 통해 A의 expiration이 B의 expiration보다 더 작을 경우 true를 반환하고, 아닐 경우 false를 반환하는 함수 입니다. 그리고 작을 경우 true를 반환하는 이유는 'list.h'의 주석에도 적혀있고, 만료시간을 오름차순으로 정렬해주었기 때문입니다. 또한 구체적인 'exp_less()'함수에 대한 설명은 우선순위 스케줄러 부분에 추가하였습니다.

1.2.3. 타이머 인터럽트 핸들러 timer_interrupt() 에서는

- 매 틱 발생 시마다 인터럽트 처리 과정에서 위 리스트 중 시간이 만료된 알람이 있는지 검사하고,
- 만료된 알람이 있는 경우, 알람은 리스트에서 제거하고 해당 스레드는 thread_unblock()을 호출, 스레드를 깨움

그리고 이제 block 시켜 놓았던 스레드를 unblock 해주기 위해 'timer_interrupt()' 함수에서 매 인터럽트마다 만료 시간을 검사해서 만료된 알람이 있는 경우, 리스트에서 알람을 제거하고 'thread_unblock()' 함수를 호출할 예정 입니다.

우선 'timer_interrupt()' 함수 안에 'thread_sleep()' 이라는 함수를 호출 해주었습니다. 이 함수는 'threads/thread.c'에 있는데, 그 이유는 앞서 'thread_sleep()' 함수를 'thread.c' 파일 안에 선언한 이유와 같습니다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_unsleep (ticks);
  thread_tick ();
}
```

그리고 'thread_unsleep' 이라는 함수를 분석하면 아래와 같습니다.

```
void
thread_unsleep (int64_t ticks){
  struct alarm *a;
  struct list_elem *b;
  struct thread *c;

  while(list_empty(&alarms) != true){
    a = list_entry (list_begin (&alarms), struct alarm, elem);
    if (a->expiration <= ticks){
      list_pop_front(&alarms);
      thread_unblock(a->th);
    }
    else break;
  }
```

우선 이 함수에서는 '*a'라는 알람 구조체, prev와 next의 주소를 가지고 있는 'b'라는 구조체, 그리고 스레드를 뜻 하는 '*c'를 선언 해주었습니다. 그리고 while문 안에 있는 'list_empty'함수를 이용해서 리스트가 빌 때 까지 반복합니다. 그리고 반복문 내에서 a의 'expiration'(만료시간)이 ticks보다 작거나 같을 경우 'list_pop_front()' 함수를 이용하여 리스트에서 제거하고 'thread_unblock()' 함수를 이용해서 해당 쓰레드를 깨워줍니다. 그리고 아닐 경우에는 'break'만 추가하였는데, 그 이유는 이미 'list_insert_ordered()' 라는 함수를 통해 정렬되어 들어갔기 때문에, 앞 부분만 확인하면 되기 때문입니다.

```
void thread_sleep (int64_t ticks);
static bool exp_less (const struct list_elem *a,const struct list_elem *b, void *aux);
void thread_unsleep (int64_t ticks);
```

그리고 마지막으로 'threads/thread.h' 라는 헤더파일에 'thread.c'에서 사용했던 함수를 'timer.c'에서도 사용하기 위해서 위와 같이 선언하였습니다.

[테스트]

make-check

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
```

alarm-single

```
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.
Timer: 361 ticks
Thread: 250 idle ticks, 114 kernel ticks, 0 user ticks
```

alarm-multiple

```
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 867 ticks
Thread: 550 idle ticks, 320 kernel ticks, 0 user ticks
```

alarm-simultaneous

```
Executing 'alarm-simultaneous':
(alarm-simultaneous) begin
(alarm-simultaneous) Creating 3 threads to sleep 5 times each.
(alarm-simultaneous) Each thread sleeps 10 ticks each time.
(alarm-simultaneous) Within an iteration, all threads should wake up on the same tick.
(alarm-simultaneous) iteration 0, thread 0: woke up after 10 ticks
(alarm-simultaneous) iteration 0, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 0, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 1, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 2, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 3, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 4, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 2: woke up 0 ticks later
(alarm-simultaneous) end
Execution of 'alarm-simultaneous' complete.
Timer: 418 ticks
Thread: 247 idle ticks, 173 kernel ticks, 0 user ticks
```

alarm-priority

```
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
Timer: 594 ticks
Thread: 471 idle ticks, 125 kernel ticks, 0 user ticks
```

alarm-zero

```
Executing 'alarm-zero':
(alarm-zero) begin
(alarm-zero) PASS
(alarm-zero) end
Execution of 'alarm-zero' complete.
Timer: 35 ticks
Thread: 1 idle ticks, 36 kernel ticks, 0 user ticks
```

alarm-negative

```
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.
Timer: 37 ticks
Thread: 1 idle ticks, 38 kernel ticks, 0 user ticks
```

2. 우선순위 스케줄러 구현

[문제정의] (1) 현재 pintos에는 라운드로빈 방식의 스케줄러가 구현되어 있다. 이를 스레드별 우선순위에 따라 스케줄링 할 수 있는 우선순위 스케줄러를 새로 구현한다. (2) 우선순위 스케줄링 방식의 문제점은 우선순위 역전 현상이 일어날 수 있다는 것이다. 이를 방지할 수 있도록 우선순위 스케줄러 priority donation 기능을 구현한다.

[해결]

2.1. Priority Scheduling 구현

Ready queue 의 운영
Ready queue 를 thread 의 priority 순으로 유지 Priority 값이 같은 thread 들이 여럿인 경우 ready queue 에 도착한 순서로 유지

(기존의 RR 방식의 스케줄링은 먼저 온 쓰레드 순서대로 ready queue 에 들어가고, time slice 마다 thread_yield()를 해주어 running 상태의 쓰레드를 ready 로 만들고, list_push_back()함수를 통하여 모든 스레드를 평등하게 동일한 time slice 동안 실행시켰다. 하지만 priority 스케줄링은 list_inset_ordered()를 통하여 priority 순서대로 ready queue 에 삽입한다.)

우선 priority 스케줄링을 구현하기 위해서 ready list 를 우선순위 순으로 유지하기 위해서 스레드가 running 상태에서 waiting 상태로 바뀌는 부분의 리스트 삽입 함수의 알고리즘을 변경할 것이다. 스레드의 상태가 running 에서 waiting 으로 바뀌는 부분은 thread_unblock(), thread_yield(), sema_up() 총 3 개이다. (하지만, sema_up()은 thread_unblock()을 통해 상태가 바뀌므로 실질적으로 고쳐야 할 부분은 두 부분이다.)

```
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_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

list_push_back() 함수 대신 void list_insert_ordered (struct list *, struct list_elem *,

list_less_func *, void *aux); 함수를 사용할건데 여기서 중요한 부분은 세번째 변수인 list_less_func 인데 다음과 같이 정의되어있다.
typedef bool list_less_func (const struct list_elem *a, const struct list_elem *b, void *aux);

의미하는 바는 list_elem 을 담고있는 struct 의 대소비교를 정의한다. 우리는 thread struct 의 priority 필드를 가지고 대소관계를 비교할 것이다.

list_elem 을 담고있는 struct 를 찾으려면 list_entry(LIST_ELEM, STRUCT, MEMBER) 함수를 이용해야 하는데 두번째, 세번째 매개변수로 struct 와, struct 의 첫 부분과 list_elem 과의 offset 을 구할 수 있고, 첫번째 매개변수와 offset 을 가지고 list_elem 을 담고있는 struct 의 시작주소를 구할 수 있다. 즉, list_elem 을 담고있는 struct 를 찾을 수 있는 것이다. (핀토스 매뉴얼 챕터 2 의 2.2.3 에서는 priority 0 가 가장 낮은 우선순위, priority 63 이 가장 높은 우선순위를 가진다고 명시 되어있다. 즉, 높은 숫자의 우선순위 값을 가지는 스레드가 높은 우선순위를 갖기 때문에, priority 값이 높은 스레드를 리스트의 앞에 연결하기 위해서 A 스레드의 priority 값이 더 클 때 true 를 반환했다.)

결론적으로 thread 의 priority 를 비교하기 위해 다음과 같은 함수를 작성했다.

```
static bool
pri_more(const struct list_elem *a, const struct list_elem *b, void *aux)
{
        struct thread *A = list_entry(a, struct thread, elem);
        struct thread *B = list_entry(b, struct thread, elem);

        if (A->priority > B->priority) return true;
        else return false;
}
```

그리고 thread_yield(void) 함수와 thread_unblock(struct thread *t) 함수 내의 list_push_back (&ready_list, &cur->elem); 함수를 list_insert_ordered (&ready_list, &cur->elem, pri_more, NULL);으로 대체한다.

```
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,
pri_more, NULL);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

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

preemption 시점의 변경
RR 의 'time slice 만료시 preempt'제거
대신 thread 가 ready queue 에 들어오는 경우, 현재 running 중인 thread 와 priority 를 비교, 그 결과에 따라 preempt 결정

RR 방식의 스케줄러는 매 tick 마다 time slice 를 검사하여 running 중인 thread 를 강제적으로 interrupt 시키는 코드가 thread_tick()함수에 있다. 그 부분을 지워주어 time slice 마다 interrupt 하지 못하게 한다. (좌측은 기존 소스코드, 우측은 바뀐 소스코드이다.)

```
void
thread_tick (void)
{
  struct thread *t =
thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Enforce preemption. */
  if (++thread_ticks >=
TIME_SLICE)
    intr_yield_on_return ();
}
```

```
void
thread_tick (void)
{
  struct thread *t =
thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;
}
```

thread 가 ready queue 로 들어오는 경우 는 아까 말한듯이 thread_yield(), thread_unblock(), sema_up()이다. thread_yield()는 sehedule()을 통해 무조건 바로 인터럽트 되기 때문에 따로 우선순위 비교를 하지 않아도 되고, sema_up()은 thread_unblock()를 통해 우선순위가 비교되므로 thread_unblock()만 현재 실행중인 스레드와 ready list 의 맨 처음의 스레드의 우선순위를 비교하여 현재 실행중인 스레드의 우선순위가 작을 경우 thread_yield()를 통해 스케줄링 해주면 가장 높은 우선순위의 스레드가 동작하게 된다.

```
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 ,pri_more, NULL);
  t->status = THREAD_READY;
  intr_set_level (old_level);

  if (thread_current() != idle_thread)
    if(thread_current()->priority < list_entry(list_begin(&ready_list),struct thread, elem)->priority)
      thread_yield();
}
```

wait queue 의 운영
Lock, semaphore, cond.varible 의 wait queue 도 priority 순으로 유지

semaphore, lock, cond.varible 의 wait queue 도 priority 순으로 유지한다. Lock 는 value 값이 1 인 특별한 경우의 semaphore 와 같으므로 semaphore 의 함수를 가지고 동작을 하게 되어있으므로 sema_down() 함수에 있는 wait 리스트에 스레드를 연결할 때, list_push_back 대신에 list_push_ordered 로 변경시키면 우선순위순으로 wait queue 가 유지된다.

(좌측은 기존 코드이고, 우측은 변경 후 코드이다.)

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema-
>waiters, &thread_current ()-
>elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_insert_ordered (&sema-
>waiters, &thread_current ()-
>elem, pri_more, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

Priority-sema 라는 테스트 파일을 구동하는 중 오류 해결을 위하여 살펴보던 중 기존 sema_up() 함수의 문제점을 발견하였다. sema_down() 함수의 thread_block() 수행 후 value 값을 낮춘 것과 대칭적으로 sema_up() 함수의 thread_unblock() 함수 수행 전에 value 값을 올려야 하는데 기존 소스코드는 sema_up() 수행 후 value 값을 올려주었다.

(좌측은 기존 코드이고, 우측은 변경 후 코드이다.)

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema-
>waiters))
    thread_unblock (list_entry
(list_pop_front (&sema->waiters),

struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
```

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  sema->value++;
  old_level = intr_disable ();
  if (!list_empty (&sema-
>waiters))
    thread_unblock (list_entry
(list_pop_front (&sema->waiters),
struct thread, elem));

  intr_set_level (old_level);
}
```

condition 의 wait 리스트는 lock 이나 semaphore 와의 그것과는 약간 다른 형태를 가진다. 바로 list 에 연결되는 list_elem 이 thread 의 list_elem 이 아니고, semaphore_elem 이라는 struct 의 list_elem 이라는 점이다. 그래서 이 condition 리스트를 정렬 할 때는 기존의 스레드의 priority 을 비교하는 것과 다른, 보다 다소 복잡한 과정이 필요하다. cond_wait()라는 함수를 보면 semaphore_elem 이라는 구조체의 semaphore 필드의 waitlist 에는 하나의 스레드 밖에 연결이 되지 않기 때문에 여기 연결된 스레드가 semaphore_elem 의 priority 라고 생각을 하면 된다.(좌측의 코드는 일반적인 스레드의 priority 우선순위 비교이고, 오른쪽은 semaphore_elem 의 우선순위 비교이다.)

```
static bool
pri_more (const struct list_elem
*a,const struct list_elem *b, void
*aux)
{
  struct thread *A = list_entry
(a, struct thread, elem);
  struct thread *B = list_entry
(b, struct thread, elem);

  if(A->priority > B->priority)
return true;
  else return false;
}
```

```
static bool
pri_sema_more (const struct
list_elem *a,const struct
list_elem *b, void *aux)
{
  struct semaphore_elem *A =
list_entry (a, struct
semaphore_elem, elem);
  struct semaphore_elem *B =
list_entry (b, struct
semaphore_elem, elem);

   if(list_entry(list_begin(&A-
>semaphore.waiters), struct
thread, elem)->priority >
      list_entry(list_begin(&B-
>semaphore.waiters), struct
thread, elem)->priority) return
true;
    else return false;
}
```

또한, cond_wait() 함수를 보면 semaphore_elem 의 semaphore 가 초기화 된 후에 semaphore_elem 을 condition 리스트에 연결하기 때문에 semaphore_elem 의 semaphore 의 리스트에는 아무 스레드도 연결되어있지 않은 상태로 연결한다. 그래서 일단은 list_push_back()으로 들어온 순서대로 리스트에 연결했다가 추후에 list_sort()함수를 통해 우선순위순으로 정렬되어 유지되게 했다. cond_wait()함수 에서 semaphore_elem 의 semaphore 에 스레드가 메달리는 시점은 sema_down()함수를 호출 해주는 때인데 이 함수를 호출하기 전에 condition 리스트를 정렬하면 마지막에 생성된 스레드가 정렬이 안되고, sema_down()함수 호출 후에 condition 리스트를 정렬하면 처음에 생성된 스레드가 정렬이 안되는 문제점이 생겨서 sema_down()함수 전후로 list 를 정렬해주었다.

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);


  lock_release (lock);
  list_sort(&cond->waiters, pri_sema_more, NULL);
  sema_down (&waiter.semaphore);
  list_sort(&cond->waiters, pri_sema_more, NULL);
  lock_acquire (lock);
}
```

2.2. Priority Donation 구현

Priority inheritance (donation) – priority inversion 의 해결책
높은 우선순위 스레드 TL(우선순위 값 H)가 낮은 우선순위 스레드 TL(우선순위 값 L, L < H)이 이미 가지고 있는 lock을 요청하는 경우, TL의 priority를 일시적으로 TH와 같아지도록 높여줌 (즉, TL 의 우선순위 값을 임시로 H로 설정)
이후 TL 이 lock을 해제하는 순간, TL의 우선순위를 원래의 우선 순위 값 L로 환원; 이후 TH이 lock을 가지게 됨

교수님께서 설명해주신 위의 설명을 우리는 다음과 같이 해석하여 적용시켜 보았다.

높은 우선순위 스레드 TH(우선순위 값 H)가 낮은 우선순위 스레드 TL(우선순위 값 L, L < H)이 이미 가지고 있는 lock을 요청하는 경우,	가지고 있는 thread 확인 :lock -> holder ;가 있는경우, NULL이 아닌경우 우선순위 확인 (현재 쓰레드와 lock 쓰레드) "요청하는 경우" => lock_acquire에다 구현.
TL의 priority를 일시적으로 TH와 같아지도록 높여줌 (즉, TL 의 우선순위 값을 임시로 H로 설정)	TL의 prioprity를 어딘가(original_priority)에 저장하고, TH의 priority를 복사해서 TL의 priority에 저장, 기존 TL의 priority 값은 original_priority에 따로 저장

이후 TL 이 lock을 해제하는 순간, TL의 우선순위를 원래의 우선 순위 값 L로 환원; 이후 TH이 lock을 가지게 됨	TL이 lock을 해제하는 순간: lock_relase에 구현 TL의 우선순위를 원래 우선 순위 값으로 환원: priority 값을 저장해놨던 priority_original donated된 TL의 priority에 저장한다. TH가 lock을 가지게 됨=> lock_acquire 해줌.

먼저, 쓰레드의 기존 priority를 저장할 수 있는 공간은 쓰레드에 만들고, thread_init()함수에서 초기화를 시켜주었다. thread 초기화는 -1로 해 주어 donation이 되었는지 안되었는지 명시적으로 구분이 가능하게 해주었다..

```
struct thread
  {
        ...
  int priority;
  /* Priority. */
    int original_priority;
        ...
};
```

```
static void
init_thread (struct thread *t, const char
*name, int priority)
{
        ...
  t->priority = priority;
  t->original_priority =  -1;
        ...
}
```

그리고 lock_acquire() 함수에서 holder의 유무와 priority 의 비교를 해주어 조건에 맞으면 priority를 donation 해주었다. original_priority의 값을 -1로 초기화 해주었으므로, 조건에 맞으면 original_priority 값에 기존 priority 값을 저장해주는 과정이 필요하다.

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

    if(lock->holder != NULL){
      if(lock->holder->priority < thread_current()->priority){
        if(lock->holder->original_priority == -1){
          lock->holder->original_priority = lock->holder->priority;
        }
        lock->holder->priority = thread_current()->priority;
      }
    }

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

즉, lock의 semaphore의 value가 0일 경우, 그 lock을 가진 semaphore와 현재 thread의 priority 값을 비교하여 lock을 지닌 쓰레드의 priority가 더 작으면 priority donation을 해주었다.

그렇게 되면 priority를 donation 받은 쓰레드가 현재 running 중인 쓰레드로 변경되게 된다.

이후 lock을 release 해줄때 만약 쓰레드의 priority 값과 original_priority 값이 다른경우는 donation 된 상태이므로, priority 값을 기존 priority 값, 즉 original_priority값으로 변경해주고, 다시 original priority는 -1로 set 해주었다.

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  int temp = 0;
  if(thread_current()->original_priority !=-1 ){
    temp = thread_current()->original_priority;
    thread_current()->original_priority = -1;
    thread_set_priority(temp);
  }

  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

Multiple donation의 처리

TL가 TH1 과 TH2 (단 우선 순위는 TH1 < TH2)로부터 donation 받는 경우

test파일인 priority-donate-multiple의 실행동작을 분석해본 결과 우리가 기존에 쓰레드에 변수로 추가했던 original_priority는 제일 처음, 한번의 priority 값만 저장하기 때문에 여러 lock이 있을 경우, 제대로 처리되지 못했다, 그래서 lock에 priority 값을 저장해주는 priority_list라는 list를 만들고, lock_init() 함수에서 초기화해주었다. 그리고 그 리스트는 push back, pop back 명령어를 통해서 스택처럼 사용했다. priority_list를 메달 element도 만들어야 했으므로, priority_elem 이라는 구조체를 만들어서, 그 안에 pre_priority 라는 정수형을 저장할 필드, 리스트를 연결시킬수 있는 list_elem형의 elem 필드를 만들어주었다.

```
struct lock
  {
    struct thread *holder;
    struct semaphore semaphore;
    struct list priority_list;
  };
```

```
struct priority_elem
{
  struct list_elem elem;
  int pre_priority;
};
```

그리고 기존 lock_acquire 함수와 lock_release 함수는 original_priority를 사용하지 않고 priority_list 를 사용하도록 개선하였다.

```
void
lock_acquire (struct lock *lock)
{
        ...
  if(lock->holder != NULL){
        if(lock->holder->priority < thread_current()->priority){
        struct priority_elem a;
        a.pre_priority = lock->holder->priority;
        list_push_back(&lock->priority_list, &a.elem);
    lock->holder->priority = thread_get_priority();
  }
        ...
}
```

```
void
lock_release (struct lock *lock)
{
        ...
    if (!list_empty(&lock->priority_list)){
    struct priority_elem *a = list_entry(list_pop_back(&lock->priority_list), struct
                                        priority_elem, elem);
    thread_set_priority(a->pre_priority);
  }

        ...

}
```

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

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);
  list_init(&lock->priority_list);
}
```

이렇게 구현해보니 priority-donate-maultiple 은 제대로 출력이 되나, priority-donate-one는 정상종료 되지 않는 문제가 생겼다. 확인해보니 lock_release() 함수에서 기존 priority를 모두 반환해주지 못하고 main함수가 종료되는 문제가 생겼다. 그래서 priority_list의 모든 값을 반환해주기 위해서 if 문을 while 문으로 바꾸었다.

```
void
lock_release (struct lock *lock)
{
        ...
    while(!list_empty(&lock->priority_list)){
    struct priority_elem *a = list_entry(list_pop_back(&lock->priority_list), struct
priority_elem, elem);
    thread_set_priority(a->pre_priority);
  }
        ...
```

Nested donation의 처리
TM 이 TH 로부터 donation 받고, 다시 TL 이 TM 로부터 donation 받는 경우

구현하지 못 했습니다.

2.3. Priority 조작 인터페이스 구현

int thread_get_priority (void)
Running thread의 priority 값을 리턴

보충자료에는 priority 값을 return 하는 함수가 없으므로 함수를 선언, 정의 하라고 되어있으나, 핀토스 소스파일에 이미 정의되어있으므로 넘어가겠다.

```
int
thread_get_priority (void)
{
  return thread_current ()->priority;
}
```

void thread_set_priority (int new_priority)
Running thread의 priority 값을 변경
prioiry 가 낮아지는 경우 context switch 가 발생할 수도 있음

보충자료에는 priority 값을 set 하는 함수가 없으므로 함수를 선언, 정의 하라고 되어있으나, 핀토스 소스파일에 이미 정의되어 있으므로 넘어가겠다.

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
}
```

Ready 리스트의 제일 앞에 있는 스레드는 ready 상태의 스레드 중 가장 우선순위가 높은 스레드이고, running 상태의 스레드는 모든 스레드 중 가장 우선순위가 높은 스레드이다. 그런데 만약, thread_set_priority()함수를 통하여 running 상태의 스레드의 priority 값을 변경할 때 ready 리스트의 첫 스레드의 priority 보다 작게 변경시킨다면, 즉시 현재 running 상태이던 스레드를 스케줄 하여 ready 상태로 만들고, ready리스트의 맨 앞의 스레드의 상태를 run으로 만들어야 한다. 그래서 조건문을 통해 ready리스트가 비어 있지 않을 경우에 running스레드의 바뀐 priority가 리스트 의 첫 스레드의 priority보다 더 작으면 thread_yield()를 통해 스케줄링 하였다.

```
void
thread_set_priority (int new_priority)
{
  struct thread *t = thread_current ();
  t ->priority = new_priority;

  if(list_empty(&ready_list) == false){
    struct thread *first = list_entry (list_front(&ready_list), struct thread, elem);
    if (t->priority < first->priority) thread_yield();
  }
}
```

[테스트]

make-check

```
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL 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
FAIL tests/threads/priority-donate-chain
```

priority-change

```
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
Timer: 64 ticks
Thread: 0 idle ticks, 66 kernel ticks, 0 user ticks
```

priority-preempt

```
Executing 'priority-preempt':
(priority-preempt) begin
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 1
(priority-preempt) Thread high-priority iteration 2
(priority-preempt) Thread high-priority iteration 3
(priority-preempt) Thread high-priority iteration 4
(priority-preempt) Thread high-priority done!
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
Timer: 79 ticks
Thread: 0 idle ticks, 82 kernel ticks, 0 user ticks
```

priority-fifo

```
Executing 'priority-fifo':
(priority-fifo) begin
(priority-fifo) 16 threads will iterate 16 times in the same order each time.
(priority-fifo) If the order varies then there is a bug.
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) end
Execution of 'priority-fifo' complete.
Timer: 256 ticks
Thread: 0 idle ticks, 259 kernel ticks, 0 user ticks
```

priority-sema

```
Executing 'priority-sema':
(priority-sema) begin
(priority-sema) Thread priority 30 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 29 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 28 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 27 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 26 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 25 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 24 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 23 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 22 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 21 woke up.
(priority-sema) Back in main thread.
(priority-sema) end
Execution of 'priority-sema' complete.
Timer: 153 ticks
Thread: 0 idle ticks, 156 kernel ticks, 0 user ticks
```

priority-condvar

```
Executing 'priority-condvar':
(priority-condvar) begin
(priority-condvar) Thread priority 23 starting.
(priority-condvar) Thread priority 22 starting.
(priority-condvar) Thread priority 21 starting.
(priority-condvar) Thread priority 30 starting.
(priority-condvar) Thread priority 29 starting.
(priority-condvar) Thread priority 28 starting.
(priority-condvar) Thread priority 27 starting.
(priority-condvar) Thread priority 26 starting.
(priority-condvar) Thread priority 25 starting.
(priority-condvar) Thread priority 24 starting.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 30 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 29 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 28 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 27 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 26 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 25 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 24 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 23 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 22 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 21 woke up.
(priority-condvar) end
Execution of 'priority-condvar' complete.
Timer: 214 ticks
Thread: 0 idle ticks, 216 kernel ticks, 0 user ticks
```

priority-donate-one

```
Executing 'priority-donate-one':
(priority-donate-one) begin
(priority-donate-one) This thread should have priority 32.  Actual priority: 32.
(priority-donate-one) This thread should have priority 33.  Actual priority: 33.
(priority-donate-one) acquire2: got the lock
(priority-donate-one) acquire2: done
(priority-donate-one) acquire1: got the lock
(priority-donate-one) acquire1: done
(priority-donate-one) acquire2, acquire1 must already have finished, in that order.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
Execution of 'priority-donate-one' complete.
Timer: 94 ticks
Thread: 0 idle ticks, 97 kernel ticks, 0 user ticks
```

priority-donate-multiple

```
Executing 'priority-donate-multiple':
(priority-donate-multiple) begin
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Main thread should have priority 33.  Actual priority: 33.
(priority-donate-multiple) Thread b acquired lock b.
(priority-donate-multiple) Thread b finished.
(priority-donate-multiple) Thread b should have just finished.
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Thread a acquired lock a.
(priority-donate-multiple) Thread a finished.
(priority-donate-multiple) Thread a should have just finished.
(priority-donate-multiple) Main thread should have priority 31.  Actual priority: 31.
(priority-donate-multiple) end
Execution of 'priority-donate-multiple' complete.
Timer: 115 ticks
Thread: 0 idle ticks, 117 kernel ticks, 0 user ticks
```

priority-donate-multiple2

```
Executing 'priority-donate-multiple2':
(priority-donate-multiple2) begin
(priority-donate-multiple2) Main thread should have priority 34.  Actual priority: 34.
(priority-donate-multiple2) Main thread should have priority 36.  Actual priority: 36.
(priority-donate-multiple2) Thread c finished.
(priority-donate-multiple2) Thread a acquired lock a.
(priority-donate-multiple2) Thread a finished.
(priority-donate-multiple2) Main thread should have priority 36.  Actual priority: 31.
(priority-donate-multiple2) Thread b acquired lock b.
(priority-donate-multiple2) Thread b finished.
(priority-donate-multiple2) Threads b, a, c should have just finished, in that order.
(priority-donate-multiple2) Main thread should have priority 31.  Actual priority: 34.
(priority-donate-multiple2) end
Execution of 'priority-donate-multiple2' complete.
Timer: 122 ticks
Thread: 0 idle ticks, 125 kernel ticks, 0 user ticks
```

priority-donate-nest

```
Executing 'priority-donate-nest':
(priority-donate-nest) begin
(priority-donate-nest) Low thread should have priority 32.  Actual priority: 32.
(priority-donate-nest) Low thread should have priority 33.  Actual priority: 32.
(priority-donate-nest) Medium thread should have priority 33.  Actual priority: 33.
(priority-donate-nest) Medium thread got the lock.
(priority-donate-nest) High thread got the lock.
(priority-donate-nest) High thread finished.
(priority-donate-nest) High thread should have just finished.
(priority-donate-nest) Middle thread finished.
(priority-donate-nest) Medium thread should just have finished.
(priority-donate-nest) Low thread should have priority 31.  Actual priority: 31.
(priority-donate-nest) end
Execution of 'priority-donate-nest' complete.
Timer: 113 ticks
Thread: 0 idle ticks, 115 kernel ticks, 0 user ticks
```

priority-donate-chain

```
Executing 'priority-donate-chain':
(priority-donate-chain) begin
(priority-donate-chain) main got lock.
(priority-donate-chain) main should have priority 3.  Actual priority: 3.
(priority-donate-chain) main should have priority 6.  Actual priority: 3.
(priority-donate-chain) interloper 2 finished.
(priority-donate-chain) main should have priority 9.  Actual priority: 3.
(priority-donate-chain) interloper 3 finished.
(priority-donate-chain) main should have priority 12.  Actual priority: 3.
(priority-donate-chain) interloper 4 finished.
(priority-donate-chain) main should have priority 15.  Actual priority: 3.
(priority-donate-chain) interloper 5 finished.
(priority-donate-chain) main should have priority 18.  Actual priority: 3.
(priority-donate-chain) interloper 6 finished.
(priority-donate-chain) main should have priority 21.  Actual priority: 3.
(priority-donate-chain) interloper 7 finished.
(priority-donate-chain) interloper 1 finished.
(priority-donate-chain) thread 1 got lock
(priority-donate-chain) thread 1 should have priority 21. Actual priority: 6
(priority-donate-chain) thread 2 got lock
(priority-donate-chain) thread 2 should have priority 21. Actual priority: 9
(priority-donate-chain) thread 3 got lock
(priority-donate-chain) thread 3 should have priority 21. Actual priority: 12
(priority-donate-chain) thread 4 got lock
(priority-donate-chain) thread 4 should have priority 21. Actual priority: 15
(priority-donate-chain) thread 5 got lock
(priority-donate-chain) thread 5 should have priority 21. Actual priority: 18
(priority-donate-chain) thread 6 got lock
(priority-donate-chain) thread 6 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 7 got lock
(priority-donate-chain) thread 7 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 7 finishing with priority 21.
(priority-donate-chain) thread 6 finishing with priority 18.
(priority-donate-chain) thread 5 finishing with priority 15.
(priority-donate-chain) thread 4 finishing with priority 12.
(priority-donate-chain) thread 3 finishing with priority 9.
(priority-donate-chain) thread 2 finishing with priority 6.
(priority-donate-chain) thread 1 finishing with priority 3.
(priority-donate-chain) main finishing with priority 0.
(priority-donate-chain) end
Execution of 'priority-donate-chain' complete.
Timer: 312 ticks
Thread: 0 idle ticks, 314 kernel ticks, 0 user ticks
```

priority-donate-sema

```
Executing 'priority-donate-sema':
(priority-donate-sema) begin
(priority-donate-sema) Thread L acquired lock.
(priority-donate-sema) Thread M finished.
(priority-donate-sema) Main thread finished.
(priority-donate-sema) end
Execution of 'priority-donate-sema' complete.
Timer: 57 ticks
Thread: 0 idle ticks, 59 kernel ticks, 0 user ticks
```

priority-donate-lower

```
Executing 'priority-donate-lower':
(priority-donate-lower) begin
(priority-donate-lower) Main thread should have priority 41.  Actual priority: 41.
(priority-donate-lower) Lowering base priority...
(priority-donate-lower) Main thread should have priority 41.  Actual priority: 21.
(priority-donate-lower) acquire: got the lock
(priority-donate-lower) acquire: done
(priority-donate-lower) acquire must already have finished.
(priority-donate-lower) Main thread should have priority 21.  Actual priority: 31.
(priority-donate-lower) end
Execution of 'priority-donate-lower' complete.
Timer: 87 ticks
Thread: 0 idle ticks, 90 kernel ticks, 0 user ticks
```

3. Multi-level feedback queue scheduler 구현

[문제정의] 평균 응답 시간의 개선을 위해 4.4BSD 운영체제의 스케줄러와 유사한 동작을 하는 multilevel feedback queue schedule(MLFQS)를 구현한다

[해결]

3.1 Niceness

nice는 어떤 thread가 더 좋은 것인지를 알려주는 변수다. nice는 -20부터 20까지의 정수값을 가지는데 음의 nice값은 우선순위를 높이고, 양의 nice는 우선순위를 낮춘다. 즉, nice는 cpu-burst-time과 관련 있다. nice 값은 부모로부터 상속받으므로 thrad_create() 함수에 't->nice = thread_current()->nice;' 을 추가해준다.

```
int
thread_get_nice (void)
{
  return thread_current()->nice;
}
```

```
int
thread_get_nice (void)
{
  return thread_current()->nice;
}
```

하지만, nice값을 변경하게 되면 priority값도 변경되므로 thread_set_priority() 함수를 호출하여 우선순위를 다시 정해주어야 한다.(recent_cpu에 관한 내용은 나중에 나온다.)

3.2 Calculating Priority

여기서 multi level feedback queue는 64단계인데 이는 쓰레드를 우선순위별로 queue에 넣기 위함이다.

B.1에서 언급한 것과 마찬가지로 우선순위는 nice의 값에 큰 영향을 받는다. 우선순위 값은 매 4tick마다 모든 쓰레드에서 다시 계산된다. 그리고 그 우선순위를 계산하는 공식은 'priority = PRI_MAX – (recent_cpu / 4) – (nice * 2)' 이다. 따라서 thread_set_nice()함수는 다음처럼 바꾸어야 한다.

```
void
thread_set_nice (int new_nice)
{
  thread_current()->nice = new_nice;

  thread_set_priority(PRI_MAX - (thread_get_recent_cpu() / 4) - (nice * 2) );
}
```

3.3 Calculating recent_cpu

recent_cpu는 쓰레드가 CPU 타임(cpu burst time)이 얼마나 되는지를 나타내는 변수이다. 쓰레드가 생성될 때 부모의 recent_cpu값을 전달받으므로 thread_create() 함수에

't->recent_cpu = thread_current()->recent_cpu;' 을 추가해준다.

recent_cpu를 구하는 공식은

'recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu+nice' 로 계산된다.

매 4틱마다 계산되므로 thread_tick() 함수에 recent_cpu를 계산하는 함수를 추가하였다.

3.4 Calculating load_avg

load_avg는 시스템의 평균 로드량을 나타내는 전역변수이다.

load_avg를 구하는 공식은

'load_avg = (59/60)*load_avg + (1/60)*ready_threads' 로 계산된다.

매 4틱마다 계산되므로 thread_tick() 함수에 load_avg를 계산하는 함수를 추가하였다.

3.5 Summary

위의 niceness, priority, recent_cpu, load_avg에 대해 만든 함수는 다음과 같다.

```
int
thread_get_load_avg (void)
{

  return load_avg;
}
```

```
int
thread_get_recent_cpu (void)
{
  return thread_current()->recent_cpu;
}
```

```
void
thread_calculate_recent_cpu (void)
{
        thread_current()->recent_cpu = (2 * load_avg) / (2 * load_avg + 1)*
thread_current()->recent_cpu + thrad_get_nice()
}
```

```
void
thread_calculate_load_avg (void)
{
  struct list_elem *e;
  int cnt = 1;
  for (e = list_begin (&ready_list); e != list_end (&ready_list); e = list_next (e)){
    if(list_entry(e,struct thread, elem) != idle_thread) cnt++;
  }
```

3.6 Fixed-Point Real Arithmetic

위의 nice나 priority는 모두 정수이나, recent_cpu나 load_avg는 소수로 표현해야 한다. 그리고, 핀토스 운영체제는 부동소수점 계산을 하지 못하여 고정소수점으로 나타내야 한다핀토스 매뉴얼 17페이지에 의하면 fixed-point.h라는 이름의 헤더파일을 만들어서 고정소수점 연산을 간편하게 할 수 있게 나타내야한다. 고정소수점은 매뉴얼 B.6절의 예시와 같이 17.14형식을 사용하였다.

다른 헤더파일의 틀을 빌려서 다음과 같이 작성하였다.

```
#ifndef THREADS_FIXED-POINT_H
#define THREADS_FIXED-POINT_H

#define P 17
#define Q 14
#define f 1<<(Q)

#define Convert_n_to_fixed_point(n) (n)*(f)
#define Convert_x_to_integer_rtz(x) (x)/(f)
#define Convert_x_to_integer_rtn(x) ((x) >= 0 ? ((x) + (f) / 2) / (f) : ((x) - (f) / 2) / (f))
#define Add_x_and_n (x, n) + (n) * (f)
#define Sub_x_and_n (x, n) (x) - (n) * (f)
#define Multiply_x_by_y (x, y) ((int64_t)(x)) * (y) / (f)
#define Divide_x_by_y (x, y) ((int64_t)(x)) * (f) / (y)


#endif /* threads/fixed-point.h */
```

위의 사항들을 고려하여 multi-level-feedback-queue를 구현하려 노력하였으나, 아쉽게도 구현해내지 못하였다. 그리고 이 코드를 추가할 경우 우선순위 스케줄러에 문제가 생겨서 제출 코드에는 추가하지 않았습니다.

[테스트]

make-check

```
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
FAIL tests/threads/mlfqs-fair-2
FAIL tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
```

부록

본 부록에는 Alarm clock을 'timer.c'만을 수정하여 구현한 것을 적어놓았습니다.

우선 기본적인 알고리즘과 함수의 내용은 비슷하기 때문에 구체적인 설명은 하지 않도록 하겠습니다. 다음은 본 구현에서 변경된 부분입니다.

수정된 함수
- 'devices/timer.c'의 'timer_init' 함수 - 'devices/timer.c'의 'timer_sleep' 함수 - 'devices/timer.c'의 'timer_interrupt' 함수

추가된 함수
- 'devices/timer.c'의 'exp_less' 함수

추가된 자료구조
- 'devices/timer.c'에 alarm 구조체 - 'devices/timer.c'에 alarms 더블 링크 리스트

그리고 다음은 본격적인 코드 부분 입니다.

1. 자료구조
- 스레드들이 요청한 알람을 유지하는 linked list - 알람을 위한 구조체의 예

우선 'threads/thread.c'에 선언 해주었던 더블 링크 리스트인 'alamrs'와 구조체인 'alarm'을 'devices/timer.c'에 선언해주었습니다. 또한 'thread_init()' 함수에 선언해주었던 alarms 리스트를 'timer_init'에 선언해주었습니다.

```
static struct list alarms;
struct alarm
{
    int64_t expiration;
    struct thread * th;
    struct list_elem elem;
};
list_init (&alarms);
```

2. timer_sleep() 에서는
- 알람을 해당 스레드와 함께 위 리스트에 매단 후 - thread_block()을 호출, 해당 스레드를 블록시킴

그리고 기존의 구현에서 와는 달리 'timer_sleep()' 함수 안에 바로 구현해주었습니다.

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

  enum intr_level old_level;

  ASSERT (intr_get_level () == INTR_ON);

  struct thread *cur;

  old_level = intr_disable();

  cur = thread_current();

  struct alarm a;
  a.expiration = ticks + start;
  a.th = cur;

  list_insert_ordered (&alarms, &a.elem, exp_less, NULL);

  thread_block();

  intr_set_level (old_level);
}
```

그리고 'exp_less' 함수 또한 'timer.c'에 추가해주었습니다.

3. 타이머 인터럽트 핸들러 timer_interrupt() 에서는
- 매 틱 발생 시마다 인터럽트 처리 과정에서 위 리스트 중 시간이 만료된 알람이 있는지 검사하고, - 만료된 알람이 있는 경우, 알람은 리스트에서 제거하고 해당 스레드는 thread_unblock() 을 호출, 스레드를 깨움

그리고 'timer_interrupt' 또한 기존의 구현과는 달리 바로 구현하였습니다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  struct alarm *a;
  struct list_elem *b;
  struct thread *c;

  while (list_empty(&alarms) != true) {
    a = list_entry (list_begin (&alarms), struct alarm, elem);
    if (a -> expiration <= ticks) {
      list_pop_front(&alarms);
      thread_unblock(a -> th);
    }
    else break;
  }

  thread_tick ();
}
```

감사의 말

여태까지 저희 보고서를 읽어주셔서 감사합니다.