운영체제

1분반

Pintos Project - Thread scheduling

이름: 한규현
학년: 4
학번: 201810896
학과: 융합전자공학전공

1. 프로젝트 개요

Pintos 에 우선순위 스케줄러(priority scheduler)를 구현한다. 또한 우선순위 스케줄링 시 발생할 수 있는 우선순위 역전 (priority inversion) 을 방지할 수 있는 priority inheritance(donation) 기능을 구현한다.

2. 프로젝트 내용

2.1 문제 정의

(1) 현재 pintos 에는 라운드로빈 스케줄러가 구현되어 있다. 이를 수정하여 스레드 별 우선순위에 따라 스케줄링 할 수 있는 우선순위 스케줄러를 새로 구현한다. 특히, preemptive dynamic priority scheduling 을 구현한다.
(2) 우선순위 스케줄링 방식의 문제점은 우선순위 역전 현상이 일어날 수 있다는 것이다. 이를 방지할 수 있도록 우선순위 스케줄러에 priority inheritance(donation) 기능을 구현한다. 단, 우선순위 역전 현상은 다양한 자원에 대해 발생할 수 있는 문제이나, 이번 과제에서는 락(lock) 관련된 우선순위 역전 현상만 해결한다.
(3) 각 스레드가 자신의 우선순위를 확인하고 우선순위를 변경할 수 있도록 두 가지 함수 void thread_set_priority(int new_priority)와 int thread_get_priority(void)를 구현한다.

2.2 파일 수정 목록

아래는 함수 프로토타입 선언을 제외하고 추가 또는 삭제한 부분을 정리한 표이다.

수정 파일	수정내용
threads/thread.h	thread 구조체 수정
threads/thread.c	bool thread_priority_cmp() 함수 추가
	thread_unblock() 함수 수정
	thread_yield() 함수 수정
	void thread_preempt_test () 함수 추가
	void priority_reset () 함수 추가
	tid_t thread_create () 함수 수정
	static void init_thread () 함수 수정
	bool thread_priority_donate_cmp () 함수 추가
	void release_lock_remove () 함수 추가
	void priority_reset () 함수 추가
	void thread_set_priority () 함수 수정
	void thread_get_priority () 함수 수정
threads/synch.c	void sema_down() 함수 수정
	void sema_up () 함수 수정
	void priority_donate () 함수 추가
	void lock_acquire () 함수 수정
	void lock_release () 함수 수정

2.3 수정 코드 설명

2.3.1 우선순위 스케쥴러 구현

현재 pintos는 ready queue에 thread가 추가될 때, list_push_back() 함수를 사용해서 ready queue에 thread를 추가한다. 따라서 ready queue에는 thread가 항상 queue에 들어가게 되고, 순서대로 round-robin 방식 스케줄링을 한다.

먼저 ready queue에 push할 때, priority 순서에 맞춰 thread가 들어가도록 구현하는 방식으로 preemptive dynamic priority scheduling을 구현한다. 이를 위해 list_insert_ordered() 함수를 사용한다. 이때 list_insert_ordered() 함수에 3번재 parameter에 priority를 비교할 수 있는 return 값을 만들기 위해 thread_priority_cmp() 함수를 구현했다.

threads/thread.c

```
bool
thread_priority_cmp (const struct list_elem *a, const struct list_elem *b, void
*aux UNUSED)
{
  struct thread *thread_a = list_entry(a, struct thread, elem);
  struct thread *thread_b = list_entry(b, struct thread, elem);

  if (thread_a != NULL && thread_b != NULL)
  {
    if (thread_a->priority > thread_b->priority)
      return true;
    else
      return false;
  }
  return false;
}
```

해당 함수는 두 개의 thread에 priority를 비교하는 함수로 thread_a가 크면 true thread_b가 크면 false를 return 하는 함수를 구현했다.

이후 list_push_back()에 해당하는 부분을 list_insert_ordered()로 바꾸고, parameter로 thread_priority_cmp() 함수에 return 값을 넣었다.

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_push_back (&ready_list, &t->elem);
  list_insert_ordered (&ready_list, &t->elem, thread_priority_cmp, 0);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

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_push_back (&ready_list, &cur->elem);
    list_insert_ordered (&ready_list, &cur->elem, thread_priority_cmp, 0);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

thread_unblock(), thread_yield() 두 함수 모두 ready queue에 들어갈 때, list_push_back() 대신, list_insert_ordered()를 사용해서 ready queue에 thread 삽입을 priority 순으로 들어가게 완료했다.

추가로 running 상태의 thread가 priority가 변경됐을 때, ready queue에 thread 보다 priority가 낮은 경우 context switching이 일어나야 한다. 이를 위해 thread_preempt_test() 함수를 구현했다.

threads/thread.c

```
void
thread_preempt_test (void)
{
  if (!list_empty (&ready_list) && thread_current ()->priority
    < list_entry (list_front (&ready_list), struct thread, elem)->priority)
  thread_yield ();
}
```

현재 thread와 ready_queue에 가장 앞 thread에 priority를 비교해서 더 높은 priority를 가지는 경우 thread_yield() 함수가 호출되서 context switching이 되도록 했다.

구현한 thread_preempt_test() 함수를 thread_create() 함수와 thread_set_priority() 함수에 추가하였다.

threads/thread.c

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  //...생략

  /* Add to run queue. */
  thread_unblock (t);
  thread_preempt_test ();

  //...생략
}
```

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

  priority_reset ();
  thread_preempt_test ();
}
```

위 두 함수에 thread_preempt_test()를 호출해주면, priority가 변경될 때, context switching이 되게 된다.

2.3.2 priority inheritance(donation) 기능을 구현

먼저 thread 구조체를 수정하고, 수정 부분을 초기화 하는 init_thread를 수정한다.

threads/thread.h

```
struct thread
  {
// 생략...

    int priority;
    int priority_origin;

    struct lock *wait_lock_release;
    struct list donations;
    struct list_elem donation_elem;

    struct list_elem allelem;             /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;                /* List element. */

// 생략...
  };
```

priority_origin 변수를 사용해서 기존에 priority 값을 저장한다. wait_lock_realse는 thread 중, lock이 release 되면 lock을 받을 수 있도록 하는 리스트이다. donations는 자신에 priority를 빌려준 thread의 list이고, donation_elem은 donations 리스트를 관리하기 위한 elem이다.

threads/thread.c

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
// 생략...
  t->priority = priority;
  t->magic = THREAD_MAGIC;
  t->priority_origin = priority;
  t->wait_lock_release = NULL;
  list_init (&t->donations);
  list_push_back (&all_list, &t->allelem);
}
```

위에 thread 구조체에서 추가한 부분을 초기화 할 수 있도록 코드를 작성했다.

thread가 하나의 공유자원을 사용하기 위해 여러 thread가 대기하고 있을 때, 공유자원을 사용했던 thread가 사용을 마친 경우 공유자원을 적절히 배분하여 우선순위 역전현상을 방지한다.

기능을 구현하기 위해 먼저 sema_down(), sema_up()을 수정했다. 자원을 사용하고자 할 때 thread는 sema_down()을 호출하고, 사용을 마치면 sema_up()을 호출한다.

threads/synch.c

```
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,
      thread_priority_cmp, 0);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

sema_down에서 value == 0인 경우 즉 공유자원이 없을 때, list_insert_ordered() 함수를 호출하도록 코드를 수정했다. 이를 통해 앞선 문제 해결방법과 같게 priority 순으로 공유자원을 기다리는 seam->waiters 리스트에 thread가 들어가도록 했다. 이때 앞에서 작성했던 thread_priority_cmp() 함수를 사용했다.

threads/synch.c

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
  {
    list_sort (&sema->waiters, thread_priority_cmp, 0);
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;
  thread_preempt_test ();
  intr_set_level (old_level);
}
```

sema_up() 은 sema->waiters 리스트를 다시 한번 priority 순으로 재정렬하고, thread_unblock()을 호출했다. 공유자원 사용을 마치면 sema->value++;을 통해 다른 thread가 사용할 수 있게 하고, thread_test_preemption()을 호출해 context switching이 일어나도록 한다.

본격적으로 donate(inheritance)를 구현하기 위해 donation_elem에서 priority를 비교하는 함수를 작성했다.

threads/thread.c

```
bool
thread_priority_donate_cmp (const struct list_elem *a,
  const struct list_elem *b, void *aux UNUSED)
{
  return list_entry (a, struct thread, donation_elem)->priority
  > list_entry (b, struct thread, donation_elem)->priority;
}
```

위 함수 호출을 통해 donation_elem에서 priority를 비교할 수 있다.

threads/thread.c

```
void
priority_donate (void)
{
  int len;
  struct thread *cur = thread_current ();

  for (len = 0; len < 8; len++) {
    if (!cur->wait_lock_release) break;
      struct thread *holder = cur->wait_lock_release->holder;
      holder->priority = cur->priority;
      cur = holder;
  }
}
```

priority를 donate 하는 함수인 priority_donate()를 작성했다. nested의 최대 깊이 8을 지정하고, wait_lock_release에 lock을 기다리는 thread가 있다면 holder에 있는 thread에게 priority를 깊이 8만큼 donate 시킨다. wait_lock_release에 더 이상 thread가 없는 경우 break 된다.

앞선 부분에서 sema_down()과 sema_up()을 수정 완료 후, lock_acquire() 함수와 lock_release() 함수를 아래와 같이 수정했다.

threads/synch.c

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

  struct thread *cur = thread_current ();
  if (lock->holder)
  {
    cur->wait_lock_release = lock;
    list_insert_ordered (&lock->holder->donations,
    &cur->donation_elem, thread_priority_donate_cmp, 0);
    priority_donate ();
  }
  sema_down (&lock->semaphore);

  cur->wait_lock_release = NULL;
```

```
  lock->holder = cur;
}
```

threads/synch.c

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

  lock->holder = NULL;

  release_lock_remove (lock);
  priority_reset ();

  sema_up (&lock->semaphore);
}
```

lock_acquire() 에서는 sema_down()을 호출하고, lock_release()에서는 sema_up()을 호출한다. lock 관련해서 sema_down()과 sema_up()을 수정했기 때문에 공유자원을 사용할 때, priority에 순서로 사용할 수 있도록 했다.

lock_acquire()에서 lock->holder는 현재 lock을 소유하고 있는 thread로 만약 lock을 이미 소유하고 있는 thread가 있다면, wait_lock_release 리스트에 thread를 추가한다.
lock->holder->donations에 현재 thread를 추가하고 priority donation을 한다.

이때 lock이 가진 holder를 비우고, sema_up을 할 때, donate 받은 priority를 donations 리스트에서 빼고, priority를 재설정해줘야 한다. 이 기능을 release_lock_remove() 함수를 구현하여 해당 기능을 하게 했다.

threads/thread.c

```
void
release_lock_remove (struct lock *lock)
{
  struct list_elem *e;
  struct thread *cur = thread_current ();

  for (e = list_begin (&cur->donations); e != list_end (&cur->donations);
       e =list_next(e)) {
    struct thread *t = list_entry (e, struct thread, donation_elem);
```

```
    if (t->wait_lock_release == lock)
      list_remove (&t->donation_elem);
  }
}
```

cur->donations 리스트를 체크하면서, wait_lock_release가 현재 release 하는 lock 인 경우 해당 thread를 donation_elem에서 삭제하도록 했다.

threads/thread.c

```
void
priority_reset (void)
{
  struct thread *cur = thread_current ();
  struct lock *lock = cur->wait_lock_release;

  cur->priority = cur->priority_origin;

  if(!list_empty (&cur->donations)) {
    list_sort (&cur->donations, thread_priority_donate_cmp, 0);

    struct thread *front = list_entry (list_front (&cur->donations), struct thread,
donation_elem);
    if (front->priority > cur->priority)
      cur->priority = front->priority;
  }
}
```

donations 리스트에 아무것도 없다면 priority_origin으로 priority가 된다. 만약 리스트에 thread가 있다면 가장 priority가 높은 thread를 고르게 list_sort()를 호출했다. 이후 priority_origin과 비교하여 priority가 더 높은 것을 priority로 설정한다.

2.3.3 thread_set_priority(), thread_get_priority() 구현

threads/thread.c

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority_origin = new_priority;

  priority_reset ();
  thread_preempt_test ();
}
```

현재 실행 중인 thread의 priority를 변경하는 함수이다. priority를 priority_reset()하고 priority를 바탕으로 동작하도록 thread_preempt_test() 함수를 호출했다.

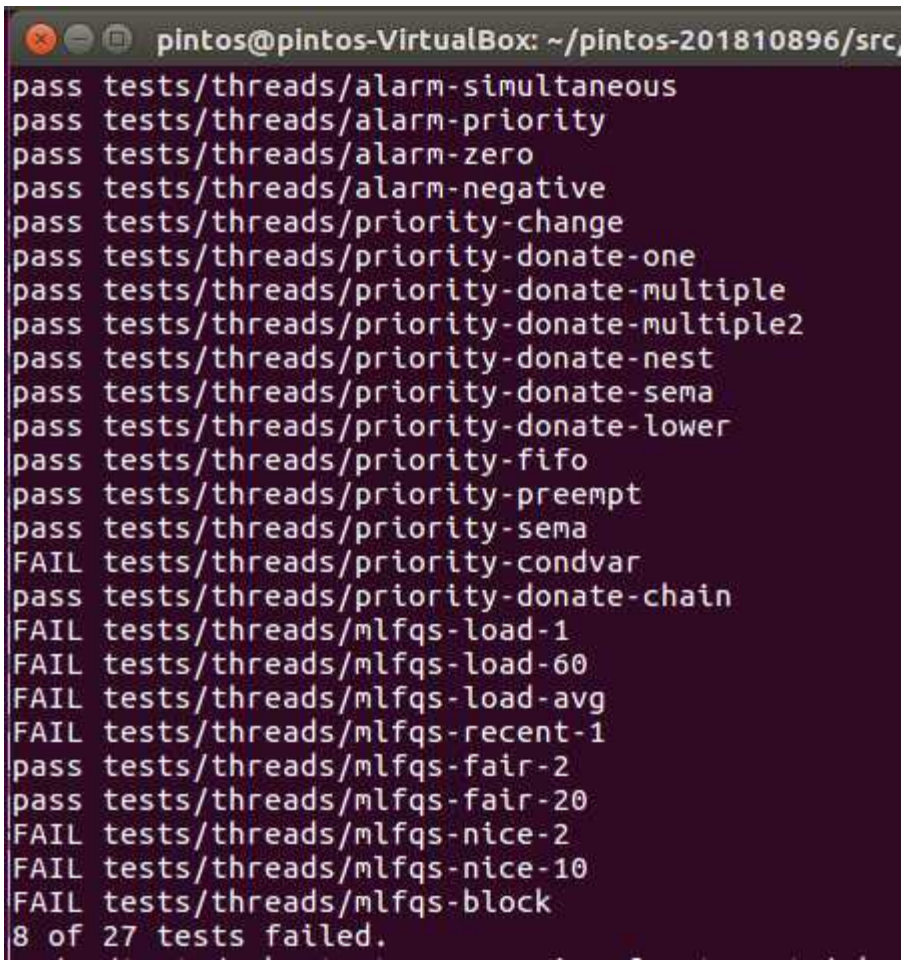
threads/thread.c

```
/* Returns the current thread's priority. */
int
thread_get_priority (void)
{
  return thread_current ()->priority;
}
```

현재 실행 중인 thread의 priority를 반환하는 함수이다.

3. 테스트

3.1 make check



priority-change, priority-preempt, priority-sema, priority-donate-one, priority-donate-multiple, priority-donate-multiple2, priority-donate-nest, priority-donate-chain, prioritydonate-sema, priority-donate-lower 에 대해서 pass tests가 된 것을 확인할 수 있다.

끝.