Priority Scheduler 구현

배수한

3학년 201910899, 융합전자공학과

201910899@sangmyung.kr

Introduction

현재 pintos에서 스레드 스케줄러는 round-robin 방식을 채택하고 있습니다. 이 방식은 스레드 간의 우선순위 없이 ready_list에 들어온 순서대로 실행되지만 제대로 된 우선순위 스케줄링이 이루어지지 않고 있습니다. 라운드로빈 스케줄링은 공정성, 간단한 구현 등 장점들이 존재하나, 이는 우선순위를 고려하지 않기 때문에 중요한 작업에 더 많은 CPU 시간을 할당하는 것이 어렵고, 스레드의 작업이 긴 경우 스레드 전환 비용이 높아질 수 있는 단점이 있습니다.

따라서 본 보고서에서는 이러한 단점들을 극복하기 위해 스케줄링 방식을 Priority 스케줄링 방식으로 개선하고, pintos가 제공하는 테스트를 통해 구현한 Priority 스케줄링이 제대로 동작하는지 확인하고자 합니다.

The proposed method

구현한 함수는 다음과 같습니다. thread.c에 구현한 함수들은 synch.c에서도 사용되므로, thread.h에도 함수들을 정의합니다.

```
bool cmp_priority (struct list_elem *x, struct list_elem *y, void *aux UNUSED);
void thread_preemption (void);
bool cmp_donate_priority (const struct list_elem *x, const struct list_elem *y, void *aux UNUSED);
void donate_priority (void);
void del_lock (struct lock *lock);
void renew_priority (void);
```

그림 1. thread.c, thread.h에 추가한 함수

1) RR방식에서 스레드가 레디 큐에 들어가는 경우는 총 3가지입니다.

1. 스레드 생성시 thread_unblock() 호출

2. 스레드가 wait->ready상태가 될 때

3. Time slice 만료 시 thread_yield() 호출

또한 RR방식은 스레드가 레디 큐에 들어갈 때 list_push_back()함수를 통해 무조건 맨 뒤에 들어가는 방식을 채택하고 있습니다.

Priority 스케줄링을 구현하기 위해, 레디 큐를 항상 priority기준 내림차순으로 정렬된 상태로 유지합니다. 이를 위해 스레드가 레디 큐에 push 할 때, priority 순서에 맞추어 push 하는 방법을 사용합니다. 우선순위 내림차순 정렬을 하기 위해, list_push_back() 대신 list_insert_ordered()함수를 사용해야 합니다. list_insert_ordered()함수는 반복문을 사용해 들어갈 위치를 찾고 위치를 찾으면 break를 수행하여 해당 위치에 삽입하는 함수이므로, less (elem, e, aux) 는 elem > e 일 때 true를 반환하는 함수를 따로 구현해야 합니다. 만약 두 스레드의 우선순위가 같은 경우, 이 함수는 false를 반환하므로 list_insert_ordered()함수에선 다음 들어갈 위치를 찾습니다. 따라서 우선순위가 같은 경우 레디 큐에 도착한 순서대로 정렬됩니다. 이를 위해 구현한 함수는 다음과 같습니다.

```
bool
cmp_priority (struct list_elem *x, struct list_elem *y, void *aux UNUSED)
{
    struct thread *th_a = list_entry (x, struct thread, elem);
    struct thread *th_b = list_entry (y, struct thread, elem);

    return th_a->priority > th_b->priority;
}
```

그림 2. 우선순위 비교를 위한 cmp_priority()함수, [thread/thread.c]

이를 바탕으로 경우 1, 경우 2에 관련된 함수들을 수정합니다. 수정해야 할 함수는 list_push_back()을 포함하는 thread_unblock(), thread_yield() 함수입니다.

```
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, cmp_priority, 0);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

그림 3. thread_unblock() 수정, [thread/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, cmp_priority, 0);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

그림 4. thread_yield() 수정, [thread/thread.c]

현재 실행중인 스레드의 우선순위가 바뀔 때, 레디 큐의 가장 높은 우선순위보다 낮다면 CPU를 넘겨주어야 합니다. 따라서 레디 큐의 맨 앞 스레드 priotity와 running 스레드 priority를 비교하고, 레디 큐의 스레드가 더 높은 priority를 가지면 CPU를 넘겨주는 함수를 구현합니다.

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

그림 5. Thread 선점을 위한 thread_preemption()함수 구현, [thread/thread.c]

CPU를 넘겨줄 수 있는 경우는 thread_create(), thread_set_priority() 이므로, thread_preemption()을 사용해 두 함수의 마지막 부분을 수정합니다.

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

그림 6, thread_create()함수의 마지막 부분 수정, [thread/thread.c]

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

  renew_priority ();
  thread_preemption ();
}
```

그림 7. thread_set_priority()함수 수정, [thread/thread.c]

renew_priority()함수와 **init_pri**는 Donation구현을 위한 코드로, 아래에서 자세히 다루겠습니다.

마지막으로, 경우3을 없애기 위해 thread_tick()함수를 수정합니다. Time slice만료 부분을 제거하면 해결됩니다.

```
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++;
}
```

그림 8. thread_tick()함수 수정, [thread/thread.c]

2) 하나의 공유자원을 사용하기 위해 여러 스레드가 대기하고 자원을 사용하던 스레드가 sema_up 할 때 어떤 스레드가 자원을 가져야 하는지에 대한 문제가 발생합니다. 이를 해결하기 위해 priority에 따라 자원을 할당받도록 합니다. 이전에 구현했던 cmp_priority()함수를 통한 list_insert_ordered()를 사용하면 해결할 수 있습니다. 현재 sema_down()은 list_push_back()을 사용하므로, 이를 list_insert_ordered()로 수정합니다.

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

그림 9. sema_down()함수 수정, [thread/synch.c]

waiters 리스트에 머무르는 동안 스레드들의 우선순위가 변경되었을 수도 있습니다. 따라서 sema_up()함수에서는 자원을 가질 때 waiters 리스트를 우선순위 내림차순으로 한번 정렬 후에 unblock될 수 있도록 list_sort()함수를 통해 코드를 수정합니다. 또한 unblock된 스레드가 현재 실행중인 스레드보다 우선순위가 높을 수 있으므로, 이전에 구현한 thread_preemption()함수를 이용해 선점할 수 있게 만들어줍니다.

```
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, cmp_priority, 0);
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;
  thread_preemption ();
  intr_set_level (old_level);
}
```

그림 10. sema_up()함수 수정, [thread/synch.c]

Lock에 관련된 함수인 lock_acquire(), lock_release()함수를 보면 함수 막바지에 sema_up(), sema_down()을 사용합니다. 이 두 함수는 이미 이전에 수정했으므로, 현재 상태에서 Lock관련 부분은 따로 처리해 줄 부분이 없습니다.

3) priority inversion을 해결하기 위해 priority donation을 구현합니다. 이를 위해 스레드가 priority를 양도받은 스레드들을 관리하고, 원래의 priority값을 저장할 수 있게 thread구조체를 변경합니다.

init_pri: 원래의 우선순위 값	waiting_lock: 스레드가 얻기 위해 기다리고 있는 lock
donations: priority를 양도받은 스레드들의 리스트	donation_elem: donations를 관리하기 위한 element

```
int init_pri;

struct lock *waiting_lock;
struct list donations;
struct list_elem donation_elem;
```

그림 11. Thread 구조체에서 추가한 항목, [thread/thread.h]

이후 init_thread()함수에서도 위 항목들을 초기화해줍니다.

```
t->init_pri = priority;
t->waiting_lock = NULL;
list_init (&t->donations);
```

그림 12. init_thread()함수 추가 코드, [thread/thread.c]

donations리스트를 priority 내림차순으로 정렬하기 위해 cmp_donate_priority()함수를 구현합니다. 이는 이전에 구현했던 cmp_priority()함수와 같은 방식입니다.

```
bool
cmp_donate_priority (const struct list_elem *x, const struct list_elem *y, void *aux UNUSED)
{
    const struct thread *th_a = list_entry (x, struct thread, elem);
    const struct thread *th_b = list_entry (y, struct thread, elem);

    return th_a->priority > th_b->priority;
}
```

그림 13. cmp_donate_priority()함수 구현, [thread/thread.c]

또한 스레드의 priority를 양도하기 위한 donate_priority()함수를 구현합니다. num_th는 priority donation이 무한정으로 이루어지는 것을 방지하기 위한 값입니다. 이 코드에선 10으로 설정했습니다. 현재 스레드의 waiting_lock이 NULL이 아니면 Lock을 기다리고 있는 상태의 스레드가 있다는 뜻입니다. 현재 Lock을 가지고 있는 holder에게 우선순위를 양도하는 방식을 대기 중인 lock을 가진 스레드를 따라가면서 10번 반복합니다. 이 과정에서 waiting_lock 이 NULL, 즉 Lock을 기다리는 thread가 없다면 donation을 할 필요가 없으므로 break를 수행합니다.

```
void
donate_priority (void)
{
  int num_th;
  struct thread *cur = thread_current ();

  for (num_th = 0; num_th < 10; num_th++){
    if (!cur->waiting_lock) break;
      struct thread *holder = cur->waiting_lock->holder;
      holder->priority = cur->priority;
      cur = holder;
  }
}
```

그림 14. donate_priority()함수 구현, [thread/thread.c]

현재 실행중인 스레드가 Lock을 요청하면, 요청한 스레드는 sema_down을 하기 전에 Lock을 가진 스레드에게 priority를 양도해야 합니다. 어떠한 스레드도 Lock을 가지고 있지 않다면 해당 Lock을 가집니다. 만약 lock을 가진 스레드가 있다면, 현재 스레드의 waiting_lock을 lock으로 만들어 Lock을 기다리는 상태로 만들어줍니다. 이후 donations 리스트에 우선순위 내림차순으로 현재 스레드를 추가하고, priority를 양도합니다. 이후 lock을 얻었다면, waiting_lock값을 NULL로 만들어줍니다. 더 이상 Lock을 기다릴 필요가 없기 때문입니다.

```
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->waiting_lock = lock;
    list_insert_ordered (&lock->holder->donations, &cur->donation_elem,
                         cmp_donate_priority, 0);
    donate_priority ();
  }

  sema_down (&lock->semaphore);

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

그림 15. lock_acquire()함수 수정, [thread/synch.c]

Priority Donation을 받은 스레드가 Lock을 반환할 때, 이 Lock을 사용하기 위해 priority를 빌려준 스레드를 donations 리스트에서 제거하고 우선순위를 재설정하는 작업이 추가로 필요합니다. 이를 위해 donations 리스트에서 스레드를 제거하는 함수와 Priority Donation을 받은 스레드의 priority를 재설정하는 함수를 구현합니다.

우선 현재 실행중인 스레드의 donations 리스트에서 스레드를 제거하는 함수인 del_lock()함수를 구현합니다. 이 함수는 현재 실행중인 스레드의 donations 리스트를 돌면서, t->waiting_lock이 주어진 lock과 동일한지 확인합니다. 만약 t->waiting_lock과 lock이 동일하다면, t가 현재 lock을 기다리고 있는 상태이므로, list_remove()함수를 통해 donations 리스트에서 제거합니다. del_lock()함수는 다음과 같습니다.

```
void
del_lock (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 *th = list_entry (e, struct thread, donation_elem);
    if (th->waiting_lock == lock)
      list_remove (&th->donation_elem);
  }
}
```

그림 16. del_lock()함수 구현, [thread/thread.c]

다음으로 Priority Donation을 받은 스레드의 priority를 재설정하는 renew_priority()함수를 구현합니다. 먼저 현재 실행중인 스레드의 우선순위를 원래의 우선순위로 변경합니다. 만약 donations리스트에 스레드가 남아있다면 남아있는 스레드들 중에서 가장 높은 우선순위 값을 가져옵니다. 이를 위해 cmp_donate_priority()함수를 이용한 list_sort()함수를 통해 우선순위 내림차순으로 donations리스트를 정렬하여, 가장 큰 우선순위를 가진 스레드가 donations리스트의 맨 앞에 올 수 있게 합니다. 이후 리스트 맨 앞의 priority가 현재 스레드의 priority보다 높다면, donations리스트 맨 앞 스레드의 우선순위를 현재 스레드의 우선순위로 만들어줍니다. 따라서 현재 스레드는 donations리스트가 비어있다면 최초의 우선순위를 가지게 될 것이고, 아닌 경우 리스트 맨 앞의 priority가 현재 스레드의 priority보다 높다면 donations리스트의 스레드들 중 가장 큰 우선순위를 가지게 될 것입니다.

```
void
renew_priority (void)
{
  struct thread *cur = thread_current ();

  cur->priority = cur->init_pri;

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

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

그림 17. renew_priority()함수 구현, [thread/thread.c]

이 두 함수를 통해 lock_release()함수를 수정합니다. Lock을 제거하고, 우선순위를 재설정하는 과정을 함수 중간에 추가하면 됩니다.

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

  del_lock (lock);
  renew_priority ();

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

그림 18. lock_release()함수 수정, [thread/synch.c]

만약 현재 실행중인 스레드의 최초우선순위(init_pri)가 donations 리스트에 있는 스레드들보다 높아진다면, donations 리스트의 가장 높은 우선순위가 아니라, 변경된 priority가 적용되어야 합니다. 이전에 구현했던 renew_priority()함수에서 최초우선순위를 사용하는 경우는 donations리스트가 비어있거나, 현재 스레드의 priority 값이 donations리스트 내에 가장 큰 priority값 보다 큰 경우입니다. 지금 같은 경우는 후자에 해당합니다. 따라서 thread_set_priority()함수에 renew_priority ()함수를 추가한다면 문제는 해결됩니다. 수정한 thread_set_priority()는 이전의 [그림 7]과 같습니다.

thread_get_priority()함수의 경우 현재 실행중인 스레드의 우선순위를 반환하는 함수로, 다음과 같습니다.

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

그림 19. thread_get_priority()함수, [thread/thread.c]

Simulation results

Priority Scheduling이 성공적으로 구현되었는지 pintos test를 통해 확인합니다. 결과는 다음과 같습니다.

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

그림 20. Test results