Operating System

Pintos Project #03

우선순위 스케줄러의 구현

컴퓨터 과학과

201710957

이유진

프로젝트 내용

<문제정의>

(1) 현재 pintos 에서는 라운드 로빈 스케쥴러가 구현되어 있다. 이를 수정하여 스레드 별 우선순위에 ᄄᆞ라 스케쥴링 할 수 있는 우선순위 스케쥴러를 새로 구현한다. 특히 preemptive dynamif prioirty scheduling 을 구현한다.

(2)우선순위 역전현상ㅇ을 방지하기 위해 proiroty inhenritance(donation) 기능을 구현한다.

(3)각 스레드가 자신의 우선순위를 확인하고 우선순위를 변경할 수 있도록 두가지 함수 void thread_priority(int new_priority) 와 int thread_get_priority()를 구현한다.

<테스트>

구현된 우선순위 스케줄러가 제대로 동작하는지 여부는 테스트

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

를 사용하여 시험한다.

우선순위 스케줄링과 관련된 테스트 중 priority-fifo 와 priority-convar 에 대해서는 시험하지 않는다.

수정한 소스코드 및 테스트 결과

<수정한 소스코드>

struct thread /thread.h

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

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

    //for priority scheduling
    int init_priority;
    struct lock *wait_on_lock;
    struct list donations;
    struct list_elem donation_elem;



#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

thread 의 자료구조 안에 int 형 변수 init_priority, lock 변수인 wait_on_lock, 우선순위 상속에 대해 우선순위를 저장할 donation 리스트, 상속과 관련된 원소를 넣을 donation_elem 리스트를 선언하였다.

define depth / thread.c

```
#define DEPTH_LIMIT 8
```

뒤에

init_thread / thread.c

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;
  list_push_back (&all_list, &t->allelem);

  t->init_priority = priority;
  t->wait_on_lock = NULL;
  list_init(&t->donations);
}
```

t->init_priorty 를 이용하여 priority 로 초기화한다.

t->wait_on_lock 을 NULL 로 초기화한다.

t 의 donations 라는 리스트를 초기화한다.

test_max_priority / thread.c

```
void test_max_priority(void){
  if(list_empty(&ready_list))
    return;

  struct thread *t = list_entry(list_front(&ready_list),struct thread, elem);
  if(intr_context())
  {
    thread_ticks++;
    if(thread_current()->priority < t->priority || (thread_ticks >=TIME_SLICE && thread_current()->priority == t->priority)){
      intr_yield_on_return();
    }
    return;
  }
  if(thread_current()->priority < t->priority)
    thread_yield();

}
```

ready queue 에 들어있는 thread 와 현재 실행중인 thread 를 비교하는 함수이다.

현재 실행중인 thread 의 우선순위보다 readyqueue 에 있는 thread 가 우선순위가 높으면 thread_yield 를 이용하여 실행할 스레드를 교체한다.

remove_with_lock / thread.c

```
void remove_with_lock(struct lock *lock){
  struct list_elem *e = list_begin(&thread_current()->donations);
  struct list_elem *next;
  while( e!= list_end(&thread_current()->donations))
  {
    struct thread *t = list_entry(e, struct thread, donation_elem);
    next = list_next(e);
    if(t->wait_on_lock == lock)
      list_remove(e);
    e = next;
  }
}
```

현재 실행중인 스레드의 donations 라는 리스트에서 반복을 하면서 donation element 인 스레드의 wait_on_lock 과 lock 이 같으면 리스트에서 제거하는 함수이다.

priority_cmp / thread.c

```
bool priority_cmp(const struct list_elem *a,const struct list_elem *b, void *aux UNUSED){
  struct thread *aa = list_entry(a, struct thread, elem);
  struct thread *bb = list_entry(b, struct thread, elem);

  if(aa->priority > bb->priority)
    return true;

  else
    return false;


}
```

priority 를 기준으로 insert 하기 위해서 만든 비교함수 이다.

thread_create() / thread.c

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux) |
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  enum intr_level old_level;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Prepare thread for first run by initializing its stack.
     Do this atomically so intermediate values for the 'stack'
     member cannot be observed. */
  old_level = intr_disable ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;
```

```
  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  intr_set_level (old_level);

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

  test_max_priority();

  return tid;
}
```

새로 생성된 스레드는 또한 우선순위를 확인해봐야 하기 때문에 test_max_priorty()를 이용하여 새로 생우선순위를 확인하는 코드를 추가하였다.

thread_unblock() / 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, priority_cmp, (void*)NULL);
  t->status = THREAD_READY;

  intr_set_level (old_level);
}
```

readylist 에 우선순위를 기준으로 정렬된상태를 유지하기위해서, list_insert_ordered 함수를 이용하여 삽입하였다.

thread_yield() / 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, (void*)NULL);
   //list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

thread_yield 함수 또한 readylist 에 우선순위를 기준으로 정렬된상태를 유지하기위해서, list_insert_ordered 함수를 이용하여 삽입하였다.

donate_priority() / thread.c

```
void donate_priority(void){
  int depth = 0;
  struct thread *t = thread_current();
  struct lock *l = t->wait_on_lock;
  while(l && depth<DEPTH_LIMIT)
  {
    depth++;
    if(!l->holder)
      return;
    if(l->holder->priority >= t->priority)
      return;

    l->holder->priority = t->priority;
    t = l->holder;
    l = t->wait_on_lock;
  }
}
```

우선순위 역전현상이 일어났을 때 우선순위상속을 위한 함수이다.

depth 가 DEPTH_LIMIT 보다 작고, l 이 true 일 때 반복문을 돌게된다.반복문을 돌면서 우선순위 상속을 진행하는데, 이 반복문 안에서 l->holder 에 아무것도 없거나 lock 의 홀더의 우선순위가 현재 우선순위보다 높은 경우에는 우선순위 상속이 필요 없기 때문에 도중에 함수를 마친다.

우선순위 상속이 반복문 안에서 돌기 때문에, multiple donation 과 nested donation 또한 처리된다.

refresh_priority() / thread.c

```
void refresh_priority(void){
  struct thread *t = thread_current();
  t->priority = t->init_priority;

  if(list_empty(&t->donations))
    return;

  struct thread *s = list_entry(list_front(&t->donations), struct thread,donation_elem);

  if(s->priority > t->priority)
    t->priority = s->priority;
}
```

우선순위 상속 마친 후 원래의 우선순위로 다시 돌리는 작업이다.

현재 스레드의 우선순위를 초기화 시키고 우선순위를 donations 리스트의 우선순위로 변경한다.

lock_aquire / sync.c

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

  enum intr_level old_level = intr_disable();

  if(!thread_mlfqs && lock->holder)
  {
    thread_current() -> wait_on_lock = lock;
    list_insert_ordered(&lock->holder->donations, &thread_current()->donation_elem,priority_cmp,NULL);
  }

  sema_down (&lock->semaphore);

  thread_current()->wait_on_lock = NULL;

  lock->holder = thread_current ();
  intr_set_level(old_level);

}
```

lock 을 요청하는 이 함수에서, 우선 intr_disable()을 통해 interrupt 를 멈춘다.

파라미터인 lock 의 holder 가 비어있지 않았다면, 현재 실행중인 스레드의 wait_on_lock 에 lock 을 저장하고 우선순위를 저장하는 donations 에 우선순위 순서대로 삽입한다.

lock 의 holder 을 현재 스레드로 바꾼 후, interrupt level 을 복구 시킨 후 마무리한다.

lock_try_aquire

```
bool
lock_try_acquire (struct lock *lock)
{
  bool success;

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

  enum intr_level old_level = intr_disable();

  success = sema_try_down (&lock->semaphore);
  if (success){
    thread_current()->wait_on_lock = NULL;
    lock->holder = thread_current ();
  }
  intr_set_level(old_level);
  return success;
}
```

lock_acquire 과 같은 맥락으로, lock 을 얻었다면(sucess) 현재 스레드의 wait_on_lock 을 NULL 로 초기화 시킨후, lock 의 holder 에 현재 스레드를 넣는다.

lock_release / sync.c

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

  enum intr_level old_level = intr_disable();

  lock->holder = NULL;

  if(!thread_mlfqs){
    remove_with_lock(lock);
    refresh_priority();
  }
  sema_up (&lock->semaphore);
  intr_set_level(old_level);
}
```

위와 동일한 방법으로 interrupt 를 제어한상태로, lock 의 holder 을 NULL 로 초기화 해주고, thread_mifqs 가 false 일 때 remove_with_lock 을 실행하고, refresh_priority()를 통하여 우선순위를 복구한다.

sema_down / sync.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)
    {
      donate_priority();
      list_insert_ordered(&sema->waiters, &thread_current ()->elem,priority_cmp,NULL);
      //list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

기존 sema_down 함수에서 sema->waiters 리스트에 우선순위 순서로 삽입하기 위해 list_insert_ordered 를 이용하여 삽입했다

sema_up() / sync.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,priority_cmp,NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));

  }
  sema->value++;

  if(!intr_context())
    test_max_priority();


  intr_set_level (old_level);
}
```

sema 의 waiters 리스트에 원소가 있다면 리스트를 우선순위순서대로 sorting 한 후, waiters 리스트의 첫 원소(우선순위가 가장높은 원소)를 unblock 한다.

timer_interrupt /timer.c

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_check();
  thread_tick ();

  test_max_priority();

}
```

timer_interrupt 가 일어날때마다, ready queue 와 실행 스레드의 우선순위를 비교해야하기 때문에 test_max_priority() 함수를 실행한다.

<테스트 결과>


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