PintOS Project – Thread scheduling

1. 프로젝트 개요

Pintos에 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 목표

현재 Pintos의 스케줄러는 Rr방식으로 구현 되어있다. 이를 우선순위를 고려하여 스케줄링 하도록 수정한다.

2-1) Ready list에 새로 추가된 thread의 우선순위가 현재 CPU를 점유중인 thread의 우선순위보다 높으면, 기존 thread를 밀어내고 CPU를 점유하도록 한다.

2-2) 우선순위 스케줄링 방식의 문제점은 우선순위 역전현상이 일어날 수 있다는 것이다. 이를 방지할 수 있도록 우선순위 스케줄러에 priority inheritance(donation) 기능을 구현한다.

3. 추가된 함수 & 수정된 함수

3-1) thread.c 추가된함수

- compare_priority(): 두 thread의 우선 순위를 비교하는 함수로, 준비 리스트의 thread를 우선 순위에 따라 정렬하기 위해 사용된다.
- sort_ready_list(): 준비 중인 thread 목록을 우선 순위에 따라 정렬하는 함수.
- search_array(): 현재 thread의 우선 순위 배열에서 특정 우선 순위를 검색하여 해당 우선 순위를 배열에서 제거하고 현재 우선 순위를 변경하는 함수.

3-2) thread.c 수정된 함수

- thread_create(): thread 생성 함수로, 우선 순위에 따라 생성된 thread를 준비 리스트에 추가한다.
- thread_yield(): 현재 실행 중인 thread를 준비 리스트에 추가하여 실행 가능한 상태로 전환하는 함수.
- thread_set_priority(): 현재 thread의 우선 순위를 변경하는 함수.

- thread_get_priority(): 현재 thread의 우선순위를 리턴 받는 함수.
- thread_unblock(): 차단된 thread를 준비 리스트에 추가하여 실행 가능 상태로 전환하는 함수.

3-3) synch.c 추가된함수

- compare_priority(): 우선 순위를 비교하는 함수. (공통)
- sort_ready_list(): 준비 중인 thread 목록을 우선 순위에 따라 정렬하는 함수. (공통)

3-4) synch.c 수정된함수

- sema_down(): thread의 우선 순위를 고려하여 Semaphore의 값을 확인하고 thread를 블록하는 부분이 수정되었다.
- sema_up(): 대기 중인 thread를 깨우고 우선 순위에 따라 스케줄링하는 부분이 추가되었다.
- lock_acquire(): thread의 우선 순위를 고려하여 lock을 획득하는 부분이 수정되었다.
- lock_release(): thread의 우선 순위를 고려하여 lock을 해제하는 부분이 수정되었다.

3-5) thread.h 수정

```
struct thread
  {
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;
    int priorities[9];
    int size;
    struct list_elem allelem;
    int64_t wakeup_time;
    int donation_no;
    struct lock *waiting_for;

    struct list_elem elem;
```

int priority

이전의 struct thread 구조체에 priority 필드를 추가하였다. 이 필드는 thread의 우선 순위를 나타낸다.

int priorities[9] 및 int size

donated priority list와 size 필드를 추가하였다. 이 필드들은 기부된 우선 순위 목록과 목록의 크기를 저장한다.

int donation_no: donation_no

필드를 추가하였다. 이 필드는 thread가 기부한 lock의 수를 저장한다.

struct lock *waiting_for

waiting_for 필드를 추가하였다. 이 필드는 thread가 Block된 상태에서 대기 중인 lock을 나타낸다.

```
extern bool thread_mlfqs;

void thread_init (void);
void thread_start (void);

void thread_tick (void);
void thread_print_stats (void);

typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);

void thread_block (void);
void thread_unblock (struct thread *);

struct thread *thread_current (void);
tid_t thread_tid (void);
const char *thread_name (void);

void thread_exit (void) NO_RETURN;
void thread_yield (void);

typedef void thread_action_func (struct thread *t, void *aux);
void thread_foreach (thread_action_func *, void *);

int thread_get_priority (void);
void thread_set_priority (int);

int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
bool compare_priority(struct list_elem *l1, struct list_elem *l2, void *aux);
void sort_ready_list(void);
void search_array(struct thread *cur,int elem);

#endif /* threads/thread.h */
```

compare_priority() 함수를 선언하였다. 이 함수는 thread 우선 순위를 비교하여 정렬에 사용된다.

sort_ready_list() 함수를 선언하였다. 이 함수는 실행 가능한 thread 목록을 우선 순위에 따라 정렬한다.

3-6) synch.h 수정

```
struct semaphore
  {
    unsigned value;
    struct list waiters;
  };

void sema_init (struct semaphore *, unsigned value);
void sema_down (struct semaphore *);
bool sema_try_down (struct semaphore *);
void sema_up (struct semaphore *);
void sema_self_test (void);

/* Lock. */
struct lock
  {
    struct thread *holder;
    struct semaphore semaphore;
    bool is_donated;
  };

void lock_init (struct lock *);
void lock_acquire (struct lock *);
bool lock_try_acquire (struct lock *);
void lock_release (struct lock *);
bool lock_held_by_current_thread (const struct lock *);


struct condition
  {
    struct list waiters;
  };

void cond_init (struct condition *);
void cond_wait (struct condition *, struct lock *);
void cond_signal (struct condition *, struct lock *);
void cond_broadcast (struct condition *, struct lock *);

bool compare_sema(struct list_elem *l1, struct list_elem *l2,void *aux);
```

struct lock 구조체

bool is_donated: 이전의 struct lock 구조체에 추가하였다. lock이 thread에 danate되었는지 여부를 나타낸다.

sema_down() 함수

compare_priority() 함수를 호출하여 대기 중인 thread를 우선 순위에 따라 정렬하는 부분을 추가하였다.

sema_up() 함수

대기 중인 thread를 깨우기 전에 list_sort() 함수를 호출하여 대기 중인 thread를 우선 순위에 따라 정렬하는 부분을 추가하였다.

lock_acquire() 함수

lock을 획득하기 전에 우선 순위를 고려하여 thread의 대기 상태 및 우선 순위를 변경하는 부분을 추가하였다.

lock_release() 함수

lock을 해제한 후에 대기 중인 thread를 깨울 때 우선 순위에 따라 정렬하는 부분을 추가하였다.

4. 코드 설명

compare_priority():

```
bool compare_priority(struct list_elem *l1, struct list_elem *l2,void *aux)
{
  struct thread *t1 = list_entry(l1,struct thread,elem);
  struct thread *t2 = list_entry(l2,struct thread,elem);
  if( t1->priority > t2->priority)
    return true;
  return false;
}
```

이 함수는 두 개의 list_elem 포인터를 인자로 받아 해당 thread의 우선 순위를 비교하여 정렬하는 함수이다. 반환 값은 true 또는 false로, l1의 우선 순위가 l2의 우선 순위보다 높은지를 나타낸다. 우선 순위 스케줄링에서 실행 가능한 thread를 우선 순위에 따라 정렬하는 데 사용된다.

sort_ready_list():

```
/*Sorts the ready_list present in thread.c*/
 void sort_ready_list(void)
{
  list_sort(&ready_list, compare_priority, 0);
}
```

이 함수는 실행 가능한 thread 목록인 ready_list를 우선 순위에 따라 정렬하는 함수이다. list_sort() 함수를 사용하여 ready_list를 compare_priority() 함수를 기준으로 정렬한다. 우선 순위 스케줄링에서 실행 가능한 thread 목록을 우선 순위에 맞게 재정렬하는데 사용된다.

search_array():

```
void search_array(struct thread *cur,int elem)
{ int found=0;
  for(int i=0;i<(cur->size)-1;i++)
  {
  if(cur->priorities[i]==elem)
    {
     found=1;
    }
  if(found==1)
    {
     cur->priorities[i]=cur->priorities[i+1];
    }
  }
  cur->size -=1;
}
```

이 함수는 주어진 thread의 우선 순위 목록인 priorities에서 특정 원소를 검색하여 삭제하는 역할을 한다. 우선 순위 목록을 순회하면서 원소를 검색하고, 원소를 찾으면 해당 원소 이후의 원소들을 한 칸씩 앞으로 이동시키고, 마지막으로 size 값을 감소시켜 원소가 삭제된 것을 반영한다. 우선 순위 기부에서 기부된 우선 순위를 검색하여 삭제하는 데 사용된다.

thread_create():

```
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;

  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 ();

  /* 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;

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

  return tid;
}
```

이 함수는 주어진 이름, 우선 순위, 함수 포인터 및 보조 인자를 사용하여 새로운 thraed를 생성한다. thread를 할당하고 초기화한 후, 필요한 스택 프레임을 설정하고 실행 대기 큐에 추가하여 thread를 실행 가능 상태로 만든다. 이 함수는 새로운 thread를 생성하여 실행 가능 상태로 전환하는 데 사용된다. 생성된 thread의 tid를 반환하며, thread 생성에 실패한 경우 TID_ERROR 값을 반환한다.

thread_yield():

```
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, compare_priority, 0);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

이 함수는 현재 실행 중인 thread를 양보하고 스케줄링을 수행한다. 인터럽트 컨텍스트에서 호출되지 않도록 ASSERT() 매크로를 사용하여 확인한다. 함수 내부에서는 현재 thread를 실행 가능 큐에 삽입한 후, thread의 상태를 THREAD_READY로 변경하고 스케줄링을 수행한다. 스케줄링은 schedule() 함수를 호출하여 이루어진다. 현재 thread를 실행 가능 큐에 삽입하고 스케줄링을 통해 다음 thread를 선택하는 데 사용된다.

thread_set_priority()

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current()->priorities[0] = new_priority;
  if(thread_current()->size==1)
  {
    thread_current ()->priority = new_priority;
    thread_yield();
  }
}
```

이 함수는 현재 thread의 우선 순위를 변경한다. 함수 내부에서는 현재 thread의 우선 순위 목록(priorities)의 첫 번째 원소를 변경하고, 만약 우선 순위 목록의 size가 1인 경우, thread의 우선 순위도 변경하고 thread를 양보한다. 현재 thread의 우선 순위를 변경하여 thread 스케줄링에 영향을 준다.

thread_get_priority():

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

이 함수는 현재 thread의 우선 순위를 리턴한다.

thread_unblock():

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

이 함수는 블록된 thread를 Unblock하고 실행 가능 상태로 전환한다. 함수 내부에서는 thread의 유효성을 확인하고, 인터럽트를 비활성화한 후 thread의 상태를 검사한다. 그리고 실행 가능 큐에 적절한 위치에 thread를 삽입하여 Unblock한다. thread의 상태도 THREAD_READY로 변경한다. 블록된 thread를 Unblock하여 실행 가능 상태로 전환하는 데 사용된다.

sema_down():

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

이 함수는 Semaphore를 내리는 동작을 수행한다. 함수 내부에서는 Semaphore의 유효성을 확인하고, 인터럽트를 비활성화한 후 Semaphore의 값을 확인한다. 값이 0인 경우, 현재 thread를

대기 리스트(waiters)에 적절한 위치에 삽입한 후 Block 한다. thread는 Semaphore 값이 양수가 될 때까지 대기 상태로 유지된다. 이후에 Semaphore 값이 감소하고, 인터럽트 레벨을 복원하여 원래의 인터럽트 상태를 되돌린다. Semaphore 값을 내리는 동작을 수행하고, thread가 필요한 자원을 기다리도록 하는 데 사용된다.

sema_up():

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    while (!list_empty (&sema->waiters)) {
        thread_unblock(list_entry(list_pop_front(&sema->waiters), struct thread, elem));
    }
  }

  sema->value++;
  thread_yield();


  intr_set_level (old_level);

}
```

이 함수는 Semaphore를 올리는 동작을 수행한다. 함수 내부에서는 Semaphore의 유효성을 확인하고, 인터럽트를 비활성화한 후 대기 중인 thread가 있는지 확인한다. 대기 중인 thread가 있다면, 대기 리스트(waiters)를 우선 순위에 따라 정렬한 후 가장 우선 순위가 높은 thread를 깨운다. 이후, 대기 중인 thread가 다시 실행 가능 상태로 전환된다.
Semaphore 값을 올리고, 대기 중인 thread를 깨워 실행 가능 상태로 전환하는 데 사용된다.

lock_acquire():

```
void
lock_acquire (struct lock *lock)
{

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

  if(lock->holder != NULL)
  {
    thread_current()->waiting_for= lock;
    if(lock->holder->priority < thread_current()->priority)
    {  struct thread *temp=thread_current();
       while(temp->waiting_for!=NULL)
       {
         struct lock *cur_lock=temp->waiting_for;
         cur_lock->holder->priorities[cur_lock->holder->size] = temp->priority;
         cur_lock->holder->size+=1;
         cur_lock->holder->priority = temp->priority;
         if(cur_lock->holder->status == THREAD_READY)
           break;
         temp=cur_lock->holder;
       }
       if(!lock->is_donated)
         lock->holder->donation_no +=1;
       lock->is_donated = true;
       sort_ready_list();
    }
  }
  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
  lock->holder->waiting_for=NULL;
}
```

이 함수는 lock을 획득하는 동작을 수행한다. 함수 내부에서는 lock의 유효성을 확인하고, 현재 컨텍스트가 인터럽트 컨텍스트가 아님을 확인하며, 현재 thread가 해당 lock을 소유하지 않는지 확인한다. lock이 이미 다른 thread에 의해 소유되어 있는 경우, 현재 thread는 해당 lock을 기다리는 상태로 변경되고, 이후 우선 순위 donate 로직을 수행한다. 기다리는 동안 lock에 의해 기부된 thread들의 우선 순위를 업데이트하고, 기부가 이루어진 경우 기부 횟수를 증가시키고 실행 가능한 thread 리스트를 재정렬한다. 마지막으로, sema_down() 함수를 사용하여 lock을 획득하기 위해 대기하고, lock을 획득한 thread를 설정하고 기다리는 상태를 초기화한다.
lock을 획득하여 실행 가능한 상태로 전환하는 데 사용된다.

lock_release():

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

  struct semaphore *lock_sema=&lock->semaphore;
  list_sort(&lock_sema->waiters, compare_priority, 0);

  if (lock->is_donated)
  {
    thread_current()->donation_no -=1;
    int elem=list_entry (list_front (&lock_sema->waiters),struct thread, elem)->priority;
    search_array(thread_current(),elem);
    thread_current()->priority = thread_current()->priorities[(thread_current()->size)-1];
    lock->is_donated = false;
  }
  if(thread_current()->donation_no ==0)
  {
    thread_current()-> size=1;
    thread_current()-> priority = thread_current()->priorities[0];
  }
  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

이 함수는 lock을 해제하는 동작을 수행한다. 함수 내부에서는 lock의 유효성을 확인하고, 현재 thread가 해당 lock을 소유하고 있는지 확인한다. lock에 기부된 우선 순위가 있는 경우, 현재 thread의 기부 횟수를 감소시키고, 기부된 thread 리스트에서 가장 높은 우선 순위를 가져와서 현재 thread의 우선 순위를 업데이트하고, 기부 플래그를 false로 설정한다. 만약 현재 thread의 기부 횟수가 0인 경우, 현재 thread의 크기를 1로 설정하고, 우선 순위를 기부된 thread 중 가장 높은 우선 순위로 설정한다. 마지막으로 lock의 소유자를 NULL로 설정하고, sema_up() 함수를 사용하여 lock의 Semaphore를 해제한다.

5. 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
```

구현된 우선순위 스케줄러가 제대로 동작하는지 여부 테스트

priority-change - pass

priority-preempt - pass

priority-sema - pass

priority-donate-one - pass
priority-donate-multiple - pass
priority-donate-multiple2 - pass
priority-donate-nest - pass
priority-donate-chain - pass
prioritydonate-sema - pass
priority-donate-lower – pass

시험결과. 우선순위 스케줄링과 관련된 테스트 모두 통과.

priority-fifo – pass의 경우 시험하지 않는다고 했지만 기능을 구현하니 pass가 나올 수 밖에 없었음.
priority-convar - Fail 시험하지 않는 파트이므로 상관X