OS Project 2 - Team 4

20185083 서영석, 20185141 이준명, 20185010 권강민

1. Introduction

This assignment involves implementing thread priority and integrating it with lock, semaphore, and condition functionalities. To solve the inverse priority problem that can occur when using locks, priority donation has been implemented to efficiently control resource distribution in a multi-threaded environment.

2. Overview

A. Problem 1 : Priority Scheduling

The first goal is to execute threads in priority order. When multiple threads are executed simultaneously, there may be threads that need to be executed first and others that need to be executed later. Therefore, it is necessary to add a priority to each thread and make them execute in that order.

To solve the above task, we implemented Priority scheduling using a `list_max` function. Additionally, when a thread is created and enters the ready queue or when the priority of an existing thread is updated, we call the `thread_yield` function to ensure that the thread with the highest priority in the ready queue is executed.

B. Problem 2 : Priority Donation

Priority Donation is one of the techniques used to address the Priority Inversion Problem that can occur in competition for shared resources such as mutexes, semaphores, and condition variables.

The priority inversion problem refers to the phenomenon where a process with a high priority is executed later than a process with a lower priority. This typically occurs due to competition for access to shared resources.

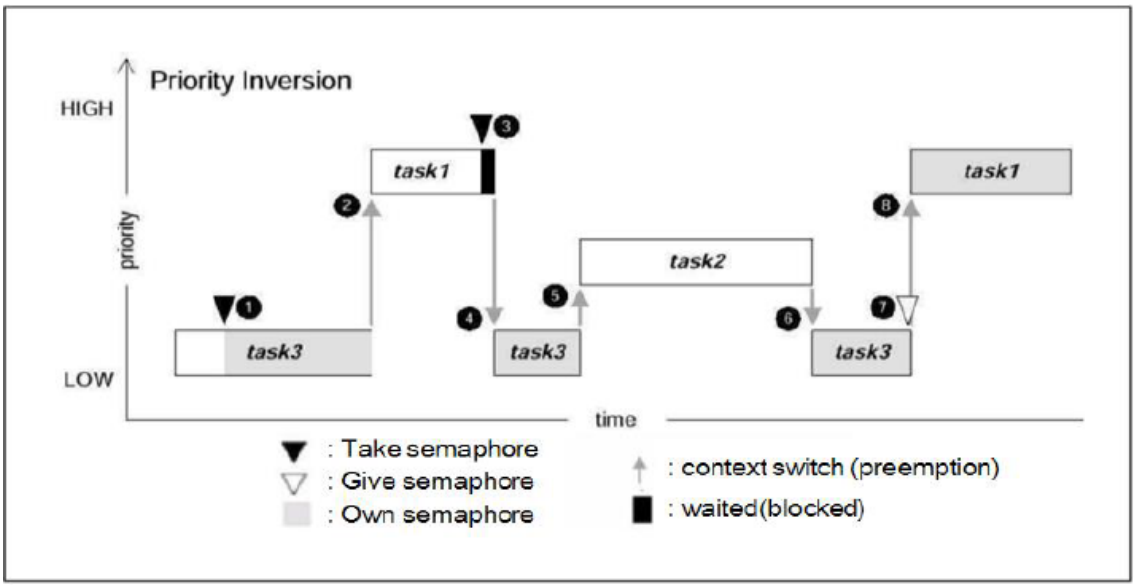


Figure 1. Priority Inversion Problem Occurs

For example, if a high-priority process is waiting for a lock to be released, but a low-priority process already holds the lock and is waiting for it to be released, the high-priority process will continue to be blocked, causing a priority inversion problem. The priority inversion problem can be problematic in systems where priority scheduling is important, such as real-time systems. Techniques such as priority donation are used to solve this problem.

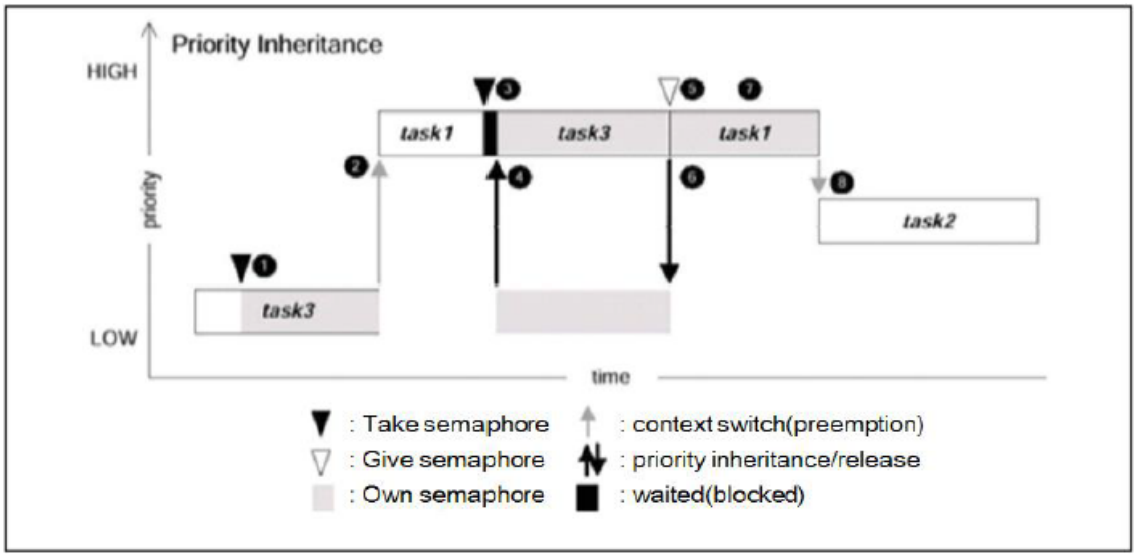


Figure 2. Using Priority Donation (http://blog.skby.net/우선순위-역전-현상/)

To address this problem, we can raise the priority of the low-priority process that is holding the lock waited by a higher-priority process to the same priority as the higher-priority process. By doing so, the low-priority process can continue to execute without being blocked and can perform its tasks effectively. This priority donation technique is widely used in many operating systems and plays a crucial role in ensuring real-time performance and efficiency.

3. Code-level development

We modified the `thread.c` and `synch.c` files, and its header files `thread.h` and `synch.h`. + is an additional line, and - is deleted line. Some unimportant changes were not noted.

A. Problem 1 : Priority Scheduling

A priority queue was used to implement Priority scheduling. Although it could have been implemented sufficiently using the list_insert_ordered function and pop_back function**,** in order to operate in FIFO order when priorities are the same the thread with the maximum priority was obtained from the `ready_list` using the `list_max` function. Then, control was passed to that thread.

- `threads/thread.c`

```
static struct thread * next_thread_to_run (void)
 {
   if (list_empty (&ready_list))
     return idle_thread;
-  //else
-    //return list_entry (list_pop_front (&ready_list), struct thread, elem);
+  else{
+    struct list_elem * max_element = list_max(&ready_list,compare_thread_priority,NULL);
+    list_remove(max_element);
+    struct thread * max_thread = list_entry(max_element,struct thread, elem);
+    return max_thread;
+  }
 }
```

- `threads/thread.c`

```
bool compare_thread_priority(const struct list_elem *a, const struct list_elem *b,
void * aux UNUSED){
  struct thread * a_thread = list_entry(a,struct thread,elem);
  struct thread * b_thread = list_entry(b,struct thread,elem);
  return a_thread->priority < b_thread->priority;
}
```

B. Problem 2 : Priority Donation

We modified the `thread.c` and `synch.c` files, along with their corresponding header files, in order to implement priority donation

1. thread structure modified

- `threads/thread.h`

```
struct thread
  {
    tid_t tid;
```

```
    enum thread_status status;
    char name[16];
    uint8_t *stack;
+   int original_priority;//added
    int priority;
    struct list_elem allelem;
    struct list_elem elem;
-   //struct list_elem blockelem   <-- removed
+   struct lock * waiting_lock;
+   struct list lock_list; //added
    int64_t wait_start_time;
    int64_t wait_ticks;

#ifdef USERPROG
    uint32_t *pagedir;
#endif
    unsigned magic;
  };
```

- `threads/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->original_priority = priority;
  t->priority = priority;
  t->magic = THREAD_MAGIC;
+ list_init(&t->lock_list);
  list_push_back (&all_list, &t->allelem);
}
```

To implement priority donation, three elements were added to the `thread` structure:

1. `original_priority`: the priority of the thread before any priority donation occurs.
2. `waiting_lock`: the lock that the thread is currently waiting on.
3. `lock_list`: a list of locks that the thread is holding.

2. Priority Donation

- `threads/synch.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(!sema_try_down(&lock->semaphore)){
+   thread_current()->waiting_lock = lock;
    sema_down (&lock->semaphore);
  }
+ list_push_back(&thread_current()->lock_list,&lock->elem);
+ thread_current()->waiting_lock = NULL;
  lock->holder = thread_current ();
+ bool is_priority_change = lock_update_priority(lock);
+ intr_set_level(old_level);
+ if(is_priority_change)
+   thread_yield();
}
```

When a thread calls the `lock_acquire` function to acquire a resource, a process checks whether semaphore value can be lowered. If the semaphore value is 0, the current thread is added to the waiting list of the semaphore and it waits. When it can acquire the semaphore, the lock's holder is changed to the current thread and the `lock_update_priority` function is called.

- `threads/synch.c`

```
void
lock_update_priority (struct lock* lock){
    if(lock->holder){
        thread_update_priority(lock->holder);
    }
}
```

When `lock_acquire` is called, a thread that holds the lock exists, so `lock_update_priority` function calls `thread_update_priority` function. The `thread_update_priority` function works as follows:

1. First, it examines the waiters (`lock_list` in thread structure) of the locks held by `m_thread`, obtains the highest priority, and compares it with the original priority of `m_thread`. If the maximum priority found is greater than the original priority, `m_thread` performs priority donation to itself. (This process is performed by calling `thread_get_lock_maximum_donation`.)
2. It then checks the holder of `waiting_lock` of thread structure. If the priority of each lock holder is lower than the priority of `m_thread`, it performs priority donation to the holder.

- `threads/thread.c`

```
...
bool
compare_lock_priority_donation(struct list_elem * a, struct list_elem * b,
 void * aux UNUSED){
    struct lock *lock_a = list_entry(a,struct lock, elem);
    struct lock * lock_b = list_entry(b,struct lock, elem);
    int a_sem = sema_get_maximum_priority(&lock_a->semaphore);
    int b_sem = sema_get_maximum_priority(&lock_b->semaphore);
    return a_sem < b_sem;
}
```

```
...

int
thread_get_lock_maximum_donation(struct thread* m_thread){
    struct list * locklist = &m_thread->lock_list;
    int max_donation = -1;//if there is no lock_maximum_donation, return -1.
    if(list_empty(locklist)){
        max_donation = -1;
    }
    else{
        struct list_elem * max_lock_elem = list_max(locklist,
                                                    compare_lock_priority_donation,NULL);
        struct lock * max_lock = list_entry(max_lock_elem,struct lock, elem);
        max_donation = sema_get_maximum_priority(&max_lock->semaphore);
    }
    return max_donation;
}
...

bool thread_update_priority(struct thread * m_thread){//return true if there is a change
    //change current thread's priority
    int thread_prev_priority = m_thread->priority;
    int thread_original_priority = m_thread->original_priority;
    int donated_priority = thread_get_lock_maximum_donation(m_thread);
    int max_priority =
donated_priority > thread_original_priority ? donated_priority : thread_original_priority;
    m_thread->priority = max_priority;
    if(thread_prev_priority == m_thread->priority){
        return false;
    }
    //apply priority change to other threads
    struct lock * waiting_lock = m_thread->waiting_lock;
    if(waiting_lock != NULL){
        lock_update_priority(waiting_lock);
    }
    return true;
}
```

- `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_push_back(&sema->waiters,&thread_current()->elem);
+     if(thread_current()->waiting_lock){
+       lock_update_priority(thread_current()->waiting_lock);
+     }
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

In the `sema_down` function, the thread is added as a waiter, and if the thread has a waiting lock, the lock's priority is updated because priority donation may occur.

- `threads/thread.c`

```
void thread_set_priority (int new_priority)
 {
-    //thread_current ()->priority = new_priority;
+    struct thread * target_thread = thread_current();
+    thread_set_priority_of(new_priority, target_thread);
 }

+void thread_set_priority_of(int new_priority, struct thread * m_thread){
+    m_thread->original_priority = new_priority;
+    if(thread_update_priority(m_thread)){
+        thread_yield();
+    }
+}
```

When the original priority of a thread is changed through the `thread_set_priority` function, the final priority must be determined by comparing it with the donated priority. Therefore, `thread_update_priority` is called, and if the priority changes as a result of this update, `thread_yield` is called.

3. Lock Release

- `threads/synch.h`

```
struct lock
{
     struct thread *holder;      /* Thread holding lock (for debugging). */
     struct semaphore semaphore; /* Binary semaphore controlling access. */
+    struct list_elem elem;
};
```

When the `lock_release` function is called, the following actions occur:

1. The holder of the lock is released. The lock is removed from the holder's lock list using the `list_remove` function.
2. Priority donation is undone and the original priority is restored. This is done by calling `thread_update_priority`.
3. The semaphore corresponding to the lock is incremented by 1, and the thread with the highest priority in the `waiters` list is unblocked.

- `threads/synch.c`

```
void
lock_release (struct lock *lock)
```

```
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
+ enum intr_level old_level = intr_disable();
// make holder to NULL
+ struct thread * holder = lock->holder;
+ list_remove(&lock->elem);
  lock->holder = NULL;
// revert thread priority to the original state
+ thread_update_priority(holder);
  sema_up (&lock->semaphore);
+ intr_set_level(old_level);
}

....

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  sema->value++;
- //if (!list_empty (&sema->waiters))
- //  thread_unblock (list_entry (list_pop_front (&sema->waiters),
- //                              struct thread, elem));
+ if (!list_empty (&sema->waiters)){
+   struct list_elem * max_elem = list_max(&sema->waiters,compare_thread_priority,NULL);
+   list_remove(max_elem);
+   struct thread * max_thread = list_entry(max_elem,struct thread,elem);
+   thread_unblock(max_thread);
+ }
  intr_set_level (old_level);
+ thread_yield();
}
```

C. Conditional variable

`cond_signal` function has been modified to signal in order of highest priority when `cond_signal` function is called. `cond_signal` was designed to give signal to the first waiter(semaphore). It was changed to give signal in order of priority using the `list_max` function.

- `threads/synch.c`

```
void cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));
-// if (!list_empty (&cond->waiters))
-//   sema_up (&list_entry (list_pop_front (&cond->waiters),
-//                         struct semaphore_elem, elem)->semaphore);
+ if (!list_empty (&cond->waiters)){
+   struct list_elem * popped_list_elem =
                      list_max(&cond->waiters,compare_sema_elem_by_priority,NULL);
+   list_remove(popped_list_elem);
+   struct semaphore_elem * popped_sema_elem =
```

```
                      list_entry(popped_list_elem,struct semaphore_elem,elem);
+   struct semaphore * sem = &popped_sema_elem->semaphore;
+   sema_up (sem);
  }
}
```

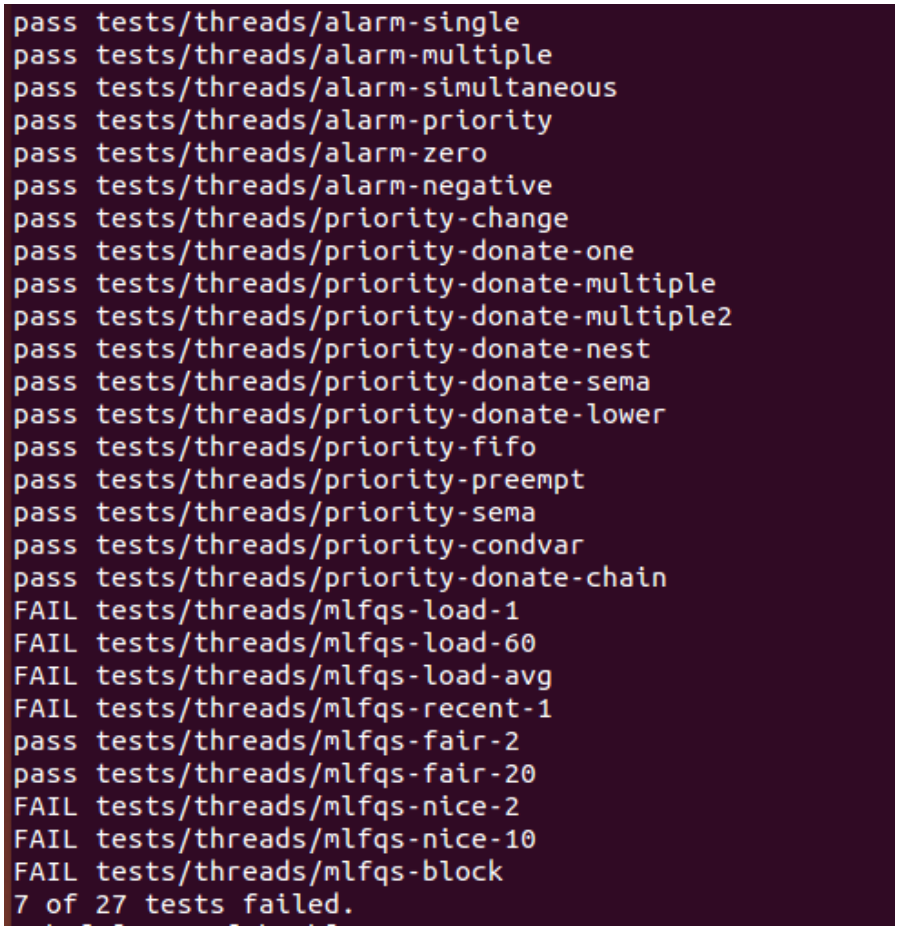
- `threads/synch.c`

```
int sema_get_maximum_priority (struct semaphore *sema)
{
  enum intr_level old_level = intr_disable();
  if(list_empty(&sema->waiters)){
    return 0;
  }
  struct list_elem * max_elem = list_max(&sema->waiters,compare_thread_priority,NULL);
  struct thread * max_thread = list_entry(max_elem,struct thread, elem);
  int priority = max_thread->priority;
  intr_set_level(old_level);
  return priority;
}
```

- `threads/synch.c`

```
bool compare_sema_elem_by_priority(struct list_elem * a, struct list_elem * b, void * AUX UNUSED){
    struct semaphore_elem * a_sema_elem = list_entry(a,struct semaphore_elem, elem);
    struct semaphore_elem * b_sema_elem = list_entry(b, struct semaphore_elem, elem);
    struct semaphore * a_sema = &a_sema_elem->semaphore;
    struct semaphore * b_sema = &b_sema_elem->semaphore;
    int a_sema_max_priority = sema_get_maximum_priority(a_sema);
    int b_sema_max_priority = sema_get_maximum_priority(b_sema);
    return a_sema_max_priority<b_sema_max_priority;
}
```

4. Result



Our code passed all of the tests for problem 1 and problem 2. Some mlfqs test for Problem 3 failed.

5. Discussion

Flexibility

One problem that our program has is the violation of the single responsibility principle at semaphore and lock. Ideally, semaphore shouldn't care about lock. Updating priority on lock waiting is lock's domain. But, before we call `sema_down` function, the thread is not added to the waiter of sema. And after we call `sema_down` function, thread cannot get control back. Therefore in our thread update structure, the only possible location to update is inside `sema_down` function. We can solve this problem by adding a new `lock_priority_update` function that has two arguments, `new_waiter_priority`, which includes `new_waiter_priority` at donation priority selection. But it makes code bloated and duplicated. So, we leave it as it is.