Selemon Yitbarek - NWEN301 - Project1 Report - ID 300223775

A Priority Ready Queue (PART 1)

Data Structures and Algorithms

in thread.c I declared/changed the following.

In the method thread_create:
//my code

```
if(thread_get_priority()<t->priority){
 thread_yield();


}
```

the purpose of this code was to test if the priority of the new thread is greater than the currently running thread and if so we would thread_yield n the new thread (t) will take over and start running and the other thread gets placed in the ready list.

```
/* Returns true if priority A is greater than prioriy B, false
  otherwise. */
//my additional code
static bool
priority_greater (const struct list_elem *a_, const struct list_elem *b_,
        void *aux UNUSED)
{
  const struct thread *a = list_entry (a_, struct thread, elem);
  const struct thread *b = list_entry (b_, struct thread, elem);

  return a->priority > b->priority;
```

}

I wrote this code to compare the priority of two threads and it returns a boolean, it returns true if a->priority is greater than b->priority else it returns false.

in the method thread_unblock:
//my code

list_insert_ordered(&ready_list, &t->elem, priority_greater, NULL);

the purpose of this code is to make sure when threads are being unblocked its being added to the ready list and when doing so making sure its sorted by its priority.

in the method thread_yield:
//my code
list_insert_ordered (&ready_list, &cur->elem, priority_greater, NULL);

the purpose of this code is to make sure if the current thread is not the idle thread then we add the current thread to the ready list and when doing so making sure its sorted by its priority.

in the method thread_set_priority:
//my code
struct thread *t = list_entry(list_begin(&ready_list), struct thread, elem);
if(t->priority> new_priority) {
    thread_yield();

}

the purpose of this code is to set the new thread with the thread with the highest priority in the ready list and test to see if that thread has a higher priority than the currently running thread if so thread yield and the new thread (t) takes over.

Algorithms

in the method thread_create the algorithm did the following:
Creates a new kernel thread named NAME with the given initial

PRIORITY, which executes FUNCTION passing AUX as the argument,

and adds it to the ready queue. Returns the thread identifier

for the new thread, or TID_ERROR if creation fails.

If thread_start() has been called, then the new thread may be

scheduled before thread_create() returns. It could even exit
before thread_create() returns. Contrariwise, the original
thread may run for any amount of time before the new thread is
scheduled. Use a semaphore or some other form of
synchronization if you need to ensure ordering.
The code provided sets the new thread's `priority' member to
PRIORITY, but no actual priority scheduling is implemented.

in the method priority_greater the algorithm did the following:
Returns true if priority A is greater than prioriy B, false otherwise.

in the method thread_unblock the algorithm did the following:
Transitions a blocked thread T to the ready-to-run state.
This is an error if T is not blocked. (Use thread_yield() to
make the running thread ready.) This function does not preempt the running thread. This can
be important: if the caller had disabled interrupts itself,
it may expect that it can atomically unblock a thread and
update other data.

in the method thread_yield the algorithm did the following:
Yields the CPU. The current thread is not put to sleep and
may be scheduled again immediately at the scheduler's whim

in the method thread_set_priority the algorithm did the following:
Sets the current thread's priority to NEW_PRIORITY.

in the method

Priority and synchronisation (PART 2)

Data Structures

in synch.c I declared/changed the following:

//my code

static bool highest_sema_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED);

the purpose of this code is to declare the function highest_sema_priority, it tells the compiler that it can use this function and expect that it will be defined somewhere.

//my code

```
static bool highest_sema_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)

{

  ASSERT(a!=NULL);

  ASSERT(b!=NULL);



  const struct semaphore_elem *semaA = list_entry(a, struct semaphore_elem,

                                    elem);

  const struct semaphore_elem *semaB = list_entry(b, struct semaphore_elem,

                                    elem);



  return (semaA->sema_pri > semaB->sema_pri);

}
```

the purpose of this code is to compare two semaphore elem priorities and return true if semaA is greater than semaB else return true.

```
/* Returns true if priority A is greater than prioriy B, false
   otherwise. */
static bool
priority_greater (const struct list_elem *a_, const struct list_elem *b_,
          void *aux UNUSED)
{
  const struct thread *a = list_entry (a_, struct thread, elem);
  const struct thread *b = list_entry (b_, struct thread, elem);

  return a->priority > b->priority;
}
```

I wrote this code to compare the priority of two threads and it returns a boolean, it returns true if a->priority is greater than b->priority else it returns false.

in the method sema_down:

```
//my code
list_insert_ordered (&sema->waiters, &thread_current ()->elem, priority_greater, NULL);
```

the purpose of this code is to add the current thread elem to the list of semaphore waiters and when doing so making sure its sorted by its priority.

in the method sema_up:

```
//my code from here
  struct thread *t;
  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)){
    t = (list_entry (list_pop_front (&sema->waiters),struct thread, elem));
    thread_unblock(t);

  }
  sema->value++;
  if(thread_get_priority()<t->priority){
```

thread_yield();

}

the purpose of this code is to get the first thread from the sema waiters (which is the one with the highest priority) into the thread t then unblock that thread. then it checks to see if the priority of the current thread is less than t's priority which is the one that just got popped out from the sema waiters if so thread yield and t thread takes over.

in the struct semaphore_elem:
//my code

int sema_pri;

The purpose of this code is to store the top's thread priority in the list of waiters of each semaphore

in the method cond_wait:
//MY CODE
waiter.sema_pri=thread_get_priority();
list_insert_ordered(&cond->waiters, &waiter.elem, highest_sema_priority, NULL);

The purpose of this code is to set the priority of the semaphore here with the top element of waiters list in the semaphore struct after setting the priority of the semaphore sort the semaphores waiters list to reflect the top thread's priority

Algorithms

in the method priority_greater the algorithm did the following:

Returns true if priority A is greater than prioriy B, false otherwise.

in the method sema_down the algorithm did the following:

Waits for SEMA's value

to become positive and then atomically decrements it.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but if it sleeps then the next scheduled

thread will probably turn interrupts back on

in the method sema_up the algorithm did the following:

Increments SEMA's value and wakes up one thread of those waiting for SEMA, if any.

the struct semaphore_elem did the following:

One semaphore in a list contains a List element, a semaphore and the top's thread priority in the list of waiters of each semaphor

in the method cond_wait the algorithm did the following:

Atomically releases LOCK and waits for COND to be signaled by
some other piece of code. After COND is signaled, LOCK is
reacquired before returning. LOCK must be held before calling
this function. The monitor implemented by this function is "Mesa" style, not
"Hoare" style, that is, sending and receiving a signal are not
an atomic operation. Thus, typically the caller must recheck
the condition after the wait completes and, if necessary, wait
again. A given condition variable is associated with only a single
lock, but one lock may be associated with any number of
condition variables. That is, there is a one-to-many mapping
from locks to condition variables.

in the method highest_sema_priority the algorithm did the following:

Returns true if priority semaA is greater than prioriy semaB, false otherwise.

Priority and synchronisation (PART 3)

Data Structures/Algorithms

Thread.h

this is used for priority scheduling

```
//my code
int init_priority;
struct lock *wait_on_lock;
struct list donations;
struct list_elem donation_elem;
```

This is used for FreeBSD scheduling

```
//my code
int nice;
int recent_cpu;
```

Note about my result (PART 3)

I have spent many hours trying to get this part to work 100% but I was not able to get it working perfectly. some tests pass and some don't for this part.

Synchronization

my various data structures are protected by the use of semaphore and condition methods making them mutually exclusive. race conditions are avoided in my code by using semaphores. one Thread can take semaphore, modify the value and release it If another thread tries to access the variable, it should acquire the semaphore, if its already acquired, it is pended and gains access after the previous thread relinquishes control.

My Design

my queue design was a list insert order design. I made a comparator that compares two threads priority and returns true or false I was able to use this comparator every time I was adding to the ready list I believe ordering it when adding to the list instead of ordering when removing made the performance more efficient.

Also, I decided whenever I was creating a thread or setting a priority to a thread I made sure I checked if the new thread created had a higher priority than the currently running if it did I would call thread yield and the new thread would start running. when it came to setting a priority, I initialized the currently running thread with the new priority and took the thread with the highest priority off the ready list and compared it with the current thread priority, if the current thread priority is smaller than the thread off of the ready list then I thread yield.