CIS – 545 ARCHITECTURE AND OPERATING SYSTEMS
PROJECT - 2

Submitted To

Janche Sang

Submitted By

Prudhvi Reddy Araga (praraga)

TASK 1: EFFICIENT ALARM CLOCK

DATA STRUCTURES AND FUNCTIONS

- **Sleeping List Queue Definition**

  static struct list sleeping_list; //linked list of thread element

- **Addon to thread struct, global reference wakeup time**

  int64_t wakeup_time;

- **Sleeping list insertion function**

  void enqueue_sleeping_list (int64_t wakeup_time, struct thread* thread);

- **Redefinition of timer sleep function to support non-busy wakeup**

  void timer_sleep (int64_t ticks);

- **Timer awake function to wake sleeping threads after sleep time**

  static void thread_awake (void);

- **Redefinition of timer interrupt handler. Does not busy wait.**

  static void timer_interrupt (struct intr_frame *args UNUSED);

- **List comparator function for "list_insert_ordered()", ASCENDING order.**

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

ALGORITHMS

Definition of sleeping list enqueue function

void enqueue_sleeping_list (int64_t wakeup_time, struct thread* thread)

- Set wakeup_time struct thread field
- Call list_insert_ordered () which inserts thread's list_elem elem in ascending wakeup_time manner
- Block current thread

We use thread's list_elem elem variable since thread can belong to only one list at a time and creating another linked list of threads will be too costly (we don't want dynamically allocated data structure)

Redefinition of timer sleep function

void timer_sleep (int64_t ticks)

- Compute wakeup_time as timer_ticks () + ticks
- Call enqueue_sleeping_list ()

Definition of thread awake function

static void thread_awake (void)

- Traverse sleeping_list starting from the beginning and check whether thread are poppable. For reference: We define poppable as timer_tics () >= wakeup_time
- If poppable, call list_remove (thread) and thread_unblock (thread)

Redefinition of timer interrupt function

static void timer_interrupt (struct intr_frame *args UNUSED)

- Call timer_tick ()
- Call thread_awake ()

Definition of wakeup time less than function

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

- Obtain thread struct pointer by calling list_entry ()
- Return thread a -> wakeup_time < thread b -> wakeup_time

RE-IMPLEMENTATION OF ALARM CLOCK ALGORITHM

Following are the list of steps that are advised for re-implementation of timer_sleep():

- Create a new list of threads i.e., waiting_list
- If we see any thread calling, then it needs to be removed from the ready list and add them to waiting_list.
- Change the status of thread to THREAD_BLOCKED
- Threads will be waiting within the waiting list until the timer expires
- In case of any interruption with the timer, need to move back the thread to ready list

Changes within the timer_sleep function are as follows:

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void timer_sleep (int64_t ticks)
{

 if (ticks<=0)
  return;

  ASSERT (intr_get_level () == INTR_ON);

enum intr_level old_level = intr_disable();
thread_current()->ticks_blocked = ticks;
thread_block();

intr_set_level(old_level);

}
```

SYNCHRONIZATION

To avoid race conditions, we have to disable interrupts at the beginning of enqueue_sleeping_list () and thread_awake (). We are iterating through a list with pointers, so we want to make sure interrupts are disabled. Thread is blocked once it's put into a sleeping_list and unblocked once it's popped out of the list.

RATIONALE

The given implementation avoids busy waiting by using the thread_block () and thread_unblock (). The data structure sleeping_list is maintained by inserting in sorted order. This ensures that we minimize the number of elements that need to be checked during thread_awake(). On average, the function will only need to peek at one. Call to thread_awake () is added to timer_interrupt () in order to make sure threads wake up exactly on the set wakeup_time.

TASK 2: PRIORITY SCHEDULER

DATA STRUCTURES AND FUNCTIONS

Nested Priority Donation function

void donate(curr_thread, dependent_thread);

Changes to the following functions

- In linked list struct: void insert_sorted_order(), to insert behind threads of the same priority, for support of round robin.

- Modify lib/kernel/list/list_sort(...) so that it implements a stable sort.
- Modify lib/kernel/list/list_insert_sorted(...) so that it implements a stable insert in rear of like elements.
- Add call to list_insert_ordered() in sema_down() to replace push_back()
- cond_wait()
- thread_unblock() update to call list_insert_ordered()
- thread_yield() update to call list_insert_ordered()

Additions to struct thread

- int effective priority
- struct thread* child_thread
- struct lock* lock, to support donation so that we have a reference for waiting lock

Ready list struct addition

- Add struct lock* lock field to ready_list, for mutual exclusion during insert, sort, and pop.

Semaphore struct field addition

- Add struct lock* lock field to wait_list for semaphores, for mutual exclusion during insert, sort, and pop.
- In the case of the wait_list, this will remain as NULL always. Added for generality of sort/insert algorithms.

Waiting list changes (none)

- We avoid changes in waiting list implementation by instead changing the implementation of insert and sort function.

Small change in thread creation function

- Change create_thread() function to initialize effective_priority to priority.

ALGORITHMS

Nested Priority Donation

void donate(struct thread* curr_thread, struct thread* blocking_thread)

- If in MLFQS mode, return.
- Initialize variable max_priority to max(dependent_thread->eff_priority, curr_thread->eff_priority)
- Create a traversal pointer, initially equal to the current thread pointer
- Set blocking_thread->eff_priority to max_priority

- Loop while blocking_thread != NULL && curr_thread->eff_priority > blocking_thread->eff_priority
  - set blocking_thread->eff_priority = curr_thread->eff_priority
  - set current_thread = blocking_thread
  - set blocking_thread = blocking_thread->waiting_lock->holder // go to next link
  - update max_priority as done above during initialization.
  - call list_sort() on waitlist
- if any priority in readylist was changed, call list_sort() on ready_list

Linked list sort modification

- Simply modify lib/kernel/list/list_sort(...) algorithm so that it is a "stable" sort. Aquires list-> lock before and releases after sort.

Insert algorithm modification

- Simply modify lib/kernel/list/list_insert_sorted(...) algorithm so that it is a "stable" insert.
- Aquires list->lock before and releases after insertion.

Choosing the next thread to run

- No changes need to be made to this function. Why?
  - We simply pop from the first thread from the ready or waiting list as code already does.
  - This is fine because we are maintaining a sorted descending ordering of the list (by effective priority), with internal sort for round robin, elsewhere in the code.

Acquiring a Lock

- In sema_down(), replace list_push_back() with insert_sorted_order(), in order to maintain sort.

Releasing a Lock

- In sema_up(), add logic to revert the current thread's (child) effective priority to its original priority.
- There is no need to modify the parent threads, as there are two cases:
  - The dependent thread (parent) has a higher effective priority, or
  - The dependent thread has a lower effective priority.
- In both cases, the child releasing the lock does not affect the parent's effective priority.

Computing the effective priority

- Effective priority is determined when a thread calls lock_acquire()
- At this point it will propagate its current effective priority to the lock holder thread and update as needed.
- This process continues "recursively" (really, implementation is iterative to preserve size of the stack frame).
- Traversal terminates when the holder thread no longers has a dependency (child).

Priority scheduling for semaphores and locks

- When inserting threads into waiting lists, place the elem in descending order using list_insert_ordered().
- This list is now maintained such that it will be sorted by effective priority.
- This guarantees that when the list is popped, the highest priority thread is chosen first.
- Furthermore, by calling list_sort() during execution of donate(), we maintain ordering after dynamic priority updtes within both the thread readylist and the semaphore waiting list.

Priority scheduling for condition variables

- Use the same method as above.

Changing thread's priority

- A thread's effective priority is only modified when donate() is called by the thread attempting to acquire a held lock.
- At this point, effective priority of all threads fighting over the lock becomes the max of their effective priorities.

RE-IMPLEMENTATION OF PRIORITY SCHEDULING ALGORITHM

Following are the list of steps that are advised:

- When we schedule a new thread, we need to find the thread with highest priority to make sure that it gets executed first.
- Returning the base priority is necessary.
- All threads in the nested loop should have the same donation.

For this implementation, the changes are done within the files – Thread.h, Thread.c, Synch.c

```
void
thread_update_priority(struct thread *t)
{
  enum intr_level old_level = intr_disable();
  int max_pri = t->base_priority;
  int lock_pri;
  if (!list_empty(&t->locks_holding))
  {
    list_sort(&t->locks_holding, lock_cmp_priority, NULL);
            lock_pri = list_entry(list_front(&t->locks_holding), struct lock, elem)->max_priority;
    if (max_pri < lock_pri)
              max_pri = lock_pri;
  }
  t->priority = max_pri;
intr_set_level(old_level);
}

Void thread_set_priority (int new_priority)
{
  /*
  thread_current ()->priority = new_priority;
  thread_yield();
  */

  /* Solution Code */
  if (thread_mlfqs)
    return;
  enum intr_level old_level = intr_disable();
  struct thread *cur = thread_current();
  int old_priority = cur->priority;
  cur->base_priority = new_priority;

  if (list_empty(&cur->locks_holding) || new_priority > old_priority)
  {
   cur->priority = new_priority;
          thread_yield();
  }
  intr_set_level(old_level);
}
```

SYNCHRONIZATION

Shared resources that are handled during this section include the semaphore waitlist, readylist, and sleeping_list. We synchronize these by adding a lock field to each of the definition of these list structs. We aquire the lock before modification of the list, and after completion of the modification.

RATIONALE

For implementation of the ready and waiting list queues: By maintaining ready/waiting lists in sorted order, we can simply keep the current method of popping the lists to determine the thread with the highest effective priority. Because we treat insertion as a LIFO queue, where threads of the same priority are internally sorted by last run time, we are able to maintain a round robin schedule by simply popping frome the sorted list, while avoiding saving runtimes and things of that nature. We considered several other options, including creating an array of size 64 of pointers to linked lists, where the i'th linked list would be a LIFO queue of threads of the i'th priority, in round robin order. However, this would require a minimum of 256 bytes of memory just for storing the pointers, which in the case of the waiting lists for semaphores, might be mostly empty (wasted space), and there would be one of these for each semaphore, which could take up a considerable amount of space. This might be unacceptable if the resource is in the kernel stack or static memory, as memory is scarce.

Another considered alternative was similar to the previous, except we would have a linked list of max size 64, which would link to "priority nodes." If there exist no threads of a given priority, then there would be no node for that priority. The existing nodes would point to their own linked lists, which would be a LIFO queue for round robin. This would get rid of the wasted space problem but would prove to be complex in implementation. Our chosen implementation would be correct, but would require additional time complexity in traversing the list for insertion, and requires a custom implementation of insert_in_sorted_order().

For implementation of nested priority donation: We considered a recursive solution but realized that the stack grows unbounded in the case of a long chain of lock dependencies, and therefore considered a solution that could be implemented iteratively. The proposed solution of saving pointers to the dependent thread was simplistic and straightforward, but then required us to create a reference to the lock that a thread is waiting on within the thread struct, in order to always have a reference to the next child when following the dependency chain (since the lock already has a reference to its owner).