Project 2 Design Doc

Group Members

Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

Task 1: Efficient Alarm Clock

Data Structures and Functions

Add a list to keep track of threads that are sleeping, sorted by time to be woken up.

threads/thread.c:

```
static struct list sleeping_list;
```

Add a field in `struct thread` to keep track of remaining ticks to sleep for.

threads/thread.h:

```
struct thread {
…
int sleep_ticks;
…
}
```

Modify `timer_sleep()` to add thread to `sleeping_list`.

devices/timer.c:

```
void timer_sleep(int64_t ticks)
```

Modify `time_interrupt()` to decrement ticks for sleeping threads and unblock any threads that need to be woken up.

```
static void timer_interrupt(struct intr_frame* args UNUSED)
```

Algorithms

Since we want to avoid busy waiting in `timer_sleep()`, we should not keep wasting cpu time on calling `thread_yield()` in a while loop. Instead, we put the thread to sleep by calling `thread_block()`. We keep track of a list of sleeping threads with their remaining sleep time recorded. In `timer_sleep()`, we add current thread into the `sleeping_list`. `timer_interrupt()` decrements ticks for each sleeping thread over time, and calls `thread_unblock()` to put the thread into the ready queue if remaining sleep time is 0.

Synchronization

We disable interrupt before calling `thread_block()` in `timer_sleep()` and enable interrupt before it returns. This ensures that `thread_block()` is correctly called and makes sure that the thread and the timer interrupt handler (via `timer_sleep()` and `timer_interrupt()` respectively) do not access `sleeping_list` at the same time.

Rationale

We also considered not to keep track of a list of sleeping threads. The desired behavior can also be achieved by iterating through all threads, decrement ticks and unblock necessary threads. However, considering that `timer_interrupt()` should run as fast as possible, keeping track of a list of sleeping threads reduces the amount of threads the interrupt handler needs to iterate through. Coding for this part should be fairly straightforward. In terms of time complexity, putting the thread to sleep using `timer_sleep()` should be O(1) since we only need to insert the thread into `sleeping_list`. `timer_interrupt()` would be O(n) (n being the number of sleeping threads) since we need to iterate through the sleeping threads.

Task 2: Priority Scheduler

Data Structures and Functions

Add a field to store locks a thread holds
Add a field to store threads that are waiting for this clock

```
struct lock {
…
struct list_elem elem;
struct list wait_thread;
…
}
```

Add several fields to struct thread.

```
struct thread {
…
struct lock waited_lock;  //the lock that thread is waiting for
Int effective_priority;  // thread's effective priority
struct list held_locks; // a list of locks this thread holds
Int priority;                 // thread's base priority
Int highest_p;          // keep track of the highest priority
…
}
```

`src/threads/threads.c`

```
/* Implement priority scheduling here*/
static struct thread* next_thread_to_run(void) {
    if (list_empty(&ready_list))
           return idle_thread;
      else
        ...
            return list_entry(highest_priority_thread);
}

/* Modify this function described in the algorithms section */
void thread_set_priority(int new_priority);

/*Change it to return the effective priority of the thread*/
thread_get_priority() {
    return thread_current()->effective_priority;
}
```

`src/threads/synch.c`

/* Modify lock_acquire to support the algorithms described below / *void lock_acquire(struct lock* lock);

/* Modify lock_release to support the algorithms described below / *void lock_release(struct lock* lock);

/* Change semaphore to support priority scheduling, see algorithms for specific description*/
Modify `sema_up(struct semaphore * sema)` to `sema_up(&lock->semaphore)`

/* Change condition variables to support priority scheduling / *void cond_signal(struct condition* cond, struct lock* lock UNUSED);

Algorithms

-Choosing the next thread to run: iterate through the ready queue to get a thread with the highest effective priority and run it next.

Priority Donation

In order to implement strict priority scheduling, we add a field `effective_priority` to the `thread` struct so that we can use `effective_priority` to determine which thread to run next. The field `priority`, on the other hand, will be used to keep track of different threads' base priorities.

-Acquiring a Lock:

First to check whether the lock has an owner.

If the lock doesn't have an owner, the thread that tries to acquire the lock should have the highest priority. We should update the lock's owner and add the lock to `held_locks`, a list of locks that thread holds.

If the lock has at least one owner, We want to compare the effective priorities of the current thread and the holders of the lock:

If the holder's effective priority is higher, we don't update any priority.
If the current thread's priority is higher, set the effective priority of the holder to the current thread's effective priority. In this case, nested donation is needed and we would need to recurse through the chain of locks that the holder is waiting on and set the effective priorities of all the holders with a lower effective priority to the effective priority of the current thread.

In either case, when priority donation is done, we set the current thread's `waited_lock` to be this lock, add it to the list of waiters of the lock, and block the current thread.

-Releasing a Lock:

First, we want to remove the lock from current threads' `held_locks`. We then want to traverse through current thread's remaining `held_locks` and find the one with the highest priority. In order to keep track of this highest priority, we name it as `highest_p`, compare it with current thread's base priority, find the max one and set it as the current thread's effective priority.

By the way that our `sema_up()` function is modified as suggested below, we can then call `sema_up(&lock->semaphore)`, which takes care of unblocking the thread with highest priority among all waiters of the lock in `wait_thread`.

In case when the current thread has been decremented due to the previous steps, we want to check whether its effective priority is lower than any thread in the read list. If that is the case, we want to immediately call `thread_yield()`.

-Priority scheduling for semaphores and locks

We modify `sema_up()` to pop off and unblock the thread with highest `effective_priority` from the list of waiting threads. This way, we up the semaphore for the thread with highest priority instead of doing it by the order they were put on the waiting list.

This also works for locks because locks are implemented using semaphore. Note that we make sure to set `thread->waited_lock` back to null when a thread is unblocked and acquires the lock that it is waiting on.

-Priority scheduling for condition variables

Since condition variables are implemented using locks and semaphores, the above mentioned modifications for `sema_up()` will achieve our desired behavior to signal the thread with highest priority.

-Changing a thread's priority

In the function `thread_set_priority(int new_priority)`, we always want to update the thread's base priority `thread->priority` to `new_priority`. However, consider the following case where `thread->effective_priority` could possibly be modified:
If `new_priority >= thread->effective_priority`, then we set `thread->effective_priority = new_priority`.
If `new_priority < thread->effective_priority && old_base_priority < thread->effective_priority`, after decrementing the base priority, we are done. Effective priority doesn't need to be modified since it is already a resultant value from priority donation.
If `new_priority < thread->effective_priority && old_base_priority = thread->effective_priority`,after decrementing the base priority, it's possible that some waiters of the locks held by the current thread have a higher priority. In this case, priority donation is needed. Namely, we want to traverse through `thread->held_locks` to get the highest effective priority of the waiters of the locks from `wait_thread` and set the effective priority of the current thread to that priority.

Synchronization

We changed synchronization primitives (locks, semaphores, condition variables) so that when it releases a lock or runs `sema_up`, they will find the thread with the highest priority on the waiting queue, and make it available to run.

Also notice that `lock_acquire` and therefore `sema_down` may sleep while `sema_up` might be called from the interrupt handler. Therefore, in order to prevent attributes of locks and semaphores used in our priority scheduler from being modified by other threads in case of a timer interrupt, where other threads might try to interfere and access these field variables, we want to disable interrupts while locks and semaphores are been accessed and modified.

Rationale

One thing we considered was to keep the ready list and waiters of each lock sorted, so we would get the next thread to unblock and the highest priority among the locks in constant time. However, a thread's priority can be dynamically changed by `thread_set_priority()` and through priority donation, and we would need to resort the lists whenever there is a small change to any effective priority, and sorting takes more than linear time. So instead, we decided to not to update the order of the list, but to traverse through the lists to get the highest priority thread instead. The function `list_max()` does this nicely so it would be easier to code, and it is linear time, which is better than sorting the lists every time.

You must also answer these additional questions in your design document:

1. In class, we studied the three most important attributes of a thread that the operating system stores when the thread is not running: the program counter, stack pointer, and registers. Where/how are each of these three attributes stored in Pintos? You may find it useful to closely read pintos/src/threads/switch.S and the schedule function in pintos/src/threads/thread.c. You may also find it useful to look over the slides from Lecture 73 and Lecture 84 of Summer 2020's CS162 offering.

The stack pointer is stored in the `stack` field in `struct thread` and is loaded when a thread is switched. The program counter and registers `ebx`, `ebp`, `esi`, `edi` are stored on the thread's stack and can be restored after the stack pointer is retrieved.

2. When a kernel thread in Pintos calls thread_exit, when/where is the page containing its stack and TCB (i.e. struct thread) freed? Why can't we just free this memory by calling palloc_free_page inside the thread_exit function? (You won't get credit for this question by just copy/pasting from comments in the Pintos skeleton – convince us that you understand what's going on here).

The page contains its stack and TCB is freed after switching the threads, and it's done by `thread_schedule_tail`. `thread_exit` will call `schedule` and `switch_thread` which needs the current thread and the next thread. If we free the current thread inside

the `thread_exit()` function and isn't called after `switch_thread`, we cannot make sure a new thread is successfully running and switch to it before the current one has been freed.

3. When the thread_tick function is called by the timer interrupt handler, in which stack does it execute?

It executes in the kernel stack since the interrupt handler runs in the kernel mode.

4. Consider a fully-functional correct implementation of this project, except for a single bug, which exists in the sema_up() function. According to the project requirements, semaphores (and other synchronization variables) must prefer higher-priority threads over lower-priority threads. However, this buggy implementation chooses the highest-priority thread based on the base priority rather than the effective priority. Essentially, priority donations are not taken into account when the semaphore decides which thread to unblock. Please design a test case that can prove the existence of this bug. Pintos test cases contain regular kernel-level code (variables, function calls, if statements, etc) and can print out text. We can compare the expected output with the actual output. If they do not match, then it proves that the implementation contains a bug. You should provide a description of how the test works, as well as the expected output and the actual output.

Let there be 5 threads A, B, C, D, E and 3 locks. Assign threads the following priority:

A: 20
B: 30
C: 3
D: 2
E: 1

E holds lock_1, D holds lock_2, C holds lock_3.
Thread B tries to acquire lock_2 held by D. D's effective priority is raised to 30.
Thread A tries to acquire lock_3 held by C. C's effective priority is raised to 20.
Both threads C and D tries to acquire lock_1 held by E. After E releases the lock, we check who is the owner of lock_1.
If the owner is D, which has a lower base priority but a higher effective priority, this test passes. If the owner is C, who has a higher base priority but a lower effective priority, there is a bug.