CS302 Operating System: Pintos Design Report

Name: 傅伟堡(Weibao Fu)

SID: 11812202

Task 1: Efficient Alarm Clock

1. Data structures and functions

Modified Structure

`int64_t blocked tick;`

- add variable to thread structs
- initialize to 0
- set to the `ticks` when sleep a thread

Added Function

```
void thread_check_block(struct thread *t, void *aux){
  if (t->status != THREAD_BLOCKED) return;
  if(!(--t->blocked_tick)) thread_unblock(t);
}
```

- to check whether the thread is blocked
- unblock the thread if `t->blocked_tick` is 0

```
bool thread_cmp_priority(const struct list_elem *a, const struct list_elem *b,
void *aux){
  return list_entry(a, struct thread, elem)->priority >
            list_entry(b, struct thread,elem)->priority;
}

void list_push_back_thread_priority (struct list *list, struct list_elem *elem){
  list_insert_ordered(list, elem, &thread_cmp_priority, NULL);
}
```

- keep the thread list ordered by their priority

Modified Function

```
static void init_thread (struct thread *t, const char *name, int priority){
    t->blocked_tick=0;
}
```

- initialize the variable `blocked_tick` to 0

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_foreach(thread_check_block, NULL);
  thread_tick ();
}
```

- call function `thread_check_block` for every threads

```
void thread_unblock (struct thread *t)
{
  # original: list_push_back (&ready_list, &t->elem);
  list_push_back_thread_priority (&ready_list, &t->elem);
}

static void init_thread (struct thread *t, const char *name, int priority)
{
    # original: list_push_back (&all_list, &t->allelem);
    list_push_back_thread_priority (&all_list, &t->allelem);
}

void thread_yield (void)
{
    if (cur != idle_thread)
        # original: list_push_back (&ready_list, &cur->elem);
        list_push_back_thread_priority (&ready_list, &cur->elem);
}
```

- ensure the thread list is ordered by their priority

2. Algorithms

Overview

When a thread goes to sleep, we set its `blocked_tick=0`, and then block the thread. Every time interrupt (tick) we let each blocked thread `blocked_tick--` and check if need to wake then up (if its `blocked_tick==0`, then we unblock it). Every time we select a queue from `ready_list`, we will select the thread with highest priority as we keep the queue ordered by thread's priority.

Call timer_sleep(int64_t ticks)

(1) the current thread's `blocked_tick` is set to the given sleep ticks
(2) disable interrupts
(3) the thread is blocked
(4) reset interrupts level to its old one

Timer interrupt handler

(1) for each thread, check whether it is blocked
(2) if one thread is blocked, minus its `blocked_tick` by 1
(3) when the blocked thread's `blocked_tick==0`, unblock this thread

Call thread_unblock(struct thread *t)

(1) insert the thread into `ready_list`, which is a priority queue ordered by the thread priority

Call init_thread (struct thread *t, const char *name, int priority)

(1) initialize the thread (set its blocked_tick as 0)
(2) insert the thread into `all_list`, which is a priority queue ordered by the thread priority

Call thread_yield (void)

(1) disable interrupts
(2) if current thread's status is not idle, then push the thread into `ready_list`
(3) change the current thread's as `THREAD_READY`
(4) schedule and pick a new thread to run
(5) reset interrupts level to its old one

3. Synchronization

Every time we call `timer_sleep(int64_t ticks)`, we set the interrupt is disabled. This method can prevent race conditions.

4. Rationale

Actually, I implemented this by adding `sleep_queue` in the first time. But this will take up unnecessary spaces to store this structure. Thus, I choose to implement this by add a thread's attribute to record how many ticks should this thread be blocked. Then I only need to check the `blocked_tick` of the thread in each interval, and decide whether unblock this thread. Besides, we do not need to modify too much code to accomplish it, I think it's efficient and reasonable.

Task 2: Priority Scheduler

1. Data structures and functions

Modified Structure

`int original_priority;`

- add variable to thread structs
- store the original priority we assign the thread

`struct list lock_holding;`

- add variable to thread structs
- initialize to empty
- store the locks that the thread is holding

`struct lock *lock_waiting;`

- add variable to thread structs
- initialize to NULL
- store the lock that the thread is waiting

`int priority;`

- add variable to lock structs
- initialize to PRI_MIN
- store the priority of the lock

`struct list_elem elem;`

- add variable to lock structs
- for compare function

Added Function

```
bool lock_cmp_priority(const struct list_elem *a, const struct list_elem *b,
void *aux){
  return list_entry(a, struct lock, elem)->priority >
            list_entry(b, struct lock, elem)->priority;
}

void list_push_back_lock_priority (struct list *list, struct list_elem *elem){
  list_insert_ordered(list, elem, &lock_cmp_priority, NULL);
}
```

- keep the lock list ordered by their priority

```
bool sema_cmp_priority(const struct list_elem *a, const struct list_elem *b,
void *aux){
  struct semaphore_elem *sa = list_entry(a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry(b, struct semaphore_elem, elem);
  return list_entry(list_front(&sa->semaphore.waiters), struct thread, elem)-
>priority >        list_entry(list_front(&sb->semaphore.waiters), struct thread,
elem)->priority;
}
```

- define the sema compare function

```
void thread_update_priority(struct thread *t){
    if(t == idle_thread) return;
    int max_priority = max(PRI_MIN, t->original_priority);
    if(!list_empty(&t->lock_holding)){
        list_sort(&t->lock_holding, lock_cmp_priority, NULL);
        int front_priority = list_entry (list_front (&t->lock_holding),
                                         struct lock, elem)->priority;
        max_priority = min(max(max_priority, front_priority), PRI_MAX);
    }
    t->priority = max_priority;
}
```

- update thread priority according to the priority of threads in list and holding locks.
- ensure the correctness of thread's priority by calling this function when we we change `lock_holding`

Modified Function

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

  struct thread *current_thread = thread_current();
  struct lock *lock_iter = lock;
```

```
  if(lock->holder != NULL){
    current_thread->lock_waiting = lock;
    while(lock_iter != NULL){
      if(current_thread->priority <= lock_iter->priority) break;
      lock_iter->priority = current_thread->priority;
      thread_update_priority(lock_iter->holder);
      lock_iter = lock_iter->holder->lock_waiting;
    }
  }

  sema_down (&lock->semaphore);

  current_thread->lock_waiting = NULL;
  list_push_back_lock_priority(&current_thread->lock_holding,&lock->elem);
  lock->holder = current_thread;
  thread_update_priority(current_thread);
}
```

- acquire the lock and check the priority

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

  list_remove(&lock->elem);
  thread_update_priority(thread_current());
  lock->holder = NULL;
  lock->priority = PRI_MIN;
  sema_up (&lock->semaphore);
}
```

- release the lock

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

  ASSERT (sema != NULL);

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

  sema->value++;
  thread_yield();
  intr_set_level (old_level);
}

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_thread_priority (&sema->waiters, &thread_current ()->elem);
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}
```

- keep the sema list ordered by their priority

2. Algorithms

Overview

I add some attributes to `strcut lock` and `struct thread` to help me implement priority donation. Whenever a thread asking for the lock, we will iteratively update the thread priority by the priority of the lock. To be more specific, if we fail to acquire the lock, we will find the thread which holding the lock and then set their `priority` to the max of the their `priority`. Besides, I use function `void thread_update_priority(struct thread *t)` to ensure the correctness of thread's priority by calling this function when we we change `lock_holding`.

Choosing the next thread to run

It is implemented in function `static struct thread *next_thread_to_run (void)`. Actually, the threads in `ready_list` is ordered by their priority, as we push the thread by its priority, and higher priority thread will be in the front. In this case, whenever we recall this function, it will return the thread with highest priority in the `ready_list`.

Acquiring a Lock

When a thread try to acquire a lock, if the lock holder is empty then the thread can get the lock directly, otherwise, it should check the current lock holder. If the lock holder is smaller than the waiting thread's priority, we choose to update the lock holder's `priority` as the waiting thread's. It is noticeable that we should implement it iteratively until the `iter_lock` is NULL since chain of locks may exist. After the thread getting the lock, we should push the lock into the thread's `lock_holding` and set the lock's `holder` as the waiting thread. At last, we call function `thread_update_priority(struct thread *t)` to guarantee the correctness of the threads' priority.

Releasing a Lock

When a thread try to release a lock, we need to remove the lock from the global lock list using `list_remove(&lock->elem)`. Besides, we call function `thread_update_priority(struct thread *t)` to guarantee the correctness of the threads' priority. After that, we will call function `void sema_up (struct semaphore *sema)` to increase the lock's sema value. By this way, the waited highest-priority thread will get the lock.

Computing the effective priority

The implementation is in function `void thread_update_priority(struct thread *t)`. In this function, it updates the priority of the thread (effective priority) based on all the locks that the current thread has and the priority of the thread itself, that is, always selects the maximum value among them. So as long as we ensure that this function is called when the thread `lock_holding` changes, I can ensure that the priority of the thread is correct.

Priority scheduling for semaphores and locks

The implementation is in function `void sema_up (struct semaphore *sema)`. In this function, it will check the whole queue and then unblock the element with the highest priority. Besides, the thread waiting for the lock will check whether the priority needs to be donated the lock holder. If the holder's `priority` (effective priority) is smaller than the waiter's priority, the waiter sets the holder's `priority` to the holder's `priority`.

Priority scheduling for condition variables

The implementation is in function `void cond_signal (struct condition *cond, struct lock *lock UNUSED)`. In the previous implementation, the queue of condition is just a normal queue rather than a priority queue. In order to accomplish the priority scheduling for condition variables, we need to modify the queue to the priority queue. In this case, we just need to sort the `cond_waiters` by compare function `sema_cmp_priority()` before we call `sema_up()`.

Changing thread's priority

The implementation is in function `void thread_set_priority (int new_priority)`. Firstly, assign the current thread's `original_priority`. Then call function `thread_update_priority(struct thread *t)` to guarantee the correctness of the threads' priority. At last, we should use `thread_yield()` to check if there is a higher priority thread.

3. Synchronization

Actually, the donor have the power to set the lock holder's priority during the priority donation. And the thread also can change its priority. If the donor and the thread set the priority in different order, which will lead to a different result. In order to deal with this problem and prevent the situation happening, we will disable interrupts when we try to read or write to `priority`. By applying this method, we can protect against the possibility of two or more waiting threads reading from this value, getting the wrong priority, and setting it. In this case, we can guarantee synchronization.

4. Rationale

I think in this task, the two most important ideas are: (1) maintain the priority of the lock is correct (2) maintain the correct thread priority through the lists of locks. Actually, only the new thread needs to acquire the lock, the priority of the lock may change. Such that, we only need to maintain the correctness of the lock's priority in function `lock_acquire()`. When it comes to the change of priority, it will happen in the following four situations: (1) before the lock is acquired (2) after the lock is acquired (3) after the lock is released (4) change the priority manually. In this case, we need to function `thread_update_priority(struct thread *t)` to guarantee the correctness of the threads' priority. In my design, I use the changes of `priority` to indicate or not a thread's priority is donated (`priority` will be the same as `original_priority` if it is not donated). And I also add other attributes such as `lock_holding` to record the thread information. By this way, I can get lock information of the thread. This design can help us keep track of donated priority, which is very important for priority donation. Above all, I think it is an efficient and reasonable design.

Task 3: Multi-level Feedback Queue Scheduler

1. Data structures and functions

Global Variable

`fixed_t load_avg;`

- record the load_avg of the whole os

Modified Structure

`int nice;`

- add variable to thread structs
- determine how nice the thread should be to other threads

`fixed_t recent_cpu;`

- add variable to thread structs
- estimate of the CPU time the thread has used recently

Added Function

```
void thread_add_recent_cpu(struct thread *t){
  if(t == idle_thread) return;
  t->recent_cpu = FP_ADD_MIX(t->recent_cpu,1);
}
```

- increase `recent_cpu` of non-idle threads by 1

```
void thread_update_recent_cpu(struct thread *t){
  if (t == idle_thread) return;
  t->recent_cpu = FP_ADD_MIX(FP_MULT(FP_DIV(FP_MULT_MIX(load_avg, 2),
                  FP_ADD_MIX(FP_MULT_MIX(load_avg, 2),1)),t->recent_cpu),t-
>nice);
}
```

- update the `recent_cpu` of a thread

```
void thread_update_load_avg(void){
  load_avg = FP_ADD(FP_DIV_MIX(FP_MULT_MIX(load_avg, 59),60),
             FP_DIV_MIX(FP_CONST(thread_ready_count(thread_current())),60));
}
```

- update `load_avg` of the os

```
size_t thread_ready_count(struct thread *t){
  size_t ready_thread = list_size(&ready_list);
  if(t != idle_thread) ready_thread++;
  return ready_thread;
}
```

- count the number of thread with *READY* status

```
void thread_update_priority_mlfqs(struct thread *t){
  if (t ==  idle_thread) return;
  t->priority = FP_ROUND(FP_SUB(FP_SUB(FP_CONST(PRI_MAX),
              FP_DIV_MIX(t->recent_cpu,4)),FP_MULT_MIX(FP_CONST(2),t->nice)));
  t->priority = max(min(PRI_MAX,t->priority),PRI_MIN);
}
```

- update the thread priority

Modified Function

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
  enum intr_level old_level = intr_disable();
  // modified part
  if(ticks % TIMER_FREQ == 0){
    thread_update_load_avg();
    thread_foreach(thread_update_recent_cpu, NULL);
  }
  if(ticks % 4 == 0){
    thread_foreach(thread_update_priority, NULL);
  }
  intr_set_level(old_level);
}
```

- register the function in `timer_interrupt()`

```
void thread_update_priority(struct thread *t){
  if(t == idle_thread) return;
  // modified part
  if(thread_mlfqs){
    thread_update_priority_mlfqs(t);
  }else {
    int max_priority = max(PRI_MIN, t->original_priority);
    if(!list_empty(&t->lock_holding)){
      list_sort(&t->lock_holding, lock_cmp_priority, NULL);
      int front_priority = list_entry (list_front (&t->lock_holding), struct
lock, elem)->priority;
      max_priority = min(max(max_priority, front_priority), PRI_MAX);
    }

    t->priority = max_priority;
  }
}
```

- update threads' priority

2. Algorithms

Overview

The goal is to balance threads' different scheduling needs. Threads that perform a lot of I/O require a fast response time to keep input and output devices busy, but need little CPU time. On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. According to the formula, we can calculate `priority`, `recent_cpu`, `load_avg` dynamically. This formula is designed so that threads that have recently been scheduled on the CPU will have a lower priority the next time the scheduler picks a thread to run. This is key to preventing starvation: a thread that has not received any CPU time recently will have a `recent_cpu` of 0, which barring a very high nice value, should ensure that it receives CPU time soon. Also, we use the lower 16 bits to indicate the fractional part since `float/double` types are not defined in Pintos.

Update `priority`

The implementation is in function `void thread_update_priority_mlfqs(struct thread *t)`. Every thread has a nice value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (PRI_MIN) through 63 (PRI_MAX), which is recalculated using the following formula every fourth tick:

$$priority = PRI\_MAX - (recent\_cpu/4) - (nice \times 2)$$

Update `recent_cpu`

The implementation is in function `void thread_update_recent_cpu(struct thread *t)`. `recent_cpu` measures the amount of CPU time a thread has received “recently.” On each timer tick, the running thread’s `recent_cpu` is incremented by 1 (implemented in function `void thread_add_recent_cpu(struct thread *t)`). Once per second, every thread’s `recent_cpu` is updated this way:

$$recent\_cpu = (2 \times load\_avg)/(2 \times load\_avg + 1) \times recent\_cpu + nice$$

Update `load_avg`

The implementation is in function `void thread_update_load_avg(void)`. load_avg estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:

$$load\_avg = (59/60) \times load\_avg + (1/60) \times ready\_threads$$

where `ready_threads` is the number of threads that are either running or ready to run at time of update (not including the idle thread). It is implemented in function `size_t thread_ready_count(struct thread *t)`.

Choosing the next thread to run

It is implemented in function `static struct thread *next_thread_to_run (void)`. Actually, the threads in `ready_list` is ordered by their priority, as we push the thread by its priority, and higher priority thread will be in the front. In this case, whenever we recall this function, it will return the thread with highest priority in the `ready_list`.

3. Synchronization

In this task, the only synchronization problem is when we compute each thread's priority value. In this case, I set the interrupts disabled whenever I calculate the priority of threads. By this method, we can guarantee synchronization.

4. Rationale

This task is not very difficult as the formula of calculating `priority`, `recent_cpu`, `load_avg` are given. The first step is to solve the problem that there is no `float/double` type in Pintos. We select the lower 16 bits to indicate the fractional part, in this case, we should follow the `fixed_point.h` function to do calculation. Since this step is closed, we can solve that only by following the formula of `fixed_t`. However, the only open question is that how to do selection when we calculate the priority. To be more specific, the types of `recent_cpu`, `load_avg` are both `fixed_t`, while the type of `priority` is `int`. In this case, we should decide how to deal with the fractional part. Compared with abandoning the fractional part and just keep the integer part, I this `round` the fractional part is more reasonable. In this case, I choose to use `FP_ROUND` to calculate the priority.

5. Additional question

Question 1

Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a `recent_cpu` value of 0. Fill in the table below showing the scheduling decision and the `recent_cpu` and priority values for each thread after each given number of timer ticks. We can use `R(A)` and `P(A)` to denote the `recent_cpu` and priority values of thread A, for brevity.

My Assumption: first update recent_cpu then update priority, and I will round(PRI_MAX*(recent_cpu/4) -(nice*2)) to get the priority, and I will choose the longest waiting thread if there are two or more threads with the same highest priority

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

Question 2

Did any ambiguities in the scheduler specification make values in the table (in the previous question) uncertain? If so, what rule did you use to resolve them?

Yes. If there are two or more threads with the same highest priority, how to choose the next thread to run is ambiguous, which further leads the value of `recent_cpu` and `priority` uncertain. In this case, I will select the longest waiting time thread with highest priority to be the next thread to run.

Reference

---

[1] : CS302-OS-Project1_2021.pdf

[2] : Priority Donation

[3] : Advanced Scheduler