Task 1: Efficient Alarm Clock

1. Data structures and functions

- Create a new variable to mark sleep ticks in thread struct.

```
struct thread
{
    int64_t thread_sleep;
}
```

- Create a new function at thread.c to call thread sleep.

```
void thread_sleeped (int64_t ticks)
```

- Create a new function at thread.c to weakup sleeping thread.

```
void thread_weakup (void)
```

- Modify **timer_sleep** function to call

```
thread_sleeped(ticks);
```

rather than

```
thread_yield ();
```

- Modify **timer_interrupt** function to call

```
thread_weakup();
```

in order to weakup sleeping thread.

- Modify **init_thread** to inital those new variable.

2. Algorithms

- Let thread sleep.
  timer_sleep function will call **thread_sleeped** function to block a thread and set it's `thread_sleep` variable to ticks. So this blocked thread will never run again until it was weakup.

- Weakup a sleeping thread.
  At **timer_interupt** function, a main thread will call

```
thread_weakup();
```

to check each sleeping thread if it's `thread_sleep` variable is count to 0, if so weakup it and minus `thread_sleep` by 1 otherwise.

3. Synchronization

This task will use a static variable `ticks` and static list `all_list` , so I call **intr_disable** before access those resource.

4. Rationale

This design is easy to implement but not the most effective, for **thread_weakup** function check every thread at `all_list` , so we can create a `blocked_list` to store blocked thread, but it will be a lot of Synchronization issue. It need about 100 line code.

Task 2: Priority Scheduler

1. Data structures and functions

- Create a new list at thread struct to store locks that hold by thread. Is a priority queue.

```
list lock_list;
```

- Create a new variable at thread struct to store original priority of thread.

```
int original_priority;
```

- Create a point at thread struct point to a lock that blocked this thread.

```
struct lock *be_lock;
```

- Create a new variable at lock struct to record the highest priority thread at it's waiters list.

```
int biggest_priority;
```

- Create a new list_less_func at thread.c to compare two thread's priority.

```
bool priority_ordered (struct list_elem *a, struct list_elem *b)
```

- Create a new function at thread.c to denote priority recursively.

```
void thread_denote_priority (struct lock *l, struct thread *t);
```

- Create a new function at thread.c to update priority while thread release a lock.

```
void thread_update_priority (struct thread *t);
```

- Create a new list_less_func at synch.c to compare two element 's priority of waiters list.

```
bool waiter_priority_order (struct list_elem *a, struct list_elem *b)
```

- Modify **thread_yield**, **init_thread** and **thread_unblock** function, use **list_insert_ordered** to insert a element orderly rather than **list_push_back**.

```
list_insert_ordered (&ready_list, &t->elem, (list_less_func *) &priority_ordered, NULL);
```

- Modify **next_thread_to_run** function, sort the `ready_list` before pop the next thread.
- Modify **lock_acquire** function, update `lock_list` of the lock holder, update `be_lock` of current thread and call **thread_denote_priority** if blocked.
- Modify **lock_release** function, to remove the lock from `lock_list` and call **thread_update_priority**
- Modify **thread_set_priority** function, make set change `original_priority` first and if thread is not be donate or `original_priority` is bigger than `priority` set `priority` to `original_priority` .
- Modify **init_thread** to inital those new variable.
- Modify **cond_signal** and **sema_up** function , sort the waiters list before pop operate.

2. Algorithms

- Choosing next thread at priority order.
  First, use **list_insert_ordered** to insert thread in priority order.
  Then, use **list_sort** to sort `ready_list` before get the next thread. Last, call **thread_yield** function after create a new thread or reset current thread's priority.
- Acquire a lock.
  At **lock_acquire** function. Call **thread_denote_priority** to denote priority recursively.

```
void lock_acquire (struct lock *lock)
{
    If this lock had hold by other
        call thread_denote_priority function.

    Then call sema_down.
```

```
    Add lock to current thread's lock_list.
    Update lock's biggest priority.
    And set lock holder.
}
```

- Denote priority.
  At **thread_denote_priority** function. Change the priority of received thread and then if received thread block by another lock, call **thread_denote_priority** for then second lock. Then priority can be denote to the lock chin.

```
void thread_denote_priority(struct lock *l, struct thread *t)
{
    Change l's holder priority and lock's biggest priority if need.
    If l's holder was block by another lock
        call thread_denote_priority(l's holder be_lock, l's holder) angin.
}
```

- Release a lock
  At **lock_release** function. Except the original statements, we should remove lock form current thread's `lock_list` and update highest priority of this lock, then call **update_priority** to update thread priority.
- Update priority At **update_priority** function. Update priority of current thread, if current thread hold locks then set priority to biggest priority of `lock_list` element's `biggest_priority` and `original_priority` otherwise set it to `original_priority` .
- Change waiters list to priority queue. Sort the waiters list at **cond_signal** and **sema_up** function before pop operation. So that the first thread unlock will have the highest priority.

3. Synchronization

I will call **intr_disable** at every where it needed.

4. Rationale

This design might be work. It need about 300 line code.

Task 3: Multi-level Feedback Queue Scheduler

1. Data structures and functions

- Create two new variable at thread struct.

```
int nice;
fixed_t recent_cpu;
```

- Create a static variable at timer.c

```
static fixed_t load_avg;
```

- Create a new function at thread.c to increase `recent_cup`

```
void thread_increase_recent_cpu(void);
```

- Create a new function at thread.c to ipdate `priority`

```
void thread_update_priority_by_mlfqs(struct thread *t);
```

- Create a new function at thread.c to update `recent_cpu`

```
void thread_update_recent_cpu(strutc thread *t);
```

- Create a new function at timer.c to update `load_avg`

```
void thread_update_load_avg();
```

- Modify **timer_interrupt** function to update `priority`, `recent_cpu` and `load_avg` punctually.
- Modify **init_thread** to inital those new variable.

2. Algorithms

The algorithms to calculate new priority is

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

Recalculate priority for each thread at `all_list` every 4 tick.

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

Recalculate `recent_cpu` for each thread at `all_list` every 1 second.

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

Recalculate `load_avg` every 1 second.

- Call update on time.
  By modify **timer_interrupt** function. For each tick call **thread_increase_recent_cpu** to increase current thread's `recent_cpu` by 1
  for each 4 ticks call **thread_foreach** and **thread_update_priority_by_mlfqs** function to recalculate priority for every thread.
  for every `TIMER_FREQ` ticks call **thread_foreach** and **thread_update_recent_cpu** function to

recalculate `recent_cpu` for every thread.
for every `TIMER_FREQ` ticks call **thread_update_load_avg** function to recalculate `load_avg` .

3. Synchronization

`load_avg` is a static variable, so **timer_interrupt** function is disabling interrupts.

4. Rationale

I only consider one solution of this task. This design is easy too implement. I just need to use float operation at fixed_point.h to implement 3 formula. It need about 100 line code.

Design Document Additional Questions

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
0	0	1	2	63	60	58	A
4	4	1	2	62	60	58	A
8	8	1	2	61	60	58	A
12	12	1	2	60	60	58	A
16	16	1	2	59	60	58	B
20	16	5	2	59	59	58	B
24	16	9	2	59	58	58	A
28	20	9	2	58	58	58	A
32	24	9	2	57	58	58	B

Did not know how threads arrange if they have the same priority.

Tell us about how pintos start the first thread in its thread system (only consider the thread part).

thread_init will be called first to initial `tid_lock` , `ready_list` and `all_list` then call **running_thread** to get a point of current thread. Then call **init_thread** to initial current as main thread and set it's thread to `THREAD_RUNNING` and call **allocate_tid** to get a tidy for main thread.

Consider priority scheduling, how does pintos keep running a ready thread with highest priority after its time tick reaching TIME_SLICE?

To implement a priority scheduling we should call **thread_yield** at **thread_set_priority** and **thread_create**, then **thread_yield** function will sort the `ready_list` with priority order to chose the highest priority thread to run.

What will pintos do when switching from one thread to the other? By calling what functions and doing what?

schedule function will call **running_thread** and **next_thread_to_run** get current thread and next thread, then if current thread and next thread are not the same it will call **switch_threads** to switch next thread to running thread then call **thread_schedule_tail** to complete a thread switch and free resource.

What do priority-donation test cases(priority-donate-chain and priority-donate-nest) do and illustrate the running process

At donate nest, low thread will first acquire lock a then create a medium thread, medium thread acquire lock b and a then blocked by a, then donate priority 32 to low thread, then low thread create high thread and acquire b, then high thread donate priority 33 to medium thread, medium thread donate priority 33 to low thread. Then low thread release lock a, medium thread received lock a then release lock b and lock a, then high thread release lock b high thread finish, then medium thread finish, low thread finish.

At donate chain, first thread create 7 thread, and each thread was locked by previous. So each new thread will donate it priority throw the lock chain, and the highest priority thread will finish first then finish in priority decline order.

Pintos floating point number operation

It use int to represent float number. The higher 16 bits is integer part, lower 16 bits is float part. such as `FP_SHIFT_AMOUNT` = 4
FP_DIV(A,B)

$$4 = 0100, 0000 = 64$$

$$3 = 0011, 0000 = 48$$

$$64 * 16/48- > 21 = 0001, 0101 = 1.3125$$

FP_MULT(A,B)

$$4 = 0100, 0000 = 64$$

$$2.5 = 0010, 1000 = 40$$

$$64 * 40/16 = 160 = 1010, 0000 = 10$$