Rewrite “Time.c” -> Timer_sleep
We need to call thread_block() instead of thread_yield() ;
add a variable ticks_blocked to store the ticks after being blocked in thread.h and initialize it

add a function check_blocked_threads in thread.c which can check the ticks that a thread has been blocked. It decrements thread->ticks_blocked and unblocks it if thread->ticks_blocked == 0;

Implement a function to check how many times that the thread has been blocked every time. The best place to call this function is inside thread_foreach.

Task 1:
priority alarm
Problem: the current implement always put the thread at the end of the queue. There is a function called : list_insert_ordered can help us do this.

```
list_insert_ordered (struct list *list, struct list_elem *elem,
                     list_less_func *less, void *aux)
{
  struct list_elem *e;

  ASSERT (list != NULL);
  ASSERT (elem != NULL);
  ASSERT (less != NULL);

  for (e = list_begin (list); e != list_end (list); e = list_next (e))
    if (less (elem, e, aux))
      break;
  return list_insert (e, elem);
}
```

In order to use that function, we implement a function to satisfy the third argument list_less_func * less, which is an simple function that compares two elements in the queue and returns true or false.

use src/lib/kernel/list.c function: list_insert_ordered instead list_push_back when we push a

similarly, modify the list_push_back to list_insert_ordered in
Thread_unblock
Thread_yield

priority change and preempt
-> need to sort every set_prority
modify thread_set_priority current->priority <= new_priority and yield()

after create a new thread, yield if its priority < priority

Task 2

donate

The system will do below in order to pass the donate test.

One thread with a low priority lock can upgrade the lock’s priority until it releases the lock. The system should do this recursively.

If a thread has been donated when we want to set its priority, set the value original_priority instead. When the system doesn’t need the donation, the system can restore the value of original_priority into the priority.

When releases a lock, the system should compare the donate priority and the current priority.

The sema and condition queue need to be a priority queue.

When a thread

First, in the struct of thread, add a variable to store the base priority, two locks, one for current lock, one for the lock the thread is waiting. Also, add a list element to store the priority donation and a max_priority. If it holds a lock and its priority is not the biggest one, then donates its lock into the lock waiting queue. It will keep going until finding the max priority then the current thread should hold the lock.

Second, implement hold_the_lock function. It reuses the list_insert_ordered function to insert the lock to the queue. In order to use list_insert_ordered, we needed a function to compare two locks’ priority, which is similar to the previous task.

Second, modify the lock_acquire function which will keep tracking the current priority and max priority.

Third, complete the thread_donate_prority function. It is simple, just remove a thread from waiting list and put in into ready list, so we can reuse the list_insert_ordered as the previous task.

Forth, implement remove_lock function to remove a lock from the queue and update the priority. There is a list_remove function in the library.

Fifth, implement update_priority function. It uses the list_sort function to sort the lock queue, and it modifies the current thread lock priority to the max value.

Sixth, modified thread_set_priority which takes new_priority as parameter. It saves the old value in the base_priority that we added in the struct. If the new priority is greater than the old one, changes the current thread priority to the new one and

Last, modify the condition(cond_signal) queue to a priority queue that is similar to the thread_update. Similarly, update sema_up and sema_down.

Task 3 Scheduler
There is a pintos library for fixed-point calculation that is provided by other universities. We used the library to do the last task.

According to the equation: priority = PRI_MAX - (recent_cpu / 4) - (nice * 2).

Struct thread:
add integer nice to store nice value.
add a fixed point value recent_cpu

Globe variable @ thread.c
fixed point load_avg

Use the formula to calculate the priority, and the system will take the thread with highest priority.

Modify timer_interrupt function.
if it is a mlfqs thread, then increase recent cpu by one (function thread_mlfqs_recent_cpu_plusplus).
then
every Timer_Freq ticks run a function thread_mlfqs_update_load_avg_and_recent_cup
every 4 ticks run a function thread_mlfqs_update_priority.

thread_mlfqs_recent_cpu_plusplus: use the fixed point library function to add one to current_thread->recent_cpu.
thread_mlfqs_update_load_avg_and_recent_cup: update load_avg and recent_cpu of all threads
thread_mlfqs_update_priority: calculate the priority value

Last step is to complete thread_set_nice, thread_get_nice, thread_get_load_avg and thread_get_recent_cpu

Run test pass 26/27, mlfqs-recent-1 fails.