Report for project's implementation

Part-1: Getting started (running hello.c)

A new hello.c file is created in src/*tests/threads* and its entry for this file is made in tests.c, tests.h and Make.tests present in the same folder. Now running the command 'pintos run hello' prints "Hello Pintos".

Part-2: Pre-emption of threads (implement thread sleep)

The earlier implementation for sleeping threads, i.e. *timer_sleep ()* in *devices/timer.c* does the job but it busy waits. It keeps on yielding the thread that to the end of the *ready_list* as soon as it gets CPU.
To handle this situation efficiently, I made a new list for sleeping threads called *sleep_list defined below.*
struct list sleep_list;

Now when a thread is made to sleep, it is moved to this *sleep_list* instead of ready list and is blocked. So, the thread will not be scheduled now and we avoided busy waiting.

The main issue faced include:
The main challenge I faced for this part was to wake up the sleeping threads. It required a lot of effort to figure out how this could be done.
Since the sleep time is an integral multiple of ticks, this can be done using *timer_interrupt ()* function which is called after each tick. This is done using 8254 timer.

The *timer_interrupt ()* in turn calls the *thread_tick ()* function in *threads/thread.c* , in which we check threads in *sleep_list* if their sleeping ticks have completed. This is done by maintaining a thread's attribute defined below *(wake_up_tick)* . If so, we unblock the thread and move it back to the ready list so that it can be scheduled again.

int wake_up_tick; // defined inside thread's structure
In the thread structure, I included a thread an integer *wake_up_tick* which is the tick at which the thread has to wake up, which is equal to current tick value plus the number of ticks it has to sleep.

To efficiently wake up sleeping threads, the *sleep_list* is maintained to be in increasing order of *wake_up_tick* of threads.
This is done by inserting the thread elements in this list in their ordered position. It is implemented by using *list_insert_ordered ()* function defined in *list.h*.
By doing so we need not traverse the whole sleep_list at each tick, instead we check the first thread's *wake_up_tick,* if it's greater or equal to the current tick, we wake up the thread and check next thread and so on. If the *wake_up_tick* is less than current tick, we skip the entire list because other threads cant wake up now because they have larger *wake_up_tick.*

Alternative approach:
- We can make a new thread's state corresponding to a sleeping thread.

Part-3: Implementation of priority scheduling

When a thread is created using *thread_create ()*, we check if it's priority is greater than the running thread's priority. If so, we run the newly created thread and put the currently running thread to the ready queue i.e. *ready_list.*
This is implemented by using *check_preemption ()* which checks if the priority of new thread is greater than the current running thread's priority. If so, calls *thread_yield ().*

To implement the priority scheduling algorithm, we maintain the *ready_list* to be sorted at all times in decreasing order of priority and pick up the highest priority present in front of the list.
This is done by inserting the thread elements in this list in their ordered position. It is implemented by using *list_insert_ordered ()* function defined in *list.h*.
All *list_push_back ()* calls are replaced by *list_insert_ordered ().*

A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.
This is implemented by modifying the *thread_set_priority ().* After changing priority of the thread, it calls *check_preemption ()* which checks if the priority of new thread is greater than the current running thread's priority. If so, calls t*hread_yield ()*.

When threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first.
Every semaphore has a list shown below for threads waiting for that semaphore defined in *threads/synch.h*
struct list waiters;

Earlier implementation inserted new waiters to the back or the list when *sema_down ()* and pops out the front thread from the list when *sema_down ()* is called.
In my implementation, I maintained the waiters list sorted at all times by using *list_insert_ordered()* instead of *list_push_back ()* in *sema_down ().* So now when *sema_up ()* is called, the waiter popped from the waiters list will be the one with highest priority.
Also, when *sema_up ()* is called and a thread unblocks from the waiters list, we need to check if its priority is greater than the current thread's priority, so we call *check_preemption (),* which checks if the priority of thread that just unblocked is greater than the current running thread's priority. If so, calls *thread_yield ().*

For monitors, a wait on condition variable causes the thread to be pushed into waiter list of that condition.
When we issue condition signal, we need to sort the waiter list and pick up the thread having highest priority. This is done in function *cond_signal ().*

*Part-4: Advanced Scheduler (MLFQS) (**BONUS**)*

My implementation of MLFQS uses only one ready queue instead of 64.
This is possible as we pick up the thread with highest priority from the ready queue. When it yields, it is inserted to the ready queue at the end of it's priority. This way, a round robin scheduling is done between threads having highest priority.

This part was the hardest among the other parts as it requires lot of issues to be handled and lot of fixed point calculations.

The main issues faced include:

- Figuring out when we have to recalculate priorities, load average, recent cpu etc.
- Fixed-point calculations.
- Figuring out the order of calculations.

An integer value *niceness* is added to the thread's structure defined in *threads/thread.h* which is initialized to 0 in *init_thread (). nice* value determines how "nice" the thread should be to other threads.

1. **Calculating *load_avg***:
   System's load average, estimates the average number of threads ready to run over the past minute.
   It's calculation is implemented in the function *recalculate_load_avg ().*
   It's value is a function of *ready_threads,* the number of threads that are either running or ready to run at time of update. *ready_threads* is calculated by iterating over the *ready_list.*
   This function is called every second from *thread_tick (). load_avg* is a global variable and common for all threads which is initialized to zero.

2. **Calculating *recent_cpu:***
   Measures how much CPU time each process has received "recently."
   It is initialized to zero when a thread is initialized in *init_thread ().*
   It is a function of *load_avg* and the nice value of the thread.
   recent_cpu is defined for a thread, so we included an integer *recent_cpu* in the thread's structure defined in *threads/thread.h*
   load_avg defines the rate of decay for *recent_cpu*.
   Every second, *recalculate_recent_cpu ()* is called for each thread present in the *all_list.*
   Also, at each tick in *thread_tick (), recent_cpu* of the current thread is increased by one.

3. **Calculating priority:**
   The priority of each thread present in *all_list* is recalculated using the function *calculate_dynamic_priority ()* for each thread.
   This function is called at every fourth tick from *thread_tick ().*
   It is also called when niceness is changed in *thread_set_nice ()* and when new thread is created.
   Thread's priority is a function of *load_avg* and it's *recent_cpu.*

load_avg and *recent_cpu* are real numbers but Pintos does not support floating point numbers. So they are stored as fixed point 17.14 format.
Calculations are also done using this format. Normal integer is first converted to this format for calculations with 17.14 format.

For optimization purposes, bitwise operators are used where-ever possible in these calculations.

The functions used by test cases to get the variables stored as 17.14 fixed point format described above, e.g. *thread_get_load_avg (), thread_get_recent_cpu ()* returns the nearest integer rounded to that value.

Alternative approaches:

- 64 ready queues could be used instead of one and then we would have to determine the highest priority queue which is not empty. By doing the work in only one queue, we saved this overhead.

- Instead of a ready list, we could use a red-black tree. Then the highest priority thread would be at the rightmost of this red-black tree. When the priorities have to be recalculated, they are inserted to the tree again.