# Design Document for Project 1: Threads

## Task 1: Efficient Alarm Clock

### Data Structures and Functions

No additional data structure is necessary for the new `timer_sleep()` to be implemented. The necessary functions to be added will be `thread_current()->status = THREAD_BLOCKED;` when the thread is put to sleep and `ASSERT (t->status == THREAD_BLOCKED); t->status = THREAD_READY; list_push_back (&ready_list, &t->elem);` For the thread t to wake it up after `ticks * TIMER_FREQ --realtime >= timer_elapsed(start);`

### Algorithms

The thread will be changed to the blocked state when `timer_sleep()` is called with a positive argument. Once the correct amount of real-time has passed, it will be changed from blocked back to ready and added to the ready list. If `timer_sleep()` is called with a negative or zero argument, nothing will happen.

### Synchronization

None of the current synchronization techniques will be modified by the change in `timer_sleep().` Only the current thread will be put to sleep and will not go back to the ready state until after the timer is up. Since no synchronization items will be modified, it is safe to assume that the existing synchronization tactics are sufficient.

### Rationale

This design is an improvement upon the existing design since there is no busy waiting involved. The thread goes to the existing blocked state throughout the duration of the given timer, then switches back to the read state afterward. This solution was chosen since it has minimal synchronization modifications and only added existing code from thread.c to the `timer_sleep()` function.

## Task 2: Priority Scheduler

### Data Structures and Functions

#### New Data Structures

Two new data structures detailed below will be added: `queue` and `queue_element`.

```
struct queue
{
    struct queue_elem root; //The head of the queue
    int count; //Number of threads in the queue
};
```

```
struct queue_elem{
    struct queue_elem* left_child;
    struct queue_elem* right_child;
    struct queue_elem* parent;
    int priority;
    int original_priority; //Stores the base priority of the thread prior to any donation.
    int internal_donation; //Ensures that no threads suffer from starvation. When a thread is blocked or
waiting for a resource past a certain limit, it will increase and therefore modify the effective priority
 of the thread in question.
    int own_lock; //Records if a thread currently owns any locks.  If it is set to 0, the thread's priori
ty should not be affected by any locks.
};
```

The main data structure that will be implemented is a priority queue to store the various threads in the ready state with the thread with the highest priority available easily accessible. This priority queue will replace the doubly linked list described as `struct thread` in thread.c. Within the `queue` struct, each `queue_elem` will have possibly null members: a left and right child, parent, priority, the threads original priority, a counter of internal donation, and a counter for whatever locks the thread might currently own.

#### New Functions

Additionally, a few functions will be added to manage the priority queue and ensure that the thread with the highest priority is always at the root. ```

# define queue*entry(QUEUE*ELEM, STRUCT, MEMBER) //A macro to mimic list*entry, which is returning a thread pointer from a queue*elem

struct queue*elem *queue*root (struct queue* queue); //Return first element of a priority queue struct queue*elem *queue*pop (struct queue* queue); //Pop out the first element of a priority queue struct queue*elem *queue*insert (struct queue* queue); //Insert the element into the queue and reorder the priority queue to maintain the property of a max heap int queue_size(struct queue* queue); //Return the size a priority queue ```

### Algorithms

#### Selecting the next Thread to Run

Currently, the strategy pintos used to chose the next thread to run is calling the function `list_pop_front` which will pop an element from the ready list. However, the proposed modification will change the ready list from a linked list to the aforementioned priority queue to store the threads in the ready state. The function `list_pop_front` will be changed to `queue_pop` which will pop out an element with the highest priority if the priority queue is not empty or return *NULL* if it is empty.

#### Acquiring and Releasing a Lock

The implementation of `lock_acquire` , `lock_try_acquire` and `lock_release` will just call the functions `sema_down` , `sema_try_down` and `sema_up` .

`sema_up` adds one to the `semaphore.value` integer and, if the value greater than zero, will call the modified function `queue_pop` which will pop out a thread with the highest priority and unblock it.

`sema_down` and `sema_try_down` will be built to handle two different situations.

If an interrupt handler wants to interact with semaphores, it will call the `sema_try_down` function. If `semaphore.value` is 0, this

function will return false, and the interrupt handler will abort because an interrupt handler can't sleep or be added to a waiting list. Otherwise, it will decrement `semaphore.value` by 1 and return true.

If a normal thread wants to acquire a lock, it will call the `sema_down` function which decrements `semaphore.value` by 1. If `semaphore.value` is 0 at this point, the current thread will be added to the waiting priority queue and call `thread_block` to unblock itself and find the next thread to run.

### Computing the Effective Priority of a Thread

The effective priority of a thread is reliant on the original priority, set when the thread is created, and internal donation, modified when an originally low priority thread gets stuck waiting for a lock/semaphore for too long. Effective priority will follow the formula: `effective priority = priority + internal donation`

### Priority Scheduling for Semaphores and Locks

The priority scheduling for locks depends on interactions with semaphores. The previously described strategy of scheduling for semaphores as well as the `sema_down,` `sema_try_down,` and `sema_up` funtions will handle most cases. However, internal donation will be handled as follows. As it stands, `sema_up` will be called, and if the waiting priority queue has a nonzero amount of members, the thread with the highest priority will be popped out. Alternatively, threads with lower priorities have few chances to acquire locks/semaphores, and starvation may occur. In order to avoid this, every time after `queue_pop` is called, the value of `internal donation` in the thread's `queue_element` will increment by 1.

### Priority Scheduling for Condition Variables

In Pintos, condition variables combine semaphores and locks. Pintos maps one lock to multiple condition variables and uses semaphores, which are initialized to 0, to materialize a waiting list. Specifically, if one thread wants to wait for a condition variable, it calls the function `cond_wait` to block itself, add itself to the waiting list, and find the next thread to run. If a condition variable is ready, the current thread will call the function `cond_signal` to unblock the thread in waiting list.

### Changing a Thread's Priority

In changing the thread's priority, `int priority` will be modified in the thread struct. When a low priority thread acquires a lock while a high priority thread is also on the ready list and attempts to acquire that lock, a priority donation will occur. Both the high priority thread and low priority thread's priority level will be obtained by calling `int thread_get_priority (void)` and saved at `int temp_priority_high` and `int temp_priority_low` respectively. Then, the effective priority of the thread with the lower base priority will be set to `temp_priority_high` by calling `void thread_set_priority (int temp_priority_high).` After the lock is released, the thread that had its priority increased will be changed back to its initial priority level prior to the donation using `void thread_set_priority (int temp_priority_low).` However, in some circumstances, this method may encounter some unexpected problems. For example, a thread could never recover its original priority in the condition of nested priority donations. To combat this, the `own_lock` and `original_priority` members of the thread's `queue_elem` will be used to recover the original priority of a thread when all locks are released.

### Priority queue

The list struct from lib/kernel/list.c will be modified to become a priority queue. The time complexity of the doubly-linked list is Θ(n) for accesses and Θ(n) for searches. Since the priority queue will be used to pop the highest priority in each iteration, it will have time complexity of Θ(1) for accesses if no specific thread is called.

## Synchronization

### Multiple Threads Accessing the same Lock and Semaphore

When a thread is acquiring a lock/semaphore, it will disable interrupts until that thread successfully gets a lock or puts itself on the

waiting list. This means acquiring a lock/semaphore is atomic and there is no thread switching when a thread is trying to acquire a lock/semaphore. Therefore, acquiring a lock and semaphore is thread-safe.

### Accessing Shared Variables

There are two possible circumstances when different threads access a shared variable. In one scenario, a normal thread **A** is accessing a shared variable and will use a lock or a semaphore to synchronize this shared variable if necessary. If another thread **B** preempts **A**, it will encounter a lock and will put itself on the waiting list, block itself, and choose the next thread to run. In the second scenario, a normal thread **A** is accessing a shared variable and will use a lock or a semaphore to synchronize this shared variable if necessary. However, if an interrupt happens, some interrupt handler will run and encounter a critical section at which point it will try to acquire a lock. If it successfully acquires a lock, the handler will continue. Otherwise, the handler won't put itself on the waiting list and will try to continue to run without accessing the variable or aborting.

### Lists and other Data Structures

Lists and other data structures may not be thread sate in Pintos on their own, but a lock can be used to restrict multiple threads from modifying the same pointers simultaneously.

### Calling Functions

When two threads call the same function, if they do not access shared variables, they don't have any problems with regards to synchronization. However, if they did access a shared variable, a lock or semaphore will be used as described previously.

### Memory Deallocation

A thread will only be deallocated when the function `thread_schedule_tail` is called. In this function, the thread tagged *THREAD_DYING* will be released. Once the function exits, another thread could be tagged *THREAD_DYING*. `thread_schedule_tail` will be modified to prevent the memory of the running thread from remaining allocated.

## Rationale

A priority queue was chosen to store the threads due to the ease of creation and ability to efficiently provide the thread with the highest priority as described above. Others have proven that this priority queue will be no worse in terms of asymptotic space growth than the existing doubly linked list struct. The listed method of obtaining and acquiring locks helps to prevent data races and ensure that all threads can return to their original priorities following priority donation.

# Task 3: Multi-level Feedback Queue Schedule

## Data Structures and Functions

```
struct thread //Add recent_cpu for priority calculation;
static struct thread * next_thread_to_run(void); //Invoke fetch_thread() and enable mlfqs
static struct thread * running_thread(); //Change the ready list and enable mlfqs
void thread_unblock(struct thread *t); //Add thread into thread_lists
void thread_block(struct thread *t); //Add thread into blocked_list
static struct thread * fetch_thread(); //Scan through the thread list, then fetch the correct one
static void init_thread(struct thread*t, chonst char *name, int priority); Set the priority
void thread_schedule_tail(struct thread *prev); // Reset the priority then sent back to the thread_lists
int thread_get_nice(void);  // Get the thead`s nice value
void thread_set_nice(int new_nice); // Set the thread`s nice value
int thread_get_recent_cpu(void); //Get recent cpu time in moving average
int thread_get_load_avg(void); //Get loaded average
int load_avg(); //Hold the load average for calculation
```

The thread struct will have the new member recent_cpu in order to calculate the priority of the thread. All functions that rely on the structure of the thread ready list will be slightly modified to accommodate the flag to switch to using the MLFQS. Additionally, the nice values of threads must now be implemented in order to help determine how willing the threads are to give up their priority. New functions will also need to be created that calculate and keep track of the load average to ensure fairness among the threads.

## Algorithms

### Creating a New Thread

A new thread will be created the same way as it is with the default scheduler, but with an added nice value and added to the MLFQS list instead.

### Choosing the Next Thread to Run

The program will start with a scan through the list, mark the highest effective priority thread, and send it to the scheduler.

### Changing a Thread's Priority

When `schedule()` is called, if the thread has not finished executing, the priority will be re-calculated for the thread, and it will be added to the bottom of the list.

### Calculating the Priority

The load average (initially 0) can be calculated by calling `list_size` to get the quantity of threads in the ready list and plug it into the equation `load_avg = (59/60) * load_avg +(1/60) * list_size(&ready_list)`

The recent CPU time can be calculated using the nice value of a thread. After appropriately incrementing the `recent_cpu` variable, the effective priority can be determined by `priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)` Every second, `recent_cpu` will be calculated according to the formula
`recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu+nice.`

## Synchronization

When the kernel calls the scheduler, interrupts are turned off, so there is no need for considering any new synchronization issues. It is safe to assume that since Pintos runs without synchronization conflict prior to these modifications, it will continue to do so after them.

## Rationale

### Alternative design 1:

Create 64 lists, each responsible for 1 priority, when scheduling happens, the thread will be moved into the list that fits its priority. This design is useful for the given situation that many threads need to be handled (typically more than 64). However, since it is rare for there to be that many ready threads in Pintos, many of the lists may be empty in most of the time. Additionally, if this design is chosen, all the functions related with ready_list will need to be heavily modified, thus this is not the choice.

### Alternative design 2

Make more than one list (for example 2), and distribute threads into those lists by their priority. For example, threads with a priority higher than 31 should go to list 1, while the rest should go to list 0. Similarly, it takes less time to find a thread to run, but it takes up more memory, and a lot of code will need to be greatly altered to implement that, creating an unnecessarily complex program.

With the proposed design, it may be less efficient if there are too many threads in the kernel, but this situation is unlikely to occur in pintos. Since ready-to-work data structures are available, the existing structure of Pintos will remain mostly intact. Assume we have n threads in the kernel, the time complexity, and space complexity will all be O(n), since we have only one list, and we survey through the

ready list to find the next thread. We chose not to modify `next_thread_to_run()` directly, and instead modify `fetch_thread()`.

# Additional Questions

## 1. Effective Priority Test Case

Create 3 threads: `thread1` with priority 1, `thread2` with priority 2, and `thread3` with priority 3. Have `thread1` and `thread3` both re-acquire a lock to access a section that prints the name of the current thread. Additionally, `thread2` should print out its name right before it finishes executing. Ready `thread1` first and have it acquire the lock. Then, as soon as it acquires a lock, ready `thread2` and `thread3` and continue until all 3 until they have finished. The correct output should be `thread1 thread3 thread2` This is because `thread1` should accept a priority donation once `thread3` asks to acquire the lock that `thread1` currently has. Therefore, `thread1` will have its priority raised to 3 and finish executing the portion of its code that re-acquires the lock and prints `thread1` first. However, the actual output will be `thread2 thread1 thread3` This is because `thread2` will execute and print after `thread3` reaches the point where it is no longer ready and waits to acquire the lock from `thread1` since it will have the highest base priority remaining.

## 2. MLFQS Scheduler Table

Since there is a `#define TIMER_FREQ 100` in the timer.h, which means 100 timer ticks for a second, and `#define TIME_SLICE 4` in thread.c implements that 4 ticks for a time slice. Then there is 25 slices in one second, each slice are 40 ms length. For each time tick, the recent*cpu add by 1, which is shown on the table. For each second the recent*cpu is updated with the formula, which did not appear in the table.

| 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 | 12 | 0 | 0 | 61 | 60 | 59 | A |
| 16 | 12 | 4 | 0 | 60 | 60 | 59 | B |
| 20 | 16 | 4 | 0 | 60 | 59 | 59 | A |
| 24 | 16 | 8 | 0 | 59 | 59 | 59 | C |
| 28 | 20 | 8 | 0 | 59 | 59 | 58 | B |
| 32 | 20 | 12 | 0 | 59 | 58 | 58 | A |
| 36 | 20 | 12 | 4 | 58 | 58 | 58 | C |

## 3. Ambiguities

Since there is no specific policy when dealing with threads with the same priority level outlined in the spec, FCFS will be used to determine which thread goes first. All functions assume no timer tick is consumed during the calculation.