50007.1 - Laboratory - Task 1 - Design Document

Robbie Buxton, Jordan Hall, Bartłomiej Cieślar and Oliver Killane

21 October 2021

Group

Fill in the names and email addresses of your group members.

Name
Bartłomiej Cieślar
Jordan Hall
Oliver Killane
Robert Buxton

Preliminaries

Sources

1. "Sequential access in splay trees takes linear time" Tarjan, Robert E. (1985) [doi:10.1007/BF02579253]
2. "A Data Structure for Dynamic Trees" Sleator, D. D.; Tarjan, R. E. (1983) [doi:10.1145/800076.802464]
3. "Maintain subtree information using link/cut trees" You, Y. (2019) [https://codeforces.com/blog/entry/67637]

Latex Formatting

To make this document easier to read, we have implemented a consistent style:

C4 (5 marks)

This is the question's text...

...it can go over several lines!

We also have syntax highlighting for:

- Functions such as **schedule** or **init_thread**.
- Variables such as **num_ready_threads** or **thread_mlfqs**.
- Structures of type definitions such as **thread** or **ready_queue**.
- Constants and values set in define such as **BINARY_POINT** and **PRI_MAX**.
- Files such as **thread.c** and **fixed-point.h**.

We also include code directly from this repository, so the contents, line numbers and file titles all match their respective places.

thread.c

```c
163 /* Returns the number of threads currently in the multilevel queue system. */
164 size_t threads_ready(void)
165 {
166         return num_ready_threads;
167 }
```

This makes our document quicker and easier to read and mark.

Priority Scheduling

Data Structures

A1 (2 marks)

Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration.

Identify the purpose of each in roughly 25 words.

donation.h

```
void donation_lock_init(struct lock *lock);
```

donation_initialised is a flag for checking if the donation system has been enabled and is being used for avoiding a circular dependency between malloc, the initialization of the donation system for the main thread and several locks related functions. This requires that all locks that are acquired by the main thread must also be released before we can enable the donation system.

thread.h

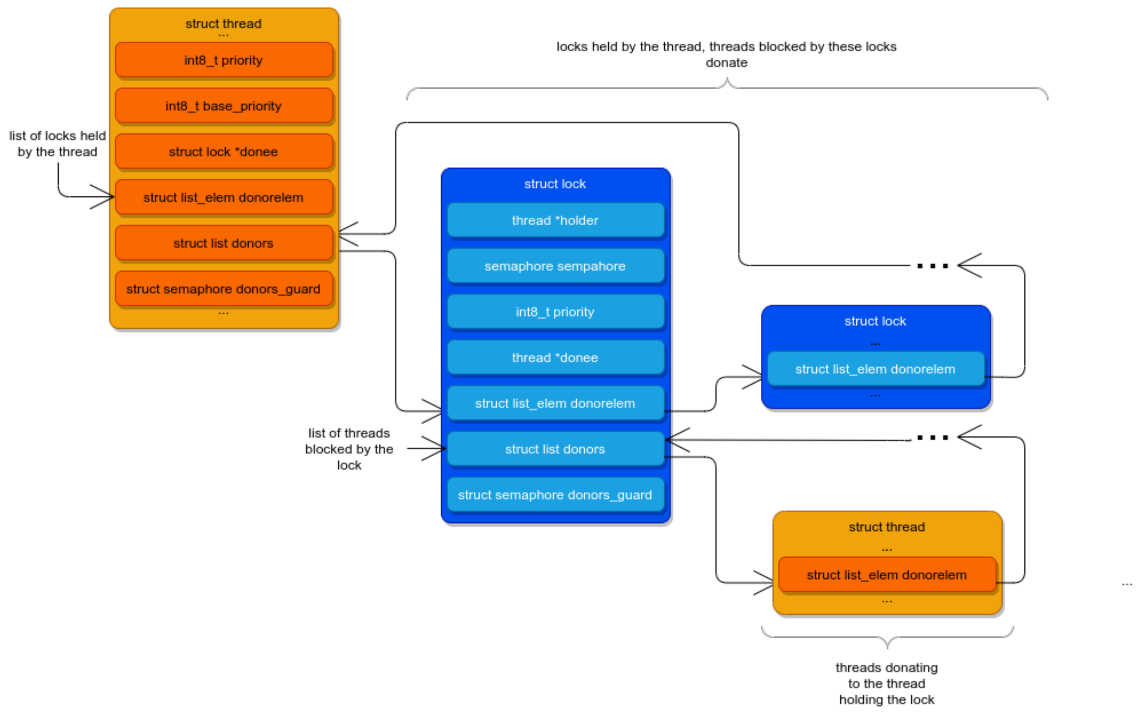
```
 * only because they are mutually exclusive: only a thread in the
 * ready state is on the run queue, whereas only a thread in the
 * blocked state is on a semaphore wait list.
 */
struct thread {
        /* Owned by thread.c. */
        tid_t tid; /* Thread identifier. */
        enum thread_status status; /* Thread state. */
        char name[16]; /* Name (for debugging purposes). */
        uint8_t *stack; /* Saved stack pointer. */
        int8_t priority; /* Thread's current computed priority */
        union {
                struct {
                        int8_t nice; /* Niceness ranges from 20 to -20 */
                        fixed32 recent_cpu; /* Measure of cpu usage */
                };
                struct {
                        /* Owned by donation.c */
                        int8_t base_priority; /* Thread's base priority */
                        struct lock *donee; /* The lock that receives the thread's
                            ↪ priority */
                        struct list_elem donorelem; /* Used in donee's list of donors
                            ↪ */
                        struct list donors; /* Locks donating their priority to the
                            ↪ thread */
                };
        };
        struct list_elem allelem; /* List element for all threads list. */
        /* Shared between thread.c and synch.c. */
        struct list_elem elem; /* List element. */

#ifdef USERPROG
        /* Owned by userprog/process.c. */
#ifndef NDEBUG
        bool may_page_fault; /* For debugging kernel */
#endif
```

The modified thread uses a union so that the thread can be used in both the mlfqs scheduler and the priority donation systems while retaining a reduced memory footprint. In the context of priority donation, we store the lock that the current thread is donating priority to, the element used in the donee's donor list, the current thread's donors, and a semaphore used in hand-over-hand locking for the priority structure.

A2 (4 marks)

Draw a diagram that illustrates a nested donation in your structure and briefly explain how this works.



struct thread
...
int8_t priority
int8_t base_priority
struct lock *donee
struct list_elem donorelem
struct list donors
struct semaphore donors_guard
...
list of locks held by the thread
locks held by the thread, threads blocked by these locks donate
struct lock
thread *holder
semaphore sempahore
int8_t priority
thread *donee
struct list_elem donorelem
struct list donors
struct semaphore donors_guard
list of threads blocked by the lock
struct lock
...
struct list_elem donorelem
...
struct thread
...
struct list_elem donorelem
...
threads donating to the thread holding the lock

Threads contain a list of donor locks. Each lock contains a list of the threads that are blocked by it. When a new thread is called lock_acquire and is blocked it is added to the lock's waiters. When a thread is blocked or updates its priority, it calls donation_cascading_update which propagates its priority to the lock blocking it, and from that lock to the thread holding the lock. This process continues until the max depth or a thread is not blocked by a lock.

Semaphores are used as guards for threads, to prevent race conditions when updating a thread's priority through donation, or thread_set_priority. Semaphores are also used to prevent race conditions from occurring when the donors lists of threads or locks are changed as threads are blocked, and start donating their priority.

Algorithms

A3 (3 marks)

How do you ensure that the highest priority waiting thread wakes up first for a lock, semaphore, or condition variable?

- **Semaphores**
  Locks are implemented using semaphores. **lock_release** calls **sema_up** internally, this means locks wake up the highest priority thread the same way semaphores do (see (ii)).

- **Locks**
  Sempahores store the list of waiting threads **semaphore.waiters**. When **sema_up** is called, if the waiters list is not empty we then get the **list_max** of the waiters. We use **priority_cmp** as the comparison function to get the highest priority waiter.

  Interrupts are disabled inside semaphores to prevent potential race conditions.

synch.c

```
        if (!list_empty(&sema->waiters)) {
                struct thread *max_waiter = list_entry(
                                                list_max(&sema->waiters,
                                                ↪ priority_cmp, NULL),
                                                ↪ struct thread, elem);
                list_remove(&max_waiter->elem);
                thread_unblock(max_waiter);
                thread_priority_yield();
        }
```

  (The semaphore wakes the highest priority waiter when signalled)

- **Condition Variables**
  A condition variable also stores the list of waiting threads **condition.waiters**. When **cond_broadcast** is called, the condition variable is repeatedly signalled until there are no waiting threads. When signalled (**cond_signal**) the highest priority waiter is woken and removed from waiters. This means the threads are woken up in order of their priority.

synch.c

```
 * make sense to try to signal a condition variable within an
 * interrupt handler.
 */
void cond_signal(struct condition *cond, struct lock *lock UNUSED)
{
        ASSERT(cond);
```

  (The condition variable wakes the highest priority waiter when signalled)

A4 (3 marks)

*Describe the sequence of events when a call to **lock_acquire** causes a priority donation.*

How is nested donation handled?

When **lock_acquire** is called we check the variable **donation_initialised** to see if we need to deal with priority donation. If the value is set to true then we call the function **donation_thread_acquire** passing in the lock and acquiring thread.

This function sets the donee of the lock as the thread. Next, the lock is added to the threads donors queue which holds all the locks donating priority to the thread. This function then calls **donation_cascading_update** passing in the lock.

This function handles the updating of priorities. It traverses down the chain of locks and threads until it reaches the **DONATION_MAX_DEPTH** or cannot go any further. It updates the threads and locks with a new priority if their current priority is smaller than the largest donating threads priority.

A5 (3 marks)

*Describe the sequence of events when **lock_release** is called on a lock that a higher-priority thread is waiting for.*

After checking that the current thread holds the lock with an assert, we call **donation_thread_release** passing in the lock. The thread is removed as the donee of the lock and has the lock removed from its donors list.

donation_thread_update_donee then updates the thread's priority to the maximum of the **base_priority** of the thread and all the priorities still being donated.

The lock is now released, and the highest priority waiter on the lock can now acquire it. We use semaphores as guards on both the lock and the thread, releasing it to prevent race conditions.

Synchronisation

A6 (2 marks)

*How do you avoid a race condition in **thread_set_priority** when a thread needs to recompute its effective priority, but the donated priorities potentially change during the computation?*

Can you use a lock to avoid the race?

The whole of the donation system hinges on the idea that, between the donees' guards being free, the list of donees for each node (thread/lock) is always ordered. This enforces some of the synchronisation requirements on the fields of the nodes which are:

When changing the priority, we need to hold the semaphore for the node's donors list and its donee's donors list. This is required by the fact that to update the priority, we need to get the largest donor from the node and after updating it we need to update that information in the donee's donors list. This is so our invariant holds.

Similarly, the donors list requires its node's guard to be acquired. The list elem representing a donor entry in the node's donee's donors list also requires a guard of the node's donee. The donee pointer of the node does not need to be protected, since it can only be changed by the thread that corresponds to it (if the node is a thread), or by the thread that is currently holding the lock (if the node is a lock). This way, in order to update the node's priority in **donation_update_priority**, called by the **thread_set_priority**, we only need to acquire the node's own donors list guard.

Rationale

A7 (3 marks)

Why did you choose this design? In what ways is it superior to another design you considered?
This design uses a simple implementation and it has good speeds for the system loads that pintos can experience with its memory limits. It also handles circular dependencies due to the close interleaving of the code in the very core of the operating system well.

- **Naïve approach**
  The first design we considered was that every time a priority needed to be fetched, we would go through the whole donation tree of threads that were donating to that node both directly or indirectly and take the maximum of all base priorities of those threads. We rejected this design from the get-go.

  The main issue with this approach is that, although its theoretical worst-case complexity is technically the same as the complexity of ours, once we add the optimization that we only update 8 levels deep into the priority donation, the number of threads that need to be accessed to update the top thread grows exponentially. Apart from that, the synchronisation cannot be directly hand-over-hand, which further slows down the code by not providing the same amount of fine-grained locking.

- **Link-cut trees**
  The primary problems of this solution are it being more complicated, requiring more testing and having more points of failure that could potentially cause the development not to be completed before the deadline.

  Another non-obvious problem with this approach is the complexity constant of link-cut trees[2,3]. This is due to the usage of splays[1] which although fast are still slower than if we limit our depth of priority update. Moreover, due to the need to use either a list-based priority queue or a splay one, it further slows it down over a simple for-loop of several iterations.

- **Backend for Priority Queue**
  We also had to choose the backend for the priority queue used in the nodes of the donation tree. We first settled on a heap-based approach which turned out to be flawed because of the circular dependency of locks used in a **malloc**. This would have been used in the heap, which would have been used by the donation system, which in turn was used by the locks.

  We then considered using splay trees[1] but like with the link-cut trees it could potentially have a higher run time over a linked list for the loads expected. This is due to more code and more branching, preventing good pipelining in the CPU. On top of that, it would require more code complexity, which could in turn hinder our ability to deliver the code on schedule.

Advanced Scheduler

Data Structures

B1 (2 marks)

Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration.

Identify the purpose of each in roughly 25 words.

fixed-point.h

```c
typedef struct fixed32 {
        int32_t val;
} fixed32;
```

The **fixed32** struct is used to wrap an int value. This can then be used in fixed-point arithmetic. We decided to wrap it in a struct so the C compiler would enforce strong type checking. In this way, an int is never conflated with a **fixed32**, and a **fixed32** is never conflated with an int.

thread.h

```c
 * only because they are mutually exclusive: only a thread in the
 * ready state is on the run queue, whereas only a thread in the
 * blocked state is on a semaphore wait list.
 */
struct thread {
        /* Owned by thread.c. */
        tid_t tid; /* Thread identifier. */
        enum thread_status status; /* Thread state. */
        char name[16]; /* Name (for debugging purposes). */
        uint8_t *stack; /* Saved stack pointer. */
        int8_t priority; /* Thread's current computed priority */
        union {
                struct {
                        int8_t nice; /* Niceness ranges from 20 to -20 */
                        fixed32 recent_cpu; /* Measure of cpu usage */
                };
                struct {
                        /* Owned by donation.c */
                        int8_t base_priority; /* Thread's base priority */
                        struct lock *donee; /* The lock that receives the thread's
                            ↪ priority */
                        struct list_elem donorelem; /* Used in donee's list of donors
                            ↪  */
                        struct list donors; /* Locks donating their priority to the
                            ↪ thread */
                };
        };
        struct list_elem allelem; /* List element for all threads list. */
        /* Shared between thread.c and synch.c. */
        struct list_elem elem; /* List element. */

#ifdef USERPROG
        /* Owned by userprog/process.c. */
#ifndef NDEBUG
        bool may_page_fault; /* For debugging kernel */
```

```
#endif
```

For MLFQS, **thread** was changed to add the variables **nice** and **recent_cpu**. These are added to a union with the data used in priority donation. These new entries are to be used in the multi-level feedback queue scheduler. This saves space in the size of **thread**.

thread.h

```
};
```

If **thread_mlfqs** is false (default) then we use the round-robin scheduler. Otherwise we use the multi-level feedback queue scheduler described in the PintOS specification.

thread.c

```
static fixed32 load_average;
```

load_average describes the estimate for the average number of ready threads over the last minute. It is calculated using the equations shown in the spec and then used in calculating **recent_cpu**.

thread.c

```
/* Array of all ready lists, each ready list at priority (index + PRI_MIN). */
static struct ready_queue ready_queue_array[1 + PRI_MAX - PRI_MIN];
```

ready_queue_array stores a **ready_queue** for each of the possible priorities. The priority of a ready queue at **index** is (**index** + **PRI_MIN**).

thread.c

```
/* Number of current ready threads (not running). */
static int32_t num_ready_threads;
```

num_ready_threads keeps track of the current number of ready threads that are currently waiting for CPU time. This is used for calculating **load_avg** as described in the PintOS specification.

thread.c

```
/* List of all non-empty ready_queues, in ascending order by priority.*/
static struct list nonempty_ready_queues;
```

non_empty_thread_queues is a list of all the non-empty thread queues stored in order of ascending priority. We can then poll the highest priority ready thread in constant time.

thread.h

```
        struct vector open_files; /* Vector of open file structs */
        struct list children; /* List of child processes */
        struct child_manager *parent; /* Struct managing the child process */
#ifndef VM
        struct file *exec_file; /* The program file the thread is running */
#else
        struct vector mmapings; /* Maps mmap ids to malloced lists of pages is uses
            ↪ */
```

ready_queue binds the ready queue for a certain priority, and a list elem so it may be placed in the **non_empty_thread_queues** and **ready_queue_array** structures.

Algorithms

B2 (3 marks)

Suppose threads A, B, and C have nice values 0, 1, and 2 and each has a ***recent_cpu*** *value of 0.*

Fill in the table below showing the scheduling decision, the priority and the ***recent_cpu*** *values for each thread after each given number of timer ticks:*

timer	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	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

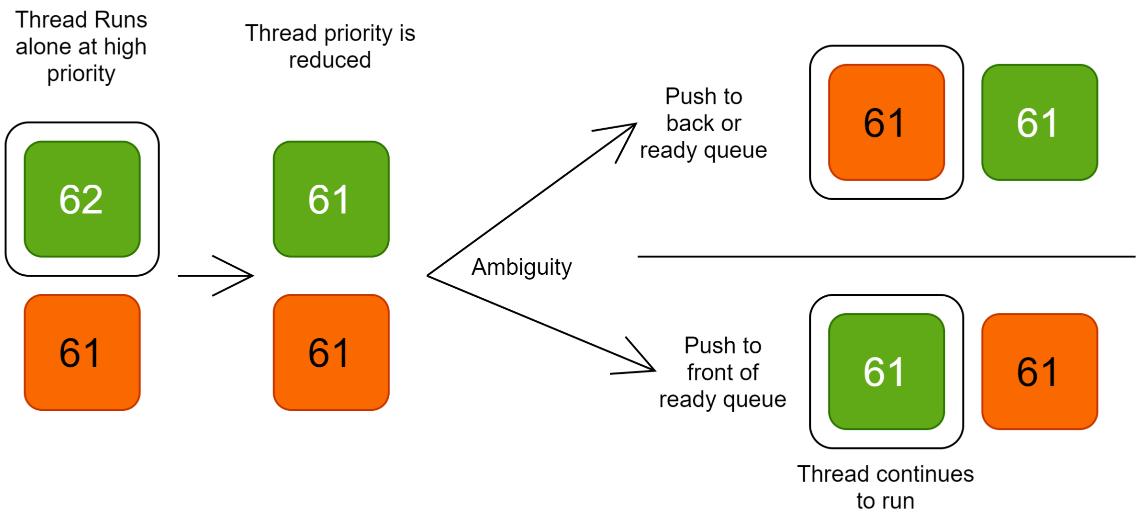
B3 (2 marks)

Did any ambiguities in the scheduler specification make values in the table uncertain?

If so, what rule did you use to resolve them?

When more than one thread has the highest priority, they are scheduled in a round-robin scheme.

When the highest priority thread has its priority reduced to that of another (or more) threads, it is moved to a new ready queue and where it should be added in the ready queue is ambiguous.


Thread Runs alone at high priority
62
61
Thread priority is reduced
61
61
Ambiguity
Push to back or ready queue
61
61
Push to front of ready queue
61
61
Thread continues to run

We decide to continue running a thread so long as its priority is greater than or equal to the highest priority of the other ready threads. When the thread yields it is inserted to the back of the ready queue associated with its priority.

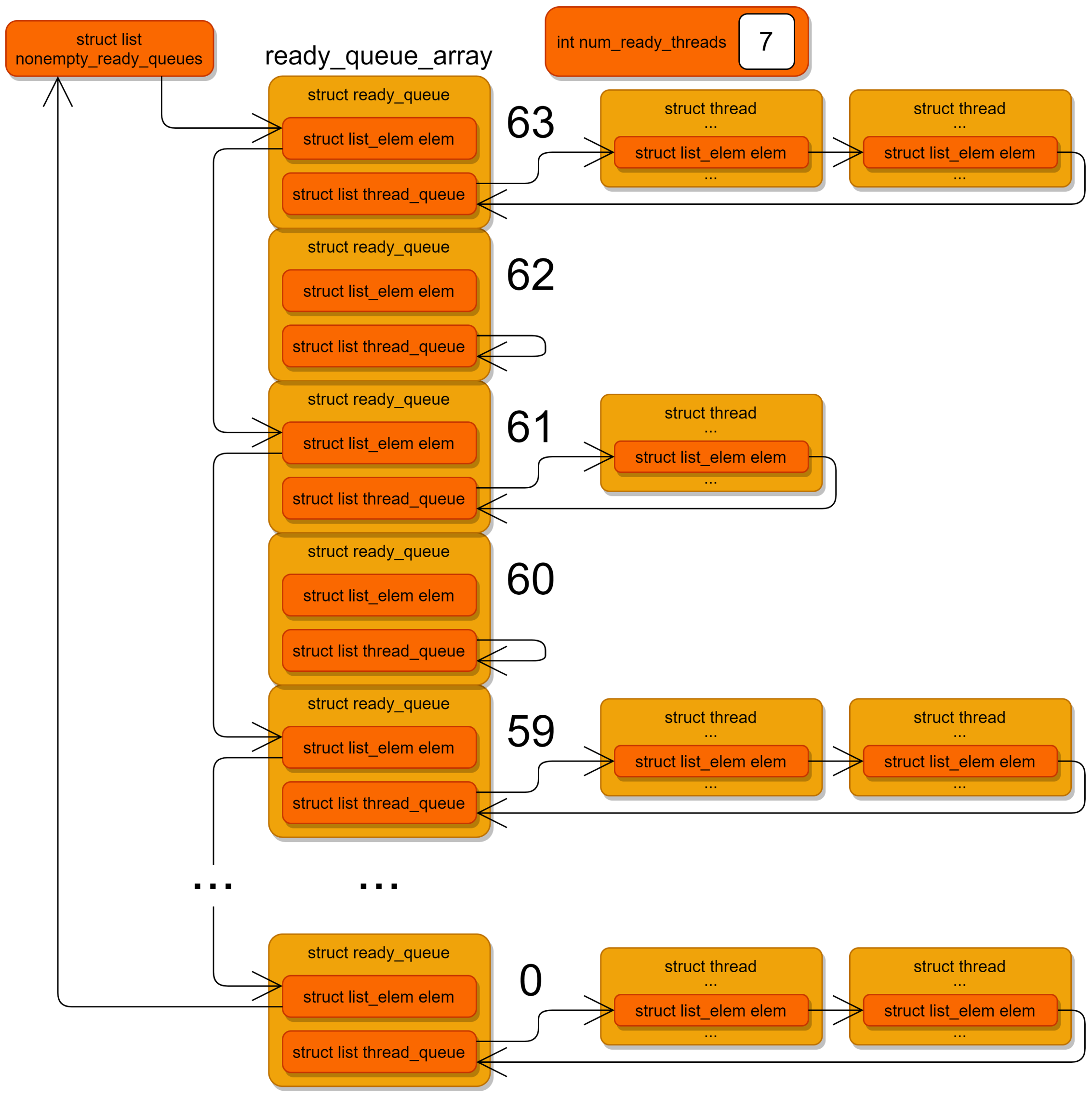
As a result, on the table when thread **A** is demoted from 62 $\rightarrow$ 61 it continues to be run until the time slice is over (it still has the highest priority). It then ends up at the back of the ready queue from priority 61, allowing **B** to be run.

Rationale

B4 (3 marks)

Briefly critique your design, pointing out advantages and disadvantages in your design choices.

Our implementation of fixed-point operations works by the following. There is a struct **fixed32** that stores a single element. The value should not be modified but instead, new **fixed32** structs should be initialized with the modified value. This makes the fixed point operations more complicated but has the added benefit that we can never confuse a fixed point value for an integer. If **fixed32** was a typedef of an integer, the C compiler would allow us to implicitly cast it to an integer. This would cause logical errors. This implementation allows us to do calculations between **fixed32** values and integers with added safety. This implementation has worked well for us.



We implemented the scheduler using an array of 64 **ready_queue**s, the index of the ready queue is equivalent to its priority. Every thread with status **THREAD_READY** is present in this system.

All non-empty ready queues were then linked together in a list, ordered by descending priority.

This approach had several advantages:

- **Insertion is fast**
  To insert a thread requires only three steps in **ready_add**:

  1. **list_push_back** the thread on the ready queue at the index of its priority.
  2. If the list was previously empty, insert the ready queue into the non-empty ready queues.

Our comparison function can use the memory location of the ready queue struct's **list_elem** directly for this.

3. Increment **num_ready_threads**.

As a result, the worst case scenario is when priorities 63 $\rightarrow$ 1 have at least one thread, priority 0 is empty, and we insert one thread into priority 0. This would require a traversal over 63 list elements with 63 direct comparisons of pointers.

- **Removal is fast**
  To remove a thread requires only three steps in **ready_remove**:

  1. Use **list_elem_alone** to determine if the thread is the last in its ready queue. If so use **get_list** to get the ready queue it is in, and **list_remove** it from the non-empty ready lists.
  2. Call **list_remove** on the threads **list_elem**.
  3. Decrement **num_ready_threads**.

  The worst case for this is when the only thread of a ready queue is removed. This is still very fast as **list_elem_alone** only checks two pointer values. **get_list** is very fast when there is a single element, and **list_remove** simply connects its adjacent elements by copying pointers.

- **Updating thread positions is fast**
  Updating a thread's position requires a **ready_remove** followed by a **ready_add**. As both are fast, so too are updates.

- **Getting the next thread is fast**
  We can simply take the first element of the first ready queue in **nonempty_ready_queues**.

- **Getting size is fast**
  As we use an **int** to keep track of the number of ready threads, we do not need to interact with our thread queue structure at all to get its size.

- **It is flexible**
  The size of the **ready_queue_array** is based off **PRI_MIN** and **PRI_MAX**, as is **ready_add**. This means priorities can be changed (e.g **PRI_MIN** = 42) without causing issues.

The disadvantage - additions to list.h:
To allow for fast removal of threads without traversing the list or making assumptions about a thread's priority, we must be able to remove the **ready_queue** it is contained in using the thread's **list_elem**.

To achieve this without exposing thread.c to the internal workings of list.c we added two new functions to list.c.

list.c

```
/* Determines if the list the elem resides holds only this element */
bool list_elem_alone(struct list_elem *elem)
{
        return elem->next && elem->prev && !elem->next->next && !elem->prev->prev;
}
```

list.c

```c
/* Returns the list the elem is part of. */
struct list *get_list(struct list_elem *elem)
{
        while (elem->next)
                elem = elem->next;

        return list_entry(elem, struct list, tail);
}
```