# Project 1: Threads

## Preliminaries

Team Number: 20

윤병준(20190766)

김치헌(20190807)

*We use GPT-4 to revise grammar errors in this report.*

## Table of Contents

* [Project 1: Threads](#project-1--threads)
  + [Preliminaries](#preliminaries)
  + [Table of Contents](#table-of-contents)
* [Analysis of the current implementation](#analysis-of-the-current-implementation)
  + [Threads](#threads)
    - [Overview of Thread life cycle implementation in PintOS](#X5638390f605920ac973ca68897bf5d9f84e128e)
    - [Thread Memory Layout](#thread-memory-layout)
    - [Thread State](#thread-state)
  + [Thread Switching](#thread-switching)
    - [Overview](#overview)
    - [schedule()](#Xb18318f20e674d0a8a727edb75f014d491f437c)
    - [switch\_threads()](#Xbf5d94f34413a8e369455631a21f4b7a1eb60d1)
    - [Special Case: Starting a Thread for the First Time](#X3128da9902ac386e5673f4a83e151572b3d0213)
  + [Synchronization](#synchronization)
    - [Disabling interrupts](#disabling-interrupts)
    - [Semaphore](#semaphore)
    - [Lock](#lock)
    - [Condition Variable](#condition-variable)
    - [Optimization barrier](#optimization-barrier)
* [Requirements](#requirements)
  + [Alarm Clock](#alarm-clock)
    - [Overview of Current Implementation](#overview-of-current-implementation)
    - [Data Structures](#data-structures)
    - [Algorithms](#algorithms)
    - [Implementation](#implementation)
    - [Rationale](#rationale)
  + [Priority Scheduling](#priority-scheduling)
    - [Overview of Current Implementation](#overview-of-current-implementation-1)
    - [Priority Inversion](#priority-inversion)
    - [Data Structures](#data-structures-1)
    - [Algorithms](#algorithms-1)
    - [Implementation](#implementation-1)
    - [Rationale](#rationale-1)
    - [Further Notes](#further-notes)
  + [Advanced Scheduler](#advanced-scheduler)
    - [Overview of Current Implementation](#overview-of-current-implementation-2)
    - [Data Structures](#data-structures-2)
    - [Algorithms](#algorithms-2)
    - [Implementation](#implementation-2)
    - [Rationale](#rationale-2)

# Analysis of the current implementation

## Threads

### Overview of Thread life cycle implementation in PintOS

The current implementation of threads in PintOS starts at main() in src/threads/init.c.

int main(void) {  
 /\* initialization \*/  
 thread\_init();  
 /\* Init memory system \*/  
 thread\_start();  
 /\* shutdown \*/  
 thread\_exit();  
}

thread\_init()’s primary purpose is to create the first thread for PintOS. A key feature of thread\_init() is to set the MAGIC value to the running thread. Details of the MAGIC value are explained at the top of thread.h. The MAGIC value is used to check whether the thread is valid or not. Since struct thread should not grow too large, its magic member will work as a validity checker.

thread\_start will create an idle thread (with thread\_create) and use sema\_down to prevent other processes from joining. Once the idle thread is created, it will sema\_up to release the last sema\_down from thread\_create. Then, it blocks itself, allowing the run\_action in main() to run another thread. The idle thread will wake up when there are no available threads to run.

thread\_create will create a new thread based on the given argument. The function receives a function name, priority, and an argument (aux). This function will allocate memory for a page for struct thread and stack frames for kernel\_thread and switching. Initially, init\_thread initializes a thread in the THREAD\_BLOCKED state. Just before returning, thread\_unblock() is called, setting the thread state to THREAD\_READY.

### Thread Memory Layout

Suppose thread\_create() is called with argument "alarm\_single". Then, the memory layout of the thread will be as follows:

struct thread locate at the very bottom of the page, the stack frame for kernel\_thread is located on the top of the page, and stack frame of called functions grow downward. If stack frame grows too large, it will overlap with the struct thread. Resulting in a memory corruption, which can be identified by checking the MAGIC value.

### Thread State

The state diagram of the thread life cycle has one exception: the THREAD\_RUNNING state can be initialized by thread\_init(), called by main() to set the first thread.

* THREAD\_READY: The thread is ready to run but isn’t running. Once the scheduler selects this thread, it will run next. Managed in ready\_list in src/threads/thread.c.
* THREAD\_RUNNING: The thread is currently running, and only one thread can be in this state.
* THREAD\_BLOCKED: The thread is blocked, waiting for an event such as a lock release or a semaphore to be upped.
* THREAD\_DYING: The thread will be destroyed soon.

This life cycle is defined in src/threads/thread.c as the enum thread\_status. This enum is stored in the status member of struct thread.

/\* States in a thread's life cycle. \*/  
enum thread\_status {  
THREAD\_RUNNING, /\* Running thread. \*/  
THREAD\_READY, /\* Not running but ready to run. \*/  
THREAD\_BLOCKED, /\* Waiting for an event to trigger. \*/  
THREAD\_DYING /\* About to be destroyed. \*/  
};

A key aspect of the thread life cycle is determining the order in which threads run next or how the OS manages the priority of threads in the ready\_list. The current implementation of ready\_list is as a FIFO list containing all threads ready to run.

## Thread Switching

### Overview

The above diagram shows the process of thread switching. The current thread is in the THREAD\_RUNNING state. If a thread has used up its time slice, thread\_tick() (called by timer\_interrupt) will call the function intr\_yield\_on\_return(). intr\_yield\_on\_return() change yield\_on\_return flag to true and let intr\_handler know. Then, intr\_handler will call thread\_yield() to switch to the next thread. thread\_yield() will call schedule() and switch\_threads() to switch to the next thread during interrupt.

### schedule()

schedule() is responsible for deciding which thread runs next. This function is invoked by: thread\_block(), thread\_exit(), and thread\_yield(). Before calling schedule(), interrupts should be disabled by using intr\_disable(). If interrupts are not disabled, an interrupt handler might be called during thread switching. schedule() chooses the next thread by invoking next\_thread\_to\_run() and switches from the current to the next thread.

thread\_schedule\_tail() completes the switching process. It sets the current thread to the THREAD\_RUNNING state and starts a new time slice. If the thread being switched from is in the THREAD\_DYING state, we free the allocated memory, which includes struct thread and the stack frame. We free the memory after switching because we need the information in this memory for switch\_threads(). Freeing the memory before switching would prevent access to this necessary information.

### switch\_threads()

Within switchs.S, we find the assembly code responsible for thread switching. The CUR thread is the one currently executing, while the NEXT thread is the one we’ll be switching to. The purpose of switch\_threads() is to save the state of CUR and restore the state of NEXT. Let’s break down the process:

1. **Save Registers**

pushl %ebx  
pushl %ebp  
pushl %esi  
pushl %edi

The registers ebx, ebp, esi, and edi are callee-saved registers. It’s imperative that a called function preserves their values.

1. **Save Current Stack Pointer**

mov thread\_stack\_ofs, %edx # uint32\_t thread\_stack\_ofs = offsetof(struct thread, stack); from thread.c  
movl SWITCH\_CUR(%esp), %eax  
movl %esp, (%eax,%edx,1)

The thread\_stack\_ofs represents the offset of the stack member in the struct thread. SWITCH\_CUR is the offset from the struct switch\_threads\_frame to the cur member. The operation movl SWITCH\_CUR(%esp), %eax yields a pointer to the current thread’s struct thread. The last line here saves the current stack pointer to the stack member of the current thread.

1. **Restore Next Stack Pointer**

movl SWITCH\_NEXT(%esp), %eax  
movl (%eax,%edx,1), %esp

Similarly, movl SWITCH\_NEXT(%esp), %eax results in a pointer to the next thread’s struct thread. Adding thread\_stack\_ofs, now stored in %edx, yields a pointer to the stack member of the next thread. The final line restores the stack pointer of the next thread. From now, esp points to the stack of next page. Instruction from now is in context of the next thread, and cur page(context) is freezed.

1. **Restore Registers**

popl %edi  
popl %esi  
popl %ebp  
popl %ebx

In the context of the next thread, restore registers using pop instructions. 5. **Return to Caller**

ret

If the thread is the first thread, we will discuss this case in the next section. If not, the ret instruction will restore eip to point right after of the switch\_threads() call in schedule().

### Special Case: Starting a Thread for the First Time

The initialization of a thread poses a unique challenge. Specifically, the stack pointer for the thread hasn’t been set yet. As noted, switch\_threads() requires information about the previous stack to switch to the next thread correctly. However, an initial thread lacks this prior stack. Thus, it becomes necessary to initialize certain stack frames for the first thread. This initialization is performed in thread\_create(), found in src/threads/thread.c.

tid\_t thread\_create(const char \*name, int priority, thread\_func \*function, void \*aux) {  
 struct thread \*t;  
 struct kernel\_thread\_frame \*kf;  
 struct switch\_entry\_frame \*ef;  
 struct switch\_threads\_frame \*sf;  
 ...  
}

At the top is the switch\_threads\_frame. Its eip points to switch\_entry(), as defined in switch.S. This means that the next function to be called will be switch\_entry().

The subsequent frame is switch\_entry\_frame, whose task is to invoke thread\_schedule\_tail() for the first time. A more detailed description follows:

# Implementation of switch\_entry() in `switch.S`  
addl $8, %esp # discard switch\_threads() arguments: cur and next  
pushl %eax # push SWITCH\_CUR(%esp) to the stack, which will be the argument for thread\_schedule\_tail()  
 # SWITCH\_CUR(%esp) points to the current thread's struct thread  
call thread\_schedule\_tail   
addl $4, %esp # clean up stack

The switch\_entry function aids switch\_threads(). Its main job is to discard the arguments to switch\_threads() and then call thread\_schedule\_tail().

Lastly, we have kernel\_thread\_frame, which calls function with aux as its argument. Within kernel\_thread(), interrupts are enabled, function is called with the argument aux, and finally, thread\_exit() is invoked.

In summary, the roles of kernel\_thread\_frame, switch\_entry\_frame, and switch\_threads\_frame are to establish the correct execution environment when creating thread. This setup ensures to initialize thread can seamlessly and invoke thread\_schedule\_tail().

## Synchronization

### Disabling interrupts

When dealing with synchronization problems, ensuring that preemption does not occur while executing the atomic function is crucial. So, it is necessary to prevent the preemption from occurring. In pintos, the preemption occurs when the thread\_ticks gets greater than TIME\_SLICE. The thread\_ticks is increased by the thread\_tick(), called by the external interrupt handler timer\_interrupt(). Thus, turning off the interrupt can be the solution. The interrupt can be controlled manually via the functions defined in src/interrupt.c. Specifically, disabling interrupts can be done by intr\_disable() and rollback interrupts by intr\_set\_level().

### Semaphore

A semaphore is a synchronization primitive invented to control access to a common resource shared by multiple threads. The semaphore S is a variable that holds an integer value, and this value can be accessed and edited by only two operators, P and V. The integer value held by the semaphore is the number of currently available units. A thread calls P before accessing the critical section(a shared resource that two or more threads should not access). If the semaphore value is over 1, it decreases the value and enter the critical section. If not, the thread waits until the value is positive, then it enters the critical section, decreasing the value. A thread calls V after completing the operation in the critical section. V increases the semaphore value.

Pintos implemented a semaphore as follows.

struct semaphore {  
 unsigned value; /\* Current value. \*/  
 struct list waiters; /\* List of waiting threads \*/  
};

A semaphore is initiated by sema\_init(). For P and V, sema\_up() and sema\_down() is implemented.

* sema\_init(): Set an initial value for the variable value and initiate a list, waiters.
* sema\_down(): P. If the value is 0, the caller thread is pushed into the waiters by list\_push\_back() and blocked by thread\_block(). When the value becomes positive, the caller thread decreases the value and returns to access the critical section.
* sema\_up(): V. If waiters is not empty (i.e., there are threads that want to access the critical section), the front thread is popped by list\_pop\_front(). Then, the popped thread takes access to the critical section. Regardless of waiters, sema\_up() increases the value.

According to the semaphore definition, it seems normal to initialize the semaphore’s value to a positive number depending on how many resources are available. However, we can find that pintos initialize the value with zero and then call sema\_down(). In this case, the caller (i.e., the thread that initialized the semaphore and called sema\_down()) immediately pushed into waiters and then blocked. As the caller whose state was running is blocked, another thread in the ready list is selected and becomes the following running thread. The selected running thread does its job and then calls sema\_up() so the previous running thread can be unblocked and go to the ready list. A typical example is the initializing step of the idle thread. The main thread initializes the semaphore idle\_started with value 0 and calls sema\_down to pass the control flow to the idle thread. After the function idle() is started, the idle thread calls sema\_up() to give the control flow back to the main thread.

There are other functions for semaphore in pintos, sema\_try\_down() and sema\_self\_test().

sema\_try\_down() is similar to sema\_down() as it is a kind of P function. However, it works only when the value is positive. In other words, if the value is 0, it returns false, and the caller neither enters the waiters nor becomes blocked. If the value is positive, it decreases the value by one and returns true.

sema\_self\_test() is a function for self-testing.

### Lock

A Lock is a synchronization primitive like a semaphore. It is used for controlling access to a shared resource like semaphore. The semaphore’s P(or “down”) is “acquire” in the lock, and the V (or “up”) of the semaphore is “release” in the lock.

There are two main differences between a lock and a semaphore.

* One is the range of the value can have. A Semaphore can have various values larger than 1, but a lock’s value can only be 0 or 1. In other words, a lock is a binary semaphore.
* Another is the presence of the lock holder. In semaphore, there is no owner or holder. It means a thread can “up” the semaphore without being the thread “downed” it. In contrast, a lock can only be “released” by the thread that “acquired” it.

Pintos implemented a lock as follows.

struct lock {  
 struct thread \*holder; /\* Thread holding lock \*/  
 struct semaphore semaphore; /\* Binary semaphore controlling access \*/  
};

As a lock performs like a binary semaphore, it has a semaphore as a member variable and a holder for the thread holding the lock. A function lock\_init() initializes a lock. There are four functions to implement the lock features.

* lock\_init(): Initialize holder and semaphore. Initially, there is no holder, so initialize holder with NULL. As the lock is the binary semaphore, semaphore is initialized with the value 1.
* lock\_acquire(): Checks whether the holder is the caller(it must not), and then invoke sema\_down(). Then, set the holder to the current thread.
* lock\_release(): Checks whether the holder is the caller(it must be), and then set the holder NULL. Then invoke sema\_up().
* lock\_try\_acquire(): Lock version of sema\_try\_down(). Acquire the lock only when the holder is NULL.
* lock\_held\_by\_current\_thread(): lock\_acquire() and lock\_release() use this function to check the holder. It returns true if the holder is the current thread. Otherwise, it returns false.

### Condition Variable

A condition variable is a synchronization primitive designed to solve a synchronization problem. A semaphore and a lock can handle the concurrency problem by limiting the number of threads to access the critical area. However, they do not focus on controlling the order in the interdependent threads. The condition variable defines how the dependent thread gives the control flow to the responsible thread and takes it back.

If a thread(Thread A) wants to access the shared resource after another thread(Thread B) works on that shared value, it uses a condition variable. In this case, the “condition” is “Does Thread B finish its work?”. Thread A waits for the condition to become true, and Thread B signals to Thread A if the condition is satisfied. Here is a brief explanation of how the condition variable works using a lock and a semaphore.

* First, Thread A needs to lock the common resource and acquire to prevent other threads from accessing.
* Then, it prepares a semaphore with an initial value of 0 to block itself until the common resource is ready.
* After preparing the semaphore, it releases the lock and blocks itself using the prepared semaphore so that Thread B can access it.
* The Thread B acquires the lock and does the work needed on the common resource.
* When Thread B finishes the job, it sends a signal to the blocked thread, Thread A, which is the V operation of the semaphore.
* Finally, it can release the lock for Thread A.

Pintos implemented a condition as follows.

struct condition {  
 struct list waiters; /\* List of waiting threads \*/  
};

The list waiters holds the struct semaphore\_elem elements. semaphore\_elem is defined like below.

struct semaphore\_elem {  
 struct list\_elem elem; /\* List element \*/  
 struct semaphore semaphore; /\* This semaphore. \*/  
};

condition is initialized by cond\_init() and performs its functionality with three functions, cond\_wait(), cond\_signal() and cond\_broadcast().

* cond\_init(): Initialize the list waiters.
* cond\_wait(): Receives a lock as an argument and has a local variable semaphore\_elem waiter.

This lock is the lock already acquired by the caller.

After calling cond\_wait(), the waiter semaphore initialized with the value 0 and list\_push\_back() into waiters. Then the lock released, and the waiter semaphore downed to make the thread wait for the “signal”. As the lock is released, it can be acquired by other threads. The thread which acquires the lock will send the signal to the waiting thread. The “signal” will make the semaphore up, and after receiving the signal, the thread re-acquires the lock.

* cond\_signal(): Called by the current thread, which acquires the lock released in cond\_wait(). It makes the first waiting semaphore up (i.e., make the waiting thread ready). After calling cond\_signal(), the caller can release the lock.
* cond\_broadcast(): If there is more than one waiter in the waiters, the caller can send signals to every waiter in waiters using this function.

### Optimization barrier

An optimization barrier is a barrier to the compiler not optimizing or reordering across it. Without the barrier, the compiler can optimize and reorder the instructions, resulting in a buggy program because the order of the instructions plays an important role in the synchronization problem.

# Requirements

## Alarm Clock

We should reimplement the timer\_sleep() function defined in device/timer.c.

### Overview of Current Implementation

/\* Sleeps for approximately TICKS timer ticks. Interrupts must  
 be turned on. \*/  
void timer\_sleep (int64\_t ticks) {  
int64\_t start = timer\_ticks ();  
  
ASSERT (intr\_get\_level () == INTR\_ON);  
while (timer\_elapsed (start) < ticks)  
thread\_yield ();  
}

The current implementation of the time\_sleep() is “busy wait”. In this case, we can call it “busy sleeping”. In the last two lines of the code, we can see the thread still sleeping is being pushed by calling the thread\_yield(). If this thread becomes the running thread before the ticks ticks elapsed, it will just call the thread\_yield() and go back to the ready list because it will still be busy sleeping. The sleeping threads will do nothing but call thread\_yield(). This implementation has room for improvement.

### Data Structures

We defined a new list and its element structure to manage the sleeping thread effectively.

static struct list sleep\_list;  
  
struct sleep\_list\_elem {  
 struct list\_elem elem; /\* List element \*/  
 int64\_t end\_tick; /\* Tick to wake up \*/  
 struct semaphore semaphore; /\* Semaphore to block a sleeping thread \*/  
};

### Algorithms

Sleeping algorithms

* A structure sleep\_list\_elem variable(says sleep\_elem) is generated by the thread that is called the new timer\_sleep().
* The thread insert sleep\_elem into the sleep\_list in ascending order of end\_tick.
* Then the thread calls the sema\_down() to sleep.
* The semaphore covers the sleeping thread until the sema\_up() is called.

Awakening algorithms

* For every tick, the timer interrupt handler will check whether there is a sleep\_list\_elem who holds the thread that needs to wake up at the current tick.
* As every sleep\_list\_elem is sorted in ascending order of end\_tick, we can check the front element.
* If the end\_tick of the front element is bigger than the current tick, we end the search.
* Otherwise,
  + We list\_pop\_front() and sema\_up() the semaphore of the popped sleep\_list\_elem.
  + Iterate until finding the sleep\_list\_elem whose end\_tick is bigger than the current tick.

### Implementation

Initializing the sleep\_list will be added to the thread\_init().

We need a new function in src/thread.c to implement the sleeping algorithms above. The new function thread\_sleep() will look like the following code and be called by the timer\_sleep().

void thread\_sleep(int64\_t end\_tick) {  
 /\* Define `sleep\_list\_elem` variable and initialize it \*/  
 ...  
  
 /\* Insert `sleep\_list\_elem` into the `sleep\_list` \*/  
 /\* We need to define `list\_less\_function`, `less()`, to use the `list\_insert\_ordered()` \*/  
 ...  
  
 /\* Semaphore down. (i.e., Start sleeping) \*/  
 ...  
}

Then the new timer\_sleep() is defined as below.

void timer\_sleep(int64\_t ticks) {  
 int64\_t start = timer\_ticks;  
   
 ASSERT (intr\_get\_level () == INTR\_ON);  
 thread\_sleep(start+ticks);  
}

To implement the awakening algorithm, we need to modify thread\_tick() or define a new function called by timer\_interrupt(). In any case, we need to have the code look like the following code. In this case, we define a new function named thread\_wakeup().

void thread\_wakeup(int64\_t current\_tick) {  
 /\* Define a placeholder for iterating \*/  
 ...  
 /\* While loop until the `sleep\_list` empty \*/  
 while(!list\_empty(sleep\_list)) {  
 /\* Get the front element \*/  
 ...  
 /\* Break the while loop if the element's `end\_tick` is greater than `current\_tick` \*/  
 if ( ... )  
 break;  
 /\* Else, pop front from the `sleep\_list` and call sema\_up for its `semaphore` \*/  
 ...  
 }  
}

Then the thread\_wakeup() will be added in timer\_interrupt().

static void timer\_interrupt(struct intr\_frame \*args UNUSED) {  
 ticks++;  
 thread\_tick();  
 thread\_wakeup(ticks);  
}

### Rationale

This implementation has some advantages.

The first one is that we do not have to change the member of the thread structure. It is essential to keep the thread structure simple. Because the more extensive the structure becomes, the lesser the memory space the thread can have. And not adding a member about sleeping to the thread structure fits logically since only some threads will sleep.

The second one is that we reduced the overhead of finding the threads to wake up. There are two methodologies we can choose for the sleep\_list. The first is to sort when inserting, then find efficiently. The second is to push back first, then search for it later. Both have pros and cons in some situations. The first approach has more overhead in inserting, while the second has more in finding. We chose the first approach because, in this situation, finding appears more frequently than inserting. (we have to find the thread to wake up every single tick!)

## Priority Scheduling

### Overview of Current Implementation

The current implementation of priority scheduling is based on the ready\_list in FIFO order. It doesn’t consider the priority of the thread. According to the state diagram of the thread life cycle, a thread is pushed to the ready\_list when thread\_unblock() and thread\_yield() are called. Popping a new thread from the ready\_list is done by schedule(). Both thread\_unblock() and thread\_yield() call list\_push\_back() to push a thread to the end of the ready\_list. schedule() calls next\_thread\_to\_run() to pop a thread from ready\_list. next\_thread\_to\_run() returns the first entry of the ready\_list. In summary, the current implementation of priority scheduling is based on the naive FIFO queue, which doesn’t account for the priority of the thread.

### Priority Inversion

Priority inversion is a problem that occurs when a high-priority thread is waiting for a CPU resource that is currently being used by a low-priority thread. A root cause of this is a lock used to protect a shared resource. Assume there are three threads: H, M, L, with priority such that H > M > L. If H is waiting for a lock A that is currently held by L, then M can preempt L. At this point, H is in the waiting list of lock A, and L is in the ready list. After M finishes its work, it yields the CPU to L, which is in the ready list. L will run and release lock A. Then, H will be able to run. Based on the priorities of the threads, H should run first. However, due to priority inversion, the run order of the threads becomes M -> L -> H.

One solution to this is **priority donation**. When a thread is waiting for a lock, it donates its priority to the holder of the lock. In the above example, H donates its priority to L. Thus, L will have the highest priority, followed by H and M.

Multiple donations occur when there are several instances of donation. If L holds locks A and B, and M wants to lock A, then M will donate its priority to L and go to the waiting list of lock A. Next, if H wants to lock B, it donates its priority to L. This scenario is referred to as multiple donation.

Nested donation is a case of recursive priority donation. The previous example’s donations are between M -> L and H -> L. Nested donation is a scenario like H -> M -> L. As a precondition, L is holding lock A, and M is holding lock B and requires lock A. If H needs lock B, it donates its priority only to M. M still needs lock A. However, lock A is held by L, which hasn’t received any donation. To address this situation, we need a recursive donation of priority, moving from H -> M and then from M -> L.

### Data Structures

To achieve priority scheduling, we need to manage the ready\_list in a sorted order. Since ready\_list is implemented as a list in PintOS, we can use list\_insert\_ordered() to insert a thread into the ready\_list in sorted order. For this, new data structures are not needed.

However, to address the priority inversion issue, we need to introduce some new members to struct thread to manage priority donation.

struct thread {  
 ...  
 int original\_priority; /\* Original priority of the thread \*/  
 struct lock \*waiting\_lock; /\* Lock that the thread is waiting for \*/  
 struct list donations; /\* List of donations to handle multiple donations \*/  
 struct list\_elem donation\_elem; /\* List element for donation list \*/  
 ...  
}

### Algorithms

* **Manage ready\_list in a sorted manner**: One effective way is to keep the ready\_list sorted when a thread is pushed into it. This can be achieved by using the pre-implemented function list\_insert\_ordered(). This function will insert a thread into the ready\_list in a sorted order.
* **Acquiring a lock**: When a thread tries to acquire a lock, the following steps are taken:

function lock\_acquire(struct lock \*lock)  
 check if the lock is already acquired by another thread  
 if yes,  
 add the lock to the thread's waiting list  
 add the thread to the donation list  
 donate the priority to the holder of the lock  
   
 set the thread's waiting lock to nil  
 acquire the lock with sema\_down

* **Donating priority**: When a thread donates its priority to another thread, the following steps are taken. A recursive loop to delve deeper will resolve the nested donation issue.

function donate\_priority()  
 doner := current thread  
 donee := doner's waiting lock's holder  
 loop   
 if donor's waiting lock is nil  
 break  
 if doner's priority is greater than donee's priority  
 donee -> priority := donor -> priority  
 donor := donee  
 donee := donor's waiting lock's holder  
 if changed priority of donee in ready list  
 sort ready list

* **Releasing a lock**: When a thread releases a lock, the following steps are taken.

function lock\_release(struct lock \*lock)  
 remove the lock from the thread's waiting list  
 remove the thread from the donation list  
 refresh the thread's priority based on the highest priority in the donation list  
 release the lock

* **Changing priority in a running thread**: When a thread’s priority is changed, the following steps are taken.

function thread\_set\_priority(int new\_priority)  
 if the current thread has no donations,   
 set priority to new\_priority  
 else   
 set original\_priority to new\_priority  
   
 if new\_priority is lower than the current priority  
 and if next\_thread\_to\_run's priority is higher than the current priority  
 yield the CPU

* **preemption**: When a thread is preempted, the following steps are taken.

function preemptive\_yield()  
 if (  
 interrupts are not disabled  
 and ready list is not empty  
 and current thread's priority is lower than the next thread's priority  
 )  
 yield the CPU

### Implementation

The following are the key functions for our implementation:

* **Managing the ready\_list**:

list\_insert\_ordered(&ready\_list, &current\_thread->elem, thread\_priority\_cmp, NULL);  
  
static bool thread\_priority\_cmp(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux) {  
 return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority;  
}

* **Function for lock\_acquire()**:

void lock\_acquire(struct lock \*lock) {  
 struct thread \*cur = thread\_current();  
 struct thread \*holder = lock->holder;  
  
 if (holder != NULL) {  
 cur->waiting\_lock = lock;  
 list\_insert\_ordered(&holder->donations, &cur->donation\_elem, thread\_priority\_cmp, NULL);  
 donate\_priority(cur, holder);  
 }  
  
 sema\_down(&lock->semaphore);  
 cur->waiting\_lock = NULL;  
 lock->holder = cur;  
}

* **Function for donate\_priority()**:

void donate\_priority() {  
 bool is\_in\_ready\_list = false; // if the donee is in the ready list, sort ready\_list  
 struct thread \*priority\_doner = thread\_current();  
 struct thread \*lock\_holder = doner->waiting\_lock->holder;  
 // Init variables  
 // TODO: Address the depth of the nested search  
 while (priority\_doner->waiting\_lock != NULL) {  
 if(lock\_holder-> status == THREAD\_READY) {  
 is\_in\_ready\_list = true;  
 }  
 if(priority\_doner->priority > lock\_holder->priority){  
 lock\_holder->priority = priority\_doner->priority;  
 priority\_doner = lock\_holder;  
 lock\_holder = priority\_doner->waiting\_lock->holder;  
 }  
 }  
 if (is\_in\_ready\_list) {  
 list\_sort(&ready\_list, thread\_priority\_cmp, NULL);  
 }  
}

* **Function for lock\_release()**:

void lock\_release(struct lock \*lock) {  
 struct thread \*cur = thread\_current();  
 struct thread \*holder = lock->holder;  
  
 for (struct list\_elem \*e = list\_begin(&holder->donations); e != list\_end(&holder->donations); e = list\_next(e)) {  
 struct thread \*t = list\_entry(e, struct thread, donation\_elem);  
 if (t->waiting\_lock == lock) {  
 list\_remove(e);  
 }  
 }  
 refresh\_priority(cur);  
  
 lock->holder = NULL;  
 sema\_up(&lock->semaphore);  
}

* **Function for refresh\_priority()**:

void refresh\_priority(struct thread \*t) {  
 t->priority = t->original\_priority;  
 if (list\_empty(&t->donations)) {  
 t->priority = t->original\_priority;  
 return;  
 }   
   
 struct thread \*max = list\_entry(list\_max(&t->donations, thread\_priority\_cmp, NULL), struct thread, donation\_elem);  
 if (max->priority > t->original\_priority) {  
 t->priority = max->priority;  
 }  
}

* **Function for preemtive\_yield()**:

// Add this function to where two conditions occurs, change or priority in running thread. new thread added to ready list  
// such as, thread\_create(), thread\_set\_priority(), sema\_up()  
void preemtive\_yield() {  
struct thread \*current = thread\_current();  
struct thread \*next = list\_entry(list\_front(&ready\_list), struct thread, elem);  
if (!intr\_context() && !list\_empty(&ready\_list) && current->priority < next->priority) {  
thread\_yield();  
}  
}

### Rationale

The aforementioned algorithm and implementation are based on the assumptions that:

1. The ready\_list is sorted in ascending order of priority.
2. waiters in the semaphore are sorted based on the ascending order of thread priority.

There are two possible approaches for list management:

1. **Always maintain the list in a sorted manner**: This approach leverages pre-implemented functions like list\_insert\_ordered().
2. **Pop elements from the list in a sorted manner but maintain an unsorted list internally**: While this approach might seem efficient, it requires additional overhead to ensure the list’s order during operations.

Both approaches have the same time complexity: inserting into a sorted list and popping in a sorted manner both have a complexity of O(n). Given the benefits of using the pre-implemented functions in list, we chose to always maintain the list in a sorted manner.

### Further Notes

There are some additional notes to consider:

* Related to lock\_release()
  + A thread will be unblocked by sema\_up(). A current implementation of sema\_up() works as FIFO. Thus, we need to add sorting process based on the priority of the thread to sema\_up()
* Related to ready\_list
  + As we mentioned above in the rationale, we need to maintain the ready\_list in a sorted manner. But, there exists edge case that the ready\_list is not sorted. Like change of priority of thread in the ready\_list due to priority donation. To handle this case, we check whether lock holder is in ready\_list, and sort the ready\_list if it is in the ready\_list.
* Related to cond variables
  + cond also use waiters list. So, we need to sort the waiters list based on the priority of the thread. We can use list\_insert\_ordered() to sort the waiters list.

## Advanced Scheduler

### Overview of Current Implementation

We need to newly implement the advanced scheduler, Multilevel Feedback Queue Scheduler(MLFQS). which is not implemented yet.

### Data Structures

To implement MLFQS we need an array of 64 ready\_list. Each entry is for each level of priority. If we implement each ready\_list separately, we need to implement the function that pop every thread from every ready\_list and re-arrange every thread to the ready\_lists again, then call this function every time the priorities calculated. However, sorting a single ready\_list every time we re-calculate the priorities does the same effect. So, we are going to use the same single ready\_list.

We need to append two new members to the thread structures, nice and recent\_cpu to calculate the priority. Also, we need a new global variable load\_average to calculate the recent\_cpu.

We are going to use fixed-point representation for recent\_cpu and load\_avg which are real numbers. It will be less confusing, if we define a new type.

typedef int fixed\_t;  
  
struct thread {  
 ...  
 int nice;  
 fixed\_t recent\_cpu;  
 ...  
};

The nice is the value that indicates how the thread is “nice” (i.e., tends to yield the cpu). The nice is an integer value and can have -20 ~ 20. The default nice value is zero.

The recent\_cpu is the value that indicates how much the process had a cpu time “recently”. The recent\_cpu is calculated with “Exponentially weighted moving average”. The initial value of the recent\_cpu is zero.

The load\_average is the value that indicates the moving average of the size of the ready list. The initial value of the load\_average is zero.

### Algorithms

Selecting the next thread to run

* Find from the highest level ready\_list which is not empty.
* If there are more threads than one, use Round Robin strategy. (i.e. FIFO with limited time slice)

Calculating the priority every 4 ticks

* priority = PRI\_MAX - (recent\_cpu / 4) - (2 \* nice)
* if priority is bigger than PRI\_MAX or smaller than PRI\_MIN, needs adjust.
* Sort ready\_list after calculate the priority.

Calculating the recent\_cpu every second

* recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice
* recent\_cpu of the running thread (except the idle thread) is increased by one every tick.

Calculating the load\_avg every second

* load\_avg = (59/60) \* load\_avg + (1/60) \* size(ready\_list)

### Implementation

Disable thread\_set\_priority() if the thread\_mlfqs is set.

void thread\_set\_priority(int new\_priority) {  
 /\* Should not modify the priority with thread\_set\_priority() if the `thread\_mlfqs` is set. \*/  
 ASSERT(!thread\_mlfqs);  
 ...  
}

Disable priority donation if the thread\_mlfqs is set.

void lock\_acquire(struct lock \*lock) {  
 if (thread\_mlfqs) {  
 /\*\* Default lock\_acquire implementation \*\*/  
 return;  
 }   
 /\*\* New lock\_acquire implementation with priority donation \*\*/  
}  
  
void lock\_release(struct lock \*lock) {  
 if (thread\_mlfqs) {  
 /\*\* Default lock\_release implementation \*\*/  
 return;  
 }   
 /\*\* New lock\_release implementation with priority donation \*\*/  
}

Fixed point calculations

* As the kernel of the pintos doesn’t support the float point calculations, we need fixed point calculations to deal with recent\_cpu and load\_avg which are real numbers. For 32-bit representation, we have 1 bit for signs, 17 bits for integer part and 14 bits for decimal part. We need some revise to get a correct calculation using fixed point representation, and each equation is in the appendix of the pintos.

Implementation of calculations

void calculate\_priority(struct thread\* t) {  
 if (t != idle\_thread) {  
 int nice = t->nice;  
 fixed\_t recent\_cpu = t->recent\_cpu;  
 t->pritority = fp2int(add\_n(div\_n(recent\_cpu, 4), PRI\_MAX - t->nice\*2));  
 if (t->priority > PRI\_MAX) t->priority = PRI\_MAX;  
 else if (t->priority < PRI\_MIN) t->priority = PRI\_MIN;  
 }  
}  
  
void increase\_recent\_cpu(struct thread\* t) {  
 if (t != idle\_thread) {  
 t->recent\_cpu = add\_n(recent\_cpu, 1);  
 }  
}  
  
void calculate\_recent\_cpu(struct thread\* t) {  
 if (t != idle\_thread) {  
 fixed\_t load\_avg\_mul\_2 = mul\_n(load\_avg, 2);  
 t->recent\_cpu = add\_n(mul\_x(div\_x(load\_avg\_mul\_2, add\_n(load\_avg\_mul\_2, 1)), t->recent\_cpu), t->nice);  
 }  
}  
  
void calculate\_load\_avg(void) {  
 int size = list\_size(ready\_list);;  
 fixed\_t c1 = div\_n(int2fp(59), 60);  
 fixed\_t c2 = div\_n(int2fp(1), 60);  
 load\_avg = add\_x(mul\_x(c1, load\_avg), mul\_n(c2, size));  
}

Implementation for sort ready\_list. We need this function in thread.h. Because the ready\_list isn’t reachable in the timer.c.

void sort\_ready\_list(void) {  
 sort\_list(&ready\_list, thread\_priority\_cmp, NULL);  
}

New implementation of timer\_interrupt()

void timer\_interrupt(void) {  
 ticks++;  
 thread\_tick();  
 thread\_wakeup(ticks);  
   
 if (thread\_mlfqs) {  
 increase\_recent\_cpu(thread\_current());  
 if (ticks % TIMER\_FREQ == 0) {  
 calculate\_load\_avg();  
 thread\_foreach(calculate\_recent\_cpu);  
 }  
 if (ticks % 4 == 0) {  
 thread\_foreach(calculate\_priority);  
 sort\_ready\_list();  
 }  
 }  
}

Getters and Setters for the new variables

int thread\_get\_nice(void) {  
 /\*\* disabling interrupts \*\*/  
 ...  
 int nice = current\_thread()->nice;  
 /\*\* re-enabling interrupts \*\*/  
 ...  
 return nice;  
}  
  
void thread\_set\_nice(int nice) {  
 /\*\* disabling interrupts \*\*/  
 ...  
 struct thread \*t = current\_thread();  
 t->nice = nice;  
 calculate\_priority(t);  
 preemptive\_yield();  
 /\*\* re-enabling interrupts \*\*/  
 ...  
}  
  
/\*\* Note   
 \* Returned value is the rounded value of the recent\_cpu multiply by 100   
\*\*/  
int thread\_get\_recent\_cpu(void) {  
 /\*\* disabling interrupts \*\*/  
 ...  
 fixed\_t recent\_cpu = thread\_current()->recent\_cpu;  
 int result = fp2int\_round(fp\_mul\_n(recent\_cpu, 100));  
 /\*\* re-enabling interrupts \*\*/  
 ...  
 return result;  
}  
  
/\*\* Note:   
 \* Returned value is the rounded value of the load\_avg multiply by 100  
\*\*/  
int thread\_get\_load\_avg(void) {  
 /\*\* disabling interrupts \*\*/  
 ...  
 int result = fp2int\_round(fp\_mul\_n(load\_avg, 100))  
 /\*\* re-enabling interrupts \*\*/  
 ...  
 return result;  
}

### Rationale

If a priority of a thread is fixed from the begging and never changes, we need to deal with the problems like the priority invasions. The priority donation can solve this problem, but we need to consider various cases to implement the correct priority donation.

As the priority invasion occurs because of the fixed priority, we can also solve it by dynamically calculate the priority of the existing threads. MLFQS is one of the those methods using the dynamic priority system.

With the nice and recent\_cpu with well-constructed weight, we can make the operating system automatically avoid the priority invasions.