Project 1: Threads

Preliminaries

Team Number: 20

윤병준(20190766)

김치헌(20190806)

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

Table of Contents

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();
}
```

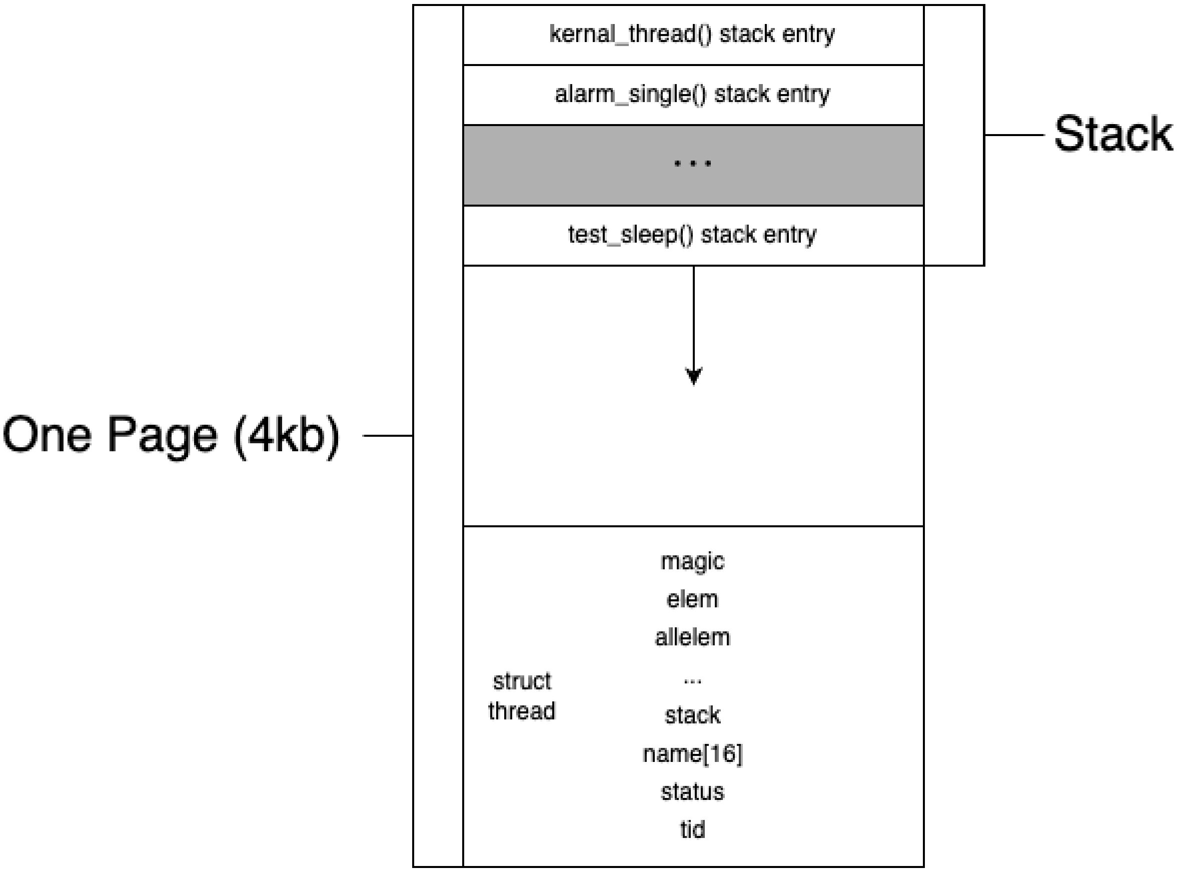
`thread_init()` 's primary purpose is to create the first thread for PintOS. A key feature of `thread_init()` is to set the MAGIC value to the running thread. Details of the MAGIC value are explained at the top of `thread.h`. The MAGIC value is used to check whether the thread is valid or not. Since `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` .

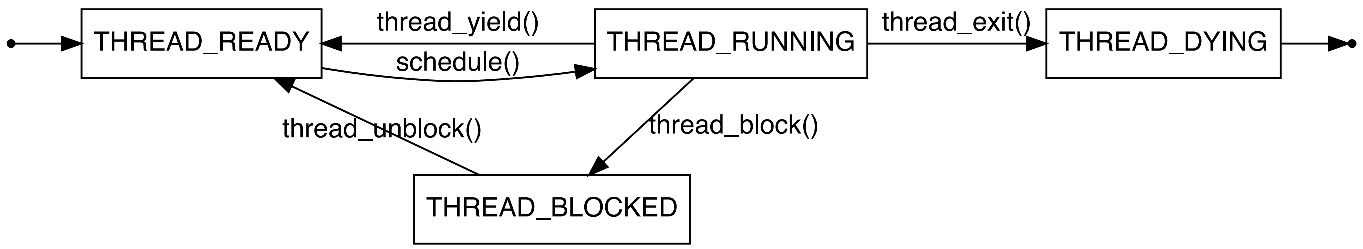
Thread Memory Layout

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.

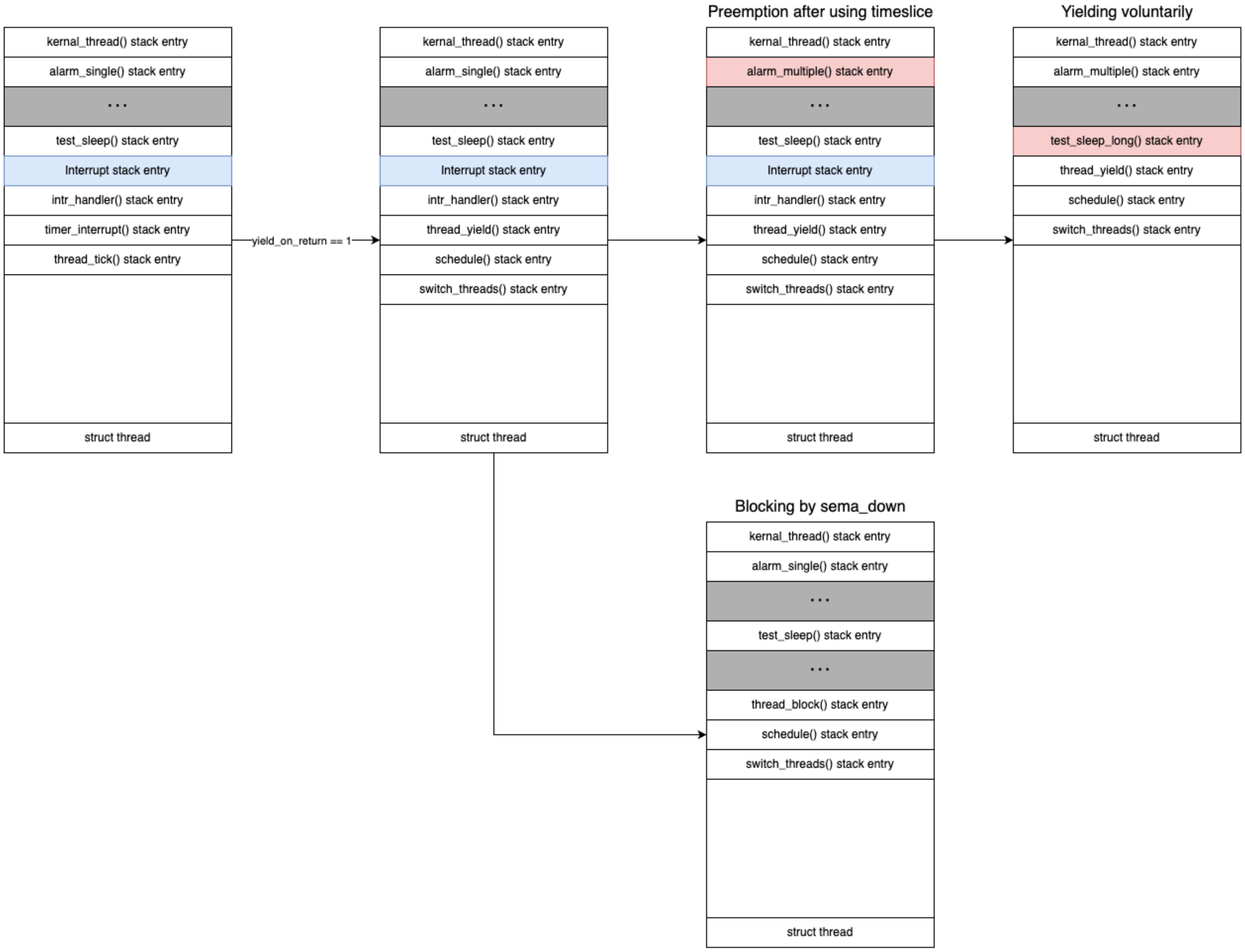
This life cycle is defined in `src/threads/thread.c` as the enum `thread_status` . This enum is stored in the `status` member of `struct thread` .

```
/* 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


kernal_thread() stack entry
alarm_single() stack entry
...
test_sleep() stack entry
Interrupt stack entry
intr_handler() stack entry
timer_interrupt() stack entry
thread_tick() stack entry
struct thread
yield_on_return == 1
kernal_thread() stack entry
alarm_single() stack entry
...
test_sleep() stack entry
Interrupt stack entry
intr_handler() stack entry
thread_yield() stack entry
schedule() stack entry
switch_threads() stack entry
struct thread
Preemption after using timeslice
kernal_thread() stack entry
alarm_multiple() stack entry
...
test_sleep() stack entry
Interrupt stack entry
intr_handler() stack entry
thread_yield() stack entry
schedule() stack entry
switch_threads() stack entry
struct thread
Yielding voluntarily
kernal_thread() stack entry
alarm_multiple() stack entry
...
test_sleep_long() stack entry
thread_yield() stack entry
schedule() stack entry
switch_threads() stack entry
struct thread
Blocking by sema_down
kernal_thread() stack entry
alarm_single() stack entry
...
test_sleep() stack entry
...
thread_block() stack entry
schedule() stack entry
switch_threads() stack entry
struct thread

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.

2. **Save Current Stack Pointer**

```
# uint32_t thread_stack_ofs = offsetof(struct thread, stack); from thread.c
mov thread_stack_ofs, %edx
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.

3. **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.

4. **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, cmp_thread_priority, NULL);

static bool
cmp_thread_priority(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() {
  // if the donee is in the ready list, sort ready_list
  bool is_in_ready_list = false;
  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_list`s 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.