



# Lab1 Design Report



Member #1

Name: Suwon Yoon

Student ID: 20210527



Member #2

Name: Jiwon Lee

Student ID: 20210706

## **Table of Contents**

### **0. Current Code Analysis**

#### **0.1 timer.c / timer.h**

##### **0.1.1 Structures**

##### **0.1.2 Functions**

##### **0.1.3 Enums**

##### **0.1.4 Variables**

#### **0.2 thread.c / thread.h**

##### **0.2.1 Structures**

##### **0.2.2 Functions**

##### **0.2.3 Enums**

##### **0.2.4 Variables**

### **1. Analysis of Current Thread System**

#### **1.1 Overview**

#### **1.2 thread\_init**

#### **1.3 thread\_start**

##### **1.3.1 Switching Threads**

##### **1.3.2 fake stack frame**

##### **1.3.3 idle thread**

#### **1.4 Extra Details**

### **2. Analysis of Synchronization Primitives**

#### **2.1 Overview**

#### **2.2 Motivation of Synchronization Primitive**

**2.3.1 Initialize value with '0'**

**2.3.2 Initialize value with '1'**

**2.3.3 Something to Check with**

**2.3.4 Functions**

**2.4 Lock**

**2.4.1 Functions**

**2.4.2 example of using lock**

**2.5 Monitors**

**2.5.1 Condition**

**2.5.2 Monitor Lock**

**2.5.3 Fucntions**

### **3. Alarm Clock Design Plan**

**3.1 Current Implementation**

**3.2 Our Design Plan**

### **4. Priority Scheduler Design Plan**

**4.1 Current Implementation**

**4.2 Important Concepts**

**4.3 Our Design Plan**

### **5. Advanced Scheduler Design Plan**

**5.1 Current Implementation**

**5.2 Important Concepts**

**5.3 Our Design Plan**

# 0. Current Code Analysis

## 0.1 timer - timer.c / timer.h

### 0.1.1 Overview

timer는 system timer로, 시스템의 정기적인 interrupt를 관리하고, thread를 깨우거나 재우는 등의 작업을 수행하는 과정에서 활용된다. Pintos의 scheduling 및 concurrency 관련 기능에 있어 중요한 역할을 한다.

### 0.1.2 Macros

#### TIMER\_FREQ

초당 timer interrupt의 수를 정의하는 매크로다.

### 0.1.3 Variables

#### ticks

OS가 부팅된 이후로 지난 timer tick수이다.

#### loops\_per\_tick

한 timer tick당 CPU가 얼마나 많은 비어있는 루프를 수행할 수 있는지 측정한 값이다.

### 0.1.4 Structures

특별히 정의된 구조체가 없다.

### 0.1.5 Functions

#### timer\_init

이 함수는 시스템의 timer에 대한 설정을 초기화하는 역할을 한다.

시스템 부팅 시 호출되어 timer 관련 하드웨어 및 소프트웨어를 초당 **TIMER\_FREQ** 만큼 timer interrupt가 일어나도록 설정한다.

#### timer\_calibrate

이 함수는 시스템의 timer를 보정하는 역할을 한다.

주어진 시간 내에 CPU가 얼마나 많은 비어있는 루프를 수행할 수 있는지를 측정하여 해당 값을 **loops\_per\_tick**에 저장한다. 이 값은 그 후 시스템에서 시간 지연을 생성할 때 사용될 수 있다.

## timer\_ticks

이 함수는 시스템이 시작된 이후 발생한 timer interrupt의 총 수, 즉 tick수를 반환하는 함수다.

특이하게 `intr_disable()` 과 `intr_set_level(old_level)` 를 사용하는 부분이 있다. 이는 tick을 보기 전에 interrupt 상태를 `old_level` 에 저장하고, interrupt를 비활성화하기 위해서이다. 이렇게 하는 이유는 `ticks` 는 글로벌 변수로서 interrupt handler에 의해 수정될 수 있기 때문에, 그 값을 읽을 때 interrupt를 잠시 비활성화하여 안전하게 접근한다.

## timer\_elapsed

이 함수는 특정 tick 값 (= `then`)이 주어졌을 때, 그 값부터 현재까지 얼마나 많은 timer interrupt가 발생했는지를 반환한다.

특정 시점 이후로 경과 시간을 측정하는데 사용된다.

## timer\_sleep

이 함수는 Pintos에서 호출하는 thread를 주어진 `ticks` 수만큼 잠자게 하는 기능을 제공한다. 함수를 호출한 thread는 `ticks` 만큼의 timer interrupt가 발생할 때까지 실행을 일시 중단한다.

단위에 따라 ms, us, ns로 나뉜다.

`timer_msleep(int64_t ms)`, `timer_usleep(int64_t us)`, `timer_nsleep(int64_t ns)`

interrupt가 꺼져 있으면 사용을 할 수 없기 때문에 delay함수를 사용해야 한다.

`timer_mdelay(int64_t ms)`, `timer_udelay(int64_t us)`, `timer_ndelay(int64_t ns)`

## timer\_print\_stats

이 함수는 현재까지 발생한 timer interrupt 총 횟수 (즉, `ticks` 수)를 출력한다.

## timer\_interrupt

이 함수는 timer interrupt가 발생할 때마다 호출되는 함수로, timer interrupt handler의 역할을 수행한다.

글로벌 변수인 `ticks` 값을 1 증가시키고, timer interrupt가 발생할 때마다 thread 관련 작업을 수행할 수 있도록 `thread_tick()` 을 호출한다.

## too\_many\_loops

이 함수는 주어진 `loops` 의 반복 횟수가 한 timer tick보다 더 오래 걸리는지를 판단하는 함수다.

`timer_calibrate` 과정에서 사용되며, `loops` 의 반복이 한 timer tick보다 더 오래 걸렸다면 `true` 를, 그렇지 않다면 `false` 를 반환한다.

## busy\_wait

이 함수는 주어진 **loops** 횟수만큼 바쁜 대기(busy-wait) 상태로 CPU를 소모하여 짧은 delay를 이르킨다.

### real\_time\_sleep

이 함수는 주어진 시간 동안 thread를 sleep 시킨다. 이 시간은 분수 형태로 주어지며, **NUM** 이 분자고 **DENOM** 이 분모다. 이 시간을 사용해 thread를 총 sleep 시켜야하는 **ticks**를 구한다.

### real\_time\_delay

이 함수는 주어진 시간 동안 thread를 delay 시킨다. 이 시간은 분수 형태로 주어지며, **NUM** 이 분자고 **DENOM** 이 분모다. 이 시간을 사용해 thread를 총 delay 시켜야하는 **ticks**를 구한다.

## 0.2 thread.c / thread.h

### 0.2.1 Overview

### 0.1.2 Macros

#### THREAD\_MAGIC

Stack Overflow를 감지하기 위한 임의의 값으로 **struct thread**의 **magic** 멤버가 이 값으로 initialize된다. 이 값이 변형되면 stack overflow가 발생된 것으로 생각한다.

#### TIME\_SLICE

각 thread에게 주어지는 timer tick의 수로 이 값을 기준으로 thread의 실행 시간을 할당하며, 이 시간이 지나면 다음 thread로 CPU를 넘겨준다.

### 0.2.3 Variables

#### ready\_list

**THREAD\_READY** 상태인 프로세스들의 목록이다. 이 thread들은 실행할 준비가 되어 있지만 (ready) 현재 실행 중이 아니다.

#### all\_list

모든 프로세스들의 목록이다. 프로세스는 첫 scheduling 때 이 목록에 추가되고, 종료가 되면 목록에서 제거된다. 모든 thread들의 관리를 위해 사용된다.

#### idle\_thread

idle 상태인 thread를 가르키는 thread다. 다른 thread가 실행할 준비가 되어 있지 않을 때 이 thread가 실행된다.

### **initial\_thread**

`init.c` 의 `main()` 함수를 실행하는 시스템 시작 시 최초로 생성되는 thread다.

### **tid\_lock**

`allocate_tid()` 함수에 의해 사용되는 lock으로, thread ID를 할당할 때 발생할 수 있는 concurrency 관련 문제를 방지하기 위해 사용된다.

### **idle\_ticks**

idle 상태에서 보낸 timer tick의 수로, 시스템이 얼마나 많은 timer ticks를 idle 상태로 보냈는지를 저장한다.

### **kernel\_ticks**

kernel thread에서 사용된 timer tick의 수로, kernel 모드에서 얼마나 많은 timer ticks를 보냈는지를 저장한다.

### **user\_ticks**

user program에서 사용된 timer tick의 수로, user 모드에서 얼마나 많은 timer ticks를 보냈는지를 저장한다.

### **thread\_ticks**

thread가 마지막으로 yield한 이후의 timer tick 수로, 이 값은 thread의 실행 시간을 구하는데 사용된다.

### **thread\_mlfqs**

스케줄링 알고리즘을 결정하는 플래그로, `false`로 설정 되었을 경우 round-robin scheduler를 사용하고, `true`로 설정 되었을 경우 multi-level feedback queue scheduler를 사용한다.

## **0.2.4 Structures**

### **struct thread**

이 구조체는 kernel 스레드나 사용자 프로세스를 나타내는 역할이다.

스레드의 상태 및 데이터를 나타내기 위해 사용된다. thread의 상태, 이름 등의 정보를 저장하며, 스택 오버플로우를 감지하는 기능도 제공한다.

각 스레드 구조체는 자체 4kB 페이지에 저장되며 페이지의 하단 (offset = 0kB)에 위치하게 된다. 페이지의 나머지 부분은 kernel 스택을 위해 예약되어 있으며, 페이지의 상단(offset = 4kB)에서 아래쪽으로 쌓이는 구조이다. kernel 스레드와 페이지를 공유하기 때문에 `struct thread`는 너무 커져서는 안된다. 그렇게 되면 kernel 스택이 차지할 수 있는 공간이 너무 적어진다. 적은

공간, 혹은 잘못된 kernel 함수로 인해 kernel 스택이 오버플로우 되면 스레드 상태가 손상된다. 따라서 kernel 함수는 큰 구조체나 배열을 non-static 로컬 변수로 할당해서는 안되고, 대신 `malloc()` 과 같은 동적 할당을 사용해야 한다.

## 멤버 변수

`tid` 스레드의 고유 식별자

`status` 스레드의 상태

`name` 디버깅 목적으로 사용되는 이름

`stack` 저장된 스택 포인터

`priority` 스레드 우선순위

`allelem` 모든 스레드 목록에 대한 리스트 요소

`elem` 실행 큐나 semaphore 대기 목록의 요소

`pagedir` 페이지 딕토리

`magic` 스택 오버플로우를 감지하는 용도

## 0.2.5 Functions

### thread\_init

이 함수는 시스템이 부팅될 때 처음으로 호출되어, 스레드 관련 구조와 변수들을 설정한다.

`lock_init()`, `list_init()` 함수들을 사용하여 스레드 ID를 관리하기 위한 lock 메커니즘, 준비 상태의 스레드와 모든 스레드를 관리하기 위한 리스트를 초기화한다. `init_thread()` 함수를 사용하여 현재 실행 중인 스레드를 `initial_thread` 변수에 저장한다. 초기 스레드의 이름은 "main", 상태는 `THREAD_RUNNING` 으로 설정한다.

### thread\_start

이 함수는 스레드 스케줄링을 시작하며, interrupt를 활성화시키고 idle thread를 생성한다.

`struct semaphore idle_started` 와 `sema_init(&idle_started, 0)` 를 통해 `idle_started` 라는 semaphore를 초기화한다. 이 semaphore는 idle 스레드가 초기화될 때까지 기다리는데 사용된다. `thread_create("idle", PRI_MIN, idle, &idle_started)` 를 사용하여 최소 우선순위인 `PRI_MIN` 으로 "idle"이라는 이름의 idle 스레드를 생성한다. idle 스레드는 시스템 내에서 다른 스레드가 실행할 준비가 되어 있지 않을 때 실행된다. `intr_enable()` 를 통해 interrupt를 활성화시

켜, 스레드 스케줄링을 시작한다. 마지막으로 `sema_down(&idle_started)` 를 통해 idle 스레드가 `idle_thread` 를 초기화할 때까지 기다린다.

### thread\_tick

이 함수는 타이머 interrupt가 발생할 때마다 호출되어, 스레드의 실행 시간을 추적하고 선점을 강제한다.

`struct thread *t = thread_current()` 를 사용하여 현재 실행 중인 스레드를 `t` 변수에 저장한다.

그 다음, `if (t == idle_thread)` 를 통해 현재 스레드가 idle 스레드인지 확인한다. 만약 맞다면, `idle_ticks++` 를 통해 idle 시간을 증가시킨다.

`#ifdef USERPROG` 내부의 조건문 `else if (t->pagedir != NULL)` 는 사용자 프로그램의 스레드가 실행 중인지 확인한다. 만약 맞다면, `user_ticks++` 를 통해 사용자 프로그램의 실행 시간을 증가시킨다.

위의 두 조건이 모두 아닐 경우, `else` 문을 통해 `kernel_ticks++` 가 실행되어, kernel 모드에서의 실행 시간을 증가시킨다.

`if (++thread_ticks >= TIME_SLICE)` 를 통해 스레드의 실행 시간이 일정 시간 슬라이스 (`TIME_SLICE`) 를 초과했는지 확인한다. 만약 초과한다면, `intr_yield_on_return()` 를 호출하여 현재 스레드의 실행을 중단하고 다른 스레드에게 실행을 양보한다. 이를 통해 CPU의 공정한 할당을 보장한다.

### thread\_print\_stats

이 함수는 idle, kernel 및 user ticks를 포함하여 스레드 실행과 관련된 통계를 출력한다.

### kernel\_thread

이 함수는 kernel 스레드의 베이스로 사용되는 함수다.

`intr_enable()` 를 사용하여 interrupt를 활성화한 다음, `function(aux)` 를 사용하여 스레드 함수를 실행하며 인자 `aux` 를 전달한다. 마지막으로 `thread_exit()` 를 호출하여 `function()` 이 반환되면 스레드를 종료한다.

### thread\_create

이 함수는 `name` 이라는 이름과 주어진 초기 우선순위 `priority` 로 새로운 kernel 스레드를 생성하고, 이를 run queue 에 추가한다. 스레드는 `function` 을 실행하며 `aux` 를 인수로 전달받는다. 스레드 생성에 성공하면 새로운 스레드의 ID를 반환하고, 실패하면 `TID_ERROR` 를 반환한다.

`palloc_get_page()` 를 사용하여 스레드를 위한 메모리를 할당하고, 할당에 실패하면 `return TID_ERROR` 를 통해 에러를 반환한다. `init_thread()` 와 `allocate_tid()` 를 통해 스레드에 이름, 우

선 순위, 그리고 tid를 설정한다. 그 이후에는 `alloc_frame()`을 사용해서 스레드의 스택 프레임을 설정한다. 마지막으로는 `thread_unblock()`를 통해 스레드를 run queue에 추가한다.

깊고 넘어가야 할 점은 `thread_create()` 가 호출된 후에는 새로운 스레드가 스케줄링되거나 `thread_create()` 가 반환되기 전에 종료될 수 있다는 것이다. 원래의 스레드는 새 스레드가 스케줄링되기 전에 얼마든지 실행될 수 있다. 실행 순서를 보장하려면 semaphore나 다른 동기화 방법을 사용해야 한다.

### thread\_exit

이 함수는 현재 스레드를 종료시키고 삭제시킨다.

이 함수가 호출되면 절대 caller한테 return하지 않는다. user program인 경우 `process_exit()`을 호출한다.

### thread\_yield

이 함수는 현재 실행 중인 스레드가 CPU를 yield하게 한다. 해당 스레드는 sleep이 되지는 않으며 스케줄러에 따라 즉시 다시 스케줄될 수 있다.

`intr_disable()` 를 통해 현재의 interrupt 레벨을 `old_level`에 저장하고 interrupt를 비활성화한다. 그 다음 `if (cur != idle_thread)` 검사를 통해 현재 스레드가 idle 스레드가 아닌 경우, `list_push_back()` 를 사용하여 현재 스레드를 `ready_list`에 추가한다. 현재 스레드의 상태를 `THREAD_READY` 상태로 변경한다. `schedule()` 를 호출하여 다음 스레드를 스케줄링한다. `intr_set_level(old_level)` 는 이후 thread가 switch되어 돌아왔을 때 실행되며 통해 interrupt 레벨을 원래 상태로 복원한다.

### thread\_block

이 함수는 현재 스레드를 잠재우어 스케줄링 대상에서 제외시킨다. `thread_unblock()` 에 의해 다시 깨어날 때까지 스케줄링되지 않는다.

`ASSERT (!intr_context())` 는 빠르게 처리되어야 하는 external interrupt가 이 함수를 부를 수 없는 것을 보장하기 위해 필요하고, `ASSERT (intr_get_level() == INTR_OFF)` 는 interrupt가 꺼져있는 상태에서만 이 함수가 호출되어야 함을 보장하기 위해 필요하다. 이는 스레드를 안전하게 잠재우기 위한 조건이다.

조건이 만족되면 현재 스레드의 상태를 `THREAD_BLOCKED`로 변경한다. 이는 스레드가 준비 큐에서 제외되고 스케줄링되지 않게 한다. 마지막으로 `schedule()` 를 호출하여 스케줄러를 통해 다른 스레드가 실행될 수 있도록 한다.

### thread\_unblock

이 함수는 주어진 스레드 `t`를 깨워서 TCB를 이후에 스케줄링 될 수 있는 상태로 만들고 `thread`를 `ready_list`에 추가한다.

`intr_disable()`를 통해 현재의 interrupt 레벨을 받아 `old_level`에 저장하고 interrupt를 비활성화시킨다. 이는 동기화를 위해 스레드 상태 변경을 안전하게 만든다. `list_push_back(&ready_list, &t->elem)`를 통해 `t`를 `ready_list`의 뒤에 추가해준 다음엔 스레드 `t`의 상태를 `THREAD_READY`로 변경한다. 이로써 스레드는 다시 스케줄링될 준비가 되었다.

`intr_set_level(old_level)`는 이후 thread가 switch되어 돌아왔을 때 실행되며 통해 interrupt 레벨을 원래 상태로 복원한다.

### `thread_name, thread_current, thread_tid`

각각 이름, 현재 실행 중인 스레드, 스레드 ID를 반환하는 함수다.

### `thread_set_priority, thread_get_priority`

각각 현재 스레드의 우선 순위를 설정하고 반환하는 함수다.

### `thread_FOREACH`

이 함수는 모든 스레드에 대해 주어진 함수 `func` (`thread_action_func *` 타입: `struct thread *t` 와 `void *` 를 입력으로 하고 리턴은 없는 함수 포인터)를 호출하며, `aux` 를 전달 인자로 함께 보낸다. 이 함수는 interrupt가 꺼진 상태에서 호출되어야 한다.

for문을 사용하여 `all_list`에 있는 각 스레드 구조체 포인터 `t`에 대해 `func(t, aux)`를 호출하여 각 스레드에 `func` 함수를 호출하고 `aux` 를 전달한다.

### `running_thread`

이 함수는 현재 실행 중인 스레드를 반환한다.

`esp`라는 변수를 만들어, 여기에 `asm ("mov %%esp, %0" : "=g" (esp))` 를 사용하여 CPU의 스택 포인터 값을 저장한다. `return pg_round_down(esp)` 를 사용하여 `esp` 를 페이지의 시작 주소로 내림한 후, 해당 주소에서의 `struct thread` 를 반환한다. `struct thread` 는 항상 페이지의 하단 부분에 위치하며 스택 포인터는 페이지 중간 어딘가에 있기 때문에, `esp` 를 내리면 무조건 현재 스레드를 찾을 수 있다.

### `is_thread`

이 함수는 주어진 포인터 `T`가 유효한 스레드를 가리키는지 확인하여 그 결과를 반환한다.

`t` 가 NULL이 아니고 `t` 의 `magic` 이 `THREAD_MAGIC` 과 같은지 확인한다. 이 두 조건이 모두 만족하면 `true`를 반환하고, 그렇지 않으면 `false`를 반환한다.

### `init_thread`

이 함수는 TCB `t`를 초기화하는 함수이다.

`PRI_MIN` 과 `PRI_MAX` 사이에 주어진 `priority` 값이 위치한다면 유효한 값으로 판단한다. `memset(t, 0, sizeof *t)` 를 사용하여 `t`의 메모리를 0으로 초기화한다. `t`의 `status` 를 `THREAD_BLOCKED` 로 설정한다. 그 다음엔 `t`의 `name` 필드에 주어진 이름 `name` 을, `stack` 필드에 `(uint8_t *) t + PGSIZE` 를, `priority` 필드에 주어진 `priority` 값을, 그리고 `magic` 필드에 `THREAD_MAGIC` 값을 설정한다. 마지막으로 `old_level = intr_disable()` 와 `intr_set_level(old_level)` 를 사용해서 안전하게 interrupt가 비활성화된 상태에서 `list_push_back()` 를 통해 안전하게 `t` 를 `all_list` 에 올린다.

### alloc\_frame

이 함수는 스레드 `t`의 스택 상단에 인자 `size` 바이트 크기의 프레임을 할당하고 프레임의 기본 포인터를 반환한다.

`t` 가 유효한 스레드 포인토이고, 요청된 `size` 가 word-size 단위로 정렬되어 있다면, `t`의 스택 포인터를 `size` 바이트만큼 감소시키고 수정된 스택 포인터 (할당된 프레임의 시작 주소)를 반환한다.

### next\_thread\_to\_run

이 함수는 스케줄링 될 다음 스레드를 `ready_list` 에서 선택하고 반환한다.

`list_empty()` 를 사용하여 `ready_list` 가 비어 있는지 확인하고, 비어 있으면 `idle_thread` 를 반환하고, 그렇지 않은 경우 `list_entry(list_pop_front())` 를 사용하여 `ready_list` 의 앞부분에서 스레드를 꺼내어 반환한다.

### idle

이 함수는 다른 스레드가 실행 준비가 되어 있지 않을 때 실행되는 idle thread의 코드다.

`thread_current()` 를 사용하여 현재 스레드를 받아와서 `idle_thread` 에 저장한다. 그 다음 `sema_up (idle_started)` 를 사용하여 semaphore `idle_started` 를 "up"하여 `thread_start()` 가 계속 진행될 수 있도록 한다. 무한 루프안에 `intr_disable()` 와 `thread_block()` 를 호출하여 interrupt를 비활성화하고 현재 스레드를 차단한다. 마지막으로, `asm volatile ("sti; hlt" : : : "memory")` 를 사용하여 interrupt를 활성화하고 다음 interrupt가 발생할 때까지 기다리고 complier가 operation을 reordering 하지 못하게 한다.

### thread\_schedule\_tail

이 함수는 새로운 스레드의 페이지 테이블을 활성화함으로써 스레드 전환을 완료하고, 이전 스레드가 종료 중이면 그것을 파괴한다.

`running_thread()` 함수를 사용하여 현재 실행 중인 스레드를 `cur` 변수에 저장한다. 현재 스레드의 상태를 `THREAD_RUNNING`으로 설정한다. `thread_ticks` 를 0으로 초기화한다. `USERPROG` 인 경우 `process_activate()` 함수를 호출하여 새 주소 공간을 활성화한다. 마지막으로, 우리가 전환한 이전 스레드가 종료 중이고 `initial_thread` 가 아니면 해당 스레드 구조를 `palloc_free_page()` 함수를 사용해서 파괴한다. `initial_thread` 는 `palloc()` 을 통해 얻어지지 않았기 때문에 이 방법을 사용해서는 안된다.

### schedule

이 함수는 새로운 프로세스를 스케줄링한다. 함수가 호출될 때, interrupt는 꺼져 있어야 하며 현재 실행 중인 프로세스의 상태는 실행 중에서 다른 상태로 변경되어야 한다. 이 함수는 다른 스레드를 찾아 실행하게 전환한다.

`running_thread()` 함수를 호출하여 현재 실행 중인 스레드를 `cur` 변수에 저장한다.

`next_thread_to_run()` 함수를 호출하여 다음에 실행될 스레드를 `next` 변수에 저장한다. 이 때, interrupt가 꺼져 있는지, 현재 스레드의 상태가 실행 중이 아닌지, `next` 변수가 유효한 스레드를 가리키는지 확인한다.

현재 스레드와 다음에 실행될 스레드가 다르면, `switch_threads()` 함수를 호출하여 스레드 전환을 수행하고, 이전 스레드 정보를 `prev` 변수에 저장한다. `thread_schedule_tail()` 함수는 이후 thread가 switch되어 돌아왔을 때 실행되며 통해 interrupt 레벨을 원래 상태로 복원한다.

### allocate\_tid

이 함수는 새로운 스레드에 할당할 스레드 ID(`tid`)를 생성하고 반환한다.

함수 내에서 정적 변수 `next_tid` 는 다음에 할당될 스레드 ID를 저장하며 초기 값은 1로 설정된다. `lock_acquire(&tid_lock)` 를 사용하여 `tid` 생성 과정 중에 다른 스레드가 이 함수에 동시 접근하는 것을 방지한다. `tid` 변수에 `next_tid` 값을 저장한 후, `next_tid` 값을 1 증가시킨다. 이렇게 해서 각 스레드는 고유한 ID 값을 가질 수 있다. 마지막으로 `lock_release(&tid_lock)` 를 사용하여 lock을 해제하고, 새로 할당된 `tid` 값을 반환한다.

# 1. Initialization of Thread system & Thread Switching

## 1.1 Overview

이번 장에서는 Current Code Analysis 장에서 살펴본 함수들을 기반으로 thread system의 initialize 과정, thread의 initialize 과정, 그리고 thread의 switch 과정을 보다 깊이있게 알아볼 것이다.

이번 글에는 아래의 질문에 답을 할 수 있는 내용이 포함되어 있다.

- `thread_create` 함수는 어떤 역할을 하는가
  - fake stack frame이 왜 필요하고 어떻게 사용되는가
- `schedule` 함수는 어떻게 작동하는가
  - thread는 언제 실행되는 것인가
  - thread는 어떻게 switch 되는가
  - TCB는 무엇인가

`threads/init.c` 의 `main()` 을 보면 pintos booting을 위한 여러 작업을 한다. 그 작업 중 thread system을 initialize하는 과정이 포함되는데 이는 `thread_init` 함수 와 `thread_start` 함수 를 통해 이루어진다. `threads/init.c` 에서 두 함수의 실행 흐름을 짚어가며 thread system의 initialize 과정을 살펴보자. 그 과정에서, thread가 어떻게 초기화되고 스케줄링 될 준비를 하는지, 어떻게 스케줄링 되는지까지 자세하게 알아보겠다. 이전에 함수들에 대한 대략적인 내용은 살펴보았으므로 이 부분에서는 함수 자체에 대한 내용보다는 유기적인 동작 방식을 위주로 살펴보겠다.

## 1.2 `thread_init`

thread를 정의하기 위해서는 thread의 상태를 기록해둔 thread control block과 그것을 실행하는데 사용하는 stack 영역이 필요하다. 이러한 점을 고려했을 때, `thread_init` 함수 는 현재 실행 중인 코드를 thread ‘화’ 시키는 함수로도 해석할 수 있다. thread system의 정상적인 동작을 위하여 `tid_lock`과 `ready_list`, `all_list`을 초기화하는 과정은 기본적으로 필요하고 주의 깊게 살펴볼 부분은 그 이후 부분이다.

`running_thread` 함수 를 통해 struct thread가 위치할 곳의 주소를 받아온다. 그리고 `init_thread` 함수 를 통해 main이라는 이름을 가지는 thread의 thread control block을 구성한다. 그 이후 status를 `THREAD_RUNNING`으로 수정해주고 tid도 부여한다. 이전까지 실행중이던 프로그램은 이 순간부터 thread로 취급할 수 있는 것이다.

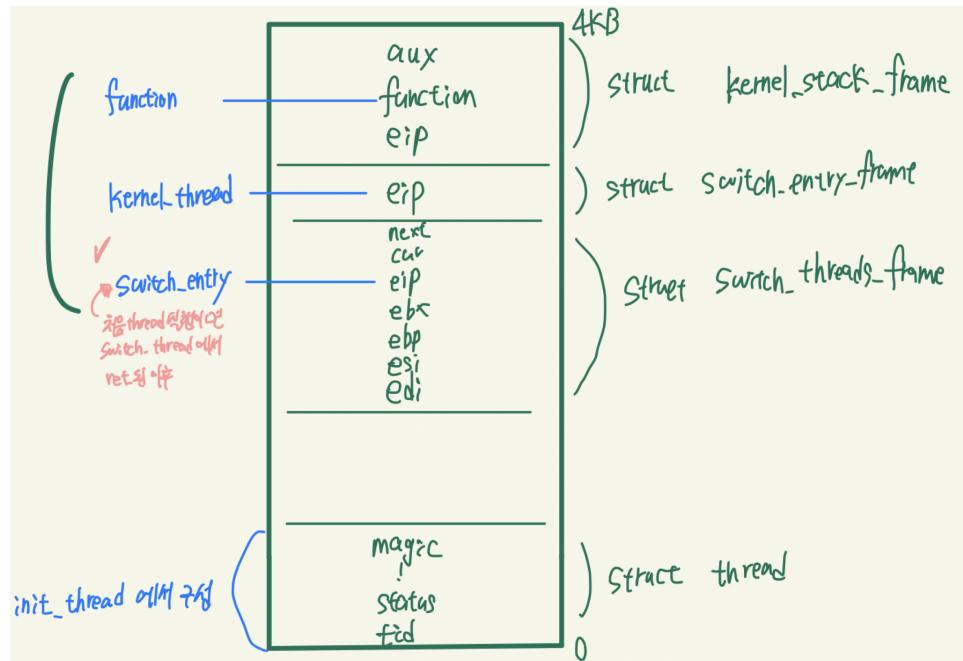
이 함수가 정상적으로 작동하기 위해서는 loader의 도움이 필요한데 그 이유는 `running_thread` 함수 의 정상적인 작동을 위해서는 stack의 bottom을 page의 경계로 맞추어 두어야 하기 때문이다. `thread_init` 함수 는 pintos 부팅시 한 번만 동작하므로 loader의 도움을 받아야만 하는 조건이 있는 것이 큰 제약이 되지는 않는다. 하지만 이후에 생성되는 thread를 이미 구현된 함수(ex. `running_thread` 함수 )로 관리하기 위해서는 page allocator를 초기화하는 것이 필요하다.

## 1.3 thread\_start

이 함수에서는 idle thread를 생성하고 interrupt를 키는 두 가지 일을 한다. 이 두 가지 일을 하는 이유는 다음과 같다. 우선 idle thread는 이후 생성되는 thread 중 어느 thread도 cpu를 점유하고 있지 않을 때 실행되는 thread이다. idle thread의 실행시간을 확인함으로써 cpu에 아무런 thread가 스케줄링 되지 않은 시간을 측정할 수 있다. 다음으로, interrupt를 켜 둠으로써 실행 중인 다른 thread를 멈추고 또 다른 thread를 실행할 수 있도록 한다.

`thread_start` 함수 가 시작되어 종료된 이후에 실행되는 `thread_create` 함수 는 언제든 block될 수 있다. 하지만 지금은 `thread_start` 함수 호출이 다 끝마쳐지지 않은 상황이라 아직 interrupt를 켜두지 않았기 때문에 idle thread를 만들기 위해 호출하는 `thread_create` 함수 실행 중간에 다른 thread가 실행될 일은 없다.

idle thread에 대해 더 살펴보자. 앞에서 살펴본 main thread와는 다르게 idle thread는 `thread_create` 함수 를 활용하여 만들어진다. `thread_create` 함수 에서 특이한 점은 fake stack frame을 구성한다는 것이다. 아래 그림과 함께 살펴보자.



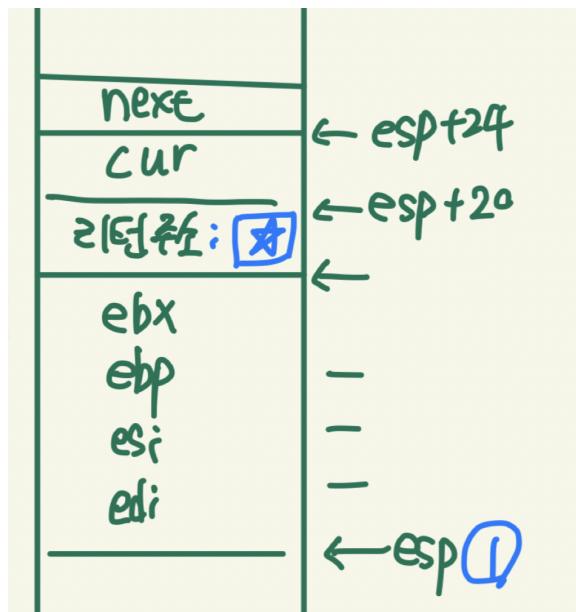
`init_thread` 함수 로 TCB 구성을 마친 이후, stack frame을 할당받아 `kernel_thread_frame`, `switch_entry_frame`, `switch_threads_frame`을 만든다. 이러한 stack frame을 만드는 이유는 `생성된 이후 한 번도 실행되지 않은 thread` 의 `switch` 방식과 `실행을 하다가 멈춘 이후 다시 실행되는 thread` 의 `switch` 방식을 동일하게 하기 위해서이다.

### 1.3.1 Switching Threads

생성된 이후 한 번도 실행되지 않은 `thread`의 `switch` 방식을 실행을 하다가 멈춘 이후 다시 실행되는 `thread`의 `switch` 방식과 동일하게 하는 것이 fake stack frame을 만드는 이유라고 하였으므로, 실행을 하다가 멈춘 이후 다시 실행되는 `thread`가 실행을 멈출 때, 해당 `thread`의 stack에 어떤 변화가 일어나는지 이해를 할 필요가 있다.

이를 위해 `thread A`가 실행하다가 멈추고 `thread B`가 실행되는 경우를 살펴보자. 이 경우 `thread A`가 `schedule` 함수를 호출하고 그 안에서 `switch_threads` 함수가 호출된다.

`switch_threads` 함수는 어셈블리 언어로 작성되어 있는데 함수 호출시 가장 먼저 하는 작업은 `ebx, ebp, esi, edi` 레지스터에 있는 값을 stack으로 push하는 것이다. 이는 `thread A`의 context를 보존하기 위함이다. 이후 `edx`에 ‘struct thread’의 멤버 변수인 stack의 구조체 내부에서의 offset’을 저장한다. 이 값은 이후 `thread A`의 stack pointer를 저장하고, `thread B`의 stack pointer를 불러오는데 사용된다. 이후, `eax`에 `thread A`의 TCB 주소를 저장한다. 그리고 `eax`와 `edx`를 활용하여 `thread A`의 TCB의 stack을 저장하는 변수에 `esp`값을 저장한다. 이는 이후 `thread A`가 다시 `switch` 되어 실행을 재개할 때 stack pointer 값을 알려주는 용도로 활용된다. 현재 `thread A`의 stack 상태를 그림으로 보면 아래와 같다.(`esp`는 파란색 1번에 위치하고 있다.)



```
static void
schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

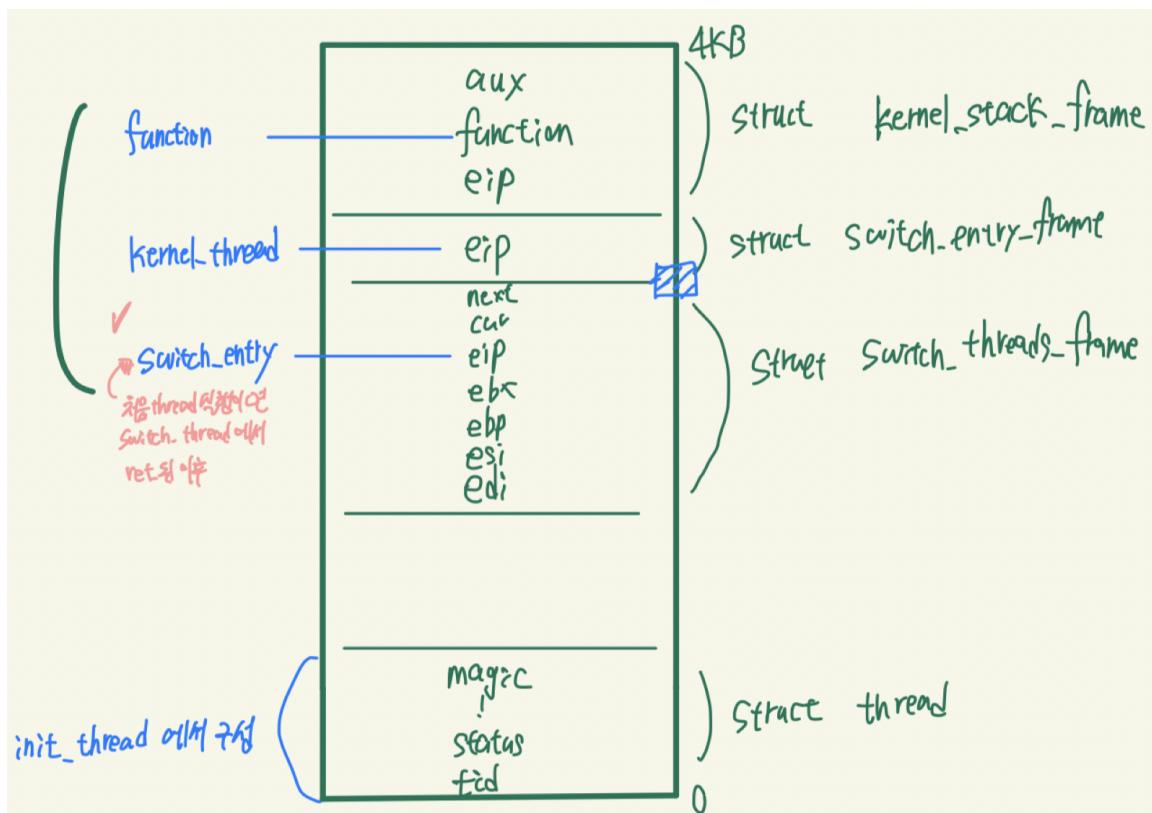
    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
        prev = switch_threads (cur, next);
    thread_schedule_tail (prev);
}
```

`switch_thread` 함수가 실행될 때, 함수 호출을 하는 것이므로, call instruction(어셈블리 언어)이 수행될 것임을 알 수 있는데 이때 리턴 주소를 stack에 저장하게 된다. 그 주소는 파란색 별표로 표시했으며 `thread_schedule_tail` 함수의 실행 부분과 같다. `switch_threads` 함수의 arguments도 위 그림과 같이 저장된다. `SWITCH_CUR`가 20으로 정의되어 있고, `SWITCH_NEXT`가 24로 정의되어 있는 것에서 이런 convention이 사용된다는 것을 알 수 있다.

이어서 thread B의 실행은 어떻게 되는지 살펴보자. 다시 `switch_threads` 함수를 살펴보면, ecx에 thread B의 TCB 주소를 저장한다. 그리고 ecx와 edx를 활용하여 thread B의 stack pointer 값을 esp(stack pointer 주소를 저장하는 레지스터)에 저장한다. 마지막으로 edi, esi, ebp, ebx를 차례로 pop하여 thread B가 실행 중이었던 context를 복구한다. 이제, cpu의 관점에서 thread B를 실행할 준비가 된 것이다. 따라서 ret instruction을 통해 기존에 실행 중이었던 위치로 eip(program counter 주소를 저장하는 레지스터, 여기서는 이전에 thread B가 실행 중이던 주소값이 들어가 있음)를 이동시킨다. thread B도 thread A가 위에서 했던 것과 같이 switching되어 다른 thread에 실행을 양보할 때 stack에 동일한 과정으로 값을 저장해 뒀을 것이기 때문에 이러한 흐름으로 thread가 switch 된다는 것을 알 수 있다.

### 1.3.2 fake stack frame

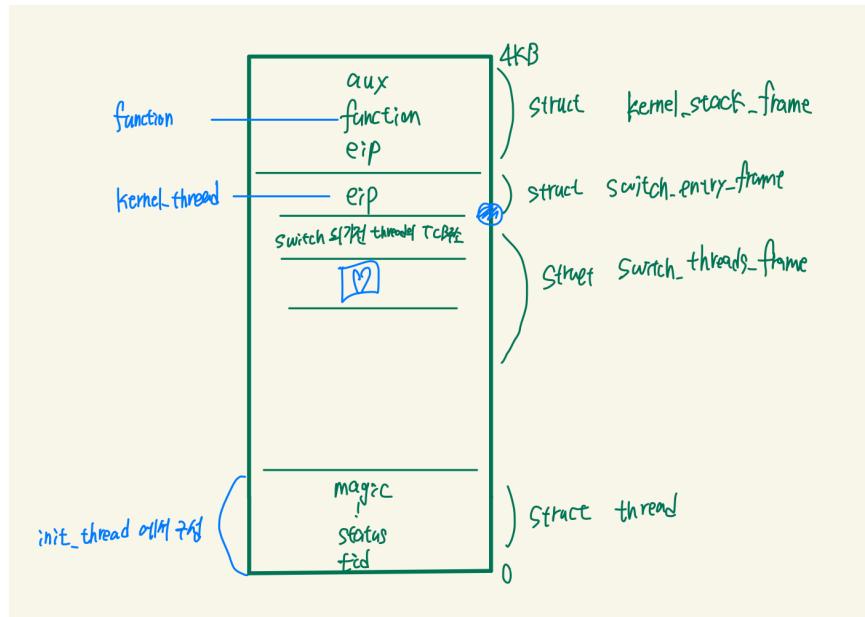


실행 중이던 thread가 어떻게 switch 되는지 이해하였다. 다시 fake stack frame 논의로 돌아가보자. 위 그림은 fake stack frame 구성의 그림을 다시 가져온 것이다. thread A와 thread B가 스케줄링 되는 과정을 돌아보면 위 그림의 `switch_threads_frame`을 이해할 수 있다. 한 가지 다른 점은 eip 부분에 thread에서 원래 실행 중이던 주소가 들어 있는 것이 아니라 `switch_entry` 함수의 주소가 있다는 것이다. 이는 새로 생성된 thread를 처음 실행하는 것이므로

기준에 실행 중이던 곳이 없으니까 충분히 다를 수 있는 부분이다.(함수 이름과 주석을 통해 추측해보기만 해도, switch되어 enter 하는 시점에 호출되니까 자연스럽다.)

`switch_entry` 함수에 대해 알아보자. 이 함수 역시 어셈블리 언어로 작성되어 있다. 이 함수에서는 esp 값에 8을 더함으로써 `switch_threads` 함수의 argument 부분을 무시한다. 이러한 처리를 하는 이유는 이미 `switch_threads` 함수의 후반부에서 pop 4번을 통해 만들어 두었던 edi, esi, ebp, ebx가 stack에서 없어진 이후 ret에 의해 eip에 있는 주소인 `switch_entry`로 이동했기 때문이다. 즉 `switch_entry` 함수가 실행되어, esp에 8을 더한 이후 esp 값은 파란색 네모 부분에 위치해있다. 이후 eax를 push하고 `thread_schedule_tail` 함수를 호출한다. 현재 eax에는 `switch_threads` 함수에서 eax에 있던 'switch 되기 전에 실행 중이던 thread의 TCB의 주소'가 있다. `thread_schedule_tail` 함수는 `thread_schedule_tail(prev)`와 같이 사용되는데, 여기서 eax에 있는 값이 stack으로 push되어 `thread_schedule_tail` 함수의 argument가 되어 사용되는 것이다. `thread_schedule_tail` 함수가 리턴되면 아래 그림 코드 부분의 파란색 하트 부분으로 돌아오게 되는데 esp 값에 4를 더하여 매개변수 값을 무시하고 esp값을 아래 그림의 파란색 동그리미 부분에 위치시킨다. 그리고 ret 연산을 통해 `kernel thread` 함수가 실행되도록 한다.

```
51 .globl switch_entry
52 → .func switch_entry
53     switch_entry:
54         # Discard switch_threads() arguments.
55         addl $8, %esp
56
57         # Call thread_schedule_tail(prev).
58         pushl %eax
59         .globl thread_schedule_tail
60         call thread_schedule_tail
61         addl $4, %esp ━━━━━━ V
62
63         # Start thread proper.
64         ret
65     .endfunc
```



`kernel_thread` 함수의 실행을 위해서 `kernel_stack_frame`이 존재한다. `kernel_thread` 함수는 매개변수로 `thread_func *` 값과 `void *` 값을 받는다. 이를 고려하여 `kernel_stack_frame`에 실행할 `function`과 매개변수인 `aux`를 미리 만들어 둔 것이다. 따라서 `thread`가 처음 실행할 때조차도 `thread_create` 함수의 argument로 전달된 함수 포인터가 가르키는 함수가 실행되는 것이다.

실행 중이던 `thread`가 `switch`되어 중단될 때 정보 저장을 위해 만들어 두는 `stack`과 동일한 형태로 fake stack frame을 구성해둠으로써 `thread`가 처음 실행되든 아니든 동일한 방법으로 `switch`될 수 있다. 지금까지 fake stack frame을 구성함으로써 특수한 방법이 아닌 일관적인 방법으로 `thread`를 `switch`할 수 있다는 사실을 살펴보았다.

### 1.3.3 idle thread

switching threads 부분과 fake stack frame 부분에서 살펴본 흐름에 따라서 `idle` 함수가 실행된다. 이제 `idle` 함수가 최초에 실행되면 main thread와 어떻게 상호작용을 하는지 살펴보겠다.

main thread가 호출하는 `thread_start` 함수에서 가장 먼저 하는 일은 세마포어 `idle_started`를 0으로 initialize 하는 것이다. 그리고 세마포어 `idle_started`를 `idle` 함수의 argument로 설정하게끔 `thread_create` 함수를 호출한다. 그리고 앞에서 살펴본 바와 같이 `thread_create` 함수를 통해 `idle` 함수를 실행하는 idle thread를 실행할 수 있는 상태를 만들도록 처리한 후 최종적으로 `sema_down(&idle_started)`를 실행한다. `thread_create` 함수에서 idle thread에 대한 기본 정보와 `switch` 되었을 때 정상적으로 동작할 수 있도록 처리를 마쳤지만 여전히 실행 중인 `thread`는 main thread라는 점을 인지해야한다. `idle` 함수는 아직 실행되지 않았다라는 점을 알아야한다.

main thread가 `sema_down` 함수를 실행하는 과정에서 세마포어 `idle_started`값이 0이기 때문에 `thread_block` 함수를 호출하게 된다. `thread_block` 함수에서 `schedule` 함수가 호출되기 때문에 이

때 thread가 switch 되며 `idle` 함수를 실행하는 idle thread가 최초로 실행된다. `schedule` 함수가 호출되기 직전에 main thread는 THREAD\_BLOCKED 상태가 되고 idle\_started 세마포어의 waiter에 main thread가 추가된다. 이제 `idle` 함수에서 어떤 일을 하는지 살펴보자.

idle thread에서 `idle` 함수가 최초로 실행될 때는 `thread_current` 함수를 호출하여 전역변수인 `idle_thread` 값을 idle thread의 TCB 시작 주소로 설정한다. 그리고 `sema_up` 함수를 호출하여 앞에서 main thread에서 down 하였던 세마포어와 동일한 세마포어인 `idle_started`를 up 시킨다. 이때, `idle_started` 값은 1이 된다. 앞에서 `sema_down` 함수를 호출하였는데도 불구하고 왜 값이 0인가 하면 `sema_down` 함수의 구현상 값이 0인 경우 `thread_block` 함수가 먼저 호출되기 때문이다. 그리고 값이 0이 아니게 되는 순간까지 while loop를 무한히 돈다.(이 loop는 `sema_down` 함수를 호출한 thread에서 돈다. 여기서는 main thread)

`sema_up` 함수에서 `thread_unblock` 함수를 호출하는데 main thread의 상태가 THREAD\_RUNNING이 된다. 하지만 아직 실제로 main thread가 실행 중인 상태가 된 것은 아니다. `schedule` 함수를 호출하여 스케줄링을 해주기 않았기 때문이다. 여기서 한 가지 알 수 있는 사실은, TCB는 단지 thread의 상태를 나타내기 위한 정보를 기록해 두는 보조적인 데이터 일 뿐이지 해당 데이터에 정보가 기록되었다고 그에 맞게 thread가 동작하지는 않는다는 것이다. 지금까지 내용을 정리하자면, `idle` 함수에서 `sema_up` 함수가 종료된 시점에 `idle_started` 세마포어 값은 1, 실행 중인 thread는 idle thread, main thread의 state는 THREAD\_RUNNING이다.

다시 `idle` 함수로 돌아가보자. `sema_up` 함수 실행을 마친 `idle` 함수에서는 `for( ; ; )`로 표현된 무한 loop로 진입하게 된다. loop에서 가장 먼저 하는 일은 interrupt를 끄는 것이다. 그리고 `thread_block` 함수를 호출하여 THREAD\_RUNNING 이었던 idle thread의 상태를 THREAD\_BLOCKED로 바꾼다. 그 직후에 `schedule` 함수에 의해 main thread가 실행된다. main thread는 실행을 멈췄던 곳(`sema_down` 함수 안에 있는 `thread_block` 함수 안에 있는 `schedule` 함수 안에 있는 `thread_schedule_tail` 함수)에서 실행을 재개할 것이다. `thread_schedule_tail` 함수가 실행되고 `schedule` 함수와 `thread_block` 함수가 차례로 리턴되어 `sema_down` 함수에서 `thread_block` 함수가 리턴된 후로 간 상황에서, `idle_started` 세마포어 값이 1이므로 while loop의 조건을 만족하지 않는다. 따라서 세마포어 값을 1 줄이고 `thread_start` 함수가 리턴된다.

이후의 작업은 init.c에 있는 `thread_start()` 아래의 작업을 의미한다. `thread_block` 함수에서 이러한 동작이 가능한 이유는 `thread_unblock` 함수에서 THREAD\_BLOCKED 상태에 있는 main thread를 THREAD\_READY 상태로 잘 바꾸어 주고, ready\_list에 넣어두는 '실제로 thread가 switching되기 위한 사전 작업'을 해두었기 때문이다. 여기서 알 수 있는 또 하나의 사실은 thread가 실질적으로 switch 되는 시점은 cpu가 stack pointer 값을 저장해두는 esp의 주소를 바꾸고, thread가 실행 중이던 context 복원을 위한 작업을 한 직후라는 것이다. 이 점을 고려하

였을 때 방금 살펴본 예시에서는 main thread가 switch 되는 시점은 `thread_unblock` 함수에서가 아니라 `thread_block` 함수가 실행되는 중에 있다는 것이다.

## 1.4 Extra Details

- `switch_threads` 함수는 `switch_threads (cur, next)`와 같이 호출되는데 이때 stack에 cur과 next 값이 저장된다.

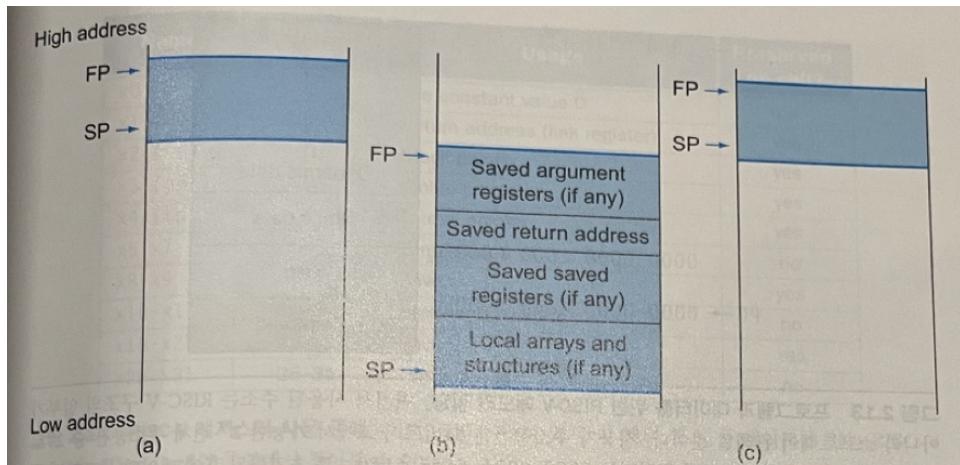


그림 2.12 프로세저 호출 (a)전, (b)중, (c)후의 스택 할당. 프레임 포인터(fp 또는 x8)는 프레임의 첫 번째 더블워드(인수 레지스터값이 저장된 것일 때가 많다)를 가리키며, 스택 포인터(sp)는 스택의 맨 위를 가리킨다. 프로세저를 호출하면 저장해야 하는 모든 레지스터와 메모리 내의 지역 변수를 넣기 위한 공간을 스택에 만든다. 약간의 주소 계산을 추가하면 sp로 변수를 참조할 수도 있지만, sp는 프로그램이 실행되면서 변할 수 있으므로 변수 참조는 변하지 않는 fp를 사용하는 것이 좋다. 스택에 지역 변수를 저장하지 않는 프로세저의 경우는 fp값을 바꾸고 나중에 원상복구하는 일을 생략함으로써 시간을 절약할 수 있다. fp를 사용하는 경우는 호출할 때의 sp값으로 fp를 초기화하고 나중에 fp를 이용하여 sp를 원상복구 한다. 이 정보는 이 책의 첫 번째 페이지에 있는 RISC-V Reference Data Card의 ④열에서도 찾아볼 수 있다.

- `thread_init` 함수에서 ASSERT 문으로 `intr_get_level` 함수의 리턴값이 `INTR_OFF`임을 보장한다. pintos가 booting되기 이전이므로 이 부분에서 문제가 될 일은 없을 것으로 보다 견고한 코드를 위해 작성해둔다.
- `switch_thread` 함수에서 `movl (%eax, %edx, 1), %esp` 실행 직후에 실행하는 코드는 switch되어 앞으로 실행될 thread를 위한 것이다.
- `timer_init` 함수는 timer interrupt handling이 가능하도록 준비한다. 일단 `pit_configure_channel` 함수를 통해 Programmable Interval Timer와의 채널설정을 해준다. 이는 초당 `TIMER_FREQ` 만큼 timer interrupt가 일어나도록 설정한다. 그 다음엔 `intr_register_ext` 함수를 통해 `timer_interrupt` 함수를 interrupt handler로 등록한다.

## 2. Analysis of Synchronization Primitives

### 2.1 Overview

이번 장에서는 `synch.c` / `synch.h`에 정의된 synchronization primitives가 어떤 방식으로 활용될 수 있는지 알아볼 것이다. 또한 필요한 경우에 대하여 `pintos` 코드에서 실제로 어떻게 활용되는지 알아볼 것이며 Synchronization primitive에 대한 추가적인 설명을 진행할 것이다.

### 2.2 Motivation of Synchronization Primitive

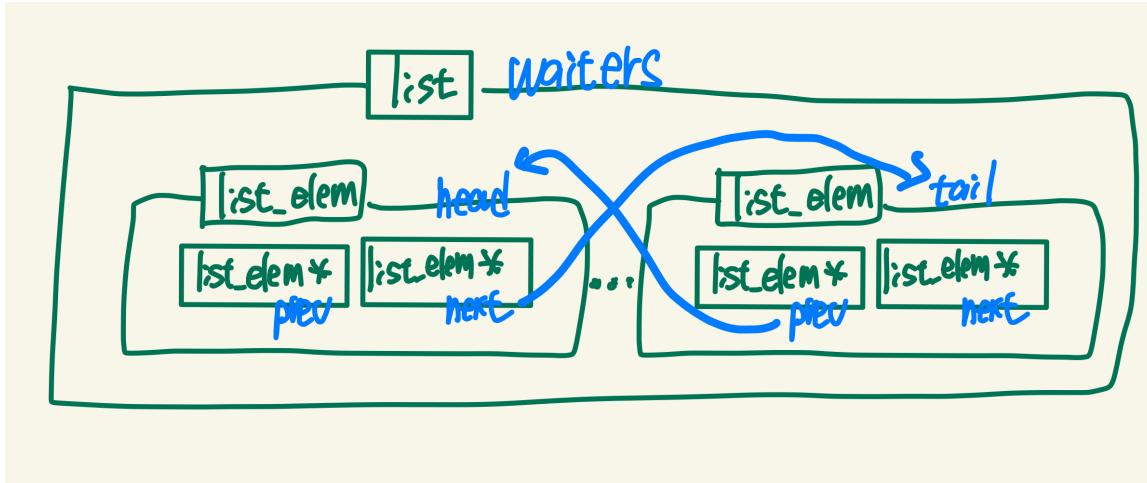
다수의 thread를 활용하여 작동하는 프로그램을 만들 때, 필연적으로 thread 사이에 공유되는 변수가 존재한다. 이러한 공유 자원들을 적절히 관리하지 않으면 큰 오류가 생길 수 있다. 이러한 공유 자원의 관리는 interrupt를 끄고 키는 동작을 통해 가능하다. interrupt를 끄면 CPU가 interrupt에 응답하는 것을 방지하여 다른 thread가 현재 실행 중인 thread를 preempt할 수 없기 때문이다.

interrupt를 끄고 키는 기능은 `interrupt.h`와 `interrupt.c` 파일에 정의된 함수를 통해 동작시킬 수 있다. 이 함수들을 살펴보면 결국엔 어셈블리 언어로 구성되어 있는 매우 low level 코드라는 것을 알 수 있다. 이러한 low level 코드를 활용하여 공유 자원을 관리하는 것을 보다 더 편히 하기 위해 synchronization primitive를 정의한다.

### 2.3 Semaphore

가장 먼저 살펴볼 것이 세마포어이다. 세마포어 구조체의 정의를 보면 `unsigned` 타입의 `value`와 `struct list` 타입의 `waiters`로 구현되어 있다.

`value`를 활용하여 세마포어가 관리하는 공유 변수에 대한 접근 가능여부를 결정할 수 있고, `waiters`에 세마포어가 관리하는 공유 변수에 접근하길 원하는 thread 정보를 리스트 형태로 저장한다. `waiters`는 아래 그림과 같은 구조를 띠고 있다.



motivation 부분의 서술에서도 알 수 있듯이 세마포어라고해서 대단한 개념이 추가된 것이 아니라 low level로 구현한 기능을 사용하기 쉽게 만들어 둔 것 뿐이다. 물론 value 값을 어떤 식으로 사용할 수 있는지에 대한 대표적인 방법들은 존재한다. 그 방법들에 대해 알아보며 세마포어가 가지는 의미에 대해 알아보자.

### 2.3.1 Initialize value with '0'

이 방법은 위에서 설명한 thread system을 initialize 하는 과정에서 main thread 와 idle thread 사이에서 사용된 방법이다. 예시를 통해 이 방법에 대해 살펴보자.

thread A가 thread B를 시작시키고, thread A는 thread B에서 어떤 작업이 끝나는 것을 기다리고 있는 상황이라고 해보자. 이때, thread A는 세마포어의 value를 0으로 만들고 이를 thread B로 넘겨준다. 그리고 thread A는 sema\_down 함수를 이용하여 thread B가 작업을 끝냈다는 신호를 기다린다(thread A가 block된다.). 그 신호는 thread B에서 sema\_up 함수를 통해 발생된다. 각 함수 동작에 대한 세부적인 내용은 아래에서 살펴보겠다.

위 예시에서, 엄밀히 말하면 thread A가 sema\_down 함수를 실행시키는 시점과 thread B가 sema\_up 함수를 실행시키는 순서는 바뀌어도 상관없다. 그런 경우에는 sema\_down 함수를 실행한 thread A가 block되지 않을 것이기 때문이다.

### 2.3.2 Initialize value with '1'

이 방법은 일반적으로 자원에 대한 접근 권한을 조정할 때 사용한다. 자원을 사용하기 전에 sema\_down 함수를 호출하여 value를 0으로 만들고, 자원을 사용한 후에 sema\_up 함수를 호출하여 value를 1로 만든다. 이렇게 세마포어를 사용하는 경우 value의 의미는 '지금 해당 세마포어가 관리하는 자원을 사용할 수 있는가?'에 대한 답과 같다. 1이면 가능한 것이고 0이면 불가능한 것이다. 이 방법에서 일반화하여 세마포어 값을 n으로 하여 세마포어가 관리하는 자원을

최대 n개의 thread까지 동시에 사용할 수 있도록 할 수도 있지만 이러한 방법은 주로 사용되지 않는다고 한다.

### 2.3.3 Something to Check with

세마포어 값을 1로 초기화 하여 세마포어를 사용할 때, 공유 자원과 세마포어의 인스턴스가 직접적으로 binding되지는 않는다. 즉, 공유 자원은 공유 자원대로 세마포어는 세마포어대로 정의된다는 뜻이다. 따라서 어떤 thread에서 공유 자원에 대한 사용을 마쳤으면 마친 것에 대한 함수를 실행시켜야 한다.

### 2.3.4 Functions

이러한 방식으로 사용되는 세마포어를 활용하기 위해 정의된 함수는 총 4가지가 있다.

#### sema\_init

이 함수는 세마포어의 value를 initialize한다. 이 함수를 사용할 때는 세마포어를 우선 정의하고 매개변수로 넘겨줘야 한다. 이때 initialize 할 value 값도 함께 넘겨준다. 이 함수가 호출된 시점에 waiters 도 초기화해준다.

#### sema\_down

이 함수는 intr\_disable 함수를 호출하여 interrupt를 끄고 세마포어의 value가 0인 경우, thread\_block 함수를 호출하여 sema\_down 함수를 호출한 thread(thread A라고 하자)를 block 시킨다. thread A가 block 될 때 세마포어의 waiters에 thread A가 push back된다.

만약 세마포어의 value가 0인 상태에서 thread A가 다시 스케줄링 된다면 while loop의 조건이 참이기 때문에 다시 block된다. 즉, 세마포어의 value가 0이 아닐 때까지 block된다. 그러다가 세마포어의 value가 0이 아닌 값이 되고 thread A가 다시 스케줄링 된 경우 세마포어 value를 1만큼 줄이고 interrupt를 함수 호출 이전 상태와 같게 만든 후 리턴한다.

이 함수에서는 thread\_block 함수 호출이 일어나 thread가 block될 수 있으므로 external interrupt handler에 의해서 호출되는 상황을 방지해야한다.

#### sema\_try\_down

이 함수도 sema\_down 함수와 같이 세마포어의 value를 1만큼 줄이는 역할을 한다. 또한 이 함수 역시 interrupt를 끄고 리턴하기 직전에 다시 interrupt를 원래 상태로 만들어 중간에 다른 thread에 의해 선점당하지 않게, atomic하게 실행되게 한다. 다른 점은 return 값이 참 또는 거짓으로 존재한다는 것인데 value를 줄이는 것을 성공하였으면 참을, 실패하였으면 거짓을 리턴한다. sema\_down과 또 다른 점은 thread\_block 함수를 호출할 일이 없어 external interrupt handler에 의해서 호출될 수 있다는 점이다. 이 함수의 이러한 특징 때문에 tight 한 loop에서 이 함수를 호출하는 것은 CPU time을 버리는 일이라는 것인데, 세마포어 값이 0인 상태로 꽤 오랜

기간 머무는 경우, tight loop에서 이 함수를 호출하는 행위는 busy waiting을 하는 상황과 동일하다. 따라서 유용한 작업을 하지 않고 cpu를 점유하는 상황이 생기므로 피해야한다.

#### sema\_up

이 함수 역시 위의 두 함수와 마찬가지로 atomic하게 실행된다.

이 함수는 세마포어의 waiters가 비어있지 않은 경우, waiters의 가장 앞에 있는 thread를 매개 변수로 하여 thread\_unblock 함수를 호출한다. (이때, 해당 thread는 waiters에서 제거된다.) 즉, 세마포어 값의 변화를 기다리고 있는, THREAD\_BLOCKED 상태에 있는 thread 를 THREAD\_READY로 만들어 스케줄링될 준비를 시키는 것이다. 그리고 세마포어 값을 1 증가 시켜 해당 세마포어가 관리하는 공유 자원을 사용하려고 하는 thread가 스케줄링 된 경우 공유 자원에 접근할 수 있게 한다. 세마포어의 value를 0으로 initialize 하는 경우에는 이 함수가 호출되어 세마포어의 value를 1로 바꾸고, 그에 따라 seam\_down 함수의 while loop의 조건이 거짓이 되어 더이상 block 되지 않게 된다. 그리고 곧바로 세마포어 value를 다시 줄여 다른 thread가 자원에 접근하는 것을 막는다.

이 함수에서 사용되는 thread\_unblock 함수는 thread가 block되는 상황을 직접적으로 만들지는 않으므로 external interrupt handler에 의해서 호출되어도 문제 없다.

## 2.4 Locks

락은 세마포어의 특수한 구현이다. 세마포어에 두 가지 조건을 추가하면 락과 동일해진다.

첫 번째 조건은 초기화하는 value가 1이라는 것이다. 따라서 1로 초기화된 세마포어에서 sema\_down 함수를 호출하는 것은 락(공유 자원에 대한 접근 권한)을 얻는 것으로 해석할 수 있고, sema\_up 함수를 호출하는 것은 락(공유 자원에 대한 접근 권한)을 해제하는 것으로 해석 할 수 있다.

두 번째 조건은 하나의 thread만이 락을 소유할 수 있다는 것이다. 락을 소유한다는 것의 의미는 락에서 struct semaphore를 활용하여 정의된 value 값을 변화시킬 수 있다는 것이다.

이러한 특징을 가지는 락 구조체의 정의를 살펴보면 struct thread \* 타입을 가지는 holder 변수 와 struct semaphore 타입을 가지는 semaphore 변수가 정의되어 있다.

holder를 통해 어떤 thread가 락을 소유하고 있는지 나타낼 수 있고, semaphore를 통하여 락 을 구현할 수 있다. 이때 semaphore로 정의된 변수는 락의 구현을 위해 존재하므로 이전 세마 포어이다.

### 2.4.1 Functions

#### lock\_init

이 함수는 lock을 initialize 한다. 기본적으로 sema\_init 함수를 활용하지만 한 가지 다른 점은 락의 개념으로 정의한 struct lock은 holder를 가지고 있기 때문에 holder에 대한 초기화도 진행 한다.

락을 초기화하는 시점에는 락을 소유하고 있는 thread가 없으므로 holder는 NULL이다.

### lock\_acquire

이 함수는 lock을 획득하려는 특정 thread에 의해 호출되어 락을 획득한다.

함수가 호출되면, 먼저 현재 thread가 매개변수로 들어온 락을 보유하고 있는지 확인한다. 우리가 현재 사용하는 락은 재귀적이지 않으므로, 이미 보유하고 있다면 락을 다시 획득하는 것은 잘못된 동작이다. 따라서 현재 락을 소유하고 있지 않은 thread가 이 함수를 호출해야 하며 이를 ASSERT를 사용하여 확인한다.

그 후, sema\_down 함수를 호출하여 락의 semaphore의 value를 감소시킨다. 세부 동작은 sema\_down 함수와 동일하여 생략한다. 간단히 언급하면 semaphore의 value가 1인 경우 값을 0으로 만들고, 1인 경우 0이 되는 것을 기다리기 위해 block된다.

sema\_down 함수를 통해 락을 소유한 경우에는 holder 변수를 현재 실행 중인 스레드로 설정하여 현재 thread가 락을 보유하고 있음을 기록한다.

이 함수에서는 sema\_down 함수가 사용되므로 thread\_block 함수가 호출될 수 있어, external interrupt handler에 의해서 호출되면 안된다.

### lock\_try\_acquire

이 함수는 sema\_try\_down 함수를 락에 맞추어 변형한 것으로, sema\_try\_down 함수가 참 값을 리턴하면 holder를 현재 실행 중인 thread로 기록한다.

이 함수 역시 lock\_acquire 함수와 같이, 락을 소유하고 있지 않은 thread에서 호출되어야 한다.

### lock\_release

이 함수는 이 함수를 호출하는 thread가 가진 락에 대한 소유를 해제하는 함수이다. 따라서 이 함수를 호출한 thread가 락을 소유하고 있다는 것을 보장해야하며, holder 값을 NULL로 설정 한다. 최종적으로 sema\_up 함수를 호출하여 semaphore의 value를 1로 만든다. 이때, 락이 release 되는 것을 기다리고 있는 thread를 매개변수로 하여 thread\_unblock 함수를 호출하여 해당 thread가 스케줄링될 준비를 하도록 한다.

### lock\_held\_by\_current\_thread

이 함수를 호출한 thread가 결국 이 함수 내부의 thread\_current 함수가 리턴하는 thread일 것인데 그 thread가 lock을 소유하고 있는지 참 또는 거짓으로 리턴한다.

관련해서 임의의 thread가 lock을 소유하고 있는지 확인하는 함수를 구현하여 이 함수의 기능을 보다 일반화할 수 있다고 생각할 수도 있는데 이는 위험한 접근이다. 왜냐하면 아래와 같은 상황이 발생할 수 있기 때문이다.

예를 들어, thread A는 thread B가 락을 소유하고 있다는 것을 확인하고, 이 정보를 기반으로 다른 작업을 수행하기로 했다고 할 때, 이후에 thread B가 락을 해제하고 thread C가 락을 획득하면 thread A가 판단한 정보는 더 이상 정확한 정보가 아니게 되므로 의도하지 않은 동작이 발생할 수 있다.

## 2.4.2 example of using lock

락이 어떤 방식으로 사용되는지 pintos 코드의 일부를 통해 알아보자.

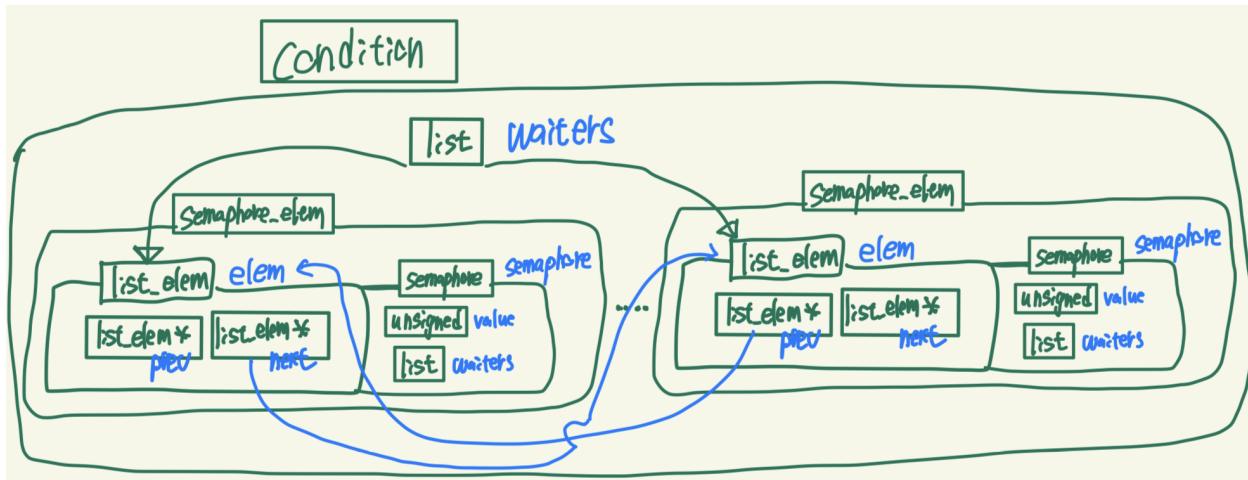
thread system을 initialize할 때 호출되는 thread\_init 함수에는 thread마다 부여되는 고유한 번호인 thread id에 대한 락을 초기화 하는 코드가 있는 것을 확인할 수 있다. thread id를 할당하는 함수인 allocate\_tid 함수의 동작을 생각해보면, thread id는 서로 다른 thread 사이에서 동시에 접근하면 안되는 공유 변수라는 것을 알 수 있다. 만약 static 변수인 next\_tid로 새로운 thread id를 만들고 이를 allocate\_tid 함수를 호출한 부분으로 전달하는 과정에 수행 중인 thread가 block되고 새로 스케줄링 된 thread가 next\_tid 값을 변경시킨다면 의도한대로 동작하지 않을 것이다. 따라서 이를 막기 위해 새로운 thread id를 만드는 과정 중에는 다른 thread가 next\_tid에 접근할 수 없도록 락을 건다. allocate\_tid 함수가 리턴되는 과정까지 락을 거는 것은 아니지만 tid라는 지역 변수에 값을 저장하기 때문에 switching이 되어도 의도한대로 동작한다.

## 2.5 Monitors

모니터는 락과 세마포어를 조합하여 만들어 낸 higher level의 synchronization primitive이다. 모니터는 락과 세마포어를 활용하여 만드는 것인 만큼 조금 더 넓은 범위로 synchronization을 커버한다. 모니터 개념을 이해하기 위해서는 두 가지 개념이 필요한데 컨디션과 모니터 락이다.

### 2.5.1 Condition

우선 컨디션에 대해 알아보자. 컨디션 개념은 코드로 struct condition으로 구현된다. struct condition은 struct list 타입의 waiters 변수 하나를 가진 구조체이다. 아래 그림을 보며 이해해보자.



waiters 변수는 본질적으로 세마포어를 원소로 하는 리스트이다. 현재 구현상 struct list 타입을 가지는 waiters를 구성하는 원소는 struct semaphore\_elem 이다. 이는 synch.c 파일에 정의되어 있으며 synch.h에 없는 것에 미루어 보았을 때 모니터를 동작시킬 때, 외부에서 볼 필요가 없어 외부로부터 숨겨둔 것이다. struct semaphore\_elem은 struct list\_elem 타입을 가지는 elem 변수와 struct semaphore 타입을 가지는 semaphore 변수로 이루어진 구조체이다. elem 변수는 struct semaphore\_elem을 리스트로 다룰 수 있는 기능을 제공해주며, 공유 변수에 대한 synchronization은 실질적으로 semaphore 변수를 활용하여 진행된다.

## 2.5.2 Monitor Lock

다음으로 모니터 락이라는 개념이 존재한다. 모니터를 일종의 공유 변수로 생각하여 이에 대한 락을 모니터 락이라고 부르는 것이다. 위에서 살펴본 struct condition에 모니터 락을 강제하는 구현은 되어있지 않다. 하지만 모니터를 정확하게 사용하기 위해서는 모니터락을 따로 선언하여 사용하여야 한다. 모니터 락을 얻은 경우, “모니터 안에 있다”고 표현하며 모니터 락을 얻은 thread는 모니터가 관리하는 자원에 접근할 수 있다.

## 2.5.3 Functions

### cond\_init

이 함수는 모니터를 초기화한다. 이는 단순히 struct list 타입을 가지는 struct condition의 waiters 변수를 초기화 하는 것과 같다. 모니터는 세마포어와 락을 활용한 higher level의 synchronization을 구현한 것이기 때문에 초기화 단계에서 많은 작업을 수행하지는 않는다.

### cond\_wait

기본적인 동작을 풀어 설명하면 다음과 같다.

1. 모니터의 waiters 변수(struct list 타입)에 push back 할 semaphore\_elem 타입의 waiter 를 선언한다.
2. waiter의 멤버변수인 semaphore 변수(struct semaphore 타입)의 value 0으로 초기화하여 waiters 에 push back한다.
3. 그 후 모니터 락을 풀고, 방금 초기화 한 세마포어에 대해 sema\_down 함수를 실행시켜 cond\_wait를 호출한 thread를 block 시킨다.
4. 이후 block 된 thread가 스케줄링 되는 경우에, cond\_wait 함수의 추상화 레벨에서 본다면 모니터 락을 얻는 것으로부터 동작한다.

pintos에서 모니터의 구현은 시그널을 보내고 받는 과정을 atomic하게 진행하지 않는다는 것이 특징이다.(세마포어를 구현한 함수를 통해 구현된 것을 보면 알 수 있다: thread\_block 함수와 thread\_unblock 함수의 동작 참고) 따라서 이 함수의 호출된 이후 리턴되는 경우 이 함수를 호출한 곳에서 모니터 락을 다시 한 번 체크해야 한다. 만약 모니터 락이 없다면 다시 cond\_wait 함수를 호출해서 모니터 락을 기다려야 한다.

이 함수는 sema\_down 함수 호출을 하므로 block 될 수 있다 따라서 external interrupt handler 에서 호출되면 안된다. 그리고 모니터 락을 얻은 thread에 대해서만 동작한다는 것도 보장해야 한다.

### **cond\_signal**

이 함수는 컨디션을 기다리고 있는 thread 중 하나에 wake up 신호를 보내는 기능을 한다.

컨디션의 waiters가 존재한다면 waiters에서 pop front 연산을 하여 해당 원소의 세마포어를 매개변수로 sema\_up 함수를 호출한다. 즉, 컨디션을 기다리고 있는 thread가 실행될 준비를 시키는 것이다.(schedule 함수가 호출될 때 컨디션을 기다리는 thread의 실행이 된다.)

이 함수는 cond\_wait 함수와 대응되는 함수이며 역시 모니터 락을 얻은 thread에 대해서만 동작한다는 것도 보장해야 한다.

### **cond\_broadcast**

이 함수는 cond\_signal 함수를 컨디션의 waiters의 모든 원소에 대해 호출한다. waiters가 빌 때까지 cond\_signal 함수를 호출하는 것으로 구현되어 있다.

## **2.6 Extra Details**

list.h에 정의되어있는 macro 중에 `list_entry` 는 리스트 요소를 포함하는 외부 구조체의 포인터로 변환하는 데 사용된다. 이를 사용해 리스트 요소가 어떤 복잡한 구조체에 들어있든 Doubly Linked List를 만들어서 관리할 수 있게 된다.

# 3. Alarm Clock Design Plan

## 3.1 Current Implementation

현재 Alarm Clock의 implementation은 devices/timer.c에 위치해 있다. 현재 구현에서의 `timer_sleep` 함수는 busy waiting 방식으로, tight loop 안에서 `thread_yield` 함수를 계속 호출하여 반복적으로 yield를 하는 방식으로 sleep 이 구현되어있다. 이런 방식은 thread를 sleep 시킨 상태여도 thread는 계속 실행되며 CPU를 사용하기 때문에 매우 비효율적이다.

## 3.2 Our Design Plan

우리 팀의 아이디어는 thread를 sleep 시킬때, `thread_yield` 함수를 사용하여 thread를 `ready_list`에 추가하는 것이 아닌, 새로 정의한 `sleep_list`에 추가하여 `timer_sleep` 함수의 매개 변수만큼의 시간이 경과하였을 때만 다시 thread가 실행되도록 하는 것이다.

이를 위해서는 현재의 TCB를 thread가 깨어나는 시점에 대한 정보를 저장할 수 있도록 수정해야 한다. 그리고 sleep 하고 있는 thread를 관리하는 `sleep_list`의 선언도 필요하다. 이는 thread system을 초기화 할 때 함께 초기화 해줄 생각이다.

추가적으로 만들 함수는 `thread_sleep` 함수이다. 이 함수는 현재 `timer_sleep` 함수가 사용된 곳에서 호출할 생각이며 함수의 이름대로 thread를 sleep시키는데 활용할 것이다. 구체적으로는 함수가 깨어날 시간을 TCB에 저장하고 `sleep_list`에 sleep시킬 thread(이 함수를 호출한 thread)를 추가한다. 그리고 `thread_block` 함수를 통해 스케줄링 되지 않도록 할 것이다.

지금까지 어떻게 thread를 sleep 시킬 것인지에 대한 이야기를 마쳤다. 기존의 방식은 busy waiting 방식이므로 깨어나야할 시간이 되면 조건문에 의해 thread가 sleep에서 벗어나는 방식이었기에 sleep된 thread를 다시 깨우는 것에 대한 별다른 구현은 필요 없었다. 하지만 변경할 방식에서는 sleep 하는 thread는 block된 상태이기 때문에 thread를 unblock 시키는 처리가 필요하다. 따라서 tick이 1만큼씩 커지는 매 순간에 깨어날 thread가 있는지를 확인하고 만약 깨어날 thread가 있다면 block 상태에서 ready 상태로 변경해주어야 한다. 이런 기능을 하는 `thread_wakeup` 함수를 만들 예정이고 `timer_interrupt` 함수 안에서 사용할 예정이다.

한 가지 추가로 고려할 사항은 다음과 같다. sleep 하는 thread가 여럿일 수 있고 `timer_interrupt`가 호출되는 매 순간 `sleep_list`를 순회하여 깨울 thread를 찾는 것은 비효율적이라고 생각했다. 따라서 `sleep_list`에 sleep 할 thread를 추가할 때 깨어날 tick을 기준으로 오름차순으로 정렬되게 하여 맨 앞(그리고 맨 앞 thread가 일어날 tick과 같은) thread만 확인하여도 문제없도록 구현할 예정이다.

# 4. Priority Scheduler Design Plan

## 4.1 Current Implementation

현재 Pintos에서 scheduler는 priority를 고려하게 구현되어있지 않다. priority를 설정하거나 불러올 수 있는 `thread_set_priority`, `thread_get_priority` 같은 함수들은 존재하지만, 막상 `schedule` 함수의 실행과정에서는 사용이 되지 않고 있다. `ready_list`에 `push_back` 된 순서로 round robin 방식으로 스케줄링된다.

## 4.2 Important Concepts

### Priority Inversion

Priority Inversion은 운영체제에서 발생할 수 있는 문제 상황으로, 우선순위가 높은 스레드가 우선순위가 낮은 스레드 때문에 원하는 자원에 접근하지 못하게 되어 불필요하게 대기하는 현상을 말한다. priority가 높은 thread가 priority가 낮은 thread와 자원을 공유하고 있을 때, priority가 낮은 thread가 그 자원에 lock을 걸어서 priority가 더 높은 thread가 CPU를 사용하지 못하는 상황이 대표적인 예시이다.

### Priority Donation

Priority Donation은 Priority Inversion 문제를 해결하기 위한 하나의 기술이다. 기본적으로 Priority Donation은 우선순위가 높은 스레드가 더 낮은 우선순위의 스레드 때문에 필요한 자원에 접근 못하게 되었을 때, 높은 우선순위 스레드의 우선순위를 낮은 우선순위의 스레드에게 빌려주는 방식으로 작동한다. 그렇게 하여 스레드의 우선순위를 일시적으로 끌어올려 작업을 빨리 끝내게 함으로써 lock이 걸린 자원이 보다 더 빨리 해제될 수 있도록 한다.

### Extended Donation

Extended Donation은 연쇄적, 다중, 또는 그 외의 복잡한 상황에서의 우선순위 기부를 관리하고 효과적으로 Priority Inversion 문제를 처리하기 위한 전반적인 접근 방식을 포괄하는 용어이다.

### Multiple Donation

Multiple Donation은 하나의 스레드가 여러 다른 스레드들로부터 동시에 우선순위를 기부받는 경우이다. 예를 들어, 스레드 A와 B 둘다 스레드 C의 락이 해제되기를 기다리고 있을 때, A와 B의 우선순위가 C에게 동시에 기부될 수 있다.

### Nested Donation

Nested Donation은 스레드의 우선순위 기부가 연쇄적으로 일어나는 경우이다. 스레드 A가 락을 기다리고 있고, 그 락을 점유하고 있는 스레드 B가 다른 락에 대해 스레드 C를 기다릴 때, A의 우선순위가 B에게 기부되었다면, 그 후 B의, 혹은 B가 기부받은 A의 우선순위가 C에게 기부될 수 있다.

## 4.3 Our Design Plan

schedule 함수의 next\_thread\_to\_run 함수를 보면 ready\_list에서 list\_pop\_front 함수로 다음 번에 실행할 thread를 정하는 것을 알 수 있다. thread\_unblock 함수와 thread\_yield 함수에서만 ready\_list에 thread를 추가하는데, 이때 thread를 ready\_list로 push back 하지 말고 우선순위에 맞게 정렬된 상태가 되게끔 ready\_list에 thread를 추가하고자 한다. 우리 팀은 이러한 방법을 통해, 우선순위가 높을수록 먼저 스케줄링 되도록 하는 기능을 구현하려고 한다.

이때 lock, semaphore, condition variables을 기다리고 있는 상태인 blocked 상태라면, blocked된 thread 중에서 가장 우선순위가 높은 것이 먼저 unblock되도록 해야한다. 이를 위해 세마포어 또는 컨디션 변수의 waiters에 저장되는 thread를 우선순위 순으로 정렬하여 구현하려고 한다.

또한 우선순위가 높은 thread가 cpu를 선점하는 기능을 구현하기 위해 thread 생성시에 현재 running 하고 있는 thread와 우선순위 비교를 하려고 한다. 그리고 thread\_set\_priority 함수로 현재 실행 중인 thread의 우선순위를 바꿀 때 ready list의 가장 우선순위가 높은 값과 우선순위를 비교하려고 한다. 이러한 비교를 통해 우선순위가 높은 thread가 cpu를 선점할 수 있도록 한다.

우리 팀의 보다 구체적인 design plan은 다음과 같다. 우리는 `ready_list`에서 다음으로 실행될 thread를 선정하는 방식에 priority를 고려하도록 할 것이다. 그럴려면 `ready_list`에 있는 threads들이 각각의 priority에 따라 정렬이 되어야 한다. 그 이유는 현재 실행 중인 스레드가 `thread_yield` 함수 또는 `thread_unblock` 함수를 실행할 때마다 `ready_list`에 있는 thread들 중에 priority가 가장 크면서도 현재 실행 중인 thread보다 priority를 가지는지 확인을 해야되기 때문이다. 정렬이 되어있지 않은 상황이라면, 매번 `ready_list`를 검사하기 위해 linear time의 시간을 소모해야된다는 문제가 생긴다. 때문에, `ready_list`에 새로운 thread를 추가할 때, priority 기준으로 된 정렬이 유지될 수 있도록 `list_insert_ordered` 함수를 사용할 것이다.

하지만 이러한 priority 기반 `ready_list`에는 문제가 될 수 있는 extended donation 상황이 현재 두개 존재한다. 바로 multiple donation과 nested donation 상황이다. 저 상황들에 대해 대처할 수 있도록 디자인을 하지 않는다면 priority inversion 문제가 발생할 수 있다.

위의 모든 상황에 대해 대처할 수 있도록 코드를 짜야 한다. 먼저 `struct thread`에 priority donation의 구현을 위한 멤버 변수들을 추가해야 한다. priority donation이 끝난 다음 priority

를 되돌리기 위해서는 (A)donation을 하기 직전의 priority 값을 저장하고 있는 변수가 필요하다. 또한, (B)thread가 기다리는 lock을 저장하고 있는 변수(lock을 통해 공유 자원에 대한 접근을 control하므로), (C)thread가 점유한 lock을 기다리는 threads들을 저장할 변수가 필요할 것이다.

멤버 변수들을 추가하고 나면 현재의 `lock_acquire` 함수의 구현을 바꿔야 한다. 일단 현재 thread가 기다리는 lock을 이미 점유한 thread가 존재하는지 확인을 한다. 그 다음엔 그 thread가 존재를 한다면, 현재 thread의 (B)를 그 lock으로 설정 후, (B)의 `holder`, 즉 lock을 점유한 thread의 (C)에 현재 thread를 추가한다.

(B)가 필요한 이유는 nested donation을 고려해서이다. Nested donation 상황이 구현되려면 여러 thread들이 또 다른 lock으로 연결되어 chain dependency가 생긴 상황에서, 현재 lock을 점유하고 있는 thread에게 기다리고 있는 thread들이 donation이 이루어져야 한다. 이를 위해 선 각각의 thread에서 (B)의 holder 정보를 사용해서 dependency chain을 traverse할 수 있어야 한다. 그래서 (B)가 필요하다.

(C)가 필요한 이유는 현재 스레드에 lock이 걸려있는 스레드들이 있는지 모든 스레드들을 검사하지 않고도 tracking 하기 위해서이다.

현재의 `lock_release` 함수 또한 수정이 필요하다. `lock_release` 함수에서 `sema_up`을 실행하기 전, 먼저 현재 thread의 (C)를 traverse하며 (C)에 담긴 모든 threads를 (C)로부터 제거해야 한다. 이후, lock이 해제되며 변화가 일어난 (C)를 통해 현재 thread의 priority를 새로 계산해야 한다.

위에 내용이 구현이 되었다면, `sema_down` 함수의 실행이 끝났다는 의미는 현재 thread가 lock을 점유하게 되었다는 의미가 된다. 그렇다면 `sema_down` 실행 후 현재 thread의 (B)를 `NULL`로 설정해주어야 한다.

`thread_set_priority` 는 주어진 `priority` 인자 값으로 thread의 priority를 설정하는 함수이다. `thread_set_priority`로 인해 priority가 바뀌는 경우에는 이에 맞춰 priority를 다시 계산하고 필요하다면 donation을 중단하는 과정도 구현해야한다. 이는 (C)로부터 priority를 계산하는 과정을 함수로 만들면 가능하다.

`thread_get_priority` 는 스레드의 priority를 반환하는 함수로, 우리의 계획으로는 특별한 변형이 필요할 것 같지는 않다.

## 5. Advanced Scheduler Design Plan

### 5.1 Current Implementation

기존 Pintos 상의 scheduler는 대부분 함수들의 implementation이 priority에만 기반하여 실행되고 있는 상태기 때문에, priority가 낮은 thread의 경우 실행되기까지 상당히 시간이 오래 걸리거나 실행이 되지 않을 수 있다. 현재 MLFQS에 관련된 알고리즘은 코드 상 존재하지 않는다.

## 5.2 Important Concepts

### Round Robin

thread간에 별도의 priority 값을 두지 않고 선입선출 순서대로 실행시키는 알고리즘이다. thread가 실행이 되면, 그 thread를 queue의 뒤로 이동시켜 모든 thread가 무조건 실행이 되도록 하는 알고리즘이다.

### MLFQS (Multi Level Feedback Queue Scheduler)

Pintos에서 MLFQS는 priority 값을 기준으로 구분되는 64 개의 ready queue를 운영한다. thread마다 가지는 `recent_cpu`, `nice` 값과 시스템이 가지고 있는 값 `load_avg` 등을 이용하여 동적으로 priority 값을 갱신해주는게 특징이다.

### Floating Point Operation - Integer Operation Conversion

Floating Point Operation들은 그 자체로도 꽤 무겁기 때문에 Pintos kernel에서는 지원하지를 않고 있다. 그렇기에 integer operation으로 변환해서 진행해주어야 한다. Pintos 공식 문서 B.6 Fixed-Point Real Arithmetic에서 자세히 설명하고 있는데, 원리를 간단히 요약하자면 우측에 존재하는 bits들을 분수로 생각하는 것이다.

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table,  $x$  and  $y$  are fixed-point numbers,  $n$  is an integer, fixed-point numbers are in signed  $p.q$  format where  $p + q = 31$ , and  $f$  is  $1 \ll q$ :

Convert $n$ to fixed point:	$n * f$
Convert $x$ to integer (rounding toward zero):	$x / f$
Convert $x$ to integer (rounding to nearest):	$(x + f / 2) / f$ if $x \geq 0$ , $(x - f / 2) / f$ if $x \leq 0$ .
Add $x$ and $y$ :	$x + y$
Subtract $y$ from $x$ :	$x - y$
Add $x$ and $n$ :	$x + n * f$
Subtract $n$ from $x$ :	$x - n * f$
Multiply $x$ by $y$ :	$((int64_t) x) * y / f$
Multiply $x$ by $n$ :	$x * n$
Divide $x$ by $y$ :	$((int64_t) x) * f / y$
Divide $x$ by $n$ :	$x / n$

## 5.3 Our Design Plan

우리의 design plan은 다음과 같다.

일단, MLFQS의 기본이 될 64개의 ready queue를 어떻게 구현할지를 정해야한다. 우리는 MLFQS가 옵션으로 주어졌을때만 작동하록 구현을 해야되기 때문에, 현재 존재하는 `ready_list` 를 수정해서는 안된다. 대신 새로 `mlfqs_ready_lists` 를 `struct list` 가 64개가 붙어 있는 배열로 만들어주어서 MLFQS 옵션이 주어졌을 때 사용할 수 있도록 하면 된다.

`struct thread` 에는 멤버 변수 `nice`, `recent_cpu` 를 추가해주어야 한다. 또한 전역변수로 `mlfqs_ready_lists[64]`, `load_avg`, `ready_threads` 를 추가해주어야한다. 이러한 전역 변수로 추가된 값들은 `thread_init` 에서 `thread_mlfqs` 가 `true` 일시 다른 전역 변수들과 같이 초기화를 해줘야한다.

64개의 ready queue가 이제 준비되었으니 이젠 MLFQS의 핵심인 실시간으로 갱신되는 priority 알고리즘을 짜줘야한다. 이 갱신이 기준이 될 event는 바로 `thread_tick` 이다. 현재 MLFQS 시스템 상에서 각 thread의 priority에 영향을 주는 변수들은 `recent_cpu`, `nice`, `load_avg` 가 있다. 이 값들이 이제 일정 tick마다 바뀌면서 이에 맞춰 priority 또한 갱신되어야 한다. 일단 매 tick 마다 현재 실행되고 있는 thread의 `recent_cpu` 에 1을 더해주어야 한다. `load_avg` 를 계산하기 위해서 추가되어야하는 변수 `ready_threads` 는 업데이트 시점의 `ready_list` 의 threads 개수와 `running_thread` 의 개수를 더한 값을 가진다.

그리고 매 초마다 `load_average` 와 `recent_cpu` 가 계산되어야 한다. 이 때 사용되는 식은 아래와 같다. `recent_cpu` 는 Exponentially Weighted Moving Average를 기반으로 만들어진 식을 사용했으며.

$$\text{load\_avg} = (59/60) \times \text{load\_avg} + (1/60) \times \text{ready\_threads}$$

$$\text{recent\_cpu} = (2 \times \text{load\_avg}) / (2 \times \text{load\_avg} + 1) \times \text{recent\_cpu} + \text{nice}$$

이렇게 수정되어가는 변수들을 기반으로 4 tick마다 thread의 priority를 새로 갱신해줘야한다. 이 때 사용되는 식은 아래와 같다.

$$\text{priority} = \text{PRI\_MAX} - (\text{recent\_cpu}/4) - (\text{nice} \times 2)$$

그리고 이렇게 priority가 바뀐 thread들은 `mlfqs_ready_lists` 에서 priority 값에 맞게 위치를 조정해줘야한다.

이제 MLFQS의 가장 기본적인 조건인 priority기반 ready queue 배열과 실시간 priority 조정이 갖추어졌으니, scheduling 과정만 신경을 쓰면 된다. Scheduling은 `mlfqs_ready_lists` 중 가장 큰 priority로부터 시작하여 비어있지 않는 list가 나올때까지 priority를 내려가며 검사할 것이다. thread가 들어있는 list가 있다면, Round Robin 방식으로 list의 가장 앞에 있는 thread를 실행 할 것이다. 다 검사를 했을 때 thread가 있는 list가 없다면, idle thread를 실행할 것이다.

마무리로는 MLFQS 옵션이 주어졌을 때 function behavior가 달라지도록 맞춰주는 것이다.

`thread_tick` 함수에서는 MLFQS 옵션이 주어지면 일정 tick마다 여러 값들을 갱신해주는 내용을 넣어줘야한다.

`thread_block` 과 `thread_unblock` 처럼 `ready_list` 와 관련된 operation을 진행하는 함수들에서도 MLFQS 옵션 존재시 `ready_threads`를 변경하도록 해야 한다.

`next_thread_to_run` 역시 `ready_list`가 아닌 `mlfqs_lists`의 highest index 배열 리스트에서 가장 앞의 값을 pop 하는 방식으로 변경하여야 한다.