Project 1 : Final Report

Team7 컴퓨터공학과 20210741 김소현 컴퓨터공학과 20210774 김주은

---

1. Analysis of synchronization

A. Sychronization이란?

Synchronization은 critical resource를 공유하는 2개 이상의 thread들의 concurrent execution을 말한다. 여러 thread들 간에 critical resource를 사용할 때 conflict를 방지하려면 thread들이 synchronize되어야 한다.

B. Meaning and their implementation in pintos

1) Semaphore

- meaning

  Semaphore은 2개 이상의 thread가 shared resource나 critical section 등에 접근할 때 여러 process 혹은 thread가 접근하는 것을 막아준다.

- implementation

  - **struct**

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

  `value` : semaphore의 현재 상태 : 사용 가능한 공유 자원의 개수를 나타내는 변수

  `waiter` : semaphore을 기다리는 thread들의 list : 공유 자원을 사용하기 위해 대기하는 waiters의 list

  - **function**

    - **void sema_init (struct semaphore *, unsigned value);**

    semaphore을 초기화하는 함수로, semaphore가 사용 가능하다면 1 이상이, 그렇지 않다면 0이 value로 assign 될 것이다. 공유자원을 사용하기 위해 여러 thread가 sema_down되어 대기하고 있고, 공유자원을 사용하던 thread가 사용한 후 sema_up할 때 어떤 thread가 가장 먼저 공유자원을 사용할 것인가에 대한 문제를 해결하는 것이 semaphore 관련 개념이라고 할 수 있다.

    - **void sema_down (struct semaphore *);**

    sema_down은 running중인 thread가 semaphore을 요청할 때 사용하는 함수다. 해당 함수는 semaphore가 사용이 가능한지 여부에 상관없이 일단 실행된다. semaphore의 value가 0(사용불가능한 상태)이라면 waiters list에 들어가고 block 상태가 된다. block 상태가 되면 thread는 thread_block()에서 멈추며 semaphore의 value가 positive가 될 때까지 기다리는 것이다. 이후에 sema_up()이 호출되면 thread는 thread_block()부터 시작하게 된다.

    - **bool sema_try_down (struct semaphore *);**

    sema_down과 기능은 비슷하나 semaphore의 값이 0이면 즉시 반환하며 이때 성공 여부를 반환한다. semaphore의 값이 positive이면 값을 1 감소시키고 true를 반환하며 0인 경우에는 block되지 않고 false를 바로

반환한다.

- **void sema_up (struct semaphore *);**

대기 중인 thread가 waiter list에 있으면 첫번째 thread를 깨워서 실행 가능하게 하며, semaphore 값을 1만큼 증가시킴으로써 사용 가능한 공유 자원을 하나 증가시켜주는 함수다.

- **void sema_self_test (void);**

semaphore의 기본 기능을 테스트하기 위한 함수이다. 2개의 semaphore을 초기화하고 새로운 thread를 생성하여 sema_test_helper을 실행한다. 이후 첫번째 semaphore을 1만큼 증가시키고 두번째 semaphore을 1만큼 감소시키는 과정을 10번 반복한 후 test를 종료한다.

- **static void sema_test_helper (void *sema_);**

sema_self_test와는 정반대로 up down 연산을 수행한다.

- sema_self_test와 sema_test_helper함수를 같이 분석해보면

  1. 메인 thread인 sema_self_test에서 2개의 semaphore을 0으로 초기화한다.
  2. 메인 thread(sema_self_test)에서 새로운 thread를 생성하고 sema_test_helper을 호출한다.
  3. 메인 thread에서는 sema[0]에 대해 sema_up 연산을 10번 수행하여 value가 10이 된다.
  4. 이후 sema[1]에 대해 sema_down 연산을 10번 수행하는데 이 때 이 연산은 sema[1]의 값이 0이기 때문에 메인 thread를 block 상태로 만들고, sema[1]이 positive가 되도록 기다린다.
  5. sema_test_helper 함수가 실행되는 새로운 thread에서는 sema[0]에 대해 sema_down 연산을 10번 수행한다.
  6. 메인 thread에서는 sema_up으로 sema[0]을 10으로 만들어놨기 때문에 새로운 thread는 sema_down을 10번 수행해도 모두 성공적으로 수행된다.
  7. sema[1]에서 sema_up 연산이 수행하면서 값이 positive가 되기 때문에 main thread에서 대기 중이던 sema_down 연산이 수행된다.

  이러한 방식으로 thread의 기본적인 synchronization mechanism을 확인할 수 있다.

2) Lock

- meaning

  synchronization mechanism 중 하나로 여러 thread나 process가 동시에 공유 자원에 접근할 때 발생할 수 있는 **race condition을 방지**하기 위해 사용된다. thread가 lock을 획득하면 해당 thread만이 그 lock이 보호하고 있는 resource나 code section에 접근할수 있게 되며 다른 thread들은 해당 lock이 release될 때 까지 대기해야 한다. semaphore와 비슷하게 동기화 문제를 해결하기 위한 도구이지만 사용 방식이나 구현 방식이 다르다.

  - **한 번에 하나의 thread만**이 공유 자원이나 critical section에 접근할 수 있으며, 다른 thread들은 해당 lock이 release될 때까지 대기해야 한다.
  - semaphore의 경우 특정 thread가 semaphore을 내릴수도 있고 다른 thread는 그것을 올릴 수 있다. 그러나 lock의 경우 lock을 요청한 thread가 lock을 보유하며, 해제할 수 있는 것도 lock을 획득한 thread만 가능하다.

- implementation

  - **struct**

```
/* Lock. */
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

`holder` : lock을 보유하고 있는 thread의 pointer을 저장한다. 이는 debugging을 위해 사용될 수 있는데 어떤 thread가 lock을 보유하고 있는지, lock을 해제하지 않고 끝난 thread가 있는지 등을 파악할 수 있다.

`semaphore` : 여기서는 semaphore가 binary semaphore로 사용되며 0 또는 1의 값을 가진다. thread가 lock을 획득하려고 할 때, semaphore의 값이 0, 1 중 어떤 값이냐에 따라서 thread가 대기하거나 lock을 획득하게 된다.

- **function**
  - **void lock_init (struct lock *);**

    lock을 초기화하는 함수다. lock 객체의 holder을 NULL로 설정하여 lock을 보유하고 있는 thread가 없음을 나타내고, semaphore을 1로 초기화하여 lock이 사용 가능한 상태임을 나타낸다.
  - **void lock_acquire (struct lock *);**

    주어진 lock을 획득하는 함수로 호출된 thread가 해당 lock을 획득하도록 한다. lock_acquire 함수는 주어진 lock을 현재 thread가 획득하도록 시도하며, 획득에 성공할 경우 해당 thread를 lock holder로 나타낸다. thread_current는 실행중인 thread를 리턴하는 함수로, 해당 thread가 lock을 보유하고 있음을 struct의 holer 변수에 저장한다.
  - **bool lock_try_acquire (struct lock *);**

    lock을 획득하려고 시도하는 함수다. 하지만 다른 thread가 이미 lock을 보유하고 있으면 대기하지 않고 실패(success 여부)를 바로 반환한다.
  - **void lock_release (struct lock *);**

    현재 thread가 보유하고 있는 lock을 해제하는 함수이다. 실제로 보유하고 있지 않으면 실행을 중지한다. 해당 thread의 lock의 holder를 NULL로 설정하고, semaphore를 1 증가시켜서 대기중인 다른 thread가 lock을 획득할 기회를 얻는다.
  - **bool lock_held_by_current_thread (const struct lock *);**

    현재 실행 중인 thread가 주어진 lock을 보유하고 있는지 확인하는 함수이다. lock pointer가 NULL이 아닌지 확인하고, 현재 실행중인 thread가 lock의 holder와 일치하는지 True/False로 return한다.
  - **One semaphore in a list**

```
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
  };
```

`struct list_elem` : 다양한 data type의 정보를 갖는 구조체를 linked list에 포함시키기 위해 만들어진 것으로 linked list에 사용되는 기본적인 요소다.

결국 semaphore_elem은 linked list의 한 element로 동작하며, 이 요소는 기본 element, semaphore가 한 쌍으로 한 element를 구성하게 한다.

3) Condition Variable

- meaning

  condition variable은 synchronization의 또 다른 형태로 주로 **thread들이 어떤 조건이 충족될 때까지 대기**하는 데 사용된다. condition variable은 주로 lock과 함께 사용되며 thread는 특정 조건의 발생을 기다리면서 해당 lock을 잠시 풀어주어 다른 thread가 해당 조건을 만족시킬 수 있도록 한다. 즉, condition variable은 한 쪽 코드에서 조건을 신호로 보내게 하고, 다른 cooperating 코드가 해당 신호를 받아 그에 따라 동작한다.

- implementation
  - **struct**

```
/* Condition variable. */
struct condition
  {
    struct list waiters;        /* List of waiting threads. */
  };
```

`waiters` : 대기 중인 thread들의 목록

- 특정 조건에 대해 대기 중인 thread들은 해당 list에 추가되며, 조건이 충족될 때 해당 thread들은 list에서 제거되어 실행을 계속하게 된다.

- **function**
  - **void cond_init (struct condition *);**

    waiters(대기 중인 thread들의 list)를 초기화하여 conditional variable을 초기화하는 함수다.

  - **void cond_wait (struct condition *, struct lock *);**

    주어진 conditional variable에 대해 대기하는 동안 관련 lock을 잠시 해제하고 나중에 다시 획득하는 역할을 한다. 이러한 동작을 통해 다른 thread가 해당 lock을 획득하고 조건 변수를 변경할 수 있게 해준다. 이때 waiter의 semaphore을 0으로 초기화하는데, 이는 thread가 이 semaphore에 대기하면 바로 block될 것임을 의미한다. 이후 waiter을 waiters(대기 list)에 추가하고, cond_signal or cond_broadcast 함수에 의해 신호가 보내질 때까지 현재 thread를 block한다. 이후 lock을 다시 획득하여 결국 이 함수는 lock을 보유한 상태가 된다.

  - **void cond_signal (struct condition *, struct lock *);**

    조건 변수 cond에 대기 중인 thread 중 하나에 신호를 보내어 해당 thread가 대기 상태에서 깨어나도록 해주는 함수다. 하나 이상의 thread가 대기 중이면, cond에 대기 중인 thread 중 하나를 깨우는 역할으로, 특정 조건이 만족될 때 대기 중인 thread를 깨워 해당 조건에 반응하도록 만든다.

  - **void cond_broadcast (struct condition *, struct lock *);**

    cond에 대기 중인 모든 thread를 깨우는 함수이다. waiters list가 비어 있지 않은 동안 반복하여 대기 중인 thread가 있다면 그 thread들을 깨우기 위한 작업을 반복한다. 이는 여러 thread가 특정 조건을 기다리고 있고 그 조건이 만족됐을 때, 모든 thread들에게 동시에 알릴 필요가 있을 때 유용하게 사용된다.

---

2. Analysis of the current thread system

A. Structure

- **thread 관리에 필요한 변수들**
  - thread.h
    - **States in a thread's life cycle**

      thread의 status를 표현하는 4가지 상태에 대한 변수를 `enum thread_status`에 저장한다.

THREAD_RUNNING	CPU에서 현재 실행 중인 상태	thread가 실행 중
THREAD_READY	실행할 준비는 되었지만 실행되고 있지 않은 상태	ready list에 있는 경우이고, scheduler가 호출되어 다음으로 실행될 thread로 선택될 수 있다.
THREAD_BLOCKED	lock, interrupt 등의 trigger을 기다리고 있는 상태	thread_block()이나 Interrupt에 의해 block된 경우 → thread_unblock()함수에 의해 다시 READY로 가지 않으면 스케쥴링 되지 않음

THREAD_DYING	다른 thread로 전환하고 사라질 상태	scheduler에 의해 destroy된 경우이고, schedule 함수에서 thread_schedule_tail에서 후처리 작업을 할 때 dying 상태로 전환된다.

- **Thread identifier type**

`typedef int tid_t;` Thread Identifier의 약자로, 각 thread의 식별 번호인 tid 값을 저장한다.

- **Thread priorities**

priority는 0부터 63 사이의 정수값을 가진다. 숫자가 클수록 우선순위가 높으며, default 값은 31이다.

```
#define PRI_MIN 0                       /* Lowest priority. */
#define PRI_DEFAULT 31                  /* Default priority. */
#define PRI_MAX 63                      /* Highest priority. */
```

- thread.c
  - `static struct list ready_list;` THREAD_READY state에 있는 process들의 리스트이다. 실행할 준비가 되었지만, 실제로는 **실행되고 있지 않은 상태**를 말한다.
  - `static struct list all_list;` 모든 process들의 리스트이다. 처음 schedule 될 때 리스트에 추가되고, exit될 때 삭제된다.
  - `static struct thread *idle_thread;` idle thread이고, ready list가 비었을 때 실행된다. system에 다른 실행할 준비가 된 thread가 없을 때 실행되며 CPU가 할 일이 없을때 idle_thread가 실행되어 CPU의 시간을 소모한다. idle_thread는 OS에 의해 가장 낮은 우선순위로 설정되며 다른 모든 thread가 실행되지 않을 때만 실행된다.
  - `static struct thread *initial_thread;` init.c의 main()에서 실행되고, initial thread이다.
  - `static struct lock tid_lock;` new thread의 tid를 allocate하기 위하여 lock을 사용한다. 동일한 tid가 2번 이상 할당되는 것을 방지하기 위해 lock이 사용된다.
    1. thread가 새 tid를 할당받고자 할 때 allocate_tid() 함수를 호출한다.
    2. 함수 내에서 먼저 tid_lock을 획득한다.
    3. lock을 획득한 후 안전하게 새로운 tid를 할당 받는다.
    4. tid 할당 후 tid_lock을 release한다.
  - `struct kernel_thread_frame` : kernel_thread()을 위한 Stack frame이다.
  - **Timer Ticks** : system의 timer interrupt가 발생하는 주기로 일정 시간 간격으로 발생한다.
    - `static long long idle_ticks;` idle thread를 위한 # of timer ticks 저장
    - `static long long kernel_ticks;` kernel threads를 위한 # of timer ticks 저장
    - `static long long user_ticks;` user programs을 위한 # of timer ticks 저장
  - `#define TIME_SLICE 4` 각 thread에게 4 timer ticks을 부여하고 실행시킨다. 각 thread에 할당된 timer tick 수가 4라는 것을 의미한다. 각 thread는 연속으로 4개의 timer tick동안 실행될 수 있다. 이는 round-robin scheduling algorithm의 일부분으로 각 thread는 일정한 시간(4개의 timer tick)동안 CPU를 점유하고 그 시간이 지나면 다음 thread에게 CPU를 양보한다.
  - `static unsigned thread_ticks;` thread가 마지막으로 CPU를 양보한 이후의 timer tick 수를 추적한다. 매 timer interrupt마다 thread_ticks는 증가하며 thread_ticks 값이 time_slice와 동일하거나 그보다 크게 되면 현재 thread는 CPU를 양보하고 다음 thread가 실행된다. 결국 thread_ticks 변수는 현재 thread가 얼마나 오랜 시간 동안 실행되었는지 추적하는 데 사용되며, 이를 통해 thread가 time_slice보다 더 오래 실행되지 않도록 한다.

- `bool thread_mlfqs;` # **thread scheduler을 정하는 flag이다.**
  - false 값일 때(default) round-robin scheduler를 사용한다.
    - 기본 scheduling 방식으로 각 thread는 일정 시간동안 실행되며 그 시간이 지나면 다른 thread로 전환된다. 이를 통해 모든 thread에게는 균등한 시간이 제공된다.
  - true 값일 때 multi-level feedback queue scheduler를 사용한다.
    - thread의 priority를 고려하여 thread끼리 전환된다.
  - kernel command-line option "-o mlfqs"에 의해 제어된다.

- **Struct thread**

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          // 각 thread 별로 고유한 tid
    enum thread_status status;          // thread state
    char name[16];                      // name
    uint8_t *stack;                     // Saved stack pointer
    int priority;                       // Priority
    struct list_elem allelem;           // List element for all threads list

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              // List element

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  // Page directory
#endif

    /* Owned by thread.c. */
    unsigned magic;                     // Detects stack overflow
  };
```

thread별로 고유의 **tid**를 가져 구별된다. 또한 status에는 해당 thread의 현재 **state**를 저장한다. **name**은 디버깅 시에 사용한다. thread는 고유의 **stack**을 가지는데 이 stack pointer의 값을 stack에 저장한다. 이 값은 switch할 때 사용된다. **allelem, elem**은 list를 관리하기 위해 쓰인다. thread의 list_elem을 통해서 다른 thread와 연결 가능하다.

- allelem: 모든 thread를 연결한 list의 element
- elem: ready_list, semaphore에서 wait하고 있는 list의 element

pintos에서는 이와 같이 **list element를 이용하여 특정 element에 접근할 수 있다.**

pagedir은 page directory의 주소를 저장하고, **magic**은 stack overflow를 찾기 위해 임의의 값으로 설정된다. thread_current function이 저장된 값과 원래 설정된 값을 비교하여 같지 않으면 stack overflow가 발생하였다고 감지하고 assertion을 발생시키게 된다.

- **List.h의 list 관련 변수들**

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };

struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
```

```
  };

#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

list_entry는 elem이 속하는 thread의 주소를 구하는 매크로이다. 이 외에 traversal, insertion, removal, elements, properties 등을 위한 여러 가지 변수들이 정의되어 있다.

B. Functions

- **void thread_init (void);**

  thread system의 초기 설정을 하는 함수로, 현재 실행 중인 code를 thread로 전환하여 thread system을 초기화하며 init.c의 main()에서 호출된다. ready_list, all_list와 tid lock을 초기화시키고 init_thread함수를 호출하여 thread 구조체를 초기화하고 해당 구조체를 initial_thread에 저장한다. initial_thread의 status는 running으로 세팅하고, tid도 새로 할당시킨다.

  - **static void init_thread (struct thread *, const char *name, int priority);**

    thread t를 주어진 이름 name으로, blocked된 thread로 초기화하는 함수다.

- **void thread_start (void);**

  idle_thread를 만들고, interrupt를 enable하여 preemptive scheduling(실행 중인 다른 thread를 중단시키고 cpu를 점유하도록 하는 것)을 시작한다. 이 후 sema down을 호출하는데 idle_started라는 semaphore가 업데이트 될 때까지 현재 thread를 block하고 sema_up이 호출될 때까지 대기한다. 즉 idle_thread가 제대로 시작되고 초기화를 완료하기 전에 이 함수가 종료되지 않도록 한다.

  idle_thread에서 실행되는 function은 **idle()**이고 ready가 된 thread가 있을 때까지 계속 loop를 돌며 대기한다.

  - **static void idle (void *aux UNUSED);**

    CPU가 사용되지 않고, 다른 thread가 모두 실행 준비가 되어 있지 않았을 때 실행되는 특별한 함수이다. 이때 semaphore는 0으로 초기화되어 thread_start()는 대기 상태에 머물게 되는데 idle thread가 시작되면 sema_up을 호출함으로써 semaphore 값을 증가시켜 thread_start()함수가 계속 진행되도록 한다. **결국 idle thread가 완전히 시작되고 준비된 후에만 thread_start() 함수가 다음 작업을 진행하도록 한다.** idle thread는 실제로 아무 작업도 수행하지 않기 때문에 CPU resource를 낭비하게 된다. 따라서 hlt 명령을 사용하여 CPU를 멈추게 하려고 한다. hlt 명령을 실행하기 전에 interrup를 활성화하여 다른 thread가 준비되거나 다른 interrupt가 발생할 때 CPU가 다시 깨어나게 한다.

- **void thread_tick (void);**

  timer tick마다 timer interrupt가 발생되고 timer interrupt handler에 의해 호출되는 함수로, external interrupt context에서 실행된다. 각 thread의 type(idel, user process, kernel)에 따라 tick 수를 업데이트 해준다. TIME_SLICE 마다 preemption을 수행하여서 현재 실행 중인 thread를 waiter로 이동시키고 새로운 thread를 scheduling하므로, 특정 시간 이상 실행된 thread를 yield 시킴으로써 균형을 맞춘다.

- **void thread_print_stats (void);**

  thread의 statistic을 출력하며 pintos가 종료될 때 호출된다.

- **tid_t thread_create (const char *name, int priority, thread_func *function, void *aux)**

  새 kernel thread를 생성하는 함수다. 새 thread를 위한 page를 할당하고 name, priority를 parameter로 받아서 thread 변수를 초기화하고, aux를 function 함수의 인자로 전달한다. kernel_threa, switch_entry, switch_thread를 위한 각 stack frame을 할당하고 thread를 ready queue에 추가하여 대기 상태로 만든다, 생성 완료되면 tid를 return하고, 실패한 경우에는 TID_ERROR를 return한다.

  만약 순서를 보장해야하는 경우에는 semaphore 등의 synchronization의 도구를 사용해야 한다. 위의 주어진 코드는 새로운 thread의 우선 순위를 PRIORITY로 설정하지만, 실제로 priority scheduling은 구현되지 않았으므로, **본 lab에서 구현해야 한다.**

- **void thread_block (void);**

  현재 실행중인 thread의 state를 running에서 blocked로 전환한다. 또한 새로운 thread로 context switching하는 schedule 함수를 호출한다. thread_unblock()함수를 이용해 깨우지 않는 이상 다시 schedule되지 않을 것이다. 또한 이 함수는 synchronization의 영향을 고려하여, interrupt가 꺼진 상태로 호출되어야만 한다.

- **void thread_unblock (struct thread *);**

  blocked된 thread를 다시 ready state로 전환한다. Interrupt 비활성화시킨 후, block 되어 있는 thread를 ready list에 추가하고, ready 상태로 변경해 thread status를 업데이트한다.

- **struct thread *thread_current (void);** 현재 실행중인 thread 주소를 return한다.

- **tid_t thread_tid (void);** 현재 실행중인 thread의 tid를 return한다.

- **const char *thread_name (void);** 현재 실행중인 thread의 name을 return한다.

- **void thread_exit (void) NO_RETURN;**

  thread가 종료될 때 호출되어 thread를 Dying 상태로 변경한다. 그리고 다음 thread로 context switch하는 schedule 함수를 호출한다.

- **void thread_yield (void);**

  현재 실행중이던 thread를 yield하고, 다음에 실행될 thread가 cpu를 점유하고 실행될 수 있도록 한다. 현재 thread는 ready list에 추가하여 ready 상태로 변경되고, block되는 것이 아니라 이후에 다시 schedule 될 수 있다.

- **void thread_foreach (thread_action_func *, void *);**

  all_list의 모든 thread를 순회하면서, parameter인 function을 수행한다. 이때 인자로 받은 aux를 function의 parameter 값으로 사용한다.

- **void thread_set_priority (int);** 현재 thread의 priority를 인자로 받은 new_priority 값으로 바꾼다.

- **int thread_get_priority (void);** 현재 thread의 priority를 return한다.

- **static void kernel_thread (thread_func *, void *aux);**

  kernel_thread 함수는 주어진 인자를 이용하여 function을 실행하는 kernel thread를 시작하고, 해당 함수가 실행 완료되면 thread를 종료한다.

- **static struct thread *running_thread (void);** 이 함수는 현재 실행 중인 thread의 stack pointer을 사용하여 해당 thread의 구조체 주소를 찾아 반환한다.

- **static struct thread *next_thread_to_run (void);**

  CPU에 다음으로 실행될 thread를 결정하는 함수이다. 실행 가능한 thread가 있으면 그 중에서 하나를 선택하여 return 하며 실행 가능한 thread가 없으면 idle_thread를 return한다.

- **static bool is_thread (struct thread *) UNUSED;**

  thread pointer를 인자로 받아, 유효한 thread가 맞는지 확인한다. 이때 thread magic 값을 비교하여 stack overflow 가 일어났는지 확인하여, 유효한지 확인하는 과정이 있다. (Stack overflow가 일어나면 해당 값을 수정하기 때문에 원래 값과 현재 값을 비교하여 확인 가능하다.)

- **static void *alloc_frame (struct thread *, size_t size);**

  여기서 thread→stack은 thread struct의 멤버변수로, stack pointer 값을 저장하는 uint8_t 형 변수이다.

  이때 `ASSERT (size % sizeof (uint32_t) == 0);`에서는 입력한 size가 uint32_t의 배수인지 검증한다. 즉, stack size가 word(32bit) 단위로 정렬되는지 확인하여, stack frame을 할당한다.

- **static void schedule (void);**

  다음에 실행할 thread를 확인하여 context switch시킨다. 이후 thread_schedule_tail() 함수를 이용하여 prev(실행이 중단된 thread)의 후처리 작업을 해준다.

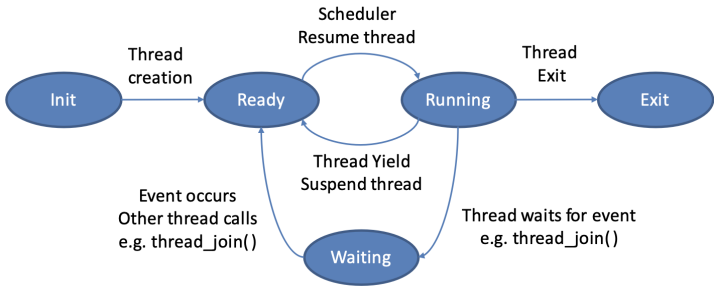
  - **void thread_schedule_tail (struct thread *prev)**

scheduling의 후처리 작업을 수행하는 함수이다. switching이 일어나서 새로 실행되는 thread가 running중이라고 표시하고, 또한 switching이 일어나기 전 원래 thread가 dying state이면 해당 thread의 page를 free시켜서 후처리 작업을 해준다.

- **static tid_t allocate_tid (void);** 새로운 thread가 사용할 tid를 return한다.

C. How to switch threads

thread들은 한 process 안에서 실행되고, CPU 자원을 나누어 점유하게 된다. 이러한 thread의 life cycle을 관리하기 위한 scheduling 작업은 매우 중요하다.



thread가 생성된 후에, ready_list에 삽입되어 **Ready** 상태에 있다가, scheduler 함수가 다음에 CPU를 점유하여 실행될 thread를 선택하여 switch 작업을 한다. **Running** 상태인 thread는 yield되어 ready_list에 삽입되거나, **r**특정 이벤트를 기다려야해서 **Block**되는 경우에는 waiting 상태에 들어가게 된다. 혹은 exit되는 경우도 존재한다. **Block**된 이후 이벤트가 발생하면 다시 **ready** 상태가 되는 방식이다.

따라서 thread가 switch되는 경우는 다음과 같이 정리할 수 있다. 이는 thread를 전환하는 함수인 schedule 함수를 호출하는 경우에 해당한다.

- 현재 scheduling의 round-robin 방식대로, 한 thread가 특정 시간 이상 CPU를 점유할 경우
- 실행하고 있던 thread가 종료되어 yield되어 CPU 사용을 끝마친 경우
- resource를 위해 waiting하거나 exit되는 경우

즉, schedule()을 호출하는 함수는 thread_yield(), thread_block(), thread_exit() 뿐이다. 모두 schedule()을 호출하기 전에 interrupt disable한 상태로 변경한다. switch를 할 때 다른 interrupt가 와서 실행되면 정상 동작할 수 없기 때문이다.

```
static void
schedule (void)
{
  struct thread *cur = running_thread (); // 현재 실행중인 thread 저장
  struct thread *next = next_thread_to_run (); // 다음에 실행될 thread
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF); // 비활성 상태인지 확인
  ASSERT (cur->status != THREAD_RUNNING); // 현재 상태가 Running인지 아닌지
  ASSERT (is_thread (next)); // 다음 실행될 thread가 유효한지

  if (cur != next) // 현재 thread와 다음에 실행될 thread가 다른 경우에 switch한다
    prev = switch_threads (cur, next); // 전환하고, 이전 실행 thread를 return하여 prev에 저장한다
  thread_schedule_tail (prev); // thread의 상태 업데이트하고 마무리한다
}
```

그래서 schedule()에서는 switch를 하기 이전에 ASSERT문으로 interrupt가 비활성 상태인지 확인한다. 이후 thread의 상태를 변경한 뒤, thread_schedule_tail(prev)를 호출하여 새로 실행된 thread의 상태를 running으로 변경하고, dying된 thread의 page를 free시키는 후처리 작업을 해준다.

현재 scheduling 방식의 단점에 대한 분석과 개선 방안의 논의는 3.B Priority Scheduling에서 이어서 하겠다.

---

3. How to achieve each requirement

A. Alarm clock

1) Problem : Current Implementation

본 문제에서는 현재 thread scheduling의 비효율적인 점을 개선한다. thread는 state에 대한 구조체에 정의되어 있듯이, 4가지 상태를 오간다. 이때 하나의 CPU를 여러 개의 thread가 사용해야 하기 때문에, OS는 thread의 자원 배분을 관리해야 한다. 특정 thread가 한 CPU를 점유하는 것을 막기 위해, 특정 시간 조건에 따라서 thread를 yield 시켜 실행되지 않도록 한다. 이러한 시간에 대한 함수들은 timer.c에 구현되어 있다.

```
int64_t
timer_ticks (void)
{
  enum intr_level old_level = intr_disable ();
  int64_t t = ticks;
  intr_set_level (old_level);
  return t;
}

int64_t
timer_elapsed (int64_t then)
{
  return timer_ticks () - then;
}
```

`timer_ticks`는 OS가 부팅된 이후 누적된 카운트를 return하고, `timer_elapsed`는 특정 시점을 인자로 받아, 그 시점부터 흐른 시간을 return한다.

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

`timer_sleep` 함수는 thread가 잠들어 있어야 하는 시간을 의미하는 ticks을 인자로 받아, 해당 thread의 실행을 지연시킨다. `timer_ticks`로 해당 thread가 실행된 시간을 start로 받고, 이 시점부터 얼마나 실행되었는지를 `timer_elapsed`로 확인한다. 그리고 이러한 시간이 ticks보다 작은 특정 기간 동안에는 `thread_yield`를 호출하여 실행되지 않도록 한다. 하지만 이 경우는 busy waiting 방식으로 구현되어 있어서 비효율적이다. while 문의 조건이 true여서 계속해서 반복문을 실행할 때, `thread_yield` 함수가 호출되어 해당 thread는 계속해서 ready list에 `list_push_back`된다. 그리고, 다른 thread가 CPU를 점유하여 실행되도록 한다.

즉, thread가 ticks 이상만큼 잠들어 있는(blocked) 되게 하고, 특정 시점을 확인하여 ready 상태로 만드는 것이 아니라, while 문 안에서 조건을 계속해서 확인함으로써 불필요한 반복 작업을 하는 것이다.

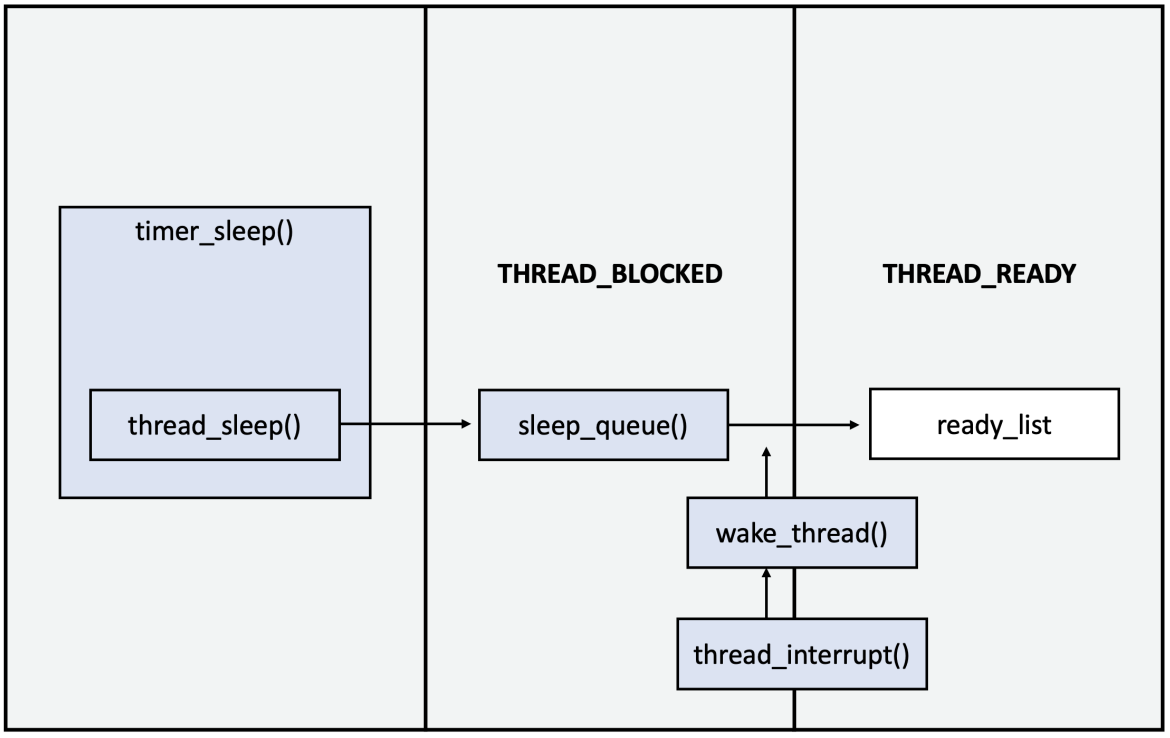
2) Our Design & Implementation

busy waiting의 대안으로 Sleep/Wake up 기반의 alarm clock을 구현하여 해결하였다. 기존과 달리 thread를 yield하여 ready_list에서 추가하는 것이 아니라, sleep list 추가한다. 이 후 sleep list에 있는 thread들 중에서 깨어날 시간이 다 된 thread는 sleep list에서 삭제하고 ready_list에 추가하였다.

즉, thread는 4가지 state에서 전환이 되고 현재의 문제는 thread가 sleep 될 때 ready list에 남아 반복 실행되는 것이다. 이를 ready list가 아니라 sleep list에 추가함으로써 blocked 상태로 만들어서 해결했다.

sleep list를 정의함으로써 추가적으로 수정해야할 함수들이 있는데, 이때 sleep list를 struct list형 변수로 정의하여, list.c에 정의되어 있는 basic operation들을 사용하였다.

이때 sleep list에 추가된 thread들에서 tick을 확인하고, 이후에 ready list으로 이동시키기 위해서는 tick값의 비교가 필요하다. sleep list에서 tick을 기준으로 **sorting**하면 효율적이라고 생각했고, list.c에 정의되어 있는 **list_insert_ordered** 함수를 이용했다.



전체적인 함수 동작과, 이때의 thread의 status는 위 그림과 같다.

- **Rationales**

위 알고리즘에서는 잠들어야하는 thread들을 blocked 상태로 전환하여 sleep list에 추가한다. 그리고 이때 정렬하는 함수를 사용하여 sleep list에 추가될 때에 sort 되도록 한다. 이로써 시간 복잡도를 줄일 수 있다. 또한, wake_thread 함수에서 thread를 깨울 때, sorting 되어있는 sleep list를 순차적으로 순회하면서 조건을 확인할 수 있다. 이때 오름차순으로 sorting 되어있기 때문에, wakeup_tick이 주어진 tick보다 더 큰 시점에서는 break 가능하다. 왜냐하면 해당 시점 이후에 sleep list에 저장되어 있는 thread의 wakeup_tick은 오름차순 정렬에 따라 더 큰 값을 가지고, 더 기다려야하는 thread임이 보장되기 때문이다. 이로써 불필요하게 모든 thread에 대해 wake 조건을 확인할 필요가 없어진다.

최종 설계와 관련하여 추가하거나 수정한 function, data stucture는 다음과 같다.

```
static struct list sleep_list;
```

`sleep_list` : global variable로 static struct list 형의 자료구조를 만들어 sleep된 thread를 저장하게 한다. 즉, `timer_sleep()`을 호출한 thread를 저장하고, 이 thread들은 blocked 상태이다.

```
struct thread
  {
    ...
    /* modified for lab1_1 */
    int64_t wakeup_tick;
    ...
  };
```

`wakeup_tick` : 각 thread가 깨어나야 할 tick을 저장할 변수이다. 각 thread에 대한 것이므로 thread struct의 member variable로 추가한다. 따라서 threads/thread.h 함수의 struct thread 안에 추가하면 된다. int64_t형 변수로 설정한다.

```
void
thread_init (void)
{
  ...
  /* modified for p1 */
  list_init (&sleep_list);
  ...
}
```

`thread_init()` : thread를 초기화하는 함수이므로, sleep list를 초기화하는 코드를 추가한다. list 변수를 사용할 것이므로 list_init함수를 사용한다.

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();
  /* modified for lab1_1 */
  thread_sleep(start+ticks);
}
```

`timer_sleep()` : 기존에는 while 문 안에서 계속하여 thread_yield하는 busy waiting하며 반복 실행하였는데, 이부분을 삭제했다. 그리고 `thread_sleep()` 함수를 호출하였다.

```
void thread_sleep(int64_t ticks)
{
  struct thread *cur = thread_current ();

  enum intr_level old_level;
  old_level = intr_disable ();

  ASSERT (cur != idle_thread)

  cur->wakeup_tick = ticks;
  list_insert_ordered(&sleep_list, &cur->elem, cmp_wakeup_tick, NULL);
  thread_block();

  intr_set_level (old_level);
}
```

`void thread_sleep(int64_t ticks);` : 실행 중인 thread를 sleep으로 만드는 함수이다. 이때 interrupt에 의해 방해받지 않도록 thread를 blocked 상태로 만들어 작업을 처리한다. idle thread인 경우 ASSERT가 발생하게 하여 확인한다. 인자로 넘겨받은 ticks을 현재 thread의 wakeup_tick으로 설정한다. 그리고 sleep list에 wakeup tick에 따라 정렬하여 삽입한다. 이

때 list.c에 정의되어 있는 list_insert_ordered 함수를 사용한다. 그리고 이때 wakeup_tick을 비교하는 `cmp_wakeup_tick` 함수를 호출한다.

```
bool cmp_wakeup_tick(const struct list_elem *prev, const struct list_elem *next, void *aux UNUSED)
{
  struct thread* prev_thread = list_entry(prev, struct thread, elem);
  struct thread* next_thread = list_entry(next, struct thread, elem);

  return ((prev_thread->wakeup_tick)<(next_thread->wakeup_tick));
}
```

`bool cmp_wakeup_tick(const struct list_elem *prev, const struct list_elem *next, void *aux UNUSED);` : 이 함수는 두 개의 thread를 받아 각각의 wakeup_tick 값을 비교하여 두 번째가 더 크면 true를 리턴하는 함수이다. 이렇게 sleep_list를 오름차순으로 정렬할 수 있다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  /* modified for lab1_1 */
  wake_thread(ticks);
}
```

`timer_interrupt()` : 매 tick마다 timer interrupt 시에 호출되는 함수이다. `wake_thread()` 함수를 호출하여 sleep list에서 깨어날 thread가 있는지 확인하고, 깨우는 기능을 수행한다.

```
void wake_thread(int64_t ticks)
{
  struct list_elem* iter;
  for (iter = list_begin(&sleep_list); iter != list_end(&sleep_list);)
  {
    struct thread* cur = list_entry(iter, struct thread, elem);
    if (cur->wakeup_tick > ticks) break;
    iter = list_remove(iter);
    thread_unblock(cur);
  }
}
```

`void wake_thread(int64_t ticks);` : sleep list에서 깨워야 할 thread를 깨우는 함수이다. 따라서 sleep list를 순회하면서 조건을 확인하는 코드를 구현한다. 이 함수에서 인자로 받는 ticks 값은 시스템이 부팅된 이후 지난 시간을 의미한다. 만약 현재 thread의 wakeup_tick값이 인자로 받은 ticks보다 크면 아직 더 기다려야하므로 깨우지 않고 break한다. 이외의 경우에는 깨워야 하기 때문에, list에서 해당 요소를 삭제하고, unblock함으로써 blocked에서 ready 상태로 전환한다.

B. Priority Scheduling

1) Problem : Current Implementation

1-1. FIFO

현재의 scheduling 방식은 thread의 우선순위를 고려하지 않고, round-robin 방식으로 생성 순서에 따라서 작동하기 때문에 비효율적이다.

- **Scheduling**
  - thread creation이 일어나면 ready_list의 마지막에 push되며, 다음 실행할 next_thread_to_run 함수에서는 맨 앞에서 pop 시킨다. 따라서 scheduling이 일어나면, FIFO 방식으로 먼저 생성되어 list에 push된 thread가 먼저 실행되게 된다.

- **Synchronization**
  - semaphore

    waiter_list에서 thread들이 공유자원을 기다리는데, 이때 기다리는 상황에서는 waiter_list에 thread 순서대로 삽입되고, 이후에 sema_up이 되어 공유 자원 사용이 가능해진 경우 waiter_list의 맨 앞 thread를 unblock시킨다.
  - conditional variable

    thread들이 어떤 조건이 충족될 때까지 대기하는 데 사용되는데, 이때 대기 중인 thread를 waiters list로 관리한다. 이때 대기할 때는 waiters list의 맨 마지막에 push_back하고, signal이 도착하여 깨우는 경우에는 front에 있는 것을 pop한다.

즉, scheduling과 synchronization에서 현재의 pintos code는 thread을 scheduling하거나, critical resource를 사용하기 위해 접근하는 순서에 따라 FIFO로 실행된다. 즉 thread 간의 우선순위를 전혀 고려하지 않는 동작 방식이다. 이러한 문제를 제시된 priority scheduling으로 해결하였다.

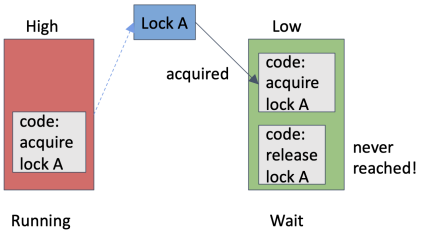
1-2. Priority inversion problem

또한, priority scheduling 과정에서 priority inversion problem이 발생할 수 있다. 해당 문제에 앞서 priority 관련 개념을 정리하고자 한다.

- **Priority**
  - Priority란?
    - 각 thread마다 부여되는 것으로 PRI_MIN(=0) 부터 PRI_MAX(=63)까지의 범위를 가지며 **숫자가 클수록 priority가 높다.**
    - thread_create() 함수로 thread를 생성할 때 인자로 초기 우선 순위를 전달한다.(기본 값으로 PRI_DEFAULT(=31))
  - 생성된 thread의 우선순위는 다음 함수를 통해 변경 가능하다.
    - void thread_set_priority (int new_priority)
      - 현재 thread의 우선순위를 new_priority 로 변경
    - int thread_get_priority (void)
      - 현재 thread의 우선순위를 반환
  - Rules of priority scheduling

> 1. ready list에 있는 것 중 가장 높은 우선순위의 thread를 먼저 execute 2. thread가 waiting하고 있을때, 가장 높은 우선순위의 thread를 먼저 unblock 3. 현재 running중인 thread보다 높은 우선순위의 thread가 ready list에 추가된 경우에, 현재 thread는 즉시 yield 4. thread는 언제든지 자신의 우선순위를 올리거나/낮출 수 있는데, 낮추어서 더 이상 highest priority가 아니게 된 경우에는, 즉시 yield

- **Priority inversion problem**
  - priority inversion이란?
    - 위의 4가지 rule을 따른 priority scheduling을 따랐을 때 priority가 높은 thread가 priority가 낮은 thread를 wait하는 현상을 말한다.
    - 아래 예시와 같이 낮은 우선순위의 thread가 lock을 acquire한 상태에서 높은 우선순위의 thread가 실행된다고 가정하자. priority scheduling 규칙에 따라 낮은 우선순위의 thread가 CPU를 점유할 수 없기 때문에 H는 L이 lock을 release하기까지 무한정 대기하게 된다. 따라서 H가 실행되더라도 lock을 L이 가지고 있기 때문에 Blocked 된다.



- 추가로, 이렇게 H가 L의 실행 완료를 대기하는 상태에서, H와 L 사이의 Priority를 가진 thread M(lock 요청하지 않는)이 ready list에 추가될 경우, CPU는 L 대신에 priority가 더 높은 M을 실행하게 되므로, M의 실행이 완료된 후에서야 L을 실행할 수 있게 되고, 그 후에서야 lock을 얻은 H가 실행될 수 있다. 결과적으로 M → L → H 순으로 동작하고, H가 M보다 우선순위가 높지만 M이 더 우선적으로 실행되게 되는 문제가 발생한다.

2) Our Design & Implementation

최종 설계와 관련하여 추가하거나 수정한 function, data stucture는 다음과 같다.

Problem1. Priority scheduling

먼저 기존의 FIFO 방식의 문제점을 해결하고자 Priority를 반영한 방식으로 구현했다.

1. **Scheduling**

   1) list insert 동작 수정

   먼저, scheduling의 경우 ready_list을 관리하는 게 중요하기 때문에, list에 insert하는 부분을 수정하고자 한다. ready_list에 insert하는 경우는 thread를 ready 상태로 만드는 경우에 해당하므로 **thread_unblock과 thread_yield** 함수에서 ready_list에 insert한다.

   - 기존 : `thread_unblock()`과 `thread_yield()` 함수에서 ready_insert하는 부분을 살펴보면 **list_push_back** 함수를 사용하여 list의 맨 뒤에 넣는 과정을 거쳤다.
   - 수정 : 해당 부분을 지우고 **list_insert_ordered**를 호출하여 list에 넣을 때마다 priority를 기준으로 sorting하여 넣는다.

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  /* modified for lab1_2 */
  //list_push_back (&ready_list, &t->elem);
  list_insert_ordered(&ready_list, &t->elem, cmp_priority, NULL);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    /* modified for lab1_2 */
```

```
    //list_push_back (&ready_list, &cur->elem);
    list_insert_ordered(&ready_list, &cur->elem, cmp_priority, NULL);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

여기서 두 element를 비교할 함수가 필요하므로 `cmp_priority`라는 함수를 정의한다. 두 thread의 priority 간의 크기를 비교해서 첫번째가 높으면 1 두번째가 높으면 0을 return한다.

```
bool cmp_priority(const struct list_elem *t1,
 const struct list_elem *t2,
void *aux UNUSED)
{
  return list_entry(t1, struct thread, elem)->priority
> list_entry(t2, struct thread, elem)->priority;
}
```

2) priority를 고려하는 동작 수정

다음으로는 scheduling을 priority에 따라 하게 되면 priority가 높은 thread가 항상 먼저 수행되어야 하는 규칙이 보장되어야 한다. 그렇기 때문에 priority가 업데이트되는 것을 고려하여 현재 thread와 새로운 thread 간의 priority를 비교하는 동작이 필요하다.

먼저 thread들의 priority를 다시 고려해야할 경우는 두가지가 있다. thread를 새로 만들거나(`thread_create()`) thread의 priority를 새로 업데이트할 때(`thread_set_priority()`)이다.

- 기존 : `thread_set_priority()` 함수는 단순히 thread_current의 priority를 인자로 받은 new_priority로 업데이트한다. `thread_create()`함수는 새로운 thread를 만든다.
- 수정 : `thread_set_priority()`에서 thread_current의 priority를 new_priority로 업데이트한 후와 `thread_create()`에서 새로운 thread를 다 만든 후에 `check_priority_and_yield()`라는 함수를 호출한다.

```
void
thread_set_priority (int new_priority)
{
   thread_current ()->priority = new_priority;

  // modified for donation
  struct thread *cur = thread_current();
  /* modified for lab1_3 */
  if (!thread_mlfqs)
  {
    thread_current ()->original_priority = new_priority;
    cur->priority = cur->original_priority;
    // 3. update current thread's priority according to donation list
    if(!list_empty(&cur->donation_list))
    {
      list_sort(&cur->donation_list, cmp_priority, NULL);
      int max_priority = (list_entry(list_begin(&cur->donation_list), struct thread, donation_elem))->priority;
      cur->priority = cur->priority < max_priority ? max_priority : cur->priority;
    }
  }

  /* modified for lab1_2 */
  //compare current thread's priority with new thread's priority
  check_priority_and_yield();

}
```

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
```

```
{
  ...
  /* Add to run queue. */
  thread_unblock (t);
  /* modified for lab1_2 */
  //compare current thread's priority with new thread's priority
  check_priority_and_yield();

  return tid;
}
```

`check_priority_and_yield()` : 현재 실행되는 thread의 priority가 ready_list에 있는 thread들의 priority보다 큰지를 체크하고, 더 큰 priority를 가지는 thread가 ready_list에 있으면 현재 thread를 yield하는 함수다. ready_list에서 priority가 가장 큰 thread는 list_insert_ordered로 삽입했기 때문에 가장 앞에 있을 것이다. 그래서 list_front함수를 사용하여 ready_list에서 가장 priority가 큰 thread의 priority를 `max_priority`로 저장한다. 그리고 현재 수행되는 thread의 priority가 `max_priority`보다 작은 경우 더이상 수행되면 안되므로 `thread_yield()`함수를 호출하여 yield하게 했다.

```
void check_priority_and_yield(void)
{
  int max_priority;
  if(!list_empty(&ready_list))
  {
    max_priority = list_entry(list_front(&ready_list),struct thread, elem)->priority;
    if(thread_get_priority() < max_priority)
    {
      thread_yield();
    }
  }
}
```

2. **Synchronization**

   synchronization mechanism도 priority를 고려하도록 수정해야 한다. thread들 간에 lock, semaphore, conditional variable을 얻으려고 할 때 이들 간에 priority를 고려해야 한다.

   기존 : semaphore을 기다리고 있는 thread들이 있는 waiter list는 FIFO 방식으로 구현되어 있다.

   수정 : waiter list를 priority를 반영하여 관리하도록 하기 위해 list insert 시 `list_insert_ordered` 을 호출하여 insert되도록 하고, waiter list에서 pop하기 전에 waiter list를 sorting하는 부분을 추가한다.

   `sema_down()` : semaphore을 기다리기 위해 thread가 waiters list에 들어갈 시 기존에는 list_push_back을 사용했다면 `list_insert_ordered`를 사용하는 코드로 수정한다. 이 때 마찬가지로 thread들 간의 priority를 비교해야 하므로 위에서 사용한 `cmp_priority` 함수를 사용한다.

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      /* modified for lab1_2 */
      //list_push_back (&sema->waiters, &thread_current ()->elem);
      list_insert_ordered(&sema->waiters,
    &thread_current()->elem, cmp_priority, NULL);
      thread_block ();
    }
```

```
  sema->value--;
  intr_set_level (old_level);
}
```

`sema_up()` : semaphore을 해제하기 전에 semaphore을 기다리는 waiter들 중에서 priority가 가장 큰 thread를 `list_pop_front`를 통해 꺼낸후 unblock을 해주는 과정이 존재한다. 하지만 waiter list에 있는 thread들이 우선순위가 변경되는 경우가 존재하기에 `list_pop_front`를 해주기 전에 waiter list를 list_sort를 사용하여 정렬한다.

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
  {
    /* modified for lab1_2 */
    list_sort(&sema->waiters, cmp_priority, NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;
  check_priority_and_yield();
  intr_set_level (old_level);
}
```

`cond_wait()` : `sema_down()`함수와 마찬가지로 conditional variable의 waiter list에 priority 기준으로 삽입되도록 `list_insert_ordered`를 사용하는 코드로 수정한다.

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  /* modified for lab1_2 */
  //list_push_back (&cond->waiters, &waiter.elem);
  list_insert_ordered(&cond->waiters, &waiter.elem,
cmp_sema_priority, NULL);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

- 이 때, conditional variable의 waiters는 위의 semaphore의 waiters와 다르다. conditional variable waiters는 같은 semaphore을 기다리는 thread들이 아니며, 같은 condition을 기다리고 있는 semaphore들이라고 할 수 있다. 그렇기 때문에 비교를 할 때 각 semaphore들을 비교하는 셈이며, 각 semaphore의 waiter list에서 가장 큰 priority끼리 비교해야 한다. 이를 위해 `cmp_sema_priority`라는 새로운 함수를 사용하고자 한다.

`cmp_sema_priority` 는 두개의 semaphore의 waiter list에서 가장 앞 thread들을 가져오고 두 thread들 간의 priority를 비교하여 결과를 반환한다.

```
/* modified for lab1_2 */
bool cmp_sema_priority (const struct list_elem *s1,
 const struct list_elem *s2, void *aux UNUSED)
{
  struct semaphore_elem *sema1 = list_entry(s1,
```

```
struct semaphore_elem, elem);
  struct semaphore_elem *sema2 = list_entry(s2,
struct semaphore_elem, elem);

  struct list_elem *max_thread_sema1 = list_begin(
&(sema1->semaphore.waiters));
  struct list_elem *max_thread_sema2 = list_begin(
&(sema2->semaphore.waiters));

  int sema1_priority = list_entry(max_thread_sema1,
struct thread, elem)->priority;
  int sema2_priority = list_entry(max_thread_sema2,
struct thread, elem)->priority;

  return sema1_priority > sema2_priority;
}
```

`cond_signal()` : waiters에서 가장 앞에 있는 semaphore을 sema_up하는 함수이고, waiters에 있는 semaphore들의 waiter list에서 thread들의 priority가 변경되는 경우를 고려하여 sorting하는 코드를 추가한다.

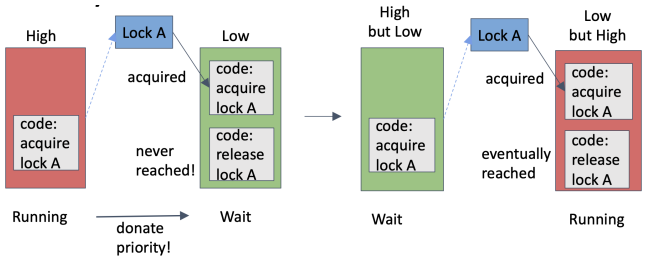
```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
  {
    /* modified for lab1_2 */
    list_sort(&cond->waiters, cmp_sema_priority, NULL);
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                struct semaphore_elem, elem)->semaphore);
  }
}
```

Problem2. Priority Inversion Problem

우리는 priority inversion problem을 priority donation을 통해 해결하고자 한다.

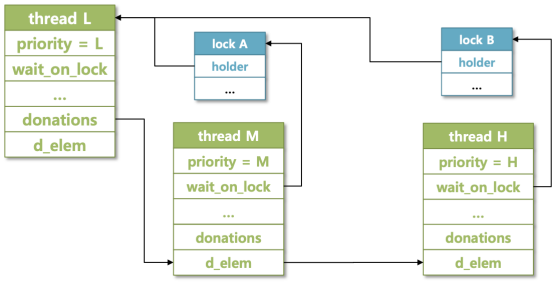
- priority donation이란?



더 높은 우선순위 thread H가 낮은 우선순위 thread L에게 priority를 “Donate”하여 priority inversion problem을 해결할 수 있다. blocked되어 있는 thread H의 priority가 lock을 가지고 있는 thread L의 priority보다 높을 때, H의 priority를 L에게 donation하여 임시적으로 두 thread H, L이 같은 priority를 갖도록 하는 방법이다. 그리고 실행 후 복구한다.

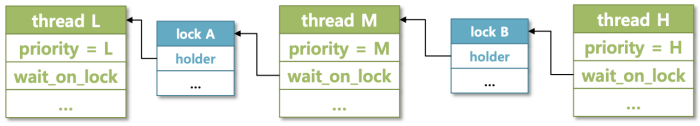
- priority donation 구현 과정에서 고려해야하는 것이 2가지가 있다.

  1) multiple donation



한 thread에 대해 여러 개의 priority가 donate될 때 발생하는 현상이다. 두 개 이상의 lock 보유시, 각 lock에 의해 donation 발생 가능하기 때문에, donation 받기 이전 상태의 우선순위를 기억하고 복구해야 한다.

2) nested donation



H가 M이 점유하고 있는 Lock B를 얻기 위해 대기하고, L이 점유하고 있는 Lock A를 얻기 위해 대기하고 있는 상황에서 발생한다. 실행을 위해서 H의 priority는 L, M 모두에게 donation 되어야 한다. 이때 nested limit은 최대 8 level이다.

- priority donation implementation

1) 자료 구조 선언 및 초기화

먼저 priority donation과 관련된 구조체를 선언하고 초기화한다.

`waiting_lock` : current thread가 acquire하고자 하는 lock을 저장할 수 있도록 struct thread에 새로운 필드를 만들어 준 것이 waiting_lock이다. 해당 thread가 대기하고 있는 lock의 주소를 저장한다.

`original_priority` : donation 이후 원래 가지고 있던 우선순위 값으로 초기화하기 위해 원래 우선순위 값을 저장

`donation_list` : multiple donation을 고려하여 현재 thread에 donation한 thread들을 관리하는 list

`donation_elem` : donation_list의 elem

```
/* modified for lab1_2 */
int original_priority; // priority before donation
struct lock *waiting_lock; // lock that thread is waiting for

// List of higher priority threads that donated to the thread
struct list donation_list; // for multiple donation
struct list_elem donation_elem;
```

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
 ...
  /* modified for lab1_2 */
  t->original_priority = priority;
  t->waiting_lock = NULL;
  list_init(&t->donation_list);

  /* modified for lab1_3 */
  t->nice = 0;
  t->recent_cpu = 0;

  old_level = intr_disable ();
  list_push_back (&all_list, &t->allelem);
```

```
  intr_set_level (old_level);
}
```

2) lock을 acquire하는 부분 수정

priority donation이 일어나야 하는 경우는 한 가지이다. lock을 acquire할 때 priority를 check하여 lock의 holder가 현재 acquire하고자 하는 thread보다 priority가 낮으면 donation을 해주면 된다. 또한 donation을 고려하여 동작을 수정해야 하는 부분은 lock을 release하는 부분이다. lock을 acquire하는 과정에서 donation이 발생했으나 lock을 release하게 되면 더이상 lock holder가 아니므로 원래 thread의 priority로 돌아가야 한다. 그러므로 **lock을 acquire하는 부분, lock을 release하는 부분**을 수정하였다.

`lock_acquire()` : 현재 lock을 acquire하는 부분은 sema_down을 호출하고 이를 성공하면 lock→holder에 현재 thread를 넣는 동작을 수행한다. 하지만 이제 priority donation을 적용해야 하므로 sema_down을 호출하기 전에 lock_holder가 있다면 priority를 donation하는 부분을 추가한다. 그리고 해당 부분을 `donate_priority()`라는 함수로 구현한다. 그리고 `donate_priority()` 함수 호출 전에 current thread의 `waiting_lock`을 현재 acquire하고자 하는 lock으로 업데이트해주고, list_insert_ordered를 통해 lock holder의 `donation_list`에 current thread를 priority 기준으로 sorting하며 추가한다.

```
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  // when lock holder exists
  // 1. update waiting lock
  // 2. Add the current thread to the donation list of the lock
  // 3. call donate_priority() for priority donation
  /* modified for lab1_3*/
  if (!thread_mlfqs)
  {
    if (lock->holder)
    {
      thread_current()->waiting_lock = lock;
      list_insert_ordered(&lock->holder->donation_list,
&thread_current()->donation_elem, cmp_priority, NULL);
      donate_priority();
    }
  }
  // try to acquire lock
  sema_down (&lock->semaphore);
  if (!thread_mlfqs)
  {
    thread_current()->waiting_lock = NULL;
  }
  lock->holder = thread_current ();
}
```

`donate_priority()` : 해당 함수는 nested donation을 고려하여 재귀적으로 구현했다. level limit은 8이므로 총 8번을 반복하는데, 다음 반복 시의 current thread는 현재 current thread가 acquire하고자 하는 lock의 holder가 되는 구조로 재귀적으로 구현한다. 또한 current thread가 acquire하고자 하는 lock이 없는 상태면 재귀를 종료한다. donation 과정은 현재 thread가 acquire하고자하는 lock의 holder priority와 비교하여 현재 thread가 priority가 더 크면 donation을 해주는 방식이다.

```
void donate_priority(void)
{
  struct thread* cur = thread_current();
  int level;
  struct thread *holder;
  for (level = 0; level<8 ;level++)
  {
    if(cur->waiting_lock != NULL)
    {
```

```
      holder = cur->waiting_lock->holder;
      if(cur->priority > holder->priority) // donation
      {
        holder->priority = cur->priority;
      }
      cur = holder;
    }
    else
    {
      break;
    }
  }
}
```

`donate_priority()` 이후 `sema_down`이 완료되고 lock을 acquire에 성공하면 더 이상 `waiting_lock` 필드에 lock을 저장할 필요가 없으므로 NULL로 업데이트해준다.

2) lock을 release하는 부분 수정

기존에 lock을 release하는 부분은 lock→holder을 NULL로 만들고 sema_up을 호출하는 동작을 수행한다. 그러나 holder을 NULL로 만들기 전에 holder의 priority에 대한 동작 수정이 필요하다. holder의 priority를 다시 수정하는 과정을 거칠 때 필요한 동작을 정리해보면 holder의 priority를 원래 priority로 복원하고 holder에게 donation해준 thread들의 priority를 다시 고려하여 수정해줘야 한다. holder는 여러 lock을 가지고 있을 수 있고 현재 release하는 lock외에 다른 lock을 가지고 있는 경우 해당 lock을 기다리는 thread들이 현재 thread에 donation해주었을 가능성이 존재한다. 그러므로 초기 priority로 업데이트 후 현재 thread에 donation한 thread들의 priority들 중 가장 큰 priority로 다시 업데이트하는 과정이 필요하다. `lock_acquire()`함수에서 donation 전에 lock→holder의 donation_list에 현재 thread를 `list_insert_ordered`를 통해 insert하는 부분을 추가했기 때문에 각 thread들을 기준으로 donation한 thread들이 donation_list에 priority 순으로 존재함이 보장되어 있다. 이를 바탕으로 `lock_release()` 함수를 수정하였다.

`lock_release()` : 먼저 lock holder에게 donation한 thread들을 donation list에서 지우고, lock holder의 priority를 수정하였다. 먼저 lock holder에게 donation한 thread들을 donation list에서 지우는 것은 `delete_donation_list(lock)`을 호출하여 수행한다. 그 다음, lock holder의 priority를 수정하는 동작은 먼저 lock holder의 priority를 `original_priority`로 수정하고, donation_list를 다시 sorting하고 donation_list에서 가장 큰 priority를 가져와 이와 비교해 lock holder의 priority를 재수정한다.

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  /* modified for donation in lab1_3 */
  if (!thread_mlfqs)
  {
    /* modified for donation in lab1_2 */
    // 1. update donation list
    delete_donation_list(lock);
    // 2. update current thread's priority to original
    struct thread *cur = lock->holder; // lock holder is current thread
    cur->priority = cur->original_priority;
    // 3. update current thread's priority according to donation list
    if(!list_empty(&cur->donation_list))
    {
      list_sort(&cur->donation_list, cmp_priority, NULL);
      int max_priority = (list_entry(list_begin(&cur->donation_list), struct thread, donation_elem))->priority;
      cur->priority = cur->priority < max_priority ? max_priority : cur->priority;
    }
  }
  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

`delete_donation_list(lock)` : donation list를 처음부터 끝까지 돌면 donation한 thread가 acquire하고자 하는 lock과 현재 lock이 일치하면 해당 thread를 donation list에서 지운다.

```
void delete_donation_list(struct lock* lock) // update donation list
{
  struct list *list_of_donation = &thread_current()->donation_list;
  struct list_elem *elem;
  struct thread* donation_thread;

  for (elem = list_begin(list_of_donation); elem != list_end(list_of_donation); elem = list_next(elem))
  {
    donation_thread = list_entry(elem, struct thread, donation_elem);
    if (donation_thread->waiting_lock == lock)
    {
      list_remove(elem);
    }
  }
}
```

3) thread_set_priority 수정

`thread_set_priority()` : 원래 현재 수행중인 thread의 priority를 new_priority로 수정하는 동작을 하는데, priority를 수정하는 곳이므로 priority donation을 고려해줄 필요가 있다. `original_priority`도 new_priority로 업데이트하도록 추가했고, new_priority로 업데이트했지만 현재 thread의 donation list를 확인하여 가장 큰 priority보다 new_priority가 작으면 이로 업데이트하도록 수정했다.

```
void
thread_set_priority (int new_priority)
{
   thread_current ()->priority = new_priority;

  // modified for donation
  struct thread *cur = thread_current();
  /* modified for lab1_3 */
  if (!thread_mlfqs)
  {
    thread_current ()->original_priority = new_priority;
    cur->priority = cur->original_priority;
    // 3. update current thread's priority according to donation list
    if(!list_empty(&cur->donation_list))
    {
      list_sort(&cur->donation_list, cmp_priority, NULL);
      int max_priority = (list_entry(list_begin(&cur->donation_list), struct thread, donation_elem))->priority;
      cur->priority = cur->priority < max_priority ? max_priority : cur->priority;
    }
  }

  /* modified for lab1_2 */
  //compare current thread's priority with new thread's priority
  check_priority_and_yield();

}
```

- **Rationales**

1. priority scheduling

   우리가 구현하고자 하는 방법은 priority scheudling을 정상적으로 동작하게 한다. ready_list에서 pop front를 하는 방식으로 thread를 꺼내게 되는데 우리는 ready_list에 insert할 때 priority를 고려하여 sorting되며 삽입되게 했기 때문에 pop front하여 꺼낸 thread가 ready_list에서 가장 큰 priority를 가진 thread라는 것이 보장된다. 또한, thread의 priority가 새로 업데이트되는 부분에서도 현재 thread와 ready_list의 가장 큰 priority를 가진 thread를 비교함으로써 항상 priority가 가장 큰 thread가 수행되는 것을 보장한다.

2. priority donation

우리가 구현하고자 하는 방법은 priority donation도 정상적으로 동작하게 한다. donation list 자료 구조를 통해 multiple donation이 정상 작동하며 donation하는 동작도 재귀로 구현하여 nested donation도 정상 작동하게 한다. 또한, lock을 release할 때 lock의 holder의 priority를 재계산하는 과정을 통해 항상 lock을 waiting하고 있는 thread들만 holder에 donation하도록 보장한다.

C. Advanced Scheduler

1) Problem : Current Implementation

현재 구현되어 있는 scheduler는 3.B에서 priority를 고려한 scheduling 방식이다. 하지만 낮은 priority를 가지는 thread는 CPU를 점유하기 어려워 오랫동안 CPU에서 실행되지 못할 수 있다. 이는 thread 실행 시간의 bottleneck이 될 것이고, project.1 문서에 제시된 요구 조건에 따라 advanced scheduler를 구현한다.

Multi-Level Feedback Queue Scheduler(MLFQS)란 각 priority 별로 ready queue를 두어서 priority를 자동적으로 관리하는 것을 의미한다. 0~63의 priority의 64개에 대해 queue로 manage하고, 이러한 priority를 4 ticks 주기로 갱신한다. MLFQS를 설계하기 위해 필요한 개념은 다음과 같다.

- **Priority**

```
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
```

priority는 thread 마다 설정되는 것을 말하며, 0(PRI_MIN)부터 63(PRI_MAX)까지의 정수 값을 가진다. MLFQS에서는 위 공식을 적용하여 매 4번째 tick마다 priority를 재계산한다.

위 공식에 쓰이는 개념을 정리하고자 한다.

- **Nice**

다른 thread에게 CPU를 yield하는 정도를 -20부터 20 사이의 정수로 나타낸 값이다. Nice가 클수록(positive) 다른 thread에게 잘 yield한다. nice가 양수일 경우 다른 thread에게 잘 양보하기 때문에 해당 thread는 priority가 낮은 값으로 설정되어 나중에 실행된다. 또한 nice가 음수이면, priority가 커져서 높은 우선순위로 실행되는 것이다.

- **Recent_cpu**

최근에 해당 thread가 CPU를 점유한 시간을 float 값으로 나타낸 변수로, 각 프로세스가 최근에 CPU time을 “얼마나” 할당받았는지 측정하기 위함이다. 즉, 최근 CPU time이 그것보다 덜 최근인 것보다 더 가중치를 가지게 하는 개선 방법이다. (by pintos.pdf)

처음 thread가 생성된 초기 상태의 값은 0이고, 나머지의 경우 parent thread에게 상속받는다. timer interrupt가 일어날 때마다 running thread에 대해서만 1씩 증가한다. (idle running하는 경우는 제외) 그리고 모든 thead(running, ready, or blocked)에 대해 1초마다 모든 스레드의 recent_cpu를 다음과 같은 공식을 통해 재계산한다.

```
recent_cpu = (2 ** load_avg) / (2 ** load_avg + 1) * recent_cpu + nice
```

최근 실행한 cpu일 경우 가중치를 더해서 계산하는 exponentially weighted moving average(EWMA)방식이다. 이때 recent_cpu는 “최근”에 thread가 받은 CPU time을 추정하는데, CPU를 경쟁하는 thread의 수를 나타내는 load_avg 값에 반비례하게 감소한다 .

이때 negative nice value를 가진 thread가 공식에 따라 negative recent_cpu를 가지게 될 수 있는데, nice value를 0으로 바꾸면 정상동작하지 못하므로 유의해야한다.

본 lab에서는 `int thread_get_recent_cpu(void)` 를 구현해야 한다. 이 함수는 현재 thread의 recent_cpu의 100배 (rounded to the nearest interget) return한다.

- **Load_avg**

지난 1분 동안 준비 상태에 있는 스레드의 평균 수를 추정한다. ready_threads는 업데이트 시점에서 실행 중이거나 준비 상태에 있는 스레드의 수(idle 스레드는 제외)를 의미한다. recent_cpu와 같은 EWMA 방식으로 계산한다. load_avg는 **시스**

템 전체적이라서 thread 별로 지정되지 않는다. (not thread-specific) 그래서 시스템 부팅될 시에 0으로 초기화되고, 매 초마다 다음과 같이 재계산된다.

***load_avg* = (59/60) * load_avg + (1/60) * ready_threads**

본 lab에서는 `int thread_get_load_avg (void)`를 구현해야 한다. 이 함수는 현재 system의 load average의 100배 (rounded to the nearest interget) return한다.

- **fixed point operation**

또한 위의 값에서 priority, nice, ready_threads값은 정수, recent_cpu, load_avg값은 실수이다. 하지만 pintos는 부동소수점 연산을 지원하지 않는다. project1 문서에서는 이를 연산할 수 있는 방법을 Appendix에서 소개하고 있다. (pintos 공식 문서의 내용과 동일하다.)

n : integer x, y : fixed-point numbers f : 17.14로 표현한 1 (즉, f는 1 << 14의 값으로 사용한다)

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 >= 0, (x - f / 2) / f if x <= 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

정리하면, 해결해야할 문제는 다음과 같은 두 가지다.

Problem1. 위의 부록 개념에 따라 MLFQS scheduler를 구현하는 것

Problem2. MLFQS에서 priority scheduling을 구현하는 것

pintos 문서에 따르면, default로는 priority scheduler가 활성화되고, mlfqs kernel option으로 MLFQS scheduler를 선택할 수 있다. (thread_mlfqs, declared in threads/thread.h, execution example : pintos –b -- -mlfqs …)

advanced scheduler에서는 priority donation을 하지 않고, priority를 4 ticks마다 계산하도록 하여 변경하도록 한다. 또한 highest priority의 thread의 경우에는 round robin scheduling method를 사용한다.

2) Our Design & Implementation

먼저 mlfqs scheduler를 구현할 때 이 scheduler가 실행될 시 전역 변수인 thread_mlfqs 값이 true로 설정된다. 따라서 이 변수 값을 확인하고 mlfqs scheduler가 실행되게 구현하면 된다. mlfqs에서 priority를 계산할 때 필요한 변수들을 위에서 살펴보았다. 이 값들을 새로 정의하고 계산하여 관리해주는 코드를 추가하여 완성했다.

- **Rationales**

Advanced Scheduling의 구현에서는 pintos 공식 문서에 나와있는 공식을 그대로 구현하였다. 그외에 timer_interrupt에서 정확한 계산 주기에 맞게 함수를 호출하여 계산하는 것이 중요하다. priority를 계산하기 위해 여러 변수 값을 사용하기 때문에 정확한 값이어야 할 것이다. 이때 pintos 공식 문서의 주기대로 주기를 설정해주었으며, load_avg 이후에 recent_cpu를 계산하도록 했다. 왜냐하면 recent_cpu는 load_avg 값을 이용하기 때문이다.

최종 설계와 관련하여 추가하거나 수정한 function, data stucture는 다음과 같다.

1. fixed-point operation 추가

```
#ifndef FIXED_POINT_H
#define FIXED_POINT_H
#include <stdint.h>
#define F (1<<14)

int convert_n_to_fp (int n) {return n * F;}
int convert_x_to_int_zero(int x) {return x/F;}
int convert_x_to_int_nearest(int x){
    if (x>=0) return (x+F/2)/F;
    else return (x-F/2)/F;
}
int add_x_y (int x, int y){return x+y;}
int sub_y_from_x(int x, int y){return x-y;}
int add_x_n(int x, int n) {return x+n*F;}
int sub_n_from_x(int x, int n) {return x-n*F;}
int mul_x_by_y(int x, int y){return ((int64_t)x)*y/F;}
int mul_x_by_n(int x, int n){return x*n;}
int div_x_by_y(int x, int y){return ((int64_t)x)*F/y;}
int div_x_by_n(int x, int n){return x/n;}
```

pintos/src/threads에 fixed_point.h라는 파일을 추가하여 부동소수점 연산에 대한 함수를 정의하여 사용했다. 이는 pintos 공식 문서에 나와있는대로 동일하게 구현하였다. 우리가 priority 계산을 하기 위해 실수값 변수를 사용해야 하기 때문에 필요한 과정이다.

2. priority 계산 함수 추가

```
struct thread
  {
    ...
    /* modified for lab1_3 */
    int nice;
    int recent_cpu;
    ...
  };

int load_avg;
```

priority 값을 구하기 위해서, nice, recent_cpu, load_avg 값이 필요하다. thread 별로 nice, recent_cpu 값이 다르기 때문에 `struct thread`에서 관리하는 변수를 추가했다. int nice와 int recent_cpu를 변수 선언한다. load_avg 값은 thread별이 아니라 시스템 전체에서 동일한 값을 사용하기 때문에 전역 변수로 선언했다.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ...
  /* modified for lab1_3 */
  t->nice = 0;
  t->recent_cpu = 0;
  ...
}

void
thread_start (void)
{
  ...
  load_avg = 0;
  ...
}
```

nice, recent_cpu를 `init_thread()` 에서 값을 0으로 초기화해준다. load_avg 값을 `thread_start()`에서 0으로 초기화한다.

priority 계산과 관련된 구현은 다음과 같다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  /* modified for lab1_3 */
  if (!thread_mlfqs)
  {
    /* modified for lab1_1 */
    wake_thread(ticks);
  }
  else
  {
    // increment recent_cpu per 1 tick
    struct thread* cur = thread_current();
    mlfqs_increment_recent_cpu(cur);
    if (ticks % 100 == 0)
      {
        mlfqs_calculate_load_avg();
        mlfqs_calculate_recent_cpu();
      }
    if (ticks % 4 == 0)
      {
      mlfqs_calculate_priority();
      }
    wake_thread(ticks);
  }

}
```

`timer_interrupt()` 에서 일정 주기에 따라 함수를 호출함으로써 각 변수의 계산 주기에 맞게 계산하는 함수를 호출하도록 한다. 1초마다 load_avg, 모든 thread의 recent_cpu를 공식에 따라 재계산하고, 실행하는 thread의 recent_cpu를 1만큼 증가시킨다. 그리고 4 tick 마다 현재 thread의 priority를 재계산한다. 또한, 이러한 과정들은 모두 thread_mlfqs의 값이 true인지 확인하고, 해당 조건일때만 실행하도록 한다.

이 함수에서 다음과 같은 함수를 호출하여 사용했고, 계산 과정에서 fixed_point.h에서 정의한 함수를 사용했다.

A. priority 계산

```
void mlfqs_calculate_priority ()
{
  struct thread* cur;
  struct list_elem* elem;
  for (elem = list_begin(&all_list); elem != list_end(&all_list);
                                    elem = list_next(elem))
  {
    cur = list_entry(elem, struct thread, allelem);
    // priority calculation = PRI_MAX - (recent_cpu/4) - (nice * 2)
    if (cur != idle_thread)
    {
      // 1. PRI_MAX - (recent_cpu/4) -> rounding since recent_cpu is real number
      int temp = sub_y_from_x(PRI_MAX,
      convert_x_to_int_nearest(div_x_by_n(cur->recent_cpu, 4)));
      cur->priority = sub_y_from_x(temp, mul_x_by_n(cur->nice, 2));
    }
  }
}
```

`void mlfqs_calculate_priority (void);`을 구현했다. 4초마다 모든 thread의 priority를 재계산한다. 모든 thread를 관리하는 리스트인 all_list를 순회하면서 공식에 맞게 priority를 계산하여 업데이트한다. 이때 해당 thread가 idle_thread가 아닌지 확인한다.

B. recent_cpu 계산

```
void mlfqs_calculate_recent_cpu ()
{
  struct thread* cur;
  struct list_elem* elem;
  for (elem = list_begin(&all_list); elem != list_end(&all_list);
                                     elem = list_next(elem))
  {
    cur = list_entry(elem, struct thread, allelem);
    // recent_cpu = (2*load_avg)/(2*load_avg+1) * recent_cpu + nice
    if (cur != idle_thread)
    {
      // 1. temp = (2*load_avg)/(2*load_avg+1)
      int temp = div_x_by_y(mul_x_by_n(load_avg, 2),
      add_x_n((mul_x_by_n(load_avg, 2)), 1));
      // 2. temp = temp * recent_cpu
      temp = mul_x_by_y(temp, cur->recent_cpu);
      // 3. recent_cpu = temp + nice
      cur->recent_cpu = add_x_n(temp, cur->nice);
    }
  }
}
```

`void mlfqs_calculate_recent_cpu (void);`를 구현했다. 1초마다 모든 thread의 recent_cpu를 재계산한다. priority 계산과 비슷하게, all_list를 순회하면서 공식에 맞게 priority를 계산하여 업데이트한다. 이때 해당 thread가 idle_thread가 아닌지 확인한다.

C. load_avg 계산

```
void mlfqs_calculate_load_avg (void)
{
  struct thread* cur = thread_current();
  // load_avg = (59/60) * load_avg + (1/60) * ready_threads
  // 1.temp = (59/60) * load_avg
  int temp = mul_x_by_y(div_x_by_y(convert_n_to_fp(59), convert_n_to_fp(60)),
                        load_avg);
  // 2. ready_threads
  int ready_threads = (cur != idle_thread) ? (list_size(&ready_list)+1)
                                           : list_size(&ready_list);
  // 3. load_avg = temp + (1/60) * ready_threads
  load_avg = add_x_y(temp, mul_x_by_n(div_x_by_y(convert_n_to_fp(1),
                               convert_n_to_fp(60)), ready_threads));
}
```

`void mlfqs_calculate_load_avg (void);`를 구현했다. load_avg 계산 공식에서 ready_threads라는 값이 사용되는데, 이를 계산하는 과정이 필요하다. ready_threads는 running 중이거나, ready 상태에 있는 thread들의 개수를 의미하며 idle_thread는 제외한다. 따라서 current thread가 idle_thread인지 조건을 확인하여 ready_threads의 값을 설정한다. 만약 idle_thread이면 ready_list의 크기에 현재 실행중인 thread를 포함하여 1을 더하면 된다. 이후 공식에 맞게 계산하여 load_avg 값을 업데이트한다.

D. 해당 thread의 recent_cpu를 1만큼 증가시키는 `void mlfqs_increment_recent_cpu (struct thread* t);`를 구현하였다.

```
void mlfqs_increment_recent_cpu (struct thread* t)
{
  if (t != idle_thread)
  {
    t->recent_cpu = add_x_n(t->recent_cpu, 1);
  }
}
```

3. **mlfqs scheduler priority donation 금지**

이전에 구현했던 priority donation을 mlfqs scheduler에서는 사용하지 않는다. 따라서 이전에 priority donation을 위해 수정했던 함수들에서 mlfqs scheduler가 아닐 때만 동작을 하도록 수정하는 과정이 필요했다. thread_mlfqs 값이 false일때만 해당 동작하도록 `thread_set_priority()` , `lock_acquire()`, `lock_release()` 함수를 수정했다.

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;

  // modified for donation
  struct thread *cur = thread_current();
  /* modified for lab1_3 */
  if (!thread_mlfqs)
  {
    thread_current ()->original_priority = new_priority;
    cur->priority = cur->original_priority;
    // 3. update current thread's priority according to donation list
    if(!list_empty(&cur->donation_list))
    {
      list_sort(&cur->donation_list, cmp_priority, NULL);
      int max_priority = (list_entry(list_begin(&cur->donation_list),
                          struct thread, donation_elem))->priority;
      cur->priority = cur->priority < max_priority
                    ? max_priority : cur->priority;
    }
  }

  /* modified for lab1_2 */
  //compare current thread's priority with new thread's priority
  check_priority_and_yield();

}
```

```
void
lock_acquire (struct lock *lock)
{
  ...
  /* modified for lab1_3*/
  if (!thread_mlfqs)
  {
    if (lock->holder)
    {
      thread_current()->waiting_lock = lock;
      list_insert_ordered(&lock->holder->donation_list,
                          &thread_current()->donation_elem, cmp_priority, NULL);
      donate_priority();
    }
  }
  // try to acquire lock
  sema_down (&lock->semaphore);
  if (!thread_mlfqs)
  {
    thread_current()->waiting_lock = NULL;
  }
  lock->holder = thread_current ();
}
```

```
void
lock_release (struct lock *lock)
{
  ...
  /* modified for donation in lab1_3 */
  if (!thread_mlfqs)
  {
```

```
    /* modified for donation in lab1_2 */
    // 1. update donation list
    delete_donation_list(lock);
    // 2. update current thread's priority to original
    struct thread *cur = lock->holder; // lock holder is current thread
    cur->priority = cur->original_priority;
    // 3. update current thread's priority according to donation list
    if(!list_empty(&cur->donation_list))
    {
      list_sort(&cur->donation_list, cmp_priority, NULL);
      int max_priority = (list_entry(list_begin(&cur->donation_list),
                        struct thread, donation_elem))->priority;
      cur->priority = cur->priority < max_priority
                    ? max_priority : cur->priority;
    }
  }
  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

4. **mlfqs 관련 변수를 return하는 함수 implement**

pintos.pdf에 구현이 필요한 함수 몇 개가 명시되어 있었다. 다음의 4가지 함수가 thread.c에 skeleton code가 주어져 있어서, 내부의 동작을 구현 완성하였다.

```
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int nice UNUSED)
{
  thread_current()->nice = nice;
}

/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
  return thread_current()->nice;
}

/* Returns 100 times the system load average. */
int
thread_get_load_avg (void)
{
  return convert_x_to_int_nearest(mul_x_by_n(load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu (void)
{
  return convert_x_to_int_nearest(mul_x_by_n(thread_current()->recent_cpu, 100));
}
```

`void thread_set_nice (int);` : 현재 thread의 nice 값을 변경한다. nice 값 변경 후에 현재 thread의 우선순위를 재계산 하고 우선순위에 의해 scheduling한다.

`int thread_get_nice (void);` : 현재 thread의 nice 값을 return한다.

`int thread_get_load_avg (void);` : 현재 system의 load average의 100배 (rounded to the nearest interget) return한다.

`int thread_get_recent_cpu (void)` : 현재 thread의 recent_cpu의 100배 (rounded to the nearest interget) return한다.

위에서 return 해야할 값들은 struct thread 안에 정의한 값으로, member 변수를 return하는 것으로 구현하고, load_avg의 경우에는 전역변수를 그대로 사용했다. load_avg, recent_cpu는 실수값이기 때문에 이때도 fixed_point.h에 있는 함수를 이용하여 Int 계산을 수행하도록 한다.

---

4. Results

team7 서버에 제출한 후 make check, make grade를 입력하여 채점한 결과 모두 통과하였다.

- make check 결과

```
team7@cse-edu ~/pintos/src/threads  project1 make check
cd build && make check
make[1]: Entering directory '/home/team7/pintos/src/threads/build'
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

- make grade 결과
  - total

```
team7@cse-edu ~/pintos/src/threads  project1 make grade
cd build && make grade
make[1]: Entering directory '/home/team7/pintos/src/threads/build'
../../tests/make-grade ../.. results ../../tests/threads/Grading | tee grade
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                    Pts Max  % Ttl  % Max
------------------------------------------- --- --- ------ ------
tests/threads/Rubric.alarm                   18/ 18  20.0%/ 20.0%
tests/threads/Rubric.priority                38/ 38  40.0%/ 40.0%
tests/threads/Rubric.mlfqs                   37/ 37  40.0%/ 40.0%
------------------------------------------- --- --- ------ ------
Total                                               100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

  - alarm

```
SUMMARY OF INDIVIDUAL TESTS

Functionality and robustness of alarm clock (tests/threads/Rubric.alarm):
             4/ 4 tests/threads/alarm-single
             4/ 4 tests/threads/alarm-multiple
             4/ 4 tests/threads/alarm-simultaneous
             4/ 4 tests/threads/alarm-priority

             1/ 1 tests/threads/alarm-zero
             1/ 1 tests/threads/alarm-negative

       - Section summary.
            6/  6 tests passed
           18/ 18 points subtotal
```

  - priority scheduler

```
Functionality of priority scheduler (tests/threads/Rubric.priority):
             3/ 3 tests/threads/priority-change
             3/ 3 tests/threads/priority-preempt

             3/ 3 tests/threads/priority-fifo
             3/ 3 tests/threads/priority-sema
             3/ 3 tests/threads/priority-condvar

             3/ 3 tests/threads/priority-donate-one
             3/ 3 tests/threads/priority-donate-multiple
             3/ 3 tests/threads/priority-donate-multiple2
             3/ 3 tests/threads/priority-donate-nest
             5/ 5 tests/threads/priority-donate-chain
             3/ 3 tests/threads/priority-donate-sema
             3/ 3 tests/threads/priority-donate-lower

        - Section summary.
             12/ 12 tests passed
             38/ 38 points subtotal
```

- advanced scheduler

```
Functionality of advanced scheduler (tests/threads/Rubric.mlfqs):
             5/ 5 tests/threads/mlfqs-load-1
             5/ 5 tests/threads/mlfqs-load-60
             3/ 3 tests/threads/mlfqs-load-avg

             5/ 5 tests/threads/mlfqs-recent-1

             5/ 5 tests/threads/mlfqs-fair-2
             3/ 3 tests/threads/mlfqs-fair-20

             4/ 4 tests/threads/mlfqs-nice-2
             2/ 2 tests/threads/mlfqs-nice-10

             5/ 5 tests/threads/mlfqs-block

        - Section summary.
              9/  9 tests passed
             37/ 37 points subtotal
```

---

5. Discussion

Design report와의 비교

how you actually implemented, and why such changes are necessary

design report를 작성할 때는 3. Advanced Scheduler에서 필요한 값들의 연산 순서에 대해 고려했었다. 그리고 실제 coding을 하면서 구현하는 과정에서, 연산하는 순서에 따라서 결과가 달라질 수 있다는 것을 직접 확인하였다. timer_interrupt 함수에서 load_avg와 recent cpu 계산을 순차적으로 하는데, 이때 load_avg를 계산한 이후에 recent_cpu를 계산해야 정확한 값을 계산할 수 있다. recent_cpu 계산 과정에서 load_avg 값을 이용하기 때문이다. 이외에도, 본 lab project1을 수행하는 과정에서 design report 설계와 동일하게 구현하였다. 초기 설계를 할 당시에 pintos files들을 꼼꼼히 분석하고, 실제로 미리 코딩을 하면서 설계를 완성해 나갔기 때문이다.

구현 과정에서 겪은 challenges

이번 pintos project1을 하면서 operating system 과목에서 배운 내용을 직접 구현하며 깊이 이해할 수 있었다. pintos를 처음 시작하면서 linux server 개발 환경을 구축하는 과정에서 어려움을 겪었었다. 팀원끼리 실시간으로 같이 토의하며 같은 서버에 접속하여 코딩을 하는 식으로 진행했는데, 이 과정에서 git 공동 작업 관련 문제가 있었다. 서버의 같은 repository에서 동시에 수정하였는데, git commit이 한 계정으로만 가능한 문제가 발생하였다. 이에 다른 repository로 2개를 만들고 각자 git에 commit하는 방식으로 해결할 수 있었다. 팀원끼리 협업하면서 git 사용에 익숙해졌고, 앞으로의 lab에서 더 잘 활용할 수 있을 것으로 기대한다.

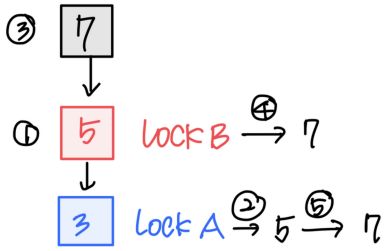
- **Lab1_1 Alarm Clock**

  alarm clock의 busy waiting을 해결하기 위해서는 먼저 thread의 life cycle에 대한 이해가 필요했다. thread를 언제 ready list에 추가하고, 언제 삭제하여 scheduling하는지 pintos 내의 코드들을 보면서 분석하였다. 이론에서 배운 life cycle의 내용만 가지고 코드를 구현하기보다, 실제로 schedule 함수가 어떠한 동작을 하고, 이 함수를 언제 호출하는지

분석하는 과정에서 더 완벽히 이해할 수 있었다. 그리고 thread_schedule_tail() 함수에서 후처리 작업까지 하는 것도 확인할 수 있었다. 이 함수에서 switch하기 전의 previous thread의 상태가 DYING이면 page를 free시켜서 메모리를 점유하는 것을 막는다는 것도 알게 되었다.

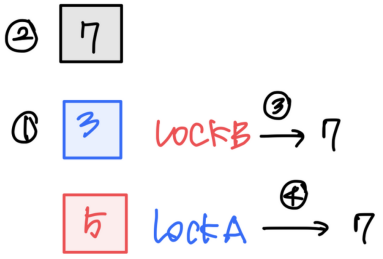
- **Lab1_2 Priority Donation**

  priority donation을 위한 `donate_priority()` 함수의 동작을 설계할 때 논의 과정을 설명하고자 한다. priority donation 구현을 위해 nested donation의 상황에서 어떻게 동작해야 하는지를 이해했다.



위 그림과 같이 lock holder(3)을 기준으로 현재 가지고 있는 lock을 waiting하는 thread(5)뿐만 아니라 그 thread(5)가 가진 lock을 waiting하는 thread의 priority에도 영향을 받는 구조였다. 이에 재귀적으로 동작하는 것이 필요하다고 생각했고 재귀적으로 구현하였다. 재귀 구현을 위해 반복 시에 current thread는 현재 current thread가 acquire하고자 하는 lock의 holder가 되는 구조로 구현했다.

그러나 재귀적으로 구현하는 과정에서 새로운 논의사항이 있었다. 우리는 초기에 재귀 구현을 할 시에 current thread가 lock holder보다 priority가 작을 시 재귀가 끝나야 한다고 생각하였다. 아래 그림을 통해 자세히 살펴보자.



위 그림을 보면 알겠지만 3이 5에게 priority donation을 하지 않더라도 재귀를 더 진행하여 7이 lock 5에게 donation 한 경우에는 결국 7이라는 priority를 가지는 thread가 5 priority를 가진 thread를 기다리는 경우가 생기기 때문에 재귀를 계속 진행하여야 한다고 판단하였다. 즉, 재귀 종료 조건이 현재 thread가 waiting lock이 없을 때만이어야 하며, 현재 thread가 lock holder보다 priority가 작다고 해서 재귀를 종료해서는 안됨을 깨달았다.