Assn1 design report

Pintos Assn1 Design Report

20200229 김경민, 20200423 성치호

이론적 배경

다음은 Pintos 구현을 위한 간단한 이론적 배경이다. 아래 코드 분석은 해당 내용을 바탕으로 한다.

Thread

스레드(Thread)란 프로세스 내에서 실행되는 흐름의 단위를 일컫는다. 효율적인 작업 수행을 위해 여러 스레드를 동시에 동작시킬 수 있는데, 이를 멀티스레딩(Multithreading)이라고 한다. 하지만 여러 스레드를 동시에 실행시키는 데 필요한 컴퓨팅 자원은 한정적이고, 다수의 스레드가 동시에 특정 메모리를 참조하고 변경하는 중 발생할 수 있는 동시성(concurrency) 문제가 존재하므로 이에 대한 체계적인 관리가 필수적으로 수반되어야 한다.

스레드가 실행 가능한 상태로 생성이 될 시, 최초로 생성된 스레드를 제외하고 스레드 생성 직후를 제외하고는 대기(Ready)중인 상태로 설정된다. 대기 중인 스레드는 스케줄러(Scheduler)에 의해 우선순위에 기반하여 한 스레드씩 실행 중인(Running) 상태로 바뀌는데, 이때 스택 및 레지스터 등의 정보들을 기존의 스레드와 교체하는 동작을 Context Switching 이라고 한다. 현재 Running 중인 스레드는 도중에 자신의 우선순위가 다른 스레드보다 낮아졌을 시 해당 스레드에게 컨텍스트를 넘겨줄 (yield) 수 있고, 실행 도중 작업이 종료(exit)되거나 `Sleep` 등 명시적인 동작에 의해 실행을 중단당할 수도 있다 (blocked).
하지만 현재 핀토스의 구현은 우선순위가 아닌 러닝 큐에 들어온 스레드 순서이고 이를 우선순위로 인한 것으로 변경하는 것이 이번 과제의 목표 중 하나이다.

이러한 방식으로 여러 스레드들을 짧은 시간동안 돌아가며 실행함으로써, 추상적인 관점에서 봤을 때 여러 동작들이 한 번에 실행되는 멀티스레딩을 구현할 수 있다.

Interrupt

스레드가 실행되는 외부 시그널 입력 등에 의한 예기치 않은 상황들을 처리해야 하는 상황들이 존재하는데, 이를 인터럽트(Interrupt)라고 한다. 인터럽트 처리 로직(handler)은 OS 커널에 미리 정의가 되어 있어 인터럽트가 발생할 시 해당 로직으로 컨텍스트 스위치가 발생하게 된다. 이때 보안 상의 이유로 낮은 권한 (User mode) 에서 실행되던 흐름을 일시적으로 Kernel mode로 상승시켜주고, 인터럽트 처리 로직 실행 중 필요한 스택 공간 역시 높은 권한의 Kernel memory로 스위칭된다.

인터럽트 처리가 끝난 뒤에는 인터럽트 상황에 따라 기존 프로그램 로직으로 복귀하거나 프로그램이 종료될 수 있는데, 복귀의 경우 실행 권한을 다시 User mode로 내린 뒤 스택 공간 또한 기존 User memory로 스위칭한다.

Pintos 구현 분석

자주 사용되는 구조체 및 함수

`ptov(uintptr_t paddr)`

```
static inline void *
ptov (uintptr_t paddr)
{
  ASSERT ((void *) paddr < PHYS_BASE);

  return (void *) (paddr + PHYS_BASE);
}
```

> 매개변수로 받은 physical 주소 `paddr`과 매핑되는 kernel의 가상 주소를 반환하는 함수이다.

리스트 in `kernel/lish.h,list.c`

Pintos에서는 `list_elem`, `list` 구조체 및 이를 이용한 함수를 통해 Double Linked List를 구현하였다. 해당 함수 및 구조체를 통해 구현한 Double Linked List는 리스트를 구성하는 element들이 연속된 메모리 공간에 위치하거나 dynamic allocated memory 사용을 요구

하지 않는다. 하지만 만일 `A` 구조체로 이루어진 리스트를 구현하고자 하면 `A` 구조체가 멤버 변수로 `list_elem` 구조체의 변수를 포함하고 그 값을 관리해주어야만 한다. 그 이유는 뒤에서 리스트 구현을 이루는 구조체와 함수를 설명하며 차근차근 설명하고자 한다.

`list_elem`

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

리스트를 이루는 각 element의 **위치 정보**를 나타내는 구조체로 리스트 내에서 해당 위치의 앞, 뒤에 위치한 element의 위치정보를 담고 있는 `list_elem` 구조체의 주소를 각각 `prev`, `next` 에 담는다.

주의할 점은 `list_elem` 구조체 자체는 실제 구현하고자 하는 리스트의 element 자체를 가지거나 가리키는 것이 아닌 해당 element의 리스트 내 **위치정보**만을 담고 있는 것이다.
Ex. `Foo` 구조체로 이루어진 리스트를 구현하였다고 할 때, `Foo` 구조체 `foo1` 의 `list_elem` 은 어떤 리스트 내 `foo1` 의 위치 정보만을 담고 있다.

`list`

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

구현하고자 하는 리스트의 처음과 끝에 해당하는 `list_elem` 구조체 변수를 `head`, `tail` 에 저장하는 구조체. 즉, 해당 `list` 구조체 변수를 가지고 있다면 `list_prev`, `list_next` 등의 함수를 이용하여 리스트의 처음, 또는 끝부터 그 반대편까지 리스트 전체를 조회하며 순회할 수 있다.

`list_entry(LIST_ELEM, STRUCT, MEMBER)`

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

> `LIST_ELEM` 이 가리키는 `list_elem` 가 포함된 `STRUCT` 구조체 변수의 주소를 반환하는 매크로. 즉 (`LIST_ELEM` 이 가리키는 `list_elem`)에 대응되는 `STRUCT` 구조체 변수 주소를 반환한다.

매개변수	설명
`STRUCT`	어떤 리스트를 이루는 구조체 (구조체 구조 그 자체)
`LIST_ELEM`	어떤 `STRUCT` 구조체 `a` 내에 저장된 `list_elem` 구조체 변수의 주소, 즉 리스트 내 `a` 의 위치정보를 저장한 `list_elem` 의 주소
`MEMBER`	`STRUCT` 내에 `list_element` 구조체를 저장하는 변수명

`LIST_ELEM` 구조체 변수의 `next` 의 주소에서 (`STRUCT` 구조체 구조 내에서 struct 시작으로부터 `MEMBER.next` 의 위치)를 빼어 `LIST_ELEM` 변수가 담긴 struct의 시작 즉, 해당 `list_elem` 에 대응되는 `STRUCT` 구조체 변수의 주소를 얻게 된다.

`list_next, list_prev`

```
struct list_elem *
list_next (struct list_elem *elem)
{
  ASSERT (is_head (elem) || is_interior (elem));
  return elem->next;
}

struct list_elem *
```

```
list_prev (struct list_elem *elem)
{
  ASSERT (is_interior (elem) || is_tail (elem));
  return elem->prev;
}
```

> 입력받은 `list_elem` 포인터가 가리키는 `list_elem` 구조체 변수의 리스트 내 뒤, 앞에 위치한 `list_elem`의 주소를 반환하는 함수
> 이를 이용해 리스트를 순회할 수 있다.

Kernel Initialization

함수 소개

`bss_init(void)`

```
static void
bss_init (void)
{
  extern char _start_bss, _end_bss;
  memset (&_start_bss, 0, &_end_bss - &_start_bss);
}
```

> BSS를 0으로 초기화하는 함수

먼저 bss란 "Blocked Started By Symbol"의 약자로 정적으로 할당된 변수를 포함하며, 초기화되지 않은 전역 변수를 포괄하며 이들은 공간 활용을 극대화하기 위해 메모리에서 분리되어 저장된다.(값을 가지고 있지 않기에)
`kernel.lds.S`에서 정의된 `_start_bss`, `_end_bss` 값을 이용한다.

```
  /* BSS (zero-initialized data) is after everything else. */
  _start_bss = .;
  .bss : { *(.bss) }
  _end_bss = .;
```

`_start_bss`는 bss가 저장된 구간의 바로 앞에 위치, `_end_bss`는 bss 바로 뒤에 위치한다.
`memset`을 통해 `_start_bss`의 주소로부터 `_end_bss`의 주소까지의 값을 모두 0으로 초기화하여 bss에 해당하는 변수(메모리)의 값을 0으로 초기화해준다.

`paging_init(void)`

```
static void
paging_init (void)
{
  uint32_t *pd, *pt;
  size_t page;
  extern char _start, _end_kernel_text;

  pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  pt = NULL;
  for (page = 0; page < init_ram_pages; page++)
    {
      uintptr_t paddr = page * PGSIZE;
      char *vaddr = ptov (paddr);
      size_t pde_idx = pd_no (vaddr);
      size_t pte_idx = pt_no (vaddr);
      bool in_kernel_text = &_start <= vaddr && vaddr < &_end_kernel_text;

      if (pd[pde_idx] == 0)
        {
          pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
          pd[pde_idx] = pde_create (pt);
        }

      pt[pte_idx] = pte_create_kernel (vaddr, !in_kernel_text);
    }
```

```
  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base Address
     of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init_page_dir)));
}
```

> 페이지 디렉토리, 페이지 테이블을 가상 메모리와 매핑하고 Page Directory와 Page Table에 free한 페이지를 allocate해 초기화하는 함수

`palloc_get_page`를 이용해 `pd`에 자유로운 페이지를 allocate한다. 해당 함수는 kernel의 init 중 일부로 매우 중요한 작업이기에 page 할당 중 실패하였을 때 패닉한다.
page의 개수만큼 순회하며 각 페이지의 물리적 주소에 대응되는 virtual address를 구하고 이에 해당하는 페이지 디렉토리 엔트리 인덱스(`pde_idx`), 페이지 테이블 엔트리 인덱스(`pte_idx`)을 구한다. 또한 페이지 테이블을 위한 페이지를 할당해주기도 한다.

`read_command_line(void)`

```
static char **
read_command_line (void)
{
  static char *argv[LOADER_ARGS_LEN / 2 + 1];
  char *p, *end;
  int argc;
  int i;

  argc = *(uint32_t *) ptov (LOADER_ARG_CNT);
  p = ptov (LOADER_ARGS);
  end = p + LOADER_ARGS_LEN;
```

> command arguments를 단어 단위로 잘라 그 문자열의 주소들을 리스트를 반환하는 함수

`argc`에 physical 주소 `LOADER_ARG_CNT`에 저장되어 있는 argument 개수를 저장하고 `p`에는 argument들이 저장되어 있는 physical 주소 `LOADER_ARGS`의 가상 주소를 저장한다.

```
  for (i = 0; i < argc; i++)
    {
      if (p >= end)
        PANIC ("command line arguments overflow");

      argv[i] = p;
      p += strnlen (p, end - p) + 1;
    }
  argv[argc] = NULL;

  /* Print kernel command line. */
  printf ("Kernel command line:");
  for (i = 0; i < argc; i++)
    if (strchr (argv[i], ' ') == NULL)
      printf (" %s", argv[i]);
    else
      printf (" '%s'", argv[i]);
  printf ("\n");

  return argv;
}
```

`p`에서부터 시작해 `\0`를 기준으로 문자열을 나누어 생각해 `0`로 구분된 각 문자열의 시작 주소를 `argv`(list 처럼 작동)에 차례대로 저장한다. 조회 중인 주소가 최대 수치의 주소(`end`)를 넘어서면 패닉을 일으킨다.
`argv[argc]`(끝 부분)에 `NULL`을 집어넣어 argument list의 끝을 표시한다.
그리고 커널에 집어넣은 커맨드를 출력하고 args가 단어 단위로 저장된 `argv`를 반환한다.

`parse_options(char **argv)`

```
static char **
parse_options (char **argv)
{
  for (; *argv != NULL && **argv == '-'; argv++)
    {
      char *save_ptr;
      char *name = strtok_r (*argv, "=", &save_ptr);
      char *value = strtok_r (NULL, "", &save_ptr);

      if (!strcmp (name, "-h"))
        usage ();
      else if (!strcmp (name, "-q"))
        shutdown_configure (SHUTDOWN_POWER_OFF);
      else if (!strcmp (name, "-r"))
        shutdown_configure (SHUTDOWN_REBOOT);
#ifdef FILESYS
      else if (!strcmp (name, "-f"))
        format_filesys = true;
      else if (!strcmp (name, "-filesys"))
        filesys_bdev_name = value;
      else if (!strcmp (name, "-scratch"))
        scratch_bdev_name = value;
#ifdef VM
      else if (!strcmp (name, "-swap"))
        swap_bdev_name = value;
#endif
#endif
      else if (!strcmp (name, "-rs"))
        random_init (atoi (value));
      else if (!strcmp (name, "-mlfqs"))
        thread_mlfqs = true;
#ifdef USERPROG
      else if (!strcmp (name, "-ul"))
        user_page_limit = atoi (value);
#endif
      else
        PANIC ("unknown option `%s' (use -h for help)", name);
    }

  /* Initialize the random number generator based on the system
     time.  This has no effect if an "-rs" option was specified.

     When running under Bochs, this is not enough by itself to
     get a good seed value, because the pintos script sets the
     initial time to a predictable value, not to the local time,
     for reproducibility.  To fix this, give the "-r" option to
     the pintos script to request real-time execution. */
  random_init (rtc_get_time ());

  return argv;
}
```

> argument의 주소들이 담긴 주소를 받아 각 option에 맞는 행위를 수행한 뒤 option이 아닌 첫번째 argument를 반환하는 함수

매개변수로 입력 받은 argument의 주소들이 담긴 list를 주소 값이 null이 아니거나 '-' 값으로 시작하는 동안 순회하며 확인한다. 순회한 주소의 문자열을 확인하여 '-h', '-q', '-r' 등에 일치하는지 확인하고 그에 맞는 작업을 수행하여 kernel의 설정을 변경한다. 그리고 '-'값으로 시작하지 않는 문자열의 주소가 담긴 주소(입력 매개변수 list element 주소 중 하나)를 반환한다. 해당 주소는 kernel이 실행할 명령어를 담고 있다.

`run_task(char **argv)`

```
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
```

```
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

> 매개변수로 받은 `argv`의 `argv[1]`가 가리키는 명령어를 `run_test` 또는 `process_wait`를 통해 수행하는 함수

매개변수로 받은 `char **argv`의 `argv[1]`가 가리키는 명령어를 `run_test` 또는 `process_wait`를 통해 수행한다.

`run_actions(char **argv)`

```
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };
```

> 매개변수로 입력 받은 argv를 순회하며 argv 내 명시된 action들을 순차적으로 수행하는 함수

`action` 구조체는 수행 가능한 액션의 이름, 필요한 argument 개수, 해당 액션을 수행하기 위한 function을 저장한다.
`actions`은 수행 가능한 action에 대응대는 `action` list로 현재로서는 `run`과 예외 처리를 위한 값인 `NULL`이 있다.

```
  while (*argv != NULL)
    {
      const struct action *a;
      int i;

      /* Find action name. */
      for (a = actions; ; a++)
        if (a->name == NULL)
          PANIC ("unknown action `%s' (use -h for help)", *argv);
        else if (!strcmp (*argv, a->name))
          break;

      /* Check for required arguments. */
      for (i = 1; i < a->argc; i++)
        if (argv[i] == NULL)
          PANIC ("action `%s' requires %d argument(s)", *argv, a->argc - 1);

      /* Invoke action and advance. */
      a->function (argv);
      argv += a->argc;
    }
```

```
}
```

매개변수로 입력받은 `argv`를 순회하며 주소가 가리키는 문자열과 `actions`의 아이템과 이름이 일치하는 것이 있는지 확인 후 있다면 해당 `action`에 명시된 argument 개수만큼 제공하였는지 확인한다. 개수가 동일하다면 함수에 해당 action에 해당되는 문자열 주소를 가리키는 argv 내 값의 주소를 해당 `action`에 해당되는 함수 `action->function`의 매개변수로 두어 해당 함수를 호출해 action을 수행한다. `argv`를 순회하며 NULL 값을 만날 때까지 이를 반복한다.
만약 `actions`에 없는 action을 요구하거나 `action`에 대한 args가 부족할 때는 패닉을 일으킨다.

Interrupt

`idt`

```
static uint64_t idt[INTR_CNT];
```

IDT(Interrupt Descriptor Table)을 저장하는 리스트 변수, `intr-stubs.S`에 있는 256개의 x86 interrupt stub과 연결하는 Gate를 저장한다. `intr_init`에서 초기화된다.

`intr_handler`

```
static intr_handler_func *intr_handlers[INTR_CNT];
```

`idt`에 있는 각 interrupt에 대한 intr handler function을 담은 리스트

`intr_level`

```
enum intr_level
  {
    INTR_OFF,             /* Interrupts disabled. */
    INTR_ON               /* Interrupts enabled. */
  };
```

Interrupt 활성화, 비활성화 여부를 표현하는 enum

`intr_get_level`

```
enum intr_level
intr_get_level (void)
{
  uint32_t flags;

  /* Push the flags register on the processor stack, then pop the
     value off the stack into `flags'.  See [IA32-v2b] "PUSHF"
     and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware
     Interrupts". */
  asm volatile ("pushfl; popl %0" : "=g" (flags));

  return flags & FLAG_IF ? INTR_ON : INTR_OFF;
}
```

processor stack에 flag register를 넣고 pop시켜 flag에 저장하여 flag 값을 확인하여 현재 Interrupt 활성화 여부를 `intr_level`로 반환하는 함수

`intr_set_level`

```
enum intr_level
intr_set_level (enum intr_level level)
{
```

```
  return level == INTR_ON ? intr_enable () : intr_disable ();
}
```

입력 받은 `intr_level` 에 따라 `intr_enable` 또는 `intr_disable` 을 통해 interrupt 활성화 여부를 조절하는 함수

`intr_enable`

```
enum intr_level
intr_enable (void)
{
  enum intr_level old_level = intr_get_level ();
  ASSERT (!intr_context ());

  /* Enable interrupts by setting the interrupt flag.

     See [IA32-v2b] "STI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("sti");

  return old_level;
}
```

> interrupt 를 enable하고 변경 전 원래의 interrupt level을 `intr_level` 꼴로 반환하는 함수
> external interrupt를 처리하고 있지 않을 때 실행되어야 한다.
> `sti` 어셈블리를 통해 interrupt flag를 설정함으로써 interrupt를 허용하고 변경 전 원래의 interrupt level을 반환한다.

`intr_disable`

```
enum intr_level
intr_disable (void)
{
  enum intr_level old_level = intr_get_level ();

  /* Disable interrupts by clearing the interrupt flag.
     See [IA32-v2b] "CLI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("cli" : : : "memory");

  return old_level;
}
```

> interrupt 를 disable하고 변경 전 원래의 interrupt level을 `intr_level` 꼴로 반환하는 함수
> `cli` 어셈블리를 통해 interrupt flag를 지움으로써 interrupt를 비활성화하고 변경 전 원래의 interrupt level을 반환한다.

`intr_init`

Programmable Interrupt controller를 초기화한다.
`idt` 를 순회하며 `intr_stubs` 를 참조하여 interrupt stub gate를 집어넣어 초기화한다.
이후 IDT register를 로드한다.
마지막으로 interrupt의 번호별 이름이 적힌 `intr_names` 를 초기화한다.

Thread

`THREAD_MAGIC`

```
#define THREAD_MAGIC 0xcd6abf4b
```

`thread` 에서 kernel stack의 overflow를 감지할 때 사용하는 상수 값. `thread` 구조체 내 thread에 관한 정보를 저장하는 공간 끝(`magic`)에 해당 상수를 저장한다. 만일 kernel stack이 커짐에 따라 overflow되어 `magic` 값이 기존 `THREAD_MAGIC` 에서 다른 값으로 변형되어 `magic` 값을 통해 kernel stack의 overflow 여부를 감지할 수 있다. 자세한 내용은 아래 `thread` 구조체에 대한 설명에서 확인할 수 있다.

`thread_status`

```
enum thread_status
  {
    THREAD_RUNNING,     /* Running thread. */
    THREAD_READY,       /* Not running but ready to run. */
    THREAD_BLOCKED,     /* Waiting for an event to trigger. */
    THREAD_DYING        /* About to be destroyed. */
  };
```

`thread` 구조체에서 `thread`의 상태를 나타낼 때 사용하는 열거형.
위의 이론적 배경에서 설명하였듯 스레드는 4개의 스레드 상태를 왔다갔다하는 lifecycle을 가진다.

`tid_t`

```
typedef int tid_t;
```

Thread를 구분하는 id를 위한 자료형(Thread identifier type)이다.

`PRI_MIN, PRI_DEFAULT, PRI_MAX`

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

스레드(`thread`)가 가질 수 있는 priority의 범위 및 기본 값을 나타내는 상수이다. `PRI_MIN`, `PRI_MAX`는 각각 스레드의 우선순위(`thread` 구조체에서 `priority`)의 하한과 상환이다. 값은 클수록 우선순위가 높은 것(더 중요한 것)이다. `PRI_DEFAULT`는 스레드 생성 시 특별한 의도가 있지 않을 때 기본적으로 사용하는 `priority` 값이다.

`thread`

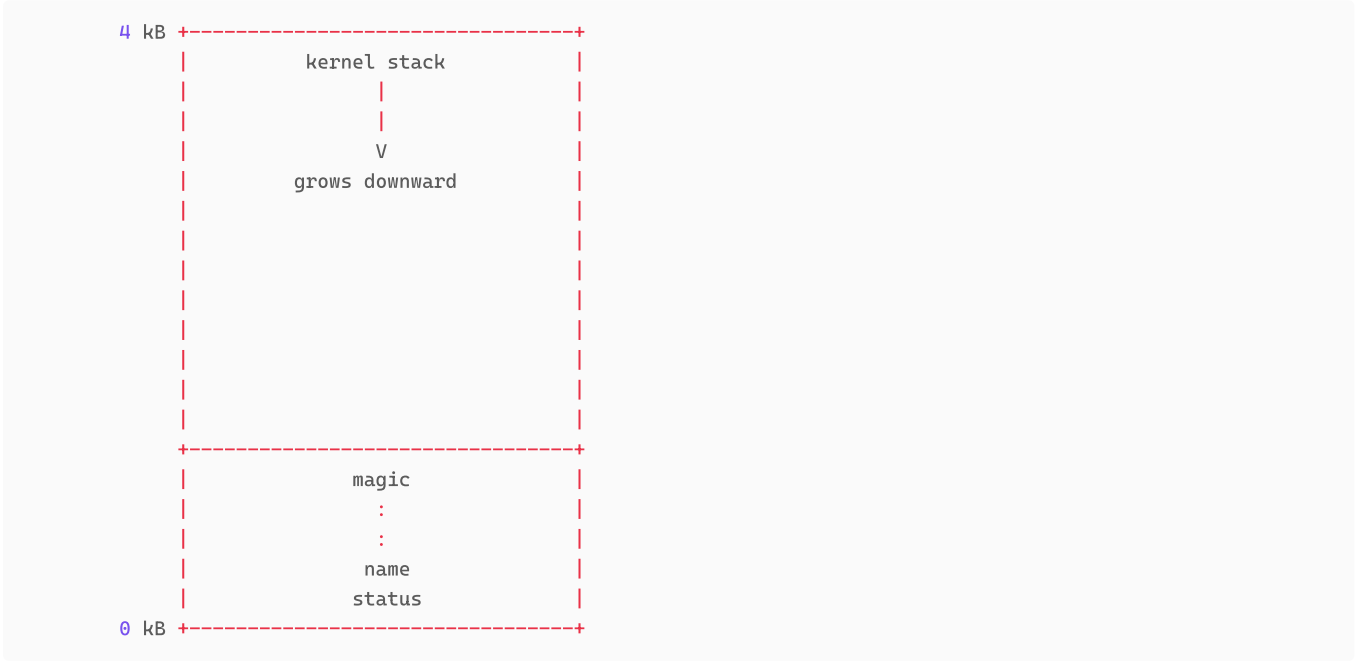
```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    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. */
  };
```

변수	자료형	설명
`tid`	`tid_t`	스레드를 서로 구별하는 id, 스레드 별로 유일한 값을 가진다.
`status`	enum `thread_status`	스레드의 현재 상태
`name`	`char`	디버깅용 스레드 이름
`stack`	`uint8_t`	해당 스레드의 stack을 가리키는 포인터 - 다른 스레드로 전환될 시 stack pointer를 해당 변수에 저장하여 이후에 thread가 다시 실행될 때 해당 pointer를 이용한다.
`priority`	`int`	스레드 우선순위(숫자가 클수록 높은 우선순위)

변수	자료형	설명
`allelem`	struct `list_elem`	모든 스레드를 포함하는 double linked list를 위한 아이템 하나
`elem`	struct `list_elem`	모든 스레드를 포함하는 double linked list를 위한 아이템 하나
`magic`	`unsigned`	`thread` 구조체의 가장 마지막 멤버 변수로, stack overflow를 감지하는 숫자. 항상 `THREAD_MAGIC` 으로 설정되어 있다. 만약 kernel stack이 커지다 thread struct 부분까지 침범하게 되면 magic 숫자가 변경되게 되고 `THREAD_MAGIC` 이 아니게 되어 이를 감지할 수 있다.

pintos는 thread를 thread에 대한 정보를 나타내는 상단 코드의 `thread` 구조체와 kernel stack으로 이루어진다. thread는 이 둘을 4kB의 page 내에 저장한다. 각 스레드는 4kB page 내 아래 구조와 같이 저장된다.



어떤 스레드에 대한 stack을 제외한 정보를 가지고 있는 `thread` 구조체는 해당 page의 가장 바닥(0)부터 위로 값을 저장한다. `thread` 를 저장한 부분의 끝에는 `THREAD_MAGIC` 가 저장된 `magic` 이 저장된다.
반대로 해당 스레드의 커널 스택은 `thread` 가 저장된 부분의 반대편인 page의 가장 천장(4kB)부터 저장하여 아래로 늘어난다.
만약 kernel stack이 길어지게 되어 `thread` 저장 공간을 침범하게 되면 `thread` 저장 부분의 가장 끝에 있는 `magic` 값이 기존 `THREAD_MAGIC` 에서 다른 값으로 변경되게 되고 이를 통해 kernel stack overflow를 감지하는 것이다.
이처럼 커널 스택을 포함한 스레드가 저장되는 공간은 4kB이기 때문에 스레드에서 이보다 더 큰 non-static local variables를 저장하고 싶다면 `malloc()`, `palloc_get_page()` 같은 dynamic allocation을 사용해야 한다.

`thread_mlfqs`

```
extern bool thread_mlfqs;
```

mlfqs(Multi level feedback queue scheduler)를 사용할지 여부를 담는 전역 변수이다.
기본 값은 false로 설정되는데 해당 값이 false라면 round-robin scheduler를 사용하고 true라면 Multi level feedback queue scheduler를 사용하는 것을 의미한다. 이 값에 따라 다른 스케줄러를 사용하도록 하는 것은 Assn1에서 우리가 구현해야할 사항이다.
해당 변수는 command-line option인 `-mlfqs` 옵션이 주어질 때, main함수에서 init 중 `parse_options` 에 의해 True로 설정된다.

`ready_list`

```
static struct list ready_list;
```

`thread_status` 가 `THREAD_READY` 인 프로세스 리스트를 저장하는 `list` 구조체

`all_list`

```
static struct list all_list;
```

지금까지 생성되었던 모든 프로세스를 저장하는 리스트를 저장하는 `list` 구조체

`idle_thread`

```
static struct thread *idle_thread;
```

`idle` 함수를 수행하는 스레드를 가리키는 포인터 변수. idle thread란 running될 준비가 된 스레드가 하나도 없을 때 실행되는 스레드이다. 해당 변수는 `thread_start`에서 `idle_thread`를 생성하고 이후 처음으로 `idle`을 실행해야할 스레드가 실행되었을 때 `idle`에서 초기화된다.

`initial_thread`

```
static struct thread *initial_thread;
```

처음으로 생성되고 running되는 스레드의 `thread`를 가리키는 포인터 변수
즉 스레드 시스템의 초기화 이후에는 커널 실행시 처음으로 작동하는 `threads/initc.c`의 `main` 함수에 해당되는 스레드를 가리킨다.

`tid_lock`

```
static struct lock tid_lock;
```

`allocate_tid()`를 수행할 때 사용되는 `lock`이다.
`allocate_tid()`에서 `tid` 즉 Thread마다 고유한 identifier를 지정할 때 tid를 할당할 때마다 이후 할당할 tid 값을 1씩 증가시키는데 증가, 할당 과정 중 다른 스레드의 개입을 막기위한 `lock`이다.
더 자세한 설명은 `allocate_tid`를 참고하면 된다.

`kernel_thread_frame`

```
struct kernel_thread_frame
  {
    void *eip;                  /* Return address. */
    thread_func *function;      /* Function to call. */
    void *aux;                  /* Auxiliary data for function. */
  };
```

`thread`의 `stack`의 주소부터 저장하는 정보의 구조. 해당 `stack` 위치에는 차례대로 `kernel_thread_frame`, `switch_entry_frame`, `switch_threads_frame` 등의 struct로 저장한다. 해당 stack frame에는 해당 커널 스레드가 어떤 매개변수를 받아 어떤 함수를 실행하고 어디로 리턴해야하는지에 대한 정보를 가지고 있다. 해당 정보는 `thread_create`에서 스레드를 생성할 때 초기화되어 스레드에 저장된다.

`idle_ticks, kernel_ticks, user_ticks`

실제 기능을 담당하는 것이 아닌 분석을 위한 변수들이다,
각각 idel Thread 보낸 총 tick, 커널 스레드에서 소요한 총 tick, user program에서 사용한 총 tick을 의미한다.

`thread_ticks`

마지막으로 yield한 이후 timer tick이 얼마나 지났는지 정하는 static 변수이다.
주기적으로 스케줄링을 수행해야 할 때 해당 변수 값을 참고한다.

`thread_init(void)`

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);
  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);
```

```
  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
}
```

> 스레드 시스템을 초기화하는 함수
> 스레드를 초기화하는 과정이기에 우선 Interrupts가 비활성화된 상태에서 실행되어야만 한다.
> 우선 스레드 시스템과 연관된 `list` 구조체 변수들(`ready_list` , `all_list`) 및 스레드 id 할당과 관련된 `lock` 구조체 변수인 `tid_lock` 을 초기화한다. 이후에는 현재 해당 함수를 실행 중인 스레드를 최초의 스레드(`initial_thread`)로 생각하여 본 스레드의 주소를 저장한다. 또한 `init_thread` 를 통해 이 함수를 실행 중인 스레드, 즉 `initial_thread` 의 `thread` 객체의 `name` 을 "main"으로 지정하고 `priority` 도 기본 값인 `PRI_DEFAULT` 로 지정해준다. 이미 해당 스레드는 `thread_init` 을 실행 중이기에 `status` 는 `THREAD_RUNNING` 으로 지정하고 `allocate_tid` 를 통해 새로운 `tid` 를 얻어 `initial_thread` 의 `tid` 로 지정한다. 이로써 해당 함수가 완료되면 본 스레드의 정보들이 올바르게 초기화 된다. **`thread_create` 를 통해 새로운 `thread` 를 생성하기 전에 page allocator 를 초기화해주어야만 한다.**

`thread_start(void)`

```
void
thread_start (void)
{
  /* Create the idle thread. */
  struct semaphore idle_started;
  sema_init (&idle_started, 0);
  thread_create ("idle", PRI_MIN, idle, &idle_started);

  /* Start preemptive thread scheduling. */
  intr_enable ();

  /* Wait for the idle thread to initialize idle_thread. */
  sema_down (&idle_started);
}
```

> `idle` 스레드를 생성, 초기화하고 interrupt를 활성화하여 스케줄링을 활성화하는 함수

여기서 idle thread란 ready된 다른 스레드가 존재하지 않을 때 실행되는 스레드이다.
interrupt를 활성화함으로써 preemptive(선점형) 스레드 스케줄링을 시작하는 함수로 idle thread도 생성한다. `idle_started` 라는 `semaphore` 를 생성하고 초기화한다.
`idle_started` 는 `idle` 함수가 실행되어 `idle_thread` 변수가 초기화가 올바르게 되었는지 확인하는 semaphore 변수이다. 본 함수 끝에서 `sema_down` 을 하였고 `sema_up` 을 할 때까지 기다린다. 본 함수를 통해 생성된 `idle` 함수를 실행하는 스레드가 running시 `idle` 함수에서 `idle_thread` 값을 본인으로 변경 후 `sema_up` 을 하여 `idle` 함수가 정상적으로 실행되었으며 `idle_thread` 가 잘 초기화 되었음을 표현한다. `sema_up` 이 되면 `thread_start` 는 완료되게 된다.

또한 해당 함수는 `thread_create` 를 통해 idle이라는 이름을 가지고 `priority` 는 `PRI_MIN` 으로 가장 낮은 `idle` 함수를 `idle_started` 의 주소를 인수로 사용하여 실행하는 thread를 생성한다.
이후에는 `intr_enable` 을 통해 interrupts를 허용한다.

`thread_tick(void)`

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;
```

```
  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
}
```

> timer tick마다 timer interrupt 핸들러에 의해 호출되는 함수이다.

현재 running 중인 스레드의 종류(idle, user program, kernel 스레드인지)에 따라 각 스레드가 소요한 시간을 계산하는 variable(`idle_ticks`, `user_ticks`, `kernel_ticks`)를 증가시켜 업데이트한다.
`thread_ticks` 를 증가시키고 만약 `TIME_SLICE` 보다 크거나 같다면 `intr_yield_on_return` 을 호출함으로써 새로운 프로세스에게 yield 하도록 한다. 즉 스케줄링이 일어나게 된다. 타이머에 의해 주기적으로 `TIME_SLICE` 틱마다 스케줄링이 일어난다.

`thread_print_stats`

```
void
thread_print_stats (void)
{
  printf ("Thread: %lld idle ticks, %lld kernel ticks, %lld user ticks\n",
          idle_ticks, kernel_ticks, user_ticks);
}
```

> 스레드의 종류별 소요 tick을 출력하는 함수

`thread_create(name, priority, function, aux)`

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);
```

> 매개변수로 받은 name과 priority를 가지고 function에 aux 인수를 넘겨 실행하는 스레드를 생성하고 ready 큐에 추가한 뒤 새로 생성한 스레드의 id를 반환하는 함수

`name` : 새로 생성할 스레드의 이름
`priority` : 새로 생성할 스레드의 priority
`function` : 스레드가 실행할 함수
`aux` : `function` 실행 때 넘길 argument

```
  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();
```

`palloc_get_page` 를 통해 스레드가 사용할 메모리, 페이지를 할당한다.
`init_thread` 를 통해 할당 받은 메모리를 0으로 초기화하고 해당 공간에 스레드의 정보를 담은 `thread` 구조체 변수 값을 입력받은 매개변수를 바탕으로 설정한다. 또한 이렇게 생성된 `thread` 의 `status` 는 처음에는 `THREAD_BLOCKED` 이다. 생성한 스레드를 `all_list` 에도 추가한다. 이후 `allocate_tid()` 를 통해 스레드의 id를 생성 후 집어 넣는다.

```
  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;
```

```
  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  /* Add to run queue. */
  thread_unblock (t);

  return tid;
}
```

각 스레드가 할당받은 메모리 중 스택의 최상단부터 `alloc_frame`을 이용해 `kernel_thread_frame`, `switch_entry_frame`, `switch_threads_frame` 공간을 차례로 할당한다. 또한 각 frame의 정보를 채워 넣어 초기화한다.
스레드 생성 및 초기화가 완료되었으므로 새로 생성한 스레드를 unblock한 뒤 스레드의 tid를 반환한다.

`thread_block(void)`

```
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}
```

> 현재 실행 중인 스레드의 `status`를 `THREAD_BLOCKED`로 변경하여 블록한 뒤 `schedule()`를 통해 스케줄링하는 함수

`!intr_context()`를 통해 현재 external interrupt를 처리 중이 아님을 확인한다. 또한 해당 함수 실행을 위해서는 interrupt가 비활성화 되어 있어야 한다.

`thread_unblock(struct thread *t)`

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

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

> 입력받은 매개변수가 가리키는 `thread`를 unblock시키는 함수

interrupt를 비활성화한 뒤 입력받은 스레드를 다시 `THREAD_READY` 상태의 스레드 리스트인 `ready_list`에 집어넣고 스레드의 `status`를 `THREAD_READY`로 변경한 뒤 원래 interrupt 레벨로 복구한다.

`thread_name`, `thread_tid`

> 현재 실행중인 스레드의 name, tid를 출력하는 함수

`thread_current`

```
struct thread *
thread_current (void)
{
  struct thread *t = running_thread ();

  /* Make sure T is really a thread.
     If either of these assertions fire, then your thread may
     have overflowed its stack.  Each thread has less than 4 kB
     of stack, so a few big automatic arrays or moderate
     recursion can cause stack overflow. */
  ASSERT (is_thread (t));
  ASSERT (t->status == THREAD_RUNNING);

  return t;
}
```

> 현재 running 중인 스레드의 주소를 반환하는 함수.

추가로 현재 실행 중인 것이 정말 올바른 형식의 스레드인지 실행 중인 스레드가 맞는지 등의 정합성 검사를 포함한다.

`thread_exit`

```
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
```

> 현재 해당 함수를 실행한 스레드를 스케줄에서 내리고 스레드 자체를 삭제하는 함수.

먼저 external interrupt를 처리하는 도중에 해당 작업을 진행해서는 안된다. 그렇기에 assert로 이를 확인하고, 또한 `intr_disable()`을 통해 interrupt를 비활성화한다. 그리고 현재 함수를 실행중인 스레드를 모든 스레드 목록이 담긴 `allelem`으로부터 삭제하고 스레드의 상태를 `THREAD_DYING`으로 변경한다. 그 뒤 `schedule`함수를 호출해 해당 스레드를 대신 실행될 스레드로 스위치하게 한다. 이후 해당 스레드는 절대 다시 running하지 않고 `thread_exit`을 호출한 이후 return되지 않는다.

`thread_yield`

```
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)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
```

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

> 현재 해당 함수를 실행 중인 스레드가 cpu 자원을 다른 스레드에게 넘기기 위해 자신을 ready 상태로 전환 뒤 스케줄링을 요청하는 함수이다.

다른 스레드 관리 함수와 마찬가지로 먼저 interrupt가 실행 중이지 않고, interrupt를 disable한 상태에서 작업을 진행한다.
해당 함수를 호출한 함수가 `idle` 스레드가 아니라면 자신을 스레드 레디큐인 `ready_list` 맨 뒤에 추가하고 스레드 상태도 `THREAD_READY` 로 전환한다. 이 때 `idle` 스레드가 아닐 때만 그러한 이유는 `idle` 스레드가 실행 중이라는 것은 실행할 수 있는 다른 스레드가 없음을 나타내기 때문으로 예상된다.
그리고 `schedule` 을 통해 스케줄링을 요청한다. 이로써 현재 핀토스 구현에서는 priority와 무관하게 현재 큐의 맨 앞의 스레드로 전환된다.
이후 interrupt 레벨을 원래대로 되돌린다.

`thread_foreach`

```
void
thread_foreach (thread_action_func *func, void *aux)
{
  struct list_elem *e;

  ASSERT (intr_get_level () == INTR_OFF);

  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e))
    {
      struct thread *t = list_entry (e, struct thread, allelem);
      func (t, aux);
    }
}
```

> 매개변수로 넘긴 함수를 함께 넘긴 매개변수 aux와 함께 모든 각 스레드를 대상으로 실행하는 함수이다.

interrupt가 비활성화된 상태에서만 실행되어야 한다.
`all_list` 를 순회하며 각 `list_elem` 이 가리키는 스레드의 주소를 매개변수로 받은 함수에 매개변수로 받은 aux와 함께 매개변수로 넣어 실행한다.

`idle`

```
static void
idle (void *idle_started_ UNUSED)
{
  struct semaphore *idle_started = idle_started_;
  idle_thread = thread_current ();
  sema_up (idle_started);

  for (;;)
    {
      /* Let someone else run. */
      intr_disable ();
      thread_block ();

      asm volatile ("sti; hlt" : : : "memory");
    }
}
```

> idle thread가 실행할 함수.

idle thread는 ready 상태의 thread가 하나도 없을 때 실행되는 스레드이다. `idle_thread` 에 현재 `idle` 을 실행 중인 자기 자신 스레드를 넣는다. 또한 `sema_up(idle_started)` 을 통해 `thread_start` 가 semaphore에서 풀려 더 진행 가능하게 한다. `sti; hlt` 는 atomically 하게 실행되며 이는 인터럽트를 활성화시키고 인터럽트 발생시까지 cpu를 대기시키는 것이다.
자기 자신이 특별한 일이 아닌 이상 running queue에 올라가지 않고 running되지 않도록 한다. (`thread_yield` 에서 `idle_thread` 일 경우 running queue에 올리지 않는다)

`next_thread_to_run`에서 `ready_list`가 비어있을 때만 `idle_thread`를 반환함으로써 `schedule`에서 다음에 `idle_thread`를 실행하게 된다.

`kernel_thread`

```
static void
kernel_thread (thread_func *function, void *aux)
{
  ASSERT (function != NULL);

  intr_enable ();       /* The scheduler runs with interrupts off. */
  function (aux);       /* Execute the thread function. */
  thread_exit ();       /* If function() returns, kill the thread. */
}
```

> kernel thread가 실행할 함수

interrupt를 허용하고 함수와 해당 함수에 사용할 매개변수를 받아 해당 함수를 실행한 뒤 실행 완료하면 `thread_exit`을 통해 `kernel_thread`를 실행 중인 스레드를 죽인다.

`running_thread`

```
struct thread *
running_thread (void)
{
  uint32_t *esp;

  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

> `esp` 변수에 CPU의 stack pointer를 복사해와 페이지 단위로 주소를 내림하여 현재 실행 중인 스레드의 `thread` 구조체의 주소를 얻는다. 앞서 말하였듯이 `thread` 구조체는 페이지 단위로 저장되기 때문이다.

`is_thread`

```
static bool
is_thread (struct thread *t)
{
  return t != NULL && t->magic == THREAD_MAGIC;
}
```

> 입력받은 포인터가 가리키는 `thread`의 정합성을 검사하여 반환하는 함수
> 입력받은 포인터가 가리키는 `thread`가 정상적인 `thread`를 가지고 있는지 `magic`을 통해 스택 오버플로우는 발생하지 않았는지 확인하여 `thread`의 정합성을 반환한다.

`init_thread`

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  enum intr_level old_level;

  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;
```

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

> 주로 이제 막 새로 생성된/공간을 할당 받은 스레드에 대해 이름, priority 설정을 비롯한 스레드를 위한 메모리 초기화 등, 기본적인 초기화를 진행하는 함수.

해당 함수는 주로 `thread_crate`에서 스레드를 위한 공간을 할당해 준 직후 실행되어 실질적으로 스레드를 초기화한다.
먼저 입력된 매개변수에 대한 조건 검사를 수행한다.
이후 입력받은 `thread`가 차지할 메모리를 0으로 초기화한다. 이후 `status`는 `THREAD_BLOCKED`**로 설정하고** 이름, 스택 위치 설정, 우선순위 설정 등 `thread` 구조체 초기화를 수행한다. 마지막으로 `interrupt`를 비활성하고 스레드 전체 리스트 `all_list`에 `thread`를 저장하고 다시 interrupt level을 되돌린다.

`alloc_frame`

```
static void *
alloc_frame (struct thread *t, size_t size)
{
  /* Stack data is always allocated in word-size units. */
  ASSERT (is_thread (t));
  ASSERT (size % sizeof (uint32_t) == 0);

  t->stack -= size;
  return t->stack;
}
```

먼저 정상적인 스레드인지, 정상적인 사이즈 할당을 요구하는지 검사한다.
`size`만큼 `thread`의 `stack`의 공간을 할당하고 할당한 크기만큼 `stack`의 위치를 옮겨준다. 그리고 그렇게 변경된 주소를 반환해준다. 이 후 `alloc_frame`을 통해 또 할당하는 경우 변경된 `stack`으로부터 이를 수행한다.

`next_thread_to_run`

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

> 다음에 어떤 스레드를 실행하게 스케줄할지 결정하여 반환하는 함수

ready된 스레드를 모아둔 `ready_list`가 비어 있다면 현재 ready된 스레드가 하나도 없으므로 `idle_thread`를 반환하고 비어있지 않다면 `ready_list` 큐의 pop 된 스레드를 반환한다. 즉 `ready_list`에 들어온 순서대로 선입선출 순서로 반환하게 될 것이다.

`thread_schedule_tail`

```
void
thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();

  ASSERT (intr_get_level () == INTR_OFF);

  /* Mark us as running. */
  cur->status = THREAD_RUNNING;

  /* Start new time slice. */
  thread_ticks = 0;

#ifdef USERPROG
```

```
  /* Activate the new address space. */
  process_activate ();
#endif

  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```

`schedule`

```
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);
}
```

> 다음에 실행할 스레드를 결정하고 그 스레드를 스케줄하는 함수

먼저 해당 함수는 반드시 interrupt가 비활성화된 상태, 현재 해당 함수를 실행하는 스레드가 `THREAD_RUNNING`이 아닌 다른 상태여야만 한다. `next_thread_to_run`을 통해 다음에 실행해야할 스레드가 무엇일지를 얻는다. 그리고 이렇게 얻은 스레드가 현재 스레드가 아니라면 `switch_threads`를 통해 해당 스레드로 switch하고 `thread_schedule_tail`을 호출한다.

`allocate_tid`

```
static tid_t
allocate_tid (void)
{
  static tid_t next_tid = 1;
  tid_t tid;

  lock_acquire (&tid_lock);
  tid = next_tid++;
  lock_release (&tid_lock);

  return tid;
}
```

> 새로 생성한 스레드를 위한 `tid`를 반환하는 함수

먼저 tid는 1부터 값을 1씩 증가해나가므로 모든 스레드는 서로 다른 tid를 가지며 늦게 생성됨에 따라 tid 값이 1씩 증가한다. static 변수 `next_tid`를 통해 다음 스레드를 위한 tid를 저장한다. 또한 현재 스레드를 위한 tid를 결정한 이후 next_tid를 1 증가시킨다.
마지막으로 이렇게 `tid`를 할당하고 `next_tid`를 변형하는 작업을 수행하는 동안 다른 스레드에서 해당 함수를 실행하지 못하게 하기 위해 lock `tid_lock`을 사용한다. 왜냐하면 여러 스레드가 함께 함수를 호출하면 같은 `tid`가 할당될 가능성이 있기 때문이다.

Switch

switch thread에 대한 실질적인 코드는 대부분 `threads/switch.S`에 구현되어 있고 `threads/switch.h`는 이에 대한 interface만을 포함하고 있다.

`switch_threads_frame`

```
struct switch_threads_frame
  {
    uint32_t edi;               /*  0: Saved %edi. */
    uint32_t esi;               /*  4: Saved %esi. */
    uint32_t ebp;               /*  8: Saved %ebp. */
    uint32_t ebx;               /* 12: Saved %ebx. */
    void (*eip) (void);         /* 16: Return address. */
    struct thread *cur;         /* 20: switch_threads()'s CUR argument. */
    struct thread *next;        /* 24: switch_threads()'s NEXT argument. */
  };
```

함수 `switch_threads`을 수행하는데 사용될 stack frame이다. 해당 구조체 변수는 각 `thread`의 `stack`에 다른 stack frame 들과 함께 저장되어 있다. 또한 이는 `thread_create`에서 초기화된다.
`switch_thread`를 수행할 때 저장할 현재 스레드의 몇몇 레지스터 값, `switch_thread`에서 사용할 매개변수인 switch 전 스레드 포인터, switch 이후 스레드 포인터를 포함한다.

`switch_threads`

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

> 현재 실행 중인 스레드 `cur`에 대한 정보를 저장하고 `next`가 가리키는 스레드를 실행하도록 변경하고 관련 정보를 복구하는 함수

```
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi
```

`pushl`을 통해 현재 스레드의 `switch_threads_frame`의 위치에 4개의 레지스터 값들을 저장한다.

```
        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

현재 스레드의 스택 pointer를 stack에 저장한다. 이후 스위치 목표 스레드의 `thread`의 `stack` 값을 복구하고 복구한 스택 내 값들을 pop해 레지스터에 집어넣는다.

Kernel Main

`threads/init.c`의 `main`는 핀토스 실행시 가장 처음으로 실행되는 함수이자 핀토스 프로그램 그 자체이다. 다른 파일의 여러 함수들을 호출하여 커널의 초기화부터 command 실행까지 이루어지게 된다.

```
int
main (void)
{
  char **argv;

  /* Clear BSS. */
  bss_init ();

  /* Break command line into arguments and parse options. */
  argv = read_command_line ();
  argv = parse_options (argv);
```

`bss_init`을 통해 초기화되지 않은 non-local varaieble, BSS를 0으로 초기화한다. 이후 `read_command_line`을 통해 핀토스 실행과 함께 입력받은 arguments를 단어 단위로 잘라 그 문자열들의 리스트를 반환 받는다. 반환 받은 문자열 리스트를 인수로 하여 `parse_options`하면 입력받은 문자열 리스트를 순회하며 각 문자열에 맞는, 즉 각 옵션에 해당하는 행위를 수행한다. 주로 config 변수를 조절한다. 옵션이 아닌 첫번째 argument(커널이 어떤 task를 수행해야할지에 대한)를 반환한다.

```
  /* Initialize ourselves as a thread so we can use locks,
     then enable console locking. */
  thread_init ();
  console_init ();

  /* Greet user. */
  printf ("Pintos booting with %'"PRIu32" kB RAM...\n",
          init_ram_pages * PGSIZE / 1024);

  /* Initialize memory system. */
  palloc_init (user_page_limit);
  malloc_init ();
  paging_init ();
```

`thread_init`을 통해 스레드 시스템을 초기화한다. `thread_init`에서는 `allocate_tid`에 사용하는 lock인 `tid_lock`을 비롯해 ready한 스레드 리스트 `ready_list`, 모든 스레드 리스트 `all_list`를 초기화하고 main 함수를 실행 중인 현재 스레드를 `initial_thread`로 지정하고 이에 맞게 스레드 연관 상태를 초기화해준다.
이후에는 `palloc_init`, `malloc_init`, `paging_init`을 통해 메모리 시스템을 초기화해준다.

```
  /* Segmentation. */
#ifdef USERPROG
  tss_init ();
  gdt_init ();
#endif

  /* Initialize interrupt handlers. */
  intr_init ();
  timer_init ();
  kbd_init ();
  input_init ();
#ifdef USERPROG
  exception_init ();
  syscall_init ();
#endif
```

User Program을 위한 기본 설정을 진행한 뒤,
`intr_init()`을 통해 PIC와 IDT를 초기화하고 `timer_init`을 통해 PIT(Programmable Interrupt Timer) 설정 및 timer interrupt에 대한 핸들러를 등록한다. 이후 키보드 및 인풋 버퍼를 초기화한다.

```
  /* Start thread scheduler and enable interrupts. */
  thread_start ();
  serial_init_queue ();
  timer_calibrate ();
```

```
#ifdef FILESYS
  /* Initialize file system. */
  ide_init ();
  locate_block_devices ();
  filesys_init (format_filesys);
#endif
```

`thread_start`를 통해 `idle` 스레드를 생성 및 초기화하고 interrupt를 활성화함으로서 timer에 의해 스케줄링도 가능하게 한다. 이후 `serial_init_queue`를 이용해 interrupt-driven I/O를 위한 serial port를 초기화한다. 균일한 틱당 딜레이를 형성하기 위해 타이머 칼리브레이션을 진행하고 몇몇 파일 설정을 진행한다.

```
  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);

  /* Finish up. */
  shutdown ();
  thread_exit ();
}
```

앞서 `parse_options`에서 구한 non-option command 및 해당 command에 대한 argument를 이용해 해당 명령을 수행한다. `run action`에서 실행한 함수가 다른 스레드를 생성해 사용하기도 한다.
command로 입력받은 명령 수행이 완료되면 `thread_exit`을 통해 `main` 함수를 실행하던 스레드를 죽이고 이에 따라 커널은 종료된다.

Scheduler

```
/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.

   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
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);
}
```

`/threads/thread.c`에 정의된 `schedule` 함수는 현재 실행중인 `cur` 스레드와 다음으로 실행할 `next` 스레드를 `switch_threads`를 통해 컨텍스트 스위칭한다. 현재 스레드의 작업을 마무리하고 `ready_list`에서 새로운 스레드를 `Running` 상태로 만들어야 하는 `thread_yield`, `thread_block`, `thread_exit`에서 호출한다.

`switch_threads` 함수는 `switch.S`에 정의되어있는 x86 어셈블리로 작성된 함수로, 두 스레드의 레지스터와 스택 공간 정보를 뒤바꾼 뒤 바꾸기 이전 기존 스레드의 정보를 반환한다.

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

`schedule` 함수를 실행한 `cur` 스레드가 `switch_threads` 함수를 호출한 뒤 스레드 정보를 반환받는 것은 `next`로 컨텍스트가 넘어간 뒤 다시 `cur`로 스위칭 된 직후일 것이므로 `prev` 스레드 정보는 `cur`의 것과 동일하다.

```
/* Chooses and returns the next thread to be scheduled.  Should
   return a thread from the run queue, unless the run queue is
   empty.  (If the running thread can continue running, then it
   will be in the run queue.)  If the run queue is empty, return
   idle_thread. */
static struct thread *next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

다음에 실행할 스레드인 `next`를 선택할 때 호출되는 함수인 `next_thread_to_run`은 현재 `run queue`로 사용되고 있는 스레드 리스트에서 가장 앞에 있는 스레드 하나를 `pop`해 반환한다.

```
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
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)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

```
/* Transitions a blocked thread T to the ready-to-run state.
   This is an error if T is not blocked.  (Use thread_yield() to
   make the running thread ready.)

   This function does not preempt the running thread.  This can
   be important: if the caller had disabled interrupts itself,
   it may expect that it can atomically unblock a thread and
   update other data. */
void thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

또한 현재 스레드를 `run queue`에 넣는 작업이 필요한 `thread_yield`, `thread_unblock` 등과 같은 경우에는 일괄적으로 `list_push_back (&ready_list, &t->elem);`를 이용하여 `run queue` 리스트의 뒤쪽 끝에다 `push`하고 있다. 위의 `pop` 동작과 연관지어 봤을 때 현재 `run queue`는 우선순위 없이 단일 큐를 이용해 먼저 `push`된 스레드가 먼저 `pop`되는 `round-robin` 형식을 사용하고 있는 것을 알 수 있다. 이를 우선순위가 높은 스레드가 먼저 `pop`되도록 `priority scheduler`로 변경하는 것이 이번 Assn1의 목표 중 하나이다.`

Synchronization Variables

멀티스레드 환경에서 Concurrency를 보장하기 위해 필요한 Semaphores, Locks, Condition Variables 구조체에 대한 정의는 `/threads/synch.h`, `/threads/synch.c`에 정의되어 있다.

Semaphore

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

현재 semaphore의 값과 후술할 `sema_down`에 의해 Block된 스레드들을 관리하는 리스트를 멤버 변수로 가진다.

```
void sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

  sema->value = value;
  list_init (&sema->waiters);
}
```

`semaphore` 구조체를 선언한 뒤 `sema_init` 함수를 통해 현재 semaphore의 값과 스레드 리스트를 초기화해줘야 한다.

```
void sema_down (struct semaphore *sema)
{
  while (sema->value == 0) wait;
  sema->value--;
}
```

```
void sema_up (struct semaphore *sema)
{
  sema->value++;
}
```

`sema_down`, `sema_up`의 정의에 따른 의사 코드는 위와 같다. 하지만 `sema_down`을 위와 같이 구현할 시 `sema->value`가 0이 되기 전까지 루프를 도는 Busy Waiting 상태가 된다.

```
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)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
```

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

따라서 본 코드에선 `sema_down`을 시도하는 스레드를 Block하고 `semaphore` 구조체 내부의 스레드 리스트에서 Block된 스레드들을 관리한다. 이 스레드들은 `sema_up`이 호출될 시 일괄적으로 Unblock된다.

OS에서 인터럽트 처리 중 `sema_down`에 의한 스레드 Block이 발생할 경우 Race Condition에 의해 OS가 멈추는 등 예기치 못한 결과를 일으킬 수 있으므로 인터럽트 중에는 `sema_down`이 실행되지 않도록 `ASSERT`문을 통해 검사한다. 또한 유사한 이유로 스레드 리스트를 변경하는 중 인터럽트가 발생하여 예기치 못한 동작이 일어날 가능성이 존재하므로 스레드 리스트를 변경하는 도중에는 인터럽트를 비활성화시켜 Atomic하게 실행되도록 한다.

```
bool sema_try_down (struct semaphore *sema)
{
  enum intr_level old_level;
  bool success;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (sema->value > 0)
    {
      sema->value--;
      success = true;
    }
  else
    success = false;
  intr_set_level (old_level);

  return success;
}
```

`sema_try_down` 함수는 `sema->value`가 0보다 커질 때까지 대기하는 대신 `down`을 1회 시도하고 성공/실패 여부를 반환한다. 따라서 실행 중 스레드가 Block되지 않는다.

Lock

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

Lock은 기본적으로 Semaphore를 기반으로 하되, Lock을 설정한 스레드와 해제하는 스레드가 동일함을 추가적으로 검증한다. 따라서 멤버 변수에 현재 해당 lock을 설정한 스레드가 무엇인지를 저장하는 `holder` 멤버 변수가 추가적으로 선언되었다.

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

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);
}
```

Lock을 초기화하는 함수. 현재 Lock을 소유 중인 `holder`가 없는 상태이고 Semaphore의 값을 1인 상태로 설정한다.

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

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

Lock을 획득하는 함수. 앞서 설명한 인터럽트 여부 확인과 더불어 현재 Lock을 보유 중인 스레드가 다시 Lock 획득을 시도하고 있지는 않은지 검사한 뒤 `sema_down` 을 호출한다. 이미 해당 Lock을 다른 스레드가 소유 중일 경우 Semaphore의 값이 0일 것이므로 Semaphore가 1이 될 때까지 Block될 것이고, 소유 중이었던 스레드가 Lock 소유를 해제할 경우 현재 스레드가 Unblock되어 Lock 소유를 시도할 것이다. 소유에 성공했거나 기존 소유자가 없었을 경우 lock의 holder를 현재 스레드로 변경한다.

```
bool lock_try_acquire (struct lock *lock)
{
  bool success;

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

  success = sema_try_down (&lock->semaphore);
  if (success)
    lock->holder = thread_current ();
  return success;
}
```

`sema_try_down` 과 마찬가지로 Lock 소유를 1회 시도한 뒤 성공 여부를 반환한다.

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

  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

Lock 소유를 해제한다. `lock_acquire` 와는 반대로, 해제를 시도하고 있는 스레드가 Lock을 소유했던 스레드가 맞는지를 검사한다. `lock->holder` 를 초기화한 뒤 `sema_up` 으로 Semaphore의 값을 다시 1로 복구한다.

Condition variable

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

`waiters` 리스트는 `wait` 하려는 스레드가 생성한 Semaphore들을 저장한다.

```
void cond_init (struct condition *cond)
{
  ASSERT (cond != NULL);

  list_init (&cond->waiters);
}
```

`waiters` 리스트를 초기화함으로써 condition 객체를 초기화한다.

```
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);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
```

```
  lock_acquire (lock);
}
```

Conditional variable에 의해 스레드를 wait하는 동작을 정의한다. Semaphore 하나를 추가로 정의하여 Conditional variable의 리스트에 저장한 뒤, Lock을 해제한 후 (호출 시 해당 스레드가 Lock을 소유하고 있어야 한다) `sema_down` 을 시도한다. 해당 Semaphore는 다른 스레드에서 `cond_signal` 혹은 `cond_broadcast` 를 호출하여 `sema_up` 되기 전까지 `down` 상태를 유지할 것이므로 `cond_wait` 을 호출한 본 스레드는 `wait` 상태에 머무르게 된다. `sema_down` 이 완료된 후에는 다시 Lock을 획득한다. Lock 관련 동작을 수행해야 하므로 해당 동작 역시 인터럽트 핸들 로직 중 수행되지 않도록 `ASSERT` 문으로 확인해준다.

```
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))
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
}
```

해당 스레드가 Lock을 보유하고 있고 인터럽트 중이 아닐 경우, `cond` 의 Semaphore list중 가장 앞의 Semaphore를 up하여 해당 Conditional variable에 의해 `wait` 상태에 있던 스레드를 깨운다.

```
void cond_broadcast (struct condition *cond, struct lock *lock)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);

  while (!list_empty (&cond->waiters))
    cond_signal (cond, lock);
}
```

해당 스레드가 Lock을 보유하고 있고 인터럽트 중이 아닐 경우, `cond` 의 Semaphore list에 있는 모든 Semaphore를 up하여 해당 Conditional variable에 의해 `wait` 상태에 있던 스레드들을 모두 깨운다.

Timer

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

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

기존 `timer_sleep` 함수의 구현 방식은 함수가 호출된 직후부터 `ticks` 틱이 경과할 때까지 스레드를 실행 대기중인 스레드 리스트로 되돌리는 `thread_yield` 를 반복 호출한다. 이로 인해 스레드를 sleep시키는 함수임에도 불구하고 CPU가 쉬지 않고 작동하며 sleep 만료 여부를 확인하는 비효율적인 `busy waiting` 방식으로 구현되어있어, 이를 `thread_block` 을 이용한 `sleep-awake` 방식의 보다 효율적인 방법으로 개선하는 것이 Assn1의 목표 중 하나이다.

```
/* Number of timer ticks since OS booted. */
static int64_t ticks;
```

```
/* Returns the number of timer ticks elapsed since THEN, which
   should be a value once returned by timer_ticks(). */
int64_t timer_elapsed (int64_t then)
{
  return timer_ticks () - then;
}
```

```c
/* Returns the number of timer ticks since the OS booted. */
int64_t timer_ticks (void)
{
  enum intr_level old_level = intr_disable ();
  int64_t t = ticks;
  intr_set_level (old_level);
  return t;
}
```

Timer는 OS 부팅 시부터 `ticks` 전역 변수를 통해 현재까지 흐른 시간을 갱신하여 관리하며, 이를 이용해 `timer_ticks`를 첫 번째 호출한 시점과 두 번째 호출한 시점의 `ticks` 값의 차이를 계산하여 경과 시간을 알아내는 등의 동작을 구현할 수 있다.

Design

Alarm Clock

`busy waiting` 방식의 `timer_sleep`을 개선하기 위해선 스레드를 block시킨 뒤 매 틱마다 `sleep`이 만료되었는지 확인해주어야 한다. 이를 위해 다음과 같이 구현 계획을 세웠다.

- `thread` 구조체에 `sleep`이 해제되어야 할 시각인 `wakeup_tick` 변수를 멤버로 추가한다.
- `sleep_list` 리스트를 `thread.c`에 선언한 뒤 `thread_init`에서 `list_init(&sleep_list)`를 통해 초기화해준다.
- `timer_sleep` 함수가 호출될 시, 현재 스레드 구조체(`thread_current`)를 `sleep_list`에 추가한 뒤 현재 스레드를 `block`한다.
  - 인터럽트 동작 중 스레드를 `block`하는 동작은 race condition을 일으킬 수 있으므로 인터럽트를 해제해줘야 한다.
- 매 틱마다 `timer.c`의 `timer_interrupt` 혹은 `thread.c`의 `thread_tick` 함수가 호출될 때마다 `sleep_list`를 순회하며, 현재 틱이 `wakeup_tick`을 지난 스레드들을 `unblock`시키고 `ready_list` 리스트로 이동시킨다.
- `sleep_list`가 정렬된 상태로 관리될 경우 `sleep`이 만료된 스레드를 탐색할 시 전체 순회를 하는 것이 아닌 리스트의 앞쪽만 확인하면 되므로, 삽입에 `O(n)`, 삭제에 `O(1)`의 시간복잡도로 개선이 가능하다. 해당 동작은 `list`의 `list_insert_ordered`와 `list_pop_front`를 이용해 구현 가능하다.
  - `list_insert_ordered`에서 아이템들 간 대소비교를 판단할 때 `list_less_func` 비교함수를 정의해줘야 하는데, 여기서는 각 스레드들의 `wakeup_tick` 간의 비교가 이루어지도록 비교함수를 정의해줘야 한다.

Priority Scheduling

현재 구현되어있는 `round-robin scheduler`는 스레드의 우선순위를 고려하지 않고 `ready_list`에 들어간 순서대로 스케줄되고 있으므로 이를 `Priority Scheduler`로 개선해야 한다. 고려해야 하는 상황은 크게 다음과 같다.

- 스레드는 언제든 우선순위가 변경될 수 있으며, 만약 실행 중인 스레드의 우선순위가 변경되어 `ready_list`에 들어있는 스레드들 중 가장 높은 우선순위보다 낮아질 시 곧바로 `thread_yield`를 해줘야 한다.
- `Semaphore`, `Lock`, `Condition Variable`에 의해 관리되는 공유자원에 대한 접근 역시 우선순위에 따라 관리되어야 한다.
- 낮은 우선순위의 스레드 `L`이 소유 중인 `lock`을 높은 우선순위의 스레드 `H`가 요청한 상황에서 `L`보다 우선순위가 높은 다른 스레드 `M`에게 우선순위에서 밀려 `lock`을 전해주지 못하는 `Priority Inversion` 현상이 발생할 수 있다. 이는 `H`가 `L`에게 `lock`을 요청할 때 일시적으로 자신의 우선순위를 양도해주는 `Priority Donation`을 구현함으로써 해결할 수 있다.
  - 낮은 우선순위의 스레드가 여러 개의 스레드에게서 우선순위를 양도받을 수 있다(`Multiple Donation`). 이때 양도받은 스레드는 양도받은 우선순위 중 가장 높은 것을 적용한다.
  - `Priority Donation`이 연쇄적으로 일어날 수 있다(`Nested Donation`). 이때 연쇄적으로 우선순위를 양도받은 스레드들은 모두 가장 높은 `Prioriry Donor`의 우선순위를 받는다.

이를 위해 다음과 같이 구현 계획을 세웠다.

- 실행 중인 스레드의 우선순위가 대기 중인 스레드의 우선순위와 역전될 수 있는 `thread_create`와 `thread_set_priority` 함수의 마지막에 현재 스레드를 우선순위에 따라 `thread_yield`할 것인지 판단하는 로직이 추가되어야 한다.
- `ready_list` 역시 `sleep_list`와 유사하게 스레드 우선순위를 기준으로 내림차순 정렬된 상태로 관리되어야 한다. 이를 위해 새로운 `list_less_func`을 정의하여 스레드의 `priority`에 따른 비교 함수를 작성한 뒤, 기존에 `ready_list`에 스레드를 추가하던 `thread_unblock`, `thread_yield`의 `list_push_back` 함수 호출을 `list_insert_ordered`로 변경한다.
- `Semaphore`와 `Condition Variable`의 스레드 관리 로직을 변경한다. `Lock`의 경우 `Semaphore`의 로직 외에 스레드 우선순위가 관여하는 부분이 없으므로 변경하지 않는다.
  - `Semaphore`와 `Condition Variable`의 `waiters` 리스트 역시 위와 동일한 방법으로 정렬을 유지한다. 삽입이 일어나는 `sema_down`과 `cond_wait`에서 `list_insert_ordered`를 호출하여 우선순위 정렬을 유지해주고, 삭제가 일어나는 `sema_up`과 `cond_signal`에서 `list_pop_front`하여 우선순위가 가장 높은 데이터를 반환시킨다.

    - 다만 공유자원이 다른 스레드에 의해 점유되어 대기하고 있는 동안 `Priority Donation` 등의 이유로 대기하고 있던 스레드들의 우선순위가 변경될 가능성이 존재한다. 때문에 `sema_up`과 `cond_signal`에서 `list_pop_front`를 하기 전 리스트를 정렬할 필요가 있다.
    - `cond`는 리스트의 원소로 스레드가 아닌 `semaphore_elem`을 사용하기 때문에 이를 비교하는 비교함수를 추가로 정의해줘야 한다.
- `lock`을 요청하고 해제할 때 일어나는 `Priority Donation`과 관련된 동작을 구현한다.
    - `lock_acquire`를 호출했을 때 대상이 되는 `lock`을 이미 다른 스레드가 소유하고 있을 경우 자신의 우선순위를 해당 스레드에게 빌려줘야 하고, 재귀적으로 우선순위 전파가 이루어져야 한다. 이를 구현하기 위해 현재 스레드가 소유 요청을 한 `lock`에 대한 포인터인 `lock_to_acquire`, 그리고 현재 스레드가 `Priority Donation`을 받은 스레드들의 리스트인 `donors`를 `thread` 구조체에 추가하고 `lock_acquire`에서 업데이트 해준다. 그 뒤

      ```
      thread_current()->lock_to_acquire->holder->priority
      = thread_current()->priority;
      ```

      와 같이 우선순위를 `donate`한다. 현재 보고 있는 스레드를 `thread_current()->lock_to_acquire->holder`로 교체하며 사전 정의된 최대 전파 깊이 `(~8)` 단계만큼 전파되거나 `lock_to_acquire == NULL`일 때까지 반복하면 `Nested Donation`을 구현할 수 있다.
    - `lock_release`가 호출될 경우 스레드는 소유중이던 `lock`을 반환함과 동시에 해당 `lock`을 이유로 `donate` 받았던 우선순위도 되돌려야 한다. 자신에게 우선순위를 `donate`해준 스레드의 리스트인 `donors`를 순회하며 해당 `lock`을 요청했던 스레드를 리스트에서 삭제한 뒤, 남아있는 스레드들의 우선순위 중 최댓값을 사용한다. 만약 `donors` 리스트가 비었거나 본 스레드의 우선순위가 바뀌어 `donate` 받은 우선순위보다 높아질 시 기존 우선순위를 사용한다.
    - `Priority Donation`이 일어날 시 `ready_list`의 정렬 상태가 깨질 가능성이 있으므로 호출 시마다 추가적인 정렬이 필요하다.

4.4BSD Scheduler

실행 시 `cmd` 옵션에서 `-mlfqs` 옵션을 사용 시 기존 스케줄러 대신 `4.4BSD Scheduler`를 사용한다. 해당 스케줄러는 실시간으로 각 스레드마다 CPU 사용 시간 등을 계산하여 우선순위를 계산한 뒤 이를 바탕으로 스케줄링을 수행한다. 해당 스케줄러는 상술한 `Priority Donation`을 사용하지 않으므로 `-mlfqs` 플래그 여부에 따라 기존에 구현한 `Priority Donation` 등의 기능들을 비활성화시키는 로직이 필요하다.

`thread` 구조체에 추가해야 할 멤버 변수들에 대한 정보는 다음과 같다.

Variable	Type	Range	Default	Calculation	Refresh rate
`nice` (ness)	`int`	-20 ~ 20	0	`thread_set_nice()`	manual
`priority`	`int`	`PRI_MIN` (0) ~ `PRI_MAX` (63)	`PRI_DEFAULT` (31)	`PRI_MAX - (recent_cpu / 4) - (nice * 2)`	4 Ticks
`recent_cpu`	`fixed-point`	any	0	`(2*load_avg)/(2*load_avg + 1) * recent_cpu + nice`	1 sec
`load_avg`	`fixed-point`	0 ~	0	`(59/60)*load_avg + (1/60)*ready_threads`	1 sec

`nice`는 다른 스레드에게 얼마나 CPU 자원 양도를 잘 하는지에 대한 정도, `recent_cpu`는 해당 스레드가 최근 사용한 CPU 시간, `load_avg`는 최근 1분간 `ready`와 `running` 중인 스레드의 수의 평균을 나타낸다.

4틱마다 `priority`를 갱신해주고, 1초마다 `recent_cpu`와 `load_avg`를 재계산해주는 로직은 `timer_interrupt` 함수를 수정하여 구현 가능한데, OS 부팅 시부터 소요된 틱을 카운트하는 `ticks`가 4의 배수, 혹은 `TIMER_FREQ`의 배수가 될 때마다 특정 로직을 수행하도록 구현할 수 있다. 또한 매 틱마다 현재 실행중인 스레드의 `recent_cpu`를 1 증가시키는 로직도 포함되어야 한다. 해당 연산들은 스레드의 우선순위와 스케줄링에 직접적인 영향을 미치므로 인터럽트를 비활성화하여 atomic하게 실행되도록 해야 한다.

현재 `thread.h`에 advanced scheduler와 관련된 함수가 4개 정의되어 있지만 구현은 되어 있지 않다. 구현해야 할 기능은 다음과 같다.

- `thread_get_nice` : 현재 스레드의 `nice` 값을 반환한다.
- `thread_set_nice` : 현재 스레드의 `nice` 값을 파라미터의 새 값으로 적용한 후 이를 바탕으로 `priority`를 갱신한다. 갱신 후 스레드의 우선순위가 다른 스레드보다 낮아졌을 시 `thread_yield`한다.
- `thread_get_recent_cpu` : 현재 스레드의 `recent_cpu`에 100배 한 값을 반올림하여 반환한다.
- `thread_get_load_avg` : 현재 스레드의 `load_avg`에 100배 한 값을 반올림하여 반환한다.

Fixed-Point Real Arithmetic

`load_avg`와 `recent_cpu`는 실수 값을 가지는데, 일반적으로 사용되는 floating-point number는 연산이 느려 커널의 성능에 악영향을 줄 수 있기 때문에 고정 소수점 표기 방식을 이용한다. fixed-point number는 다음과 같이 표현된다.

```
0 00000000000000000 00000000000000
| |    Decimal    | | Fractional |
| +---(17 bits)---+ +-(14 bits)--+
sign
```

이를 바탕으로 Fixed-Point number 간의 연산을 정의할 수 있다.

Function	Expression
`f`	`1 << 14`
int to fp	`n * f`
fp to int (rounding toward zero)	`x / f`
fp to int (rounding to nearest)	`(x + f / 2) / f` (x >= 0), `(x - f / 2) / f` (x <= 0)
add fp, fp	`x + y`
sub fp, fp	`x - y`
add fp, int	`x + n * f`
sub fp, int	`x - n * f`
mul fp, fp	`((int64_t) x) * y / f`
mul fp, int	`x * n`
div fp, fp	`((int64_t) x) * f / y`
div fp, int	`x / n`