thread control block 내부적으로 바꾸는 거임. 실제로 thread가 바뀌어서 실행되는 거랑 다르다.

thread_unblock.

interrupt handler 안에서 호출할 수 있고 없고 차이?

언제 interrupt disable 하고 enable 해야하는지 모르겠다.

- sema_down 함수.

Certainly. The function `sema_down` is designed to operate on a semaphore, which is a synchronization primitive used in concurrent programming to control access to a shared resource.

Here's a breakdown of why this function cannot be called within an interrupt handler:

1. **May Sleep**: The function uses `thread_block()` which puts the calling thread to sleep if the semaphore's value is zero. This is unacceptable behavior in an interrupt context because interrupt handlers are supposed to execute quickly and must not sleep.
2. **Context Switching**: The function could lead to a context switch, which is also not acceptable within an interrupt handler. Interrupts should be processed quickly and shouldn't involve complex operations like context switching.
3. **Interrupt Disabling**: The function uses `intr_disable()` and `intr_set_level()` functions to disable and later restore the interrupt level. Manipulating interrupt levels within an interrupt handler could lead to unexpected behavior.
4. **Resource Contention**: If called within an interrupt handler, you might end up in a situation where the interrupt handler is waiting for a resource that it itself has locked, leading to deadlock.

To sum up, `sema_down` involves operations that are too slow or complicated for an interrupt context, which requires quick, predictable execution.

- sema_up 함수.

but this function calls intr_disable function but it can be called in interrupt handler

Great observation! Both `sema_down` and `sema_up` use `intr_disable()` to disable interrupts temporarily, but there is a key difference in why `sema_up` can be used within an interrupt handler while `sema_down` cannot.

Why `sema_up` Can Be Called from an Interrupt Handler

1. **No Sleeping**: Unlike `sema_down`, the `sema_up` function does not put the calling thread to sleep. Therefore, it can execute quickly, meeting the requirements of an interrupt context.
2. **No Waiting**: `sema_up` is generally non-blocking. It increments the semaphore's value and potentially unblocks a waiting thread. This is much quicker than waiting for a resource to become available, as is the case with `sema_down`.
3. **Quick Execution**: The operations in `sema_up` are quick and don't involve any complex operations like looping until a condition is met, which would be unacceptable in an interrupt context.
4. **Controlled Behavior**: The disabling of interrupts in `sema_up` is temporary and quickly restored by `intr_set_level(old_level)`. This ensures that the interrupt mask level is reset promptly, making the function suitable for use within an interrupt handler.

In summary, even though `sema_up` uses `intr_disable()`, the nature of its operations is such that they are acceptable within the context of an interrupt handler. The operation is quick, non-blocking, and does not involve context switching or sleeping, which are typically the main concerns when executing code in an interrupt context.

Rege

init.c / main

thread_init()

thread_start
— idle thread 생성

```c
/* Initial thread, the thread running init.c:main(). */
static struct thread *initial_thread;

/* Lock used by allocate_tid(). */
static struct lock tid_lock;
```

```c
/* Initializes the threading system by transforming the code
   that's currently running into a thread.  This can't work in
   general and it is possible in this case only because loader.S
   was careful to put the bottom of the stack at a page boundary.

   Also initializes the run queue and the tid lock.

   After calling this function, be sure to initialize the page
   allocator before trying to create any threads with
   thread_create().

   It is not safe to call thread_current() until this function
   finishes. */
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 ();
}
```

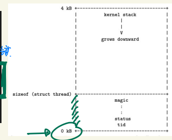
pintos loader가 일렬해놓음.

```c
/* Returns the current interrupt status. */
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;
}
```

요약: 최초의 thread가 생긴다. - 이름: main
이미 돌고 있던 main thread로 취급한다.

```c
/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

할당을 페이지 단위로 하니까 가능.




```c
/* Does basic initialization of T as a blocked thread named
   NAME. */
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);
}
```

문자열 복사. / 왜 여기서 disable?

값 설정.

how: pintos의 initial thread를 위한 struct thread를 만들어서
what: thread system을 initialize한다.
비고: (pintos loader가 initial thread의 stack을 page의 top에 설정해둔다.
그래서 다른 thread처럼 initial thread 생성 가능.
- 처음은 특이해. (thread를 만드는데 thread_create를 안쓰고 아니 못써) 처음이라서.

```c
/* Returns a tid to use for a new thread. */
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;
}
```

전역변수로 되어있다.
기본적으로 lock은 전역변수로 처리하는듯.

lock_init / thread_init

```c
/* Initializes LOCK.  A lock can be held by at most a single
   thread at any given time.  Our locks are not "recursive", that
   is, it is an error for the thread currently holding a lock to
   try to acquire that lock.

   A lock is a specialization of a semaphore with an initial
   value of 1.  The difference between a lock and such a
   semaphore is twofold.  First, a semaphore can have a value
   greater than 1, but a lock can only be owned by a single
   thread at a time.  Second, a semaphore does not have an owner,
   meaning that one thread can "down" the semaphore and then
   another one "up" it, but with a lock the same thread must both
   acquire and release it.  When these restrictions prove
   onerous, it's a good sign that a semaphore should be used,
   instead of a lock. */
void
lock_init (struct lock *lock)
{
  ASSERT (lock != NULL);

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

이미 가진 lock도 acquire 하면 안된다.

2. 소유의 개념이 없다.
- lock은 up한 애가 down
  ,, down ,, up 가능.
- sema는 아무나 up, down 모두가능.

```c
/* Initializes semaphore SEMA to VALUE.  A semaphore is a
   nonnegative integer along with two atomic operators for
   manipulating it:

   - down or "P": wait for the value to become positive, then
     decrement it.

   - up or "V": increment the value (and wake up one waiting
     thread, if any). */
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

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

```c
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
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);
}
```

이건 interrupt handler에서 부르면 이상해지나?
얘 때문에 interrupt handler에서 못부름.

```c
/* Acquires LOCK, sleeping until it becomes available if
   necessary.  The lock must not already be held by the current
   thread.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but interrupts will be turned back on if
   we need to sleep. */
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 ();
}
```

semaphore와의 차이.

```c
/* Releases LOCK, which must be owned by the current thread.

   An interrupt handler cannot acquire a lock, so it does not
   make sense to try to release a lock within an interrupt
   handler. */
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

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

```c
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.

   This function may be called from an interrupt handler. */
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);
}
```

```c
/* 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);
}
```

thread_start()

"idle" thread를 create한다.
인터럽트를 enable 한다.

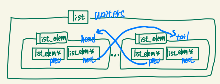
```c
/* Starts preemptive thread scheduling by enabling interrupts.
   Also creates the idle thread. */
void
thread_start (void)
{
  /* Create the idle thread. */
  struct semaphore idle_started;
  sema_init (&idle_started, 0);
  thread_create ("idle", PRI_MIN, idle, &idle_started);

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

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

thread_start 함수가 끝나면 어떻게 되는거지? 어떻게?
여기 sema_up될까

요약: idle thread를 만들어 준다.
what: Schedular를 시작시킴 : ~~idle thread가 schedule 되려면~~
how : idle thread를 만듦 — 얘는 다른 ready thread 없으면 실행됨.
what: interrupts를 enable한다
— ?

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

semaphore
- value : 숫자
- waiters : 숫자를 노리는 thread 들




```c
/* Initializes semaphore SEMA to VALUE.  A semaphore is a
   nonnegative integer along with two atomic operators for
   manipulating it:

   - down or "P": wait for the value to become positive, then
     decrement it.

   - up or "V": increment the value (and wake up one waiting
     thread, if any). */
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

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

```c
/* Initializes LIST as an empty list. */
void
list_init (struct list *list)
{
  ASSERT (list != NULL);
  list->head.prev = NULL;
  list->head.next = &list->tail;
  list->tail.prev = &list->head;
  list->tail.next = NULL;
}
```

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

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

```c
/* Creates a new kernel thread named NAME with the given initial
   PRIORITY, which executes FUNCTION passing AUX as the argument,
   and adds it to the ready queue.  Returns the thread identifier
   for the new thread, or TID_ERROR if creation fails.

   If thread_start() has been called, then the new thread may be
   scheduled before thread_create() returns.  It could even exit
   before thread_create() returns.  Contrariwise, the original
   thread may run for any amount of time before the new thread is
   scheduled.  Use a semaphore or some other form of
   synchronization if you need to ensure ordering.

   The code provided sets the new thread's `priority' member to
   PRIORITY, but no actual priority scheduling is implemented.
   Priority scheduling is the goal of Problem 1-3. */
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  printf("==== thread_create called: %s %d ====\n", name, priority);
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);

  /* 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 ();

  /* 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;
}
```

thread는 함수실행용이지 그래서...
create 도중에 thread가 바뀔수도
kernel pool의
idle
위에서함 → thread 기본정보로 struct thread를 알차게 만든다.
BLOCKED.

① 스택을 만들기위한 page를 할당
② thread 기본정보로 struct thread를 initialize한다.
③ fake stack frame을 setup한다.
— why fake stack frame?
thread를 처음시작(create) 할때를 위함.
switch_entry → kernel_thread → idle 순으로 불림.
④ thread 실행할 준비.

switch_thread에서 실행함

Stack 만들어 둔걸 언제 실행하는지 결정하는 역할을 ready 상태에 넣어서 넘겨줌.

Stack frame 내부 변수가 지금 그림이랑 반대인것 같기도

처음 그림임.

Switch_entry에서 push %eax 왜하지?

idle 맨뒤의 asm은 실행될 thread 없을때 실행되나? idle thread가 실행 될때?

③ 을 위한 준비.

```c
/* Stack frame for kernel_thread(). */
struct kernel_thread_frame
  {
    void *eip;                  /* Return address. */
    thread_func *function;      /* Function to call. */
    void *aux;                  /* Auxiliary data for function. */
  };
```

```c
/* Stack frame for switch_entry(). */
struct switch_entry_frame
  {
    void (*eip) (void);
  };
```

What is an EIP register?
EIP is a register in x86 architectures (32bit). It holds the "Extended Instruction Pointer" for the stack. In other words, it tells the computer where to go next to execute the next command and controls the flow of a program. 2022. 9. 23.

```c
/* switch_thread()'s stack 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. */
  };
```

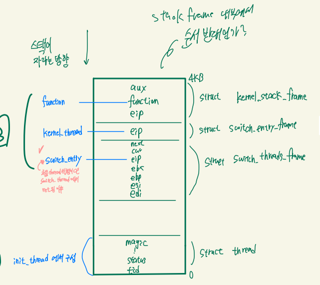
```c
/* Allocates a SIZE-byte frame at the top of thread T's stack and
   returns a pointer to the frame's base. */
static void *
alloc_frame (struct thread *t, size_t size)
{
  /* Stack data is always allocated in word-size units. */
  ASSERT (is_thread (t));
  ASSERT (size % sizeof (uint32_t) == 0);

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

```c
/* Function used as the basis for a 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. */
}
```

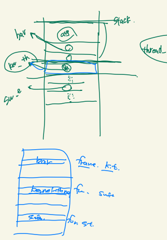
```asm
.globl switch_entry
.func switch_entry
switch_entry:
	# Discard switch_threads() arguments.
	addl $8, %esp

	# Call thread_schedule_tail(prev).
	pushl %eax
.globl thread_schedule_tail
	call thread_schedule_tail
	addl $4, %esp

	# Start thread proper.
	ret
.endfunc
```

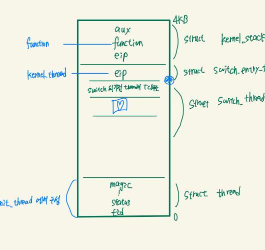
※ esp register는 stack의 꼭대기를 가리킴.

스택이 자라는 방향
stack frame 내부에서 순서 반대인가?



There is a special case when a newly created thread runs for the first time. Such a thread calls intr_enable() as the first action in kernel_thread(), which is at the bottom of the call stack for every kernel thread but the first.






```c
/* 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);
}
```

READY.

```c
/* Function used as the basis for a 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. */
}
```

무조건 불려짐 (왜?) 꼭 무엇.

```c
/* Enables interrupts and returns the previous interrupt status. */
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;
}
```

→ 인터럽트 활성화

```c
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
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);
}
```

external interrupt 실행아니어야함.

이전 level로.

이거 call하는 애가 main thread이다 block하면 바로 idle이 (처음) 실행된다.

sema→value 0

- 아까 thread_create에서 만든 idle 스레드를 ready list에 넣어줘서 ok.

main thread는 BLOCKED 되고 sema→waiters...

```c
/* Disables interrupts and returns the previous interrupt status. */
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;
}
```

— ON/OFF

이것까지 한번에 실행하려고 인터럽트 껐다.

sema 값이 0임 → (sema→waiter)에 지금 thread 등록. 그 후에 thread block (다른 thread가 돌겠임)

sema 값이 0아님 → 값을 1만큼 감소

```c
/* Idle thread.  Executes when no other thread is ready to run.

   The idle thread is initially put on the ready list by
   thread_start().  It will be scheduled once initially, at which
   point it initializes idle_thread, "up"s the semaphore passed
   to it to enable thread_start() to continue, and immediately
   blocks.  After that, the idle thread never appears in the
   ready list.  It is returned by next_thread_to_run() as a
   special case when the ready list is empty. */
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 ();

      /* Re-enable interrupts and wait for the next one.

         The `sti' instruction disables interrupts until the
         completion of the next instruction, so these two
         instructions are executed atomically.  This atomicity is
         important; otherwise, an interrupt could be handled
         between re-enabling interrupts and waiting for the next
         one to occur, wasting as much as one clock tick worth of
         time.

         See [IA32-v2a] "HLT", [IA32-v2b] "STI", and [IA32-v3a]
         7.11.1 "HLT Instruction". */
      asm volatile ("sti; hlt" : : : "memory");
    }
}
```

idle thread 만들때 만들었던 semaphore.

→ idle.

idle thread가 실행 → thread_block → 이후에 main thread 실행. idle thread는 BLOCKED.

└ intr_enable()에 씀.

```c
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.

   This function may be called from an interrupt handler. */
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);
}
```

이거 실행할때가 idle thread

1 아직 idle.

main thread를 READY로 만든다.

```c
/* Optimization barrier.

   The compiler will not reorder operations across an
   optimization barrier.  See "Optimization Barriers" in the
   reference guide for more information.*/
#define barrier() asm volatile ("" : : : "memory")
```

synch.h에서도 같은거씀.

```c
/* Puts the current thread to sleep.  It will not be scheduled
   again until awoken by thread_unblock().

   This function must be called with interrupts turned off.  It
   is usually a better idea to use one of the synchronization
   primitives in synch.h. */
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

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

이렇게 BLOCKED 하고 나중에 우선순위 낮은 것 순으로 깨워주는 건가. - 언젠가 next thread to run에서 리턴되겠.

```c
/* Returns the running thread.
   This is running_thread() plus a couple of sanity checks.
   See the big comment at the top of thread.h for details. */
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;
}
```

```c
/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
```

```c
/* Returns true if T appears to point to a valid thread. */
static bool
is_thread (struct thread *t)
{
  return t != NULL && t->magic == THREAD_MAGIC;
}
```

```c
/* 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);
}
```

Cur == next → thread_ticks 초기화 만된다.

Cur != next → (아마도 높은확률로) 지금 돌고있던 thread가 멈추고 우선순위에 의해 다음 thread가 실행. ☆ 얘는 다음에 돌때 여기서부터 시작한다.

※ TCB는 thread 관리를 위해 정보 기록해주는 곳이지 TCB에 정보가 기록되었다고 thread가 그에 맞게 동작하진X (status가 BLOCKED라고 멈추는게 아니다. 관리 처리는 따로한다.)

- thread block에서 BLOCKED되는데 실제로 멈추는건 switch_threads. (status가)
- ☆에서 실행되는데 status는 thread_schedule_tail 내부에서 RUNNING.

※ thread의 switch는 결국 esp를 해당 thread의 stack에 가져다 놓고 ret으로 ☆가 원래 돌던 형태로 가는것을 의미하는듯.

```c
/* 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);
}
```

명확X임.

```c
/* Converts pointer to list element LIST_ELEM into a pointer to
   the structure that LIST_ELEM is embedded inside.  Supply the
   name of the outer structure STRUCT and the member name MEMBER
   of the list element.  See the big comment at the top of the
   file for an example. */
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

```c
/* Offset of `stack' member within `struct thread'.
   Used by switch.S, which can't figure it out on its own. */
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

```asm
.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

	# 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
```

Caller save reg.

pushl: '현재' CPU에 있는 reg 저장 / 현재 thread의 stack

edx: struct thread에서 'stack'이 있는 부분의 offset

eax: [esp+20]: 현재 thread 구조체 주소. [eax+edx]에 esp 저장 (현재 thread의 stack주소)

ecx: [esp+24]: 새로운 thread 구조체 주소. esp에 [ecx+edx] 저장.

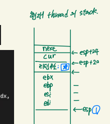
popl: '새로운' thread가 돌고있을때 저장해뒀던 값들 복구

ret → eip로 간다. / eax에 있는 값은 리턴값: 현재 thread 구조체 주소.

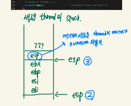
'현재': switch threads가 실행될때 돌고 있는 thread

'새로운': switch 되어 새로 돈 thread

현재 thread의 stack.



새로운 thread의 stack.



⓪ register값 스택에 저장
① 현재 CPU의 stack pointer를 현재 struct thread의 'stack' member에 저장
② 새로운 thread의 'stack'을 복구
④ 그다음 register 값 복구.

```c
/* Completes a thread switch by activating the new thread's page
   tables, and, if the previous thread is dying, destroying it.

   At this function's invocation, we just switched from thread
   PREV, the new thread is already running, and interrupts are
   still disabled.  This function is normally invoked by
   thread_schedule() as its final action before returning, but
   the first time a thread is scheduled it is called by
   switch_entry() (see switch.S).

   It's not safe to call printf() until the thread switch is
   complete.  In practice that means that printf()s should be
   added at the end of the function.

   After this function and its caller returns, the thread switch
   is complete. */
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 the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```

printf조심

이걸로 지금 돌고있는 thread 찾는다.

여기서 running 된다.

run할때 tick이 0 되는구나.

```c
static unsigned thread_ticks;   /* # of timer ticks since last yield. */
```

- semaphore
  - semaphore는 nonnegative integer이고 down(aka Probeer meaning `try` in Dutch) and up(aka Verhoog meaning increment).
  - 보통 semaphore를 0으로 initialize 해서 많이 사용한다. 용도는 어떤 한 번만 일어날 이벤트에 대해서 프로세스가 synchronization을 하게 하기 위함이다. 예를 들어, A라는 thread가 있고 B라는 thread가 있다고 했을 때, B가 어떤 이벤트가 끝나는 걸 알려줄 때까지 A를 기다리게 하고 싶은 경우가 있다고 하자. 이때 A에 sema_down을 걸어놓으면, A는 semaphore가 positive integer가 될 때까지 waiting에 들어간다. semaphore를 0으로 initialize 했기 때문에 A는 계속 기다리게 된다. 이때 B에서 어떤 event가 끝나고 나서 sema_up을 해주게 하면, semaphore 의 숫자가 1이 된다. 동시에 sema_up은 waiting list에 있는 thread를 깨워주게 된다. waiting list에 있던 Thread A가 깨어나고 semaphore를 살펴봤더니 양수가 된 것을 확인하고 다시 0으로 decrement 해주고 return 하게 된다.

└ A: main
B: idle

여기서 main thread가 계속유지된다해도 idle 만들어지면 떠넘어갈걸 바로.

```c
/* 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);
}
```

```c
/* 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);
}
```

→ 기본적인 thread 생성, block, unblock에 대해 이해함.

부족한 것은 언제 intr disable/enable 하는지.

~~인터럽트.~~

~~필요~~

디버깅 툴 필요

~~2번 과제 보기~~

~~부과제에서 뭘 해야하는지 알아야 해 (1/2 부분)~~

— 이미 세팅가 되어있어

~~— 분류 2 하여보기~~

· ~~3강 간단 파일 한 코드 들 보기~~

intr_init

timer_init

thread_ticks가 있고 thread.c

ticks가 있다 timer.c

```c
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}
```

```c
/* Registers external interrupt VEC_NO to invoke HANDLER, which
   is named NAME for debugging purposes.  The handler will
   execute with interrupts disabled. */
void
intr_register_ext (uint8_t vec_no, intr_handler_func *handler,
                   const char *name)
{
  ASSERT (vec_no >= 0x20 && vec_no <= 0x2f);
  register_handler (vec_no, 0, INTR_OFF, handler, name);
}
```

```c
/* Registers interrupt VEC_NO to invoke HANDLER with descriptor
   privilege level DPL.  Names the interrupt NAME for debugging
   purposes.  The interrupt handler will be invoked with
   interrupt status set to LEVEL. */
static void
register_handler (uint8_t vec_no, int dpl, enum intr_level level,
                  intr_handler_func *handler, const char *name)
{
  ASSERT (intr_handlers[vec_no] == NULL);   —▷ 이제 예약할거니까 비어있어야지.
  if (level == INTR_ON)
    idt[vec_no] = make_trap_gate (intr_stubs[vec_no], dpl);
  else
    idt[vec_no] = make_intr_gate (intr_stubs[vec_no], dpl);
  intr_handlers[vec_no] = handler;
  intr_names[vec_no] = name;
}
```

```c
/* Creates an interrupt gate that invokes FUNCTION with the given
   DPL. */
static uint64_t
make_intr_gate (void (*function) (void), int dpl)
{
  return make_gate (function, dpl, 14);
}

/* Creates a trap gate that invokes FUNCTION with the given
   DPL. */
static uint64_t
make_trap_gate (void (*function) (void), int dpl)
{
  return make_gate (function, dpl, 15);
}
```

```c
/* Creates an gate that invokes FUNCTION.

   The gate has descriptor privilege level DPL, meaning that it
   can be invoked intentionally when the processor is in the DPL
   or lower-numbered ring.  In practice, DPL==3 allows user mode
   to call into the gate and DPL==0 prevents such calls.  Faults
   and exceptions that occur in user mode still cause gates with
   DPL==0 to be invoked.  See [IA32-v3a] sections 4.5 "Privilege
   Levels" and 4.8.1.1 "Accessing Nonconforming Code Segments"
   for further discussion.

   TYPE must be either 14 (for an interrupt gate) or 15 (for a
   trap gate).  The difference is that entering an interrupt gate
   disables interrupts, but entering a trap gate does not.  See
   [IA32-v3a] section 5.12.1.2 "Flag Usage By Exception- or
   Interrupt-Handler Procedure" for discussion. */
static uint64_t
make_gate (void (*function) (void), int dpl, int type)
{
  uint32_t e0, e1;

  ASSERT (function != NULL);
  ASSERT (dpl >= 0 && dpl <= 3);
  ASSERT (type >= 0 && type <= 15);

  e0 = (((uint32_t) function & 0xffff)     /* Offset 15:0. */
        | (SEL_KCSEG << 16));              /* Target code segment. */

  e1 = (((uint32_t) function & 0xffff0000) /* Offset 31:16. */
        | (1 << 15)                        /* Present. */
        | ((uint32_t) dpl << 13)           /* Descriptor privilege level. */
        | (0 << 12)                        /* System. */
        | ((uint32_t) type << 8));         /* Gate type. */

  return e0 | ((uint64_t) e1 << 32);
}
```

▷ 뭐 어떻게 잘해서 TIMER_FREQ마다 timer_interrupt를 실행하겠지!

```c
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

```c
/* Called by the timer interrupt handler at each timer tick.
   Thus, this function runs in an external interrupt context. */
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 ();
}
```

```c
/* During processing of an external interrupt, directs the
   interrupt handler to yield to a new process just before
   returning from the interrupt.  May not be called at any other
   time. */
void
intr_yield_on_return (void)
{
  ASSERT (intr_context ());
  yield_on_return = true;
}
```

intr-stubs.S 봐야하네.

```c
338  /* Interrupt handlers. */
339
340  /* Handler for all interrupts, faults, and exceptions.  This
341     function is called by the assembly language interrupt stubs in
342     intr-stubs.S.  FRAME describes the interrupt and the
343     interrupted thread's registers. */
344  void
345  intr_handler (struct intr_frame *frame)
346  {
347    bool external;
348    intr_handler_func *handler;
349
350    /* External interrupts are special.
351       We only handle one at a time (so interrupts must be off)
352       and they need to be acknowledged on the PIC (see below).
353       An external interrupt handler cannot sleep. */
354    external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
355    if (external)
356      {
357        ASSERT (intr_get_level () == INTR_OFF);
358        ASSERT (!intr_context ());
359
360        in_external_intr = true;
361        yield_on_return = false;
362      }
363
364    /* Invoke the interrupt's handler. */
365    handler = intr_handlers[frame->vec_no];
366    if (handler != NULL)
367      handler (frame);
368    else if (frame->vec_no == 0x27 || frame->vec_no == 0x2f)
369      {
370        /* There is no handler, but this interrupt can trigger
371           spuriously due to a hardware fault or hardware race
372           condition.  Ignore it. */
373      }
374    else
375      unexpected_interrupt (frame);
376
377    /* Complete the processing of an external interrupt. */
378    if (external)
379      {
380        ASSERT (intr_get_level () == INTR_OFF);
381        ASSERT (intr_context ());
382
383        in_external_intr = false;
384        pic_end_of_interrupt (frame->vec_no);
385
386        if (yield_on_return)
387          thread_yield ();
388      }
389  }
```

One at a time 이니까. (line 357) → INTR_OFF 여야한다.

(line 358) → 아직 external interrupt 없음 표시하는 것 true 안함.

(line 367) → timer_interrupt handler라고 생각해보면 handler 안에서 intr_yield_on_return()을 호출해서 yield_on_return은 true로함. 그래서 386 line 참. → thread_yield()

```c
/* 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);
}
```

(ASSERT (!intr_context ())) └→ external interrupt 돌고있는 중에는 yield 못한다.

(cur->status = THREAD_READY) → 지금 돌던 것 READY. ready_list.

```c
/* Returns true during processing of an external interrupt
   and false at all other times. */
bool
intr_context (void)
{
  return in_external_intr;
}
```

```c
/* External interrupts are those generated by devices outside the
   CPU, such as the timer.  External interrupts run with
   interrupts turned off, so they never nest, nor are they ever
   pre-empted.  Handlers for external interrupts also may not
   sleep, although they may invoke intr_yield_on_return() to
   request that a new process be scheduled just before the
   interrupt returns. */
static bool in_external_intr;   /* Are we processing an external interrupt? */
static bool yield_on_return;    /* Should we yield on interrupt return? */
```

- external interrupt로 thread가 yield되는 시나리오에 대한 분석이 이뤄졌다. — 하소연g
- interrupt frame 관련 내용 이해 필요.

인터럽트 frame이 thread의 stack에 어떻게 들어갈까? / thread가 잘 돌다가 external interrupt에 의해 강제적으로 바뀐다고 해보자. 그럼 어떻게 되는가. 결국 switch_threads 함수가 처리할 것 같긴한데,...

코드는 [illegible] A.4.3 부분 보면 좋음.

```
/* Interrupt stubs.

   This defines 256 fragments of code, named `intr00_stub'
   through `intrff_stub', each of which is used as the entry
   point for the corresponding interrupt vector.  It also puts
   the address of each of these functions in the correct spot in
   `intr_stubs', an array of function pointers.

   Most of the stubs do this:

        1. Push %ebp on the stack (frame_pointer in `struct intr_frame').

        2. Push 0 on the stack (error_code).

        3. Push the interrupt number on the stack (vec_no).

   The CPU pushes an extra "error code" on the stack for a few
   interrupts.  Because we want %ebp to be where the error code
   is, we follow a different path:

        1. Push a duplicate copy of the error code on the stack.

        2. Replace the original copy of the error code by %ebp.

        3. Push the interrupt number on the stack. */

        .data
.globl intr_stubs
intr_stubs:

/* This implements steps 1 and 2, described above, in the common
   case where we just push a 0 error code. */
#define zero                                    \
        pushl %ebp;                             \
        pushl $0

/* This implements steps 1 and 2, described above, in the case
   where the CPU already pushed an error code. */
#define REAL                                    \
        pushl (%esp);                           \
        movl %ebp, 4(%esp)

/* Emits a stub for interrupt vector NUMBER.
   TYPE is `zero', for the case where we push a 0 error code,
   or `REAL', if the CPU pushes an error code for us. */
#define STUB(NUMBER, TYPE)                      \
        .text;                                  \
.func intr##NUMBER##_stub;                      \
intr##NUMBER##_stub:                            \
        TYPE;                                   \
        push $0x##NUMBER;                       \
        jmp intr_entry;                         \
.endfunc;                                       \
                                                \
        .data;                                  \
        .long intr##NUMBER##_stub;
```

대체.

```c
/* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
  };
```

```
/* Main interrupt entry point.

   An internal or external interrupt starts in one of the
   intrNN_stub routines, which push the `struct intr_frame'
   frame_pointer, error_code, and vec_no members on the stack,
   then jump here.

   We save the rest of the `struct intr_frame' members to the
   stack, set up some registers as needed by the kernel, and then
   call intr_handler(), which actually handles the interrupt.

   We "fall through" to intr_exit to return from the interrupt.
*/
.func intr_entry
intr_entry:
        /* Save caller's registers. */
        pushl %ds
        pushl %es
        pushl %fs
        pushl %gs
        pushal          -- reg 여러개 push

        /* Set up kernel environment. */
        cld                     /* String instructions go upward. */
        mov $SEL_KDSEG, %eax    /* Initialize segment registers. */
        mov %eax, %ds
        mov %eax, %es
        leal 56(%esp), %ebp     /* Set up frame pointer. */

        /* Call interrupt handler. */
        pushl %esp
.globl intr_handler
        call intr_handler
        addl $4, %esp
.endfunc
```

※ intr-stub.S에서만 쓰는 static function이다.
그래서 .func intr_entry 말고 .global intr_entry가 없다.

https://www.cs.unc.edu/~porter/courses/cse306/s13/slides/x86-assembly.pdf

- Stack Manipulation - PUSH, POP
  - PUSHAL, POPAL – push/pop "big 7" registers at once
  - PUSHF, POPF - push/pop eflags register

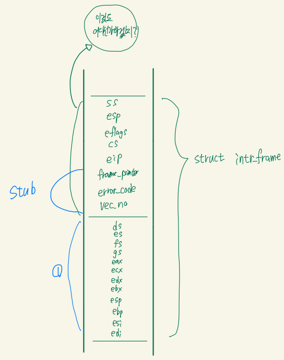
https://c9x.me/x86/html/file_module_x86_id_270.html

x86 Instruction Set Reference

PUSHA/PUSHAD

Push All General-Purpose Registers

```c
/* Interrupt handlers. */

/* Handler for all interrupts, faults, and exceptions.  This
   function is called by the assembly language interrupt stubs in
   intr-stubs.S.  FRAME describes the interrupt and the
   interrupted thread's registers. */
void
intr_handler (struct intr_frame *frame)
{
  bool external;
  intr_handler_func *handler;

  /* External interrupts are special.
     We only handle one at a time (so interrupts must be off)
     and they need to be acknowledged on the PIC (see below).
     An external interrupt handler cannot sleep. */
  external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
  if (external)
    {
      ASSERT (intr_get_level () == INTR_OFF);
      ASSERT (!intr_context ());

      in_external_intr = true;
      yield_on_return = false;
    }

  /* Invoke the interrupt's handler. */
  handler = intr_handlers[frame->vec_no];
  if (handler != NULL)
    handler (frame);
  else if (frame->vec_no == 0x27 || frame->vec_no == 0x2f)
    {
      /* There is no handler, but this interrupt can trigger
         spuriously due to a hardware fault or hardware race
         condition.  Ignore it. */
    }
  else
    unexpected_interrupt (frame);

  /* Complete the processing of an external interrupt. */
  if (external)
    {
      ASSERT (intr_get_level () == INTR_OFF);
      ASSERT (intr_context ());

      in_external_intr = false;
      pic_end_of_interrupt (frame->vec_no);

      if (yield_on_return)
        thread_yield ();
    }
}
```



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

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

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

```
/* One semaphore in a list. */
struct semaphore_elem
  {
    struct list_elem elem;      /* List element. */
    struct semaphore semaphore; /* This semaphore. */
  };
```

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

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

```
/* Initializes LOCK.  A lock can be held by at most a single
   thread at any given time.  Our locks are not "recursive", that
   is, it is an error for the thread currently holding a lock to
   try to acquire that lock.

   A lock is a specialization of a semaphore with an initial
   value of 1.  The difference between a lock and such a
   semaphore is twofold.  First, a semaphore can have a value
   greater than 1, but a lock can only be owned by a single
   thread at a time.  Second, a semaphore does not have an owner,
   meaning that one thread can "down" the semaphore and then
   another one "up" it, but with a lock the same thread must both
   acquire and release it.  When these restrictions prove
   onerous, it's a good sign that a semaphore should be used,
   instead of a lock. */
void
lock_init (struct lock *lock)
{
  ASSERT (lock != NULL);

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

```
/* Initializes semaphore SEMA to VALUE.  A semaphore is a
   nonnegative integer along with two atomic operators for
   manipulating it:

   - down or "P": wait for the value to become positive, then
     decrement it.

   - up or "V": increment the value (and wake up one waiting
     thread, if any). */
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

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

```
/* Initializes condition variable COND.  A condition variable
   allows one piece of code to signal a condition and cooperating
   code to receive the signal and act upon it. */
void
cond_init (struct condition *cond)
{
  ASSERT (cond != NULL);

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

```
/* Acquires LOCK, sleeping until it becomes available if
   necessary.  The lock must not already be held by the current
   thread.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but interrupts will be turned back on if
   we need to sleep. */
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 ();
}
```

```
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
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);
}
```

```
/* Tries to acquires LOCK and returns true if successful or false
   on failure.  The lock must not already be held by the current
   thread.

   This function will not sleep, so it may be called within an
   interrupt handler. */
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;
}
```

```
/* Down or "P" operation on a semaphore, but only if the
   semaphore is not already 0.  Returns true if the semaphore is
   decremented, false otherwise.

   This function may be called from an interrupt handler. */
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;
}
```

```
/* Releases LOCK, which must be owned by the current thread.

   An interrupt handler cannot acquire a lock, so it does not
   make sense to try to release a lock within an interrupt
   handler. */
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

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

```
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.

   This function may be called from an interrupt handler. */
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);
}
```

```
/* Returns true if the current thread holds LOCK, false
   otherwise.  (Note that testing whether some other thread holds
   a lock would be racy.) */
bool
lock_held_by_current_thread (const struct lock *lock)
{
  ASSERT (lock != NULL);

  return lock->holder == thread_current ();
}
```

`bool lock_held_by_current_thread (const struct lock *lock)` [Function]
Returns true if the running thread owns *lock*, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

In the documentation for the Pintos operating system, it's stated that there is no function to test whether an arbitrary thread owns a lock, specifically in the context of `lock_held_by_current_thread`.

The primary reason for this is likely the issue of race conditions. Imagine a function exists that allows you to check whether an arbitrary thread owns a lock. By the time you receive the result and act upon it, the status of the lock ownership could have changed, leading to unpredictable behavior or bugs in the program.

For example, let's say Thread A checks and finds out that Thread B owns a lock. Based on this information, Thread A decides to perform some other operation. However, just after this, Thread B releases the lock and Thread C acquires it. Now the information that Thread A acted upon is outdated and potentially incorrect, which could lead to errors or unintended behavior.

In a multi-threaded environment, operations often happen so quickly that even a small delay can cause the state to change dramatically. Therefore, the documentation notes that there is no function to test arbitrary ownership because the answer could be invalidated almost immediately, making the function unreliable and potentially harmful.

Here's the Korean translation for your convenience:

Pintos 운영 체제에 대한 문서에서는 임의의 스레드가 락을 소유하고 있는지 확인하는 함수가 없다고 명시적으로 언급되어 있습니다. `lock_held_by_current_thread`의 맥락에서 특히 그렇습니다.

이에 대한 주된 이유는 경쟁 조건(race conditions) 문제일 가능성이 높습니다. 임의의 스레드가 락을 소유하고 있는지 확인할 수 있는 함수가 있다고 상상해보십시오. 결과를 받고 그것을 기반으로 무언가를 수행하기 전에 락 소유 상태가 바뀔 수 있으므로, 프로그램에서 예측할 수 없는 동작이나 버그가 발생할 수 있습니다.

예를 들어, 스레드 A가 스레드 B가 락을 소유하고 있다는 것을 확인하고, 이 정보를 기반으로 다른 작업을 수행하기로 결정했다고 가정합니다. 그러나 이후에 스레드 B가 락을 해제하고 스레드 C가 그것을 획득합니다. 이제 스레드 A가 행동을 취한 정보는 구식이며 잠재적으로 부정확하므로, 오류나 의도하지 않은 동작이 발생할 수 있습니다.

다음 스레드 환경에서는 작업이 너무 빨리 발생하기 때문에 작은 지연조차도 상태를 크게 바꿀 수 있습니다. 따라서 문서에서는 임의의 소유권을 테스트하는 함수가 없다고 언급하는 것은 이러한 함수가 불안정하고 잠재적으로 해로울 수 있기 때문입니다.

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

  struct thread *cur = thread_current ();
  if (lock->holder) {
    cur->wait_on_lock = lock;
    list_insert_ordered (&lock->holder->donations, &cur->donation_elem,
                         thread_compare_donate_priority, 0);
    donate_priority ();
  }

  sema_down (&lock->semaphore);

  cur->wait_on_lock = NULL;
  lock->holder = cur;
}
```

```c
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

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

기본 idea : sleep할때 thread_yield로 ready_list에 추가하지말고 sleep list 에 추가하여 시간이 전부 경과하였을 때만 다시 thread가 실행되도록.

- thread에 일어나야하는 tick 정보 추가
- sleep list initialize 해줘야함.
- thread_sleep 함수로 " 일어날 시간 TCB에 저장 / sleep list에 추가 / thread block시키기 "
- timer_sleep 구현을 바꾸기. thread_sleep(현재틱 + 잠들틱) ↳ 일어날틱
- 그럼 thread는 어떻게 깨우는가?
  - sleep-list를 정렬된 queue로한다! 그리고 timer 틱 오를때 마다 정렬된 큐가장 앞부분이 깨날시간인지 확인. 같은 우선순위면?

1차

next thread_to_run 함수로 오면 지금은 ready_list에서 list_pop_front 함수로 다음번 실행할 thread를 정한다.

thread_unblock 함수랑 thread_yield 함수에서 ready_list로 push_back 하지말고 정렬된 상태로 넣자.

ready list를 정렬된 상태로! 여기에서만 ready list에 뭐 넣는다.

선점가능!

1. thread_create : thread 생성시에 지금 돌아가는 thread와 비교하기. → 함수 자체에서 해줌.
2. thread_set_priority : 지금 돌아가는 thread 우선순위 바뀔때 ready list랑 비교하기. → 함수자체에서 switch하고 assert하기
3. ready_list에 추가될것이 지금도는것보다 우선순위에 (여기에서 이렇게 구현되든) ∵ 추가되는것에 ready list에

그리고 thread switch해서. 결과에 맞게

unblock 함수에서 바꾸는것 X ∵ 여긴 switch 불가면안되는함수.

ready list에 있는 거 priority 바뀌어지나?

, cond var을 기다리고 쉬고있는... 우선순위 높은게 먼저 unblock 되도록! synch.c 파일 수정하자.

sema

- 세마포어의 waiters를 priority로 정렬하여 구현.
- ~~세마포어 정렬 ...~~ (block된 thread prio 바뀔수 없긴함. by donation → 여기서 정렬)

cond

세마포어를 cond의 waiters에 추가할때 정렬하여 구현.

donate될때 발생하는 정렬

세마포어 앞 얼굴 (thread 우선순위) 끼리 비교해서 가장 높은걸 기준.

---

lock을 얻으려는데 남이 가지고 있다면 받아오자

- 코드작성

- 보고서작성

- flow그리기
  - 디테일하게.
- design 바꾸기.
- 자료더찾기
  - 심화.