

# Pintos Internals

# main()

```
int main (void)
{
    ram_init ();
    argv = read_command_line ();
    argv = parse_options (argv);

    ①/* make ourselves as a thread */
    thread_init ();
    console_init ();

    printf ("Pintos booting with '%zu'
    \"kB RAM.\n", ram_pages*PGSIZE/1024);

    ②/* initialize memory system */
    palloc_init ();
    malloc_init ();
    paging_init ();

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

    ③intr_init (); /* intr handlers */
    timer_init ();
    kbd_init ();
    input_init ();

    #ifdef USERPROG
        exception_init ();
        syscall_init ();
    #endif

    ④/* start thread scheduler */
    thread_start ();
    serial_init_queue ();
    timer_calibrate ();

    #ifdef FILESYS
        disk_init ();
        filesys_init (format_filesys);
    #endif

    printf ("Boot complete.\n");

    ⑤/* run tests or user programs */
    run_actions (argv);

    if (power_off_when_done)
        power_off ();
    thread_exit ();
}
```

# **1. THE INITIAL THREAD**

# The Initial Thread


- Create a struct `thread` for the initial thread by transforming the code that's currently running into a thread

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

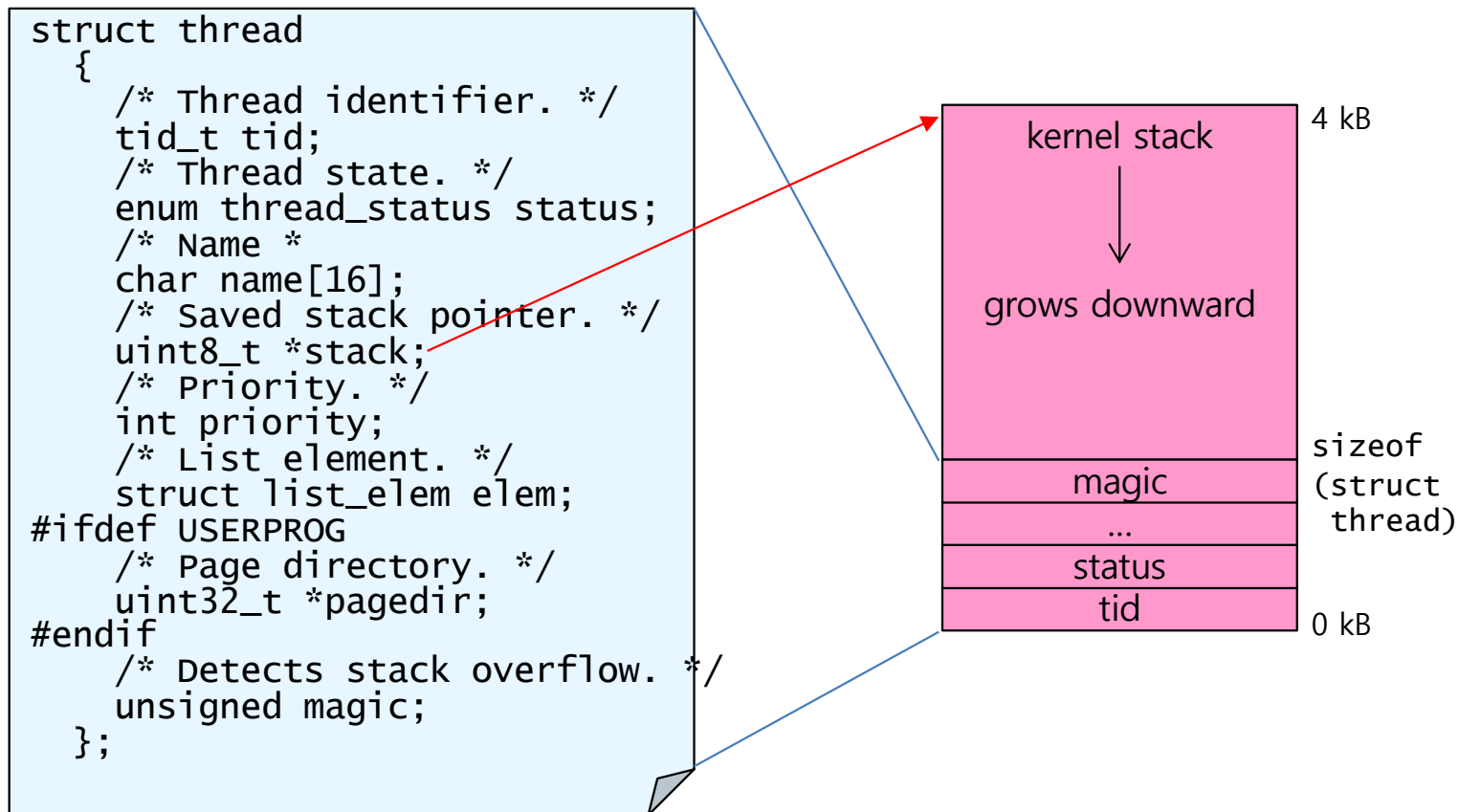
void thread_init (void)
{
    ASSERT (intr_get_level () == INTR_OFF);

    lock_init (&tid_lock);
    list_init (&ready_list); /* initialize ready queue */

    /* 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 ();
}
```

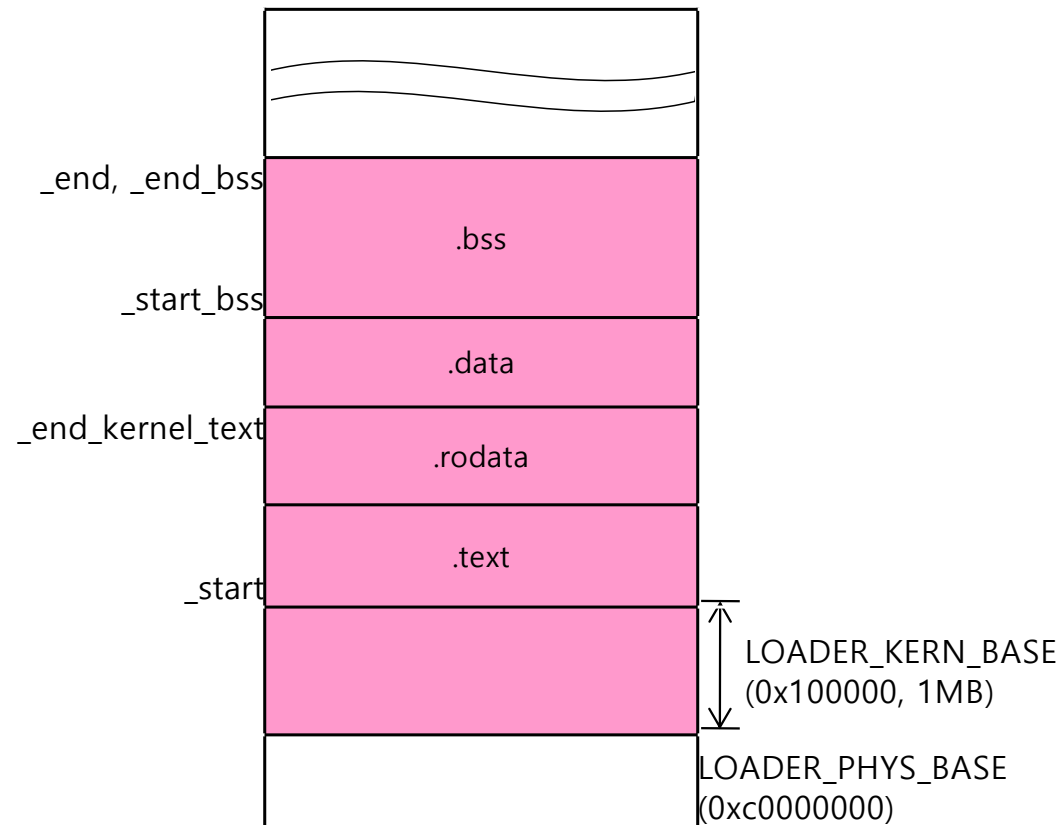


# struct thread – pintos<sup>9</sup> process control block

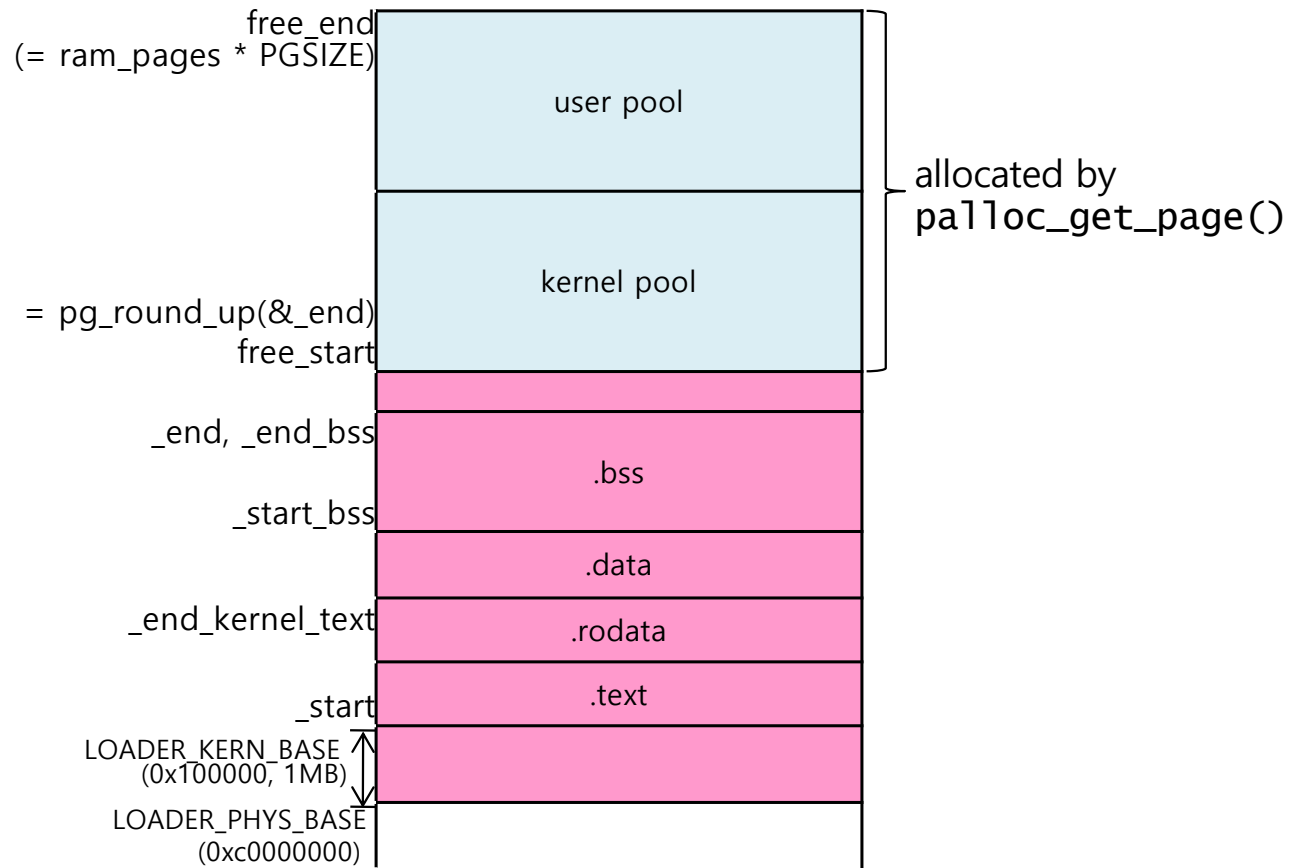


## **2. INITIALIZING MEMORY**

# 부팅 직후 최초 메모리 (in kernel linker script kernel.lds.S)



# 동적 메모리 할당 영역 초기화 palloc\_init()



# 페이징 시스템 초기화 paging\_init ()

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

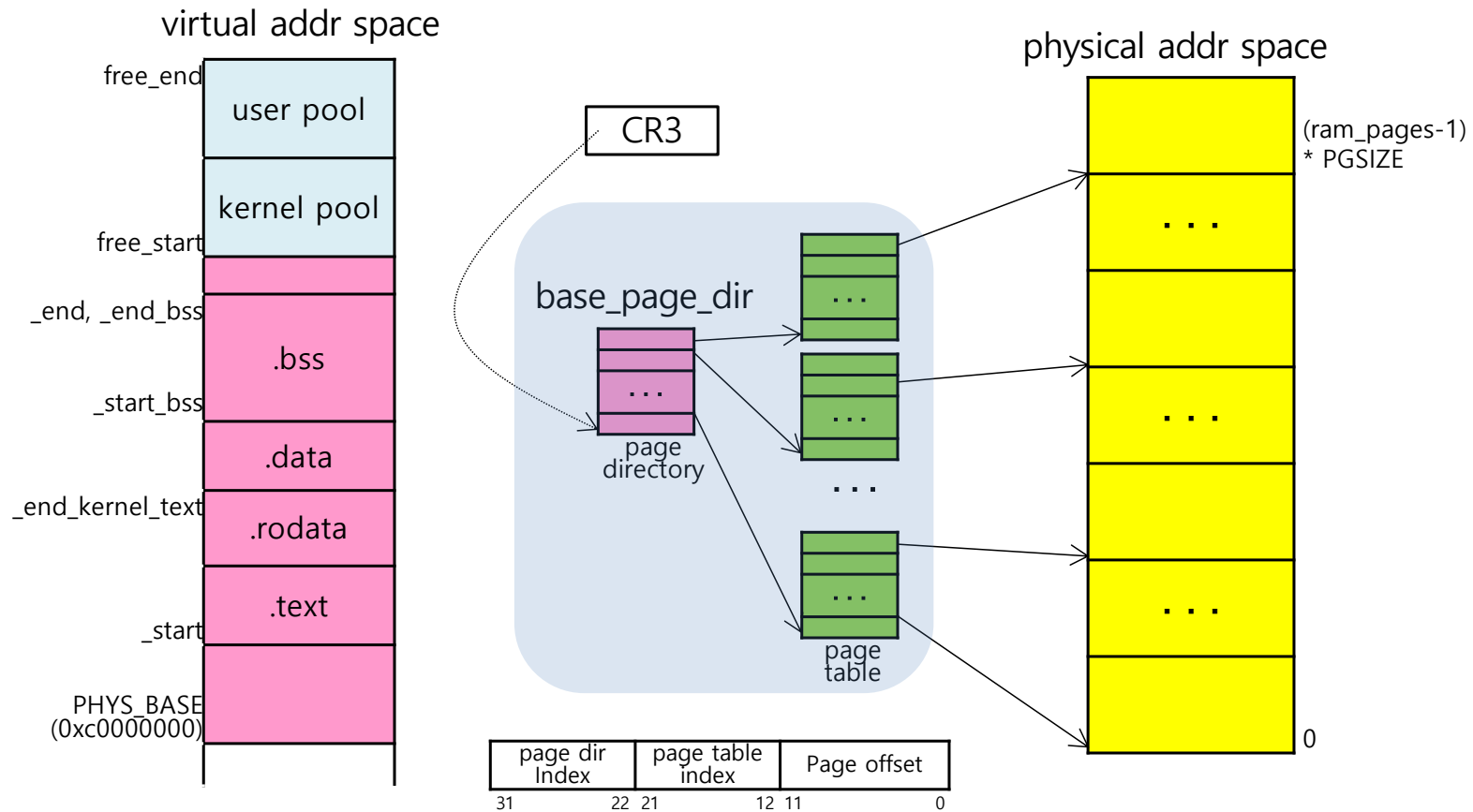
    pd = base_page_dir =
        palloc_get_page (PAL_ASSERT |
                        PAL_ZERO);
    pt = NULL;
    for(page=0;page<ram_pages;page++) {
        uintptr_t paddr=page*PGSIZE;
        char *vaddr = ptov(paddr);
        size_t pde_idx= pd_no(vaddr);
        size_t pte_idx= pt_no(vaddr);
   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

    asm volatile ("movl %0, %%cr3" : :
        "r" (vtop
            (base_page_dir)));
}
```

# Virtual Address Translation



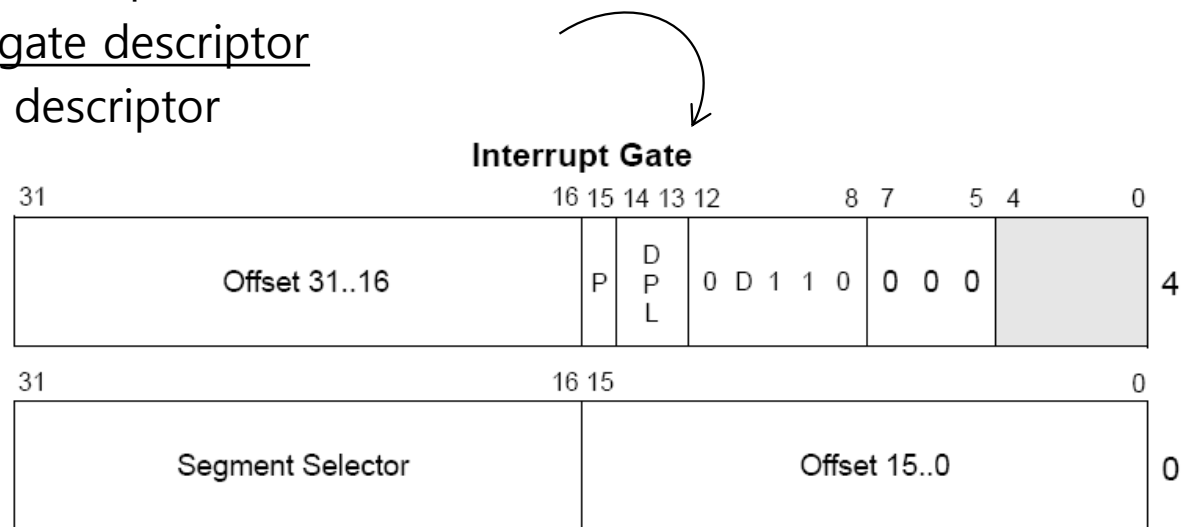
### **3. INITIALIZING INTERRUPT HANDLING**

# Internal vs. External Interrupts

- Internal interrupts (trap 등의 SW interrupt)
  - Interrupts caused directly by CPU instructions
  - System calls, invalid memory access, divide by zero, ...
  - Internal interrupts are **synchronous**, meaning that their delivery is synchronized with CPU instructions
  - `intr_disable()` does not disable internal interrupts
  - 내부 인터럽트 처리 중에는 아무 짓이나 할 수 있으며, 따라서 다른 커널 코드/다른 인터럽트와의 동기화 필요
- External interrupts (I/O interrupt)
  - Interrupts originating outside the CPU
  - H/W devices such as timer, keyboard, disks, ...
  - External interrupts are **asynchronous**, meaning that their delivery is NOT synchronized with CPU instructions
  - Handling of external interrupts can be postponed with `intr_disable()`
  - 외부인터럽트 처리 중에는 sleep/yield 할 수 없으며, 따라서 다른 인터럽트는 발생하지 않도록 하고, 가능한 빨리 처리를 마무리

# Interrupt Descriptor Table

- Interrupt Descriptor Table (IDT)
  - Pointed by special register `idt_r`
- IDT descriptors
  - Task-gate descriptor
  - Interrupt-gate descriptor
  - Trap-gate descriptor

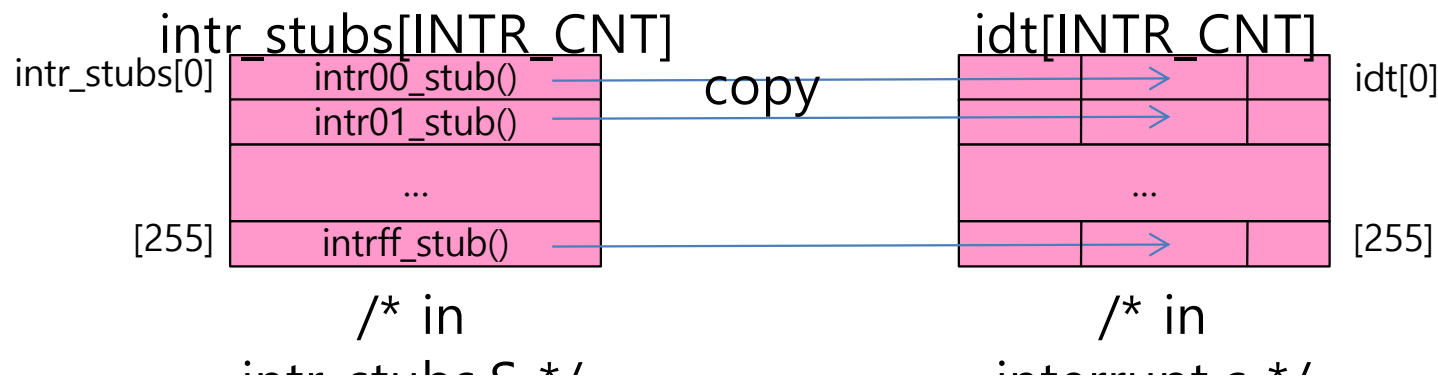


# Interrupt Infrastructure

```
void intr_init (void)
{
    pic_init (); /* Initialize interrupt controller. */

    /* Initialize IDT. */
    for (i = 0; i < INTR_CNT; i++)
        idt[i] = make_intr_gate (intr_stubs[i], 0);

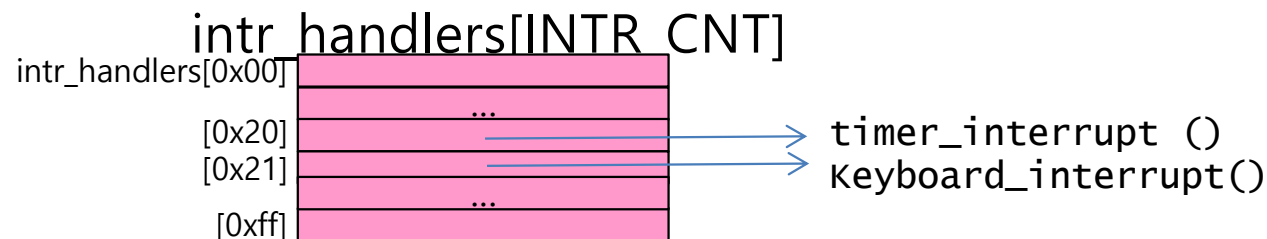
    /* Load IDT register. */
    idtr_operand = make_idtr_operand (sizeof idt - 1, idt);
    asm volatile ("lidt %0" : : "m" (idtr_operand));
}
```



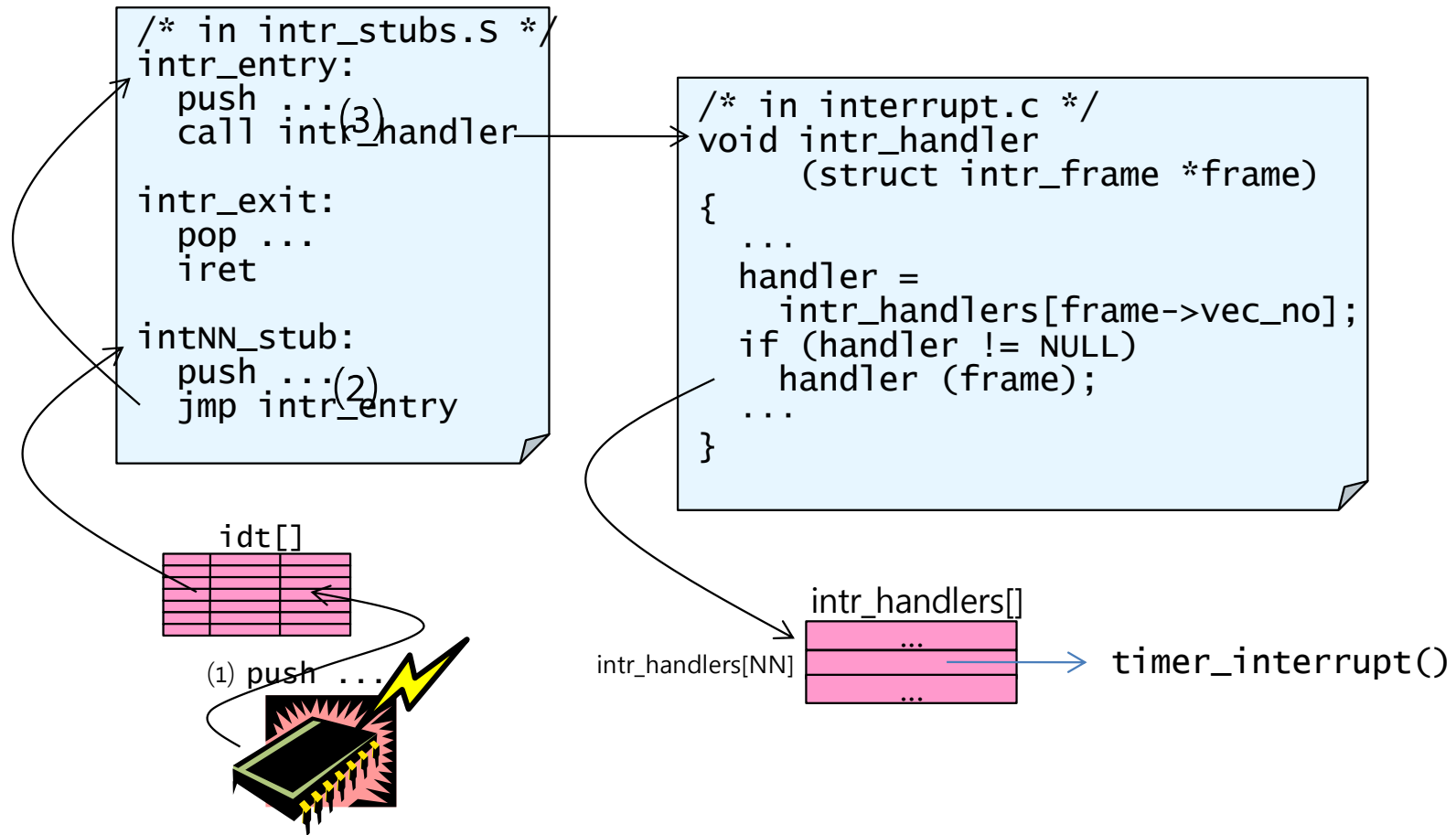
# Interrupt Initialization

```
void timer_init (void)
{
    uint16_t count = (1193180 + TIMER_FREQ / 2) / TIMER_FREQ;
    outb (0x43, 0x34);
    outb (0x40, count & 0xff);
    outb (0x40, count >> 8);
    intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}

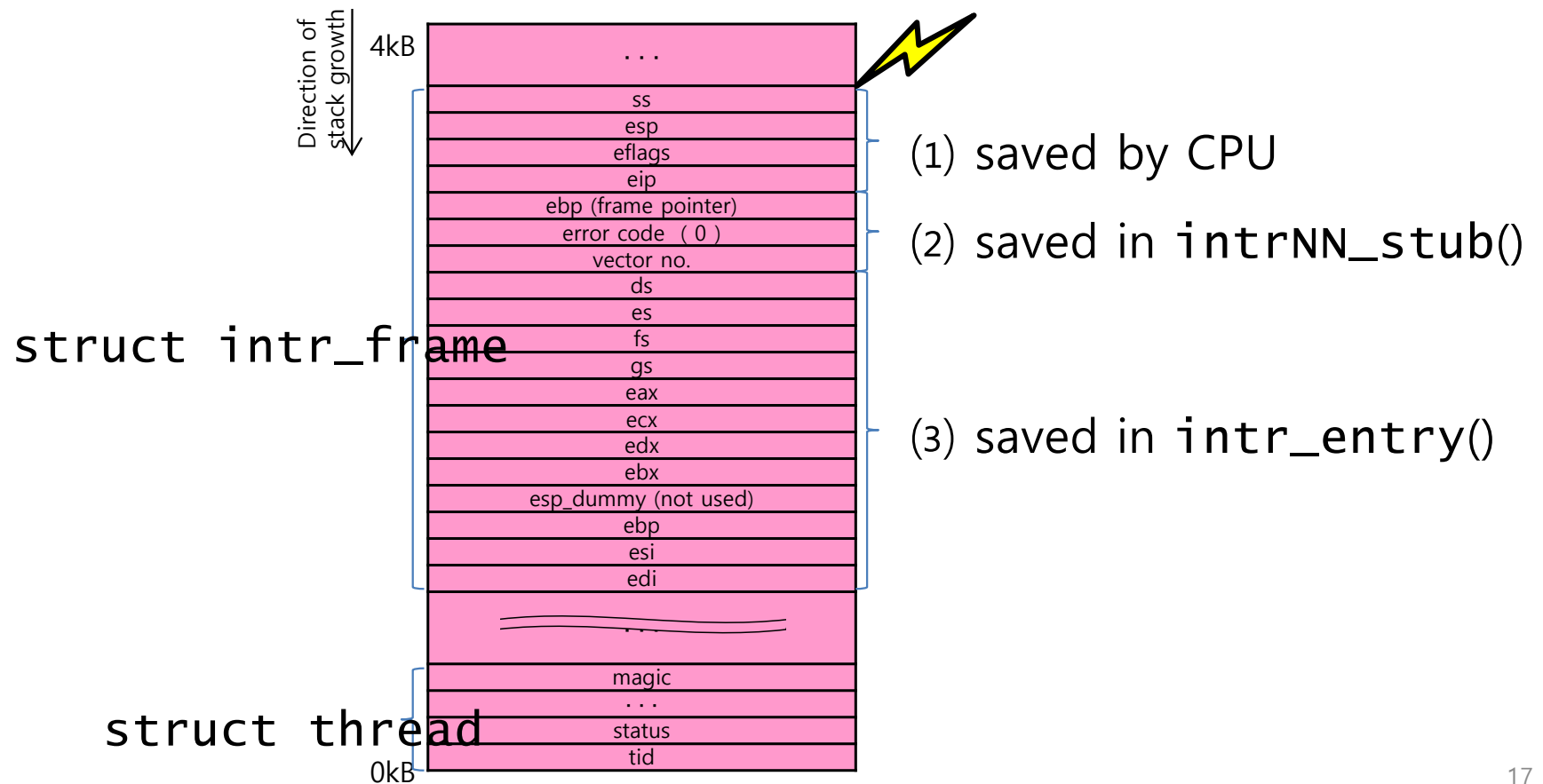
void kbd_init (void)
{
    intr_register_ext (0x21, keyboard_interrupt, "8042 Keyboard");
}
```



# Interrupt Handling



# Interrupt Frame on Interrupted Thread's Stack



# intr\_handler()

```
/* 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 & 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 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 h/w fault or
        h/w race condition. Ignore it. */
    }
    else {
        /* No handler & not spurious. Invoke
        the unexpected interrupt
        handler*/
        intr_dump_frame (frame);
        PANIC ("Unexpected interrupt");
    }

    /* 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 ();
    }
}
```

## **4. STARTING THREAD SCHEDULER**

# Starting Thread Scheduling

```
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);
}
```

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

    idle_thread =
    thread_current();
    sema_up (idle_started);

    for (;;) {
        intr_disable ();
        thread_block ();

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

# Thread Creation

```
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);  
  
    /* 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 ();
```

allocate struct thread (and stack), and initialize it

```
        /* 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;  
  
        /* Add to run queue. */  
        thread_unblock (t);  
  
        return tid;  
    }
```

creates some  
**fake stack frames**  
for switch\_threads()  
and kernel\_thread()

# Thread Switching – Case I

- `thread_yield()`



```
intr_handler ()
{
    yield_on_return = false;
    ...
    handler();
    ...
    if (external) {
        ...
        if (yield_on_return)
            thread_yield ();
    }
}
```

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

`yield_on_return ← 1`

# thread\_yield()

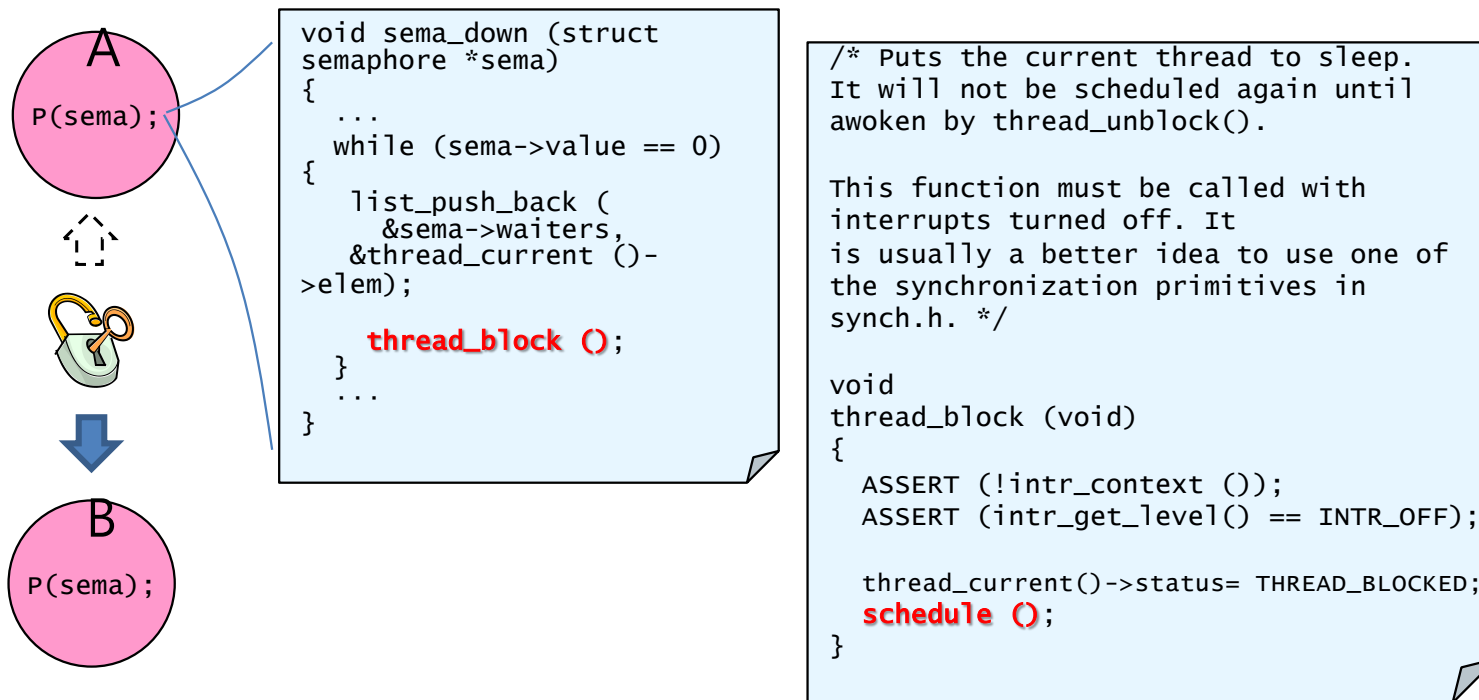
```
/* Yields the CPU. The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim.
*/
void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    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);
}
```

# Thread Switching – Case II

- `thread_block()`



# Thread Switching – Case III

- `thread_exit()`

```
/* Function used as the basis for a
kernel thread. */
static void kernel_thread
    (thread_func *function, void *aux)
{
    ASSERT (function != NULL);

    /* The scheduler runs with
    interrupts off. */
    intr_enable ();

    /* Execute the thread function. */
    function (aux);

    /* If function() returns,
    kill the thread. */
    thread_exit ();}
```

```
/* Deschedules the current thread
and destroys it. Never returns to
the caller. */
void thread_exit (void)
{
    ASSERT (!intr_context ());

#ifdef USERPROG
    process_exit ();
#endif

    /* Just set our status to dying
    and schedule another process.
    We will be destroyed during
    the
    call to schedule_tail(). */
    intr_disable ();
    thread_current ()->status =
    THREAD_DYING;
    schedule ();
    NOT_REACHED ();
}
```

# schedule()

```
/* 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 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);
    schedule_tail (prev);
}
```

# switch\_threads()

```
.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 offset of (struct thread,
stack).
.globl thread_stack_ofs

    mov thread_stack_ofs, %edx

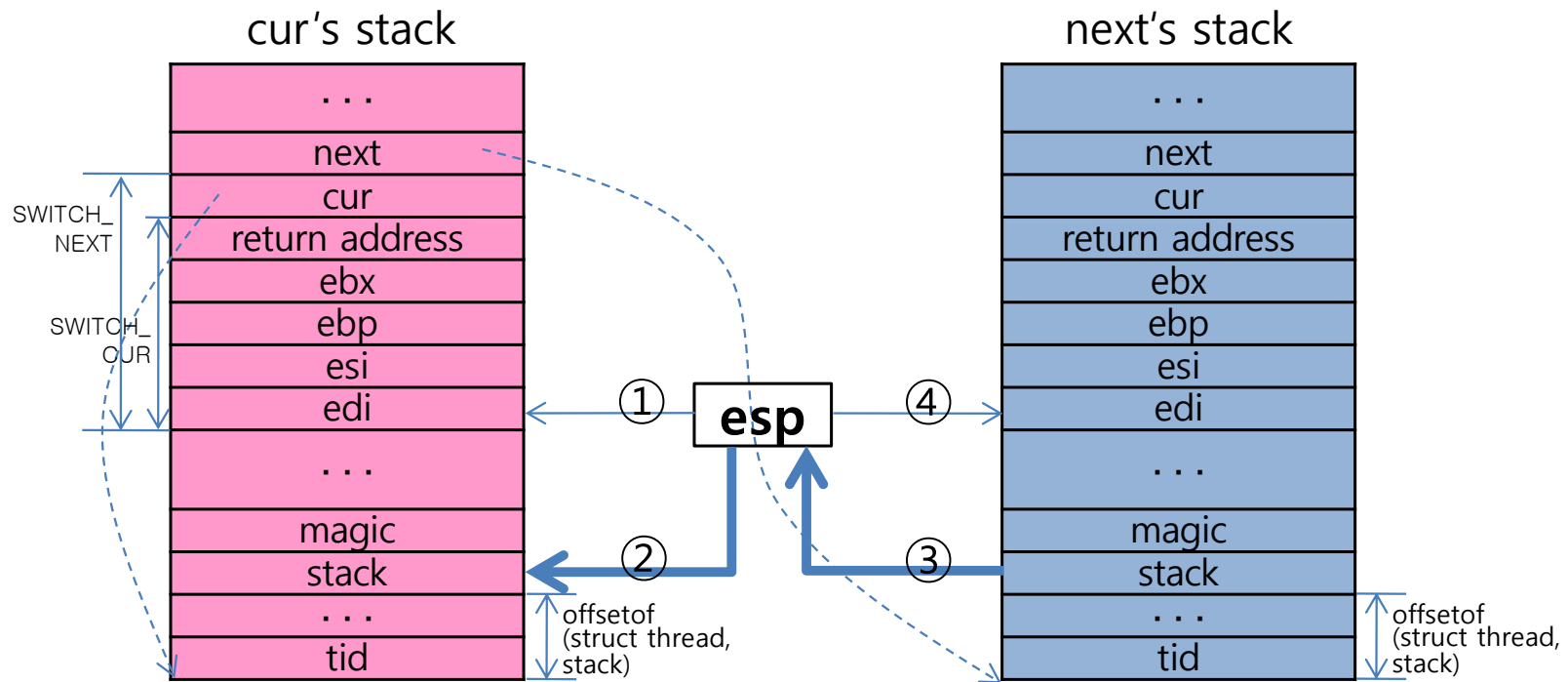
    # Save current stack pointer to
    # old thread's stack, if any.
    movl SWITCH_CUR(%esp), %eax
    movl %esp, (%eax,%edx,1)

    # 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

    # Get offset of (struct thread,
    .endfunc
```

# Context Switch



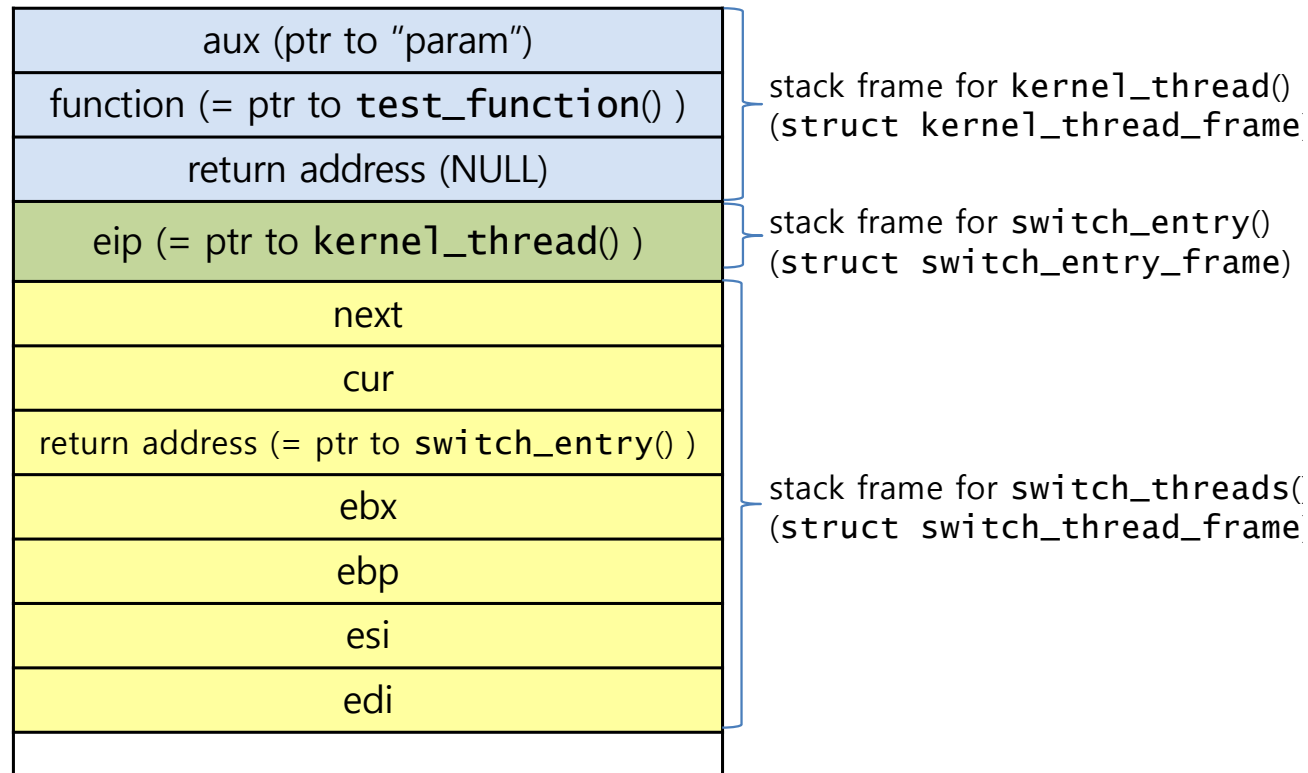
# schedule\_tail()

```
void schedule_tail (struct thread *prev)
{
    struct thread *cur = running_thread ();
    ASSERT (intr_get_level () == INTR_OFF);
    cur->status = THREAD_RUNNING; /* Mark us as running. */
    thread_ticks = 0; /* Start new time slice. */
#ifdef USERPROG
    process_activate (); /* Activate the new address space. */
#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);
    }
}
```

# Special Case – When a Thread Is Newly Created

```
thread_create ("test", PRI_DEFAULT, test_function, "param");
```



# switch\_entry()

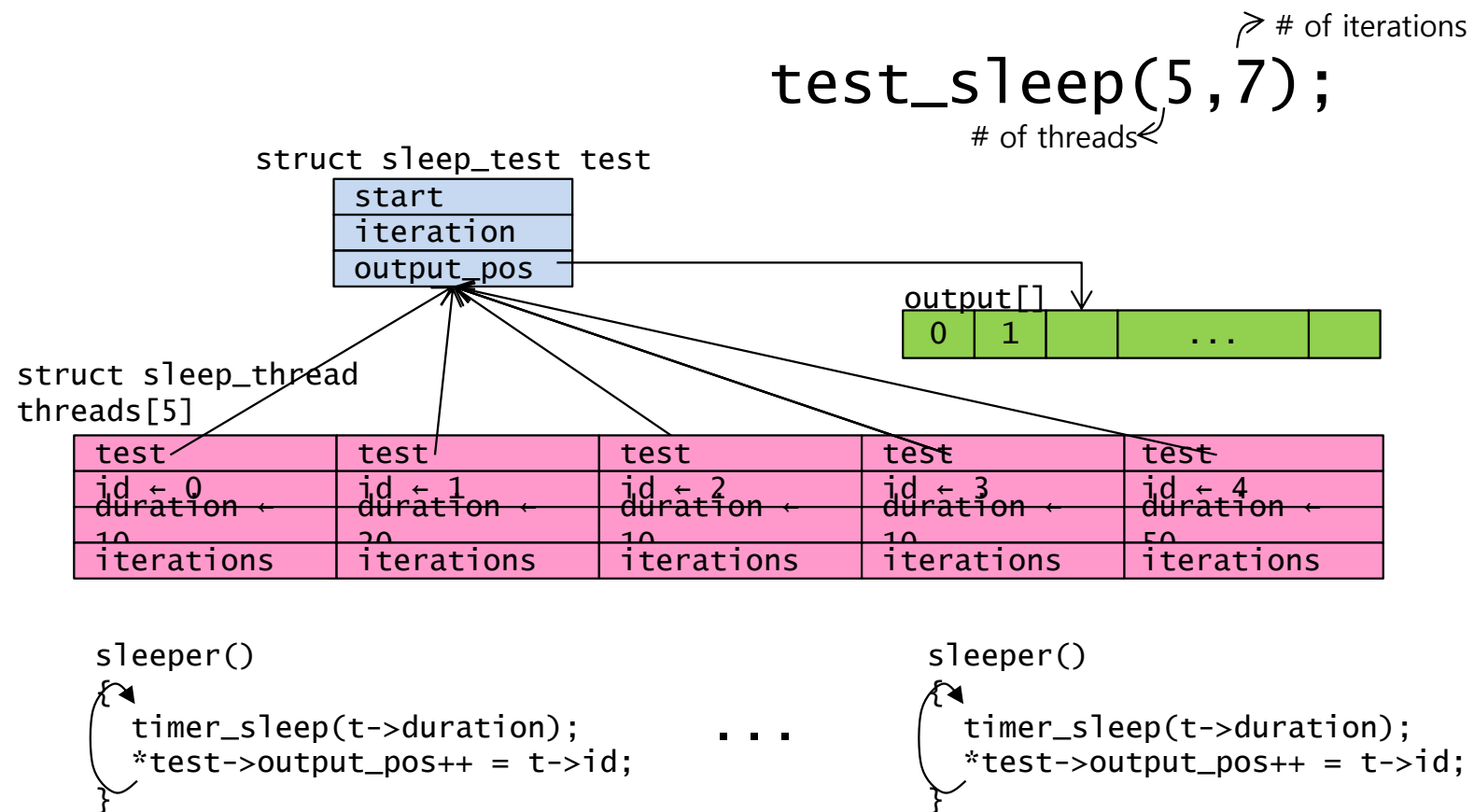
```
.globl switch_entry
.func switch_entry
switch_entry:
    # Discard switch_threads() arguments.
    addl $8, %esp

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

    # Start thread proper.
    ret
.endfunc
```

## **5. RUNNING TESTS**

# Running Kernel Command Line – alarm-multiple



# Output of alarm-multiple

