# Operating Systems Lab:

# Context Switch in Pintos


Youjip Won

# Overview

- Process structure
- Process state
- Process context
- schedule()
- switch thread

# Process structure: `struct thread`

pintos/src/threads/thread.h

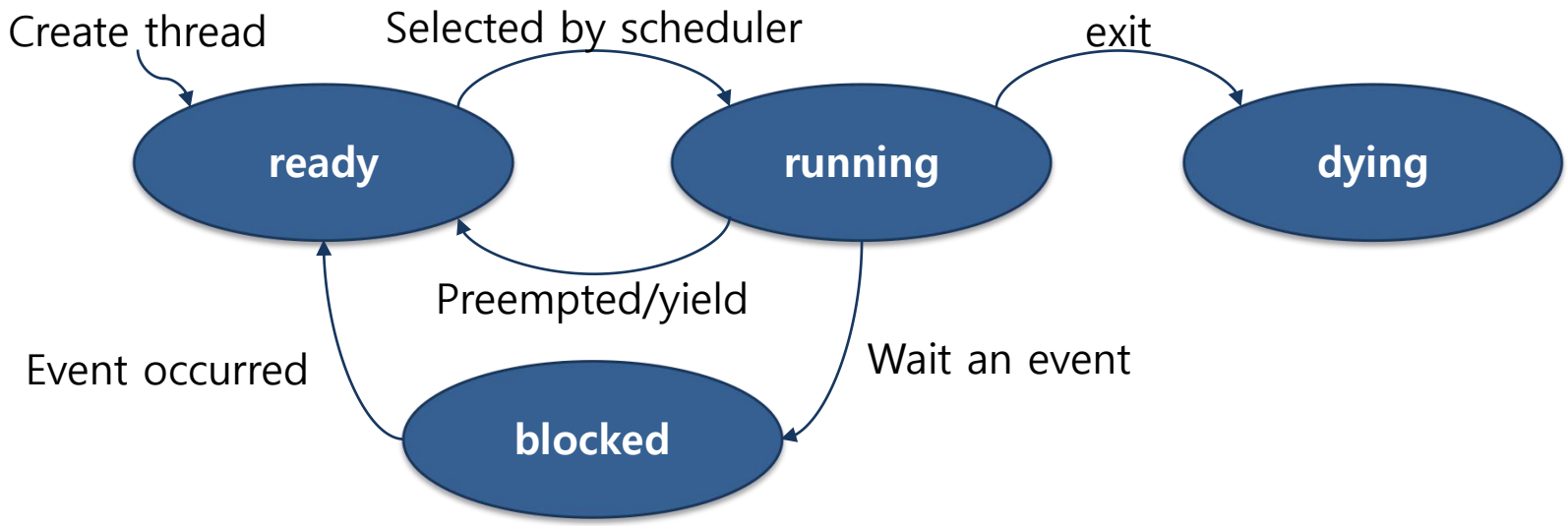
```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack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

```
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. */
  };
```

# Process state

- Process state

```
/* States in a thread's life cycle. */
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. */
  };
```



# Creating a thread

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

- Creates a new kernel thread named NAME with the given 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.

# Creating a thread

Allocating a `struct thread` object

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  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;
  list_push_back (&all_list, &t->allelem);
}
```

# Process list

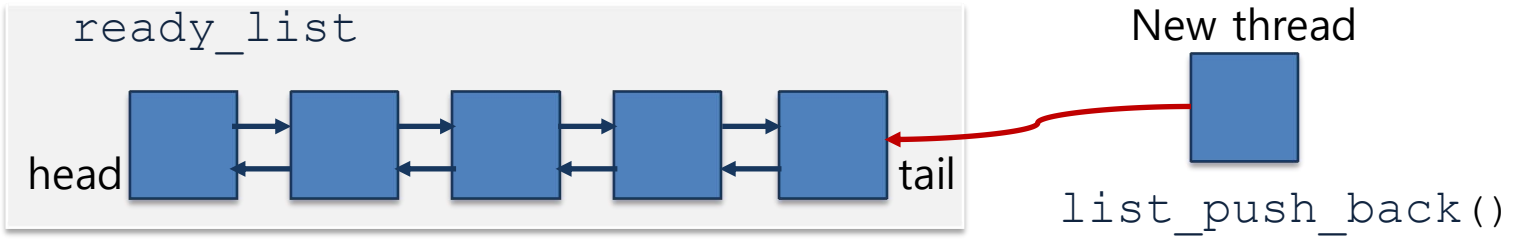
pintos/src/threads/thread.c

```
/* List of processes in THREAD_READY state, that is, processes that
are ready to run but not actually running. */
static struct list ready_list;

/* List of all processes.  Processes are added to this list
   when they are first scheduled and removed when they exit. */
static struct list all_list;
```

- `ready_list`: a set of threads that are ready for execution
- `all_list`: A set of all threads in the system.

# Creating a thread

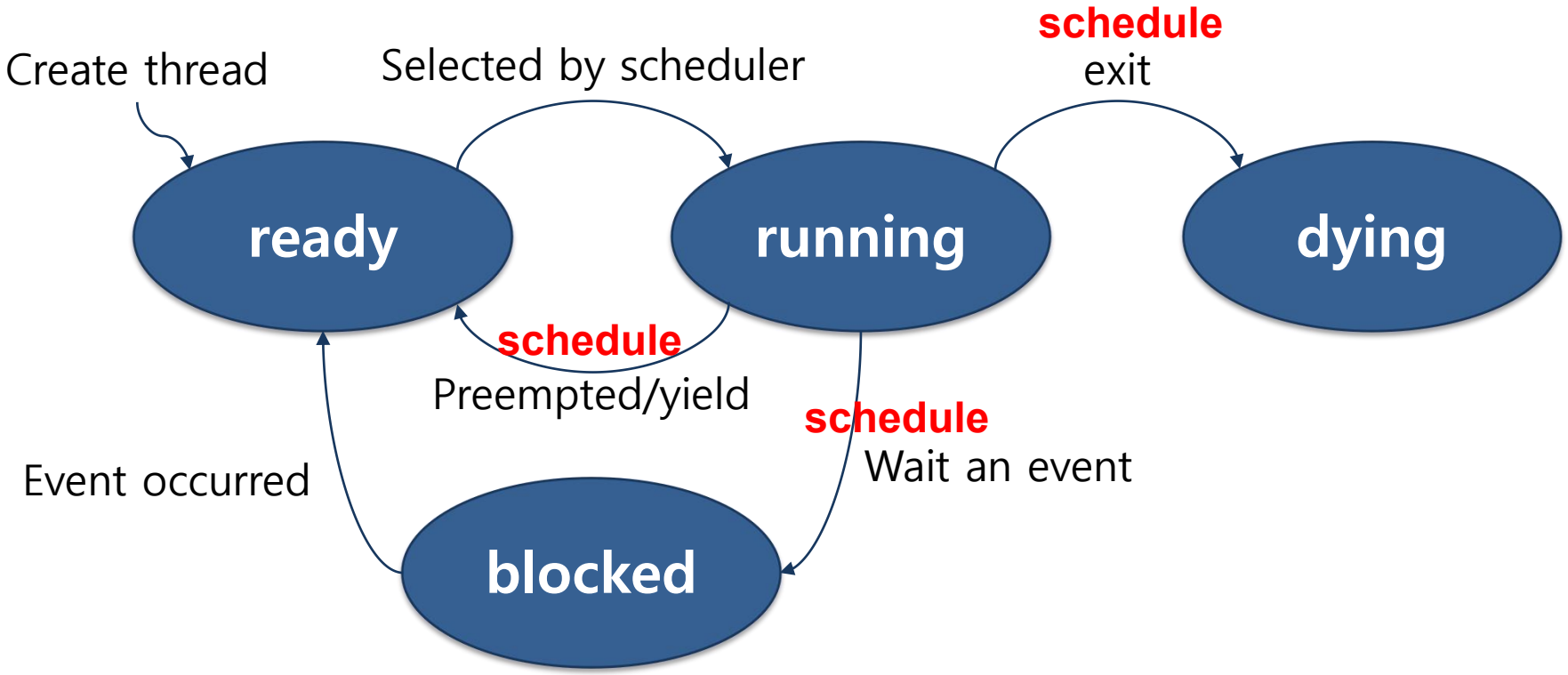
- Newly created thread is inserted at the end of the list `all_list (list_push_back())`
  - `thread_create()`
  - `init_thread`
- After the thread is ready, it is inserted at the `ready_list`.
  - `thread_unblock`



# schedule()

- Schedule a new process.
  - Get the currently running process.
  - Get the next process to run.
  - Switch context from current to next.
- Who calls `schedule()`?
  - exit, block, yield
  - *Or pre-empted, (when the time quantum expires…)*
- Before calling schedule
  - Disable interrupt.
  - Change the state of the running thread from running to something else.

# schedule()


Create thread
Selected by scheduler
schedule
exit
ready
running
dying
schedule
Preempted/yield
schedule
Wait an event
Event occurred
blocked

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

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

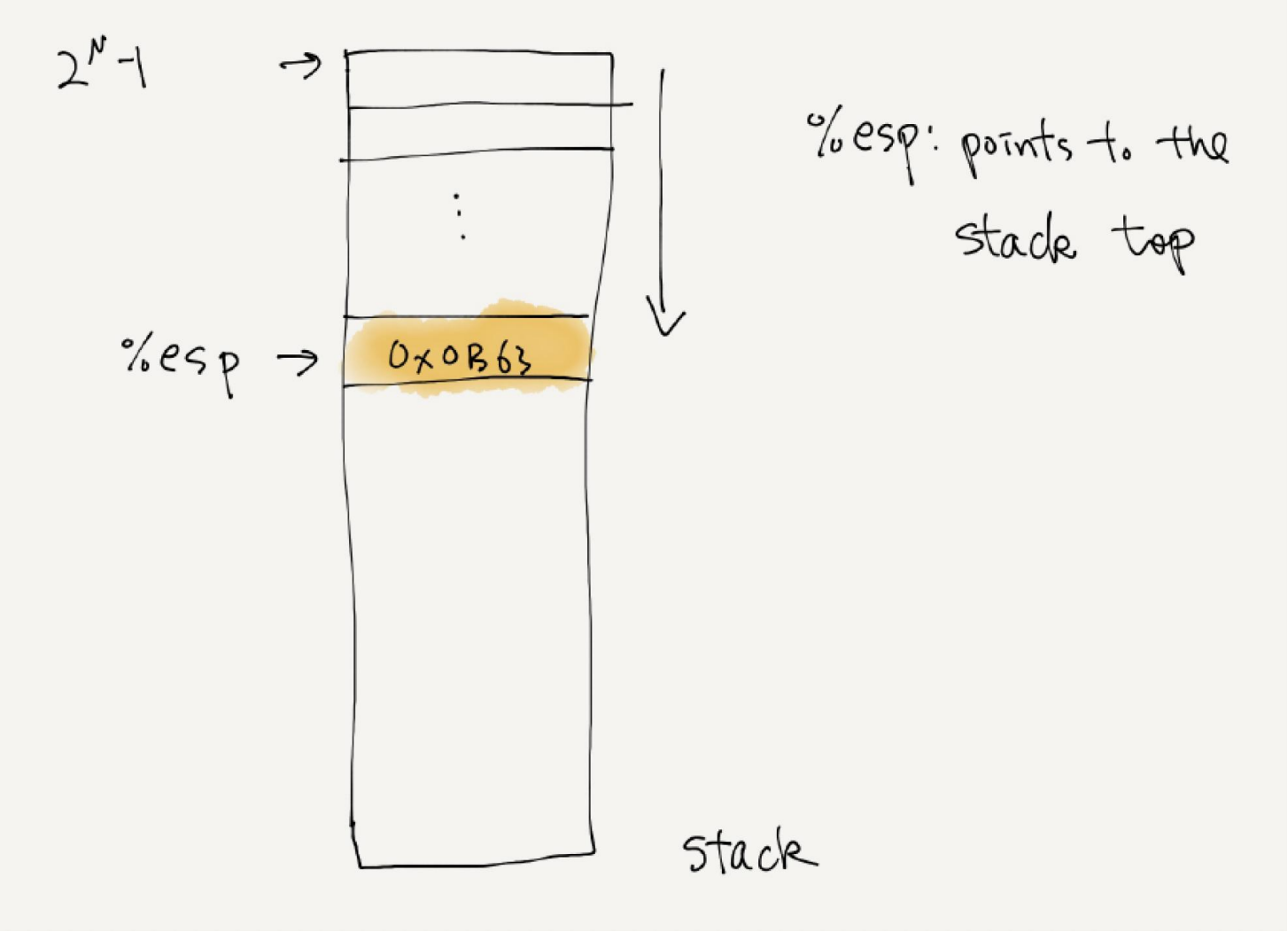
```c
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  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
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 ();
}
```

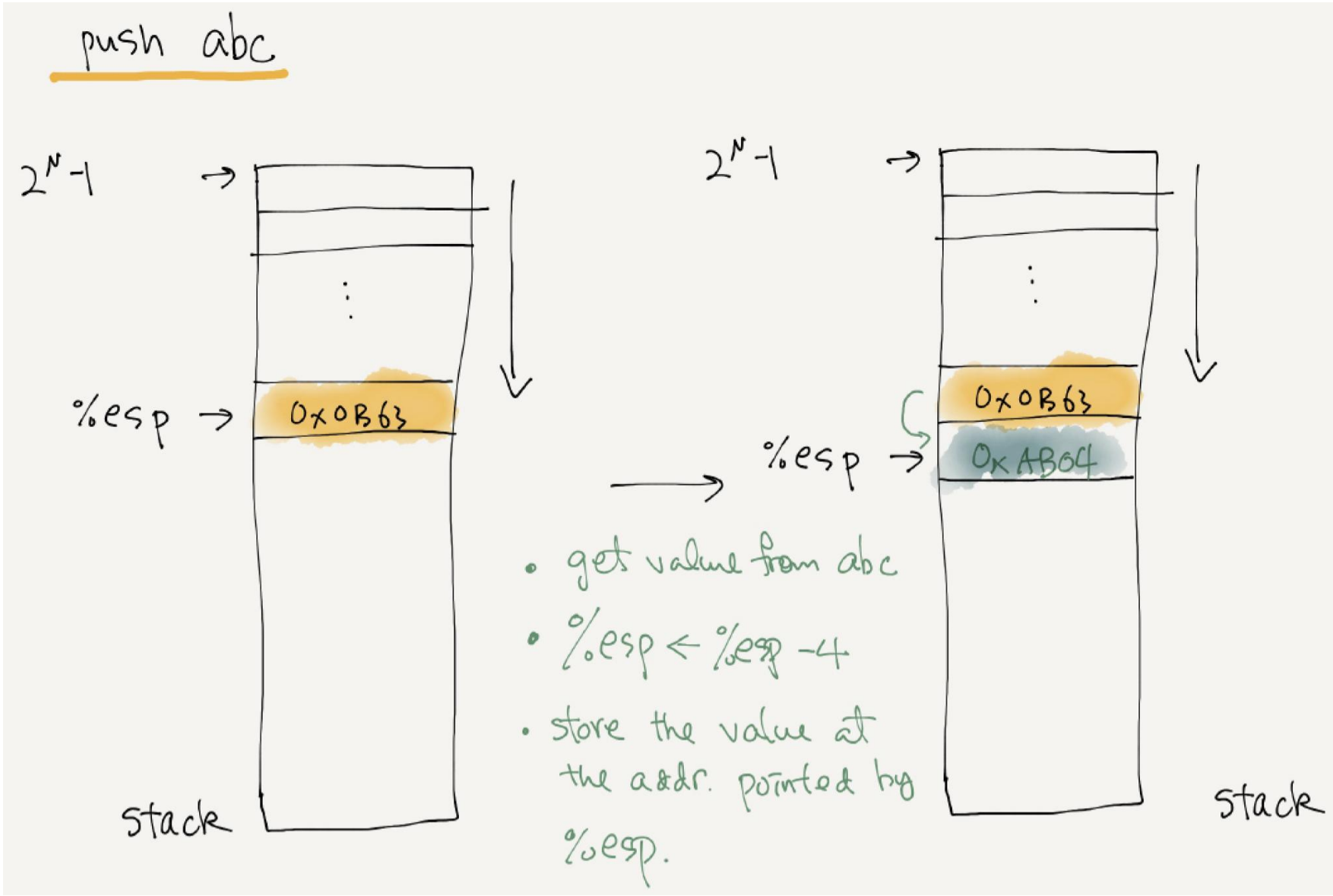
# schedule (void)

```
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);
}
```

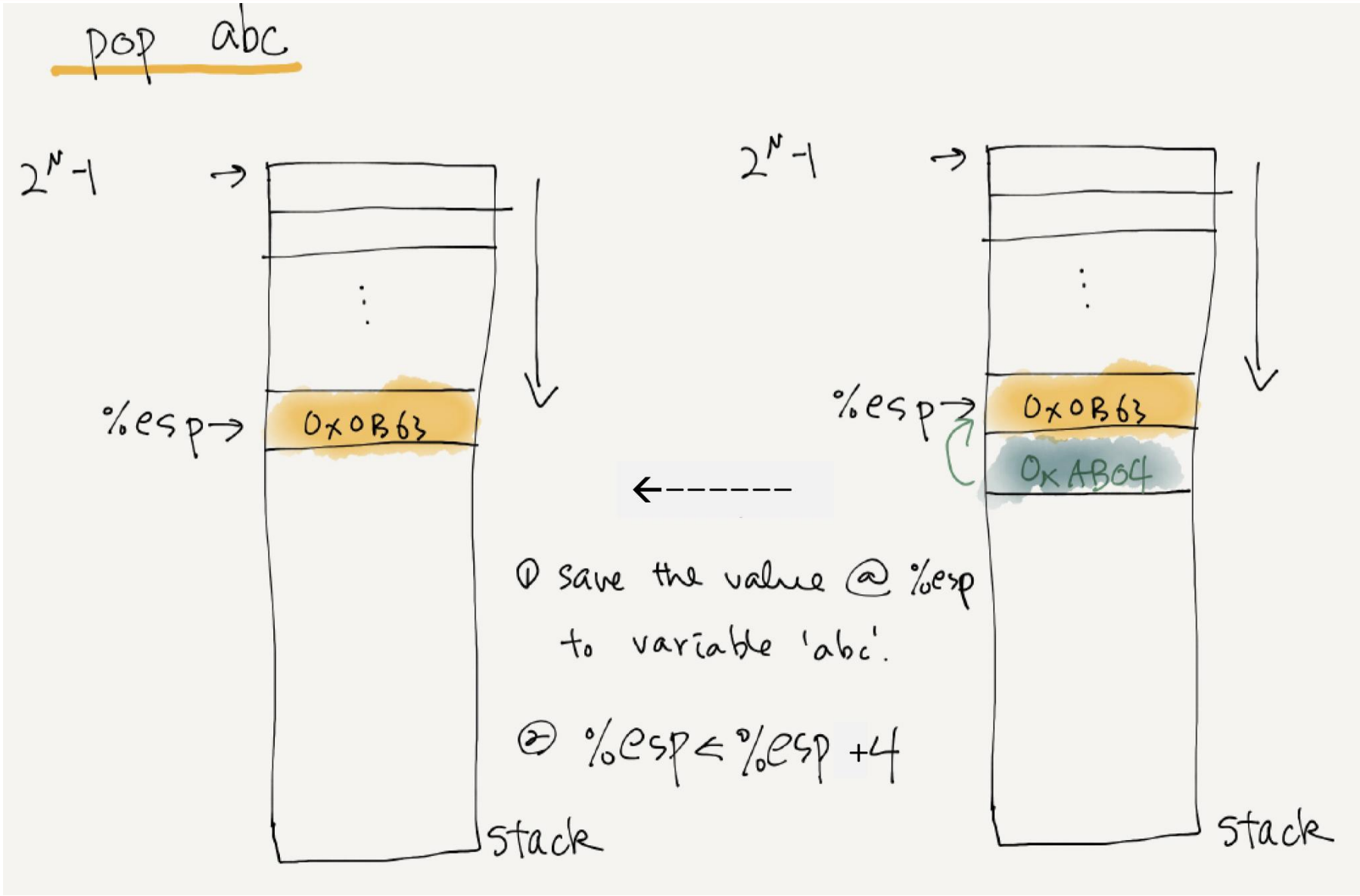
# Stack



# Process context



# Process context

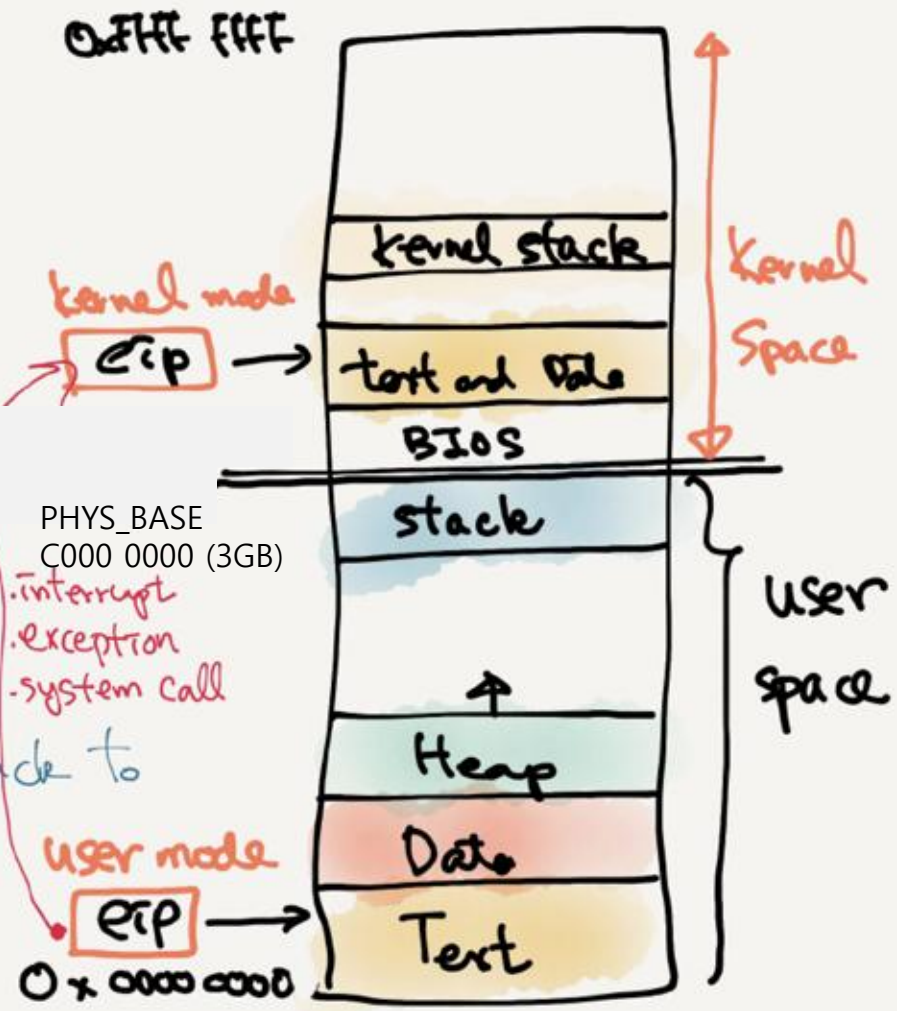


# User stack vs. kernel stack

Executing in User Mode
- User stack

executing in the kernel
- use kernel stack

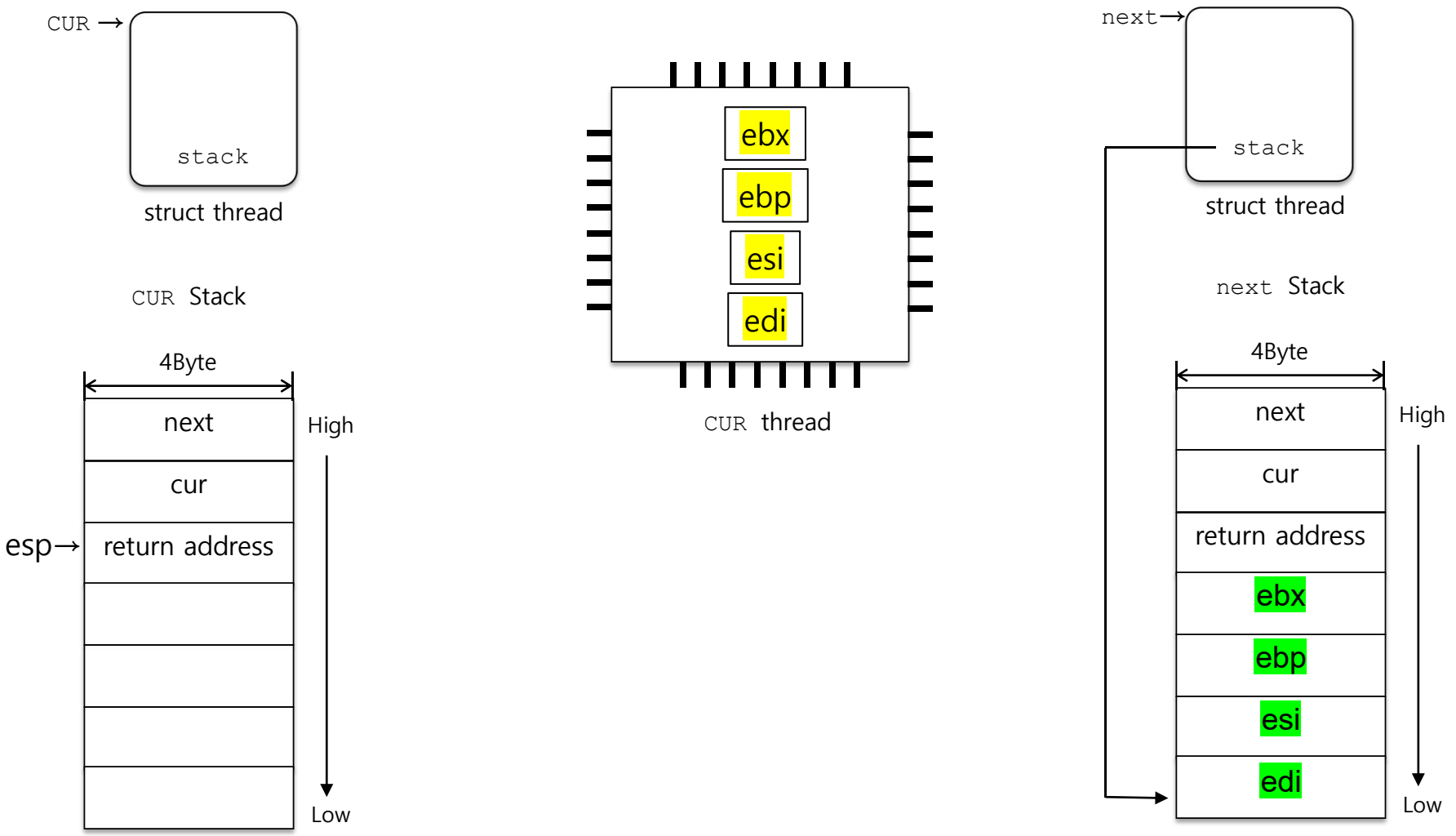
- System Call
  - entering the kernel
  ① switch from user stack to kernel stack
  ② raise privilege level



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

- Save the registers on the kernel stack of `cur`.
- Save the location of the current stack top at the current `struct thread`'s `stack` member.
- Restore the new thread's stack top (kernel stack) into CPU's stack pointer (`esp`).
- Restore registers from the stack (kernel stack).

# Call switch_threads

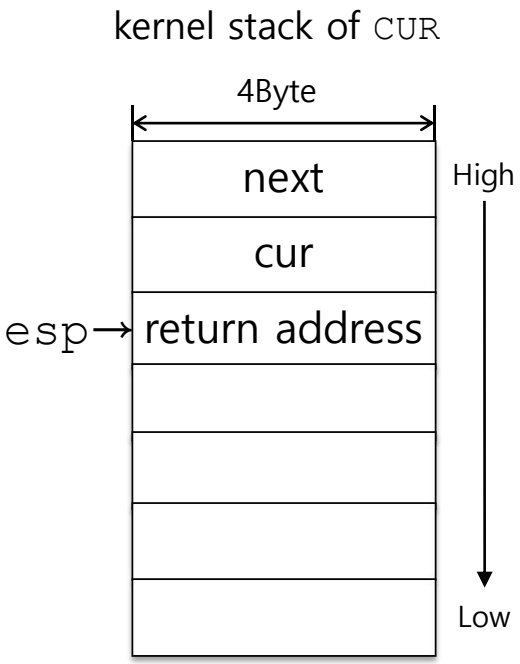


# switch_threads(struct thread *cur, struct thread *next)

- Two tasks
  - Switch the CPU context from CUR thread to NEXT thread.
  - Return `cur` (as `prev` below).

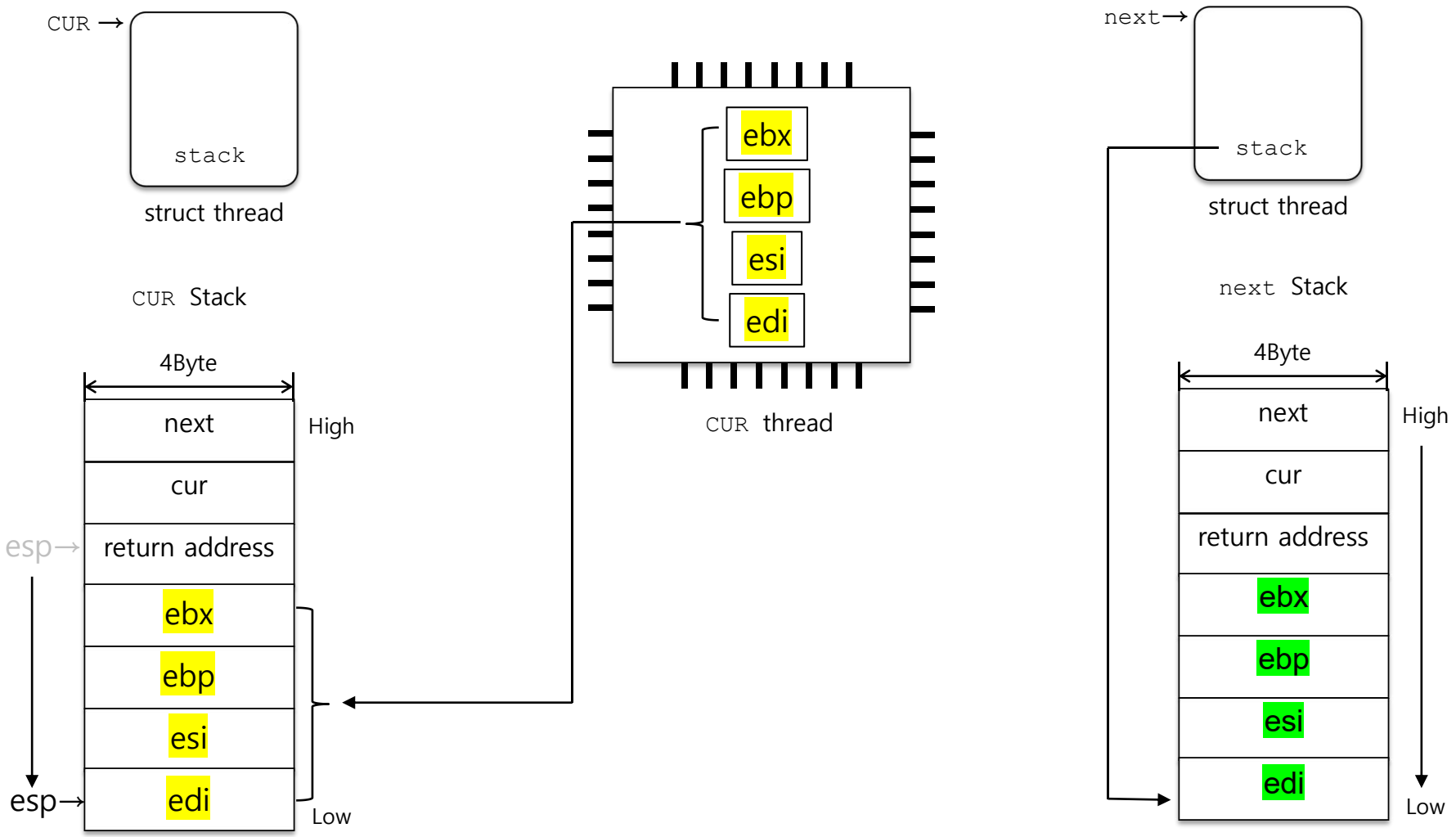
thread/thread.c

```
static void
schedule (void)
{
  ...

  prev = switch_threads(cur, next);

  ...
}
```




Just before jump into `switch_thread`

# Store current context

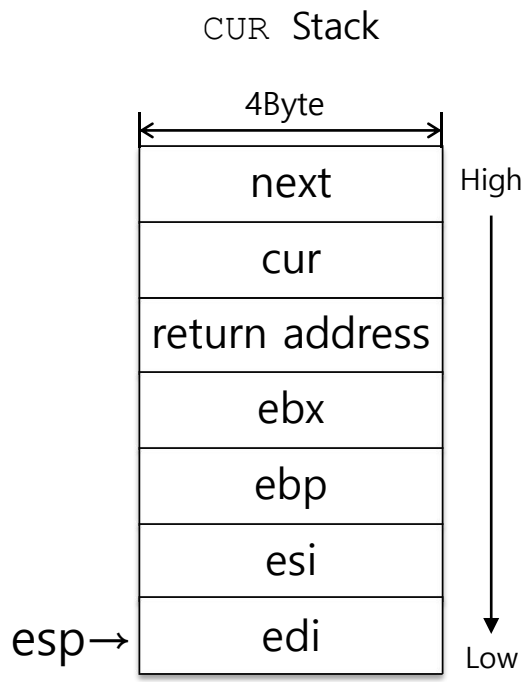


# switch_threads: I

- Save current thread's registers at its kernel stack.

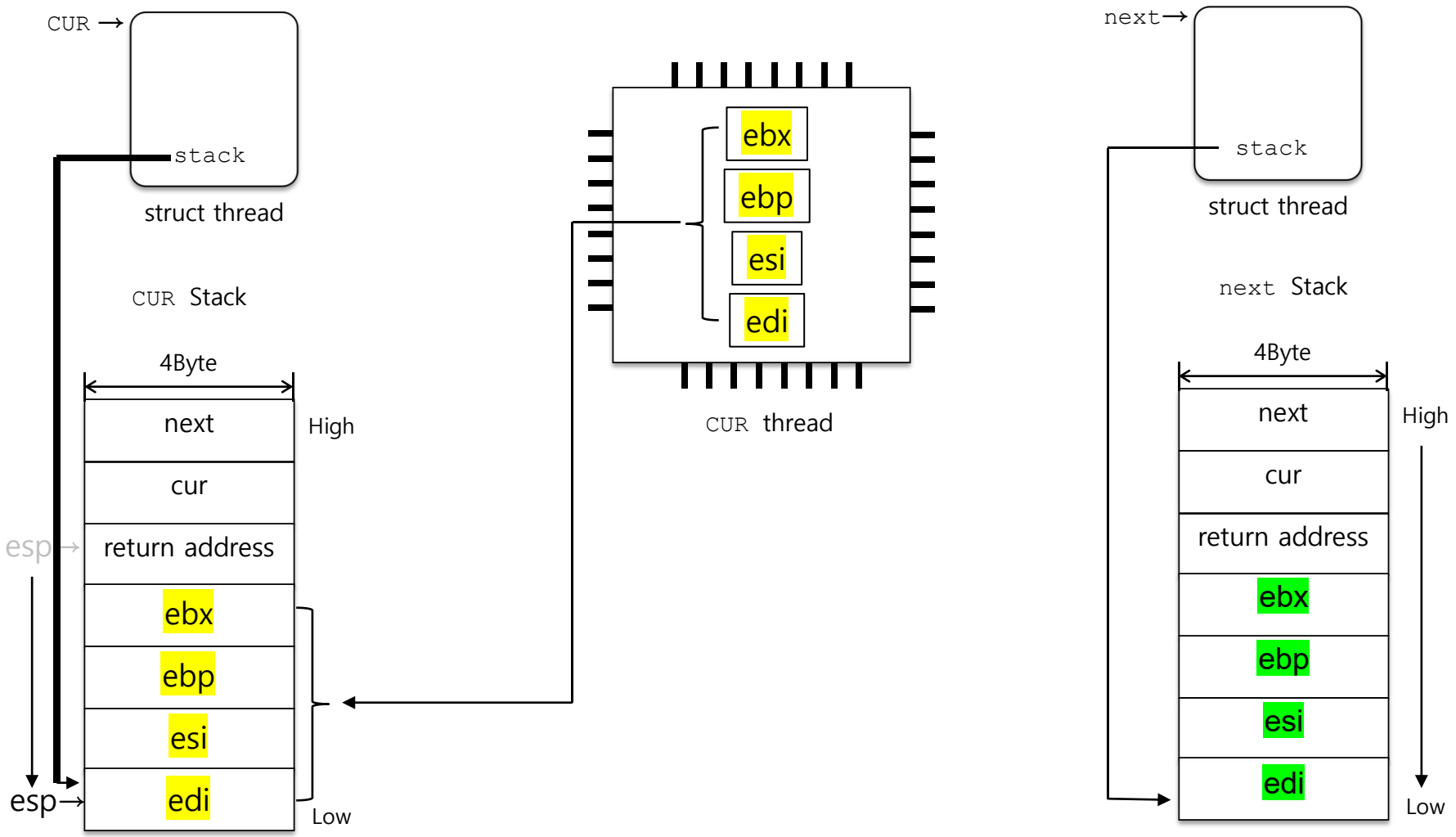
**thread/switch.S**

```
switch_threads:
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi

.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```




After storing the context of `cur` at Its kernel stack

# Store current context



# switch_threads

1. Get the offset of `stack` pointer in `thread` structure.

thread/thread.c

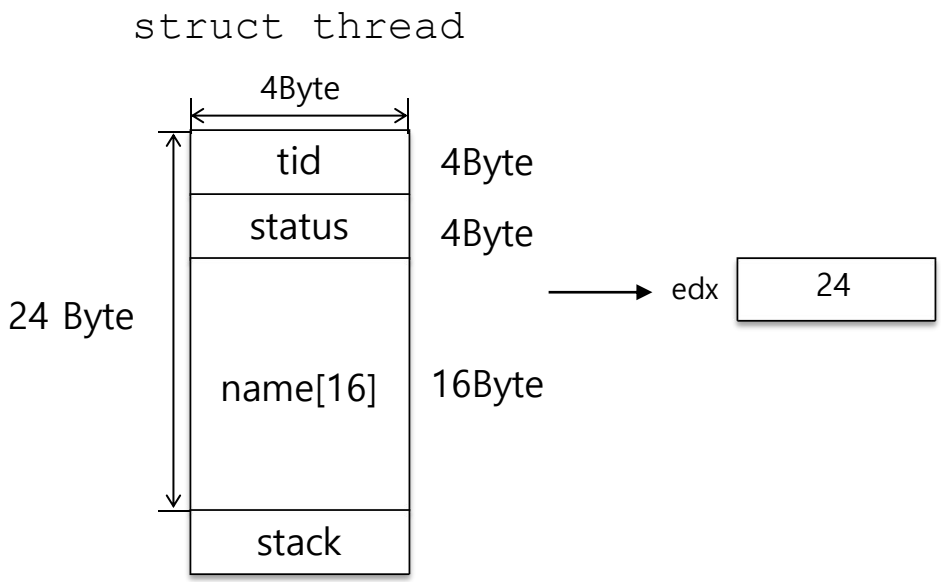
```
uint32_t thread_stack_ofs = offsetof (struct thread, stack)
```

lib/stddef.h

```
#define offsetof(TYPE, MEMBER)((size_t)&((TYPE*)0)->MEMBER)
```

2. Load the offset to `edx`.

```
.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```



# switch_threads(struct thread *cur, struct thread *next)

3. Save the location of the current `struct thread` to `eax`. (return value)

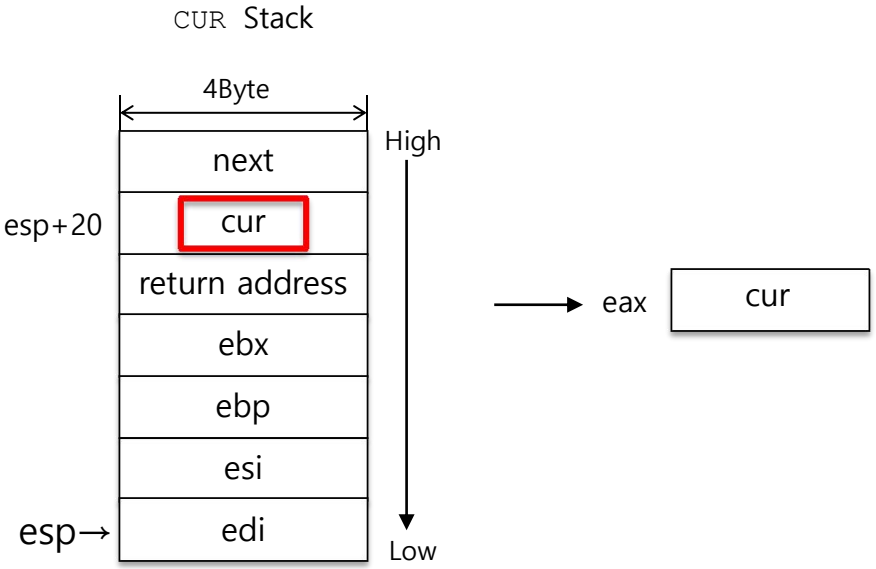
**thread/switch.S**

```
switch_threads:
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi

.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```

**thread/switch.h**

```
#define SWITCH_CUR 20
```

- SWITCH_CUR(%esp) = %esp + 20



# switch_threads

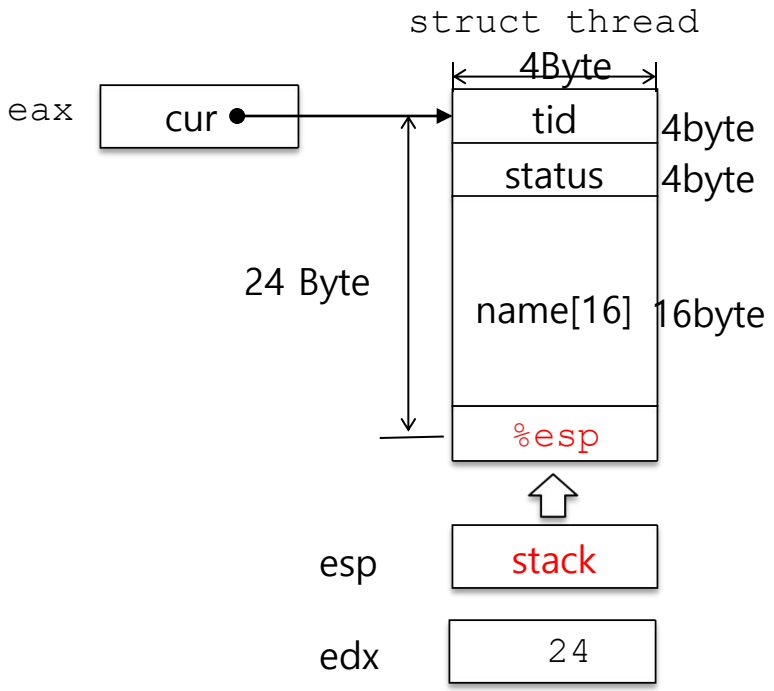
- Save current thread's stack top address at the `struct thread` (`stack` field).

(%eax,%edx,1) = %eax + %edx * 1

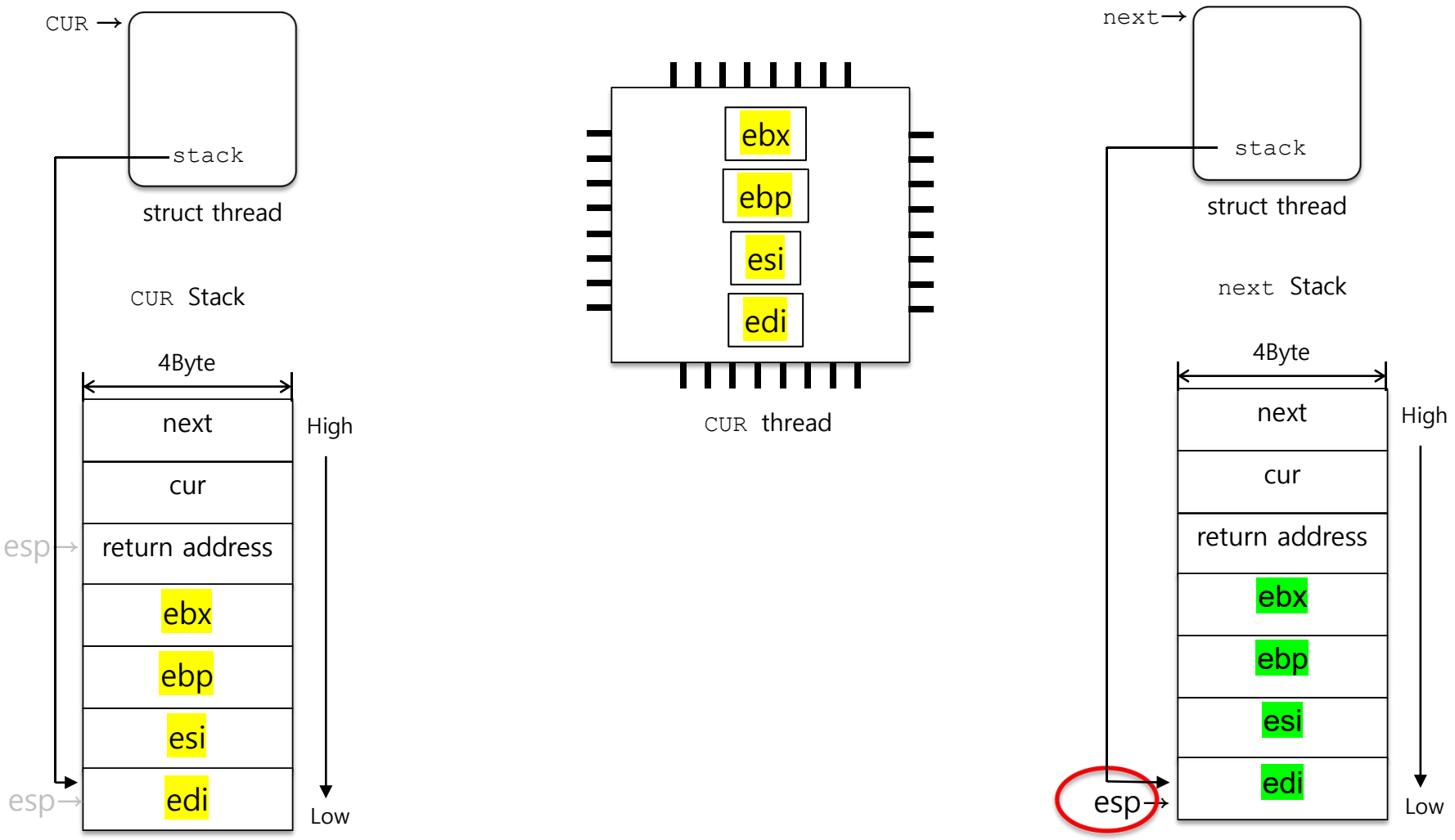
%eax: location of the struct thread of `CUR`

thread/switch.S

```
.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```



# Switch kernel stack



# switch_threads

- Load address of the struct thread of `NEXT` to `ecx`.

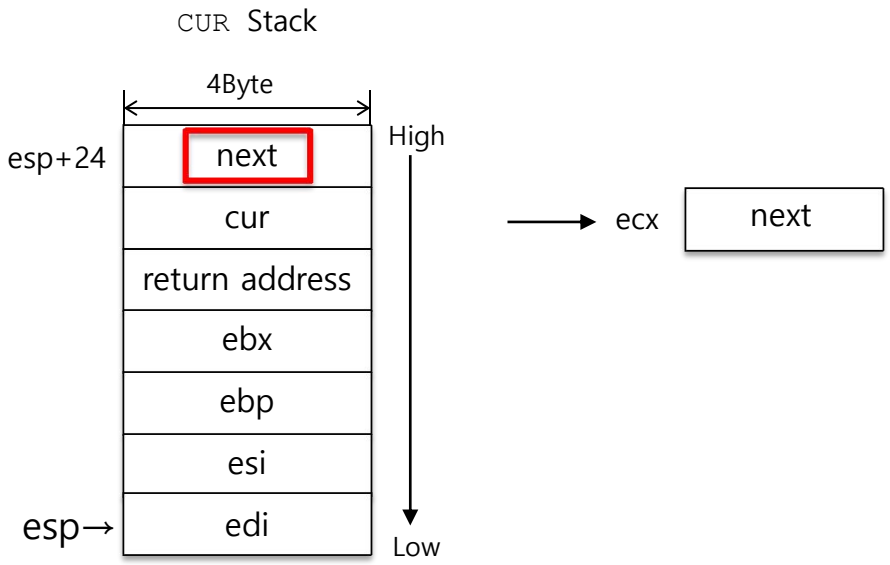
**thread/switch.S**

```
switch_threads:
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi

.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```

**thread/switch.h**

```
#define SWITCH_NEXT 24
```

SWITCH_NEXT(%esp) = %esp + 24



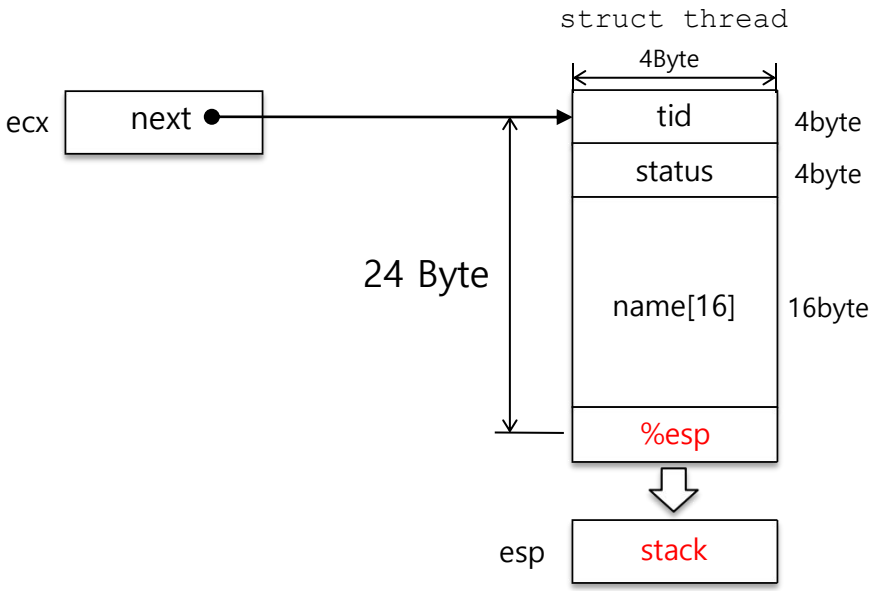
# switch_threads

- Stack switch: Load address of the kernel stack of `NEXT` to `esp`.

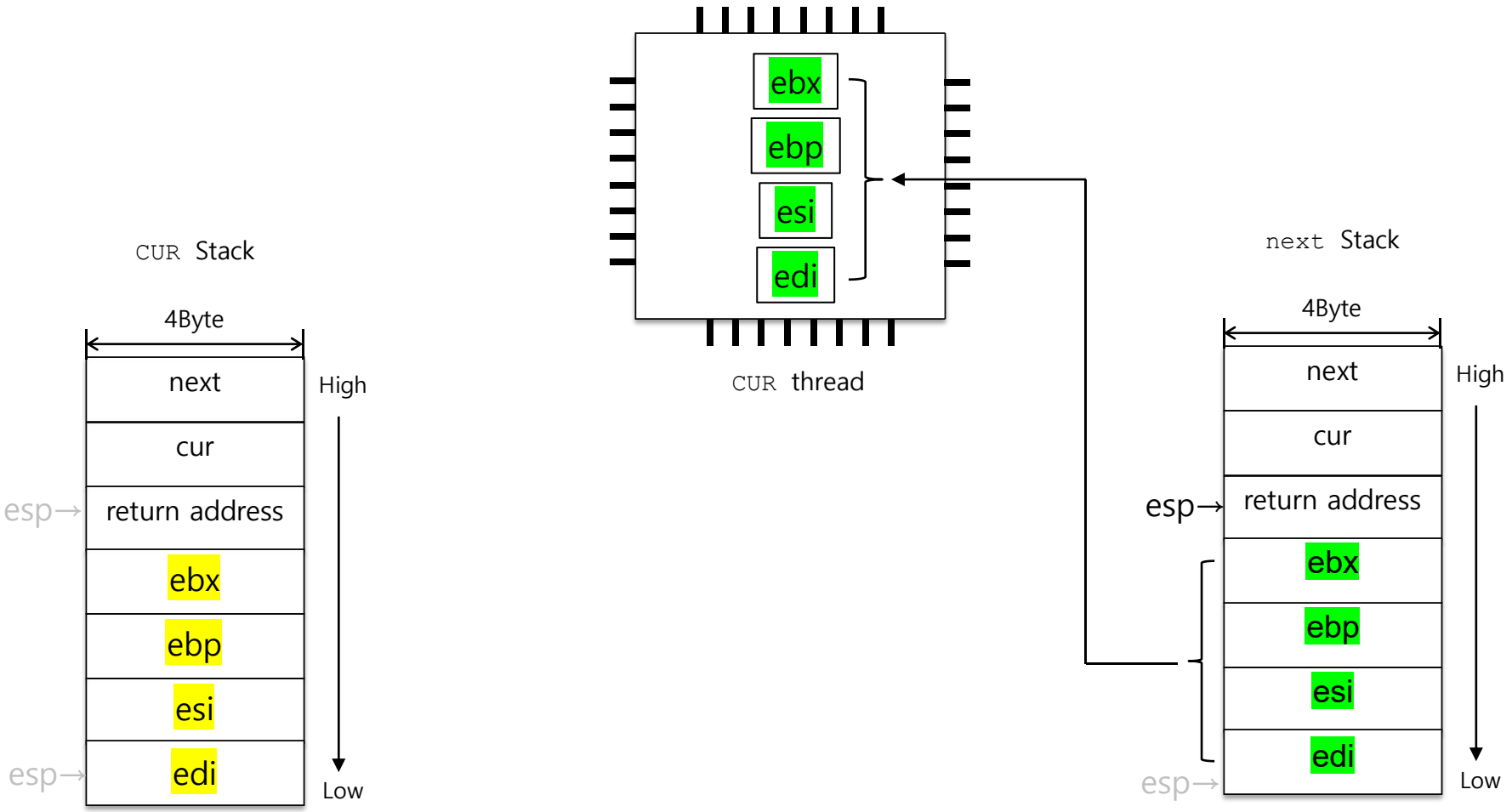
thread/switch.S

```
.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
      ret
```

(%ecx,%edx,1) = %ecx + %edx * 1



# Restore the new context

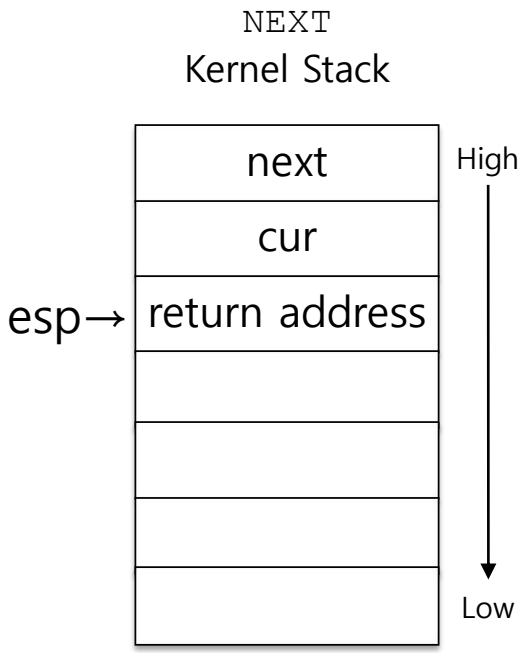


# switch_threads

- Restore next thread's registers.
- Return and run instruction of return address.

**thread/switch.S**

```
switch_threads:
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi

.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  movl SWITCH_CUR(%esp), %eax (return value)
  movl %esp, (%eax,%edx,1)

  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
  ret
```



# thread_schedule_tail()

- Mark the new thread as running.
- If the previous thread was in the dying state, then it also frees the pag e.

# Change the state of new current

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

  ASSERT (intr_get_level () == INTR_OFF);

cur->status = THREAD_RUNNING;

thread_ticks = 0;

#ifdef USERPROG
process_activate ();
#endif

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

# Summary

- schedule()
  - Called in exit, yield and block.
  - Get the new process to the CPU.
- Context switch
  - Save the context of the currently running thread to the stack.
  - Save the current stack top at the currently running `struct thread`.
  - Restore the stack top of the next thread to esp register.
  - Restore the context from the stack of the next thread to run.
- Change the state of the next process to running and frees the memory fro m the dying process.