

# CS 378 (Spring 2003)

## Linux Kernel Programming

**Yongguang Zhang**

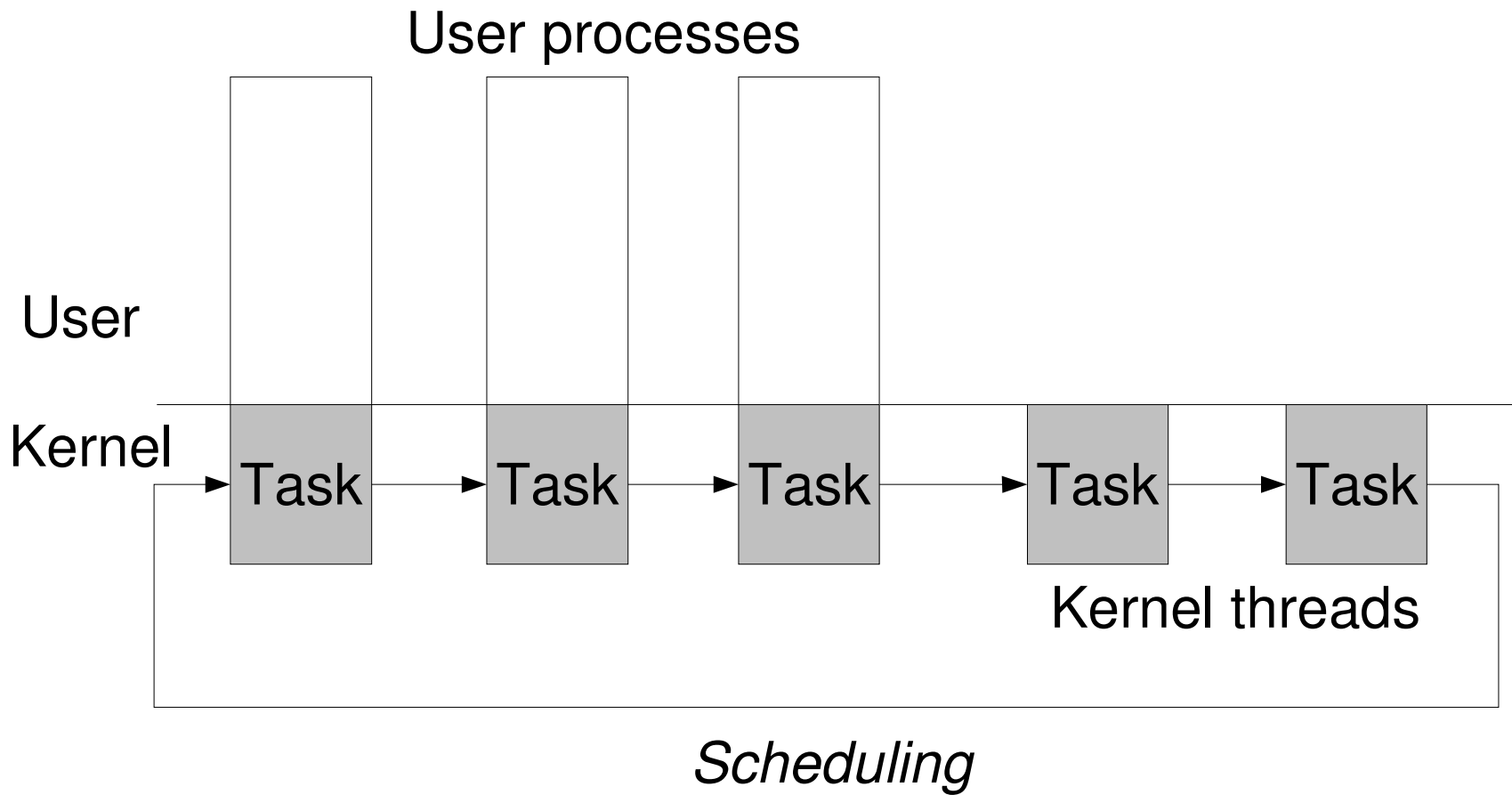
**([ygz@cs.utexas.edu](mailto:ygz@cs.utexas.edu))**

# This Lecture

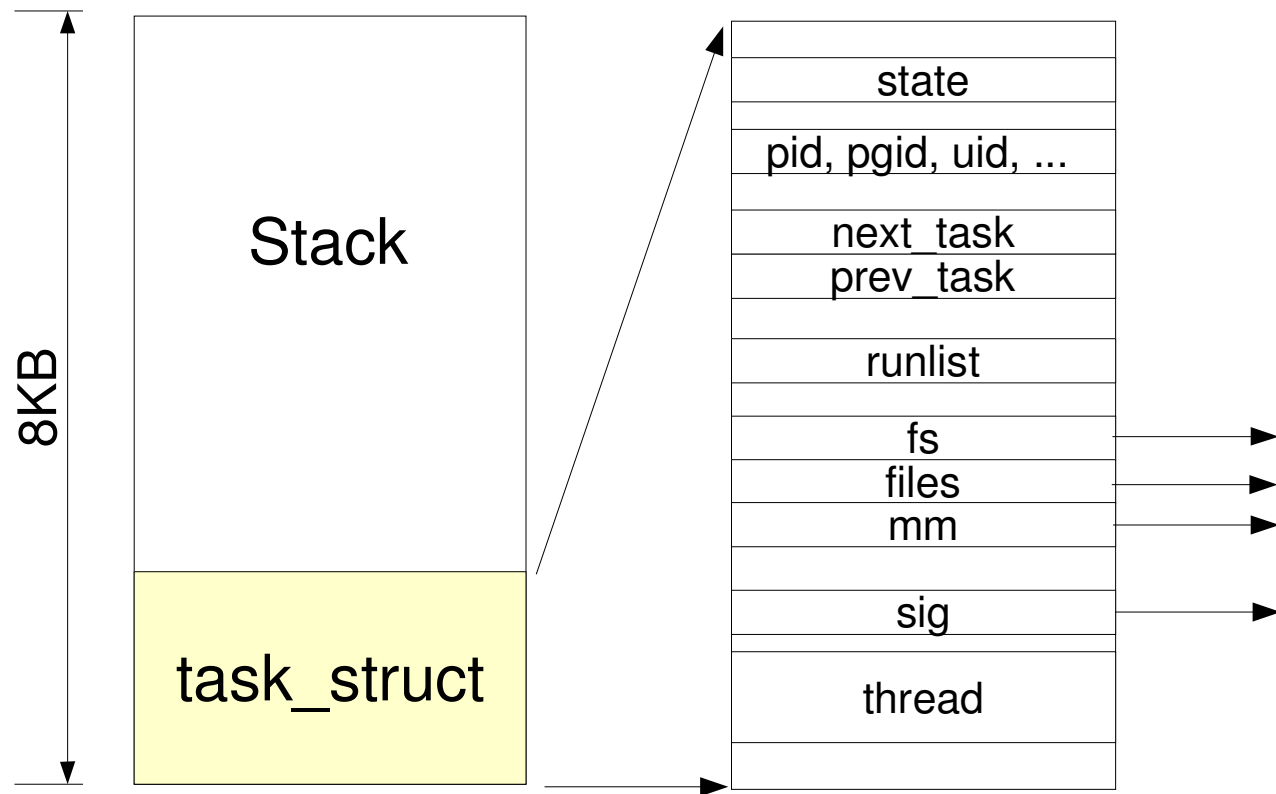
---

- Last Lecture: Process Management
- This Lecture: More Process Management
  
- Questions?

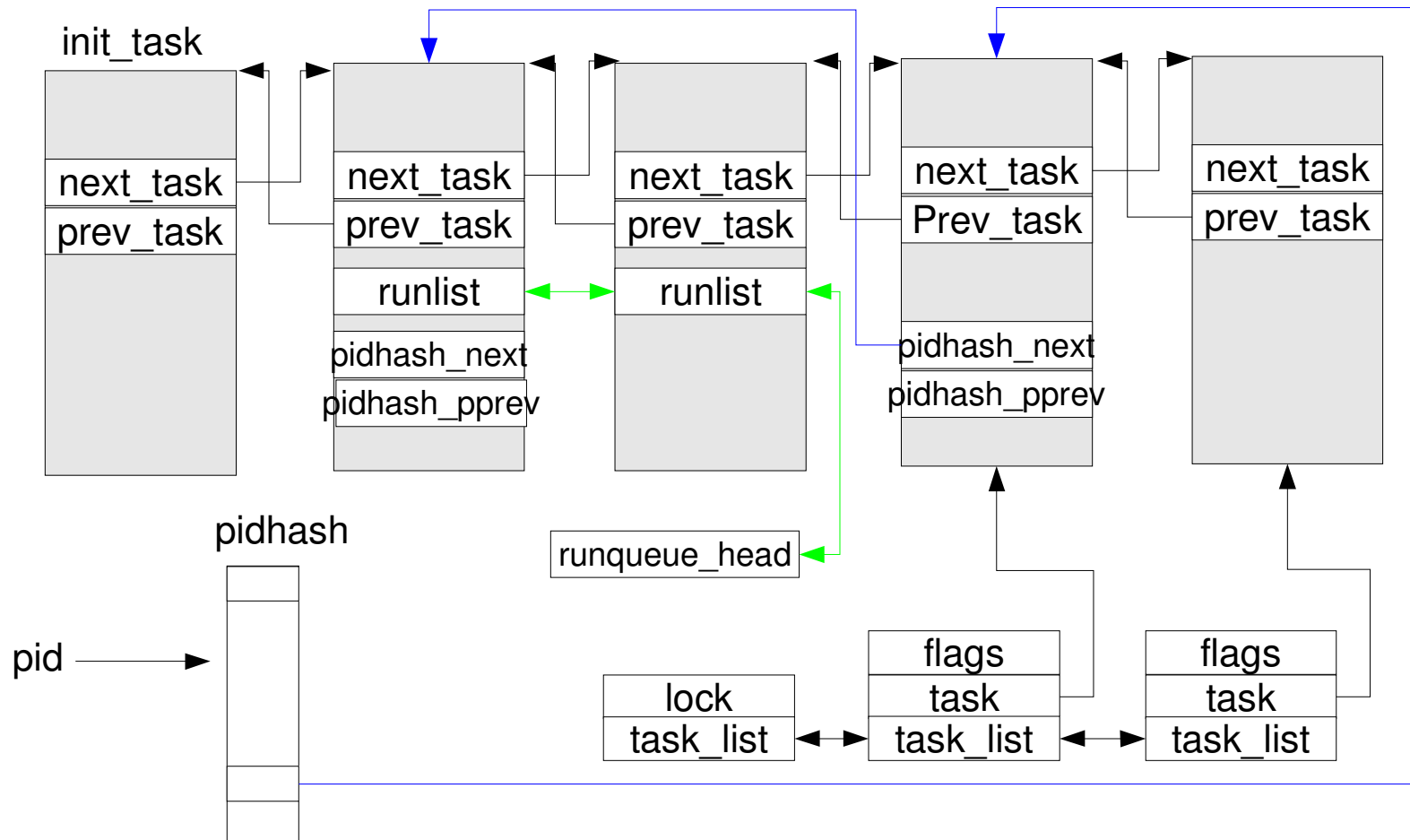
# Managing Process/Thread



# Task Descriptor



# Task Lists



# Run Queue

---

- Variables
  - `runqueue_head`
  - `nr_running`
- Functions (`kernel/sched.c`)
  - `add_to_runqueue(struct task_struct * p),`  
`del_from_runqueue(struct task_struct * p)`
  - `wake_up_process(struct task_struct * p)`

# Wait Queues

---

- Different types of wait
  - Exclusive and non-exclusive waits
  - Interruptible and non-interruptible
- Define a new wait queue
  - `DECLARE_WAIT_QUEUE_HEAD(...)`
- Functions:
  - `add_wait_queue()`, `remove_wait_queue()`
  - `sleep_on` macros
  - `wake_up` macros

# Creating Process

---

- `fork()` System Call
  - Duplicate the entire process: its virtual memory and all per-process kernel resources
  - “Heavyweight”
  - Often wasteful if followed immediately by `execve()` , which releases all these resources and creates its own
- `clone()` System Call
  - Selectively duplicate the process resources:
    - VM, FS, FILES, and/or SIGHAND
  - “Lightweight”

# do\_fork()

---

- Called by `sys_clone()`, `sys_fork()`, `sys_vfork()`

- Defined in `kernel/fork.c`:

```
p = alloc_task_struct();
```

```
Lots of checking and filling in *p;
```

```
copy_files(clone_flags, p)
```

```
copy_fs(clone_flags, p)
```

```
copy_sighand(clone_flags, p)
```

```
copy_mm(clone_flags, p)
```

```
copy_thread(0, clone_flags, stack_start, stack_size, p,  
regs);
```

```
More checking and filling in *p;
```

```
wake_up_process(p);
```

# Context Switch

---

- Suspend the Execution of Current Process (running on CPU) and Resume the Execution of Another Process
  - Save the current context (hardware and software states) in the process descriptor or in stack
  - Load the next context
- Context
  - Hardware: CPU states
  - Software: Virtual Memory Management (Page Tables)

# Hardware Context

---

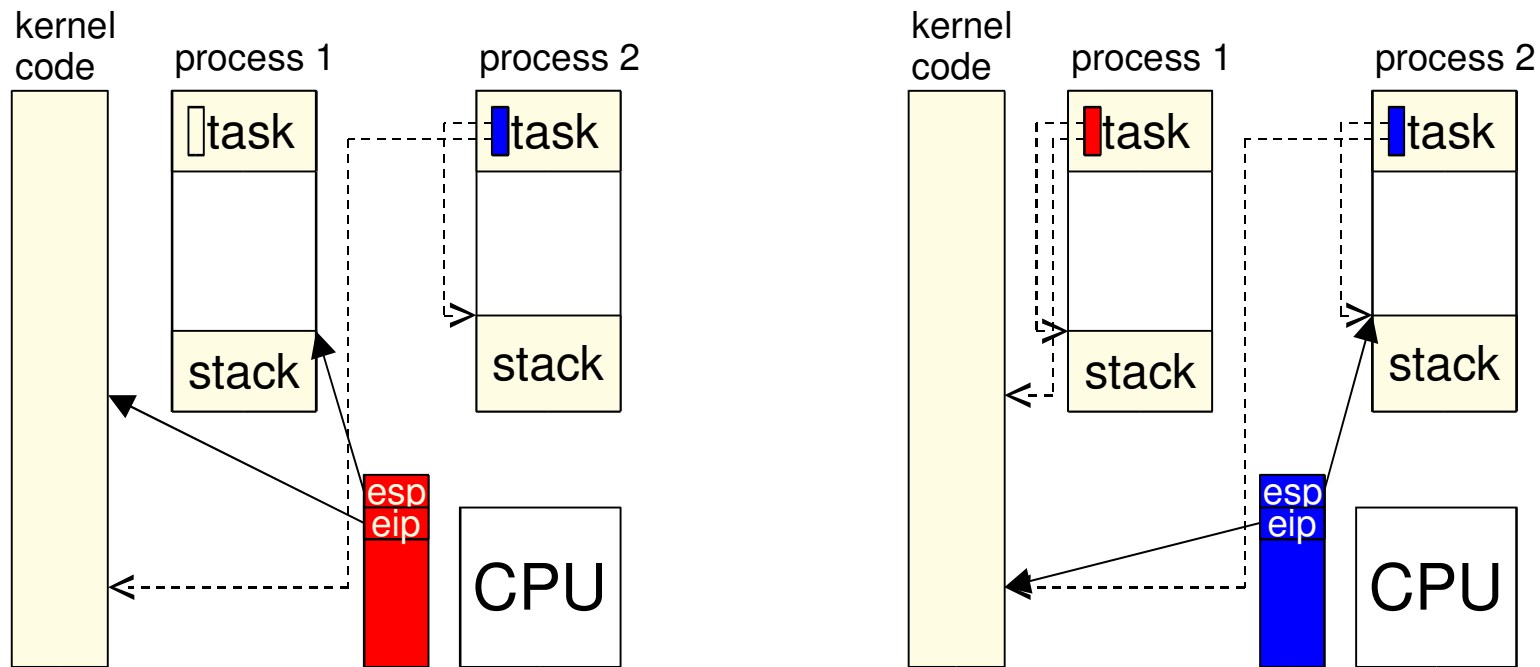
- Remember struct `thread_struct` ?
  - Hardware Registers
    - unsigned long `esp0, eip, esp, fs, gs`;
  - Hardware debugging registers
    - unsigned long `debugreg[8]`;
  - Fault info
    - unsigned long `cr2, trap_no, error_code`;
  - Floating point info
    - union `i387_union i387`;
  - Virtual 86 mode info
  - IO permissions

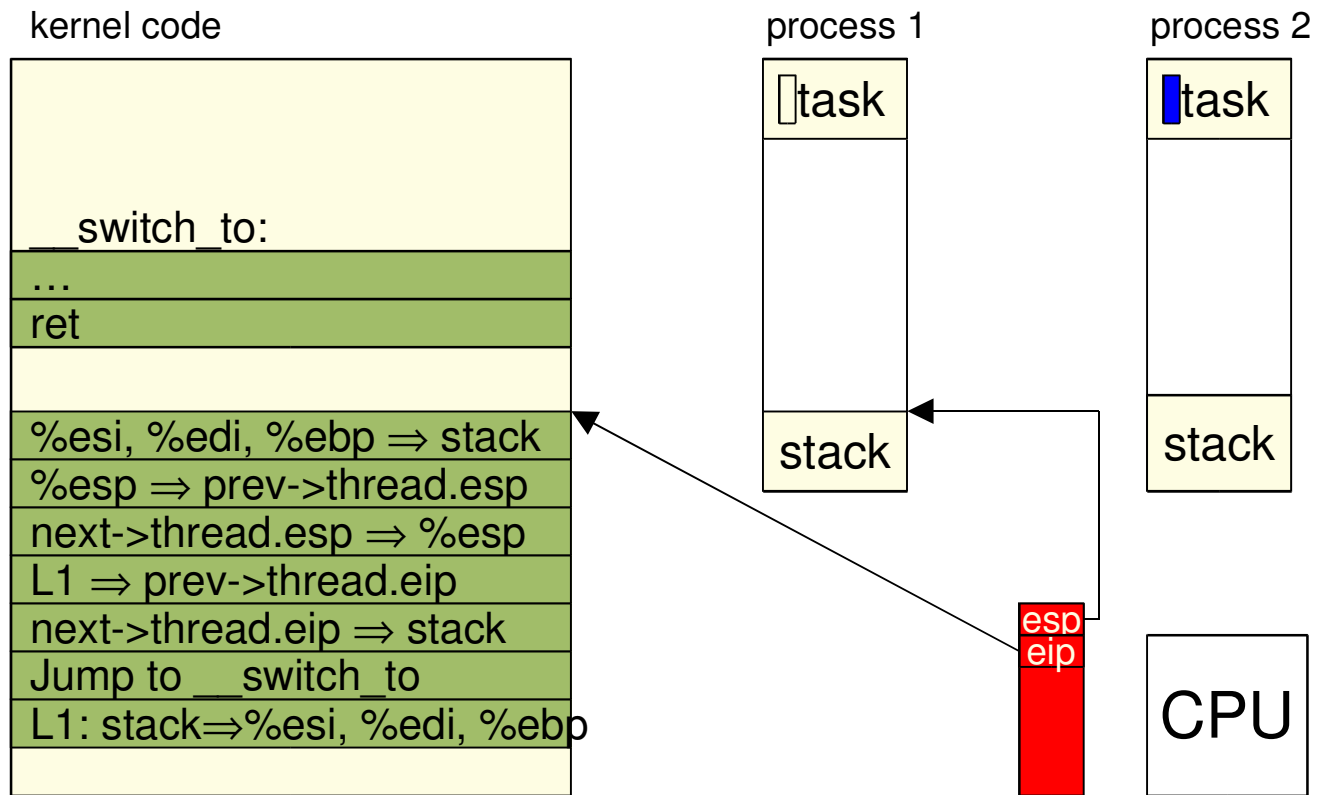
# switch\_to()

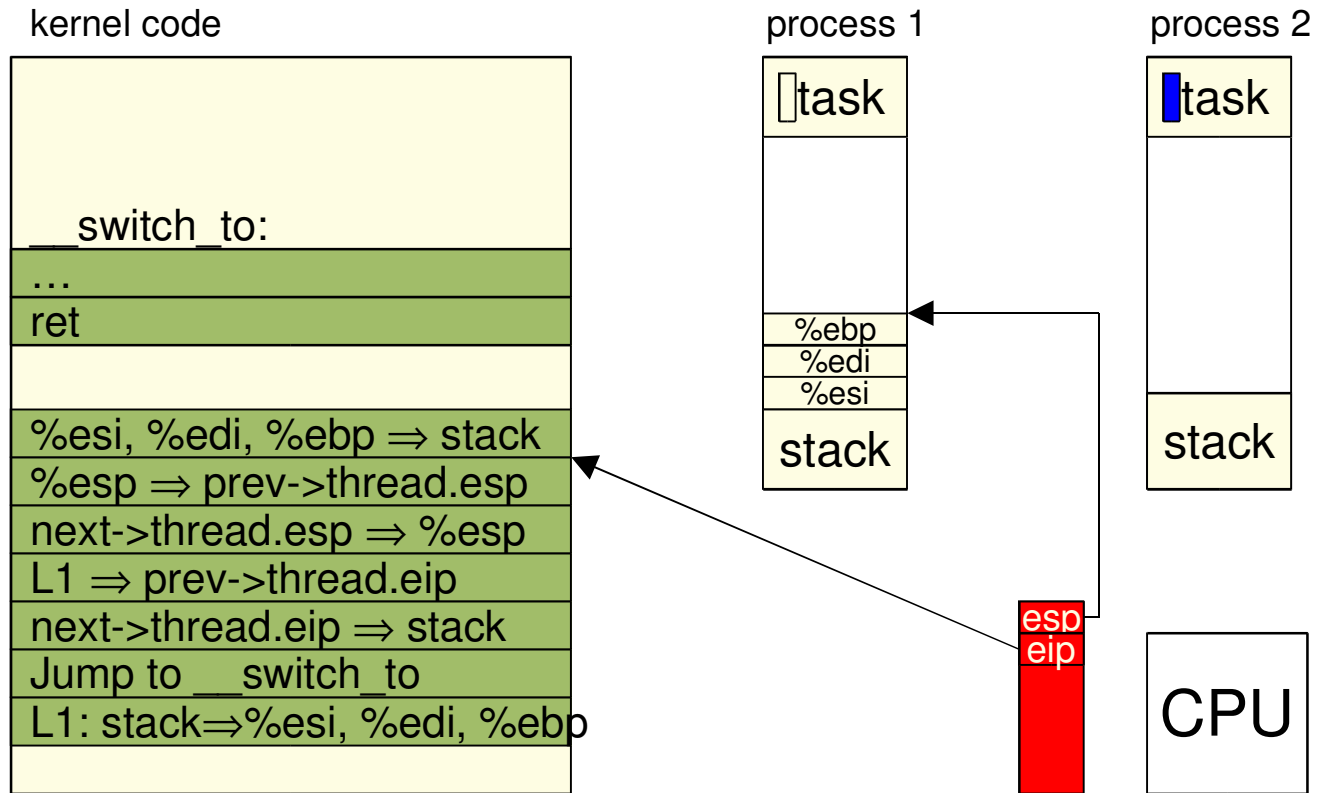
- A macro to switch context from `prev` to `next`
  - Defined in `include/asm-i386/system.h`
- Written in assembly, but essentially:
  - `%esi, %edi, %ebp`  $\Rightarrow$  stack
  - `%esp`  $\Rightarrow$  `prev->thread.esp`
  - `next->thread.esp`  $\Rightarrow$  `%esp`
  - Label 1 address  $\Rightarrow$  `prev->thread.eip`
  - `next->thread.eip`  $\Rightarrow$  stack
  - Jump to `__switch_to()` (exchange rest of the context)
    - End with `ret` (stack  $\Rightarrow$  `%eip`)
  - Label 1: stack  $\Rightarrow$  `%esi, %edi, %ebp`

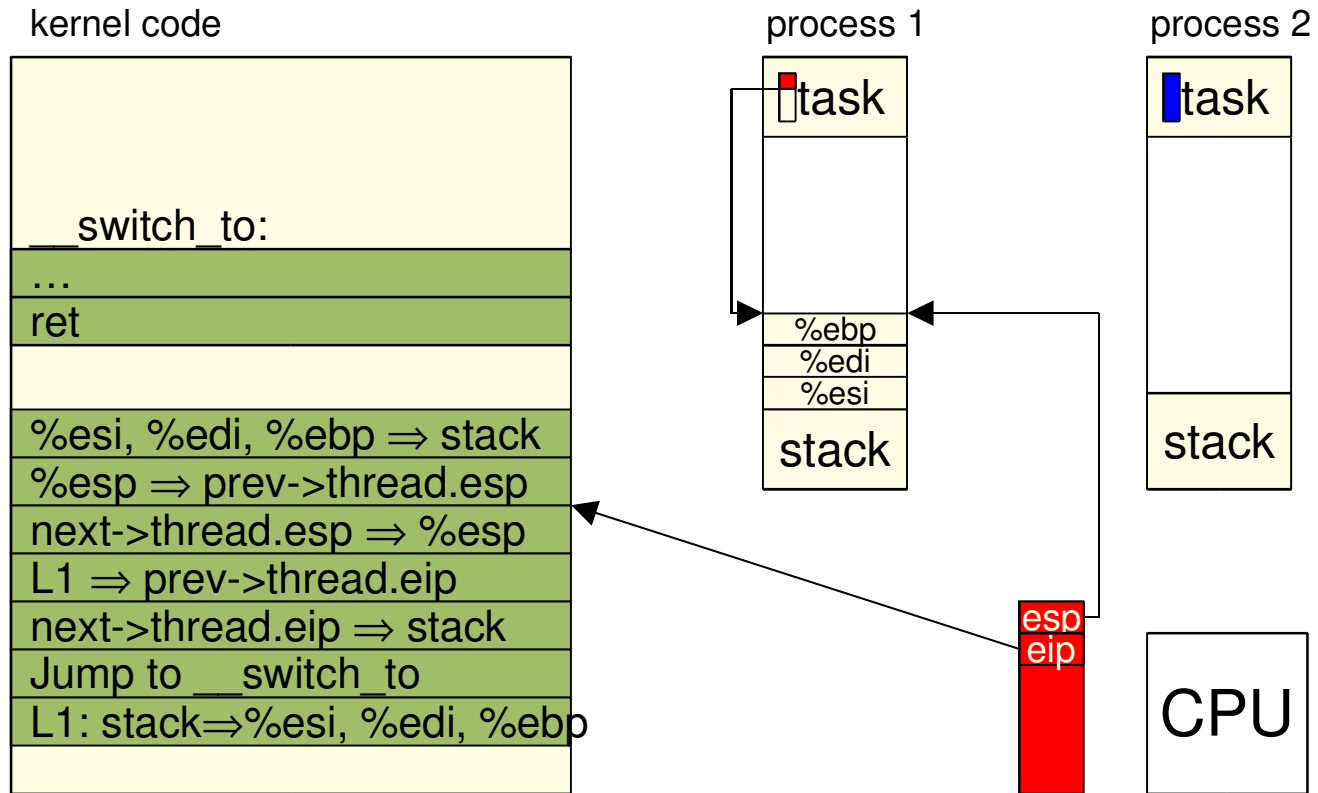
# How Exactly Does it Work?

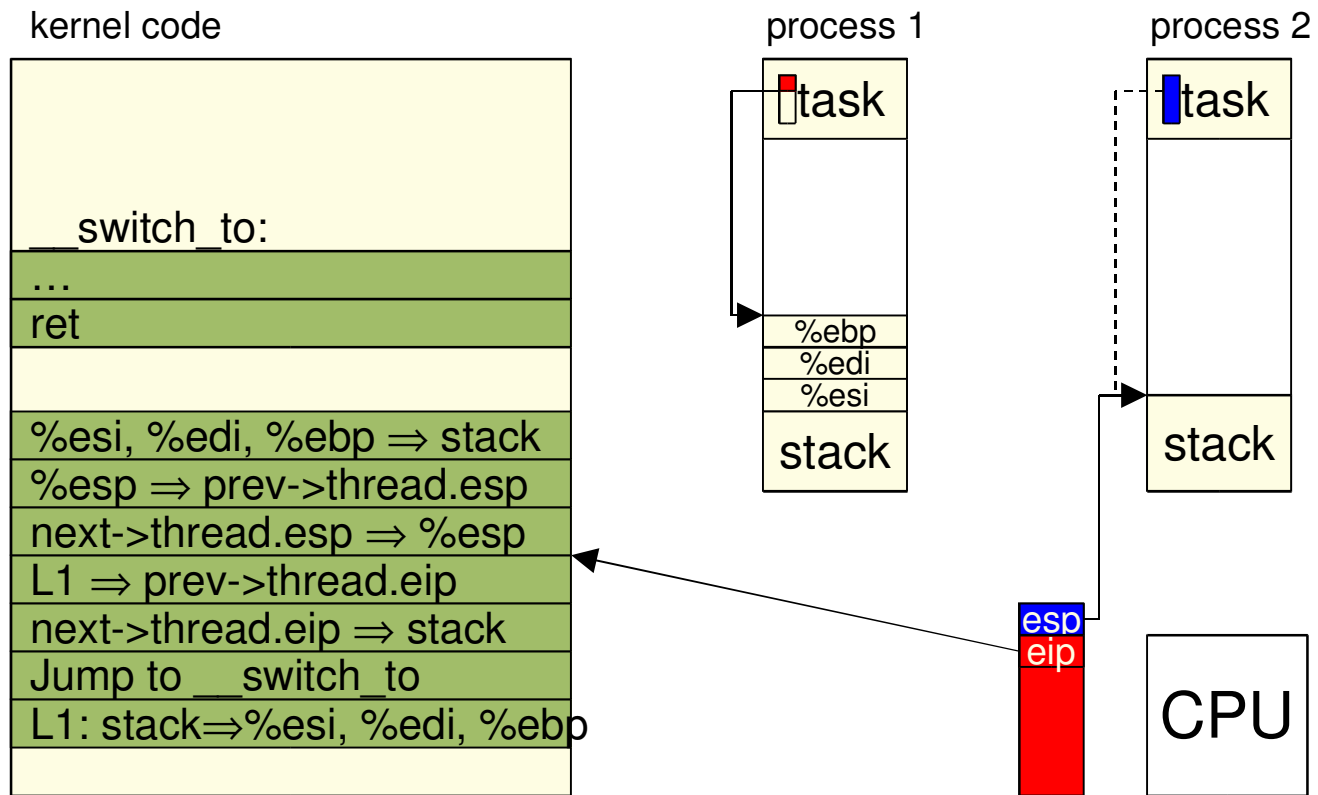
- Register ESP: stack pointer
- Register EIP: code pointer

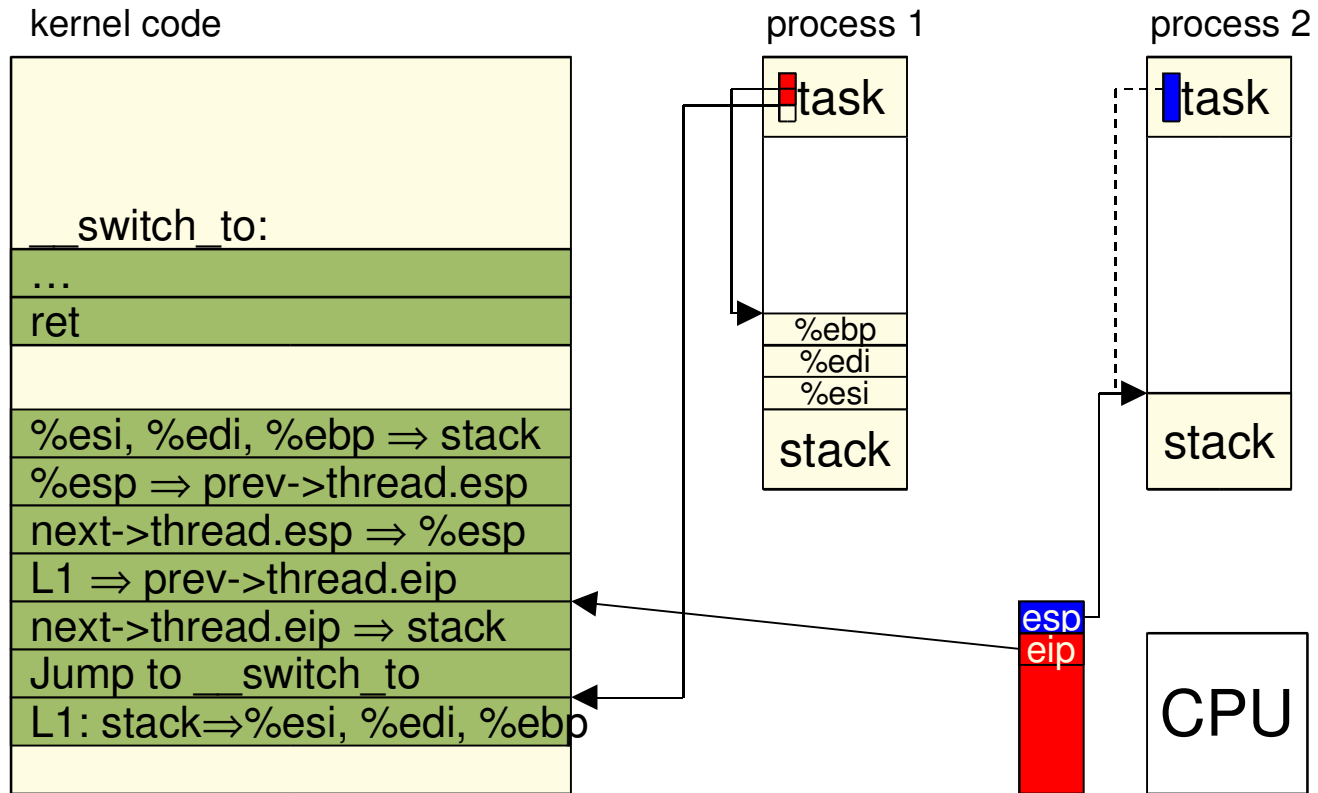


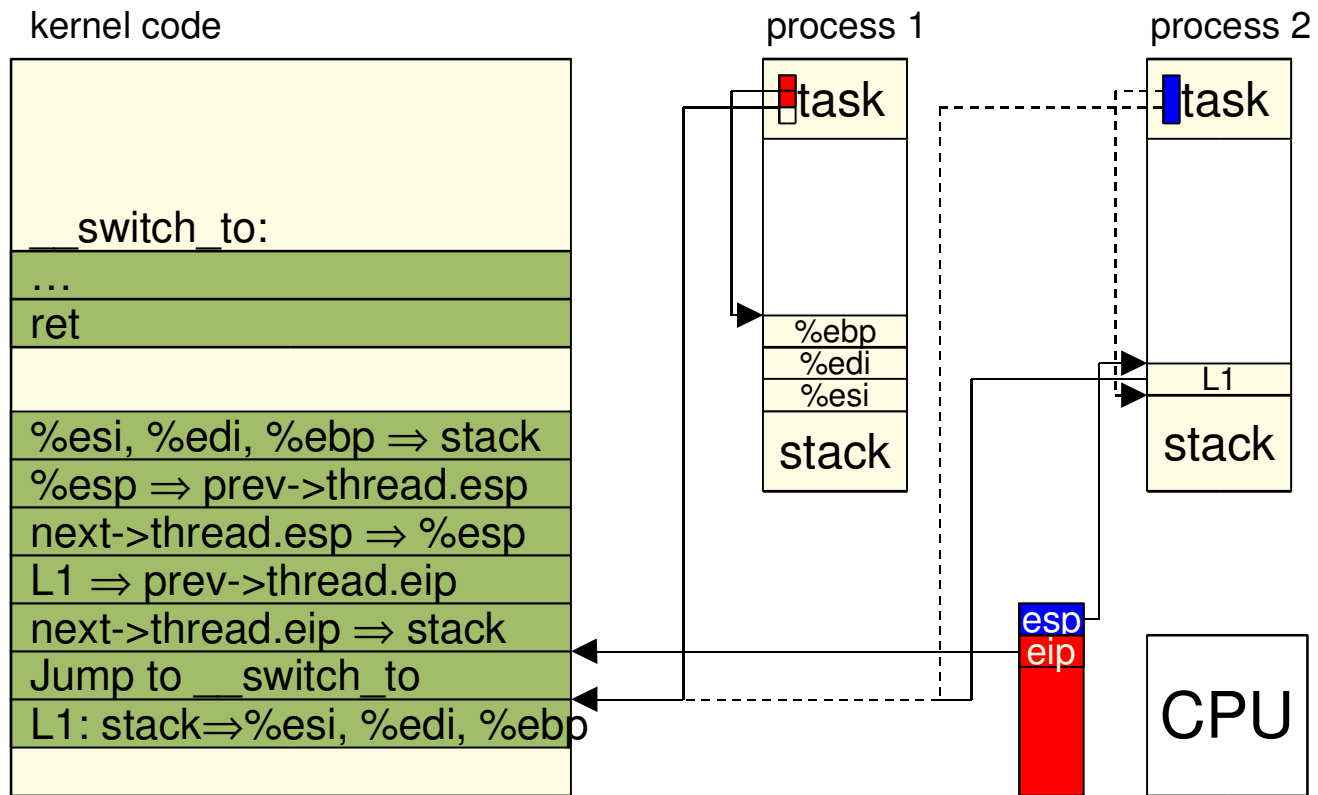


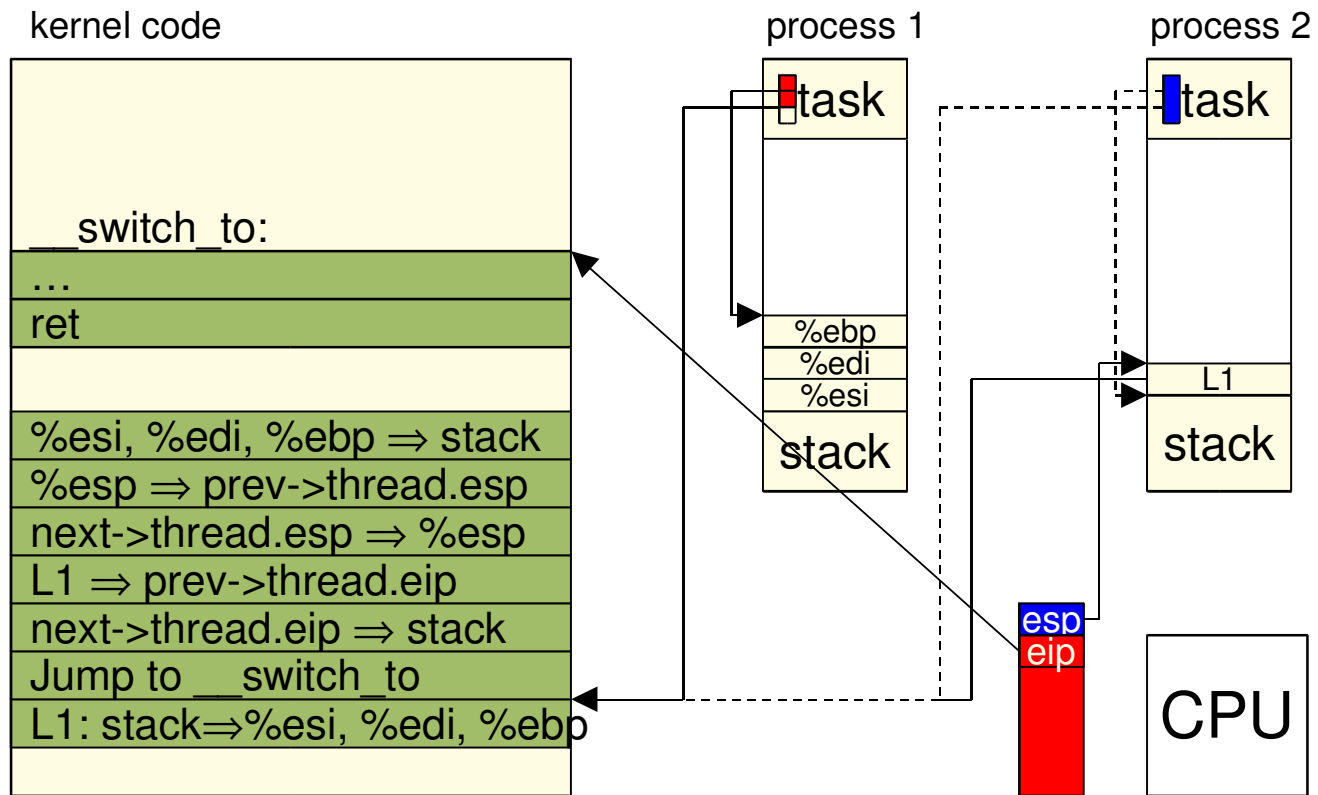


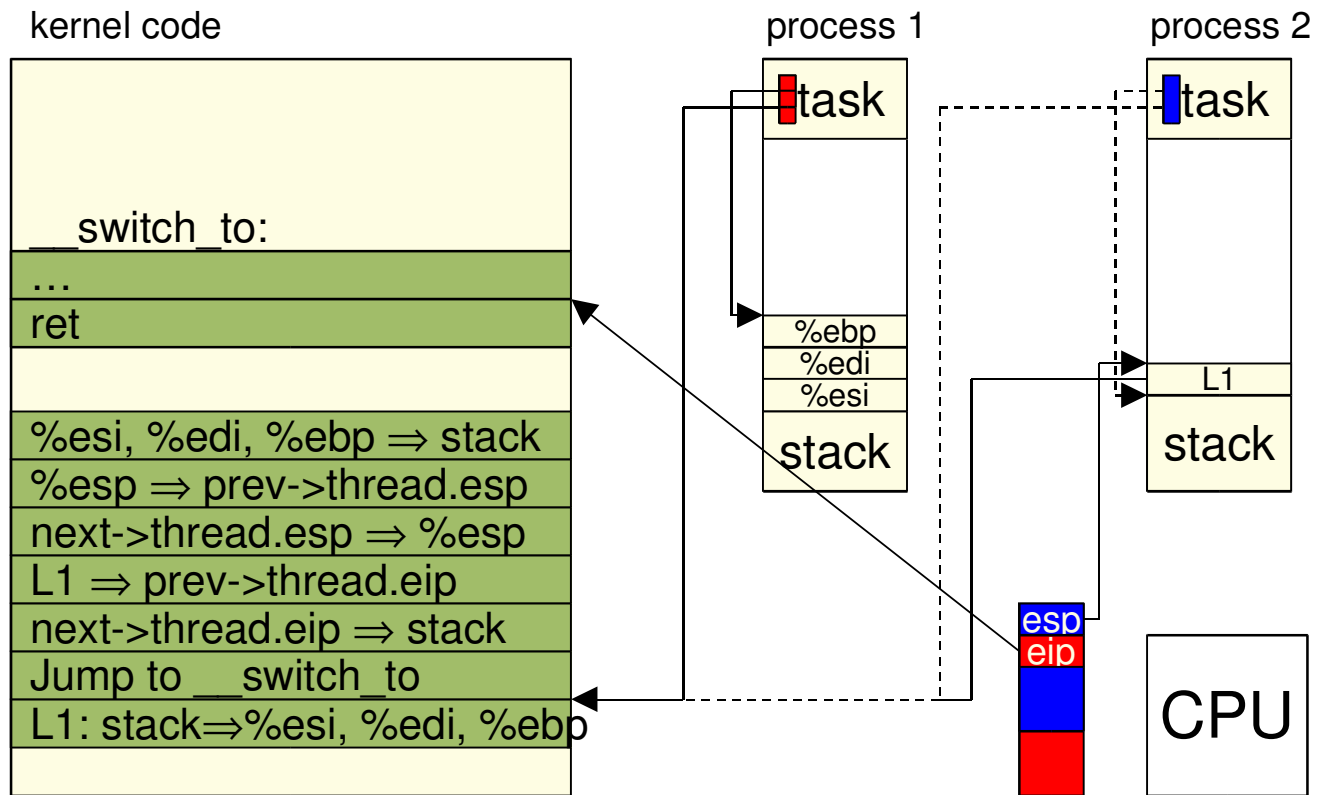


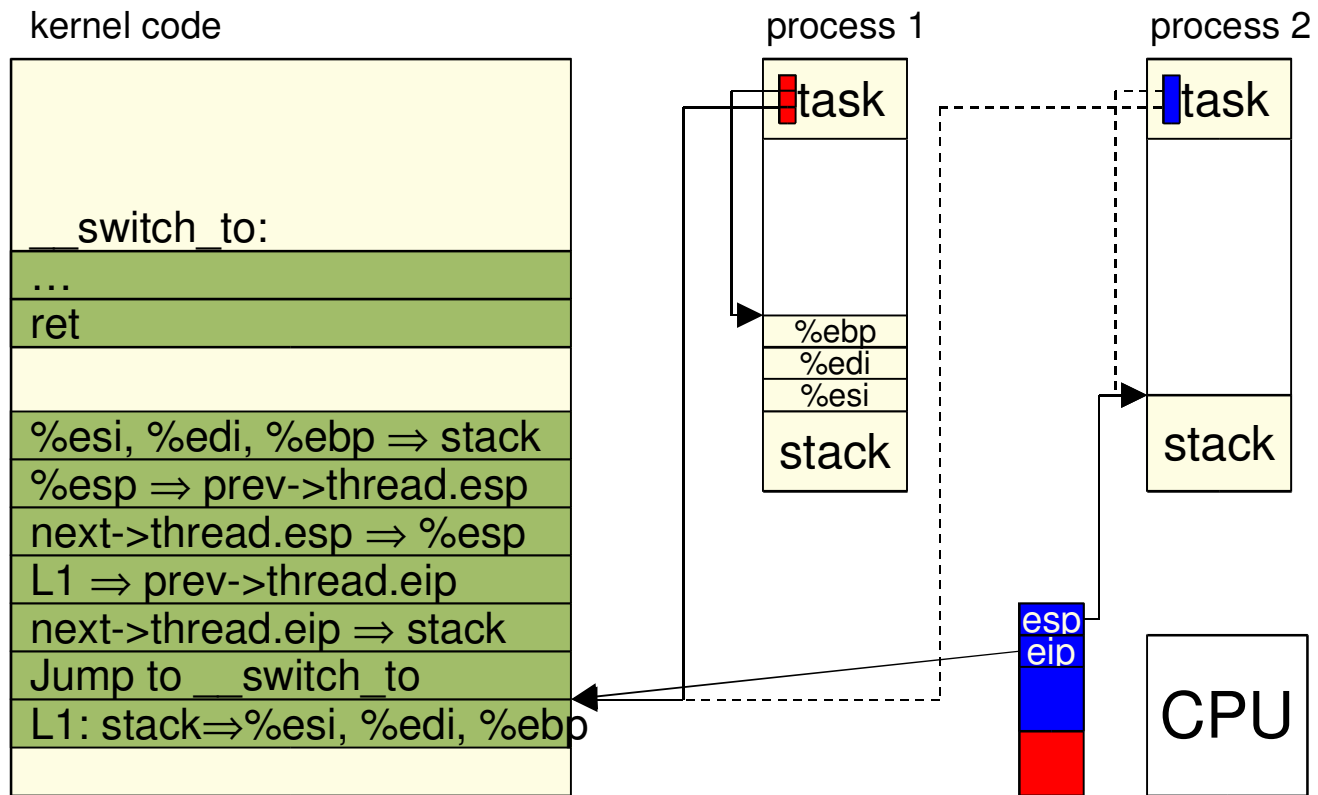


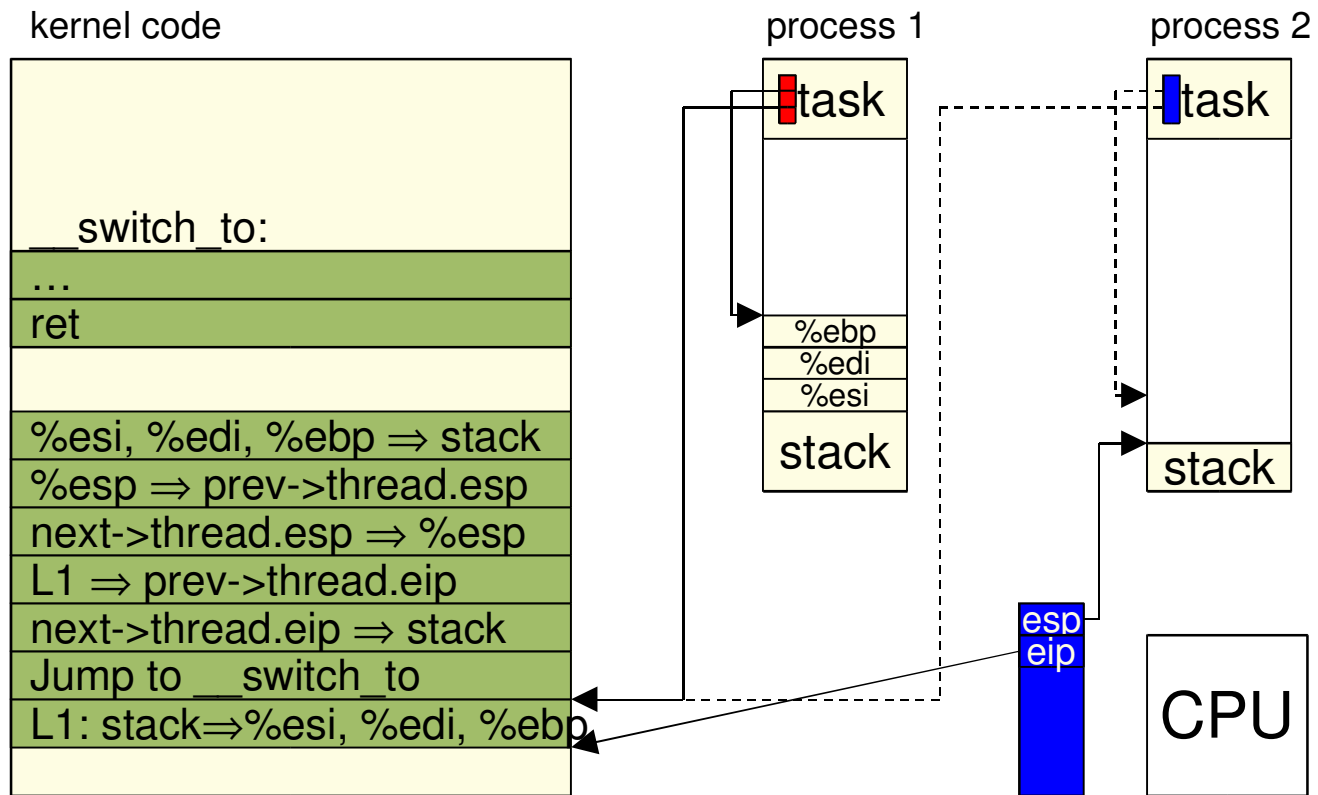












# Another Type of Context Switch

---

- Between Kernel Mode and User Mode of a Process
  - Example: system call
- Also Need to Save the Stacks, CPU states.
- Exercise:
  - Figure out how context switch is done entering a system call and returning from a system call

# Process Scheduling

---

- Scheduling Policy: to determine which process (in the run queue) to run next
  - Time-sharing
  - Preemptive (in user-mode only)
  - Normal (plain time-sharing) or real-time (FIFO or RR)
- Scheduling Mechanisms
  - The Scheduler routine (`schedule()`)
  - Process Switch (`switch_to()`)
  - System Calls

# Scheduler Data Structure

---

- In Each Process Descriptor (`task_struct` fields)
  - long `need_resched`
    - Whether to call `schedule()` when return from interrupt
  - unsigned long `policy`
    - Type of scheduling class
  - long `counter`
    - Number of ticks left before the process uses up its time slice
  - long `nice`
    - The “nice” value of the process (some form of priority)
  - unsigned long `rt_priority`
    - The fixed priority for Real-Time process

# schedule()

---

- In kernel/sched.c :

```
void schedule(void) {  
    lots of checking ...  
    find one with highest goodness() in the run queue  
    ...  
    switch_mm()  
    switch_to(prev,next,prev)  
    ...  
}
```

# Process Selection

---

- Transverse All Processes on the Run Queue for the Most Deserving Process to Run
- Actual code within `schedule()`:

```
list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

# goodness()

---

- In kernel/sched.c
- Higher value is the more desirable (to run next)
- Normal process (non-real-time process):

```
weight = p->counter;  
if (p->processor == this_cpu)  
    weight += PROC_CHANGE_PENALTY;  
if (p->mm == this_mm || !p->mm) weight += 1;  
weight += 20 - p->nice;
```

- Real-time process:  
weight = 1000 + p->rt\_priority;

# Summary

---

- Process Management:
  - LKP §3.1.1 & 3.1.2
  - ULK §3
- Group Project 1
  - Memory Management
  - Due next Monday