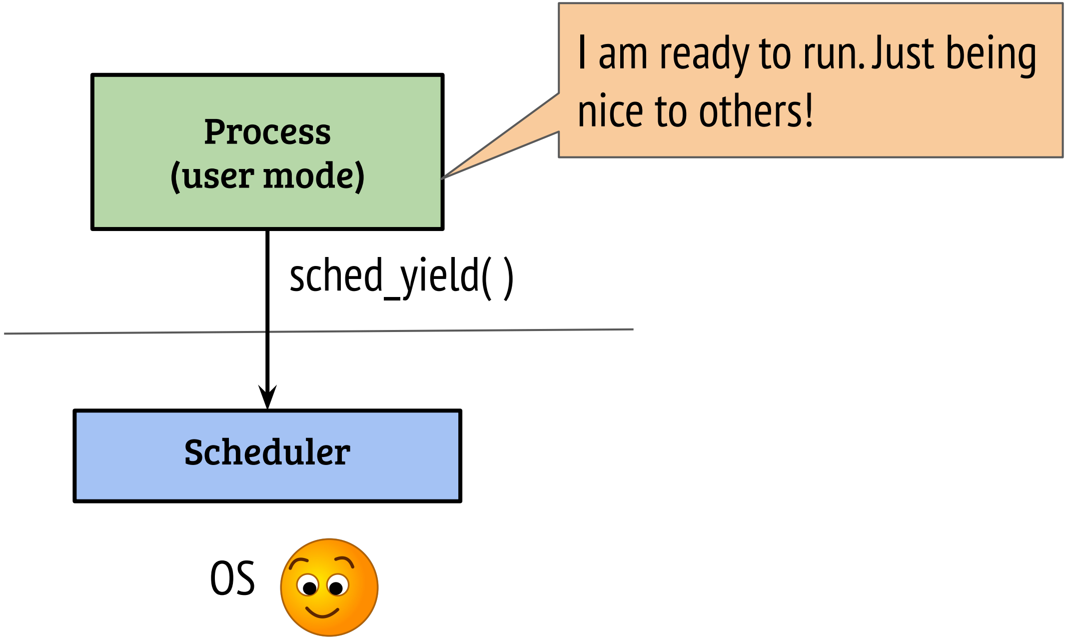
CS614: Linux Kernel Programming

Scheduling Mechanisms

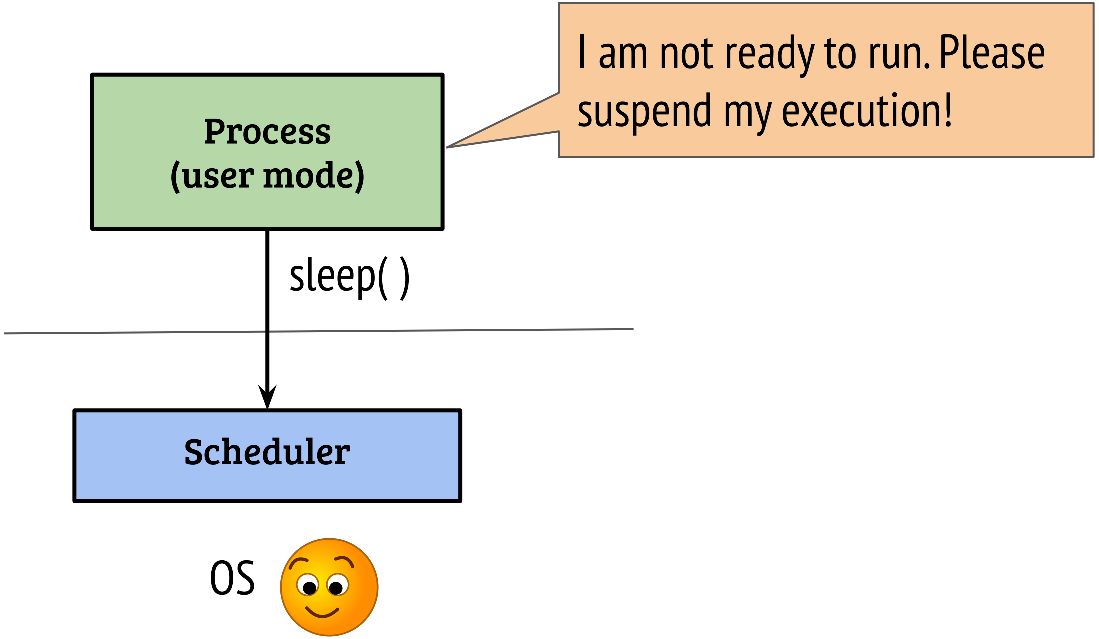
Debadatta Mishra, CSE, IIT Kanpur

Triggers for process context switch



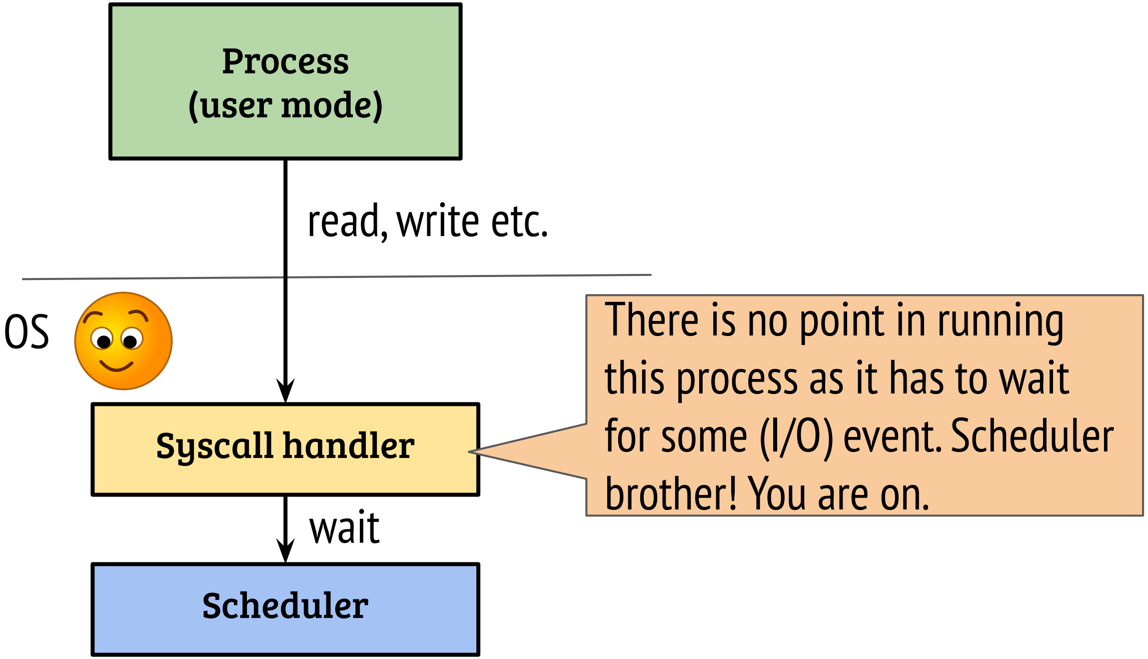
- The user process can invoke the scheduler through explicit system calls like sched_yield (see man page)

Triggers for process context switch



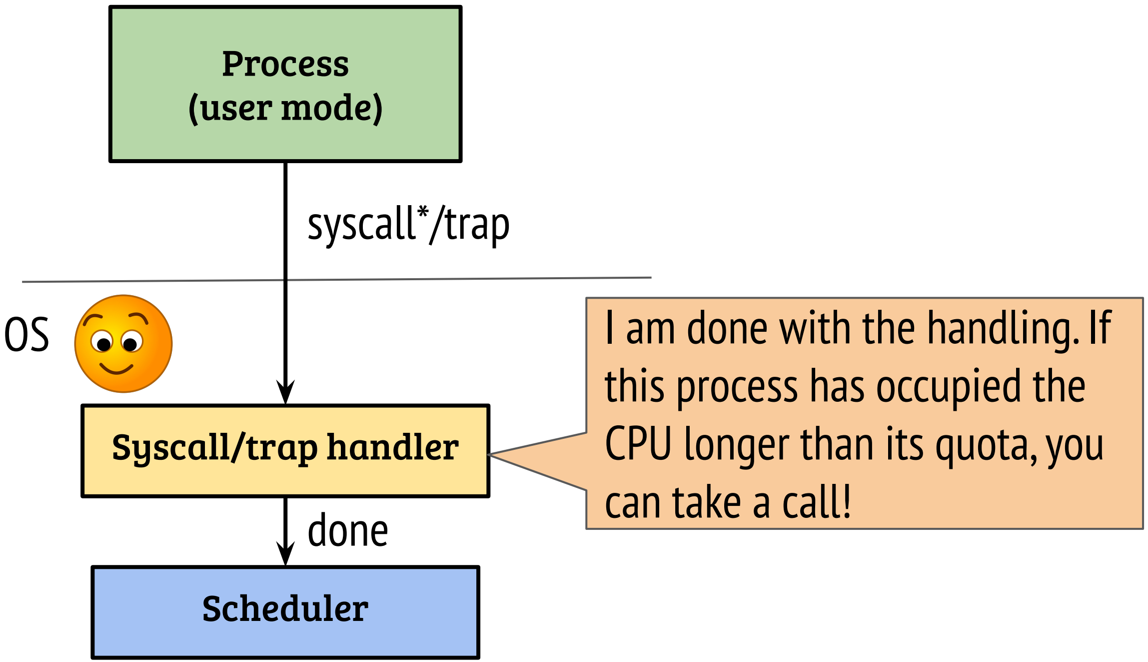
- The user process can invoke sleep( ) to suspend itself
  - sleep( ) is not a system call in Linux, it uses nanosleep( ) system call

Triggers for process context switch



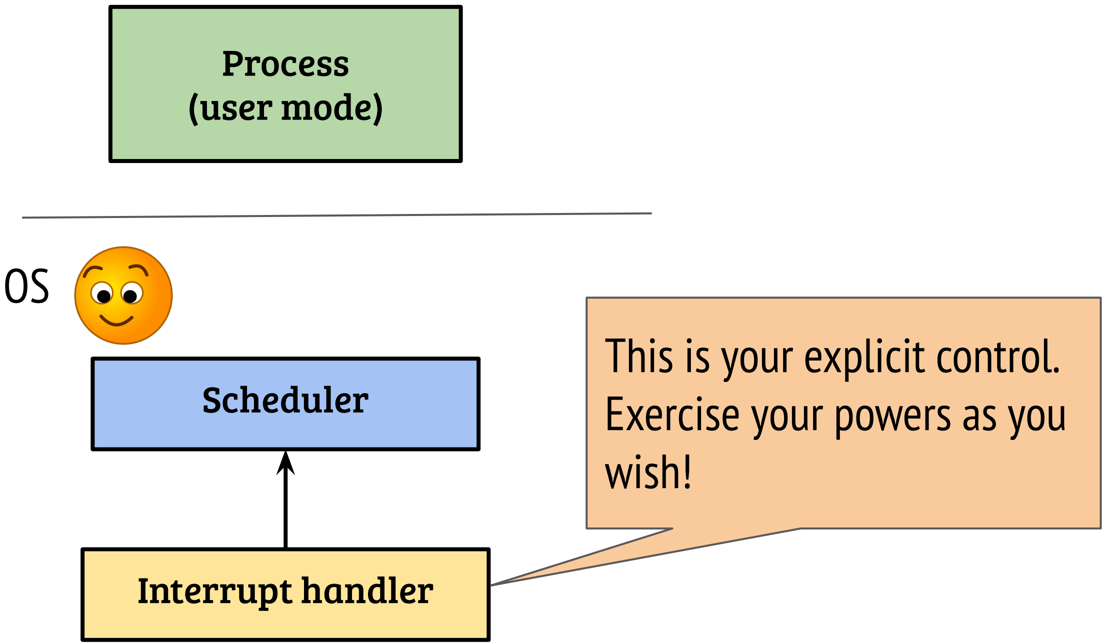
- This condition arises mostly during I/O related system calls
    - Example: read( ) from a file on disk

Triggers for process context switch



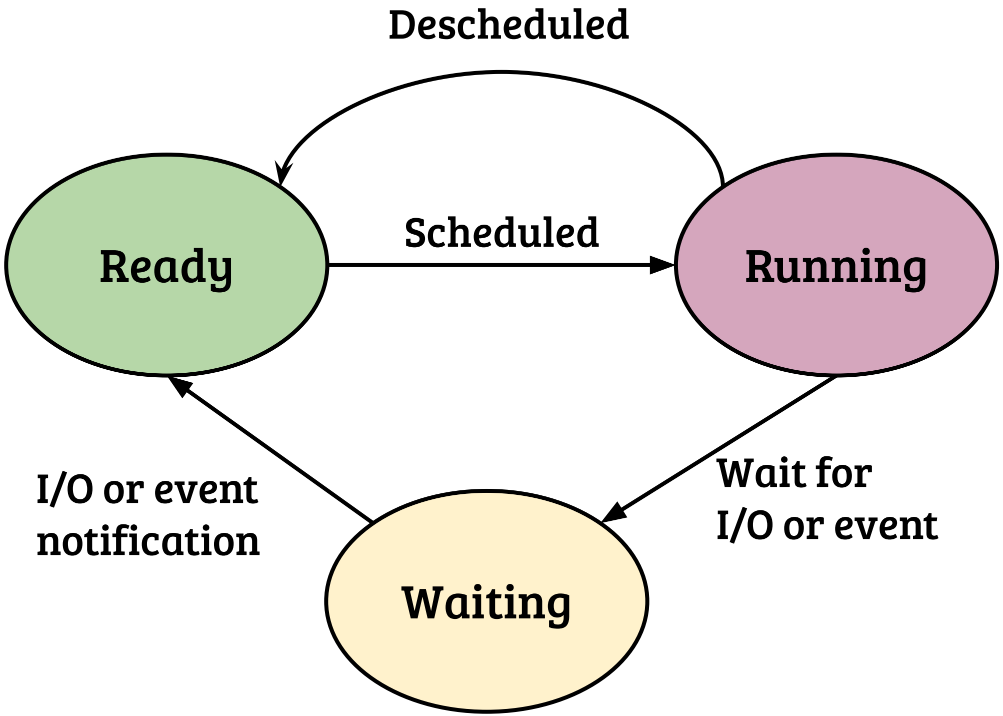
- The OS gets the control back on every system call and exception
- Before returning from syscall, the schedule can deschedule

Triggers for process context switch



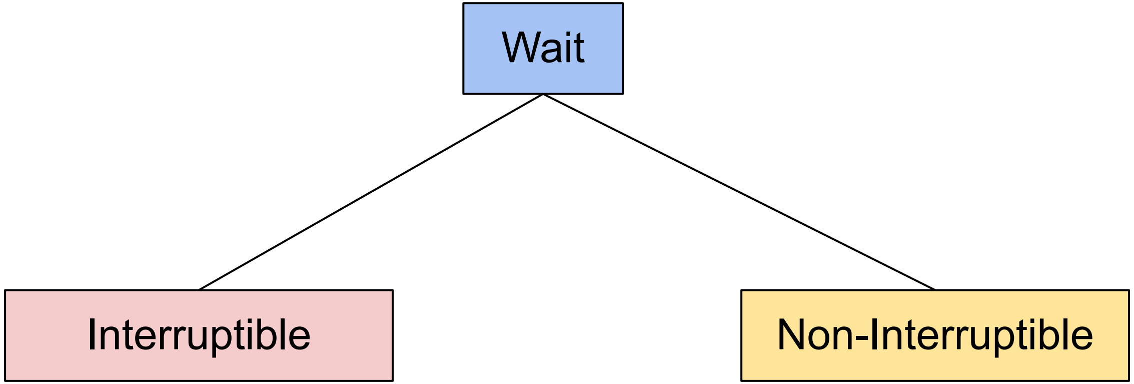
- Timer interrupts can be configured to generate interrupts periodically or after some configured time
- The OS can invoke the scheduler after handling any interrupt

Process states and transitions (simplified)



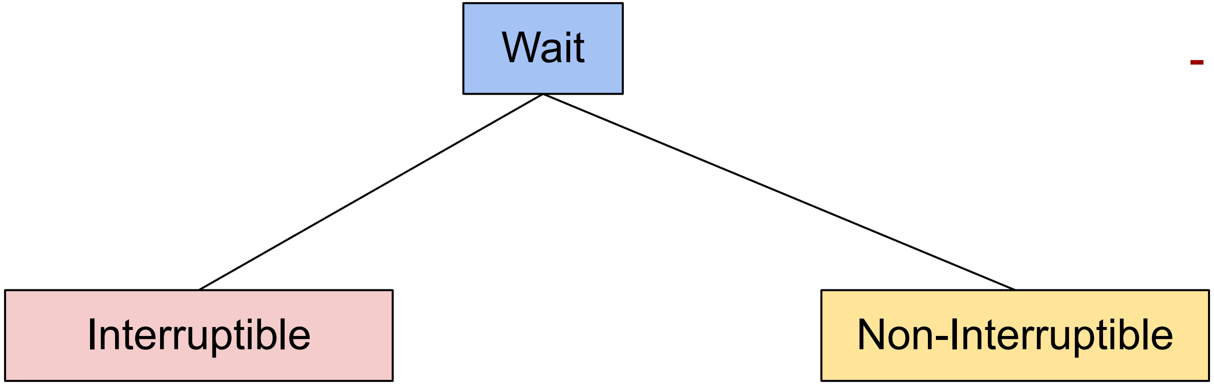
- Most processes perform a mixture of CPU and I/O activities
- When the process is waiting for an I/O, it is moved to waiting state
- A process becomes ready again when the event completion is notified (e.g., a device interrupt)

Interruptible vs non-interruptible Wait in Linux



- In Linux, a process going to waiting mode, need to be either interruptible or non-interruptible

Interruptible vs non-interruptible Wait in Linux

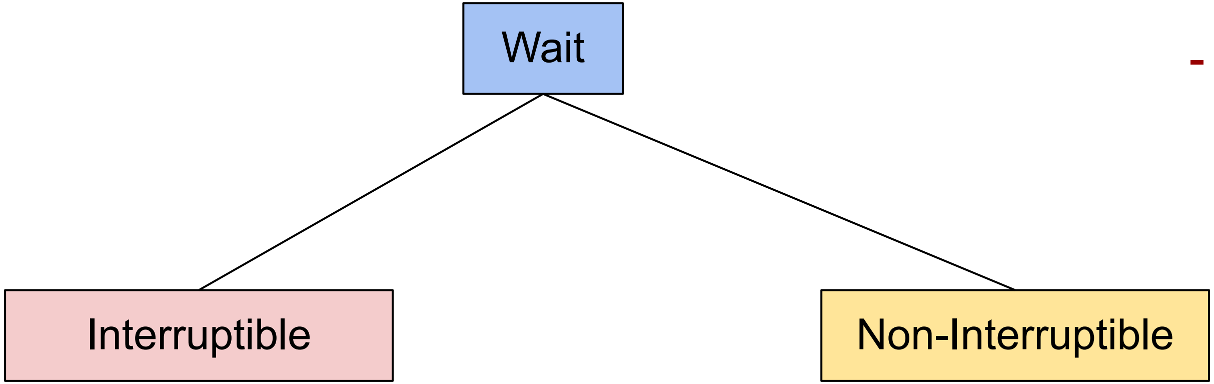


- In Linux, a process going to waiting mode, need to be either interruptible or non-interruptible

An interruptible wait can be woken up by an external event such as a signal. Example?

Non-interruptible waits can be terminated by explicitly calling wake up. Example?

Interruptible vs non-interruptible Wait in Linux



- In Linux, a process going to waiting mode, need to be either interruptible or non-interruptible

An interruptible wait can be woken up by an external event such as a signal. Example? Many, mostly user space induced waits (nanosleep syscall handler)

Non-interruptible waits can be terminated by explicitly calling wake up. Example? Parent waiting for child in case of vfork (wait_for_vfork_done)

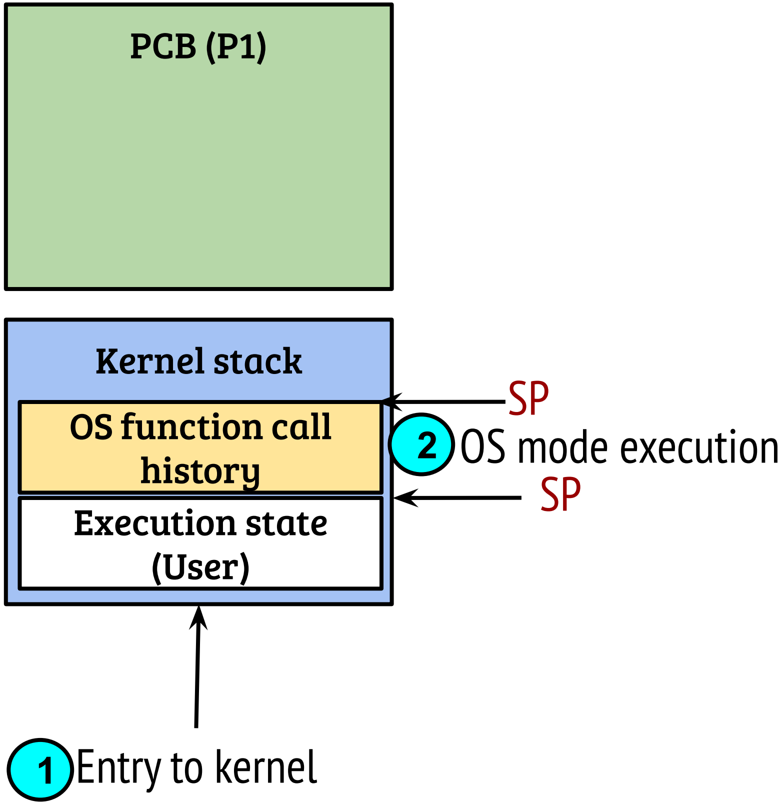
Linux: Scheduler invocations

- User initiated scheduler invocations
  - Explicit: yield, sleep
  - Implicit: blocking system calls (read, select ...) and faults
  - Kernel mode state need to be maintained

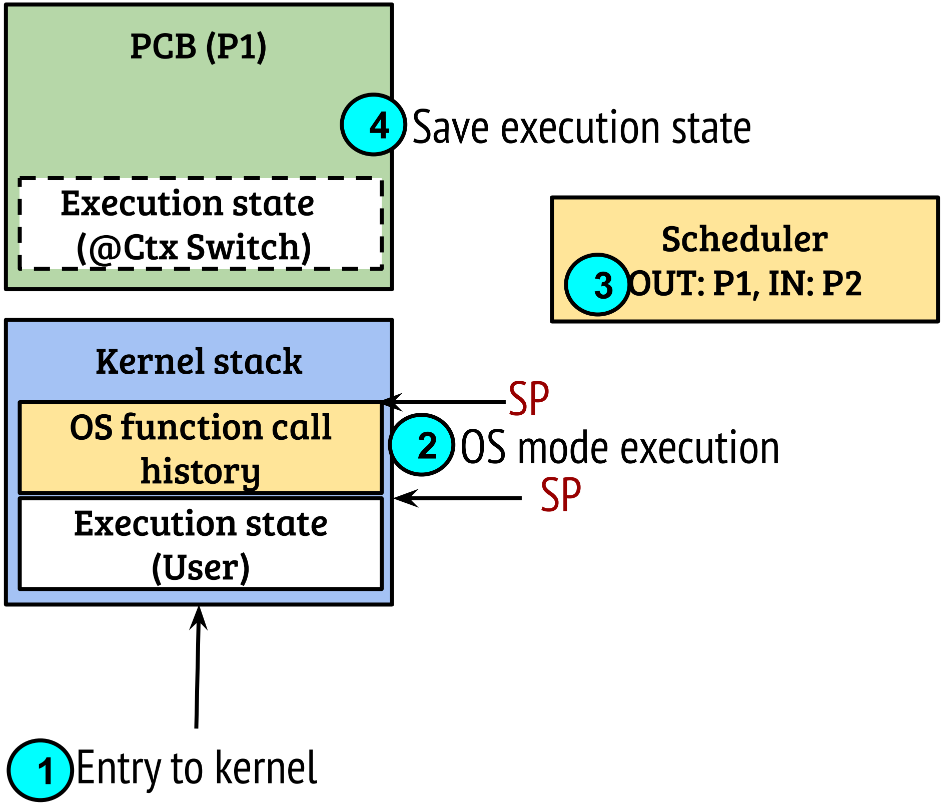
Linux: Scheduler invocations

- User initiated scheduler invocations
  - Explicit: yield, sleep
  - Implicit: blocking system calls (read, select ...) and faults
  - Kernel mode state need to be maintained
- Kernel invocations
  - A designated flag (TIF_NEED_RESCHED) represents if a task needs to be descheduled (scheduler invocation required)
  - While returning to user mode, this flag is checked and scheduler is invoked to perform context switching
  - Relevant functions: `set_tsk_need_resched`, `resched_curr`

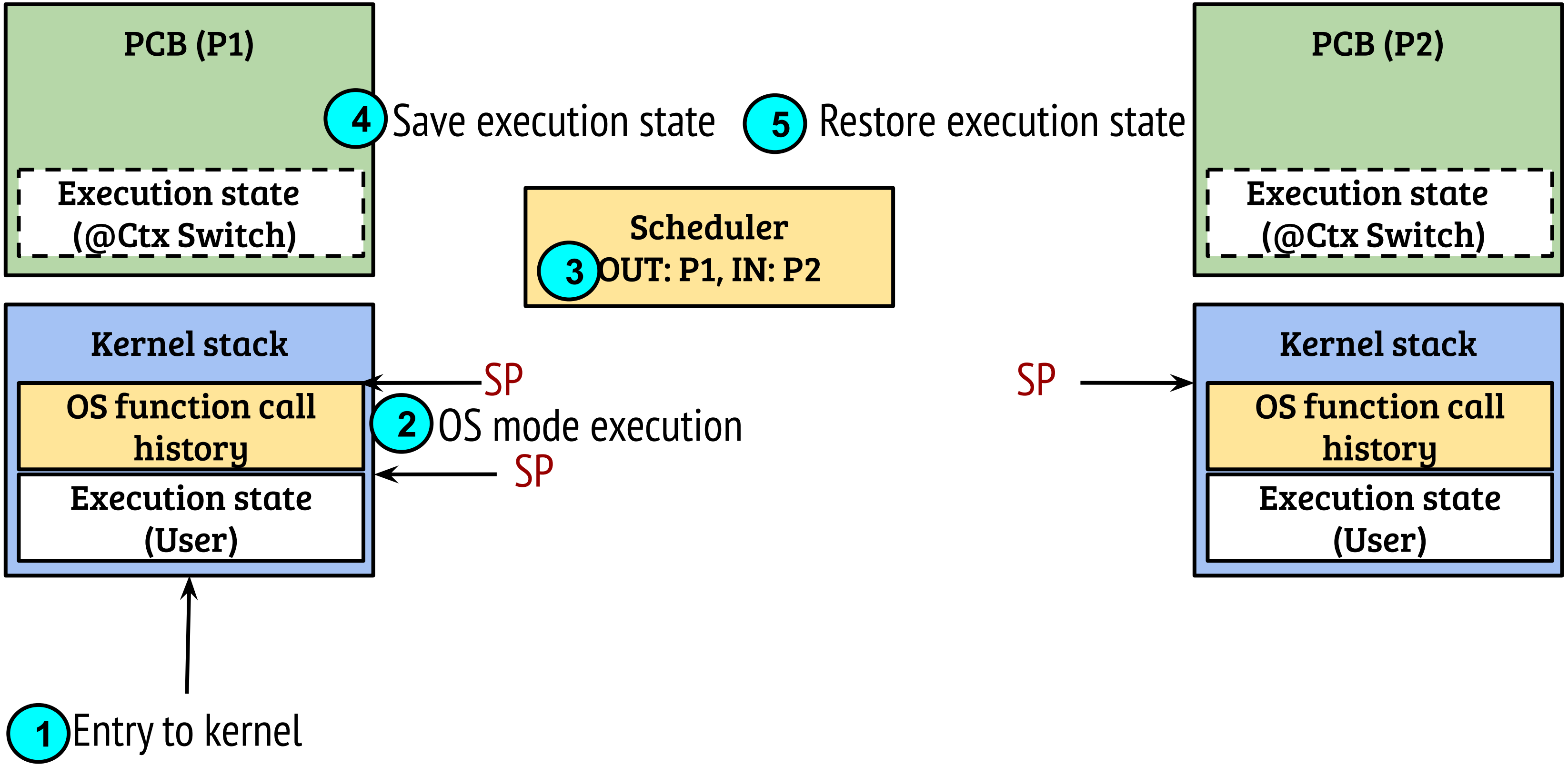
Process context switch (Generic view)



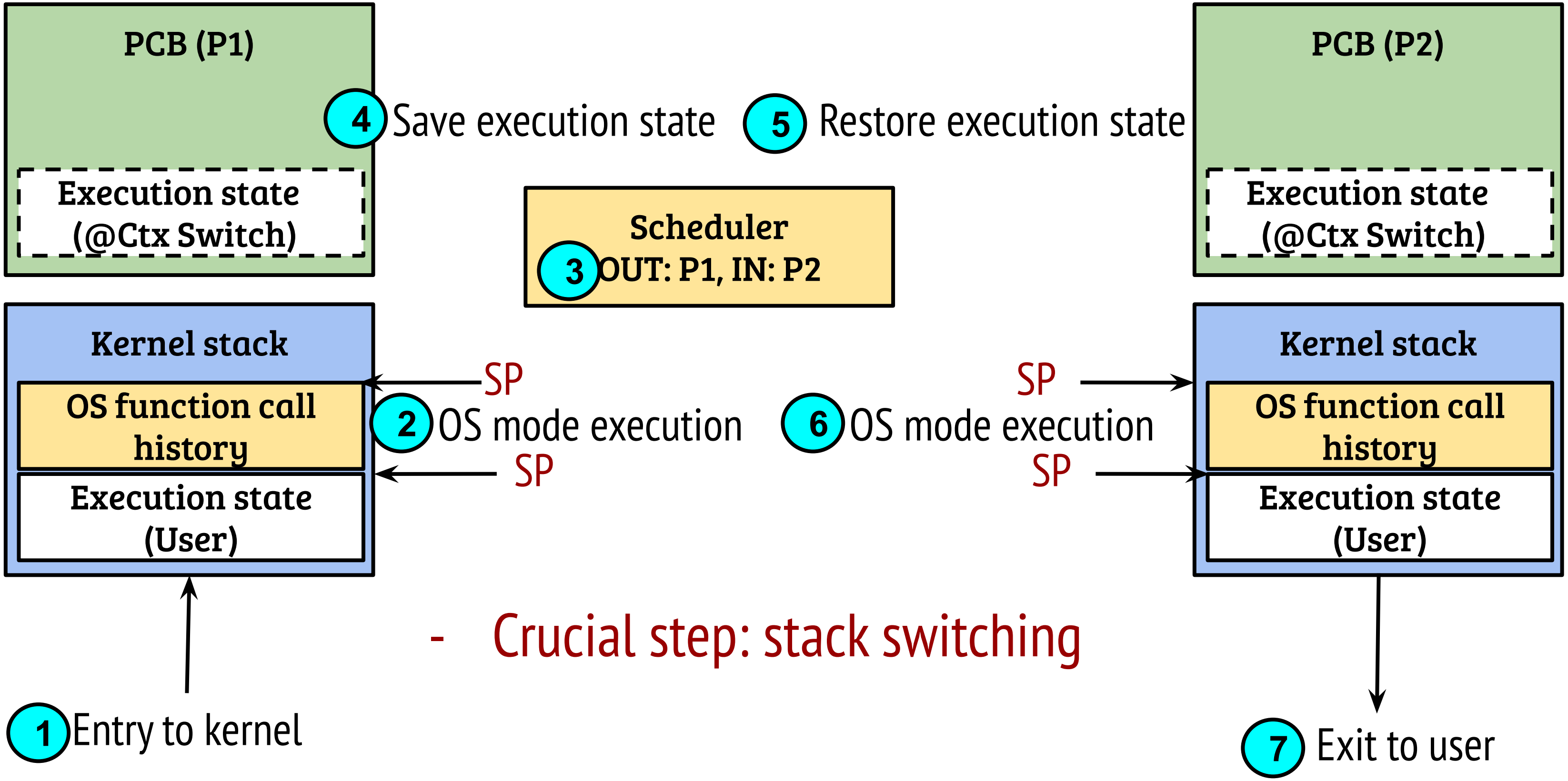
Process context switch



Process context switch



Process context switch



- Crucial step: stack switching

Context switching: Saving the state

- Is it always necessary to copy the execution state into the task_struct?
- What if only the stack pointer is saved in task_struct?

Context switching: Design choices

- Is it always necessary to copy the execution state into the task_struct?
  - Switching out user processes/threads entering the kernel mode through system calls and exceptions have their complete execution history in the kernel stack (user regs saved in kernel stack on entry)
  - A pointer can be maintained in the task_struct; the kernel stack can not be freed anyway!
- What if only the stack pointer is saved in task_struct?

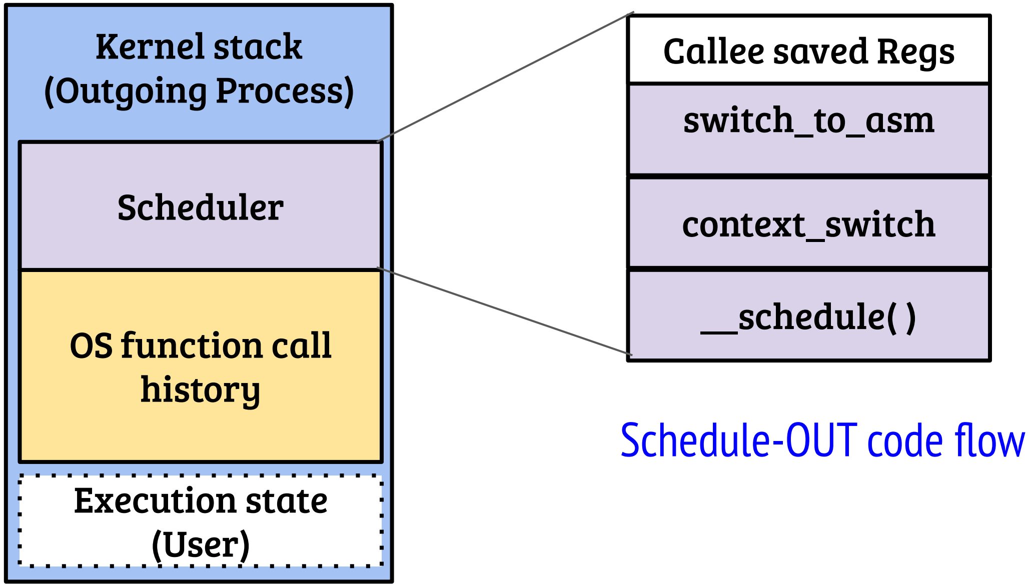
Context switching: Design choices

- Is it always necessary to copy the execution state into the task_struct?
  - Switching out user processes/threads entering the kernel mode through system calls and exceptions have their complete execution history in the kernel stack (user regs saved in kernel stack on entry)
  - A pointer can be maintained in the task_struct; the kernel stack can not be freed anyway!
- What if only the stack pointer is saved in task_struct?
  - Require special handling; example context switch scenarios
  - Case 1: A user context that entered into the kernel through external interrupt
  - Case 2: A user context in kernel mode interrupted by an external interrupt (timer)

Switching the context: A closer look

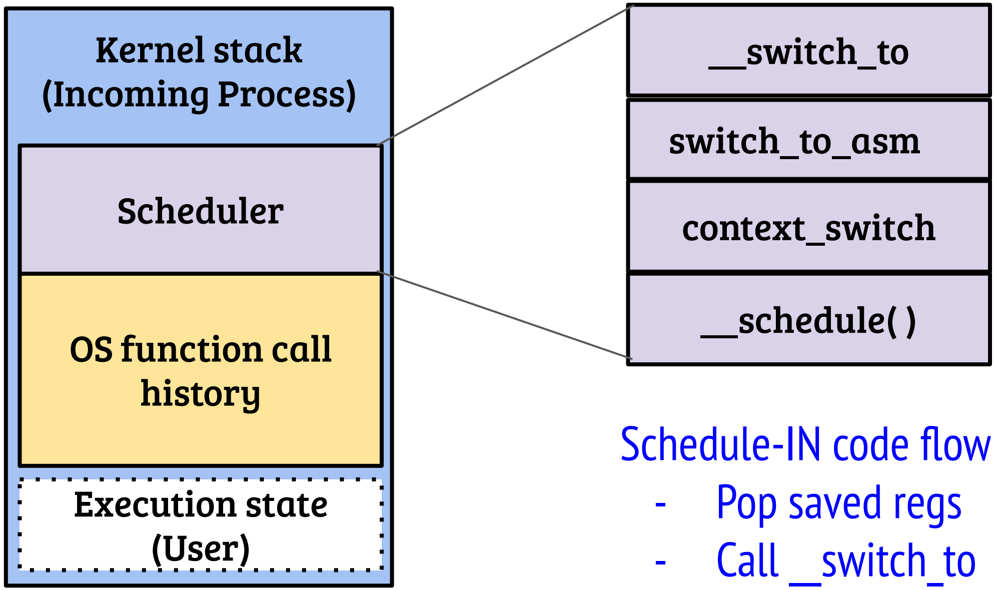
- Assume that the saved stack pointer of the incoming task corresponds to the last execution point
    - How is the last execution point for outgoing process captured?
    - What is the exact point of context switch?
    - Is there a precise point in execution of the scheduler code where we say that the context switch has taken place?
    - Which page table to use during context switching?

State of kernel stack across context switches



- The kernel stack switch occurs in switch_to_asm after saving the registers into the outgoing process kernel stack

State of kernel stack across context switches

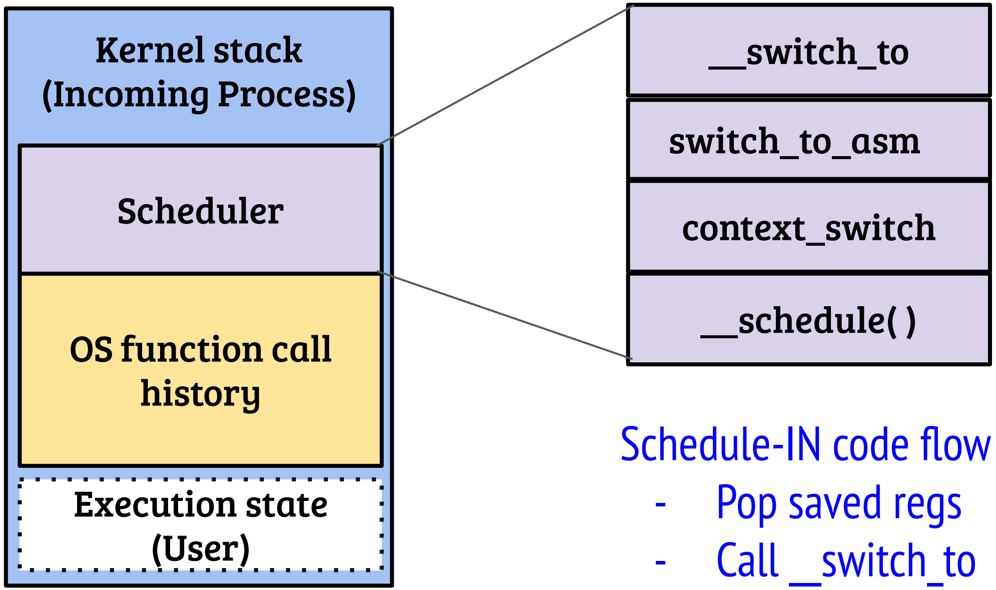


Schedule-IN code flow

- Pop saved regs
- Call __switch_to

- __switch_to performs bulk of state switching (including the change of per-CPU current process value)

State of kernel stack across context switches



Schedule-IN code flow

- Pop saved regs
- Call __switch_to

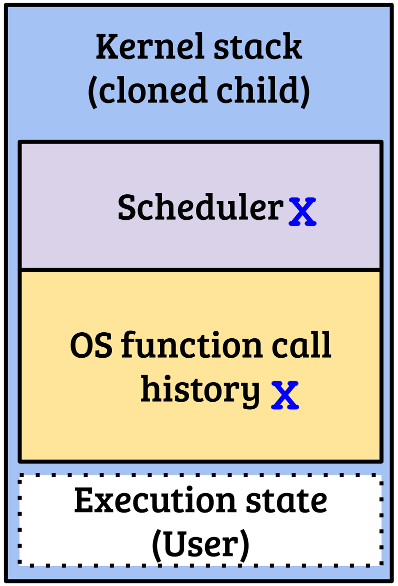
- __switch_to performs bulk of state switching (including the change of per-CPU current process value)

Switching the context: A closer look

- Assume that the saved stack pointer of the incoming task corresponds to the last execution point
  - How is the last execution point for outgoing process captured? In the stack of the outgoing process
  - What is the exact point of context switch? Very hazy when inside the scheduler code
  - Is there a precise point in execution of the scheduler code where we say that the context switch has taken place? Switching of the stack pointer
  - Which page table to use during context switching? Does not matter, any kernel-mode page table is fine

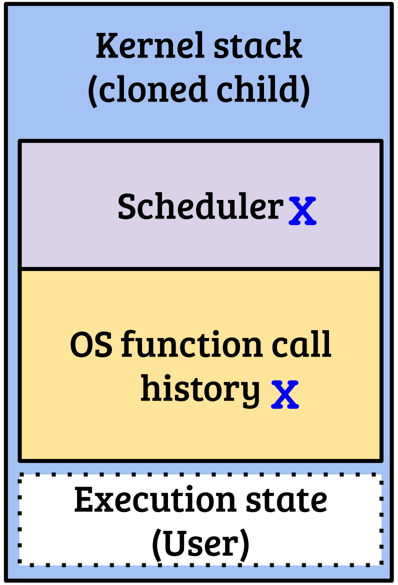
Special handling for newly created user entity

- A newly created execution entity does not have any kernel/sched state



Special handling for newly created user entity

- A newly created execution entity does not have any kernel/sched state



- Before the parent returns from clone, it sets up a special stack frame (a.k.a. fork_frame) for the newly created process
- When the child process is scheduled, it returns to the user through the fork_frame return path i.e., ret_from_fork