Virtualizing the CPU

2023-24 COMP3230B

Contents

- System Calls
- Process switch
- Interrupt

Related Learning Outcome

- ILO 2a - explain how OS manages processes/threads and **discuss the mechanisms** and policies in efficiently sharing of CPU resources.

Reading & Reference

- Required Reading
  - Chapter 6, Mechanism: Limited Direct Execution, Operating Systems: Three Easy Pieces by Arpaci-Dusseau et. Al
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

Process Control

- How to perform restricted operations?
  - Mechanism: System Calls

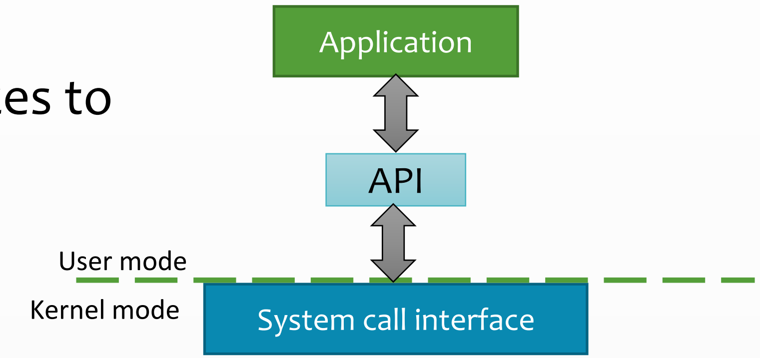
- How to provide illusion of having many CPUs?
  - Virtualizing the CPU
  - Mechanism: Context Switching

Transparent:
- Process does not know when it is running and when it is not
- Programmer does not need to worry about this situation

- How to regain control of the CPU?
  - Voluntary release of CPU
    Mechanism: System Calls

  - Involuntary release of CPU
    Mechanism: Interrupt

To request OS services



- System calls allow the kernel to carefully expose certain key services to applications
  - Most OSs expose a few hundreds such functions
- Applications mostly accessed via a high-level **Application Program Interface** (API) rather than directly invoke the specific system call
  - Common APIs are Windows API for Windows, and POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Usually, in Unix system, the API is included in the run-time support library (e.g. C library)

To request OS services

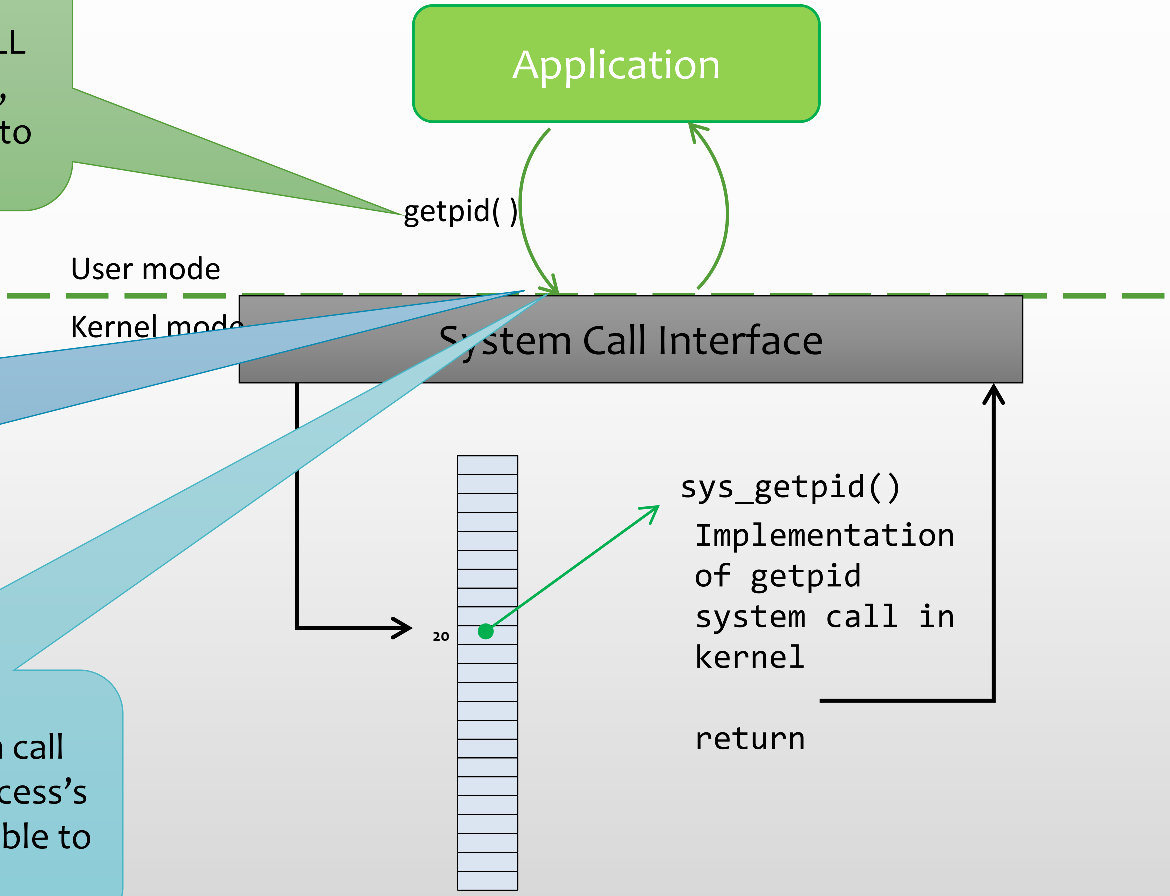
- Why use APIs rather than directly invoke system calls?
  - Caller does not need to know the details on how to invoke a system call under that OS & hardware
    - Different OSs use different techniques in implementing system calls
    - Usually, involves assembly code
  - Programmer Just needs to know how to use the **standard** API to invoke the system call and understand what information will be returned by the system call via the API
    - Details of the OS interface are hidden from programmer by the API
    - The system call interface invokes intended system call in kernel and returns status and results of the system call and pass back to calling program via the API
    - The same API can be provided by different OS platforms ➔ allows running the applications on different platforms

System Call Implementation

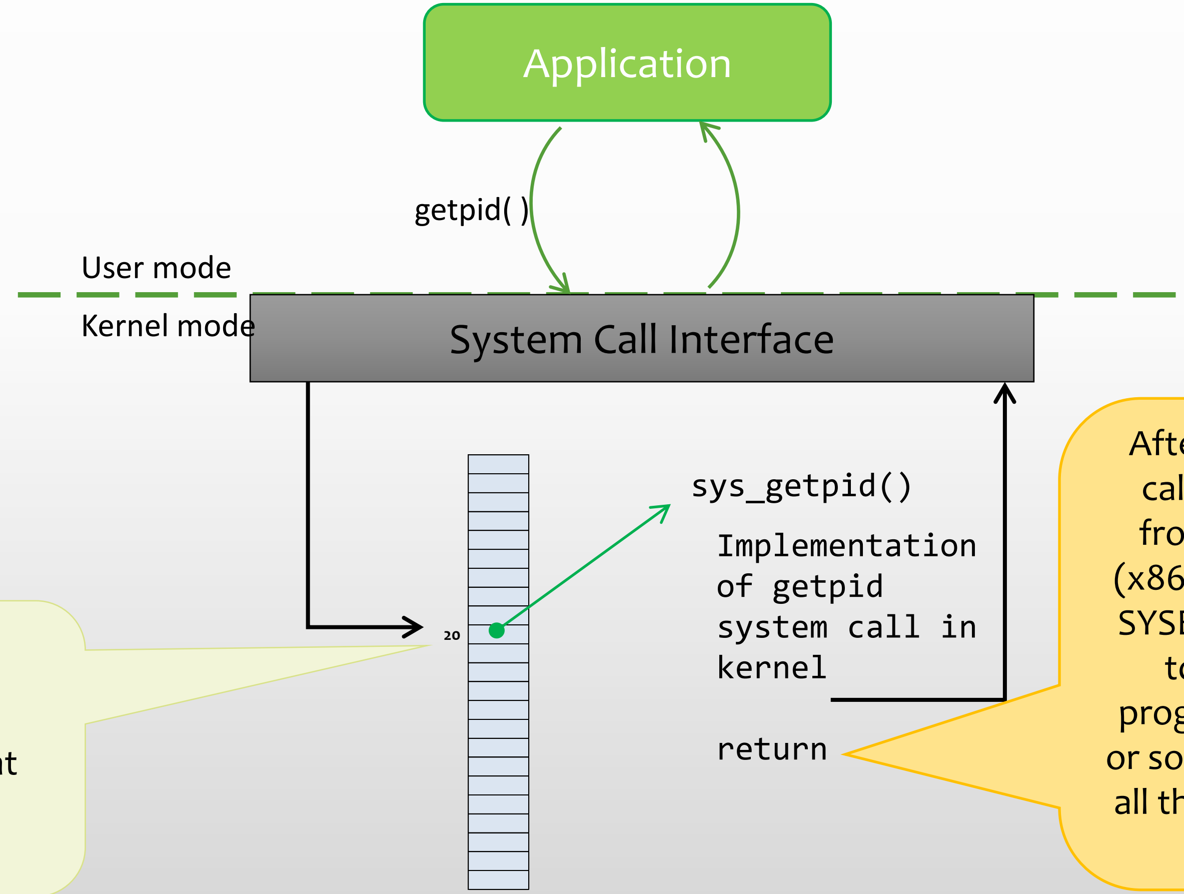
Typically, the corresponding library call contains a special machine instruction (x86 – INT or SYSCALL or SYSENTER) causing a “trap”, which results in transfer control to the kernel

- Each system call is associated with a number.
- Before “trap” into kernel, the library stores the call parameters and the system call number in the specific registers.

Either the hardware or system call function must save enough process’s register context in order to be able to return correctly



System Call Implementation



- Within kernel, the system-call implementation maintains a table indexed by these numbers.
- Each entry consists of an address that points to the corresponding system function in the kernel

After finished, handler calls a special return-from-trap instruction (x86 - IRET or SYSRET or SYSEXIT) to return back to the calling user program. The hardware or software also resumes all the process's register context.

Virtualizing the CPU

- To provide illusion that each process has its own CPU
- By virtualizing, does not mean giving a virtual CPU
  - The program is directly running on the real CPU
  - Mechanism:
    - By running one process, then stopping it and running another, and so forth
- The Crux
  - How to **transparently** and **temporarily** **stop** a process and **resume** it?
  - With direct execution of application process on CPU, how can OS **regain control** of the CPU?

Switching from one process to another process

- To stop a process, one just needs to remove it from running to other state
- How can OS make sure that the process can be resumed **without affecting** its execution logic?
  - OS needs to **save the context** (e.g., CPU registers) of current running process before stopping it
  - To resume, OS needs to **restore the context** of the soon-to-be running process back to the CPU before resuming it
  - Question: Where to store the execution context of a process?
- **Context Switch**
  - Switching of the execution context of one process to another

Mode Switch and Context Switch

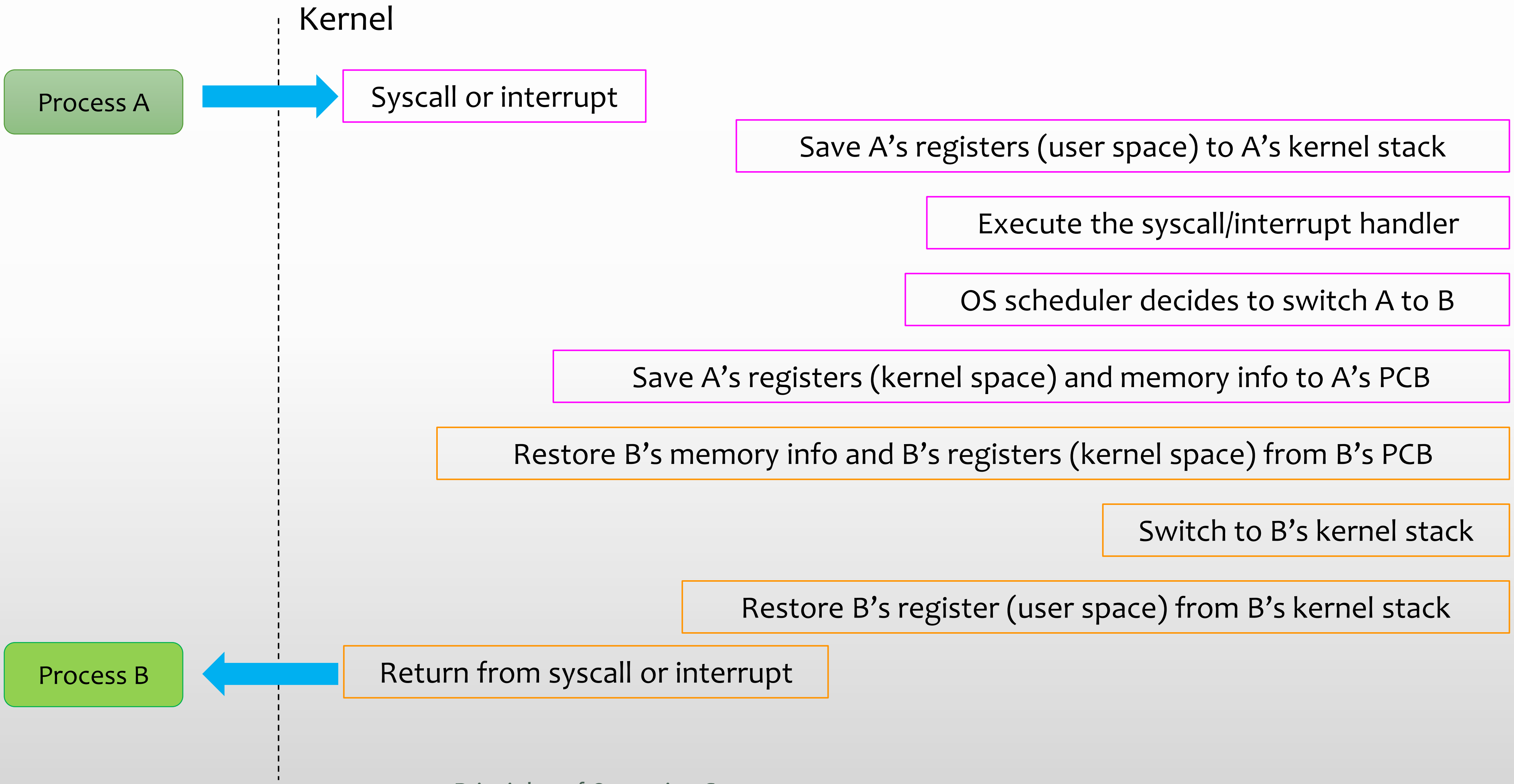
- Mode switch
  - Switching from user mode to kernel mode
  - the Kernel is said to be “executing on behalf of the process”
  - the process’s context remains accessible (e.g., address space)
  - upon exiting kernel, the process *may* resume and return to execute in user space

- Context switch
  - Switching from one process to another
  - is an essential feature of multiprogramming or time-sharing systems
  - Kernel must
    - (1) suspends the progression of current process and stores its context
    - (2) retrieves the context of another process (B) and restores it to the CPU
    - (3) resumes execution of process B

Context Switch

- Action
  - Save the “current” process’s execution context
  - Change the process’s state in PCB to appropriate status
  - Move the process to appropriate queue
  - Select a ready process (according to **scheduling policy**)
  - Load the “to be dispatched” ready process’s execution context
  - Update the ”to be dispatched” process’s state of that going to be run process
  - Resume that process by turning control over to that process

Switching contexts



Context Switch Overhead

- The CPU is considered as not performing any “useful/real” computation for the application process
  - Thus, with frequent context switching, application would take longer time to complete its execution
  - OS must minimize context-switching time
    - There are hardware instructions in some architectures that facilitates the switching

- Has some indirect cost (induced overheads)
  - The newly scheduled process may not have data and/or instructions in the physical memory
  - The newly scheduled process does not have any part of its address space in the CPU cache

To regain control of CPU

- A Passive Approach
  - OS just waits for a process to make system calls or an illegal operation
  - Indeed, this works most of the time as processes quite frequently invoke system calls
  - However, in worst case, a process gets into an infinite loop and never makes a system call, then what can OS do!!

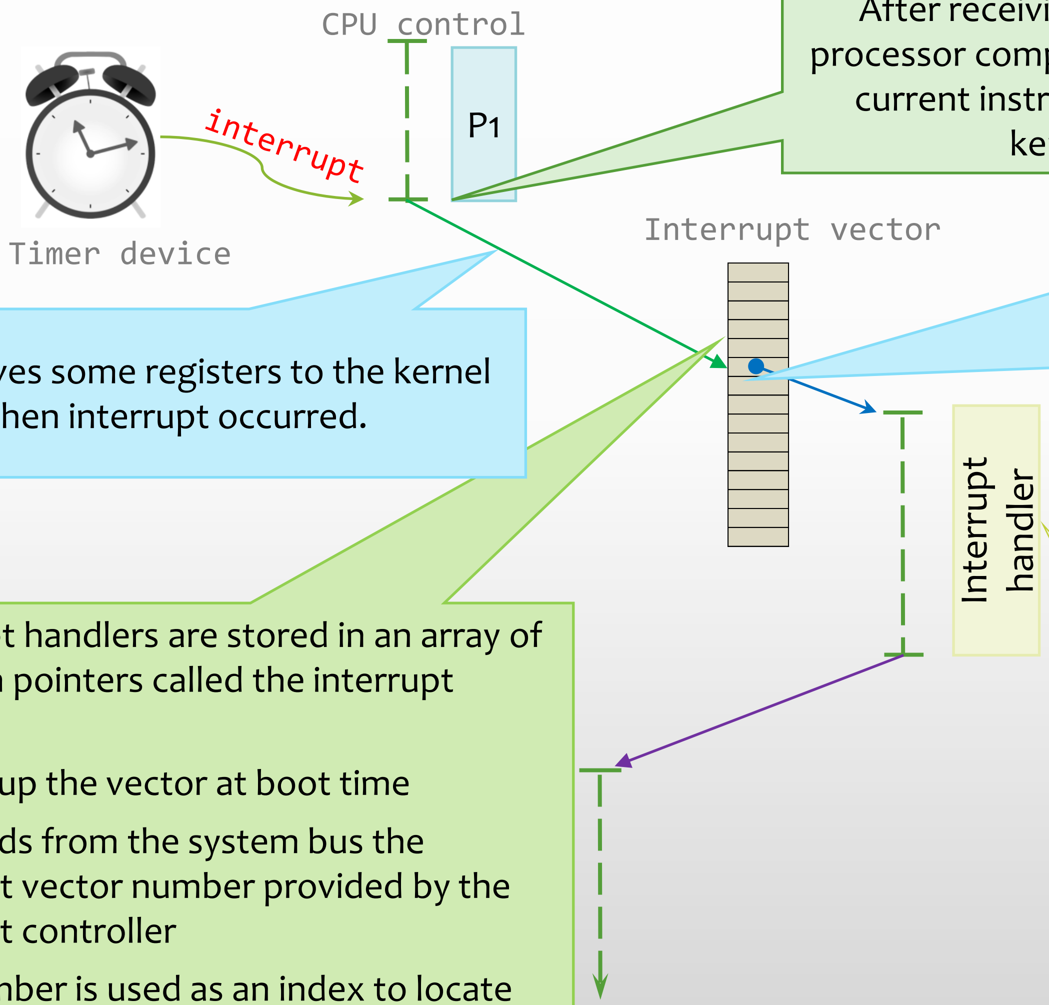
- An Active Approach (assume non-cooperative processes)
  - Needs hardware support
  - A timer device periodically generates an interrupt, e.g. every few milliseconds
  - When the interrupt is raised, OS will be called in to handle the interrupt, and thus can do what it wants

Interrupts

- An interrupt is an event raised by software or hardware when it needs the CPU's attention
  - Interrupts enable OS to respond to “alerts” immediately
- Hardware device applies signal to the interrupt line directly or via APIC to the CPU
  - e.g., a key is pressed or the mouse is moved
  - **Asynchronous** with the operation of the **current** running process
- May be triggered by the current running process
  - **Synchronous** with the operation of the **current** process
    - e.g., dividing by zero or referencing protected memory
  - Interrupt in this case is called an exception
    - Can be classified into 3 categories: Faults, Traps, & Aborts
      - Faults - CPU detects an issue before executing an instruction, e.g., page fault, segmentation fault, etc.
        - Some faults can be corrected and the interrupted program can resume
      - Traps – CPU detect the exception immediately following the execution of the instruction, e.g., INT instruction
        - After handling the trap, the interrupted program is allowed to continue
      - Abort – CPU experiences an unrecoverable error, e.g., double faut – another fault happens in the middle of handling previous fault.

Interrupt Handling

The processor responds to interrupts and exceptions in essentially the same way.



After receiving an interrupt, the processor completes execution of the current instruction and jumps to kernel space

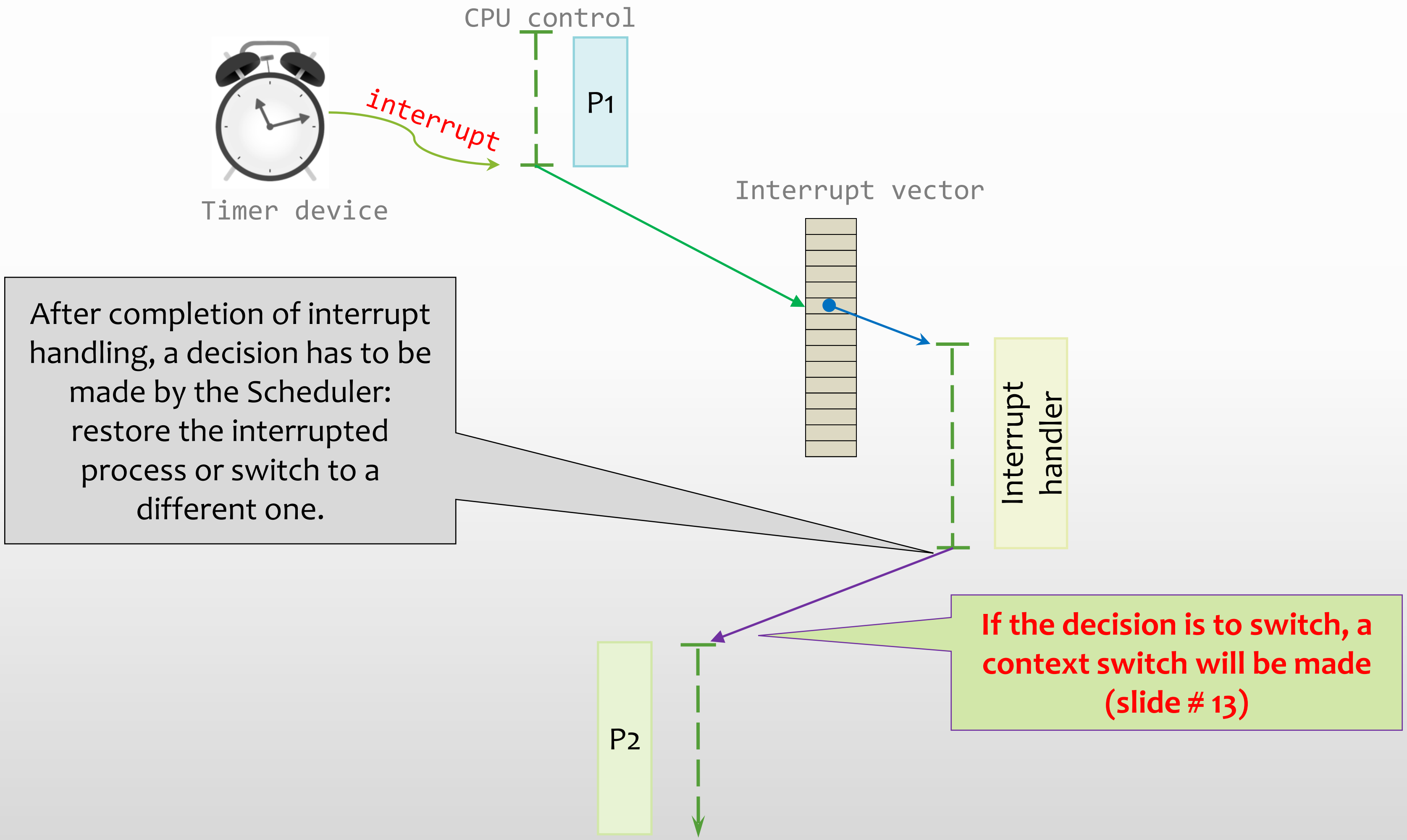
CPU saves some registers to the kernel stack when interrupt occurred.

- OS may save remainder of process state in somewhere (e.g. kernel stack)
- Jump to interrupt handler and process the interrupt

- Interrupt handlers are stored in an array of function pointers called the interrupt vector
- OS sets up the vector at boot time
- CPU reads from the system bus the interrupt vector number provided by the interrupt controller
- This number is used as an index to locate the interrupt handler

The interrupt handler determines how the system should respond

Interrupt Handling



Summary

- Three mechanisms
  - System call
    - Each call represent a service provided by the OS to user application
    - System mode will switch from user mode to kernel mode, and after the service, will switch back from kernel mode to user mode
  - Context switch
    - The key action of virtualizing the CPU
    - The whole set of register context must be saved for the interrupted process
    - The register context of coming process are restored
    - Context switch involves certain amount of performance overhead
  - Interrupt processing
    - A mechanism for hardware devices to alert OS for handling high-priority events
    - This gives a chance to OS to regain control of CPUs