Trap handling and context switching

Mythili Vutukuru

CSE, IIT Bombay

Recap: OS runs processes

- OS manages multiple active processes concurrently
- What is a process?
  - Memory image in RAM = compiled code, data (compile-time, run-time)
  - CPU context (in CPU registers when running, else saved in PCB)
  - Other things like I/O connections, ..
- Processes created by fork from parent processes
- Periodically, OS scheduler loops over ready processes
  - Find a suitable process to run, save old process context, restore new context
- Once process is context switched in, OS is out of picture, CPU in user mode, runs user code directly
- When does the OS run again?

User mode vs. Kernel mode of a process

- CPU runs user code in user mode (low privilege) most of the time
- CPU switches to kernel mode execution when
  - Process makes system call, needs OS services
  - External device needs attention, raises interrupt
  - Some fault has happened during program execution
- All such events are called traps: CPU “traps” into OS code
  - CPU shifts to high privilege level (kernel mode), runs OS code to handle event
  - Later, CPU switches to low privilege level, back to user code in user mode
- Process P goes to kernel mode to run OS code, but it is still process P itself that is in running state
- OS not a separate process, runs in kernel mode of existing processes

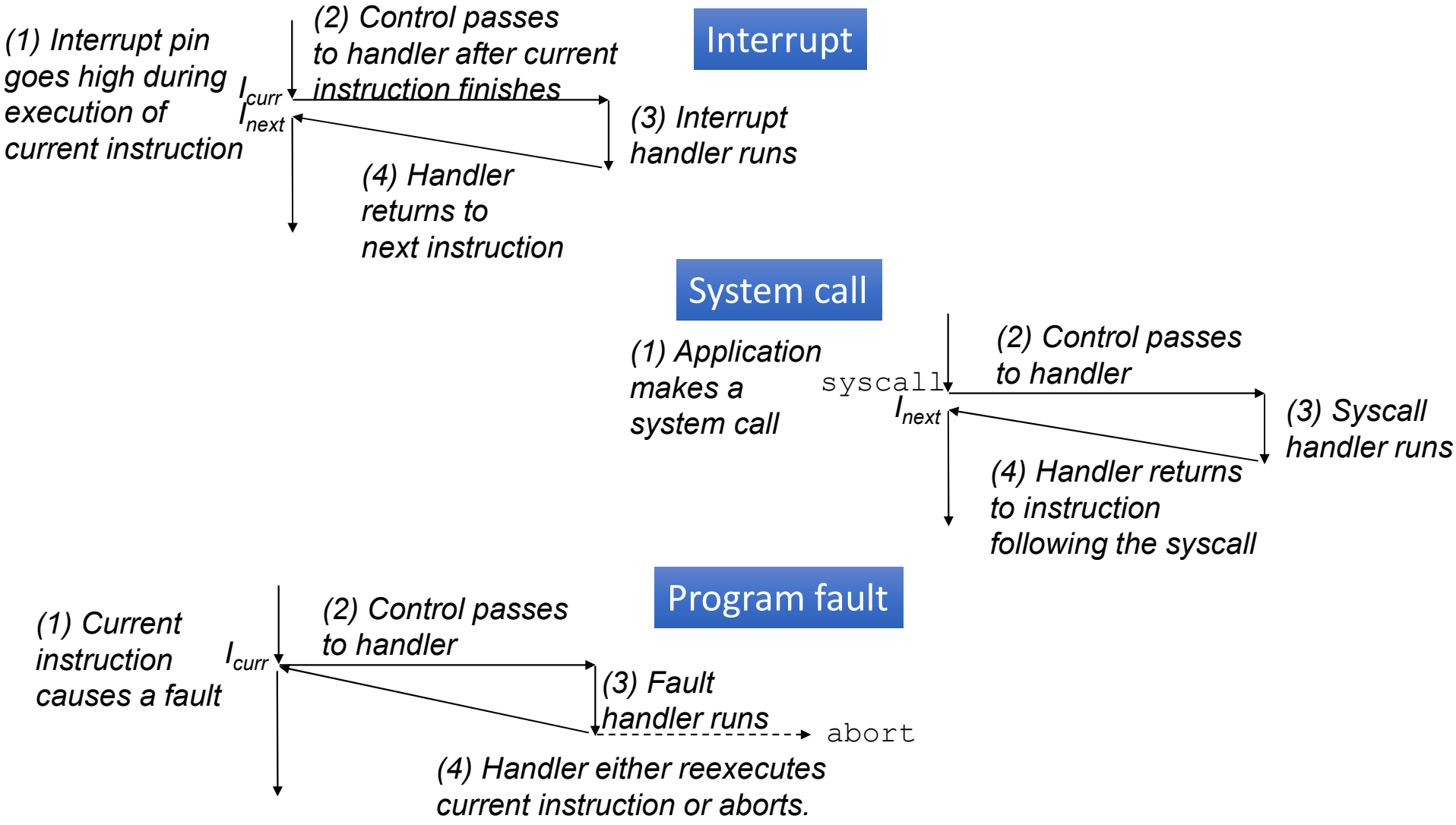
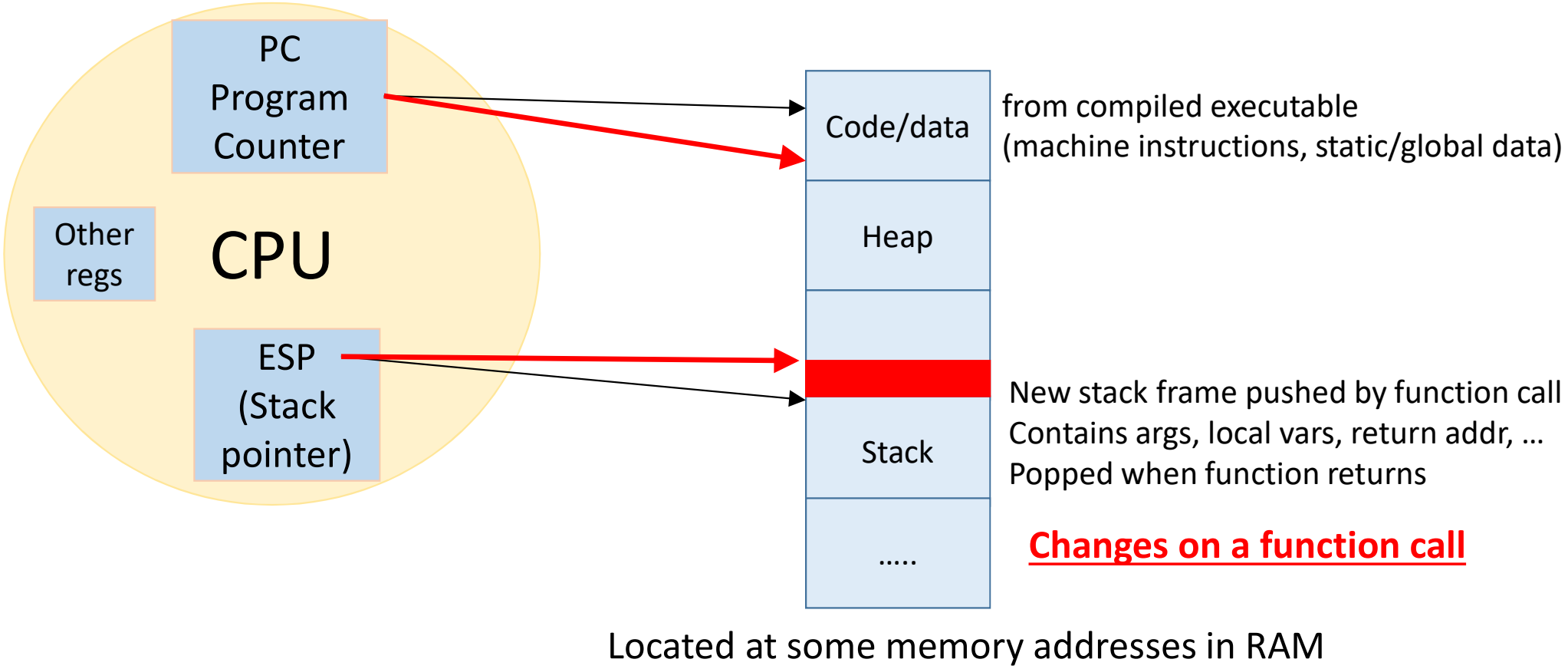


Image credit: CSAPP

Function call vs. system call

- What happens when a user program makes a function call?
  - Allocate memory on user stack for function arguments, local variables, ..
  - Push return address, PC jumps to function code
  - Push register context (to resume execution when function returns)
  - Execute function code
  - When returning from function, pop return address, pop register context
- System call also must
  - Use a stack to push register context
  - Save old PC, change PC to point to OS code to handle system call
  - Run system call, restore context back to user code

Understanding a function call



What is different for a system call?

- Changing PC in function call vs. system call
  - In function call, address of function code known in executable, can jump to function code directly using a CPU instruction ("call" in x86)
  - For system call, cannot trust user to jump to correct OS code (what if user jumps to inappropriate privileged code?)
- Saving register context on stack in function call vs. system call
  - In function call, register context is saved and restored from user stack
  - For system call, OS does not wish to use user stack (what if user has setup malicious values on the stack?)
- We require: a secure stack, a secure way of jumping to OS code

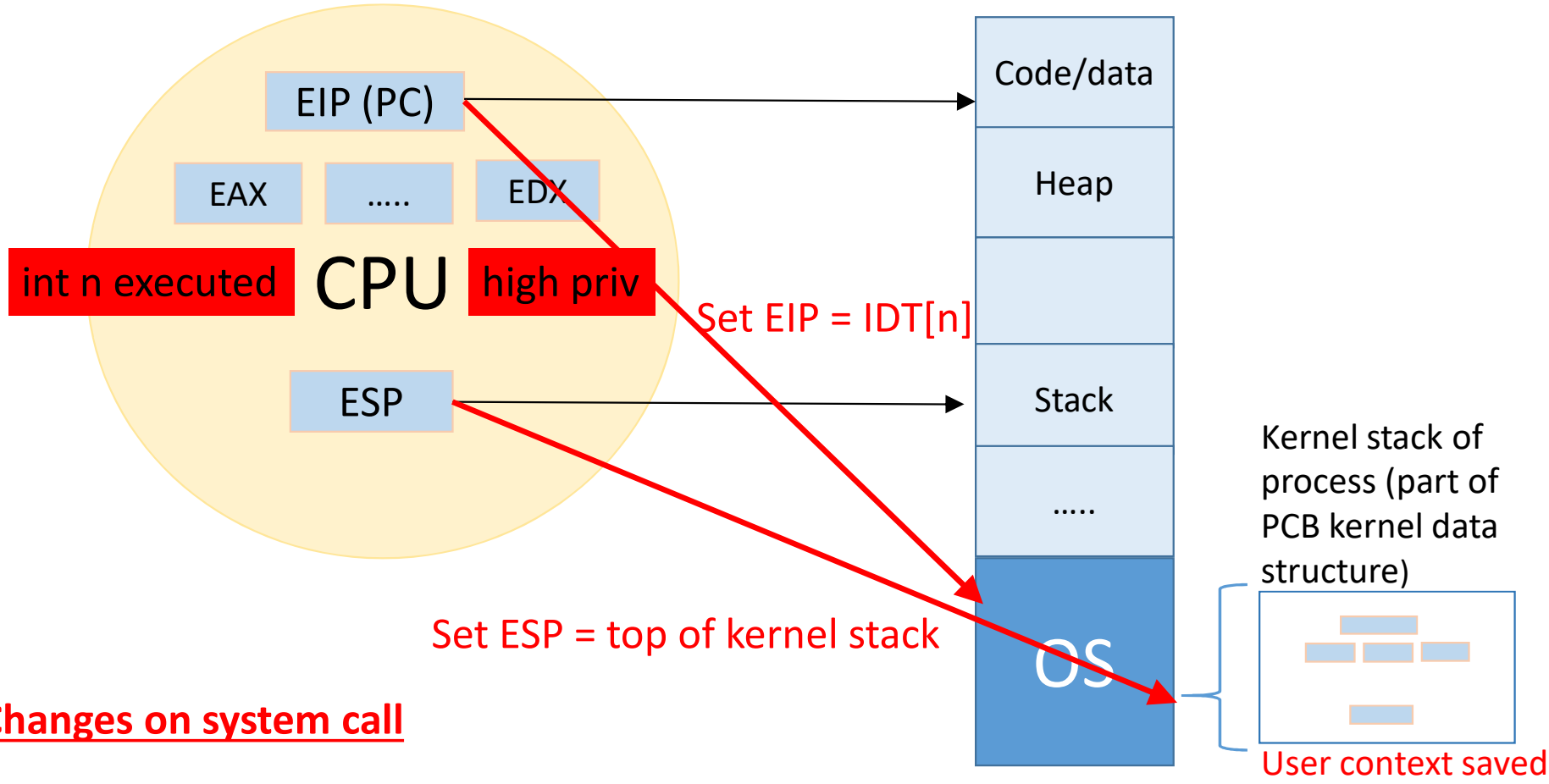
Kernel stack and IDT

- Every process uses a separate kernel stack for running kernel code
  - Part of PCB of process, in OS memory, not accessible in user mode
  - Used like user stack, but for kernel mode execution
  - Context pushed on kernel stack during system call, popped when done
- To set PC, CPU accesses Interrupt Descriptor Table (IDT)
  - Data structure with addresses of kernel code to jump to for events
  - Setup by OS during bootup, not accessible in user mode
  - CPU uses IDT to locate address of OS code to jump to
- Together: secure way of locating OS code, secure stack for OS to run

Hardware trap instruction

- When user code wants to make system call, it invokes special "trap instruction" with an argument
  - Example: "int n" in x86, argument "n" indicates type of trap (syscall, interrupt)
  - The value of "n" specifies index into IDT array, which OS function to jump to
- When CPU runs the trap instruction:
  - CPU moves to higher privilege level
  - CPU shifts stack pointer register to kernel stack of process
  - Register context is saved on kernel stack (part of PCB)
  - Address of OS code to jump to is obtained from IDT, PC points to OS code
  - OS code starts to run, on a secure stack

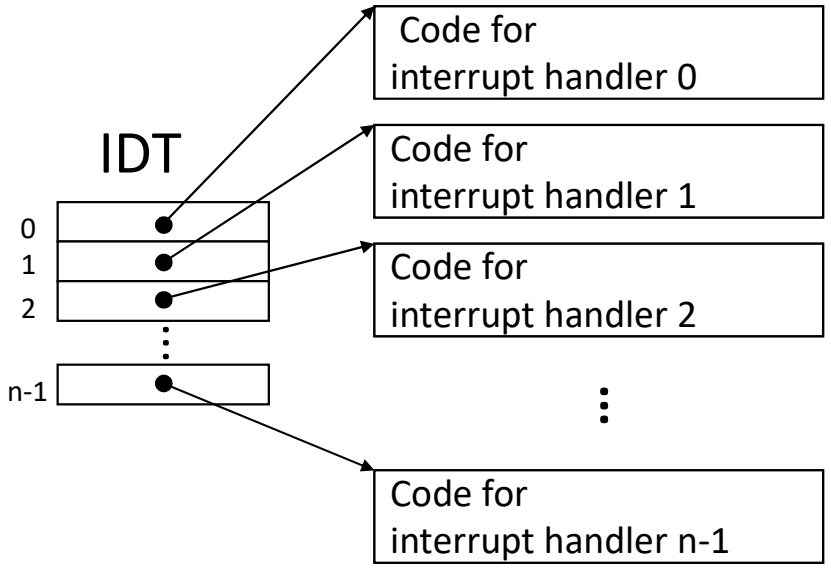
Trap handling



Changes on system call

IDT lookup

- IDT configured by OS
- Base address of IDT stored in CPU register
- Upon trap, CPU looks up IDT to find address of interrupt handler



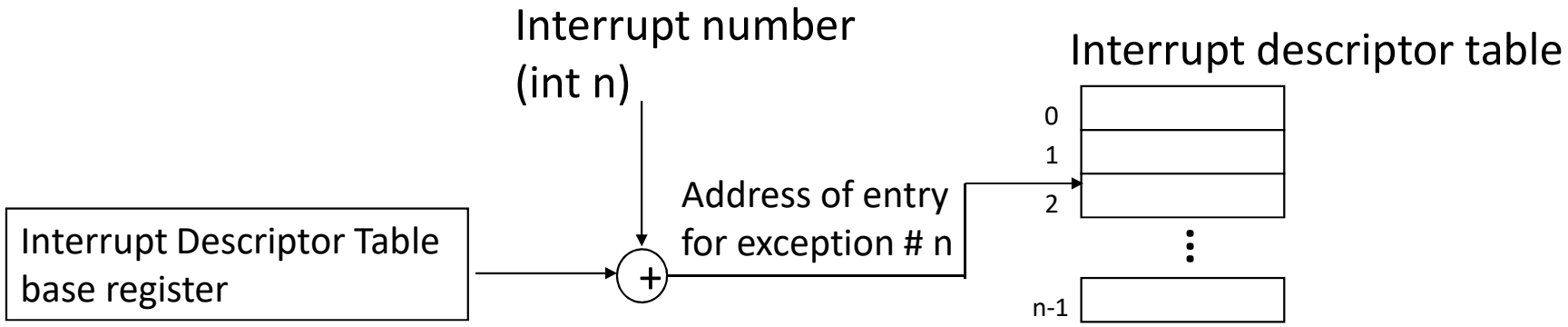


Image credit: CSAPP

Why trap instruction?

- Need a secure way of jumping to OS code to handle traps
  - User code cannot be trusted to jump to correct OS code
  - Only CPU can be trusted to handover control from user to OS securely
- Who calls trap instruction?
  - System call code in a language library (printf invokes system call via int n)
  - External hardware raises interrupt, causes CPU to execute “int n”
  - Argument “n” indicates whether system call /IRQ number of hardware device
- Across all cases, the mechanism is: save context on kernel stack, switch to OS address in IDT, run OS code to handle trap

Return from trap

- When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap
  - Restore context of CPU registers from kernel stack
  - Change CPU privilege from kernel mode to user mode
  - Restore PC and jump to user code after trap
- User process unaware that it was suspended, resumes execution at the point it stopped before
- Always return to the same user process from kernel mode? No
  - Before returning to user mode, OS checks if it must switch to another process

Why switch between processes?

- Sometimes when OS is in kernel mode, it cannot return back to the same process that was running in user mode before
  - Process has exited or must be terminated (e.g., segfault)
  - Process has made a blocking system call
- Sometimes, the OS does not want to return back to the same process
  - The process has run for too long
  - Must timeshare CPU with other processes
- In such cases, OS performs a context switch from one process to another
  - Switch from kernel mode of one process to kernel mode of another
  - OS scheduler decides which process to run next and switches to it

OS scheduler

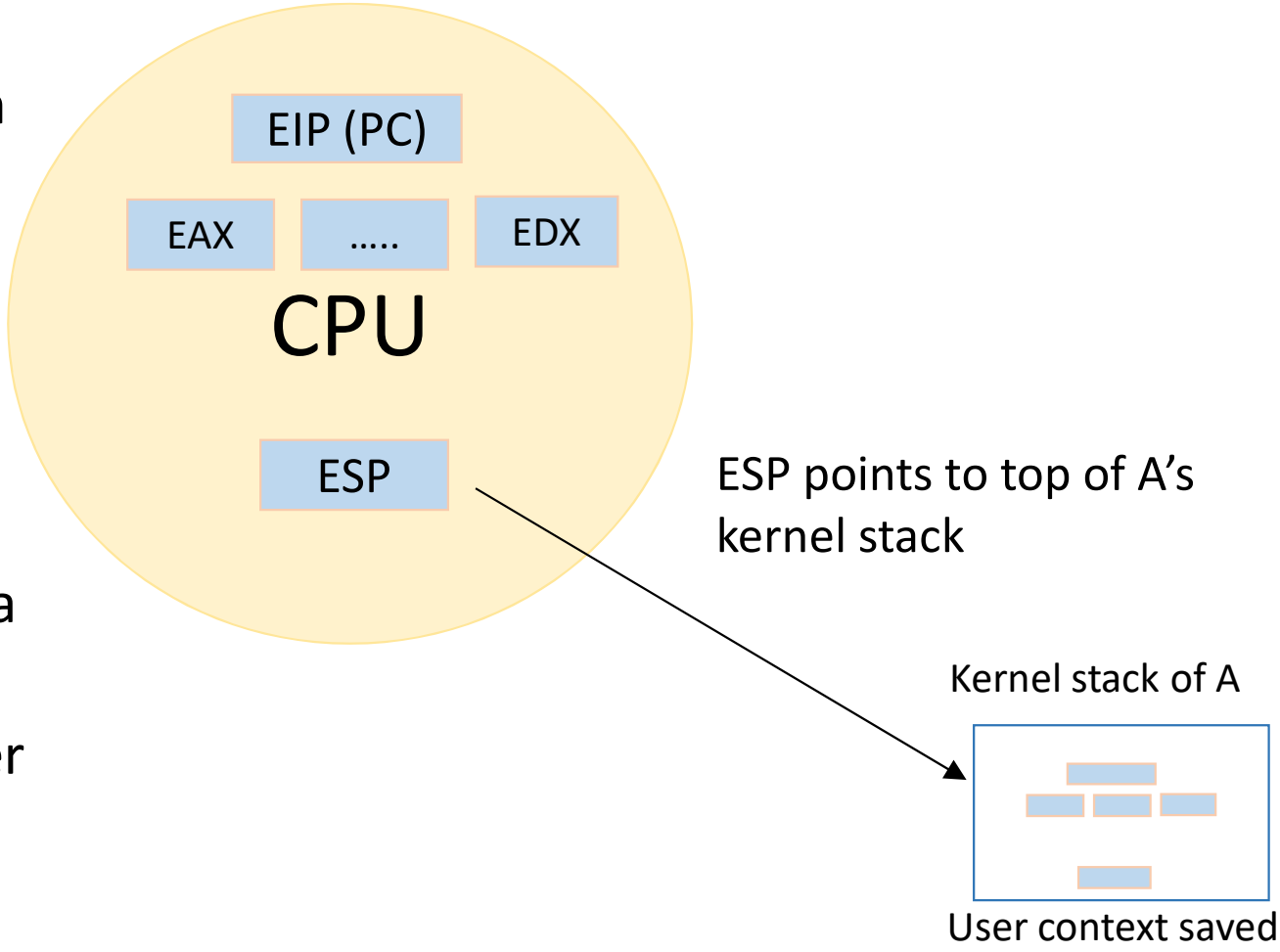
- OS maintains list of all active processes (PCBs) in a data structure
  - Processes added during fork, removed after clean up in wait
- OS scheduler is special code in the OS that periodically loops over this list and picks processes to run
- Basic outline of scheduler code
  - When invoked, save context of currently running process in its PCB
  - Loop over all ready/runnable processes and identify a process to run next
  - Restore context of new process from PCB and get it to run on CPU
  - Repeat this process as long as system is running

Scheduling and context switching

- OS scheduling involves two tasks
  - Policy to pick which process to run (next lecture)
  - Mechanism to switch to that process (this lecture)
- Non preemptive (cooperative) schedulers are polite
  - Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
  - CPU generates periodic timer interrupt
  - After servicing interrupt, OS checks if the current process has run for too long

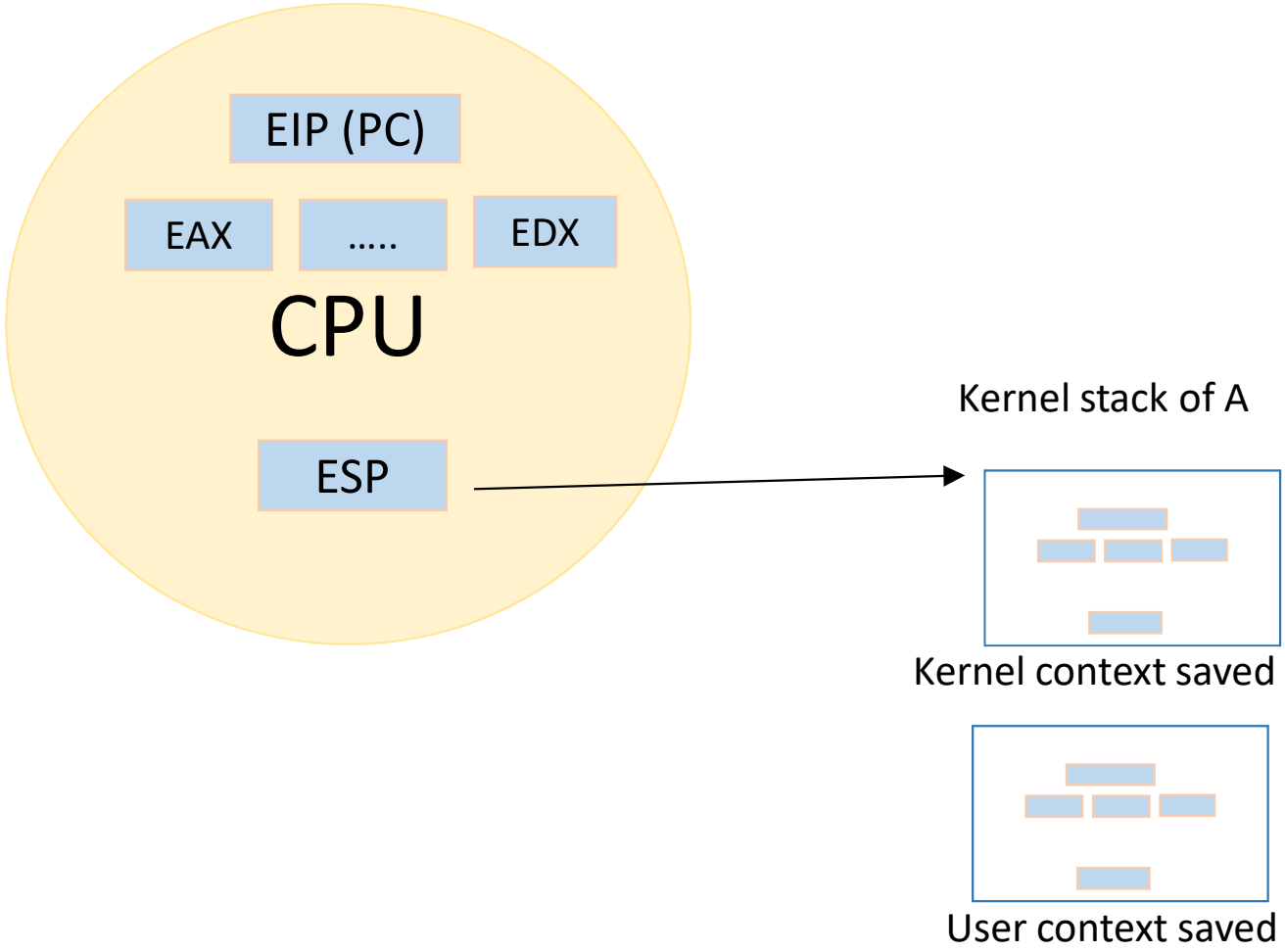
Mechanism of context switch (1)

- Process A has moved from user to kernel mode
- Kernel stack of A already has user register context
- After running for some time in kernel mode, A cannot run anymore (e.g., disk read initiated but data takes time to arrive)
- OS scheduler picks another process B to run next



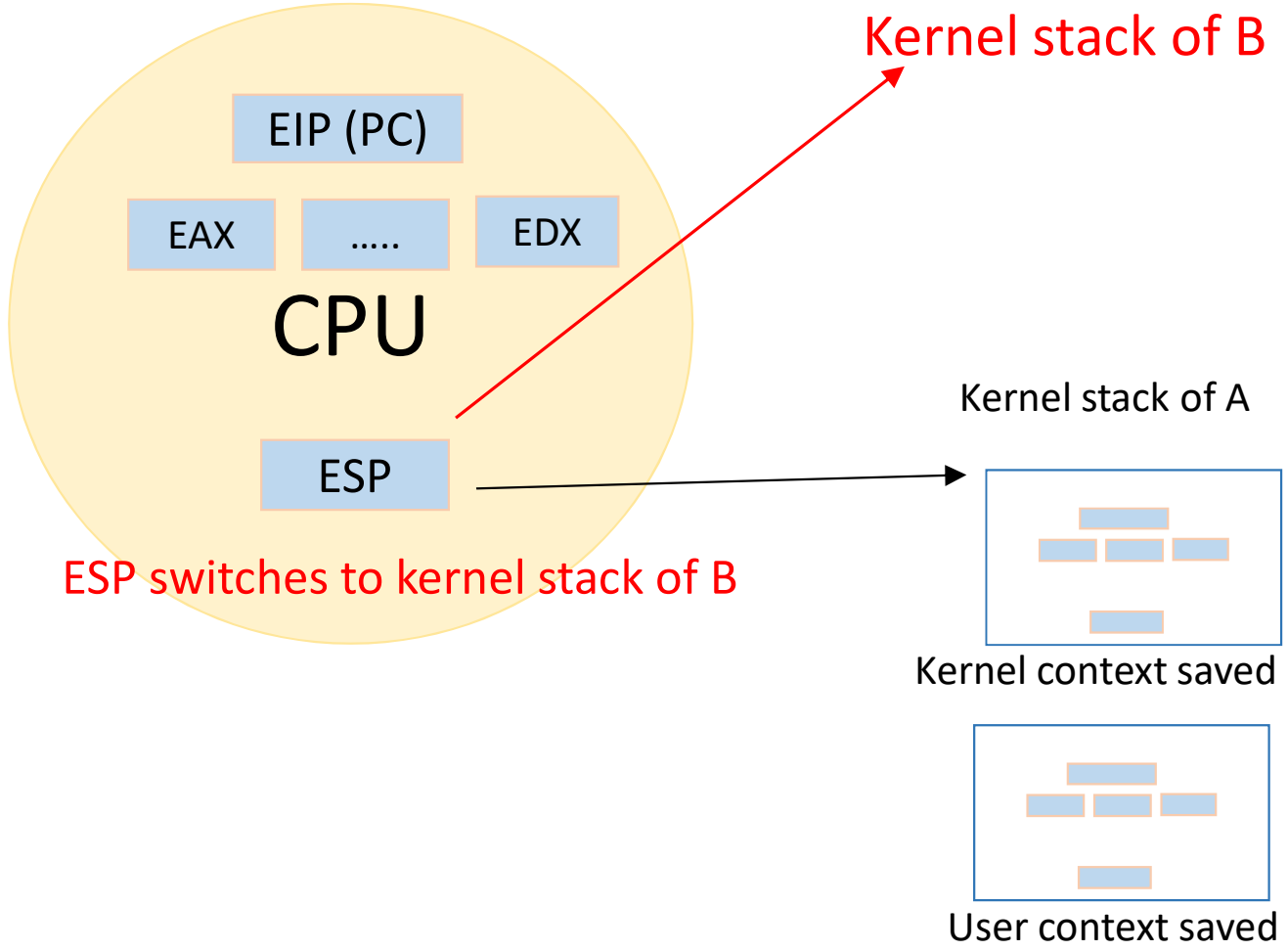
Mechanism of context switch (2)

- OS saves kernel context (PC, registers, kernel stack pointer) of A on kernel stack
- Why save context again?
- User context captures where execution stopped in user mode
- Kernel context captures where execution stopped in kernel mode



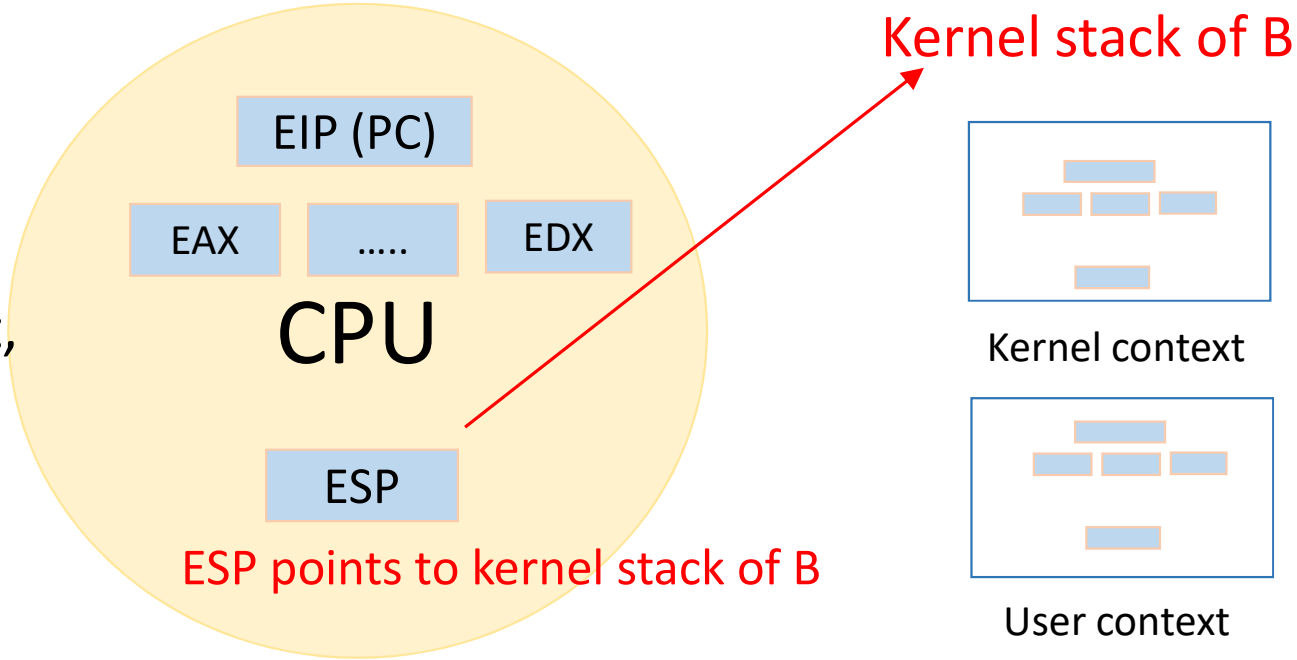
Mechanism of context switch (3)

- The actual moment of the context switch: OS switches ESP from kernel stack of A to kernel stack of next process B
- What will we find on the kernel stack of B?
  - Whatever the OS stored on it when it switched B out in the past



Mechanism of context switch (4)

- Kernel stack of B contains kernel context and user context of B
- OS restores kernel context, resumes execution in kernel mode of B, at the point it gave up CPU
- OS pops user context, resumes execution in user mode of B where it trapped into OS
- Context switch complete!



Understand saving and restoring context

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user code you stopped at) is saved on kernel stack by the trap instruction
  - Restored by return-from-trap when process goes to user mode
- During a context switch, kernel context (e.g., where you stopped in the OS code) is saved on the kernel stack by the context switching code
  - Restored when the process is ready to run and switched back in again