CS330: Operating Systems

Limited direct execution

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?
  - The OS performs multiplexing of physical resources efficiently
  - Maintains mapping of virtual view to physical resource

Virtualization: Efficiency/performance

- Resource virtualization should not add any overheads
- Efficient when programs use the resources directly, no OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?

Virtualization: Efficiency/performance

- Resource virtualization should not add any overheads
- Efficient when programs use the resource directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Virtualization: Efficiency/performance

- Resource virtualization should not add any overheads
- Efficient when programs use the resource directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Conclusion: Some limits to direct access must be enforced.

Limited direct execution

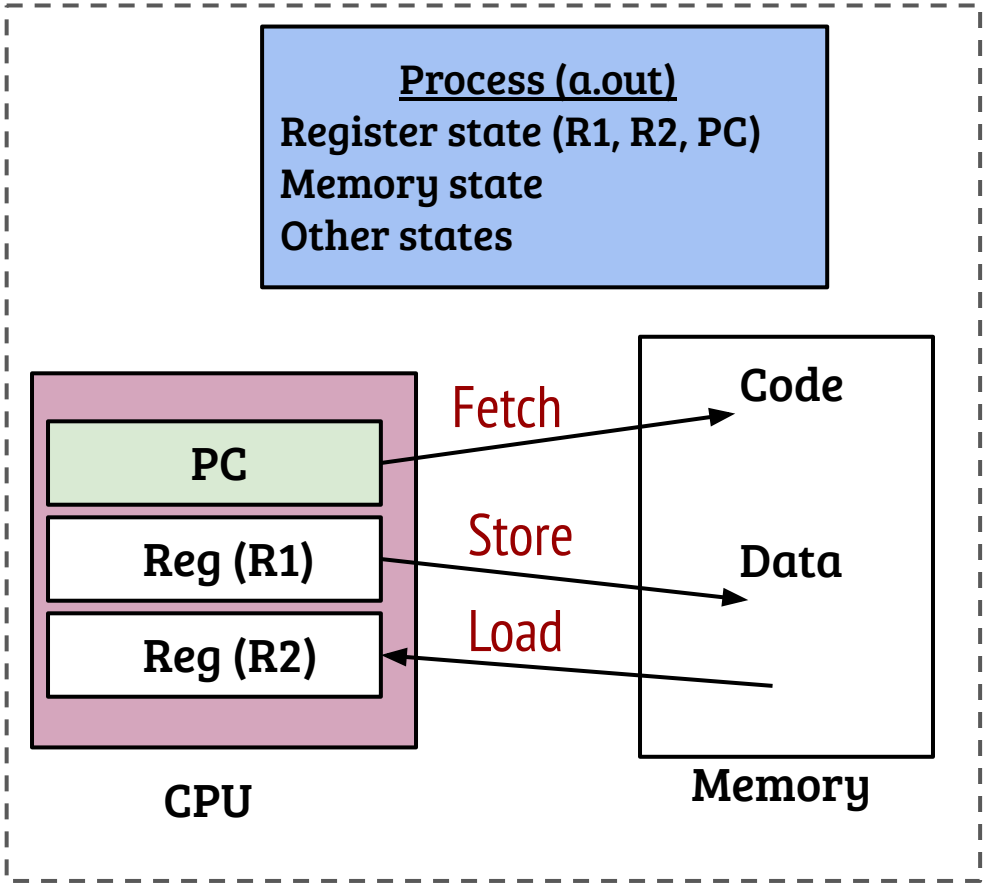
- Can the OS enforce limits to an executing process by itself?

A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

OS

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



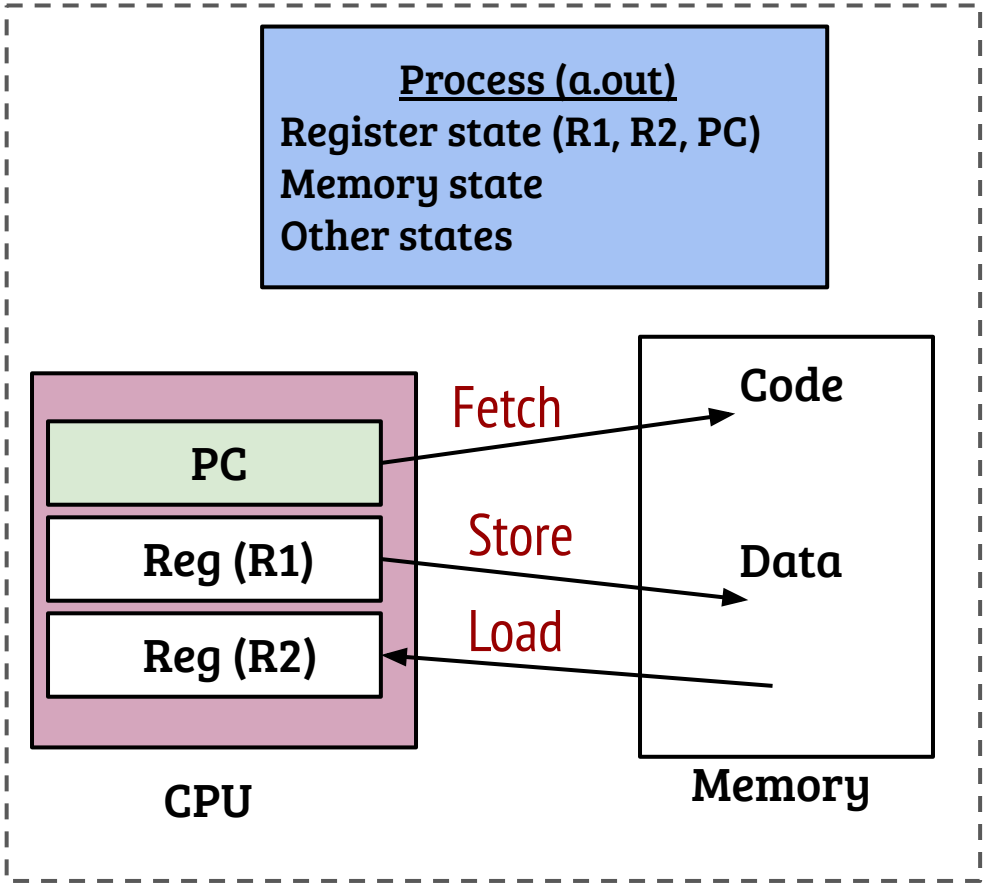
A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

Help me!

OS

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



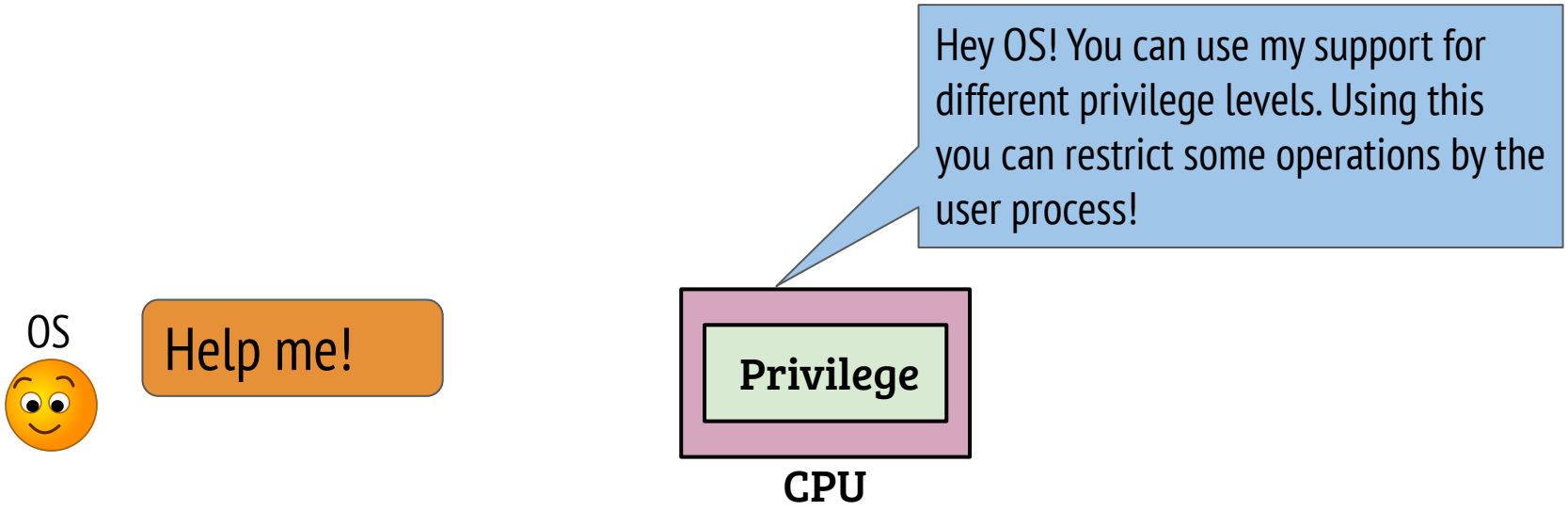
Limited direct execution

- Can the OS enforce limits to an executing process by itself?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!

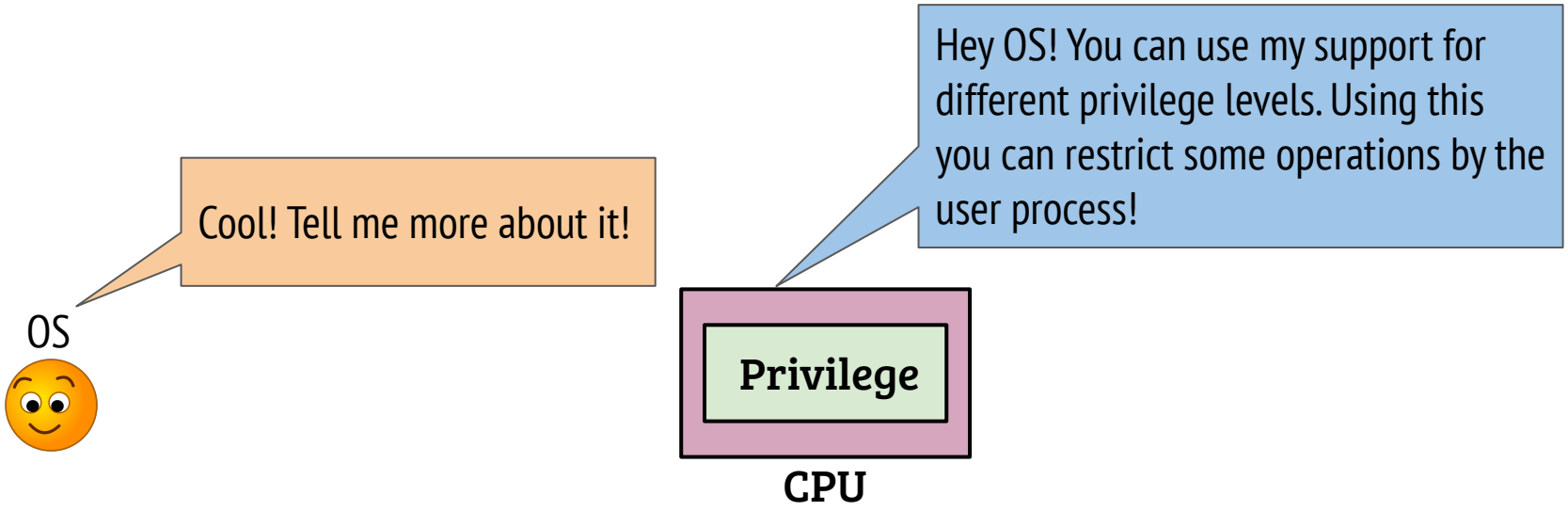
Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?

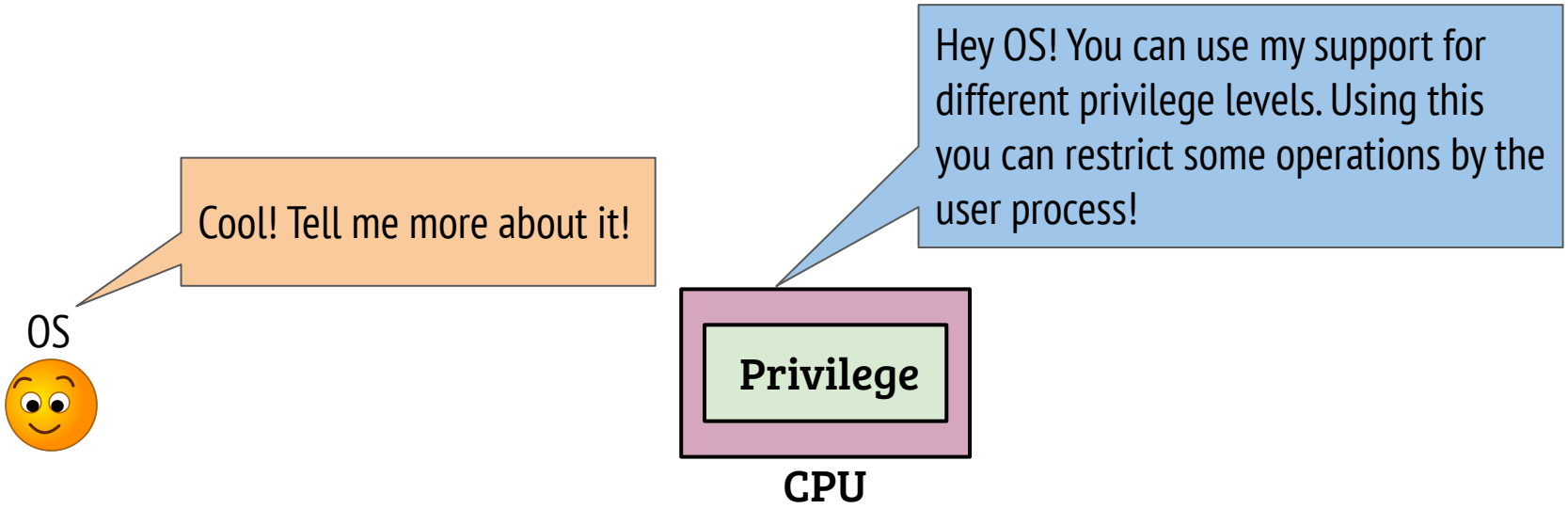
Hardware support: Privilege levels



Hardware support: Privilege levels

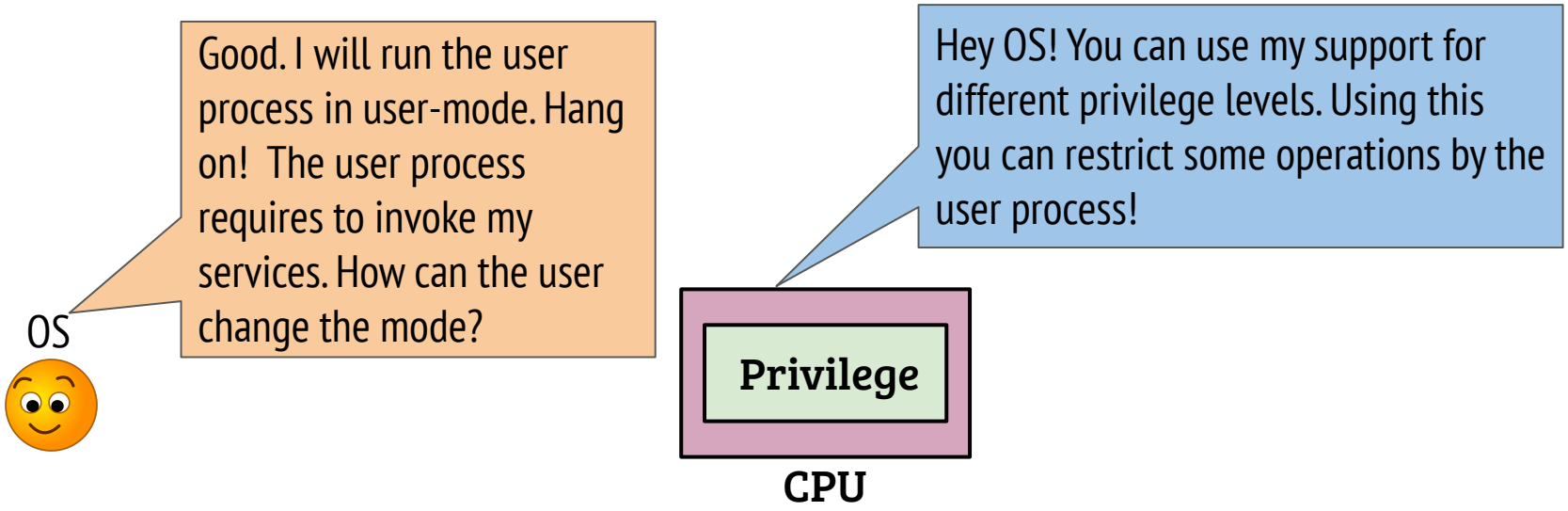


Hardware support: Privilege levels

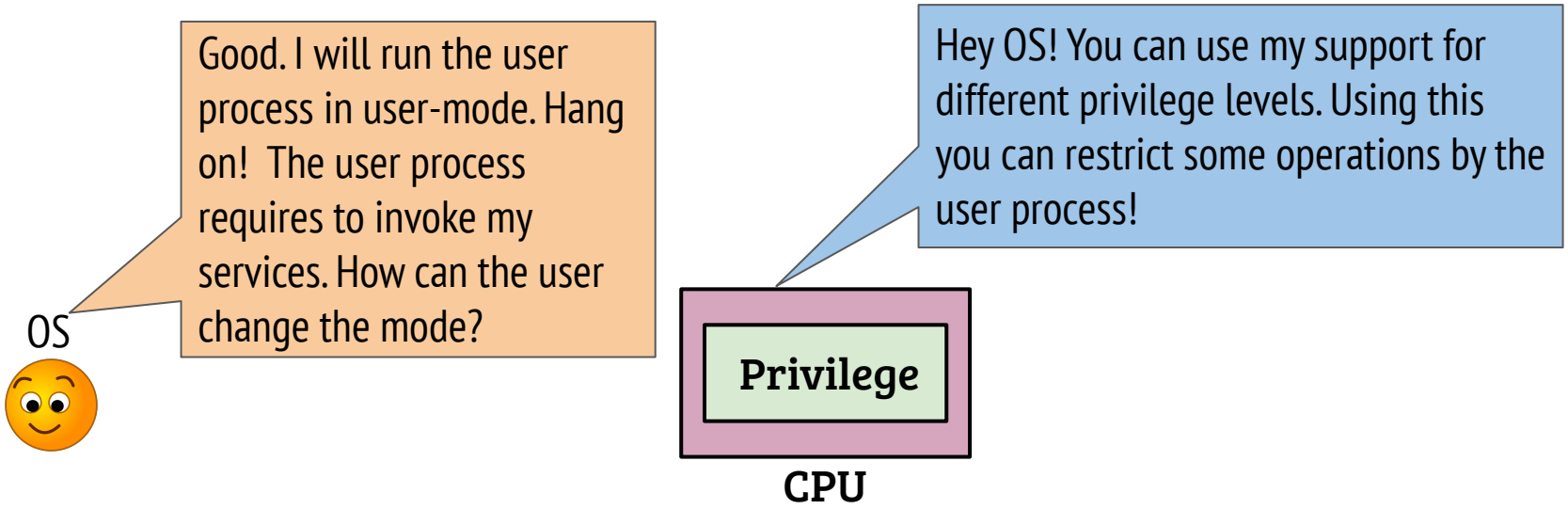


- CPU can execute in two modes: *user-mode* and *kernel-mode*
- Some operations are allowed only from kernel-mode (privileged OPs)
  - If executed from user mode, hardware will notify the OS by raising a fault/trap

Hardware support: Privilege levels

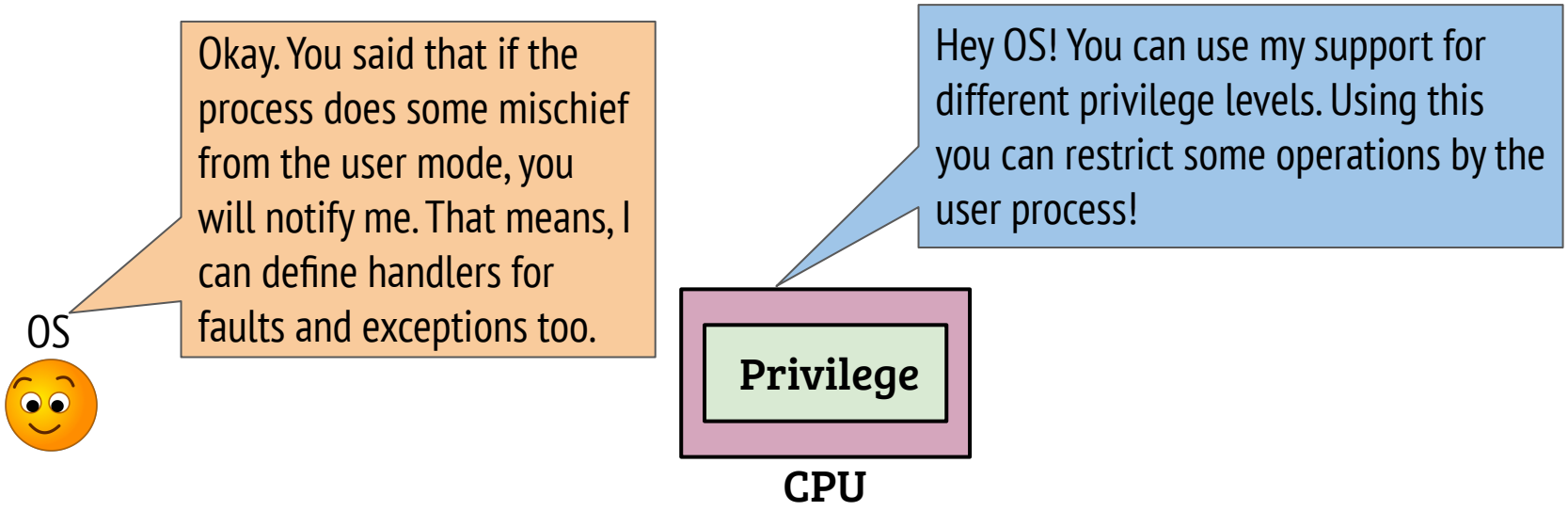


Hardware support: Privilege levels

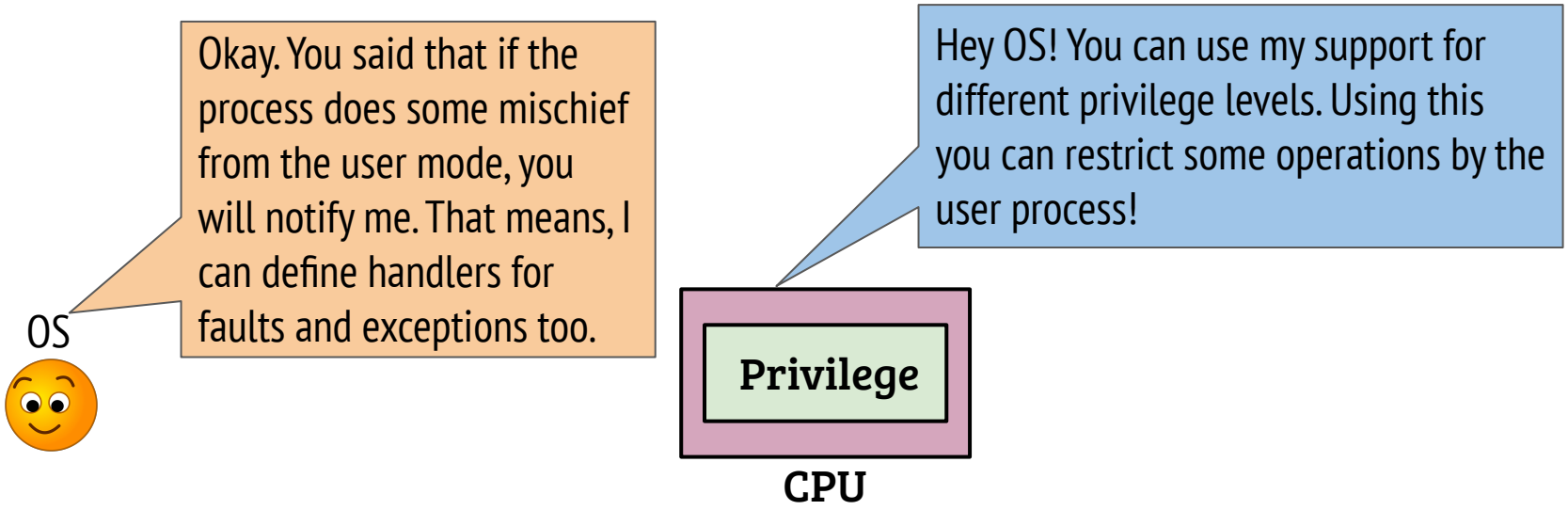


- From user-mode, privilege level of CPU can not be changed directly
- The hardware provides entry instructions from the user-mode which causes a mode switch
- The OS can define the handler for different entry gates

Hardware support: Privilege levels

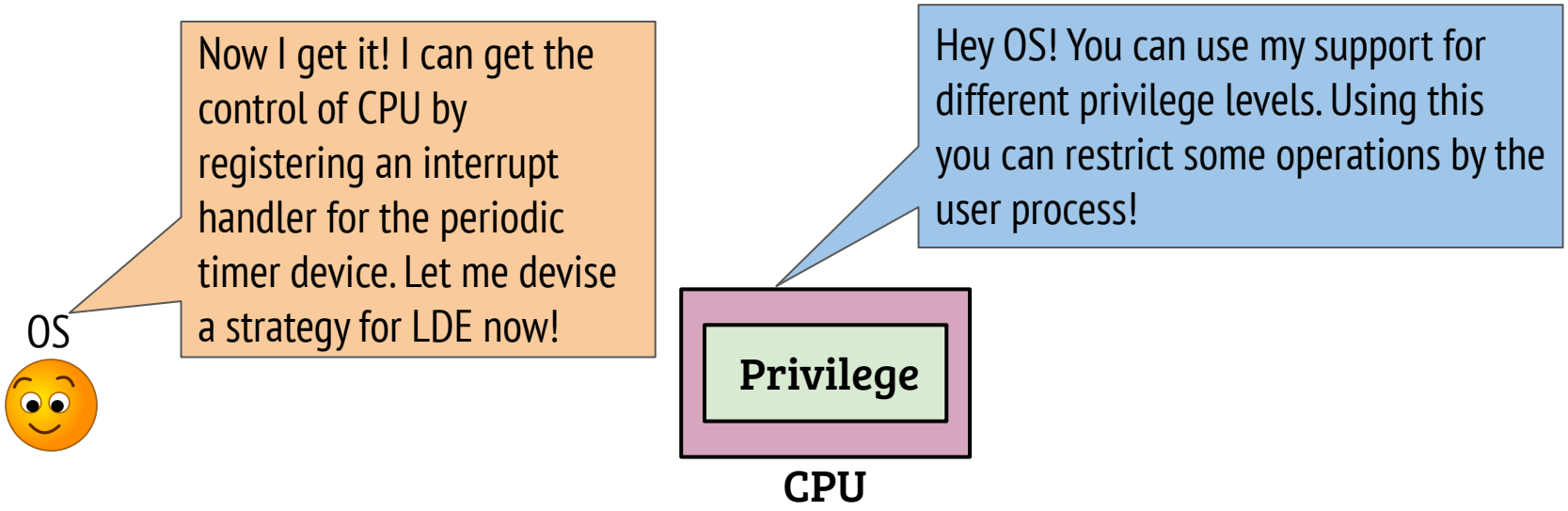


Hardware support: Privilege levels

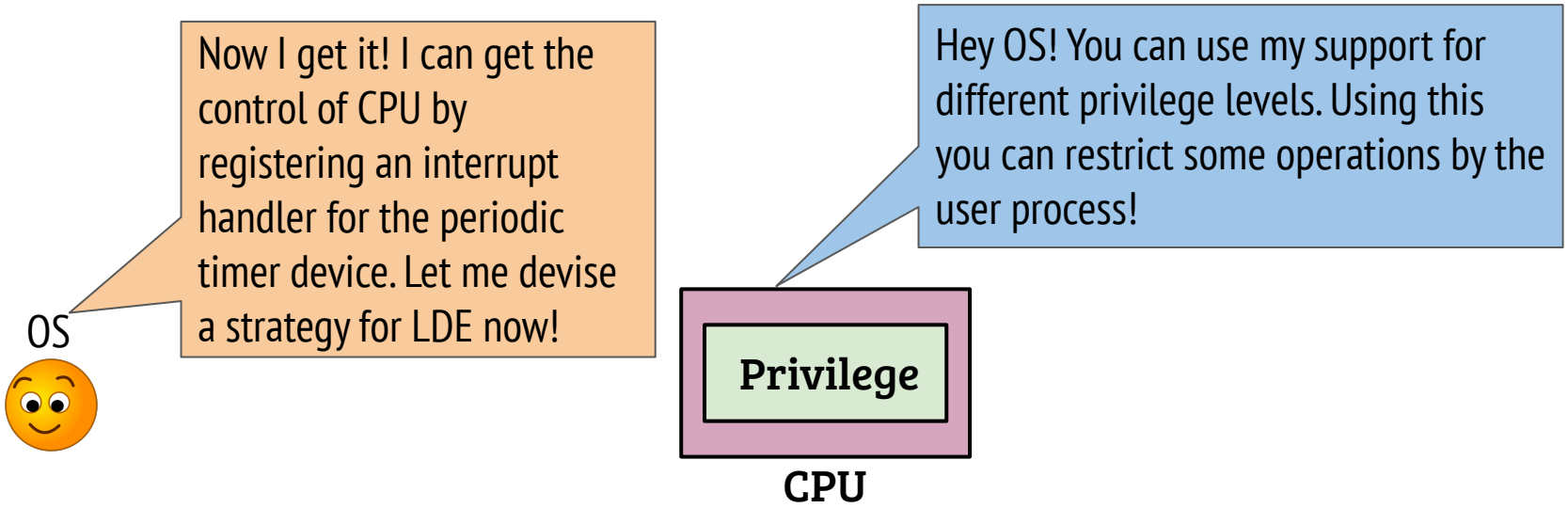


- The OS can register the handlers for faults and exceptions
- The OS can also register handlers for device interrupts
- *Registration of handlers is privileged!*

Hardware support: Privilege levels



Hardware support: Privilege levels



- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts
- The handler code is invoked by the hardware when user invokes a system call or an exception or an external interrupt

Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

CS330: Operating Systems

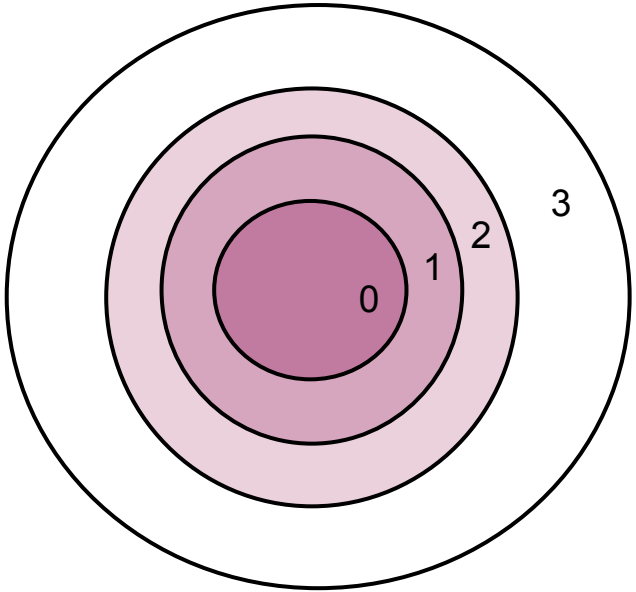
Privileged ISA (X86_64)

Recap: Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

Today's agenda: High-level view of x86_64 support for privileged mode

X86: rings of protection



- 4 privilege levels: 0⟶ highest, 3⟶ lowest
- Some operations are allowed only in privilege level 0
- Most OSes use 0 (for kernel) and 3 (for user)
- Different kinds of privilege enforcement
  - Instruction is privileged
  - Operand is privileged

Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
  asm("hlt;");
}
```

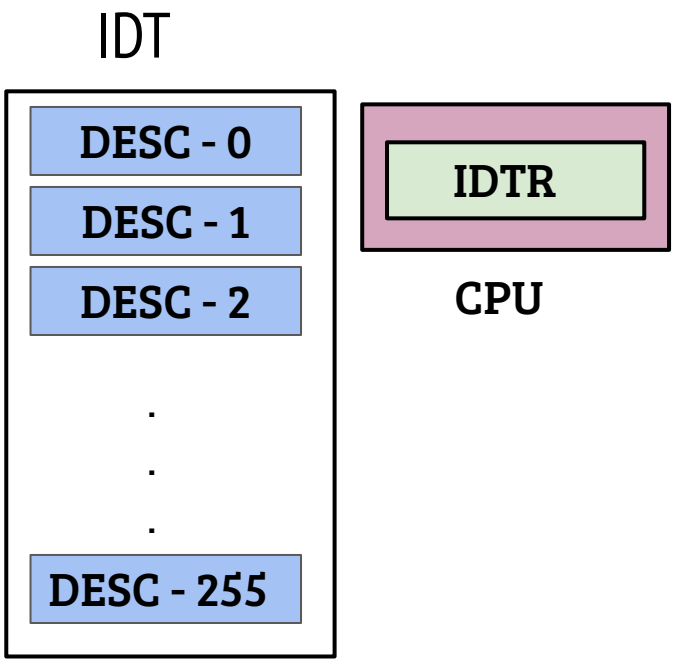
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
  unsigned long cr3_val;
  asm volatile("mov %%cr3, %0;"
            : "=r" (cr3_val)
            :: );
 printf("%lx\n", cr3_val);
}
```

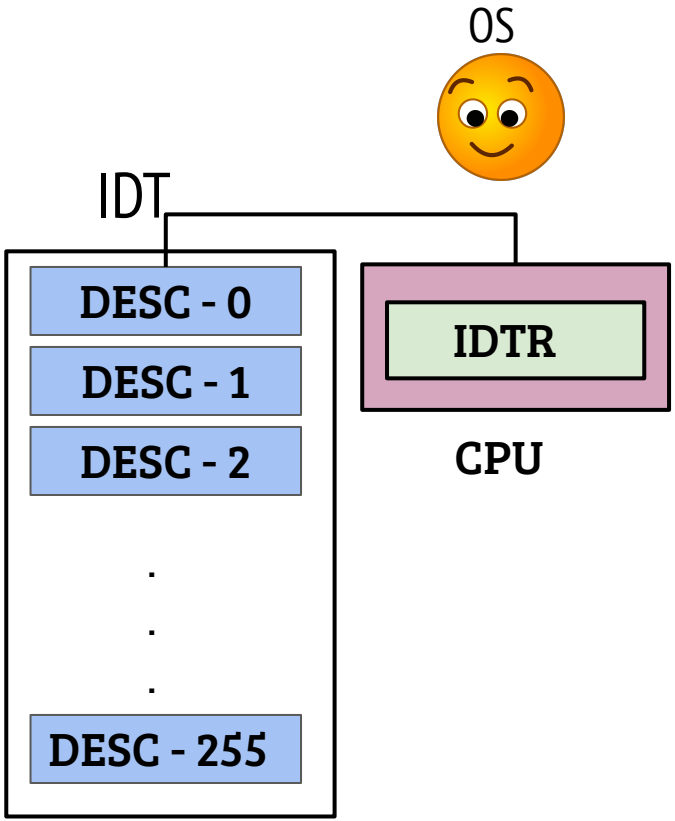
- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

Interrupt Descriptor Table (IDT): gateway to handlers



- Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
- Descriptors 0-31 are for predefined events e.g., 0 ⟶ Div-by-zero exception etc.
- Events 32-255 are user defined, can be used for h/w and s/w interrupt handling

Defining the descriptors (OS boot)



- Each descriptor contains information about handling the event
  - Privilege switch information
  - Handler address
- The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using LIDT instruction)

System call using software interrupt (INT instruction)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function. How?

System call using software interrupt (INT instruction)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function, how?
  - Every system call is associated with a number (defined by OS)
  - User process sends information like system call number, arguments through CPU registers which is used to invoke the actual handler

System call in Linux using INT

- Linux kernel defines system call handler for descriptor 0x80
- System call number is passed in EAX register and the return value is stored in EAX register
- Parameters are passed using the registers in the following order
  - EAX (syscall #), EBX (param #1), ECX (param #2), EDX(param #3), ESI (param #4), EDI (param #5), EBP (param #6)
- Syscall numbers can be found at /usr/include/x86_64-linux-gnu/asm/unistd_32.h

CS330: Operating Systems

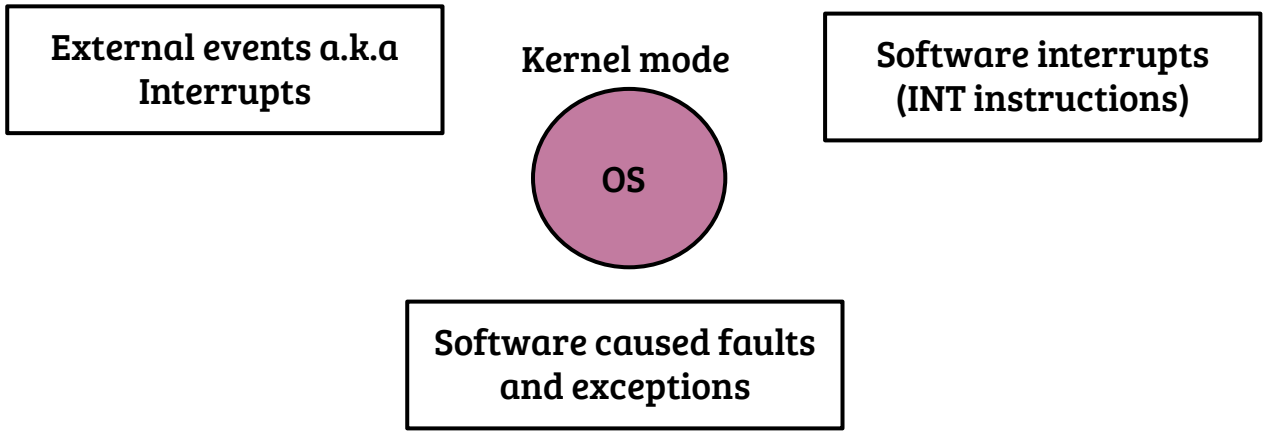
OS mode execution

Recap: Limited direct execution support in X86

- What kind of support is needed from the hardware?
- CPU privilege levels, switching, entry points and handlers
- X86 support
  - privilege levels (ring-0 to ring-3)
  - interrupt descriptor table to define handlers for hardware and software entry points (system calls, interrupts, exceptions)
  - entry point behavior can be defined by the OS to enforce limitations on the user space execution

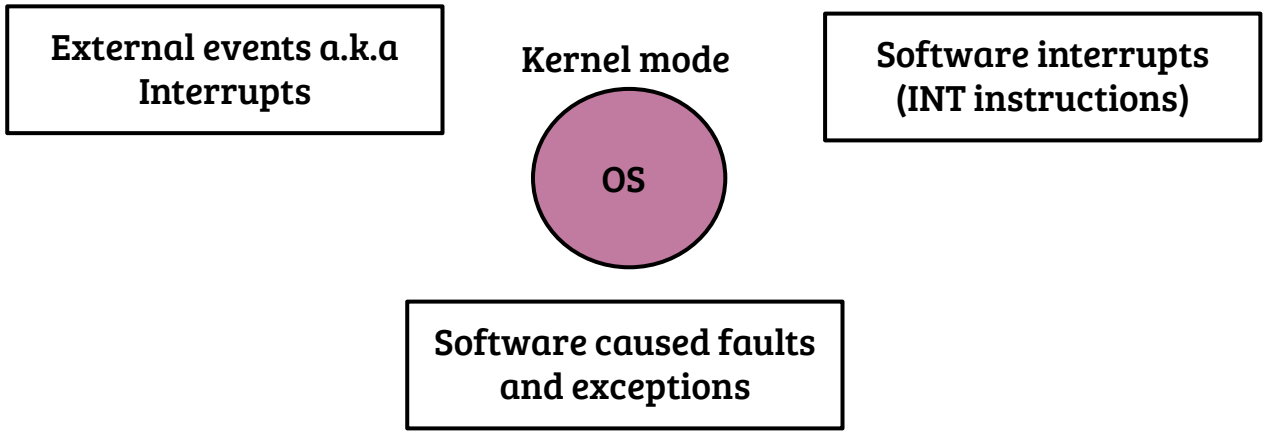
Today's agenda: Execution in privileged (kernel) mode

Post-boot OS execution



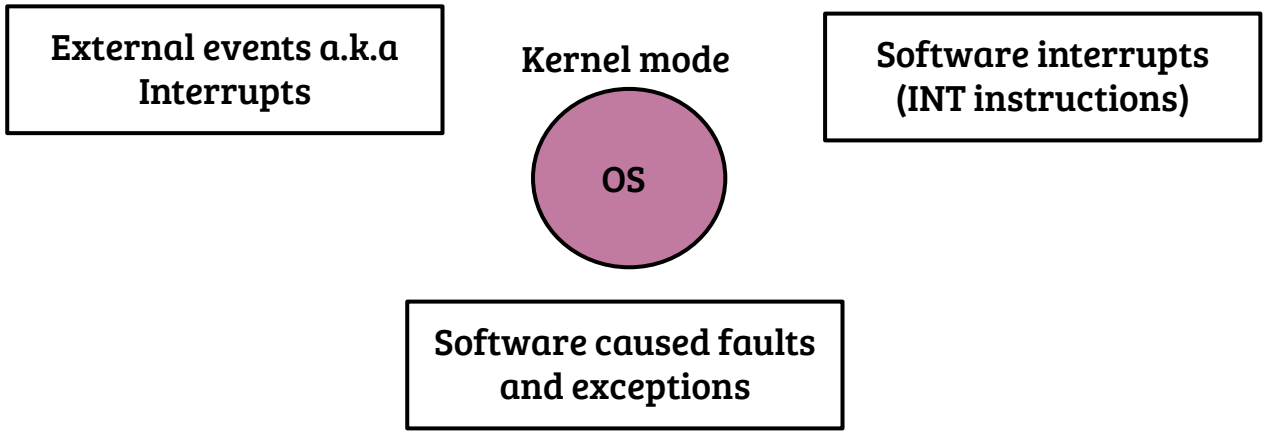
- OS execution is triggered because of interrupts, exceptions or system calls

Post-boot OS execution



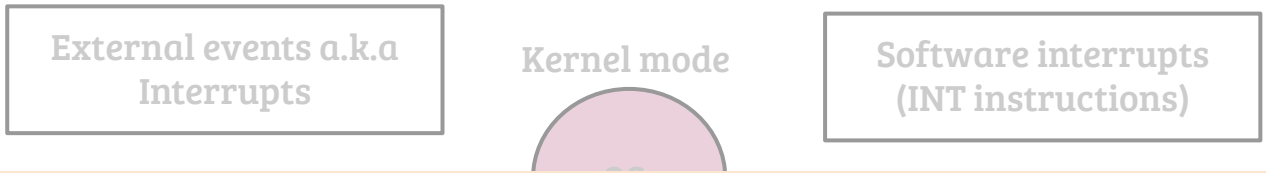
- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?

Post-boot OS execution



- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?
- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

Post-boot OS execution



- Does the OS need a separate stack?
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

for this event to happen. What can go wrong and how to handle it?

- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning

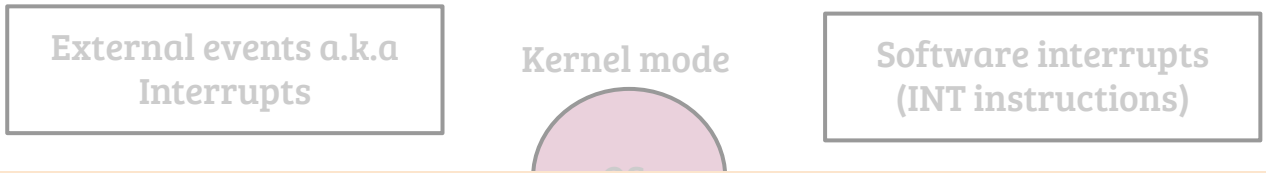
The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?
- On X86 systems, the hardware switches the stack pointer to the stack address configured by the OS

Post-boot OS execution



- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

to save the user execution state and restore it on return

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working?

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception
- What about external interrupts?

Management of OS stacks

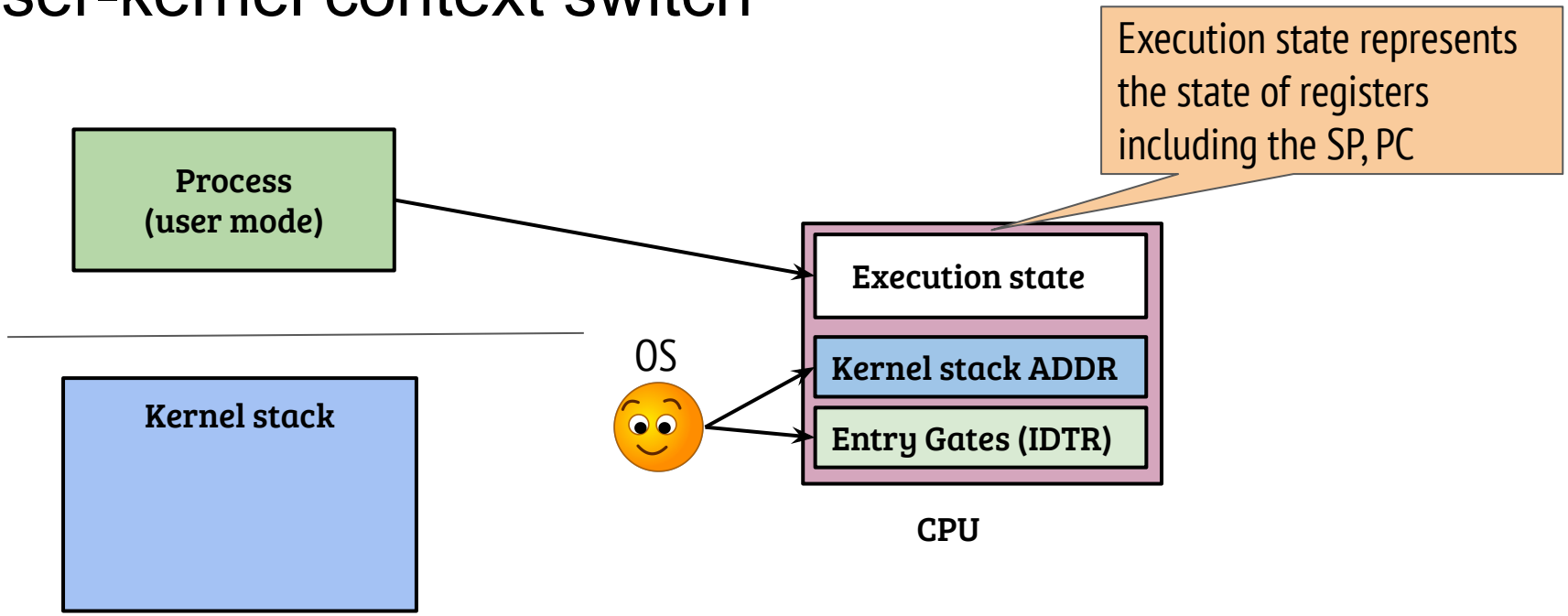
- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception
- What about external interrupts?
  - Separate interrupt stacks are used by OS for handling interrupts

Post-boot OS execution

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How is the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

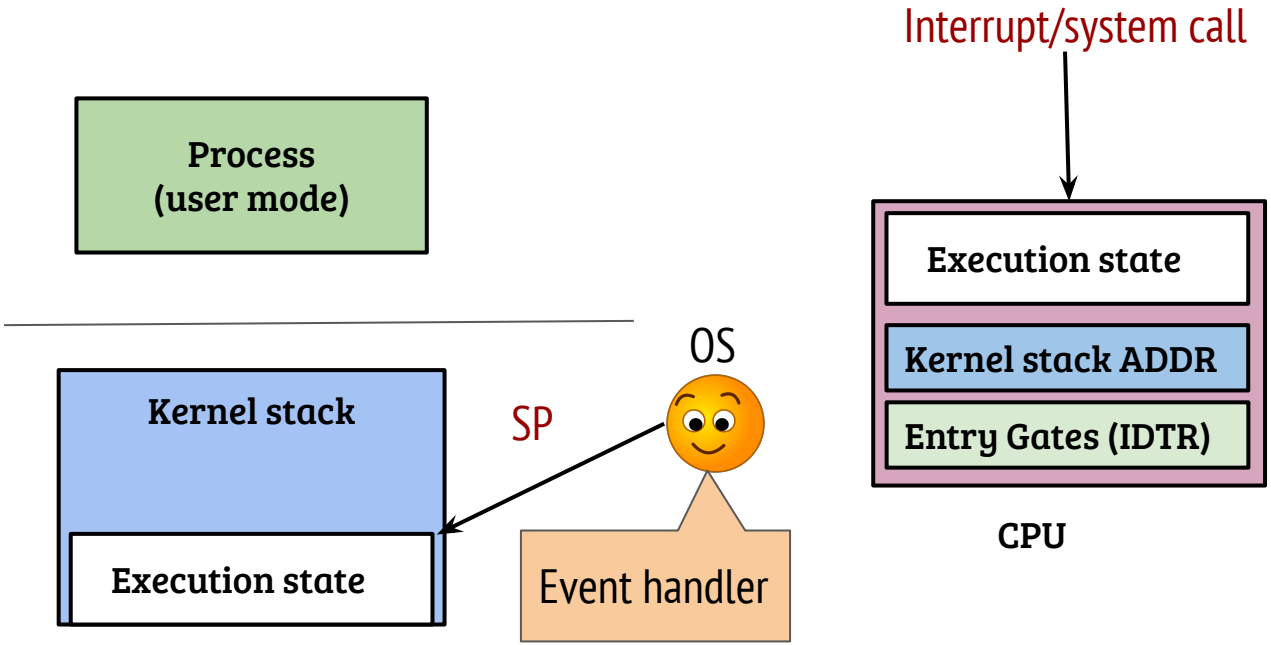
to save the user execution state and restore it on return

User-kernel context switch



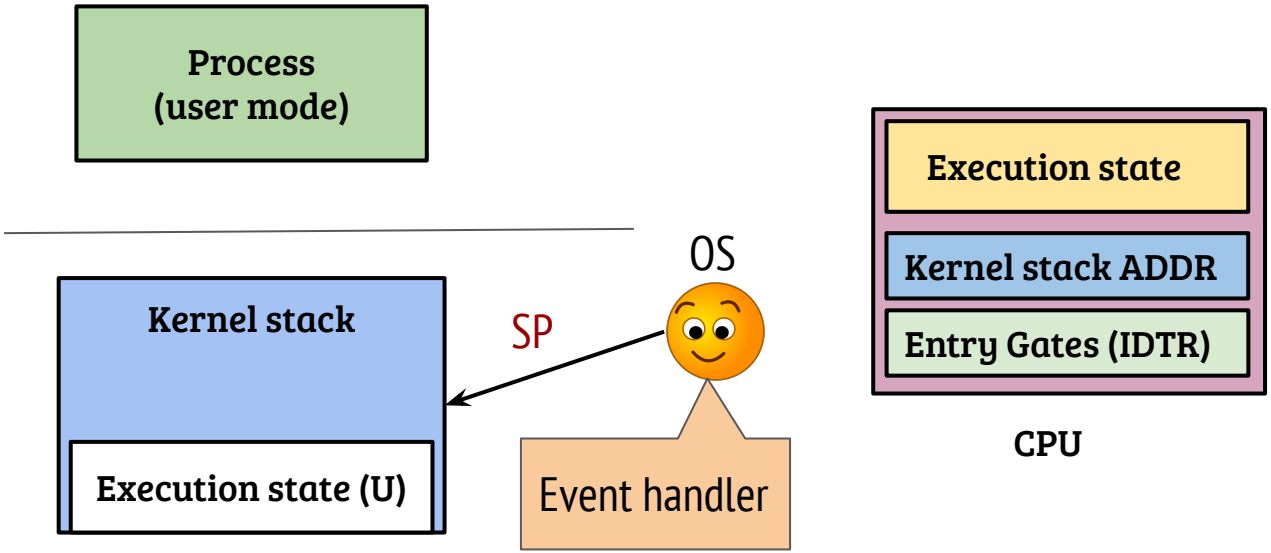
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



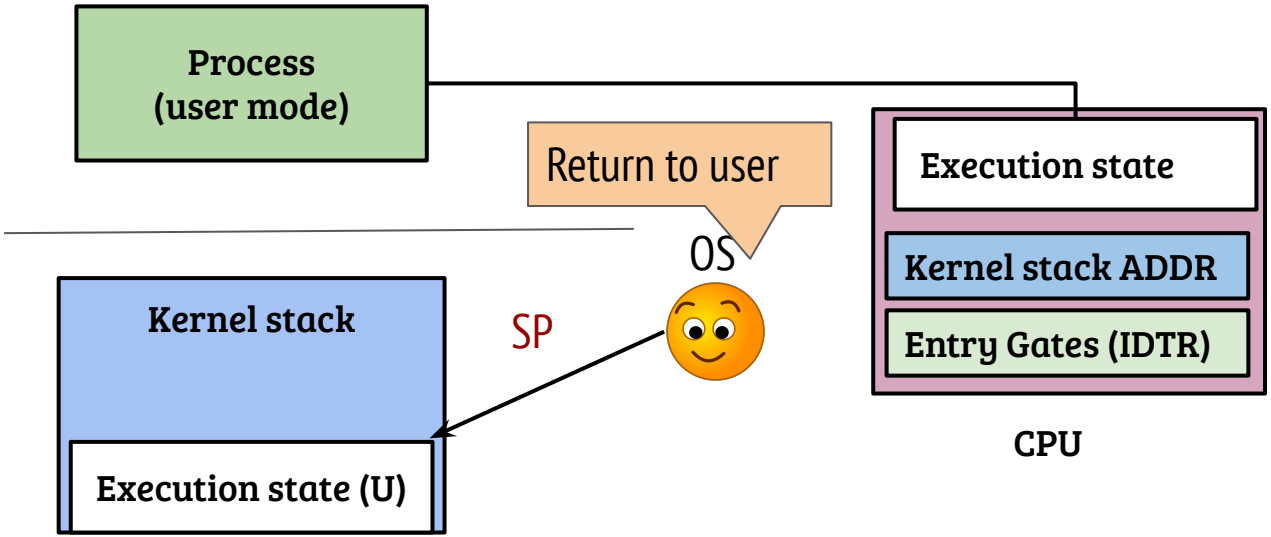
- The CPU saves the execution state onto the kernel stack
- The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)

User-kernel context switch



- The OS executes the event (syscall/interrupt) handler
  - Makes uses of the kernel stack
  - Execution state on CPU is of OS at this point

User-kernel context switch

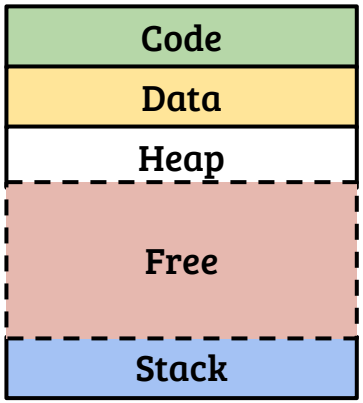


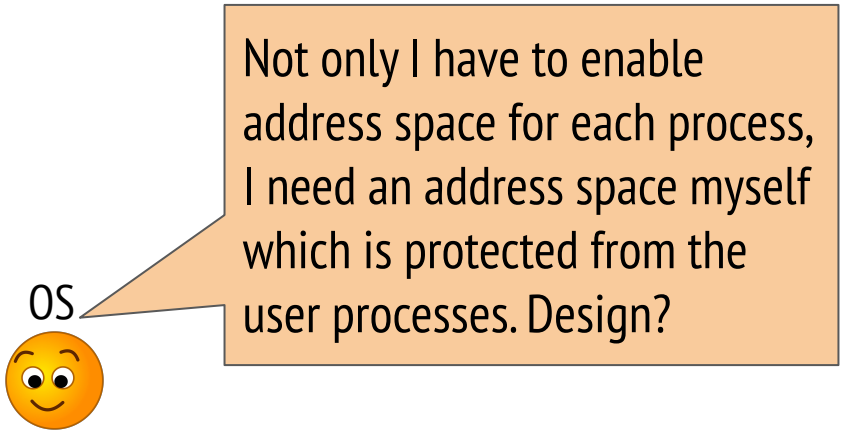
- The kernel stack pointer should point to the position at the time of entry
- CPU loads the user execution state and resumes user execution

Post-boot OS execution

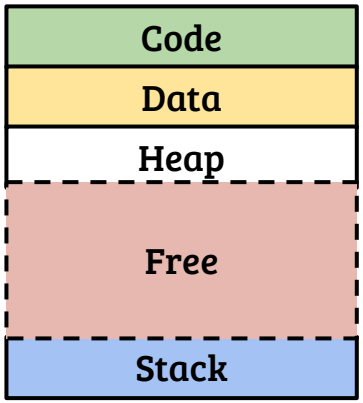
- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?

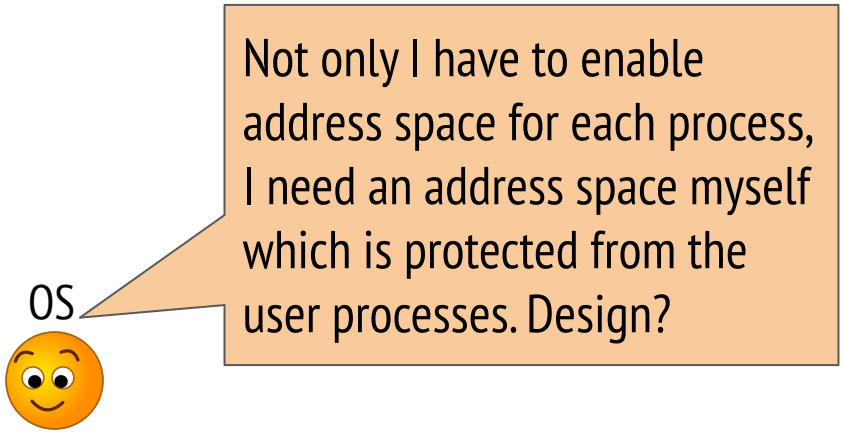
The OS address space





The OS address space





- Two possible design approaches
  - Use a separate address space for the OS, change the translation information on every OS entry (inefficient)
  - Consume a part of the address space from all processes and protect the OS addresses using H/W assistance (most commonly used)

Post-boot OS execution

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?
- A part of the process address space is reserved for OS and is protected