

CS330: Operating Systems

OS mode execution

Recap: Limited direct execution support in X86

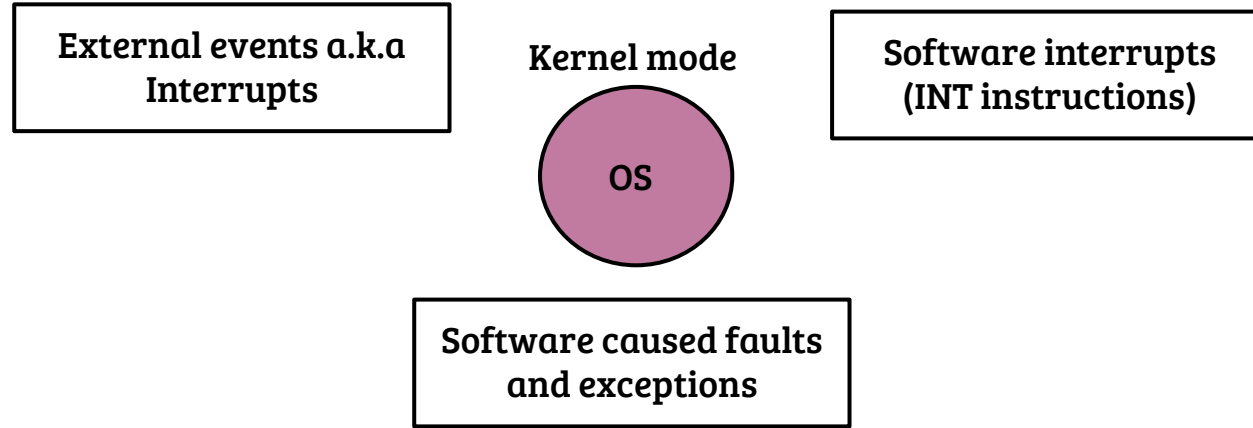
- What kind of support is needed from the hardware?
- CPU privilege levels, switching, entry points and handlers
- X86 support
 - privilege levels (ring-0 to ring-3)
 - interrupt descriptor table to define handlers for hardware and software entry points (system calls, interrupts, exceptions)
 - entry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

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

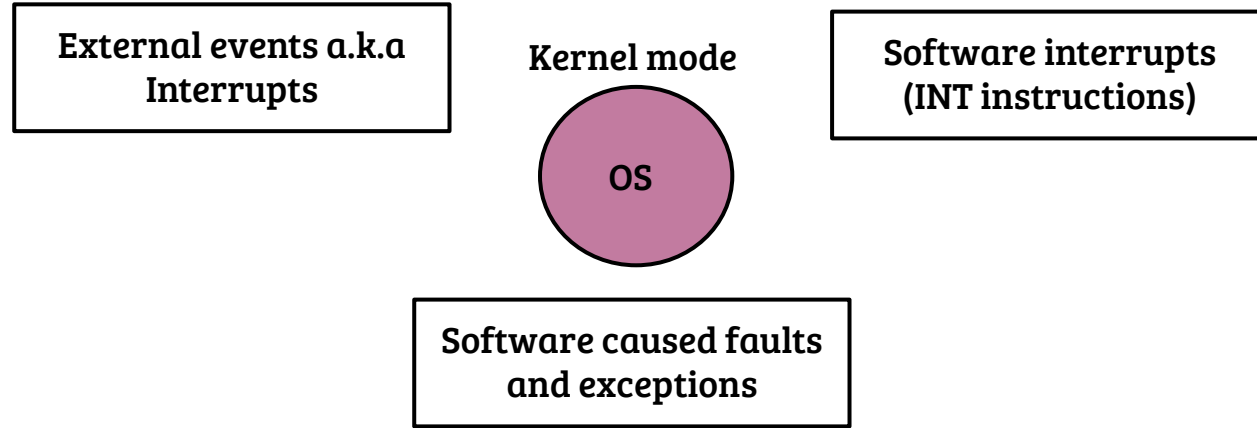
Agenda: Execution in privileged (kernel) mode

Post-boot OS execution



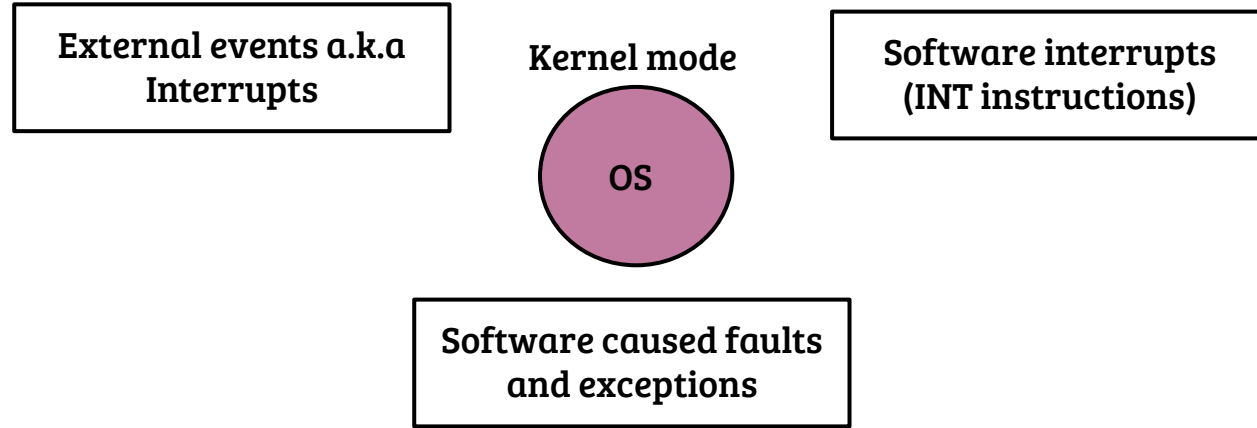
- OS execution is triggered because of interrupts, exceptions or system calls

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

External events a.k.a
Interrupts

Kernel mode

Software interrupts
(INT instructions)

- 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

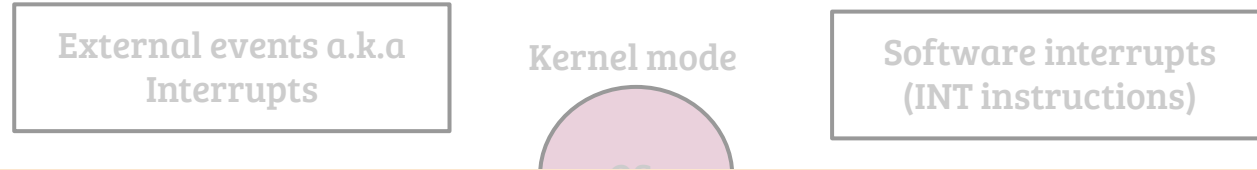
The OS stack

- 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?

The interrupted program may become corrupted after resuming. The OS needs 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

External events a.k.a
Interrupts

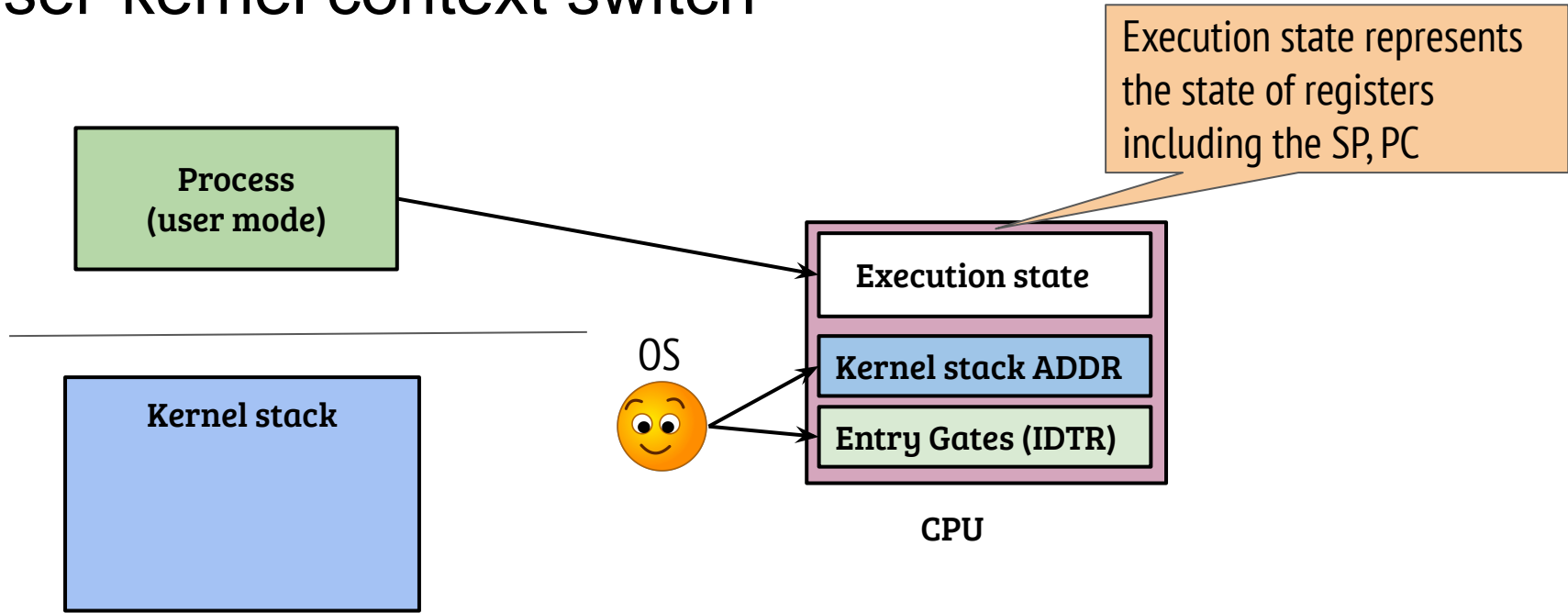
Kernel mode

Software interrupts
(INT instructions)

- 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?

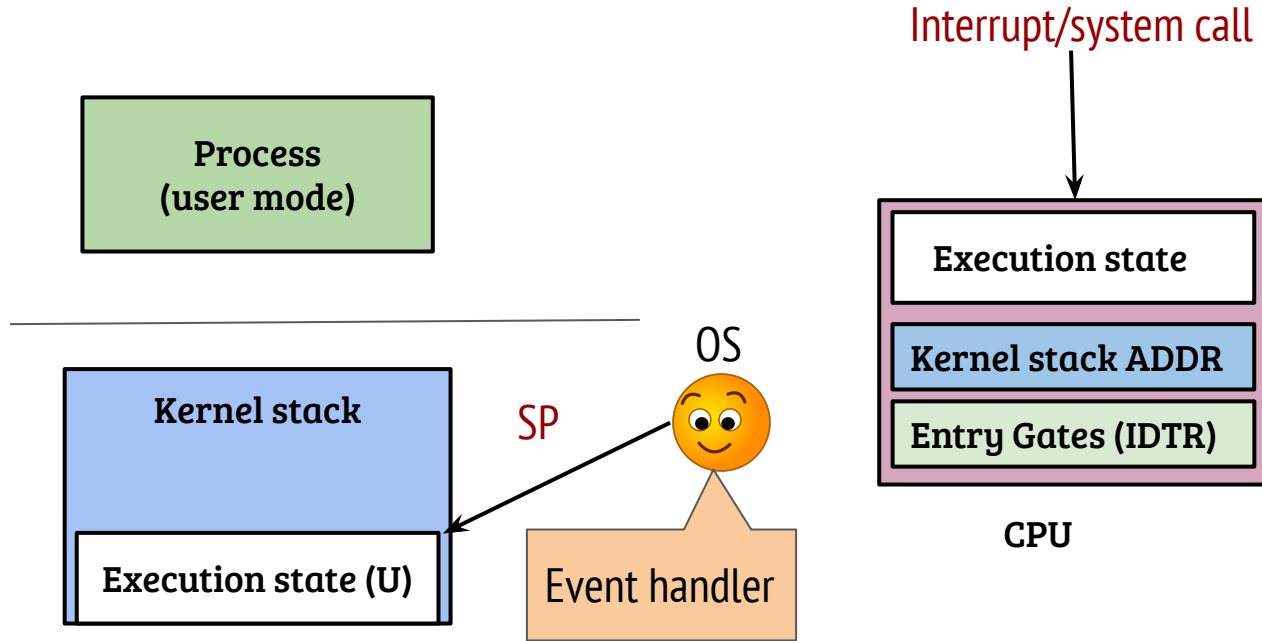
The interrupted program may become corrupted after resume. The OS needs to save the user execution state and restore it on return.

User-kernel context switch



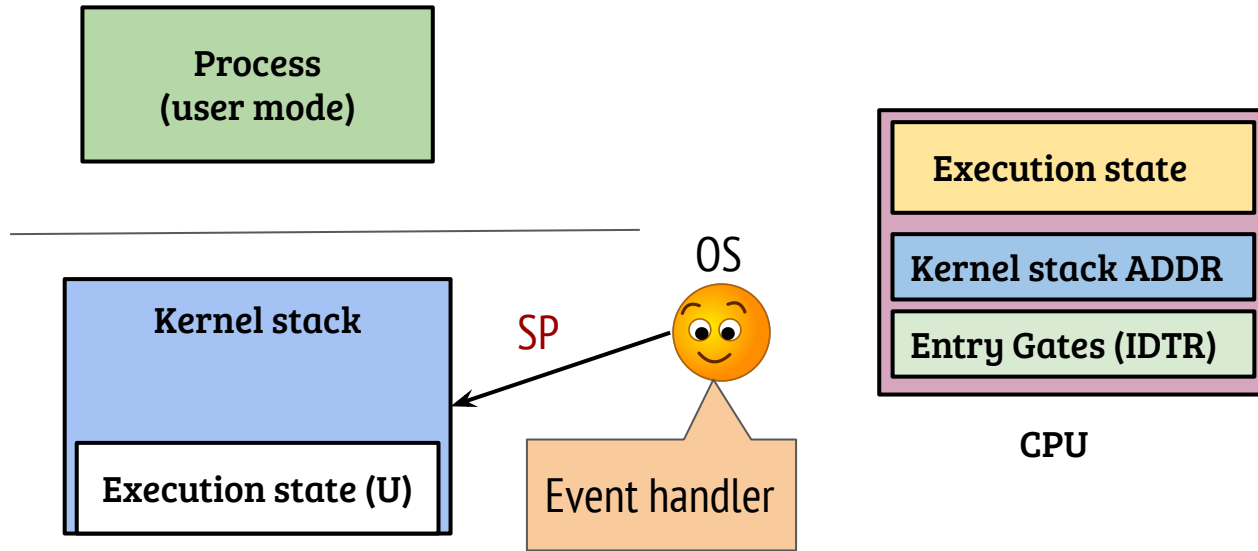
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



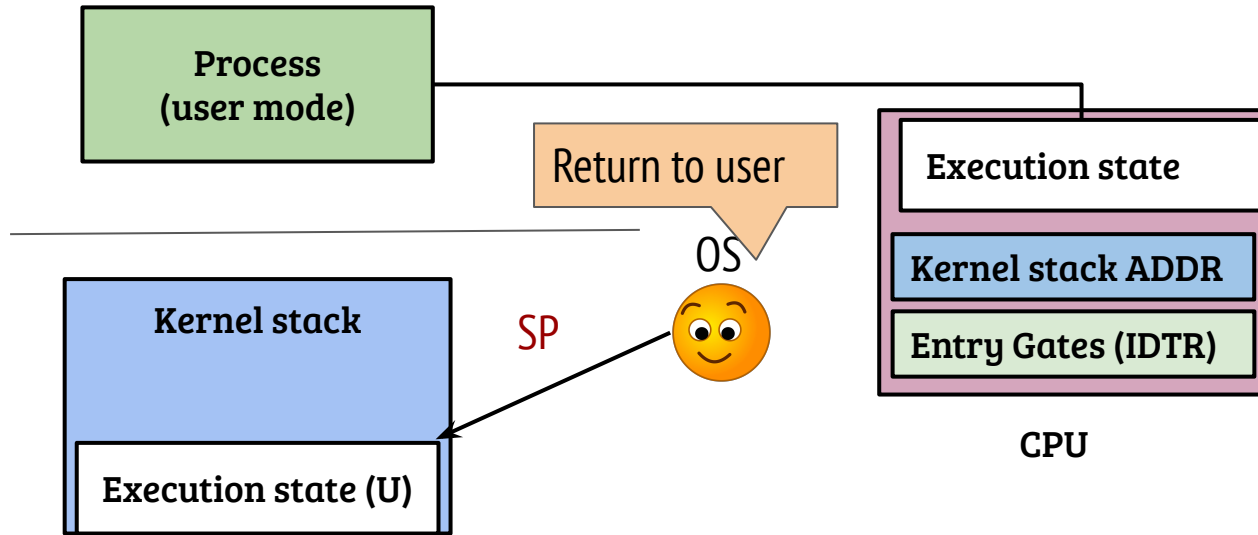
- The CPU saves the execution state onto the kernel stack
- The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)

User-kernel context switch



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

User-kernel context switch

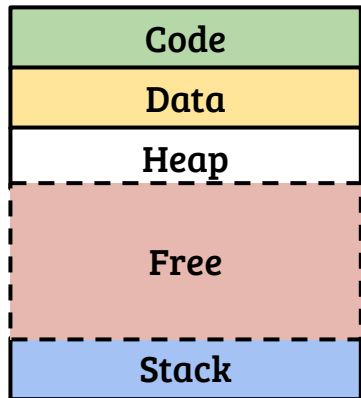


- The kernel stack pointer should point to the position at the time of entry
- CPU loads the user execution state and resumes user execution

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?

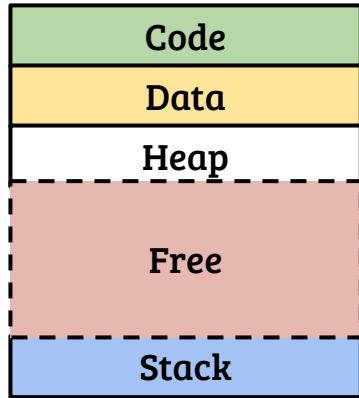
The OS address space



OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

The OS address space



OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

- 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