CS614: Linux Kernel Programming

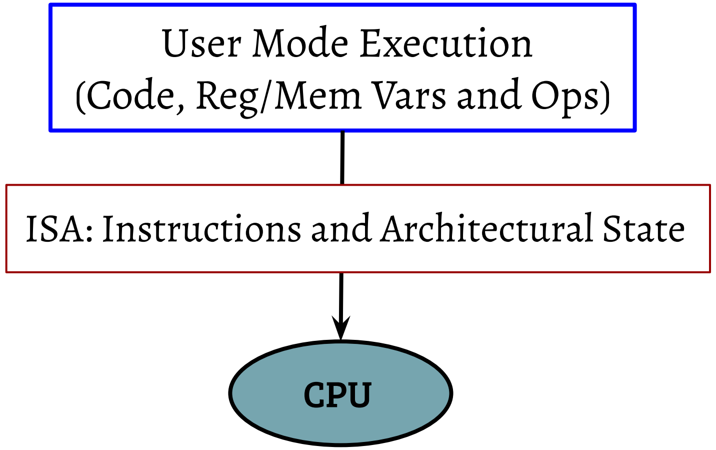
Privileges and Execution Contexts

Debadatta Mishra, CSE, IIT Kanpur

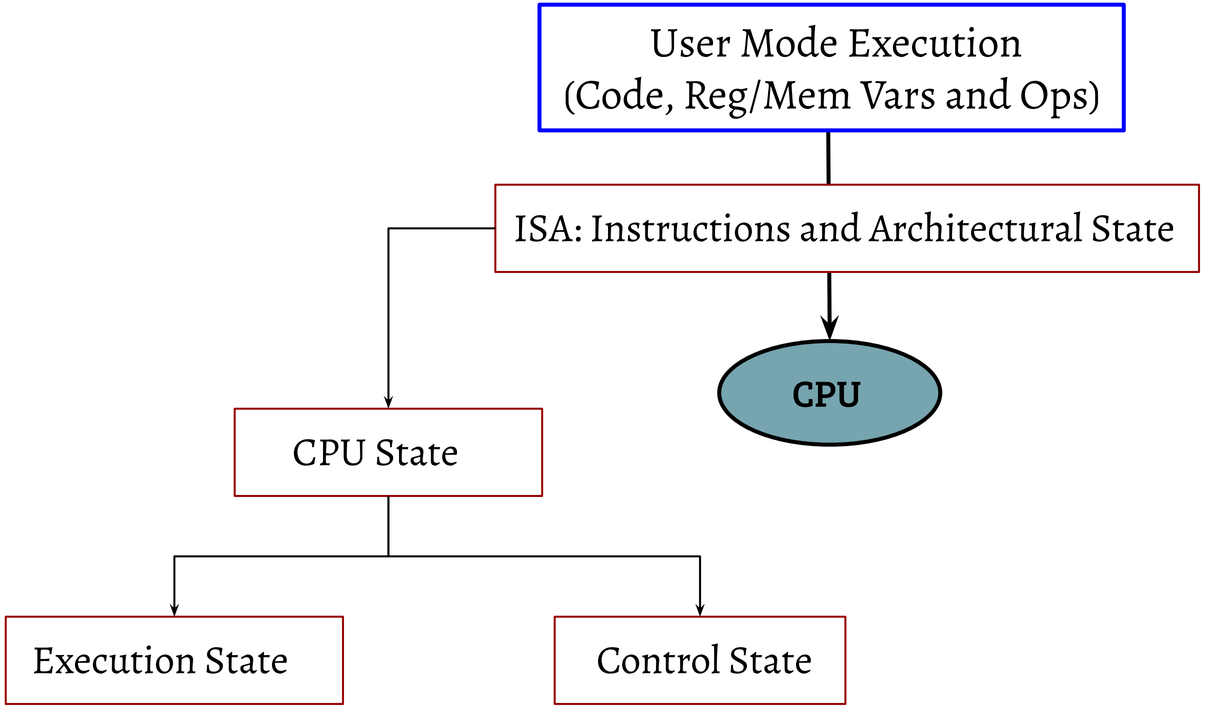
Recap: Limited direct execution

- 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

User Mode Execution: ISA and hardware resources


User Mode Execution
(Code, Reg/Mem Vars and Ops)
ISA: Instructions and Architectural State
CPU

User Mode Execution: ISA and hardware resources

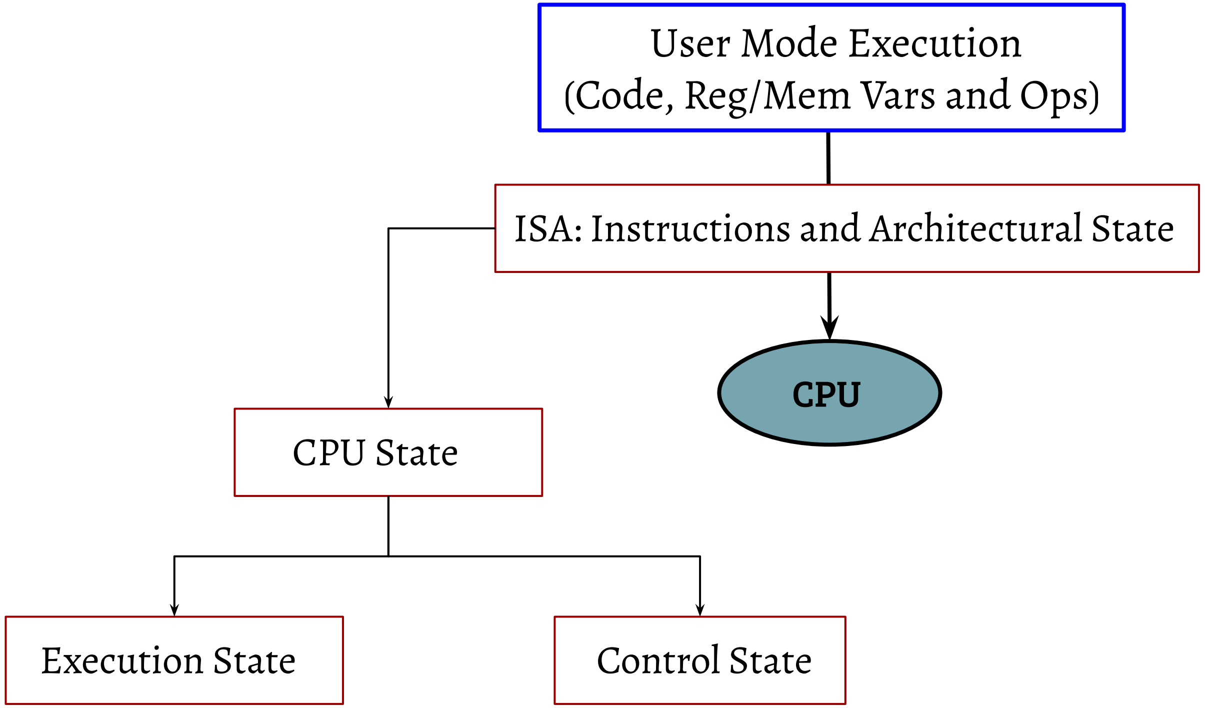

User Mode Execution
(Code, Reg/Mem Vars and Ops)
ISA: Instructions and Architectural State
CPU
CPU State
Execution State
Control State

General purpose registers and special registers (IP, SP etc.)

Control registers dictating the CPU behavior (e.g., CRs in X86)

- What is the OS role to ensure correct user mode execution?
- What about memory state? Is the stack a memory state or a register state?

User Mode Execution: ISA and hardware resources

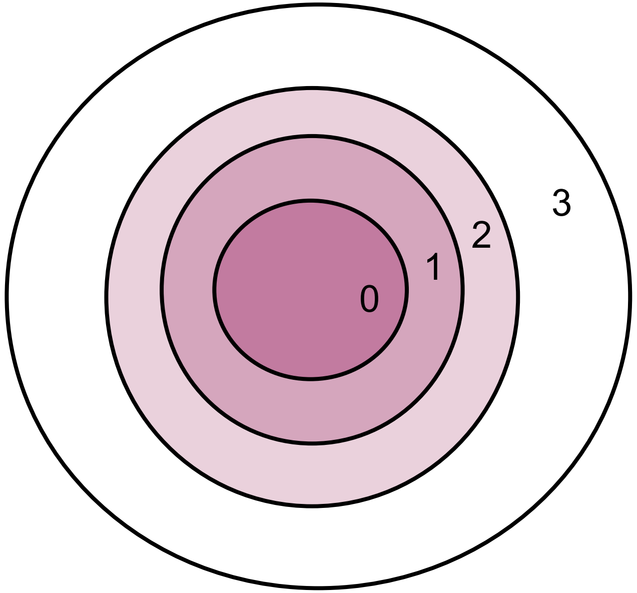

User Mode Execution
(Code, Reg/Mem Vars and Ops)
ISA: Instructions and Architectural State
CPU
CPU State
Execution State
Control State

General purpose registers and special registers (IP, SP etc.)

Control registers dictating the CPU behavior (e.g., CRs in X86)

- What is the OS role to ensure correct user mode execution? OS intervention should not mess-up the state
- What about memory state? Is the stack a memory state or a register state? Stack is maintained in memory, accessed using SP

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;”);
}
```

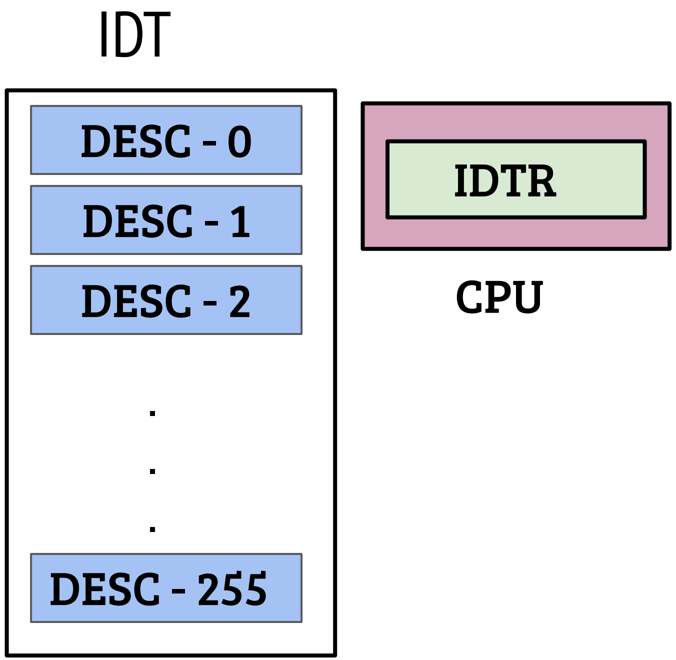
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

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);
}
```

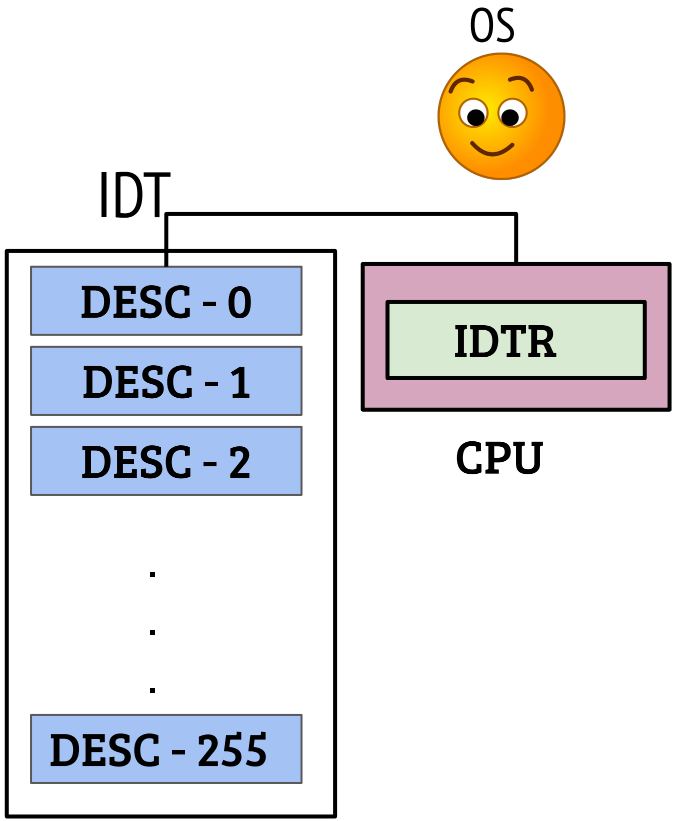
- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

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 INT instruction (Conventional Method)

- 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?

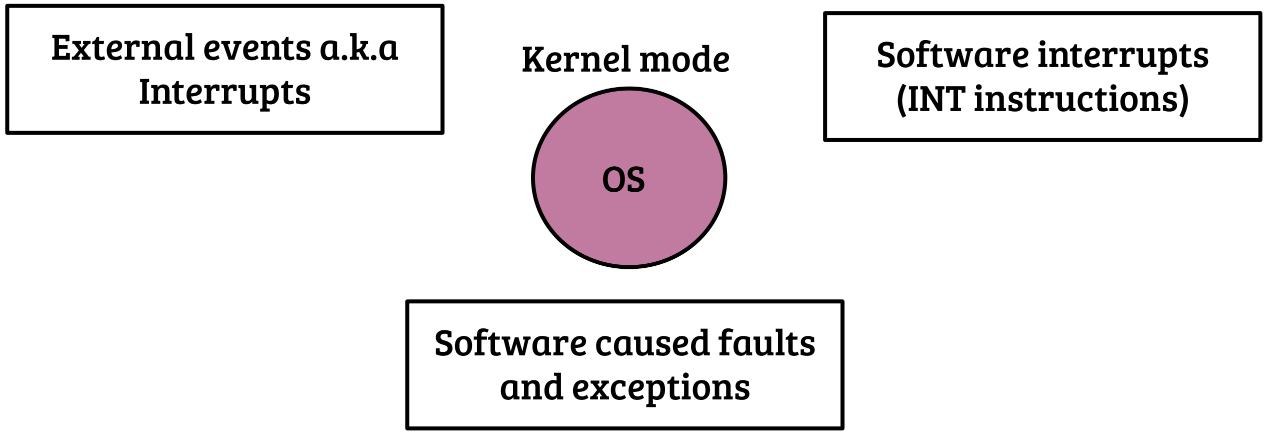
System call INT instruction (Conventional Method)

- 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 Kernel (using syscall inst.)

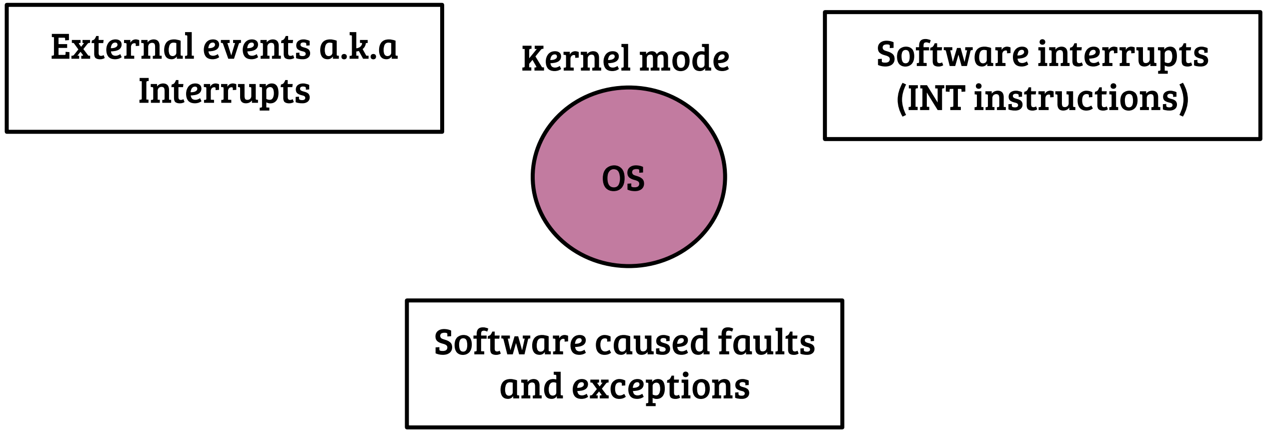
- X86 provides a fast system call method through the "syscall" instruction
- OS configures designated privileged registers with the entry address (and other information related to privilege change)
- The hardware saves the next instruction address (user return address) into RCX, change privilege levels and sets RIP to the syscall entry address. (SP and CR3 are not modified)
- Arguments and return value
    - RAX: System call # and return value
    - Arguments passed: RDI, RSI, RDX, R10, R8, R9

Post-boot OS execution


External events a.k.a Interrupts
Kernel mode
OS
Software interrupts (INT instructions)
Software caused faults and exceptions

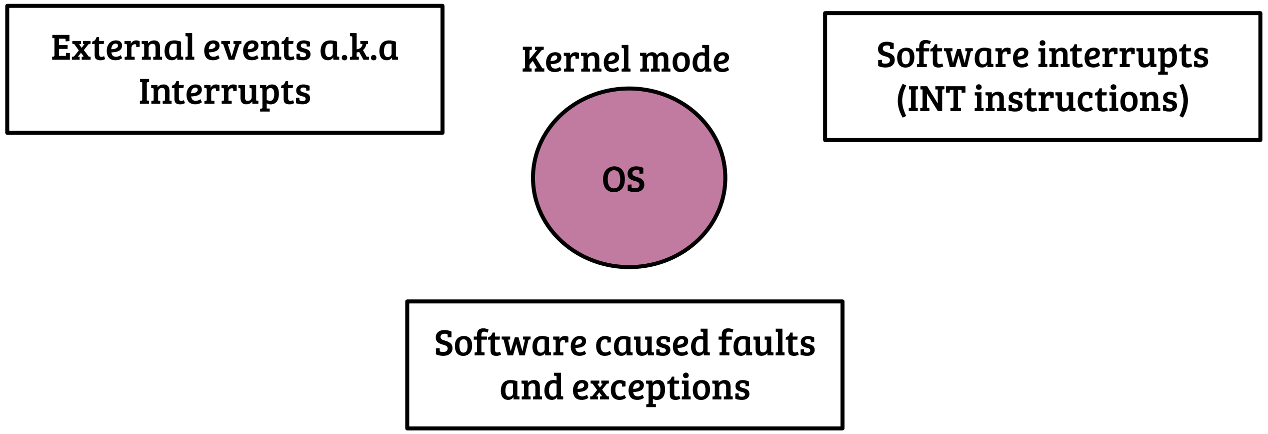
- OS execution is triggered because of interrupts, exceptions or system calls

Post-boot OS execution


External events a.k.a Interrupts
Kernel mode
OS
Software interrupts (INT instructions)
Software caused faults and exceptions

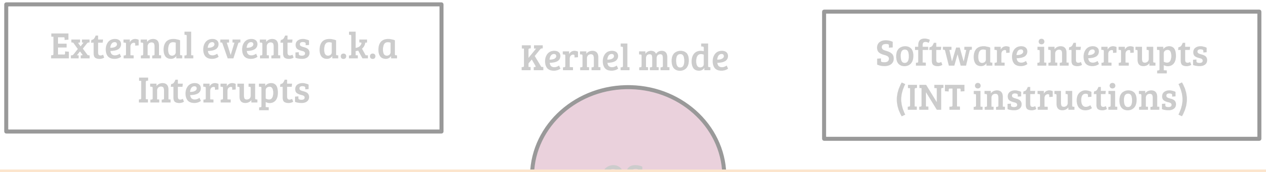
- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?

Post-boot OS execution


External events a.k.a Interrupts
Kernel mode
OS
Software interrupts (INT instructions)
Software caused faults and exceptions

- 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

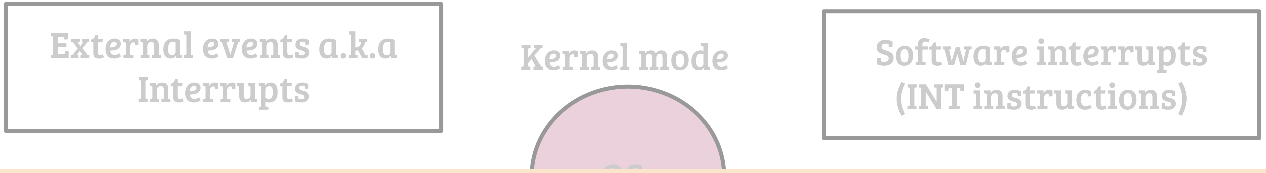
The OS stack

- 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 (or OS in case of “syscall”) switches the stack pointer to the stack address configured by the OS

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?
- 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 resume. The OS need 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

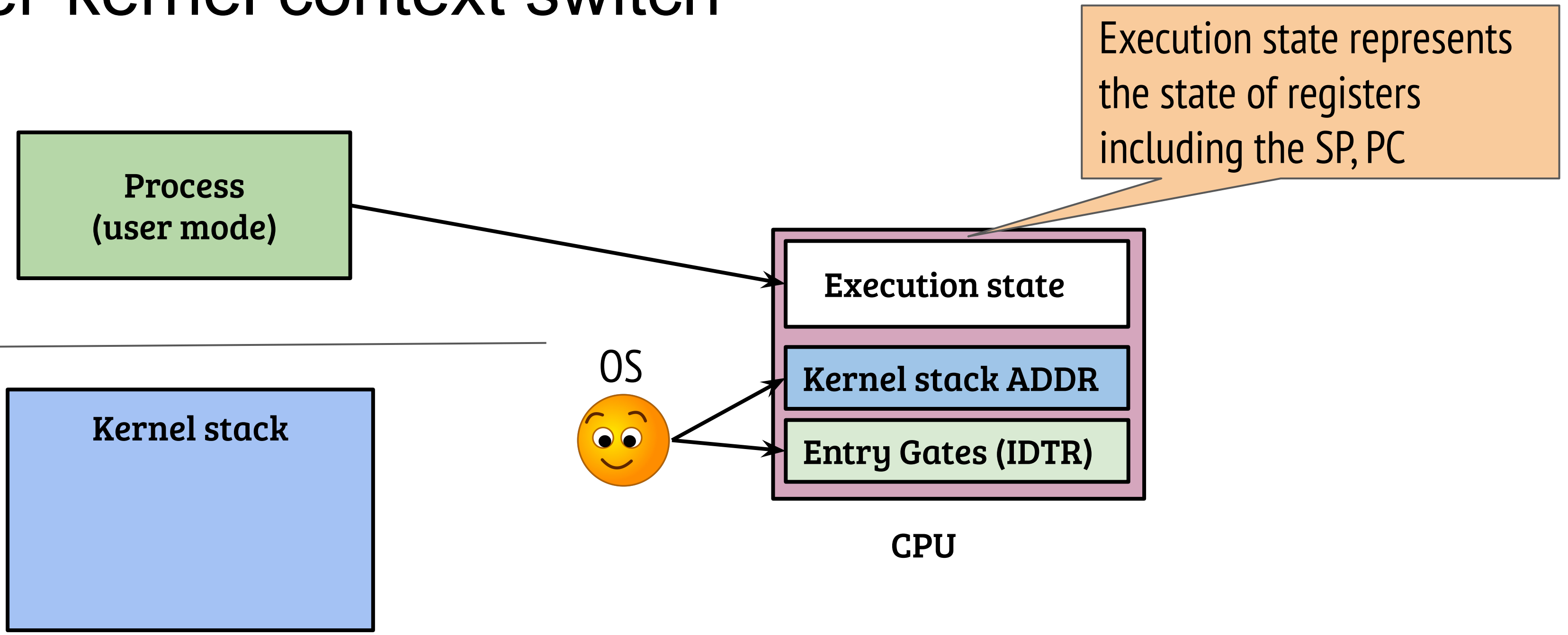
Kernel mode

Software interrupts

- 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?

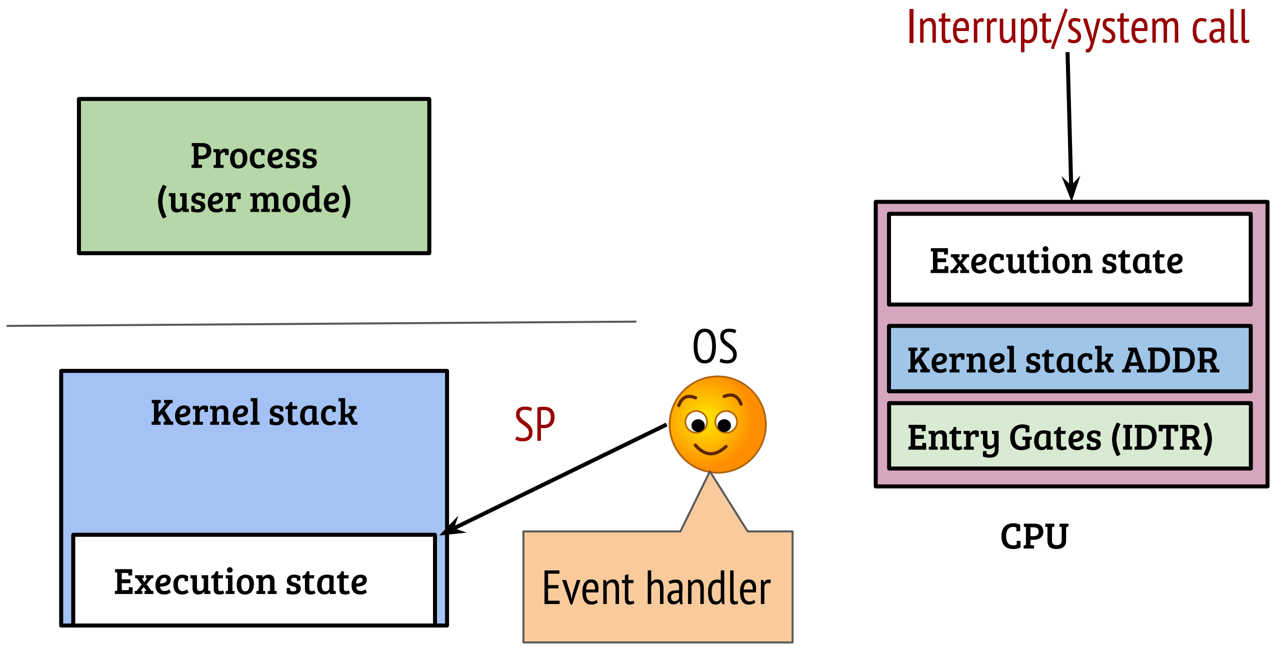
The interrupted program may become corrupted after resume? The OS need to save the user execution state and restore it on return

User-kernel context switch



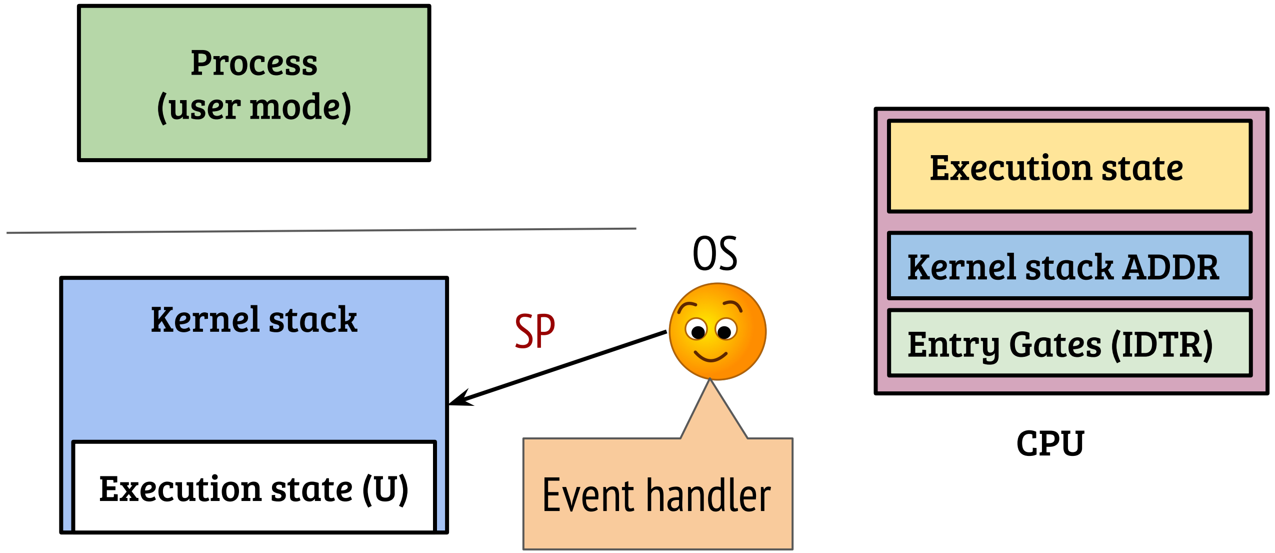
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



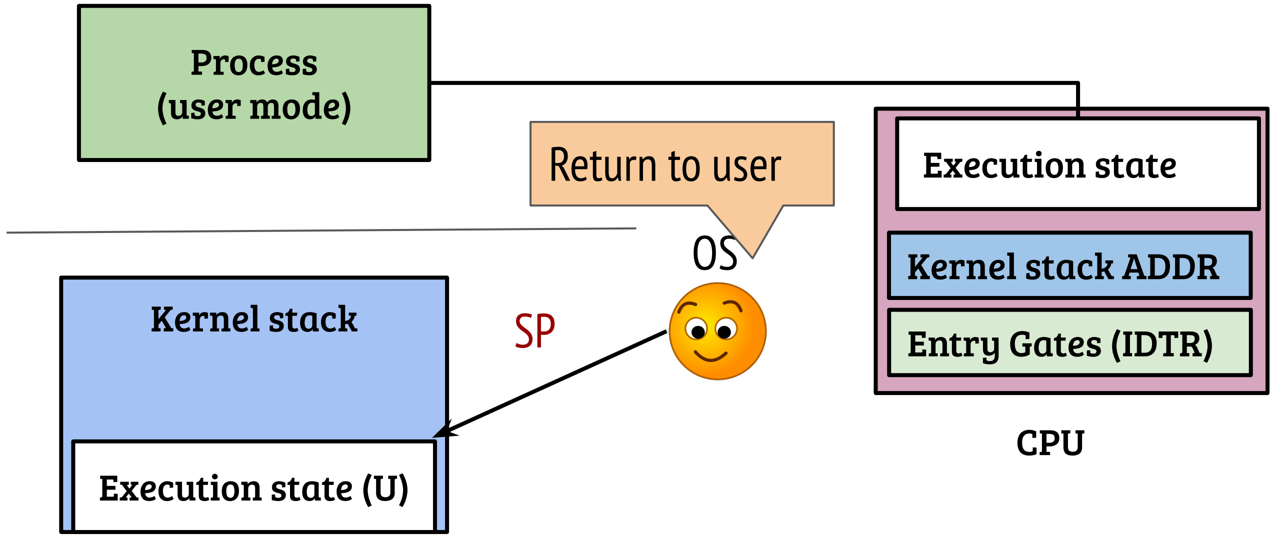
- The CPU saves the execution state onto the kernel stack
- The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)

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



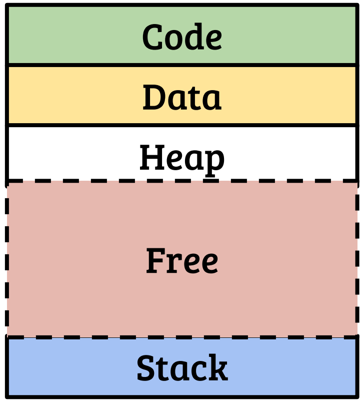
- The kernel stack pointer should point to the position at the time of entry
- CPU loads the user execution state and resumes user execution

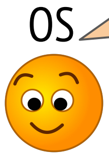
Post-boot OS execution

- 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?

to save the user execution state and restore it on return

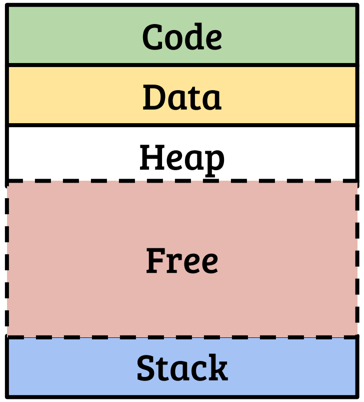
The OS address space

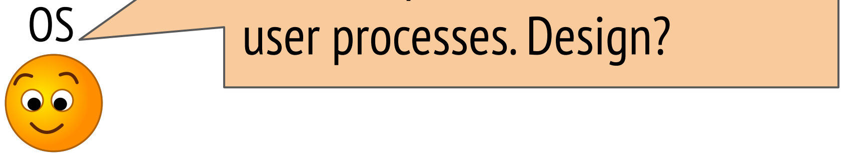




Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

The OS address space





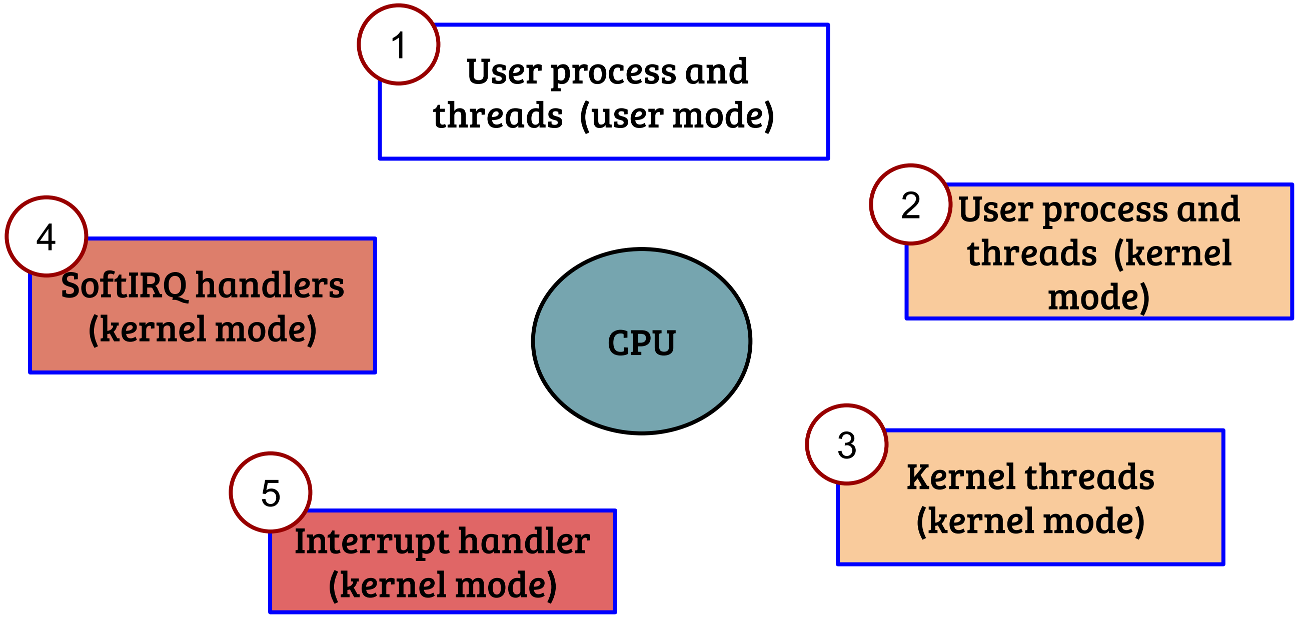
Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

- 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

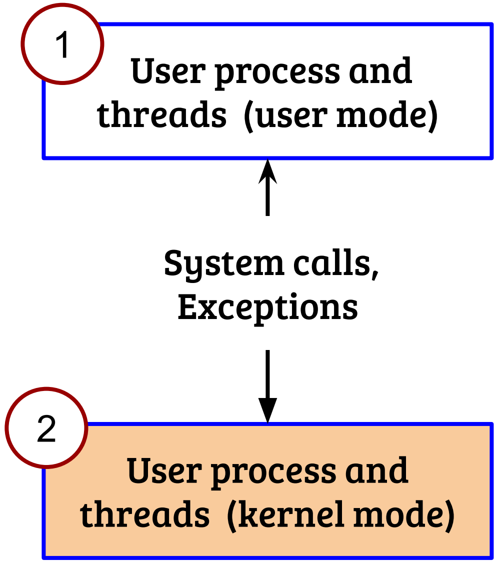
- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution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

Execution contexts in Linux


1
User process and threads (user mode)
2
User process and threads (kernel mode)
3
Kernel threads (kernel mode)
4
SoftIRQ handlers (kernel mode)
5
Interrupt handler (kernel mode)
CPU

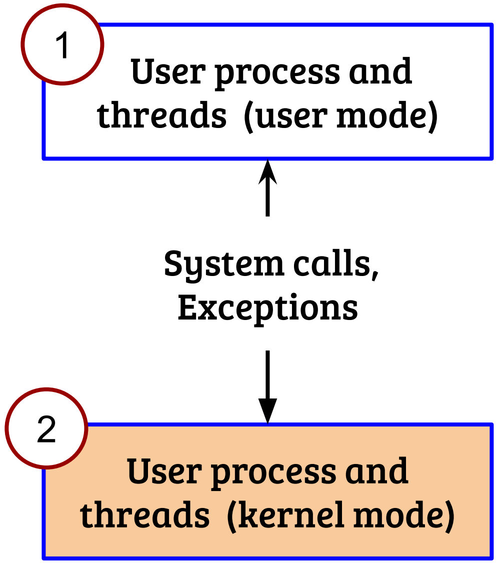
- In a linux system, the CPU can be executing in one of the above contexts
- For (3), (4) and (5), the context is not associated with any user process

User contexts


1
User process and threads (user mode)
System calls, Exceptions
2
User process and threads (kernel mode)

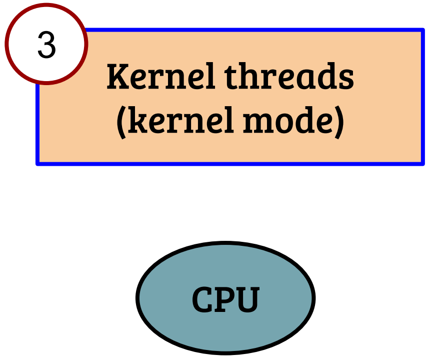
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can a process sleep { in (1) and (2) }?
- Can a process in user mode preempted?
- Can a process in kernel mode preempted?

User contexts


1
User process and threads (user mode)
System calls, Exceptions
2
User process and threads (kernel mode)

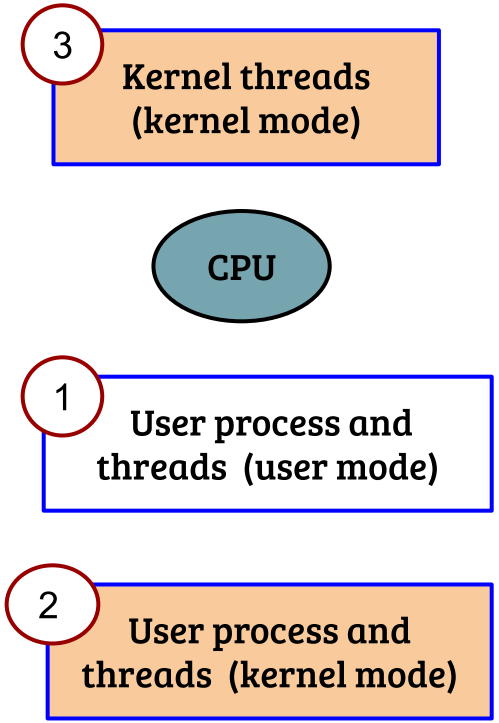
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change, CR3 changes if PTI enabled
- Can a process sleep { in (1) and (2) }?
- Yes, it can (lock holding conditions apply for 2)
- Can a process in user mode be preempted?
- Yes
- Can a process in kernel mode be preempted?
- Yes (if not explicitly disabled)

Kernel threads



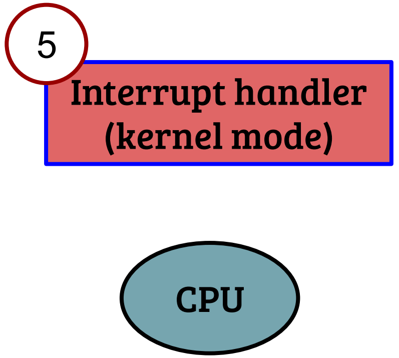
- Kernel threads are independent of user processes and threads
- Created in kernel using *kthread_create*
- How is a kernel thread different?
- Can it sleep?
- Can it be be preempted?
- Which contexts can preempt a kernel thread?

Kernel threads


3
Kernel threads (kernel mode)
CPU
1
User process and threads (user mode)
2
User process and threads (kernel mode)

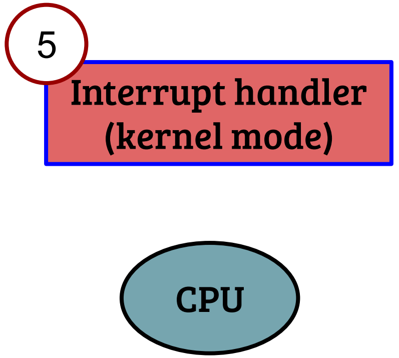
- How is a kernel thread different?
- Kernel thread never executes in user mode
- Does not require a MM context of its own
- Can it sleep?
- Yes, it can (lock holding conditions apply)
- Can it be be preempted?
- Yes (if not explicitly disabled)
- Which contexts can preempt a kernel thread?
- User, Interrupt and SoftIRQ

Hardware interrupts (Background)



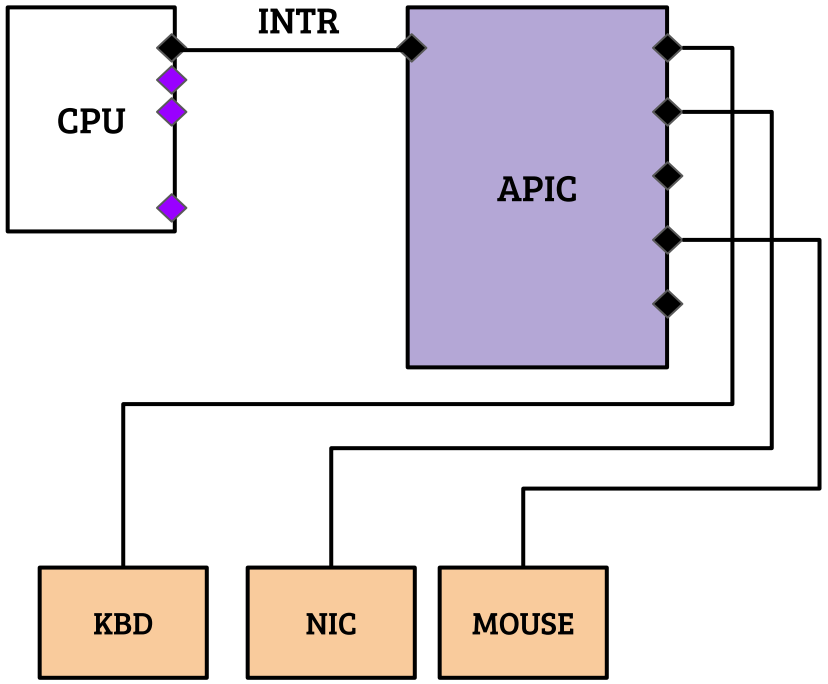
- Why interrupts?
- Example: Receive a packet from network
- What are the architectural support?

Hardware interrupts (Background)



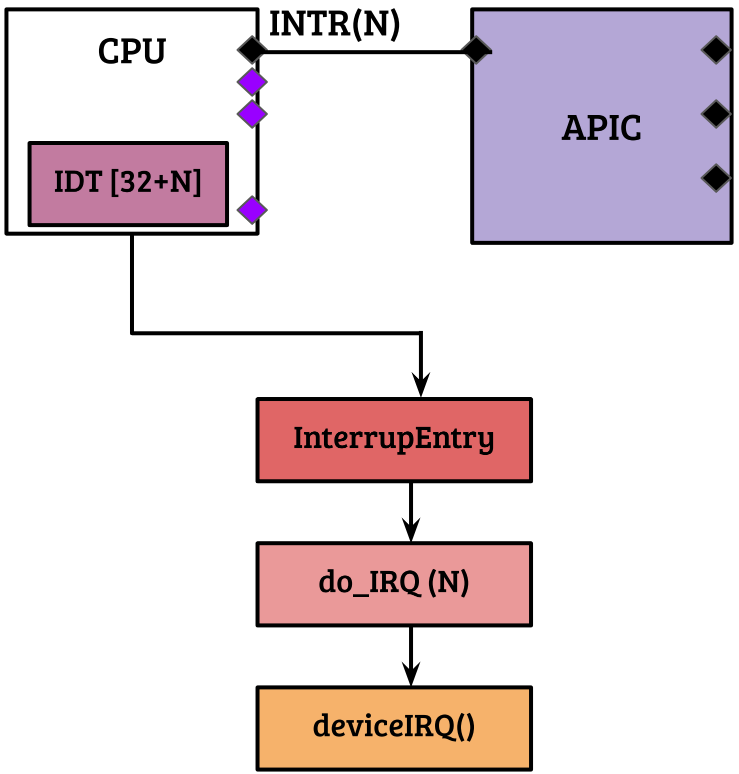
- Why interrupts?
- Example: Receive a packet from network
- Avoid CPU wastage due to polling
- Responsive and scalable systems
- What are the architectural support?
- CPU has limited #of interrupt PINs $\longrightarrow$ How to multiplex many devices?

Interrupt architecture - PIC and APIC



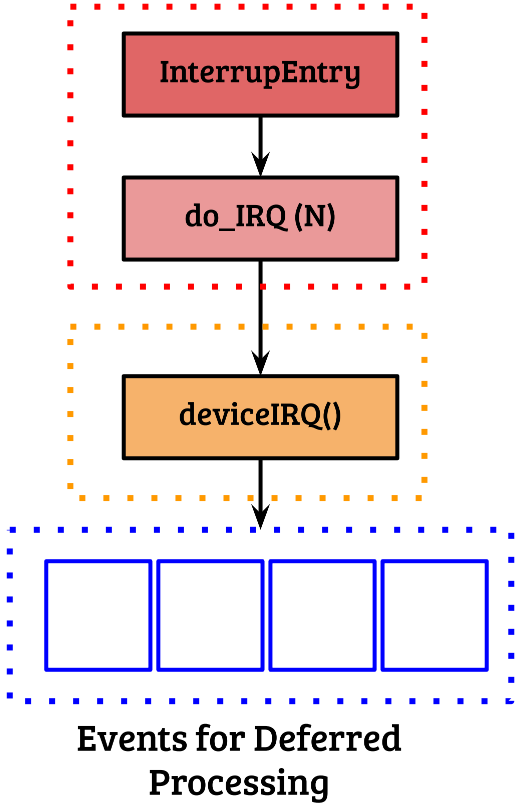
- Every device attached to the APIC is configured with a unique IRQ number
- APIC saves the IRQ in a control port register and raise CPU interrupt line on receipt of device interrupt
- CPU reads the IRQ number and invokes the interrupt handler
- Waits for acknowledgement before clearing the INTR line
- Selective disabling of IRQs possible
    - != cli (CPU interrupt disable)
    - New interrupts not lost

Interrupt handling



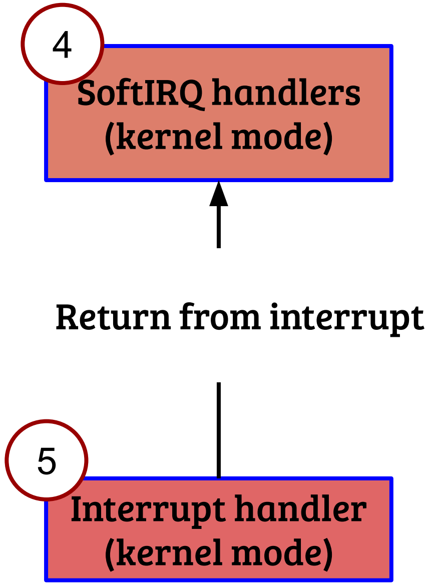
- IDT configured to load the interrupt execution context (CPL and stack)
- Interrupt entry: save regs, switch CR3 if needed
- do_IRQ checks the descriptor flags and invokes the real handler
- The device driver handler implements the device specific functionalities
- When is the interrupt acknowledged (i.e., INTR is cleared)?
- How long is the device interrupt masked?
- Not all interrupts can be handled quickly, e.g., NIC RCV

Interrupt handling in three stages



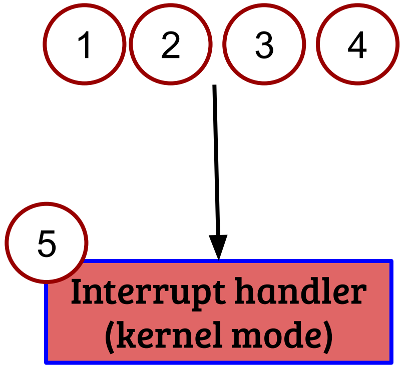
- Critical tasks: Interrupt context setup, APIC acknowledgement
- Semicritical: Accessing/updating device state, e.g., update receive queue pointers of a NIC
- Deferrable: Actions that are device independent e.g., Network stack processing

Interrupt handling: SoftIRQ


4
SoftIRQ handlers (kernel mode)
Return from interrupt
5
Interrupt handler (kernel mode)

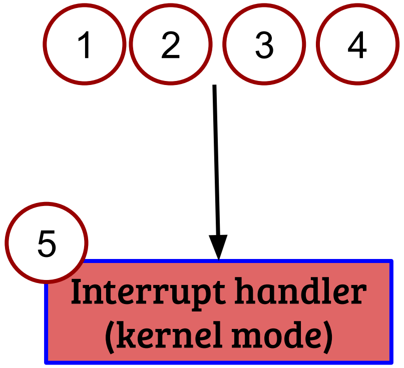
- Carry out deferrable operations, can be preempted by interrupts
- Like an interrupt, it can be raised, disabled, enabled, masked
- Executed by the local CPU kernel thread (*ksoftirqd*, one per CPU)
    - Infinite loop checking for pending softIRQ (set when softirq is raised)
    - Often scheduled on irq_exit( ) or explicit wakeup

Interrupt context



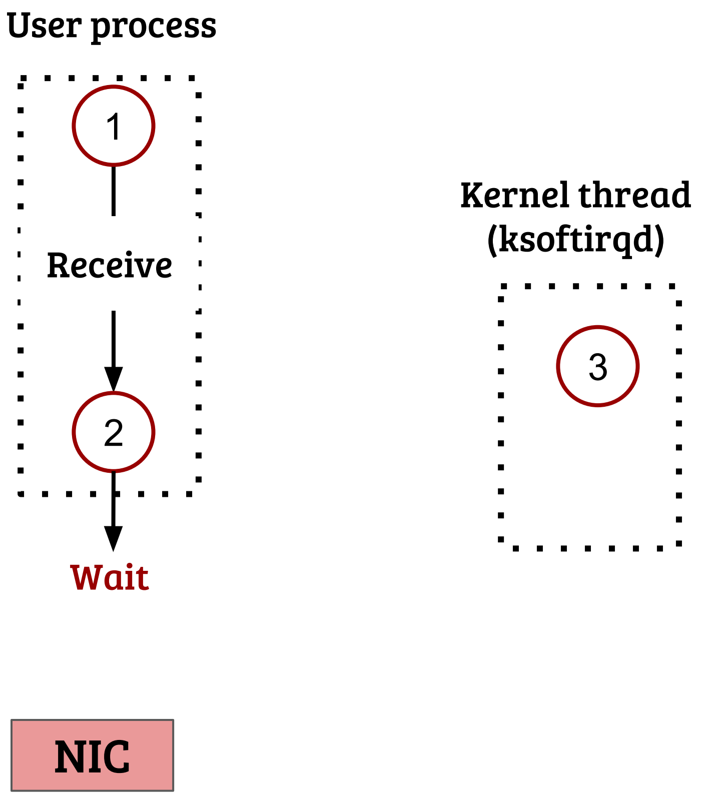
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can an interrupt handler sleep?
- Can it be preempted?

Interrupt context



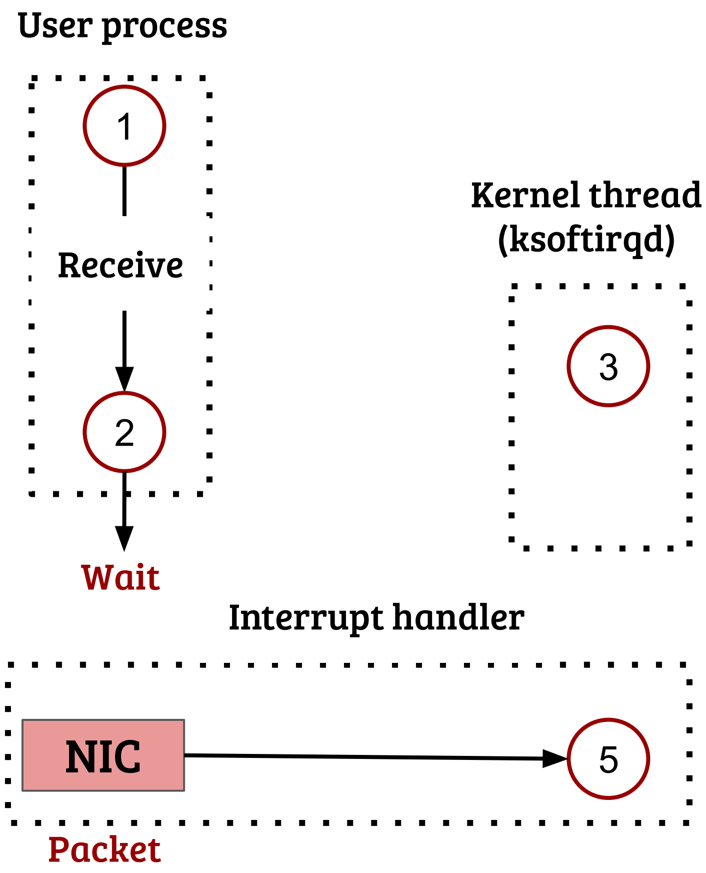
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change (interrupt stack used), CR3 changes if entering from user mode in a PTI enabled system
- Can an interrupt handler sleep?
- No, Linux does not allow sleeping (directly/indirectly) in an interrupt handler
- Can it be preempted?
- Only by another interrupt (if APIC Acked and interrupts enabled on CPU)

Contexts in action: network receive



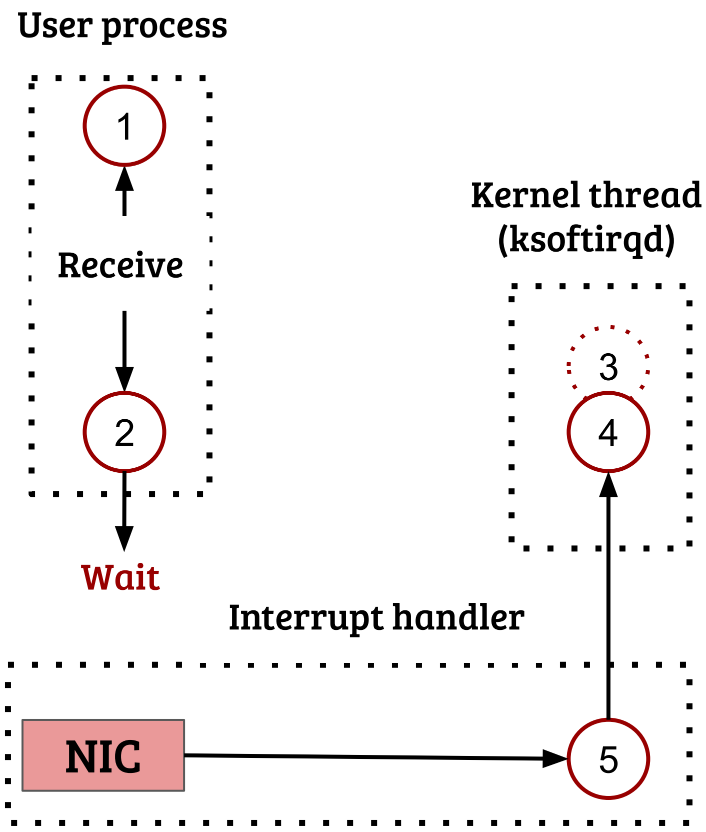
- The user process invokes recv( ) system call (blocking)
- No processed payload found, the process is descheduled and put into a wait queue
- Ksoftirqd is either suspended or processing other pending softIRQs

Contexts in action: network receive



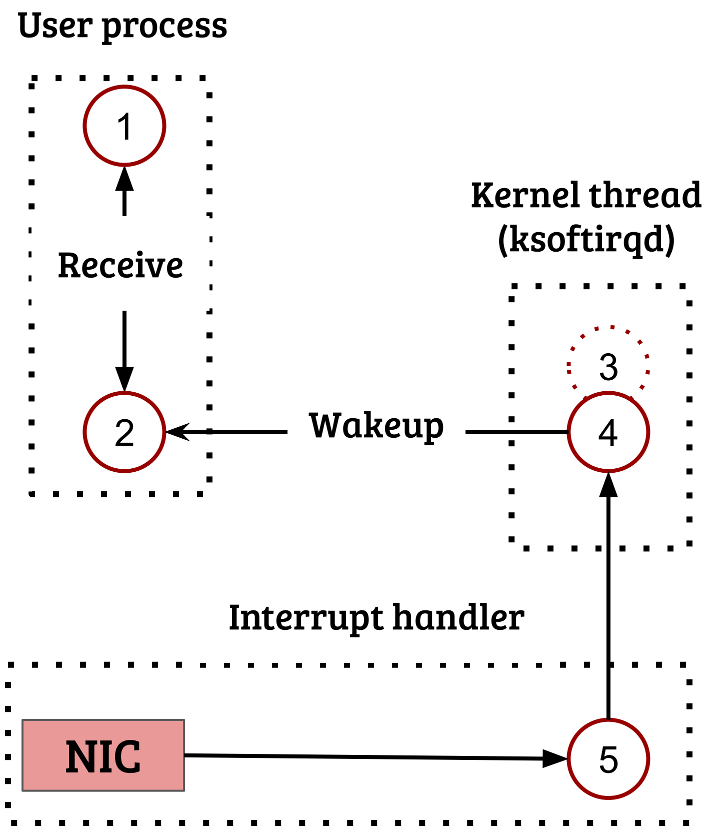
- The NIC copies the packet (using DMA) into memory buffers (a.k.a. skbuffs) and triggers the interrupt
- Before the device specific interrupt handling, APIC is acknowledged
- The device interrupt handler update the device state while masking device interrupts
- Queues the packet for further processing and triggers a softIRQ

Contexts in action: network receive



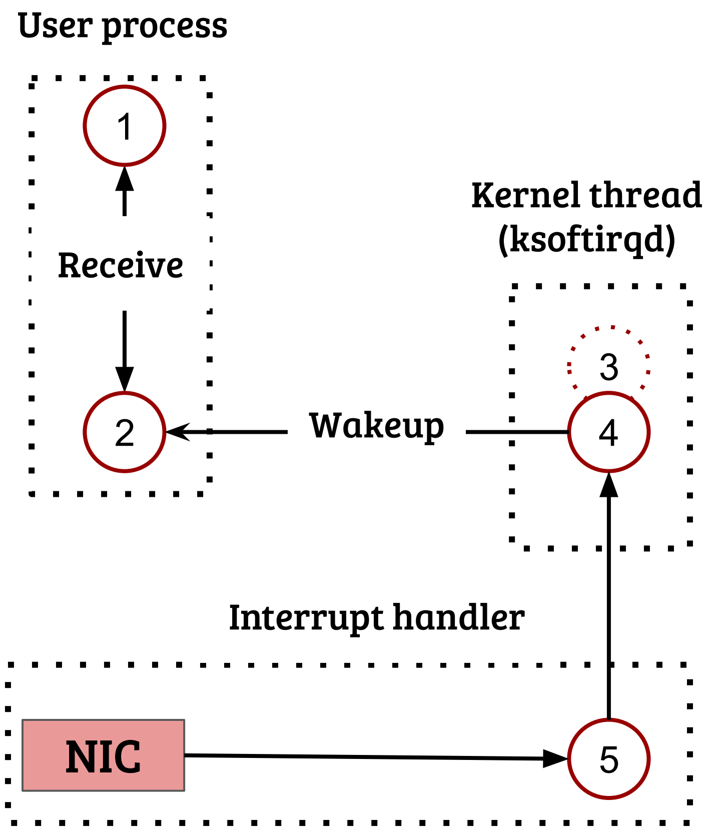
- The softIRQ is scheduled using the ksoftirqd kernel thread context
- Protocol stack processing is performed in this context
- As part of the protocol processing, the destination process is derived

Contexts in action: network receive



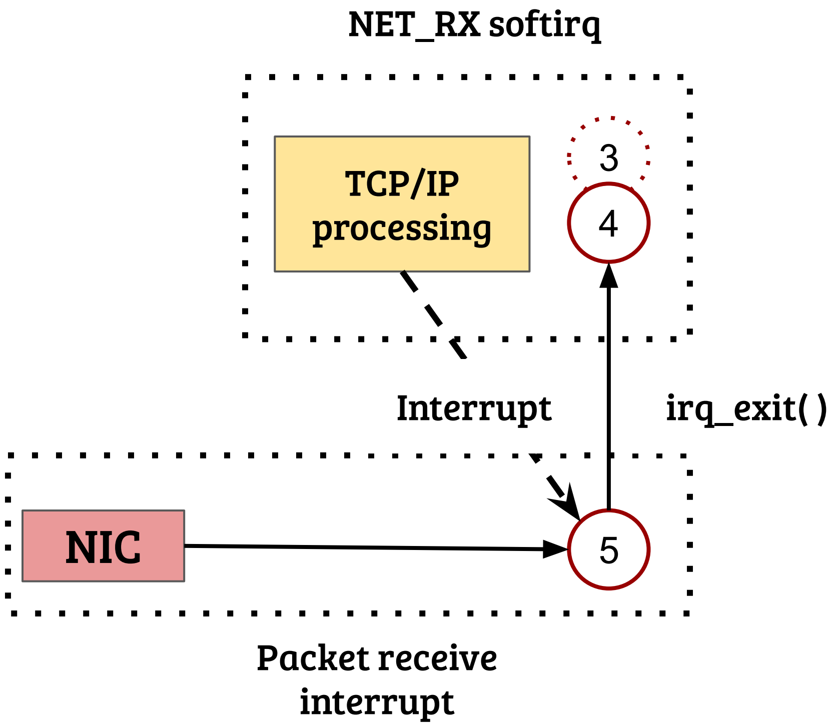
- The softIRQ processing wakes up the user process
- The user process returns from syscall (copy payload to user)
- Now, what could be the issues with this approach?

Challenges in network receive



- Minimize network packet copy across the contexts
- Precise scheduling: application progress and fairness
- Network is always overdriven and self-adjusting in nature ⟶ rate limit as early as possible
- Issues
  - Receive livelock: CPU is always handling interrupts
  - User process starvation due to softIRQ processing

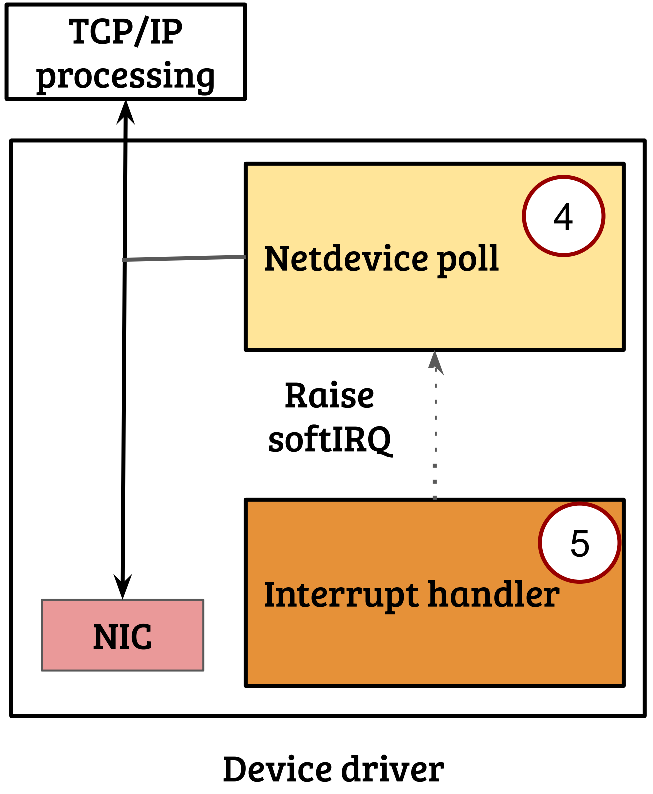
Receive livelock [1]



- Root cause: Interrupts have the highest priority over other contexts
- If the rate of interrupts is high, the system remains in interrupt handling mode, resulting in *receive livelock*
- Solution approach: Lower the priority of interrupts under heavy load
- How?

1. https://www.usenix.org/legacy/publications/library/proceedings/sd96/mogul.html

NAPI: Interrupt + Polling



- Interrupt handler raises softIRQ after disabling packet receive interrupts
- Driver registered poll method is invoked
  - Executes till receive queue is empty or an upper threshold (budget)
  - Enable the interrupt (if queue is empty) and return
- Advantages
  - Low network load, more interrupt driven
  - High load, less interrupt processing
  - Avoid wasted work, drop packets early (in the device buffer)

Context related helper routines

- bool in_irq( )
  - True if the current execution is in hardware interrupt
- bool in_softirq( )
  - True if the current execution is in a softIRQ or it is disabled
- bool in_interrupt( )
  - True if we are in NMI, IRQ, softIRQ context or have softIRQs disabled
- bool in_task( )
  - True if executing in a task context, *current* is valid
- Disabling/enabling interrupts
  - local_irq_disable/enable( )
- Disabling/enabling softIRQs
  - local_bh_disable/enable( )