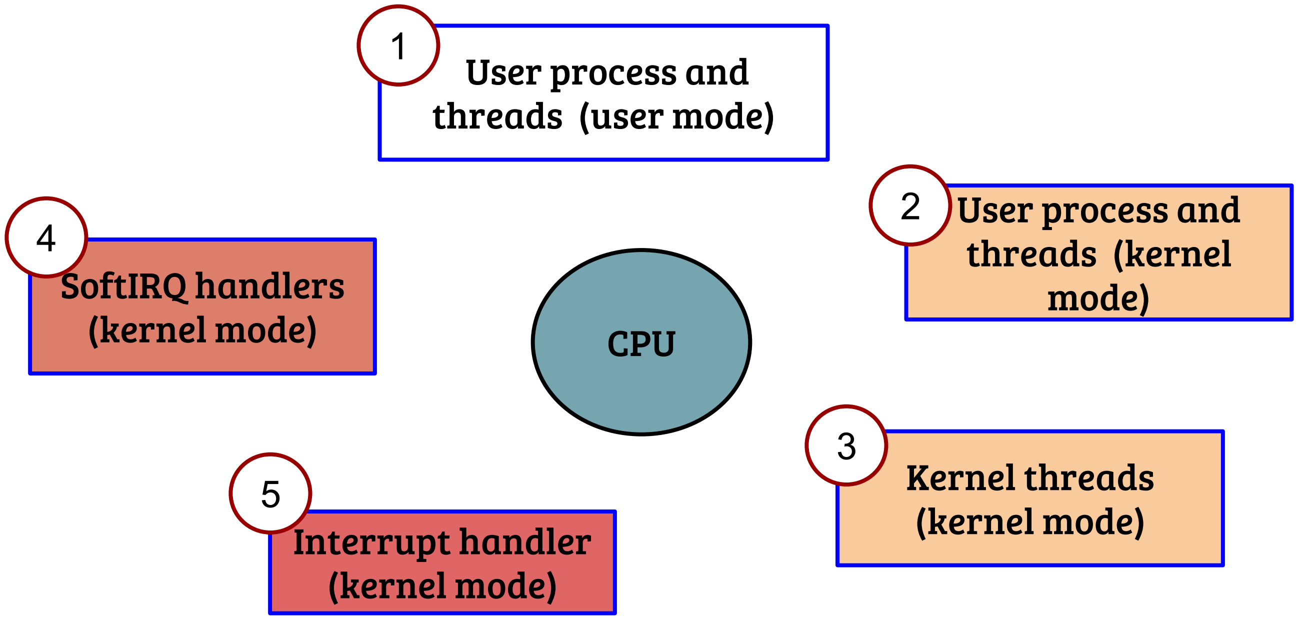
# Topics in Operating Systems

## Execution Contexts

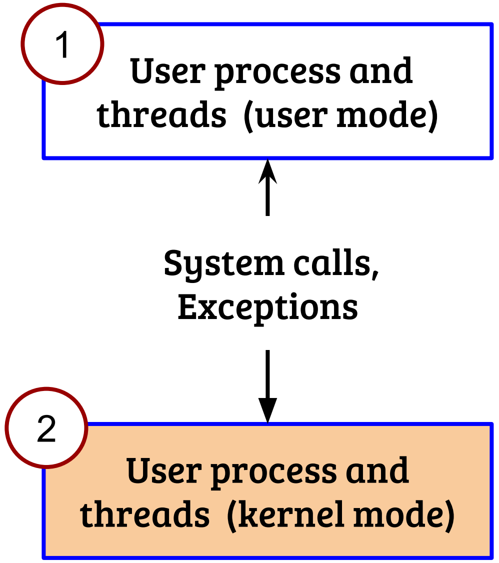
Debadatta Mishra, CSE, IITK

## Execution contexts in Linux


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

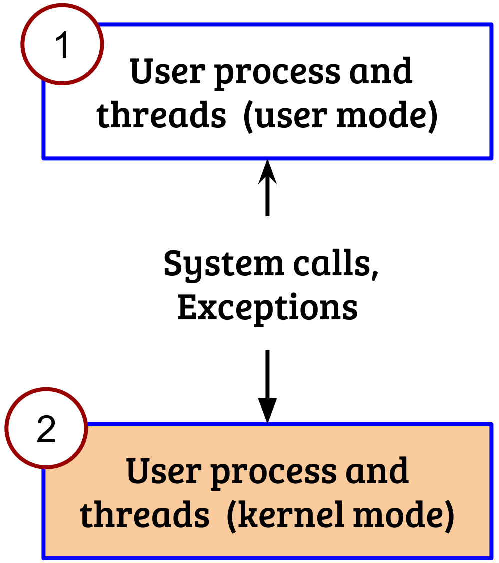
- In a linux system, the CPU can be executing in one of the above contexts
- For (3), (4) and (5), the context is not associated with any user process

## User contexts


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

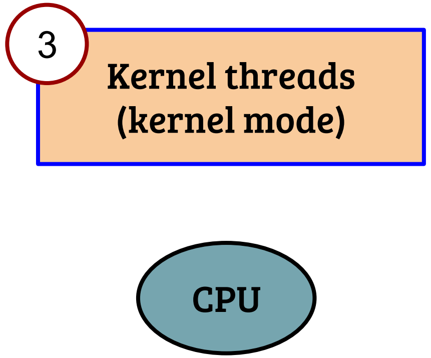
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can a process sleep { in (1) and (2) }?
- Can a process in user mode preempted?
- Can a process in kernel mode preempted?

# User contexts


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

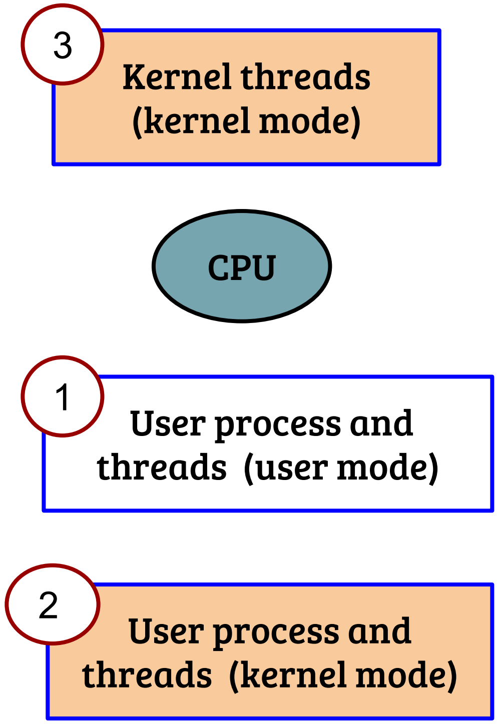
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change, CR3 changes if PTI enabled
- Can a process sleep { in (1) and (2) }?
- Yes, it can (lock holding conditions apply for 2)
- Can a process in user mode be preempted?
- Yes
- Can a process in kernel mode be preempted?
- Yes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

# Kernel threads



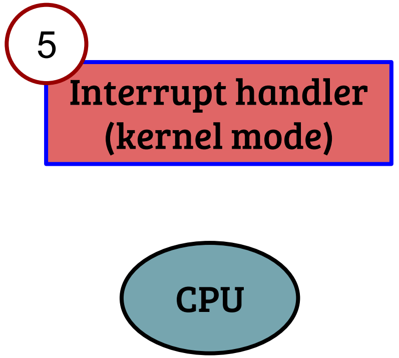
- Kernel threads are independent of user processes and threads
- Created in kernel using *kthread_create*
- How is a kernel thread different?
- Can it sleep?
- Can it be be preempted?
- Which contexts can preempt a kernel thread?

# Kernel threads


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

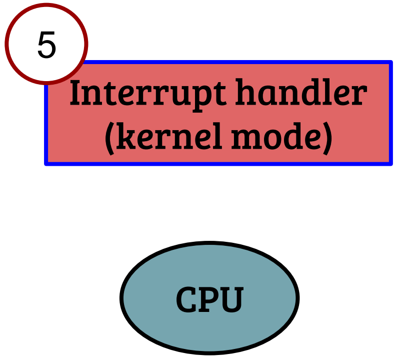
- How is a kernel thread different?
- Kernel thread never executes in user mode
- Does not require a MM context of its own
- Can it sleep?
- Yes, it can (lock holding conditions apply)
- Can it be be preempted?
- Yes (if not explicitly disabled)
- Which contexts can preempt a kernel thread?
- User, Interrupt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

## Hardware interrupts (Background)



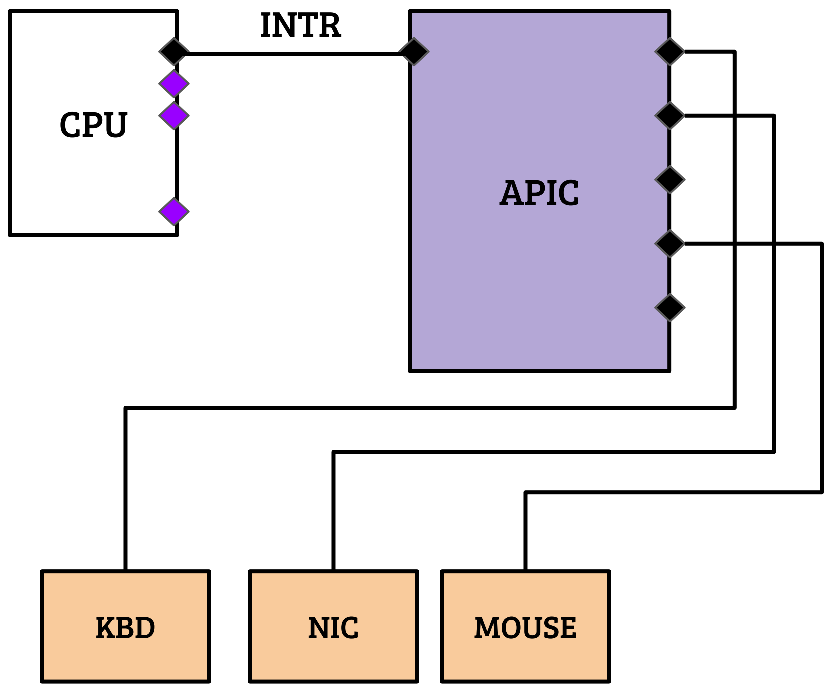
- Why interrupts?
- Example: Receive a packet from network
- What are the architectural support?

## Hardware interrupts (Background)



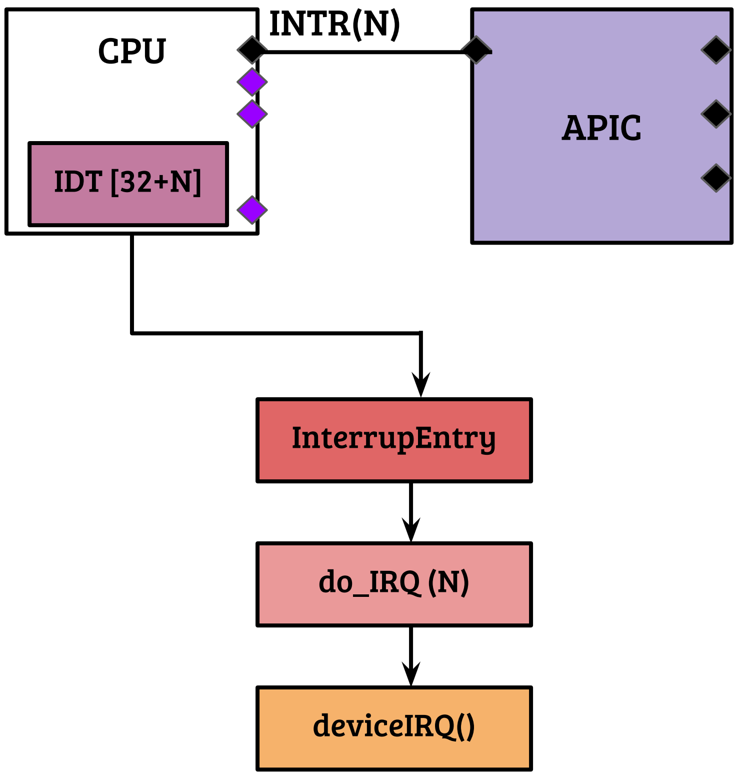
- Why interrupts?
- Example: Receive a packet from network
- Avoid CPU wastage due to polling
- Responsive and scalable systems
- What are the architectural support?
- CPU has one interrupt PIN ⟶ How to multiplex many devices?

# Interrupt architecture - PIC and APIC



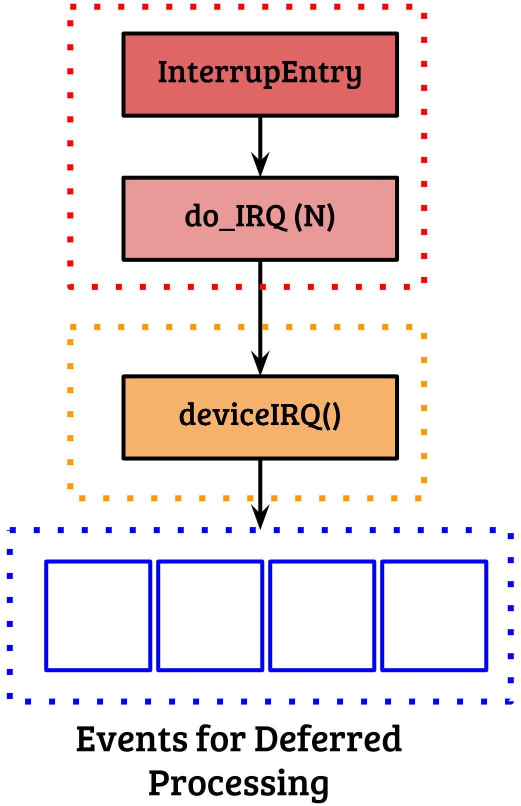
- Every device attached to the APIC is configured with a unique IRQ number
- APIC saves the IRQ in a control port register and raise CPU interrupt line on receipt of device interrupt
- CPU reads the IRQ number and invokes the interrupt handler
- Waits for acknowledgement before clearing the INTR line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

# Interrupt handling



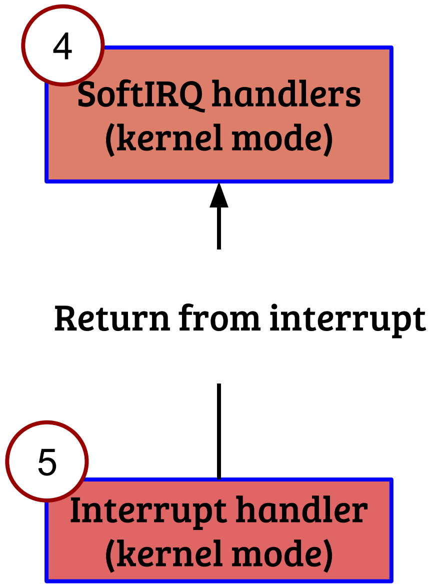
- IDT configured to load the interrupt execution context (CPL and stack)
- Interrupt entry: save regs, switch CR3 if needed
- do_IRQ checks the descriptor flags and invokes the real handler
- The device driver handler implements the device specific functionalities
- When is the interrupt acknowledged (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



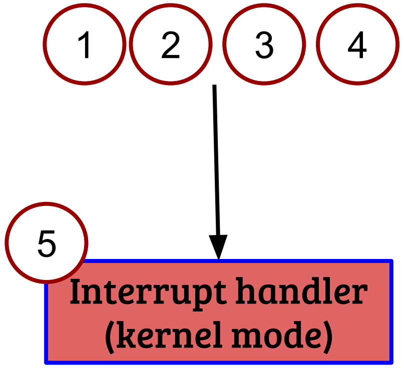
- Critical tasks: Interrupt context setup, APIC acknowledgement
- Semicritical: Accessing/updating device state, e.g., update receive queue pointers of a NIC
- Deferrable: Actions that are device independent e.g., Network stack processing

# Interrupt handling: SoftIRQ


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

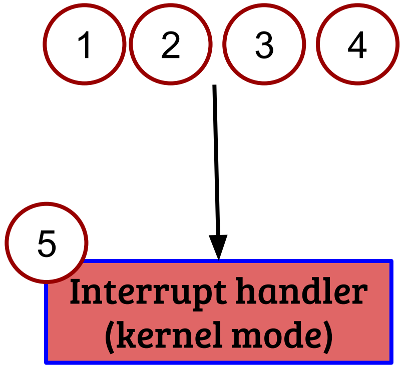
- Carry out deferrable operations, can be preempted by interrupts
- Like an interrupt, it can be raised, disabled, enabled, masked
- Executed by the local CPU kernel thread (*ksoftirqd*, one per CPU)
    - Infinite loop checking for pending softIRQ (set when softirq is raised)
    - Often scheduled on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

# Interrupt context



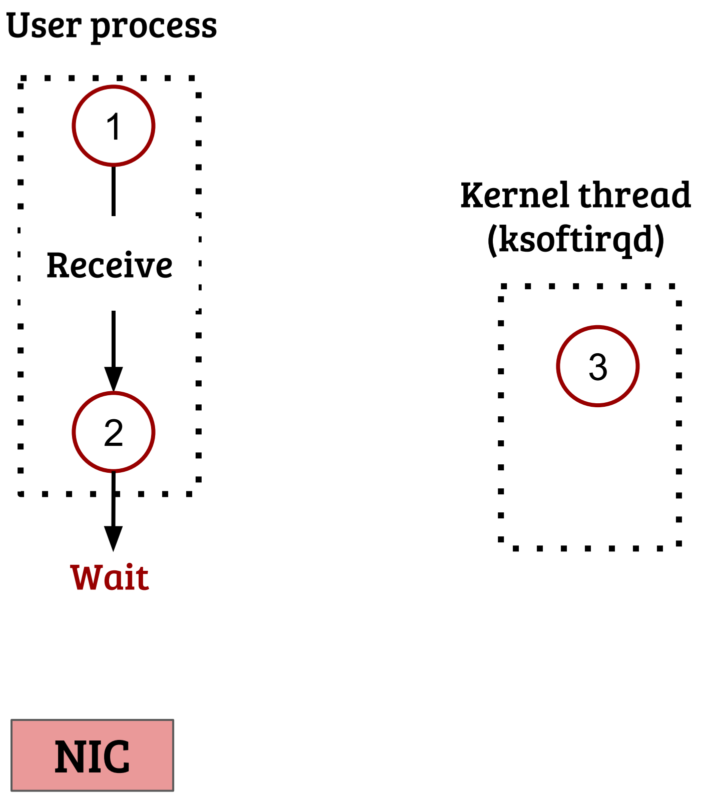
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can an interrupt handler sleep?
- Can it be preempted?

# Interrupt context



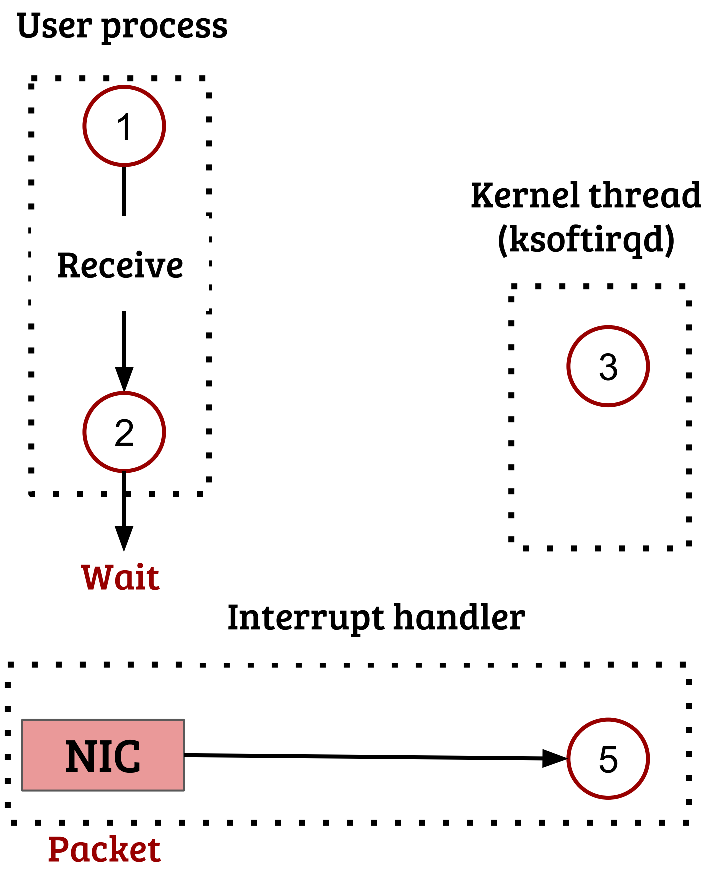
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change (interrupt stack used), CR3 changes if entering from user mode in a  PTI enabled system
- Can an interrupt handler sleep?
- No, Linux does not allow sleeping (directly/indirectly) in an interrupt handler
- Can it be preempted?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

# Contexts in action: network receive



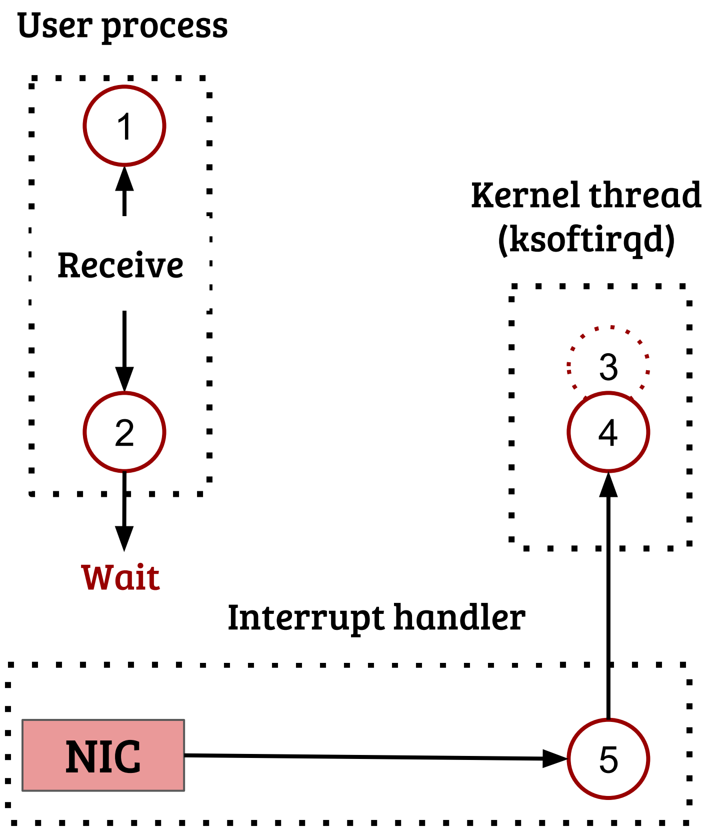
- The user process invokes recv( ) system call (blocking)
- No processed payload found, the process is descheduled and put into a wait queue
- Ksoftirqd is either suspended or processing other pending softIRQs

# Contexts in action: network receive



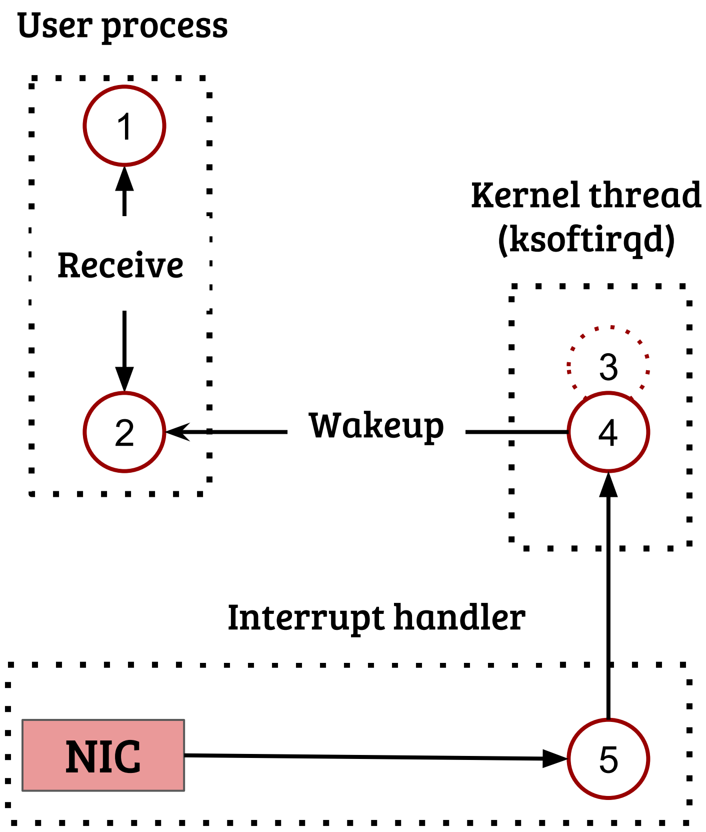
- The NIC copies the packet (using DMA) into memory buffers (a.k.a. skbuffs) and triggers the interrupt
- Before the device specific interrupt handling, APIC is acknowledged
- The device interrupt handler update the device state while masking device interrupts
- Queues the packet for further processing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

# Contexts in action: network receive



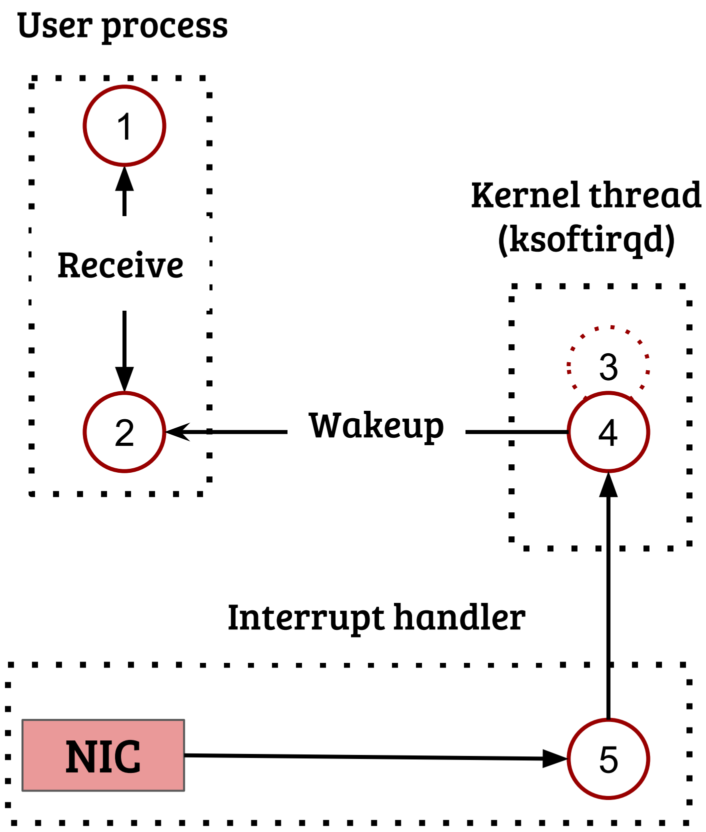
- The softIRQ is scheduled using the ksoftirqd kernel thread context
- Protocol stack processing is performed in this context
- As part of the protocol processing, the destination process is derived

# Contexts in action: network receive



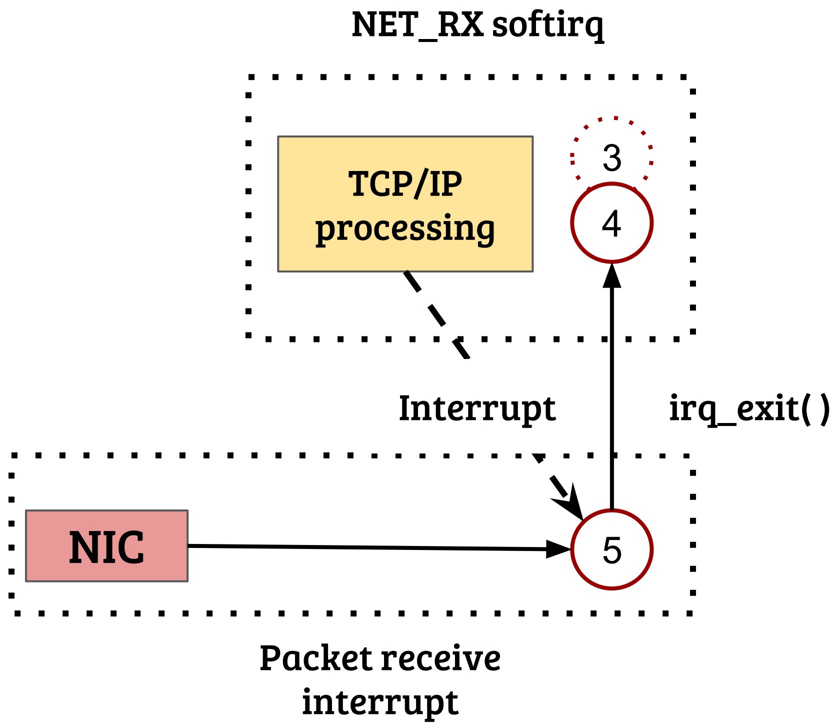
- The softIRQ processing wakes up the user process
- The user process returns from syscall (copy payload to user)
- Now, what could be the issues with this approach?

# 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

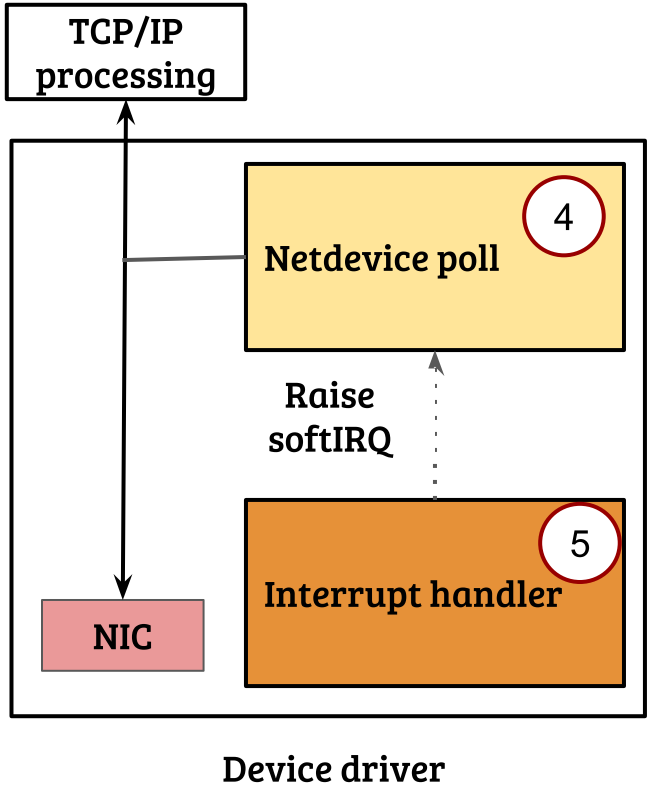
# Receive livelock [1]



- 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( )