

Interrupts

1

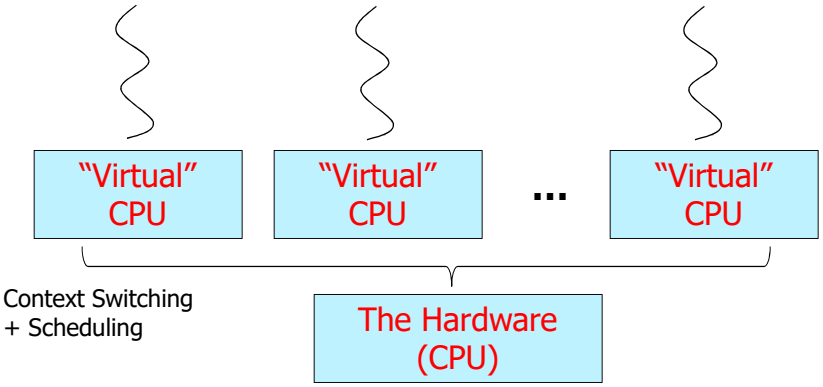


Where We Are:

The Hardware (CPU)

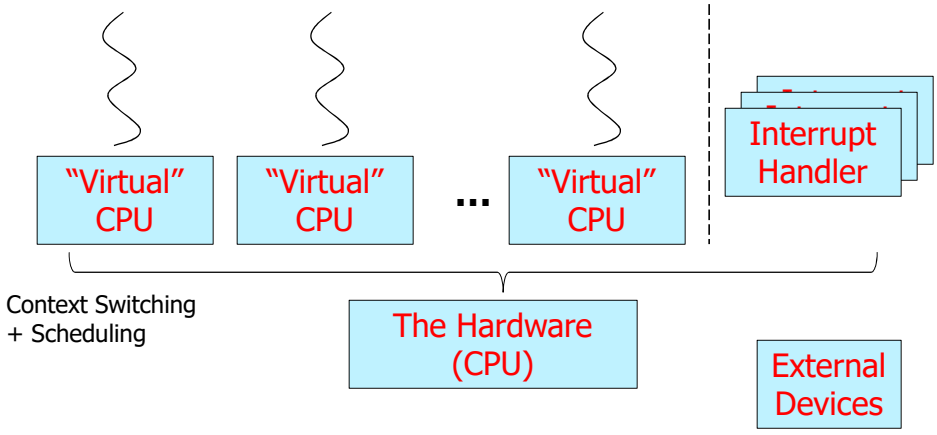
2

Where We Are:



3

Where We Are:



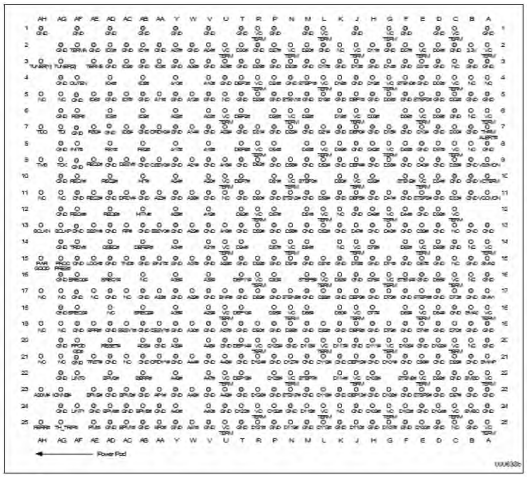
4

Hardware Interrupts

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an "Interrupt controller"
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passes an interrupt number
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler

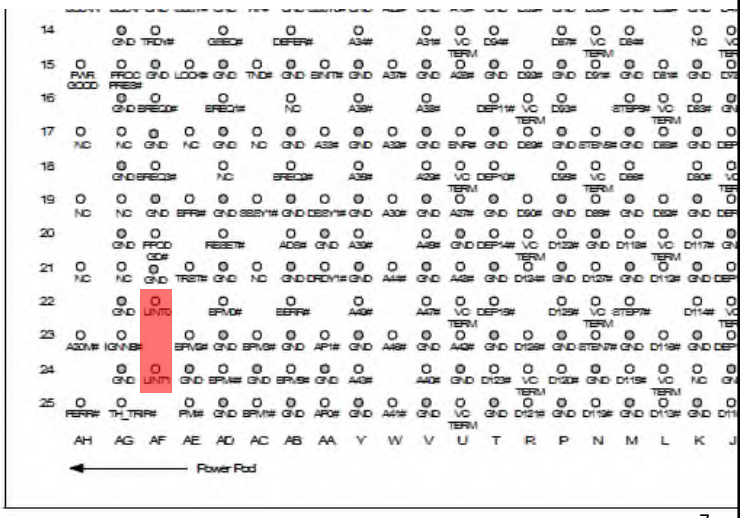
5

Example: Itanium 2 Pins



6

Itanium 2 Pins



7

Why Hardware Interrupts?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)

8



Other Interrupts

- Software Interrupts:
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Example: The system call interrupt
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel

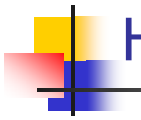
9



Other Interrupts

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

10



Registering an Interrupt Handler (top half)

- request_irq (irq, handler, flags, name, dev)
- free_irq (irq, dev)
- Notes:
  - *handler* is a pointer to the interrupt handler function
  - Interrupt handlers need not be re-entrant (same irq is masked until handler exits)
  - IRQ lines can be shared by multiple devices. The parameter dev is a unique "cookie" to be supplied by the given device and checked by the handler
  - The kernel sequentially invokes all handlers registered for a given irq

11



Interrupts, Priorities and Blocking

- Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  - What are the implications on regular program execution?

12



Interrupts, Priorities and Blocking

- Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  - What are the implications on regular program execution?
    - Must keep interrupt code short in order not to keep other processing stopped for a long time
    - Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system "hangs")

13



Interrupts, Priorities and Blocking

- Can an interrupt handler use malloc() ?
- Can an interrupt handler write data to disk?
- Can an interrupt handler use busy wait such as:
  - while (!event) loop;

14



Interrupt Context

- Interrupt handlers execute in the kernel. They are historically "trusted" (at least in Linux)
  - Problem?

15



Interrupt Context

- Interrupt handlers execute in the kernel. They are historically "trusted" (at least in Linux)
  - Problem: A bad handler (e.g., a buggy device driver that does not correctly handle interrupts from its I/O device) can crash the kernel or cause inefficiencies and poor performance

16



Interrupt Context

- Interrupt stack is a configuration option
  - Option 1: Use the stack of the interrupted process
  - Option 2: Share the common single kernel stack
  - Option 3 (v2.6 and higher): Use own stack

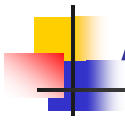
17



A Note on Multicore

- How are interrupts handled on multicore machines?

18



A Note on Multicore

- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts get routed to which CPUs.
  - The default (unless OS states otherwise) is to route all interrupts to processor 0

19



Bottom Halves

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
- The deferred portion of interrupt processing is called the "Bottom Half"

20



soft_irq

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses raise_softirq() to mark that a given soft_irq must execute the bottom half
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled

21



soft_irq Types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- …

22



soft_irq Types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

23



Tasklets

- Bottom halves multiplexed on top of soft_irq's
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)

24



Work Queues

- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
  - schedule_work(&work);

25



Example: Network Interrupts

- When packets are received from the network, packet reception triggers a hardware interrupt from the network card
- The hardware interrupt copies a packet from the network card memory and raises a soft_irq
- The soft_irq executes later and processes the packet in the kernel (enqueues it to the receiving application)
- The application is scheduled when priority permits and reads the packet

26



Example: Using Interrupts for Denial of Service

- Can one use interrupts for denial of service attacks?

27

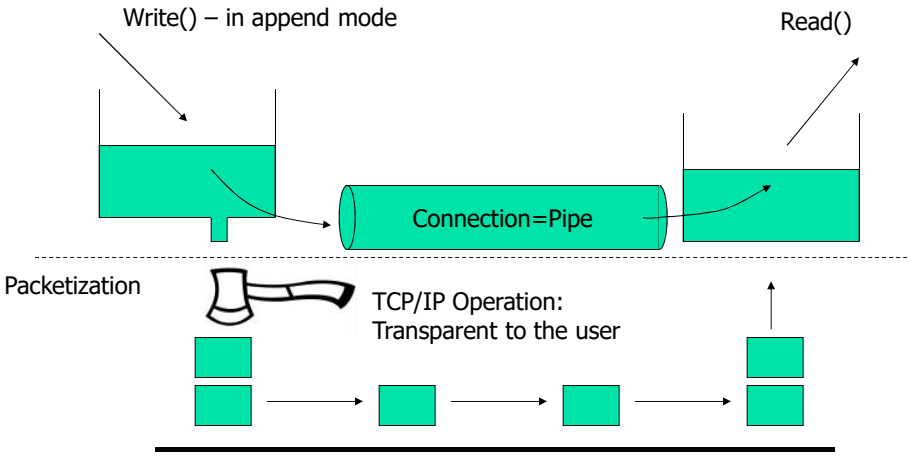


Example: Using Interrupts for Denial of Service

- Can one use interrupts for denial of service attacks?
  - Yes, since the network card interrupts you when it receives a packet from the network, external parties can use that to attack you
  - Example: The SYN attack in TCP

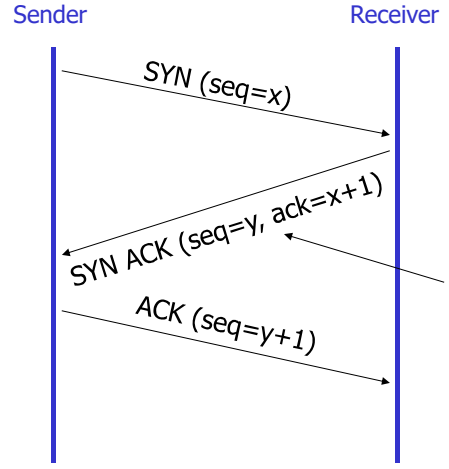
28

TCP Background: A Reliable Stream Abstraction



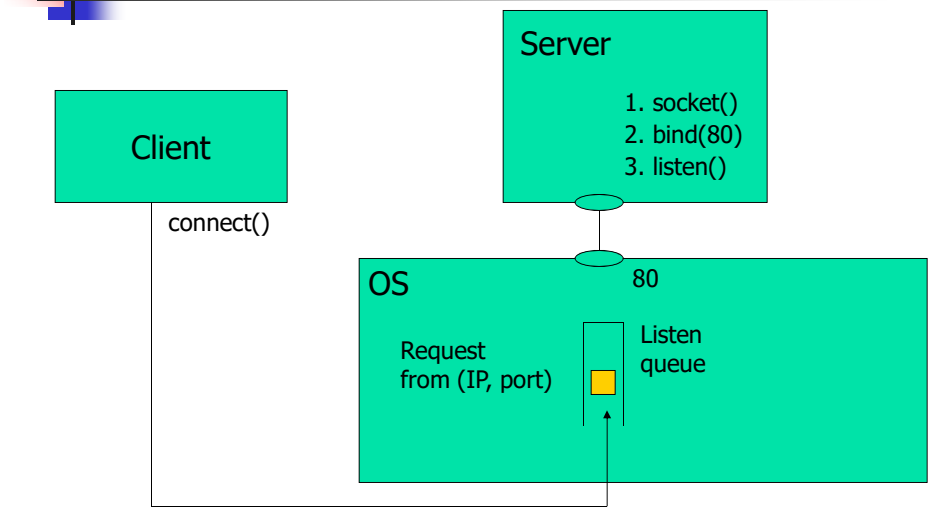
TCP Background: Connection Establishment





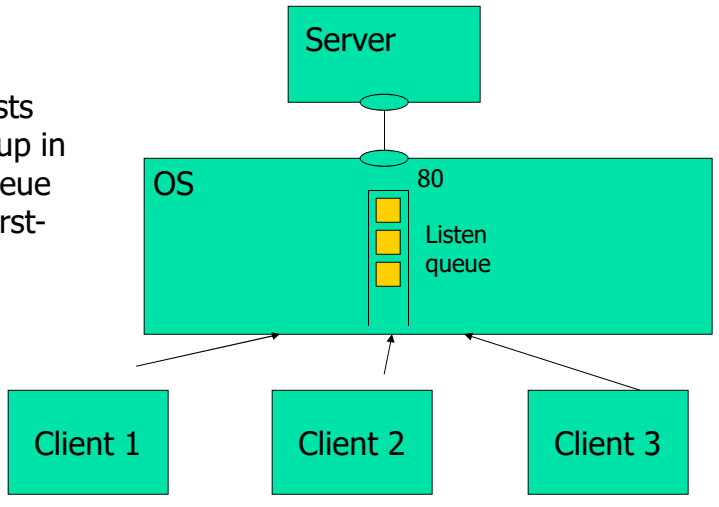
Example:
SeqNo = y
AckNo = x+1
SYN bit is on
ACK bit is on

Connecting to a Server



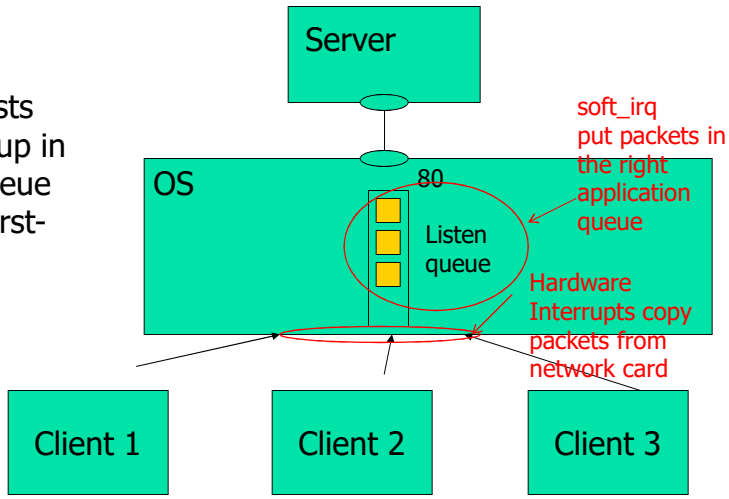
Busy Server Operation

Client requests get queued-up in the listen queue
First-come first-served



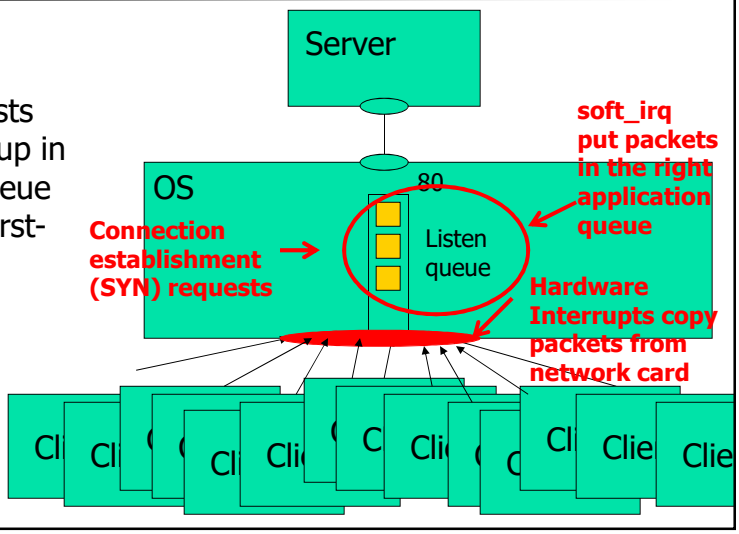
Busy Server Operation

Client requests get queued-up in the listen queue
First-come first-served



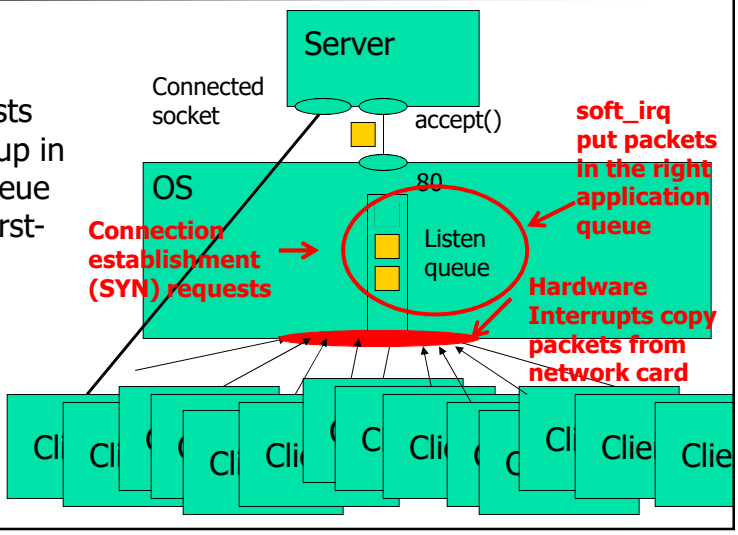
Denial of Service

Client requests get queued-up in the listen queue
First-come first-served



Denial of Service

Client requests get queued-up in the listen queue
First-come first-served



Denial of Service

Has a lower priority than the OS kernel (hence, does not get to run)

Client requests get queued-up in the listen queue
First-come first-served

