Operating Systems

Recitation 5

Plan

- Modes of operations
- What causes them

- Interrupts & Exceptions

Need protection

- Kernel privileged – cannot trust user processes
  - User processes may be malicious or buggy

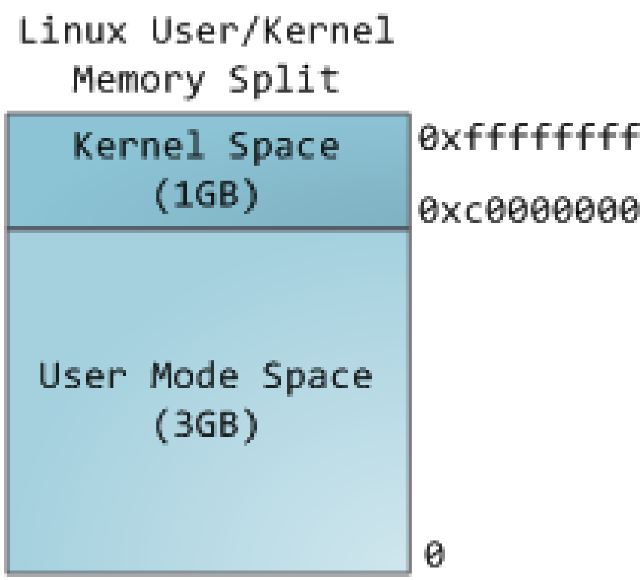
- Must protect
  - User processes from one another
  - Kernel from user processes

Hardware assisted memory protection

- <u>**Paging**</u> – we discussed virtual memory…
  - Every process has its own virtual memory address space
  - OS uses tables for translation, etc.
  - Mapped to physical memory
  - We occasionally swap to disk
  - ***P1*** can’t access memory of ***P2***

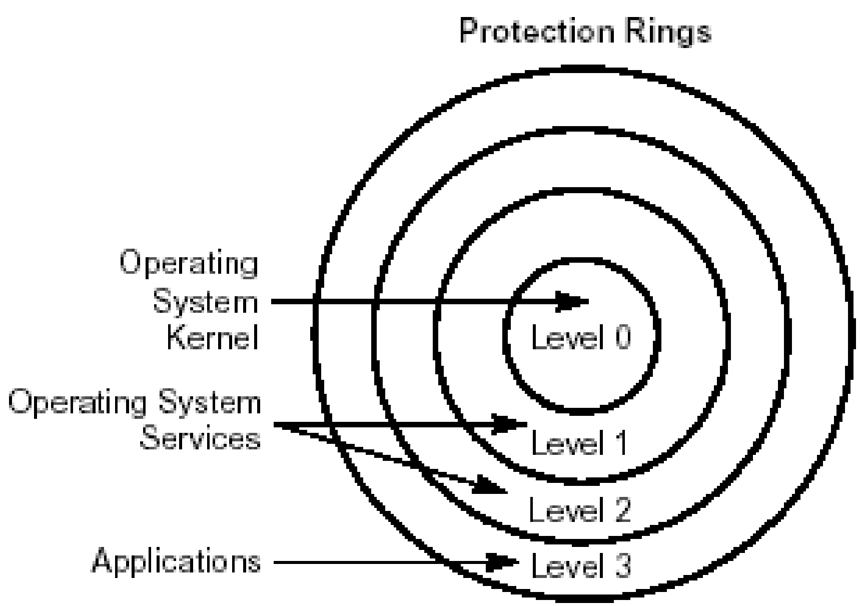
Segmentation

- Briefly mentioned too
  - Kernel code and data mapped in every process to same segments
  - User code etc. in different segments



Dual mode of operation

- ***Hardware*** (again) ***to the rescue!***
- Privileged (+ non-privileged) operations in kernel mode
- Only non-privileged operations in user mode



Segmentation + modes operation = ?

- Don’t want meddling user processes to mess with kernel stuff
- Each segment has **Descriptor Privilege Level** (DPL)
    - kernel segments: 0
    - user segments: 3
- Each thread has **Current Privilege Level** (CPL)
- Can only access segments when CPL <= DPL
- CPL controlled by OS (when switching modes) and checked against each segment’s DPL

Scheduling

- Timer interrupt (scheduler)
- Kernel periodically gets control back

Event driven mode-switch

- Events cause mode-switch
  - Interrupts: raised by devices to get OS attention
  - System calls: issued by user processes to request system services
  - Exceptions: illegal instructions (e.g., division by 0)

Why do we need interrupts?

- People like connecting devices
  - A computer is much more than the CPU
  - Keyboard, mouse, screen, disk drives…
- These devices occasionally need CPU service
  - But we can’t predict when…
- External events typically occur on a macroscopic timescale
- We want to keep the CPU busy between events

$\Rightarrow$ **Need a way for CPU to find out devices need attention**

Interrupts

Electric signal to processor, signifies event that requires immediate attention

- Causes kernel to stop performing current code and use “special code” to handle interrupt
- Mostly asynchronous and raised by devices to get OS attention
  - Keyboard keystroke
- /proc/interrupts - lists all hardware devices and how many interrupts received by each on each CPU (more in /proc/pci)
- In x86 (32bit) 256 types of interrupts

Interrupt Handler

- Often does very little at the expense of current running process
- Defers whatever it can to later execution by some kernel thread
- Example: Networking
  - time-critical work - copy network data packet off network card, respond to card and continue
  - deferred work - process data packet, pass to correct application (browser,…)

Interrupt view of CPU

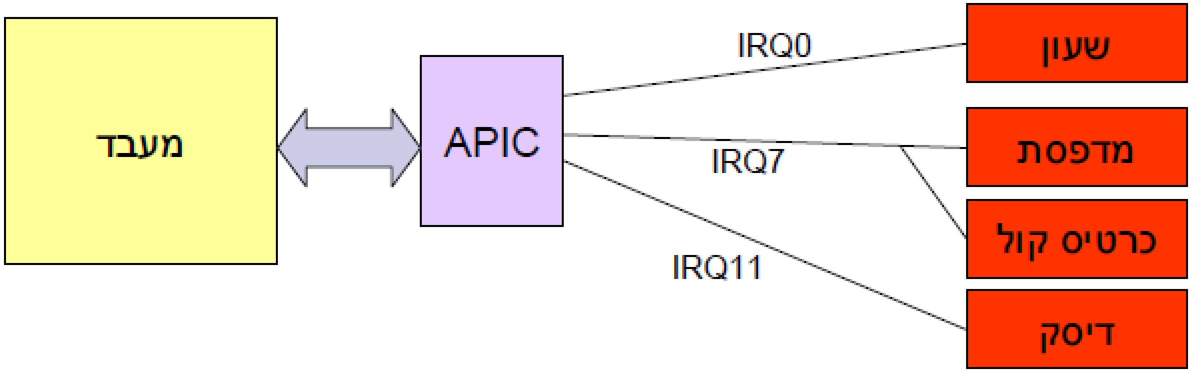
```
while (fetch next instruction) {
    run instruction;

    if (there is an interrupt) {
        process interrupt
    }
}
```

A deeper look

```
while (fetch next instruction) {
    run instruction;
    if (there is an interrupt) {
        switch to process kernel stack if necessary
        save CPU context and error code if any
        find OS-provided interrupt handler
        jump to handler
        restore CPU context when handler returns
    }
}
```

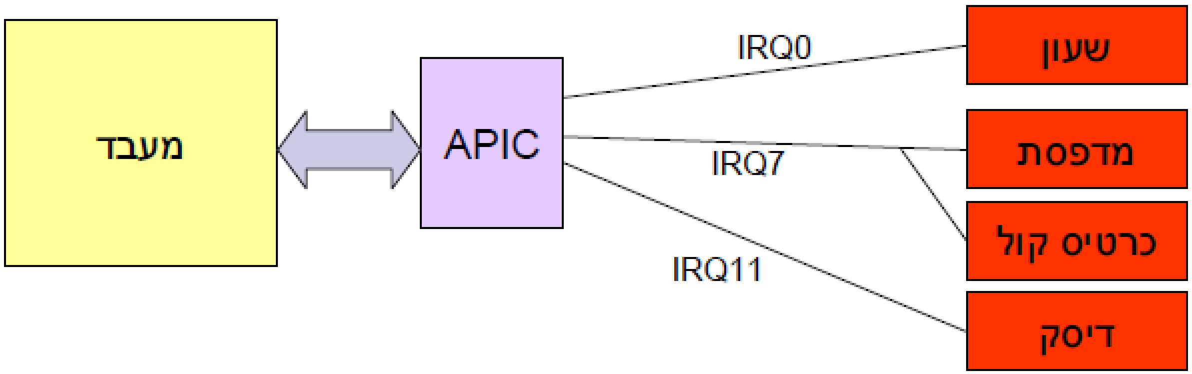
Finding the interrupt handler

- Processor has **[A]PIC**
  [Advanced] Programmable Interrupt Controller
  - In <u>multicore</u> systems - sophisticated OS mechanisms, frontend APIC routes to core-specific local APIC. Routing done with masking…
  - /proc/irq/XXX/smp_affinity



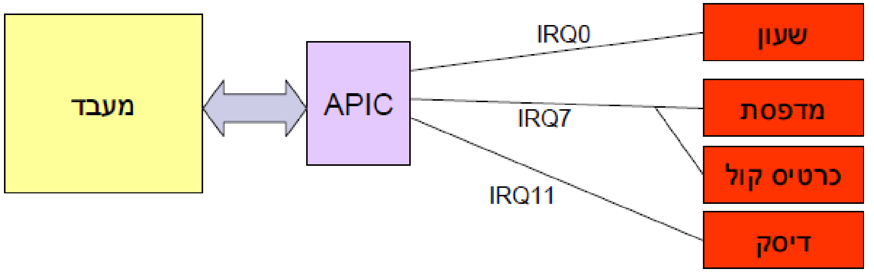
APIC

- On interrupt device sends signal to IRQ (Interrupt ReQuest) line
- IRQ sharing – need to check all devices that might have caused it
- Exceptions (sync. interrupts) do not use APIC



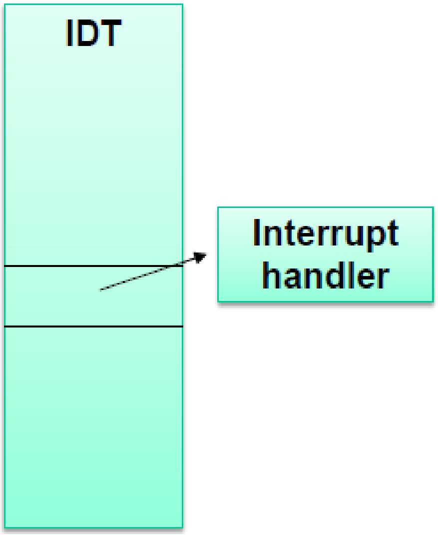
APIC

- APIC notifies CPU about specific interrupt and holds
  - Doesn’t serve interrupts meanwhile
- Linux ACKs specific interrupt
  - Allows APIC to receive more interrupts from <u>other</u> types
- Later when done, OS informs APIC it is ready to handle interrupts from specific type too

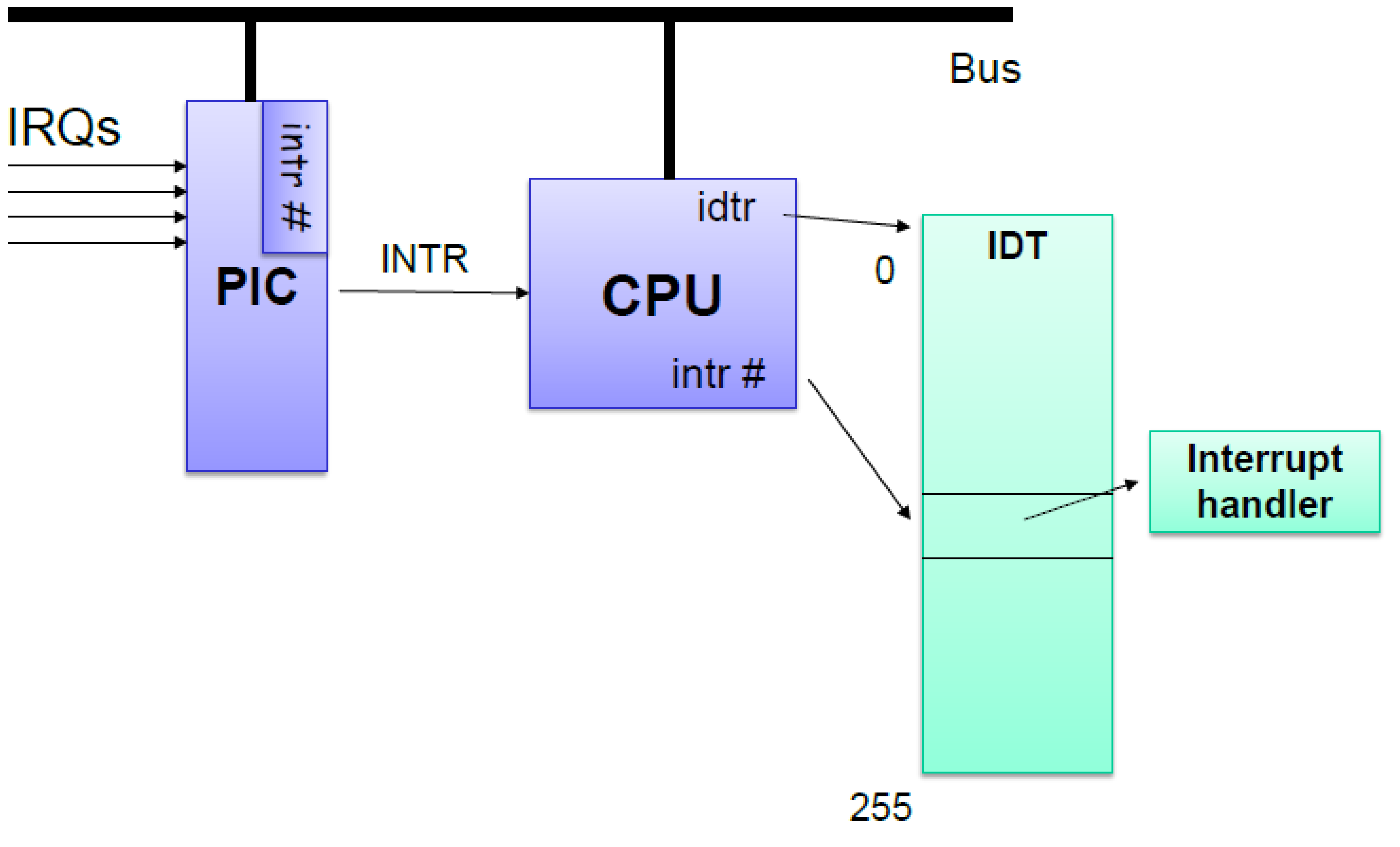


Interrupt Descriptor Table

- Hardware maps interrupt type to interrupt number
- On boot OS sets up “interrupt vector” or Interrupt Descriptor Table (IDT) in memory
  - Each entry is an interrupt handler
- OS lets hardware know IDT base in *idtr* register
- Hardware finds handler using interrupt num: IDT[intr_number]



All together now



IDT entries

- 0: divide by 0
- 1: debug (for single stepping)
- 3: breakpoint
- 14: page fault
- [128 used for syscall, soon more details…]
- 251-255 inter-processor interrupts

Interrupt Request

unsigned int do_IRQ(struct pt_regs regs);

- C function
- In kernel 2.6 arch/i386/kernel/irq.c
- Find relevant interrupt (from APIC registers)
- Invoke relevant handler called Interrupt Service Routine (ISR)
- ACK to APIC on relevant IRQ
  - Can handle more such interrupts

Interrupt Service Routine (ISR)

- Handles interrupts for specific device
  - Read keystroke code
  - Read data from network card
  - Perform I/O from disk
- Implemented by device drivers

Why not polling?

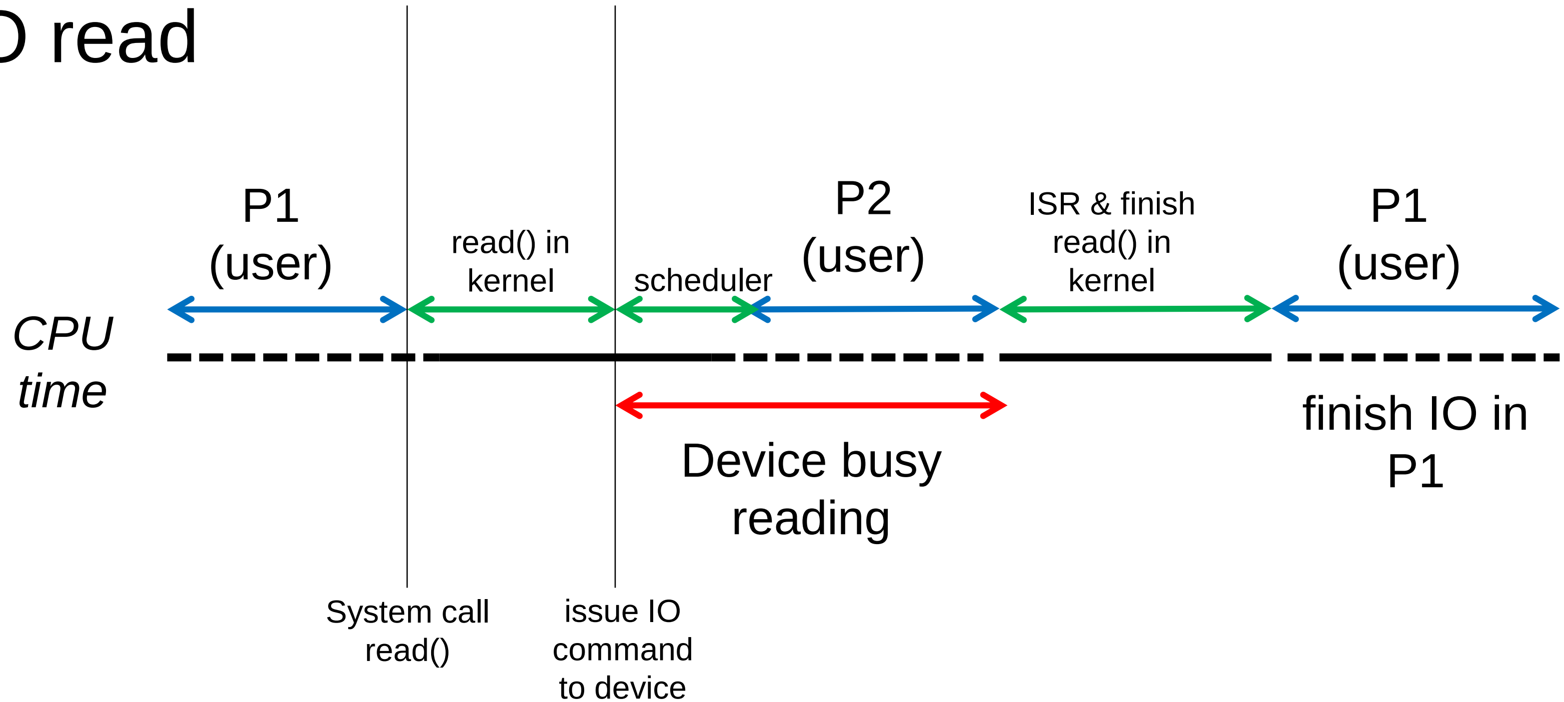
- CPU *periodically checks each device* to see if it needs service

"Polling is like picking up your phone every few seconds to see if you have a call. …"

- Takes CPU time when no requests are pending

Why not polling?

- IO read

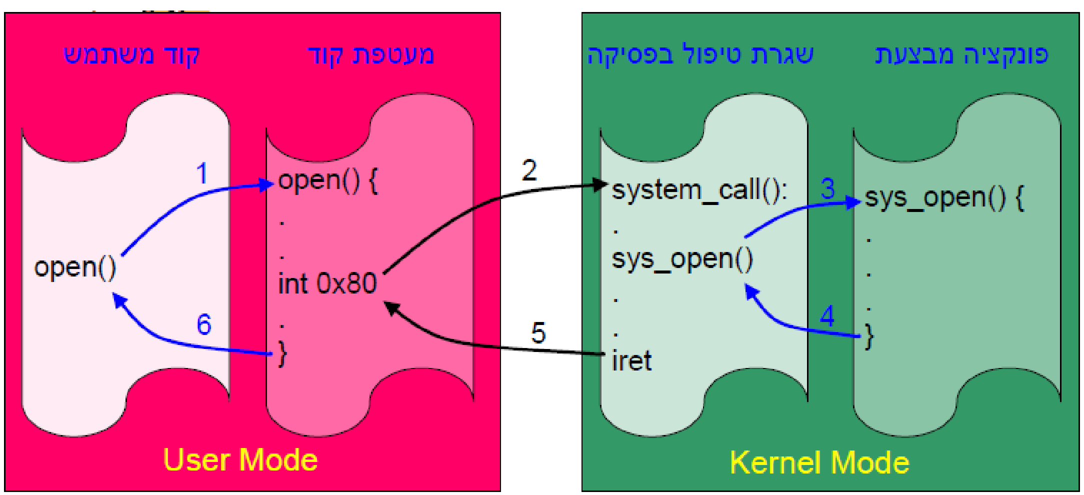


Exceptions == sync. interrupts

- Issued by processor
- *Exception in program execution*
  - Faults; offending instruction is retried
    - Division by 0
  - Traps; deliberate in code. instruction is not retried
    - Usually for debugging
  - Aborts; major error
    - hardware failure
- In multicore, cores often communicate via interrupts
    - And each core can handle different interrupts…

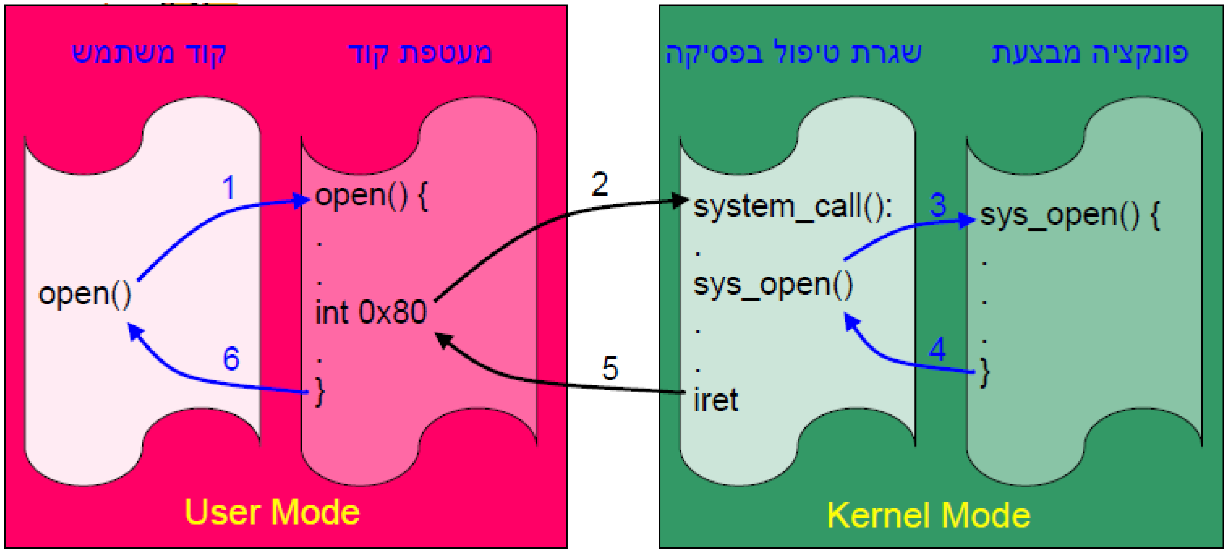
Programmed exceptions == software interrupts

- Mechanisms for special exceptions likely to occur
- Used for system calls! (and debugging)
  - Interrupt 128 (int 0x80) which invokes routine system_call()
  - Windows: int 0x2e



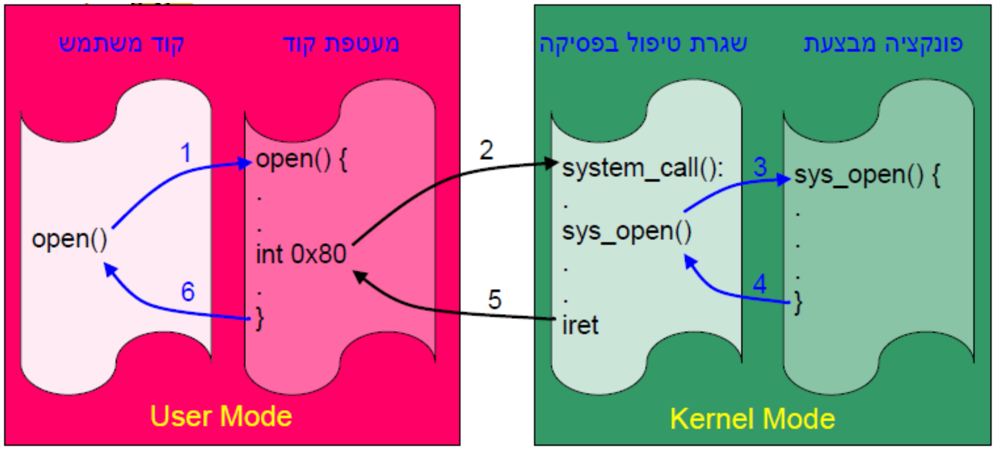
Programmed Exceptions

- On boot system call table (vector) initialized in kernel
  - Maps system call number to relevant implementation
- On demand user process sets up system call number & arguments in designated registers
- User process causes interrupt 128



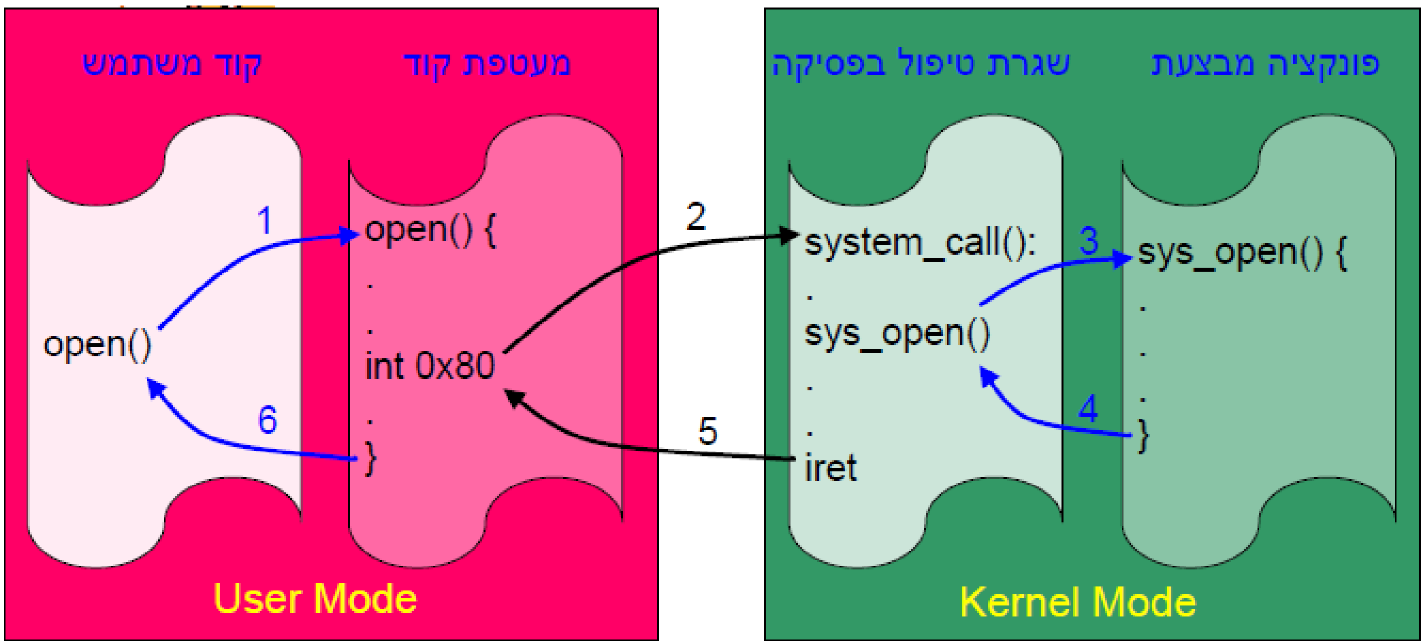
Programmed Exceptions

- Hardware switches to kernel mode and invokes relevant interrupt handler system_call()
- Kernel looks up syscall table using system call number from relevant register
- Kernel invokes the corresponding function
- Kernel returns to original execution point



Programmed Exceptions

- How is it used for debugging?



Linux Signals

Inform process async. on some event

- Form of IPC for messages from OS about events
  - 32 types in UNIX
- Some signals are handled automatically
  - KILL, STOP
- Others handled by signal handler
  - OS queues signals (only 1 for every type!)
  - Upon returning to user-mode process handles them using handler

Exception handling in Linux

- Reminder - exceptions are sync. Interrupts (faults, traps,…)
- Exception handler - saves relevant data on stack
- Then sends signal (async event…) to relevant process
- Signal type specific to exception
  - Divide by zero will send **`SIGFPE`**
  - Ctrl + C sends **`SIGINT`**
- Before returning to user mode, process checks signal handler

Linux Signals

Some signals are process-wide (KILL…)

- Threads can handle signals separately (using signal masks, somewhat complex)
- Want to handle some signals on your own? Implement your own signal handler…

Signals vs. interrupts

- Who’s talking
  - Interrupts → communication between CPU and OS kernel
  - Signals → communication between the OS kernel and OS processes (and themselves)
- Assumption
  - Interrupts assume OS is frozen and waiting for them to finish
  - Signals obviously do not