- Q1: When does the OS run?

- The OS is a **giant handler of events**, designed to handle two types of events:
  - **Asynchronous events**: Events that occur due to reasons external to what program instructions the processor is currently executing (e.g., a packet arrives from the web and the OS is notified to copy it to memory).
  - **Synchronous events**: Events that occur synchronously as a result of the execution of a program's instructions (e.g., a user program asks the OS to perform a privileged operation on its behalf; or, the processor executes an illegal instruction such as a division by zero).

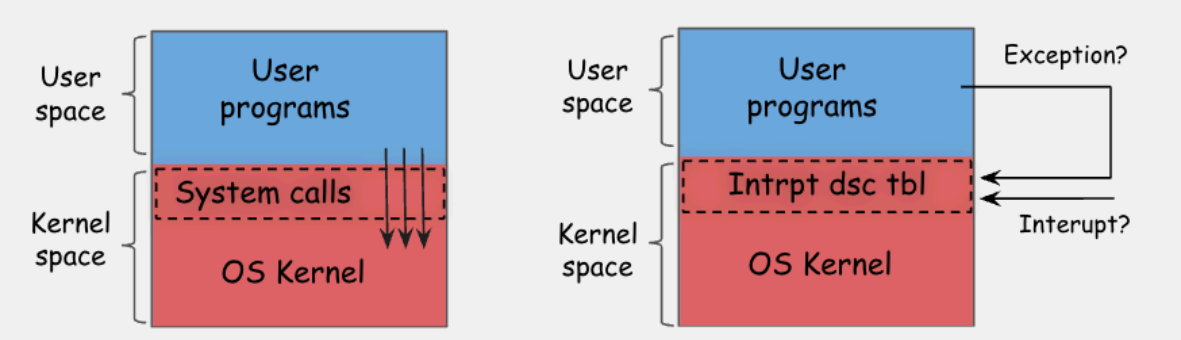
- Q2: What synchronous events invoke the OS?

- **System calls:** System calls (**syscalls**) are the interface between user programs and the OS kernel, and are the designated mechanism allowing user-space programs to request services from the OS. (For example, Linux x86_64 defines ~540 syscalls and aarch64 defines ~466 syscalls.)

- **Exceptions:** Synchronous events that are generated when the processor detects predefined "special" conditions while executing instructions. There are three classes of exceptions: faults, traps, and aborts—depending (i) on whether the OS can fix the precondition that caused them, or not; and (ii) on whether the instruction that triggered them will be re-executed after the OS returns control to the program whose execution caused the processor exception, or not.
  - **A fault** is an exception that can generally be corrected and that, once corrected, allows the program to be restarted without loss of continuity.
    - When a fault is reported in user mode, the processor automatically pushes its registers to the program's dedicated kernel stack.
    - This state is necessary so that the processor can restore its state prior to the beginning of the execution of the faulting instruction.
    - The return address for the fault handler points at the faulting instruction, rather than to the instruction following the faulting instruction. [restartable / e.g., page fault.] [Q: Why? So that after the fault is handled, it can be re-executed / re-started.]
  - **A trap** is an exception that is reported immediately following the execution of the trapping instruction.
    - Traps allow the execution of a program to be continued without loss of the program's continuity.
    - The return address for the trap handler points at the instruction following the trapping instruction. [not restartable / e.g., int 0x80 (the old way of making syscalls; also called a "software interrupt.")]
  - **An abort** is an exception that does not always report the precise location of the instruction that caused it.
    - Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables. [Not restartable, severe errors / e.g., hardware failures.]

- Q3: What asynchronous events invoke the OS?

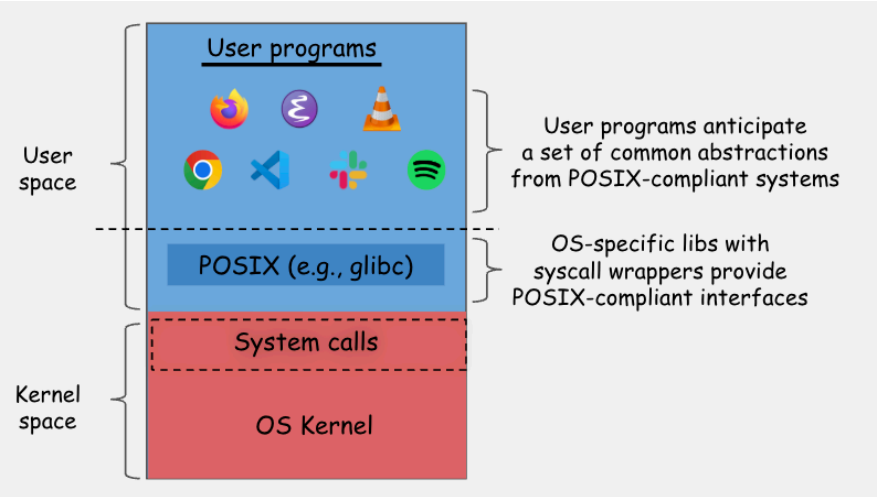
- **Interrupts:** Asynchronous events are typically triggered by I/O devices. Interrupts occur at random times during the execution of a program, in response to signals sent to the processor from external hardware. (The software can also generate synchronous interrupts, called "software" interrupts—e.g., in x86_32, by executing the int 0x80 instruction, which is the old way of making system calls.)



- **Q4: How do user programs know what services the OS offers?**
  - Since syscalls are the single way for user programs to request services from the OS, their interfaces must be well-defined.
  - User programs must not break on kernel updates
  - User programs must be **portable** across OSes of "the same family."
    - OK to recompile my code if I move to another OS
    - Don't make me rewrite my code if I move from one OS variant to another, because syscall SYS_read() takes three, instead of two, arguments.
  - Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces.
    - It describes a set of fundamental abstractions needed for efficient construction of applications.
    - While perfect portability was never a reality, the uniformity added by POSIX has always been valuable.
    - User programs can directly invoke system calls, but going through POSIX (e.g., by using glibc in GNU/Linux) ensures portability across different architectures of **POSIX-compliant** OSes.

- **Q5: What services do POSIX-compliant OSes offer?**
  - **POSIX defines 1,200 interfaces**, around the following core abstractions and mechanisms.
    - "A **process** is an address space with one or more threads executing within that address space, and the required system resources for those threads." IEEE Std 1003.1-2008 (POSIX.1-2008), Base Definitions, Section 3.189.
    - "A **thread** is a single flow of control within a process, with its own thread ID, scheduling priority and policy, errno value, floating-point environment, thread-specific key/value bindings, and the required system resources to support a flow of control." IEEE Std 1003.1-2008 (POSIX.1-2008), Base Definitions, Section 3.190.
    - "A **file is** an object that can be written to, read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, symbolic link, socket, and directory." IEEE Std 1003.1-2008 (POSIX.1-2008), Base Definitions, Section 3.139.
    - All definitions are here: https://pubs.opengroup.org/onlinepubs/9799919799/.

POSIX-compliant OSes



- **Q6: What interrupts and exceptions are supported by the x86 and ARM processor architectures?**

x86 interrupt and exception vectors

Vector NR	Exception/Interrupt Name	Type
	Divide Error	Fault
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	Trap
4	Overflow Exception	Trap
5	Bound Range Exceeded	Fault
6	Invalid Opcode	Fault
7	Device Not Available (No Math Coprocessor)	Fault
8	Double Fault	Abort
9	Coprocessor Segment Overrun (Legacy)	Fault
10	Invalid TSS	Fault
11	Segment Not Present	Fault
12	Stack-Segment Fault	Fault
13	General Protection Fault (GPF)	Fault
14	Page Fault	Fault
15	Reserved	-
16	x87 Floating-Point Exception	Fault
17	Alignment Check	Fault
18	Machine Check	Abort
19	SIMD Floating-Point Exception	Fault
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt

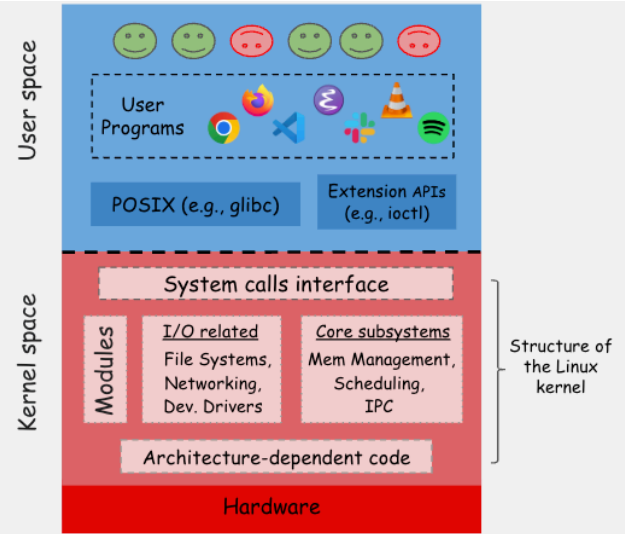
ARM interrupt and exception vectors

Vector NR	Exception/Interrupt Name	Type
-	Initial Stack Pointer	-
1	Reset Handler	
2	NMI Handler	
3	HardFault Handler	Fault
4	MemManage Fault	
5	BusFault Handler	Abort
6	UsageFault Handler	Fault
7-10	Reserved	Reserved
11	SVCall Handler	
12	Debug Monitor	
13	-	Reserved
14	PendSV Handler	
15	SysTick Handler	
16+	External Interrupts	Reserved

- Q7: What is Linux?

- A modern, open-source, POSIX-compliant operating system kernel.
- Written in '91 by Linus Torvalds from scratch ~ 100 KLoC (Jan '25: > 40 MLoC).
- Strictly speaking, Linux is an OS kernel and does not include user-space tools, libraries, or a full system environment on its own.
- In common usage, "Linux" refers to Linux-based operating systems (also called distributions).
- The most common Linux distributions include Ubuntu, Debian, Fedora, and RedHat.
- **The Linux kernel structure**
    - Core kernel subsystems
        - Memory management
        - Processor time management
        - Interprocess Communication (IPC).
    - I/O subsystems
        - Device drivers
        - File systems
        - Networking
    - Dynamically loadable modules (~60% of the codebase)
        - Originally developed to support the conditional inclusion of device drivers.
        - Otherwise, the kernel would either need to include code for all possible devices or to be recompiled to add support for every new device.
- Linux is by far the most popular Unix-like operating system today
    - Runs on **billions** of processors
        - Servers: Over 96% of the world's top 1 million websites run on Linux servers.
        - Cloud: AWS, Google Cloud, and Azure rely heavily on Linux.
        - Supercomputers: Almost 100% of the top 500 supercomputers run Linux.
        - Android devices: >3 billion devices → Android is built atop the Linux kernel.

Structure of the Linux kernel



- **Q8: What per-processor state does the Linux kernel set during early boot in x86?**
  - To properly handle syscalls, exceptions, and interrupts, the operating system must set up several critical tables in dedicated per-processor memory areas, including
    - An Interrupt Descriptor Table (IDT), which holds descriptors to Interrupt Service Routines (**ISRs**) and exception handlers, along with their respective Descriptor Privilege Level (**DPL**)—which, the processor automatically checks against its Current Privilege Level (**CPL**) of execution every time an ISR is invoked**.**
    - A Global Descriptor Table (GDT), which holds descriptors with information about
      - User and kernel, code and data segments (e.g., base address and size).
      - A Task State Segment (TSS) descriptor with information about the address of the kernel stack for the program that the processor is currently executing (e.g., the TSS.sp0 field, which, the processor automatically uses to update the value of the %esp register and switch to the program's kernel stack in cases of exceptions and interrupts; or the TSS.sp1 field, used by the OS for the stack switching during "fast" syscalls).
      - Relevant registers
        - The **%idtr** register holds the base address of the IDT.
        - The **%gdtr** register holds the base address of the GDT.
        - The **%tr** register points at the GDT, the TSS descriptor of the program that the processor is running at each point in time.
        - The **%cs** register, which the processor automatically updates during privilege level transitions to point to the current code segment descriptor in the GDT, and, thus, the CPL is always implicitly derived from the lowest two bits of the %cs register: CS = 0x08 is ring-0 (kernel mode) and CS = 0x1B is ring-3 (user mode).
      - Relevant instructions
        - The **lgdt** instruction is used to load the address of the GDT into the %gdtr register (see this).
        - The **lidt** instruction is used to load the address of the IDT into the %idtr register (see this).
        - The **ltr** instruction is used to load the address of the TSS descriptor from a GDT selector to the %tr register (see this).

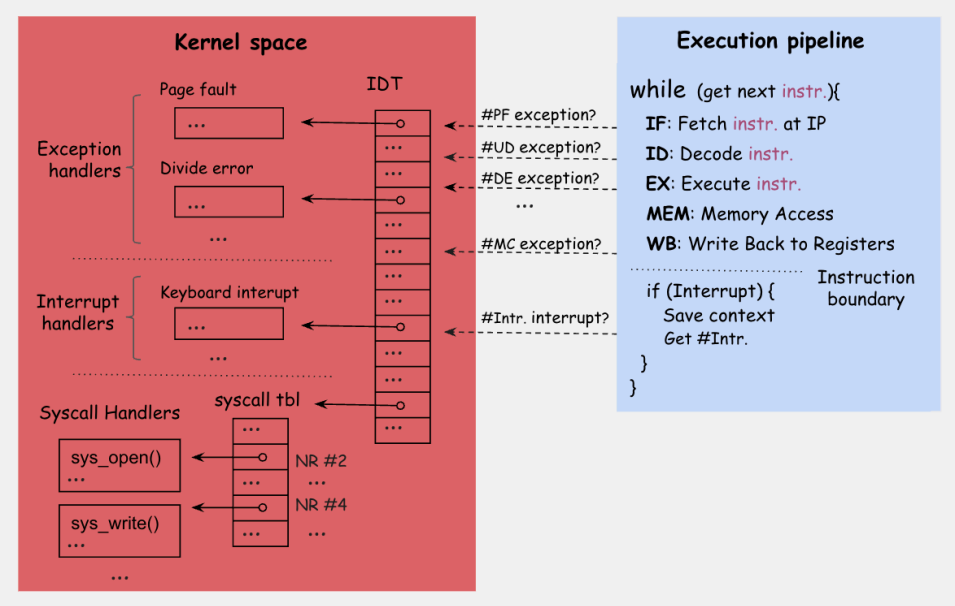
- **In summary:** The start_kernel() function in init/main.c: (1.) sets up the per-processor GDT, and loads its base address into the %gdtr register; (2.) sets up the processor's stacks for handling critical exceptions; (3.) sets up and loads the TSS descriptor (required for stack switching during exceptions) into the %tr register, and allocates necessary interrupt/exception stacks; (4.) configures processor fault handlers, and finally (5.) initializes hardware interrupt handlers (IRQ) in the IDT, and loads the IDT into the %idtr register.

- **Linux kernel init/main.c:start_kernel() call graph**
  - start_kernel()
    - setup_per_cpu_areas() / * Allocates and sets up per-processor GDT */
      - switch_gdt_and_percpu_base()
        - load_direct_gdt() / * Loads the original GDT address and size from the kernel's structure into the processor's entry area */
          - load_gdt() / * Loads the GDT into the GDTR */
            - native_load_gdt() / * Uses the lgdt instruction */
    - trap_init() / * Sets up early exception and fault handlers */
      - setup_cpu_entry_areas() / * Allocates and sets up per-processor areas for exception handling */
        - setup_cpu_entry_area() /* For each processor */
          - percpu_setup_exception_stacks() / * Sets up the processor's stacks for handling critical exceptions (such as non-maskable interrupts, double faults, and machine check failures) that need dedicated stacks since they can happen at any point */
      - cpu_init_exception_handling() / * Loads TSS and prepares handlers for early processor faults */
        - set_tss_desc() / * Allocates an entry in the processor's GDT to hold the TSS descriptor */
          - __set_tss_desc()
        - load_TR_desc() / * Load the TSS descriptor */
          - native_load_tr_desc() / * Uses the ltr instruction */
        - idt_setup_traps() / * Sets up the IDT with exception handlers */
          - idt_setup_from_table()
            - Uses INTG for interrupt descriptor gates of interrupts that don't need their interrupt-specific kernel stack.
            - Uses ISTG for interrupt descriptor gates of critical interrupts that can't be ever masked, and since they may occur at any point in time, must always run on a dedicated, per-interrupt-specific, kernel stack.
    - init_IRQ() / * Initializes interrupt handling (hardware IRQs) */
      - x86_init.irqs.intr_init() /* Architecture-specific init */
        - native_init_IRQ() /* Populates the IDT with IRQ entries */
          - idt_setup_apic_and_irq_gates() / * APIC handlers */
            - load_idt() / * Loads the IDT into IDTR */
              - native_load_idt() / * Uses the lidt instruction */

- **Q9: What are the steps involved in handling a legacy int 0x80 syscall in Linux x86?**
  - **int 0x80 / iret (**known as **"**software interrupts")
    - The following description is for the x86_32, but the flow is similar for x86_64 as well.
    - The user program places the syscall number in the %eax register and the respective arguments in the %ebx, %ecx, %edx, %esi, %edi, and %ebp registers, and executes the int 0x80 instruction from user mode (ring-3).
    - The processor uses interrupt vector 0x80 in the IDT, which points at the syscall handler.
    - The processor checks if CPL (user mode: ring-3) <= DPL (ring-3 for vector #128), and raises a general-protection exception if the CPL is greater than the DPL value specified in the IDT for the selected gate descriptor.
    - Since the selected descriptor (i.e., the syscall handler) targets a kernel code segment (__KERNEL_CS, ring-0), the processor performs a privilege-level stack switch
      - Saves (internally) the values of the %ss, %esp, %eflags, %cs, and %eip registers.
      - Updates the values of %esp and %ss registers to the address of the task's kernel stack address for interrupt handling, stored in the sp0 field of the TSS pointed to by the %tr register in the GDT.
      - Pushes the temporarily saved (internally) values of the %ss, %esp, eflags, %cs, and %eip registers onto the new stack.
      - Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt or trap gate) into the %cs and %eip registers, respectively.
      - Clears the IF flag in the %eflags register, effectively disabling maskable interrupts.
      - Begins executing the entry_INT80_32 handler at the new privilege level (updated %cs).
    - The entry_INT80_32 handler does the following
      - Saves all general-purpose registers in the task's kernel stack, and uses "switch_stacks=1" indicating that the execution must also be saved in the dedicated kernel stack for handling critical exceptions (such as non-maskable interrupts) that may occur at any time, and require their dedicated stack to be already populated.
      - Executes the requested syscall handler (whose number is held in the %eax register).
      - Switches back to the entry's kernel stack in case a nested exception had occurred while handling the syscall.
      - Restores general-purpose registers from the kernel stack.
      - Uses the iret instruction, which pops values from the stack to restore the %eflags, %cs, and %eip registers—re-enabling maskable interrupts and resuming execution at the previous privilege level and point—and if a stack switch occurred, also restores the values of the %ss and %esp registers, switching back to the user-space stack.
    - For more details, see: “Call and Return Operation for Interrupt or Exception Handling Procedures,” Intel Software Developer's Manual, Vol.1, Sec. 6.5.1.
    - **Overall**: The above is considered the legacy "slow path" (~600 processor cycles; see this), and the faster alternative (~70 processor cycles; see this) for making system calls is the sysenter-sysexit (x86_32) / syscall-sysret (x86_64) mechanisms, described next.

- **Q10: What are the steps involved in handling a fast syscall in Linux x86?**
    - Around the Pentium-4 era (2000), processors became very deeply pipelined (e.g., from ~10 to ~20 stages), and therefore pipeline stalls/flushes became very expensive.
    - A typical int 0x80 system call took about twice as long on Pentium 4 compared to Pentium 3 because the IDT entry might have not been in the cache, which would cause a cache miss and a pipeline stall.
    - Since syscalls are common enough, caching the syscall entry point in a special processor register to make the common path fast, instead of looking it up in the IDT, is worth the transistor budget.
    - The preferred method for making syscalls in x86 is the **sysenter** / **sysexit** (x86_32) or the **syscall / sysreturn** (x86_64) instruction pairs.
    - These methods depend on a set of Model-Specific Registers (MSRs) configured during kernel initialization to define the associated kernel context and the syscall entry point.
        - The **IA32_SYSENTER_CS** holds the kernel code segment selector indicating the kernel privilege level (ring-0) for sysenter, initialized here.
        - The **IA32_SYSENTER_EIP** holds the %eip to be used for sysenter—that is, the kernel entry point for sysenter, initialized here.
        - The **IA32_SYSENTER_ESP** holds the stack pointer. (Unused since stack switching is software-based in sysenter.)
    - The user program places the syscall number in the %eax register, and the respective arguments in the %ebx, %ecx, %edx, %esi, %edi, and %ebp registers, and executes the **sysenter** instruction (x86_32) from user mode (ring-3).
    - The processor automatically
        - Loads the value of the IA32_SYSENTER_CS register into the %cs register, effectively switching to ring-0.
        - Loads the value of the IA32_SYSENTER_ESP register into the %esp register—will be overwritten, since stack switching is software-based.
        - Loads the value of the IA32_SYSENTER_EIP register, which holds the address of the entry_SYSENTER_32, into the %eip register and jumps at it.
    - The entry_SYSENTER_32 handler does the following
        - Updates atomically the value of the %esp register to the address of the task's kernel stack stored in the sp1 field of the TSS. (The TSS.sp1 field is updated by the OS during switching from one program to another so that it always points at the current program's kernel trampoline stack.) [Q: Why not use TSS.sp0?]
        - Pushes placeholder register values into the stack to emulate an interrupt frame for later sysexit return, and saves all general-purpose registers to the stack using the SAVE_ALL macro to.
        - Disables maskable interrupts.
        - Executes the requested syscall handler.
        - Prepares for the sysexit instruction by manually popping the stack and setting the values for the %ebx, %esi, %edi, %ebp, and %esp registers.
        - Executes the sti instruction (set interrupt flag) to reenable maskable interrupts.
        - Executes the sysexit instruction, which performs a fast transition back to user mode (ring-3) using values from %ecx (return %eip) and %edx (return %esp).
    - Unlike the int 0x80 syscall mechanism, the sysenter syscall mechanism (i) does not perform interrupt vector lookup via the IDT, (ii) does not incur the overhead of a full privilege level switch via the gate descriptor and the task segment, and (iii) does not trigger a hardware-based task switch. (See instruction tables microbenchmarks.)

- **Q11: How are interrupts and exceptions handled in Linux x86?**
  - The OS uses the same dispatching mechanism for both interrupts and exceptions.
  - The main difference is that exceptions are synchronous events that are being raised by the processor when an "exceptional" condition is identified while executing instructions; whereas, interrupts are asynchronous events that can appear at any point in time, regardless of the execution state the processor is in. Therefore, interrupts must be handled on instruction boundaries (i.e., at the end of the execution pipeline) to avoid corrupting the processor's internal state by interrupting its execution pipeline mid-execution, in unanticipated ways.
  - The Interrupt Stack Table (IST) in x86 architecture allows specifying different stacks for different interrupts. (Up to 7 IST entries per processor.)



  - **Handling interrupts**
    - The interrupt controller (e.g., the PIC) informs the CPU that there is a pending interrupt with a specific interrupt vector number.
    - Unless maskable interrupts are disabled, the processor automatically
      - Uses the vector number to index the respective gate descriptor in the IDT.
      - If the interrupt requires a privilege level transition (i.e., DPL of the target code segment < of the CPL in the %cs register), a stack switch occurs
        - The processor saves (internally) the values of the %ss, %esp, %eflags, %cs, and %eip registers.
        - Updates the values of %esp and %ss registers to the address of the task's kernel stack address for interrupt handling, stored in the sp0 field of the TSS pointed to by the %tr register in the GDT.
        - Pushes the temporarily saved (internally) values of the %ss, %esp, eflags, %cs, and %eip registers onto the new stack.
      - If a privilege level transition is not required
        - The processor only pushes the values of the %ss, %esp, %eflags, %cs, and %eip registers onto the new stack.

        - Pushes an error code on the stack (if applicable).
        - Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt or trap gate) into the %cs and %eip registers, respectively.
        - Clears the IF flag in the %eflags register to disable maskable interrupts.
        - Loads the ISR handler's address (offset) from the gate descriptor into the %eip register, effectively transferring control to the ISR handler
    - The ISR handler
        - Saves the current execution state (i.e., registers and flags) to the stack.
        - Executes the respective handler.
        - Restores general-purpose registers from the stack.
        - Uses the iret instruction, which pops values from the stack to restore the %eflags, %cs, and %eip registers—re-enabling maskable interrupts and resuming execution at the previous privilege level and point—and if a stack switch occurred, also restores the values of the %ss and %esp registers, switching back to the user-space stack.

- **Handling exceptions**
    - The difference between interrupts and exceptions lies in their origin: interrupts originate from the hardware, while exceptions are caused by software errors or exceptional conditions within the running program.
    - Both mechanisms rely on the same general flow, utilizing the IDT and processor state saving/restoring mechanisms.

- Q12: How are interrupts, exceptions, and syscalls handled in ARM?

- The ARM architecture manages interrupts, exceptions, and system calls using a hierarchical exception model with Exception Vector Tables **(EVTs)** and Exception Levels **(EL)** registers.
- **CurrentEL**: Holds the current exception level (EL)—i.e., the current privilege level.
    - aarch32 has two exception levels: EL0 is user mode and EL1 is kernel mode.
    - aarch64 has two additional exception levels, that is EL0–4. [Q: what for?]
- **Exception Syndrome Register** (**ESR_ELn**): Holds the cause of an exception (e.g., syscall, IRQ, etc.) that occurred in ELn. (The ESR_EL0 register is used to check the cause of an exception in user mode.)
- **Saved Program Status Register** (**SPSR_ELn**): Holds the processor's status when transitioning from one exception level to another. That is, the SPSR_EL1 is used to save state when the processor switches from one privilege level to another.
- **Current Program Status Register (CPSR):** Holds all the status and control bits for the processor, akin to the PSTATE register in aarch64, or to the eflags register in x86.
- **Exception Link Register** (**ELR_ELn**): Holds the address of the instruction to return to after handling an exception in ELn. That is, the ELR_EL1 register holds the return address when switching to kernel mode.
- There is neither a VBAR_EL0 nor a SPSR_EL0 register, since no exception is to be handled in user mode (EL0).
- The following description is for aarch32, but the flow is similar for aarch64 as well.
- When an exception (including an interrupt or syscall) occurs, the processor
    - Saves its state and return address in the SPSR_ELn and ELR_ELn registers.
    - Updates the CurrentEL field in the state register to the new privilege level.
    - Jump to the appropriate entry in the table pointed to by the VBAR_ELn register.
- The user program places the syscall number in the %r7 register, and the registers %r0–r6 are used for argument passing.
- The user program executes the **svc** instruction from user mode (EL-0).

- The **svc** instruction raises an exception, causing the processor to automatically
    - Switch from EL-0 (user mode) to EL-1 (kernel mode) by updating the value of the CurrentEL register.
    - Save its context in the SPSR_EL1 and LR_EL1 registers.
    - Uses the exception number in the ESR_ELn register to find the respective handler in the vector table indexed by the VBAR_L1 register.
    - Invoke the handler (e.g., in this case, the requested syscall handler).
- The handler checks the syscall number in the %r7 register, extracts the args from the %r0–r6 registers, and invokes the appropriate kernel syscall handler.
- Upon completion, the kernel uses the **eret** instruction to
    - Restore the %pc and state registers from ELR_EL1 and SPSR_EL1.
    - Switch back the value of the currentEL register to EL0 (user mode).

- Q13: How is the validity of user-provided data assessed by the Linux kernel?

- The Linux kernel maps its physical memory into the virtual address space of each running user program.
- This helps keep the TLB entries "hot," and reduces the need to flush the TLB on every transition between user and kernel mode.
- Since user programs interact with the kernel by passing data during system calls, and user programs cannot be trusted
    - Before reading from or writing to user memory, the kernel verifies that the user program has the appropriate permissions.
    - This prevents user programs from tricking the kernel into accessing memory they shouldn't be able to.
    - The kernel provides controlled error handling, returning errors like EFAULT when an invalid memory region is accessed as a result of a user program providing an invalid address (e.g., unmapped memory, swapped-out pages, or zero pages).

- **From kernel to user space**
    - `long copy_to_user(...);`
    - `int set_user(...);`

- **From user to kernel space**
    - `long copy_from_user(...);`
    - `int get_user(...);`