K22 – Operating Systems: Design Principles and Internals

Fall 2026 @dit (α.μ. mod 2 = 1)

Vaggelis Atlidakis et al.

Lecture 03

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

Overview

- We'll follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - Events [Q: When does the OS run?]
  - Runtime [Q: How does a program look like in memory?]
  - Processes [Q: What is a process?]
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* **Abstractions**
* **Primitives**
* **Mechanisms**

Overview

- We'll follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - Events [Q: When does the OS run?]
  - Runtime [Q: How does a program look like in memory?]
  - Processes [Q: What is a process?]
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* Abstractions
* Primitives
* Mechanisms

Overview

- **Events**
  - Q1: When does the OS run?
  - Q2: What synchronous events invoke the OS?
  - Q3: What asynchronous events invoke the OS?
  - Q4: What exceptions are supported by the x86 processor architecture?
  - Q5: How do user-programs invoke the OS?
  - Q6: What are the steps involved in handling a legacy int 0x80 syscall in Linux x86?
  - Q7: How do user programs know what services the OS offers?
  - Q8: What is POSIX?
  - Q9: What is Linux?

...

When does the OS run?

When does the OS run?

The OS is a giant handler of events, which runs in response on two types of events

When does the OS run?

The OS is a giant handler of events, which runs in response on two types of events

- ***Asynchronous events***: Events that occur due to reasons external to the program instructions that the processor was currently executing

When does the OS run?

The OS is a giant handler of events, which runs in response on two types of events

- ***Asynchronous events***: Events that occur due to reasons external to the program instructions that the processor was currently executing ⇒ Interrupts

When does the OS run?

The OS is a giant handler of events, which runs in response on two types of events

- ***Asynchronous events***: Events that occur due to reasons external to the program instructions that the processor was currently executing ⇒ Interrupts
- ***Synchronous events***: Events that occur synchronously as a result of the execution of a program's instructions

What synchronous events invoke the OS?

What synchronous events invoke the OS?

Exceptions invoking the OS

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction
  - Does not break program continuity

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction
  - Does not break program continuity
  - Next instruction is the one following the trapping instruction

What synchronous events invoke the OS?

Exceptions invoking the OS

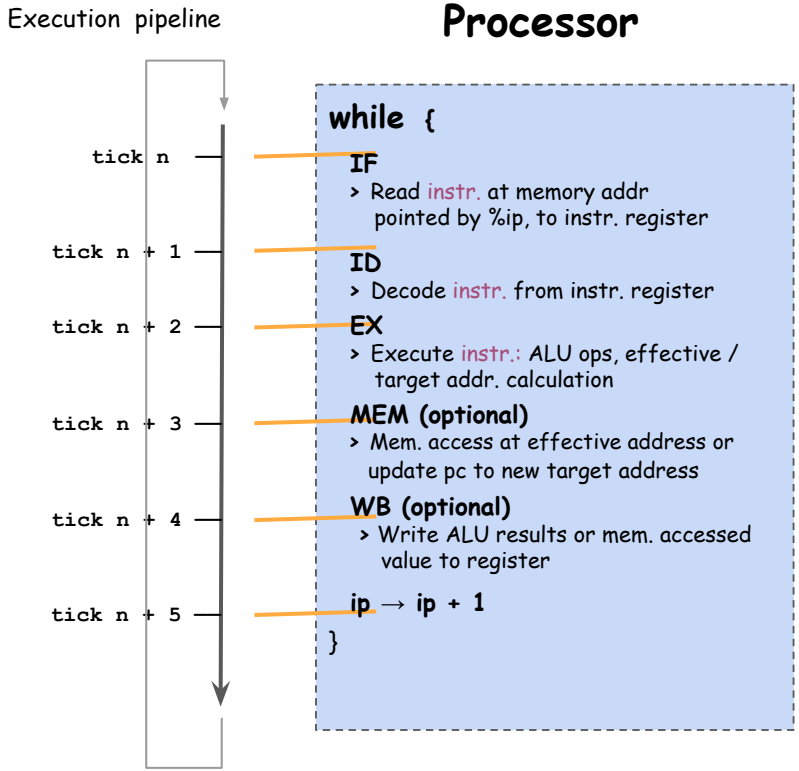
- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction
  - Does not break program continuity
  - Next instruction is the one following the trapping instruction
  - The old way of making system calls!

X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	Divide Error	
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	
4	Overflow Exception	
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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt

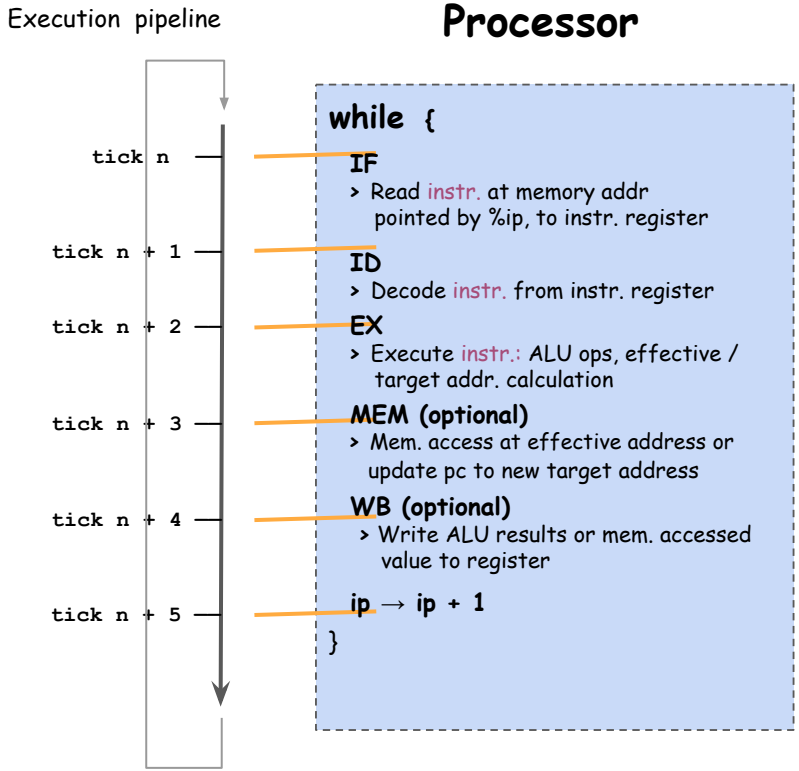
X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	Divide Error	
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	
4	Overflow Exception	
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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt



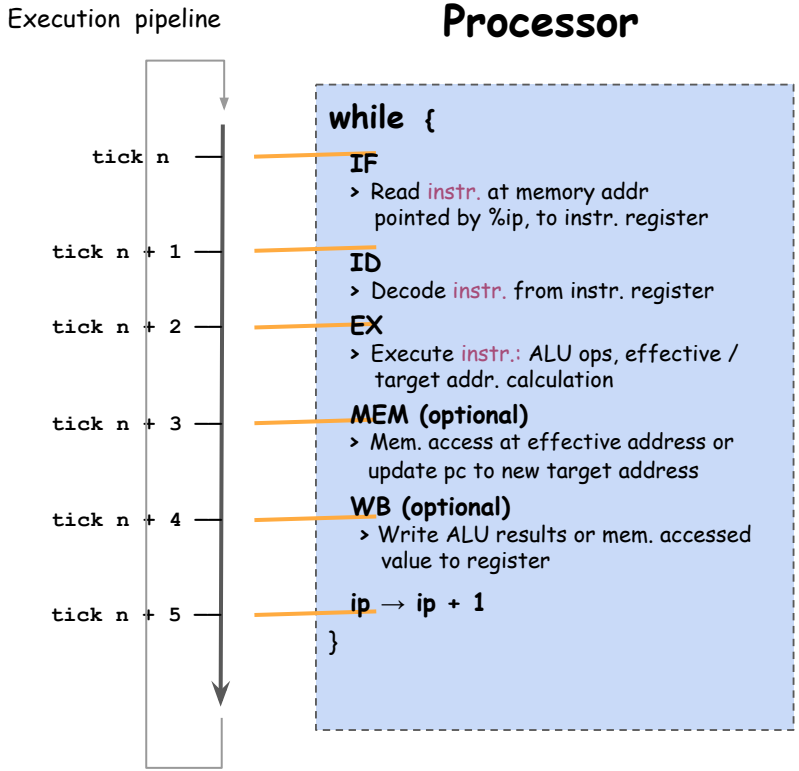
X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	Divide Error	Fault
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	
4	Overflow Exception	
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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt



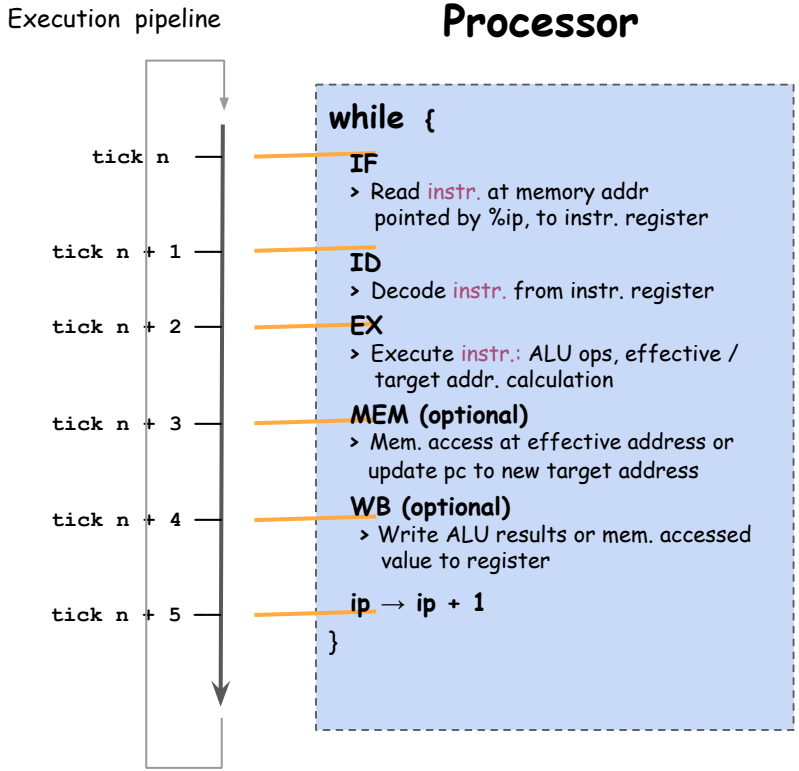
X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	Divide Error	Fault
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	Trap
4	Overflow Exception	
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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt



X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt



X86 exceptions vector

Vector NR	Exception/Interrupt Name	Type
0	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	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt

- From Intel's Software Developer's Manual, p. 3268

- Check also Tab.-7-4/5: Conditions for a Double Fault

- And see kernel/idt.c, for where the page fault exception handler is one of the first to be added

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

- The defacto mechanism for user programs to request services from the OS

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

- The defacto mechanism for user programs to request services from the OS
- Linux x86_64 implements ~380 syscalls and aarch64 ~320:

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

- The defacto mechanism for user programs to request services from the OS
- Linux x86_64 implements ~380 syscalls and aarch64 ~320:

x86/entry/syscalls/syscall_64.tbl

/ arch / x86 / entry / syscalls / syscall_64.tbl

```
60      common  exit            sys_exit
61      common  wait4           sys_wait4
62      common  kill            sys_kill
                ...
```

What synchronous events invoke the OS?

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

- The defacto mechanism for user programs to request services from the OS
- Linux x86_64 implements ~380 syscalls and aarch64 ~320:

x86/entry/syscalls/syscall_64.tbl

/ arch / x86 / entry / syscalls / syscall_64.tbl

```
60      common  exit            sys_exit
61      common  wait4           sys_wait4
62      common  kill            sys_kill
...
```

arm64/tools/syscall_64.tbl

/ arch / arm64 / tools

```
93  common  exit            sys_exit
94  common  exit_group      sys_exit_group
95  common  waitid          sys_waitid
...
```

Syscall argument passing (AArch64 example)

Syscall argument passing (AArch64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

Syscall argument passing (AArch64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}
```

Syscall argument passing (AArch64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}

int main(void) {
   __asm__ __volatile__ ("mov x0, #123");   // Syscall arg0 to reg. %x0, and so on
   __asm__ __volatile__ ("mov x8, #93");    // Syscall NR to reg. %x8 (NR 93 is exit)
   __asm__ __volatile__ ("svc  #0");        // Execute the syscall
}
```

Syscall argument passing (AArch64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}
```

```
int main(void) {
   __asm__ __volatile__ ("mov x0, #123");   // Syscall arg0 to reg. %x0, and so on
   __asm__ __volatile__ ("mov x8, #93");    // Syscall NR to reg. %x8 (NR 93 is exit)
   __asm__ __volatile__ ("svc  #0");        // Execute the syscall
}
```

```
$ ./svc_test

$ echo $?

123
```

Syscall argument passing (x86_64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}
```

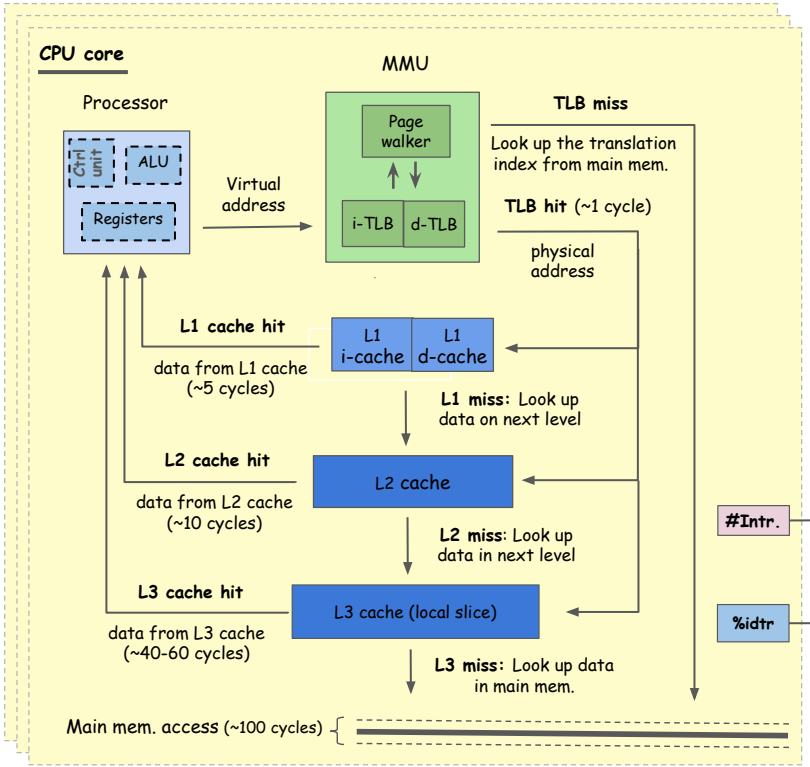
```
int main(void) {
   __asm__ __volatile__ ("mov $123, %rdi");  // Syscall arg0 to reg. %rdi, and so on
   __asm__ __volatile__ ("mov $60, %rax");   // Syscall NR to reg. %rax (NR 60 is exit)
   __asm__ __volatile__ ("syscall");         // Execute the syscall
}
```

```
$ ./syscall_test

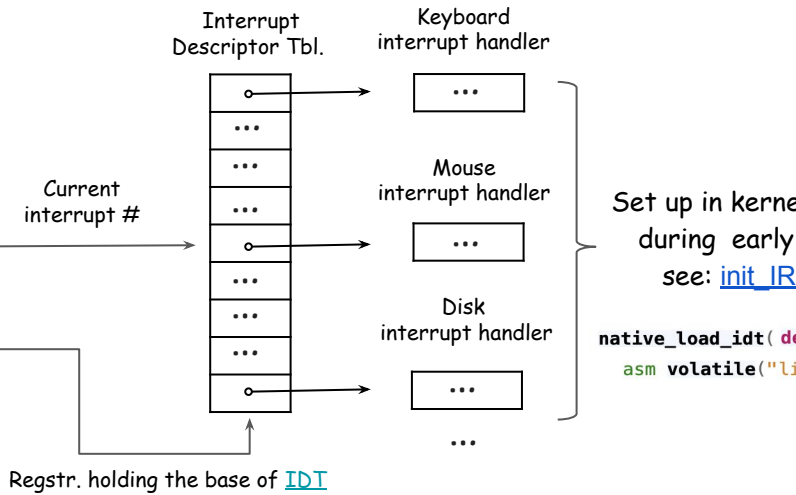
$ echo $?

123
```

Interrupt handling



do_handle_interrupt()



Set up in kernel space during early boot
see: init_IRQ()

```
native_load_idt( desc_ptr *dtr)
  asm volatile("lidt %0"::"m" (*dtr));
```

Syscall argument passing (x86_64 example)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}

int main(void) {
   __asm__ __volatile__ ("mov $123, %rdi");  // Syscall arg0 to reg. %rdi, and so on
   __asm__ __volatile__ ("mov $60, %rax");   // Syscall NR to reg. %rax (NR 60 is exit)
   __asm__ __volatile__ ("int 0x80");        // Software interrupt on vector 128
}
```

```
$ ./syscall_test

$ echo $?

123
```

How do user programs know what services an OS offers?

How do user programs know what services an OS offers?

We need a standard for that

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

- Describes a set of fundamental abstractions needed for portable application development

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

- Describes a set of fundamental abstractions needed for portable application development

- User programs can directly invoke system calls, but going through POSIX ensures portability across POSIX-compliant OSes

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

- Describes a set of fundamental abstractions needed for portable application development

- User programs can directly invoke system calls, but going through POSIX ensures portability across POSIX-compliant OSes

- Programs must not break on kernel updates

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

- Describes a set of fundamental abstractions needed for portable application development

- User programs can directly invoke system calls, but going through POSIX ensures portability across POSIX-compliant OSes

- Programs must not break on kernel updates
- OK to recompile code, if I move to another OS

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

- Describes a set of fundamental abstractions needed for portable application development

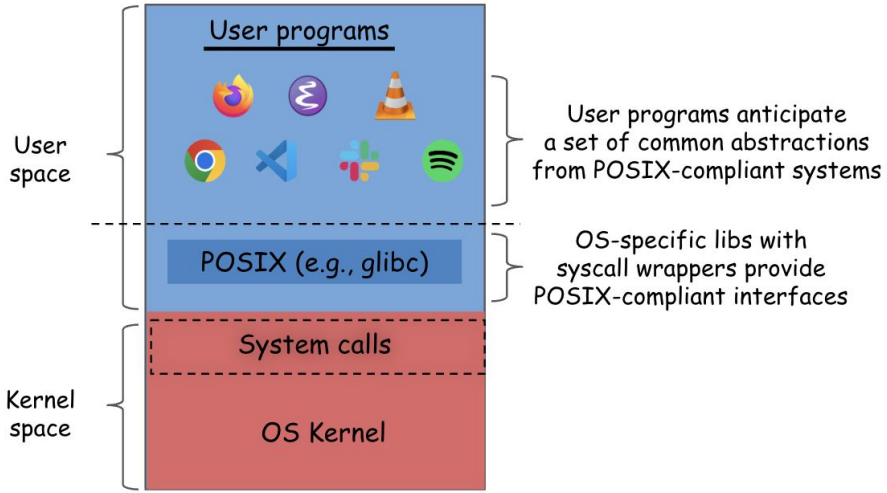
- User programs can directly invoke system calls, but going through POSIX ensures portability across POSIX-compliant OSes

- Programs must not break on kernel updates
- OK to recompile code, if I move to another OS
- But…don't make me rewrite it

How do user programs know what services an OS offers?

We need a standard for that

› Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces



What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

- ***Process***: An address space with one or more threads executing within, and the required system resources for those threads

What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

- ***Process***: An address space with one or more threads executing within, and the required system resources for those threads
- ***Thread***: A single flow of control within a process, with its own system resources required to support a flow of control

What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

- ***Process***: An address space with one or more threads executing within, and the required system resources for those threads
- ***Thread***: A single flow of control within a process, with its own system resources required to support a flow of control
- ***File***: An object that can be written to, read from, or both

What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

- ***Process***: An address space with one or more threads executing within, and the required system resources for those threads
- ***Thread***: A single flow of control within a process, with its own system resources required to support a flow of control
- ***File***: An object that can be written to, read from, or both

The complete spec with all definitions is here

What is Linux?

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***
  - **Servers**: Over 96% of the world's top 1 million websites run on Linux servers.

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***
  - **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

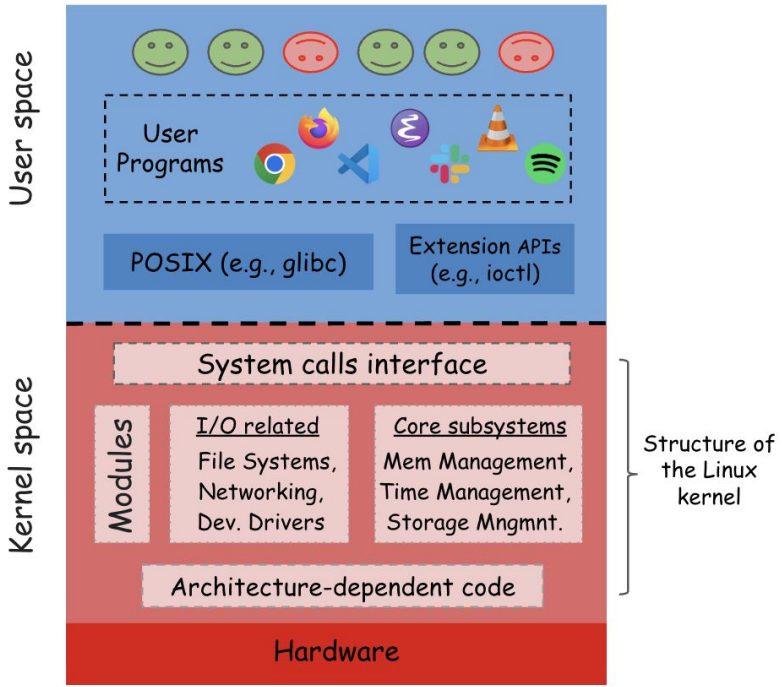
What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***
  - **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

What is Linux?

- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***
  - **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

What is Linux?



- A modern, open-source, POSIX-compliant OS kernel
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC)**
- ***The most popular Unix-like OS today***
  - **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