A practical course on

Advanced systems programming in C/Rust

Redha Gouicem

Today's topic!

Kernel and System calls

Kernel

The kernel is the core component of a computer system that:

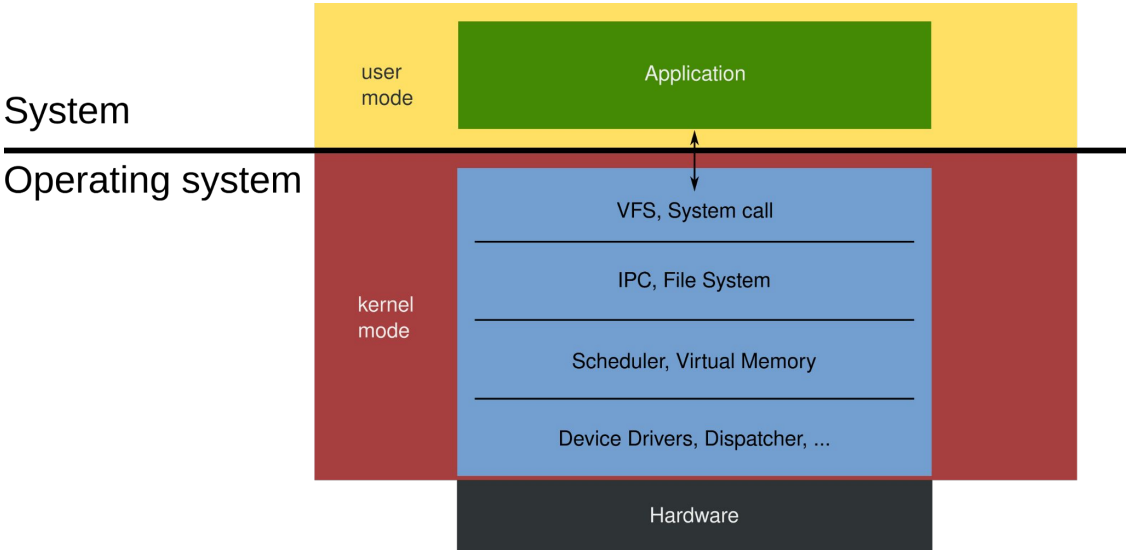
- manages resources (memory, threads, I/O devices, etc. . .)
- facilitates communication between applications (IPC)

It is executed in a privileged mode that gives it complete control over everything.

Various kernel designs define the boundary between privileged/unprivileged modes.

Monolithic Kernels

All OS services live in the kernel, in a single address space.
Single binary, modules can be added at run time.



Pros:

Performance: mode switches are relatively rare

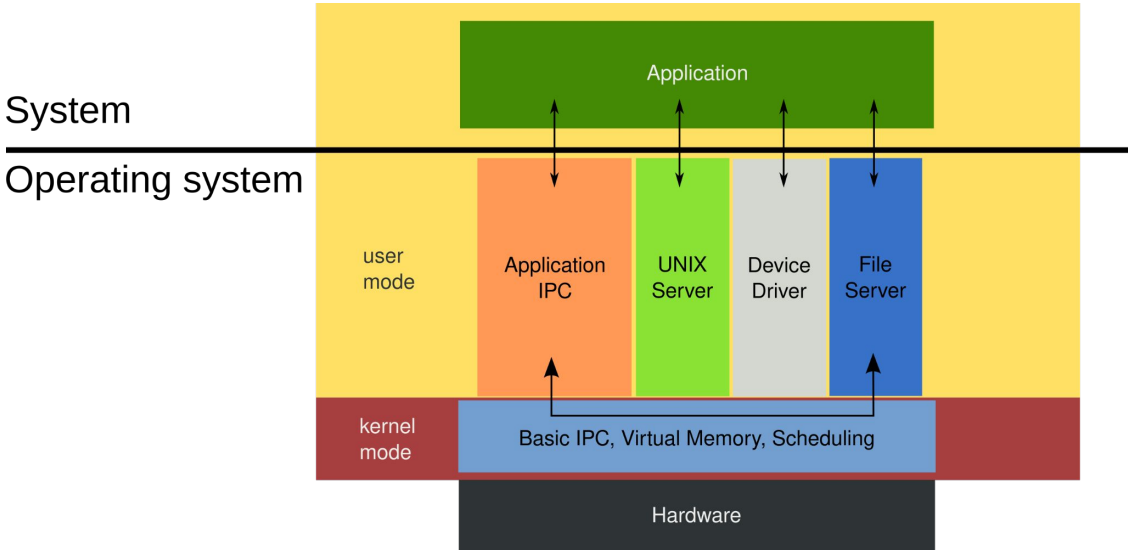
Cons:

Reliability: failure of a kernel service crashes the system

Examples: Linux, *BSD, MS-DOS, ...

Micro Kernels

Minimal services live in the kernel. Non essential services run in user mode.
If a user service crashes, it can be restarted without crashing the system.



Pros:

Reliability: services can crash and restart
Safety: easier to formalize

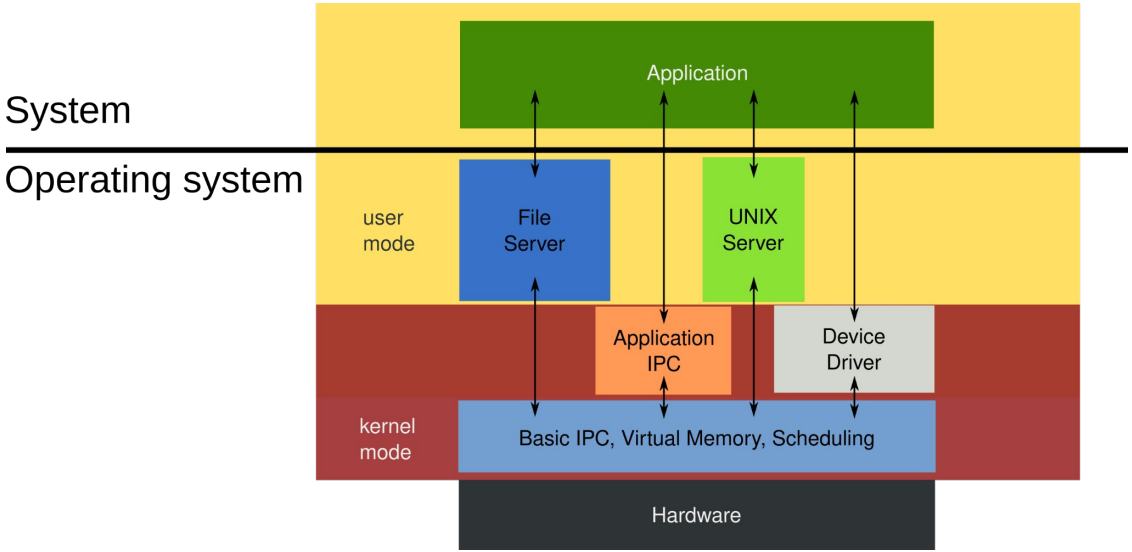
Cons:

Performance: lots of mode switches and communication

Examples: MINIX, Mach, ...

Hybrid Kernels

Between monolithic and micro kernels.
Performance-critical services live in the kernel, others run in user mode.

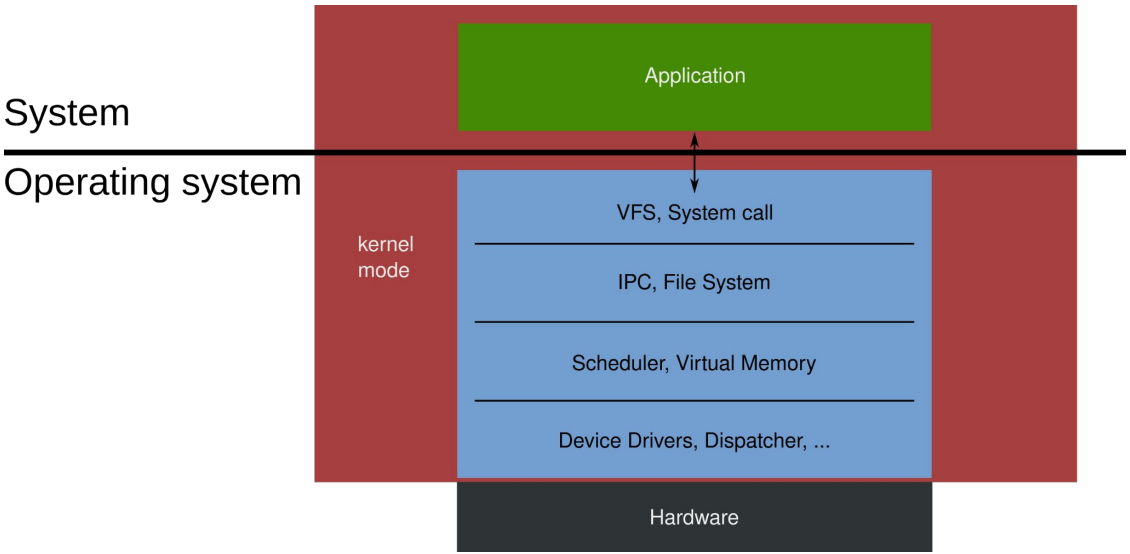


Depending on how *monolithic* or *micro* the kernel is, performance and reliability are on a spectrum between both designs.

Examples: NT, XNU, ReactOS, ...

UniKernels

Everything lives in the kernel, even applications.
Usually tailored for a particular application.



Pros:

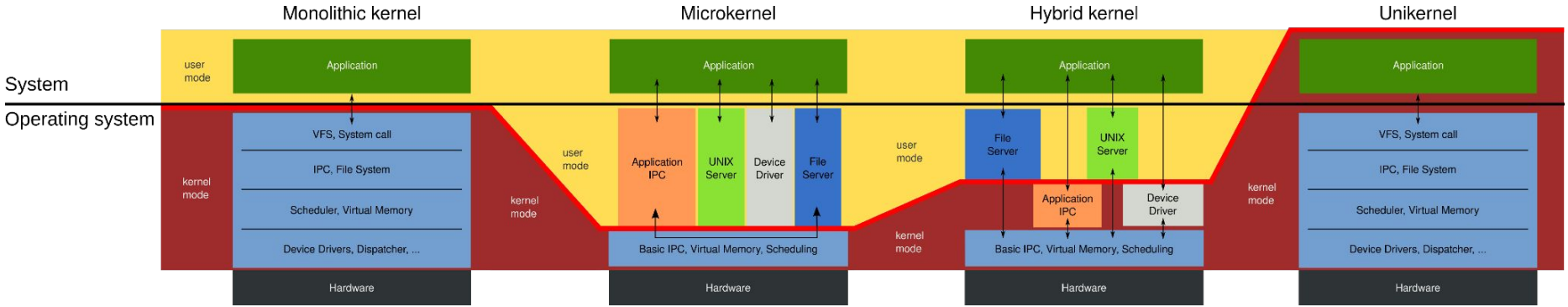
Performance: no mode switches, tailored to application.

Cons:

Usability: hard to Create, single purpose

Examples: ClickOS, MirageOS, Graphene, ...

~~Fifty~~ Four Shades of Kernel

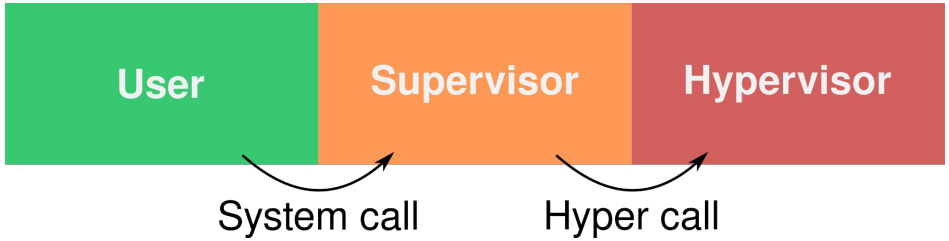


Privilege Levels

The privilege level determines what a piece of code can do on the system.

- **User mode**: Restricted access to hardware
  => Used by applications
- **Supervisor mode**: Access to privileged instructions, registers, management hardware
  => Used by the kernel
- **Hypervisor mode**: Access to virtualization-specific instructions
  => Used by hypervisors

Switching between privilege requires specific instructions.



System Calls

System calls allow applications to execute privileged code to perform certain tasks:

- I/Os (disk, network)
- Resource management (threads, memory)
- Communication (signals, IPCs)
- Access to specific hardware

They are the API of the kernel for user applications.
We'll only focus on **Linux** system calls here.

System Calls Wrappers

Programmers rarely directly use system calls. They usually use `libc` functions that perform system calls.

Example: The `ssize_t write(int, void*, size_t)` function is **not** an actual system call, but a wrapper from the `libc`.

Wrappers can make some argument checks and error handling (i.e. `errno`).

Some wrappers may do more, e.g. memory allocation can reuse previously freed memory without going through the kernel.

Making a System Call: The Interrupt Way

System calls are just a software interrupt:

1. Place the system call number in a register

eax := SYS_read

Making a System Call: The Interrupt Way

System calls are just a software interrupt:

1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)

```
eax := SYS_read
<place args>
```

Making a System Call: The Interrupt Way

System calls are just a software interrupt:

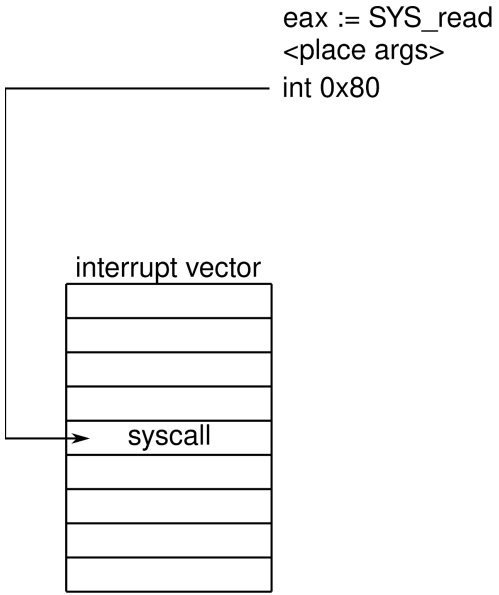
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Trigger the "system call" interrupt

```
eax := SYS_read
<place args>
int 0x80
```

Making a System Call: The Interrupt Way

System calls are just a software interrupt:

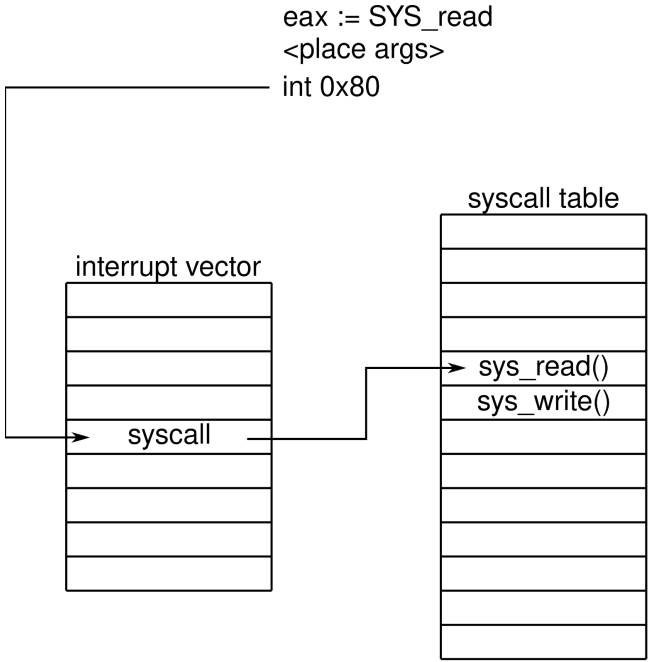
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Trigger the "system call" interrupt
4. Jump to the correct interrupt handler (the one for "system call")



Making a System Call: The Interrupt Way

System calls are just a software interrupt:

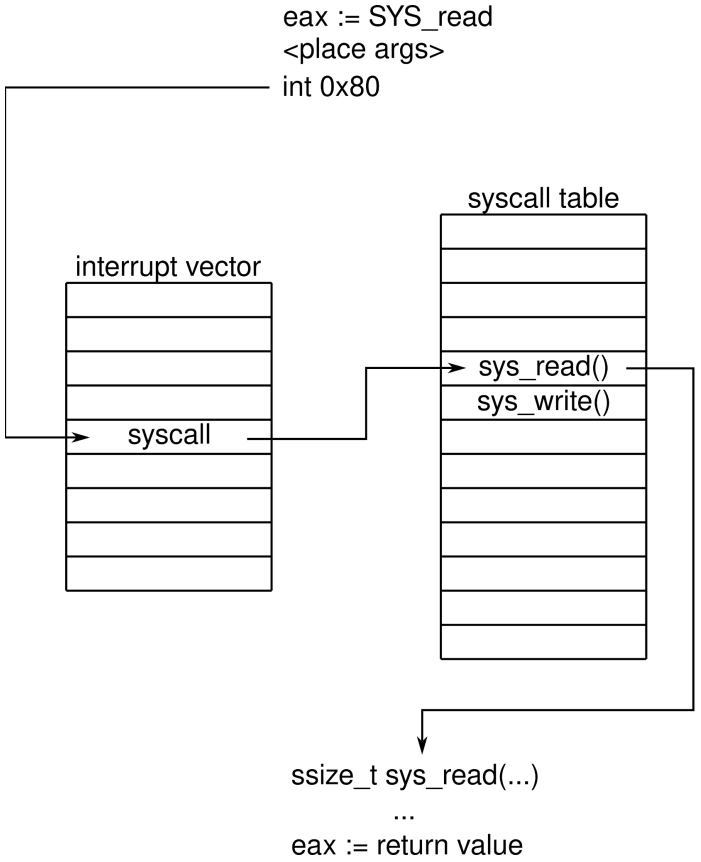
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Trigger the "system call" interrupt
4. Jump to the correct interrupt handler (the one for "system call")
5. Load the system call table and jump to the index given by the system call number



Making a System Call: The Interrupt Way

System calls are just a software interrupt:

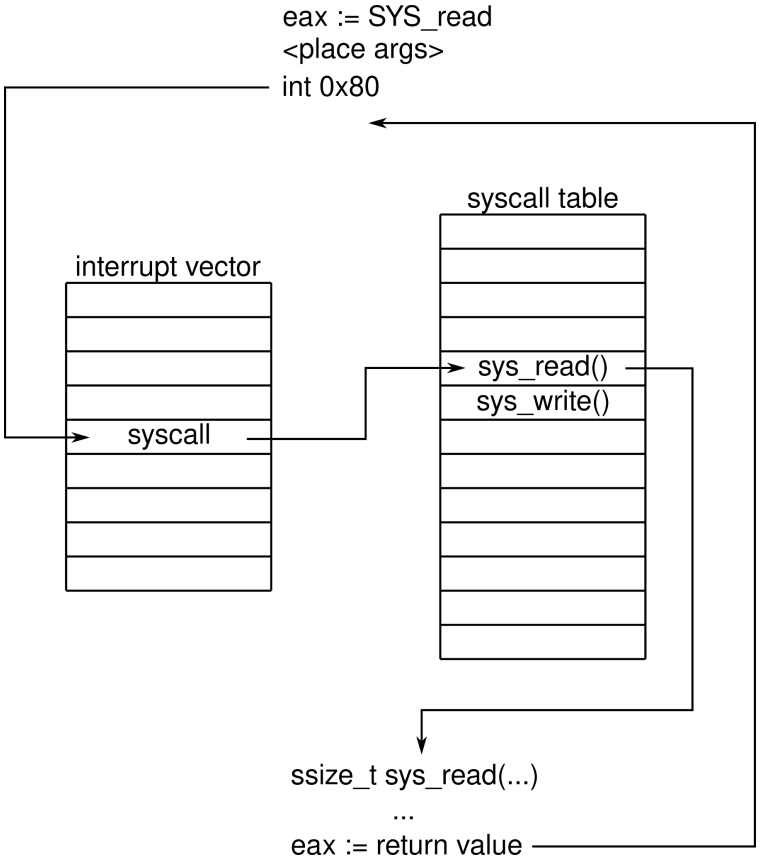
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Trigger the "system call" interrupt
4. Jump to the correct interrupt handler (the one for "system call")
5. Load the system call table and jump to the index given by the system call number
6. Perform the system call function



Making a System Call: The Interrupt Way

System calls are just a software interrupt:

1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Trigger the "system call" interrupt
4. Jump to the correct interrupt handler (the one for "system call")
5. Load the system call table and jump to the index given by the system call number
6. Perform the system call function
7. Return the result to user space



Making a System Call: The Instruction Way

eax := SYS_read

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

1. Place the system call number in a register

Making a System Call: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)

```
eax := SYS_read
<place args>
```

Making a System Call: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

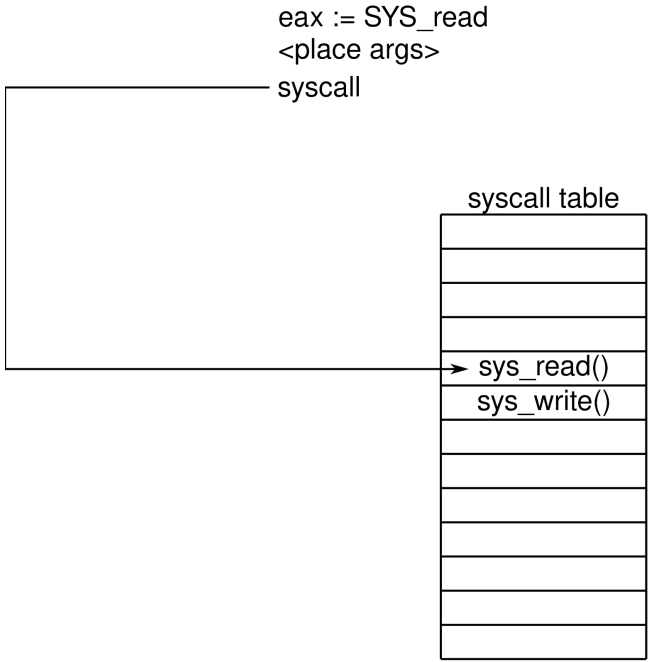
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Use the "system call" instruction

```
eax := SYS_read
<place args>
syscall
```

Making a System Call: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

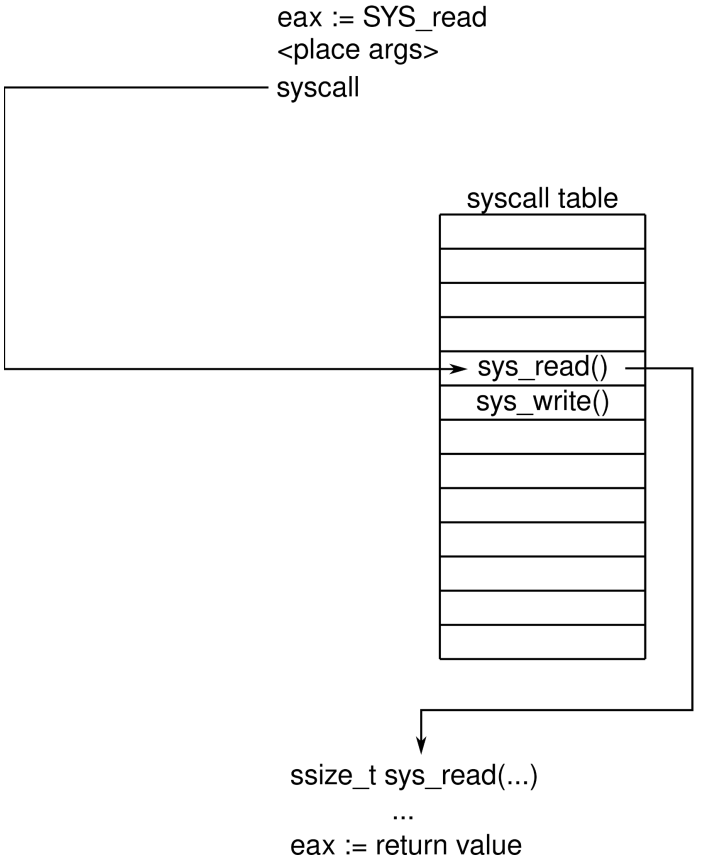
1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Use the "system call" instruction
4. Jump to the index given by the system call number in the system call table



Making a System Call: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Use the "system call" instruction
4. Jump to the index given by the system call number in the system call table
5. Perform the system call function

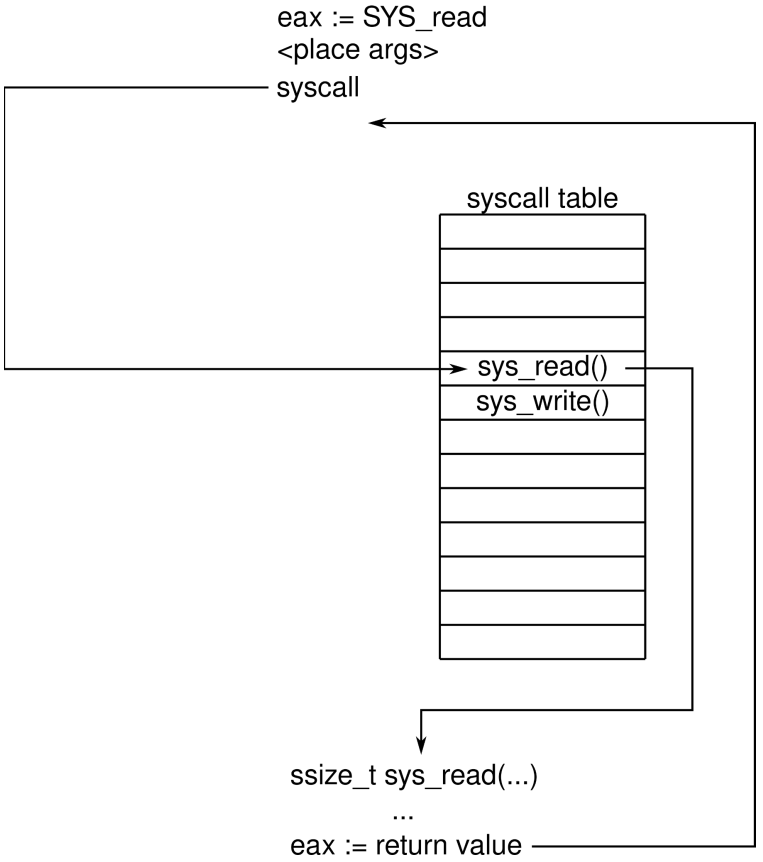


Making a System Call: The Instruction Way

Some architectures provide a specific instruction (`syscall`, `svc`, `sysenter`, …):

1. Place the system call number in a register
2. Place arguments according to ABI (we'll see that later on)
3. Use the "system call" instruction
4. Jump to the index given by the system call number in the system call table
5. Perform the system call function
6. Return the result to user space

One level of indirection is bypassed.



Application Binary Interface

API: High-level interface for programmers (function prototypes, data types, …)

ABI: Low-level interface for compilers/OS (calling conventions, architecture-specific)

Arch	syscall#	retval	arg1	arg2	arg3	arg4	arg5	arg6	arg7
Arm EABI	r7	r0	r0	r1	r2	r3	r4	r5	r6
arm64	w8	x0	x0	x1	x2	x3	x4	x5	-
mips	v0	v0	a1	a1	a2	a3	a4	a5	-
riscv	a7	a0	a0	a1	a2	a3	a4	a5	-
x86-64	rax	rax	rdi	rsi	rdx	r10	r8	r9	-

Note: Linux allows at most 6 arguments for system calls. More examples at `man syscall`.

Note 2: Conforming to the ABI is the job of the `long syscall(long nr, ...)` function from the `libc`.

This Week's Tasks

Training exercises:

- Invoke a system call with the syscall libc function
- Invoke a system call in assembly

Main exercise:

- Implement a system call tracer

Going further (not graded):

- Implement a new system call in the Linux kernel

Hijacking System Calls (sort of …)

For the training exercises

System call wrappers (libc) are usually provided through a shared library.

```
redha@tum:~$ ldd /usr/bin/echo
linux-vdso.so.1 (0x00007ffd71f08000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f1af627a000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f1af6475000)
redha@tum:~$
```

The addresses of these functions are resolved at runtime by the linker (`ld-linux-x86-64.so.2`).

You can override a system shared library with your version with the `LD_PRELOAD` environment variable.

Tracing System Calls with `strace`

For the main exercise

`strace` is a tool used to trace system calls and signals. It relies on the `ptrace()` system call.

```
redha@tum:~$ strace -e execve,brk,open,read,write echo foo bar > /tmp/toto
execve("/usr/bin/echo", ["echo", "foo", "bar"], 0x7fff7460ee10 /* 48 vars */) = 0
brk(NULL)                               = 0x55d47d60c000
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`|\2\0\0\0\0\0"..., 832) = 832
brk(NULL)                               = 0x55d47d60c000
brk(0x55d47d62d000)                     = 0x55d47d62d000
write(1, "foo bar\n", 8)                = 8
+++ exited with 0 +++

redha@tum:~$
```

See you at the Q&A on Thursday!