🧩 System Calls in Linux and How They Work

In Linux (and Unix-like systems), **system calls** are the **primary interface between user-space applications and the kernel**. When a program wants to perform a privileged operation—like reading a file, allocating memory, or creating a process—it uses a system call.

🔗 User Space vs Kernel Space

- **User Space**: Where applications like editors, browsers, and your shell run.
- **Kernel Space**: Privileged mode with direct access to hardware and core system resources.

System calls act as the **gateway between these two spaces**. Direct access is not allowed for security and stability reasons.

🧮 Registers Used in x86-64 Syscalls

Purpose	Register
Syscall number	`rax`
arg1	`rdi`
arg2	`rsi`
arg3	`rdx`
arg4	`r10`
arg5	`r8`

Purpose	Register
arg6	`r9`
Return value	`rax`

🧪 Example: Raw syscall in Assembly (x86-64)

```
section .data
    msg db "Hello, world!", 0xA
    len equ $ - msg

section .text
    global _start

_start:
    mov rax, 1          ; syscall number for write
    mov rdi, 1          ; file descriptor: stdout
    mov rsi, msg        ; message to write
    mov rdx, len        ; message length
    syscall             ; make the syscall

    mov rax, 60         ; syscall: exit
    xor rdi, rdi        ; exit code 0
    syscall
```

Assembled and run with:

```
nasm -f elf64 hello.asm && ld -o hello hello.o && ./hello
```

🔐 System Call Filtering: seccomp

Linux provides **seccomp** (Secure Computing Mode) to restrict the system calls a process can make.

Example: Block everything except `read`, `write`, and `exit`.

Used in:

- Containers (e.g., Docker)
- Sandboxing tools (e.g., Firejail)
- Browser and VM isolation

🧠 Bonus: System Call vs Function Call

Feature	**Function Call**	**System Call**
Context Switch	No	Yes (user → kernel → user)
Scope	Within the process	Into the kernel
Overhead	Low	High
Example	`strlen("abc")`	`write(1, "abc", 3)`