System Calls in Novix OS

So far, all our code has run in **kernel mode** — the most privileged level on RISC-V.
But real operating systems don't execute user programs directly inside the kernel.

Instead, user programs run in **user mode**, where they are isolated and safe.
When they need a service (like printing text, reading input, or allocating memory), they perform a **system call (syscall)** — a controlled jump into the kernel.

This chapter introduces **syscalls** and builds a small **user library (libuser)** that lets user programs call the kernel safely.

Learning Goals

By the end of this section, you'll understand:

- How syscalls connect **user space <==> kernel space**
- How RISC-V uses the `ecall` instruction
- How the kernel dispatches and handles syscalls
- How user programs use wrappers like `read()`, `write()`, and `printf()`

What Are System Calls?

System calls (or *syscalls*) are the **bridge between user programs and the operating system**.

Instead of directly accessing hardware or memory, a user program requests services from the kernel — for example:

Function	Syscall	Description
`write()`	64	Write data to stdout
`read()`	63	Read input from stdin
`cls()`	406	Clear the screen (custom Novix syscall)

When the user program executes `ecall`, the CPU traps into the kernel's **trap handler**, which recognizes that it's a syscall and jumps to the appropriate kernel function.

The Novix Syscall Interface

Let's start in **userland** with the code that invokes syscalls.

`user/lib/syscall.c`

This small library defines a generic syscall wrapper using inline assembly:

```
int64_t syscall(int64_t sysno, int64_t arg0, int64_t arg1, int64_t arg2) {
    register int64_t a0 __asm__("a0") = arg0;
    register int64_t a1 __asm__("a1") = arg1;
    register int64_t a2 __asm__("a2") = arg2;
    register int64_t a7 __asm__("a7") = sysno;

    __asm__ __volatile__("ecall" : "+r"(a0)
                         : "r"(a1), "r"(a2), "r"(a7)
                         : "memory");

    return a0;
}
```

The key instruction is `ecall`, which signals to the CPU:

```
"Jump to supervisor mode and let the kernel handle this syscall!"
```

After the kernel finishes, it places the result back in register `a0`.

Helper Wrappers

To make the interface easier to use, Novix defines thin wrappers:

```
ssize_t read(int fd, void *buf, size_t count) {
    return syscall(SYS_READ, (int64_t)fd, (int64_t)buf, (int64_t)count);
}

int write(int fd, const void *buf, size_t len) {
    return (int)syscall(SYS_WRITE, fd, (intptr_t)buf, (intptr_t)len);
}

void cls() {
    syscall(SYS_CLEAR, 0, 0, 0);
}
```

Now, user programs can simply call:

```
write(1, "Hello, World!\n", 14);
cls();
```

without worrying about the low-level `ecall` instruction.

Defining Syscall Numbers

The syscall numbers follow the **POSIX/Linux RISC-V convention**, with a few custom ones added by Novix:

```
#define SYS_READ      63
#define SYS_WRITE     64
#define SYS_EXIT      93
#define SYS_YIELD     124
#define SYS_GETPID    172
#define SYS_CLEAR     406   // Novix custom: clear the screen
#define SYS_PUTCHAR   400   // Custom putchar
```

By using the same numbering scheme as Linux, we make future compatibility easier — and our kernel code more intuitive.

Kernel-Side: Handling Syscalls

Now, let's look inside the kernel implementation in `kernel/syscall.c`.

Dispatcher Function

All syscalls are handled by a single function:

```
ssize_t sys_read(int fd, char *buf, size_t len) {
    // Enable SUM
    enable_sum();
    if (fd != 0 || buf == NULL || len == 0) {
        return -1;
    }

    size_t i = 0;
    while (i < len) {
```

```
        long c = uart_getc();
        if (c == -1) continue; // busy-wait tot input
        buf[i++] = (char)c;
        if (c == '\n') break;
        yield();
    }

    // Restore SUM state
    disable_sum();
    return i;
}

int sys_write(uint64_t fd, uint64_t user_buf, uint64_t len) {
    if (fd != 1) return -1;

    // Enable SUM
   enable_sum();

    char *buf = (char *)user_buf;

    for (uint64_t i = 0; i < len; i++) {
        char c = buf[i];  // direct access
        uart_putc(c);
    }

    // Restore SUM state
    disable_sum();

    return len;
}

void handle_syscall(struct trap_frame *f) {
    switch (f->regs.a7) {
        case SYS_READ:
            f->regs.a0 = sys_read((int)f->regs.a0, (char *)f->regs.a1, (size_t)f->regs.a2);
            break;

        case SYS_WRITE:
            f->regs.a0 = sys_write(f->regs.a0, f->regs.a1, f->regs.a2);
            break;

        case SYS_CLEAR:
            uart_cls();
            f->regs.a0 = 0;
            break;

        default:
            PANIC("[syscall] unknown syscall %d\n", f->regs.a7);
    }
}
```

Here's what happens step-by-step:

1. The CPU traps to supervisor mode via `ecall`.

2. The kernel saves all registers into the trap frame `f`.

3. The dispatcher reads `a7` (the syscall number).
4. It routes the call to the correct kernel function.
5. The return value is placed in `a0`, which goes back to userland.

Memory Access: SUM Bit

Syscalls often need to read or write user memory.
RISC-V protects supervisor mode from directly accessing user memory unless the **SUM bit** is enabled.

Novix provides two helpers:

```
void enable_sum(void) {
    uint64_t s = READ_CSR(sstatus);
    WRITE_CSR(sstatus, s | SSTATUS_SUM);
}

void disable_sum(void) {
    uint64_t s = READ_CSR(sstatus);
    WRITE_CSR(sstatus, s & ~SSTATUS_SUM);
}
```

These are used internally by `sys_read()` or `sys_write()` to safely copy user data.

Building a User Library (Bare Metal Style)

Before writing complex programs, userland needs a few basic utilities.
Just like the kernel, our `libuser` implements:

- `printf()`, `puts()`, and `putc()`
- String manipulation functions (`strlen`, etc.)
- A minimal `syscall()` interface

`user/lib/printf.c`

This is a lightweight version of the kernel `printf`, but it uses syscalls instead of UART registers.

```
void putc(char ch) {
    syscall(SYS_PUTCHAR, ch, 0, 0);
}

void puts(const char *buf) {
    write(1, buf, strlen(buf));
}
```

And of course, it includes a small but capable `printf()` implementation supporting:

- `%d`, `%u`, `%x`, `%s`, `%c`, `%p`
- Zero-padding (`%05d`)
- Field width modifiers

Hello from Userland

With our library ready, let's test it using `user/shell.c`:

```c
#include "include/stdio.h"
#include "include/syscall.h"

void main(void) {
    cls();
    puts("\nHello world from userland! (shell.elf)\n");

    puts("Input test.\n");
    puts("Enter your name : ");

    int c;
    while (1) {
        int n = read(0, &c, 1);  // read 1 byte
        if (n == 1) {
            putc(c);             // echo to stdout
        }
    }
}
```

When you run this, you'll see:

```
Hello world from userland! (shell.elf)
Input test.
Enter your name :
```

And as you type characters in QEMU, they're echoed back — by **userland code**, not the kernel!

Summary

In this first part of Userland development, we've built:

- The syscall interface connecting user mode to the kernel
- Kernel dispatcher that routes requests to system functions
- Userland library (`libuser`) with basic I/O and printing
- Our first "Hello from userland" program

This is a **major step** in any OS project — the moment your operating system starts running programs *outside* the kernel.

A smile on my face — because Novix OS has officially entered Userland!