Scheduling and Context Switching in Novix OS

In the last chapter, we built the process layer and taught the kernel to load ELF programs such as `idle.elf` and `shell.elf`.
But until now, these processes were *static* — only one could run at a time.

Now it's time to bring your operating system to life with **scheduling** and **context switching**, allowing Novix to **run multiple processes** and switch between them seamlessly.

Learning Goals

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

- What **context switching** means on RISC-V
- How Novix implements a minimal **scheduler**
- How registers and stacks are saved and restored
- How `yield()` switches between the **idle** and **shell** processes
- How the kernel transitions from **booting** to **multitasking**

The Big Picture

When Novix starts, it creates two main user processes:

Process	PID	Purpose
`idle`	0	The background process (runs when nothing else is runnable)
`shell`	2	The user process that simulates activity or the command shell

Both are loaded from ELF binaries by the kernel during startup.
The scheduler's job is to **decide which process runs next** and **save/restore** their CPU state.

What Is a Context Switch?

A *context switch* happens when the CPU stops running one process and starts running another.
The kernel must save the **registers** of the current process and restore those of the next process.

On RISC-V, we use 13 **callee-saved registers** (from `s0`–`s11` and `ra`), plus the **stack pointer** (`sp`).

Low-Level Context Switching

Defined in `kernel/context.c`, the function `switch_context_sp()` performs the actual register save/restore operation:

```
__attribute__((naked)) void switch_context_sp(uint64_t *prev_sp,
                                              uint64_t *next_sp) {
    __asm__ __volatile__(
        // Save callee-saved registers
        "addi sp, sp, -13 * 8\n"
        "sd ra,  0  * 8(sp)\n"
        "sd s0,  1  * 8(sp)\n"
        "sd s1,  2  * 8(sp)\n"
        "sd s2,  3  * 8(sp)\n"
        ...
```

```
        "sd s11, 12 * 8(sp)\n"

        // Save current stack pointer
        "sd sp, 0(a0)\n"          // *prev_sp = sp

        // Load new stack pointer
        "ld sp, 0(a1)\n"          // sp = *next_sp

        // Restore next process registers
        "ld ra,  0  * 8(sp)\n"
        "ld s0,  1  * 8(sp)\n"
        ...
        "ld s11, 12 * 8(sp)\n"
        "addi sp, sp, 13 * 8\n"
        "ret\n"
    );
}
```

This tiny piece of assembly is the *heart* of multitasking.
It stores all the registers of the current process, updates both stack pointers, and restores the next process's state — all without touching C code.

The Scheduler (Round-Robin Style)

The function `yield()` in `kernel/scheduler.c` is the Novix scheduler.
It runs in a loop, deciding which process should execute next.

Step-by-Step Breakdown

1. **Save Current Process State**

```
struct process *prev = current_proc;
struct process *next = NULL;
if (current_proc->state == PROC_RUNNING)
    current_proc->state = PROC_RUNNABLE;
```

2. **Select the Next Process**

The scheduler scans the `procs[]` array to find a process in state `PROC_RUNNABLE`.

```
for (int i = 1; i < PROCS_MAX; i++) {
    proc_t *p = &procs[i];
    if (p->state == PROC_RUNNABLE && p != current_proc) {
        next = p;
        break;
    }
}
```

If none are available, the system runs the **idle process**.

Step 3. Switch Page Tables

When switching processes, the kernel must also change the **address space** (page table):

```
__asm__ __volatile__(
    "sfence.vma\n"
    "csrw satp, %[satp]\n"
    "sfence.vma\n"
    "csrw sscratch, %[sscratch]\n"
    :
```

```
    : [satp] "r" (SATP_SV39 | ((uint64_t) next->page_table / PAGE_SIZE)),
      [sscratch] "r" ((uint64_t) &next->tf_stack[sizeof(next->tf_stack)])
);
```

This ensures the new process runs in *its own* virtual memory space.

Step 4. Perform the Context Switch

Finally, the kernel swaps stacks using our assembly routine:

```
switch_context_sp(&prev->sp, &next->sp);
```

At this point, the CPU registers, stack, and page tables all belong to the next process — the switch is complete.

Startup Flow

When Novix first calls `yield()` from `kernel_main()`, the scheduler detects this is the very first run.
The second calls to `yield`, the context switch is done bij `user_return()` using `trapframe` :

```
if (g_startup) {
    g_startup = 0;
    switch_context_sp(&prev->sp, &next->sp);
} else {
    user_return();
}
```

After this initial switch, the system alternates between `idle` and `shell`, forming the foundation for **multitasking**.

Idle and Shell Processes

- **Idle Process (`idle.elf`)**
  Runs when no other process is active. It usually contains an infinite loop doing nothing.
- **Shell Process (`shell.elf`)**
  In this minimal version, it's also an infinite loop — but in the future, this will become an interactive user shell.

```
void main(void) {
    for (;;);
}
```

Even though both do "nothing," they alternate in memory — demonstrating real **context switching** at work!

Putting It Together: kernel_main()

Here's the complete boot sequence from `kernel/kernel.c`:

```
void kernel_main(void) {
    bss_init();
    uart_cls();
    uart_printf("Novix RISC-V 64 OS, Version 0.0.1\n\n");

    trap_init();
    stack_init();
    ram_init();
    heap_init();
    paging_init();

    date_time_test();
    virtio_bus_init_scan();
```

```
    // Load and create idle process
    size_t fs;
    void *idle_elf_file = tarfs_lookup("idle.elf", &fs, 1);
    idle_proc = extract_flat_binary_from_elf(idle_elf_file, CREATE_PROCESS);
    idle_proc->pid = 0;
    strcpy(idle_proc->name, "idle");
    current_proc = idle_proc;

    // Load and create shell process
    void *shell_file = tarfs_lookup("shell.elf", &fs, 1);
    struct process *s = extract_flat_binary_from_elf(shell_file, CREATE_PROCESS);
    strcpy(s->name, "shell");

    LOG_INFO("Start scheduler ...");
    while (1) {
        yield();
    }

    system_halt();
}
```

At this point, the OS alternates between **idle** and **shell** forever, showing true process scheduling.

Summary

By now, you've implemented the *core heartbeat* of Novix OS:

- Low-level context switching using inline RISC-V assembly
- A working round-robin scheduler (`yield()`)
- Support for multiple user processes
- Integration with the ELF loader and process system

This is a huge step — your OS can now **multitask**, swapping between processes, each with its own memory and CPU state.

`*A smile on my face - because Novix OS now truly feels alive!*`