Back to the Kernel and the Shell

After exploring **Inter-Process Communication (IPC)** and building servers like `window_server` and `test_shell`, it's time to return to our **core kernel + shell setup**.

This step prepares the system for the **next phase of the course (Part 2)**, where we'll continue developing Novix OS features — starting again from a clean and stable baseline.

Learning Goals

By the end of this short chapter, you will:

- Understand how to switch back from the *server-based boot sequence* to the *classic kernel + shell* configuration
- Learn which parts of the kernel to modify
- Restore the original scheduler logic for `idle` and `shell` processes
- Boot back into a fully working **Novix shell** environment

1. Returning from Servers to the Shell

In the last chapters, our kernel booted the **init process**, which in turn started multiple servers, such as the window server and test shell — to demonstrate **IPC**.

Now we'll temporarily **disable the init process** and **load the shell directly** again, just like before the IPC chapters.

2. Kernel Boot Code

Let's look at how the kernel is modified to load the **shell** instead of the **init server**.

kernel/kernel.c

```c
pagetable_t kernel_pagetable;

struct process *current_proc;   // Currently running process
struct process *idle_proc;      // Idle process
int process_count = 0;          // Number of created processes

void kernel_main(void) {

    // clear bss
    bss_init();

    // clear screen, welcome message
    uart_cls();
    uart_printf("Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1\n\n");
    uart_puts("Booting ...\n");

    // initialize trap_vector
    trap_init();

    // initialize stack, ram & heap
    stack_init();
    ram_init();
    heap_init();
    // initialize paging
    paging_init();

    // testing date & time
    date_time_test();

    // initialize virtio
    virtio_bus_init_scan();

    // load idle process
    size_t fs;
    void *idle_elf_file = tarfs_lookup("idle.elf", &fs, 1);
    if (!idle_elf_file) {
        PANIC("[kernel_main] idle.elf not found!");
    }

    // create idle process
    idle_proc = extract_flat_binary_from_elf(idle_elf_file, CREATE_PROCESS);
    idle_proc->pid = 0; // idle
    strcpy(idle_proc->name, "idle");
    current_proc = idle_proc;
    LOG_INFO("[kernel_main] idle.elf loaded.");

    // load shell process
    void *shell_file = tarfs_lookup("shell.elf", &fs, 1);
    if (!shell_file) {
        PANIC("[kernel_main] shell.elf not found!\n");
    }

    // create shell process
    struct process *s = extract_flat_binary_from_elf(shell_file, CREATE_PROCESS);
    strcpy(s->name, "shell");
    LOG_INFO("[kernel_main] shell.elf loaded.");

/*
    // load init process (DISABLED)
    void *init_file = tarfs_lookup("init.elf", &fs, 1);
    if (!init_file) {
        PANIC("[kernel_main] init.elf not found!\n");
    }

    struct process *s = extract_flat_binary_from_elf(init_file, CREATE_PROCESS);
    strcpy(s->name, "init");
    LOG_INFO("[kernel_main] init.elf loaded.");
*/

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

    system_halt();  // Should never be reached
}
```

Explanation:

- The kernel now loads only **two processes**:
  1. `idle.elf` (PID 0)
  2. `shell.elf` (PID 1)
- The **init process** (which previously loaded multiple servers) is commented out.

- This ensures the system boots directly into the shell for testing and interaction.

3. Simplified Scheduler for Kernel → Shell

When running only two processes (`idle` and `shell`), we can simplify the **scheduler** logic to ensure smooth switching between them.

`kernel/scheduler.c`

```
#include "process.h"
#include "context.h"
#include "regs.h"
#include "uart.h"
#include "user.h"
#include "riscv.h"
#include "debug.h"

extern struct process *current_proc;
extern struct process *idle_proc;
extern struct process procs[];
extern int process_count;
int g_startup = 1; // startup flag

// Note: Only using for KERNEL >> SHELL
void yield(void) {
    LOG_USER_INFO("[yield] <<<<< YIELD >>>>>");

    struct process *prev = current_proc;
    struct process *next = NULL;

    // look for another RUNNABLE process
    print_process_table();
    if (current_proc->state == PROC_RUNNING)
        current_proc->state = PROC_RUNNABLE;

    for (int i = 1; i < PROCS_MAX; i++) {
        proc_t *p = &procs[i];

        if (p == NULL || p->pid == 0) continue;

        // only idle & shell
        if (p->state == PROC_RUNNABLE && process_count == 2) {
            next = p;
            break;
        }

        if (p->state == PROC_RUNNABLE && p != current_proc) {
            next = p;
            break;
        }
    }
    dump_trap_frame(current_proc->tf);

    // Default to idle process if no other process found
    if (next == NULL) {
        next = idle_proc;
    }

    if (next == current_proc)
        return;

    if (prev->state == PROC_RUNNING)
        prev->state = PROC_RUNNABLE;

    if (next->state == PROC_RUNNABLE)
        next->state = PROC_RUNNING;

    // Switch address space
    __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)])
    );

    current_proc = next;

    LOG_USER_INFO("[scheduler] switching to pid=%d tf@0x%x sp=0x%x epc=0x%x",
                  next->pid, (uint64_t)next->tf, next->tf->sp, next->tf->epc);

    if (g_startup) {
        g_startup = 0;
        LOG_USER_INFO("[scheduler] <<< SWITCH CONTEXT SP >>>");
        dump_trap_frame(next->tf);
        switch_context_sp(&prev->sp, &next->sp);
    } else {
        LOG_USER_INFO("[scheduler] <<< USER RETURN >>>");
        dump_trap_frame(current_proc->tf);
        print_process_table();
        user_return();
    }
}
```

Explanation:

- The scheduler cycles only between **idle** and **shell**.
- The logic for handling multiple servers is commented out for now.
- This simplified version is ideal for single-user testing and shell interaction.

4. Testing the Shell Boot

When you rebuild and boot Novix now, you'll see:

```
Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1

Booting ...
[kernel_main] idle.elf loaded.
[kernel_main] shell.elf loaded.
Start scheduler ...
$
```

Success!
The shell prompt (`$`) confirms that:

- The kernel successfully switched context to the shell process.
- The IPC/Server logic is temporarily disabled.
- You're back to a simple, interactive user-space environment.

Summary

Component	Purpose
`kernel_main()`	Loads only `idle` and `shell` processes
`yield()` (simple)	Switches between two runnable processes
`init.elf` (disabled)	Temporarily bypassed to simplify testing
Shell prompt	Confirms working kernel-to-shell loop

You've now **restored a clean working shell environment**, preparing Novix OS for the **next major development stage** in Part 2.

    A smile on my face - because Novix OS is ready for Part 2 of this course!