Processes and ELF Loading in Novix OS

In the previous part, we built the *foundation* of process management: the **Process Control Block (PCB)**, memory mapping, and user-space setup.
Now, it's time to load *real executable files* — specifically, **ELF binaries** — and turn them into living processes.

Learning Goals

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

- How ELF files are structured (headers, sections, and segments)
- How Novix extracts flat binaries from ELF images
- How the OS locates key symbols like `__user_stack_top`
- The full boot flow: from **kernel_main() → ELF loader → process creation**
- Why ELF is used universally for programs and kernels

What Is an ELF File?

ELF stands for **Executable and Linkable Format** — it's the standard binary format used by Linux, BSD, and now… Novix OS!

Each ELF file contains: - A **header** that identifies the file and architecture
- One or more **program headers** describing segments to load into memory
- Optional **section headers** with metadata and symbols

When Novix boots and loads something like `idle.elf`, it uses these structures to copy the program code, stack, and heap into memory.

ELF Header Overview

The ELF header is defined as follows in `include/elf-loader.h`:

```
typedef struct {
    uint32_t e_magic;   // "ELF" signature
    uint8_t  e_elf[12]; // Architecture, endianness, etc.
    uint16_t e_type;
    uint16_t e_machine;
    uint64_t e_entry;
    uint64_t e_phoff;   // Program header offset
    uint16_t e_phnum;   // Number of program headers
} Elf64_Ehdr;
```

Novix checks this signature first:

```
if (ehdr->e_magic != ELF_MAGIC)
    PANIC("Invalid ELF magic");
```

Only if it's valid does the kernel proceed with loading.

Program Headers

Each **program header** describes one segment of the program (for example, `.text`, `.data`, or `.bss`):

```
typedef struct {
    uint32_t p_type;   // Type: PT_LOAD means "load this segment"
    uint32_t p_flags;  // Flags: PF_R, PF_W, PF_X
    uint64_t p_offset; // Offset in the file
```

```
    uint64_t p_vaddr;  // Virtual address in memory
    uint64_t p_filesz; // File size
    uint64_t p_memsz;  // Size in memory (might include zeros)
} Elf64_Phdr;
```

Novix reads each `PT_LOAD` segment and copies it into a flat memory buffer, preserving its virtual layout.

From ELF to Flat Image

The main function responsible for this is:

```
struct process* extract_flat_binary_from_elf(const void *elf_data, int create_process_flag);
```

It performs four critical steps:

Parse and Validate

It first reads the ELF header and all program headers, computing the address range that needs to be allocated:

```
uint64_t min_vaddr = (uint64_t)-1;
uint64_t max_vaddr = 0;

for (int i = 0; i < ehdr->e_phnum; ++i) {
    if (phdr[i].p_type != PT_LOAD) continue;
    if (phdr[i].p_vaddr < min_vaddr)
        min_vaddr = phdr[i].p_vaddr;
    if (phdr[i].p_vaddr + phdr[i].p_memsz > max_vaddr)
        max_vaddr = phdr[i].p_vaddr + phdr[i].p_memsz;
}
```

It then rounds these addresses to **page boundaries**, since memory management in Novix is page-based.

Allocate a Flat Image

A flat memory region is allocated to hold all program segments contiguously:

```
size_t image_size = max_vaddr - min_vaddr;
void *flat_image = alloc_pages((image_size + PAGE_SIZE - 1) / PAGE_SIZE);
memset(flat_image, 0, image_size);
```

This serves as the **in-memory binary** used by the process creator.

Copy Segments

Each loadable segment (`PT_LOAD`) is copied into the buffer:

```
const void *src = (const uint8_t *) elf_data + phdr[i].p_offset;
void *dest = (uint8_t *) flat_image + (phdr[i].p_vaddr - min_vaddr);
memcpy(dest, src, phdr[i].p_filesz);
```

If a segment's `memsz` is larger than `filesz`, the remaining bytes are automatically zeroed (for `.bss`).

Resolve Special Symbols

After the flat image is built, Novix extracts special **linker-generated addresses**:

```
g_user_stack_top    = get_symbol_address(elf_data, "__user_stack_top");
g_user_stack_bottom = get_symbol_address(elf_data, "__user_stack_bottom");
g_user_heap_start   = get_symbol_address(elf_data, "__user_heap_start");
g_user_heap_end     = get_symbol_address(elf_data, "__user_heap_end");
```

These are defined in the user linker script (`user.ld`), and tell the kernel where to map the stack and heap.

Symbol Lookup

The helper `get_symbol_address()` scans the ELF's **symbol table** (`.symtab`) to find named symbols:

```
const Elf64_Sym *symbols = (const Elf64_Sym *)((const uint8_t *)elf_data + symtab->sh_offset);
for (size_t i = 0; i < symbol_count; ++i) {
    const char *name = strtab_p + symbols[i].st_name;
    if (strcmp(name, symbol_name) == 0)
        return symbols[i].st_value;
}
```

This allows the OS to locate `__user_stack_top` and similar symbols dynamically at boot.

Creating the Process

Finally, depending on the flag provided, the ELF loader either **creates** or **executes** a process:

```
if (create_process_flag)
    return create_init_process(flat_image, image_size, 1);
else
    return exec_process(flat_image, image_size, 0);
```

- `CREATE_PROCESS` → called during boot (e.g., `idle.elf`, `init.elf`)
- `EXEC_PROCESS` → used later when spawning new processes (e.g., `shell.elf`)

Putting It All Together: kernel_main()

Let's see how everything connects inside the kernel:

```
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();

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

    // Convert ELF → flat → process
    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;

    LOG_INFO("[kernel_main] idle.elf loaded.");
```

```
    for(;;);
}
```

This completes the boot chain: 1. Load ELF from initramfs (`tarfs_lookup`)
2. Extract and flatten it
3. Create a process and load it into memory
4. Run forever — until the scheduler and multitasking layer take over

Summary

In this chapter, we've:

- Parsed ELF headers and program segments
- Extracted flat binaries for process creation
- Resolved stack and heap symbols dynamically
- Integrated ELF loading into the Novix boot sequence
- Prepared for user program execution

With ELF loading now working, Novix can run **real user programs** — starting with our idle task, and soon, the shell!

A smile on my face — because our OS can now load and execute real ELF programs!