exec() in Novix OS

In the last lesson, we built `fork()`, allowing processes to **clone themselves**.
Now we'll take the next logical step — replacing a process's code entirely with a **new program**.

This is the essence of the **`exec()`** system call.

Learning Goals

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

- What `exec()` does in Unix-like systems
- How Novix replaces a process's memory image
- How a user program invokes `exec()`
- How process switching and cleanup work internally

The Concept of exec()

In Unix-like systems, `exec()` loads a new program into the current process's address space.
It does **not** create a new process — instead, it transforms the existing one into a different program.

fork + exec

In most OSes, new programs start like this:

```
$ shell
$ ls
```

Internally:

1. The shell calls `fork()` to create a child process.
2. The child calls `exec("ls")` to replace its memory with the `ls` binary.

So, **`fork()` creates**, and **`exec()` replaces**.

Userland Interface

From the user's perspective, calling `exec()` is simple.
In `user/lib/syscall.c`, we wrap the syscall:

```c
int exec(const char *program_name) {
    return syscall(SYS_EXEC, (intptr_t)program_name, 0, 0);
}
```

This sends the program name (e.g. `"hello.elf"`) to the kernel.

Inside the Kernel: sys_exec()

Now let's walk through the kernel-side implementation in `kernel/syscall.c`.

When the user triggers `SYS_EXEC`, the kernel switches into supervisor mode and handles it:

```c
case SYS_EXEC:
    enable_sum();
    const char* filename = (const char*)f->regs.a0;
    f->regs.a0 = sys_exec(filename);
    disable_sum();
    break;
```

Here we:

- Enable **SUM** (Supervisor User Memory access), so the kernel can safely read user-space strings.
- Call the internal `sys_exec()` function.
- Disable SUM again afterward.

sys_exec() Implementation

Here's the full breakdown of how Novix executes a new program:

```
int sys_exec(const char *progname) {
    char progname_buf[PROC_NAME_MAX_LEN];
    size_t fs;

    // 1. Search in tarfs
    enable_sum();
    strncpy(progname_buf, progname, sizeof(progname_buf) - 1);
    progname_buf[sizeof(progname_buf) - 1] = '\0';
    const void *elf_data = tarfs_lookup(progname, &fs, 0);
    if (!elf_data) {
        uart_printf("[sys_exec] %s not found!\n", progname);
        return -1;
    }
    disable_sum();

    // 2. Switch to kernel page table
    set_active_pagetable((uintptr_t)kernel_pagetable);

    // 3. Free current process's old memory
    proc_t *proc = current_proc;
    process_free_userspace(proc);

    // 4. Load the new ELF binary
    struct process *ep = extract_flat_binary_from_elf(elf_data, EXEC_PROCESS);

    // 5. Rename process
    strip_elf_extension(progname_buf, ep->name, sizeof(ep->name));

    // 6. Activate new page table and stack
    set_active_pagetable((uintptr_t)ep->page_table);
    WRITE_CSR(sscratch, (uint64_t)(ep->tf_stack + sizeof(ep->tf_stack)));

    // 7. Setup trapframe
    uint64_t sp = g_user_stack_top;
    trap_frame_t *tf = ep->tf;
    memset(tf, 0, sizeof(*tf));
    tf->epc = USER_BASE;              // entry point
    tf->regs.ra = (uint64_t)user_return;
    tf->sp = sp;

    LOG_USER_INFO("[sys_exec] ready...");
    yield();

    return 0;
}
```

Step-by-Step Breakdown

Let's walk through what happens:

1. **User Request**
   The process calls `exec("hello.elf")`. The kernel receives the filename as a pointer to user memory.
2. **Access Control**
   The kernel enables **SUM**, allowing it to safely copy data from user space.
3. **File Lookup**
   The kernel searches the TAR filesystem (`tarfs_lookup`) for the ELF binary.
4. **Cleanup**
   The current process's user memory (stack, heap, text, data) is freed — we're about to replace it.
5. **ELF Loading**
   The new binary is loaded into memory with `extract_flat_binary_from_elf()`.
6. **Rename & Reset**
   The PCB (Process Control Block) updates its name to match the new program (without `.elf`).
7. **Trapframe Setup**
   The new process gets a fresh **trapframe** with correct `epc` (entry point) and `sp` (stack pointer).
8. **Scheduler Call**
   The kernel calls `yield()` — transferring control back to the scheduler, which resumes user mode in the new program.

User Test: exec() in Action

To test this, we'll use two simple programs.

user/hello.c

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

void main() {
    char *msg = "[hello] Hello from exec!\n";
    puts(msg);
    for(;;); // no exit yet
}
```

A tiny program that just prints a message.

user/shell.c

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

void main() {
    cls();
    puts("[shell] Welcome to the shell...\n");

    int pid = fork();
    if (pid == 0) {
        // Child replaces itself with a new program
        exec("hello.elf");
    } else {
        printf("[shell] created child with pid %d\n", pid);
        yield(); // temporary until timer is active
    }
```

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

Expected Output

```
[shell] Welcome to the shell...
[shell] created child with pid 1
[hello] Hello from exec!
```

You've just seen process creation (`fork`) and execution replacement (`exec`) in action — together, they form the classic **Unix process model**.

Summary

In this chapter, we implemented **`exec()`**, one of the most powerful system calls in any OS.

- The process's memory is replaced with a new ELF binary
- Its trapframe and stack are reinitialized
- The kernel hands control to the new program safely
- Combined with `fork()`, Novix can now launch new applications dynamically

This combination — `fork()` followed by `exec()` — is what powers all process creation in Unix and Linux systems today.

```
*A smile on my face - because Novix OS can now run multiple programs, launched by other programs!*
```