exit() in Novix OS

In previous lessons, we've built up the **core process model** of Novix OS:

- `fork()` — creates a new process
- `exec()` — replaces the process image

But one crucial piece has been missing: **how to end a process properly**.

That's where `exit()` comes in.

Learning Goals

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

- What the `exit()` system call does
- How Novix OS terminates user processes
- How process state transitions to "zombie"
- How the kernel prepares for future `wait()` support

What is exit()?

In Unix-like systems, `exit()` ends the current process and returns an **exit code** to the parent. The kernel then cleans up the process's memory and marks it as terminated (a "zombie") until the parent collects its status.

For now, in Novix, we'll focus on the first part — marking the process as terminated.

Userland Implementation

The userland `exit()` is very simple — it just triggers a system call:

`user/lib/syscall.c`

```c
void exit(int status) {
    syscall(SYS_EXIT, status, 0, 0);
    while (1) {}    // safety net if kernel does not return
}
```

Note:

> The infinite loop at the end acts as a "safety net."
> Normally, control should never return from the kernel after `exit()`.
> But if it does, we trap the process to prevent undefined behavior.

Inside the Kernel

When the user process calls `exit()`, the kernel handles the request in `kernel/syscall.c`.

System call dispatch

```c
case SYS_EXIT:
    sys_exit(f->regs.a0);
    f->regs.a0 = 0;
    break;
```

Here the kernel passes the exit code from register `a0` into the helper function `sys_exit()`.

Kernel Logic

The `sys_exit()` function performs the actual process termination:

```
void sys_exit(int code) {
    proc_t *proc = current_proc;

    LOG_USER_INFO("[sys_exit] pid=%d exit(%d)", proc->pid, code);

    proc->state = PROC_ZOMBIE;
    proc->exit_code = code;
}
```

Let's break this down:

1. **Get current process**
   The kernel retrieves the process structure (`current_proc`) of the calling process.
2. **Log for debugging**
   Helpful info for tracing process lifecycle events.
3. **Mark as ZOMBIE**
   The process is no longer runnable but still exists in the process table.
   This allows the parent process to later `wait()` for it — a behavior we'll implement later.
4. **Store exit code**
   The kernel keeps the provided exit status (`exit(0)` means success).

Process Lifecycle Recap

Phase	Description
Running	Actively executing instructions
Runnable	Ready to run, waiting for scheduler
Blocked	Waiting for I/O or resource
Zombie	Terminated, waiting for parent to `wait()`
Free	Slot reusable for a new process

`exit()` transitions the process from **Running** → **Zombie**.

Testing exit()

Now, let's see it in action by updating our existing user programs.

user/hello.c

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

void main() {
    char *msg = "[hello] Hello from exec!\n";
    puts(msg);

    exit(0);

    for (;;);  // no wait yet, hang indefinitely for safety
}
```

This program now exits cleanly instead of running forever.

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
        exec("hello.elf");
    } else {
        printf("[shell] created child with pid %d\n", pid);
        yield(); // temporary until timer is active
    }

    for(;;);
}
```

The shell forks a child, the child replaces itself with `hello.elf`, and then exits cleanly.

Expected Output

```
[shell] Welcome to the shell...
[shell] created child with pid 1
[hello] Hello from exec!
[sys_exit] pid=1 exit(0)
```

The kernel logs confirm the process exited normally.

Summary

In this lesson, we introduced the **`exit()`** system call — completing the **fork/exec/exit** process lifecycle.

- User processes can now terminate gracefully
- Kernel marks terminated processes as zombies
- Exit codes are stored for future parent retrieval
- Foundation laid for implementing `wait()` next

A smile on my face - because Novix OS can now start, run, and end user programs cleanly!