fork() in Novix OS

Until now, Novix OS could run a single user process — the shell — and perform system calls.
But a real OS must be able to **spawn new processes** dynamically.

In this chapter, we'll implement `fork()`, which duplicates the current process, creating an identical **child process**.

Learning Goals

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

- How `fork()` duplicates a running process
- How Novix copies trapframes, page tables, and memory
- How parent and child differ after a fork
- How process state transitions work in multitasking

What is fork()?

In Unix-like systems, `fork()` creates a new process by duplicating the calling one.

It returns:

- **0** in the **child process**
- **child's PID** in the **parent process**

So both parent and child continue execution from the same instruction — but with **different return values**.

Example:

```c
int pid = fork();

if (pid == 0)
    printf("Hello from child!\n");
else
    printf("Hello from parent, child pid=%d\n", pid);
```

In this example, you'll see both lines printed — one from the parent, one from the child.

Userland Interface

In user space, the implementation is minimal.
We simply wrap the `SYS_FORK` syscall inside a small function:

`user/lib/syscall.c`

```c
int fork(void) {
    return syscall(SYS_FORK, 0, 0, 0);
}
```

That's it! When the user calls `fork()`, it triggers an `ecall` with syscall number `SYS_FORK`.
The real work happens in the kernel.

Kernel-Side Fork Implementation

Now, let's explore the heart of process duplication: the kernel's `sys_fork()` function.

Step-by-Step Overview

Here's what happens during a fork in Novix OS:

Step	Action	Description
1	Allocate new PCB	Create a new `process` struct for the child
2	Copy trapframe	Duplicate CPU register state from the parent
3	Allocate a new page table	Each process gets its own address space
4	Copy user memory pages	Duplicate program and data segments
5	Adjust stack & PC	Ensure both parent and child resume correctly
6	Return PID	Parent gets child PID; child gets `0`

kernel/syscall.c

```
case SYS_FORK:
    f->regs.a0 = sys_fork(1);
    break;
```

And here's the full implementation of `sys_fork()`:

`sys_fork()` Implementation

```
int sys_fork(int debug_flag) {
    proc_t *parent = current_proc;

    // 1. Find a free PCB
    proc_t *child = alloc_free_proc();
    if (!child) return -1;
    process_count += 1;
    if (debug_flag) LOG_USER_DBG("[sys_fork] child allocated at: 0x%x", child);

    // 2. Allocate trapframe
    child->tf = (trap_frame_t *)alloc_pages(1);
    if (!child->tf) return -1;

    // 3. Copy trapframe from parent
    memcpy(child->tf, parent->tf, sizeof(trap_frame_t));

    // 4. Return values differ for parent/child
    child->tf->regs.a0 = 0;           // child sees 0
    parent->tf->regs.a0 = child->pid; // parent sees child PID

    // 5. Set PC/SP
    child->tf->sp = parent->tf->sp;
    child->tf->epc = parent->tf->epc + 4; // move to next instruction

    // 6. Allocate new page table
    pagetable_t new_pt = (pagetable_t)alloc_pages(1);
    child->page_table = new_pt;
```

```c
    // 7. Inherit kernel mappings
    for (paddr_t pa = (paddr_t)__kernel_base;
         pa < (paddr_t)__free_ram_end;
         pa += PAGE_SIZE) {
        map_page(new_pt, pa, pa, PTE_V | PTE_R | PTE_W | PTE_X, 0);
    }

    // 8. Copy user pages
    vaddr_t start_va = USER_BASE;
    vaddr_t end_va = parent->heap_end > parent->user_stack_top
                   ? parent->heap_end
                   : parent->user_stack_top;

    for (vaddr_t va = start_va; va < end_va; va += PAGE_SIZE) {
        paddr_t pa_parent = walk_page(parent->page_table, va);
        if (!pa_parent) continue;

        paddr_t pa_child = alloc_pages(1);
        memcpy((void*)pa_child, (void*)PA2KA(pa_parent), PAGE_SIZE);
        map_page(new_pt, va, pa_child, PTE_V | PTE_U | PTE_R | PTE_W | PTE_X, 0);
    }

    // 9. Inherit process attributes
    child->state = PROC_RUNNABLE;
    child->heap_end = parent->heap_end;
    child->user_stack_top = parent->user_stack_top;
    child->entry_point = parent->entry_point;
    child->parent = parent;
    child->parent_pid = parent->pid;
    strncpy(child->name, parent->name, sizeof(child->name) - 1);

    LOG_USER_INFO("[fork] fork with pid %d ready.", child->pid);
    return child->pid;
}
```

Step Breakdown

1. **PCB Allocation**
   A new `proc_t` is reserved for the child. It inherits no data yet — just a fresh structure.
2. **Trapframe Copy**
   The child gets an exact copy of the parent's CPU state, ensuring both resume execution correctly.
3. **Return Values**
   Fork's magic behavior is here:
   - The parent's `a0` (return register) = child PID
   - The child's `a0` = 0
4. **Memory Duplication**
   Each user page from the parent's memory is cloned into the child's new page table.
   This gives both processes identical memory contents — initially.
5. **Stack & PC**
   The child resumes right *after* the fork instruction, preventing recursive forking.
6. **State Setup**
   The child's process state becomes `PROC_RUNNABLE`, ready to be scheduled.

Testing fork()

Now let's test it from userland.

user/shell.c

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

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

    int pid = fork();
    if (pid == 0) {
        // Child process
        puts("[shell] hello from child\n");
    } else {
        // Parent process
        printf("[shell] created child with pid %d\n", pid);
        yield(); // simple cooperative scheduling
    }

    for(;;);
}
```

Expected Output

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

You'll see both parent and child messages printed — a simple but powerful proof that two processes now exist in memory.

Summary

In this lesson, we implemented the **fork() system call**, one of the cornerstones of multitasking in Unix-like systems.

- Introduced process duplication using `fork()`
- Copied trapframes, stacks, and memory safely
- Managed independent address spaces for parent and child
- Verified behavior with a working shell test

This marks a **huge step** for Novix OS:
we now have **multiple user processes** running in isolation — a foundation for true multitasking!

    A smile on my face - because Novix OS can now create new processes, just like Unix!