wait() in Novix OS

In the last lessons, we built:

- `fork()` — to create child processes
- `exec()` — to load new programs
- `exit()` — to terminate gracefully

Now it's time to let the **parent process** wait for its **child** to finish.
This is done using the **`wait()`** system call.

Learning Goals

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

- What the `wait()` system call does
- How Novix synchronizes parent and child processes
- How the kernel detects and cleans up zombie processes
- How exit codes are communicated back to the parent

What does wait() do?

In Unix, `wait()` allows a parent process to pause execution until one of its child processes finishes.
It then retrieves that child's **exit code** and cleans up its process slot.

Conceptually:

Parent forks child :

- Child: does some work, calls exit()
- Parent: calls wait(), pauses until child finishes

When the child terminates, the parent "reaps" it — meaning the OS removes it from the process table.

Without `wait()`, zombies would pile up in the system!

Userland Implementation

Let's start with the user library call — just a thin syscall wrapper.

`user/lib/syscall.c`

```c
int wait(int *wstatus) {
    return syscall(SYS_WAIT, (uintptr_t)wstatus, 0, 0);
}
```

The user passes a pointer (`wstatus`) to store the child's exit code.
The kernel will write into that memory once the child is reaped.

Inside the Kernel

When the syscall is triggered, the dispatcher routes it to `sys_wait()`:

kernel/syscall.c

```c
case SYS_WAIT:
    f->regs.a0 = sys_wait((int *)f->regs.a0, 1);
    break;
```

We pass the user-space address for the exit status and a debug flag.

Kernel Logic: sys_wait()

Now let's study the main function that does all the work.

```c
int sys_wait(int *status, int debug_flag) {
    if (debug_flag) LOG_USER_DBG("[sys_wait] started ...");

    proc_t *parent = current_proc;

    while (1) {
        for (int i = 0; i < PROCS_MAX; i++) {
            proc_t *p = &procs[i];

            // Skip empty entries
            if (p == NULL) continue;

            // Check for a zombie child
            if (p->state == PROC_ZOMBIE && p->parent == parent) {
                if (debug_flag) LOG_USER_DBG("[sys_wait] found zombie child pid=%d", p->pid);

                // Copy exit code to user space
                if (status != NULL) {
                    enable_sum();
                    *status = p->exit_code;
                    disable_sum();
                }

                int dead_pid = p->pid;

                // Cleanup process memory and PCB
                free_proc(p);

                if (debug_flag) LOG_USER_DBG("[sys_wait] cleaned up pid=%d", dead_pid);
                return dead_pid;
            }
        }

        // No zombies found yet - yield CPU and try again
        yield();
    }
}
```

Step-by-Step Explanation

Let's walk through it:

1. **Identify the parent**
   `current_proc` is the parent process that called `wait()`.
2. **Scan the process table**
   Iterate through all process slots to find child processes in `PROC_ZOMBIE` state.

3. **Match the parent-child relationship**
   Check if `p->parent == parent`.
4. **Return exit code**
   If a zombie is found, the kernel copies its `exit_code` back to the user via the pointer `status`. (We use `enable_sum()` so the kernel can safely access user memory.)
5. **Cleanup**
   The process structure and memory are released with `free_proc(p)`.
6. **Return PID**
   The parent receives the child's PID and can continue execution.
7. **Otherwise...**
   If no zombie is found, the scheduler `yield()`s — letting the CPU run other tasks until a child exits.

Zombie Cleanup Cycle

State	Description
PROC_RUNNING	Currently executing
PROC_ZOMBIE	Terminated, waiting for parent
Parent calls wait()	Kernel finds zombie, frees memory
PROC_FREE	Slot becomes reusable

This mechanism ensures that **no terminated process stays in memory** indefinitely.

Testing wait()

Let's use our shell and hello programs to verify.

`user/hello.c`

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

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

    exit(0);
}
```

This child program simply prints a message and exits.

`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);

        puts("[shell] waiting for child...\n");
        int status;
        int dead = wait(&status);
        printf("[shell] child %d exited with code %d\n", dead, status);

        yield(); // temporary until timer is active
    }

    for(;;);
}
```

Expected Output

```
[shell] Welcome to the shell...
[shell] created child with pid 3
[shell] waiting for child...
[hello] Hello from exec!
[sys_exit] pid=3 exit(0)
[shell] child 3 exited with code 0
```

The parent successfully waited for its child and retrieved its exit code.

Summary

We've now implemented the **full process lifecycle** in Novix OS!

- `fork()` — Create a new process
- `exec()` — Load a new program
- `exit()` — Terminate the process
- `wait()` — Synchronize with terminated child

Novix OS now behaves like a real Unix-style multitasking kernel — with proper **process management, cleanup, and synchronization**.

```
*A smile on my face - because Novix OS can now start, run, and gracefully
 clean up user processes just like Linux!*
```