Homework 19

Extended Bridge to CS, Spring 2025

Akihito Chinda

September 13, 2025

Question : *UNIX Fork*

The UNIX "Fork" command has been a critical aspect of network programming over the years. I would like you to write a (roughly) one page summary of what the function does and what activities you'd expect to see in the operating system after calling this function from a program. Please pay careful attention to the various states that a process goes through and explain which states you'd expect each process to be in and why.

Solution:

Overview

The UNIX `fork()` system call is a fundamental mechanism for process creation. When a process calls `fork()`, the operating system creates a new process, known as the *child*, which is a near-exact copy of the calling process, known as the *parent*. This duplication includes the memory space, file descriptors, and execution context.

Return Values

The `fork()` function returns:

- `0` to the child process.
- The child's process ID (PID) to the parent process.
- `-1` if the fork fails.

Process States After `fork()`

After the call to `fork()`, both the parent and child processes enter the following states:

1. **New**: The child process is created and initialized.
2. **Ready**: The child is placed in the ready queue, waiting to be scheduled.
3. **Running**: The scheduler selects either the parent or child to execute next.
4. **Waiting/Blocked**: Either process may enter this state if waiting for I/O or other resources.
5. **Terminated**: When a process completes execution, it enters the terminated state. The parent may use `wait()` to clean up the child process and prevent it from becoming a zombie.

System Activities

Upon calling `fork()`, the operating system performs several key activities:

- Allocates a new Process Control Block (PCB) for the child.
- Copies the memory space and file descriptors of the parent.
- Updates scheduling queues and process tables.
- Begins concurrent execution of both processes.

Practical Use

The `fork()` system call is widely used in network servers to handle multiple client connections, in shell environments to execute commands, and in parallel processing tasks. Understanding the behavior and states of processes after a `fork()` is essential for writing efficient and reliable UNIX programs.