NYU Tandon Bridge **Spring 2024**

Homework 19

Student: Shichen Zhang (sz4968)

UNIX `fork()` Summary

`fork()` is a system call in UNIX used to create new processes. The process that calls `fork()` is referred to as the parent process. This system call creates a child process, which is an exact copy of the parent process, meaning the child inherits the parent's code segment, data segment, and other attributes. The main difference is that the child process receives a unique Process ID (PID) in its Process Control Block (PCB).

When creating the child process, instead of immediately duplicating all of the parent's memory, copy-on-write (COW) is implemented to optimize memory usage. Memory is shared between the parent and child processes and marked as read-only (RO) in the Page Map Table (PMT). Both processes can read from the same memory pages, avoiding the need to duplicate large amounts of code and data, thus improving efficiency. If either process attempts to modify (write to) a shared page, the operating system allocates a new page in memory, copies the data from the shared page to the new page, and updates the page table of the process that initiated the write to point to this newly allocated page, marking it as read-write (RW).

After `fork()` is called, it returns two values: for the parent process, it returns the PID of the child process; for the child process, it returns 0. If an error occurs, `fork()` returns -1 to the parent process, indicating the failure to create a new process.

Regarding process states, when the `fork()` system call is executed, the parent process is in the running state. During the creation of the child process, the child enters the new state as the operating system allocates resources and creates its PCB. Once the child process is successfully created, it transitions to the ready state, where it waits for the CPU to schedule it. When the CPU scheduler selects the child process, it moves to the running state.

After `fork()` is executed, the parent process can either continue running or be suspended until the child process finishes. If the parent process continues running, it remains in the running state, executing the next lines of code until it completes and transitions to the exit state. Alternatively, if the parent process is suspended, it enters the blocked state and waits for the child process to finish.

Once the child process completes its execution, it transitions to the exit state and sends a signal to the parent process, letting it know the child has finished. The parent then moves from the blocked state to the ready state, waiting to be scheduled again. When the CPU scheduler selects the parent process, it transitions back to the running state. Eventually, the parent process will complete its execution and move to the exit state.