A fork is the creation of a “child” process that is an exact copy of a “parent” process. They’re almost exactly the same, except the fork command returns a Process Identifier to the parent from the child to differentiate which one was the original and which one was the child, so there’s no philosophical ramifications about the child’s existence and history. The child gets a 0 for that value.

The return value of the fork command will tell you which instance of the process you have. If it returns 0, it’s the child, if it’s greater than 0, then it’s likely the PID of the parent process, and if it’s less than 0 then there was likely an error.

IEEE (Side note, I had to go to Wikipedia to find out what that stood for because it wasn’t immediately available on their website) standardized UNIX into a version called POSIX to make it easier for programmers to write broadly compatible programs for different systems. The purported reason was to minimize system calls. According to the textbook, new processes can only be created by a fork call in POSIX. After the thread is created, either the parent or the child can be changed however it is needed to continue running.

According to this video, the child process does not inherit the memory locks, a lock keeping a process in main memory rather than being swapped to the hard disk so it is always ready to be run. I couldn’t find anything solid on the specific states, but my gut says that new processes of course start out in the new state. However, since the child is a perfect copy of the parent, it would already have any IO or preparatory loading that was performed for the parent ready to go. Some of the other things I read seem to indicate that the parent process is either put to sleep or put back to the ready state while the child process executes. Every invocation of fork() I’ve seen also usually runs the following commands twice, so the child process would finish executing the commands requested, then the parent process would continue and execute its own commands. The video I mentioned above as well as the book say that the particular OS determines which command is executed first, so it’s entirely possible that the parent would

execute first, then the child.

I lost 5 points for not mentioning the following topics:

Paging and Fork: When a fork system call is made, the operating system uses paging to manage the memory of the parent and child processes. Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This is particularly relevant in the context of fork because the child process initially shares the same pages of memory as the parent. The operating system creates page table entries for the child that point to the same physical pages as the parent's entries. This is efficient in terms of memory usage, as only modified pages need to be duplicated.

Copy-On-Write (COW): Copy-on-write is a critical optimization used in conjunction with fork. With COW, when a fork system call is executed, the parent and child processes initially share the same physical memory pages. Only when one of the processes writes to a shared page, a copy of that page is made (page duplication). This strategy significantly reduces the overhead of forking, as unnecessary copying of memory pages is avoided. It ensures that memory is only duplicated when it is absolutely necessary, thus making the forking process more efficient in terms of memory and CPU usage.

Zombie Processes: After the child process finishes its execution, it enters a 'zombie' state. A zombie process is a process that has completed execution but still has an entry in the process table, primarily to report its exit status to its parent process. The parent process reads this exit status via the wait system call, at which point the operating system removes the zombie process's entry from the process table. If a parent process doesn't properly use wait, the child remains in a zombie state, leading to resource leakage.

System Calls and Fork: The fork is a system call that the operating system provides to create a new process. System calls are a mechanism used by a program to request service from the kernel. When fork is called, the operating system performs several tasks: it creates a process control block for the new process, duplicates the context of the parent process, and initializes the necessary kernel data structures. This process involves interaction with various subsystems of the kernel, like the scheduler, memory manager, and file system.