The UNIX “Fork” command and the functionality it enables, exemplifies the concept of concurrent operations or multiple processes running at the same time (or at least appearing to run at the same time). As Tanenbaum says in *Modern Operating Systems*, “Without the process abstraction, modern computing could not exist” (Tanenbaum 85). In order to harness the power of pseudo parallelism (disregarding multiprocessor systems for the moment) UNIX needed a method to create and manage more than one process at a time. And so, the Fork command came into existence.

Before we dive deeper into how fork influences operating system activities, we will first need to better understand what the function does and how it works. Fork is (or at least was for a time) the only system call by which UNIX could create new processes. Fork is a function that receives no arguments and returns a pid_t or process ID value as an integer. When fork is called a copy of the current process is made. Once the process has been forked there will be two instances of the process, the “parent” and the “child”. Tanenbaum notes, “In UNIX, the child’s initial address space is a copy of the parent’s, but there are definitely two distinct address spaces involved; no writable memory is shared” (Tanenbaum 90). In the case that memory is shared through copy-on-write, the moment that either of the processes wants to update any portion of memory a copy of the memory will occur before the change is made to the respective process. Once the call to the fork has returned, the return value will be either 0 for a newly created child, a positive value (the pid) for the parent, or a negative value if the fork was unsuccessful. Once the child exists, it will usually update the necessary information and memory in order to run the desired process.

After fork has been called, both the parent and child will continue to run from the point immediately following the call to fork, which created the child. At this stage, the operating system will want to manage how the processes are run and in which order. There are many scheduling algorithms (outside the scope of this paper) which are used to manage how processes run but will most certainly utilize the Sleep and Wait functions to manage which processes run when and wait until others are completed before running. As Professor Katz mentions in module 23, there are 5 states in which a process can be at any given time, 1. New, 2. Ready, 3. Blocked, 4. Running, 5. Exit. For a process to call the fork function it must be in a running state. If it is New, Ready, or Blocked, it is waiting to have CPU access to run the process, at which time it will call fork.

An important item to note is how fork will influence the states a process will be in after fork is called. Each system has a process table that stores relevant information for each process that currently exists and is in one of the five states mentioned above. When fork is called the parent’s address space is copied for the child. Once the child program terminates it sends a signal to the parent which let’s the parent know that it can clear the child process from the process table. However, there are scenarios where if not managed carefully, a parent process terminates before the child process, which results in the process never being cleared from the process table. This is known as a “Zombie” process. Since the process table has a limited capacity, if the scheduling of processes isn’t managed, it could result in a process table filled with zombie processes bringing the operating system to a gridlock where it cannot create new processes. Therefore, it is important to make use of the Wait function to ensure this issue is avoided. If Wait is called on parent processes to ensure the child process terminate first, then the parent process would be expected to be in either the ready or blocked state until the child process has passed the exit state and has completed.