The fork system call is run in kernel mode which, when called, duplicates the process in question, resulting in two identical processes running simultaneously. The duplication includes all information about the process, such as the data in the registers and file descriptors. A couple important notes about the fork system call include the following:

1. Numerous fork system calls have an exponential effect. Imagine a scenario, where the process necessitates two fork system calls. Starting with the parent process (P), when the first fork system call is made, the child process P-C1 is created. There are now 2 identical processes running. When each of these process reaches the subsequent fork call, now both of the processes (P and P-C1) are duplicated, leaving us with four processes: P, P-C1, P-C2, PC1-C1. Mathematically speaking, for every fork call made, the resulting number of processes is 2^n where n represents the number of fork calls.
2. The primary differing element between the parent process and the child process following a fork call is the process ID number, the PID. Once a child process is created, it is conceptually independent from the parent process. This means that when a change to data on the parent process is made, the child process will not know about it, and vice versa.

Copying all the pages in the address space of a large process not only would be time consuming, but also costly memory-wise. Therefore, nowadays the implementation of fork instead leverages a Copy-on-Write (COW) ideology to reduce the amount of memory being used. Since the processes are identical, they will share pages, until one of the processes makes an attempt to modify a page. At that moment, duplication of the page that is being modified will occur. The process that references the modified page will continue forward referencing the modified page, and the process that did not request the change will continue forward referencing the original, unedited page.

When the fork system call is made the parent is responsible for executing the fork call and creating the child process. If successful, the fork call will return the PID of the child process to the parent, and will return the value of 0 to the child process. If the fork is unsuccessful, the parent will receive a return value of -1, and no child process will have been created.

After the child process is created by the parent process in kernel mode, the system is brought back into user mode and user mode process execution can continue on each process, starting with the line immediately following the fork system call. The operating system scheduler will determine whether the child process or the parent process runs. In either case, whether it is the parent or the child that runs according to the scheduler, after a fork system call, often times in the program code, a programmer will include an if statement such as: if(pid == 0){ //run child process; }else{ //run parent process; }. It is this way that these two processes can essentially run concurrently because if we are in the parent process then the parent process code will run, and if we are in the child process the child process code will run. When the parent process is in the running state, then the child process is suspended and vice versa, until one or both of the processes reaches the exit state.

Sources in no particular order:
https://blog.codingconfessions.com/p/the-magic-of-fork
https://www.geeksforgeeks.org/difference-between-process-parent-process-and-child-process/
https://unix.stackexchange.com/questions/207918/what-does-it-mean-fork-will-copy-address-space-of-original-process
https://www.geeksforgeeks.org/fork-system-call/
https://en.wikipedia.org/wiki/Fork_(system_call)
https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
https://www.youtube.com/watch?v=IFEFVXvjiHY
https://www.geeksforgeeks.org/introduction-of-system-call/