Homework:

3.1)

The variable **value** is a global variable and its value at the beginning is 5. After we fork() the system creates a child process and the child process gets a copy of the **value** variable, but all the changes that happen to it in the child process will not affect the parent process because each has its own copy. When the pid is 0 we are in the child process, when the pid is > 0 we are in the parent process. In the child the value variable becomes 20 because it adds 15 to value, and after this the child profess returns 0 and terminates. The parent process waits for the child to finish because of wait(NULL) and after the child process finishes the value of the variable **value** is still 5. Now the parent prints the value of **value** at line A, so the output will be:
PARENT : value = 5.

Changes made in the child are not reflected in the parent.

3.2)

At the beginning we start with 1 process. After the first fork() we create one extra process. After the second fork() each process creates a new process, so we have to extra process and the current number is 4. After the third fork() each of the 4 process creates a new process so we have four new process and the total number is 8.

We also know that the total number of process after n calls to fork() is 2^n. n in this case is 3 because we fork three times and 2^3 is 8.

3.5)

Explanation:

The child process gets its own copies of the parent's stack and heap, and changes made to any of them in the child process do not affect the parent process at all, and vice versa. The right answer is Shared Memory Segment because it can be accessed by multiple processes. If the parent process and the child process have shared memory segments, they both can read and write to the same memory region even after a fork().

3.8)

In operating systems, scheduling is split into three main categories: short-term, medium-term. The **short-term scheduler** decides which process gets to use the CPU next. This happens very frequently, sometimes every 100 milliseconds or so, since processes often need only a short burst of CPU time before waiting for something else, like an I/O request. Because of how often it runs, the short-term scheduler needs to be lightning fast. The goal here is to keep things efficient so the CPU stays busy without unnecessary delays.

The **long-term scheduler** works on a much slower schedule. It decides which processes get to enter the system in the first place and controls how many are allowed in at once. The difference with the short-term scheduler is that the long-term scheduler might only run every few minutes. Its job is more about balance than speed. It needs to maintain a good mix of I/O-bound and CPU-bound processes. I/O-bound processes spend more time doing input and output operations, while CPU-bound processes have a heavy focus on computations.The long-term scheduler ensures a good mix so both the CPU and I/O devices stay busy, which keeps the system running efficiently.

The main job of the **medium-term scheduler** is to temporarily remove processes from memory (swapping) to reduce the load on the system when things get crowded. Later, it can bring those processes back and pick up right where they left off. This is particularly useful when memory is tight, or when the system needs to prioritize certain tasks. Swapping helps free up space and keep everything running smoothly without overloading the system.

In summary, the short-term scheduler focuses on quick and frequent decisions to keep the CPU busy. The long-term scheduler looks at the bigger picture, controlling which processes get admitted to keep the system balanced. Lastly, the medium-term scheduler helps smooth things out by managing memory and giving processes a break when needed.

3.9)

A **context switch** happens when the operating system pauses one process and starts another. This process is triggered by an interrupt, which causes the CPU to stop what it's doing and jump to a kernel routine. The main goal of a context switch is to save the current process's state so it can be resumed later, then load the state of the next process that's ready to run.

How does it work? First, the kernel performs a state save, capturing the current state of the running process. Then it switches to the next process by restoring its saved state from its PCB. This means loading its CPU registers and memory state so it can continue running as if it was never interrupted. While this switch is happening, the system isn't doing any productive work, which is why context-switching time is seen as overhead. How long it takes depends on things like memory speed, how many registers need to be saved, and whether the hardware has special tricks to speed things up.

Some processors, like the Sun UltraSPARC, have multiple sets of registers, which makes context switching much faster. More complex operating systems need to do even more during a context switch. For example, in systems with advanced memory management, the kernel might also need to save and reload a process's address space, ensuring that each process's memory is exactly where it should be.

So basically context switching is like putting one process on pause while bringing another back to life. IT is very important but also comes with a performance cost.

3.14)

When we first call fork() we will have the parent process getting the pid of the child process (2603) and the child process returning a pid of 0 because a fork always returns 0 in the child.

So in line A we have child: pid = 0

in line B the child process calls getpid() which returns its actual pid (2603) so the output of line B will be: child: pid1 = 2603.

As we previously said, in the parent process, fork() returns the pid of the child process so the output of line C is: parent: pid = 2603.

At the end, in line D the parent process calls getpid() and it returns its own pid which is 2600. So line D output will be: parent: pid1 = 2600.

Line A -> child: pid = 0

Line B -> child: pid1 = 2603

Line C -> parent: pid = 2603

Line D -> parent: pid1 = 2600