**Viva questions on fork() system call**

**Q1. What does the fork() system call return on success?  
Q2. What is the PID of the child process?  
Q3. Which function is used to get the PID of a process?  
Q4. How many total process are created with the below code**

**int main()**

**{**

**fork();**

**fork();**

**}**

**Answer to Question 1:**

On success, the fork() system call returns:

• **0** to the newly created (child) process.

• **The PID of the child process** to the parent process.

**Answer to Question 2:**

The **PID (Process ID) of the child process** is returned to the **parent process** by the fork() system call.

**Answer to Question 3:**

The function used to get the PID of a process is getpid().

**Answer to Question 4:**

With the following code:

int main() {

fork();

fork();

}

Here’s how the processes are created:

1. The first fork() creates **one new process** (so now there are 2 total processes: parent + child).

2. The second fork() is executed by both the original parent and the newly created child process from the first fork(). This results in **two new processes** (one from the parent and one from the first child).

Thus, in total, **4 processes** are created:

• Original parent process.

• Child from the first fork().

• Two new processes from the second fork().

So, the total number of processes created is **4**.

Here are the answers to your questions about the `wait()` system call:

**### Q1. \*\*Can we use `wait()` to make the child process wait for the parent process to finish?\*\***

No, the `wait()` system call is used by a parent process to wait for its child process to finish, not the other way around. The child process cannot use `wait()` to wait for the parent process. If the child needs to synchronize with the parent’s completion, other mechanisms like inter-process communication (IPC) would be required, such as signals, pipes, or shared memory.

**### Q2. \*\*What does the `wait()` system call return on success?\*\***

On success, `wait()` returns the process ID (PID) of the terminated child process whose status has been reported. If there are multiple child processes, it will return the PID of the first child that terminates. If the calling process has no children, or if an error occurs, `wait()` returns `-1` and sets `errno` to indicate the error.

. The NULL inside the wait() means that we are not interested to know the status of change of state of child process.

Let me know if you'd like to dive deeper into any related concepts!

Here are explanations for the viva questions related to creating an orphan process:

\*\***Q1. What is an orphan process?\*\***

- An \*\*orphan process\*\* is a process whose parent has finished or terminated, but it is still running. In operating systems, when a parent process exits before its child process, the child process becomes an orphan. Orphan processes are typically adopted by the `init` process (or `systemd` in modern systems), which becomes their new parent.

**\*\*Q2. What is the importance of using the `sleep()` function in the code?\*\***

- The `sleep()` function is used to introduce a delay, ensuring that the parent process terminates before the child. This is critical in demonstrating the creation of an orphan process. The sleep makes sure that the child process keeps running for a while after the parent has exited, leading to the child becoming an orphan. Without `sleep()`, both processes might exit at the same time, which would not show the orphan process scenario clearly.

**\*\*Q3. Why didn’t the program return the command prompt even after termination?\*\***

- If the program didn’t return the command prompt immediately after termination, it could be because the child process is still running. Since the child has become an orphan and was adopted by `init` (or `systemd`), it might still be alive, and if the terminal is waiting for this process to complete, it will not return the command prompt until the child process also exits. Alternatively, the shell might not realize the child process was reparented, resulting in a delay before the prompt returns.

Would you like to go into more detail about process management or specific code examples?

**### Q1. How can you identify the existence of a zombie process in the system?**

A zombie process can be identified in the system by:

- \*\*Using the `ps` command\*\*: In Linux/Unix-based systems, running the `ps aux` or `ps -el` command will display the list of processes. If you see a process with a "Z" in the STAT (state) column, it indicates a zombie process. The "Z" stands for "zombie" or "defunct."

- \*\*Using `top` command\*\*: The `top` command shows system processes in real-time. Zombie processes can be identified by looking for processes marked with a "Z" in the state column.

- \*\*Checking `/proc` filesystem\*\*: Zombie processes may also appear under the `/proc` directory, with their process ID (PID) listed but no executable content.

**### Q2. What is the importance of using the `sleep()` function in the above code?**

The `sleep()` function is often used to introduce a delay in the execution of a process, and its importance in the context of zombie processes is as follows:

- \*\*Parent waiting for the child\*\*: When creating child processes using `fork()`, the `sleep()` function allows the parent process to wait or perform some tasks while the child process runs. If the parent process exits before the child, the child may be re-parented to the `init` process.

- \*\*Keeping the parent alive\*\*: If a parent process calls `sleep()`, it can keep the process alive, allowing the child process to finish its execution. This avoids the creation of a zombie process because the parent can call `wait()` or `waitpid()` to reap the child's exit status when it terminates.

- \*\*Demonstration purposes\*\*: In some code examples, `sleep()` is used to deliberately create a scenario where the child process exits and becomes a zombie while the parent is still running, so that you can observe the zombie state before the parent reaps the child.

Would you like more clarification or further details on zombie processes?

Here are the answers to your viva questions:

**### Q1. \*\*exec() function returns only if \_\_\_\_\_\_\_\_\_\_\_\_\_\_\*\***

- The `exec()` function returns only if it \*\*fails\*\* to replace the current process image. If successful, the `exec()` function does not return because the calling process is replaced.

**### Q2. \*\*The statements that appear after exec() function also execute – True/False.\*\***

- \*\*False\*\*. The statements that appear after an `exec()` call do not execute if `exec()` is successful, because the process image is replaced. If `exec()` fails, then the following statements will be executed.

**### Q3. \*\*How is exec() function different from system() function?\*\***

- The `exec()` function replaces the current process with a new one, and control does not return to the calling process unless there is an error. On the other hand, the `system()` function creates a new child process to run a shell command, and after it finishes, control returns to the calling process. So, in `exec()`, the current process is entirely replaced, while in `system()`, the parent process remains active and waits for the child process to finish.