Operating Systems Shell Project – ProShell

Task 1.5: Provide a concise and descriptive answer to the following questions.

Question 1.5.1: What is the purpose of the fork() system call?

The purpose of fork() is to allow a process to create a new process, which is known as the child process, which is an identical copy of the calling process, which is known as the parent process.

The fork() system call is fundamental for multitasking and multiprocessing in Unix-like systems. It enables the creation of new processes that can execute concurrently with the parent process.

Question 1.5.2: When a process makes a system call and runs kernel code:

A. How does the process obtain the address of the kernel instruction to jump to?

B. Where is the userspace context of the process (program counter and other registers)

stored during the transition from user mode to kernel mode?

A. When a process makes a system call and transitions from user mode to kernel mode, it needs to obtain the address of the kernel instruction to jump to in order to execute the appropriate kernel code. This address is typically stored in a special system register called the "system call table" or "system call vector." The system call number provided by the process is used as an index into this table to fetch the address of the corresponding kernel function.

B. During the transition from user mode to kernel mode, the userspace context of the process, including the program counter (PC) and other registers, is typically stored in a data structure known as the "kernel stack" or "process control block" (PCB). This data structure is maintained by the operating system for each process and contains information about the process's execution state. When the process makes a system call, the processor switches from user mode to kernel mode, and the current state of the process (including the program counter and registers) is saved onto its kernel stack. After the system call is executed, the process's state is restored from the kernel stack, and it resumes execution in user mode.

Question 1.5.3: Explain the following code snippet and write down the list of process state transitions that occur during the following program. You may assume that this is the only process that the CPU is executing.

```
int i = 1;

while (i < 100) { i++; }

printf("%d ", i);

while (i > 0) { i--; }

printf("%d ", i);
```

The program initializes the variable i to 1, then enters a loop where i is incremented until it reaches 100. After the first loop, it prints the value of i. Then, it enters another loop where i is decremented until it reaches 0, and finally, it prints the value of i again.

List of process state transitions:

Start: The process starts in the "Running" state when it is scheduled by the CPU.

Loop 1 (Incrementing i):

While executing the first loop, the process remains in the "Running" state.

Print Statement (After Loop 1):

The process transitions to the "Blocked" state when it performs I/O (printing to the console).

Loop 2 (Decrementing i):

After printing the first value of i, the process resumes the second loop and transitions back to the "Running" state.

Print Statement (After Loop 2):

The process transitions to the "Blocked" state again when performing I/O for the second print statement.

End:

After printing the second value of i, the process completes its execution and transitions to the "Terminated" state.