Muhammad Bilal

2022360

CS311-D

Question 1:

a) The state of P1 will be ready.
b) The state of P2 will also be ready.

At first, P2 is stuck and waiting for a disc I/O operation to finish, while P1 is operating on the CPU in a single-core machine. A disc interrupt happens during this period, and the operating system converts P1 from user mode to kernel mode in order to handle the interrupt. P2 is no longer waiting for the disc read and is able to transition from the blocked to the ready state when the interrupt service brings in the disc blocks needed to unblock P2.

The system is ready to return to user mode and continue running P1 after the interrupt service function has completed. But only one process can be active at once due to the single-core architecture. P1 is currently prepared to continue running, but it hasn't started up just yet because it's still waiting on the CPU to go back to user mode. P2, which was previously blocked owing to the completion of the disc I/O, is now unblocked and in the ready state, awaiting scheduling on the CPU.

In conclusion, both P1 and P2 are in the ready state, awaiting CPU time, at the point when interrupt handling is finished and before the system switches back to user mode. Currently, neither process is active.

Question 2:

a) This instruction is **privileged**.

The interrupt descriptor table is a crucial operating system data structure that describes how the CPU handles exceptions and interrupts (IDT). By writing to the IDT register, changes can be made to the interrupt descriptor table, which controls how hardware and software interrupts are directed to the right handlers. Malicious software may disable the handling of important system exceptions or reroute interrupts, which could lead to system instability, crashes, or attempts at privilege escalation. If this could be altered by unprivileged processes, there would be a serious security issue.

If an unprivileged process could write to the interrupt descriptor table register, it could:

- Disable interrupt handling altogether, causing the system to become unresponsive.
- Redirect interrupts to malicious handlers, allowing an attacker to gain control over system resources.
- Bypass security measures, potentially leading to privilege escalation or denial of service (DoS).

b) This instruction is **unprivileged**.

Processes employ general-purpose CPU registers to carry out calculations, store temporary data, and control flow while they are executing. These registers are utilised by both kernel-level and user-level programs, and they are necessary for fundamental operation. Since these registers are saved/restored during context changes and are isolated to the running process's context, allowing user-level programs to write to them does not provide a serious security concern. Because every process has a unique register context, it is impossible for one process to interfere with another.

Programming at the user level requires writing to general-purpose registers to operate correctly. Common data processing operations like arithmetic operations, function calls, and loop control are performed using these registers. Permitting unprivileged access to these registers does not jeopardise system security or stability because the CPU isolates the register state of every process.

Question 3:

Possible outcome 1:

Child Process (ret == 0):

- In the child process, the value of ret is 0, so the code inside the first if block is executed.
- The child process prints "Hello1\n".
- After that, the child process attempts to execute the command exec("some_executable"). If the exec() system call succeeds, the current process image is replaced by the executable specified ("some_executable"), and the child process will no longer continue with the code that follows. This means "Hello2\n" will not be printed if exec() succeeds.
- If exec() fails (i.e., the specified executable could not be run), the process will not be replaced, and the next line will execute, printing "Hello2\n".

Possible output for the child process:

- Case 1: exec() succeeds:
  "Hello1\n" will be printed.
- Case 2: exec() fails:
  Both "Hello1\n" and "Hello2\n" will be printed.

Outcome 2:

Parent Process (ret > 0):

- In the parent process, the value of ret is the process ID of the child, which is greater than 0.
- The parent process will call wait(), which waits for the child process to finish execution.
- Once the child process finishes, the parent process prints "Hello3\n"

Outcome 3:

Error in fork() (ret < 0):

- If fork() fails (which is rare but possible if system resources are low), ret will be negative.
- In this case, the third else block will execute, and "Hello4\n" will be printed.

Question 4:

Outcome:

Child 1 started

```
Hello from good executable
Child 1 finished
Child 2 started
Child reaped
Child reaped
Parent finished
```