**SYSC4001 Assignment 1**

**PART1:**

**a)**

**i)** The special cards have a $ in front to differentiate from command data and program data. When the $RUN card is encountered, many software components come into play, such as the job entry subsystem (JES) that manages job processing by reading the batch job’s control language which includes commands like $RUN. We also have the system loading component that is responsible for loading the program into memory. When the $RUN card is met, the loader fetches executable code from the storage device and places it in memory. Furthermore, we have the scheduler component that allocates the system’s resources such as CPU and memory to execute the program. Finally, we have the execution component where the OS sets up the necessary environment for the execution of the program, such as managing files, I/O devices and memory allocation. After the components have all been executed, the OS monitors the program’s process, continuing its management of system resources and hadles any interrupts or errors that occur during execution. Once the execution is complete, the OS deallocates the resources, updating job status, and logs the job’s completion for the end user. If there are any additional jobs queued the OS will move onto those jobs.

**ii)** If the $LOAD command was encountered during a FORTRAN compilation process, it would most likely result in an error because the executable is not yet ready. The OS would then either stop the process then produce an error message or simply skip the $LOAD command if error handling allows it to do so.

**b)**

Kernel and user modes work together with memory protection mechanisms to safeguard the memory space assigned to the operating system. When a user program needs to access the hardware or perform a sensitive task, it must make a system call to switch to kernel mode temporarily to execute the privileged instructions on behalf of the user program. Memory protection mechanisms prevent user programs from directly accessing the memory allocated to the OS or other programs, so if a user program tries to access memory outside this range, the hardware raises a trap preventing the operation and ensuring system stability. Together these mechanisms ensure that user applications cannot access or modify the memory of the operating system, maintaining security and integrity within the system.

**c)**

Here are 4 examples of privileged instructions, along with explanations:

1. I/O Instructions: These control input and output operations, such as reading from or writing to hardware devices. User programs are not allowed to execute I/O instructions directly to prevent unauthorized access to hardware devices, as this could interfere with other programs or the OS. These instructions must be managed by the operating system to ensure correct and safe access.
2. Memory Management Instructions: These instructions handle operations like setting up or modifying memory protection. As mentioned in question b), the user program cannot access memory directly allocated to the OS or other programs so it must temporarily enter Kernel mode to do so. Ensuring security and isolation.
3. System Call Instruction: This instruction switches the CPU from user mode to kernel mode so that a user program can request services from the operating system. It is privileged because only the OS should handle sensitive tasks such as I/O operations, process management, or file manipulation, which requires full system access.
4. Interrupt Management: INstructions for enabling or disabling interrupts are also privileged because they directly influence the system’s ability to respond to events. Improper use of these could make the system unresponsive or prone to error by blocking critical interrupts.

These are privileged because allowing the user programs to execute them directly would undermine system stability and security.

**d)**

Here’s a detailed explanation of the interrupt mechanism using both software and hardware components:

i) External signal (HARDWARE): An interrupt begins when an external device sends a signal to the CPU. This external signal is triggered by events such as an I/O operation completion or a timer reaching zero.

ii) Interrupt handling (HARDWARE): Upon detecting the interrupt signal, the CPu stops executing the current instruction, saves its state, and switches to kernel mode. The CPU then loads the address of the appropriate interrupt handle from the interrupt vector, a table of pointers to interrupt handling routines stored in memory.

iii) Interrupt Handler (SOFTWARE): The operating system takes control at this point. The interrupt handler determines the source of the interrupt, processes the interrupt by executing the necessary routine, and completes the requested service.

iv) Return from interrupt(HARDWARE & SOFTWARE): After the interrupt has been dealt with, the CPU restores the saved state and returns to user mode, resuming the execution of the interrupted instruction.

**e)**

A system call is the mechanism through which user-level programs request services from the operating system’s kernel. When a user program invokes a system call, it triggers a software interrupt and the CPU switches from user mode to kernel mode. In kernel mode, the operating system identifies the requested service based on the system call number and executes the corresponding instruction. The CPU then switches back to user mode after the requested instruction has been performed. Examples of system calls are: open(), read(), write(), fork(), exec(). A system call is a type of software interrupt. When a user program makes a system call, it triggers a software-generated interrupt. The interrupt mechanism is used in system calls to facilitate the switch from user mode to kernel mode.

**f)**

A time sharing operating system allows multiple users to interact with a single computer system simultaneously. It achieves this using time-slicing and efficient scheduling algorithms.

**g)**

A process in the ready state can only move directly to the end state if it has completed its execution successfully or has been terminated due to an external event. In a program completion case, the OS schedules the process to run then issues an exit system call. Then the kernel removes the process from the ready queue and moves it to the End state. In a forced termination case, a user can issue a kill command or the process may have encountered a fatal error. The kernel will then forcibly move the process from the ready state to the end state by terminating it.

A process in the running state can move from running to ready in a time slice expiration case or an I/O operation. In a time slice expiration situation, the slice from the time-shared system expires and the process is preempted. The scheduler moves the process back to the ready state and selects another process to execute. In an I/O operation situation, the process will voluntarily relinquish the CPU and move from the running state to a waiting/blocked state. Once the I/O completes the process moves to the ready state, waiting for its turn to execute again.

**h)**

There will be 800 interrupts, and 0.8 seconds to display the characters, and 0.04 seconds to process the interrupts. Total time being 0.84 seconds. For the high resolution screen, there are a total of 400,000 interrupts. With 0.4 seconds to display the pixels, there is a whooping 20 seconds to process the interrupts. The total time being 20.4 seconds. The dumb terminal has a reasonable total time while the high resolution screen has an interrupt processing which lasts 20 seconds, dominating the total time. The problem is the large number of interrupts in the high-resolution screen. In order to solve this problem, we should try implementing batch processing instead of interrupting at every pixel. If an interrupt is generated every 1000 pixels, we can reduce the number of interrupts to 400 reducing the interrupt processing to 0.02 seconds instead of 20 seconds. This makes the total time 0.42 seconds, which is much more manageable.