SYSC 4001

Assignment 1

Mithushan Ravichandramohan 101262467

Tarek Ibrahim 101258978

Part 1 - Concepts

Question 1:

1. The ‘$’ symbol distinguishes command data from program data in job control scripts. Its main purpose is to signal the start of a command, helping the operating system recognize and process it correctly. The use of ‘$’ reduces the likelihood of errors because it's an uncommon character to begin commands with, ensuring clear identification by the system.

In terms of the ‘$RUN’ card, the Job Control Language (JCL) Interpreter reads the ‘$RUN’ card and interprets it as a directive to execute the program. Then, the operating system's scheduler steps in to manage CPU allocation. When the ‘$RUN’ command is processed, the scheduler assigns CPU time for executing the job. Before this, the ‘$LOAD’ card would have been read by the system, which loads the executable into memory using the loader, preparing it for execution. Once loaded, the CPU switches to user mode to run the program. During this transition, context-saving software ensures that important data is saved onto the stack so the program can run without losing critical information.

1. If the operating system detects a ‘$LOAD’ card in the middle of the FORTRAN compilation process, it would terminate the process. This is because the Job Control Language (JCL) system would recognize that the ‘$LOAD’ card is out of sequence. The system knows that the ‘$LOAD’ card is only valid after a successful compilation when an executable has been generated, therefore detecting it prematurely would cause the system to return an error. Upon recognition, the operating system would stop the ‘$LOAD’ operation and notify the user of the issue. It would then terminate the current job, release any resources in memory that were previously allocated for the process, and clean up by removing unnecessary resources. This is to ensure proper memory management and prevent the continuation of faulty operations in the job execution process.

Question 2:

* In user mode, programs are restricted from performing privileged operations. On the other hand, in kernel mode programs have access to all system resources. Memory protection is enforced by the Memory Management Unit (MMU), by isolating the memory space of each process using virtual memory and page tables. This is done to prevent user-mode programs from accessing memory allocated to the OS. If a user-mode program attempts to access protected memory, the MMU blocks the operation and causes a segmentation fault. User programs interact with the operating system through system calls because they allow the OS to safely transition the CPU from user mode to kernel mode. System calls are key to maintaining system integrity and preventing unauthorized access to critical resources.

Question 3:

1. Memory Management: specifically the ‘LOAD’ and ‘STORE’ instructions, where ‘LOAD’ is loading an address into memory, and ‘STORE’ is writing data from a register into memory. These are privileged instructions because allowing user programs to modify memory mappings could lead to data breaches and system corruption by enabling them to access memory belonging to other processes or the operating system.
2. I/O Control: Operations like the ‘IN’ and ‘OUT’ instructions transfer data between registers and I/O devices. These are privileged instructions because direct access to I/O devices is a security risk. After all, poorly written programs can disrupt or take control of hardware resources.
3. Processor Mode Control: ‘CLEAR INTERRUPT’ and ‘SET MODE’ instructions fall under this category. ‘CLEAR INTERRUPT’ ensures uninterrupted kernel code execution and ‘SET MODE’ switches the processor from kernel mode to user mode after a system call. These are privileged instructions because if user programs can change the processor mode, they can also switch the CPU to kernel mode and bypass security restrictions allowing unauthorized access to sensitive system data.
4. Interrupt Handling: The ‘ENABLE INTERRUPT’ and ‘DISABLE INTERRUPT’ instructions control hardware interrupts by enabling or disabling the CPU’s ability to respond to them. These are privileged instructions because controlling interrupts is essential for multitasking with an efficient throughput while responding to hardware events. If user programs can disable interrupts, they can monopolize CPU time and prevent the operating system from performing critical tasks such as job scheduling and resource allocation.

Question 4:

* An interrupt begins when it is triggered by an external hardware signal. This external device sends an interrupt signal to the CPU. Hardware components are mainly responsible for signaling, prioritizing interrupts, and mode switching while software manages context saving, ISR execution, and state restoration.

1. Interrupt Request (Hardware): When a device requests an interrupt, the interrupt controller decides the priority of the request and forwards it to the CPU. The CPU acknowledges the interrupt after completing the current instruction and identifies the source of the interrupt.
2. Mode Switching and Context Saving (Hardware and Software): The CPU switches from user mode to kernel mode using hardware. Before handling the interrupt, the current state of the running program, including the program counter (PC), status registers, and other critical data, is saved onto the stack. This is managed by hardware (for mode switching) and software (for storing register values and other state data). The software that manages this is part of the Interrupt Service Routine (ISR).
3. Determining the Correct ISR (Hardware and Software): The interrupt controller uses the signal to determine the correct ISR by referencing the interrupt vector table stored in memory. Both hardware and software are involved in locating the ISR, but the actual execution of the ISR is entirely handled by the software.
4. Executing the ISR (Software): The ISR performs the necessary actions to handle the interrupt.
5. Signaling Interrupt Completion (Hardware and Software): Once the ISR completes its task, an End of Interrupt (EOI) signal is sent back to the interrupt controller. This informs that the interrupt is complete. The CPU can now return to normal operations. Both hardware (sending the signal) and software (managing the completion process) are involved in this step.
6. Restoring the Program State (Software and Hardware): After the ISR has been completed, the CPU retrieves the saved context from the stack to restore the state of the program that was interrupted (handled by software). The CPU then switches back to user mode (handled by hardware) and continues from where it left off.

Question 5:

* A system call allows user-mode applications to request services from the operating system. This means programs can perform tasks that require privileged access to hardware that user-mode applications can not directly access. Since user programs run in a restricted environment, interrupts are used to make system calls and switch to kernel mode where the operating system can handle these tasks securely.

When a system call is made, a software interrupt is triggered. This causes the CPU to transition from user mode to kernel mode. The request is then handled by a predefined system call handler in the operating system. A few common examples of system calls include ‘CreateFile()’, ‘ReadFile()’, ‘CloseHandle()’ in Windows, and ‘read()’, ‘write()’, ‘open()’, ‘exec()’, ‘exit()’ in Linux.

Batch operating systems also rely on system calls to perform tasks like file management and job scheduling. Commands such as ‘LOAD’, ‘RUN’, and ‘END’ are used to start, manage, and terminate jobs within the system. These system calls ensure that jobs are executed in a controlled manner, while the operating system manages critical resources like memory, I/O, and CPU usage. Allowing user programs to access them directly without the use of system calls would pose a risk to system instability or security risks.

Question 6:

* A Time-Sharing Operating System allows multiple users or jobs to use the system simultaneously by switching around the CPU between different processes. Each process is allocated a specific amount of CPU time. This is known as a time slice. The operating system’s scheduler manages this switching to ensure that all processes are handled fairly.

When a process uses up its time slice the operating system saves the current process's state (context) and moves to the next one. The switching between processes is called context switching. Context switching creates the illusion that each process is using the CPU even though only one process runs at any given moment. Interrupts trigger context switches so that multiple users can interact with the system using only one CPU efficiently with good throughput.

Question 7:

1. In a multitasking operating system, there are two main ways a process can move from the Ready state to the End state.

The first occurs when the process completes its execution normally. The operating system’s scheduler selects the process from the Ready Queue and moves it to the Running state by allocating CPU time. Once the process finishes, the Process Termination Routine within the kernel is triggered. This routine ensures that the process’s resources, such as memory, are released, and the process is moved to the End state.

The second case involves abnormal termination, such as when the process encounters an error (such as an invalid instruction being provided). In this case, the kernel’s Interrupt Handler detects the issue and initiates the Process Cleanup Routine, which safely terminates the process, deallocates its resources, and transitions it to the End state. After either type of termination, the scheduler selects another process from the Ready Queue to continue running.

1. A process can move from the Running state back to the Ready state due to several events:
   1. Time Slice Expiration: This is a common scenario in time-sharing systems which occurs when a process has used up its allotted CPU time, the Timer Interrupt signals the kernel to pause the running process. The Context Switch Routine is then called to save the process’s current state (registers and program counter). The process is then moved back to the Ready Queue. The scheduler will then select another process from the queue to run, and the context switch resumes the new process.
   2. Preemption by a Higher-Priority Process: Another reason a process may move from Running to Ready is the arrival of a higher-priority process, such as one that just completed an I/O operation and is now ready to execute. When this occurs, the scheduler preempts the currently running process, saves its state, and moves it back to the Ready Queue. The higher-priority process then moves into the Running state.

Question 8:

Scenario 1: Dumb Terminal (40x20 characters)

Specifications:

* Terminal size: 40 characters wide by 20 lines tall = 40 \* 20 = 800 characters total
* Time to display one character: 1 millisecond = 1000 microseconds
* Time to process one interrupt: 50 microseconds

Calculations:

1. Total Interrupts:

* Each character generates one interrupt.
* Total interrupts needed = 800 (one per character).

1. Time spent displaying characters:

* Displaying one character takes 1 millisecond = 1000 microseconds.
* For 800 characters, time spent displaying = 800 \* 1000 = 800,000 microseconds.

1. Time spent processing interrupts:

* Processing one interrupt takes 50 microseconds.
* For 800 interrupts, total interrupt processing time = 800 \* 50 = 40,000 microseconds.

1. Total time (displaying + interrupt processing):

* Total time = time spent displaying + time spent processing interrupts
* Total time = 800,000 + 40,000 = 840,000 microseconds = 0.84 seconds.

Scenario 2: High-Resolution Screen (1000x400 pixels)

Specifications:

* + Screen size: 1000 pixels wide by 400 pixels tall = 1000 \* 400 = 400,000 pixels
  + Time to display one pixel: 1 microsecond
  + Time to process one interrupt: 50 microseconds

Calculations:

1. Total Interrupts:

* Each pixel generates one interrupt.
* Total interrupts needed = 400,000 (one per pixel).

1. Time spent displaying pixels:

* Displaying one pixel takes 1 microsecond.
* For 400,000 pixels, time spent displaying = 400,000 \* 1 = 400,000 microseconds.

1. Time spent processing interrupts:

* Processing one interrupt takes 50 microseconds.
* For 400,000 interrupts, total interrupt processing time = 400,000 \* 50 = 20,000,000 microseconds = 20 seconds.

1. Total time (displaying + interrupt processing):

* Total time = time spent displaying + time spent processing interrupts
* Total time = 400,000 + 20,000,000 = 20,400,000 microseconds = 20.4 seconds.

Discussion of Results:

1. Dumb Terminal (40x20):

* The time to display a full screen is 0.84 seconds. This is fast considering the small size of the display. The overhead for processing interrupts is small compared to the time spent displaying characters.

1. High-Resolution Screen (1000x400):

* In this scenario, displaying all the pixels takes only 400,000 microseconds (0.4 seconds). However, 20 seconds are spent processing interrupts, making the total time to display the screen 20.4 seconds. This impedes the flow of the system and is very inneficient.

Problem Analysis:

* + The bottleneck in the high-resolution screen scenario is the excessive number of interrupts. Each pixel generates an interrupt, and since processing an interrupt takes significantly more time than displaying a pixel (50 times longer), the system is inefficient.

Solution: Batching

* + The number of interrupts can be reduced by using batching. Instead of generating an interrupt for every pixel, the system can generate an interrupt after processing a batch of pixels. For example, an interrupt is generated every 1000 pixels.

New Calculations Using Batching:

* + Assumption: The system generates one interrupt for every 1000 pixels.

1. Total Interrupts with Batching:

* Total pixels = 400,000.
* One interrupt is generated for every 1000 pixels, total interrupts = 400,000 / 1000 = 400.

1. Time spent displaying pixels:

* Displaying 400,000 pixels still takes 400,000 microseconds (since the display time remains the same).

1. Time spent processing interrupts:

* Now, there are only 400 interrupts, each taking 50 microseconds.
* Total interrupt processing time = 400 \* 50 = 20,000 microseconds.

1. Total time (displaying + interrupt processing):

* Total time = time spent displaying + time spent processing interrupts
* Total time = 400,000 + 20,000 = 420,000 microseconds = 0.42 seconds.

Observations Of Solution:

* From the above calculations, the solution is proven to resolve the problem. By batching and reducing the number of interrupts, the total time to display the high-resolution screen drops from 20.4 seconds to 0.42 seconds. Minimizing interrupt overhead while maintaining fast display times makes the overall system much more efficient.