Assignment 1 - Part I

SYSC 4001

Abdalla Omar - 101261926

Fareed El Bokl - 101256495

**Part I - Concepts**

i.

The reason the special cards had a sign ($) in the beginning of them was to help the computer differentiate between program and command data when dealing with a batch of cards, since both program and command data were on punch cards. Therefore the “$” sign was used to differentiate between the two. To be even safer, usually two signs were used “$$”.

There are multiple software components that are used to interact with cards. The job control language is responsible for interpreting job control commands such as “$RUN”. Right as the $ is detected, the interpreter parses the instructions and passes them to relevant system components, such as loading an executable or initiating a program to run

Additionally there is also a compiler, The Fortran compiler in this case. When the system encounters a “$FORTRAN” card, the OS will load the FORTRAN compiler. The compiler then reads the following cards containing Fortran code, compiles it into machine code, then stores the output on a large tape/disk. Once compilation is finished, the “$LOAD” card instructs the OS to load the executable (the compiled FORTRAN code) into memory. The loader component of the OS is responsible for this, as it places the program into memory and prepares the program to be executed. Then there is also the memory manager, which allocates necessary space in memory to load the compiled FORTRAN program into memory. Once the program is loaded into memory, the $RUN tells the OS to start the execution of the program itself. The scheduler will assign CPU time for the process to run. Lastly, the input/output manager is used as while the program runs, it interacts with several input/output devices such as reading data from a card or writing output data to a printer. The I/O manager handles these operations by interacting with drivers and ensuring that data is transferred between program and external devices.

Once all the software components have been executed, the OS will firstly have the program begin execution. The OS will get the necessary resources such as memory and CPU time. The OS will also continue monitoring for interrupts or errors. Then there is cleanup after execution, which consists of the OS reclaiming the resources it had allocated such as memory. The OS will also check if there are more jobs in the batch that require processing, and if so it will move onto the next control card.

ii.

If the “$LOAD” card is detected during the FORTRAN compilation process, it would create a conflict as we’d be trying to load an executable before it is actually prepared due to the compilation process not yet being finished. What may happen is incomplete compilation, as the OS may attempt to stop the compiler, preventing it from completely transforming the source code into machine code. This would cause there to be incomplete machine code that is loaded into memory to be compiled. Likewise there may be premature loading, where there is no actual executable code loaded, which would cause an error.

The OS should raise an error when encountering the “$LOAD” card during the compilation process. This would indicate that the sequence of the batch cards is incorrect and that the “$LOAD” card has been issued too early. The system would make the operator aware of this issue, which would allow them to fix this problem in the card deck. From there the OS should either cancel or delay the $LOAD command, allowing the FORTRAN code compilation to continue and for the executable to be generated.

b)

In an operating system, **kernel mode** and **user mode** are distinct modes of operation that enforce security and prevent unauthorized access to critical system resources.

* **Kernel Mode:** This mode has unrestricted access to the system’s hardware and memory, allowing the OS to execute critical system operations such as I/O control, memory management, and process scheduling. Only trusted OS components and privileged instructions are executed in kernel mode.
* **User Mode:** In this mode, user applications run with restricted access, only able to interact with specific memory areas allocated to them by the OS. Any attempt by a user program to access memory outside its boundaries is denied, ensuring the system's stability.
* **Memory Protection:**
  + Memory protection is typically implemented using hardware, like the **Memory Management Unit (MMU)**, to enforce access boundaries. The MMU works in conjunction with the CPU to ensure that processes only access memory segments allocated to them.
  + When a process attempts to access memory reserved for the OS (or another process), the MMU detects this and triggers an **exception** or **trap**, halting the process. This prevents unauthorized reading or writing of system-level memory, preserving OS integrity.

c)

Privileged instructions are CPU commands that can only be executed in **kernel mode** because they control hardware and critical system operations. If a user-mode process tries to execute these instructions, the OS raises an exception.

1. **I/O Control Instructions:** These instructions allow direct interaction with hardware devices like disk drives or printers. Since improper I/O control can crash the system or corrupt data, only the kernel can execute these instructions.
   * **Why Privileged:** They ensure controlled access to I/O devices, preventing system instability.
2. **Interrupt Management:** Disabling or enabling interrupts is a privileged instruction, allowing the kernel to control when external signals are processed.
   * **Why Privileged:** If user-mode programs could disable interrupts, the system could miss critical events, leading to data loss or crashes.
3. **Memory Management Instructions:** Commands that modify memory allocation, page tables, or the MMU are privileged. These are required for allocating or deallocating memory for processes.
   * **Why Privileged:** To prevent user programs from corrupting memory areas or accessing unauthorized regions.
4. **Switching Between User and Kernel Mode:** Instructions that change the mode of the CPU from user mode to kernel mode and vice versa are privileged.
   * **Why Privileged:** Unauthorized mode switching would let user applications bypass memory protection.

d)

The **interrupt mechanism** is a fundamental process in operating systems that allows external or internal events to be handled efficiently. The process can be divided into two main parts: **hardware operations** and **software operations**.

#### **1. External Signal (Hardware)**

* An **external device** (e.g., keyboard, network card) or an internal event (e.g., timer) triggers an **interrupt request (IRQ)**.
* This IRQ signal is sent to the **interrupt controller** (which could be part of the CPU or a separate chip), notifying the CPU of an event that needs attention.

#### **2. Interrupt Acknowledgment (Hardware)**

* **Interrupt Controller**: The interrupt controller informs the CPU about the interrupt.
* **Interrupt Flag (IF):** At this point, the CPU disables further interrupts by **clearing the interrupt flag (IF)** in its status register. This prevents additional interrupts from interfering with the current interrupt handling process.

#### **3. Saving the Current Process State (Hardware & Software)**

* **Registers (Hardware)**: The CPU automatically saves the current **program counter (PC)**, and sometimes other **registers**, onto the stack to remember where to return after the interrupt handling is completed.
* **ALU Flags (Hardware)**: The **Arithmetic Logic Unit (ALU) flags** (e.g., zero flag, carry flag) are also saved, ensuring that any conditions or results from the current computation are preserved for resuming the interrupted process.

#### **4. Switch to Kernel Mode (Hardware)**

* The CPU transitions from **user mode** to **kernel mode** because interrupt handling is considered a privileged operation. This ensures that the interrupt service routine (ISR) has full access to system resources to manage the interrupt appropriately.

#### **5. Interrupt Vector Lookup (Hardware)**

* **Interrupt Vector Table (IVT):** The CPU looks up the interrupt vector table (a special memory location) to determine which **Interrupt Service Routine (ISR)** to run. Each type of interrupt has a specific ISR associated with it.
* The CPU loads the address of the corresponding ISR from the interrupt vector table.

#### **6. Executing the ISR (Software)**

* **Software Execution (Software):** The **Interrupt Service Routine (ISR)** is a software routine specifically written to handle the interrupt event. This routine is part of the OS kernel and depends on the specific interrupt triggered. The ISR processes the event (e.g., reading data from an I/O device, resetting a timer).

#### **7. Restoring Flags and State (Software & Hardware)**

* **ALU Flags (Hardware):** Once the ISR is complete, the ALU flags saved earlier are restored to ensure the CPU can continue with the interrupted process under the same computational conditions.
* **Program Counter & Registers (Hardware):** The CPU retrieves the saved program counter and registers from the stack, restoring the state of the interrupted process.

#### **8. Enabling Interrupts (Hardware)**

* The CPU sets the **interrupt flag (IF)** back to 1, allowing new interrupts to be detected and processed. This enables the system to handle new incoming interrupts after the current one is processed.

#### **9. Return to User Mode (Hardware)**

* Once the ISR is finished, the CPU exits **kernel mode** and returns to **user mode**, continuing the previously interrupted process from where it left off. The **program counter** is restored to its previous state, allowing the process to resume normally.

e)

A **system call** is a mechanism that allows user-mode programs to request services from the kernel. When a user application needs to perform an action that requires elevated privileges, such as file access, it makes a system call.

* **Examples of System Calls:**
* open() – opens a file.
* read() – reads data from a file or device.
* fork() – creates a new process.
* exec() – runs a new program in a process.
* exit() – terminates a process.
* **System Calls and Interrupts:** System calls trigger **software interrupts**. When a system call is made, it raises a software interrupt (called a **trap**) that switches the CPU from user mode to kernel mode, allowing the OS to execute the requested operation.
* **Interrupt Mechanism in System Calls:** The software interrupt (trap) triggers the kernel to take control of the CPU, just like handling hardware interrupts. Once the system call is complete, the process’s state is restored, and it returns to user mode.

f)

#### **Process Scheduling and Timer Mechanism:**

1. **Time Slice and Preemption:**

Each process is assigned a fixed amount of CPU time called a time slice or quantum. A hardware timer (typically integrated with the CPU) is used to measure how long each process runs. If a process’s time slice expires before it completes its task, the OS preempts the process. This means the current state of the process (e.g., registers, program counter, stack) is saved so that it can resume from this point later.

1. **Saving Process State:**

When the time slice for a process ends, the context of the process is saved. This context includes the program counter (which tracks where the process was executing), CPU registers, and other necessary information. The OS then switches to the next process in the ready queue.

1. **Handling Multiple Processes with Context Switching:**

The OS uses context switching to move from one process to another. This involves:

* Saving the current process's state when its time slice expires or it voluntarily gives up the CPU (e.g., waiting for I/O).
* Restoring the state of the next process from the ready queue, allowing it to resume execution where it left off.

1. **Round-Robin Scheduling:**

One of the common scheduling algorithms used in time-sharing systems is Round-Robin scheduling. In this approach, each process is given an equal time slice, and the OS cycles through the processes in the queue, giving each one a chance to execute for a short period. If a process finishes during its time slice, it exits the queue, freeing up resources for other processes.

1. **Preemptive Multitasking:**

In a time-sharing system, preemptive multitasking is crucial. If a process doesn't finish within its assigned time slice, the hardware timer generates an interrupt, signaling the OS to save the process's state and switch to another one. This ensures that no process can hold the CPU indefinitely, allowing interactive users to experience minimal delays.

**Handling Multiple Users and Processes:**

1. **Fair CPU Allocation:**

The OS maintains a ready queue that contains all processes waiting for CPU time. By rapidly switching between processes, the OS ensures each user process gets an equal share of the CPU, ensuring fairness.

1. **Interactive User Experience:**

Since each process receives a small, fixed time slice and the CPU switches rapidly between them, the system can handle multiple interactive users concurrently. While only one process executes at a time, the frequent switching (every few milliseconds) creates the perception that all processes are running simultaneously.

1. **Efficient I/O Management:**

When a process is waiting for I/O (e.g., reading from disk or receiving network input), it voluntarily gives up the CPU and moves to a waiting state. During this time, the CPU switches to other ready processes, optimizing resource use and improving system responsiveness.

g)

In a multitasking OS, processes transition between states (Ready, Running, Waiting, End) as they are managed by the OS kernel.

i. **Events for Process Moving from "Ready" to "End":**

* **Ready to Running:** The OS scheduler selects the process from the ready queue and assigns CPU time.
* **Running to End:** The process completes its execution or encounters a termination signal (e.g., exit() system call or fatal error), leading the OS to deallocate resources and move the process to the "End" state.

ii. **Events for Process Moving from "Running" to "Ready":**

* **Preemption:** The process’s time slice expires, and the scheduler preempts the process, placing it back in the ready queue for future execution.
* **I/O Request:** If the process issues an I/O request (e.g., reading from a disk), it moves to a waiting state, and the OS schedules another process.
* **OS Kernel Reaction:**
  + **Context Switching:** The OS kernel handles context switching between states. It saves the process’s state (registers, program counter) before transitioning and restores it when the process is rescheduled.
  + **Scheduler:** The OS’s scheduler selects processes from the ready queue based on scheduling algorithms (e.g., Round-Robin, Priority-based scheduling).

### 

### 

### 

### h)

1. **Dumb Terminal (40 characters wide by 20 lines):**
   * **Total Characters:** 40 x 20 = 800 characters.
   * **Time to Display:** Each character takes 1 millisecond, so total display time is 800 milliseconds.
   * **Interrupts:** 800 characters generate 800 interrupts, each taking 50 microseconds.
   * **Interrupt Processing Time:** 800 x 50 microseconds = 40 milliseconds.
   * **Total Time:** 800 milliseconds (display) + 40 milliseconds (interrupt processing) = **840 milliseconds**.
2. **High-Resolution Screen (1000 pixels wide by 400 pixels):**
   * **Total Pixels:** 1000 x 400 = 400,000 pixels.
   * **Time to Display:** Each pixel takes 1 microsecond, so total display time is 400,000 microseconds (400 milliseconds).
   * **Interrupts:** 400,000 pixels generate 400,000 interrupts, each taking 50 microseconds.
   * **Interrupt Processing Time:** 400,000 x 50 microseconds = 20,000,000 microseconds (20 seconds).
   * **Total Time:** 400 milliseconds (display) + 20 seconds (interrupt processing) = **20.4 seconds**.

**Problem and Solution:**

* **Problem:** The interrupt processing time for the high-resolution screen is significantly larger than the display time, leading to inefficiency.
* **Solution:** A more efficient solution is to implement **buffering** or use **Direct Memory Access (DMA)**. Buffering allows multiple pixels or characters to be processed per interrupt, reducing the number of interrupts.
* **With Buffering:** If we process 100 pixels per interrupt:
  + **Interrupts:** 400,000 / 100 = 4,000 interrupts.
  + **Interrupt Processing Time:** 4,000 x 50 microseconds = 200 milliseconds.
  + **Total Time:** 400 milliseconds (display) + 200 milliseconds (interrupt processing) = **600 milliseconds**.