SYSC 4001 Operating Systems Fall 2024 – Assignment 1

Lucas
Muhammad Maahir Abdul Aziz, 101244916

Part I – Concepts [4.5 marks]

a) i. [0.5 marks] Explain why these special cards have a special sign in front ($) and describe the software components used to interact with those cards. Explain all the software components that are needed for the card $RUN, how they interact with the card, and describe what action the OS takes after all the software components have been executed.

Ans. a) i. The $ sign is a separator to differentiate commands from regular data input which prevents misinterpretation of data as executable commands. The software components needed for $RUN are the card reader, job scheduler, command interpreter and loader.

- The card reader reads input from the job control cards and identifies commands
- The job scheduler determines the order of job execution based on priorities
- The command interpreter parses the command and translates it into actions for the OS
- The loader loads commands like $FORTRAN or $LOAD

After executing all components linked to $RUN, the OS allocates resources, eg: memory and CPU time, and initiates the execution of the job by transferring control to the execution environment.

a) ii. [0.5 marks] Explain what would happen if, in the middle of the FORTRAN compilation process, we detect the card $LOAD. What should the Operating System do in that case?

Ans. a) ii. If $LOAD is detected in the middle of the FORTRAN compilation, the OS should pause the current compilation process. The OS would save the current state of the FORTRAN compilation process, including required registers and memory states so it can resume compilation later. The OS would call the loader to load the $LOAD. After loading, the OS transfers control to the loaded program for execution. After the loaded program finishes executing, the OS would return to the paused FORTRAN compilation process.

b) [0.5 marks] Explain how kernel and user modes, combined with Memory Protection prevent a user from accessing (reading/writing to) the memory area assigned to the OS.

Ans. b) In kernel mode, the OS has unrestricted access to hardware and memory, while in user mode, applications run with limited access. Memory protection prevents user processes from accessing memory areas assigned to the OS. The hardware Memory Management Unit (MMU) enforces access controls. It uses memory segmentation to isolate user space from kernel space, ensuring that user applications cannot read or write to kernel memory. If there's unauthorized access it will trigger exceptions.

c) [0.4 marks] Write examples of four privileged instructions and explain what they do and why they are privileged instructions.

Ans. c) Privileged instructions are operations that can only be executed in kernel mode, as they directly interact with hardware or manage system resources. 4 examples are:

1. I/O operations: direct access to I/O devices, which can interfere with other processes.

2. Memory management: instructions for modifying page tables or changing memory protection settings.

3. Interrupt control: enabling or disabling interrupts to manage system responsiveness.

4. System timer control: modifying the system clock or timers, which affects process scheduling.

These instructions are privileged to prevent user applications from compromising system integrity and security.

d) [0.5 mark] Explain, in detail, the complete Interrupt mechanism, starting from an external signal until completion. Differentiate clearly what part of the process is carried out by hardware components, and what part is executed by software.

Ans. d) The interrupt mechanism begins when an external hardware signal, such as from an I/O device, indicates that an event needs immediate attention from the CPU. The CPU, upon receiving this signal, acknowledges it and temporarily halts its current task by saving the execution context, including the program counter and register values. This part of the process is handled by hardware, as the interrupt signal is generated and processed by physical components. Next, the CPU uses the interrupt vector table, which maps the interrupt signal to the correct interrupt service routine (ISR), to determine the appropriate software handler. This transition involves the software, where the operating system executes the ISR to address the interrupt, such as reading data from a device or responding to an error. Once the interrupt is handled, the CPU, again through hardware processes, restores the previously saved execution context and resumes the interrupted

program. Thus, hardware manages signal generation, acknowledgment, and context saving/restoration, while software takes over for interrupt vectoring, handling, and executing the ISR.

e) [0.5 marks] Discuss what is a System Call. Explain, in detail, what a System Call is and give examples of known system calls. Explain how system Calls are related to Interrupts and explain how the Interrupt hardware mechanism is used to implement System Calls.

Ans. e) System Call is a mechanism that allows user programs to request services from the OS. It acts as an interface between user applications and kernel services, enabling access to hardware and system resources.

Some examples are open(), read(), write(), exit()

System calls are closely related to interrupts. When a user program invokes a system call, it generates a software interrupt/trap. The CPU switches from user mode to kernel mode, allowing the OS to execute the appropriate service routine. The interrupt mechanism ensures safe and controlled access to critical OS functions.

f) [0.6 marks] Explain briefly how a Time-Sharing Operating System works. How does it manage to handle multiple interactive users with a single CPU?

Ans. f) A Time-Sharing OS allows multiple users to interact with a computer system simultaneously, using a single CPU by rapidly switching between users' processes (which gives an illusion of parallelism).

How does it do this? Using:

- Time Slicing: CPU is allocated to each process for a fixed time interval (quantum). After quantum, the scheduler interrupts the running process and switches to the next process in the queue.

- Process Scheduling: The OS maintains a ready queue and employs scheduling algorithms (like Round Robin) to determine which process to run next.

- Context Switching: The OS saves the state of the currently running process and restores the state of the next scheduled process.

g) [0.5 marks] (Kernel structure) Let us suppose we have a process in a multitasking OS.

i. What are the events needed to make a new process move from the “Ready” to the “End” state?

Ans. g) i. To transition from Ready state to End state, the process first moves to the Running state via a scheduler dispatch. This involves the kernel's scheduler, which is responsible for selecting the next process to execute based on whatever scheduling algorithm. Once the process is executing, it may reach completion and invoke the exit() system call. This system call generates an interrupt that the kernel recognizes as a signal for termination. Once the kernel receives this interrupt, it performs a series of tasks like updating the process control block (PCB) to reflect that the process is no longer active, releasing any allocated resources/memory, and removing the process from the process table. The final state transition occurs as the kernel moves the process from the Running state to the End/Terminated state, completing its cycle.

ii. How about events to move the process from “Running” to “Ready”? Explain how the OS Kernel will react to these different events (in terms of the states of the system and the transitions between these states, and the routines of the kernel involved in the process).

Hint: explain at least two kinds of events to be considered, which will produce different state changes and their corresponding transitions. Explain how the OS Kernel will react to this event in detail. Include the OS components involved in these transitions.

Ans. g) ii. When a process in the Running state requires I/O or needs to wait for an event, it transitions to the Waiting state. This transition occurs when the process issues an I/O request, eg: reading from a disk or waiting for user input, which cannot be done immediately. At this point, the kernel's I/O manager marks the process as blocked and updates its PCB to indicate waiting status. If the process is waiting for output to be sent to a device, the output spooler holds the data until the device is ready to process it.

Once the I/O operation is completed or the event occurs, the kernel receives an interrupt indicating event's completion. This interrupt triggers kernel to execute interrupt handler, which is responsible for managing the event. The handler updates the PCB of the waiting process, marking it as ready for execution again. Then, the process is moved back to the Ready state and enqueued in the ready queue.

Ans. h)

Dumb Terminal:

- Total display time: 800 ms (800 characters x 1ms/character)
- Total interrupt time: 40 ms (800 x 0.05ms/interrupt)
- Total time to display a full screen: 840 ms (sum of the above)
- Total interrupts: 800 (1 interrupt/character)

High-Resolution Screen:

- Total display time: 400 ms (400,000 pixels x 0.001ms/pixel)
- Total interrupt time: 20,000 ms (400,000 interrupts x 0.05ms/interrupt)
- Total time to display a full screen: 20,400 ms (20.4 seconds) (sum of above)
- Total interrupts: 400,000 (1 interrupt/pixel)

Discuss:
The results show a big increase in total time for the high-resolution screen, due to the overwhelming number of interrupts generated. Processing 400,000 interrupts, each taking 50 μs, leads to a total processing time of 20 seconds, which overshadowes the display time of 400 ms.

Proposed solution:
To reduce the interrupt processing time, batch processing can be used. Instead of processing each interrupt immediately, the system can batch multiple display operations together. For eg: processing 100 pixels at once would greatly reduce the number of interrupts generated.

Repeat calculation with batch processing:
Assume 100 pixels/interrupt,

- Total interrupts: 400,000 pixels / 100 = 4,000 interrupts
- Interrupt processing time = 4,000 x 0.05ms = 200ms
- Total time = 400ms + 200ms = 600ms

This shows that implementing a mechanism like batch processing can drastically reduce total processing time, making it more efficient to handle high-resolution displays while minimizing CPU overhead.