Team Apol - Nieva, Perez, Pol

1. Explain what happens during an interrupt?

What are the contents of the stack when an interrupt happens and returns?

When an interrupt happens, all the register values are copied to a special data structure or to a memory area(stack) which is in primary memory. and they stay in the stack while CPU starts executing the ISR. Once the routine is over, the registers are loaded back with their original values from the stack and continue what they are doing before the interrupt occur.

A stack is contained and identified by the segment selector in the SS register. The ESP register is the stack pointer that gives the address of the top of the stack. The register values are places in the stack by PUSH and removed by POP. The processor decrements the ESP register if an item is pushed in the stack. When an item is popped off the stack, the processor

reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows down in memory (towards lesser addresses) when items are pushed on the stack and shrinks up (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory. When a system sets up many stacks, only one stack—the current stack—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

2. What is context-switching?

It is the switching of the CPU from one process or thread to another. It can be described as the kernel performing the different activities with regard to processes (including threads) on the CPU: (a) suspending the progression of one process and storing the CPU's state for that process somewhere in memory, (b) retrieving the context of the next process from memory and restoring it in the CPU's registers and (c) returning to the location indicated by the program counter in order to resume the process. It is generally computationally intensive because it requires considerable processor time.

A context switch is used for thread or process switch and is used to mean domain crossing. It is sometimes described as the kernel suspending execution of one process on the CPU and resuming execution of some other process that had previously been suspended. A context is represented by a PCB (Process Control Block). It enables multiple processes to share a single CPU and an essential feature of a multitasking operating system. During a context switch, the kernel will save the context of the old process in its PCB and then load the saved context of the new process scheduled to run.

There are two types of context switching, the software and the hardware context switching. Software context switching is to provide a function that saves the current stack pointer (ESP) and reloads a new stack pointer (SS:ESP) while hardware context switching does not save all the registers but stores nearly all registers whether they are required or not.

3. How do we achieve SW-based context-switching during an interrupt?

Software-based context switching, unlike hardware-based, stores and reloads only the necessary states needed for switching. It saves the current state of the stack pointer (ESP) and loads the new stack pointer (SS:ESP) for execution. To be able to achieve and perform multitasking using context switching, a scheduler would be needed. A scheduler schedules a thread by putting it in a queue. This scheduler generates interrupts at regular time interval and, saves the current state of the running process and then loads and executes the next thread in the queue.

All context switches are initiated by an interrupt. This could be an actual hardware interrupt that runs a driver or a software call, that performs a hardware-interrupt-like call sequence to enter the OS. In the case of a driver interrupt, the OS provides an entry point that the driver can call instead of performing the normal direct interrupt-return & so allows a driver to exit via the OS scheduler if it needs the OS to set a thread ready. Core state for the interrupted thread has to be saved. On a simple embedded system, this is pushing all registers onto the thread stack and saving the stack pointer. It would be necessary to mark the thread stack position where the change to interrupt-state occurred to allow for nested interrupts. The driver/system call runs and may change the set of ready threads by adding/removing TCB's(thread control block) from internal queues for the different thread priorities, so that the thread will be added to the ready set, or a running thread may have called sleep() and so elected to remove itself from the ready set. The OS scheduler algorithm is run to decide which thread to run next, typically the highest-priority ready thread that is at the front of the queue for that priority. The saved stack pointer from the TCB for that thread is retrieved and loaded into the hardware stack pointer. The registers would be popped from the stack of the selected thread to restore the core state of the selected thread. An interrupt-return is performed and transferring execution to the selected thread.