Question 1:

***Answer: yes, because the hardware timer is something that would be controlled at kernel level***

**Hardware timers** are used for timing and counting operations, allowing the processor to carry on with some other process while the **timer** process runs. ... A value is loaded into a register, which is then continuously compared with a **timer** register as it runs

The **Instructions** that can run only in Kernel Mode are called **Privileged Instructions** . **Privileged Instructions** possess the following characteristics : (i) If any attempt is made to execute a **Privileged Instruction** in User Mode, then it will not be executed and treated as an illegal **instruction**.

In any Operating System, it is necessary to have [Dual Mode Operation](https://tutorialspoint.dev/slugresolver/dual-mode-operations-os/) to ensure protection and security of the System from unauthorized or errant users . This Dual Mode separates the User Mode from the System Mode or Kernel Mode.



The Instructions that can run only in Kernel Mode are called Privileged Instructions .

Privileged Instructions possess the following characteristics :

(i) If any attempt is made to execute a Privileged Instruction in User Mode, then it will not be executed and treated as an illegal instruction. The Hardware traps it to the Operating System.

(ii) Before transferring the control to any User Program, it is the responsibility of the Operating System to ensure that the **Timer** is set to interrupt. Thus, if the timer interrupts then the Operating System regains the control.  
**Thus, any instruction which can modify the contents of the Timer is a Privileged Instruction.**

(iii) Privileged Instructions are used by the Operating System in order to achieve correct operation.

(iv) Various examples of Privileged Instructions include:

* I/O instructions and Halt instructions
* Turn off all Interrupts
* Set the Timer
* Context Switching
* Clear the Memory or Remove a process from the Memory
* Modify entries in Device-status table

//----------------------------------------------------------------------------------------------------------------------

Question 2:

***Answer: They chose this type of implementation to avoid user programs from interfering with memory used by other programs or the operating system memory itself.***

**Interrupt** and **trap** numbers are defined by the hardware which is also responsible for calling the procedure in the kernel space. An **interrupt** handler is called in response to a signal from another device while a **trap** handler is called in response to an instruction executed within theCPU.

A **trap** is a software-generated **interrupt**. An **interrupt** can be used to signal the completion of an I/O to obviate the need for device polling. A **trap** can be used to call operating system routines or to catch arithmetic errors. ... **Interrupts** are hardware **interrupts**, while **traps** are software-invoked **interrupts**.

The **process stack** or task **stack** is typically an area of preserved main storage (system memory) that is used for return addresses, procedure arguments, temporarily saved registers, and locally allocated variables. The processor typically contains a register that points to the top of the **stack**.

A **stack** is a special area of computer's **memory** which stores temporary variables created by a function. In **stack**, variables are declared, stored and initialized during runtime. It is a temporary storage **memory**. ... The **stack** section mostly contains methods, local variable, and reference variables

The operating system cannot continue using user-accessible memory when it is handling an interrupt, that program may be buggy, malicious, or worse. It may overwrite any part of its address space at any time. That includes the stack.

The stack is often a critical part of the execution of a program, so the operating system has no choice other than to switch to another stack, which the user mode program can't corrupt.

A **process** terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its **process** control block (**PCB**) is erased i.e., the **PCB's** memory space is returned to a free memory pool.

//----------------------------------------------------------------------------------------------

Question 3:

***The hardware interrupt request run the interrupt handler which creates a thread to be run and then it is queued, and the CPU runs the thread shortly thereafter.***

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**.

When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread.  This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request.

//--------------------------------------------------------------------------------------------------

Question 4:

***DONE, see Carlos for diagram***

Question 5:

1. ***Answer: Kernel mode, also known as supervisory mode, since P33 has kernel level threads, that means that it is going to run in supervisory mode.***
2. ***P92 might not even need Context switching, but if it does it should be the same for both of them. P33 will always need a context switch because it is in kernel mode***
3. ***When a thread is blocked it is unable to run until some external even happens. In the case of P92 it might slow down the user program, but in the case of P33 it might slow down or crash the computer.***