EXCEPTION AND INTERRUP HANDLING ON THE x86:

Sections:

- Loading of kernel
- Reload GDT
- Exceptions and interrupts handling
- Saving registers
- Setting up the interrupt controller
- Getting into the Ring3
- Interrupts Handling in ring 3
- Spurious interrupt Handling

1)LOADING OF KERNAL:

kernel refers to the core component that forms the central part of the operating system. It is a software program that acts as an intermediary between the hardware and software layers of the system. The kernel is responsible for managing various system resources, providing essential services, and enforcing security and protection mechanisms. Some of the key functions performed by the kernel include:

Process Management: The kernel manages the creation, execution, and termination of processes. It allocates system resources, schedules processes for execution, and provides mechanisms for inter-process communication and synchronization.

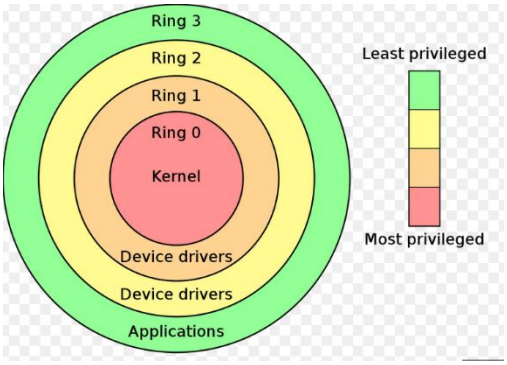
Memory Management: The kernel oversees the allocation and deallocation of memory to processes. It manages virtual memory, which includes mapping virtual addresses to physical memory, handling memory paging or swapping, and enforcing memory protection.

Device Management: The kernel interacts with hardware devices, such as input/output (I/O) devices, storage devices, and networking devices. It provides device drivers to facilitate communication between the devices and software applications.

File System Management: The kernel manages the organization, storage, and retrieval of files on storage devices. It handles file creation, deletion, and manipulation, as well as managing access permissions and file metadata.

Security and Protection: The kernel enforces security policies, access control, and protection mechanisms to ensure the integrity, confidentiality, and availability of system resources. It controls access to sensitive resources and handles authentication and authorization.

System Calls: The kernel provides an interface for applications to interact with the underlying hardware and system resources through system calls. Applications can request services from the kernel by invoking system calls, which provide an abstraction layer for accessing low-level functionalities.



Loading the Kernel into Memory:

```
LoadKernel:
    mov si,ReadPacket
    mov word[si],0x10
    mov word[si+2],100
    mov word[si+4],0
    mov word[si+6],0x1000
    mov dword[si+8],6
    mov dword[si+0xc],0
    mov dl,[DriveId]
    mov ah,0x42
    int 0x13
    jc  ReadError
```

1. Clearing the Direction Flag:

- Set the direction flag (DF) to forward direction (CLD).
- Ensures that the move instruction processes data from low memory address to high memory address.

2. Setting the Source and Destination Addresses:

- Store the destination address in the RDI register.
- Store the source address in the RSI register.
- Use the "move" instruction (move qword) to copy data between the addresses.

3. Configuring the Counter:

- Use the RCX register as a counter for the number of iterations.
- Divide the desired size by 8 (size of each quadword) and store the result in RCX.
- This determines the number of times the move instruction will execute.

4. Executing the Move Instruction:

- Use the "repeat" prefix along with the "move quadword" instruction.
- This repeats the move instruction the specified number of times stored in RCX.
- Each iteration copies 8 bytes (a quadword) from the source to the destination address.

5. Copying the Kernel into Memory:

- After the move instruction executes, the kernel is copied into the specified destination address.
- In this case, the kernel is loaded from disk into memory at address 200000 (hexadecimal).

6. Transferring Execution to the Kernel:

- Include a jump instruction to transfer execution to the kernel at the base address 200000.
- This ensures that the processor starts executing the code of the loaded kernel.

Note: The code described above is written in 64-bit mode, assuming familiarity with x86 assembly language and associated registers. By structuring the information in this way, it becomes easier to understand and follow the sequence of steps involved in loading the kernel into memory. Finally we output the character K to show we are in kernel file.

```
1  [BITS 64]
2  [ORG 0x200000]
3
4  start:
5      mov byte[0xb8000],'K'
6      mov byte[0xb8001],0xa
7
8  End:
9      hlt
10     jmp End
```

2)RELOAD GDT:

Managing System Resources and Loading GDT and IDT:

```
start:
    mov byte[0xb8000],'K'
    mov byte[0xb8001],0xa

    lgdt [Gdt64Ptr]

    retf
End:
    hlt
    jmp End

Gdt64:
    dq 0
    dq 0x0020980000000000

Gdt64Len: equ $-Gdt64


Gdt64Ptr: dw Gdt64Len-1
          dq Gdt64
```

1. GDT and IDT Introduction:

- In the kernel, we start by managing system resources such as the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT).
- The GDT contains segment descriptors that define memory regions and access privileges.
- The IDT contains interrupt gate descriptors for handling interrupts and exceptions.

2. Copying and Loading GDT:

- Copy the GDT code to the kernel.
- Use the "load gdt" instruction to load the GDT pointer.

- The GDT pointer should have a size of 8 bytes (quadword) in 64-bit mode.
- Modify the double-word to a quadword to match the size.

3. Loading Code Segment Descriptor:

- Instead of using a jump instruction, load the code segment descriptor into the CS register.
- Use a "far return" instruction to load the descriptor, indicating a return to the same privilege level.
- A normal return instruction would not load the descriptor in the CS register

```
start:
    lgdt [Gdt64Ptr]

    push 8
    push KernelEntry
    db 0x48
    retf

KernelEntry:
    mov byte[0xb8000],'K'
    mov byte[0xb8001],0xa
```

4. Preparing for Far Return:

- Fabricate a scenario where we are called by another caller and the far return instruction executes.
- Push the return address on the stack manually.
- Define a label, "kernel entry," to represent the location we want to branch to.

5. Stack Preparation for Far Return:

- The top of the stack should contain the code segment selector (8 bytes) and the offset (8 bytes).
- Push the code segment selector value (e.g., 8) onto the stack.
- Save the address of the location we want to branch to and define it as the "kernel entry" label.

6. Executing Far Return:

- Push the necessary data on the stack and execute the return instruction. Note that the default operand size of far return is 32 bits. Add an operand-size override prefix (48) to change the operand size to 64 bits.

7. Kernel Entry Point:

- In the "kernel entry" label, print the character 'K' to signify entry into the kernel. Add an infinite loop to keep the kernel running.

3)Exceptions and interrupts handling

Before looking into interrupt, we will look some terminologies associated with interrupt handling in an Operating System.

1. IRQ (Interrupt Request): An IRQ is a hardware signal sent to the processor to request attention or service. It is used by devices to interrupt the normal execution of the processor and transfer control to a specific interrupt handler. Each device is assigned a specific IRQ line, and when the device requires attention, it asserts its corresponding IRQ line to notify the processor.

2. IRR (Interrupt Request Register): The Interrupt Request Register is a hardware register used in interrupt controllers to store the status of the interrupt requests from various devices. It keeps track of the pending

interrupts and helps the interrupt controller to prioritize and handle them accordingly.

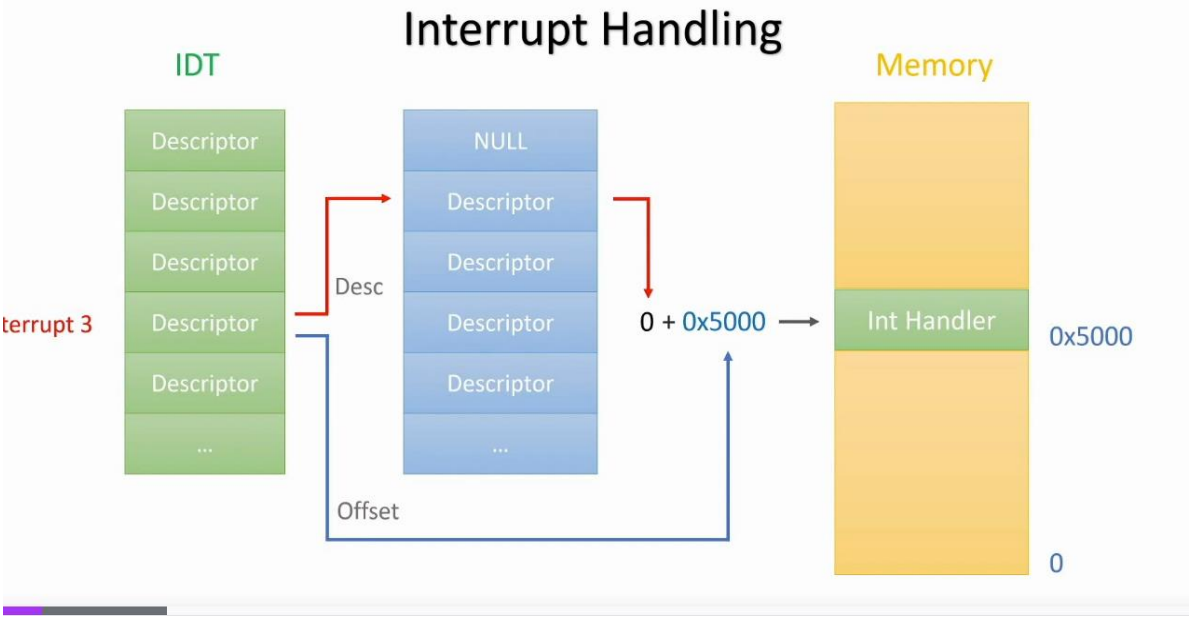
3. IMR (Interrupt Mask Register): The Interrupt Mask Register is a hardware register used in interrupt controllers to enable or disable specific interrupts. By manipulating the bits in the IMR, an operating system or the interrupt controller itself can choose which interrupts to allow and which ones to mask (ignore).

4. ISR (Interrupt Service Routine): An Interrupt Service Routine is a software routine or handler that is executed in response to an interrupt. When an interrupt occurs, the processor saves its current execution state and transfers control to the appropriate ISR associated with that interrupt. The ISR performs the necessary tasks to handle the interrupt, such as reading data from a device, updating system states, or initiating further processing.

5. PIT (Programmable Interval Timer): The Programmable Interval Timer is a hardware device used to generate regular timer interrupts in a computer system. It is commonly used for tasks that require precise timing, such as scheduling, timekeeping, and generating periodic events. The PIT can be programmed to generate interrupts at specific intervals by setting its countdown value.

6. PIC (Programmable Interrupt Controller): The Programmable Interrupt Controller is a device that manages and prioritizes interrupts from multiple devices in a computer HARDWARE INTERRUPT 2 system. It acts as an intermediary between devices and the processor. The PIC receives interrupt requests from devices through their respective IRQ lines, prioritizes them based on their importance, and routes them to the appropriate interrupt handlers (ISRs). The PIC also allows masking or enabling of interrupts through the IMR. These terms are fundamental to the functioning of an operating system, allowing it to

handle interrupts, manage device interactions, and schedule tasks efficiently.



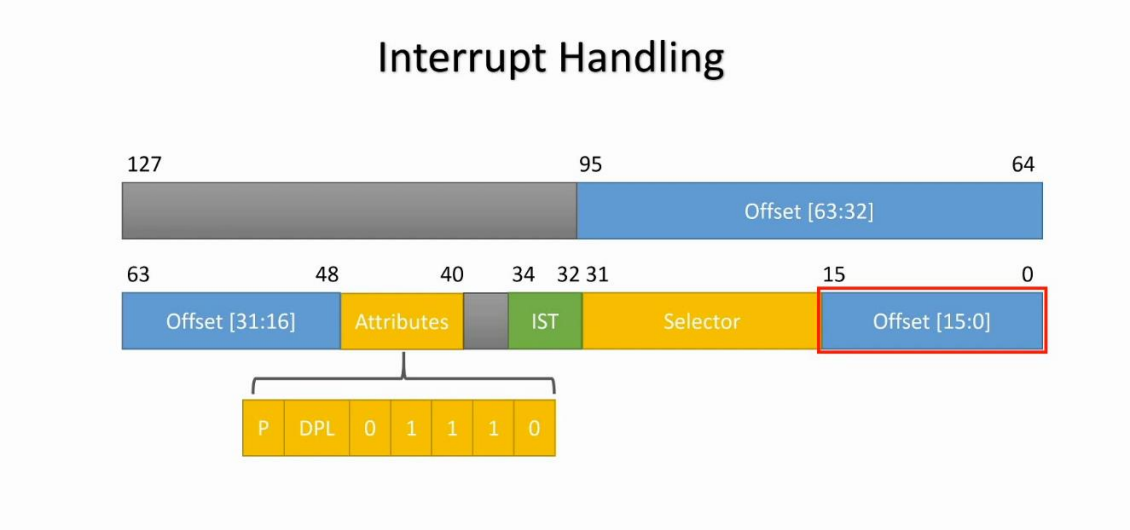
Interrupt handling is an essential part of an operating system as it enables interaction with various hardware devices and facilitates crucial functionalities like process scheduling. Without proper interrupt handling, important peripherals such as the keyboard would be unusable. Additionally, process scheduling heavily relies on timer interrupts to ensure fair and efficient utilization of system resources.

In the realm of interrupts, there are hardware interrupts, such as timer and keyboard interrupts, which are triggered by external devices. On the other hand, exceptions occur as a result of execution errors or internal processor errors. Although interrupts and exceptions have distinct origins, the process of handling them is similar.

To handle interrupts effectively, it is necessary to set up the Interrupt Descriptor Table (IDT) and write specific interrupt service routines (ISRs) to handle each interrupt or exception. When an interrupt is fired, the corresponding IDT entry is selected. This entry contains vital

information about the interrupt handler, including the code segment it resides in, attributes, offset, and other relevant details.

By properly configuring the IDT and writing dedicated ISRs, the operating system can respond to interrupts and exceptions in a controlled manner. This enables the system to perform necessary actions and provide appropriate responses based on the specific interrupt or exception encountered.



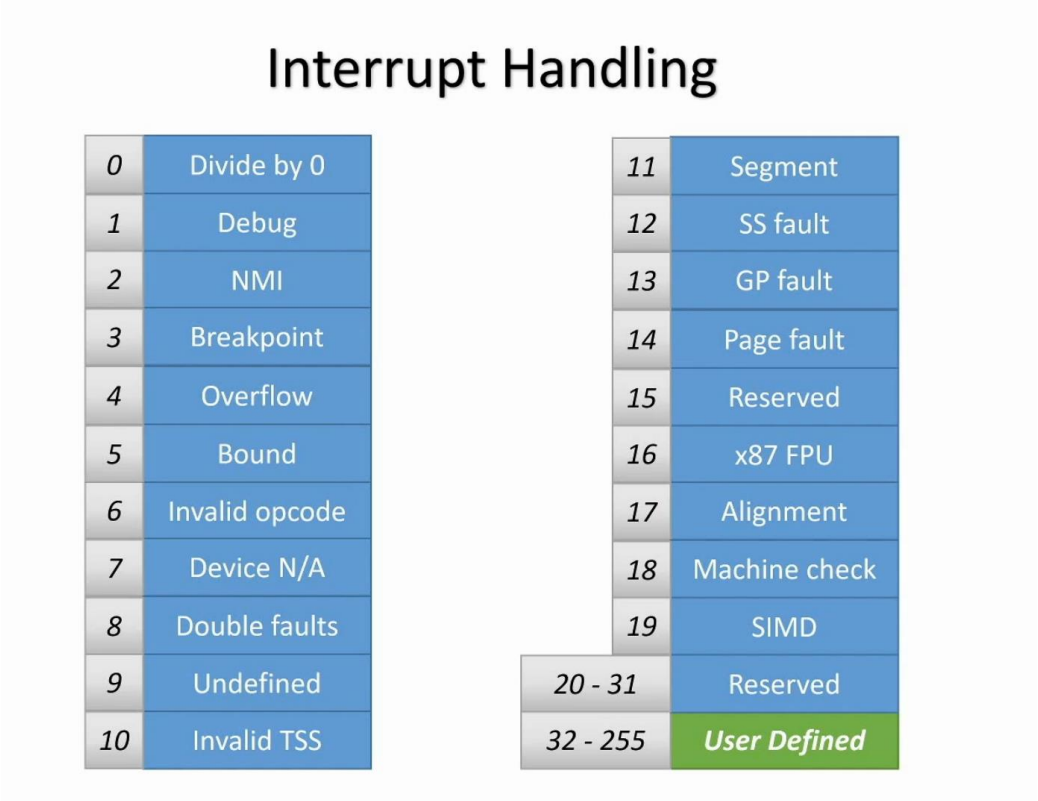
IDT Entry and Interrupt Handler:

- The IDT entry for an interrupt occupies 16 bytes in total.
- The 64-bit offset of the interrupt handler is divided into three parts.
- The lower 16 bits of the offset are stored in the first two bytes.
- Bits 16 to 31 hold the selector of the code segment descriptor where the interrupt handler resides.

IST and Attributes:

- IST (Interrupt Stack Table) is a field in the IDT entry that references a stack pointer.

- Assigning a value to the IST field loads the referenced stack pointer into the RSP register.
- In this course, IST is not used, so it is set to zero (not utilized).
- The only modifiable field in the IDT entry attributes is the DPL (Descriptor Privilege Level) field.
- The present bit is set to 1, and the value 01110 indicates an interrupt descriptor table entry.
- The DPL field specifies the privilege level (CPL) that can access the descriptor.



Interrupt Vectors and Usage:

- There are 256 different exception and interrupt vectors available.
- Interrupt vectors are numbers that identify specific exceptions and interrupts.
- For example, the vector for the divide by zero exception is 0, and 1 represents the debug exception.

- Vectors from 0 to 31 are predefined by the processor and cannot be redefined.
- Vectors from 32 to 255 are user-defined and can be used for hardware and software interrupts.

Example: Divide by Zero Exception Handling:

```
Idt:
    %rep 256
        dw 0
        dw 0x8
        db 0
        db 0x8e
        dw 0
        dd 0
        dd 0
    %endrep

IdtLen: equ $-Idt

IdtPtr: dw IdtLen-1
        dq Idt
```

The structure of the IDT pointer is similar to that of the GDT pointer. The first two bytes store the length of the IDT, and the next 8 bytes store the address of the IDT.In this example, all the fields of the IDT entries are initially set to zero, and their values are changed when necessary in the code. However, the attribute and selector values are the same for all entries in this case, so they are set here.The attribute field is located in the 6th byte of each entry. In binary, the value copied to this field is 8E, which corresponds to 10001110. This indicates that the present bit is set to 1 and the descriptor is an interrupt gate descriptor. The DPL (Descriptor Privilege Level) field is simply set to 0.

The other field of interest is located in the 3rd byte of each entry. In this example, it is set to 8, representing the code segment descriptor that is currently in use.It is important to note that when using an interrupt gate, the jump to the code segment can only occur if it is at the same privilege level or a higher privilege level than the current one. For example, if the DPL of the code segment descriptor referenced by the selector 8 is 3, then this interrupt gate descriptor cannot be used to jump to the handler because the current privilege level is ring 0, which has higher privilege than ring 3. Thus, using an interrupt gate descriptor allows jumping only from a lower privilege level to a higher privilege level segment.

```
start:
    mov rdi,Idt
    mov rax,handler0

    mov [rdi],ax
    shr rax,16
    mov [rdi+6],ax
    shr rax,16
    mov [rdi+8],eax

    lgdt [Gdt64Ptr]
    lidt [IdtPtr]
```

```
ernelEntry:
    mov byte[0xb8000],'K'
    mov byte[0xb8001],0xa

    xor rbx,rbx
    div rbx
          rbx
nd:
    hlt
    jmp End

andler0:
    mov byte[0xb8000],'D'
    mov byte[0xb8001],0xc

    jmp End

    iretq
```

We begin by copying the offset of the handler to the first IDT entry. We move the address of the IDT to the register RDI and store the offset of the handler in RAX. The offset is divided into three parts in the IDT entry. We first copy the lower 16 bits of the offset by executing "move RDI, AX", which copies the lower 16 bits to the first two bytes of the IDT entry. Then we shift the offset in RAX rightward by 16 bits, so the bits 16 to 31 are in the register AX.

We move "RDI + 6, AX" to copy the bits 16 to 31 of the offset to the second part, which is at the 7th byte of the entry. We continue by shifting the offset 16 bits again, so the bits 32 to 63 are now in register EAX. We move "RDI + 8, EAX" to copy the value in EAX to the third part of the offset. With the IDT set up and loaded, we proceed to write the handler, which is named Handler0. The handler simply prints the character 'D' on the screen to indicate that the divide by zero exception is handled by this service routine. We copy the code for printing characters and set the character to 'D' with the attribute value of 'c' to make it appear in red.

The return instruction used in the handler is different from the normal return instruction in procedures. We use the "interrupt return" instruction, which pops more data and can return to a different privilege level. We will manually jump to ring 3 using this instruction later. For now, we include it in the handler code. Lastly, we stop the system by jumping to the end, as it is important to halt the kernel and print an error message if there is an exception in privilege level 0 where the system kernel runs. To trigger a divide by zero error, we use the "div" instruction with a register that contains zero, such as RBX. If everything is set correctly, we will see the character 'D' printed on the screen in red. The result should be the appearance of the red character 'D' on the screen. If any error then K will be printed.

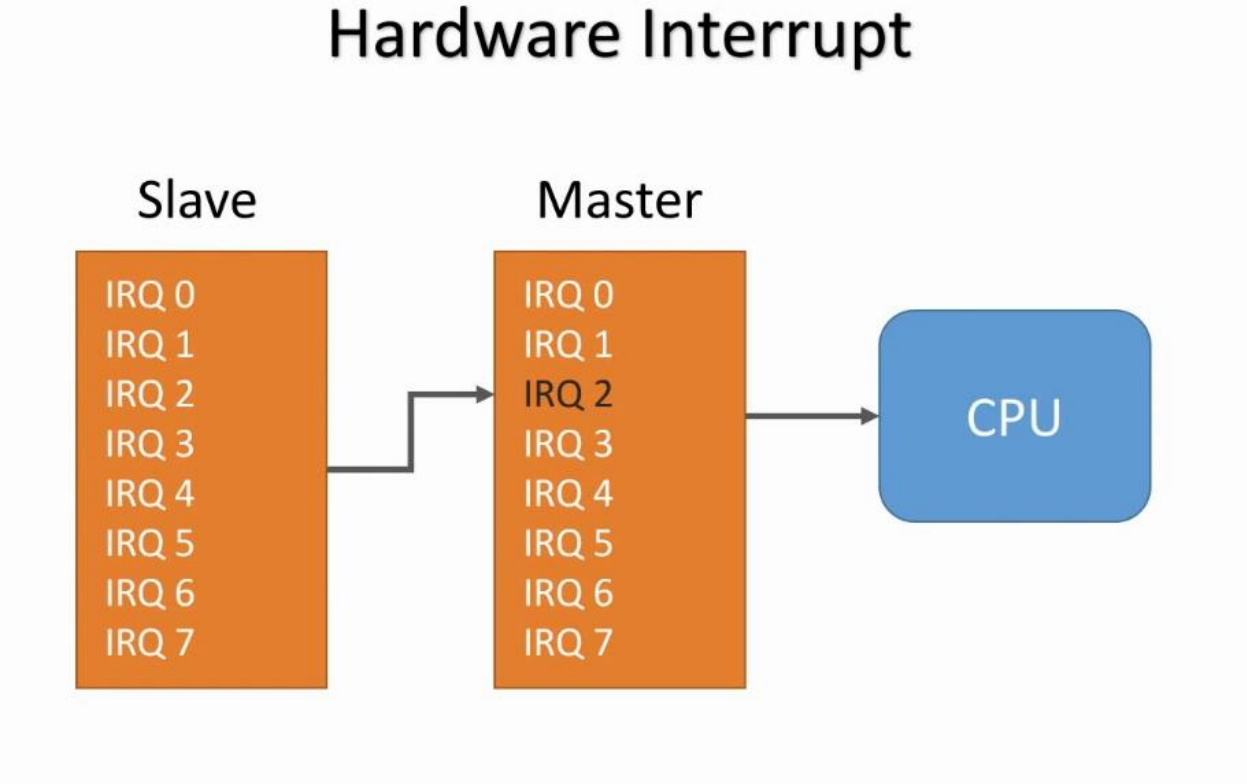
4)SAVING THE FILES:

```
    jmp End

Handler0:
    push rax
    push rbx
    push rcx
    push rdx
    push rsi
    push rdi
    push rbp
    push r8
    push r9
    push r10
    push r11
    push r12
    push r13
    push r14
    push r15

    mov byte[0xb8000],'D'
    mov byte[0xb8001],0xc

    jmp End
```

During interrupt or exception handling, it is important to save the state of the CPU. The "push" instructions are used to save the state when an interrupt or exception occurs. Only certain registers, such as the rip and rsp registers, are saved in this process. In the handler, registers are used and modified to perform specific tasks. As a result, when returning from the handler, the values of the registers may have changed, and the state of the CPU is not the same as it was before the interrupt occurred. To restore the CPU's state after interrupt handling is complete, we store all the general-purpose registers on the stack. In 64-bit mode, there are 16 general-purpose registers, and RSP is pushed on the stack when the CPU calls the handler. Therefore, we push the other 15 registers onto the stack. After handling the interrupt, we pop the original values from the stack and return. Since the stack operates as a last-in-first-out structure, we pop the values in reverse order. When we return from the

handler, the CPU resumes its previous work as if the interrupt never happened.

5)Setting up the interrupt controller:

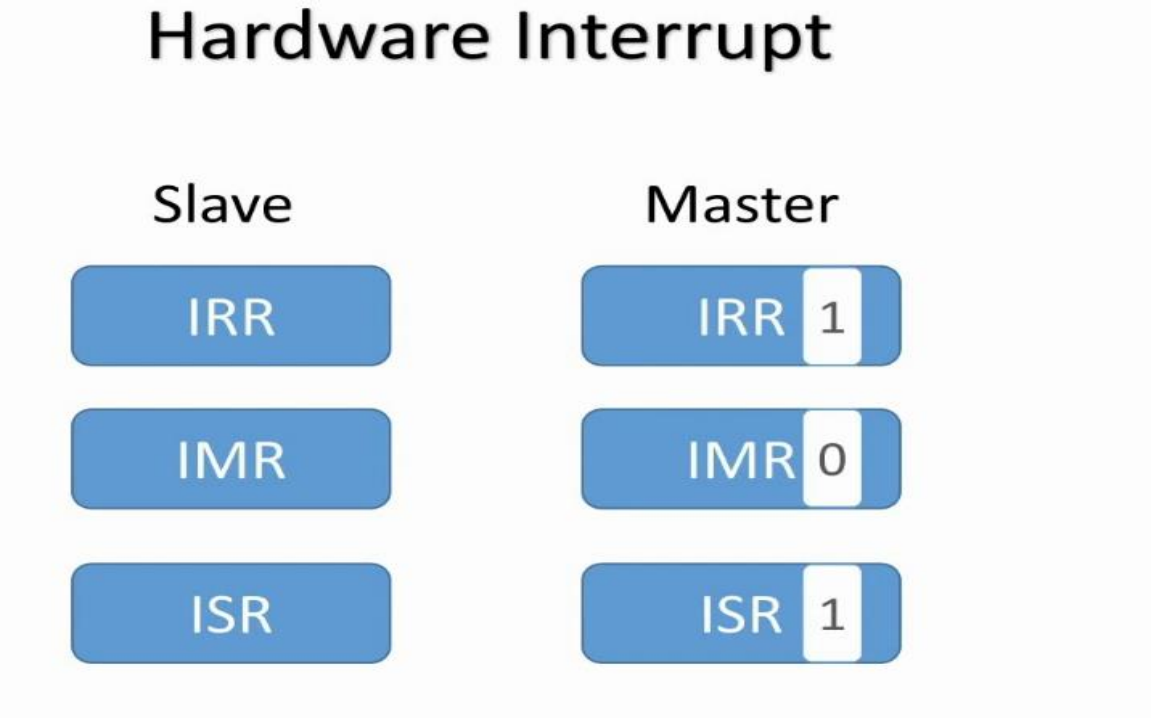


Programmable Interrupt Controller (PIC). The PIC is a device that manages multiple hardware interrupts and directs them to the appropriate interrupt handlers. It is typically used in systems with multiple devices that generate interrupts, such as servers, workstations, and embedded systems.

The PIC is programmed by the operating system to identify the priority levels of different interrupts and assign them to specific interrupt service routines. This allows the operating system to handle multiple interrupts simultaneously and ensure that critical events are

handled first. In modern computer systems, the PIC has been largely replaced by more advanced interrupt controllers, such as the Advanced Programmable Interrupt Controller (APIC) and the System Management Interrupt (SMI) controller.

However, the basic principles of interrupt-driven computing remain the same, and the PIC remains an important part of computer history.



PIC has 2 chips each with 8 IRQ signals linked together. These chips are called slave and master. The slave chip is linked to master through IRQ2. There are total of 15 interrupts.

PROGRAMMABLE INTERRUPT TIMER In addition to the Programmable Interrupt Controller (PIC), another important type of interrupt controller is the Programmable Interrupt Timer (PIT). The PIT is a hardware component that generates periodic interrupts at a fixed frequency, typically in the range of tens to hundreds of Hertz. The PIT is

used for a variety of purposes, such as scheduling tasks, measuring time intervals, and generating system ticks.

When the PIT generates an interrupt, the HARDWARE INTERRUPT 4 operating system's interrupt handling routine is invoked, allowing the system to perform the necessary tasks. Like the PIC, the PIT is programmed by the operating system to specify the frequency and other parameters of the interrupt signal. This allows the operating system to use the PIT for a wide range of purposes, making it an essential component of modern computer systems. In conclusion, the Programmable Interrupt Timer (PIT) is an important hardware component that generates periodic interrupts for a variety of purposes in modern computer systems. Understanding how the PIT works is essential for anyone working in the field of computer hardware or software development.

The PIT uses IRQ0 of master chip, keyboard uses IRQ1. Processor finds the handler for that interrupt by the vector number. There are 3 registers in each chip. These registers are used to service interrupts. 8-bit registers corresponding to IRQ. Example: When keyboard is pressed , it sends a signal to ther chip and the corresponding bit in the master IRR register is set.

IRR- Interrupt Request register

IMR- Interrupt Mask Register

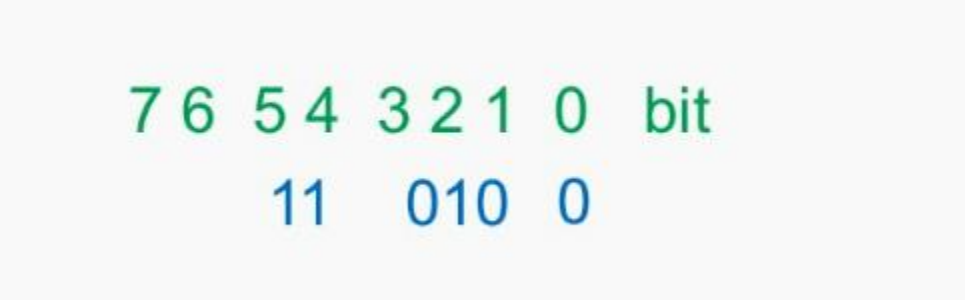
ISR- Interrupt service register

NOTE: If the corresponding bit in the interrupt mask register IMR is 0 meaning that the IRQ is not masked, the PIC will send the interrupt to the processor and set the corresponding bit in the in-service register ISR. Generally, we could have multiple IRQs come at the same which means more than one bit in the IRR will be set. The PIC will choose

which one should be processed first according to the priority. Normally, the IRQ0 has the highest priority and IRQ7 the lowest priority.

Let’s start with the code part for that we have to initialize,

we initialize the PIT and PIC which requires several steps. First, we look at the PIT. There are three channels in the PIT, through channel 0 to channel 2. Channel 1 and 2 may not exist and we don’t use them in our system. So, we only talk about channel 0. The PIT has four registers, one mode command registers and three data registers for the three channels respectively. We set the command and data registers to make the PIT works as we expect, that is fire an interrupt periodically.



The mode command register of the Programmable Interval Timer (PIT) consists of four parts. Bit 0 indicates whether the value used by the PIT is in binary or BCD (Binary Coded Decimal) form. In this case, we set it to 0, indicating binary form. Bits 1 through 3 represent the operating mode, which determines how the PIT operates. In this example, we set it to 010, which corresponds to mode 2, known as the rate generator mode used for generating reoccurring interrupts. Bits 4 to 5 represent the access mode, which determines the order in which data is written to the data register of the PIT. Since the data registers are 8 bits, if we want to write a 16-bit value, we need to write two bytes in a specific order. In this example, we set the access mode to 11, indicating that we want to write the low byte first and then the high byte. The last part of the mode command register is used for selecting the channel. In this case, we only use channel 0, so we simply set it to 0. The value of the

mode command register is moved to the AL register for further processing.

```
InitPIT:
    mov al,(1<<2)|(3<<4)
    out 0x43,al

    mov ax,11931
    out 0x40,al
    mov al,ah
    out 0x40,al
```

So we define the label init pit. The PIT works simply like this, we load a counter value and PIT will decrement this value at a rate of about 1.2 mega HZ which means it will decrement the value roughly 1.2 million times per second. In our system, we want the interrupt to be fired at a frequency of 100 Hz, which means 100 times per second. To achieve this, we perform a simple calculation by dividing 1.2 million (the standard frequency of the PIT) by 100.

The result, 11,931, is the count value that we want to write to the PIT's data register. Since the counter value is a 16-bit value, we write the low byte first and then the high byte.To set the count value, we move the value 11,931 to the AX register. The address of the data register for channel 0 of the PIT is 40, so we use the OUT instruction to write the low byte (AL) of the value to the data register at address 40. Then we move the high byte (AH) of the value to AL, and again use the OUT instruction to write it to the data register at address 40.Now that the PIT is running with the desired count value, the next step is to set up the interrupt controller.

```
InitPIC:
    mov al,0x11
    out 0x20,al
    out 0xa0,al

    mov al,32
    out 0x21,al
    mov al,40
    out 0xa1,al

    mov al,4
    out 0x21,al
    mov al,2
    out 0xa1,al

    mov al,1
    out 0x21,al
    out 0xa1,al
```

Initialization Command Word 4

7 6 5 4 3 2 1 0 bit

00 00 00 0 1

To set up the interrupt controller, we write specific command words to the command registers of both the master and the slave. We move the value 11 to AL and use the OUT instruction to write it to the command register of the master at address 0x11. Similarly, we move the value 11 to AL and write it to the command register of the slave at address 0xa1.

Next, we specify the starting vector number for the IRQs. For the master, we move the value 32 to AL and write it to the data register at address 0x21. For the slave, we move the value 40 to AL and write it to the data register at address 0xa1.

We then set the command word for the connection between the master and the slave. We move the value 4 to AL and write it to the data register at address 0x21.

The command word for the slave, indicating its identification, is set to 2. We move the value 2 to AL and write it to the data register at address 0xa1.Finally, we set the command word for selecting the mode. We move the value 1 to AL and write it to the data register at address 0x21.After setting up the interrupt controller, we mask all IRQs except IRQ0 of the master, which is used by the PIT. To mask an interrupt, we

set the corresponding bit of the IRQ and write the value to the data register. In this case, we set bits 1 through 7 and write the value to the data register at address 0x21. Since the slave IRQs are not used in this course, we set all the bits and write the value to the data register at address 0xa1.

Next, we set the IDT entry for the timer. The vector number of the timer is set to 32 in the PIC, so the address of the entry in the IDT is calculated as the base of the IDT plus 32 multiplied by 16 (since each entry takes up 16 bytes).

We write the code for the timer handler, which follows the standard process of interrupt handling. We save the state of the CPU by pushing the registers on the stack, and before the handler returns, we use the interrupt return instruction to restore the state by popping the registers back. Note: It is possible to wrap the code for setting the IDT entry as a function and call that function to set the entry. The same applies to the timer handler code.

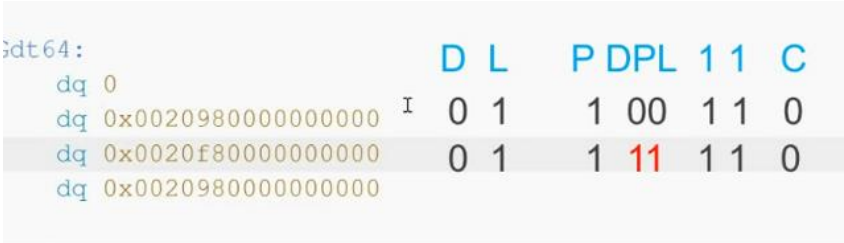
```
mov rax,Timer
mov [rdi],ax
shr rax,16
mov [rdi+6],ax
shr rax,16
mov [rdi+8],eax
```

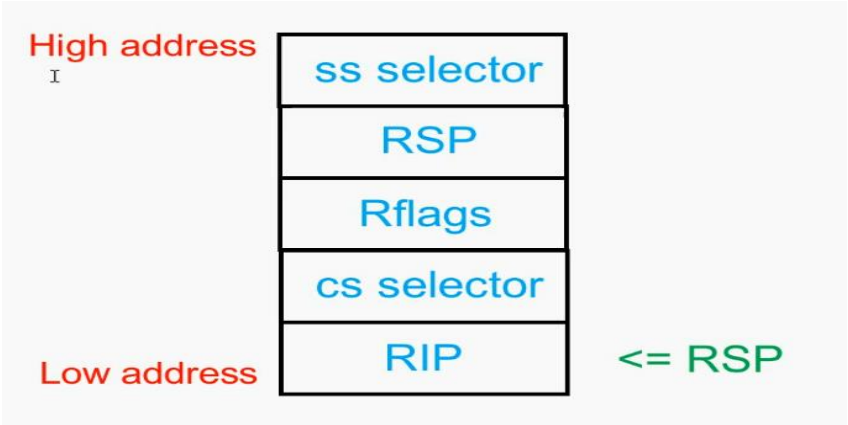
To test the timer handler, we can print the character 'T' to indicate that we have entered the timer interrupt. We can choose a different location and color for printing, such as yellow, to differentiate it. After printing, we jump to the end of the code.

Finally, to enable interrupts, we use the "set interrupt flag" instruction. It is important to note that when we switch from real mode to protected mode in the loader file, we disabled interrupts. Therefore, at this point, after setting up the timer and the interrupt controller, we need to enable interrupts in order to respond to interrupt requests.

6)Getting into the Ring3:

The CPL is stored in the lower 2 bits of cs and ss registers. We are running in ring 0 so far. If we check the lower 2 bits of cs register we will see the value is 0 meaning that we are running in ring0. What we need to do is prepare the code segment descriptor for ring3 and load the descriptor to cs register. Once the descriptor is successfully loaded into cs register, we will run in ring3. In this process, the data segment descriptor for ss register is needed. This is the only case where we load the data segment descriptor ourselves.





To return from ring 0 to ring 3, we need to prepare 5 8-byte data on the stack in reverse order:

1. The top of the stack should contain the RIP value, which specifies the address where we will return.

2. The next data should be the code segment selector that will be loaded into the CS register after the return.

3. The value of RFLAGS, which contains the status of the CPU, should be stored next. When we return, this value will be loaded into the RFLAGS register.

4. The stack pointer (RSP) should be stored in the next location on the stack.

5. The last data should be the stack segment selector. In this example, the stack segment selector is 18 in hexadecimal, so the descriptor being referenced is the fourth one. The RPL (Requested Privilege Level) is set to 3.

By pushing these 8-byte data items onto the stack in reverse order, we can use the "interrupt return" instruction to transition from ring 0 to ring 3.

```
push 18|3
push 0x7c00
push 0x2
push 0x10|3
push UserEntry
iretq
```

To return from ring 0 to ring 3, we prepare 5 8-byte data items on the stack. These data items are pushed onto the stack in reverse order.First, we push the value 18 OR 3, which combines the stack segment selector (18 in hexadecimal) and the Requested Privilege Level (RPL) set to 3. This ensures that the stack segment is correctly loaded when returning to ring 3.

Next, we push the value 7c00, representing the stack pointer. In this example, the stack pointer is set to 7c00.After that, we push the value 2, which sets the necessary flag in the RFLAGS register. Specifically, we set bit 1 to 1 while keeping other flags such as the carry flag and overflow flag at 0.Then, we push the value 10 OR 3, which combines the code segment selector (10 in this case) and the RPL set to 3. This ensures that the code segment is correctly loaded when returning to ring 3.

Finally, we push the return address, which is referred to as "user entry" in this context. This address indicates the location in the code where we want to resume execution in ring 3.Once these 8-byte data items are pushed onto the stack, we can execute the "interrupt return" instruction. This instruction will set the stack pointer (RSP) to 7c00 and jump to the "user entry" address. The CS and SS registers will be loaded with the appropriate descriptors based on the values we pushed, transitioning the execution from ring 0 to ring 3.

It's important to note that the specific label "user entry" mentioned here needs to be defined and implemented in the code to indicate the desired entry point for resuming execution in ring 3.

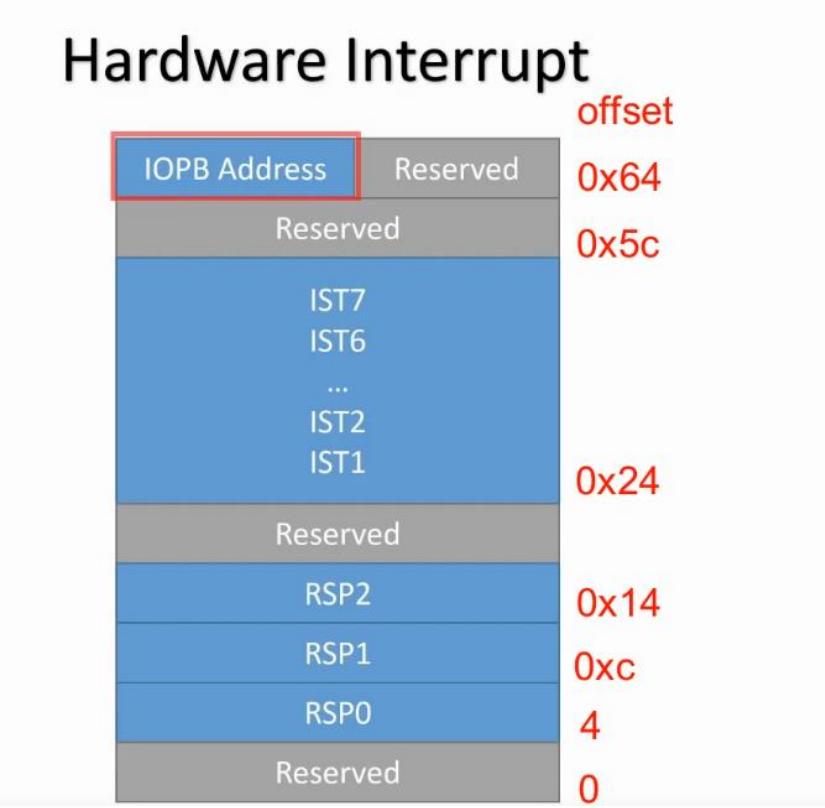
```
UserEntry:
    mov ax,cs
    and al,11b
    cmp al,3
    jne UEnd

    mov byte[0xb8010],'U'
    mov byte[0xb8011],0xE

UEnd:
    jmp UEnd
```

To determine the current privilege level, we check the lower 2 bits of the CS register. We move the value of CS into the AX register and then set AL to 11 in binary using an instruction that preserves the lower 2 bits of AL while clearing other bits. If the value in AL is now 3, it indicates that we are running in ring 3.

We compare AL with 3, and if they are not equal, we jump to the label "UEnd". It's important to note that the halt instruction cannot be executed in ring 3. Therefore, in the "UEnd" section, we simply jump to the current location, effectively ending the program.

If we are in ring 3, we print the character 'U' to indicate that we are running in user mode. Additionally, the displayed number should be in hexadecimal format. As a side note, we comment out the STI (Set Interrupt Flag) instruction because we do not want to enable interrupts in this context.

7)INTERRUPT HANDLERS IN RING 3:



1. **Setting up TSS and Transferring Control to Ring 0:**
    - When running in ring 3 and an interrupt is fired, control is transferred to the interrupt handler in ring 0.
    - The interrupt handler's segment descriptor has a DPL (Descriptor Privilege Level) of 0.
    - During the transfer, the SS segment register is loaded with a null descriptor by the processor.
    - The RSP register is also loaded with a new value.

2. **Task State Segment (TSS) Structure:**

    - The TSS is a data structure that contains various fields.
    - The field of interest for us is RSP0, which holds the value to be loaded into RSP when control transfers to ring 0.

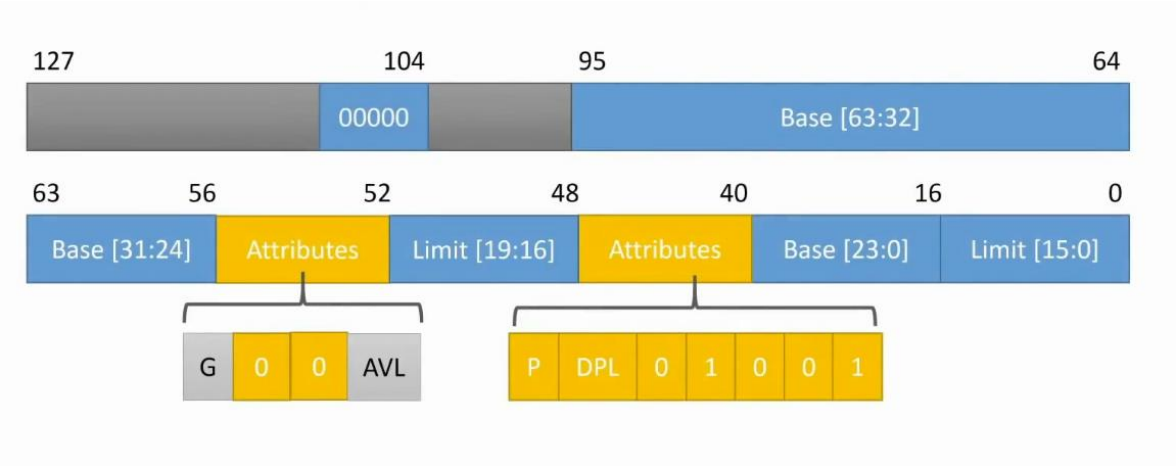
3. **Setting up TSS Fields:**

- We need to set up the TSS to specify the new value of RSP0.
- Since we don't use ring 1 and ring 2, we don't set those fields in the TSS.
- The IST (Interrupt Stack Table) fields in the TSS are not used in our system as our interrupt descriptors have IST fields set to 0.

4. **I/O Permission Bitmap:**

- The TSS also includes the address of the I/O Permission Bitmap.
- The I/O Permission Bitmap is used for protecting I/O port addresses.
- However, in our system, we don't use the I/O Permission Bitmap.

By setting up the TSS and specifying the desired value for RSP0, we enable smooth control transfer from lower privilege levels to ring 0 when an interrupt occurs in our system.



1. **TSS Descriptor in GDT:**

- The TSS descriptor is stored in the Global Descriptor Table (GDT).
- Unlike code segment and data segment descriptors, the base and limit attributes in the TSS descriptor are used.
- It is important to set the base address and limit correctly when setting up the TSS descriptor.

- The attributes of the TSS descriptor include a value indicating a 64-bit TSS descriptor, P (Present) flag, DPL (Descriptor Privilege Level), G (Granularity), and an available bit.
- In our system, the G and available bit are not used, so they are set to 0.
- The upper part of the descriptor contains reserved bits and 0s.

2. **TSS Selector:**

- In addition to the TSS descriptor, we also need a selector to reference the TSS.
- Loading the TSS selector is different from loading the code segment selector.
- We use the "Load Task Register" instruction to load the TSS selector.
- Once the selector is loaded, the processor uses it to locate the TSS descriptor in the GDT.

By correctly setting up the TSS descriptor in the GDT and loading the TSS selector, we ensure that the processor can access and use the TSS for controlling privilege level transitions during interrupt handling.

```
Tss:
    dd 0
    dq 0x150000
    times 88 db 0
    dd TssLen

TssLen: equ $-Tss
```

To define the Task State Segment (TSS) structure, we begin by creating a structure called TSS. The first four bytes of the structure are reserved, and we assign a value of 0 to them. The next field in the structure is RSP0, which is the field where we set the new address for the RSP. In this example, we want the RSP to be loaded with the address 150000 when the interrupt handler is called. The remaining fields in the TSS structure are not used in our system, so we simply set them to 0. To define these fields, we use the directive "times." The last field in the TSS structure is the address of the IO permission bitmap, but in our system, we assign the size of the TSS to it, indicating that the IO permission bitmap is not used. Finally, we define the length of the TSS structure.

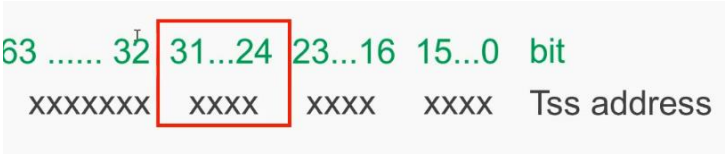
By defining the TSS structure and setting the appropriate values, we ensure that the TSS is properly configured for managing privilege level transitions during interrupt handling.

```
Gdt64:
    dq 0
    dq 0x0020980000000000
    dq 0x0020f80000000000
    dq 0x0000f20000000000
TssDesc:
    dw TssLen-1
    dw 0
    db 0
    db 0x89
    db 0
    db 0
    dq 0
```

In the Global Descriptor Table (GDT), we add a new entry called the TSS descriptor. This entry helps us easily identify and reference the TSS. The first two bytes of the TSS descriptor represent the lower 16 bits of the

TSS limit. We set this value to the length of the TSS minus 1. The base address of the TSS is currently set to 0, as we will assign the actual address in the code. The next three bytes are set to 0, representing the lower 24 bits of the base address. The following byte is the attribute field, where we assign the value 89. In binary format, this value signifies that the present bit is 1, the DPL (Descriptor Privilege Level) is 0, and the 0,1,0,0,1 sequence indicates that this is a 64-bit TSS. The remaining fields in the TSS descriptor are set to zero. By defining the TSS descriptor in the GDT, we establish the necessary information for accessing and managing the TSS structure.

```
SetTss:
    mov rax,Tss
    mov [TssDesc+2],ax
    shr rax,16
    mov [TssDesc+4],al
    shr rax,8
    mov [TssDesc+7],al
    shr rax,8
    mov [TssDesc+8],eax

    mov ax,0x20
    ltr ax
```



At the start of the code, we begin by loading the TSS descriptor. Firstly, we copy the address of the TSS structure to the descriptor. We move the value of the TSS structure into the rax register using "move rax, tss". The lower 16 bits of the address are stored in the third byte of the TSS descriptor. We move the value from "tss descriptor plus 2" into ax. Next,

we shift the value in rax right by 16 bits, so that the bit 16 to 23 of the address are now in register al. We then move this value to "tss descriptor plus 4" to set the fifth byte of the descriptor. Continuing, we shift rax right by 8 bits to prepare for the next part of the base address, which is in the eighth byte of the descriptor. We move the value from "tss descriptor plus 7" into al to set the eighth byte. The upper 32 bits of the address are in the next location, so we shift rax right by 8 bits. Eax now holds the upper 32 bits of the address, and we move it to "tss descriptor plus 8" to complete the descriptor setup.

With the TSS descriptor properly set, we can proceed to load the selector using the "load task register" instruction. The selector we use is 20 in hexadecimal, as it corresponds to the fifth entry in the GDT where the TSS descriptor resides. We move the value 20 into ax and then load the task register with "ax" to load the TSS selector.

At this point, the TSS descriptor is properly configured, and the TSS selector is loaded, allowing us to access and manage the TSS structure.

```
push 0x18|3
push 0x7c00
push 0x202
push 0x10|3
push UserEntry
iretq
```

63 ..... 9 .... 1 0 bit

0 ..... 1 .... 1 0 = 0x202

In the previous instruction set, instead of pushing the value 2 to enable interrupts, we should push the value 202. This value indicates that the interrupt flag is set, enabling interrupts. Bit 9 of the rflags register corresponds to the interrupt flag. Therefore, we push the value 202 to the stack, which will be loaded into rflags upon returning to user entry, effectively enabling interrupts.

Once we reach the user entry point, and if everything goes as expected, the value of rflags will be loaded with the value 202, enabling

interrupts. When the timer interrupt is fired, the execution is interrupted, and the processor responds by invoking the timer handler. In this case, we expect to see the character "T" printed as the timer interrupt is handled successfully.

Overall, with the correct value pushed to enable interrupts, and the timer interrupt firing, we can expect to observe the characters "K", "U", and "T" printed, validating that the interrupt handling mechanism is functioning correctly.

```
inc byte[0xb8020]
mov byte[0xb8021],0xe

mov al,0x20
out 0x20,al
```

By using this code and removing jump we can see multiple times T is printed indicating that timer interrupt, in the before code actually only once T is printed and it is jumping to end. To know if the interrupt is executed multiple times we use inc and remove jump.

```
UserEntry:

    inc byte[0xb8010]
    mov byte[0xb8011],0xF

UEnd:
    jmp UserEntry
```

Even in the user entry we have to change increment.

7) Spurious interrupt Handling

In a computer system, interrupts are signals that indicate an event or request for attention from a device or software component. Interrupts allow the system to respond quickly to external events and prioritize tasks. However, sometimes interrupts can occur unexpectedly, even when there is no corresponding event or request. These unexpected interrupts are known as spurious interrupts.

Spurious interrupts can arise from various sources, including electrical noise, signal interference, timing issues, or hardware glitches. These sources can introduce transient changes or fluctuations in the interrupt signal, leading to false triggers of the interrupt handling mechanism.

When a spurious interrupt occurs, the system behaves as if a genuine interrupt has occurred. The interrupt controller signals the processor, which then suspends the current task and transfers control to the appropriate interrupt handler routine. However, since there is no actual event or request to service, the interrupt handler may execute unnecessary operations or perform incorrect actions.

Handling spurious interrupts poses a challenge because they can disrupt the normal flow of execution and lead to unpredictable behavior. To mitigate the impact of spurious interrupts, systems employ several techniques:

1. **Interrupt Filtering:** The interrupt controller or the hardware responsible for receiving interrupts may include mechanisms to detect and filter out spurious interrupts. These mechanisms can involve additional checks and criteria to determine the validity of an interrupt before it is acknowledged by the system.
2. **Interrupt Debouncing:** In scenarios where spurious interrupts are caused by electrical noise or signal fluctuations, interrupt debouncing techniques can be employed. Debouncing circuits or algorithms ensure

that only stable and valid interrupt signals are recognized, filtering out transient or noisy signals. HARDWARE INTERRUPT 11

3. Interrupt Priority and Masking: System designers can assign priorities to interrupts and use interrupt masking mechanisms to suppress or disable lower-priority interrupts temporarily. This approach helps prevent spurious interrupts from disrupting critical interrupt handling routines.

4. Interrupt Verification: The interrupt handler routine itself can incorporate additional checks to verify the legitimacy of the interrupt source. By examining the status registers or polling the interrupting device, the handler can confirm whether the interrupt is genuine or spurious. If determined to be spurious, the handler can ignore the interrupt or take appropriate action to prevent undesired consequences.

It is essential to address spurious interrupts to ensure the reliability and stability of interrupt-driven systems. By employing appropriate hardware and software techniques, system designers can minimize the impact of spurious interrupts and maintain the integrity of interrupt handling routines