Unit V Multitasking and Virtual 8086 Mode

Multitasking- Task State Segment, TSS Descriptor, Task Register, Task Gate Descriptor, Task Switching, Task Linking, Task Address Space.

Virtual Mode – Features, Memory management in Virtual Mode, Entering and leaving Virtual mode.

Introduction of Multitasking :

Multitasking is the ability of a computer to run more than one program or task at the same time.

To provide efficient, protected multitasking, the 80386 employs several special data structures.

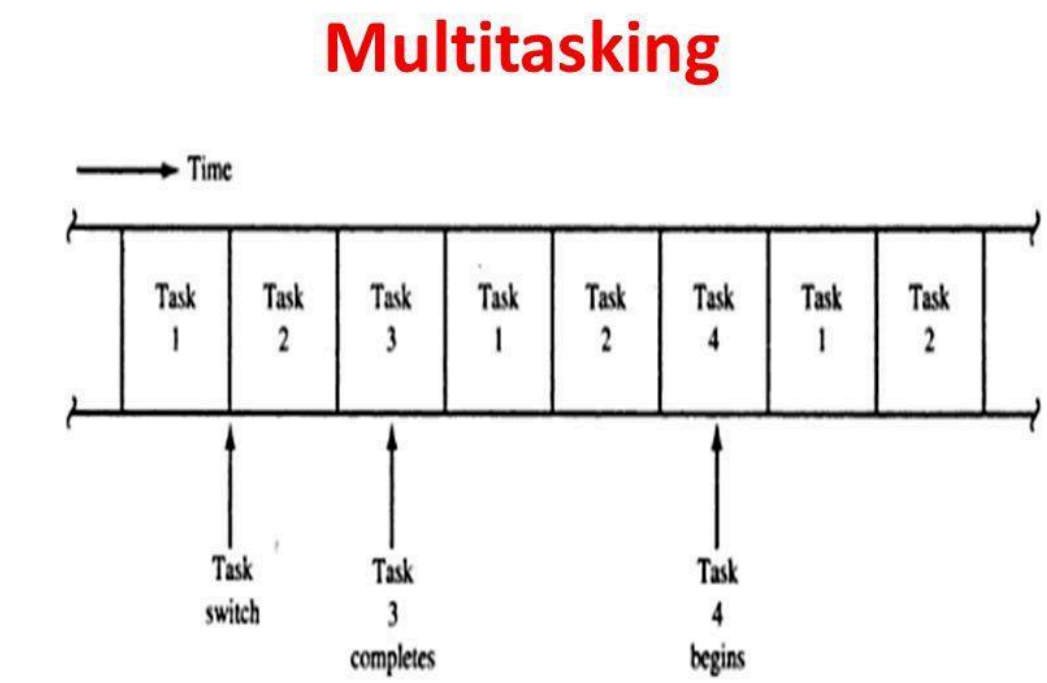


FIGURE 11.12 Running multiple tasks simultaneously

It does not, however, use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures.

The registers and data structures that support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later. In addition to the simple task switch, the 80386 offers two other task-management features:

1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.
2. With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

Task State Segment :

Q. Draw and Explain the Task State Segment of 80386.

Figure 7-1. 80386 32-Bit Task State Segment

31 — 23 — 16	15 — 7 — 0	
I/O MAP BASE	0 0 0 0 0 0 0 0 0 0 0 0 0 | T	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LDT	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
INSTRUCTION POINTER (EIP)		20
CR3 (PDPR)		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	8
ESP0		4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	BACK LINK TO PREVIOUS TSS	0

NOTE
0 MEANS INTEL RESERVED. DO NOT DEFINE.

Fig. shows the format of a TSS.It is a special type of segment, used to manage the task.

The processor state information needed to restore a task is saved in a system segment called the task-state segment (TSS)

The 80386 uses TSS like a scratch-pad. It stores everything it needs to know about the task in TSS.

TSS is not accessible to the general user program or program even privilege level 0.

The fields within TSS are accessible to only 80386.

The fields of a TSS belong to two classes:

1) Dynamic Set 2) Static set

1. A dynamic set that the processor updates with each switch from the task. This set includes the fields that store:
    - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).
    - The segment registers (ES, CS, SS, DS, FS, GS).
    - The flags register (EFLAGS).
    - The instruction pointer (EIP).
    - The selector of the TSS of the previously executing task (updated only when a return is expected).

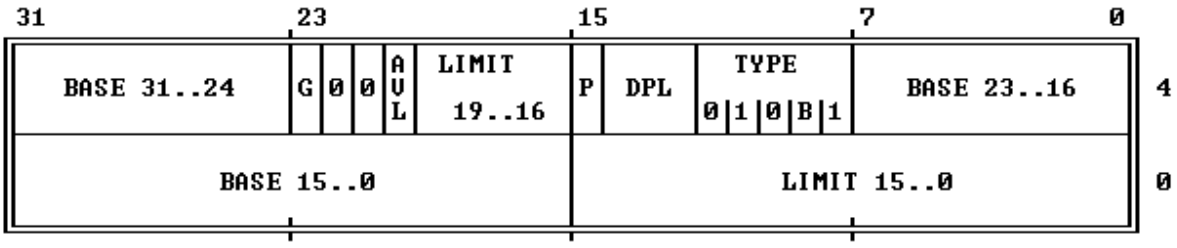
2. A static set that the processor reads but does not change. This set includes the fields that store:
    - The selector of the task's LDT.
    - The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
    - Pointers to the stacks for privilege levels 0-2.
    - The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs
    - The I/O map offset.

TSS Descriptor :

Q.Explain the TSS descriptor of 80386 with a neat diagram.

The task state segment, like all other segments, is defined by a descriptor.

Figure 7-2. TSS Descriptor for 32-bit TSS



The B-bit in the type field indicates whether the task is busy.

A type code of 9 indicates a non-busy task; a type code of 11 indicates a busy task. Tasks are not reentrant.

The B-bit allows the processor to detect an attempt to switch to a task that is already busy.

The BASE, LIMIT, and DPL fields and the G-bit and P-bit have functions similar to their counterparts in data-segment descriptors.

The LIMIT field, however, must have a value equal to or greater than 103. An attempt to switch to a task whose TSS descriptor has a limit less that 103 causes an exception.

A larger limit is permissible, and a larger limit is required if an I/O permission map is present.

A larger limit may also be convenient for systems software if additional data is stored in the same segment as the TSS.

A procedure that has access to a TSS descriptor can cause a task switch.

In most systems the DPL fields of TSS descriptors should be set to zero, so that only trusted software has the right to perform task switching.

Having access to a TSS-descriptor does not give a procedure the right to read or modify a TSS.

Reading and modification can be accomplished only with another descriptor that redefines the TSS as a data segment.

An attempt to load a TSS descriptor into any of the segment registers (CS, SS, DS, ES, FS, GS) causes an exception.

TSS descriptors may reside only in the GDT.

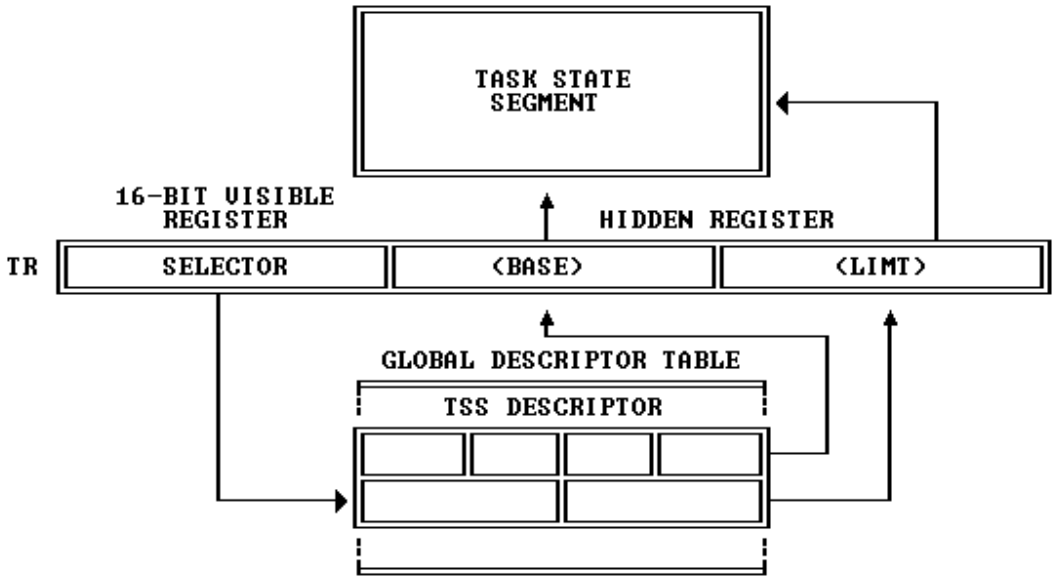
An attempt to identify a TSS with a selector that has TI=1 (indicating the current LDT) results in an exception.

Task Register :

Q.Explore the role of Task Register in multitasking and the instructions used to modify and read Task Register.

Task Register TSS Descriptor may only be loaded into GDT.

Figure 7-3. Task Register



The Task Register (TR) identifies the currently executing task by pointing to the TSS. Figure 7-3 shows the path by which the processor accesses the current TSS.

The task register has both a "visible" portion (i.e., can be read and changed by instructions) and an "invisible" portion (maintained by the processor to correspond to the visible portion; cannot be read by any instruction).

The selector in the visible portion selects a TSS descriptor in the GDT.

The processor uses the invisible portion to cache the base and limit values from the TSS descriptor.

Holding the base and limit in a register makes execution of the task more efficient, because the processor does not need to repeatedly fetch these values from memory when it references the TSS of the current task.

Q.Instructions used to modify and read Task Register :

The instructions LTR and STR are used to modify and read the visible portion of the task register.

Both instructions take one operand, a 16-bit selector located in memory or in a general register.

LTR **(Load task register)**

loads the visible portion of the task register with the selector operand, which must select a TSS descriptor in the GDT.

LTR also loads the invisible portion with information from the TSS descriptor selected by the operand.

LTR is a privileged instruction; it may be executed only when CPL is zero. LTR is generally used during system initialization to give an initial value to the task register; thereafter, the contents of TR are changed by task switch operations.

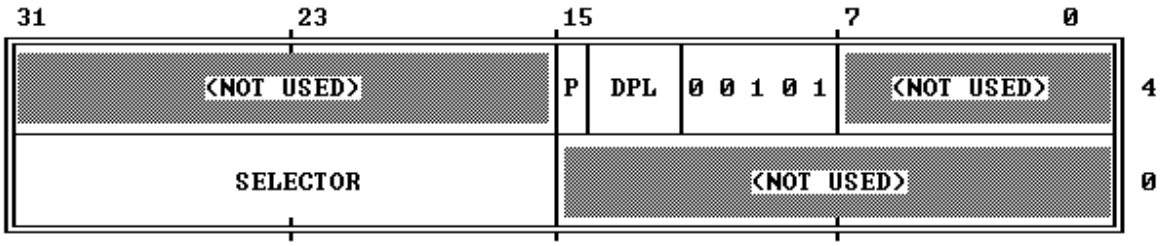
STR **(Store task register)**

It stores the visible portion of the task register in a general register or memory word. STR is not privileged instruction.

Task Gate Descriptor :

A task gate descriptor provides an indirect, protected reference to a TSS.
Figure 7-4 illustrates the format of a task gate.

Figure 7-4. Task Gate Descriptor



The SELECTOR field of a task gate must refer to a TSS descriptor. The value of the RPL in this selector is not used by the processor.

TSS descriptors must reside in the GDT. Task Gate descriptors, on the other hand, may reside in the GDT, an LDT, or the IDT (Interrupt Descriptor Table).

It contains a 16-bit value that selects an entry in the GDT containing a TSS descriptor.

The DPL field of a task gate controls the right to use the descriptor to cause a task switch.

A procedure may not select a task gate descriptor unless the maximum of the selector's RPL and the CPL of the procedure is numerically less than or equal to the DPL of the descriptor.

MAX (CPL,RPL) ≤ task gate DPL

This constraint prevents untrusted procedures from causing a task switch. (Note that when a task gate is used, the DPL of the target TSS descriptor is not used for privilege checking.)

A procedure that has access to a task gate has the power to cause a task switch, just as a procedure that has access to a TSS descriptor.

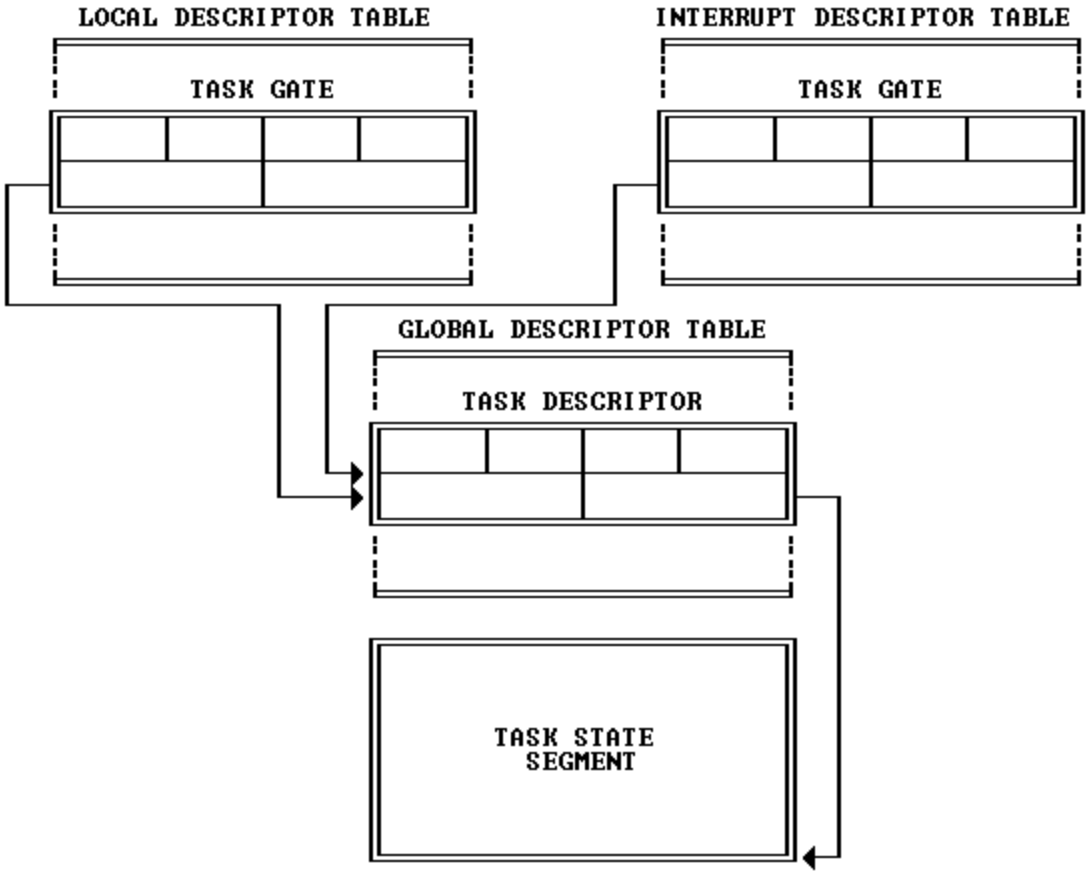
The 80386 has task gates in addition to TSS descriptors to satisfy three needs:

1. The need for a task to have a single busy bit. Because the busy-bit is stored in the TSS descriptor, each task should have only one such descriptor. There may, however, be several task gates that select the single TSS descriptor.

2. The need to provide selective access to tasks. Task gates fulfill this need, because they can reside in LDTs and can have a DPL that is different from the TSS descriptor's DPL. A procedure that does not have sufficient privilege to use the TSS descriptor in the GDT (which usually has a DPL of 0) can still switch to another task if it has access to a task gate for that task in its LDT. With task gates, systems software can limit the right to cause task switches to specific tasks.
3. The need for an interrupt or exception to cause a task switch. Task gates may also reside in the IDT, making it possible for interrupts and exceptions to cause task switching. When interrupt or exception vectors to an IDT entry that contains a task gate, the 80386 switches to the indicated task. Thus, all tasks in the system can benefit from the protection afforded by isolation from interrupt tasks.

Figure 7-5 illustrates how both a task gate in an LDT and a task gate in the IDT can identify the same task.

Figure 7-5. Task Gate Indirectly Identifies Task



Task Switching :

Q. Define Task Switching and explain the steps involved in task switching operation?

The 80386 switches execution to another task in any of four cases:
The task Switching

1. The current task executes a JMP or CALL that refers to a TSS descriptor.
2. The current task executes a JMP or CALL that refers to a task gate.
3. An interrupt or exception vectors to a task gate in the IDT.
4. The current task executes an IRET when the NT flag is set.

JMP, CALL, IRET, interrupts, and exceptions are all ordinary mechanisms of the 80386 that can be used in circumstances that do not require a task switch.

Either the type of descriptor referenced or the NT (nested task) bit in the flag word distinguishes between the standard mechanism and the variant that causes a task switch.

To cause a task switch, a JMP or CALL instruction can refer either to a TSS descriptor or to a task gate. The effect is the same in either case: the 80386 switches to the indicated task.

An exception or interrupt causes a task switch when it vectors to a task gate in the IDT. If it vectors to an interrupt or trap gate in the IDT, a task switch does not occur.

A task switching operation involves these steps:

1. **Privilege Check** :Checking that the current task is allowed to switch to the designated task. Data-access privilege rules apply in the case of JMP or CALL instructions. **The DPL of the TSS descriptor or task gate must be numerically greater (e.g., lower privilege level) than or equal to the maximum of CPL and the RPL of the gate selector.** Exceptions,

interrupts, and IRET are permitted to switch tasks regardless of the DPL of the target task gate or TSS descriptor.

2. **Limit and Present bit Checking** : Checking that the TSS descriptor of the new task is marked present and has a valid limit. Any errors up to this point occur in the context of the outgoing task. Errors are restartable and can be handled in a way that is transparent to applications procedures.

3. **Saving the state of the current task**: The processor finds the base address of the current TSS cached in the task register. It copies the registers into the current TSS (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, and the flag register). The EIP field of the TSS points to the instruction after the one that caused the task switch.

4. **Loading the task register**: with the selector of the incoming task's TSS descriptor, marking the incoming task's TSS descriptor as busy, and setting the TS (task switched) bit of the MSW. The selector is either the operand of a control transfer instruction or is taken from a task gate.
5. **Loading the incoming task's state from its TSS and resuming execution**: The registers loaded are the LDT register; the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; the segment registers ES, CS, SS, DS, FS, and GS; and PDBR. Any errors detected in this step occur in the context of the incoming task. To an exception handler, it appears that the first instruction of the new task has not yet executed.

Task Address Space

The LDT selector and PDBR fields of the TSS give software systems designers flexibility in utilization of segment and page mapping features of the 80386.

By appropriate choice of the segment and page mappings for each task, tasks may share address spaces, may have address spaces that are largely distinct from one another, or may have any degree of sharing between these two extremes.

The ability for tasks to have distinct address spaces is an important aspect of 80386 protection.

A module in one task cannot interfere with a module in another task if the modules do not have access to the same address spaces.

The flexible memory management features of the 80386 allow systems designers to assign areas of shared address space to those modules of different tasks that are designed to cooperate with each other.

1 Task Linear-to-Physical Space Mapping :

The choices for arranging the linear-to-physical mappings of tasks fall into two general classes:

1. **One linear-to-physical mapping shared among all tasks.**

   When paging is not enabled, this is the only possibility. Without page tables, all linear addresses map to the same physical addresses.

   When paging is enabled, this style of linear-to-physical mapping results from using one page directory for all tasks.

   The linear space utilized may exceed the physical space available if the operating system also implements page-level virtual memory.

2. **Several partially overlapping linear-to-physical mappings.**

This style is implemented by using a different page directory for each task. Because the PDBR (page directory base register) is loaded from the TSS with each task switch, each task may have a different page directory.

2 Task Logical Address Space

By itself, a common linear-to-physical space mapping does not enable sharing of data among tasks.

To share data, tasks must also have a common logical-to-linear space mapping; i.e., they must also have access to descriptors that point into a shared linear address space.

There are three ways to create common logical-to-physical address-space mappings:

1. **Via the GDT** :

   All tasks have access to the descriptors in the GDT.

   If those descriptors point into a linear-address space that is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.

2. **By sharing LDTs** :

   Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT segment.

   Those LDT-resident descriptors that point into a linear space that is mapped to a common physical space permit the tasks to share physical memory.

   This method of sharing is more selective than sharing by the GDT; the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared areas.

3. **By descriptor aliases in LDTs** :

It is possible for certain descriptors of different LDTs to point to the same linear address space.

If that linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share the common space. Such descriptors are commonly called "aliases".

This method of sharing is even more selective than the prior two; other descriptors in the LDTs may point to distinct linear addresses or to linear addresses that are not shared.

Virtual 8086 Mode:
Features of Virtual 8086 Mode:

Executing 8086 Code

The processor executes in V86 mode when the VM (virtual machine) bit in the EFLAGS register is set. The processor tests this flag under two general conditions:

1. When loading segment registers to know whether to use 8086-style address formation.
2. When decoding instructions to determine which instructions are sensitive to IOPL.

Except for these two modifications to its normal operations, the 80386 in V86 mode operated much as in protected mode.

Registers :

- The register set available in V86 mode includes all the registers defined for the 8086 plus
- The new registers introduced by the 80386: FS, GS, debug registers, control registers, and test registers.

Instructions :

- New instructions that explicitly operate on the segment registers FS and GS are available, and the new segment-override prefixes can be used to cause instructions to utilize FS and GS for address calculations.
- Instructions can utilize 32-bit operands through the use of the operand size prefix.

8086 programs running as V86 tasks are able to take advantage of the new applications-oriented instructions added to the architecture by the introduction of the 80186/80188, 80286 and 80386:

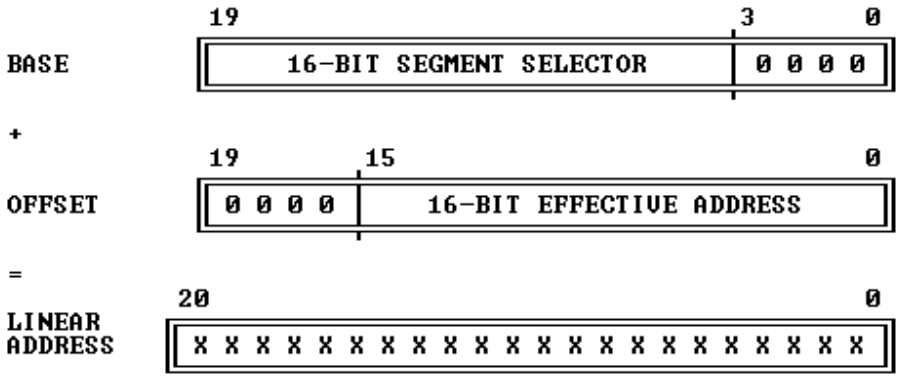
- New instructions introduced by 80186/80188 and 80286.
  - PUSH immediate data
  - Push all and pop all (PUSHA and POPA)
  - Multiply immediate data
  - Shift and rotate by immediate count
  - String I/O
  - ENTER and LEAVE
  - BOUND

- New instructions introduced by 80386.
  - LSS, LFS, LGS instructions
  - Long-displacement conditional jumps
  - Single-bit instructions
  - Bit scan
  - Double-shift instructions
  - Byte set on condition
  - Move with sign/zero extension
  - Generalized multiply

Memory Management in Virtual Mode :

Linear Address Formation :

Figure 15-1. V86 Mode Address Formation



- In V86 mode, the 80386 processor does not interpret 8086 selectors by referring to descriptors; instead, it forms linear addresses as an 8086 would.
- It shifts the selector left by four bits to form a 20-bit base address. The effective address is extended with four high-order zeros and added to the base address to create a linear address.
- Because of the possibility of a carry, the resulting linear address may contain up to 21 significant bits. An 8086 program may generate linear addresses anywhere in the range 0 to 10FFEFH (one megabyte plus approximately 64 Kbytes) of the task's linear address space.
- V86 tasks generate 32-bit linear addresses.
- While an 8086 program can only utilize the low-order 21 bits of a linear address, the linear address can be mapped via page tables to any 32-bit physical address.

- Unlike the 8086 and 80286, 32-bit effective addresses can be generated (via the address-size prefix); however, the value of a 32-bit address may not exceed 65,535 without causing an exception.

- For full compatibility with 80286 real-address mode, pseudo-protection faults (interrupt 12 or 13 with no error code) occur if an address is generated outside the range 0 through 65,535.

Structure of a V86 Task

- A V86 task consists partly of the 8086 program to be executed and partly of 80386 "native mode" code that serves as the virtual-machine monitor.
- The task must be represented by an 80386 TSS (not an 80286 TSS). The processor enters V86 mode to execute the 8086 program and returns to protected mode to execute the monitor or other 80386 tasks.

To run successfully in V86 mode, an existing 8086 program needs the following:

- A V86 monitor.
- Operating-system services.
- The V86 monitor is 80386 protected-mode code that executes at privilege-level zero.
- The monitor consists primarily of initialization and exception-handling procedures. As for any other 80386 program, executable-segment descriptors for the monitor must exist in the GDT or in the task's LDT.
- The linear addresses above 10FFEFH are available for the V86 monitor, the operating system, and other systems software.
- The monitor may also need data-segment descriptors so that it can examine the interrupt vector table or other parts of the 8086 program in the first megabyte of the address space.

In general, there are two options for implementing the 8086 operating system:

1. The 8086 operating system may run as part of the 8086 code. This approach is desirable for any of the following reasons:
    - The 8086 applications code modifies the operating system.
    - There is not sufficient development time to reimplement the 8086 operating system as 80386 code.
2. The 8086 operating system may be implemented or emulated in the V86 monitor. This approach is desirable for any of the following reasons:

- Operating system functions can be more easily coordinated among several V86 tasks.
- The functions of the 8086 operating system can be easily emulated by calls to the 80386 operating system.

Using Paging for V86 Tasks

Paging is not necessary for a single V86 task, but paging is useful or necessary for any of the following reasons:

- To create multiple V86 tasks. Each task must map the lower megabyte of linear addresses to different physical locations.

- To emulate the megabyte wrap. On members of the 8086 family, it is possible to specify addresses larger than one megabyte.

- For example, with a selector value of 0FFFFH and an offset of 0FFFFH, the effective address would be 10FFEFH (one megabyte + 65519).
- The 8086, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to 0FFEFH.
- The 80386, however, which can form addresses up to 32 bits long does not truncate such an address.
- If any 8086 programs depend on this addressing anomaly, the same effect can be achieved in a V86 task by mapping linear addresses between 100000H and 110000H and linear addresses between 0 and 10000H to the same physical addresses.
- To create a virtual address space larger than the physical address space.
- To share 8086 OS code or ROM code that is common to several 8086 programs that are executing simultaneously.
- To redirect or trap references to memory-mapped I/O devices.

Protection within a V86 Task

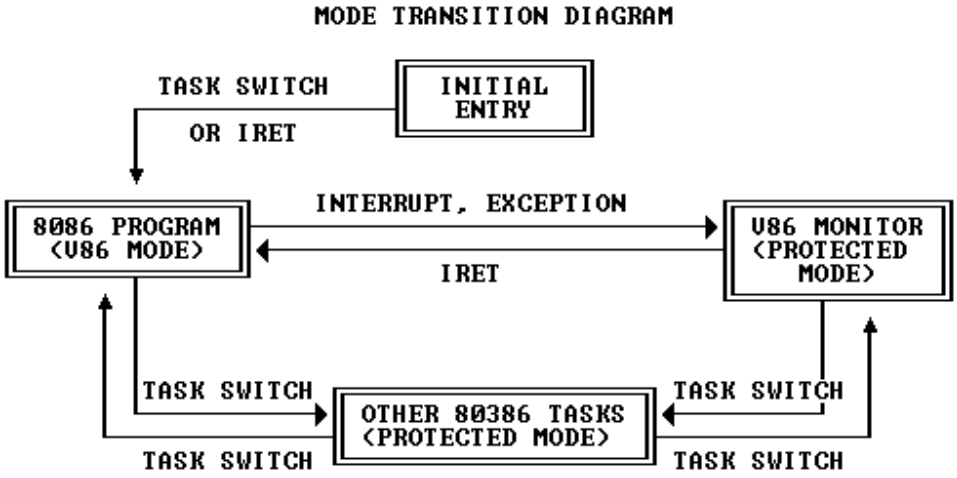
Because it does not refer to descriptors while executing 8086 programs, the processor also does not utilize the protection mechanisms offered by descriptors.

To protect the systems software that runs in a V86 task from the 8086 program, software designers may follow either of these approaches:

- Reserve the first megabyte (plus 64 kilobytes) of each task's linear address space for the 8086 program. An 8086 task cannot generate addresses outside this range.
- Use the U/S bit of page-table entries to protect the virtual-machine monitor and other systems software in each virtual 8086 task's space. When the processor is in V86 mode, CPL is 3.
- Therefore, an 8086 program has only user privileges. If the pages of the virtual-machine monitor have supervisor privilege, they cannot be accessed by the 8086 program.

Entering and Leaving V86 Mode

Figure 15-2. Entering and Leaving the 8086 Program



The 80386 enters or leaves 8086 virtual mode due to any of the three reasons as :

1) An interrupt that vectors to a task gate.
2) An action of the schedule of the 80386 operating system.
3) An IRET when the NT(Nested Task) flag is set.

Entering 8086 Virtual Mode :

The 8086 can enter 8086 virtual mode by either of two means:

1) A task switch to an 80386 task loads the image of EFLAGS from the new TSS. The TSS of the new task must be an 80386 TSS, not an 80286 TSS, because the 80286 TSS does not store the high-order word of EFLAGS, which contains the VM flag.

   A value of one in the VM bit of the new EFLAGS indicates that the new task is executing 8086 instructions

2) An IRET from a procedure of an 80386 task loads the image of EFLAGS from the stack.

   A value of one in VM in this case indicates that the procedure to which control is being returned is an 8086 procedure.

Leaving 8086 Virtual Mode :

The processor leaves V86 mode when an interrupt or exception occurs.

1) The interrupt or exception causes a task switch. A task switch from a V86 task to any other task loads EFLAGS from the TSS of the new task.

   If the new TSS is an 80386 TSS and the VM bit in the EFLAGS image is zero or if the new TSS is an 80286 TSS,

   then the processor clears the VM bit of EFLAGS, loads the segment registers from the new TSS using 80386-style address formation,

   and begins executing the instructions of the new task according to 80386 protected-mode semantics.

2) The interrupt or exception vectors to a privilege-level zero procedure. The processor stores the current setting of EFLAGS on the stack, then clears the VM bit.

   The interrupt or exception handler, therefore, executes as "native" 80386 protected-mode code.

   If an interrupt or exception vectors to a conforming segment or to a privilege level other than three, the processor causes a general-protection exception; the error code is the selector of the executable segment to which transfer was attempted.

Q.Difference between Real Mode , Protected Mode and Virtual 8086 Mode.(End Sem)

Difference Between Real, Protected and Virtual modes

Sr. No	Real Mode	Protected Mode	Virtual Mode
1	In this mode 80386 maintains the compatibility of the object code with 8086	This mode provides a sophisticated memory management and the hardware assisted protection mechanism.	This mode allows the execution of real mode programs that are incapable of running directly in protected mode
2	Memory size is limited to 1 Mbyte.	It increases the linear address space to 4 GB and allows the running of virtual memory programs of 64 TB.	Memory address range of virtual 8086 mode task is limited by 1 Mbyte.
3	Paging mechanism is in-active	Paging mechanism is active	Paging mechanism is active
4	Only A2-A19 address lines are active and A20-A31 address lines are high.	All address lines are active	All address lines are active
5	Protection Mechanism is not available	Protection Mechanism is available	It provides mechanism to selectively trap and manage I/O and interrupt activity.
6	Multitasking is not supported	Multitasking is supported	Multitasking is supported
7	Linear address is same as physical address	Physical address is generated from linear address with the help of page translation.	Physical address is generated from linear address with the help of page translation.