The protected mode

M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica

Introduction

Starting from 80286, Intel processors supported two modes

- **Real mode**
- **Protected mode.**

Virtual mode was introduced starting from the 80386.

Real mode

Allows accessing the memory like in the 8086.

Effective addresses are computed using the formula

$$offset + segment\_register \times 16$$

Protected mode

Was introduced for two purposes:

- To prevent the different tasks in a multitasking operating system from performing invalid or incorrect accesses
- Effective addresses are computed in a different way.

Task

A *task* is the combination of

- **A program**
- **Its data**
- **The necessary system functions.**

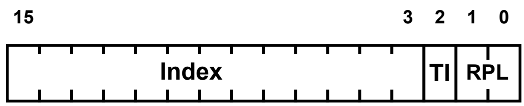
Selectors

In protected mode, the values in the segment registers represent *selectors* (and not base addresses).

Information provided by selectors on the segment allow to

- **Access to a global or local descriptor table**
- **Identify the minimum privilege level required to access the segment (*Requested Privilege Level*, or *RPL*).**

Segment selector



Effective Privilege Level

The value of the RPL field of the segment selector of the code segment at a given time is the *Current Privilege Level* (CPL), i.e., the privilege level of the currently active program.

The active program can only access data segments that have a privilege level the same as or larger than the CPL.

The larger between CPL and RPL is the the *Effective Privilege Level* (EPL), i.e., the privilege of the current task.

Example

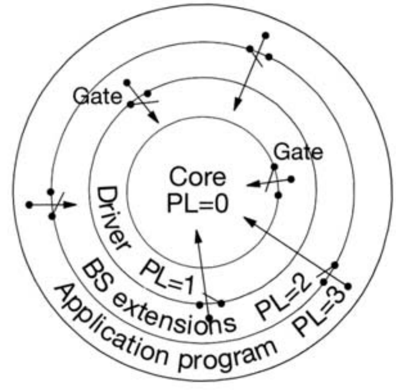
If the CPL is 2 the task can

- Make access to data contained in segments whose RPL is 2 or 3
- Make a branch to instructions belonging to code segments whose RPL is 2 or 3.

Privilege levels

There are 4 privilege levels:

- Privilege level 0 (the max) is that of the OS kernel.
- Privilege level 1 is that of the OS functions.
- Privilege level 2 is that of the least critical OS functions.
- Privilege level 3 is that of the application programs.



Stack

Starting from 80386, a separate stack is provided for every privilege level of a task.

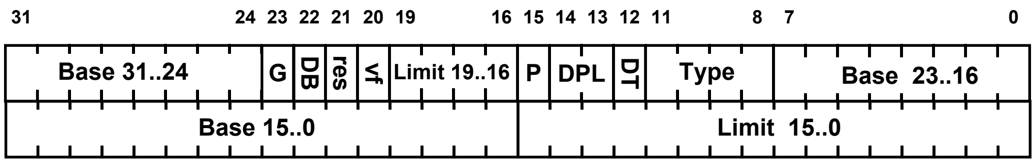
Descriptor tables

The memory stores

- **A *Global Descriptor Table* (GDT), containing the descriptors of segments available for all tasks**
- **A *Local Descriptor Table* (LDT), containing the descriptors of segments available to the currently active task, only.**

Each segment selector refers either to the GDT, or to the LDT, depending on the value of the TI bit (0→GDT, 1→LDT).

Segment descriptor



Base and limit

Each descriptor provides:

- The *base address* for the segment (32 bits)
- The *limit* (20 bits), whose meaning depends on the value of the G bit:
  - If G=0 (byte granularity), the limit is the segment size in bytes; the maximum segment size is 2^{20} bytes
  - If G=1 (page granularity), the limit is the segment size in pages; the maximum segment size is 2^{20}*4 Kb = 4Gb.

Other fields

- *DT* defines the type of the segment:
  - If DT=0, it is a system segment
  - If DT=1, it is an application segment
- *DPL* (Descriptor Privilege Level) gives the segment privilege level
- *P* says whether the segment is in memory or not
- *vf* is an available field.

Other fields

- *DT* defines the type of the segment:
  - If DT=0, it is a system segment
  - If DT=1, it is an application segment
- *DPL* (Descriptor Privilege Level) gives the segment privilege level
- *P* says whether the segment is in memory or not
- *vf* is an available field.

Gate, interrupt and trap segments

Other fi

If a segment with P=0 is accessed, the processor triggers a "segment not available" exception.

- *DT* defines the type of the
  - If DT=0, it is a system segm
  - If DT=1, it is an application segment
- *DPL* (Descriptor Privilege Level) gives the segment privilege level
- *P* says whether the segment is in memory or not
- *vf* is an available field.

Type field for system segments

1-7	for 286	system segment or gate
8	reserved	
9	available TSS	system segment
10	reserved	
11	active TSS	system segment
12	call gate	gate
13	reserved	
14	interrupt gate	gate
15	trap gate	gate

DB field

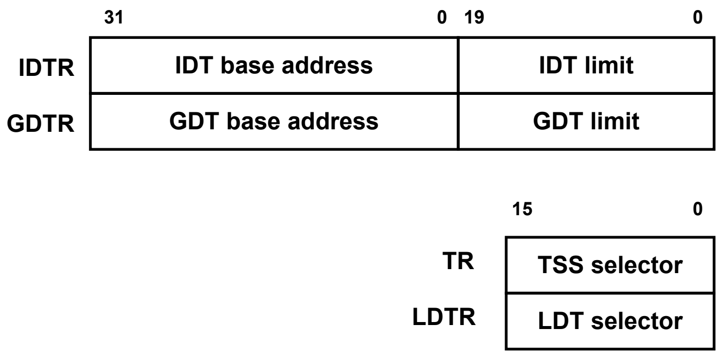
The DB field forces the processor to work with 16- or 32-bit operands when executing the code contained in the segment.

The 80386 can also switch from 32- to 16-bit operands when the operand or address size prefix is used.

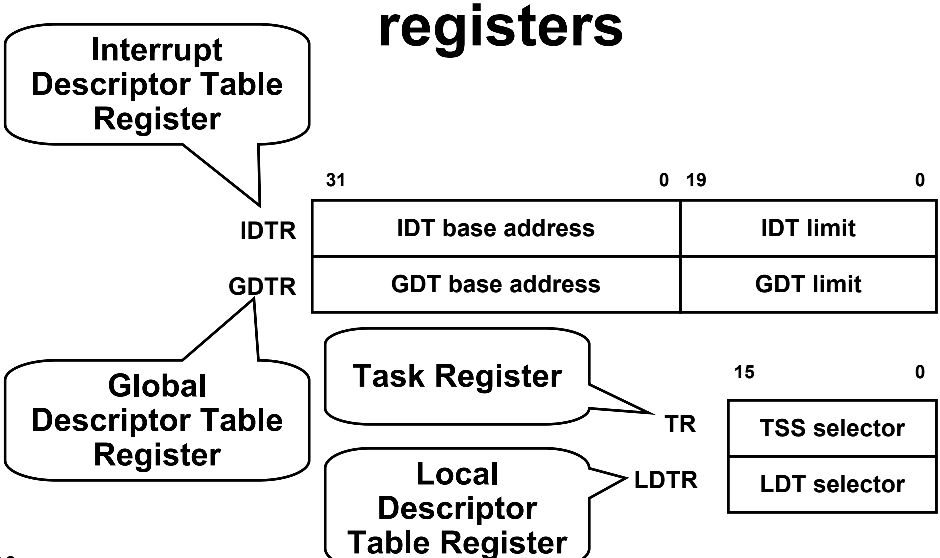
Privilege level

When the loader loads to memory a task, normally assigns a given value to the DPL of its segments, and to the RPL in the selectors used by the task.

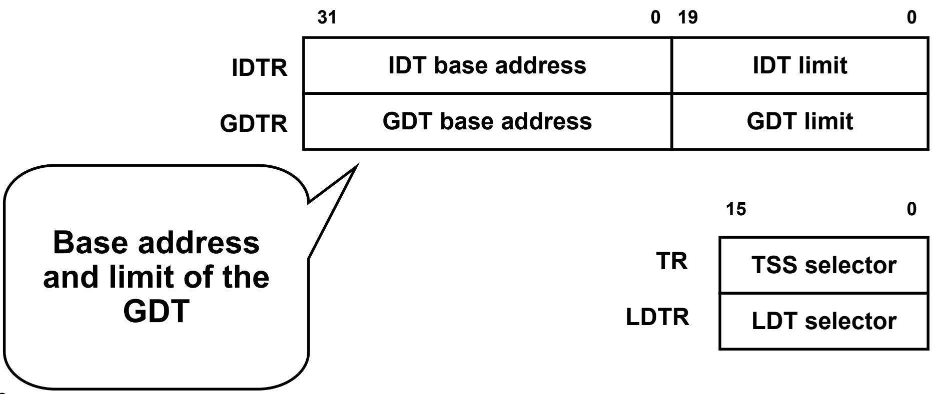
Memory management registers



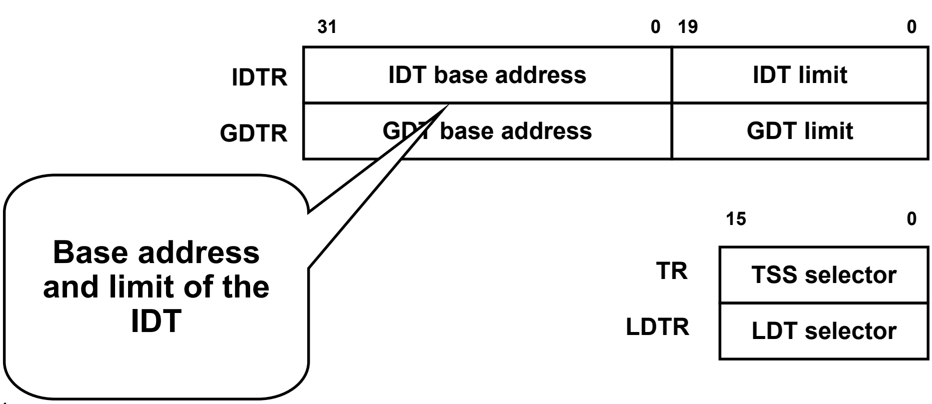
Memory management registers



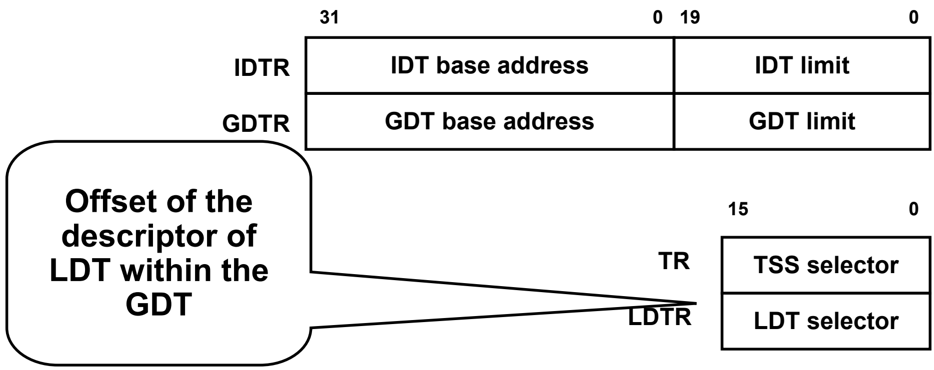
Memory management registers



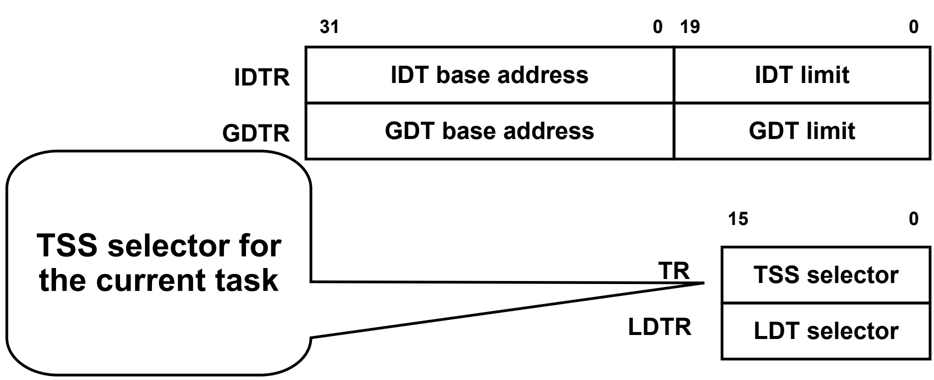
Memory management registers



Memory management registers



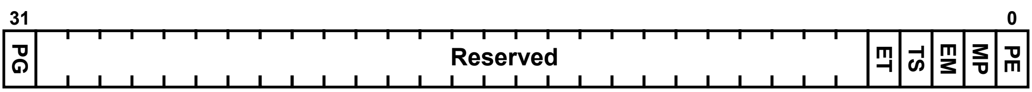
Memory management registers



Control Register CR0

CR0 stores the basic information about the processor behavior.

By changing the content of CR0, one can change the processor behavior.

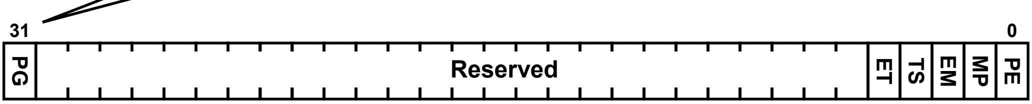


Control Register CR0

CR0 stores ... the processor behavior.

By chang... n change the processor...

Paging:
- PG=0→paging unit deactivated
- PG=1→paging active

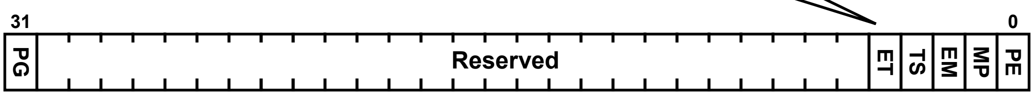


Control Register CR0

CR0 stores the basic information about the processor behavior.

By chang ... n change the processo

Extension Type:
- ET=0→80287
- ET=1→80387

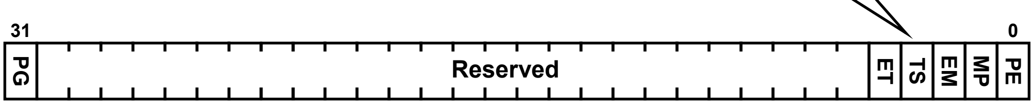


Control Register CR0

CR0 stores the ... ocessor behavior.

By changing th ... nge the processor behavi

Task switched:
- TS=0→no task switch
- TS=1→task switch carried out

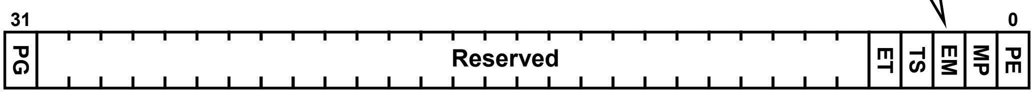


Control R

Emulate coprocessor
- EM=0→no emulation
- EM=1→emulate i387

CR0 stores the basic info
behavior.

By changing the content of CR0, one can ch the
processor behavior.

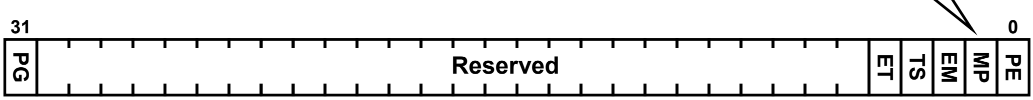


Control Register CR0

Coprocessor
- MP=0→no coprocessor
- MP=1→i387 installed

CR0 stores the basic or
behavior.

By changing the cont the
processor behavior.



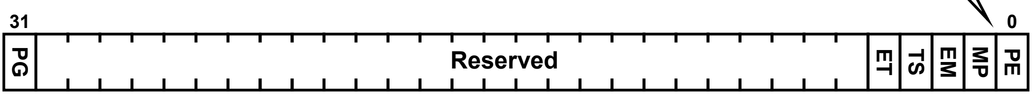
Contro

Protected mode

- **PE=0→i386 runs in real more**
- **PE=1→i386 runs in protected mode**

CR0 stores the basic ssor behavior.

By changing the content of CR0, one can e the processor behavior.

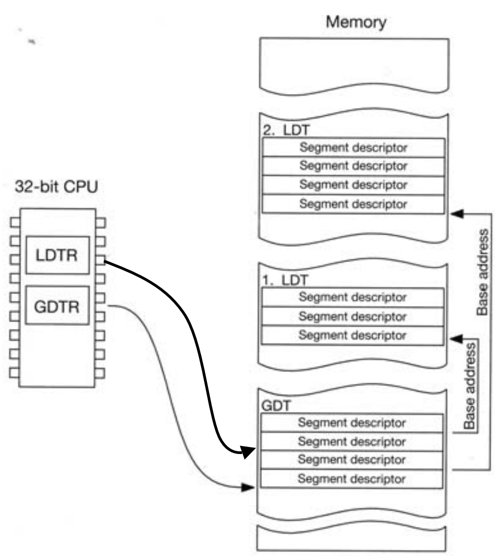


LDT

Since there are several LDTs, their segment descriptors are stored in the GDT.

GDTR stores the base address of the GDT.

LDTR stores the selector of a LDT descriptor within the GDT.



GDTR and LDTR updates

For each task, the OS delivers the GDT and a LDT.

GDTR and LDTR updates are performed by the OS kernel (working with PL=0) using the `LGDT` and `LLDT` instructions.

Switching into protected mode

The least significant bit of CR0 (PE) determines whether the processor works in real (PE=0) or protected (PE=1) mode.

The switch to protected mode can be performed in several ways:

- Through the `INT 15h` 89h function
- Using the `LMSW` (Load Machine Status Word) instruction
- Using the `MOV CR0, imm` instruction.

Memory addressing in protected mode

The processor

- Accesses the segment descriptor and checks whether the GDT or a LDT must be used
- Retrieves from GDTR or LDTR the base address of the appropriate descriptor table
- Multiplies the selector index field by 8, and sums this value to the base address
  - If the result is greater than the limit, a *general protection fault* exception is triggered.
  - If not, the segment descriptor gives the base and limit of the segment
- Sums the base to the offset (checking whether it is lower than the limit)
- Accesses to memory.

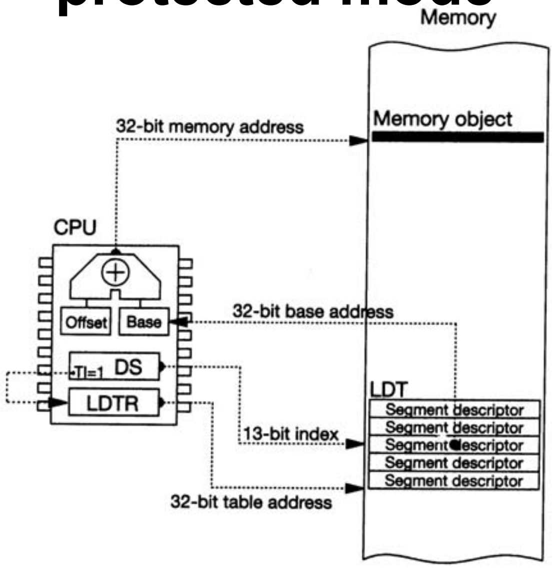
Memory addressing in protected mode

The processor

- accesses the segment descriptor and checks whether the GDT or a LDT must be used
- Retrieves from GDTR or LDTR the base address of the appropriate descriptor tabl
- Multiplies the selector i the base address
  - If the result is great *fault* exception is tri
  - If not, the segment the segment
- Sums the base to the of the limit)
- Accesses to memory.

Thanks to segment descriptor cache register, this procedure is followed only for the instructions accessing to a new segment; for most of the memory accesses the computation of the memory address is as fast as for real mode.

Memory addressing in protected mode



Intersegment control transfer

When a far call, a far jump, or an interrupt occur, not only the EIP value, but also the code segment changes.

In real mode, this means changing the value of CS.

In protected mode, this involves executing the following:

- **If the target segment has the same (or higher) PL of the current one, the CPU loads the target segment selector in the CS register**
- **If the target segment has a lower PL than the current one, the far call can only be accomplished through the use of a *call gate*.**

Gates

Gates permit jumping to a routine in another segment with a different privilege level.

In this case, the segment selector of the target segment doesn't point first to the target segment itself but to a *call gate*.

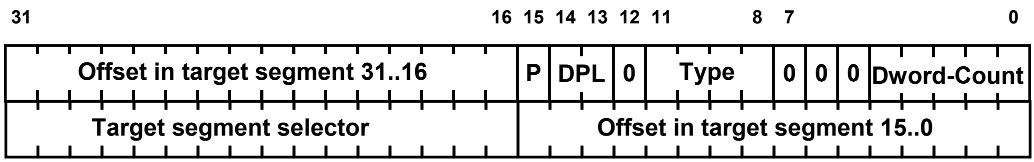
The type and DT fields in the target segment descriptor let the processor know whether it is a gate descriptor or a code segment.

Gate descriptor identification

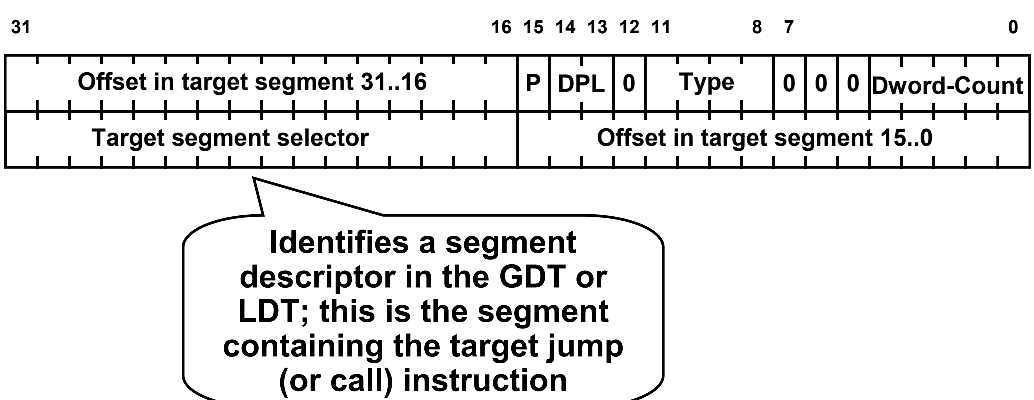
Gates are defined by

- the DT=0 bit in the segment descriptor
- values from 4 to 7 and from 12 to 15 in the type field.

Call gate descriptor



Call gate descriptor



Call g

Corresponds to an offset in the target segment

31 16	15	14 13	12	11 8	7			0
Offset in target segment 31..16	P	DPL	0	Type	0	0	0	Dword-Count
Target segment selector	Offset in target segment 15..0							

Call g

Forces the processor to transfer Dword-count double words from the stack of the calling procedure to that of the called one

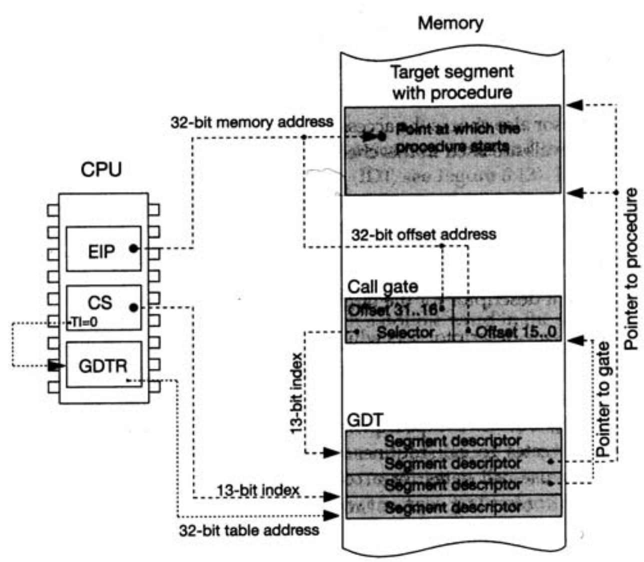
31 16	15	14 13	12	11 8	7			0
Offset in target segment 31..16	P	DPL	0	Type	0	0	0	Dword-Count
Target segment selector	Offset in target segment 15..0							

Call gate descriptor

It must be 12 (*call gate*)

31 16	15	14 13	12	11 8	7			0
Offset in target segment 31..16	P	DPL	0	Type	0	0	0	Dword-Count
Target segment selector				Offset in target segment 15..0				

Far call via a gate descriptor



Rationale for gates

Gates are important to limit the effects of incorrect jumps:

- If a jump to an incorrect point is performed, the system experiences a system crash
- If a jump to an incorrect gate is performed, only the task is aborted.

Call and jump

Both call and far jump instructions can resort to call gates. However

- Call instructions may use call gates to access procedures belonging to code segments owning a higher PL, also
- Jump instructions may use call gates to access instruction belonging to code segments owning the same or lower PL, only.

Stack management

Every privilege level has its own stack.

When a call through a gate is executed, the processor automatically transfers from the stack of the calling to that of the called procedure as many double words as indicated by the DWord-count field.

Interrupt descriptor table

In real mode, calls to interrupt service procedures are managed through the *Interrupt Vector Table* (IVR).

For each interrupt type, the IVR contains 4 bytes (2 for EIP and 2 for CS).

In protected mode, a jump to an interrupt service routine happens through the *Interrupt Description Table* (IDT), which can be accessed through the IDTR.

The IDT only contains descriptors of interrupt gates. Each descriptor is 8-byte long.

The length of IDT can be modified by changing the value of the IDT limit field. Its maximum size is 256 × 8 bytes.

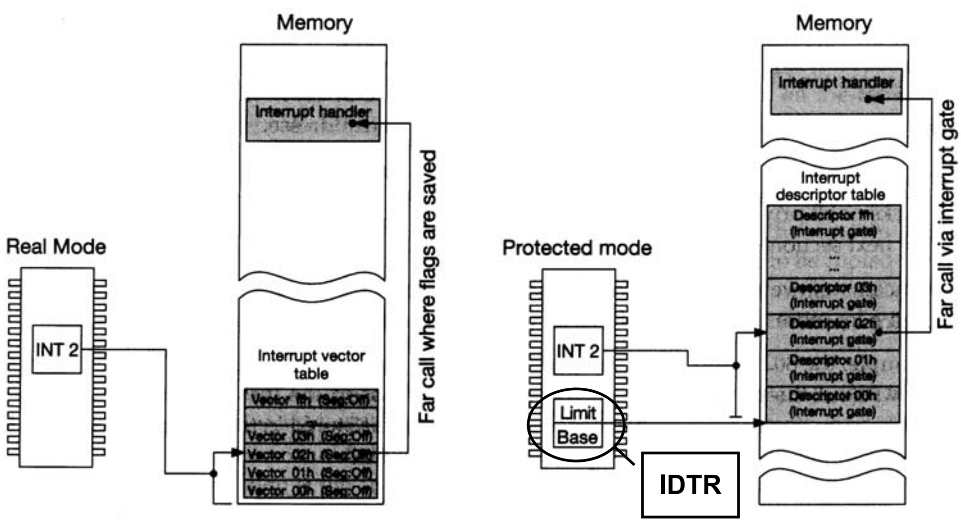
The IDT can be located anywhere in memory.

Interrupt gates

They are similar to call gates, but

- The type field contains the value 14
- The dword-count field is meaningless.

Interrupt tables



Multitasking

Multitasking is based on having several tasks running at the same time.

Periodically, each active task is activated and run for a short time; then, it is interrupted. After another short period it is restarted again from the same point where it was interrupted.

A mechanism is required, to efficiently and safely save/restore the status of a task.

All the information about a task are stored in the *Task State Segment* (TSS), which is 104 byte wide.

Task switch

When a task switch occurs, the processor automatically

- saves all the due information about the current task in the TSS identified by the *Task Register* (TR)
- loads the values of the task to be started into the segment, offset, and control registers.

Task State Segment (I)

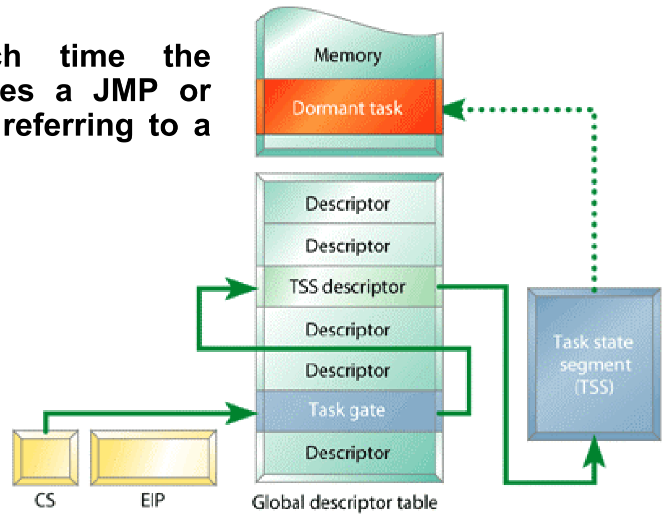
31 … 16	15 … 0	Offset
I/O map base	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 T	64h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Task LDT selector	60h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS selector	5ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS selector	58h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS selector	54h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS selector	50h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS selector	4ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES selector	48h
EDI		44h
ESI		40h
EBP		3ch
ESP		38h
EBX		34h
EDX		30h

Task State Segment (II)

31 … 16	15 … 0	Offset
ECX		2ch
EAX		28h
EFLAG		24h
EIP		20h
CR3 (PDBR)		1ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL2	18h
ESP for CPL2		14h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL1	10h
ESP for CPL1		0ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL0	08h
ESP for CPL0		04h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Back link to previous TSS	00h

Task switch

It happens each time the processor executes a JMP or CALL instruction referring to a *Task Gate*.



Task gate descriptor

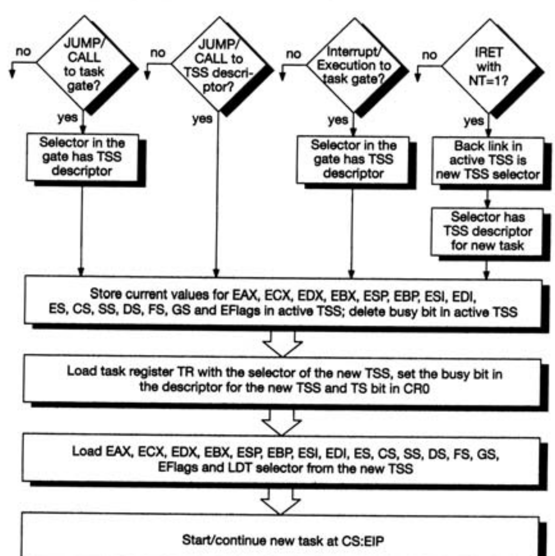
31 16	15	14 13	12	11			8	7 0
Reserved	P	DPL	0	0	1	0	1	Reserved
TSS segment selector	Reserved							

Task switch triggering

When the processor encounters a task gate during the execution of a call instruction, jump instruction, or interrupt, it

- Stores the current condition of the active task in the TSS identified by the TR
- Writes the value 1 (80286) or 9 (80386) in the type field of the old TSS descriptor. In this way, the TSS is identified as an *available* TSS.
- Loads into the task register the new TSS segment selector from the task gate descriptor
- Reads the base address, limit, and access privileges of the task gate descriptor from the LDT or GDT
- Identifies the new TSS descriptor as busy by writing the value 3 (80286) or 11 (80386) in the type field
- Loads the values for CS and EIP from the new TSS.

Triggering and executing a task switch



New task creation

When the OS creates a new task, it must provide it with

- a task gate
- a TSS descriptor
- a TSS.

Preemptive multitasking

In true multitasking OSs, it is the OS that decides when to execute the task switch: applications can not influence this process (*preemptive multitasking*).

On the contrary, in Windows 3.x it was up to the application to decide when to transfer control to another task (either OS or application). This mechanism is called *non-preemptive multitasking*.

From Windows 95, both forms of multitasking are supported, depending on how the specific application is written.