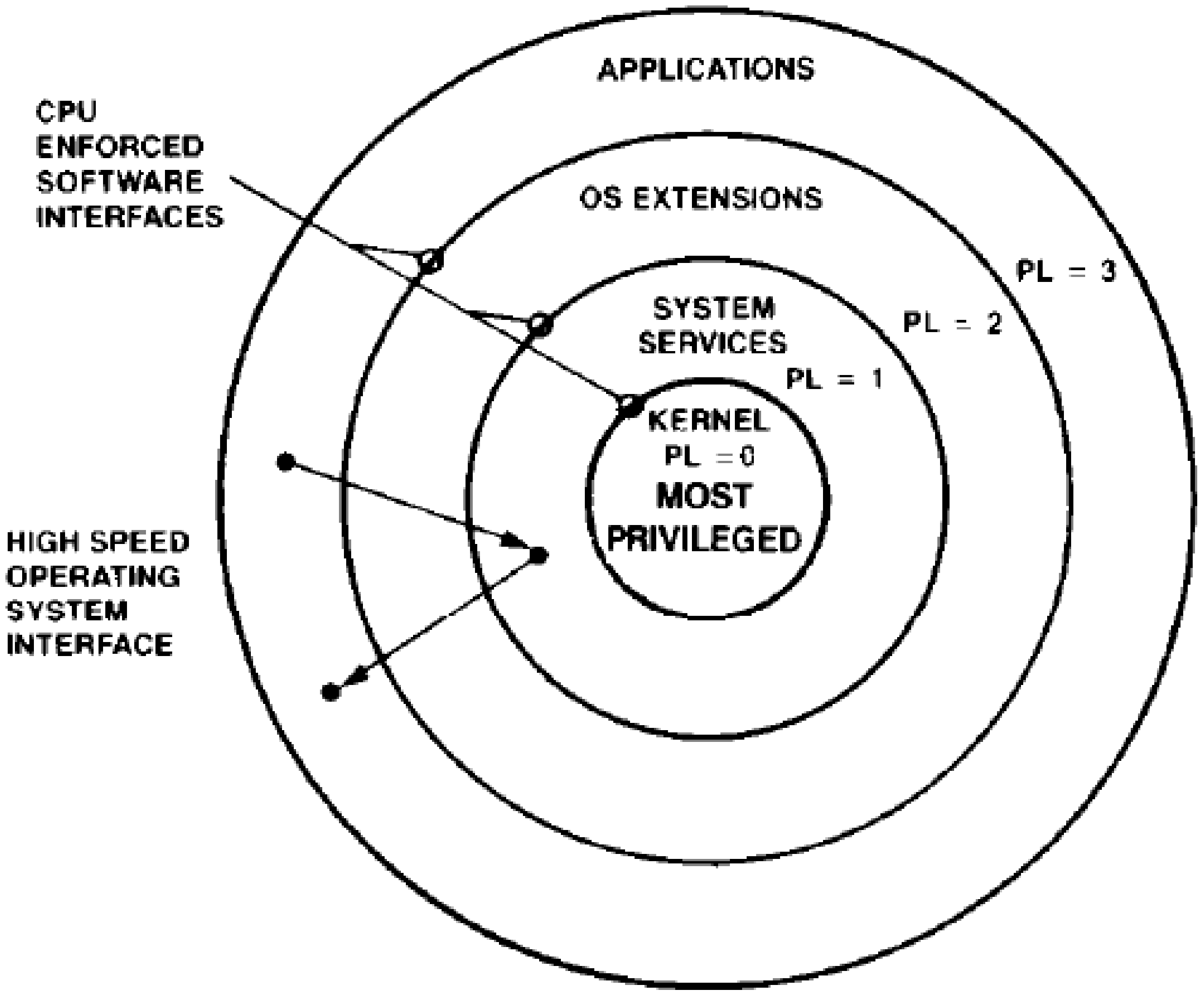
Protection in 80386DX

UQ: Explain the protection mechanism of X86 Intel family microprocessor

Protection

- 80386 DX has four levels of protection which isolate and protect user programs from each other and the operating system.
- It offers an additional type of protection on a page basis, when paging is enabled(using U/S and R/W fields)
- The four-level hierarchical privilege system is illustrated as follows:

Protection


APPLICATIONS
OS EXTENSIONS
SYSTEM SERVICES
KERNEL
PL = 0
MOST PRIVILEGED
PL = 1
PL = 2
PL = 3
CPU ENFORCED SOFTWARE INTERFACES
HIGH SPEED OPERATING SYSTEM INTERFACE

Protection

- The privilege levels (PL) are numbered 0 through 3.
- Level 0 is the most privileged or trusted level.

Rules for Privileges

- Intel 386Dx controls access to both data and procedures according to the following rules:

(1) **Data** segment with privilege **level p** can be accessed only by the **code** executing at a privilege level **atleast as privileged as p**

(E.g. Application programs are prevented from reading or changing OS Tables)

Rules for Privileges

(2) A code segment with a privilege level p can only be called by a task executing at the same or lesser privilege level than p

(E.g. An Application Program may call an OS routine)

Privilege Level

- There are 3 different types of privilege level entering into the privilege level checks:
  - **Current Privilege Level (CPL)**
  - **Descriptor Privilege Level (DPL)**
  - **Requestor Privilege Level (RPL)**

Current Privilege Level (CPL)

- CPL is stored in the selector of currently executing CS register
- It represents the privilege level(PL) of the currently executing task.
- It is also PL in the descriptor of the code segment.
- It is also designated as Task Privilege Level(TPL)

Descriptor Privilege Level (DPL)

- It is the PL of the object which is being attempted to be accessed by the current task
- It is PL of target segment and is contained in the descriptor of the segment

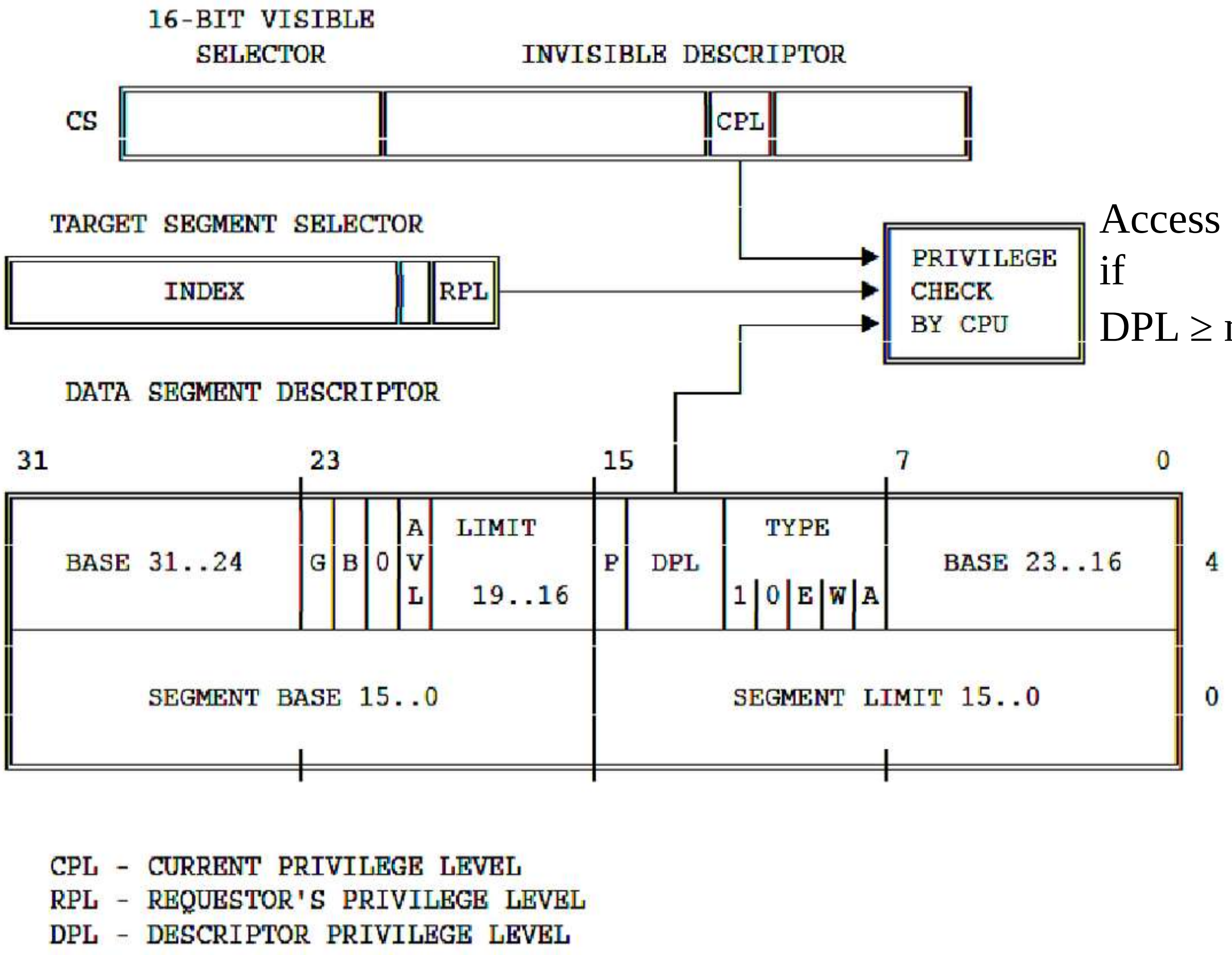
Requestor Privilege Level (RPL)

- It is the lowest two bits of any selector.
- It can be used to weaken the CPL if desired.
- The Effective Privilege Level(EPL) is
  EPL = max (CPL,RPL) (here numbers)
- Thus the task becomes less privileged.

Restricting Access to Data

- Assume that a task needs data from data segment.
- The privilege levels are checked at the time a selector for the target segment is loaded into the data segment register.
- Three privilege levels enter into privilege checking mechanism
  - CPL
  - RPL of the selector of target segment
  - DPL of the descriptor of the target segment

Restricting Access to Data



CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

Access is allowed only if
DPL ≥ max (CPL,RPL)

Restricting Access to Data

- A procedure can only access the data that is at the same or less privilege level (not numerically)

Restricting Control Transfer

- Control transfer (except interrupts) are accomplished by JMP, CALL and RET instructions.
- The near forms of JMP and CALL transfer within current code segment and requires only limit checking
- The far forms of JMP and CALL refer to other segments and require privilege checking.

Restricting Control Transfer

- The far JMP and CALL can be done in 2 ways:
  1. Without Call Gate Descriptor
  2. With Call Gate Descriptor

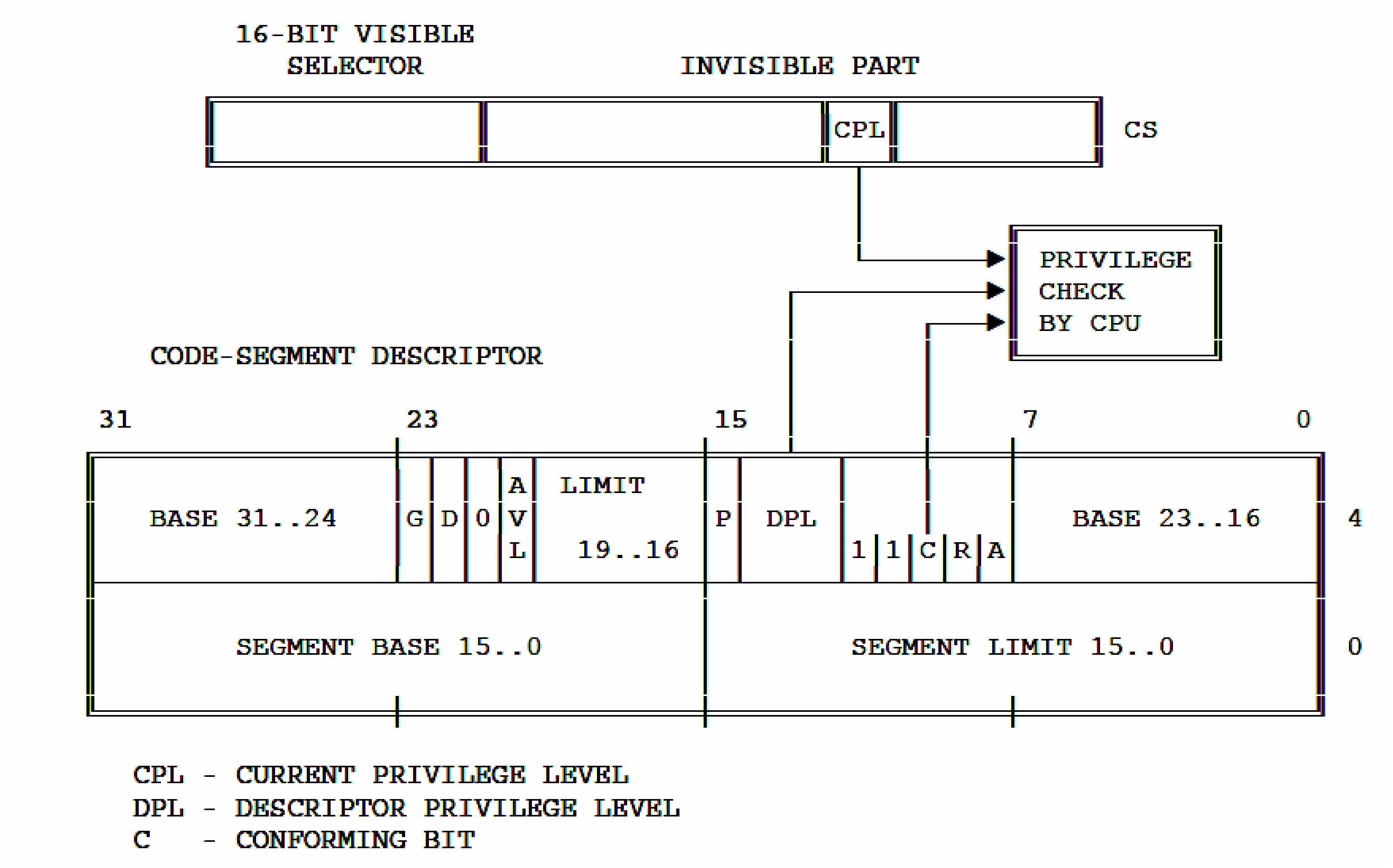
Without Call Gate

- The processor permits a JMP or CALL directly to another segment only if

1. DPL of the target segment = CPL of the calling segment
2. Conforming bit of the target code is set and DPL of the target segment ≤ CPL

- **Confirming Segment:** These segments may be called from various privilege levels but execute at the privilege level of the calling procedure. (e.g. math library)

Privilege Check for Control Transfer without gate

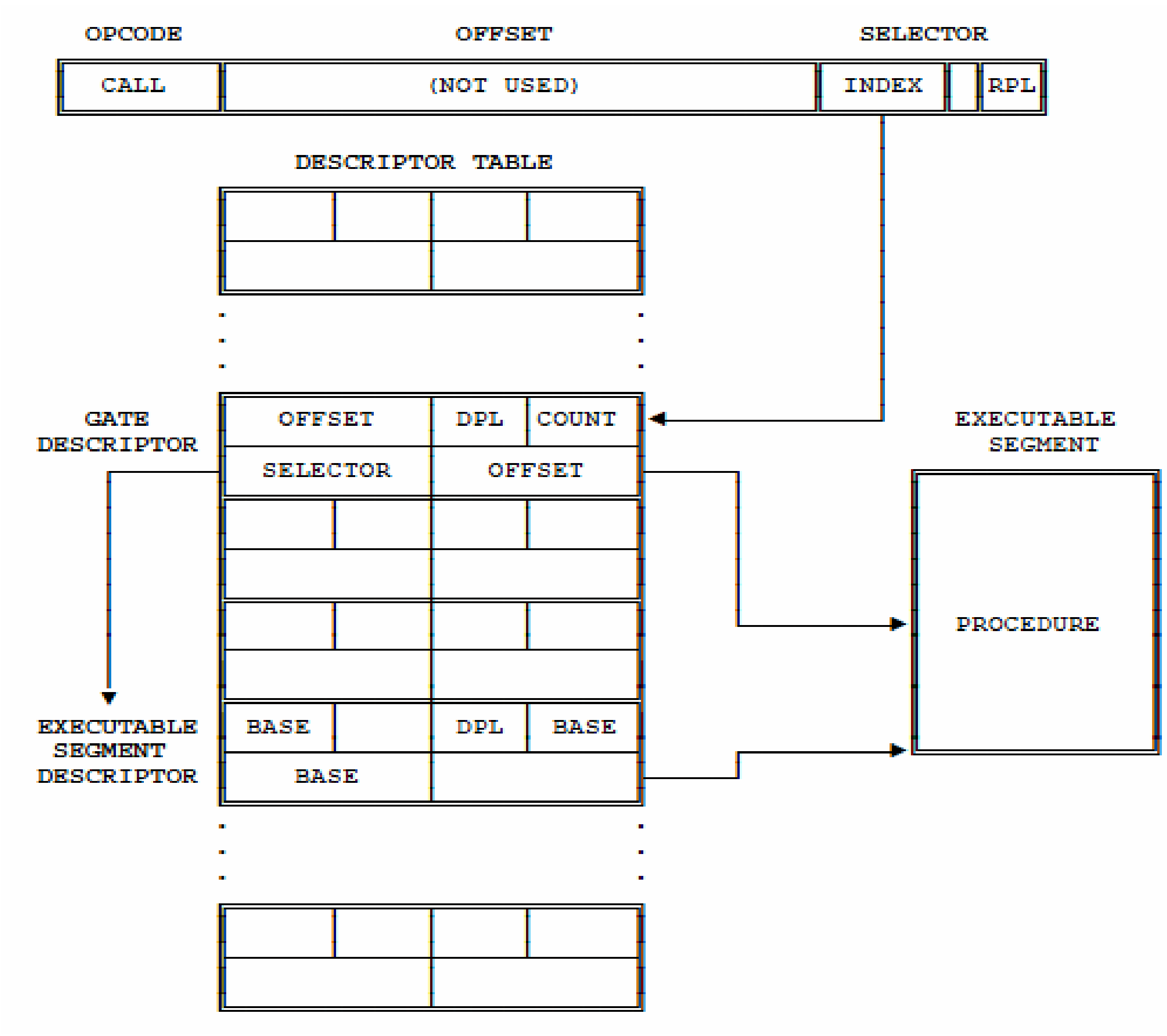


CPL - CURRENT PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL
C - CONFORMING BIT

With Call Gate

- The far pointer of the control transfer instruction uses the selector part of the pointer and selects a gate.
- The selector and offset fields of a gate form a pointer to the entry of a procedure.

With Call Gate



With Call Gate

- Four privilege levels are used to check the validity of the control transfer via a call gate:

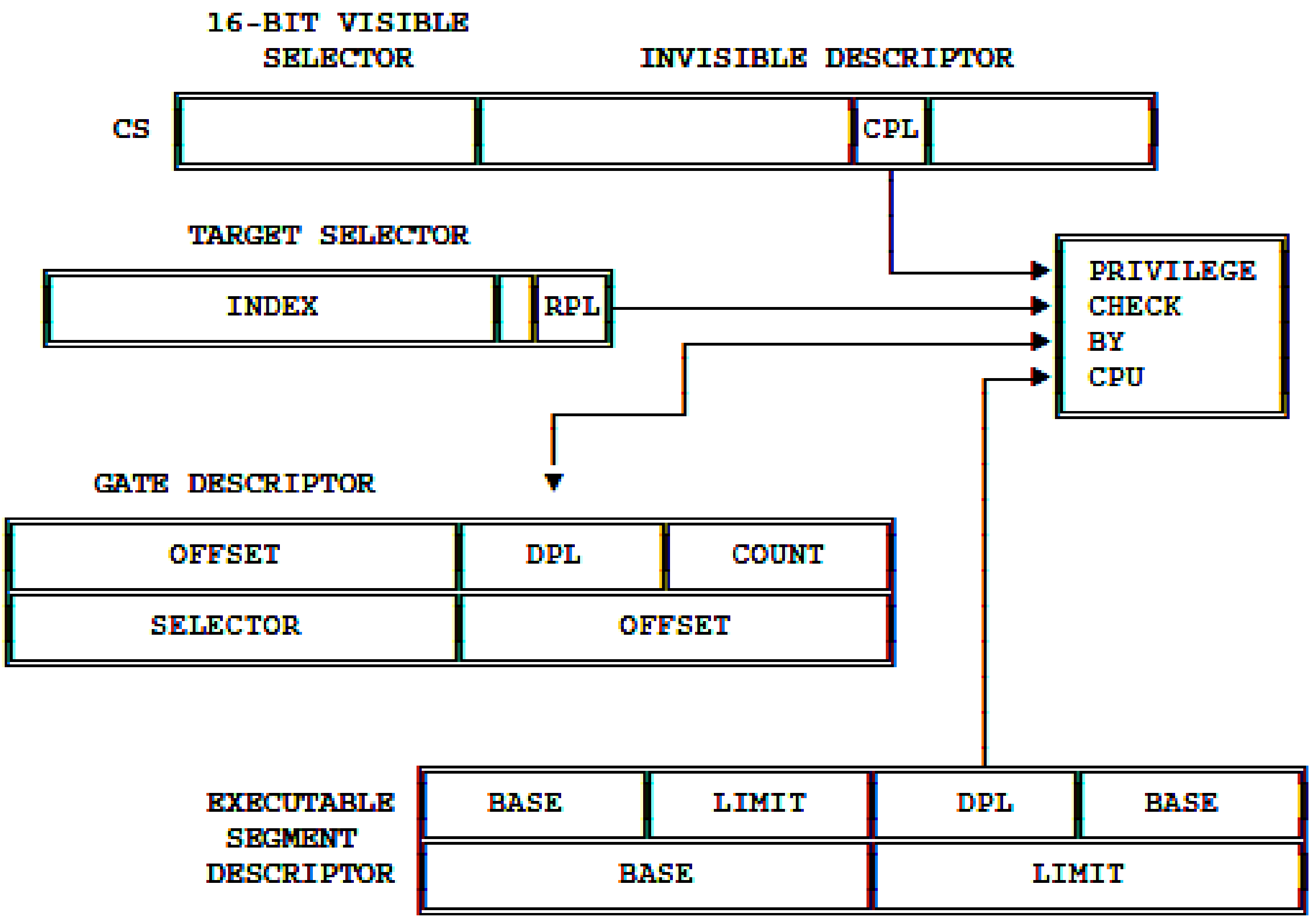
1. CPL
2. RPL of the selector used to specify call gate
3. DPL of the gate descriptor
4. DPL of the descriptor of target segment.

- Only CALL instruction can use gates to transfer to smaller privilege levels.

With Call Gate

- For a JMP instruction, the privilege rules are
  $\text{MAX(CPL,RPL)} \leq \text{gate DPL}$
  $\text{target segment DPL} = \text{CPL(numerically)}$
- For a CALL instruction, the rules are
  $\text{MAX(CPL,RPL)} \leq \text{gate DPL}$
  $\text{target segment DPL} \leq \text{CPL(numerically)}$

Privilege Check via Call Gate



CPL	-	CURRENT PRIVILEGE LEVEL
RPL	-	REQUESTOR'S PRIVILEGE LEVEL
DPL	-	DESCRIPTOR PRIVILEGE LEVEL

Unit IV

Protection

Topics To Cover Part I **protection**

- Need of Protection
- Overview of 80386DX Protection Mechanisms
- Segment Level Protection
- Page Level Protection
- Combining Segment and Page Level Protection.

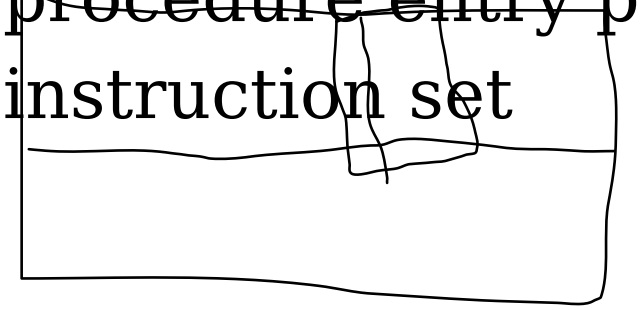
1.Need of Protection

- To help detect and identify bugs
- The 80386 supports sophisticated applications that may consist of hundreds or thousands of program modules.
- In such applications, the question is how bugs can be found and eliminated as quickly as possible and how their damage can be tightly confined.
- To help debug applications faster and make them more robust in production, the 80386 contains mechanisms to verify memory accesses and instruction execution for conformance to protection criteria.
- These mechanisms may be used or ignored, according to system design objectives.

2. Overview of 80386 Protection Mechanisms

Protection in the 80386 has five aspects:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set



- The protection hardware of the 80386 is an integral part of the memory management hardware.
- Protection applies both to segment translation and to page translation.
- Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria.
- All these checks are made before the memory cycle is started
- Any violation prevents that cycle from starting and results in an exception.
- Since the checks are performed concurrently with address formation, there is no performance penalty.

- Invalid attempts to access memory result in an exception.
- Protection violations that leads to exceptions

[illegible]

“Privilege” ?????

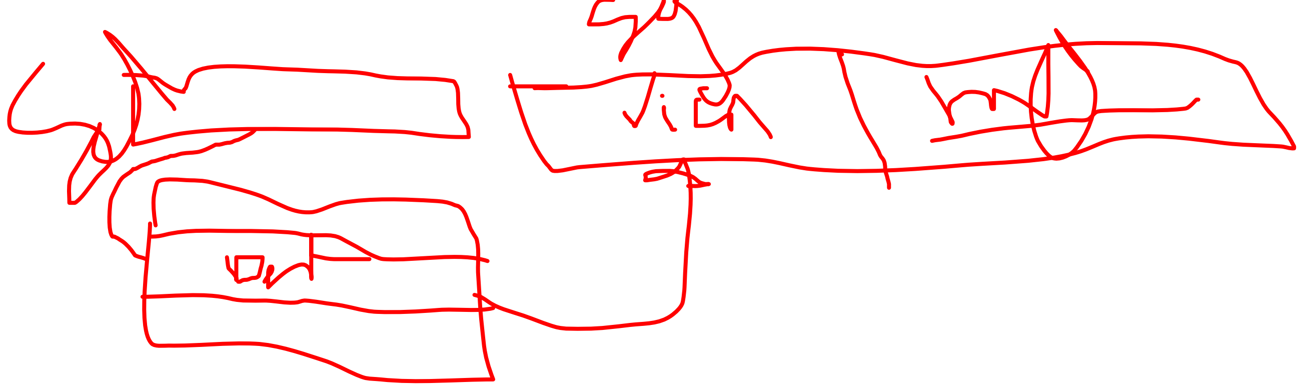
- The concept of "privilege" is central to several aspects of protection.
- Applied to procedures, privilege is the degree to which the procedure can be trusted not to make a mistake that might affect other procedures or data.
- Applied to data, privilege is the degree of protection that a data structure should have from less trusted procedures.

- The concept of privilege applies both to segment protection and to page protection.

3. Segment-Level Protection

All five aspects of protection apply to segment translation:

1. Type checking ✓
2. Limit checking ✓
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set



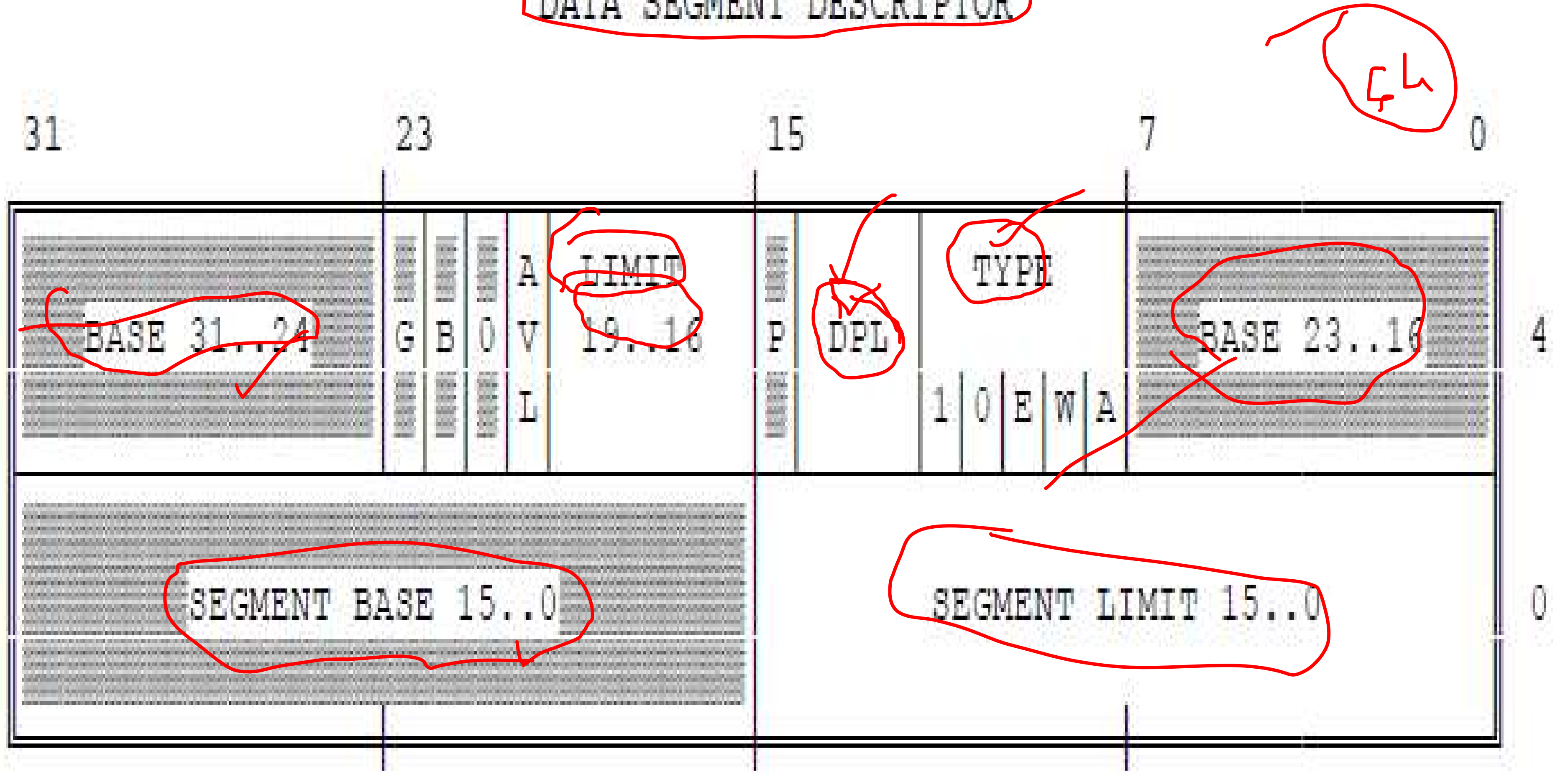
- The segment is the unit of protection, and segment descriptors store protection parameters.
- Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access.
- Segment registers hold the protection parameters of the currently addressable segments.

3.1 Descriptors Store Protection Parameters

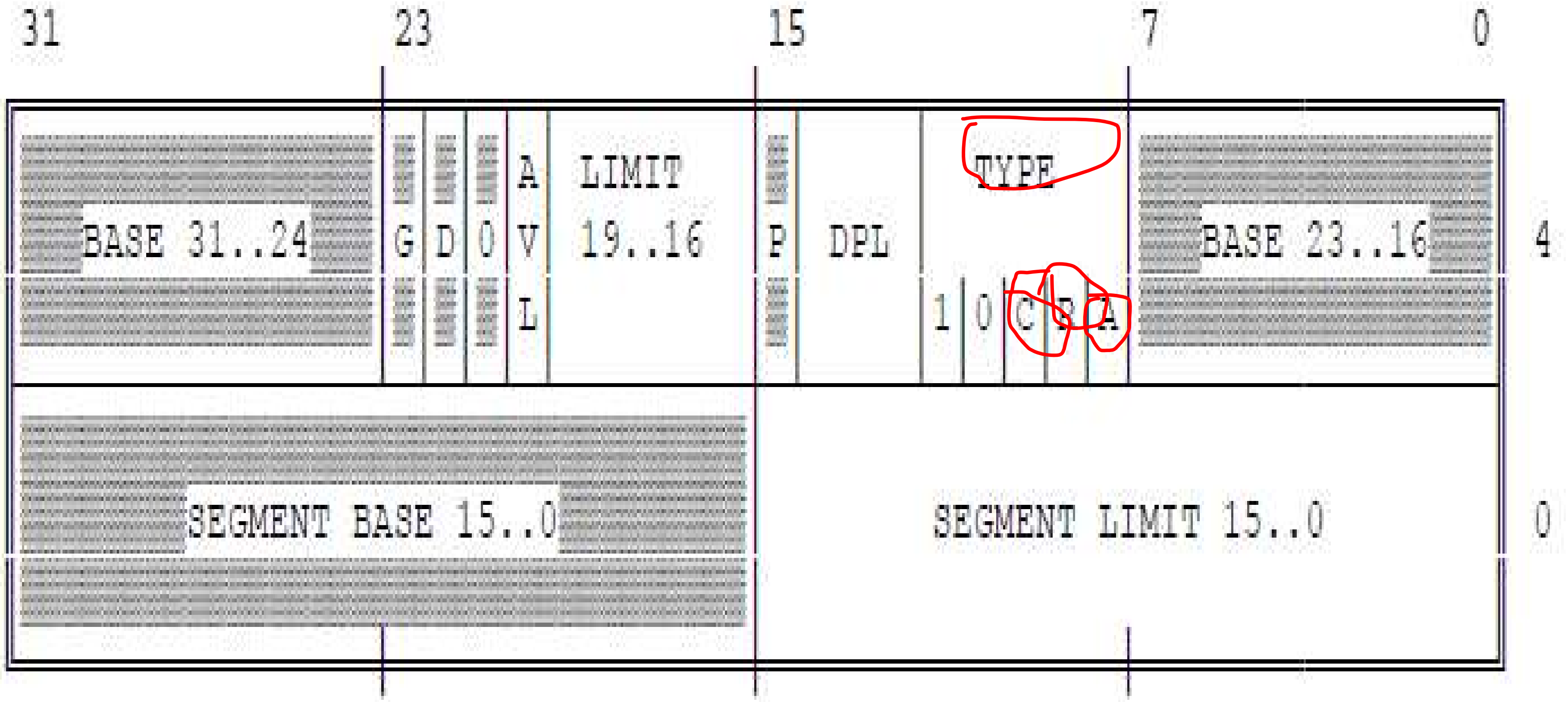
- Figure (Next slide) highlights the protection-related fields of segment descriptors.
- The protection parameters are placed in the descriptor by systems software at the time a descriptor is created.
- In general, applications programmers do not need to be concerned about protection parameters.

- When a program loads a selector into a segment register, the processor loads not only the base address of the segment but also protection information.
- Each segment register has bits in the invisible portion for storing base, limit, type, and privilege level
- That’s why, subsequent protection checks on the same segment do not consume additional clock cycles.

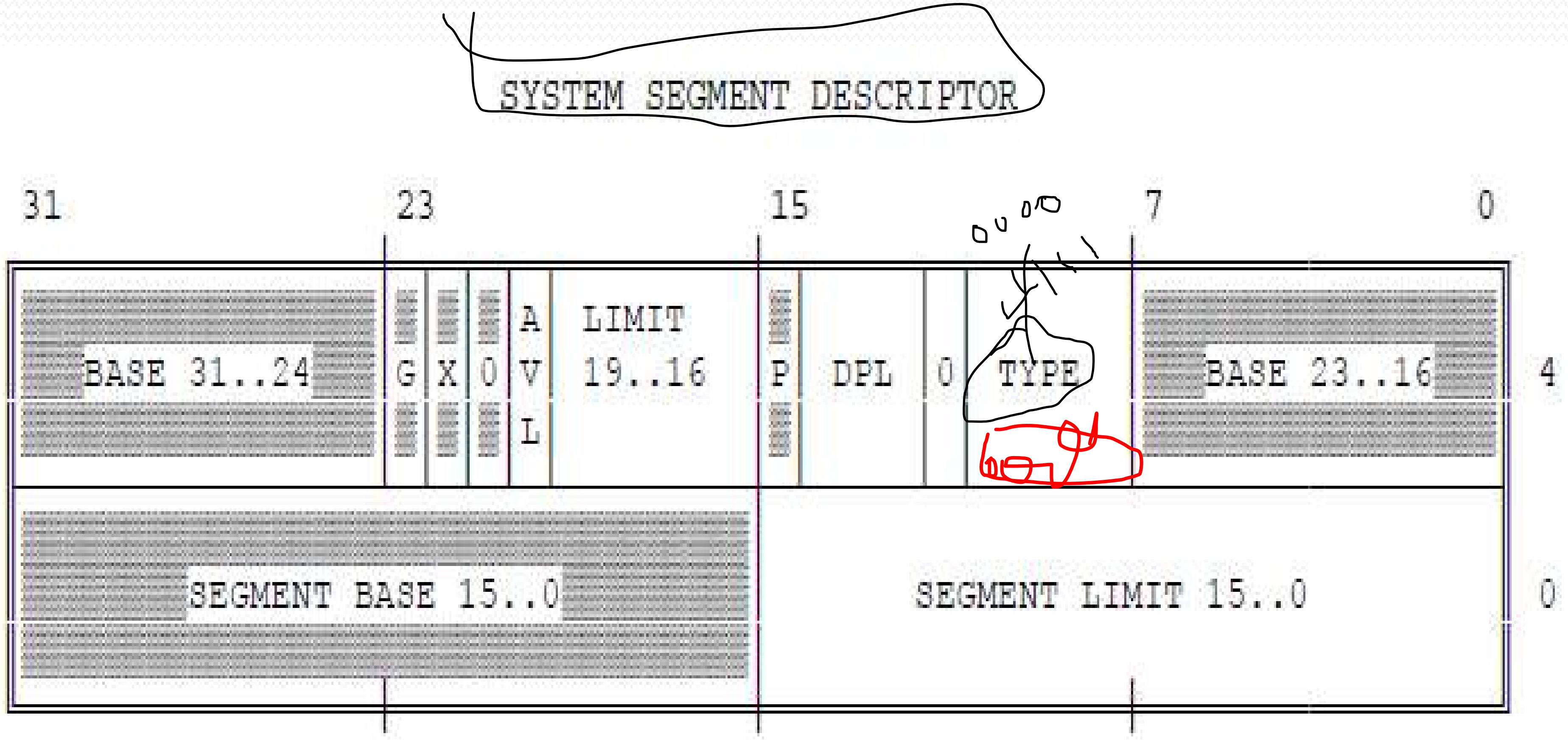
DATA SEGMENT DESCRIPTOR



EXECUTABLE SEGMENT DESCRIPTOR



SYSTEM SEGMENT DESCRIPTOR



A	-	ACCESSED	E	-	EXPAND-DOWN
AVL	-	AVAILABLE FOR PROGRAMMERS USE	G	-	GRANULARITY
B	-	BIG	P	-	SEGMENT PRESENT
C	-	CONFORMING	R	-	READABLE
D	-	DEFAULT	W	-	WRITABLE
DPL	-	DESCRIPTOR PRIVILEGE LEVEL			

Figure 6-1. Protection Fields of Segment Descriptors

3.1.1 Type Checking

- The TYPE field of a descriptor has two functions:

1. It distinguishes among different descriptor formats.
2. It specifies the intended usage of a segment.

- Besides the descriptors for data and executable segments commonly used by applications programs, the 80386 has descriptors for special segments used by the operating system and for gates.
- Table (Next) lists all the types defined for system segments and gates.

Table 6-1. System and Gate Descriptor Types

Code	Type of Segment or Gate
0 000	-reserved
1	Available 286 TSS
2	LDT
3 0011	Busy 286 TSS
4	Call Gate
5	Task Gate
6	286 Interrupt Gate
7	286 Trap Gate
8	-reserved
9	Available 386 TSS
A	-reserved
B	Busy 386 TSS
C	386 Call Gate
D	-reserved
E	386 Interrupt Gate
F	386 Trap Gate

- The type fields of data and executable segment descriptors include bits which further define the purpose of the segment:
    - The writable bit in a data-segment descriptor specifies whether instructions can write into the segment.

    - The readable bit in an executable-segment descriptor specifies whether instructions are allowed to read from the segment (for example, to access constants that are stored with instructions).

- A readable, executable segment may be read in two ways:

1. Via the CS register, by using a CS override prefix.
2. By loading a selector of the descriptor into a data-segment register (DS, ES, FS, or GS).

- Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer.

- The processor examines type information on two kinds of occasions:

1. When a selector of a descriptor is loaded into a segment register.
2. When an instruction refers (implicitly or explicitly) to a segment register.

1. When a selector of a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types

- The CS register can be loaded only with a selector of an executable segment.
- Selectors of executable segments that are not readable cannot be loaded into data-segment registers.
- Only selectors of writable data segments can be loaded into SS.

2. When an instruction refers (implicitly or explicitly) to a segment register. Certain segments can be used by instructions only in certain predefined ways

- No instruction may write into an executable segment.
- No instruction may write into a data segment if the writable bit is not set.
- No instruction may read an executable segment unless the readable bit is set.

3.1.2 Limit Checking

- The limit field of a segment descriptor is used by the processor to prevent programs from addressing outside the segment.
- The processor's interpretation of the limit depends on the setting of the G (granularity) bit.
- For data segments, the processor's interpretation of the limit depends also on the E-bit (expansion-direction bit) and the B-bit (big bit)
- Table next

Table 6-2. Useful Combinations of E, G, and B Bits

Case:	1	2	3	4
Expansion Direction	U	U	D	D
G-bit	0	1	0	1
B-bit	X	X	0	1
Lower bound is:				
0	X	X		
LIMIT+1			X	
shl(LIMIT,12,1)+1				X
Upper bound is:				
LIMIT	X			
shl(LIMIT,12,1)		X		
64K-1			X	
4G-1				X
Max seg size is:				
64K	X			
64K-1		X		
4G-4K			X	
4G				X
Min seg size is:				
0	X	X		
4K			X	X

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

- When G=0, the actual limit is the value of the 20-bit limit field as it appears in the descriptor.
- In this case, the limit may range from 0 to 0FFFFFH (2^{20}-1 or 1 MB).
- When G=1, the processor appends 12 low-order one-bits to the value in the limit field.
- In this case the actual limit may range from 0FFFH (2^{12}-1 or 4 kilobytes) to 0FFFFFFFFH(2^{32}-1 or 4 GB).

- For all types of segments except expand-down data segments, the value of the limit is one less than the size (expressed in bytes) of the segment.
- The processor causes a general-protection exception in any of these cases:

1. Attempt to access a memory byte at an address > limit.
2. Attempt to access a memory word at an address ≥limit.
3. Attempt to access a memory doubleword at an address ≥(limit-2).

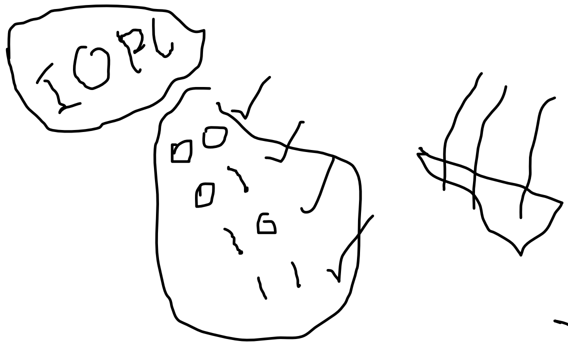
- For expand-down data segments, the limit has the same function but is interpreted differently.
- In these cases the range of valid addresses is from limit + 1 to either 64K or 2^{32}-1 (4 Gbytes) depending on the B-bit.

- An expand-down segment has maximum size when the limit is zero.
- The expand-down feature makes it possible to expand the size of a stack by copying it to a larger segment without needing also to update intrastack pointers.

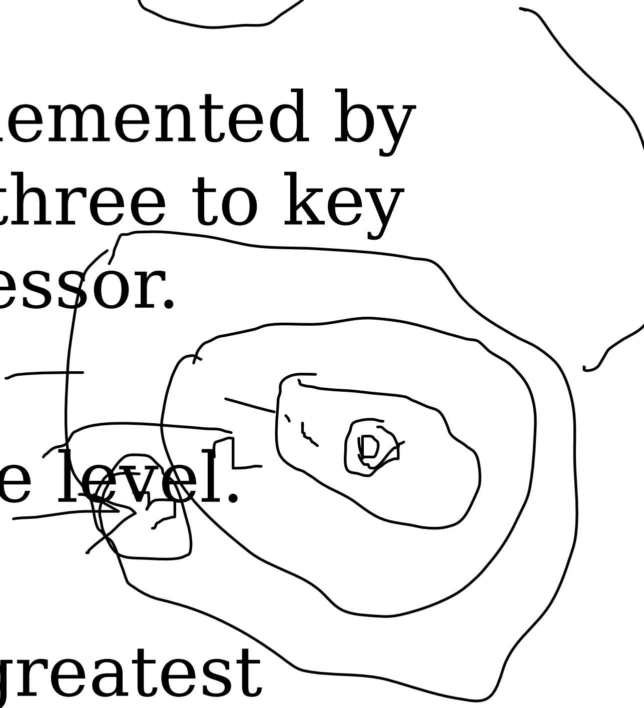
- The limit field of descriptors for descriptor tables is used by the processor to prevent programs from selecting a table entry outside the descriptor table.
- The limit of a descriptor table identifies the last valid byte of the last descriptor in the table.
- Since each descriptor is eight bytes long, the limit value is N * 8 - 1 for a table that can contain up to N descriptors.

- Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations.
- Such errors are detected when they occur, so that identification of the cause is easier.
- Without limit checking, such errors could corrupt other modules; the existence of such errors would not be discovered until later, when the corrupted module behaves incorrectly, and when identification of the cause is difficult.

3.1.3 Privilege Levels



- The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor.
- This value is called the privilege level.
- The value zero represents the greatest privilege, the value three represents the least privilege.



- The following processor-recognized objects contain privilege levels:

1. Descriptors contain a field called the descriptor privilege level (DPL).
2. Selectors contain a field called the requestor's privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates a selector.
3. An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

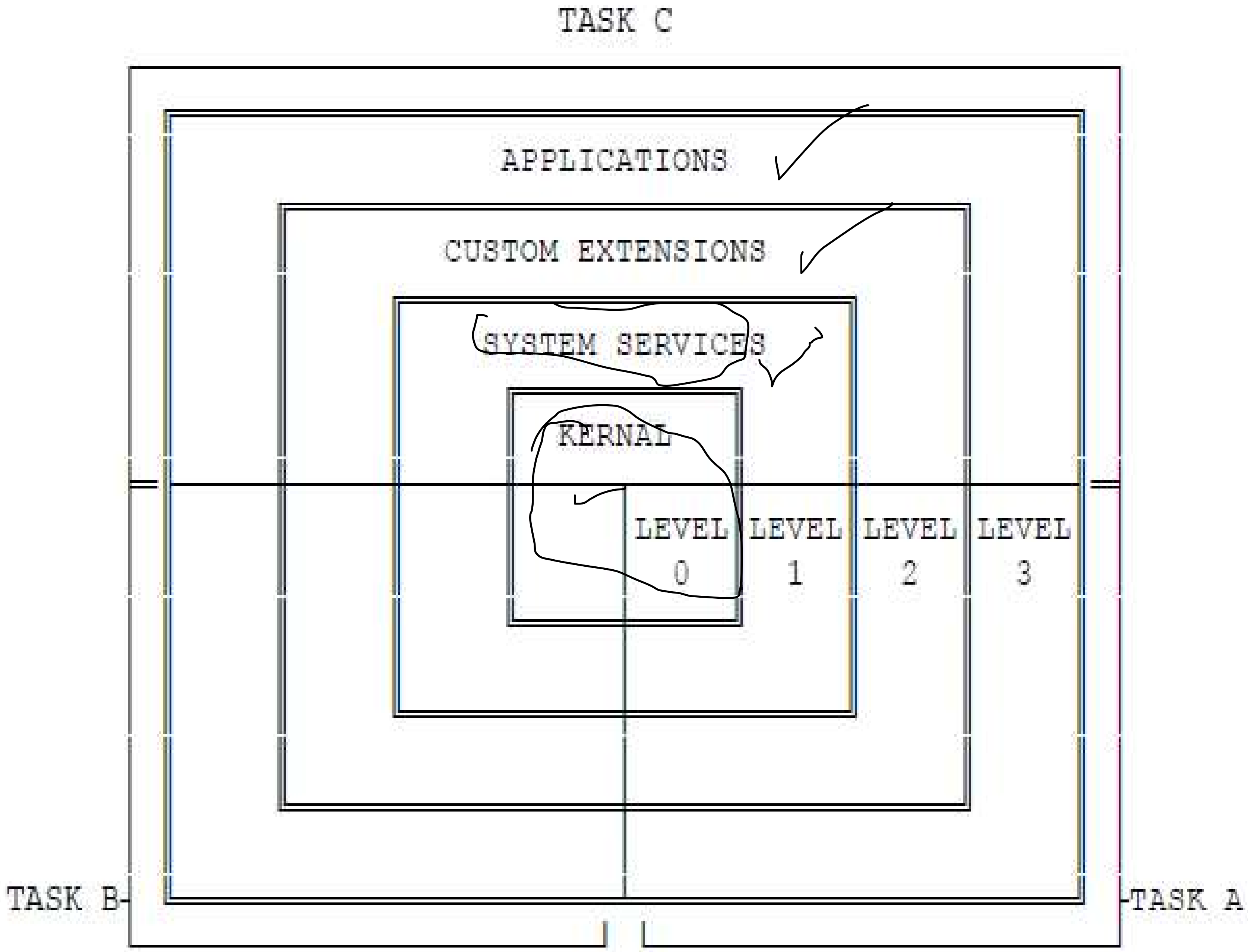
- The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels.

- The evaluation is performed at the time the selector of a descriptor is loaded into a segment register.

- The criteria used for evaluating access to data differs from that for evaluating transfers of control to executable segments:

- Diagram
- The levels of privilege can be interpreted as rings of protection.
- The center is for the segments containing the most critical software, usually the kernel of the operating system.
- Outer rings are for the segments of less critical software.

Figure 6-2. Levels of Privilege



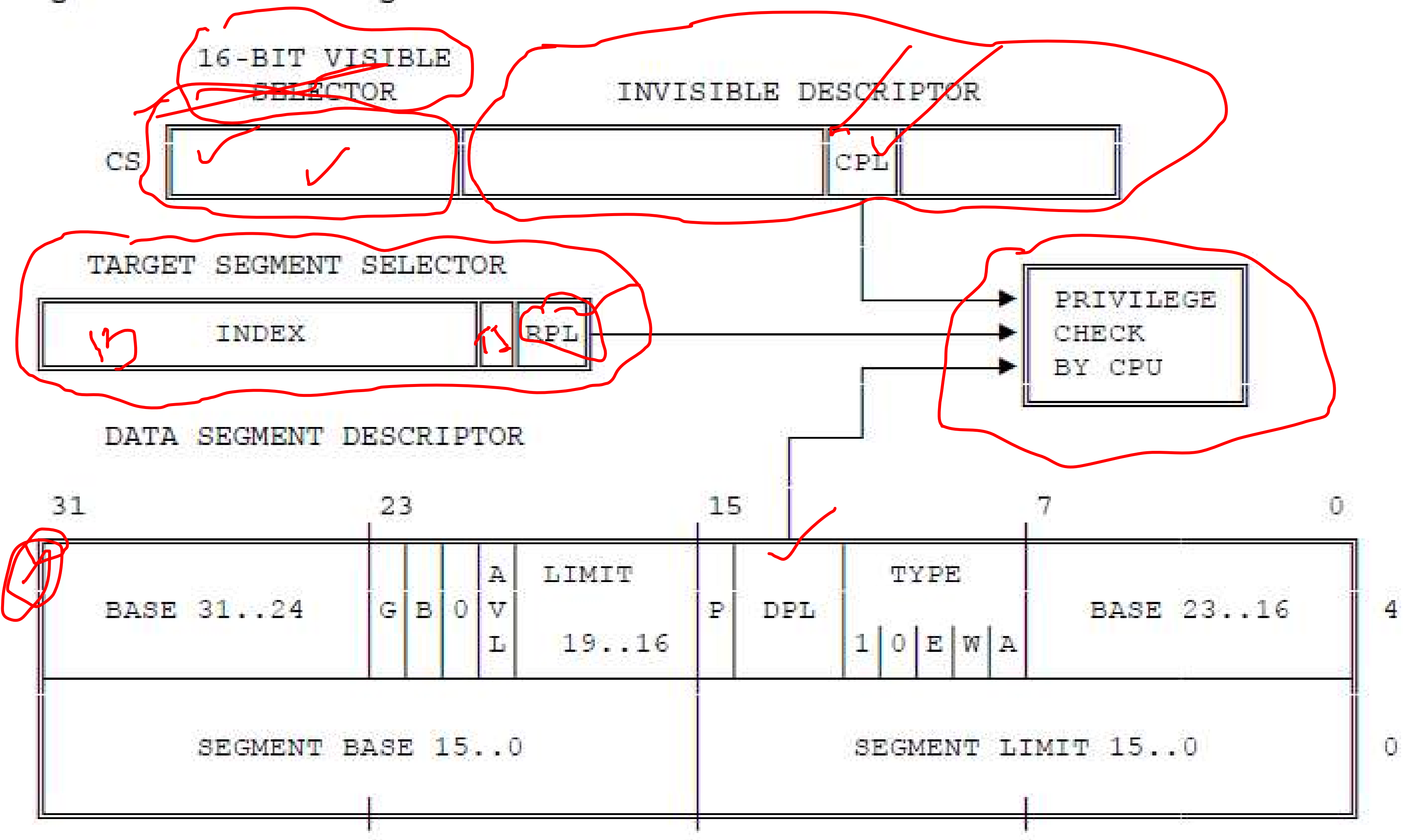
- It is not necessary to use all four privilege levels.
- Existing software that was designed to use only one or two levels of privilege can simply ignore the other levels offered by the 80386.
- A one-level system should use privilege level zero; a two-level system should use privilege levels zero and three.

3.2 Restricting Access to Data

- To address operands in memory, an 80386 program must load the selector of a data segment into a data-segment register (DS, ES, FS, GS, SS).
- The processor automatically evaluates access to a data segment by comparing privilege levels.
- The evaluation is performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register.
- 3 different privilege levels enter into this type of privilege check Diagram (Next) :

1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the selector used to specify the target segment.
3. The DPL of the descriptor of the target segment.

Figure 6-3. Privilege Check for Data Access



CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

- Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL.

$DPL \geq [\max(CPL, RPL)]$

- In other words, a procedure can only access data that is at the same or less privileged level.

- The addressable domain of a task varies as CPL changes.

1. When CPL is zero, data segments at all privilege levels are accessible
2. when CPL is one, only data segments at privilege levels one through three are accessible
3. when CPL is three, only data segments at privilege level three are accessible.

- This property of the 80386 can be used, for example, to prevent applications procedures from reading or changing tables of the operating system.

Note

- **Conforming Segment:** These segments may be called from various privilege levels but execute at the privilege level of the calling procedure. (e.g. math library)

3.2.1 Accessing Data in Code Segments

- Less common than the use of data segments is the use of code segments to store data.
- Code segments may legitimately hold constants; it is not possible to write to a segment described as a code segment.
- The following methods of accessing data in code segments are possible:

1. Load a data-segment register with a selector of a nonconforming, readable, executable segment.

2. Load a data-segment register with a selector of a conforming,

readable, executable segment.

3. Use a CS override prefix to read a readable, executable segment whose selector is already loaded in the CS register.

- The same rules as for access to data segments apply to case 1.
- Case 2 is always valid because the privilege level of a segment whose conforming bit is set is effectively the same as CPL regardless of its DPL.
- Case 3 always valid because the DPL of the code segment in CS is, by definition, equal to CPL.

3.3 Restricting Control Transfers

- With the 80386, control transfers are accomplished by the instructions JMP, CALL, RET, INT, and IRET, as well as by the exception and interrupt mechanisms.
- Exceptions and interrupts are special cases
- discussion of JMP, CALL, and RET instructions (Here only)
- The "near" forms of JMP, CALL, and RET transfer within the current code segment, and therefore are subject only to limit checking.
- The processor ensures that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current executable segment.
- This limit is cached in the CS register; therefore, protection checks for near transfers require no extra clock cycles.

[illegible]

- The operands of the "far" forms of JMP and CALL refer to other segments
- So the processor performs privilege checking.
- There are two ways a JMP or CALL can refer to another segment:

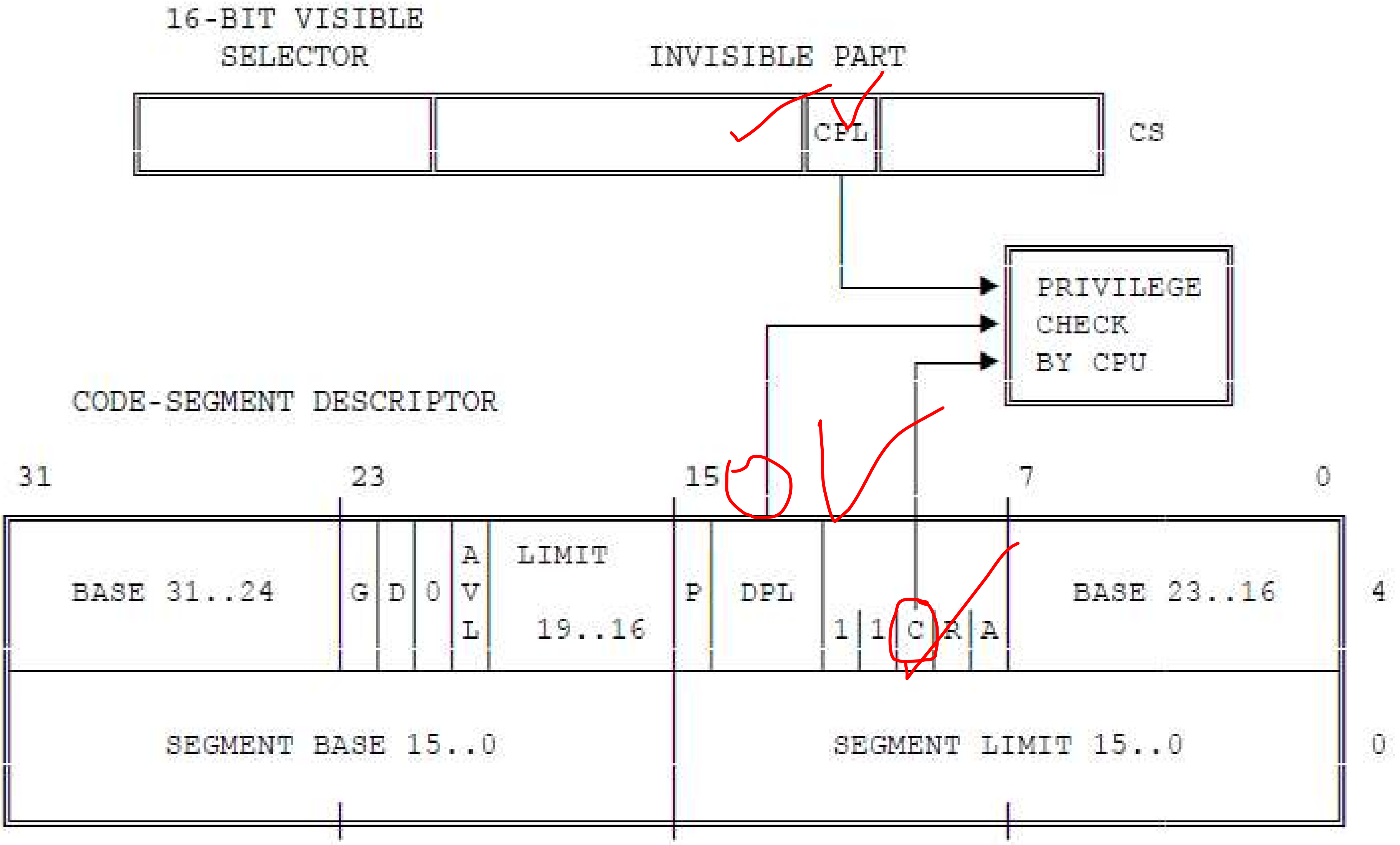
1. The operand selects the descriptor of another executable segment.
2. The operand selects a call gate descriptor.

- Diagram (Next)
- 2 different privilege levels enter into a privilege check for a control transfer that does not use a call gate:

1. The CPL (current privilege level).
2. The DPL of the descriptor of the target segment.

- Normally the CPL is equal to the DPL of the segment that the processor is currently executing.
- CPL may, however, be greater than DPL if the conforming bit is set in the descriptor of the current executable segment.
- The processor keeps a record of the CPL cached in the CS register; this value can be different from the DPL in the descriptor of the code segment.

Figure 6-4. Privilege Check for Control Transfer without Gate



CPL - CURRENT PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL
C - CONFORMING BIT

- The processor permits a JMP or CALL directly to another segment only if one of the following privilege rules is satisfied:

1. DPL of the target is equal to CPL.

OR

2. The conforming bit of the target code-segment descriptor is set, and the DPL of the target is less than or equal to CPL.

- An executable segment whose descriptor has the conforming bit set is called a conforming segment.
- The conforming-segment mechanism permits sharing of procedures that may be called from various privilege levels but should execute at the privilege level of the calling procedure.
- Examples of such procedures include math libraries and some exception handlers.
- When control is transferred to a conforming segment, the CPL does not change.
- This is the only case when CPL may be unequal to the DPL of the current executable segment.

- Most code segments are not conforming.
- The basic rules of privilege : for nonconforming segments, control can be transferred without a gate only to executable segments at the same level of privilege.
- There is a need, however, to transfer control to (numerically) smaller privilege levels
- This need is met by the CALL instruction when used with call-gate descriptors,.
- The JMP instruction may never transfer control to a nonconforming segment whose DPL does not equal CPL.

3.4 Gate Descriptors Guard Procedure Entry Points

- To provide protection for control transfers among executable segments at different privilege levels, the 80386 uses gate descriptors.
- There are four kinds of gate descriptors:

1. Call gates
2. Trap gates
3. Interrupt gates
4. Task gates

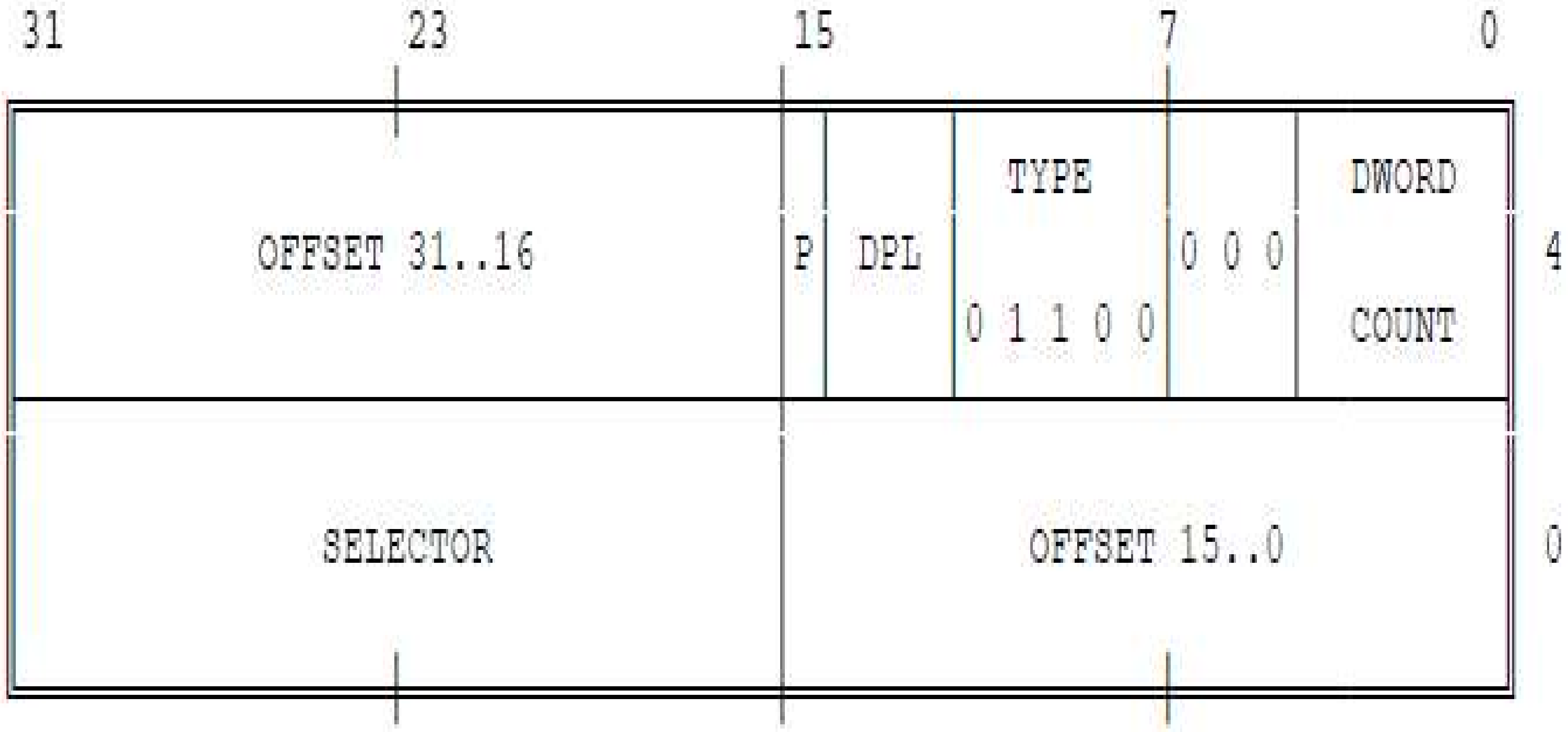
- Task gates are used for task switching
- Trap gates and interrupt gates are used by exceptions and interrupts
- A call gate descriptor may reside in the GDT or in an LDT, but not in the IDT.

Call Gate

- A call gate has two primary functions:

1. To define an entry point of a procedure.
2. To specify the privilege level of the entry point.

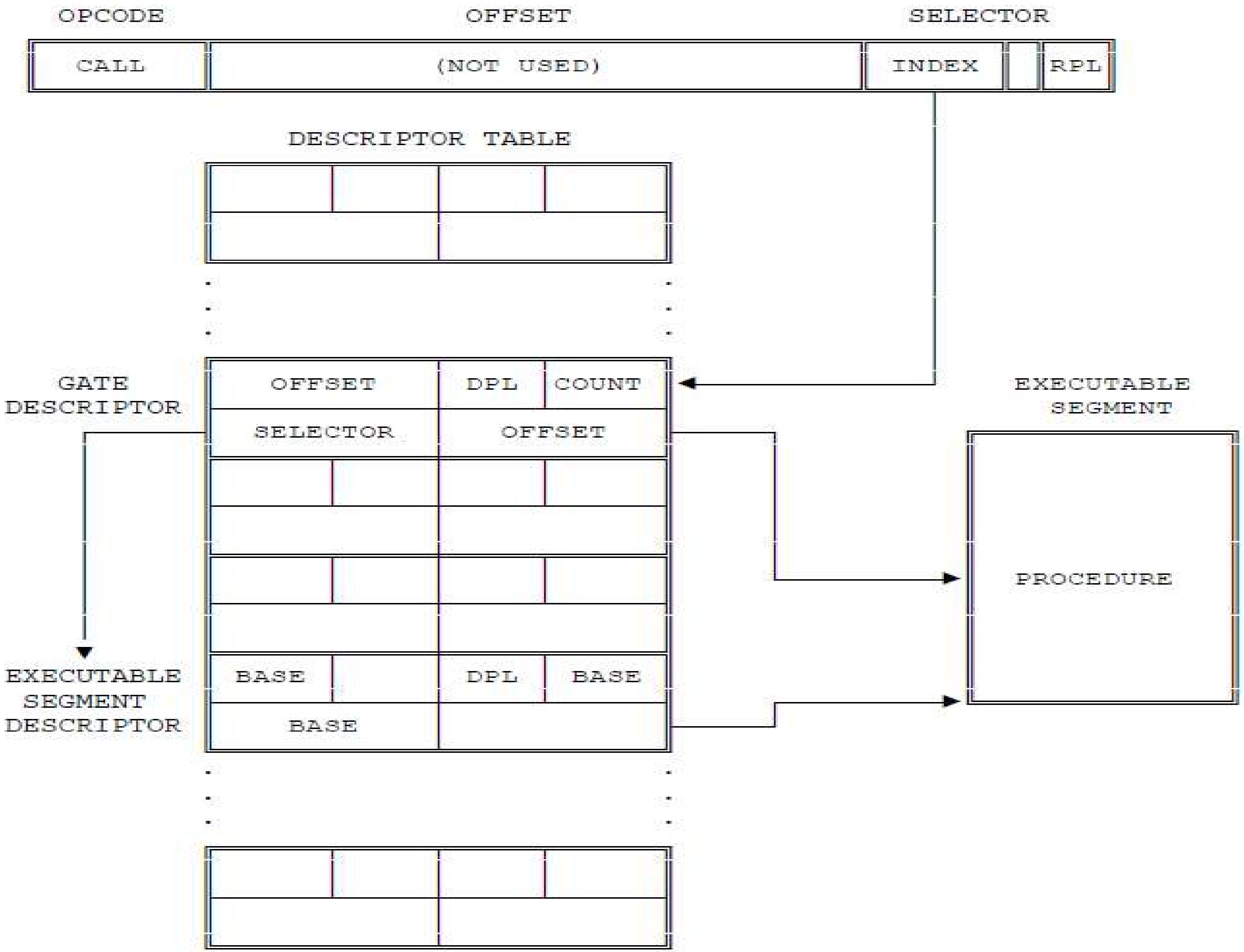
Figure 6-5. Format of 80386 Call Gate



- Call gate descriptors are used by call and jump instructions in the same manner as code segment descriptors.
- When the hardware recognizes that the destination selector refers to a gate descriptor, the operation of the instruction is expanded as determined by the contents of the call gate.

- The selector and offset fields of a gate form a pointer to the entry point of a procedure.
- A call gate guarantees that all transitions to another segment go to a valid entry point, rather than possibly into the middle of a procedure (or worse, into the middle of an instruction).
- The far pointer operand of the control transfer instruction does not point to the segment and offset of the target instruction
- rather, the selector part of the pointer selects a gate, and the offset is not used.
- Diagram (next)

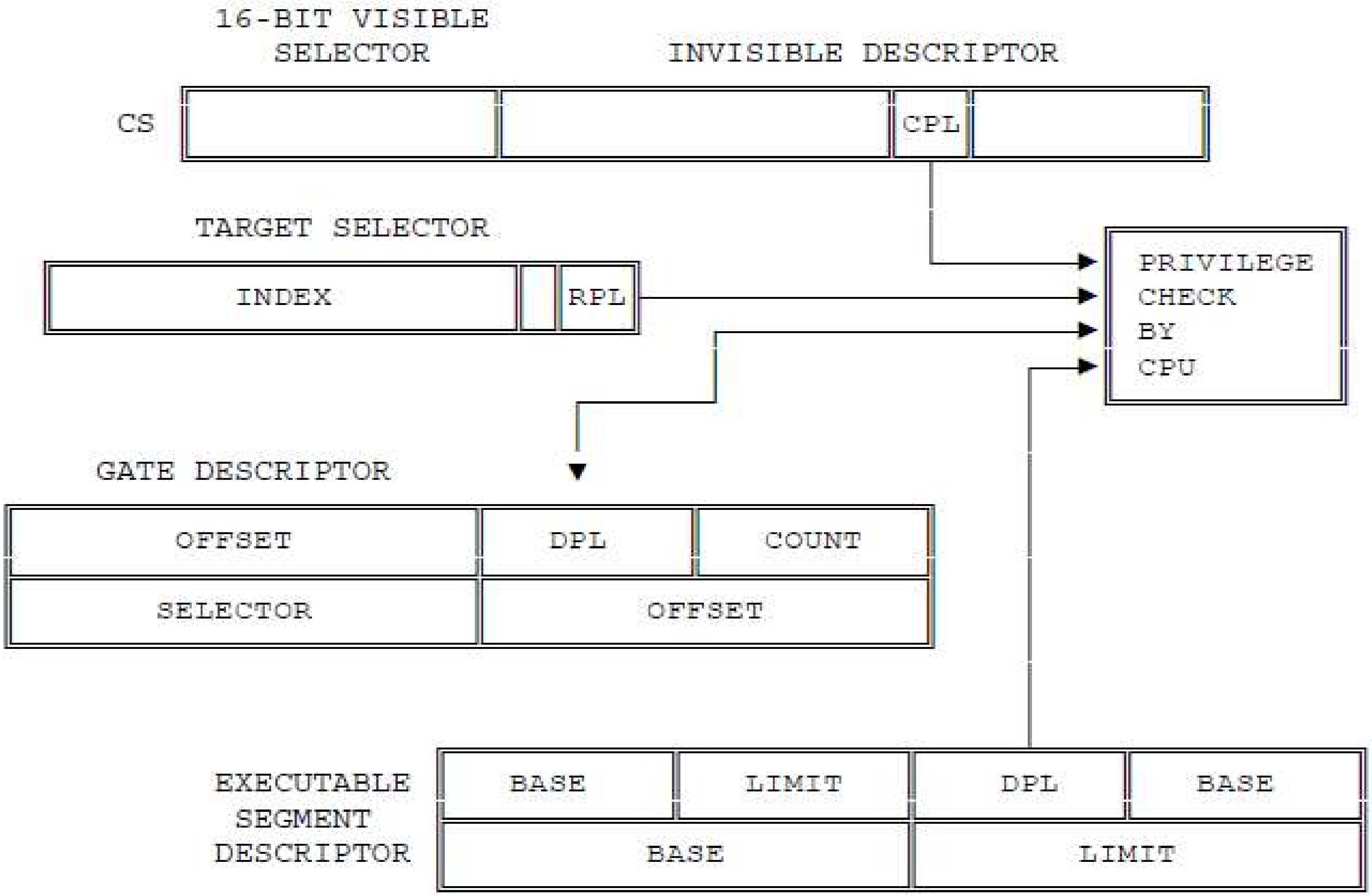
Figure 6-6. Indirect Transfer via Call Gate



- Diagram (Next)
- 4 different privilege levels are used to check the validity of a control transfer via a call gate:

1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the selector used to specify the call gate.
3. The DPL of the gate descriptor.
4. The DPL of the descriptor of the target executable segment.

Figure 6-7. Privilege Check via Call Gate



CPL	-	CURRENT PRIVILEGE LEVEL
RPL	-	REQUESTOR'S PRIVILEGE LEVEL
DPL	-	DESCRIPTOR PRIVILEGE LEVEL

- The DPL field of the gate descriptor determines what privilege levels can use the gate.

- One code segment can have several procedures that are intended for use by different privilege levels.

- For example, an operating system may have some services that are intended to be used by applications, whereas others may be intended only for use by other systems software.

Use of Gates

- Gates can be used for control transfers to numerically smaller privilege levels or to the same privilege level (though they are not necessary for transfers to the same level).

- Only CALL instructions can use gates to transfer to smaller privilege levels.

- A gate may be used by a JMP instruction only to transfer to an executable segment with the same privilege level or to a conforming segment.

- For a JMP instruction to a nonconforming segment, both of the following privilege rules must be satisfied :

1. MAX (CPL,RPL) $\leq$ gate DPL
2. target segment DPL = CPL

- otherwise, a general protection exception results.

- For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied:

1. MAX (CPL,RPL) ≤ gate DPL
2. target segment DPL ≤ CPL

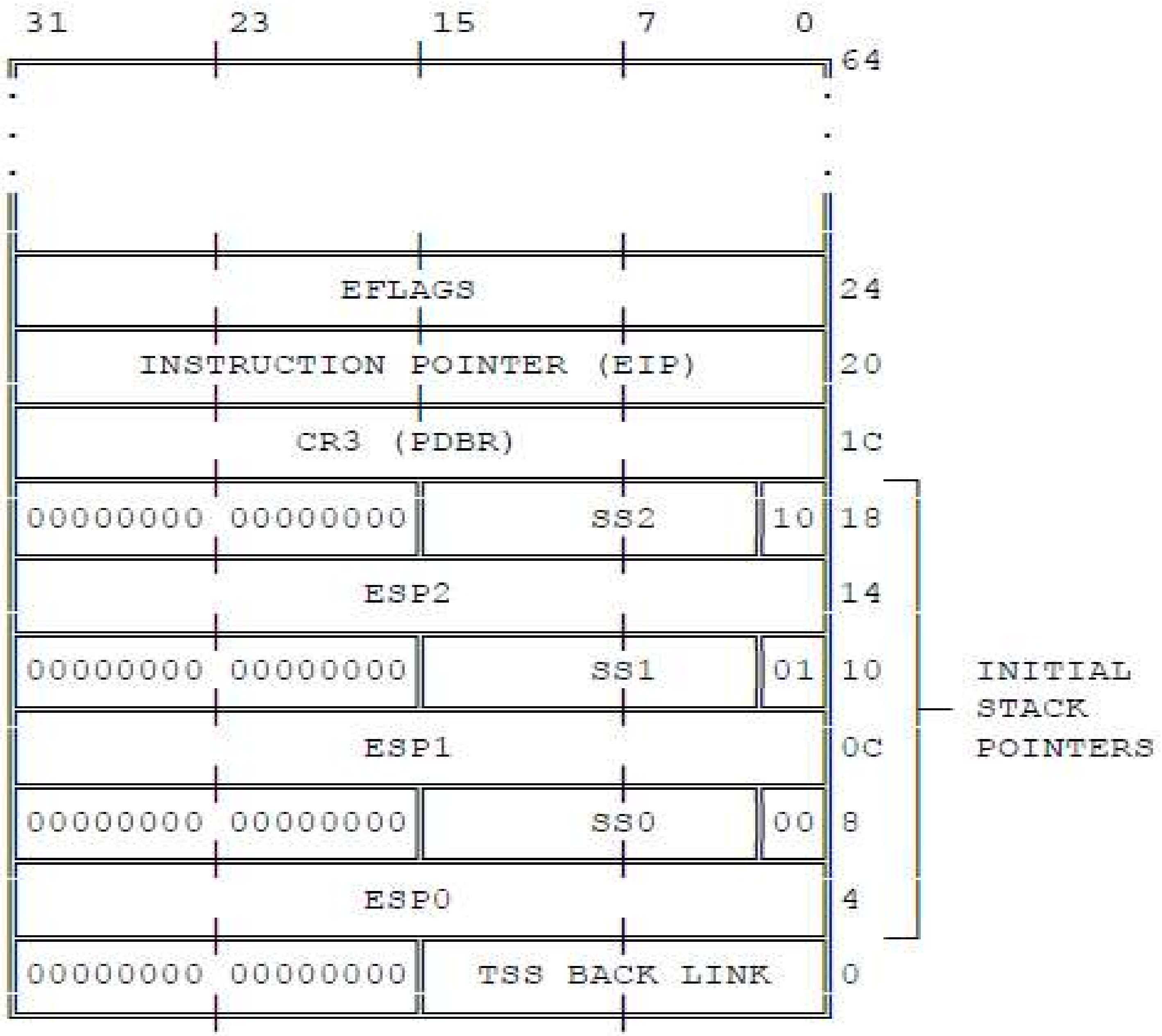
- otherwise, a general protection exception results.

3.4.1 Stack Switching

- If the destination code segment of the call gate is at a different privilege level than the CPL, an interlevel transfer is being requested.
- To maintain system integrity, each privilege level has a separate stack.
- These stacks assure sufficient stack space to process calls from less privileged levels.
- Without them, a trusted procedure would not work correctly if the calling procedure did not provide sufficient space on the caller's stack.

- The processor locates these stacks via the task state segment
- Diagram (next)
- Each task has a separate TSS, thereby permitting tasks to have separate stacks.
- Systems software is responsible for creating TSSs and placing correct stack pointers in them.
- The initial stack pointers in the TSS are strictly read-only values.
- The processor never changes them during the course of execution.

Figure 6-8. Initial Stack Pointers of TSS



- When a call gate is used to change privilege levels, a new stack is selected by loading a pointer value from the Task State Segment (TSS).
- The processor uses the DPL of the target code segment (the new CPL) to index the initial stack pointer for PL 0, PL 1, or PL 2.

- The DPL of the new stack data segment must equal the new CPL
- if it does not, a stack exception occurs.
- It is the responsibility of systems software to create stacks and stack-segment descriptors for all privilege levels that are used.
- Each stack must contain enough space to hold the old SS:ESP, the return address, and all parameters and local variables that may be required to process a call.

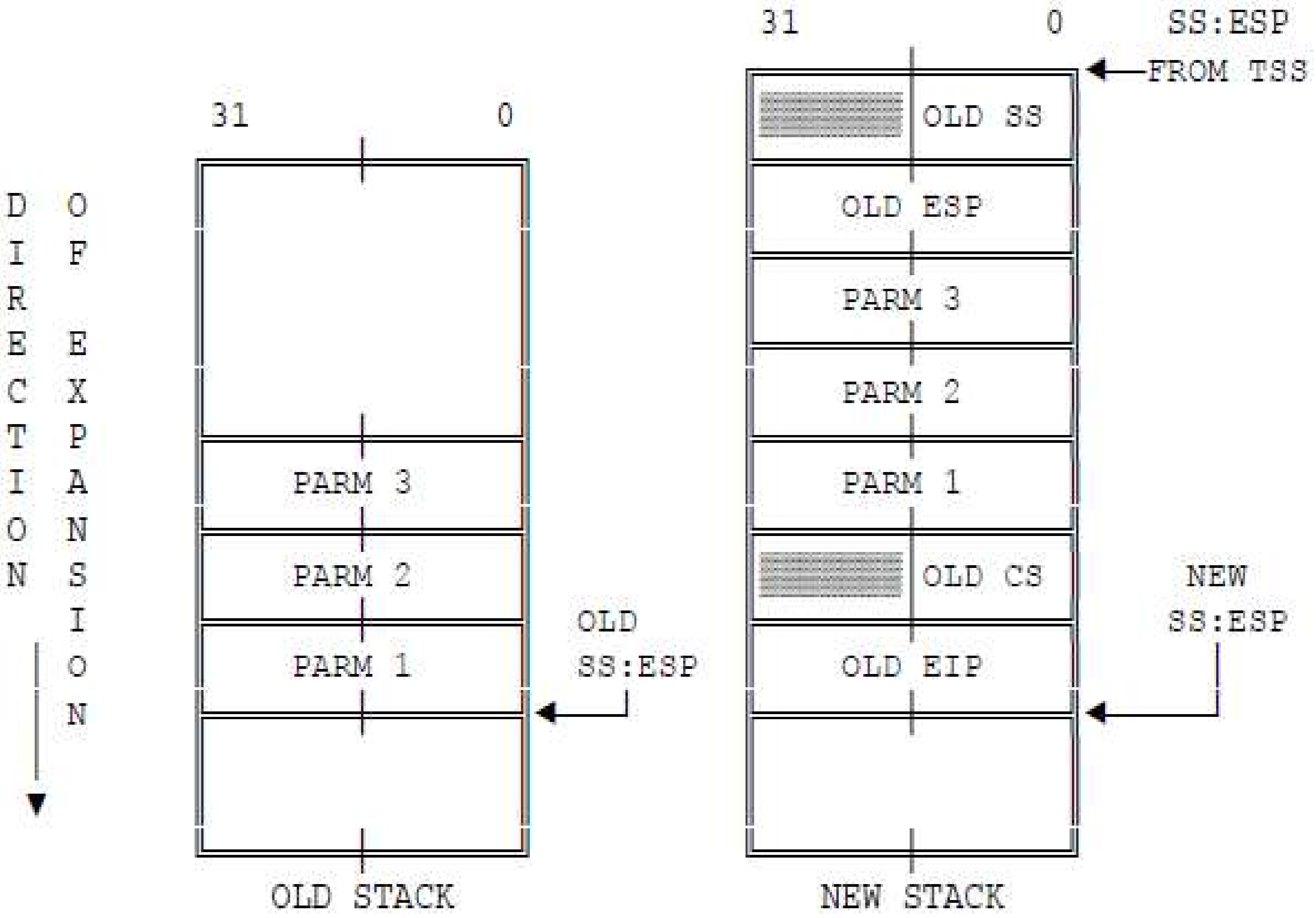
- As with intralevel calls, parameters for the subroutine are placed on the stack.
- To make privilege transitions transparent to the called procedure, the processor copies the parameters to the new stack.
- The count field of a call gate tells the processor how many doublewords (up to 31) to copy from the caller's stack to the new stack.
- If the count is zero, no parameters are copied.

- The processor performs the following stack-related steps in executing an interlevel CALL :

1. The new stack is checked to assure that it is large enough to hold the parameters and linkages; if it is not, a stack fault occurs with an error code of 0.
2. The old value of the stack registers SS:ESP is pushed onto the new stack as two doublewords.
3. The parameters are copied.
4. A pointer to the instruction after the CALL instruction (the former value of CS:EIP) is pushed onto the new stack. The final value of SS:ESP points to this return pointer on the new stack.

- Diagram :Stack contents after a successful interlevel call.
- The TSS does not have a stack pointer for a privilege level 3 stack, because privilege level 3 cannot be called by any procedure at any other privilege level.

Figure 6-9. Stack Contents after an Interlevel Call



- Procedures that may be called from another privilege level and that require more than the 31 doublewords for parameters must use the saved SS:ESP link to access all parameters beyond the last doubleword copied.

- A call via a call gate does not check the values of the words copied onto the new stack.
- The called procedure should check each parameter for validity.

3.4.2 Returning from a Procedure

- The "near" forms of the RET instruction transfer control within the current code segment and therefore are subject only to limit checking.
- The offset of the instruction following the corresponding CALL, is popped from the stack.
- The processor ensures that this offset does not exceed the limit of the current executable segment.

- The "far" form of the RET instruction pops the return pointer that was pushed onto the stack by a prior far CALL instruction.
- Under normal conditions, the return pointer is valid, because of its relation to the prior CALL or INT.
- However, the processor performs privilege checking because of the possibility that the current procedure altered the pointer or failed to properly maintain the stack.
- The RPL of the CS selector popped off the stack by the return instruction identifies the privilege level of the calling procedure.

- An intersegment return instruction can change privilege levels, but only toward procedures of lesser privilege.
- When the RET instruction encounters a saved CS value whose RPL is numerically greater than the CPL, an interlevel return occurs.
- Steps:

Interlevel Return Steps

Step 1

- The checks (Table) are made, and CS:EIP and SS:ESP are loaded with their former values that were saved on the stack.

Table 6-3. Interlevel Return Checks

Type of Check	Exception	Error Code
SF Stack Fault		
GP General Protection Exception		
NP Segment-Not-Present Exception		
ESP is within current SS segment	SF	0
ESP + 7 is within current SS segment	SF	0
RPL of return CS is greater than CPL	GP	Return CS
Return CS selector is not null	GP	Return CS
Return CS segment is within descriptor table limit	GP	Return CS
Return CS descriptor is a code segment	GP	Return CS
Return CS segment is present	NP	Return CS
DPL of return nonconforming code segment = RPL of return CS, or DPL of return conforming code segment ≤ RPL of return CS	GP	Return CS
ESP + N + 15 is within SS segment		
N Immediate Operand of RET N Instruction	SF	Return SS
SS selector at ESP + N + 12 is not null	GP	Return SS
SS selector at ESP + N + 12 is within descriptor table limit	GP	Return SS
SS descriptor is writable data segment	GP	Return SS
SS segment is present	SF	Return SS
Saved SS segment DPL = RPL of saved CS	GP	Return SS
Saved SS selector RPL = Saved SS segment DPL	GP	Return SS

Step 2

- The old SS:ESP (from the top of the current stack) value is adjusted by the number of bytes indicated in the RET instruction.
- The resulting ESP value is not compared to the limit of the stack segment.
- If ESP is beyond the limit, that fact is not recognized until the next stack operation.

Note: The SS:ESP value of the returning procedure is not preserved; normally, this value is the same as that contained in the TSS.

Step 3

- The contents of the DS, ES, FS, and GS segment registers are checked.
- If any of these registers refer to segments whose DPL is greater than the new CPL (excluding conforming code segments), the segment register is loaded with the null selector (INDEX = 0, TI = 0).
- The RET instruction itself does not signal exceptions in these cases; however, any subsequent memory reference that attempts to use a segment register that contains the null selector will cause a general protection exception.
- This prevents less privileged code from accessing more privileged segments using selectors left in the segment registers by the more privileged procedure.

3.5 Some Instructions are Reserved for Operating System

- Instructions that have the power to affect the protection mechanism or to influence general system performance can only be executed by trusted procedures.
- The 80386 has two classes of such instructions:

1. Privileged instructions: used for system control.
2. Sensitive instructions: used for I/O and I/O related activities.

Class I: **Privileged Instructions**

- The instructions that affect system data structures can only be executed when CPL is zero.
- If the CPU encounters one of these instructions when CPL is greater than zero, it signals a general protection exception.
- These instructions include:

CLTS	**Clear Task–Switched Flag**
HLT	**Halt Processor**
LGDT	**Load GDL Register**
LIDT	**Load IDT Register**
LLDT	**Load LDT Register**
LMSW	**Load Machine Status Word**
LTR	**Load Task Register**
MOV to/from CRn	**Move to Control Register n**
MOV to /from DRn	**Move to Debug Register n**
MOV to/from TRn	**Move to Test Register n**

Class II: **Sensitive Instructions**

- Instructions that deal with I/O need to be restricted but also need to be executed by procedures executing at privilege levels other than zero.

3.6 Instructions for Pointer Validation

- Pointer validation is an important part of locating programming errors.
- Pointer validation is necessary for maintaining isolation between the privilege levels.
- Pointer validation consists of the following steps:

1. Check if the supplier of the pointer is entitled to access the segment.
2. Check if the segment type is appropriate to its intended use.
3. Check if the pointer violates the segment limit.

- 80386 processor automatically performs checks 2 and 3 during instruction execution
- software must assist in performing the first check.
- The unprivileged instruction ARPL is provided for this purpose.
- Software can also explicitly perform steps 2 and 3 to check for potential violations.
- The unprivileged instructions LAR, LSL, VERR, and VERW are provided for this purpose.

LAR : Load Access Rights

- Use: to verify that a pointer refers to a segment of the proper privilege level and type.
- one operand—a selector for a descriptor whose access rights are to be examined.
- The descriptor must be visible at the privilege level which is the max(CPL, selector's RPL).
- If the descriptor is visible, LAR obtains a masked form of the second doubleword of the descriptor, masks this value with 00FxFF00H, stores the result into the specified 32-bit destination register, and sets the zero flag.

LAR…….

- Once loaded, the access-rights bits can be tested.
- All valid descriptor types can be tested by the LAR instruction.
- If the RPL or CPL is greater than DPL, or if the selector is outside the table limit, no access-rights value is returned, and the zero flag is cleared.
- Conforming code segments may be accessed from any privilege level.

LSL : Load Segment Limit

- Allows software to test the limit of a descriptor.
- If the descriptor denoted by the given selector (in memory or a register) is visible at the CPL, LSL loads the specified 32-bit register with a 32-bit, byte granular, unscrambled limit that is calculated from fragmented limit fields and the G-bit of that descriptor.
- This can only be done for segments (data, code, task state, and local descriptor tables); gate descriptors are inaccessible.
- (Table (Next) lists in detail which types are valid and which are not.
- Interpreting the limit is a function of the segment type.
- For example, downward expandable data segments treat the limit differently than code segments do.

Note

- For both LAR and LSL, the zero flag (ZF) is set if the loading was performed; otherwise, the ZF is cleared.

Table 6-4. Valid Descriptor Types for LSL

Type Code	Descriptor Type	Valid?
0	(invalid)	NO
1	Available 286 TSS	YES
2	LDT	YES
3	Busy 286 TSS	YES
4	286 Call Gate	NO
5	Task Gate	NO
6	286 Trap Gate	NO
7	286 Interrupt Gate	NO
8	(invalid)	NO
9	Available 386 TSS	YES
A	(invalid)	NO
B	Busy 386 TSS	YES
C	386 Call Gate	NO
D	(invalid)	NO
E	386 Trap Gate	NO
F	386 Interrupt Gate	NO

3.6.1 Descriptor Validation

- 2 instructions : VERR and VERW
- To determine whether a selector points to a segment that can be read or written at the current privilege level.
- Neither instruction causes a protection fault if the result is negative.

VERR : Verify for Reading

- To verify a segment for reading and to load ZF with 1 if that segment is readable from the current privilege level.
- VERR checks that:
  - ✓ The selector points to a descriptor within the bounds of the GDT or LDT.
  - ✓ It denotes a code or data segment descriptor.
  - ✓ The segment is readable and of appropriate privilege level.

VERR....

- The privilege check for data segments and nonconforming code segments is that the DPL must be numerically greater than or equal to both the CPL and the selector's RPL.

- Conforming segments are not checked for privilege level.

VERW : Verify for Writing

- provides the same capability as VERR for verifying writability.
- VERW loads ZF if the result of the writability check is positive.
- The instruction checks that the descriptor is within bounds, is a segment descriptor, is writable, and that its DPL is numerically greater or equal to both the CPL and the selector's RPL.
- Code segments are never writable, conforming or not.

3.6.2 Pointer Integrity and RPL

- The Requestor's Privilege Level (RPL) feature can prevent inappropriate use of pointers that could corrupt the operation of more privileged code or data from a less privileged level.

ARPL :Adjust Requestor's Privilege Level

- adjusts the RPL field of a selector to become the larger of its original value and the value of the RPL field in a specified register.
- The latter is normally loaded from the image of the caller's CS register which is on the stack.
- If the adjustment changes the selector's RPL, ZF is set
- Otherwise, ZF is cleared.

How…..(Skip)

- To take advantage of the processor's checking of RPL, the called procedure need only ensure that all selectors passed to it have an RPL at least as high (numerically) as the original caller's CPL.
- This action guarantees that selectors are not more trusted than their supplier.
- If one of the selectors is used to access a segment that the caller would not be able to access directly,
- i.e., the RPL is numerically greater than the DPL, then a protection fault will result when that selector is loaded into a segment register.

4. Page-Level Protection

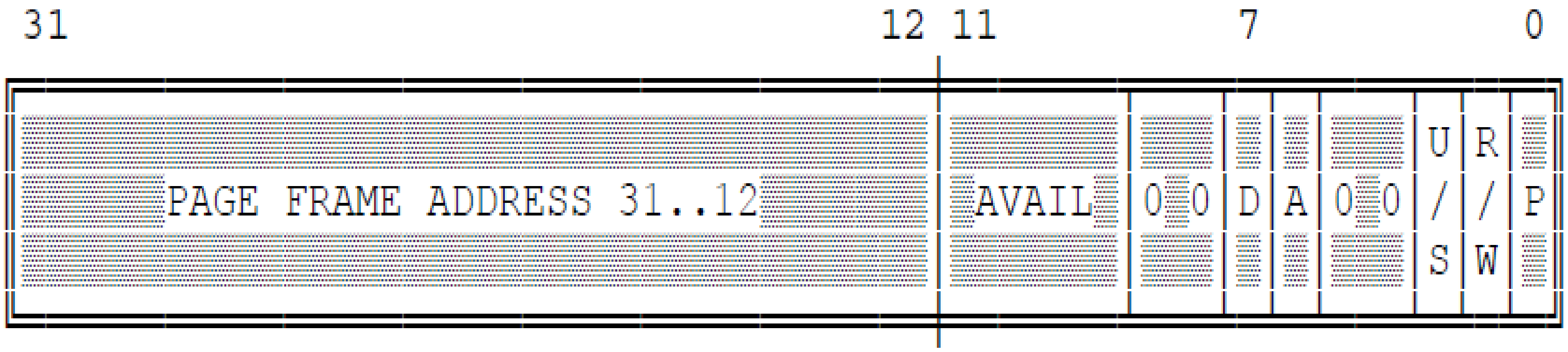
- Two kinds of protection are related to pages:

1. Restriction of addressable domain.
2. Type checking.

4.1 Page-Table Entries Hold Protection Parameters

- Figure highlights the fields of PDEs and PTEs that control access to pages.

Figure 6-10. Protection Fields of Page Table Entries



R/W - READ/WRITE
U/S - USER/SUPERVISOR

4.1.1 Restricting Addressable Domain

- The concept of privilege for pages is implemented by assigning each page to one of two levels:

1. Supervisor level (U/S=0) — for the operating system and other systems software and related data.

OR

2. User level (U/S=1) — for applications procedures and data.

- The current level (U or S) is related to CPL.
- If CPL is 0, 1, or 2, the processor is executing at supervisor level.
- If CPL is 3, the processor is executing at user level.

- When the processor is executing at supervisor level, all pages are addressable
- When the processor is executing at user level, only pages that belong to the user level are addressable.

4.1.2 Type Checking

- At the level of page addressing, two types are defined:

1. Read-only access (R/W=0)
2. Read/write access (R/W=1)

- When the processor is executing at supervisor level, all pages are both readable and writable.
- When the processor is executing at user level:
  - only pages that belong to user level and are marked for read/write access are writable;
  - pages that belong to supervisor level are neither readable nor writable from user level.

4.2 Combining Protection of Both Levels of Page Tables

- For any one page, the protection attributes of its page directory entry may differ from those of its page table entry.
- The 80386 computes the effective protection attributes for a page by examining the protection attributes in both the directory and the page table.
- Table (Next)
- The effective protection provided by the possible combinations of protection attributes

Page Directory Entry		Page Table Entry		Combined Protection	
U/S	R/W	U/S	R/W	U/S	R/W
S-0	R-0	S-0	R-0	S	x
S-0	R-0	S-0	W-1	S	x
S-0	R-0	U-1	R-0	S	x
S-0	R-0	U-1	W-1	S	x
S-0	W-1	S-0	R-0	S	x
S-0	W-1	S-0	W-1	S	x
S-0	W-1	U-1	R-0	S	x
S-0	W-1	U-1	W-1	S	x
U-1	R-0	S-0	R-0	S	x
U-1	R-0	S-0	W-1	S	x
U-1	R-0	U-1	R-0	U	R
U-1	R-0	U-1	W-1	U	R
U-1	W-1	S-0	R-0	S	x
U-1	W-1	S-0	W-1	S	x
U-1	W-1	U-1	R-0	U	R
U-1	W-1	U-1	W-1	U	W

NOTE

S	— Supervisor
R	— Read only
U	— User
W	— Read and Write

x indicates that when the combined U/S attribute is S, the R/W attribute is not checked.

Table 6-5. Combining Directory and Page Protection

4.3 Overrides to Page Protection

- Certain accesses are checked as if they are privilege-level 0 references, even if CPL = 3:

✓ LDT, GDT, TSS, IDT references.

✓ Access to inner stack during ring-crossing CALL/INT.

5. Combining Page and Segment Protection

- When paging is enabled, the 80386 first evaluates segment protection, then evaluates page protection.
- If the processor detects a protection violation at either the segment or the page level, the requested operation cannot proceed; a protection exception occurs instead.

PART II

MULTITASKING

Extra

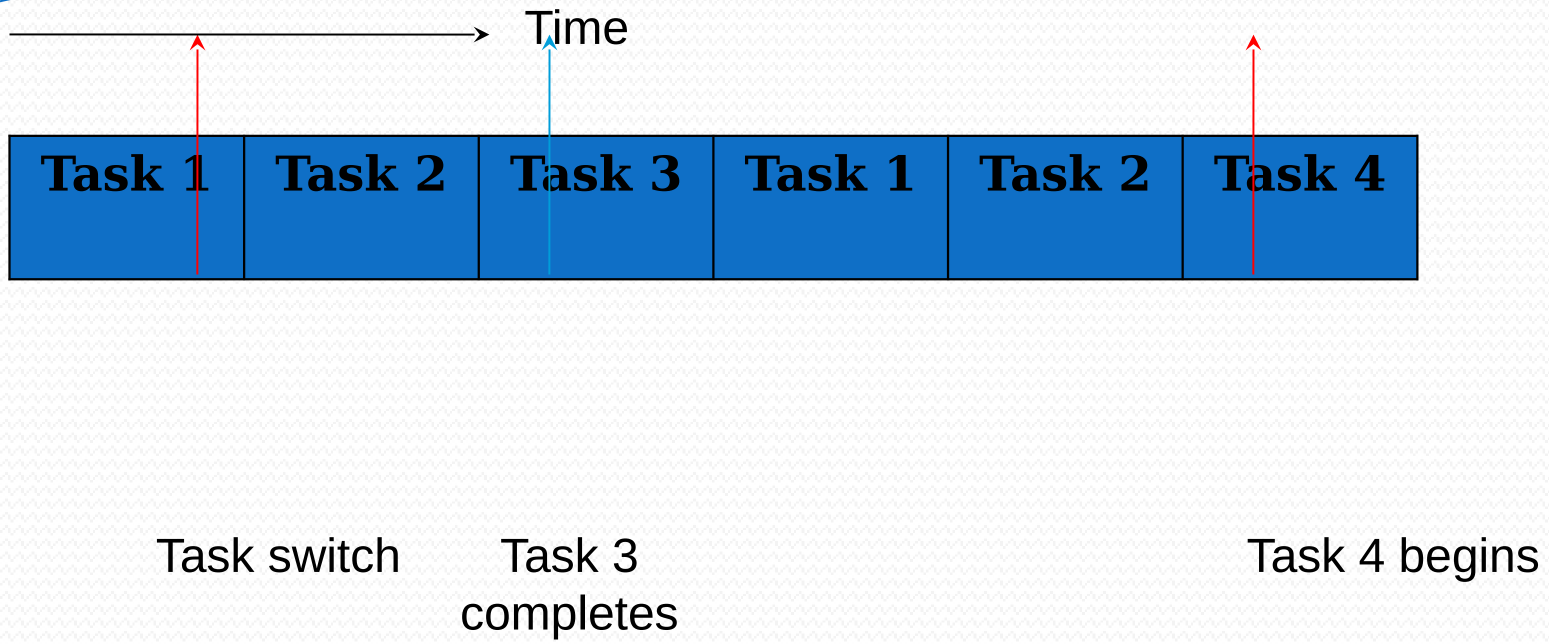


Fig: Running Multiple Tasks Simultaneously

- The x386 processor provides hardware support for multitasking.
- A task is a program which is running, or waiting to run while another program is running.
- A task is invoked by an interrupt, exception, jump, or call.
- When one of these forms of transferring execution is used with a destination specified by an entry in one of the descriptor tables, this descriptor can be a type which causes a new task to begin execution after saving the state of the current task.
- There are two types of task-related descriptors which can occur in a descriptor table: task state segment descriptors and task gates.
- When execution is passed to either kind of descriptor, a task switch occurs

- A task switch is like a procedure call, but it saves more processor state information.
- A task switch transfers execution to a completely new environment, the environment of a task.
- This requires saving the contents of nearly all the processor registers, including the EFLAGS register and the segment registers.
- Unlike procedures, tasks are not re-entrant.
- A task switch does not push anything on the stack.
- The processor state information is saved in a data structure in memory, called a task state segment.

31	15	0	
I/O MAP BASE ADDRESS	0 0 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	SELECTOR FOR TASK'S 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
EIP			20
CR3 (PDBR)			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			C
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	LINK (OLD TSS SELECTOR)		0

ADDRESSES ARE SHOWN IN HEXADECIMAL.
NOTE: BITS MARKED AS 0 ARE RESERVED. DO NOT USE.

APM62

Figure 13-1. 32-Bit Task State Segment

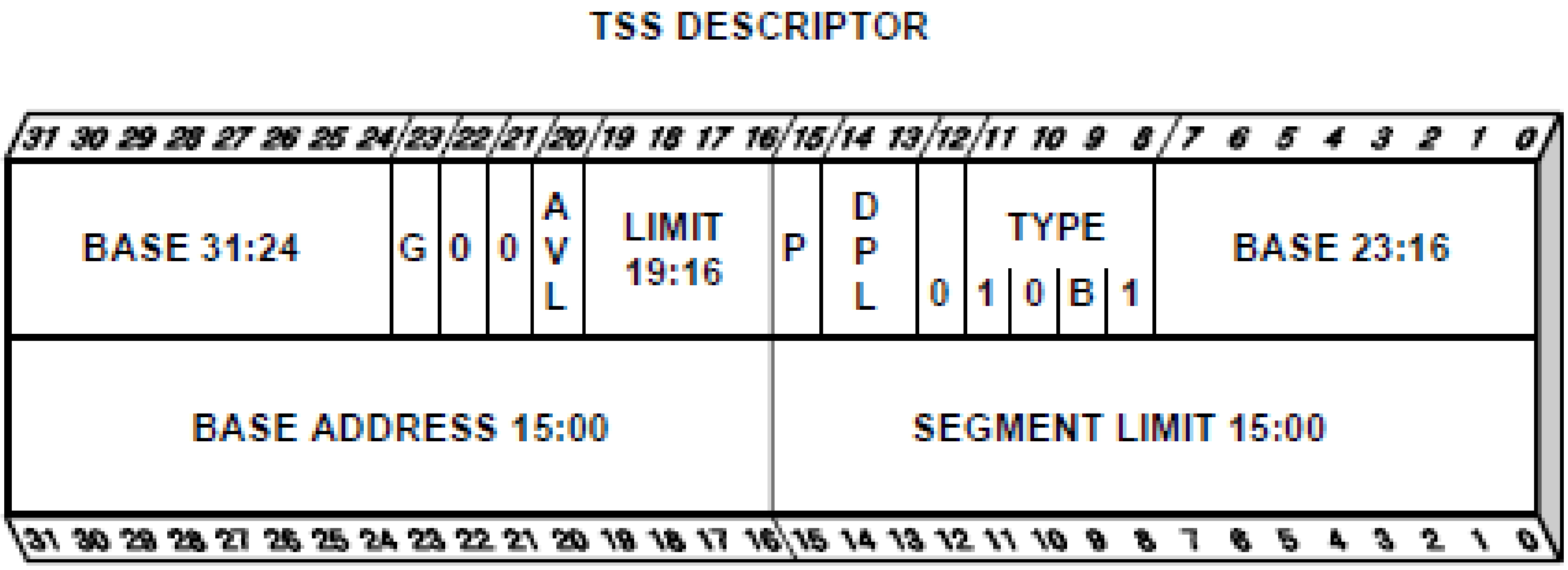
TASK STATE SEGMENT

- The processor state information needed to restore a task is saved in a type of segment, called a task state segment or TSS.
- Figure :format of a TSS for tasks designed for 32- bit CPUs.
- The fields of a TSS are divided into two main categories:

1. Dynamic fields the processor updates with each task switch.
2. Static fields the processor reads, but does not change.

TSS DESCRIPTOR

- The task state segment, like all other segments, is defined by a descriptor.
- The format of a TSS descriptor.
- The Base, Limit, and DPL fields and the Granularity bit and Present bit have functions similar to their use in data-segment descriptors.



AVL	AVAILABLE FOR USE BY SYSTEM SOFTWARE
B	BUSY BIT
BASE	SEGMENT BASE ADDRESS
DPL	DESCRIPTOR PRIVILEGE LEVEL
G	GRANULARITY
LIMIT	SEGMENT LIMIT
P	SEGMENT PRESENT
TYPE	SEGMENT TYPE

APM61

Figure 13-2. TSS Descriptor

- The Busy bit in the Type field indicates whether the task is busy.
- A busy task is currently running or waiting to run.
- A Type field with a value of 9 indicates an inactive task; a value of 11 (decimal) indicates a busy task.
- Tasks are not recursive.
- The processor uses the Busy bit to detect an attempt to call a task whose execution has been interrupted.

Extra Ends

multitasking

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

Multitasking……

- To provide efficient, protected multitasking, the 80386 employs several special data structures.
- It does not use special instructions to control multitasking
- It interprets ordinary control-transfer instructions differently when they refer to the special data structures.
- The registers and data structures that support multitasking are:

1. Task state segment
2. Task state segment descriptor
3. Task register
4. Task gate descriptor

Task MGMT Feature #1

- 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.

Task MGMT Feature #2

- 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.

Task MGMT Feature #3

- 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.

6. Task State Segment

- All the information the processor needs in order to manage a task is stored in a special type of segment, a task state segment (TSS).
- Figure
- Format of a TSS for executing 80386 tasks

Diagram :
80386 32-Bit Task State Segment

31 23	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

TSS Fields

- The fields of a TSS belong to two classes:

1. A dynamic set that the processor updates with each switch from the task.
2. A static set that the processor reads but does not change.

Dynamic Set

- This set includes the fields that store:

1. The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI)
2. The segment registers (ES, CS, SS, DS, FS, GS)
3. The flags register (EFLAGS)
4. The instruction pointer (EIP)
5. The selector of the TSS of the previously executing task (updated only when a return is expected)

Static Set

- This set includes the fields that store:

1. The selector of the task's LDT.
2. The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
3. Pointers to the stacks for privilege levels 0-2.
4. The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs.
5. The I/O map base

Bit map

- The base address for the I/O permission bit map and interrupt redirection bitmap.
- If present, these maps are stored in the TSS at higher addresses.
- The base address points to the beginning of the I/O map and the end of the 32-byte interrupt map.

Task State Segment (TSS)

Two classes of TSS format

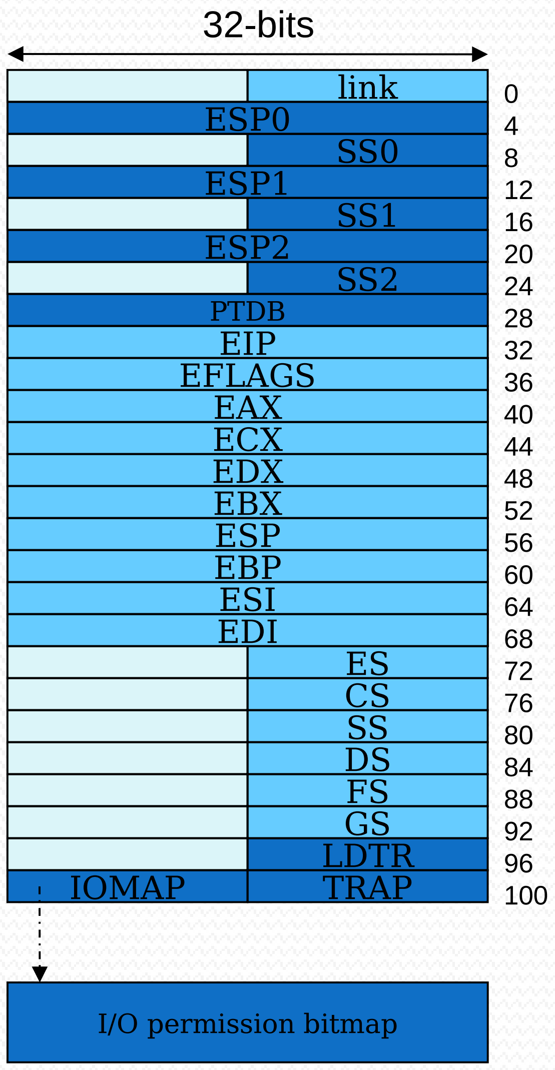
Static set

- **It is that where 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
- The I/O map base

Dynamic set

- **That where 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).

Format of 80386 TSS (32 bit)



🟦	= field is 'static'
🟦	= field is 'Dynamic'
⬜	= field is 'reserved'

Fig. 1 :Task state

- Task state segments may reside anywhere in the linear space.
- The only case that requires caution is :
- when the TSS spans a page boundary and the higher-addressed page is not present.
- In this case, the processor raises an exception if it encounters the not-present page while reading the TSS during a task switch.

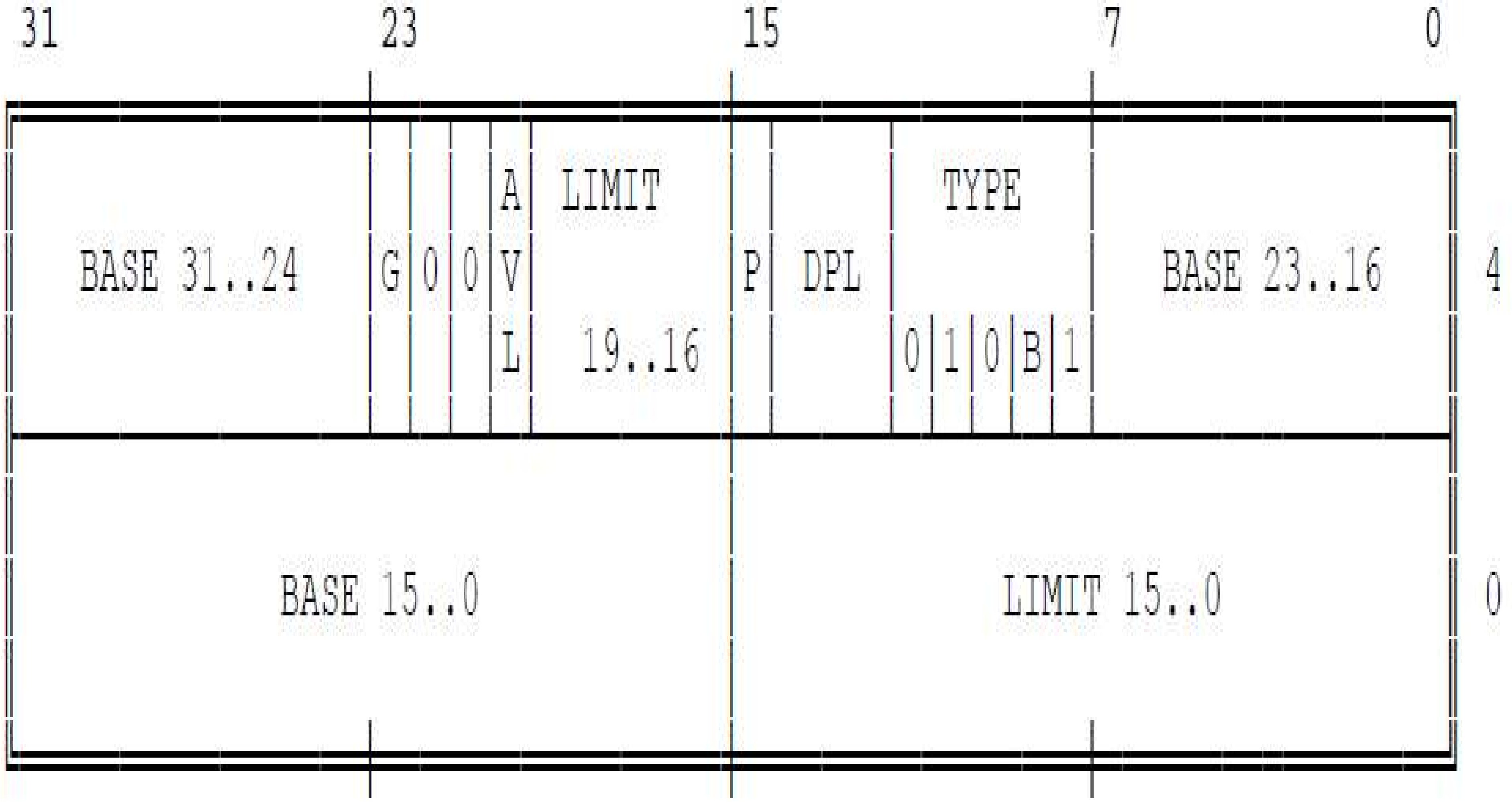
- Such an exception can be avoided by either of two strategies:

1. By allocating the TSS so that it does not cross a page boundary.
2. By ensuring that both pages are either both present or both not-present at the time of a task switch. If both pages are not-present, then the page-fault handler must make both pages present before restarting the instruction that caused the task switch.

7. TSS Descriptor

- The task state segment, like all other segments, is defined by a descriptor.
- Figure (Next) :the format of a TSS descriptor
- 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.

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



Fields of TSS Descriptor

- 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, 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.

8.TR

- TSS descriptors may only be loaded into GDT.

- When multiple TSS descriptors exist in GDT, the TSS currently in use is accessed through the use of the Task Register.

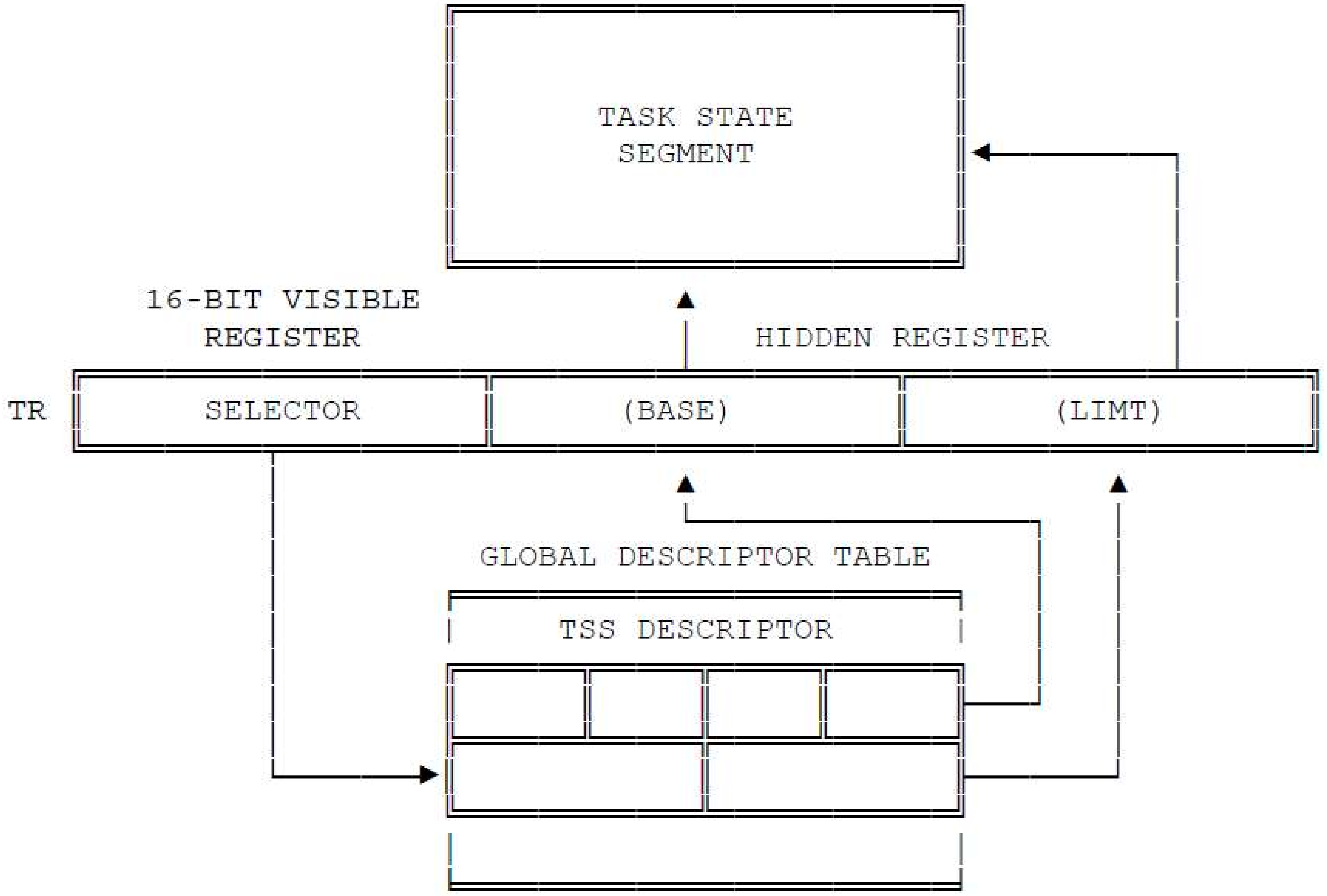
- TR is used as an index pointer into the GDT to locate a TSS descriptor.

TR…..

- The task register (TR) identifies the currently executing task by pointing to the TSS.

- Figure (Next):

  the path by which the processor accesses the current TSS



Task Register

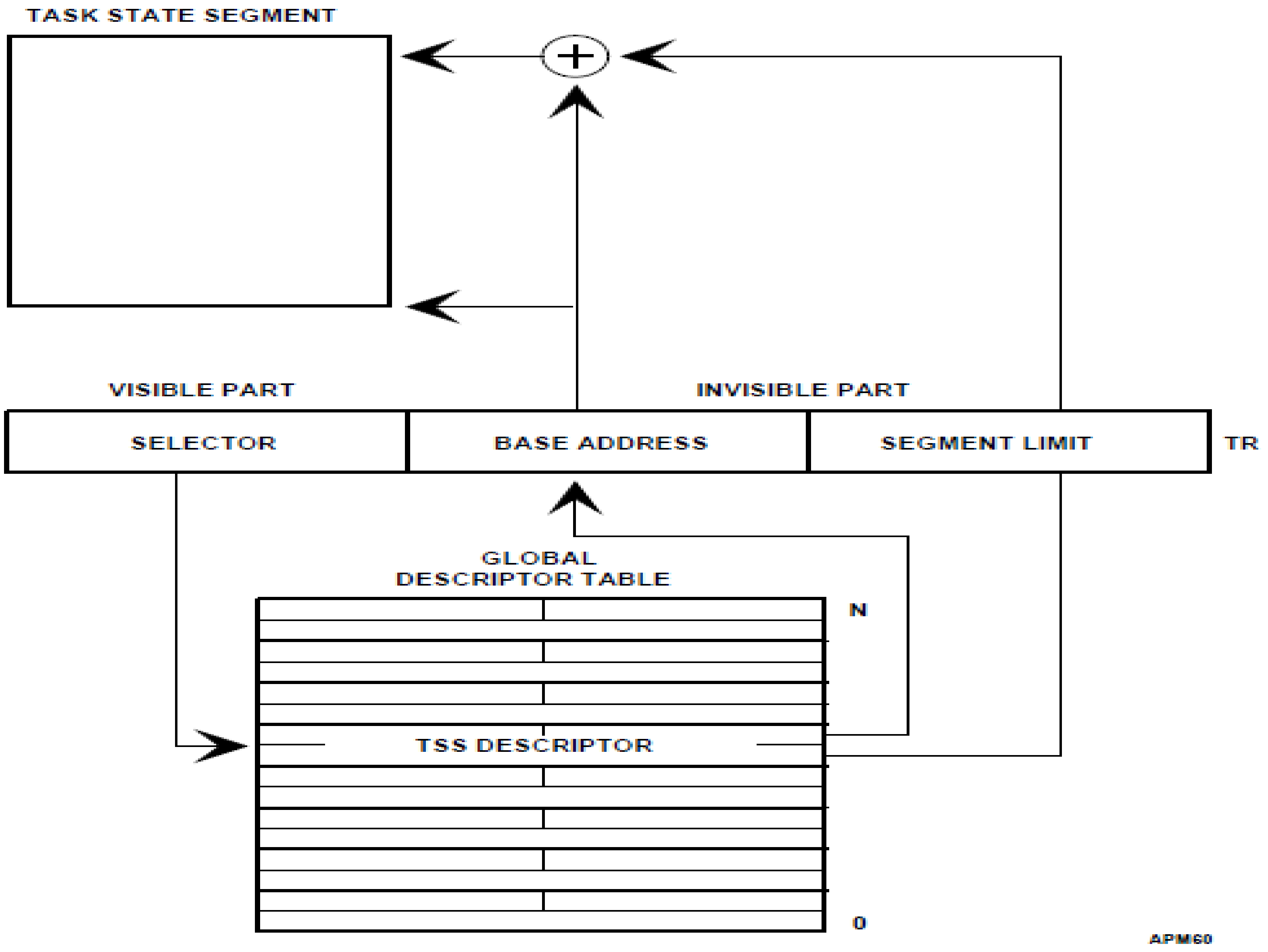


Figure 13-3. Task Register

TASK REGISTER

- The task register (TR) is used to find the current TSS.
- The task register has :
  - a visible part (i.e., a part which can be read and changed by software),
  - an invisible part (i.e., a part maintained by the processor and inaccessible to software).

- The selector in the visible portion indexes to a TSS descriptor in the GDT.
- The processor uses the invisible portion of the TR register to retain the base and limit values from the TSS descriptor.
- Keeping these values in a register makes execution of the task more efficient, because the processor does not need to fetch these values from memory to reference the TSS of the current task.

- The LTR and STR instructions are used to modify and read the visible portion of the task register.
- Both instructions take one operand, a 16-bit segment selector located in memory or 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.
- generally use during system initialization to give an initial value to the task register
- Then, contents of TR are changed by task switch operations.

STR : Store task register

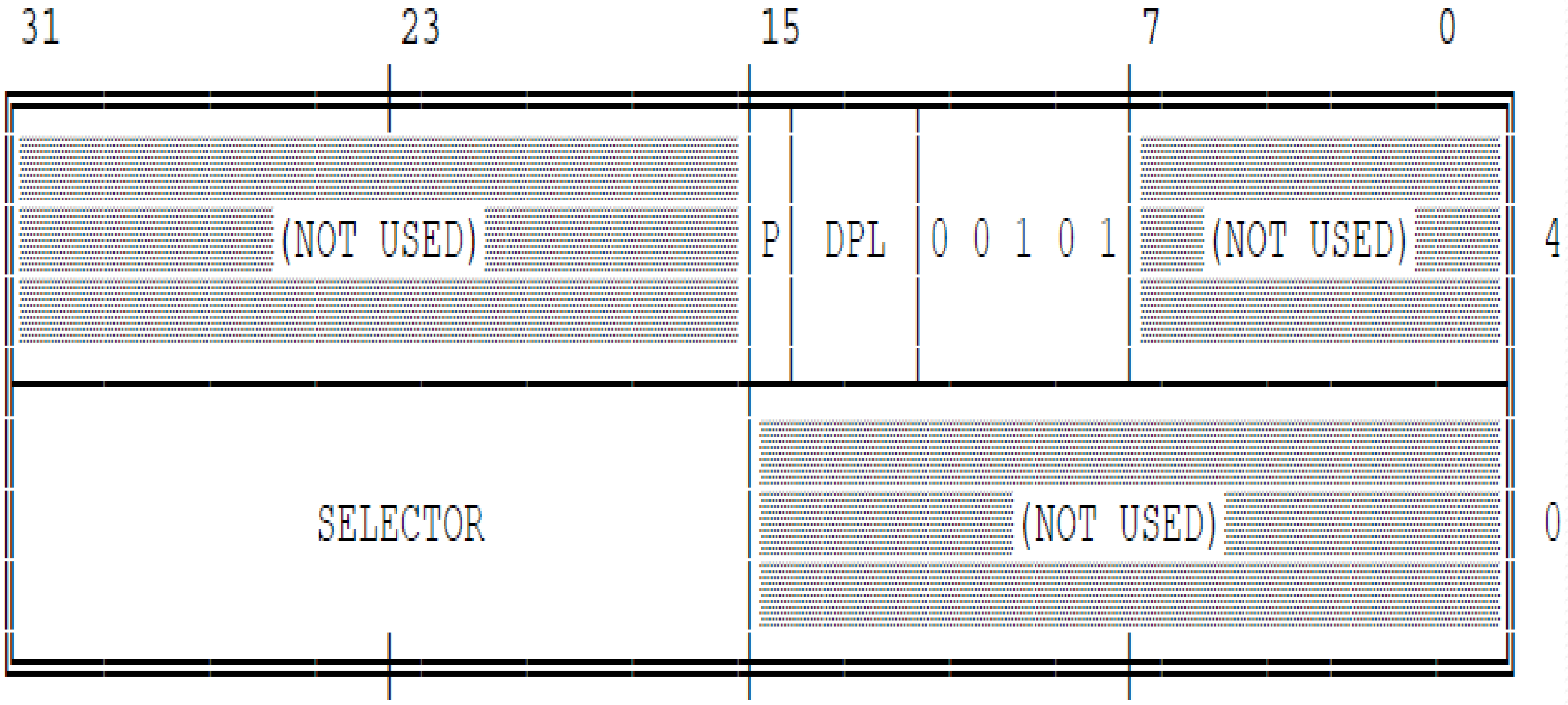
- stores the visible portion of the task register in a general register or memory word.

- STR is not privileged.

9.Task Gate Descriptor

- A task gate descriptor provides an indirect, protected reference to a TSS.
- Figure (Next) :the format of a task gate
- 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.

Figure 7-4. Task Gate 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.
- This constraint prevents untrusted procedures from causing a task switch.
- Note : 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.
2. The need to provide selective access to tasks.
3. The need for an interrupt or exception to cause a task switch.

Need #1

To have a single busy bit for a task:

- 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.

Need #2

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.

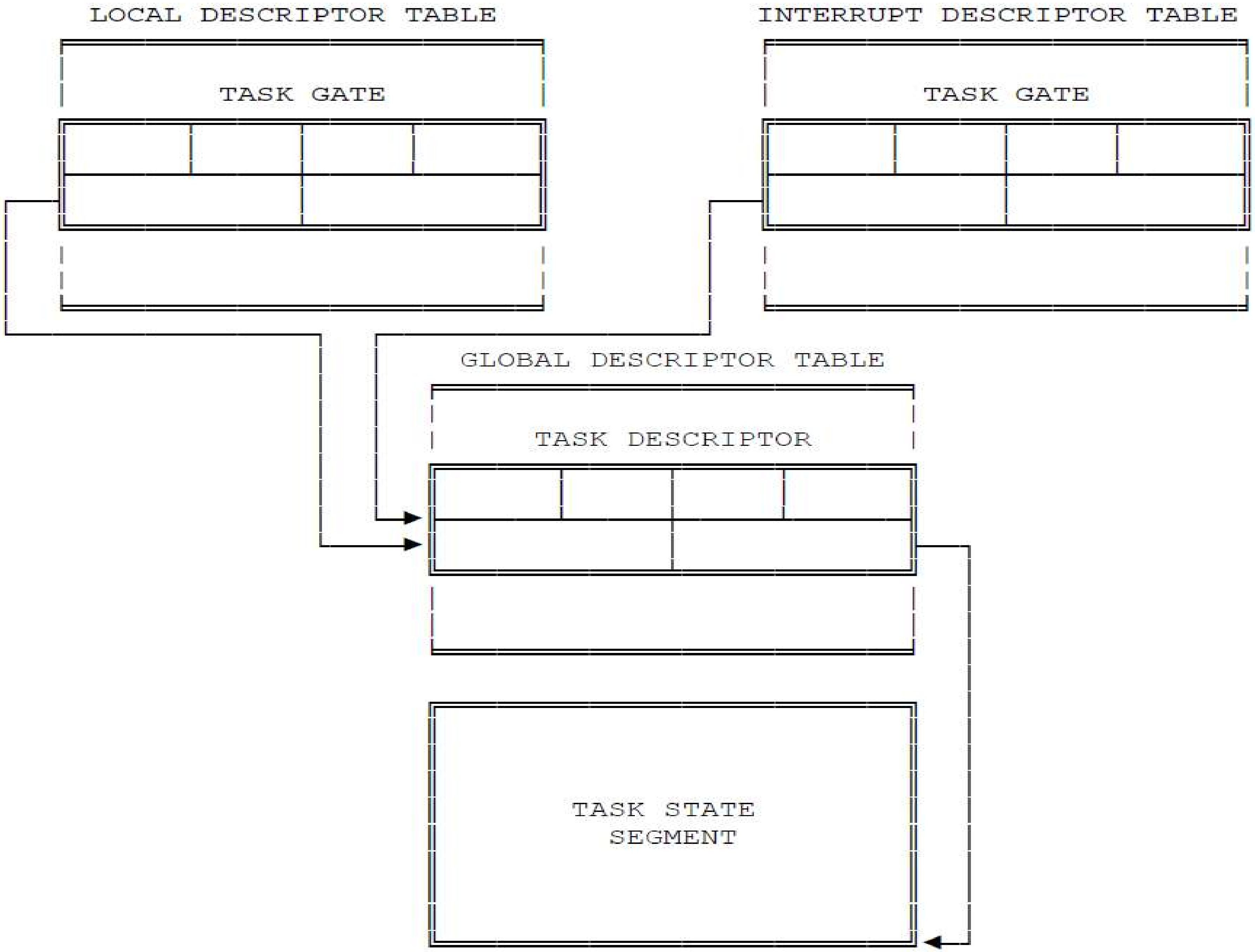
Need #3

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 :
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



10. Task Switching

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

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.

- Whether invoked as a task or as a procedure of the interrupted task, an interrupt handler always returns control to the interrupted procedure in the interrupted task.
- If the NT flag is set, however, the handler is an interrupt task, and the IRET switches back to the interrupted task.

When a task switch is called, the following steps take place:

1. The new TSS descriptor or task gate must have sufficient privilege to allow a task switch.

2. The new TSS descriptor must have its present bit set and have a valid limit field.

3. The state of the current task(also called its context) is saved.

4. The TR is loaded with the selector of the new TSS descriptor.

5. The state of the new task is loaded from its TSS and execution is resumed.

Step 1

- 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 less than or equal to the maximum of CPL and the RPL of the gate selector.
- Exceptions, interrupts, and IRETs are permitted to switch tasks regardless of the DPL of the target task gate or TSS descriptor.
- The DPL, CPL, and RPL values are compared before any further processing takes place.
- Interrupts and exceptions do not force protection checking.

Step 2

- 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.

Step 3

- Saving the state of the current task.
- This involves copying the contents of all processor registers into the TSS for 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.

Step 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, as is the busy bit in the new TSS descriptor.

- The selector is either the operand of a control transfer instruction or is taken from a task gate.

Step 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.

- The privilege level at which the old task was running has no relation to the privilege level of the new task.
- Because the tasks are isolated by their separate address spaces and task state segments, and because privilege rules control access to a TSS, no privilege checks are needed to perform a task switch.
- The new task begins executing at the privilege level indicated by the RPL of the new contents of the CS register, which are loaded from the TSS.

Note:

- The state of the outgoing task is always saved when a task switch occurs.
- If execution of that task is resumed, it starts after the instruction that caused the task switch.
- The registers are restored to the values they held when the task stopped executing.

11. Task Linking

- The back-link field of the TSS and the NT (nested task) bit of the flag word together allow the 80386 to automatically return to a task that CALLed another task or was interrupted by another task.

- When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task's TSS and, at the same time, sets the NT bit in the new task's flag register.

- The NT flag indicates whether the back-link field is valid.
- The new task releases control by executing an IRET instruction.
- When interpreting an IRET, the 80386 examines the NT flag.
- If NT is set, the 80386 switches back to the task selected by the back-link field.
- Table (Next) summarizes the uses of these fields.

Table 7-2. Effect of Task Switch on BUSY, NT, and Back-Link

Affected Field	Effect of JMP Instruction	Effect of CALL Instruction	Effect of IRET Instruction
Busy bit of incoming task	Set, must be 0 before	Set, must be 0 before	Unchanged, must be set
Busy bit of outgoing task	Cleared	Unchanged (already set)	Cleared
NT bit of incoming task	Cleared	Set	Unchanged
NT bit of outgoing task	Unchanged	Unchanged	Cleared
Back-link of incoming task	Unchanged	Set to outgoing TSS selector	Unchanged
Back-link of outgoing task	Unchanged	Unchanged	Unchanged

Busy Bit Prevents Loops

- The Busy bit of the TSS descriptor prevents re-entrant task switching.
- There is only one saved task context, the context saved in the TSS, therefore a task only may be called once before it terminates.
- The chain of suspended tasks may grow to any length, due to multiple interrupts, exceptions, jumps, and calls.
- The Busy bit prevents a task from being called if it is in this chain.
- A re-entrant task switch would overwrite the old TSS for the task, which would break the chain.

- The processor manages the Busy bit as follows:

1. When switching to a task, the processor sets the Busy bit of the new task.

2. When switching from a task, the processor clears the Busy bit of the old task if that task is not to be placed in the chain (i.e., the instruction causing the task switch is a JMP or IRET instruction). If the task is placed in the chain, its Busy bit remains set.

3. When switching to a task, the processor generates a general-protection exception if the Busy bit of the new task already is set.

- In this way, the processor prevents a task from switching to itself or to any task in the chain, which prevents re-entrant task switching.

- The busy bit is effective even in multiprocessor configurations, because the processor automatically asserts a bus lock when it sets or clears the busy bit.
- This action ensures that two processors do not invoke the same task at the same time.

Modifying Task Linkages

- Any modification of the linkage order of tasks should be accomplished only by software that can be trusted to correctly update the back-link and the busy-bit.
- Such changes may be needed to resume an interrupted task before the task that interrupted it.
- Trusted software that removes a task from the back-link chain must follow one of the following policies:

1. First change the back-link field in the TSS of the interrupting task, then clear the busy-bit in the TSS descriptor of the task removed from the list.
2. Ensure that no interrupts occur between updating the back-link chain and the busy bit.

12. Task Addressing 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.

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

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 OS also implements page-level virtual memory.

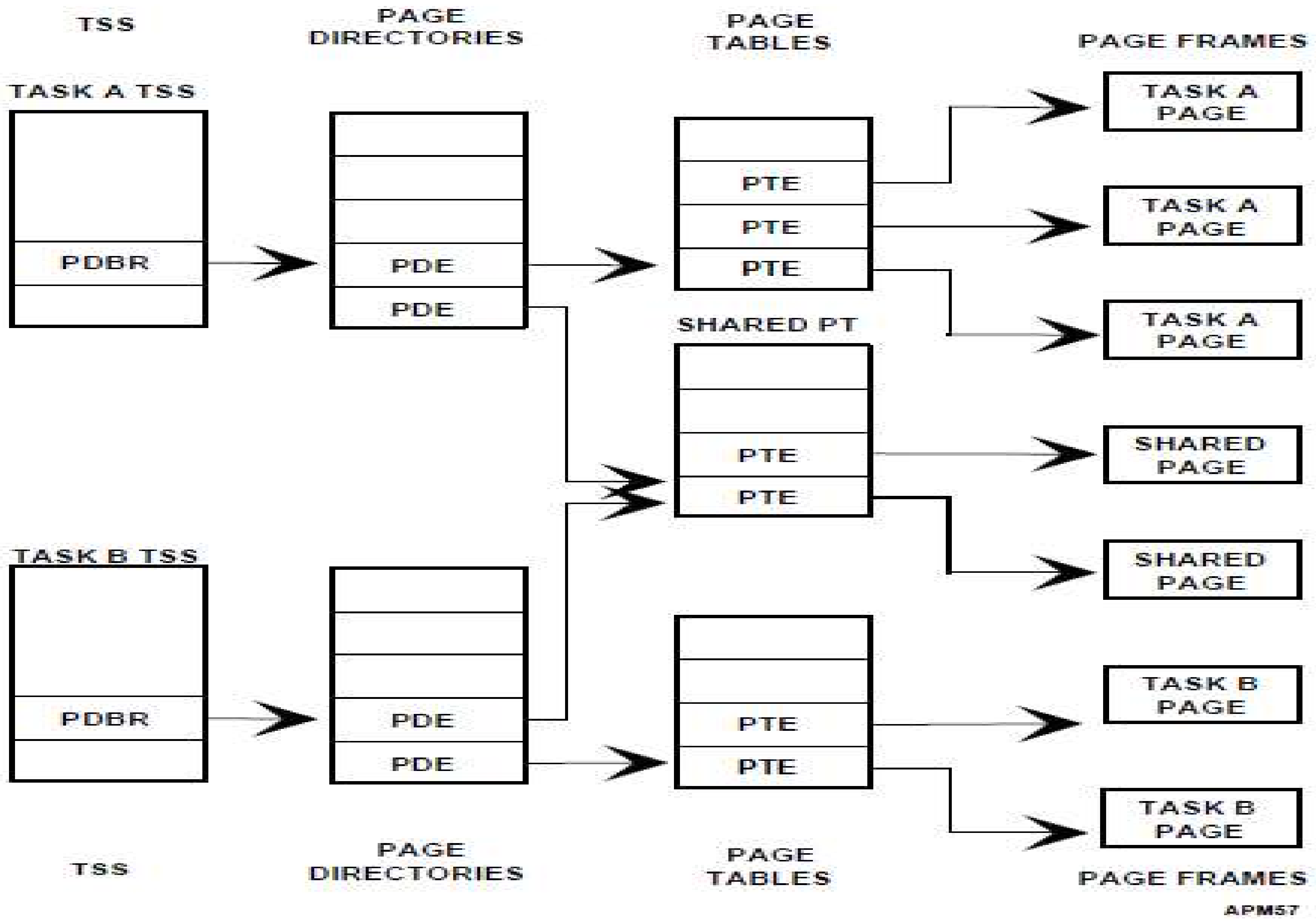


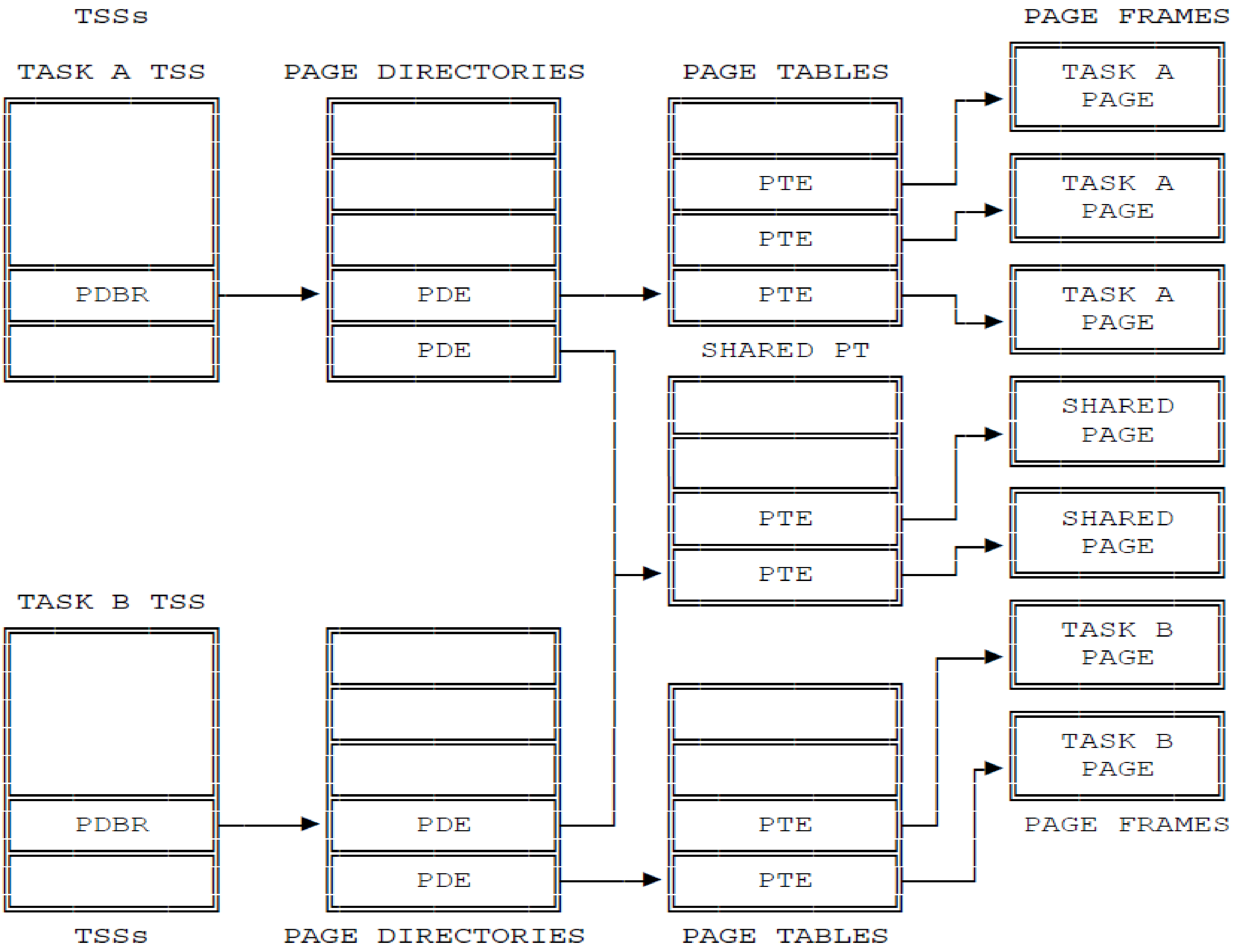
Figure 13-7. Overlapping Linear-to-Physical Mappings

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.

Figure 7-6. Partially-Overlapping Linear Spaces



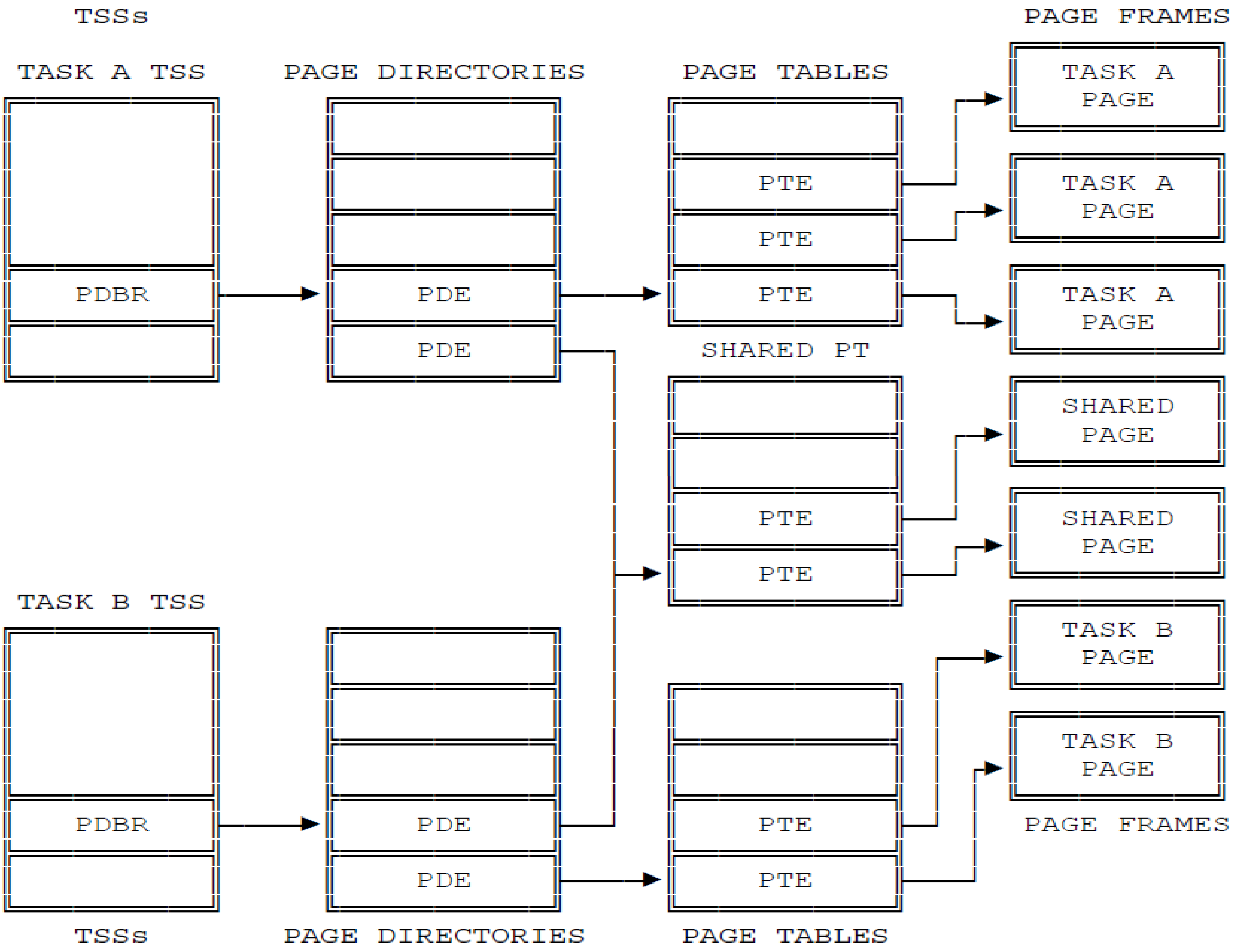
Theoretically,

- The linear address spaces of different tasks may map to
- completely distinct physical addresses.
- If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

Practically,

- Some portion of the linear address spaces of all tasks must map to the same physical addresses.
- The TSSs must lie in a common space so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch.
- The linear space mapped by the GDT should also be mapped to a common physical space;
- otherwise, the purpose of the GDT is defeated.
- Figure :how the linear spaces of two tasks can overlap in the physical space by sharing page tables.

Figure 7-6. Partially-Overlapping Linear Spaces



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.

Logical-to- Physical Address Space Mapping

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

1. Via the GDT.
2. By sharing LDTs.
3. By descriptor aliases in LDTs.