Memory Management

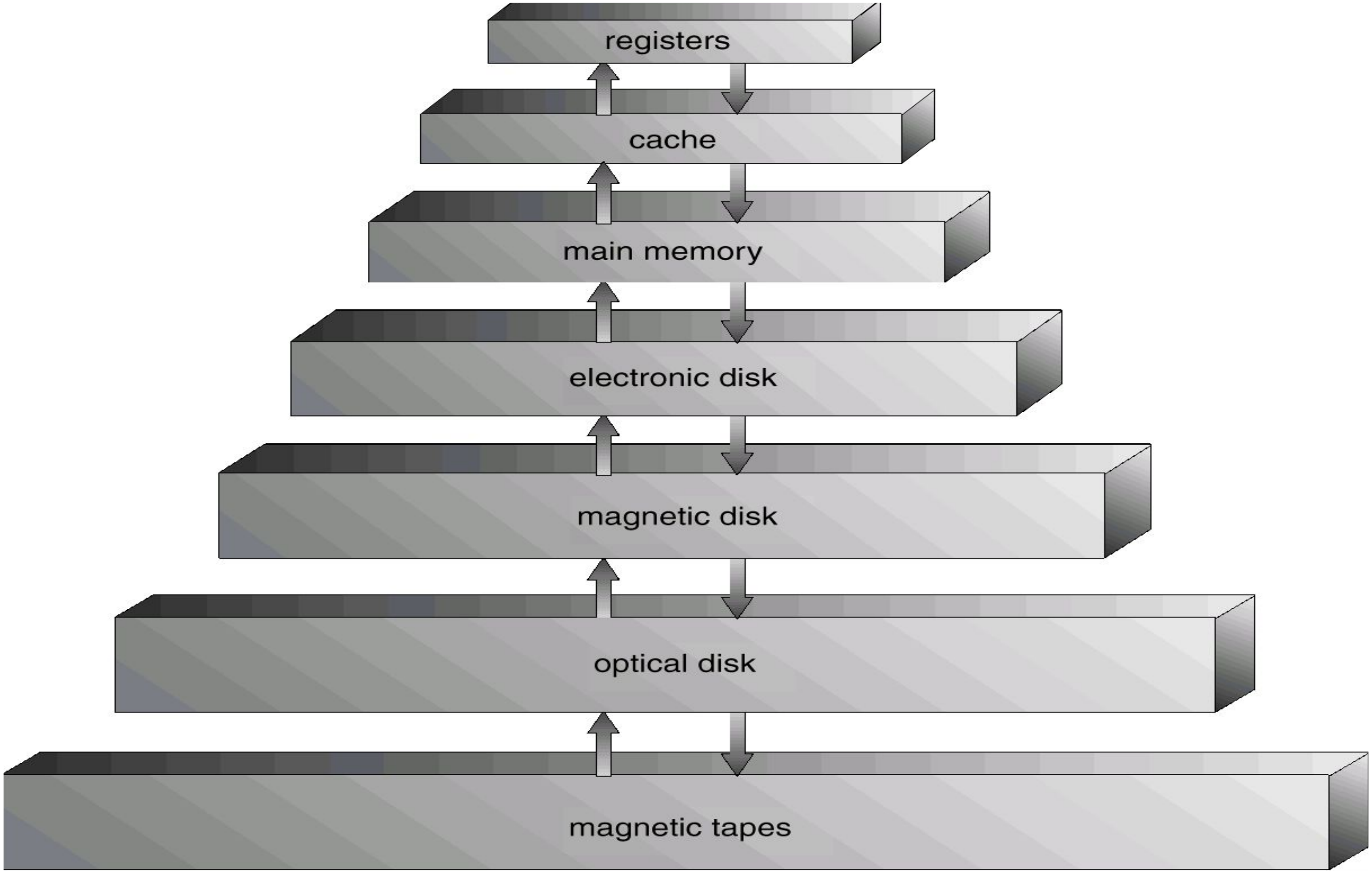
Memory

- CPU can only access content from main memory and registers (built into processor)
- Data / instruction can not be read from secondary storage by processor . It needs to be moved from secondary storage to main memory.

Memory Management

- Three design constraints of memory subsystem
  - Size
  - Speed
  - Cost
- Across the spectrum of the technologies following relationship holds
  - Smaller access time , greater cost per bit
  - Greater capacity, smaller cost per bit
  - Greater capacity , greater access time
- Memory subsystem requirement
  - Large capacity, fast access time and low cost

- To meet the contradictory design requirement , organize memory in hierarchical manner



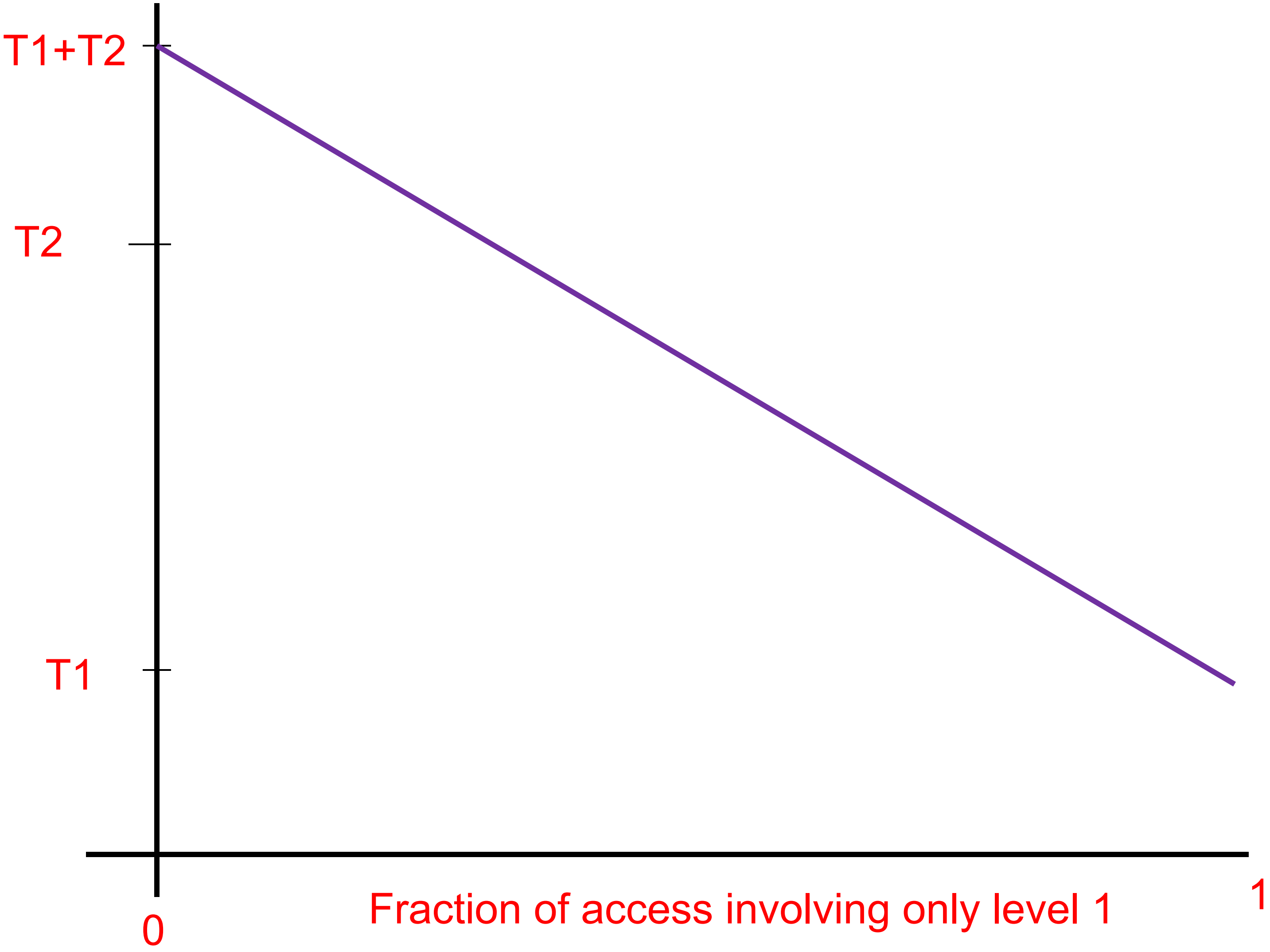
- As one goes down the Hierarchy , the following conditions occur:

1. Decreasing cost per bit
2. Increasing capacity
3. Increasing access time
4. Decreasing frequency of access of the memory .

Example (two level memory)

- Processor has access to two level of memory
  - Level 1 contains 1000 words and access time(T1) is 0.1 Micro second
  - Level 2 contains 100,000 words and access time (T2) is 1 Micro second

- If the word is found in level 1
  - then it is accessed in 0.1 Micro sec
  - else 1.1 micro sec


T1+T2
T2
T1
0
Fraction of access involving only level 1
1

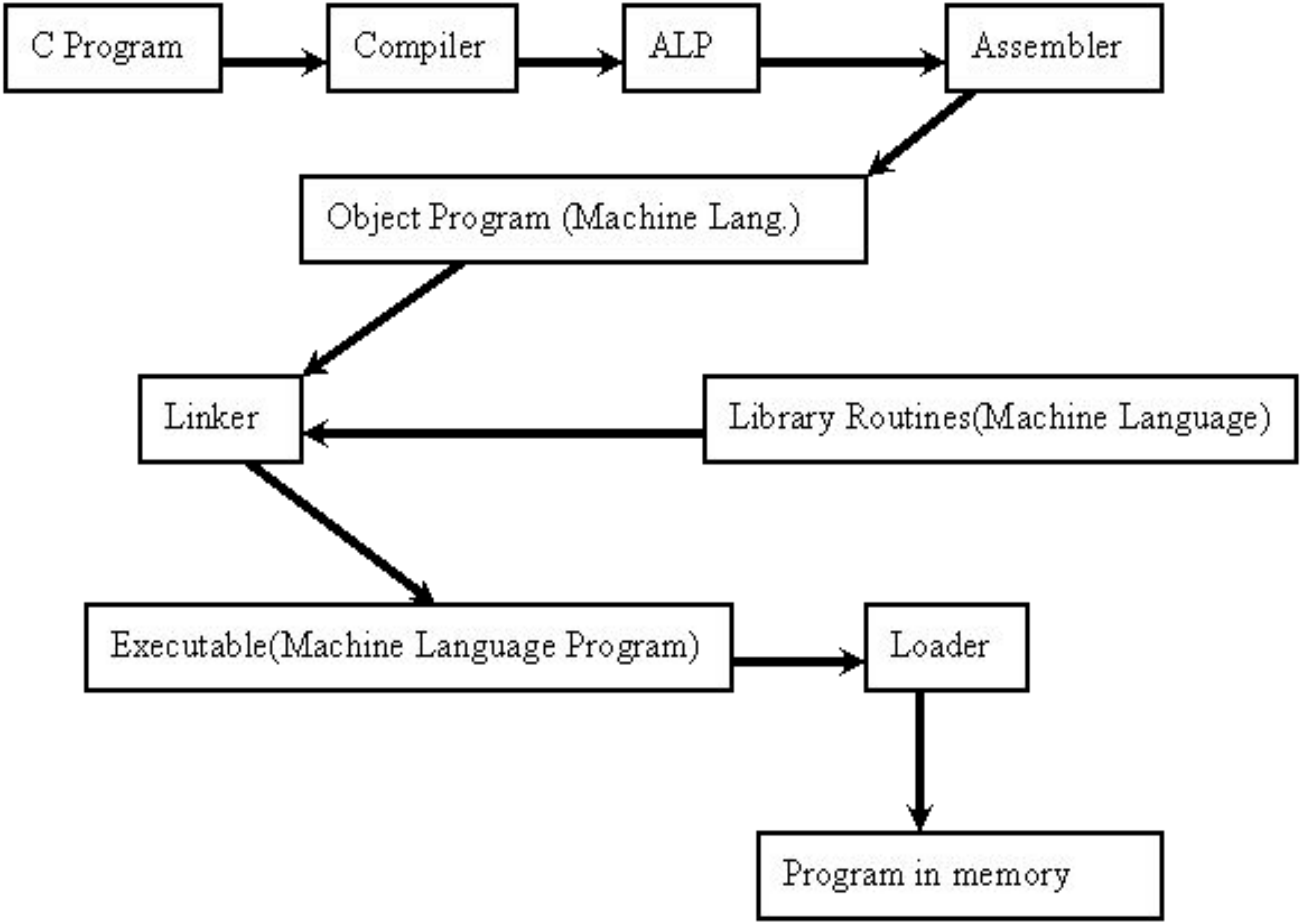
Observations

- We observe that if memory access at level 2 is less frequent then overall access time is close to level 1 access time

- The basis for validity of this condition is a principal known as locality of reference
- During course of execution of program , memory references for both data and instruction tends to cluster

Memory Management Requirement

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization


C Program
Compiler
ALP
Assembler
Object Program (Machine Lang.)
Linker
Library Routines(Machine Language)
Executable(Machine Language Program)
Loader
Program in memory

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages.
  - **Compile time**: If memory location known a priori, (starting location) absolute code can be generated; must recompile code if starting location changes. Example MS DOS .COM format programs.
  - **Load time**: Must generate *relocatable* code if memory location is not known at compile time. Final binding is delayed until load time.
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*). Most general purpose operating systems use this method.

Loading Program into main memory

- it is assumed that OS occupies some fixed portion of memory and rest is available to user processes.
- Based on requirement and functionality different memory management methods are adopted.

  - Should the process be allocated memory in contiguous manner?
  - How the User program / data area is managed?
    - Static/dynamic partitioning
    - Equal /unequal partition

Fixed Partitioning

- Main memory is divided into number of fixed size partition at system generation time.
  - A processes can be loaded into a partition of equal or greater size

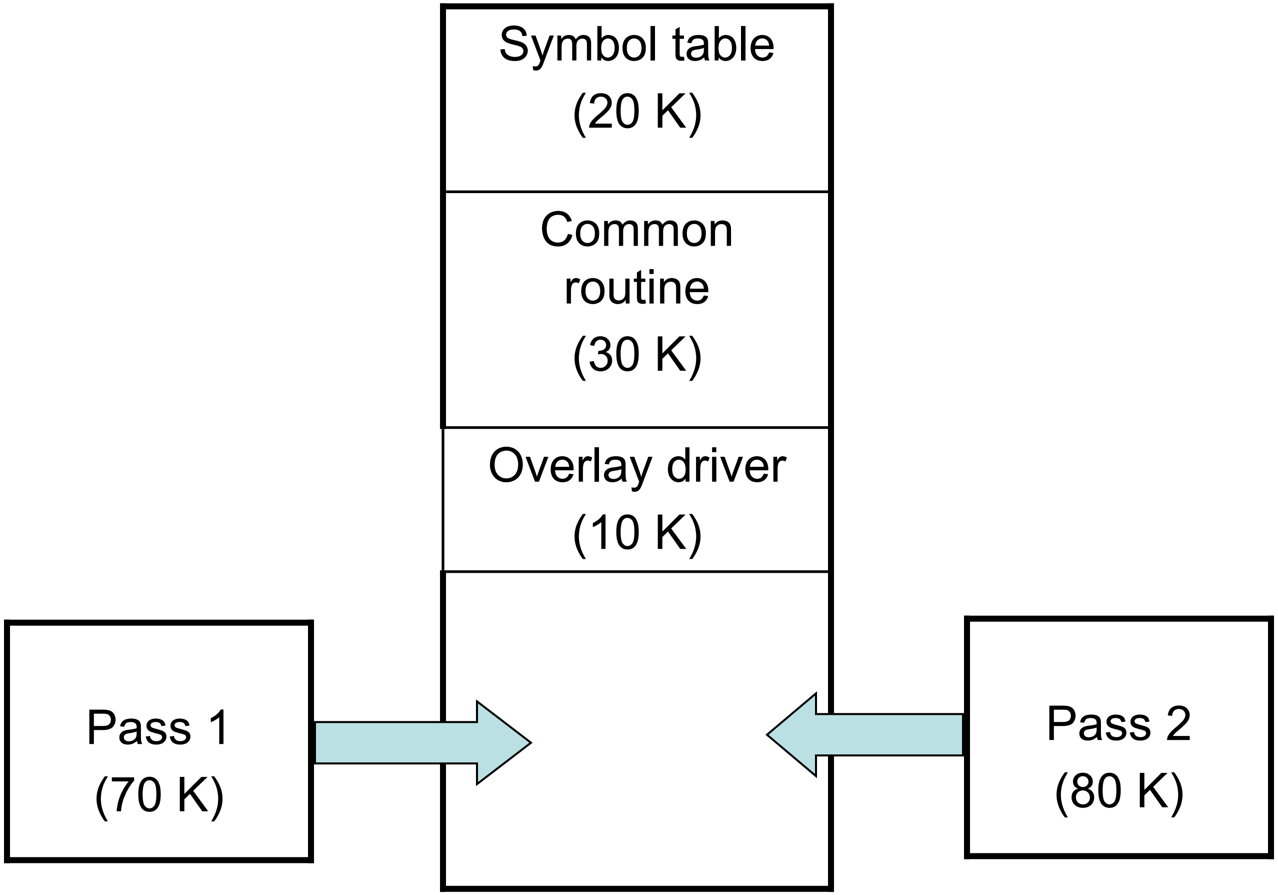
Equal size fixed Partition

- Easy to implement
- If total user memory space is X and size of partition is Y, (Y< X) then number of partitions in system will be X / Y. This is the maximum number of processes that can be loaded in the memory at any given time
- If program size is much smaller than the size of partition, the remaining space is unutilized
- A program may be too big to fit into a partition

Overlays

- Needed when process is larger than amount of memory allocated to it.
- Keep in memory only those instructions and data that are needed at any given time
- Implemented by user, no special support from operating system, programming of overlay structure is complex

Assembler example -Overlay cont..



Symbol table

(20 K)

Common routine

(30 K)

Overlay driver

(10 K)

Pass 1

(70 K)

Pass 2

(80 K)

Unequal size partition

- We create fixed number of unequal size partition
- Program is loaded into best fit partition
  - processes are assigned in such a way as to minimize wasted memory within a partition
- queue for each partition

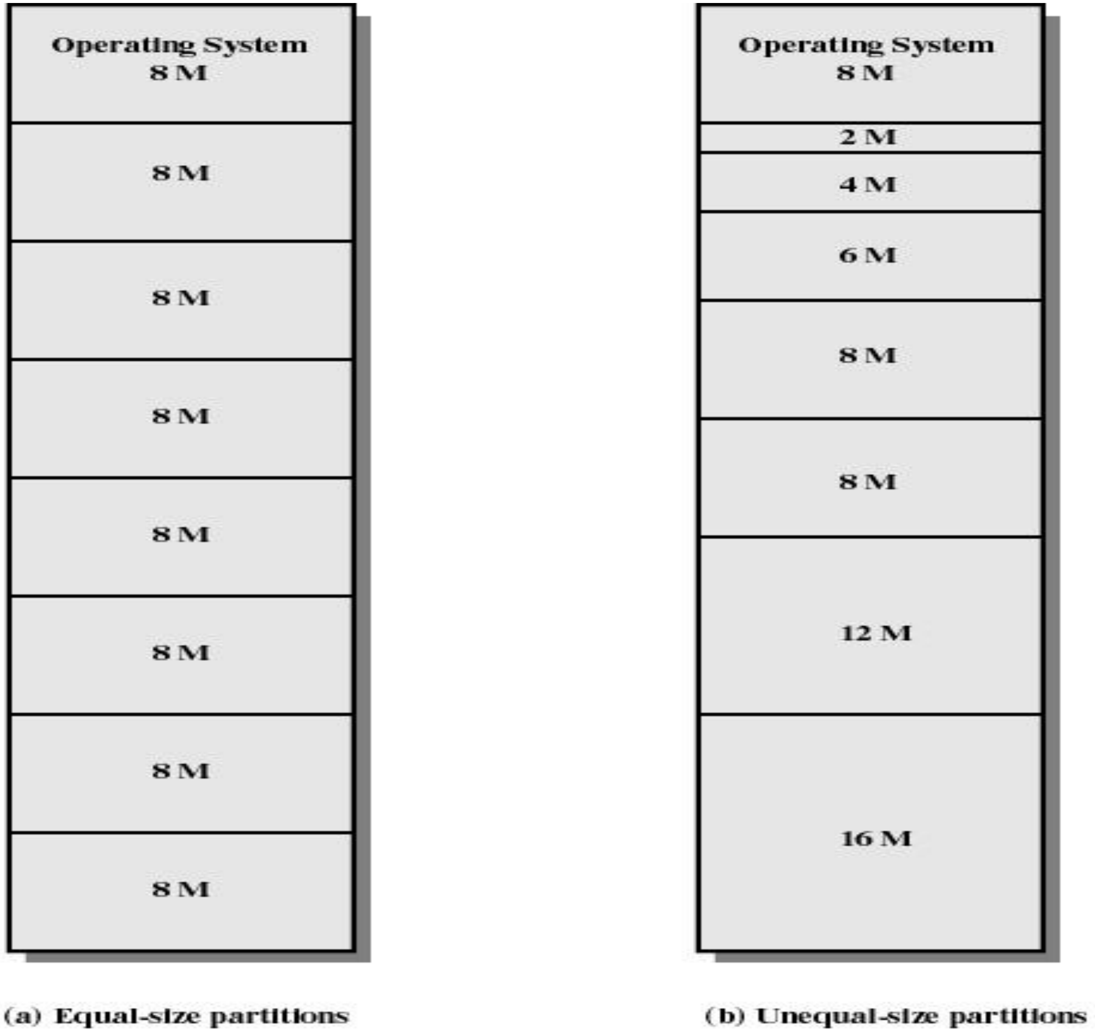
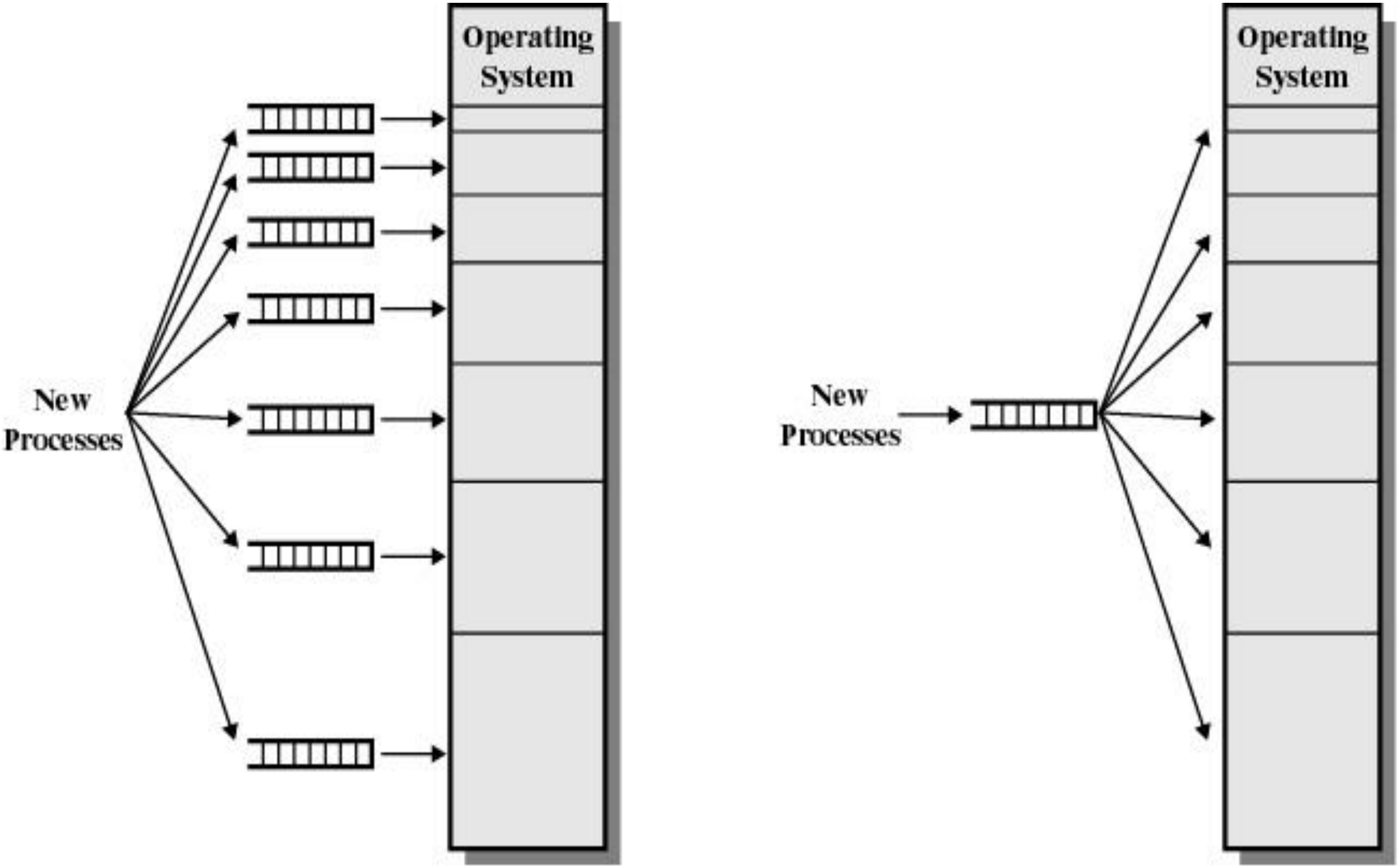


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory



(a) One process queue per partition

(b) Single process queue

Figure 7.3 Memory Assignment for Fixed Partitioning

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually we get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

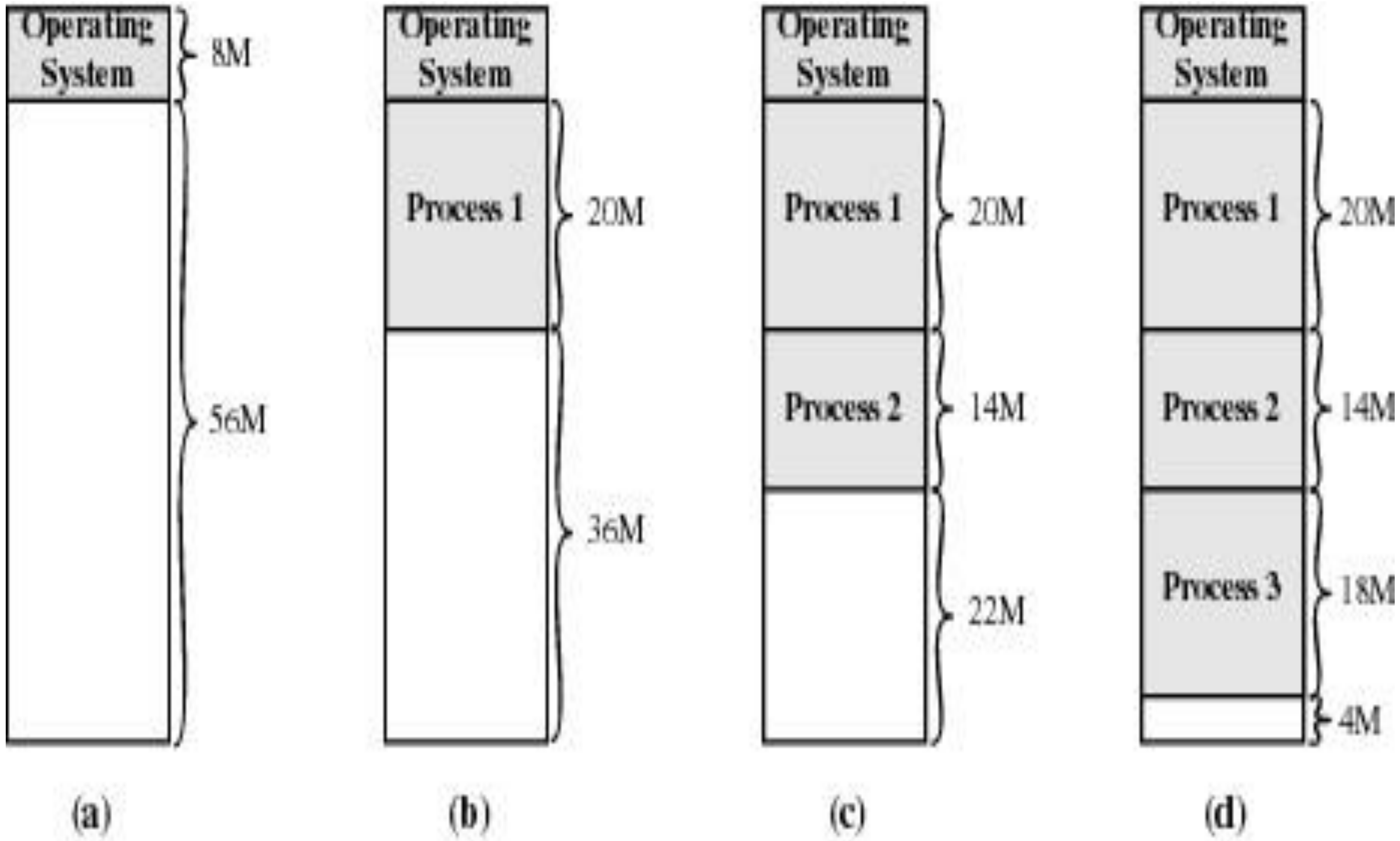


Figure 7.4 The Effect of Dynamic Partitioning

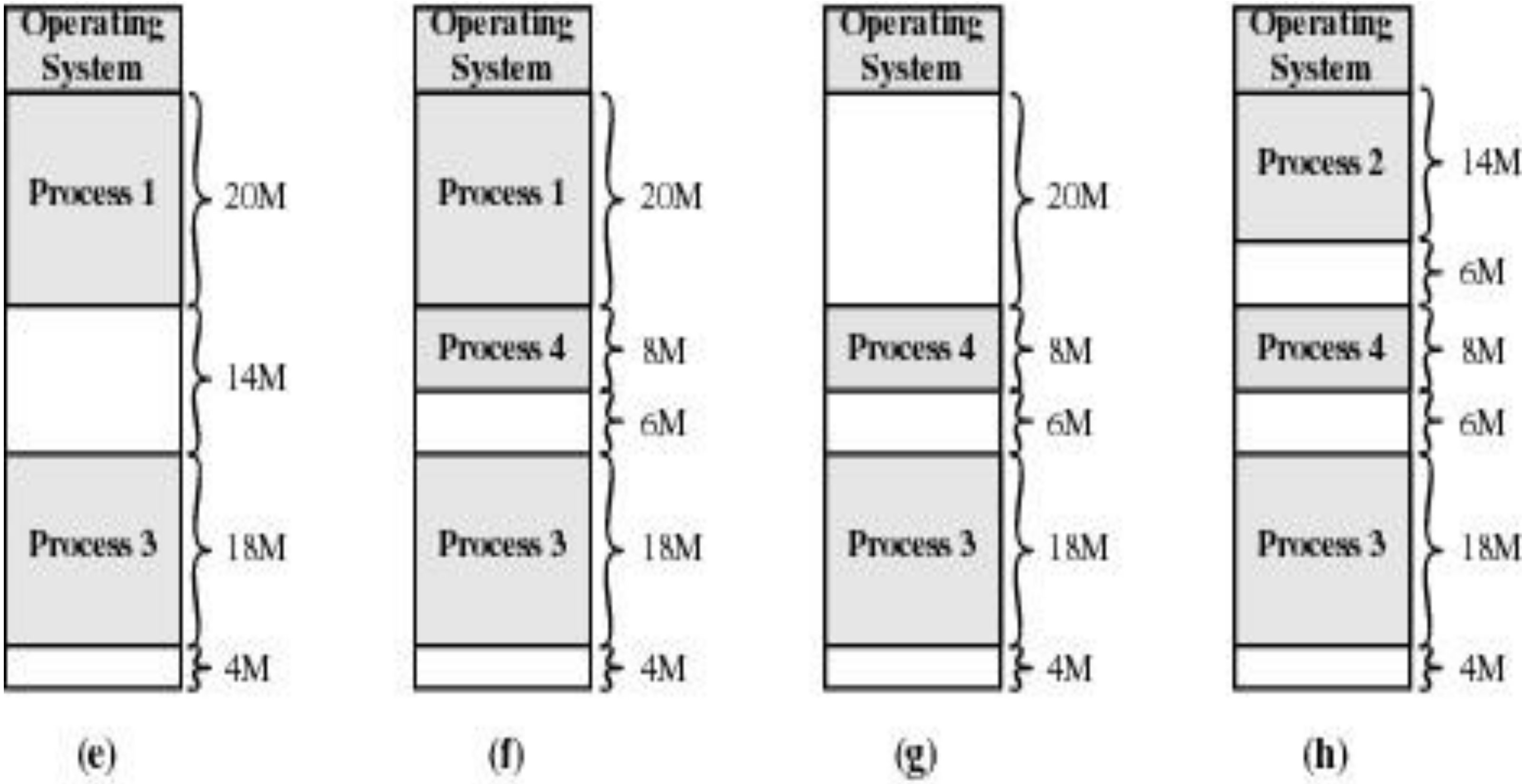


Figure 7.4 The Effect of Dynamic Partitioning

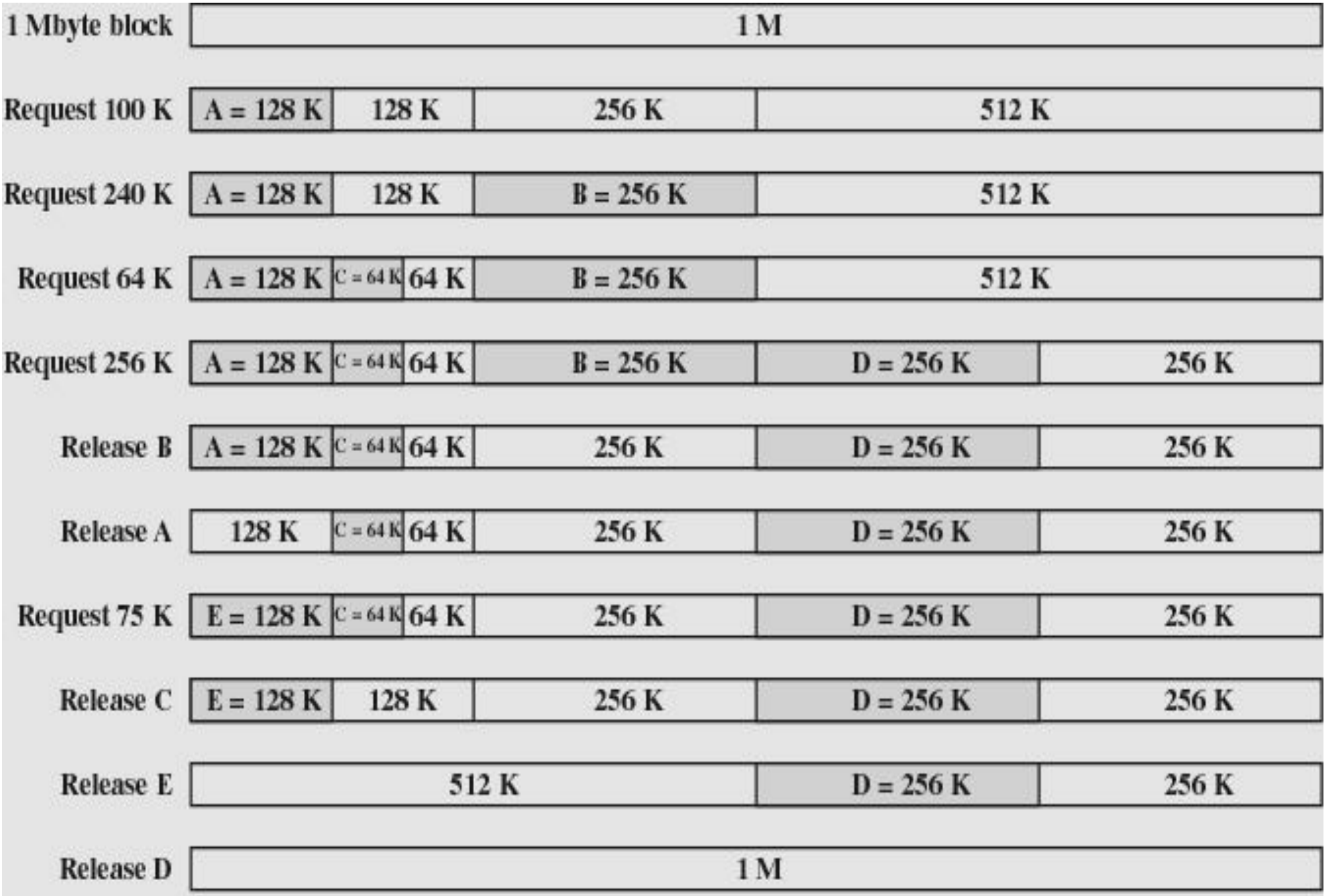
How to satisfy a request of size *n* from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough
  - – Fastest
  - – May have many process loaded in the front end of memory and it must be searched over when trying to find a free block

- **Next-fit**
  - It begins to scan memory from the location of the last placement and chooses the next available block
- **Best-fit :** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
- Compaction
  - Compaction is required to obtain a large block at the end of memory
  - Since smallest block is found for process, the smallest amount of fragmentation is left memory compaction must be done more often

Buddy System

- In Buddy system memory blocks are available of size 2^K where $L <= K <= U$
- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s <= 2^U$, entire block is allocated
  - Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to S is generated
- Buddy system is compromise to overcome the shortcoming of fixed and variable partitioning scheme



1 Mbyte block | 1 M

Request 100 K | A = 128 K | 128 K | 256 K | 512 K

Request 240 K | A = 128 K | 128 K | B = 256 K | 512 K

Request 64 K | A = 128 K | C = 64 K | 64 K | B = 256 K | 512 K

Request 256 K | A = 128 K | C = 64 K | 64 K | B = 256 K | D = 256 K | 256 K

Release B | A = 128 K | C = 64 K | 64 K | 256 K | D = 256 K | 256 K

Release A | 128 K | C = 64 K | 64 K | 256 K | D = 256 K | 256 K

Request 75 K | E = 128 K | C = 64 K | 64 K | 256 K | D = 256 K | 256 K

Release C | E = 128 K | 128 K | 256 K | D = 256 K | 256 K

Release E | 512 K | D = 256 K | 256 K

Release D | 1 M

Figure 7.6 Example of Buddy System

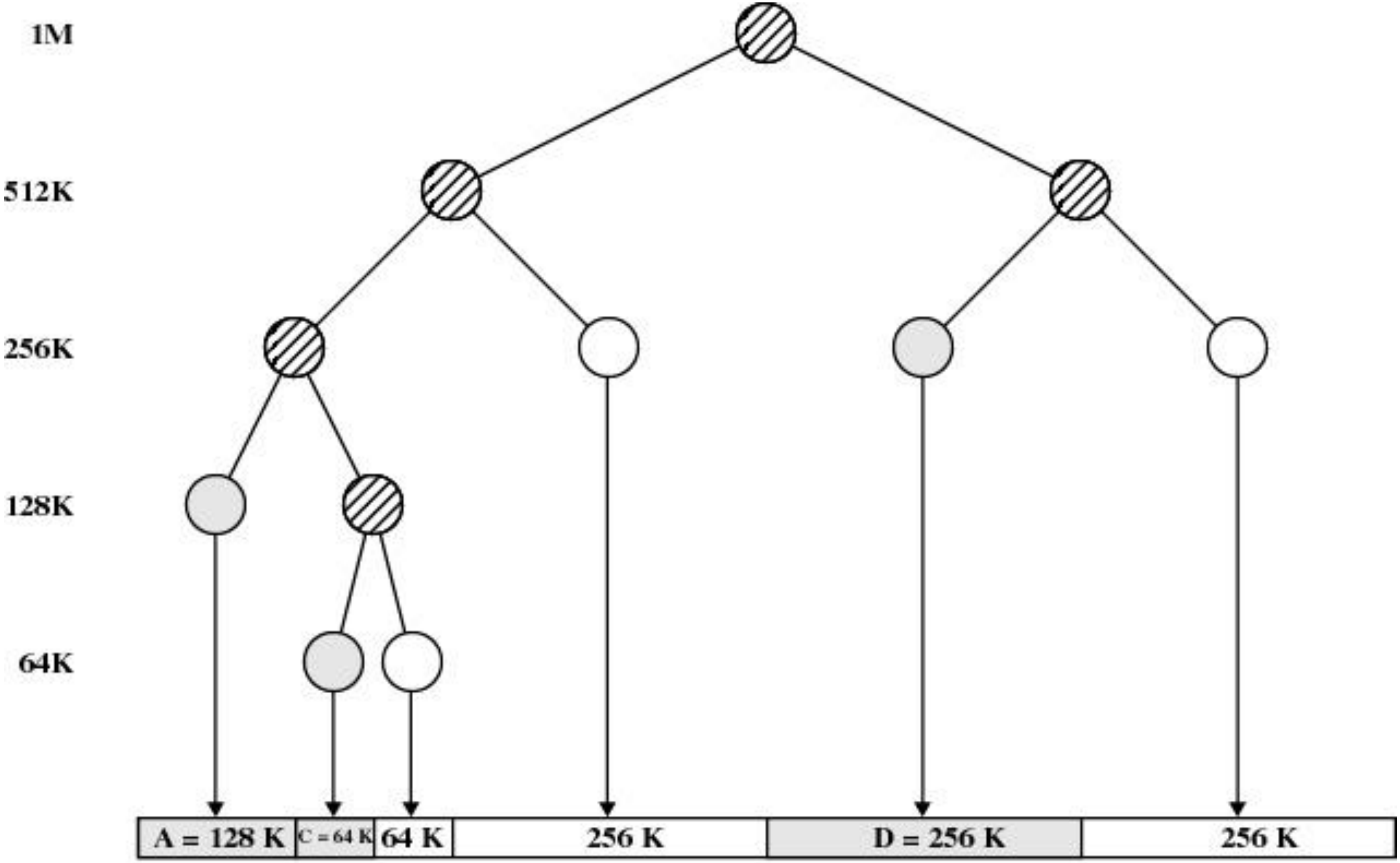


Figure 7.7 Tree Representation of Buddy System

Fragmentation

- External fragmentation –memory space exists to satisfy a request, but it is not contiguous.
- Internal fragmentation – allocated memory may be slightly larger than requested memory; The additional space is unused
  - 50 % rule. (N allocated block & 0.5N waste)
- Reduce external fragmentation by compaction
  - Compaction is possible ***only*** if relocation is dynamic, and is done at execution time.
  - I/O problem
    - Latch job in memory while it is involved in I/O.
    - Do I/O only into OS buffers

Assumptions

- Allocate total required amount of memory
- Allocate memory in contiguous manner

Observations

- We realize that loading entire program in memory is wasteful as all the functionality of a program is not used simultaneously.
- Can We load on Demand ?

- In this case when the additional code is brought in memory the memory may not be available in contiguous manner
- Relax the assumption of contiguous memory allocation

- With Load on demand and Discontiguous allocation, we require
  - logical address to physical address mapping
  - Memory can be allocated in fixed size chunks
  - Or it can be allocated in variable size chunks

Memory allocation Problems

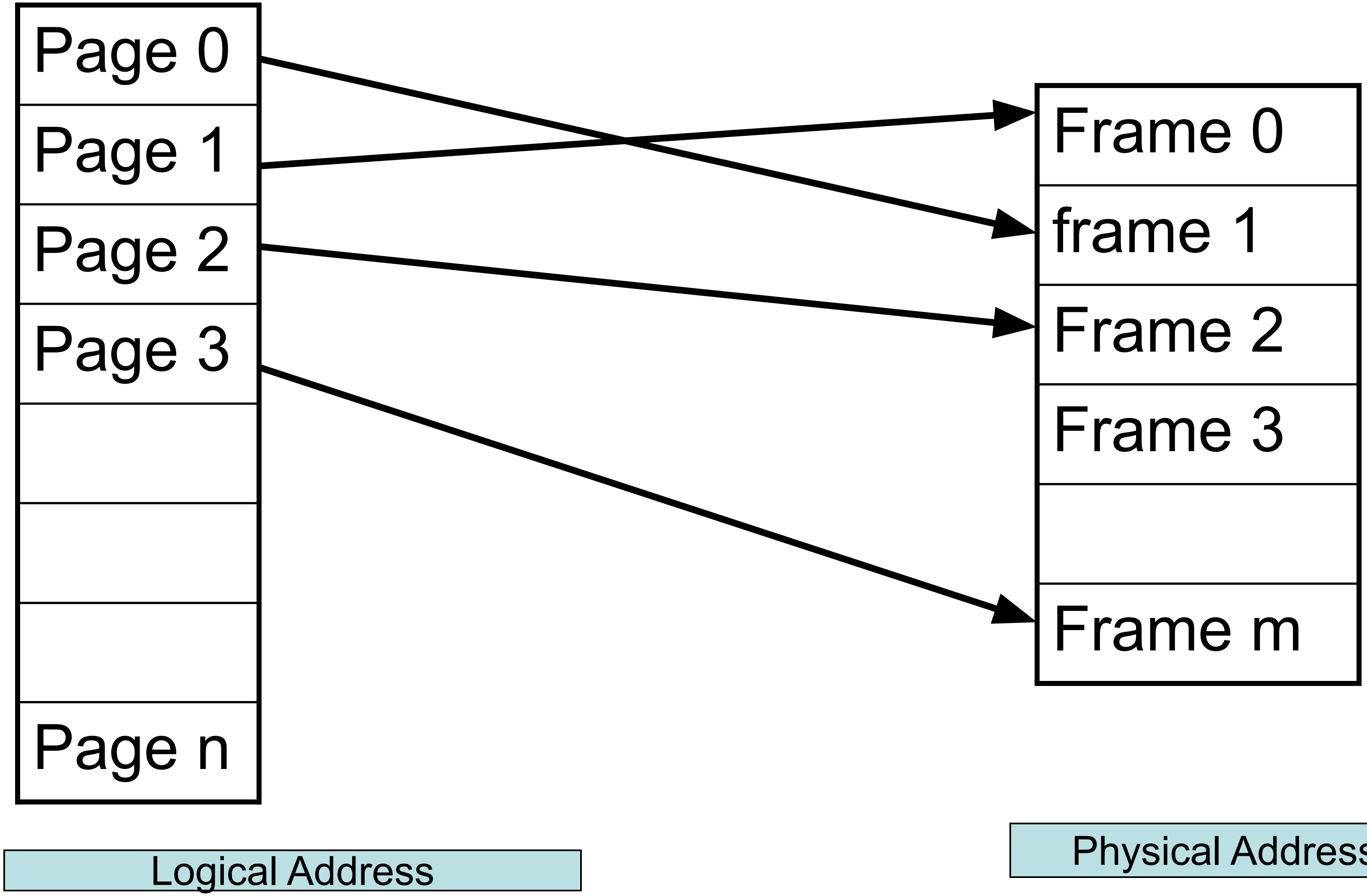
- Internal fragmentation (fixed Partition)
- External fragmentation (variable Partition)
- Compaction
- Contiguous allocation
- The system may not have enough memory to hold complete program(Overlaying technique required)

Dynamic Partitioning (paging)

- Memory is dynamically partitioned at run time and is allocated to processes
- Logical address : is a reference to memory location independent of current assignment of data to memory
- The logical address space is divided into fixed size small chunks (usually 4 KB to 4 MB) known as page
- The Physical memory is also divided into fixed size chunk and are known as frames.
- The size of frame and page is always equal for a given system
- The frame and page size are always power of 2. Mainly for easy address computation

- When process pages are required to be loaded into system , OS allocates required number of frames where process pages are loaded.
- Frame need not be allocated in contiguous manner
- If we have a process of size 13KB and page size is 4KB then process is allocated 4 frames .
- Internal fragmentation occurs (18% unused space It is less sever as compared to fixed partitioning)

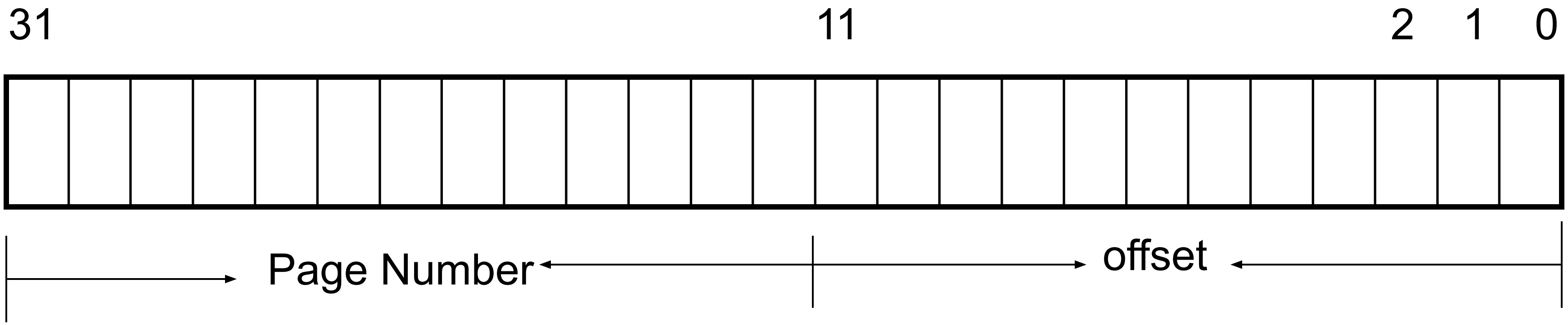
Logical to Physical Address Mapping



Address Translation Example and Issues

- Consider 80X86 Processor
- Byte organized Memory
- 32 Bit IP (logical address space 4GB)
- Address lines 32 (maximum Physical Memory 4 GB)
- Page size 4KB

Logical Address

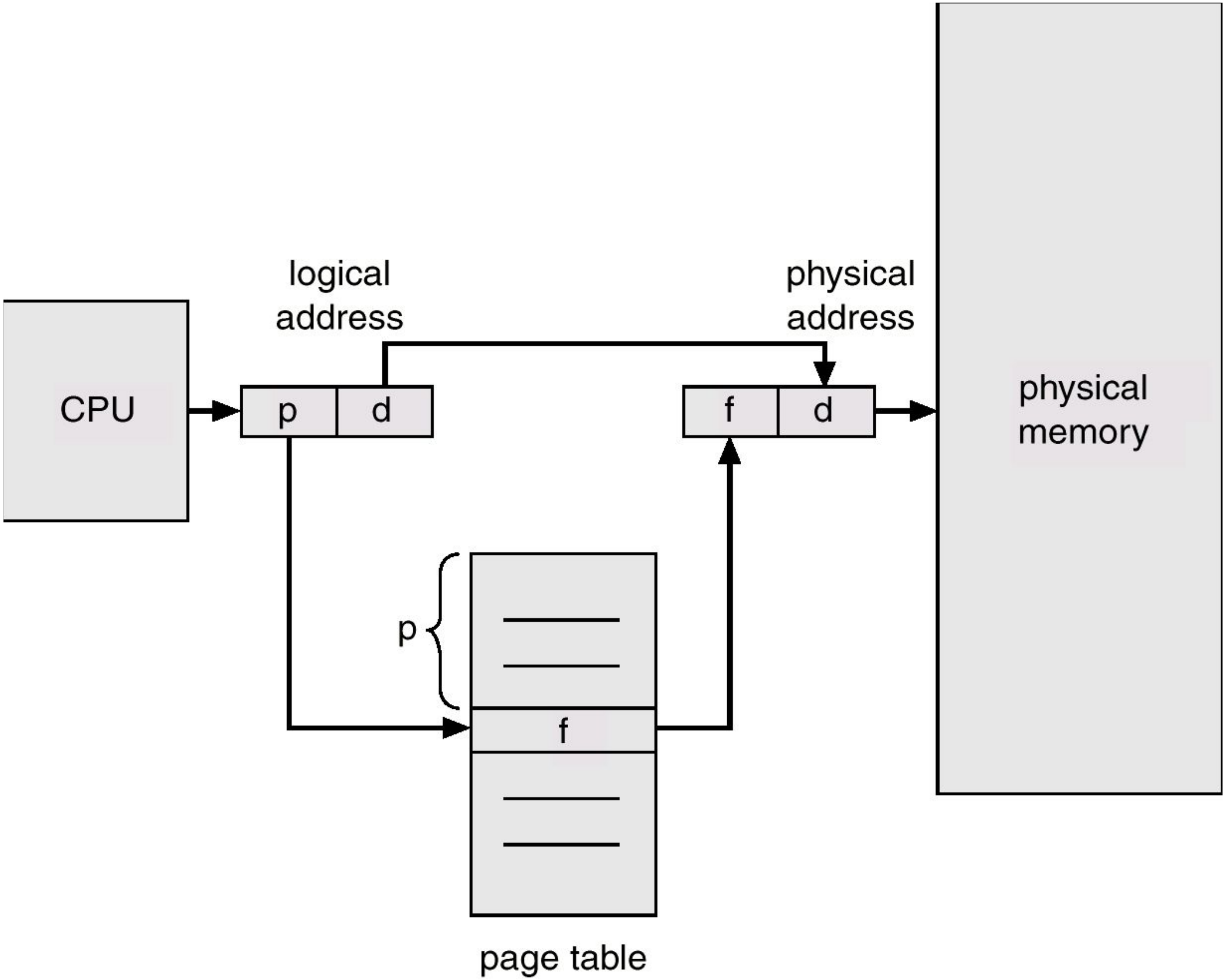


Logical address is divided into two parts , 12 bit offset within a page and 20 bit page number

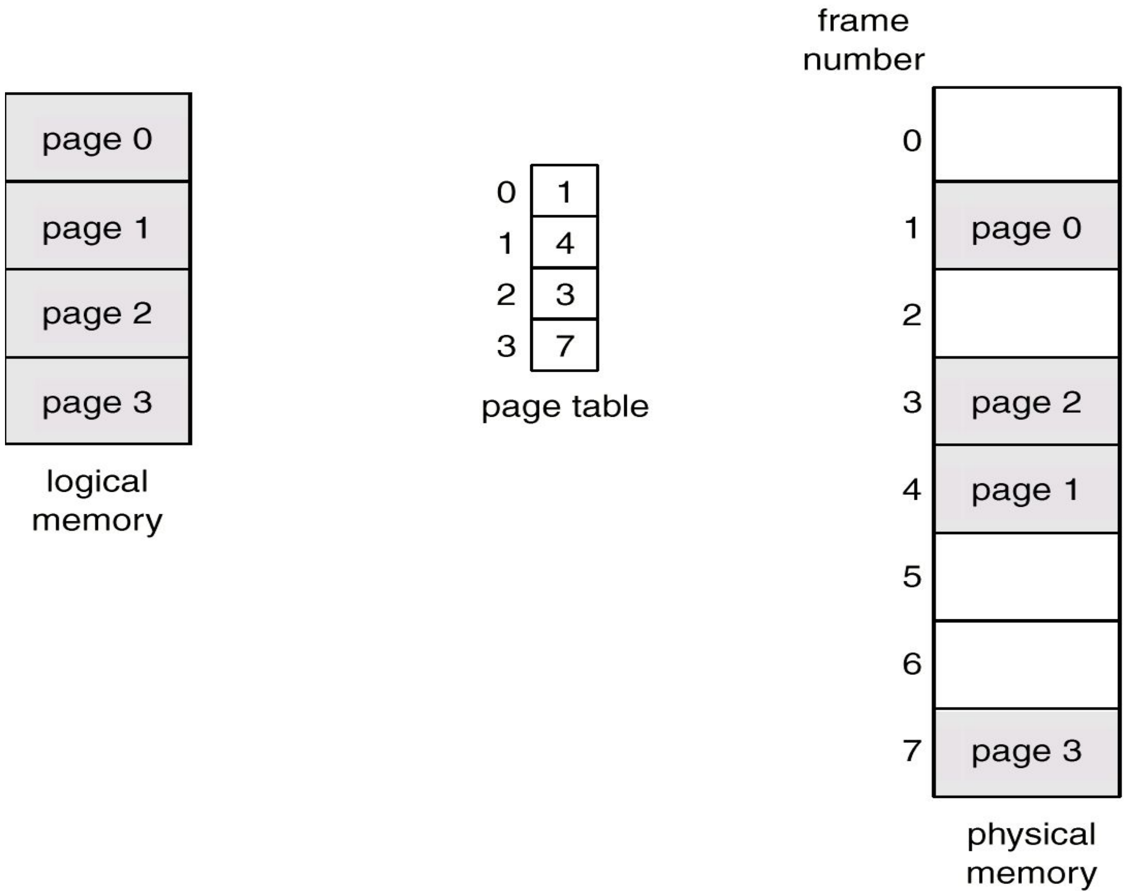
This logical address is translated to 32 Bit Physical address consisting of 12 bit Offset and 20 bit frame number

The address translation is the job of Memory management unit (MMU) Hardware

Address Translation Architecture



Paging Example



page 0
page 1
page 2
page 3

logical memory

0	1
1	4
2	3
3	7

page table

frame number

0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

physical memory

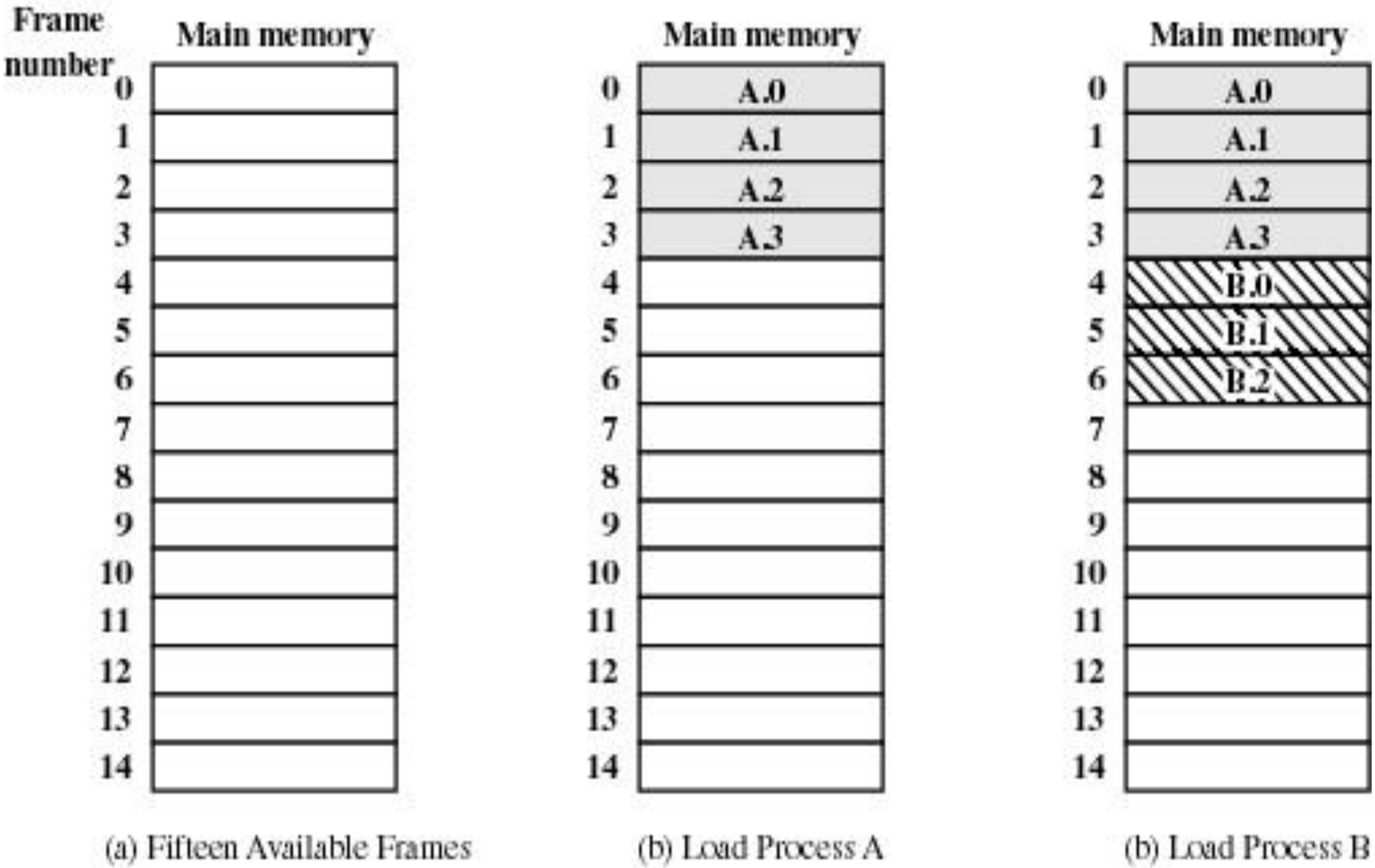


Figure 7.9 Assignment of Process Pages to Free Frames

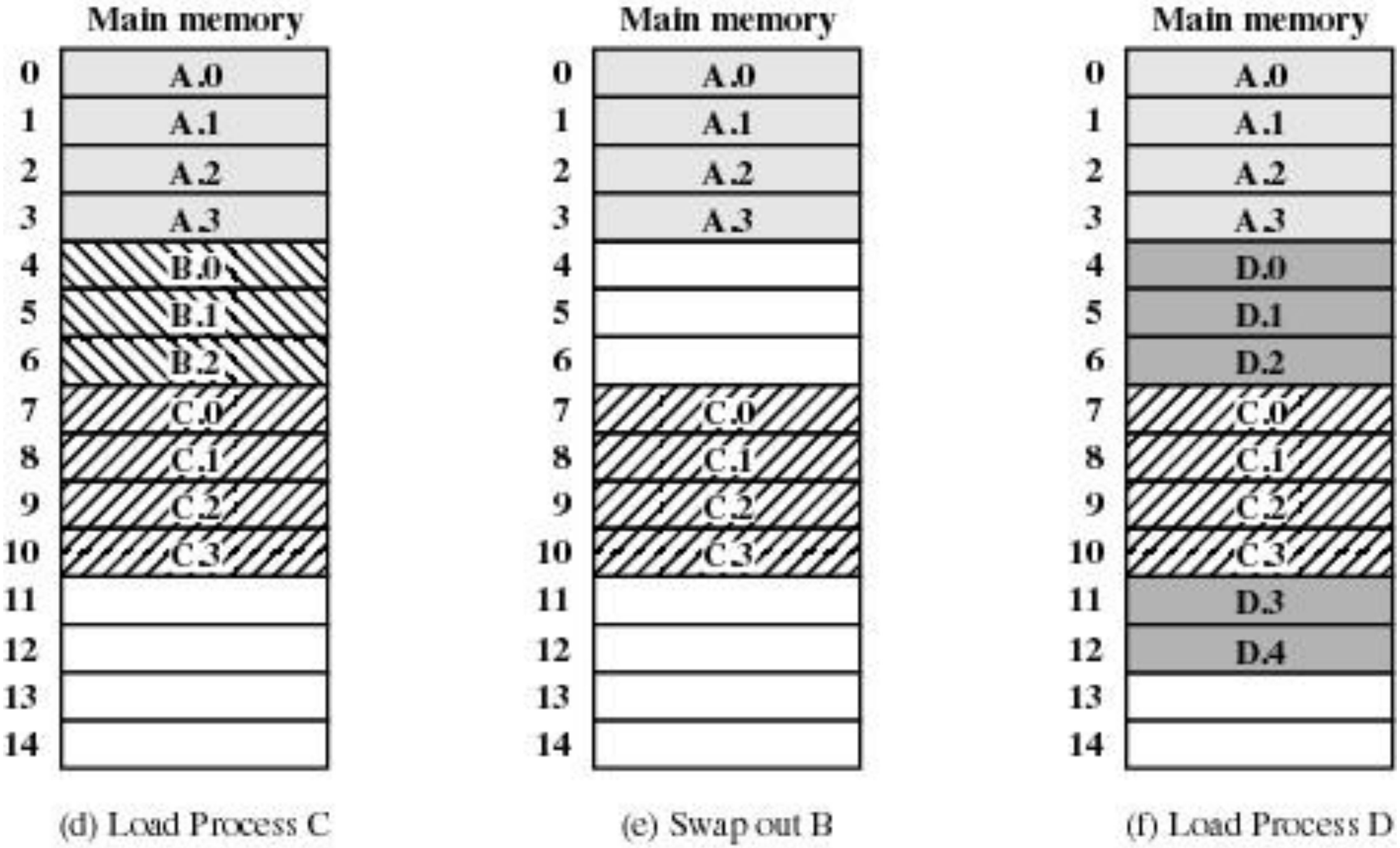
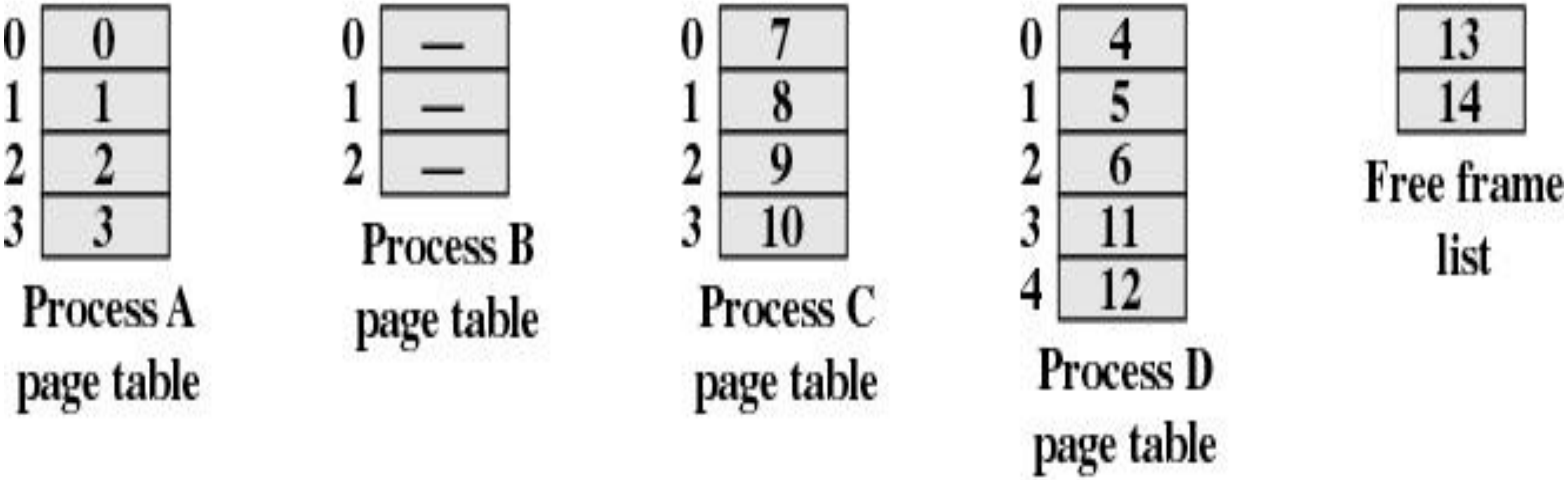


Figure 7.9 Assignment of Process Pages to Free Frames



Process A page table

Page	Frame
0	0
1	1
2	2
3	3

Process B page table

Page	Frame
0	—
1	—
2	—

Process C page table

Page	Frame
0	7
1	8
2	9
3	10

Process D page table

Page	Frame
0	4
1	5
2	6
3	11
4	12

Free frame list

13
14

Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

- Internal fragmentation results
- To reduce internal fragmentation we need to reduce the page size.
- If page size is small there will be an overhead of page table entry.
- Page sizes now in use are typically between 4KB and 8KB.
- Some CPU and kernel supports multiple page sizes. Example : Solaris 8KB and 4MB pages
- Main advantage of paging is the separation of user view of memory & actual physical memory.
- User process cannot access memory outside of its page table.

Implementation Issues

- Where does page table resides in memory ?
- What is the minimum number of frame that can be allocated?
- How to reduce page table size?

Multilevel Page Directory

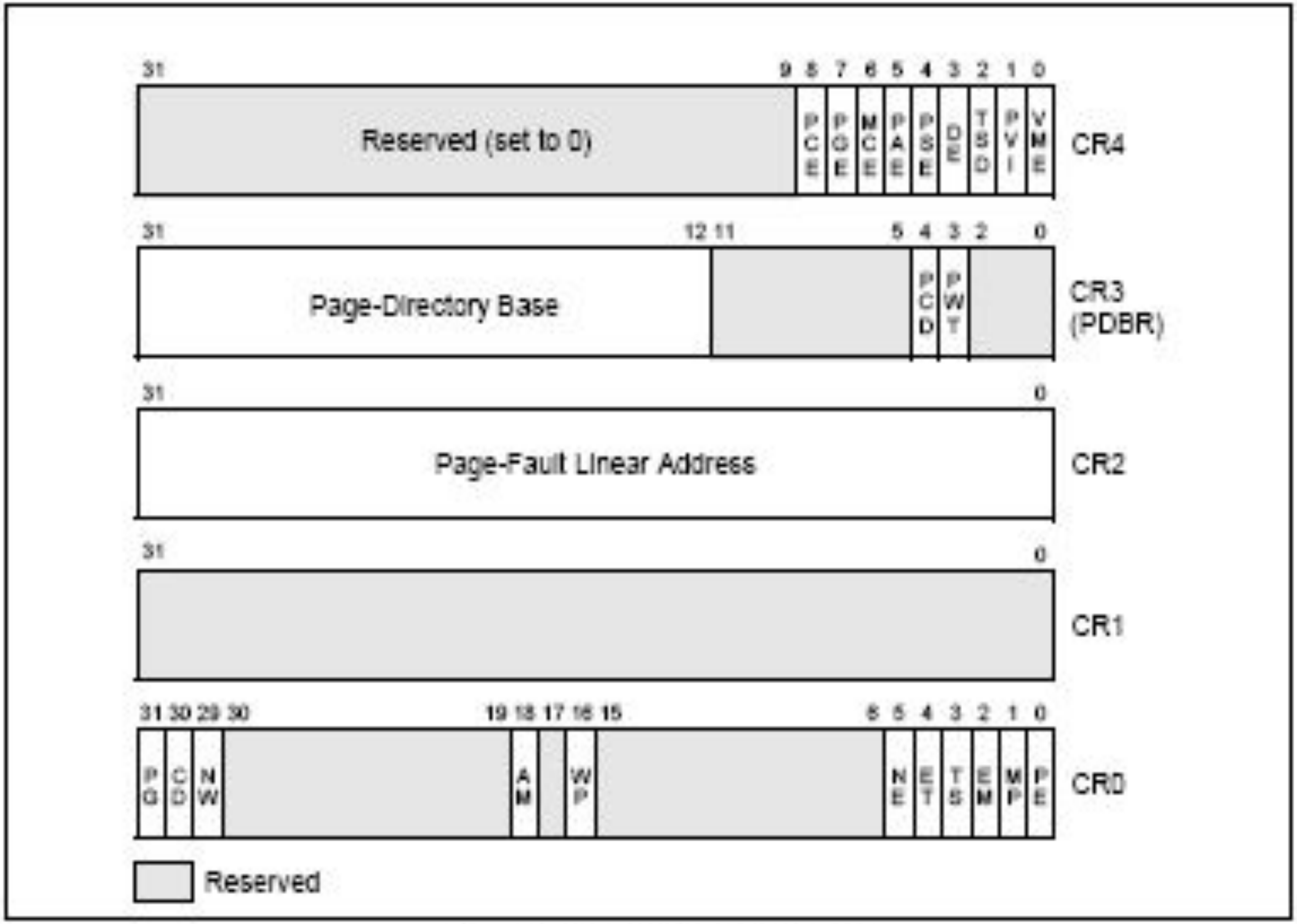
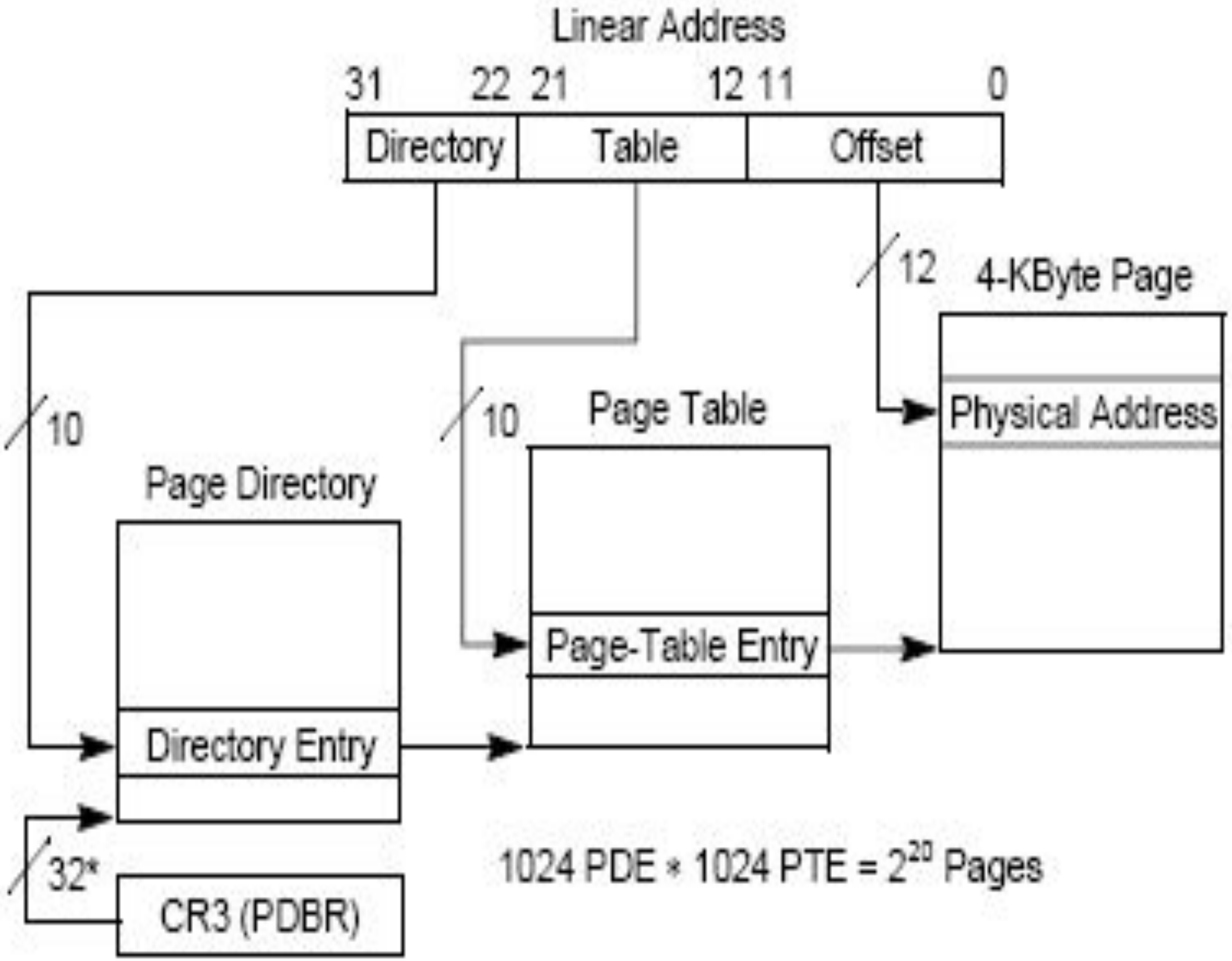


Figure 2-5. Control Registers



*32 bits aligned onto a 4-KByte boundary.

Figure 3-12. Linear Address Translation (4-KByte Pages)

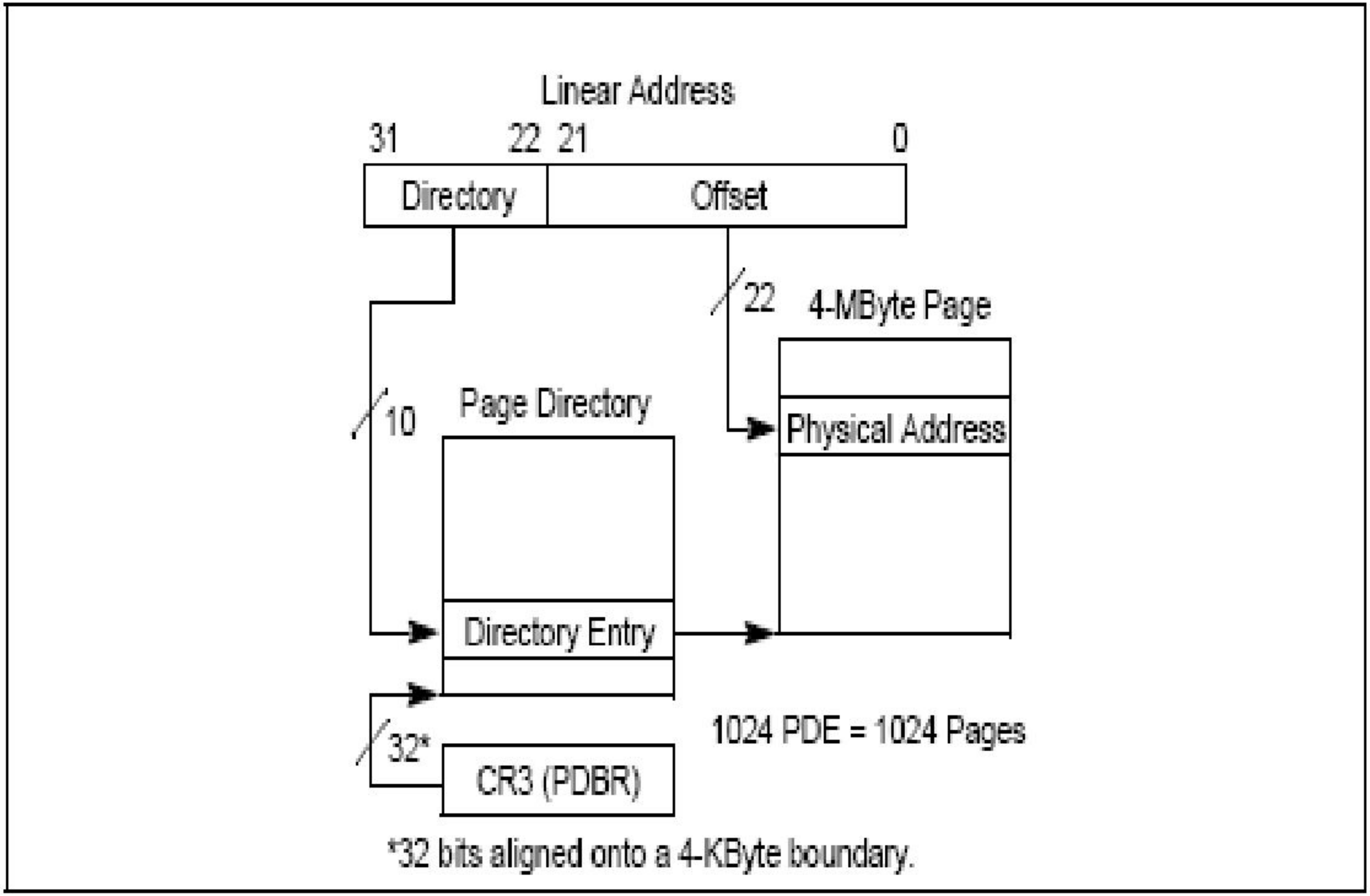
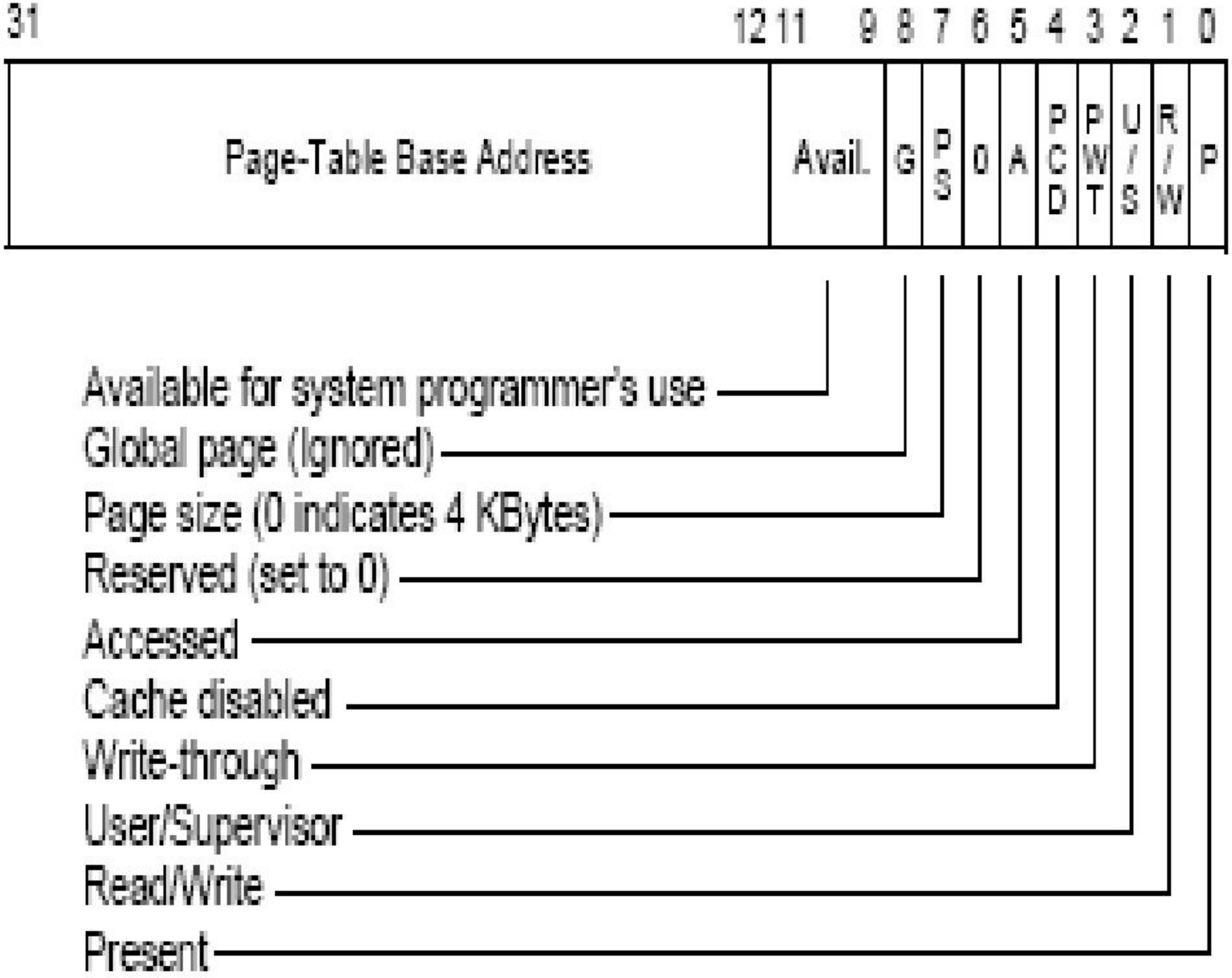


Figure 3-13. Linear Address Translation (4-MByte Pages)

Page-Directory Entry (4-KByte Page Table)



31 12 11 9 8 7 6 5 4 3 2 1 0

Page-Table Base Address

Avail. G PS 0 A PCD PWT U/S R/W P

Available for system programmer's use

Global page (Ignored)

Page size (0 indicates 4 KBytes)

Reserved (set to 0)

Accessed

Cache disabled

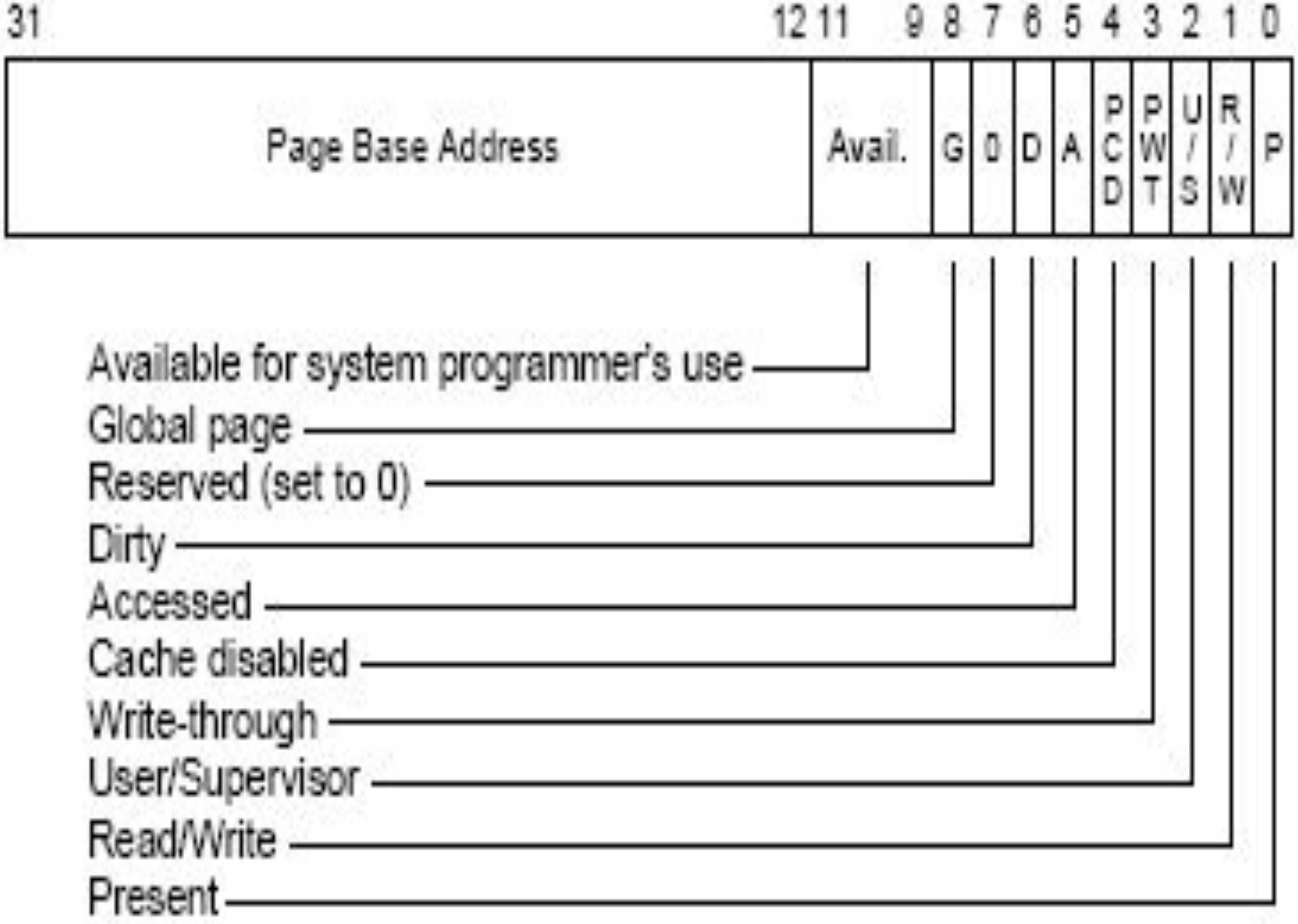
Write-through

User/Supervisor

Read/Write

Present

Page-Table Entry (4-KByte Page)



31 … 12	11 … 9	8	7	6	5	4	3	2	1	0
Page Base Address	Avail.	G	0	D	A	PCD	PWT	U/S	R/W	P

Available for system programmer's use

Global page

Reserved (set to 0)

Dirty

Accessed

Cache disabled

Write-through

User/Supervisor

Read/Write

Present

Associative Register

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs)*
- Associative registers – parallel search

Page #	Frame #

- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory

Effective Access Time

- Example
  - Hit ratio   80%
  - 20 nsec to search in TLB
  - 100 nsec to access Memory
  - Effective Memory Access Time
  - (E M A T) = (0.80 X 120) + (0.20 X 220) = 140 nsec

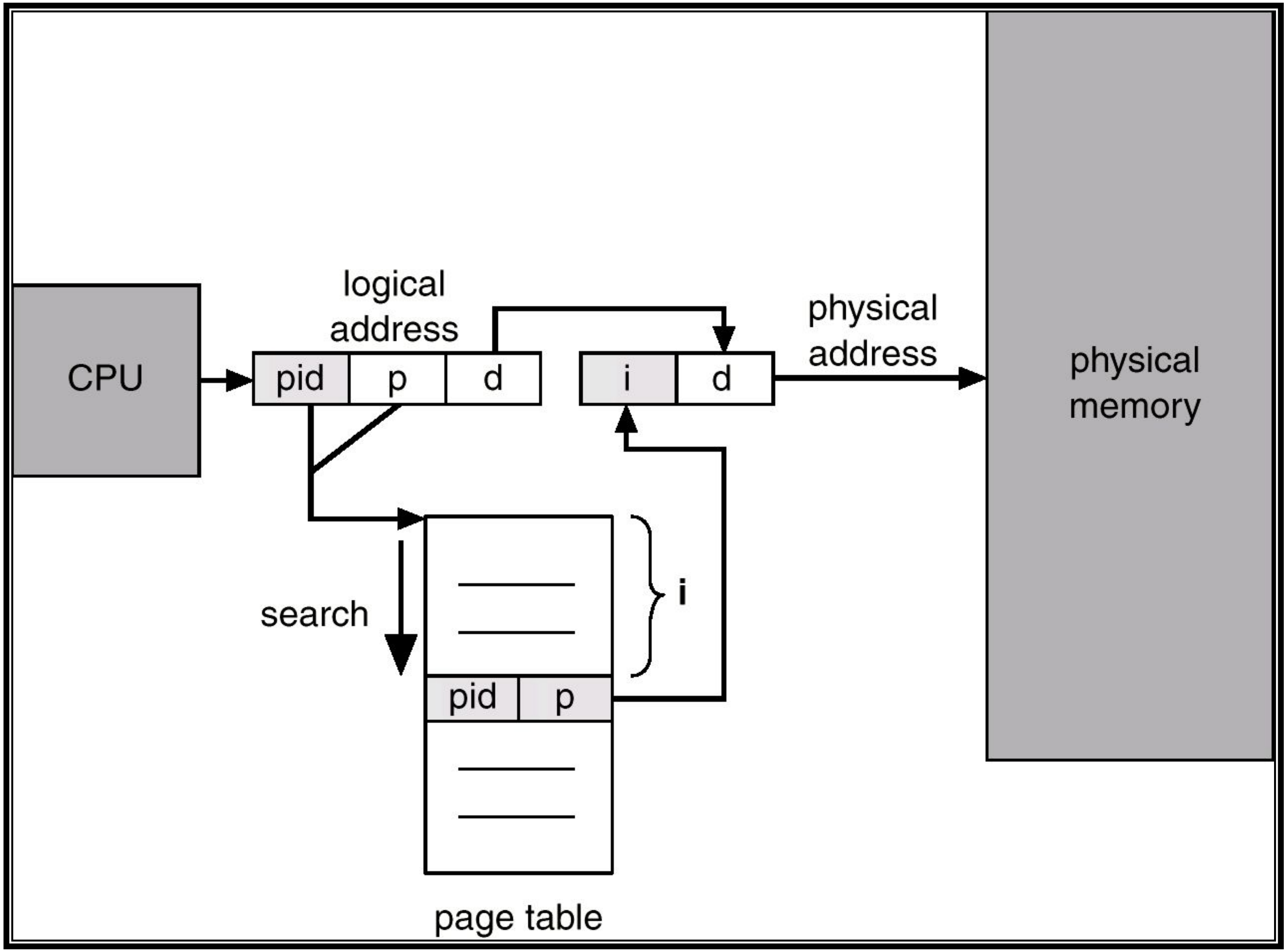
If hit ratio is 98% then

EAT = (0.98 X 120) + (0.02 X 220) = 122 nsec

Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

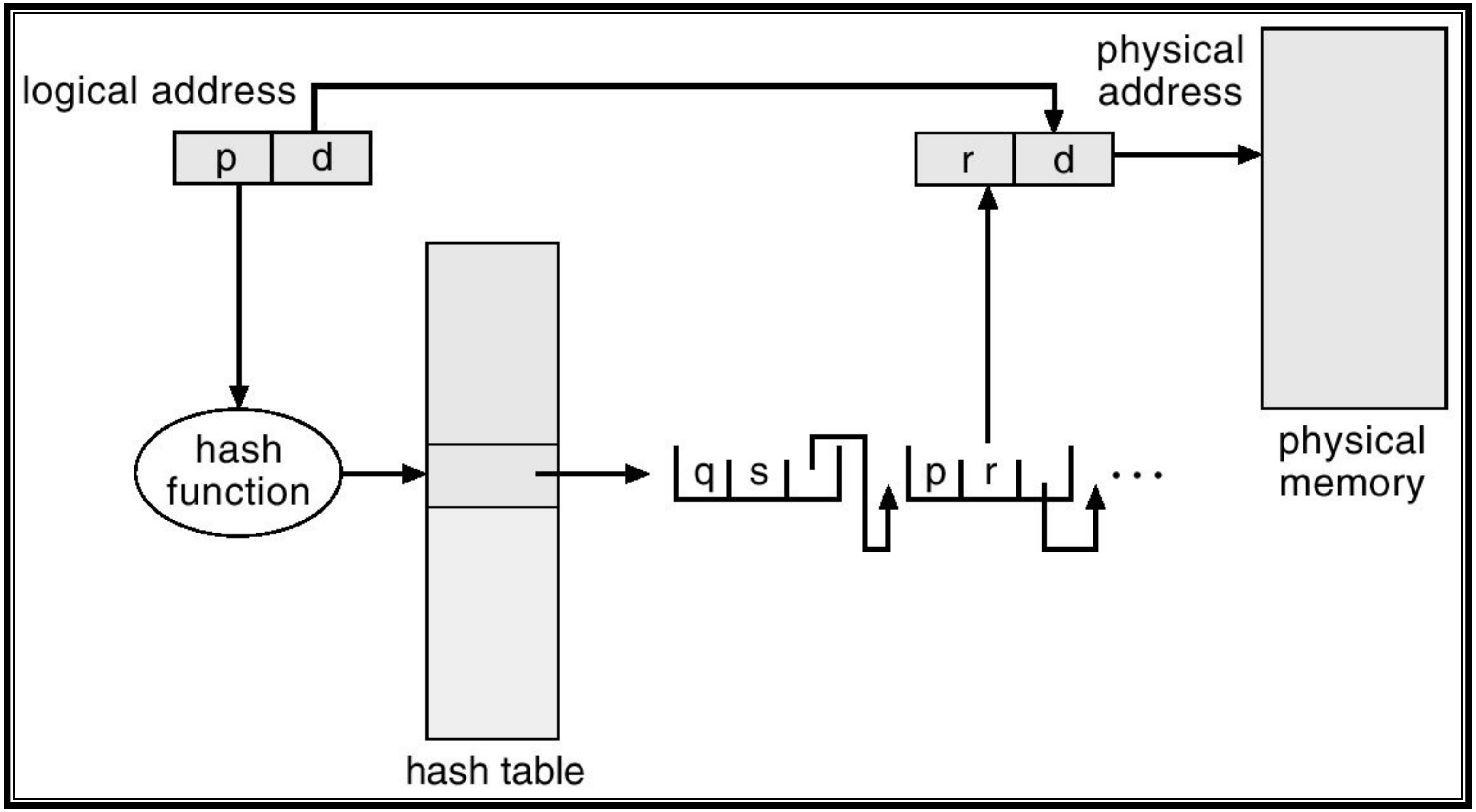
Inverted Page Table



Hashed Page Tables

- Common in logical address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Table



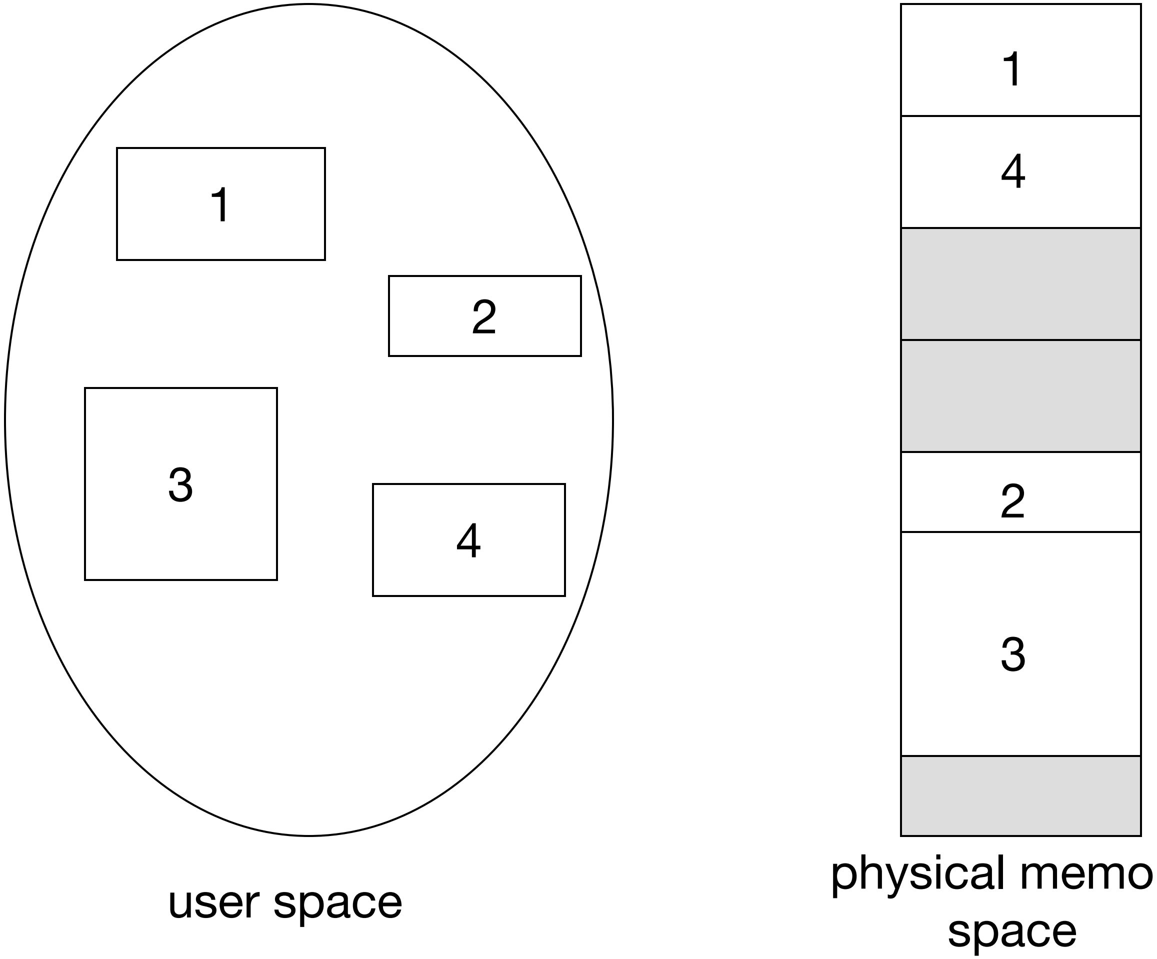
Segmentation

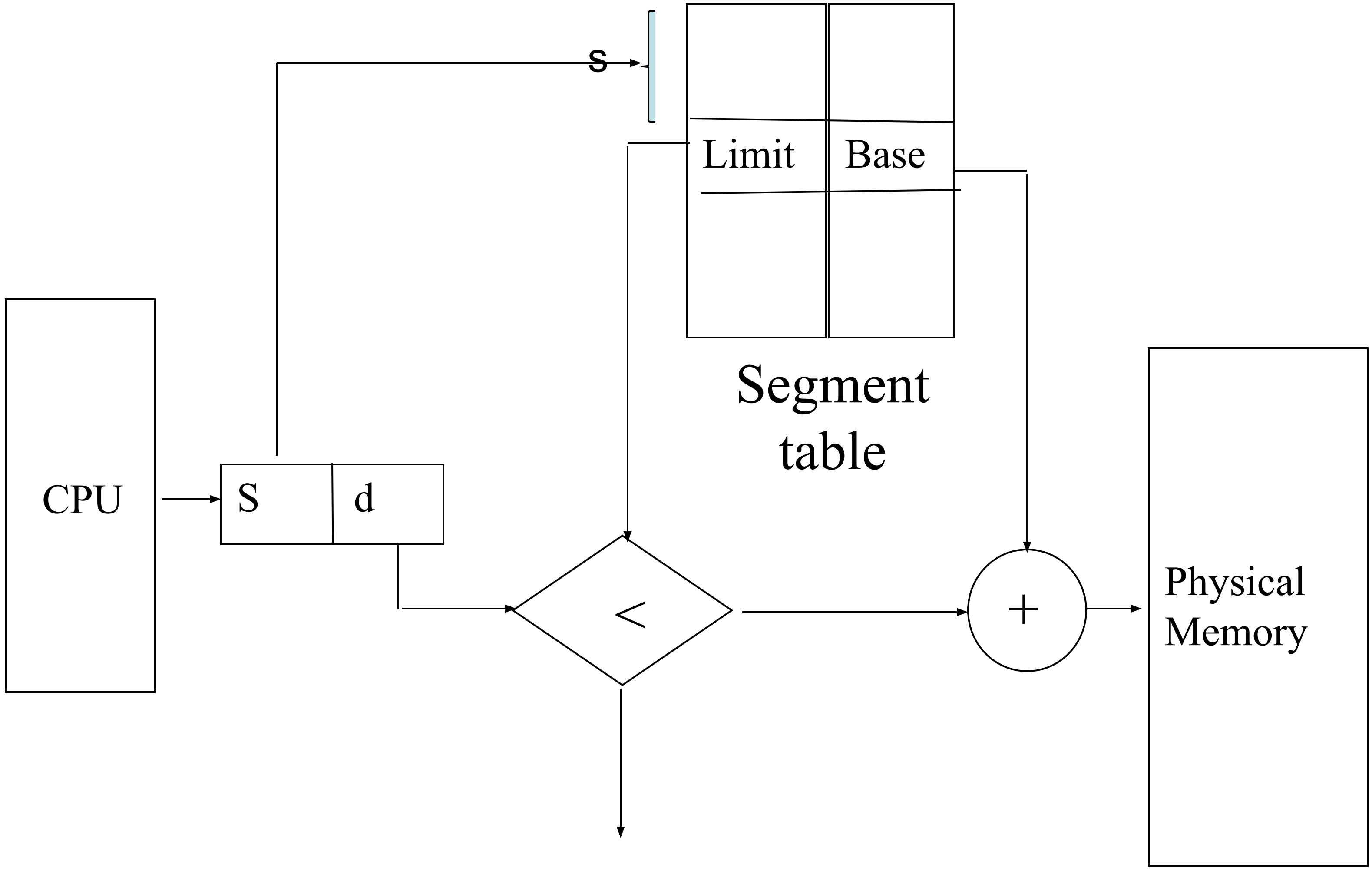
- Memory-management scheme that supports user view of memory. (User's view of memory is not same as physical memory.)
- A logical address space is a collection of segments.
- A program is a collection of segments. A segment is a logical unit such as:

  main program,
  procedure,
  function,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays

Address : segment : offset (segment no, offset)

Logical View of Segmentation





s

Limit

Base

Segment table

CPU

S

d

<

+

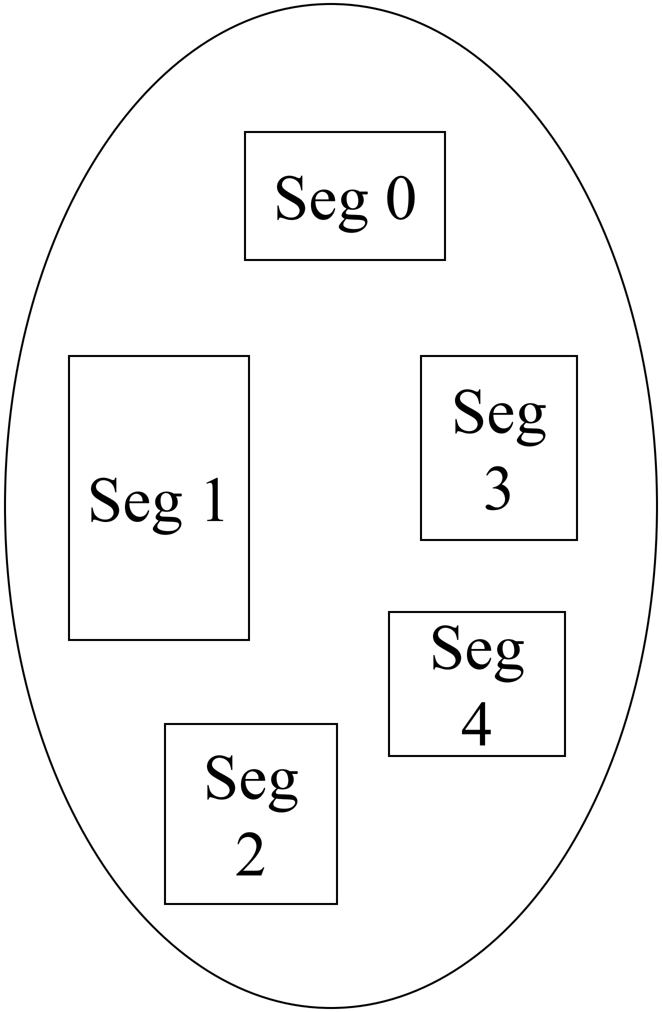
Physical Memory

Trap; addressing error

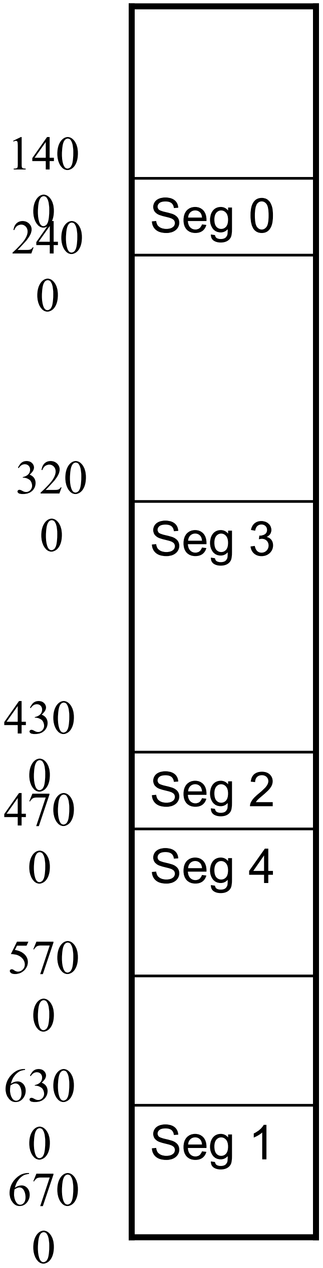
Segmentation Architecture

- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table’s location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number *s* is legal if *s* < STLR.



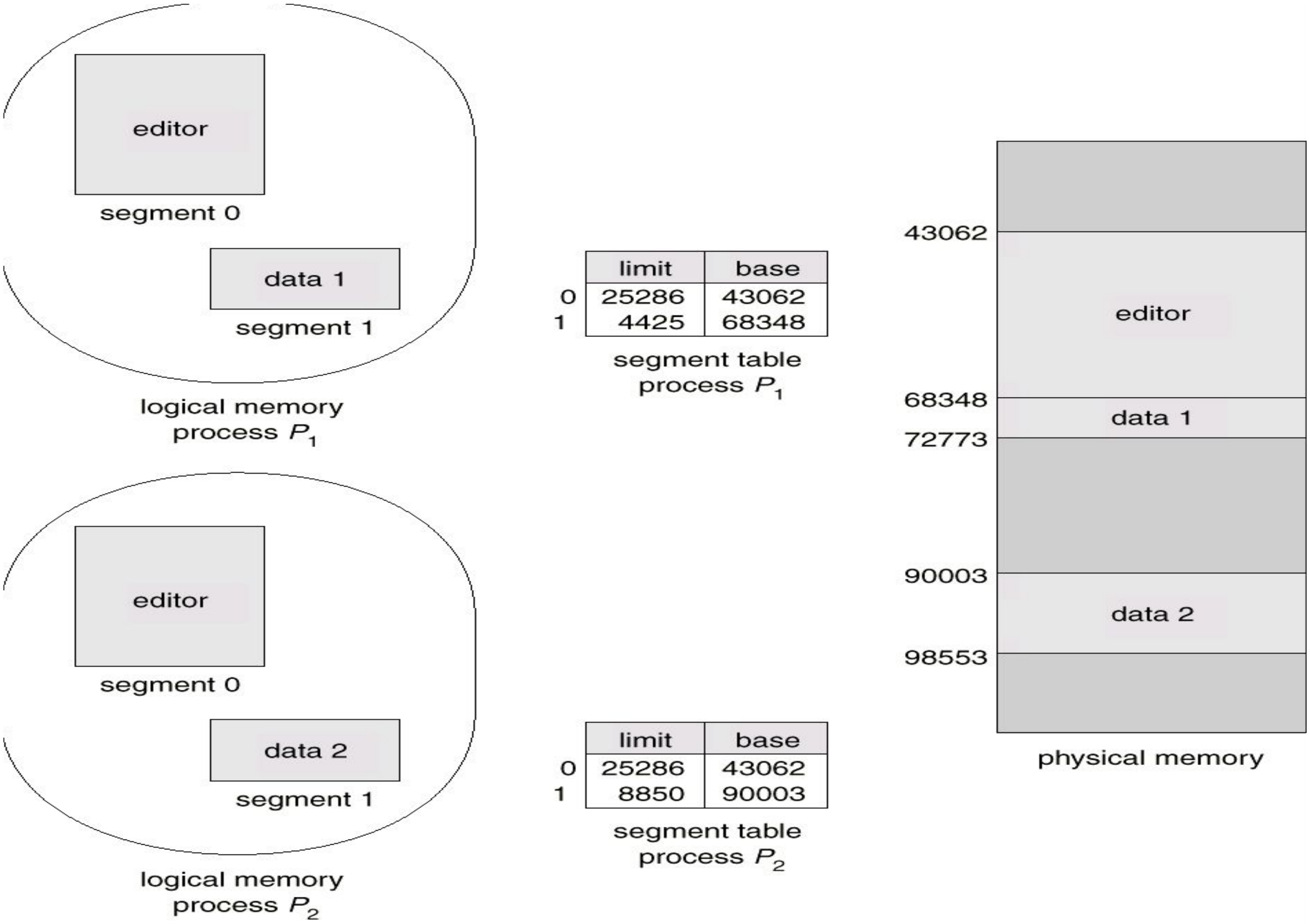
	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700



Segmentation Architecture (Cont.)

- Relocation.
  - dynamic
  - by segment table
- Sharing.
  - shared segments
  - same segment number
- Allocation.
  - first fit/best fit
  - external fragmentation

Sharing of segments



	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

Segmentation In 80X86 Processor

- Address is specified as Segment : Offset
- X86 Processor has Several segment registers
  - CS (code segment)
  - DS (Data Segment)
  - SS (Stack segment)
  - ES (Extra segment)
  - GS & FS segment

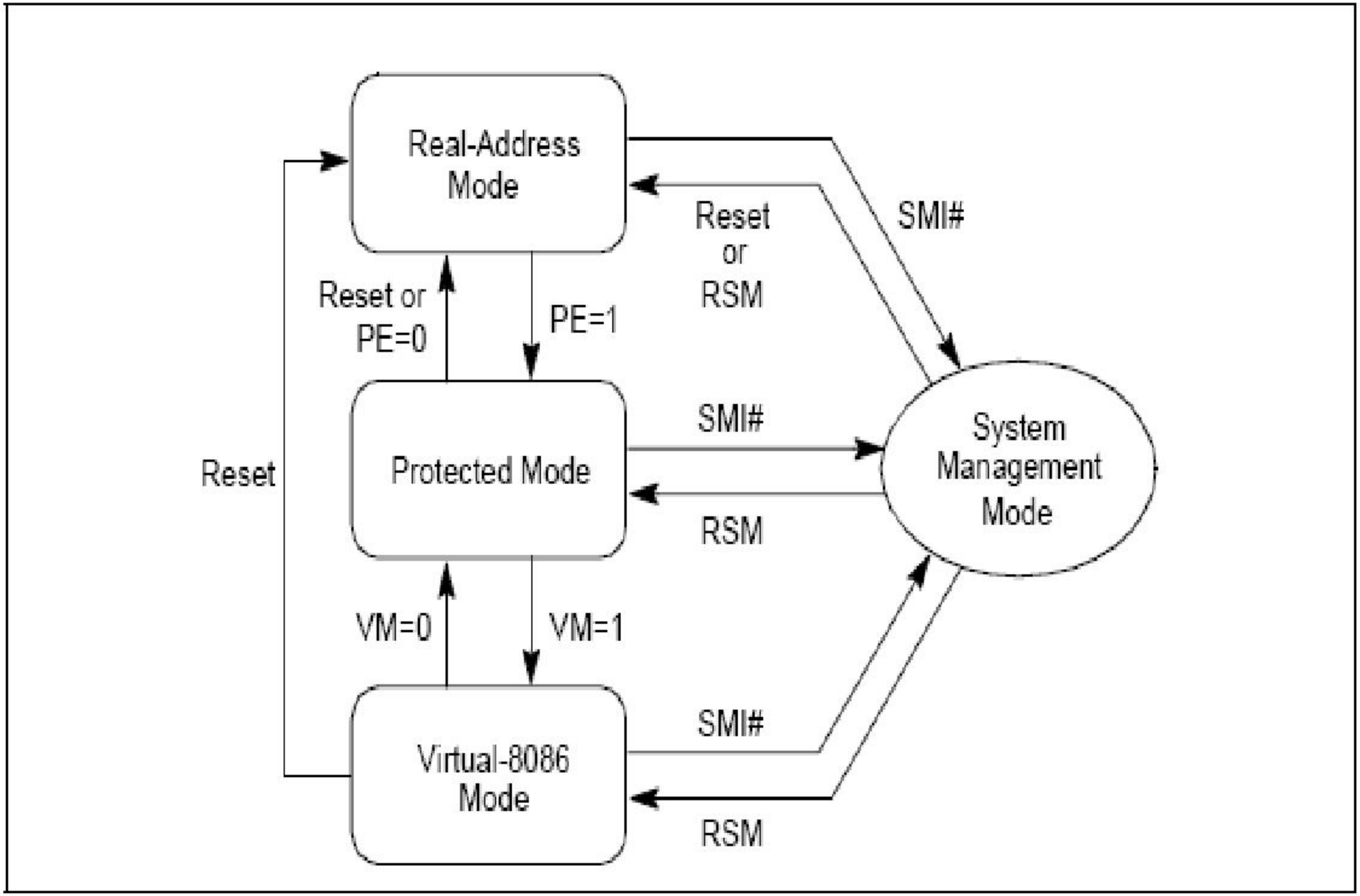
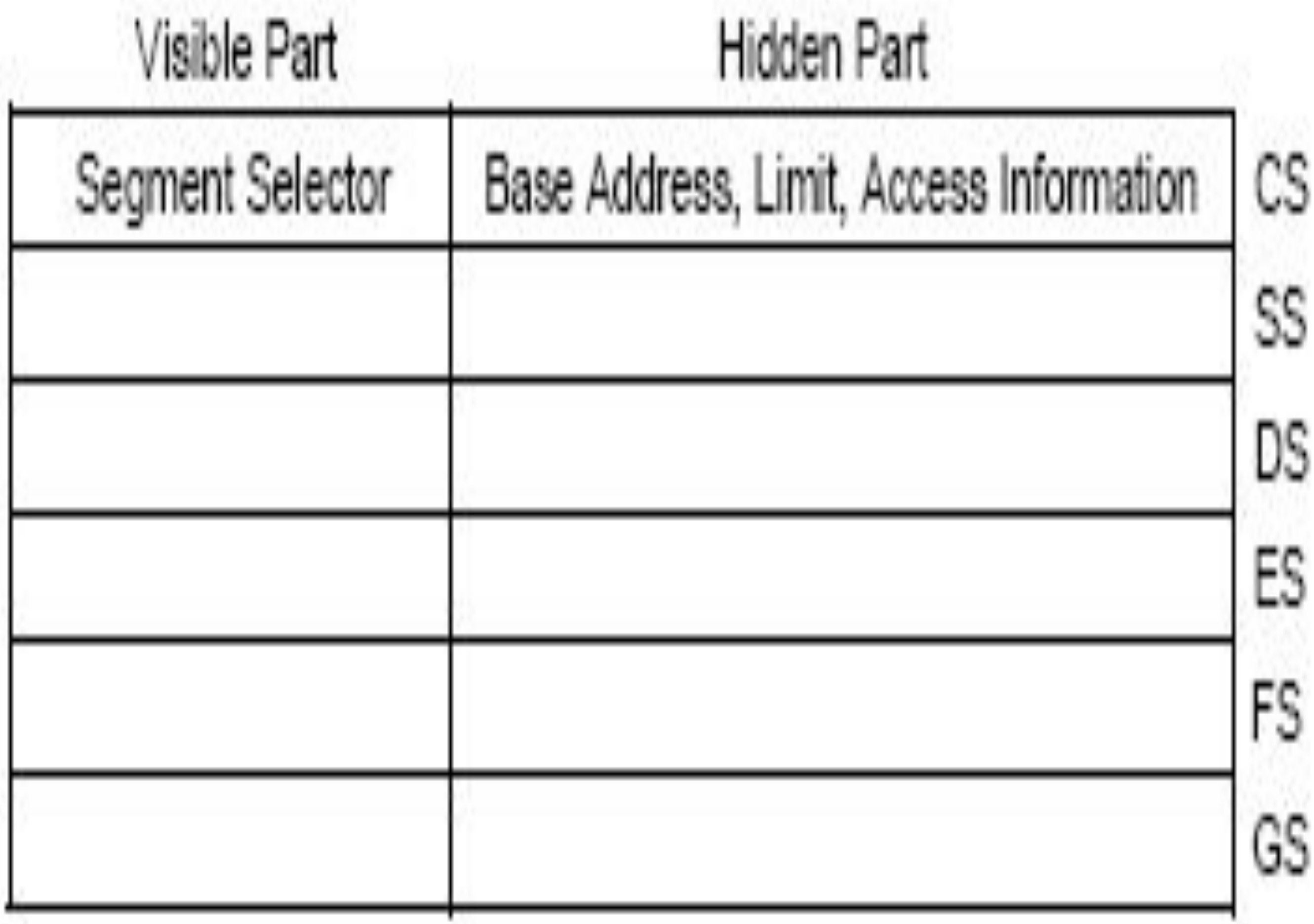
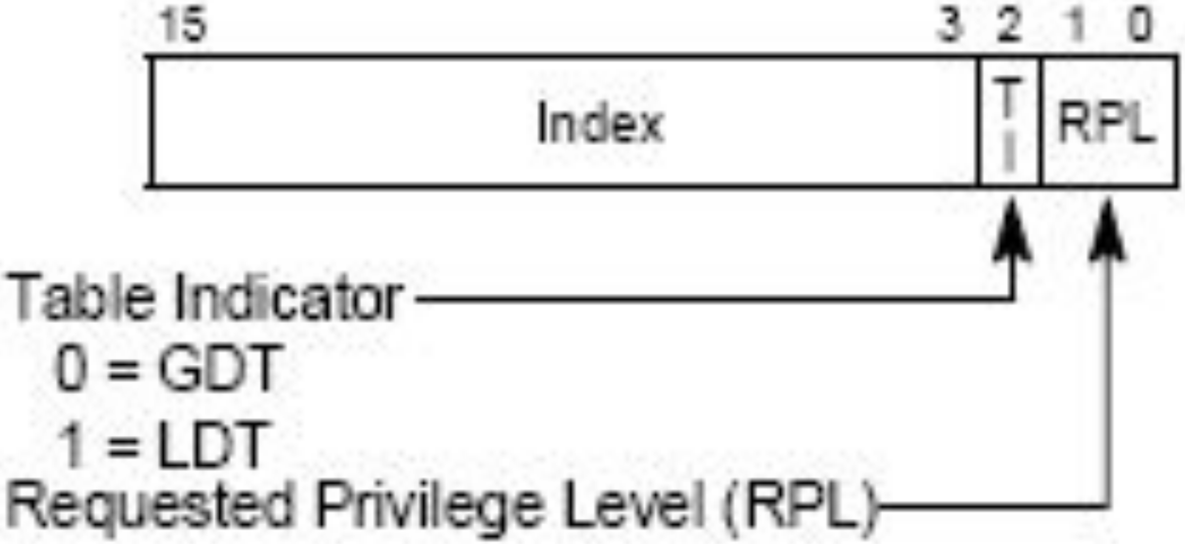


Figure 2-2. Transitions Among the Processor's Operating Modes



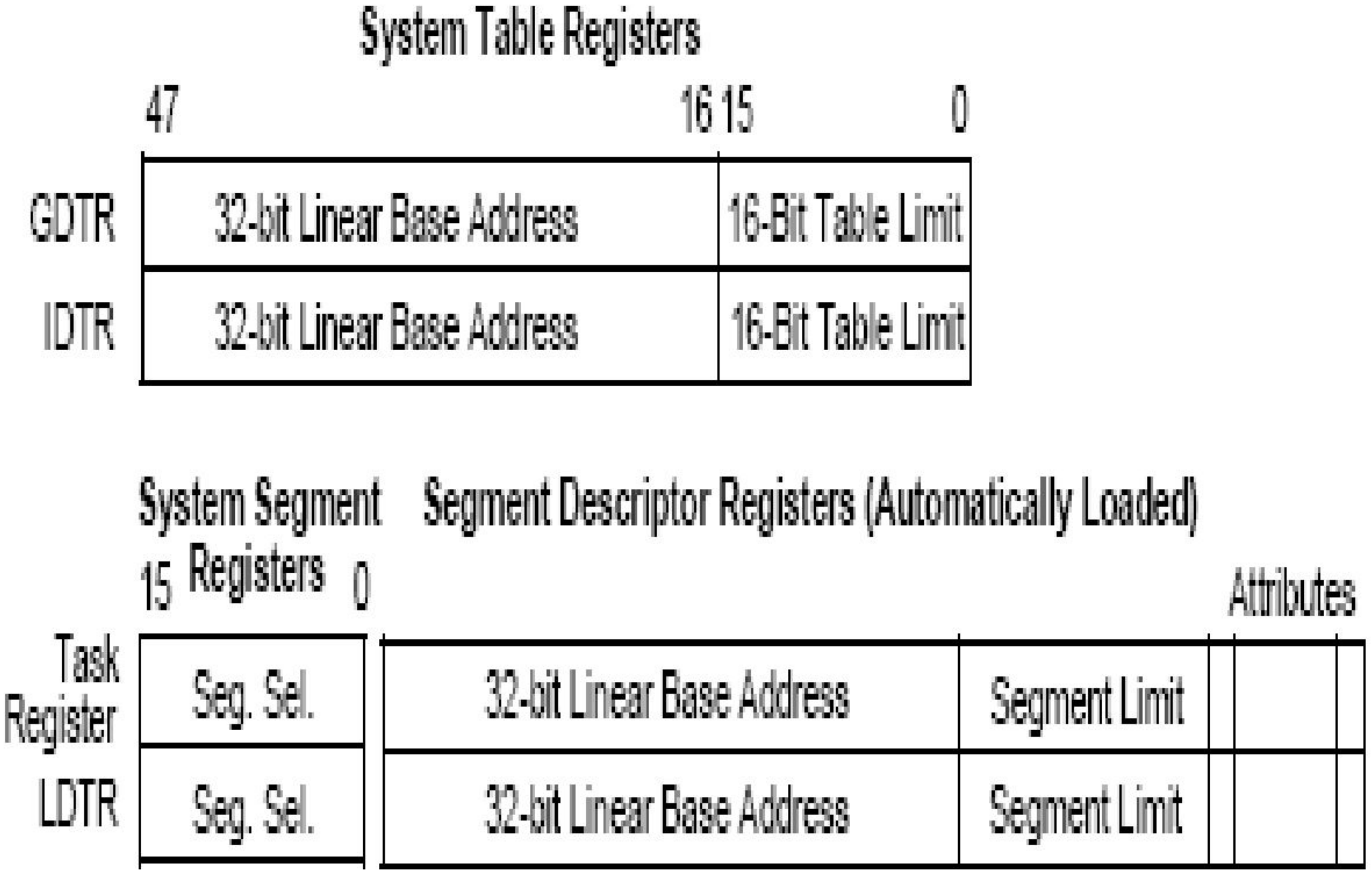
Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

Segment Registers


15
3
2
1
0
Index
TI
RPL
Table Indicator
0 = GDT
1 = LDT
Requested Privilege Level (RPL)

Memory Management Registers

- Processor provides 4 memory management registers :
  - GDTR Global descriptor table register
  - LDTR Local Descriptor Table register
  - IDTR Interrupt descriptor Table register
  - TR Task register
- These registers specify the location of data structure which control segmented memory management



System Table Registers

	47 16	15 0
GDTR	32-bit Linear Base Address	16-Bit Table Limit
IDTR	32-bit Linear Base Address	16-Bit Table Limit

System Segment Registers

Segment Descriptor Registers (Automatically Loaded)

	15 Registers 0			Attributes		
Task Register	Seg. Sel.	32-bit Linear Base Address	Segment Limit			
LDTR	Seg. Sel.	32-bit Linear Base Address	Segment Limit			

GDTR Register

- GDTR Holds
  - 32 Bit Base address of GDT
  - 16 bit limit value , defining size of GDT
  - On Power up/reset, Base part is set to 0 and limit is set to FFFFH
- Each entry of GDT is of 8 bytes
- LGDT, SGDT instruction can be used to load / store GDTR

IDTR Register

- IDTR Holds
  - 32 Bit Base address of IDT
  - 16 bit limit value , defining size of IDT
  - On Power up/reset, Base part is set to 0 and limit is set to FFFFH
- Each entry of IDT is of 8 bytes
- LIDT, SIDT instruction can be used to load / store IDTR

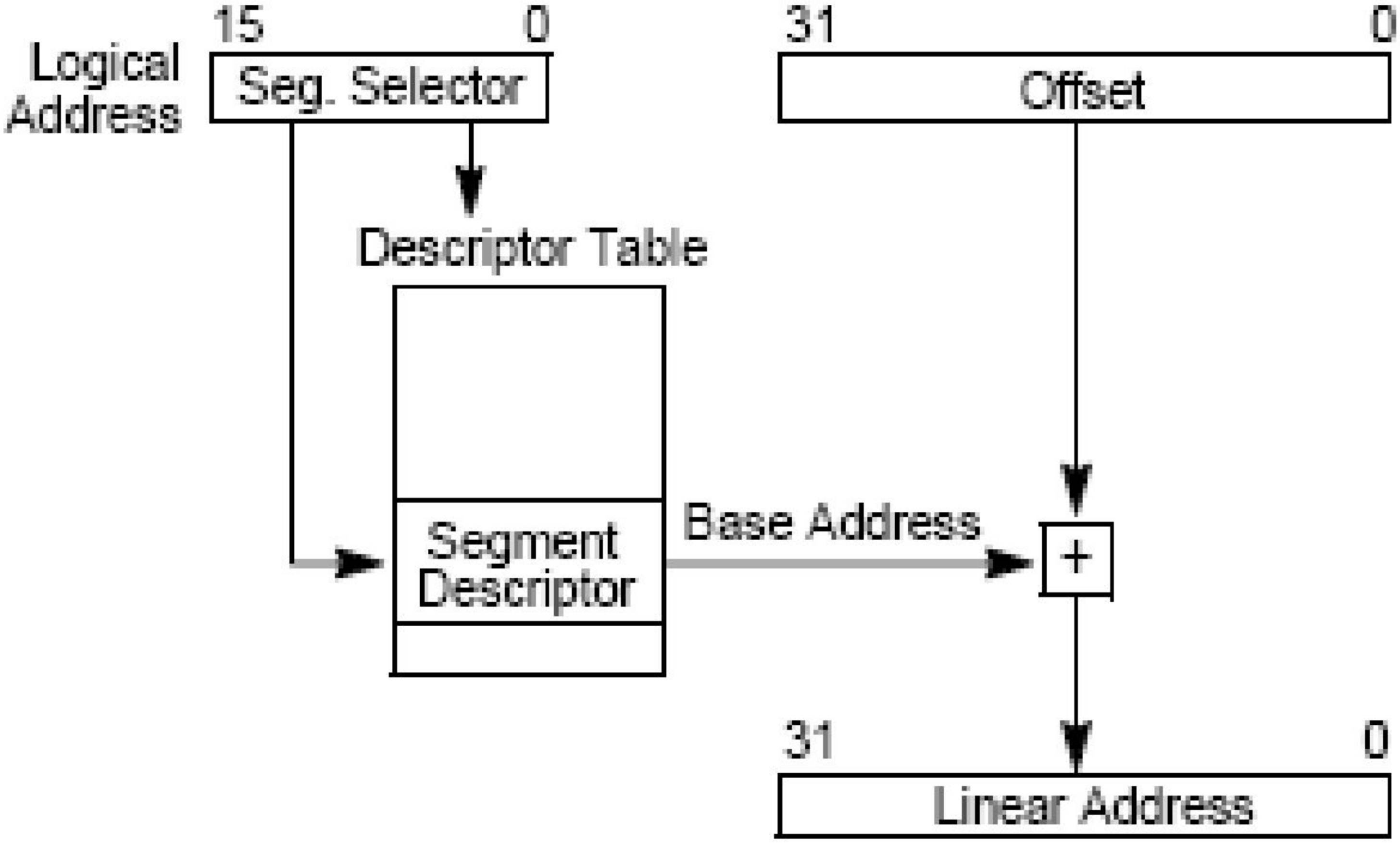
LDTR Register

- LDTR Holds
  - 32 Bit Base address of LDT
  - 16 bit limit value , defining size of LDT
  - Attribute for descriptor table
  - On Power up/reset, Base part is set to 0 and limit is set to FFFFH
- Each entry of LDT is of 8 bytes
- LIDT, SIDT instruction can be used to load / store IDTR

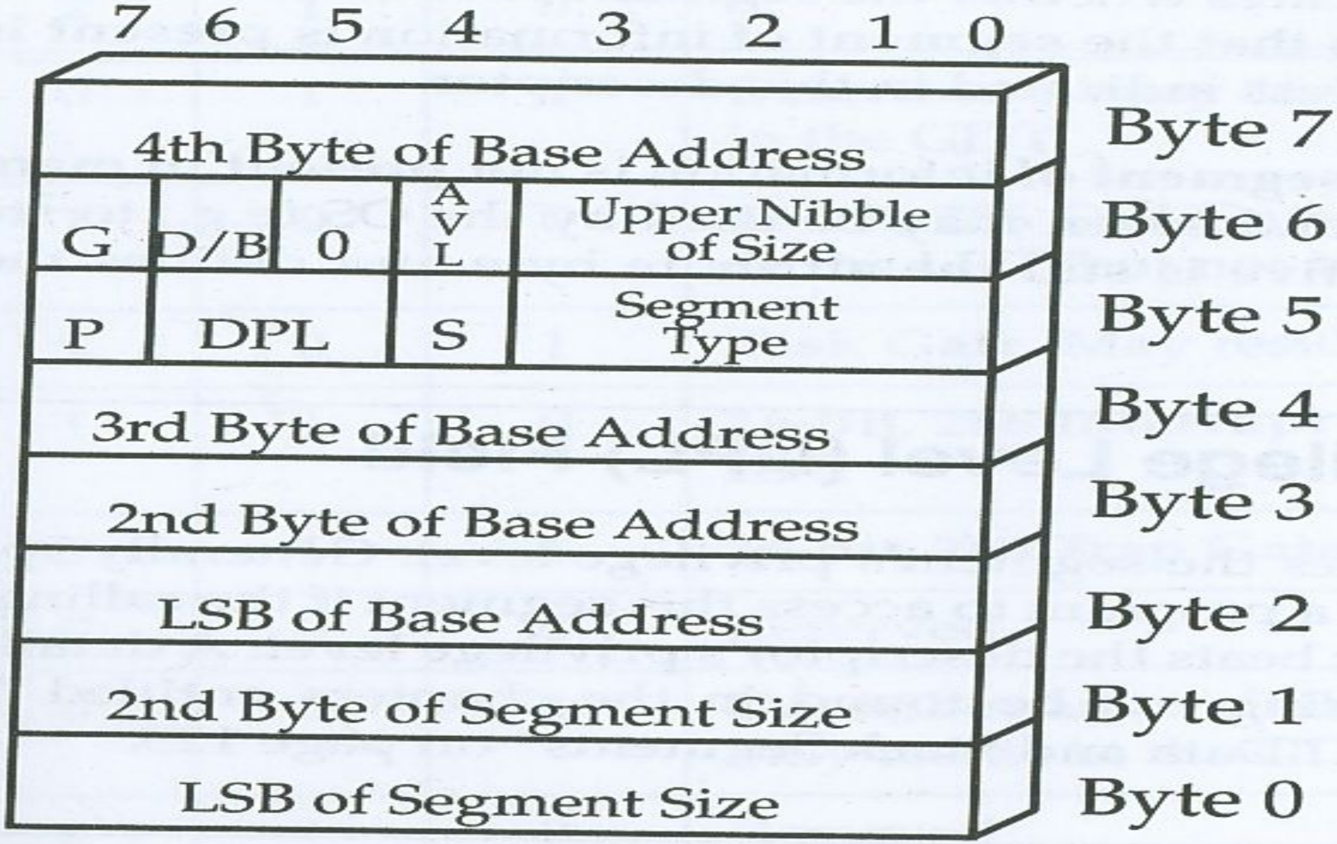
IDTR Register

- IDTR Holds
  - 32 Bit Base address of GDT
  - 16 bit limit value , defining size of GDT
  - On Power up/reset, Base part is set to 0 and limit is set to FFFFH
- Each entry of IDT is of 8 bytes
- LIDT, SIDT instruction can be used to load / store IDTR

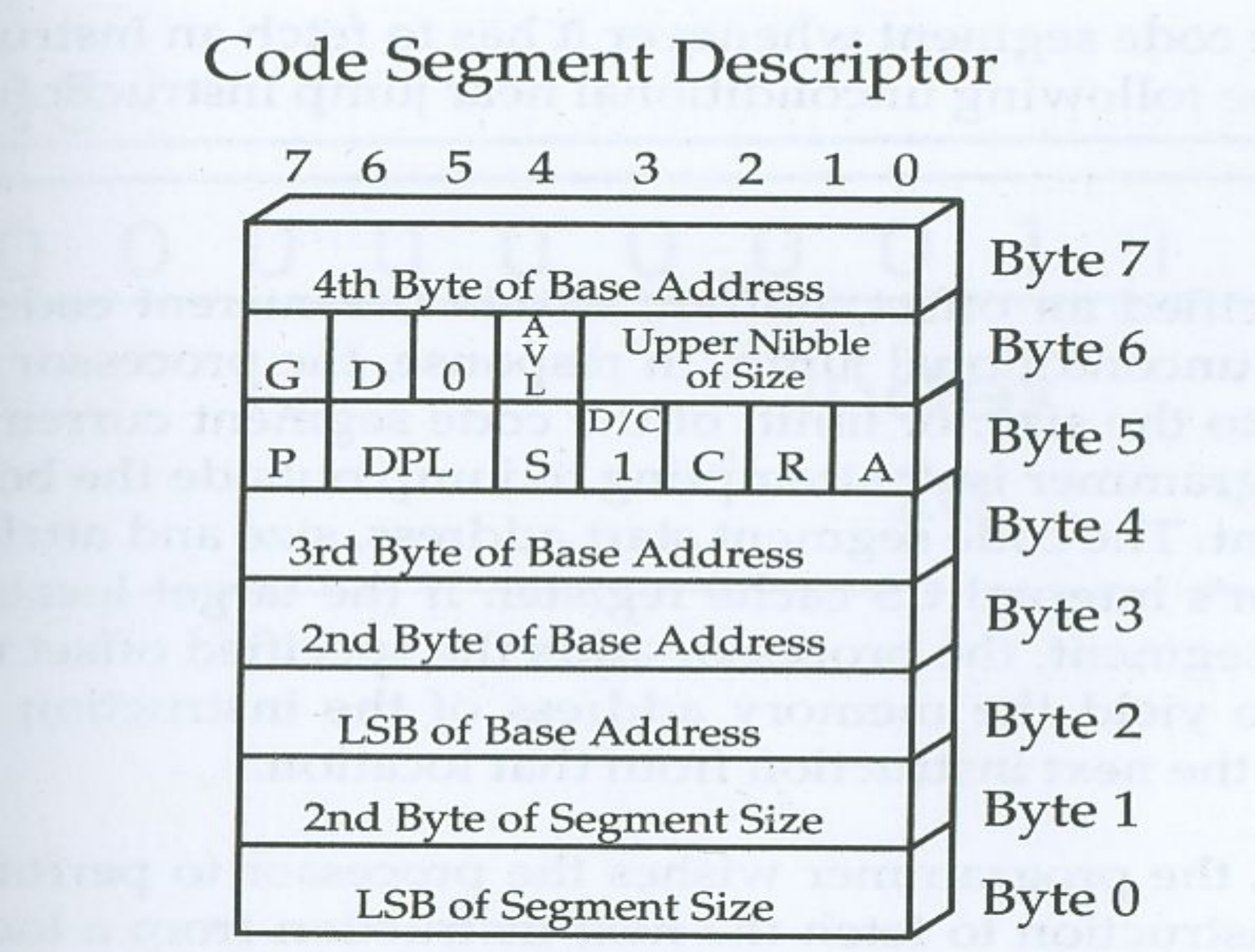
Logical Address to linear address Translation



General Format for Descriptor table Entry



7	6	5	4	3	2	1	0	
4th Byte of Base Address								Byte 7
G	D/B	0	AVL	Upper Nibble of Size				Byte 6
P	DPL		S	Segment Type				Byte 5
3rd Byte of Base Address								Byte 4
2nd Byte of Base Address								Byte 3
LSB of Base Address								Byte 2
2nd Byte of Segment Size								Byte 1
LSB of Segment Size								Byte 0



Code Segment Descriptor

	7	6	5	4	3	2	1	0
Byte 7	4th Byte of Base Address							
Byte 6	G	D	0	AVL	Upper Nibble of Size			
Byte 5	P	DPL		S	D/C 1	C	R	A
Byte 4	3rd Byte of Base Address							
Byte 3	2nd Byte of Base Address							
Byte 2	LSB of Base Address							
Byte 1	2nd Byte of Segment Size							
Byte 0	LSB of Segment Size							

G Bit
Granularity bit defines meaning of limit value.
0 = length of segment in bytes.
1 = length of segment in 4KB pages.

D Bit
In code segment, Default bit defines default size of operands and effective addresses. 0 = 16-bit, 1 = 32-bit.

AVL Bit
Available for use by system software

P Bit
Segment Present bit (must be 1 if the code segment is present in memory).

DPL Field
Descriptor Privilege Level (0-3)

S Bit
System bit. When 0, indicates system segment.
Must be 1 in a code segment descriptor.

D/C
This could be called the Data/Code bit.
A 0 indicates a data segment and a 1 indicates a code segment.

C Bit
Conforming bit. Set to 1 if code segment is conforming. See text for a detailed description.

R Bit
Readable bit. A 0 indicates an execute-only segment, while a 1 indicates the segment may be read from by both the prefetcher and for data accesses.

A Bit
Accessed bit. Set to 1 by the processor when a code segment is accessed.

Conforming / non conforming code segment

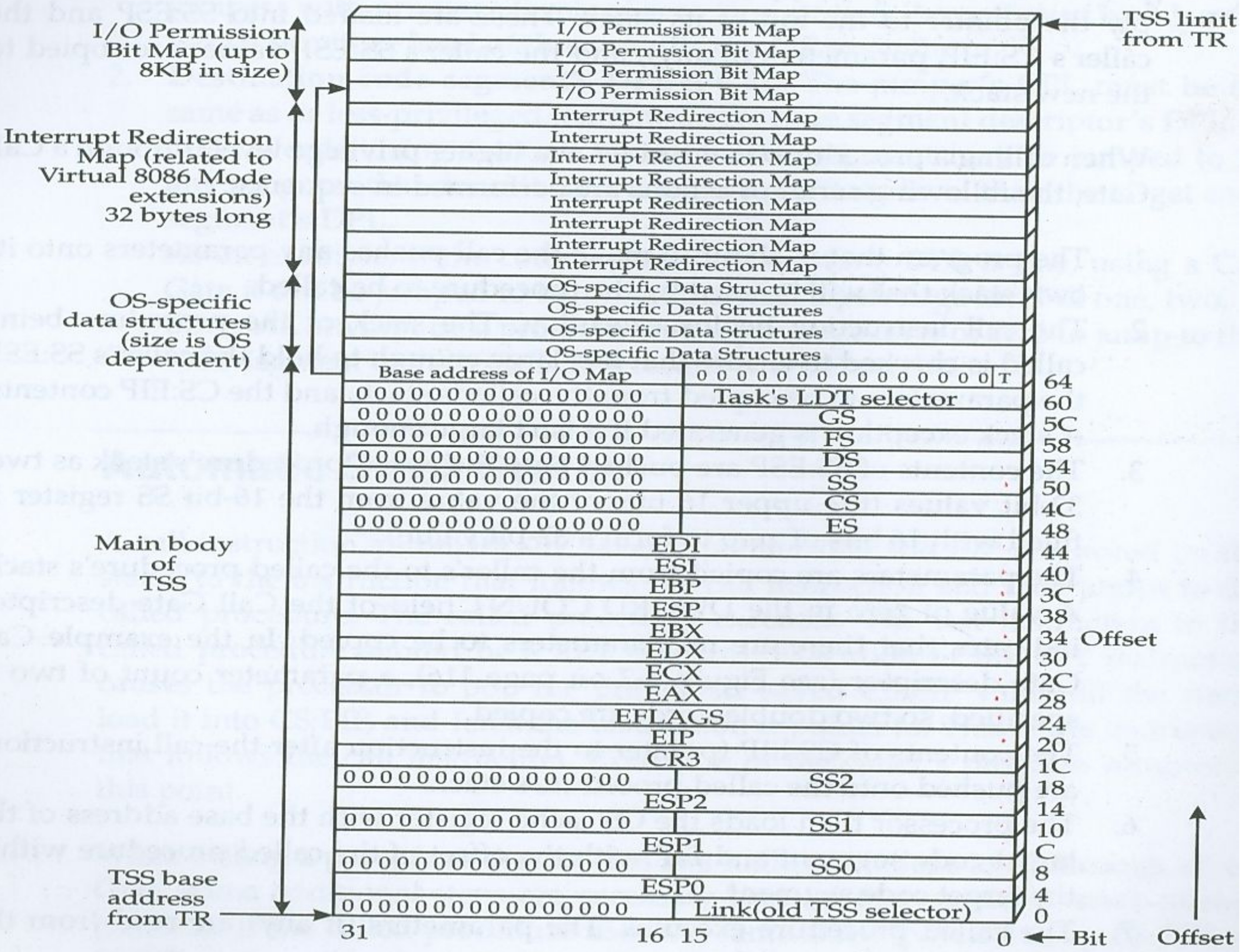
- C= 0 Non conforming
  - Non conforming code segment can only be accessed by programs whose CPL matches the target code DPL
- C=1 Conforming code
  - Conforming code can be accessed by program whose CPL > Accessed code DPL
  - Accessed code runs at privilege of calling program

How a new code segment is accessed?

- Execution of far jump
- Execution of far call
- Hardware interrupt or software exception
- Initiation/resumption of new/ existing task
- Execution of far Return
- Execution of IRET

Context /task switch

- On expiry of time slice, hardware interrupt is generated
- On interrupt, control is returned to the OS
  - OS takes the snap shot of the executing process and save it in current task’s TSS
  - OS determines the next task to be executed, loads the context of new task from its TSS and starts execution

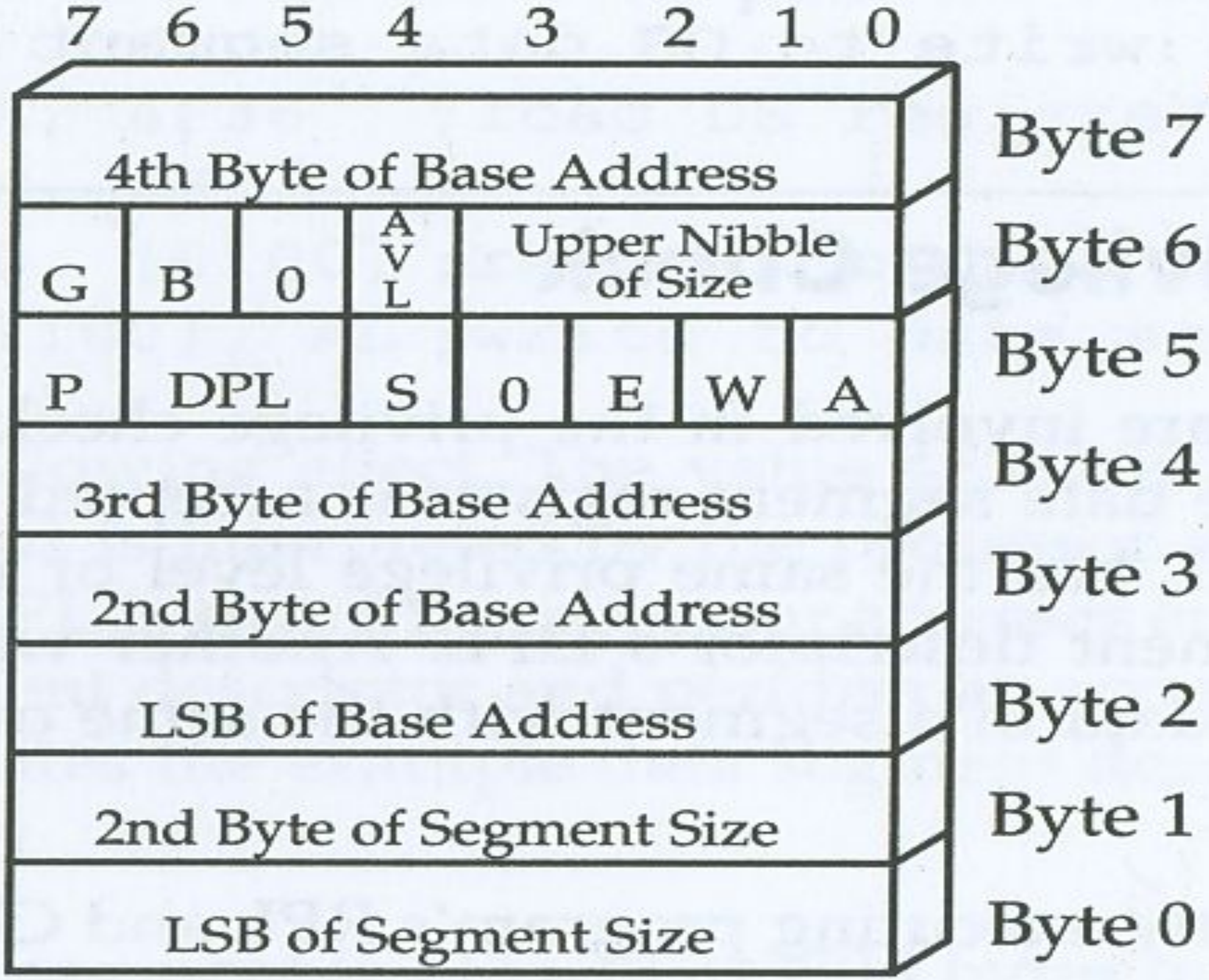

I/O Permission Bit Map (up to 8KB in size)
Interrupt Redirection Map (related to Virtual 8086 Mode extensions) 32 bytes long
OS-specific data structures (size is OS dependent)
Main body of TSS
TSS base address from TR
TSS limit from TR
31
16 15
0 ← Bit
Offset

31–16	15–0	Offset
I/O Permission Bit Map		
I/O Permission Bit Map		
I/O Permission Bit Map		
I/O Permission Bit Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
Interrupt Redirection Map		
OS-specific Data Structures		
OS-specific Data Structures		
OS-specific Data Structures		
OS-specific Data Structures		
Base address of I/O Map	000000000000000 T	64
0000000000000000	Task's LDT selector	60
0000000000000000	GS	5C
0000000000000000	FS	58
0000000000000000	DS	54
0000000000000000	SS	50
0000000000000000	CS	4C
0000000000000000	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34 Offset
EDX		30
ECX		2C
EAX		28
EFLAGS		24
EIP		20
CR3		1C
0000000000000000	SS2	18
ESP2		14
0000000000000000	SS1	10
ESP1		C
0000000000000000	SS0	8
ESP0		4
0000000000000000	Link(old TSS selector)	0

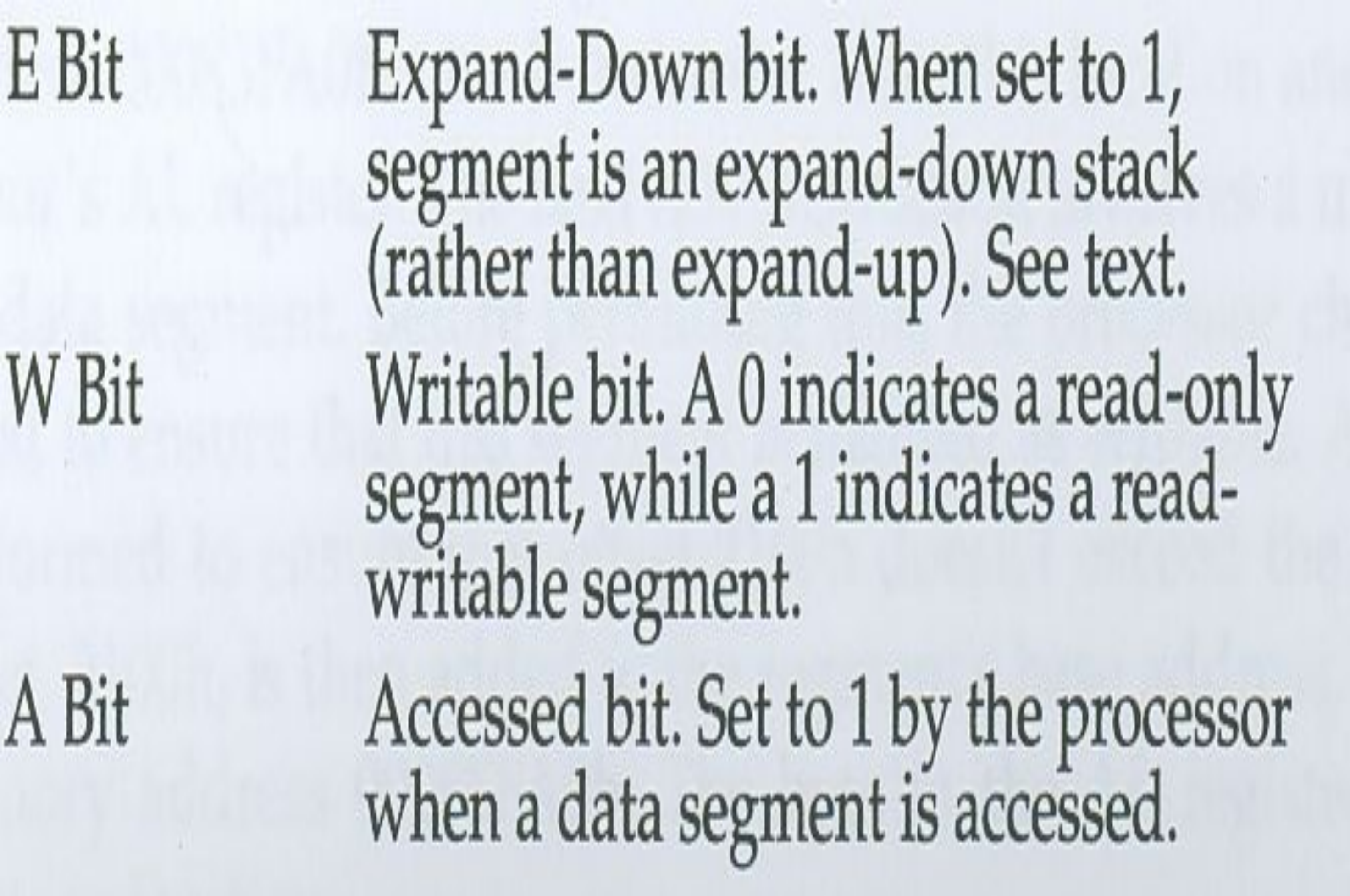
Role of RPL, CPL, DPL

- When currently executing program executes a far jump
  - DPL is compared with max(CPL, RPL)

Data Segment Descriptor



7	6	5	4	3	2	1	0	
4th Byte of Base Address								Byte 7
G	B	0	AVL	Upper Nibble of Size				Byte 6
P	DPL		S	0	E	W	A	Byte 5
3rd Byte of Base Address								Byte 4
2nd Byte of Base Address								Byte 3
LSB of Base Address								Byte 2
2nd Byte of Segment Size								Byte 1
LSB of Segment Size								Byte 0



E Bit Expand-Down bit. When set to 1, segment is an expand-down stack (rather than expand-up). See text.

W Bit Writable bit. A 0 indicates a read-only segment, while a 1 indicates a read-writable segment.

A Bit Accessed bit. Set to 1 by the processor when a data segment is accessed.

Expand UP/Down

- Expand up
  - Lowest address (BOS)= Base address
  - Highest Address (TOS)= Base +limit
- Expand down
  - Lowest address (BOS)= Base +limit-1
  - Highest Address (TOS)= Base + FFFFh or FFFF FFFFh

Privilege check in CS : Issues

- Assume OS has a code segment with privilege level 0
- It has number of procedures within this segments
- Some of these procedures should only be accessed by OS code residing at privilege 1 or 2
- Some of the procedures can be accessed by application at privilege 3

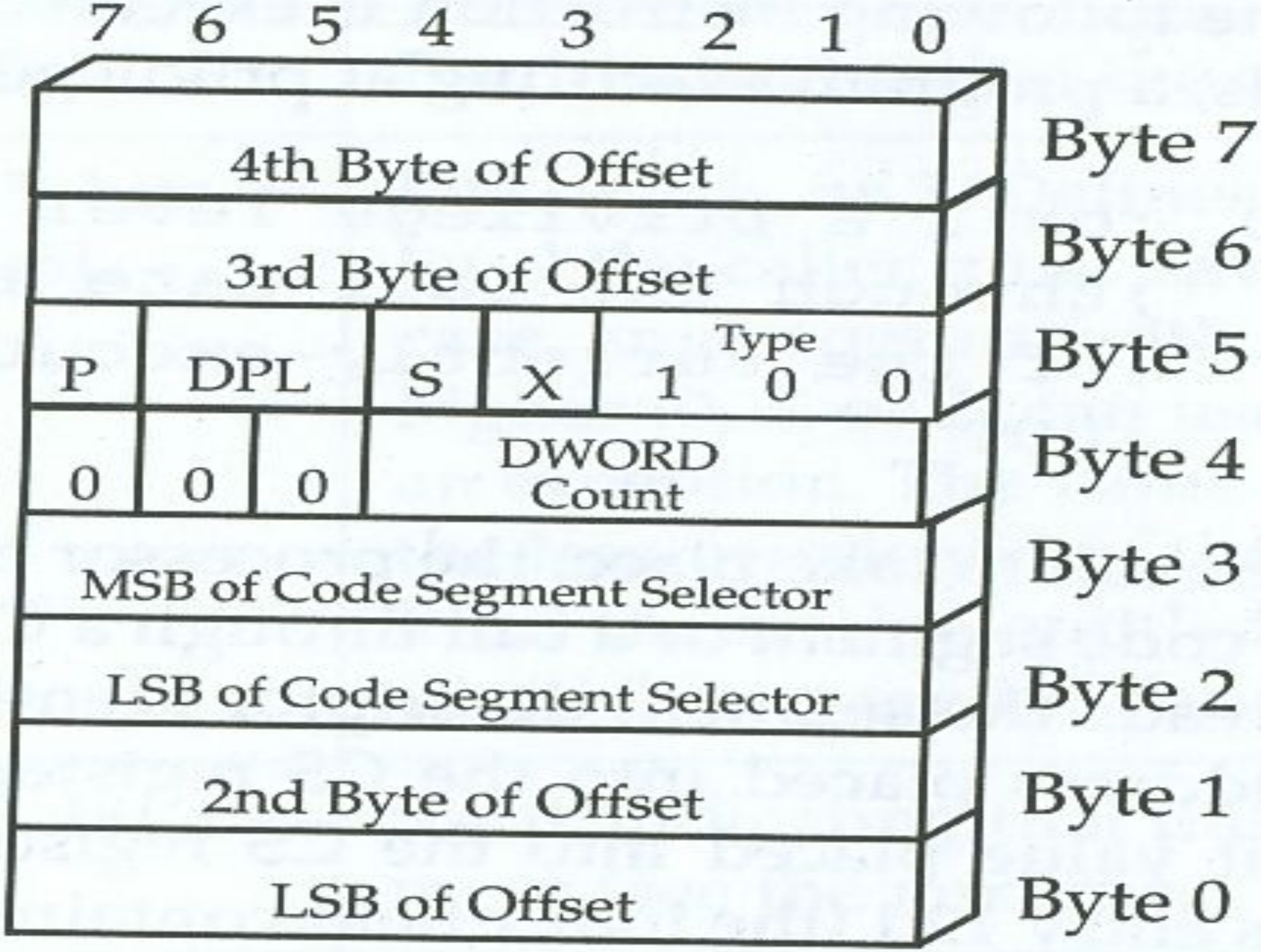
Call Gate

- Call gate is special OS descriptor
  - May reside in GDT or LDT
  - It is used for transferring control to a procedure whose privilege level is equal or greater than calling program
  - Far call instruction can use call gate to transfer control to procedure with a higher privilege level
  - Far jump instruction can use call gate to transfer control to procedure with same privilege level or to a conforming code segment

Call gate

- It contains indirect pointer to code segment and entry point
- When Far call is executed
  - CS selector identifies the GDT/ LDT entry containing call gate
  - Offset of the far call is discarded . The actual offset is obtained from call gate

Call Gate Descriptor Format

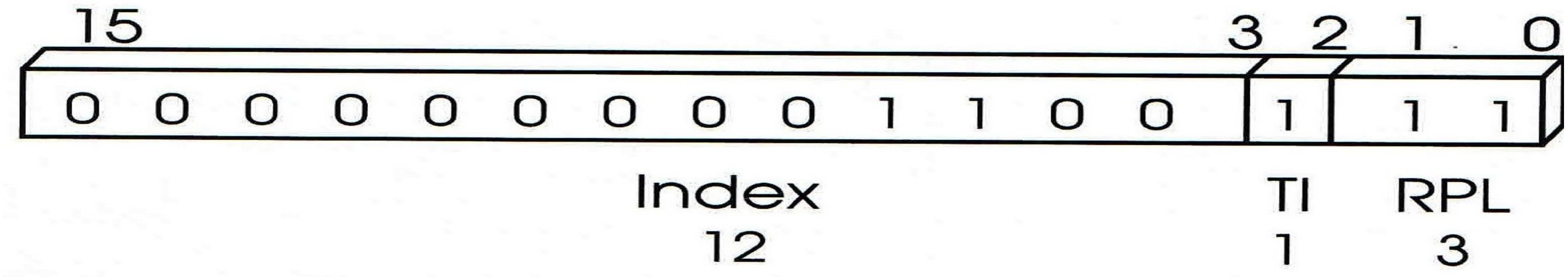


7	6	5	4	3	2	1	0	
4th Byte of Offset								Byte 7
3rd Byte of Offset								Byte 6
P	DPL		S	X	Type 1	0	0	Byte 5
0	0	0	DWORD Count					Byte 4
MSB of Code Segment Selector								Byte 3
LSB of Code Segment Selector								Byte 2
2nd Byte of Offset								Byte 1
LSB of Offset								Byte 0

- X =0 16 bit call gate
- X=1 32 bit call gate
- Dword count : number of dwords to be copied from caller stack to called stack
- DPL defines minimum privilege level caller must have to use the gate
- If DPL is 3 then any program with privilage level 3 or higher (0,1,2) can use gate

Call gate example

- Call 0067h:0000h

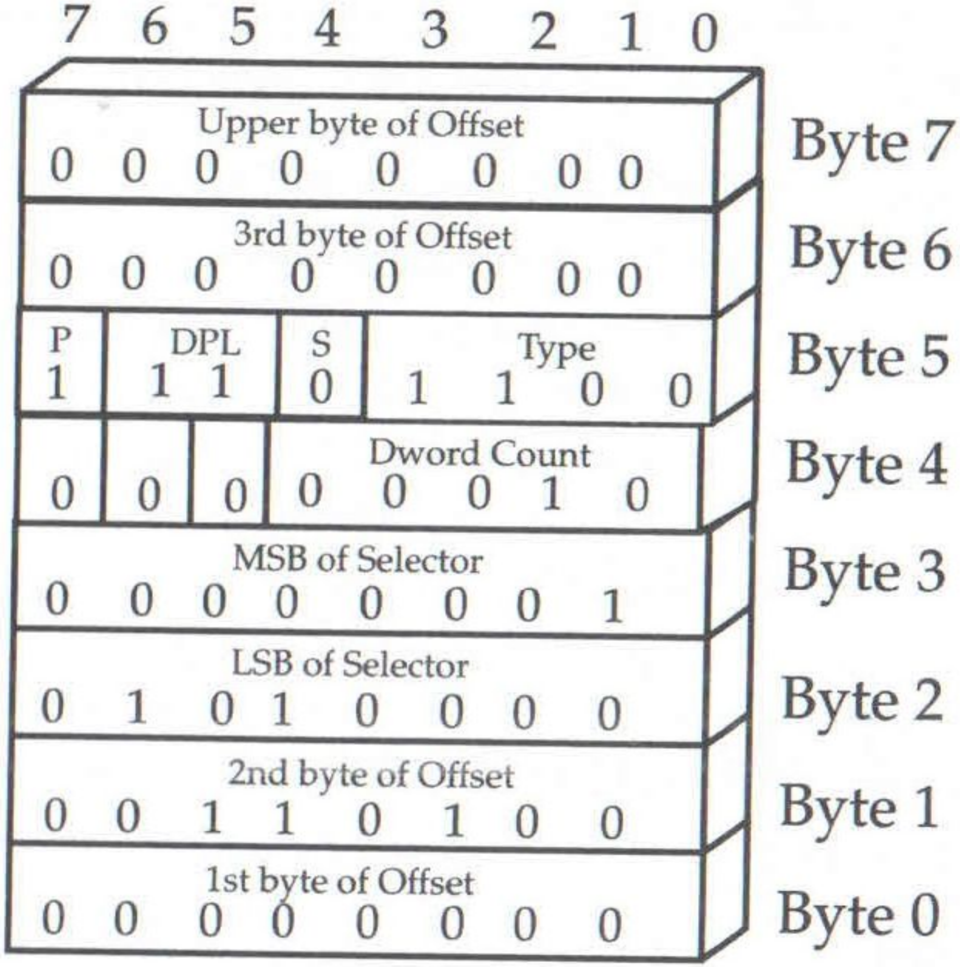


15 3 2 1 0

0 0 0 0 0 0 0 0 0 1 1 0 0 | 1 | 1 1

Index 12 | TI 1 | RPL 3

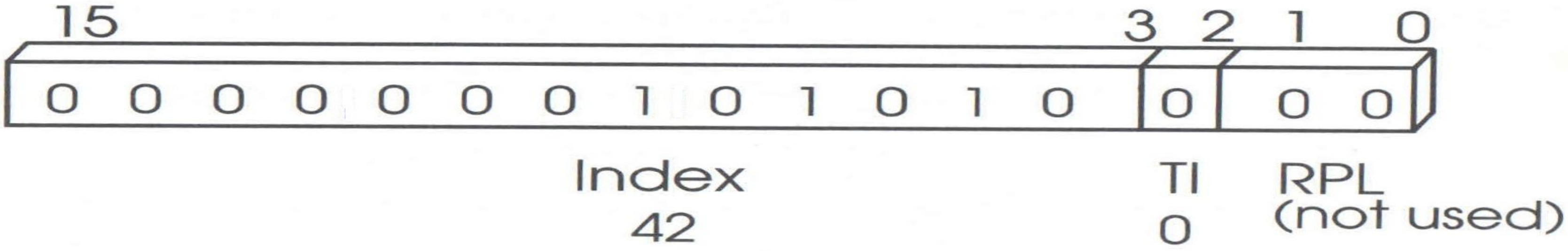
Call gate descriptor



Field	7	6	5	4	3	2	1	0	
Upper byte of Offset	0	0	0	0	0	0	0	0	Byte 7
3rd byte of Offset	0	0	0	0	0	0	0	0	Byte 6
P / DPL / S / Type	P 1	DPL 1	1	S 0	Type 1	1	0	0	Byte 5
Dword Count	0	0	0	0	0	0	1	0	Byte 4
MSB of Selector	0	0	0	0	0	0	0	1	Byte 3
LSB of Selector	0	1	0	1	0	0	0	0	Byte 2
2nd byte of Offset	0	0	1	1	0	1	0	0	Byte 1
1st byte of Offset	0	0	0	0	0	0	0	0	Byte 0

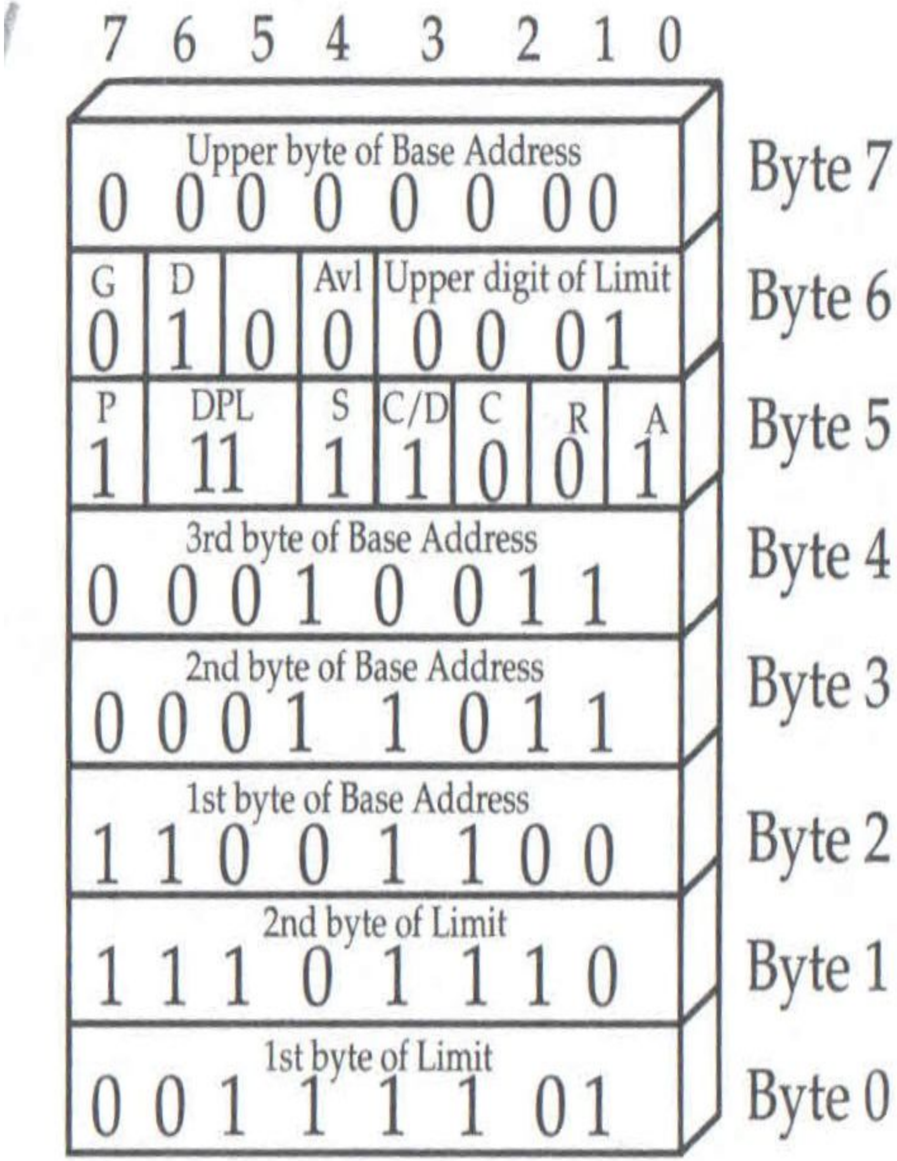
- Offset 3400h
- Selector 0150h
- D word count 2

Call gate selector



- Non conforming code
- Base address :00131bcch
- Size 1ee3dh
- R=0 execute only
- DPL 3

Code segment descriptor

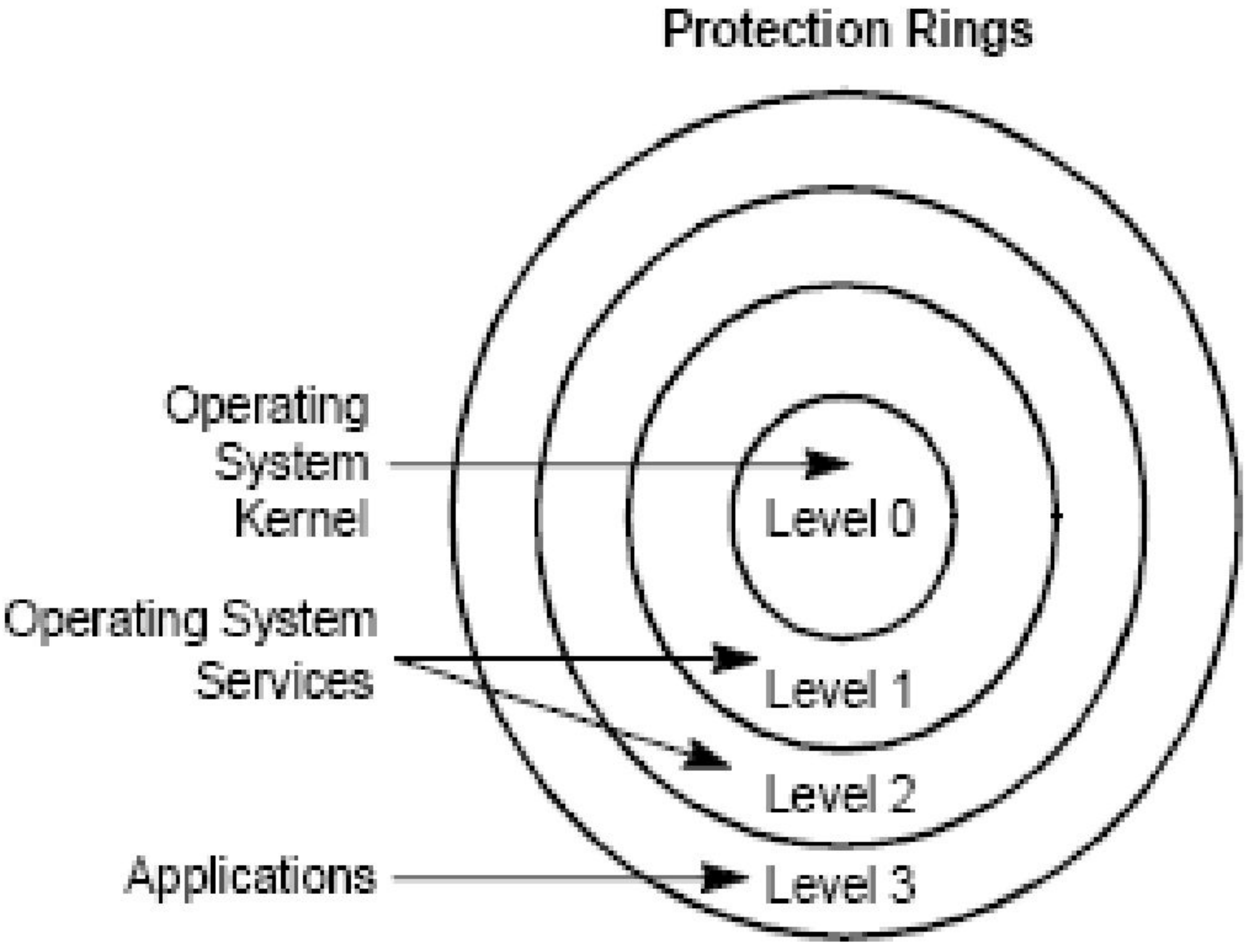


7	6	5	4	3	2	1	0	
Upper byte of Base Address								
0	0	0	0	0	0	0	0	Byte 7
G	D		Avl	Upper digit of Limit				
0	1	0	0	0	0	0	1	Byte 6
P	DPL		S	C/D	C	R	A	
1	11		1	1	0	0	1	Byte 5
3rd byte of Base Address								
0	0	0	1	0	0	1	1	Byte 4
2nd byte of Base Address								
0	0	0	1	1	0	1	1	Byte 3
1st byte of Base Address								
1	1	0	0	1	1	0	0	Byte 2
2nd byte of Limit								
1	1	1	0	1	1	1	0	Byte 1
1st byte of Limit								
0	0	1	1	1	1	0	1	Byte 0

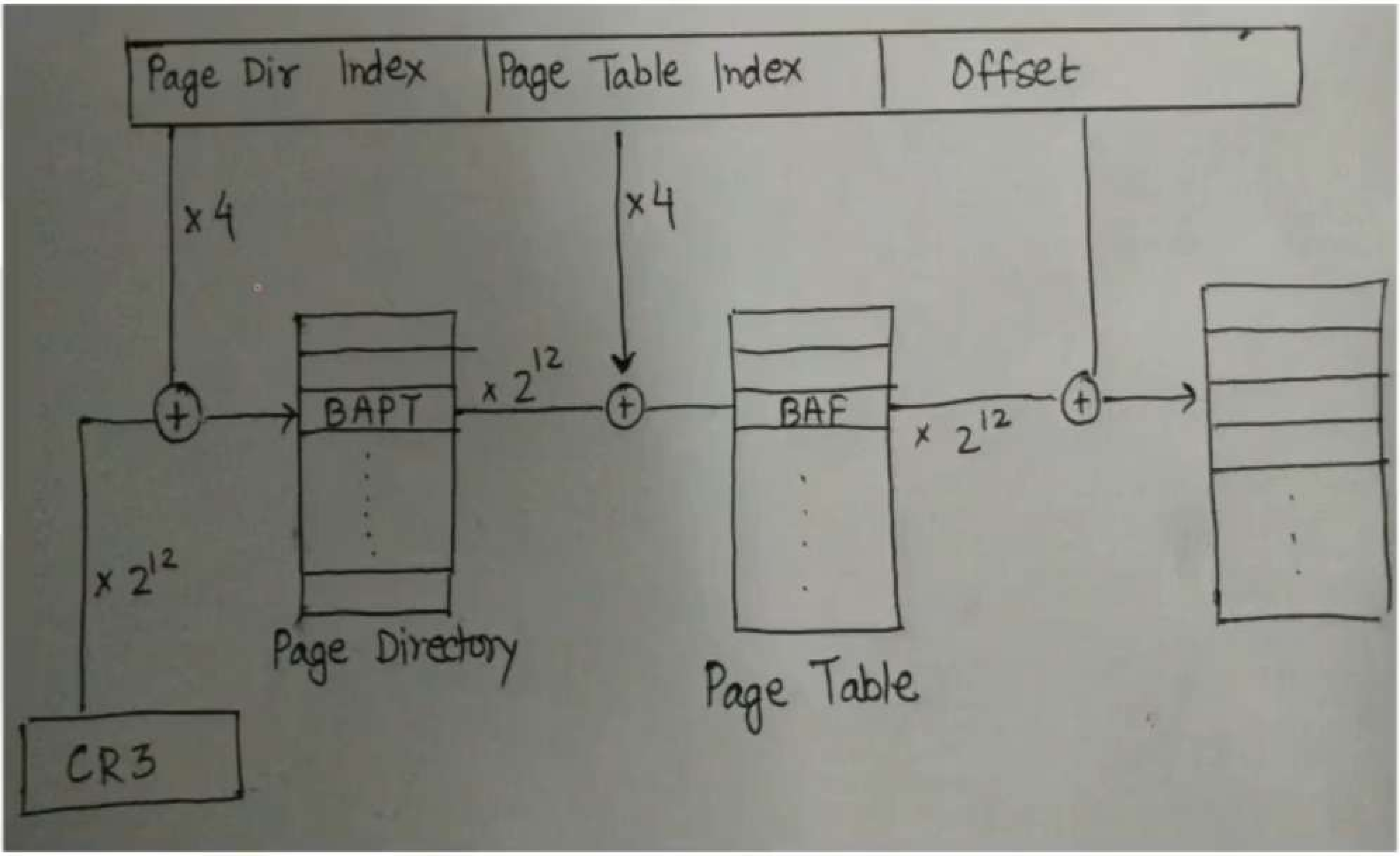
Call gate privilege check using Far call

- Call gate privilege check using Far call
  - Numerically greater of CPL and RPL <= call gate DPL
  - DPL of destination code segment <= CPL
- Call gate privilege check using Far jump
  - Numerically greater of CPL and RPL <= call gate DPL
  - DPL of destination code segment = CPL

Protection Ring



Linear Address Translation



TLB

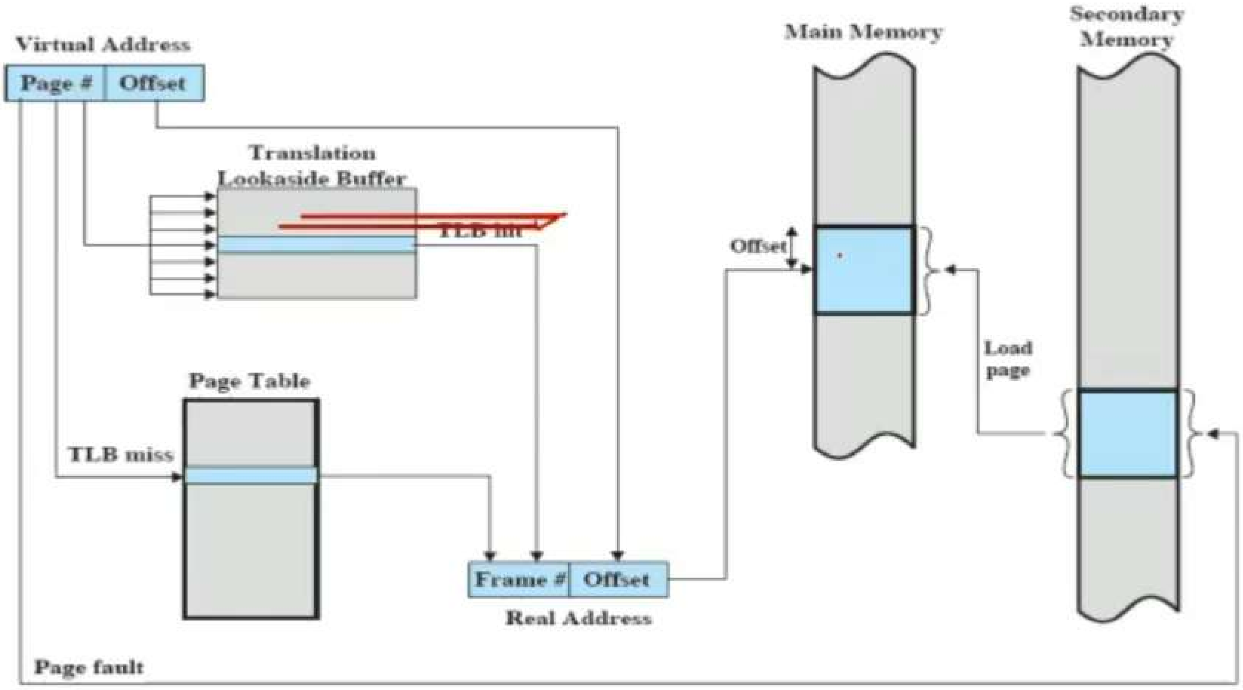


Figure 8.7 Use of a Translation Lookaside Buffer

TLB Operation

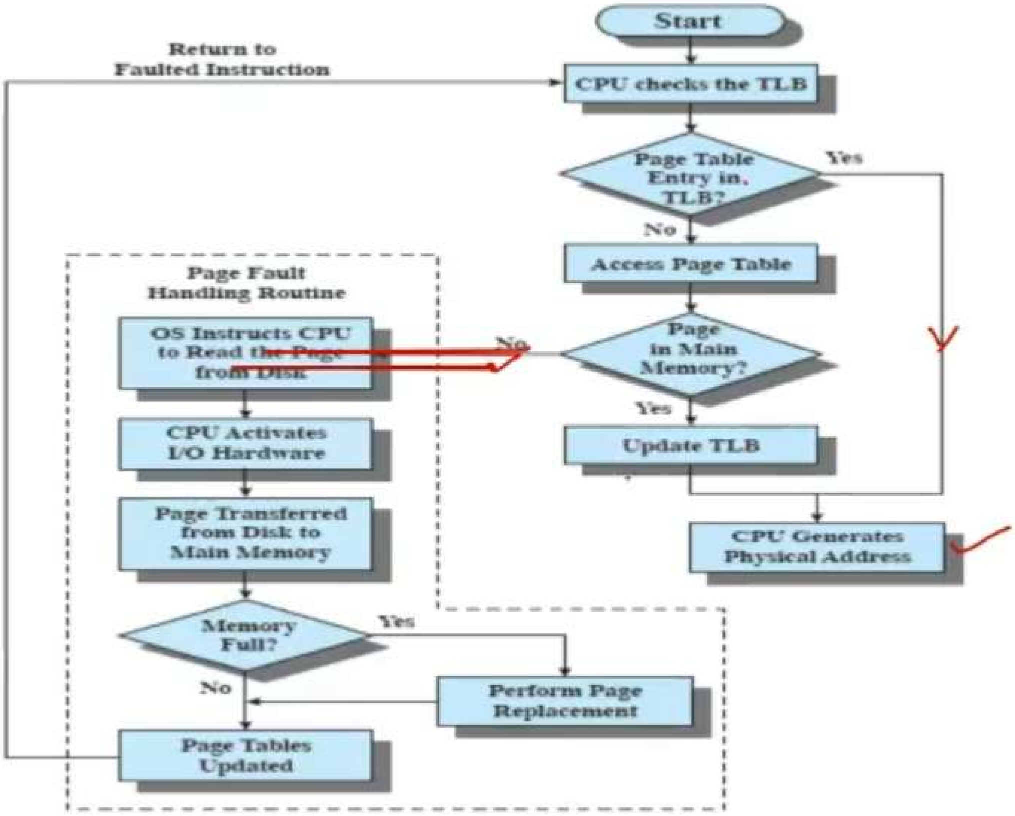

Start
CPU checks the TLB
Return to Faulted Instruction
Page Table Entry in TLB?
Yes
No
Access Page Table
Page in Main Memory?
No
Yes
Update TLB
CPU Generates Physical Address
Page Fault Handling Routine
OS Instructs CPU to Read the Page from Disk
CPU Activates I/O Hardware
Page Transferred from Disk to Main Memory
Memory Full?
Yes
No
Perform Page Replacement
Page Tables Updated

Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]