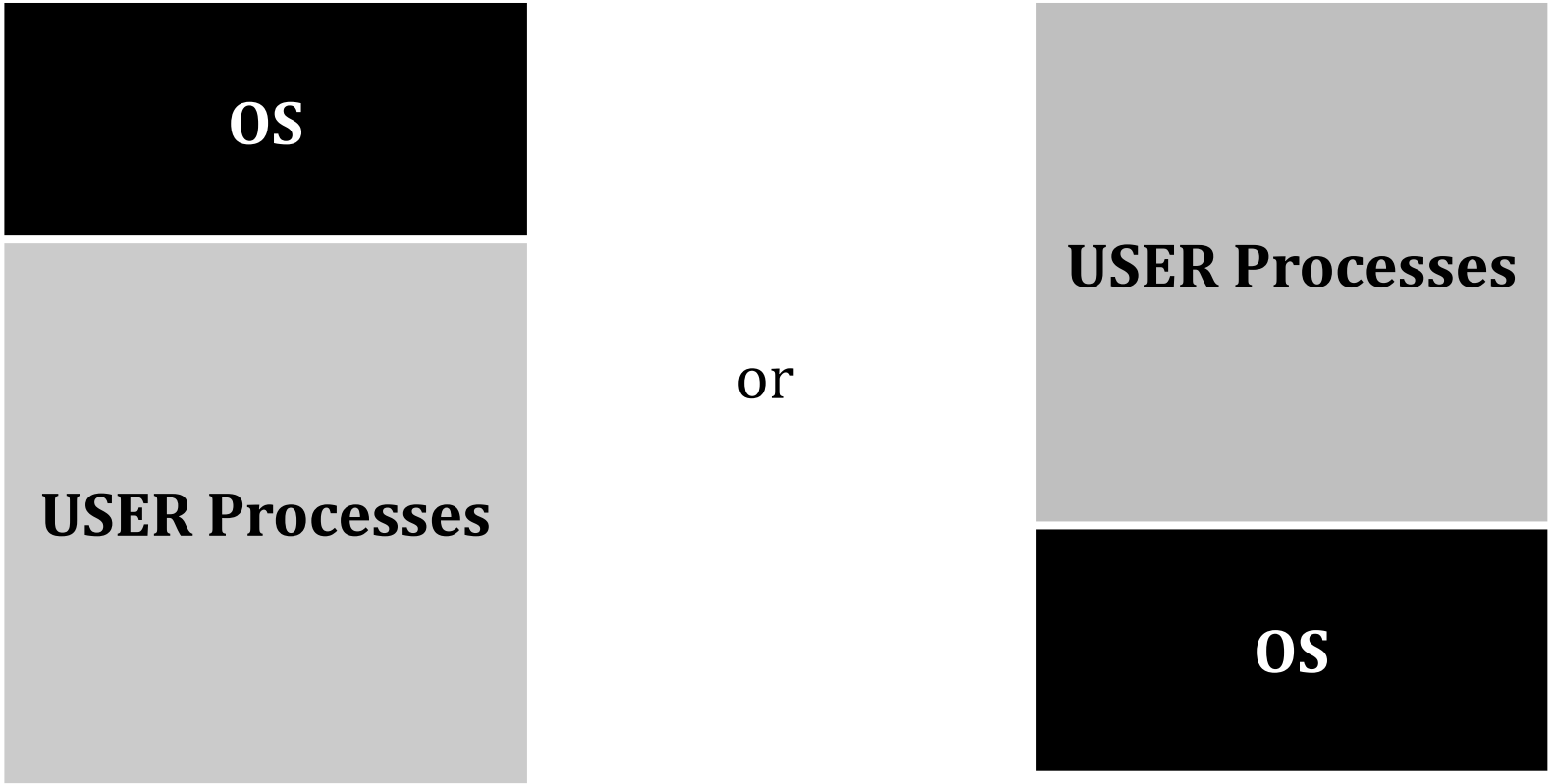
Memory Management

Dr. Manmath N. Sahoo
Dept. of CSE, NIT Rourkela

OS and User Area

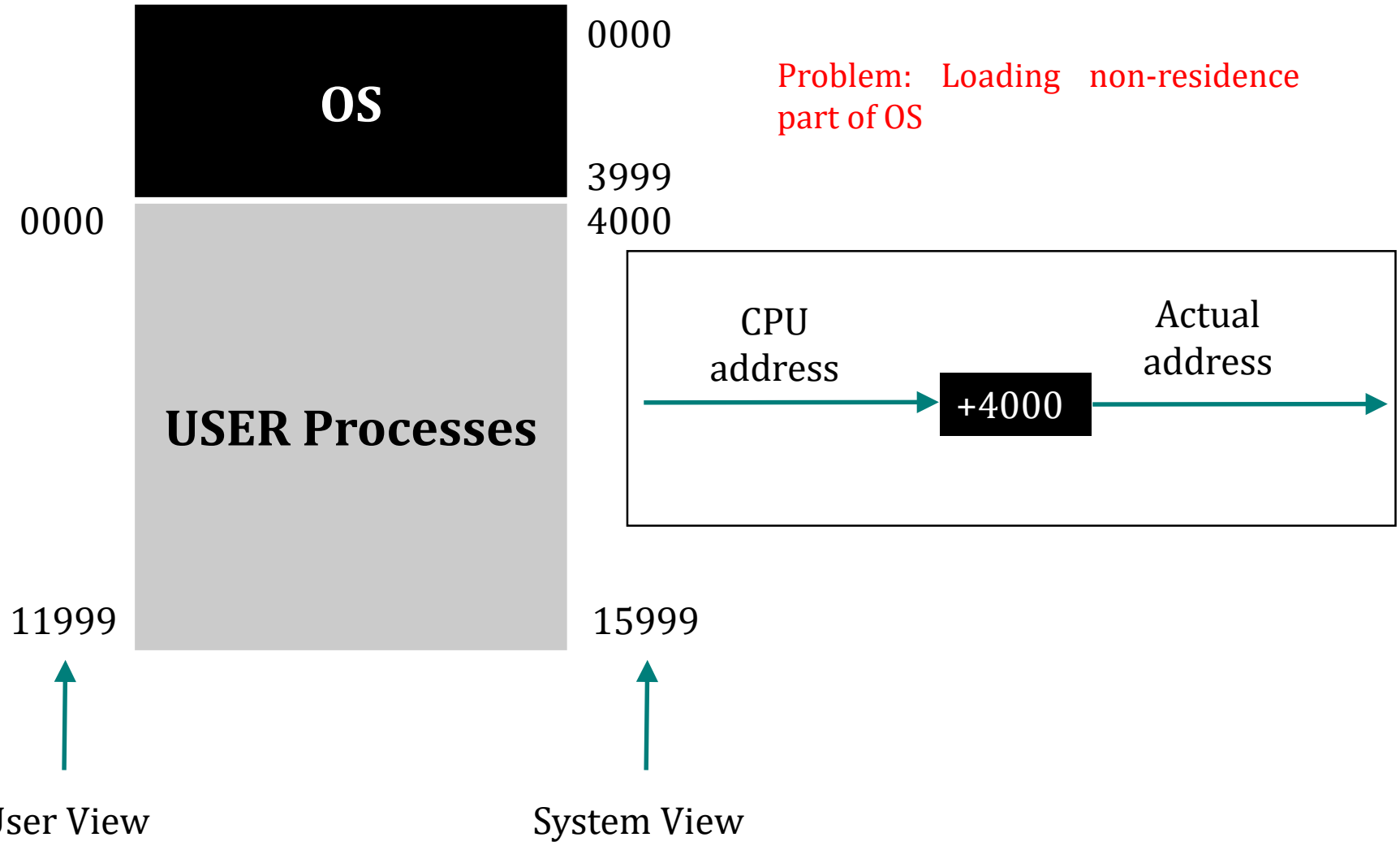


Protection of OS area from users

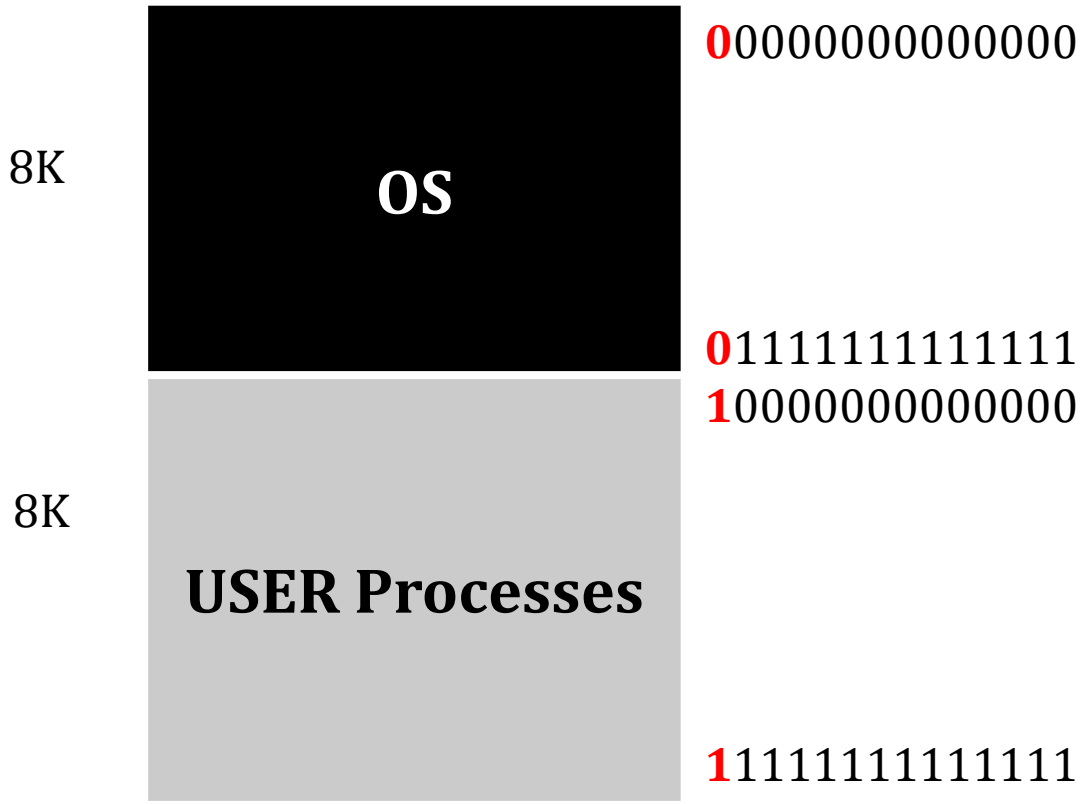
3 Schemes

- Automatic address translation
- Page-0 and Page-1 addressing
- Limit register

Automatic address translation



Page-0 and Page-1 addressing

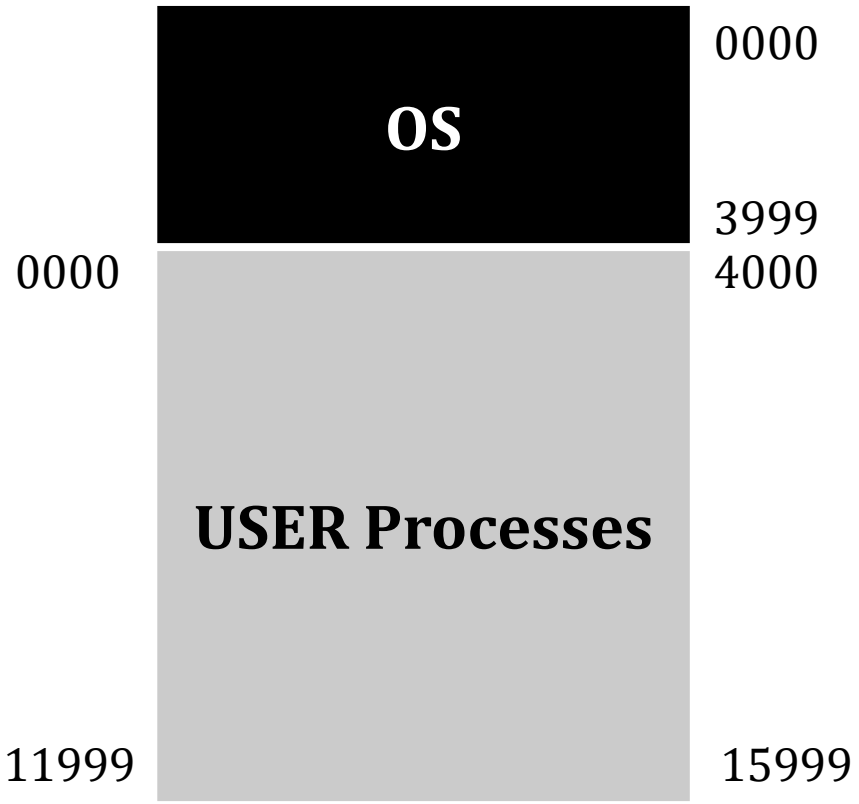


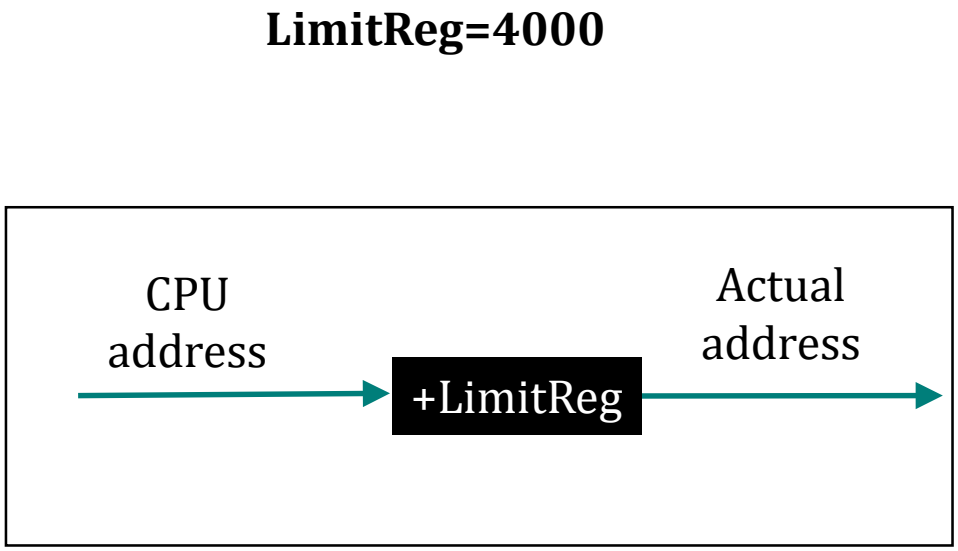
Problems:

- Loading non-residence part of OS, may exceed page-0
- If OS is small then memory wastage

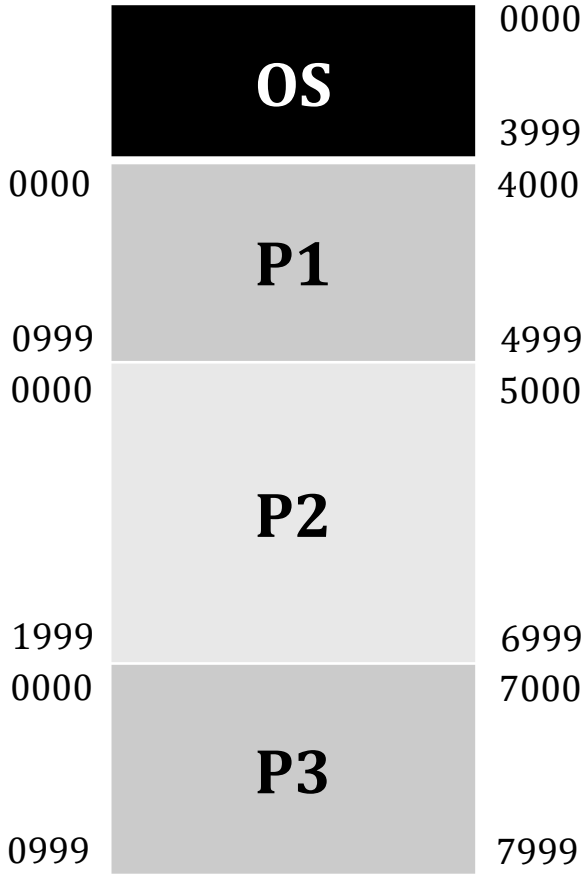
Limit Register

- Limit register stores the size of the OS





Interprocess protection

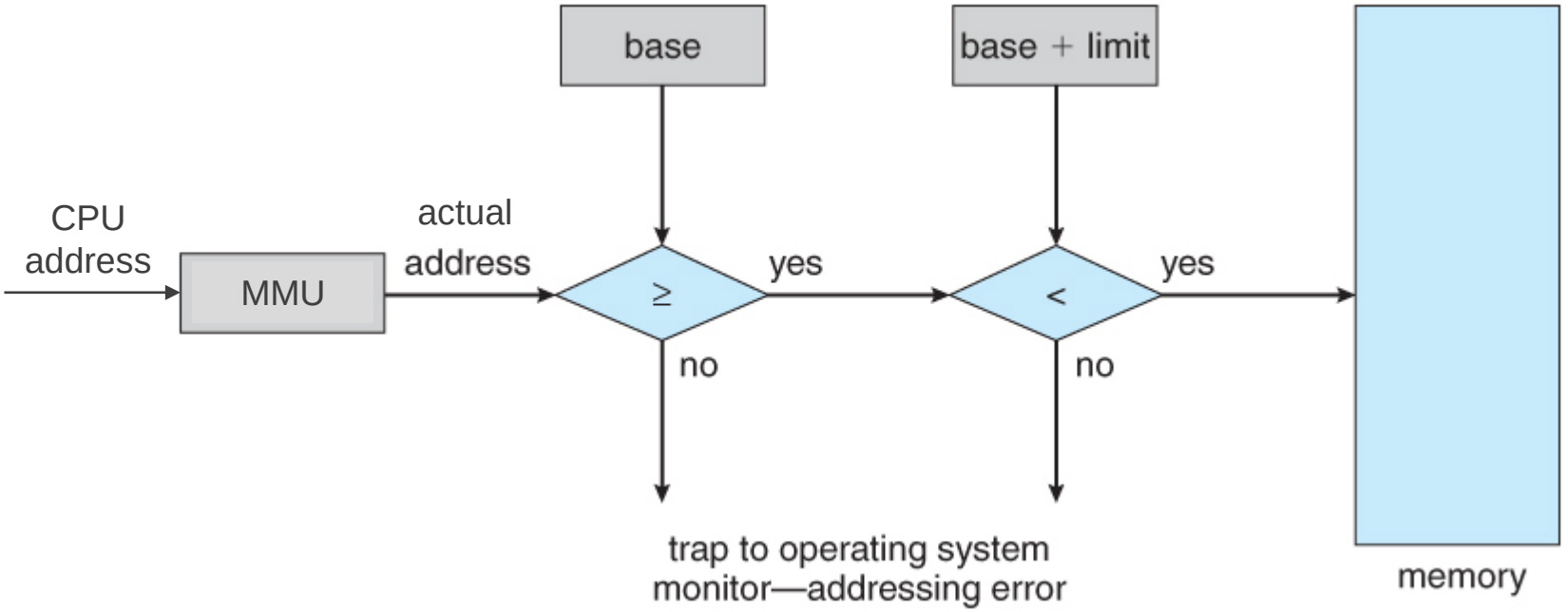


Suppose P2 is running

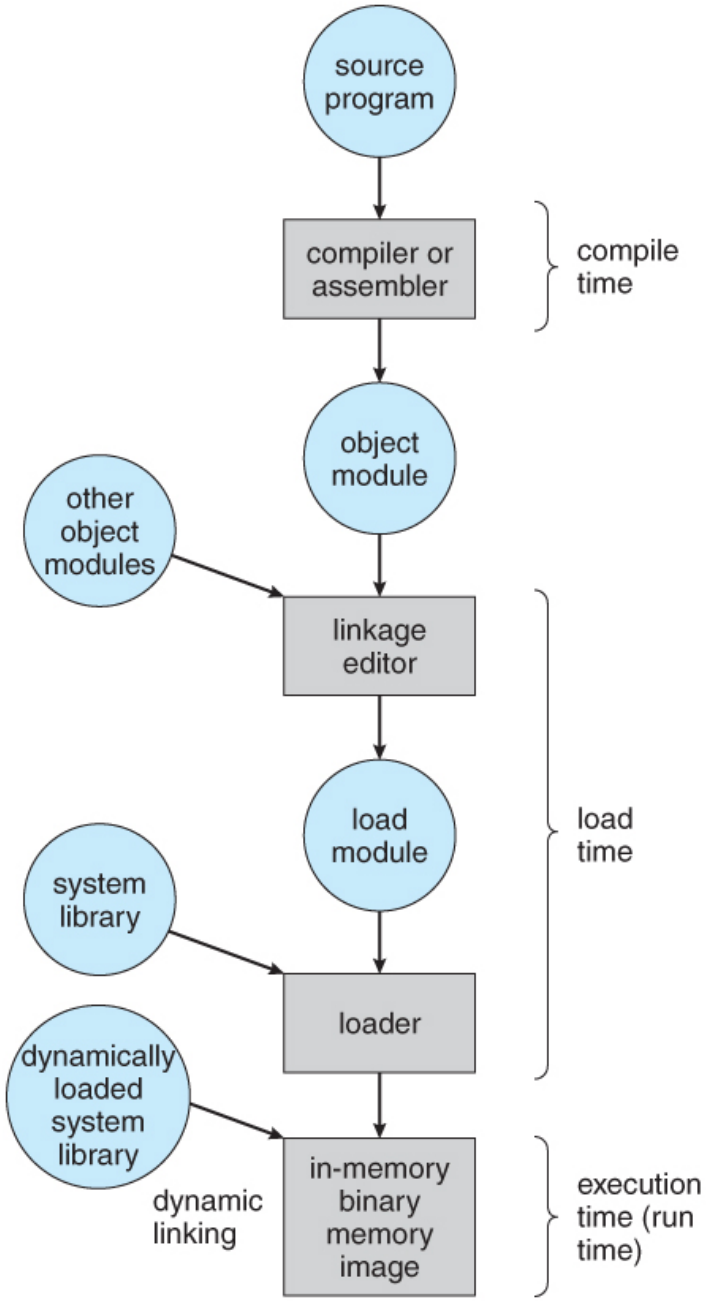
Base=5000 //base address of P2

Limit=2000 //size of P2

Interprocess protection



Multistep processing of a user program



Address binding

- Program must be brought into memory and converted into a process for it to be executed.
- During execution processes make references to physical memory locations.
- Binding of instructions/data to physical memory is known as address binding.
- Address binding can happen at three different stages.

Binding of Instructions/Data to Memory

Compile time:

- If the location where the process is to be placed in memory is known at the compilation time, then compiler can replace symbolic references to data and program instructions with actual addresses in memory where that data is to be held.
  - You must recompile code if location of process in memory is to change.

Binding of Instructions/Data to Memory

Load time:

- If the location where the process is to be placed in memory is NOT known when the program is compiled - compiler assigns ***relative address*** to data and code.
- The actual base address is added to all addresses when program is loaded into memory.
  - If starting address changes then the process then add the new starting address to all the relative addresses.

Binding of Instructions/Data to Memory

Load time binding without runtime check:

- Advantage: No address translation overhead during running
- Problem: total memory requirement of a process needs to be known a-priori
- Problem: Process cannot be moved during execution
- Problem: Rogue process can still overwrite other process's memory by writing out of bounds, no runtime check

Binding of Instructions/Data to Memory

Load time binding with runtime check:

- Address bound at load time, but checked at run time if within bound
- Solves the problem of overwriting other process's memory, but increases cost of access

Binding of Instructions/Data to Memory

Execution time binding:

- The address binding is done at execution time (run time binding).
- Advantage:
  - Processes can be moved during execution,
  - Protects one process from another,
  - Can grow process' memory at run time
- Problem: Address translation overhead at run time

Logical address space vs Physical address space

- CPU generates logical/virtual addresses for instructions and data
- Logical addresses are converted/mapped to physical addresses by Memory Management Unit (MMU) – hardware unit.

Memory allocation schemes

- Contiguous
  - Each process allocated a single contiguous chunk of memory
    - Fixed equal-size partitions
    - Fixed unequal-size partitions
    - Dynamic partitioning
- Noncontiguous
  - Parts of a process can be allocated non-contiguous chunks of memory

Fixed equal-size partitions

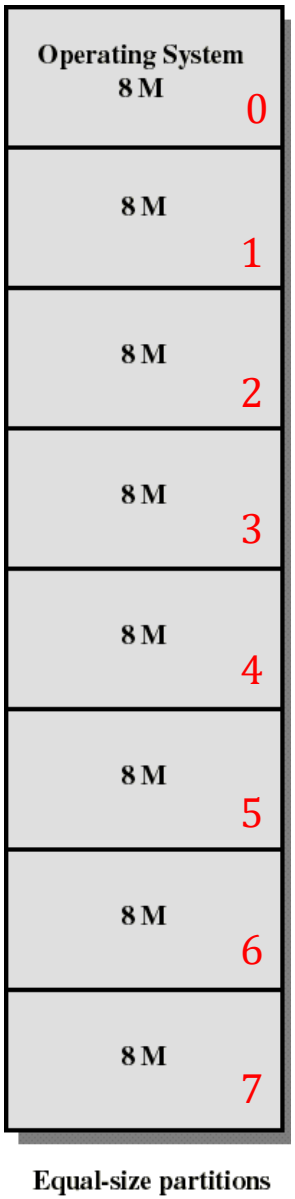
- If there is an available partition, a process can be loaded into that partition
- because all partitions are of equal size, it does not matter which partition is used.

MAT

Partition#	Process#

MFT

Partition#	Availability
0	0
1	1
2	1
3	1
4	1
5	1
6	1
7	1



Equal-size partitions

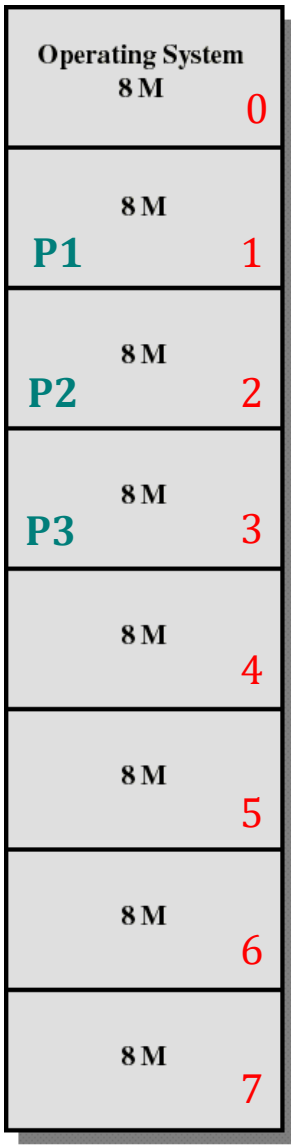
Fixed equal-size partitions

MAT

Partition#	Process#
1	P1
2	P2
3	P3

MFT

Partition#	Availability
0	0
1	0
2	0
3	0
4	1
5	1
6	1
7	1



Equal-size partitions

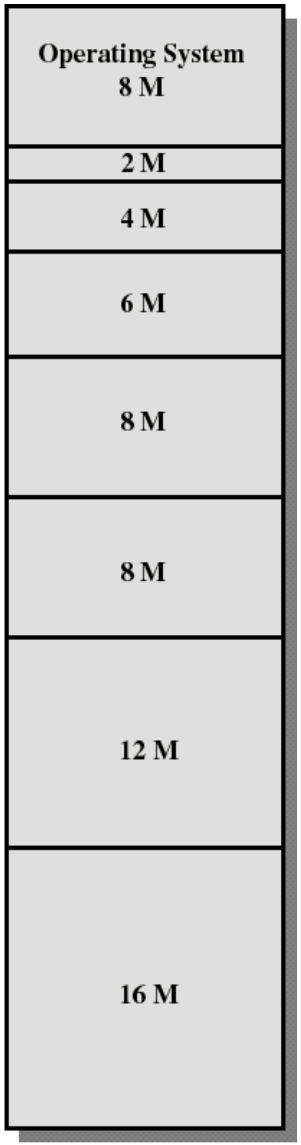
Fixed Unequal-size partitions

MAT

Partition#	Process#

MFT

Partition#	Availability
0	0
1	1
2	1
3	1
4	1
5	1
6	1
7	1

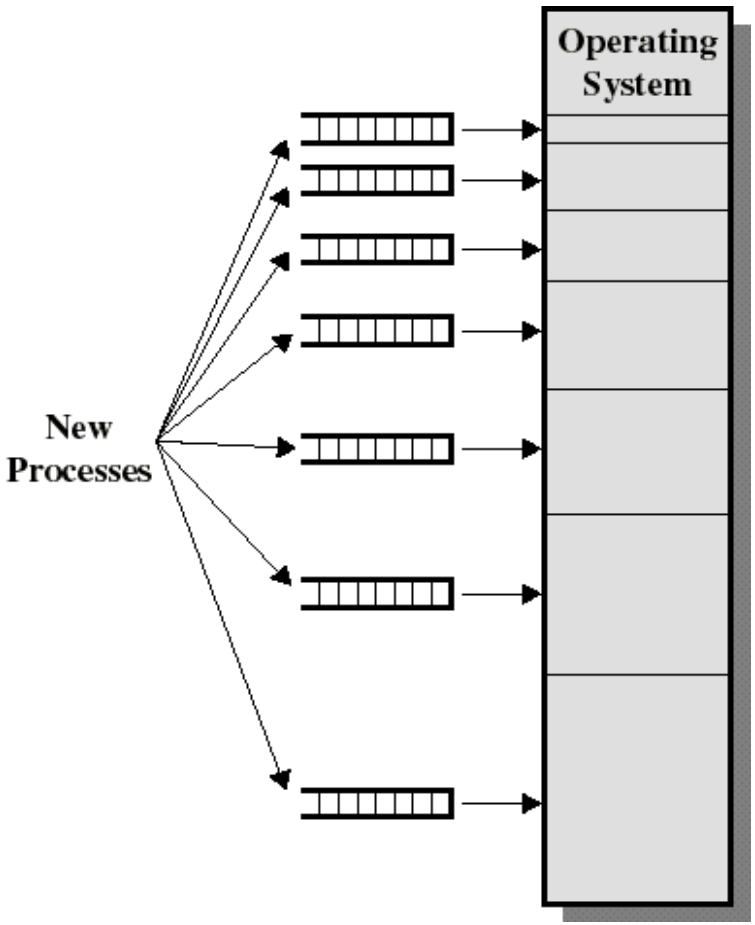


Unequal-size partitions

Placement Algorithm with Unequal-size Partitions

Unequal-size partitions, use of multiple queues:

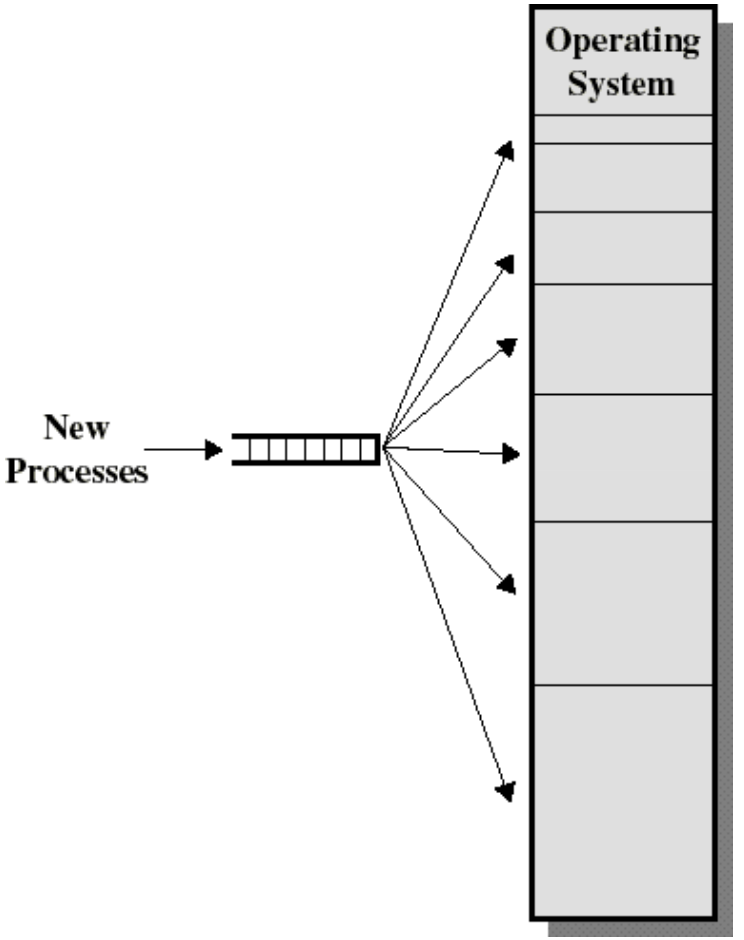
- a queue exists for each partition size.
- assign each process to the smallest partition within which it will fit.
- tries to minimize internal fragmentation.
- problem: some queues might be empty while some might be loaded. – decreases degree of multiprogramming



Placement Algorithm with Unequal-size Partitions

Unequal-size partitions, use of a single queue:

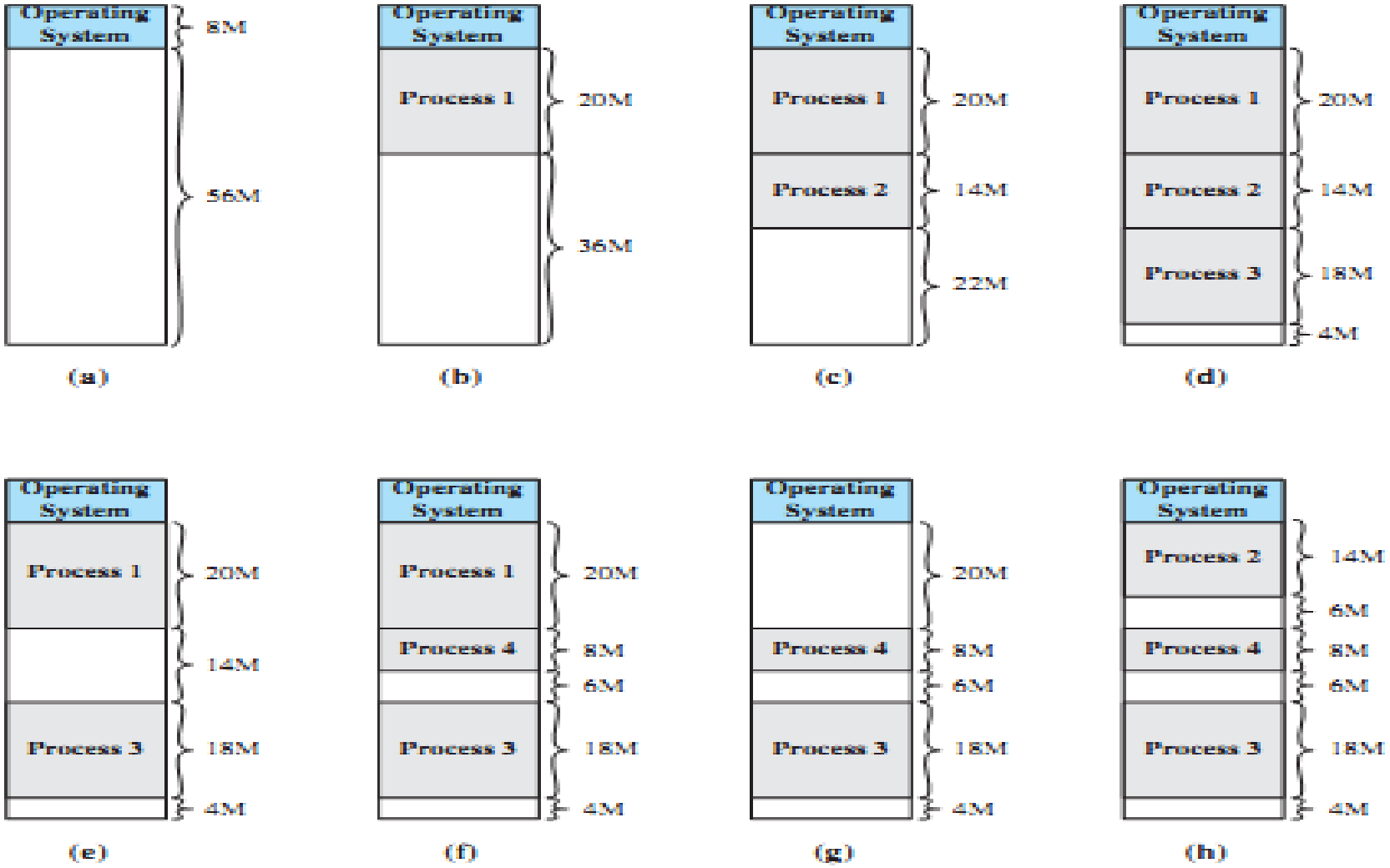
- the smallest available partition that will hold the process is selected.
- increases the level of multiprogramming at the expense of internal fragmentation.



Dynamic partitioning

- In fixed partitioning, degree of multiprogramming limited by number of partitions.
- **Hole** – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined.

Dynamic partitioning



Dynamic partitioning

MAT

Process	Starting address	Displacement

MFT

Starting address	Displacement
8000	63999

External fragmentation

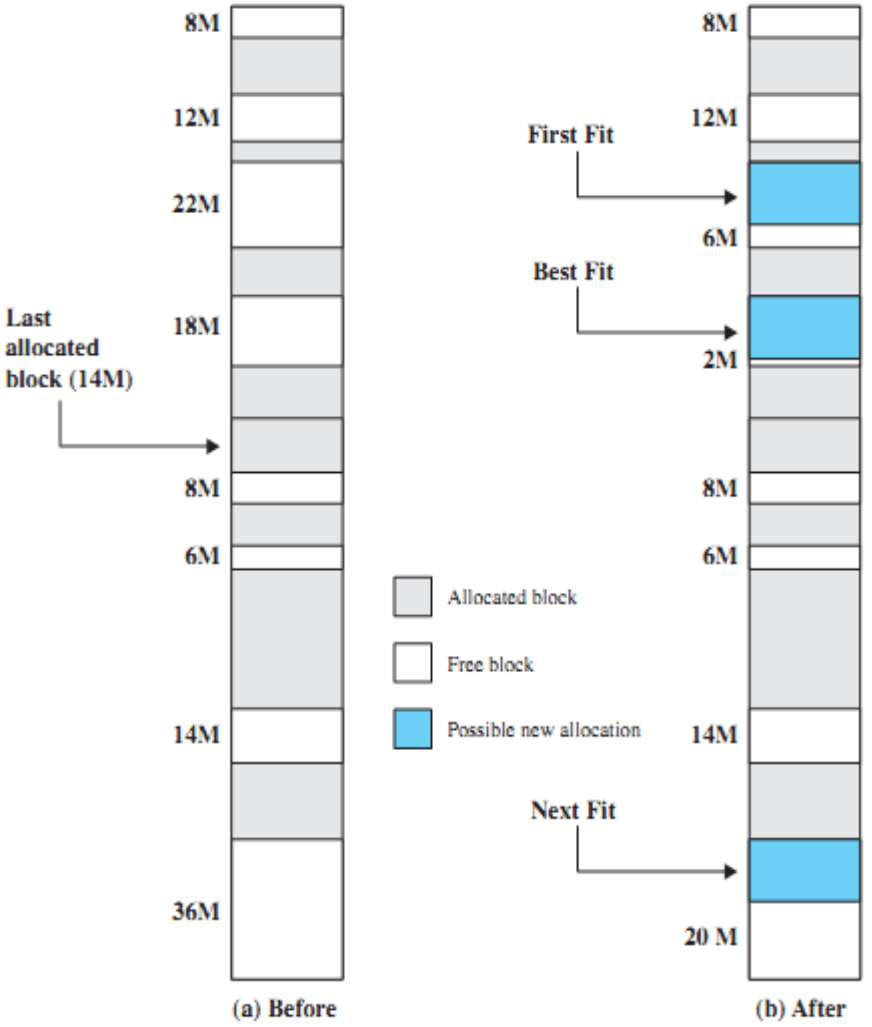
Placement Algorithms in Dynamic partitioning

To satisfy request of size n from list of free holes – four basic methods:

- **First-fit**: Allocate the first hole that is big enough.
- **Next-fit**: Same logic as first-fit but starts search always from the last allocated hole (need to keep a pointer to this) in a wraparound fashion.
- **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.

Placement Algorithms in Dynamic partitioning

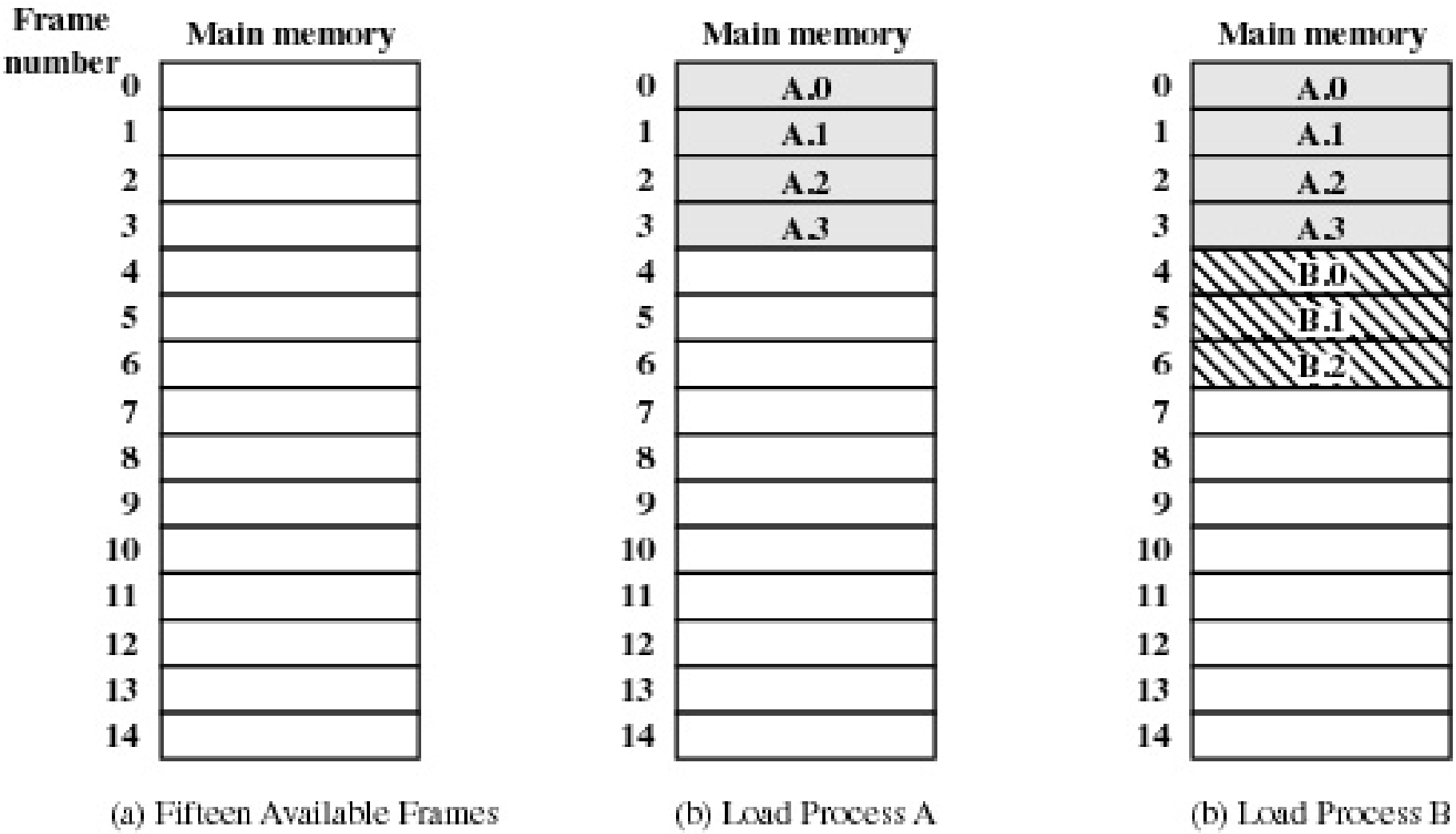
- New process of 16MB



Paging (Noncontiguous allocation)

- Physical memory is divided up into fixed-sized blocks called ***frames***.
- Logical memory of a process is also divided into blocks of same size as the page frames. These blocks called ***pages***.
- OS keeps track of all free frames.
- To run a program of size n pages, need to find n free frames and load program into them.
- Then set up a page table to translate logical to physical addresses. Separate page table for each process.

Paging

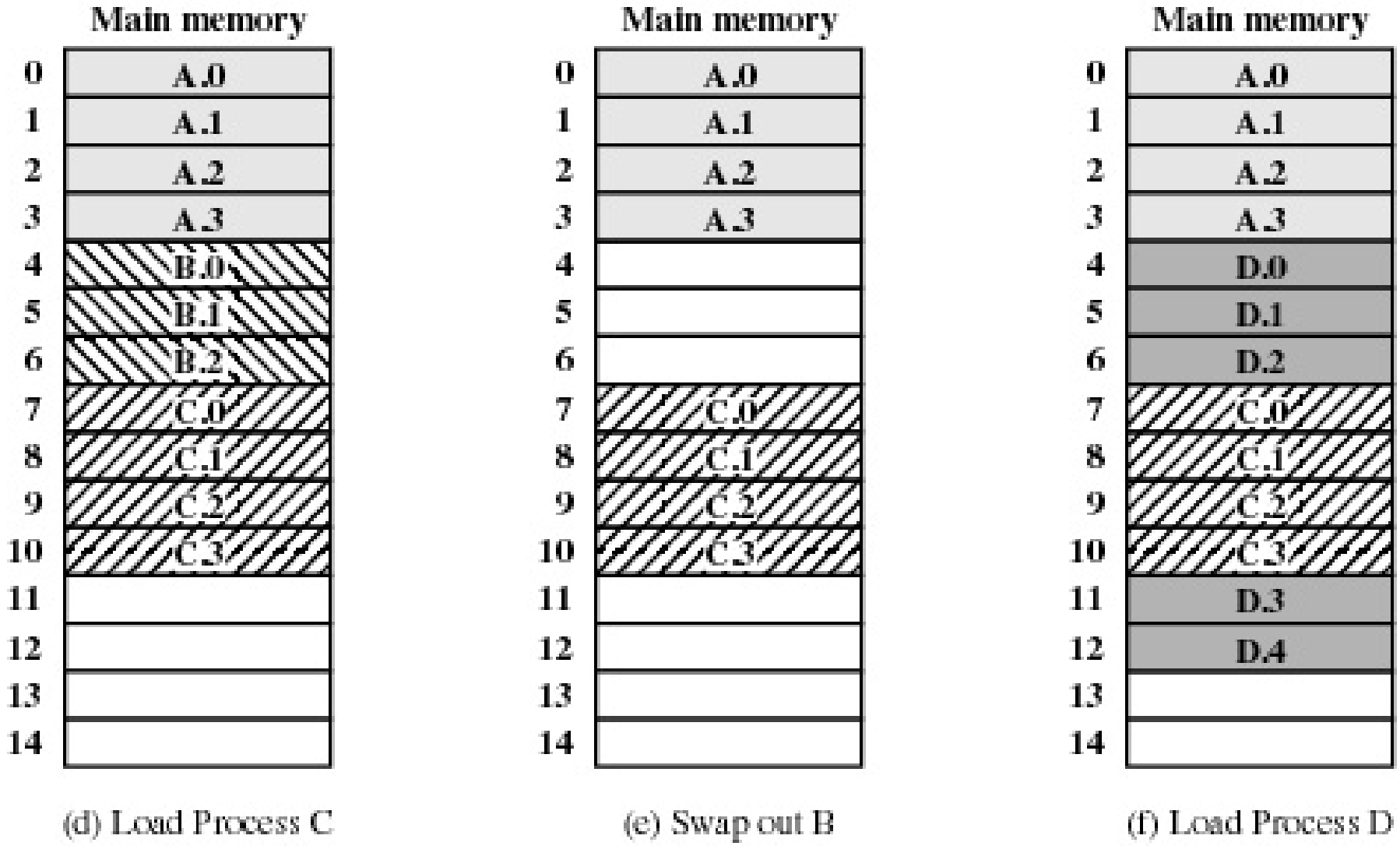


(a) Fifteen Available Frames

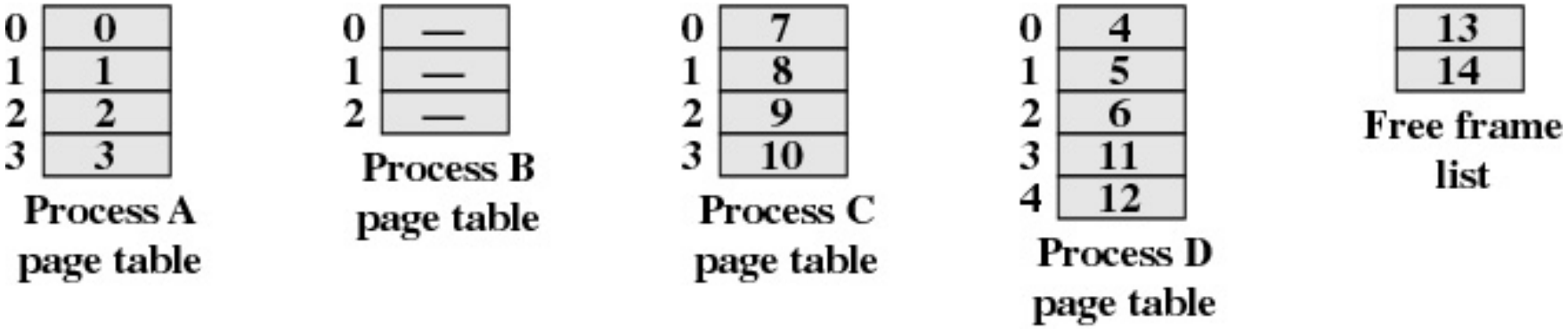
(b) Load Process A

(b) Load Process B

Paging


Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4 B.0
5 B.1
6 B.2
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
(d) Load Process C
Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4
5
6
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
(e) Swap out B
Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4 D.0
5 D.1
6 D.2
7 C.0
8 C.1
9 C.2
10 C.3
11 D.3
12 D.4
13
14
(f) Load Process D

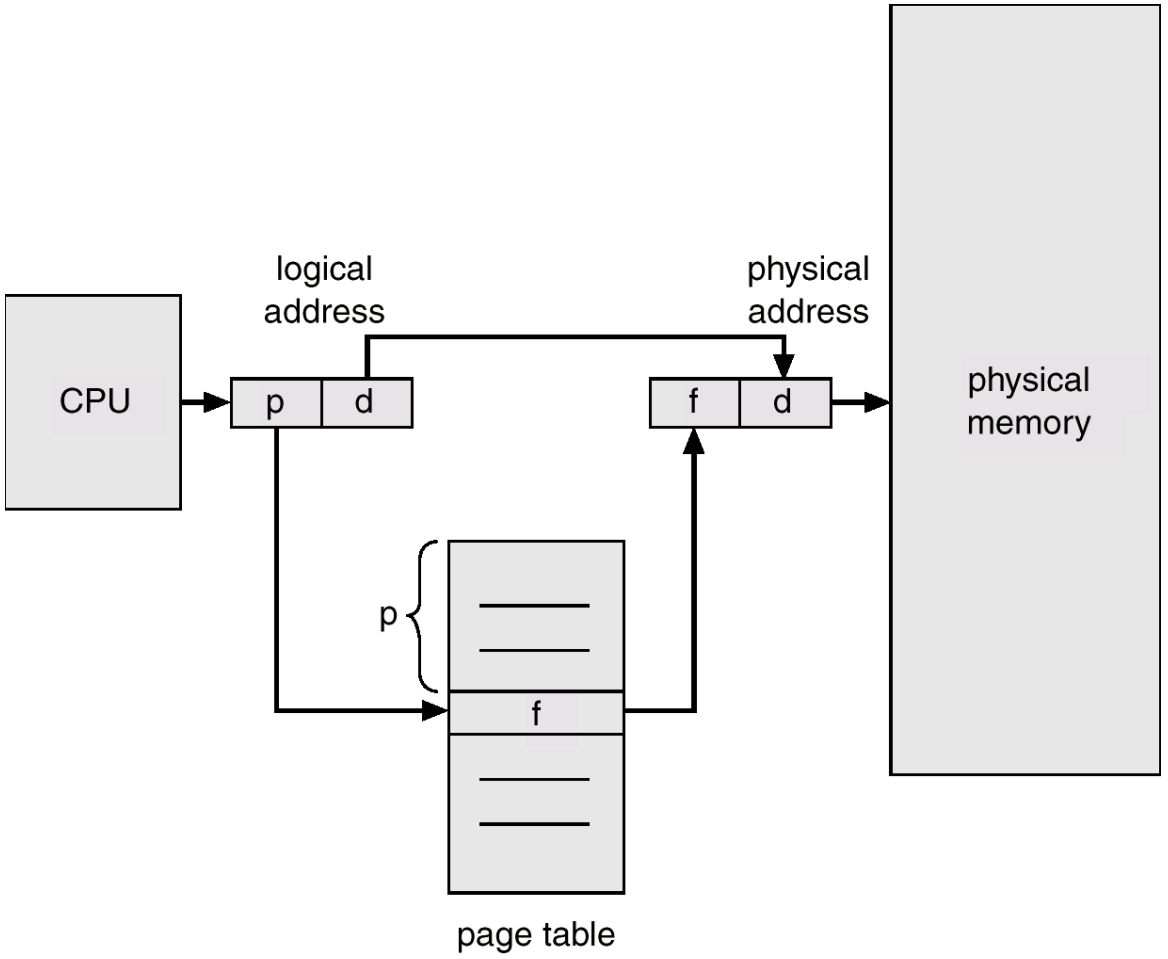
Page Tables for Example



Address Translation Scheme: Paging

- Address generated by CPU is divided into:
- Page number (p) – used as an index into a page table which contains base address of each page in physical memory.
- Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.
- Logical address space is 2^m, and frame size is 2^n.
  - Higher order (m-n) bits are used for page number (p) and lower order n bits are used for page offset (d)

Address Translation Scheme: Paging



Implementation of Page Table

- Set of dedicated registers:
  - Let 16-bit logical address, and page size=8K.
  - Page table contains maximum 8 entries.
  - 8 dedicated registers can store the page-frame mappings.
  - Not practical if page table contains large number of entries.

Implementation of Page Table

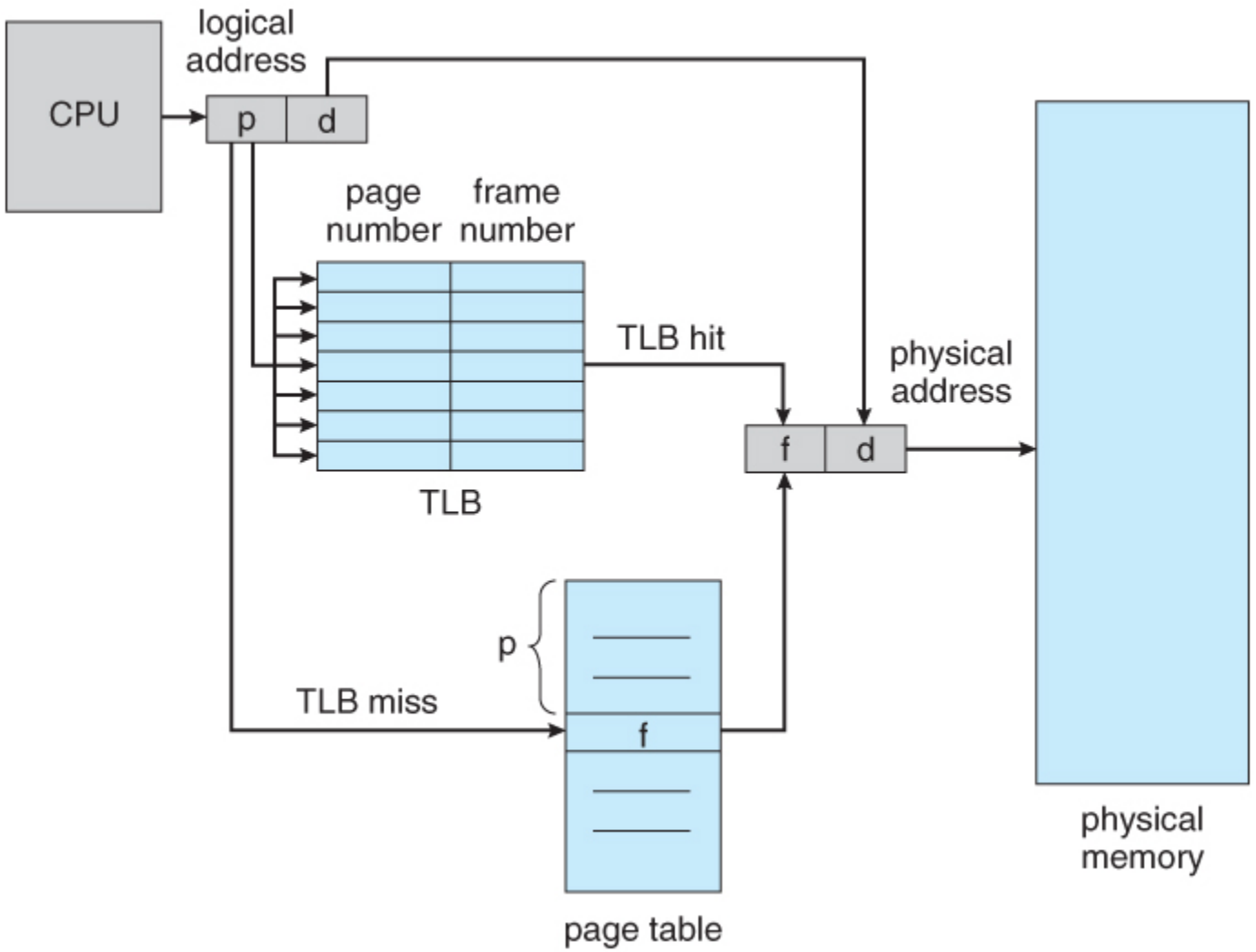
- PMT kept in main memory.
  - Page Table Base Register (PTBR) points to the PMT.
  - Page Table Length Register (PTLR) stores size of the PMT.
  - Memory access is slowed down by a factor of 2.

Implementation of Page Table

- PMT kept in main memory and TLB
  - A small, high speed associative cache called Translation Look-aside Buffer (TLB) stores some entries of PMT.
  - PMT, as a whole is stored on main memory.
- TLB (Associative Cache) – allow parallel search

Page#	Frame#

TLB hardware for paging



Effective memory access time

- Memory access time = 100ns
- TLB lookup time = 20ns
- TLB hit ratio = 80%
- Effective access time =

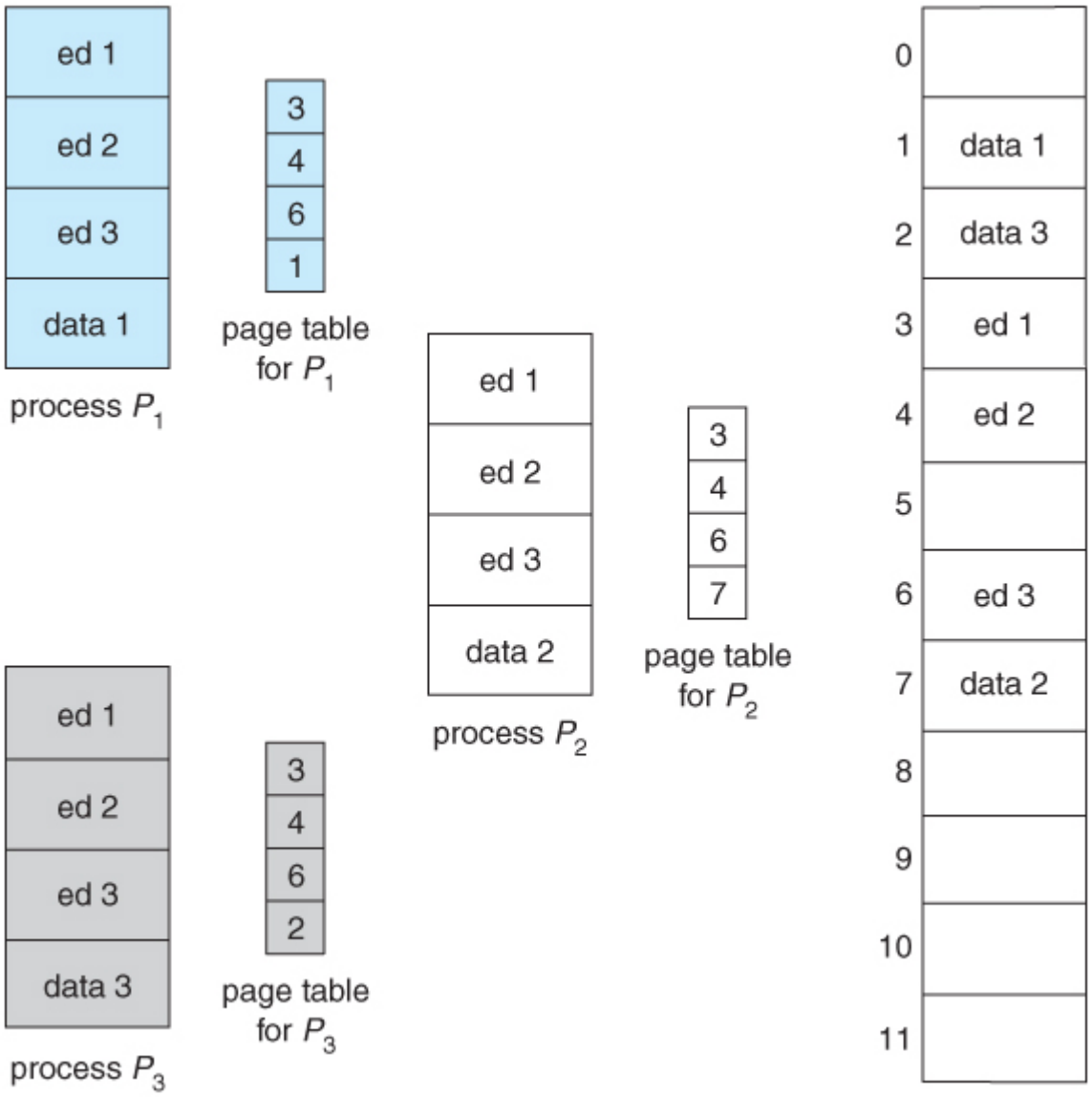
$$0.8 * (20+100) + 0.2 * (20+2*100) = 140 \text{ ns}$$

Shared Pages

- If the code is *reentrant* (not self-modifying) then it can be shared among processes.
- Let 40 users executing text editor
- Text editor: 150KB code + 50KB data
- Total memory required = 8000KB (more)

- However, code pages can be shared
- Let page size = 50KB then 3 code pages and 1 data page per user

Shared Pages



150KB shared code + 40*50KB data

=2150KB (40 users)

Page Table Structure

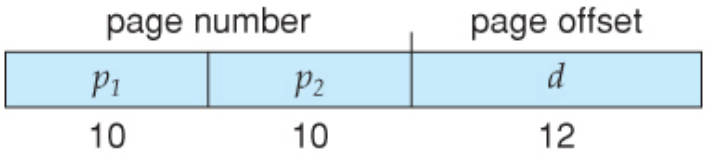
- Most modern computer systems support 32bit or 64bit logical address.
- Consider a system with 32bit logical address, and page size of 4K

- # of entries in PMT=2^{20} (worst case)
- Each entry of PMT stores a 21bit frame# and other attributes (for a 8GB memory).
- If each entry is of 4bytes then memory required by PMT=$4*2^{20}$=4MB (large)
- Do we need to apply paging on the PMT itself??

YES – Hierarchical Paging

2-level paging example

Page size=3K, entry in PMT=1K



Process A=18K

0	
1	
2	
3	
4	
5	

Inner Page Table

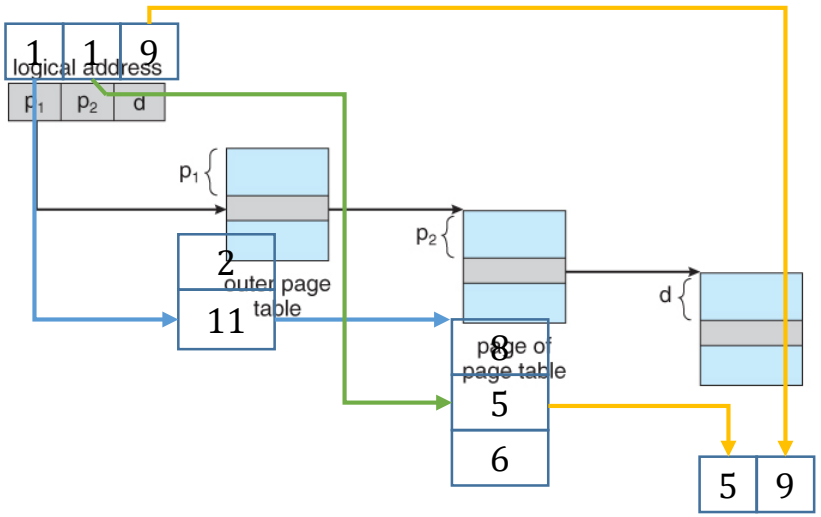
0	3	
1	1	pg0
2	7	
3	8	
4	5	pg1
5	6	

Outer Page Table

pg0	2
pg1	11

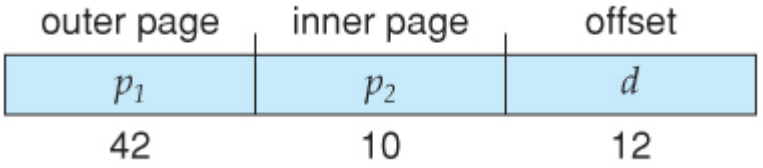
Main memory

0	
1	A.1
2	pg0
3	A.0
4	
5	A.4
6	A.5
7	A.2
8	A.3
9	
10	
11	pg1
12	
13	
14	

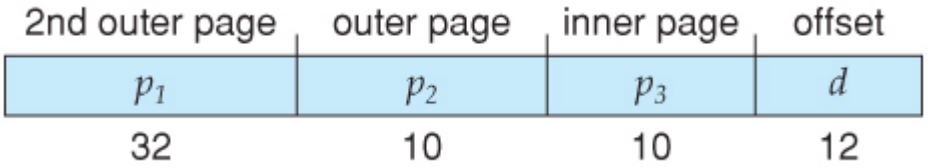


Hierarchical paging

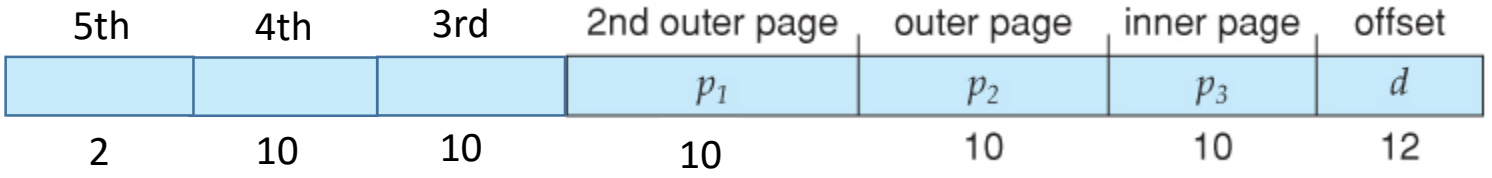
- For a 64-bit logical address system:



64-bits Two-level leaves 42 bits in outer table

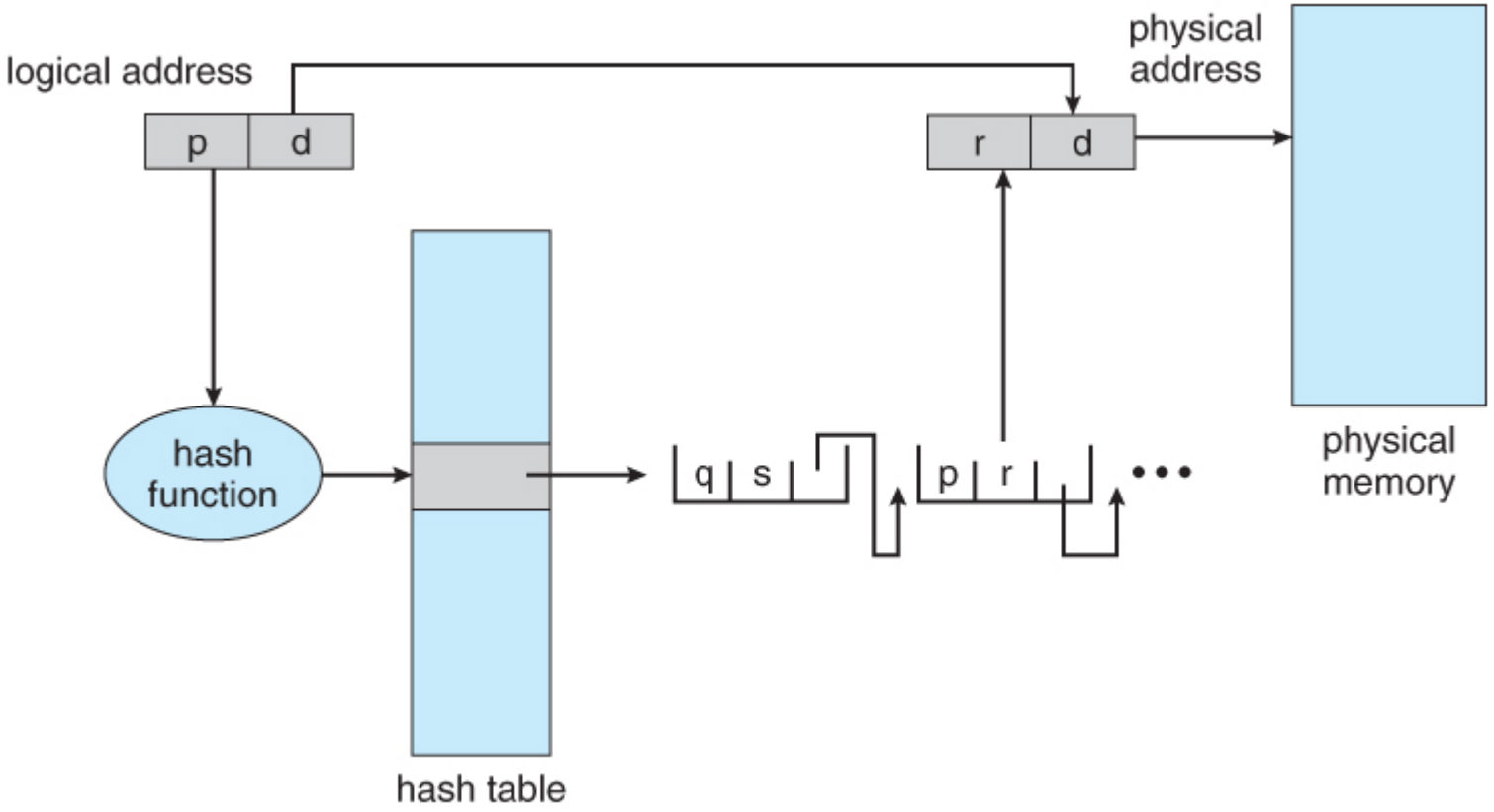


Going to a fourth level still leaves 32 bits in the outer table.



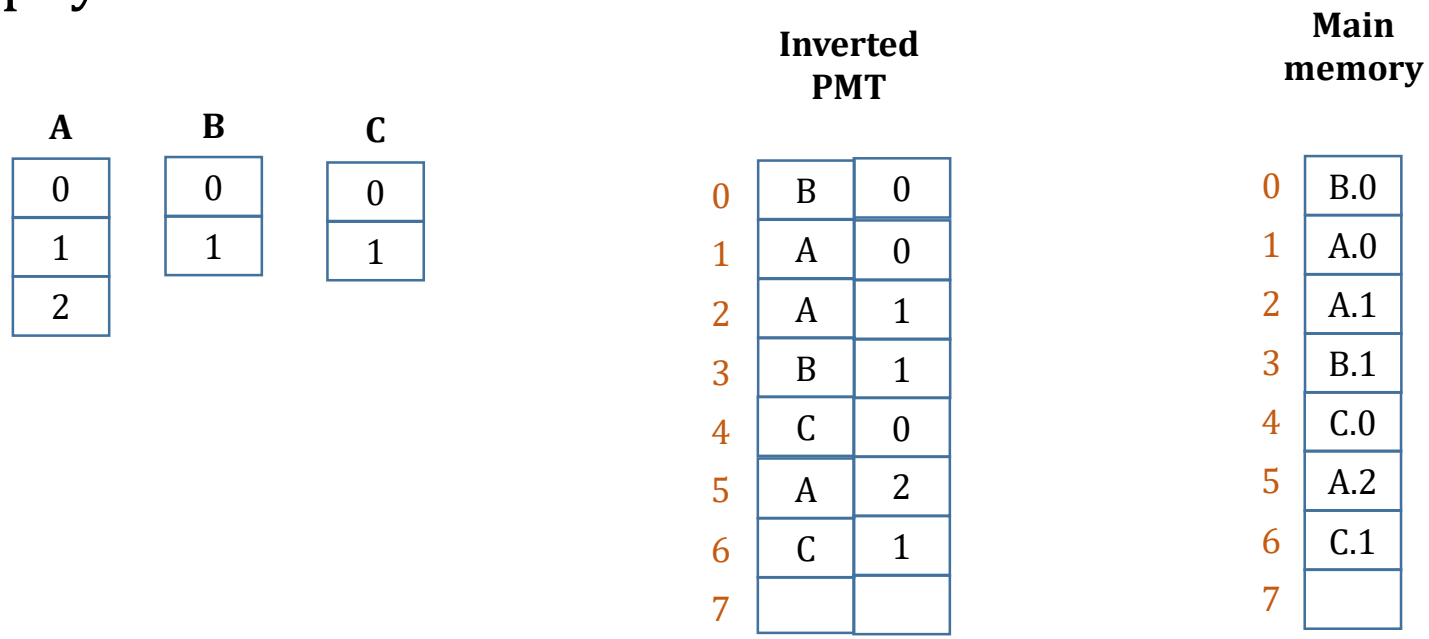
7 levels of indirection, which would be prohibitively slow memory access

Hashed Page Tables



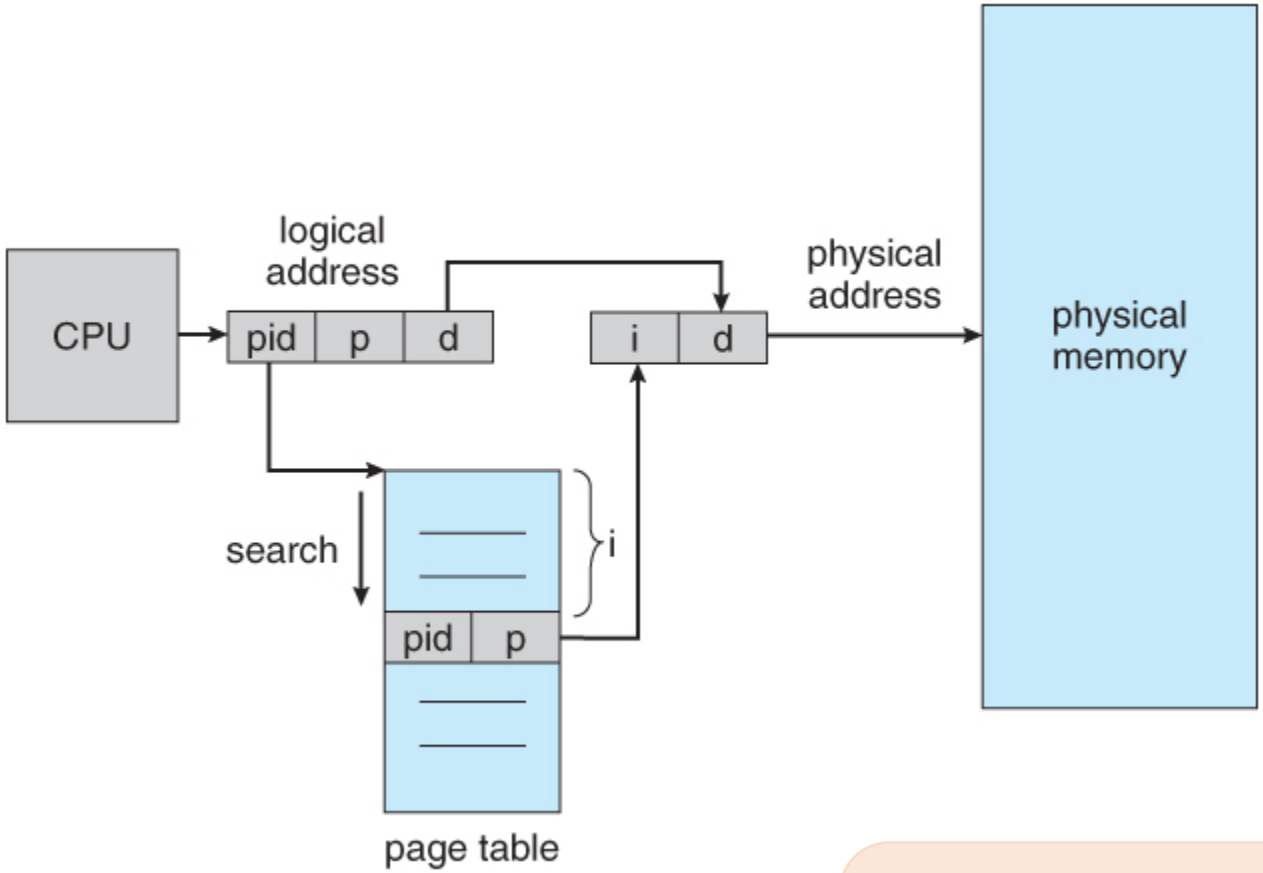
Inverted Page Table

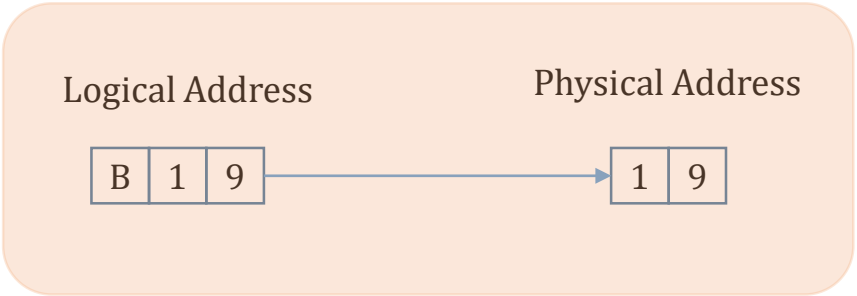
- Managing per-process PMT is complex.
- Inverted page table keeps an entry corresponding to each physical frame



Search time is more
Solution: Hashing

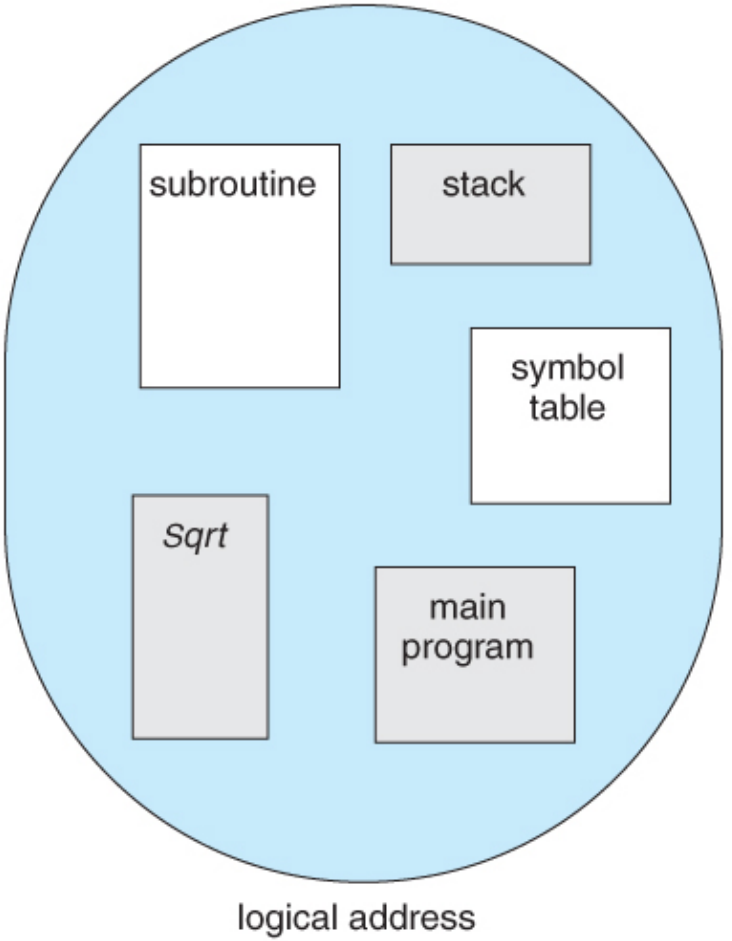
Inverted Page Table



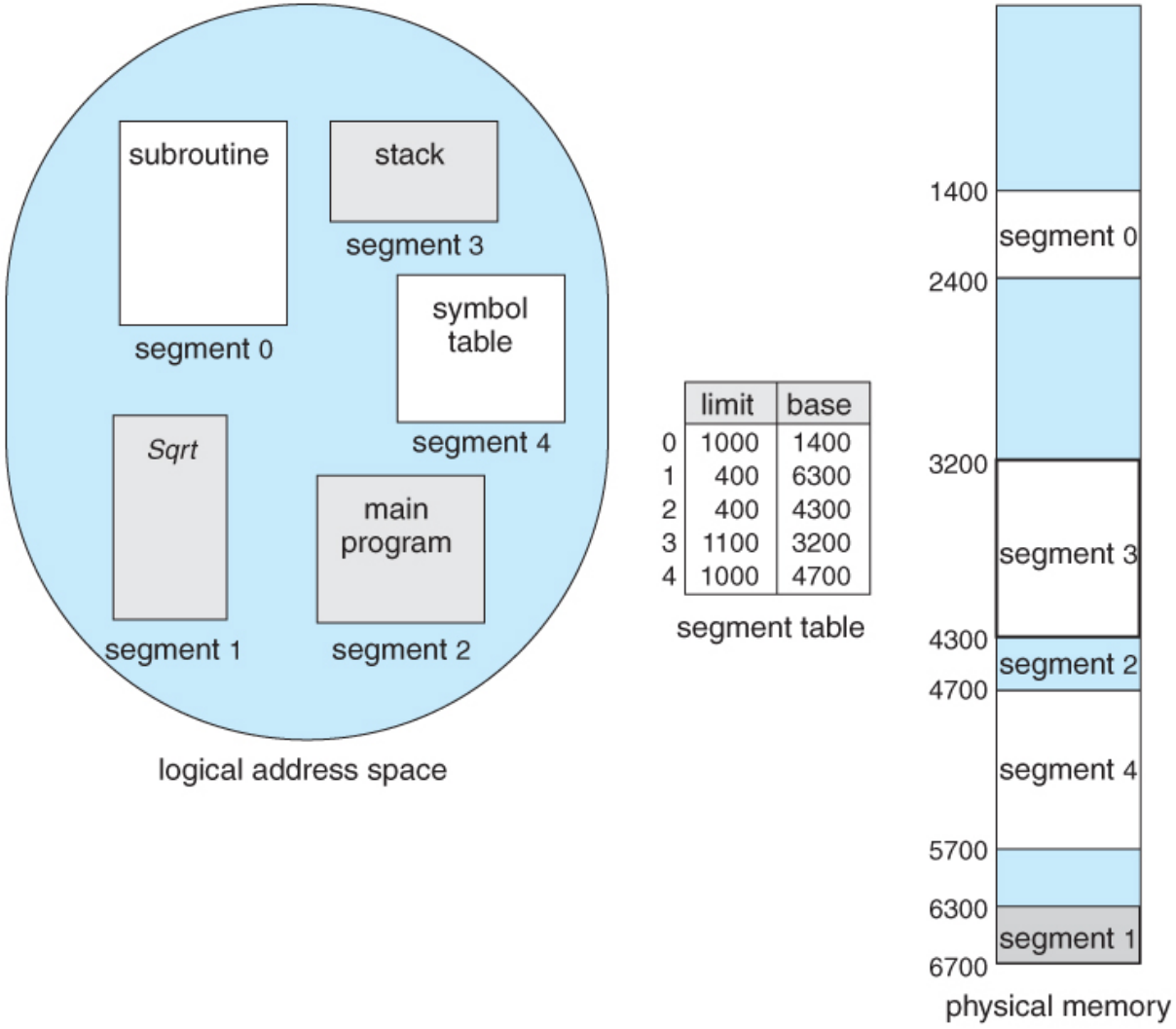


Segmentation

- User view (at higher level of abstraction) - Process as a collection of blocks (segments)
- E.g., main(), sqrt(), stack etc



Segmentation Example



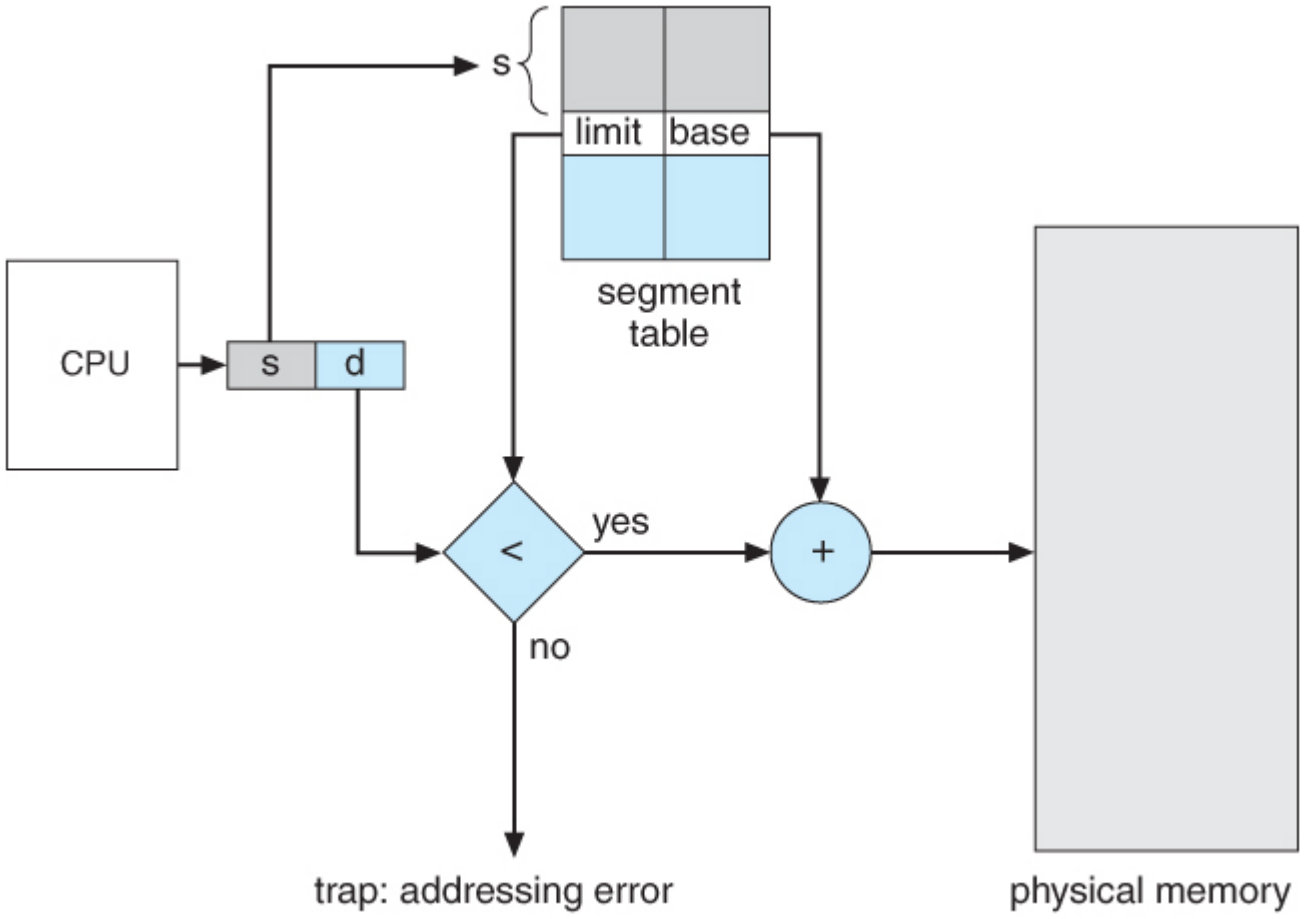
Segmentation Hardware

- Logical address consists of a pair: <segment-number, offset>,
  - Segment number is an index into the segment table - with one segment table for each process
  - offset is the value to be added to start address of segment to give real physical address
- each entry in segment table has:
  - base value – contains the starting physical address where the segment resides in memory.
  - limit – specifies the length of the segment.
- Offset address is legal if offset < limit value

Segmentation Hardware

- 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 process;
- segment number s is legal if s < STLR.

Segmentation Hardware



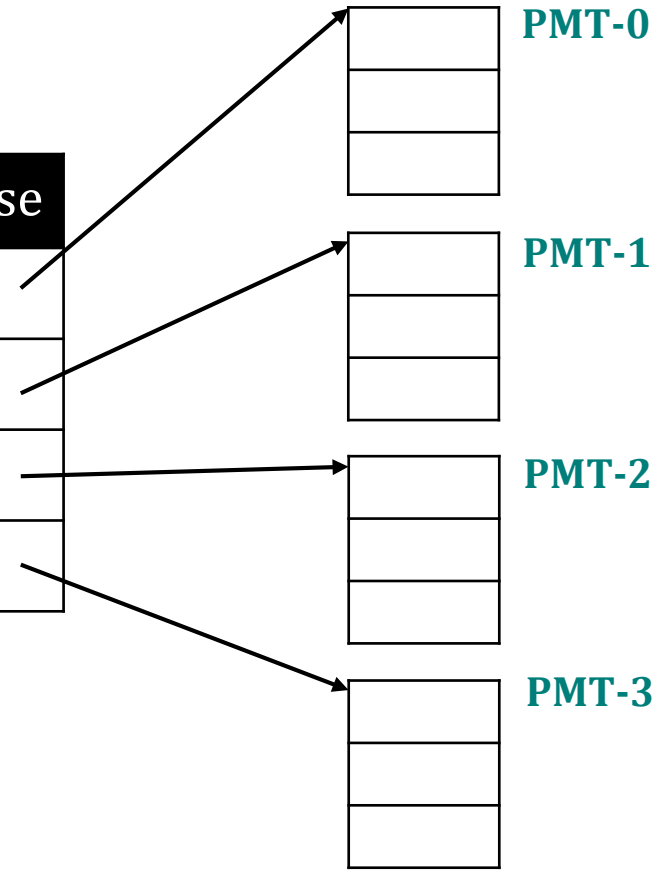
Segmentation with Paging

Logical Address

Segment#	Page#	Offset

Segment Table

Segment#	Page Table Base
0	
1	
2	
3	



Segmentation with Paging:
Address Translation