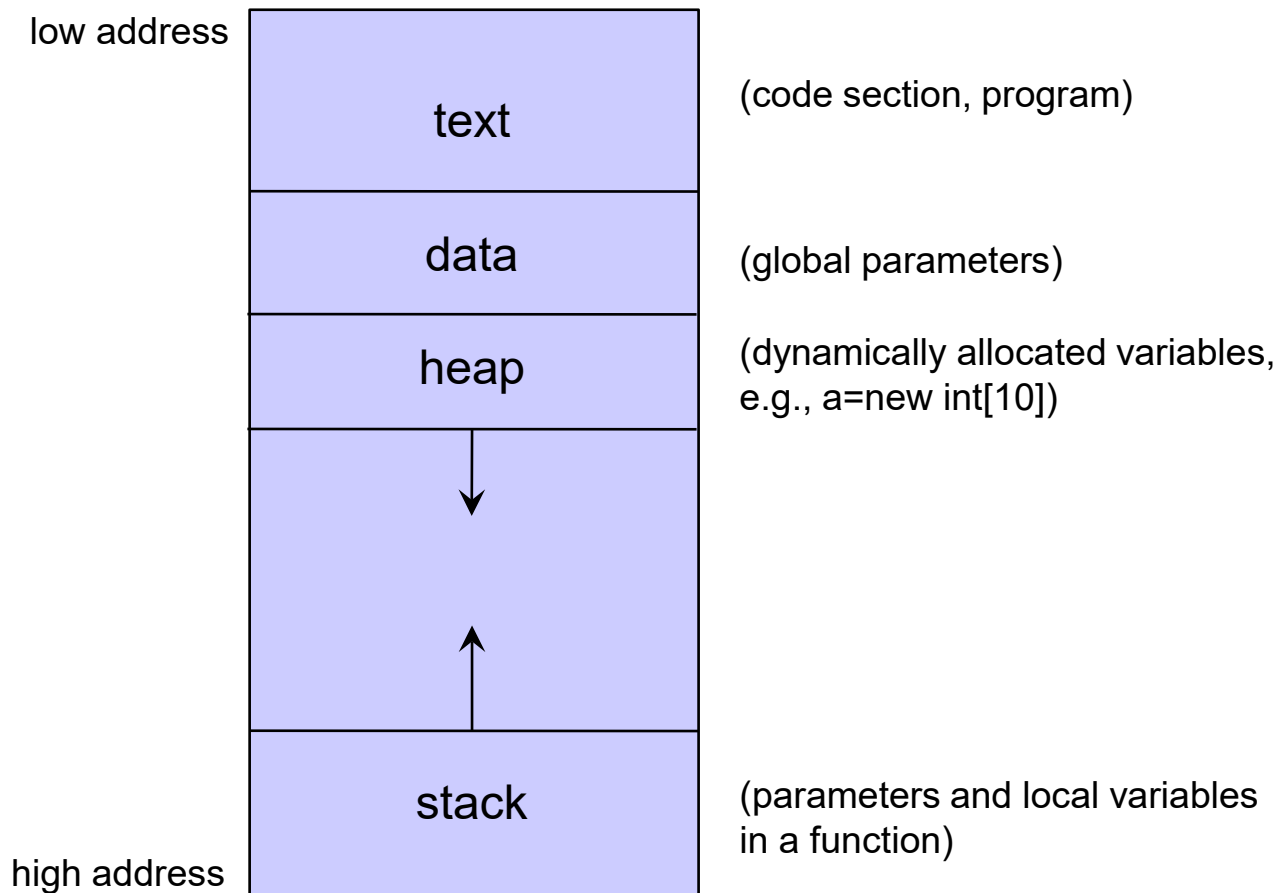


## Part 6: Memory Organization

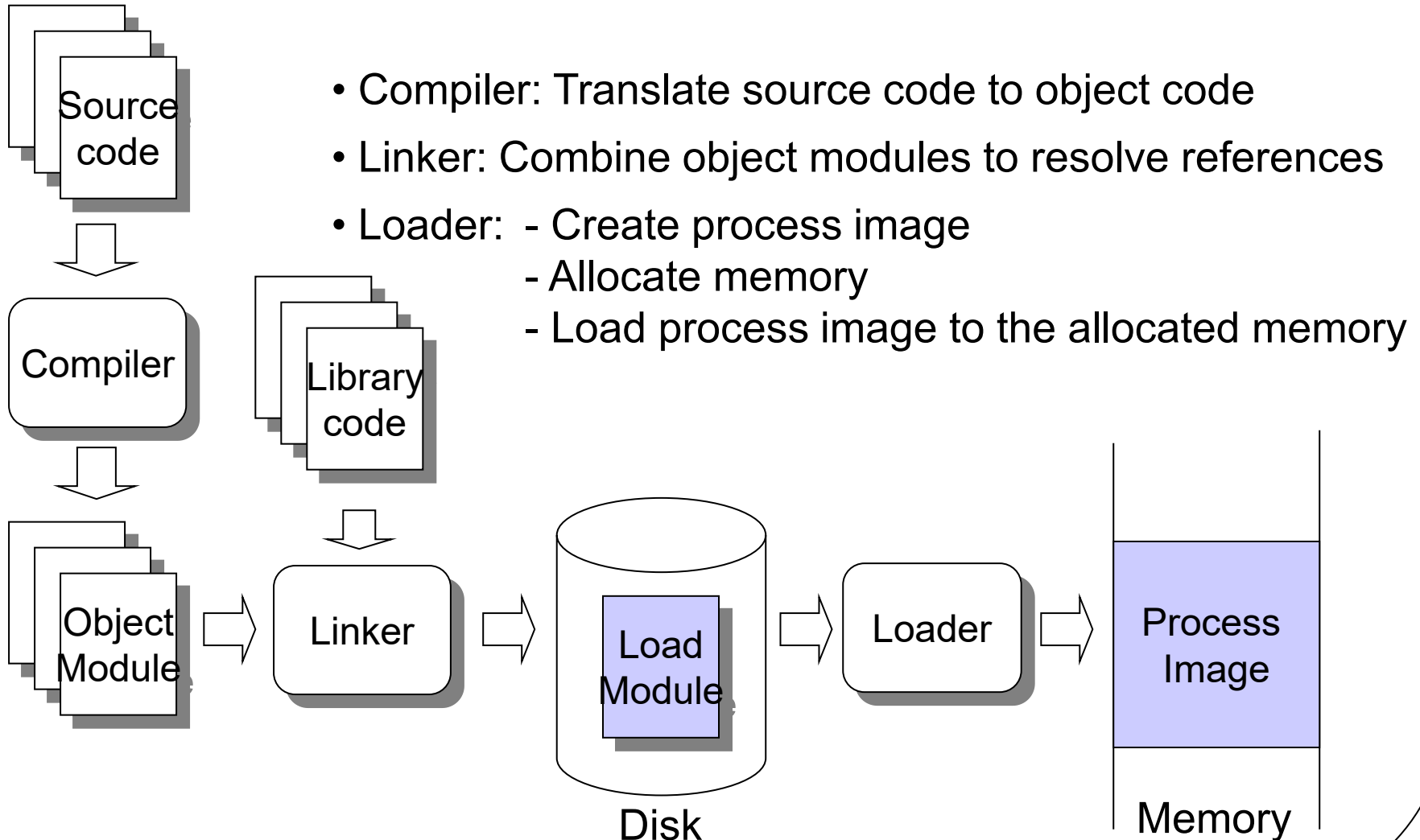
- Bind Code and Data to Memory
  - address binding, logical versus physical address space
- Contiguous Allocation
  - fixed vs. dynamic partitioning, fragmentation
- Paging
  - address translation, page table implementation, shared pages, two-level page-table, inverted page table
- Segmentation
  - address translation

# Bind Code and Data to Memory

- To run a program, a process image must be created and loaded into memory



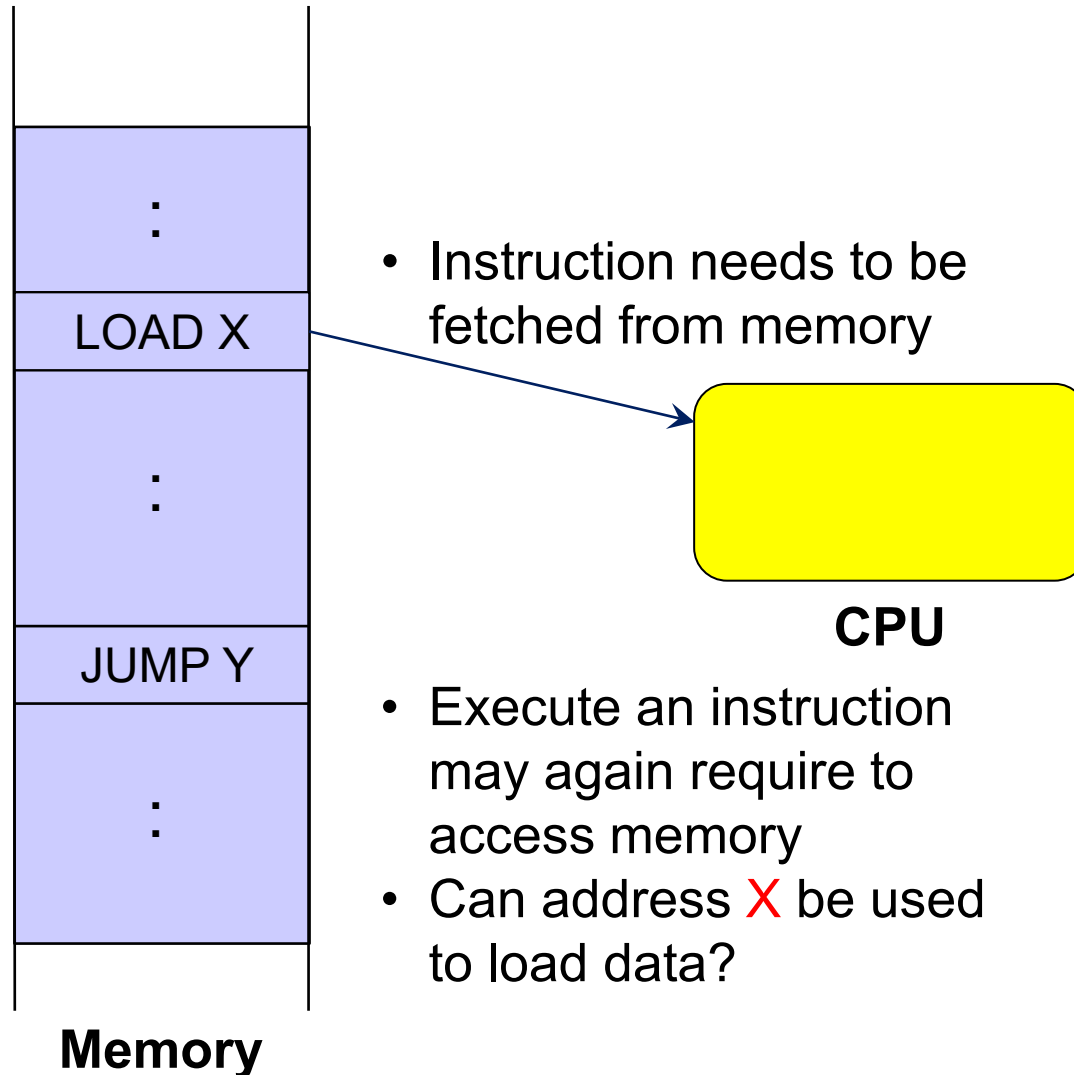
# Binding of Code and Data to Memory (Cont.)



# Logical vs. Physical Address Space

- *Logical address* – address used in the code, generated by the CPU when executing an instruction.
- *Physical address* – address used to access physical memory, seen by the memory unit.
- *Address Space* – all addresses accessible by a process

# Binding of Code and Data to Memory (Cont.)

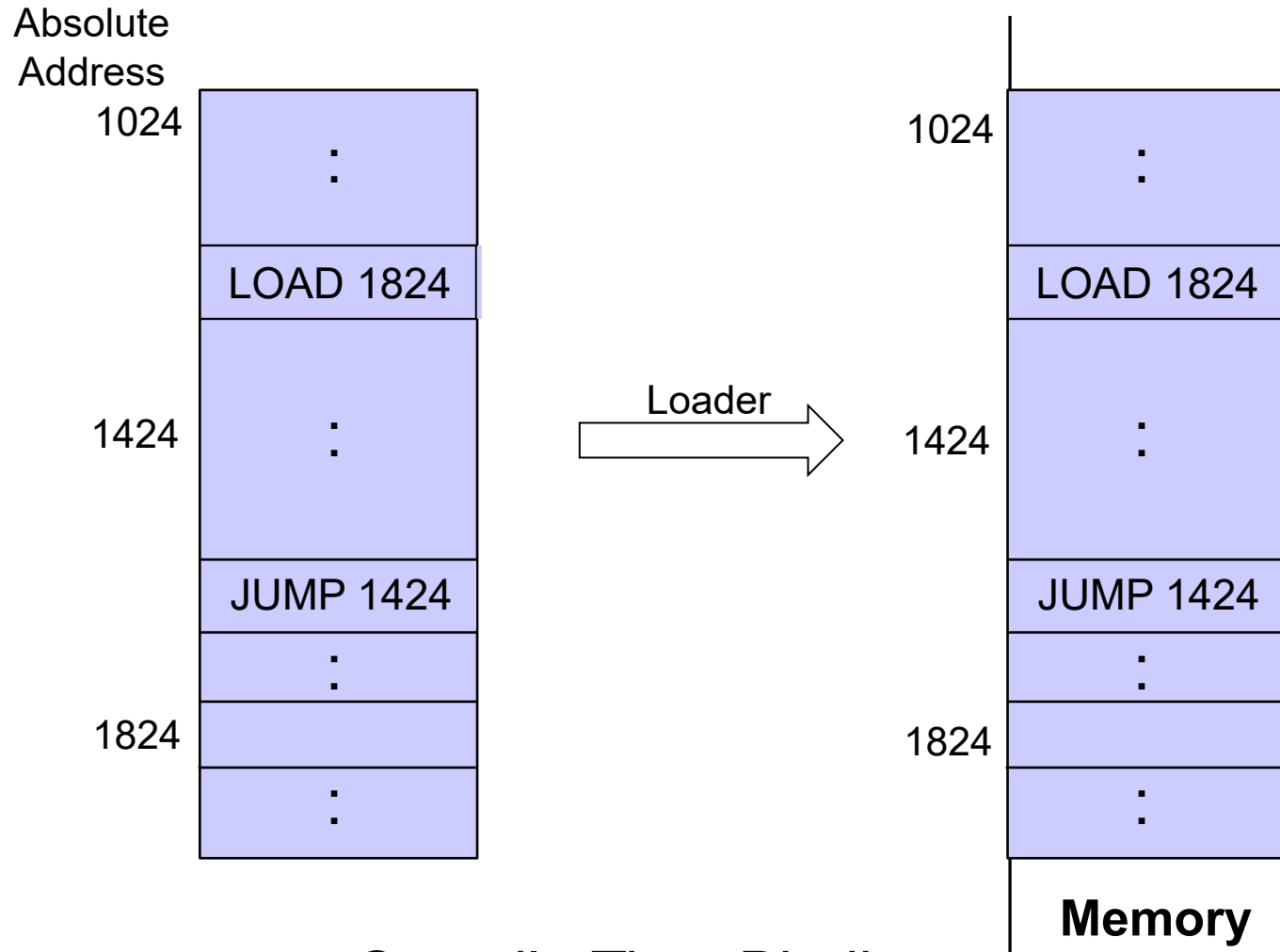


## Binding of Code and Data to Memory (Cont.)

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes.
- **Load time:** Compiler generates *relocatable code*. Binding is performed by the loader.
- **Execution time:** If the process can be moved during its execution from one memory segment to another, binding is delayed until run time.

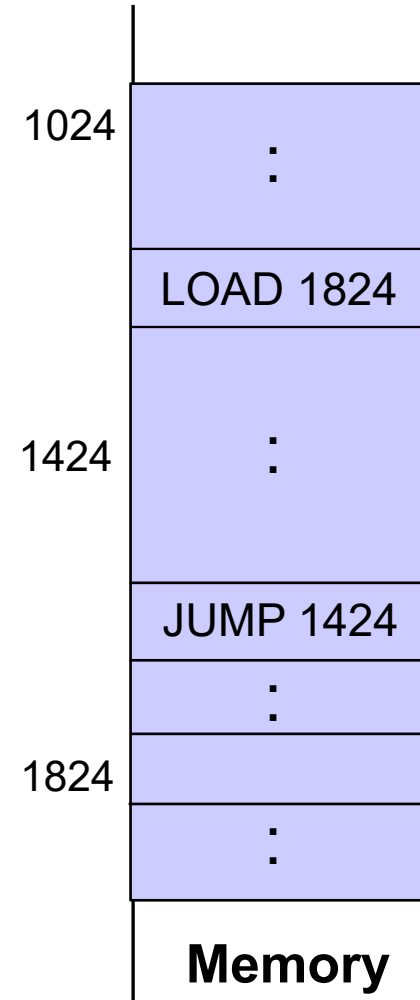
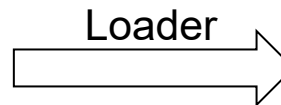
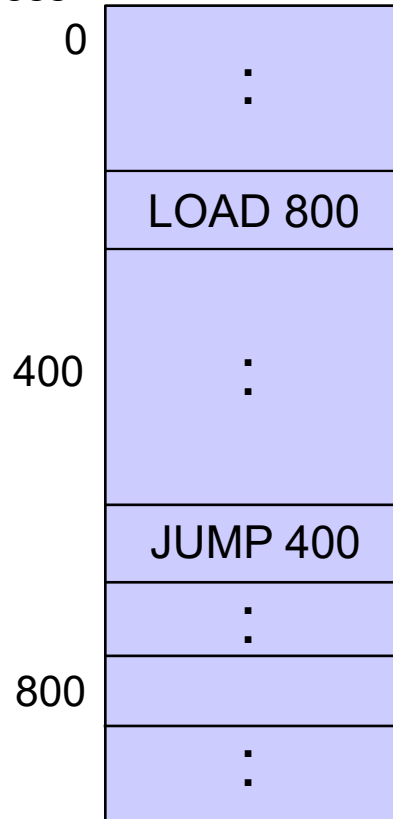
# Binding of Code and Data to Memory (Cont.)



Compile Time Binding

# Binding of Code and Data to Memory (Cont.)

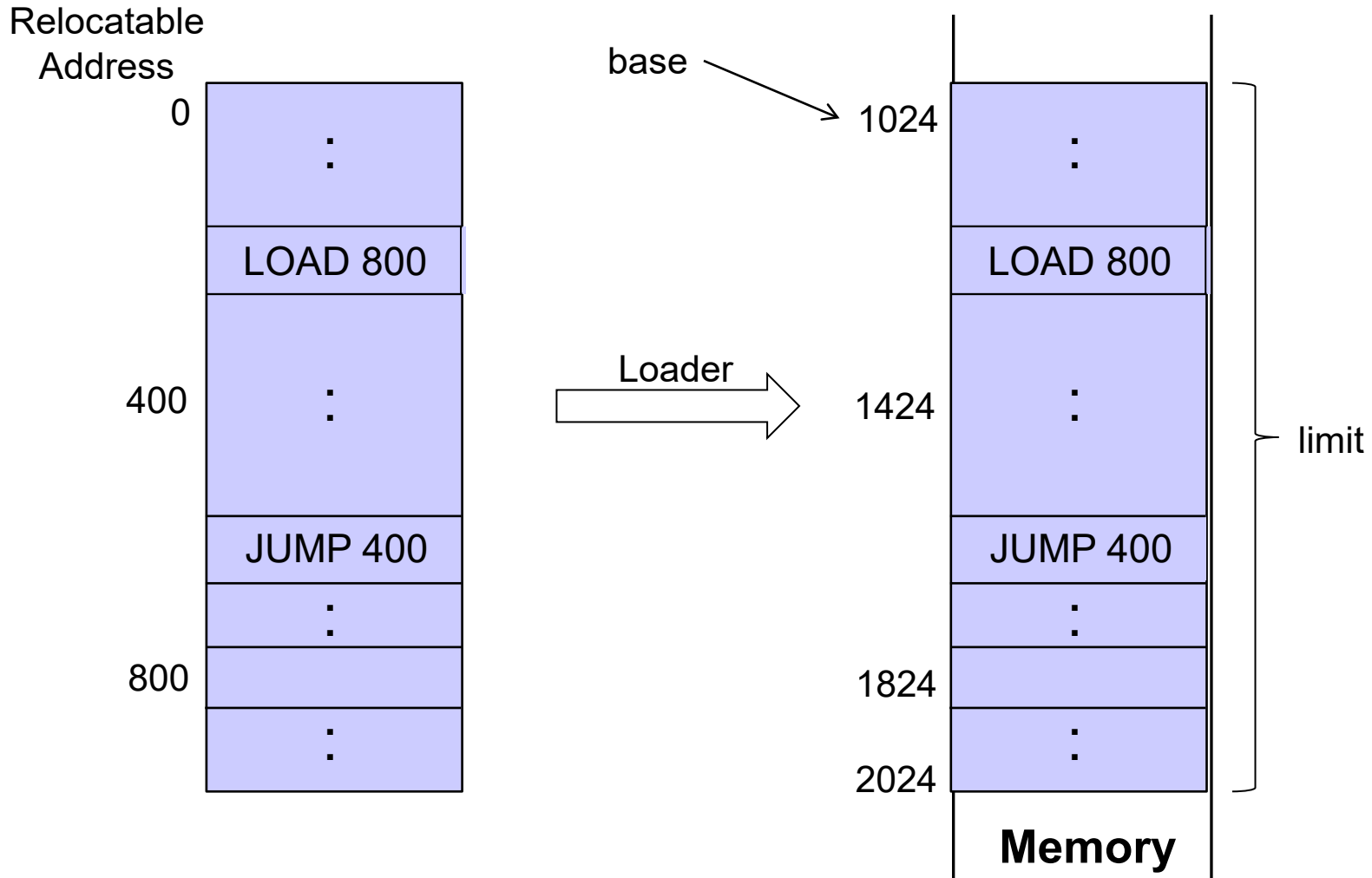
Relocatable  
Address



Load Time Binding

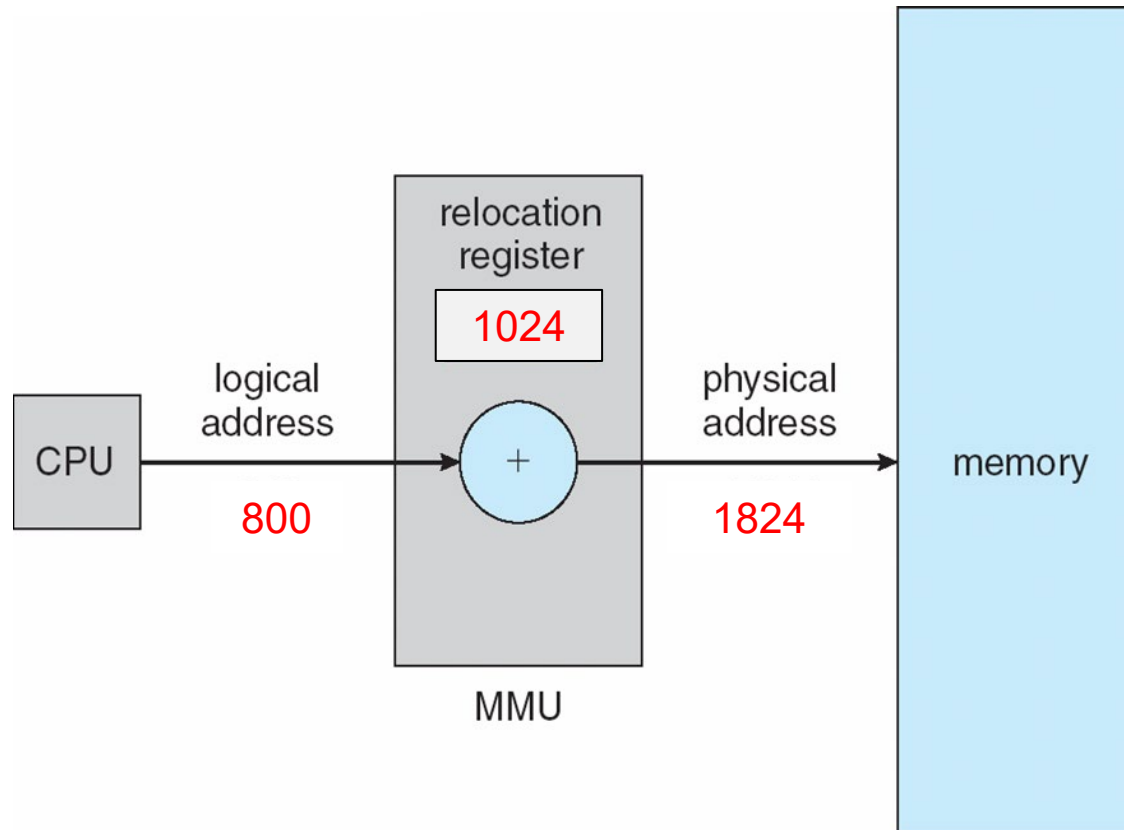


# Binding of Code and Data to Memory (Cont.)



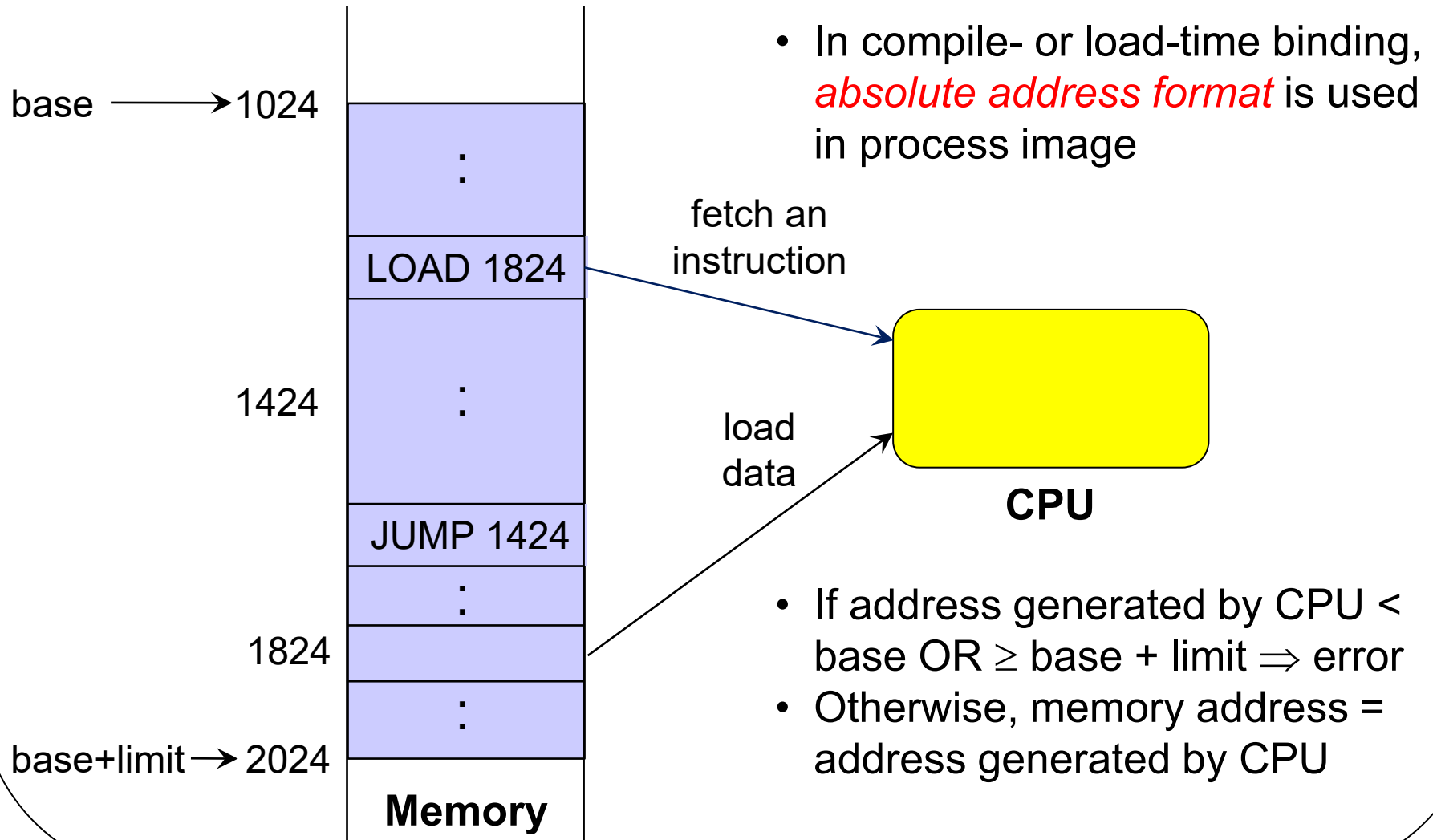
Execution Time Binding

# Binding of Code and Data to Memory (Cont.)

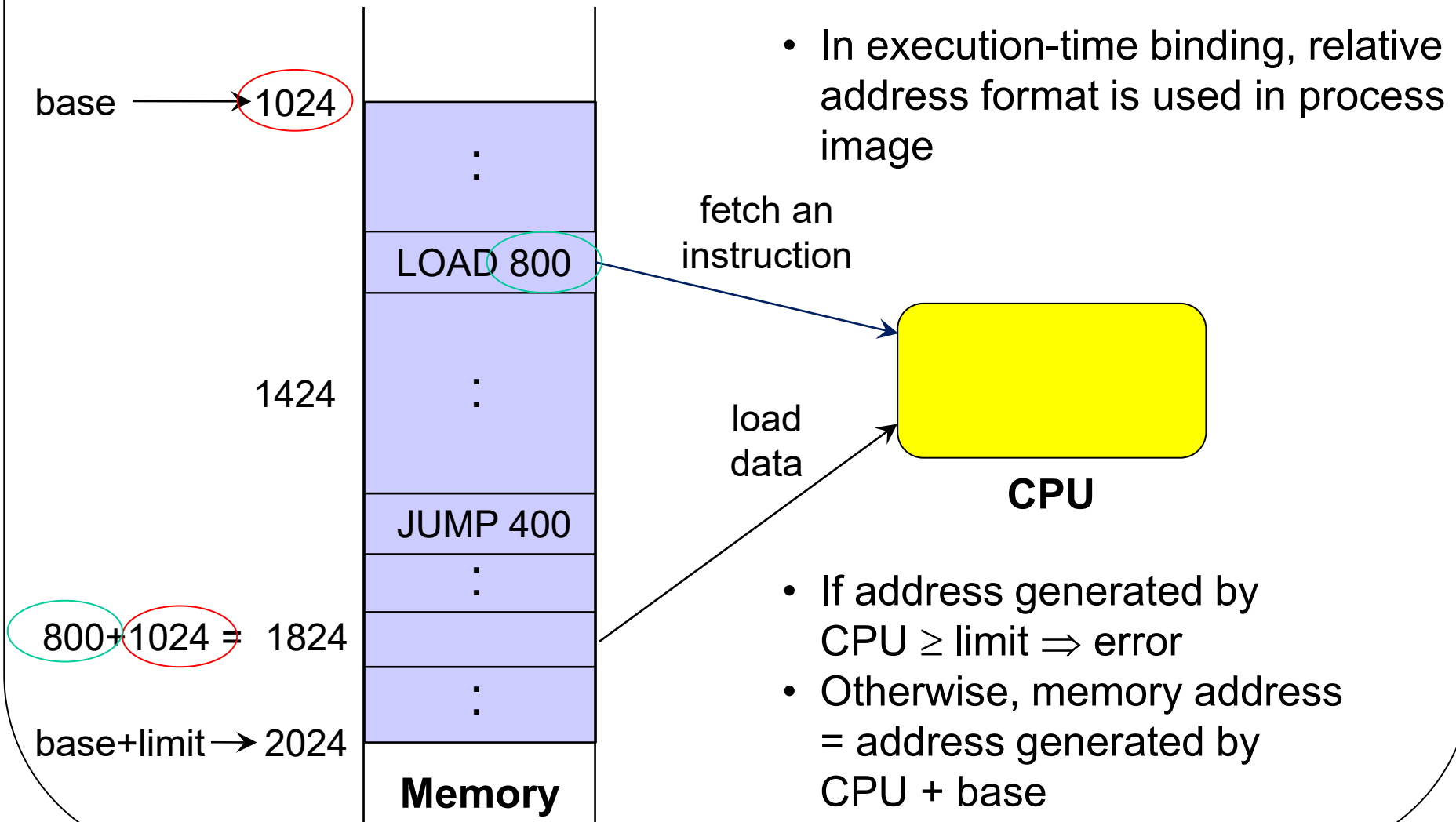


Execution Time Binding

# Binding of Code and Data to Memory (Cont.)

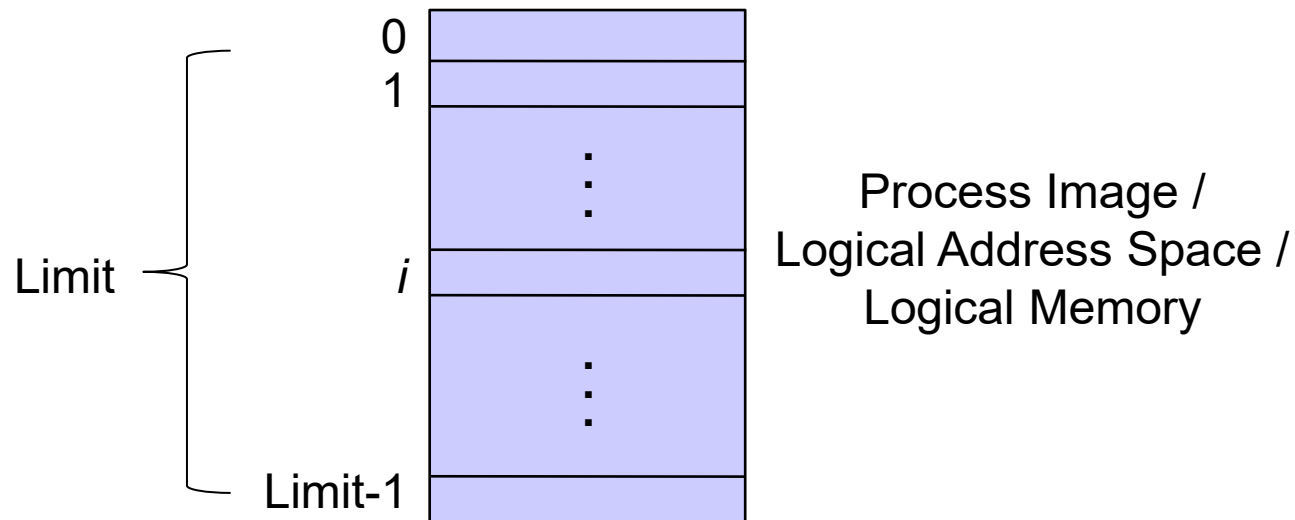


# Binding of Code and Data to Memory (Cont.)



# Logical vs. Physical Address Space

- Logical address space can be viewed as a linear, or one-dimensional, continuous address space consisting of a sequence of bytes
- For execution time binding, logical address space always starts from 0.



# How to Allocate Memory among Processes

- Two approaches:

- *Contiguous Allocation*

The logical address space of a process remains contiguous in physical memory

- Fixed Partitioning
    - Dynamic Partitioning

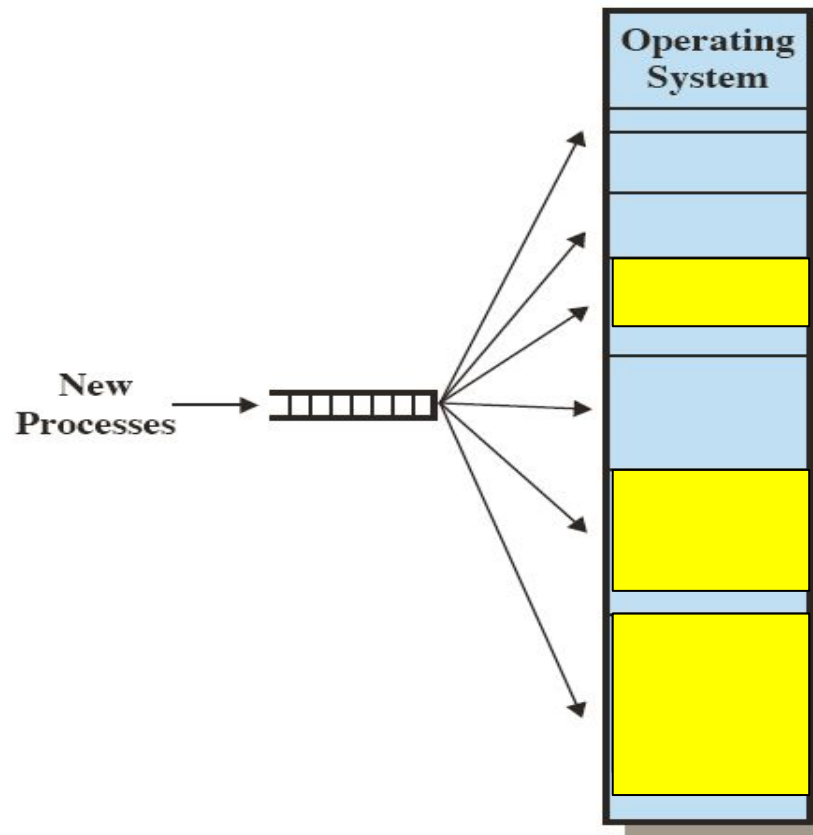
- *Non-contiguous Allocation*

A process logical address space is scattered over different regions in physical memory

# Contiguous Allocation

- *Fixed Partitioning*

Memory is partitioned into regions with fixed boundaries



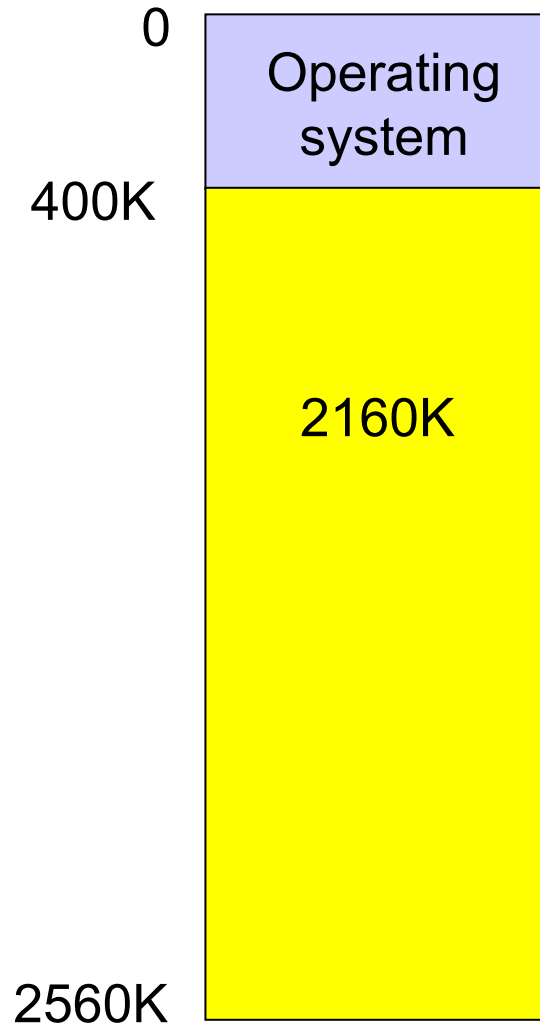
## Contiguous Allocation (Cont.)

- *Dynamic Partitioning*

- *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.
- Operating system maintains information about
  - allocated partitions
  - free partitions (hole)

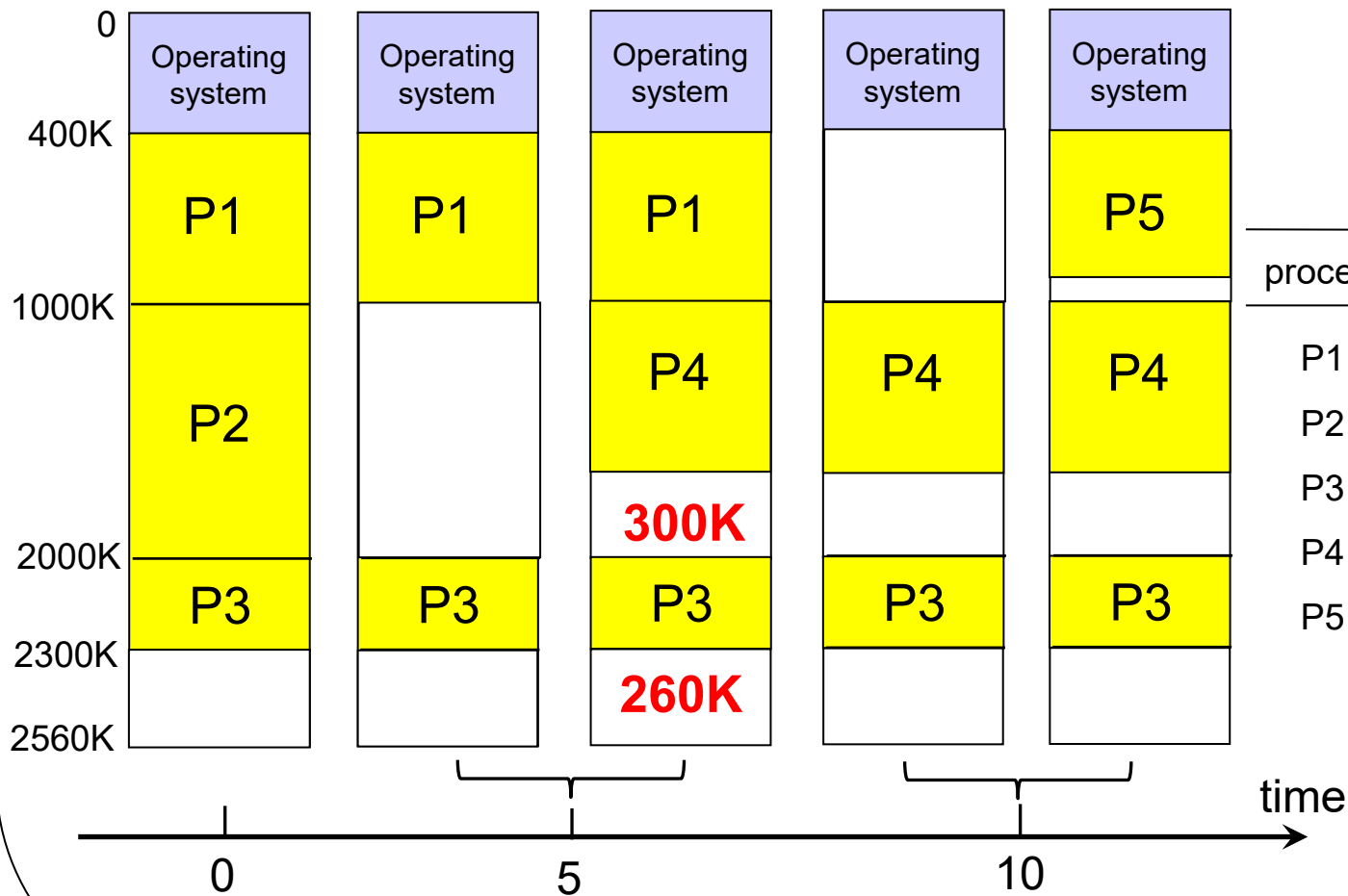


# Contiguous Allocation (Cont.)



Job queue		
process	memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

# Contiguous Allocation (Cont.)



# Dynamic Storage-Allocation Problem

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

- **First-fit:** Allocate the *first* hole that is big enough.
- **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.

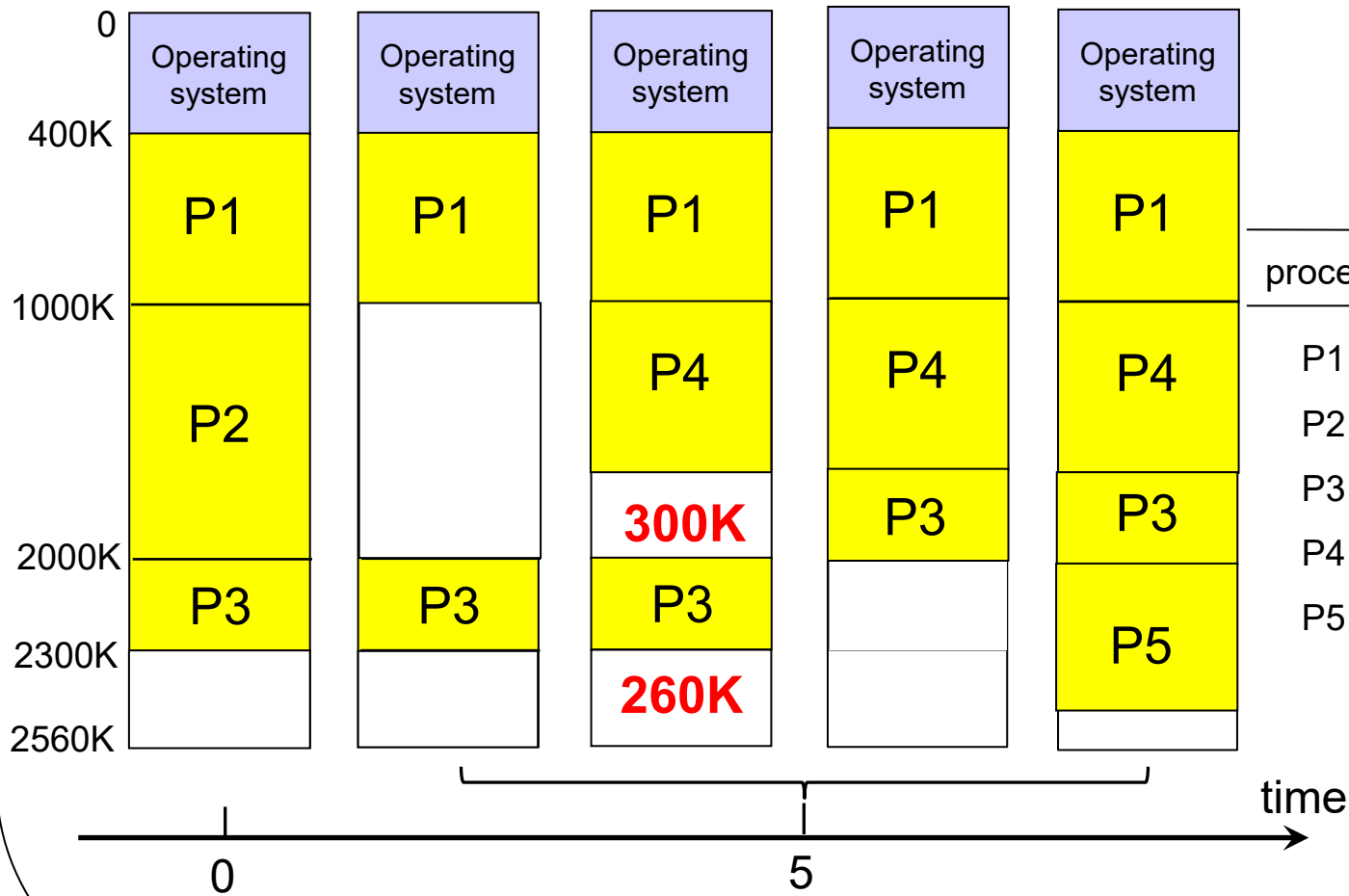
# Fragmentation

- *External fragmentation*: when enough total memory space exists to satisfy a request, but it is not contiguous. Happens outside a partition.
- *Internal fragmentation*: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

## Fragmentation (Cont.)

- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible only if re-locatable address format is used in process image and binding is done during execution-time.

# Fragmentation (Cont.)



Job queue		
process	memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

# Paging

- Memory space allocated to a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called *frames* (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called *pages*.

## Paging (Cont.)

- Keep track of all free frames.
- To run a program of size  $n$  pages, need to find  $n$  free frames and load program.
- Set up a *page table* to translate logical to physical addresses.
- External fragmentation is eliminated
- Internal fragmentation is still possible, on average half of a page is unused.



# Paging (Cont.)

0	a	0
1	b	
2	c	
3	d	
4	e	1
5	f	
6	g	
7	h	
8	i	2
9	j	
10	k	
11	l	
12	m	3
13	n	
14	o	
15	p	

logical memory

page #	0	5	frame #
	1	6	
	2	1	
	3	2	

page table

- To load process to memory, need to find 4 frames
- Page Table to remember the mapping

Logical address space:  
4x4 = 16 bytes  
4 Pages

Page Size (and Frame Size): 4 bytes

0		0
4	i	1
	j	
	k	
8	m	2
	n	
	o	
	p	
12		3
16		4
20	a	5
	b	
	c	
	d	
24	e	6
	f	
	g	
	h	
28		7

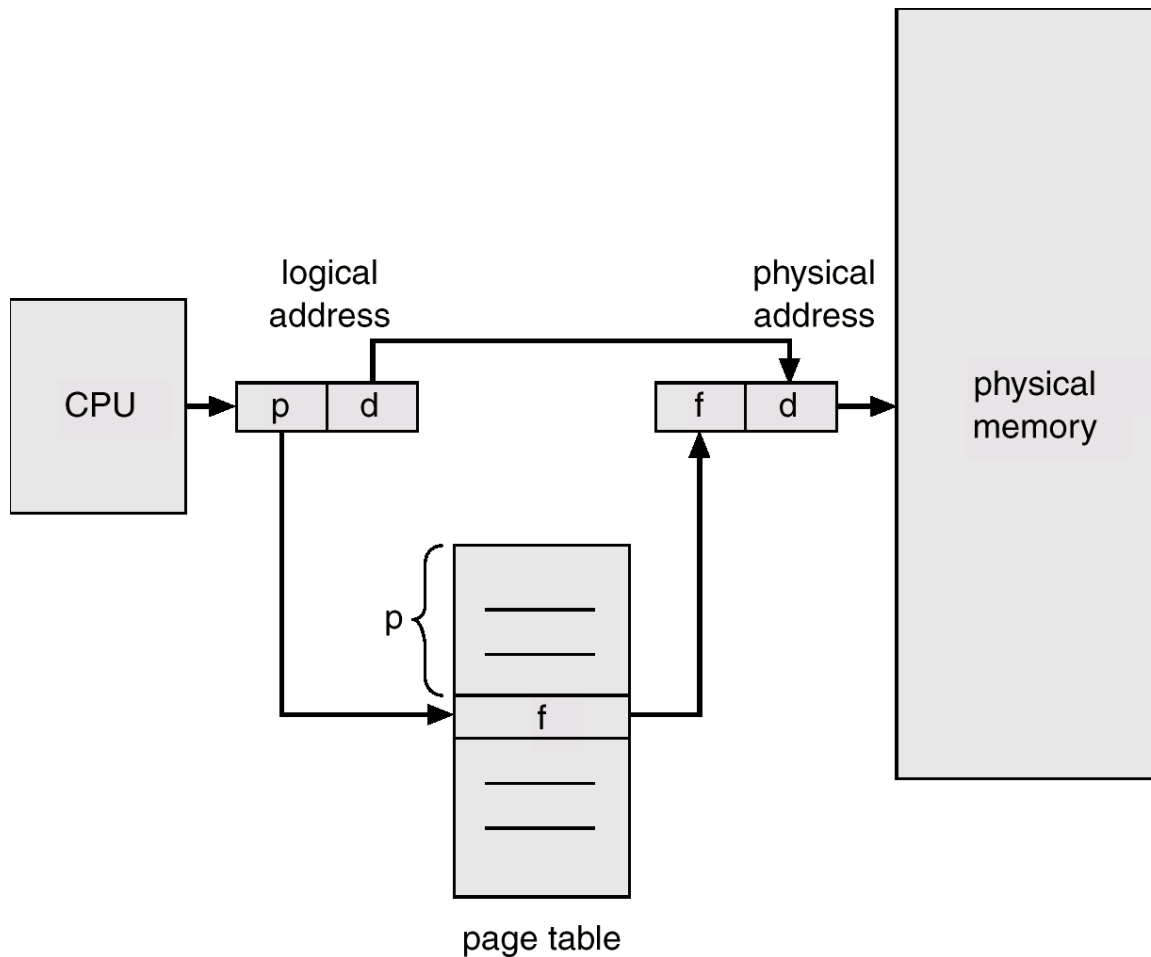
physical memory

Size of  
Physical  
memory:  
8x4 = 32 bytes  
8 Frames

# Address Translation Scheme

- Logical address contains:
  - *Page number* ( $p$ ) – used as an index into a *page table* entry which contains *frame number* in physical memory.
  - *Page offset* ( $d$ ) – combined with frame number to define the physical memory address that is sent to the memory unit.

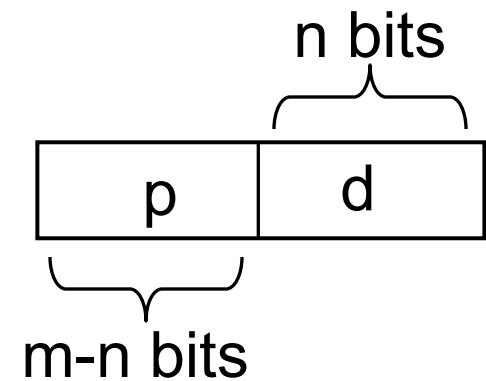
# Address Translation Scheme (Cont.)



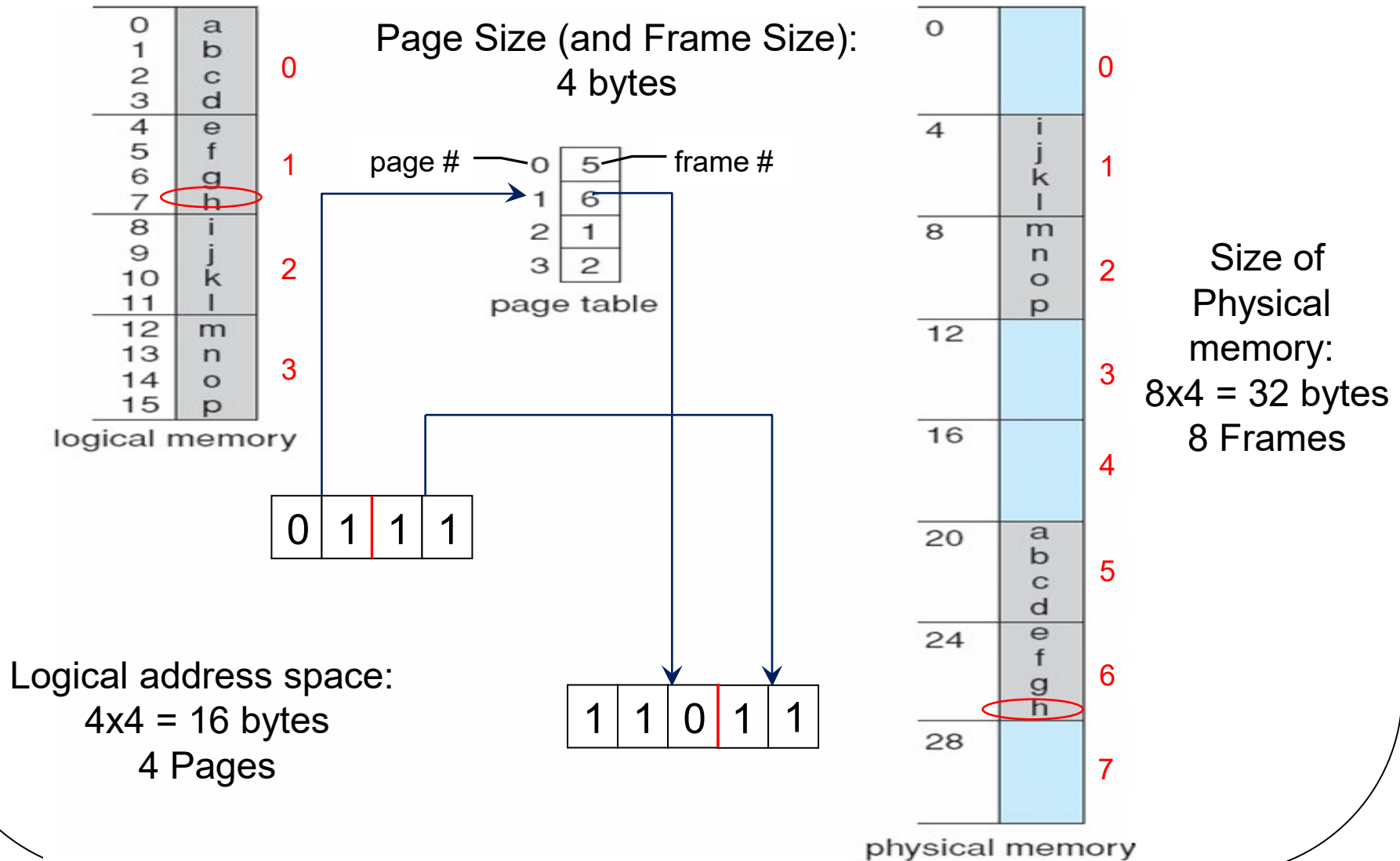
Page size:  $2^n$

Logical address space:  $2^m$

Number of pages:  $2^{m-n}$



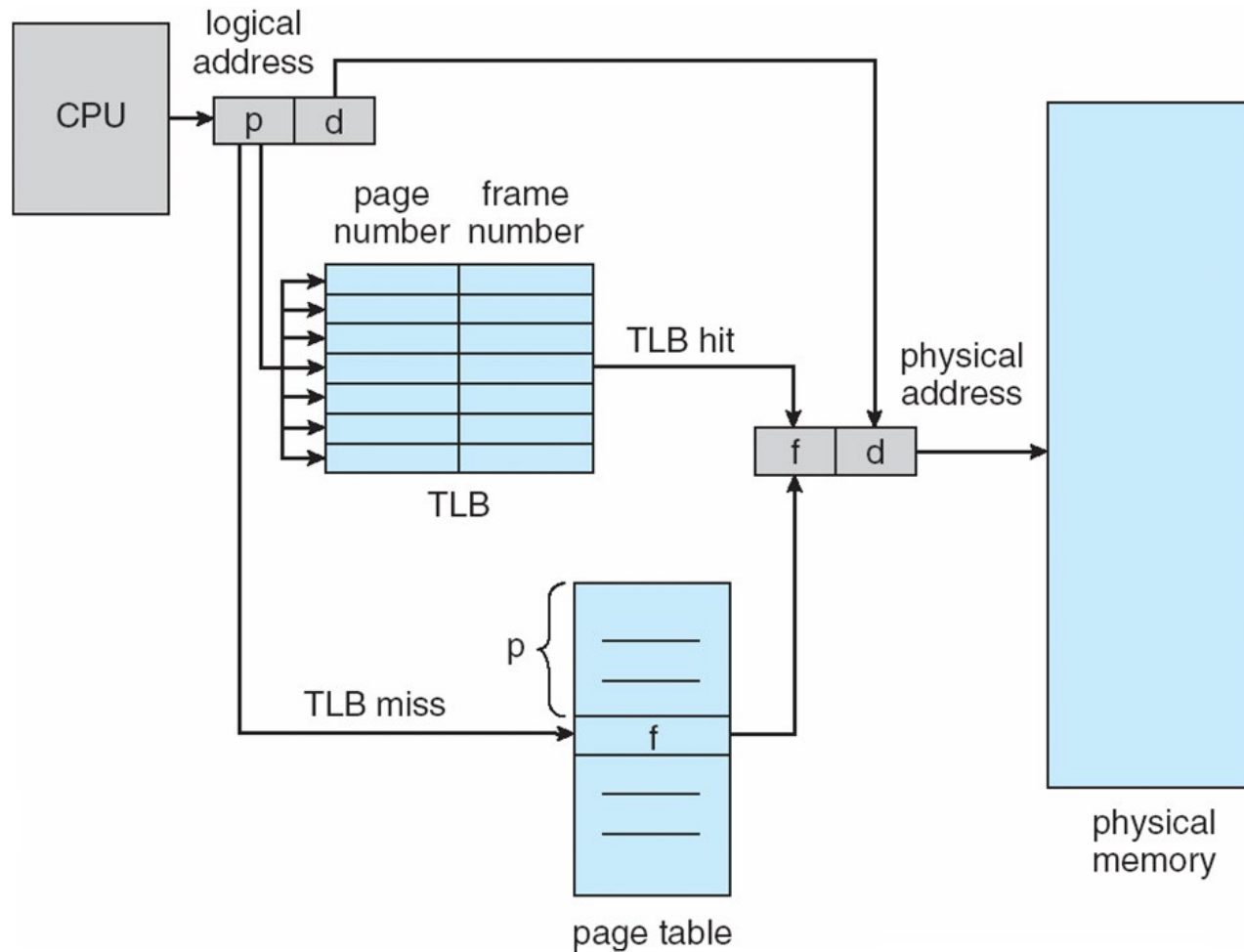
# Address Translation Scheme (Cont.)



# Implementation of Page Table

- Page table is kept in physical memory.
- *Page-table base register* (PTBR) points to the page table (for each process).
- *Page-table length register* (PTLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses: one for the page table and one for the data/instruction.
  - Effective memory access time =  $2 \mu$  (assuming that memory cycle time is  $\mu$  time unit)
- Memory access time can be reduced by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers* (TLBs)

# Paging Using TLBs

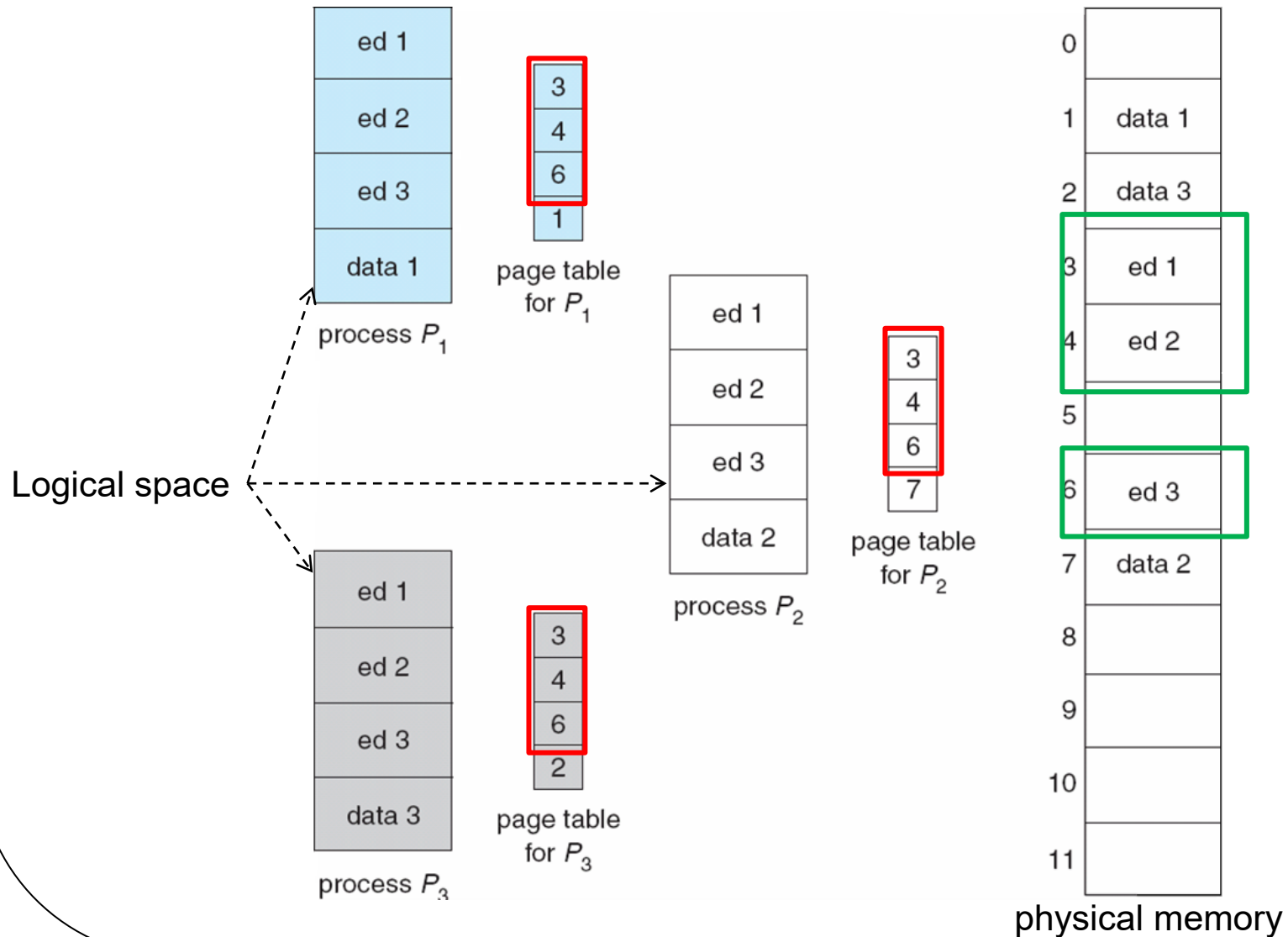


## Effective Access Time

- TLBs Lookup =  $\varepsilon$  time unit
- Assume memory cycle time is  $\mu$  time unit
- Hit ratio – percentage of times that a page number is found in the associative registers.
- Hit ratio =  $\alpha$
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (\mu + \varepsilon) \alpha + (2 \mu + \varepsilon)(1 - \alpha) \\ &= (2 - \alpha) \mu + \varepsilon \end{aligned}$$

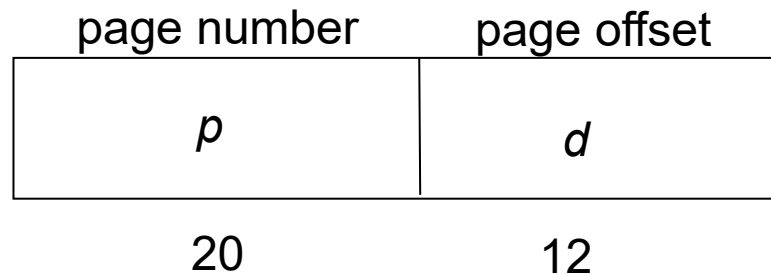
# Shared Pages





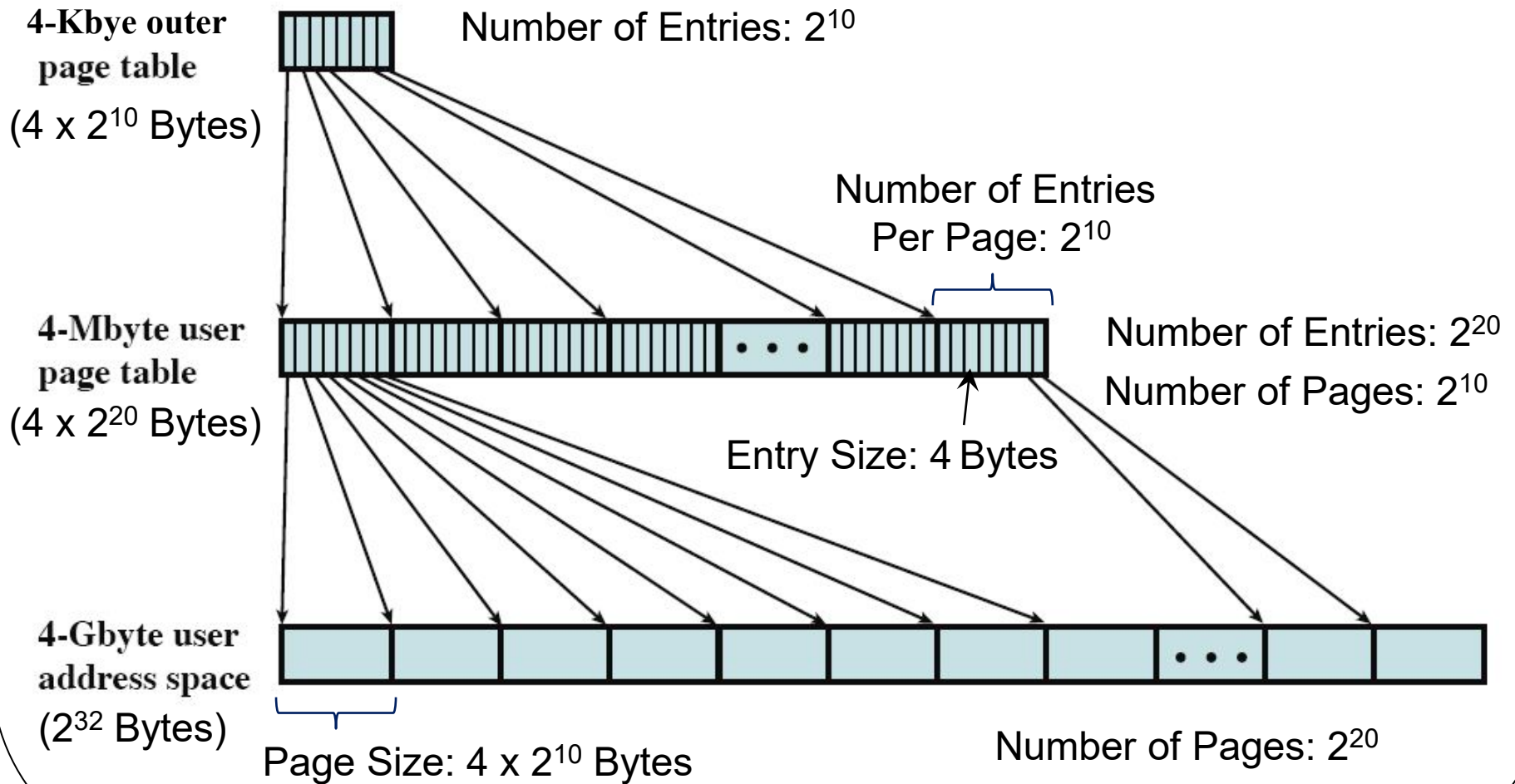
# Two-Level Page-Table Scheme

- A logical address (on 32-bit machine with 4K page size)
  - A logical address is divided into:
    - a page number consisting of 20 bits.
    - a page offset consisting of 12 bits.



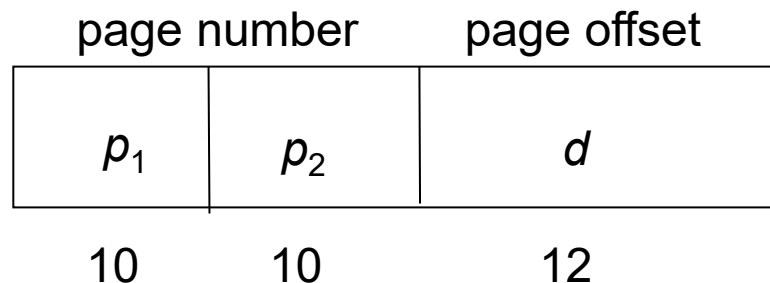
- $2^{20}$  entries in a page table
  - If each entry consists of 4 bytes, each page table occupies 4 megabyte of memory!
- *A large page table is divided up to be easier to allocate in physical memory, with a small increase in effective access time*

# Two-Level Page-Table Scheme (Cont.)



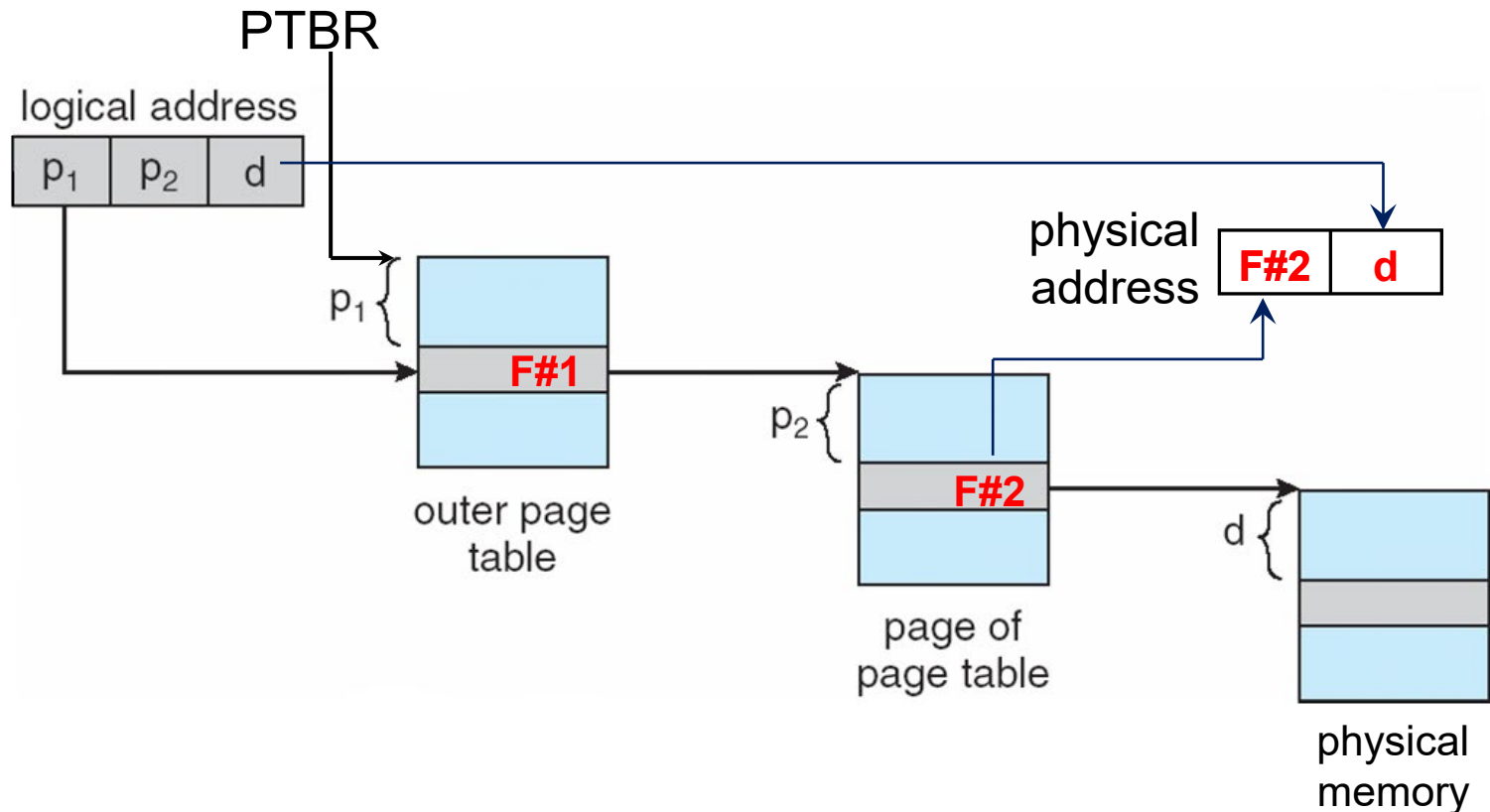
## Two-Level Page-Table Scheme (Cont.)

- Since the page table is paged, the page number is further divided into:
  - $p_1$ : an index into the outer page (second level) table
    - The outer page table contains  $(4 \times 2^{20}) / 4K = 2^{10}$  number of entries  $\rightarrow$  10 bits for  $p_1$
  - $p_2$ : is an index into a page of the inner (first level) page table
    - Each page in the page table contains  $(4 \times 2^{10}) / 4 = 2^{10}$  number of entries  $\rightarrow$  10 bits for  $p_2$
- Thus, a logical address is as follows:



## Two-Level Page-Table Scheme (Cont.)

- Address-translation scheme for a two-level 32-bit paging architecture

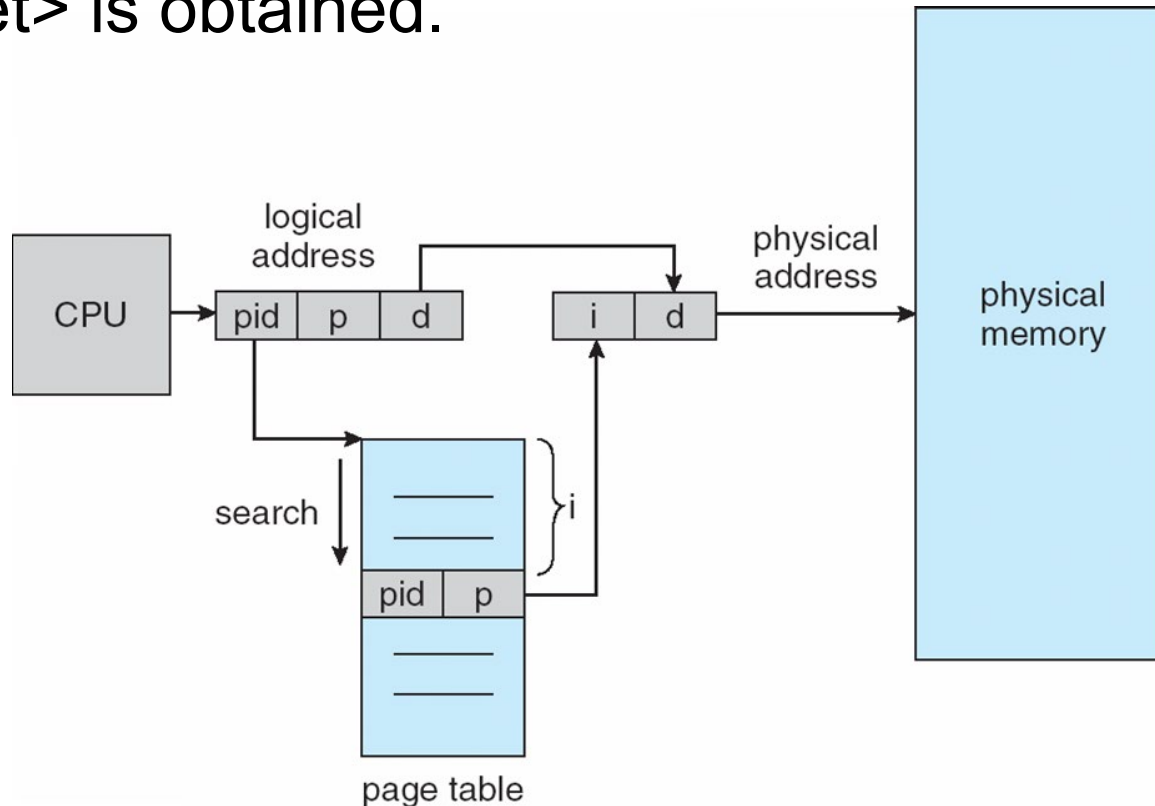


# Inverted Page Table

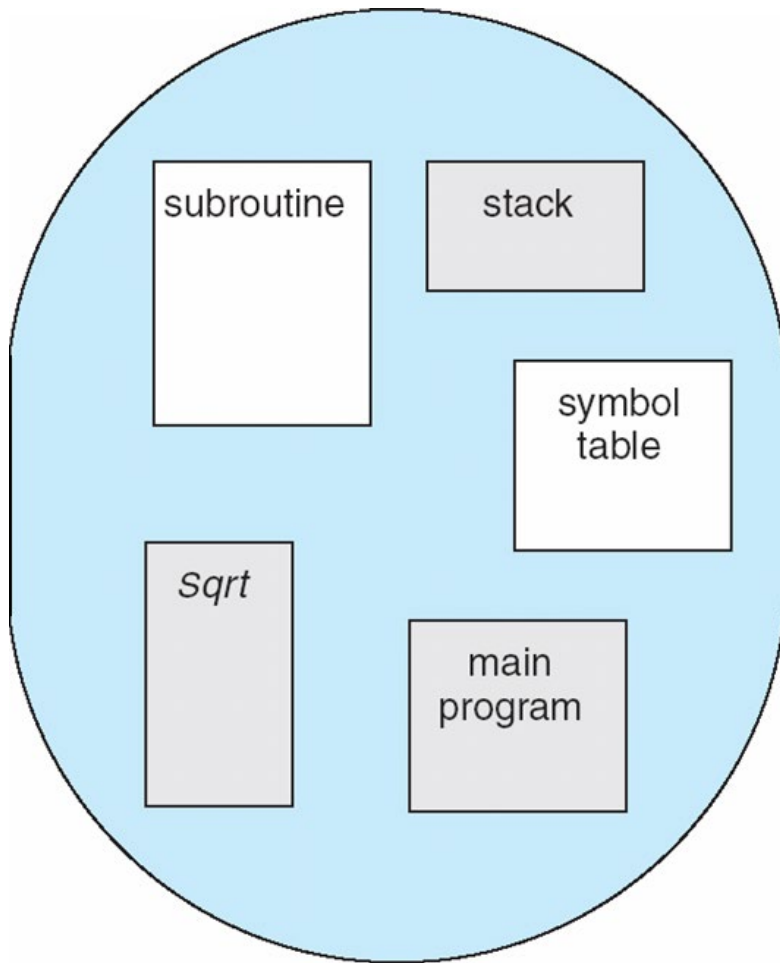
- Usually each process has its own page table. Hence the system could have many page tables, consuming substantial memory space
  - The page table size is proportional to that of the logical address space.
- **Alternative:** have a single table with one entry for each physical frame, as <process-id, page-no>. This is called an *Inverted Page Table*
- Logical address: <process-id, page-no, offset>
- Increases search time: table sorted by physical address but lookups occur on logical address

## Inverted Page Table (Cont.)

- To access memory, the pair  $\langle \text{process-id}, \text{page-no} \rangle$  is presented to inverted page table to find a match.
- If match is found, say at entry  $i$ , then physical addr  $\langle i, \text{offset} \rangle$  is obtained.



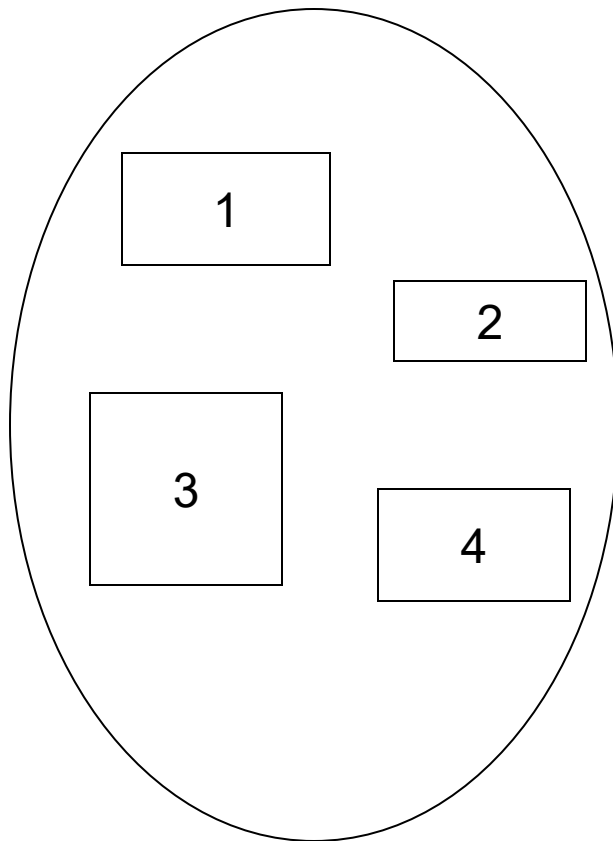
# Segmentation



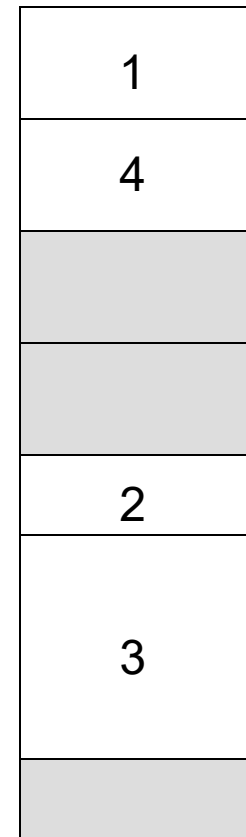
logical address space

- A memory-management scheme that break program up into its logical *segments*, and allocating space for these segments into memory separately.
- Unlike pages, segments can be of variable size
- A process has a collection of segments.
- Like pages of a process, segments of a process may not be allocated contiguously

## Segmentation (Cont.)



logical address space



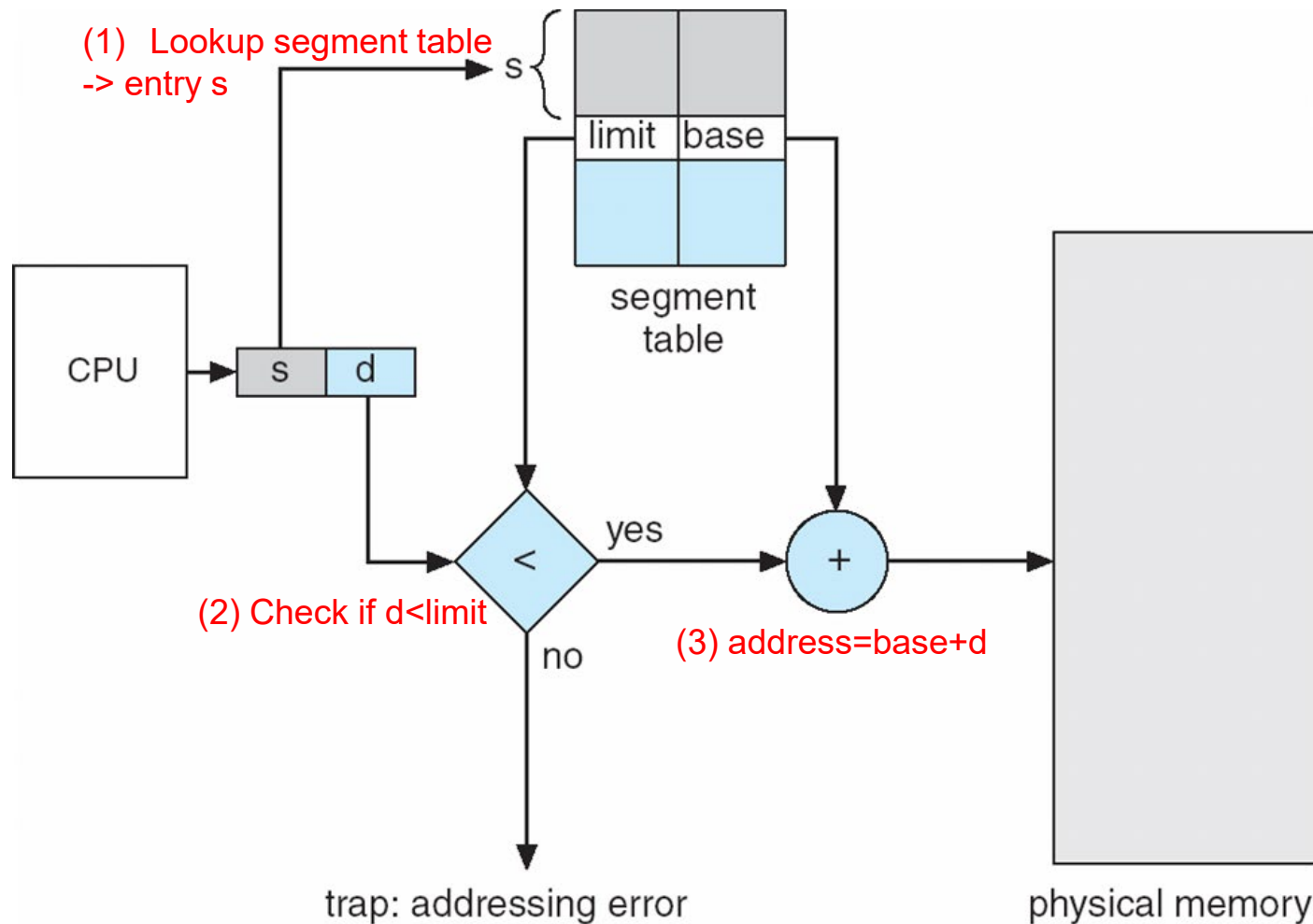
physical memory space



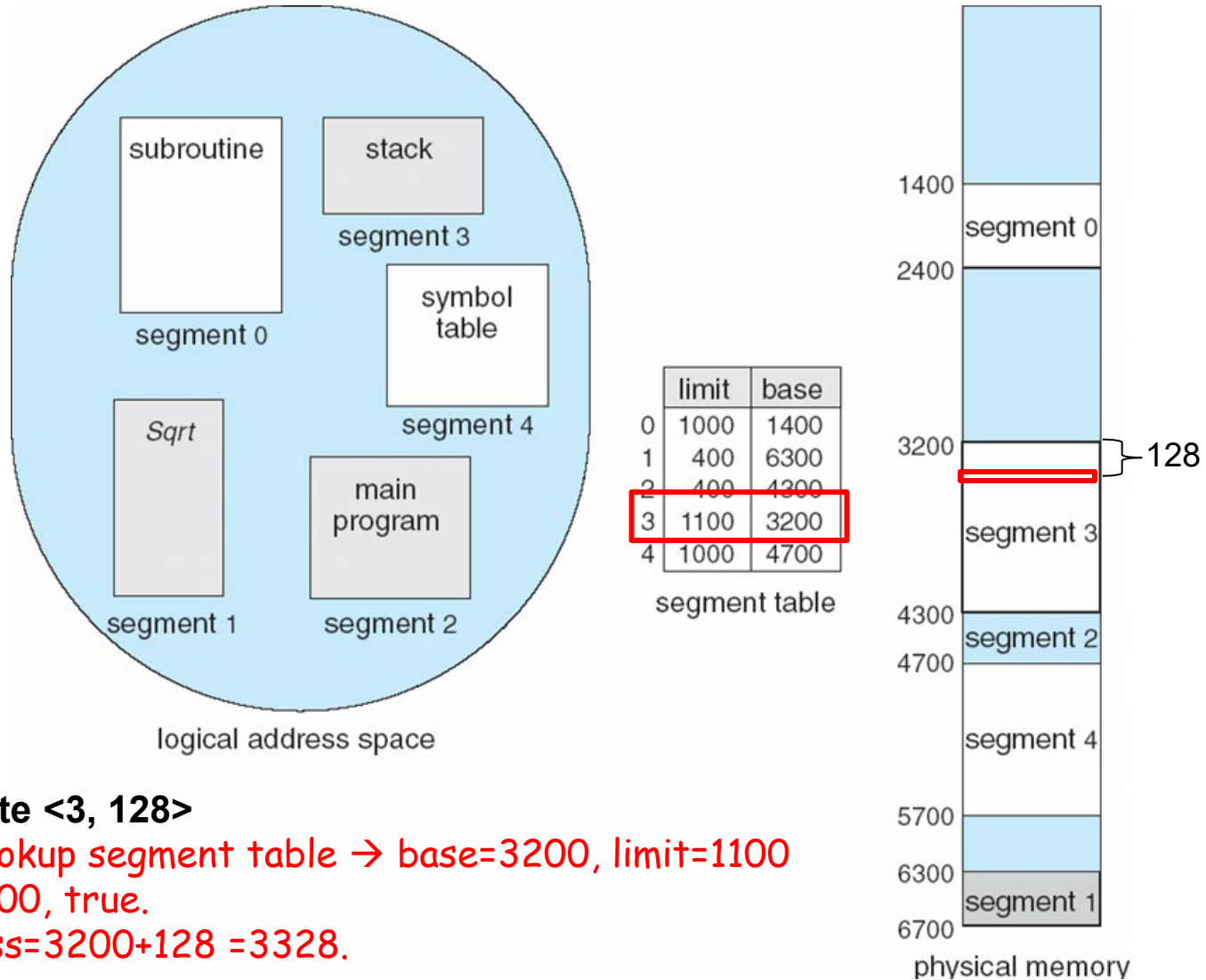
# Address Translation

- Each segment has a segment no. & offset, i.e., its logical address is:  
    <segment-no, offset>,
- *Segment table*. 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;

# Address Translation (Cont.)



# Address Translation (Cont.)



**Q: translate <3, 128>**

**Ans: (1) lookup segment table → base=3200, limit=1100**

**(2)  $128 < 1100$ , true.**

**(3)  $\text{address} = 3200 + 128 = 3328$ .**

## Fragmentation in Segmentation

- Since segments vary in length, memory allocation is a dynamic storage-allocation problem. Usually use best-fit or first-fit.
- Suffer from external fragmentation as process leaves the system, its occupied segments become holes in the memory.
- As a process leaves the system, its occupied segments become holes of varying sizes in the memory.

# Summary

- Programs must be brought into memory for execution  $\Rightarrow$  Memory Allocation
  - Contiguous Allocation
    - fix partition vs. dynamic partition
  - Non-contiguous Allocation
    - paging vs. segmentation
  - Fragmentation Problem

## Summary (Cont.)

- Process executes in its logical address space, but the actual code and data are stored in physical memory  $\Rightarrow$  Mapping of logical address to physical address
  - Paging
    - basic scheme
    - using translation look-aside buffer
    - double-level paging
    - inverted page table
  - Segmentation
    - basic scheme