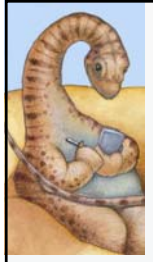
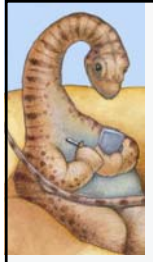
Chapter 8: Memory Management



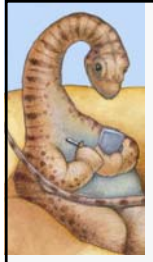
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation



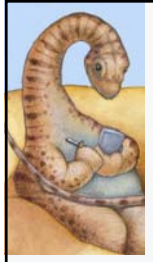
Background

- Program must be brought into memory and placed within a process for it to be run
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run



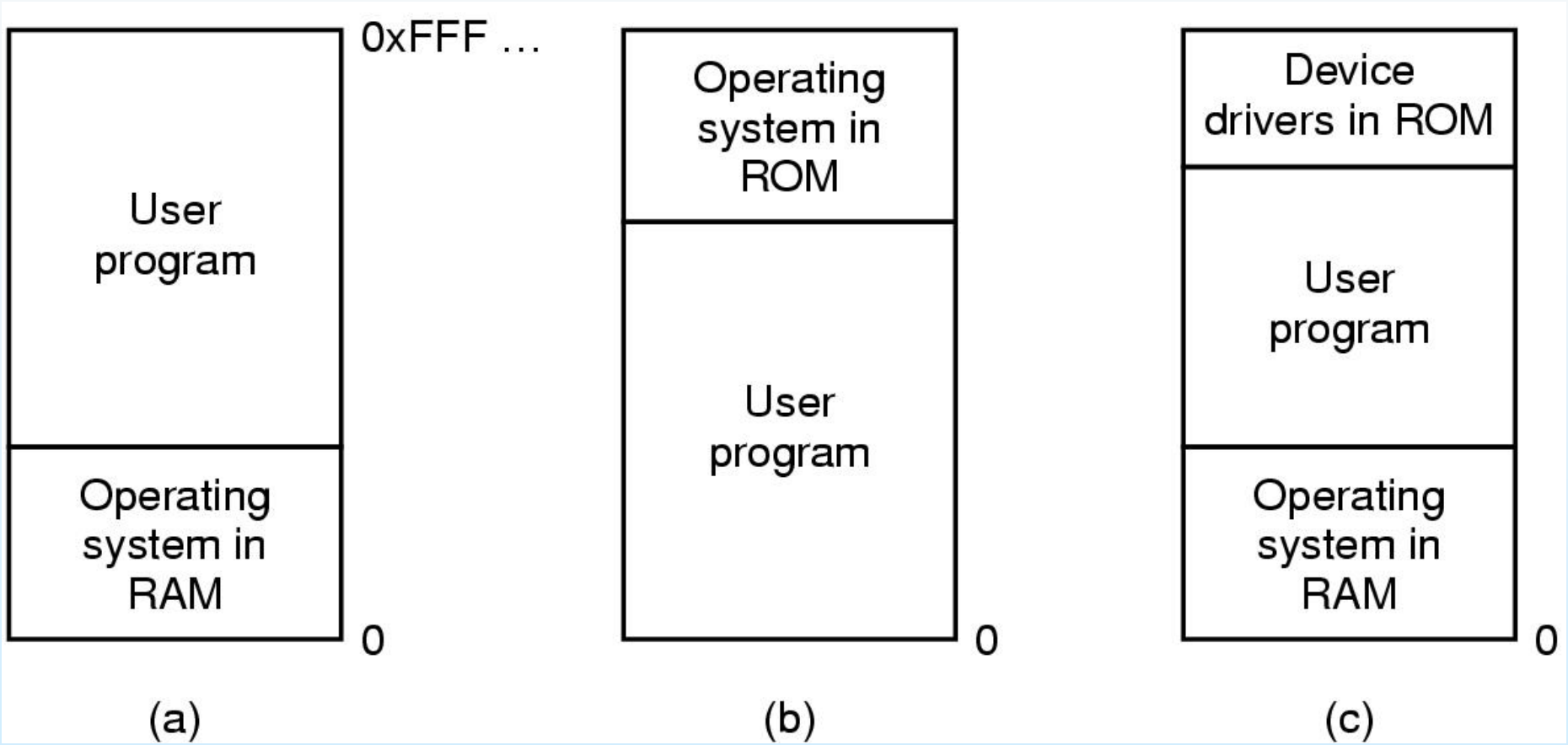
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy



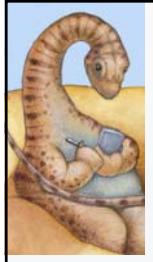
Basic Memory Management

Monoprogramming without Swapping or Paging

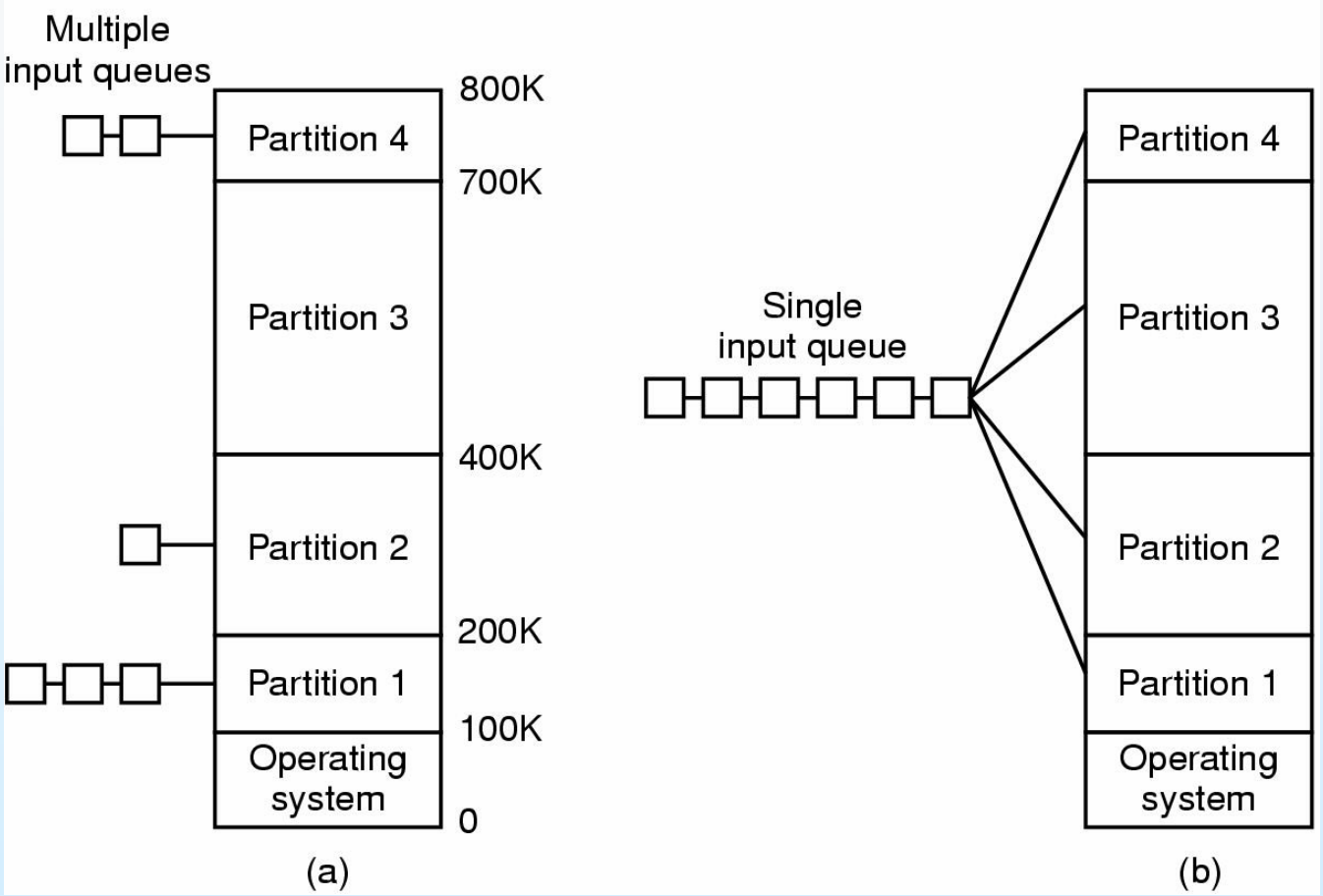


Three simple ways of organizing memory

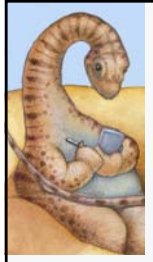
- an operating system with one user process



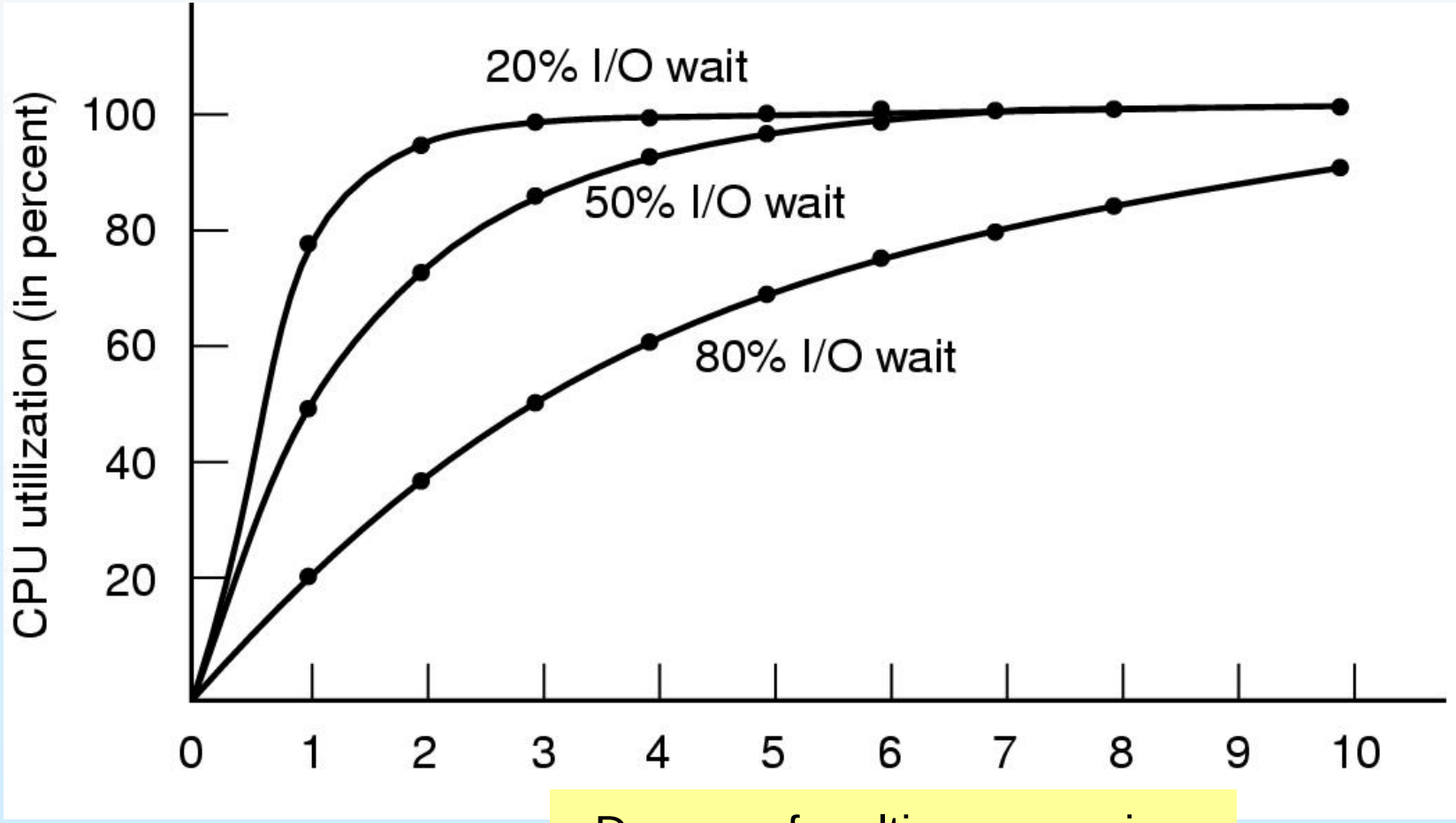
Multiprogramming with Fixed Partitions



- Fixed memory partitions
  - separate input queues for each partition
  - single input queue

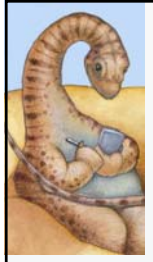


Modeling Multiprogramming



CPU utilization as a function of number of processes in memory is modeled as

CPU utilization = $1-p^n$, where p is the fraction of time waiting for I/O



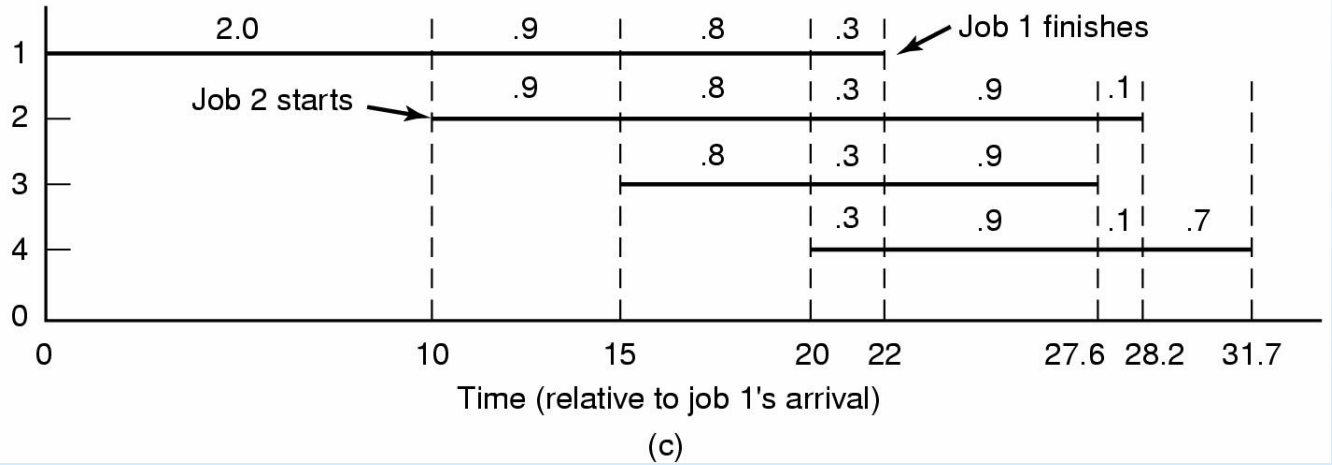
Analysis of Multiprogramming System Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

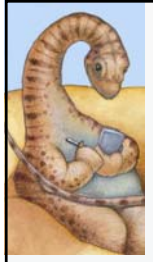
	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

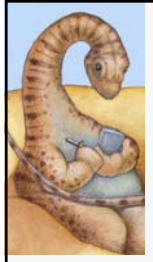
- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval



Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions

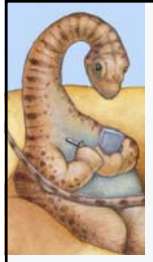
- Use base and limit values
  - address locations added to base value to map to physical address
  - address locations larger than limit value is an error



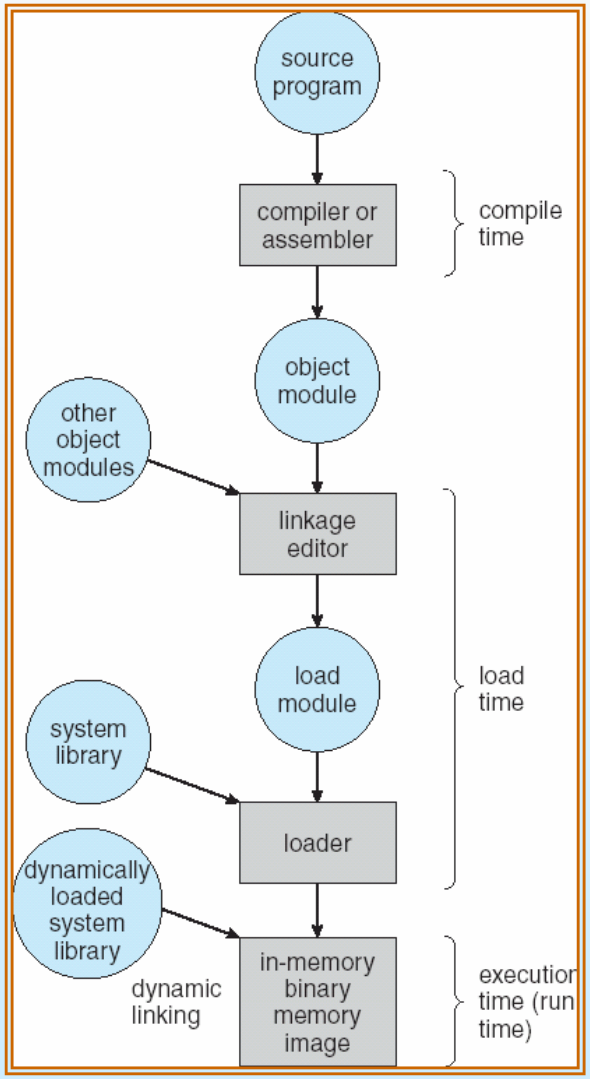
Binding of Instructions and Data to Memory

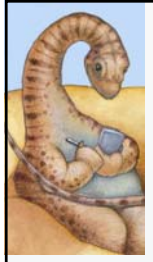
Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
- **Load time**: Must generate *relocatable code* if memory location is not known at compile 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*).



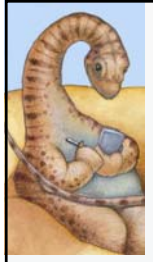
Multistep Processing of a User Program





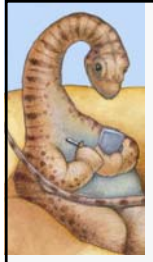
Logical vs. Physical Address Space

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as *virtual address*
  - **Physical address** – address seen by the memory unit
- Compile-time and load-time address-binding schemes generate identical logical and physical addresses ;
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme

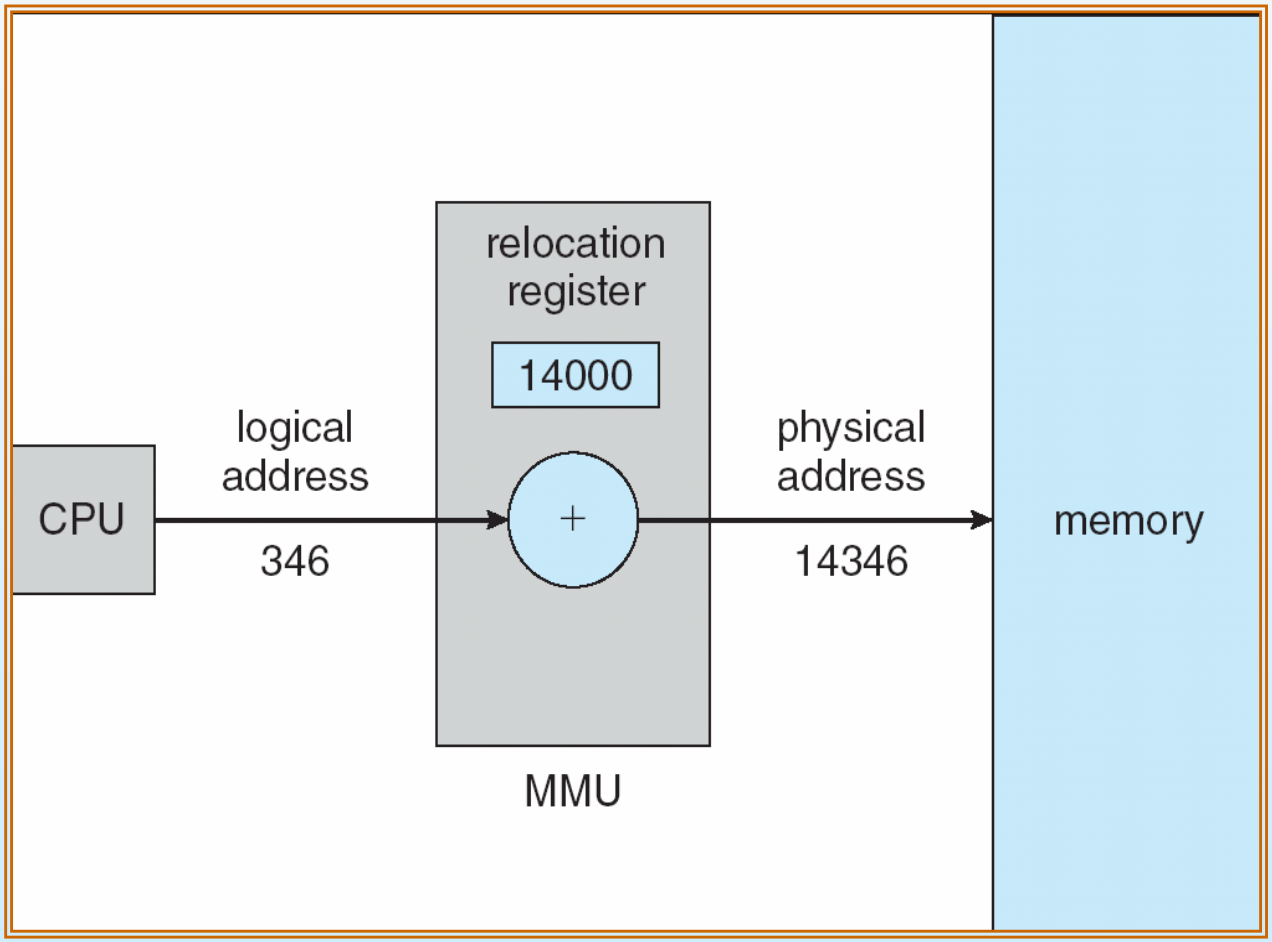


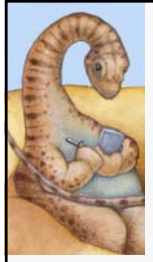
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses



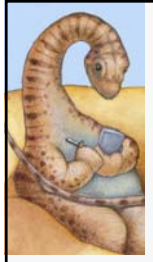
Dynamic relocation using a relocation register





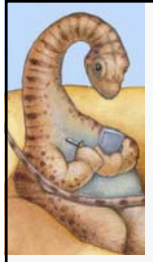
Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design



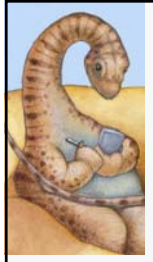
Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for language subroutine libraries
  - Without this, each program must include a copy of its language library in the executable image.
    - This wastes both disk space and main memory.

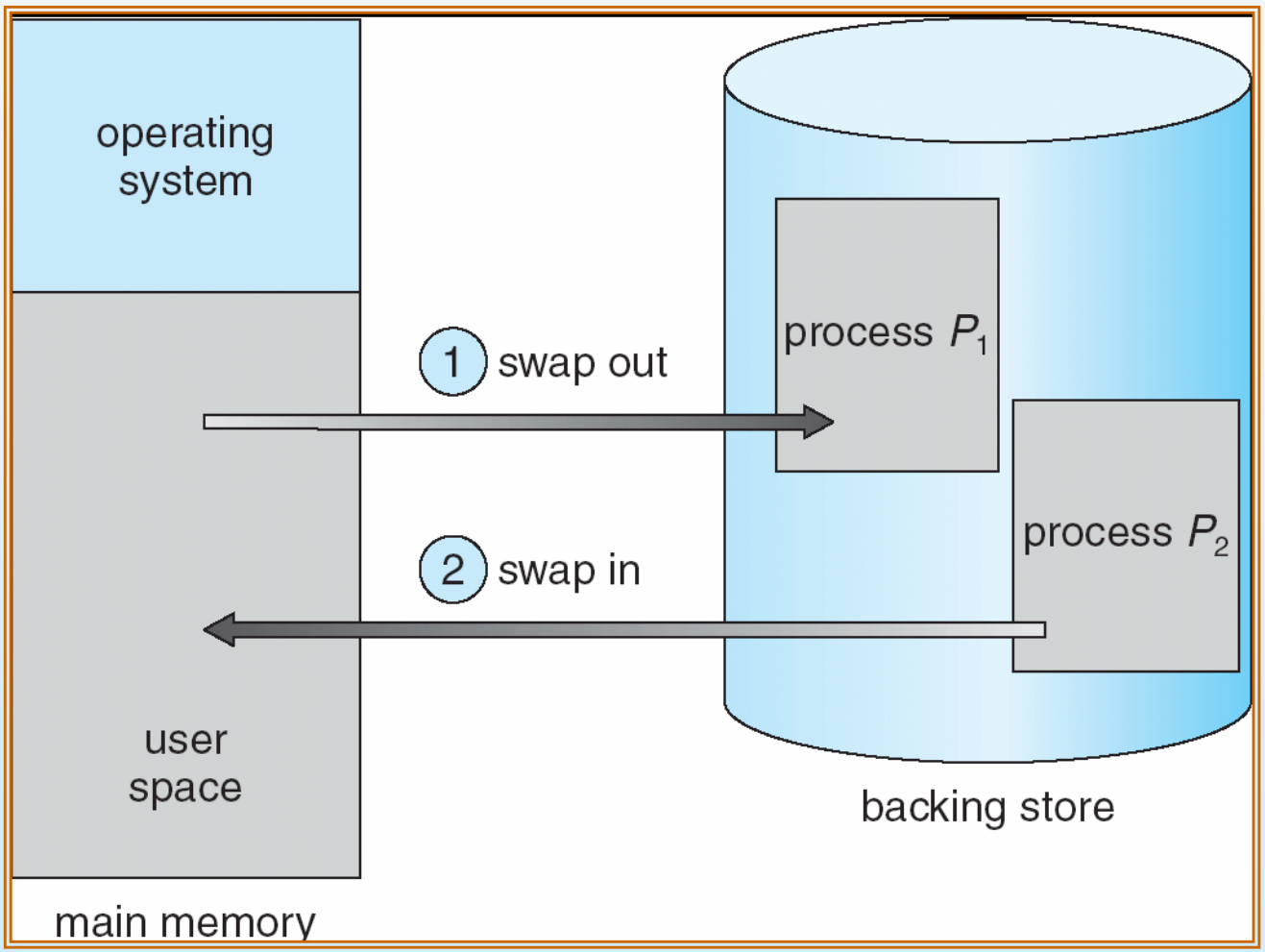


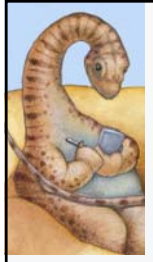
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)



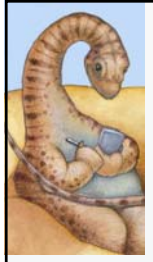
Schematic View of Swapping



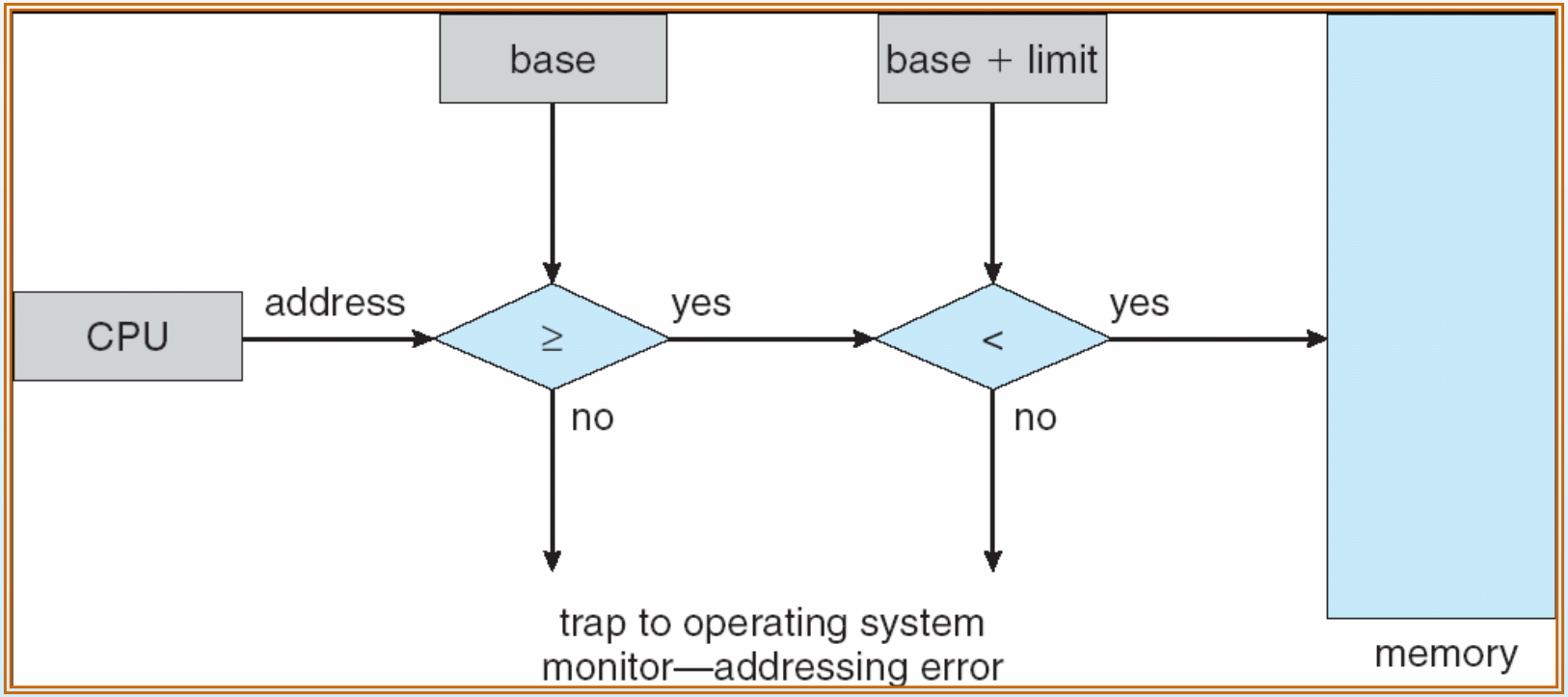


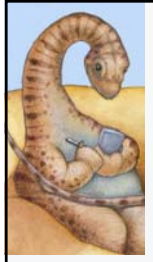
Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Single-partition allocation
  - Each process is contained in a single contiguous section of memory
  - Relocation-register scheme with a limit register used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register



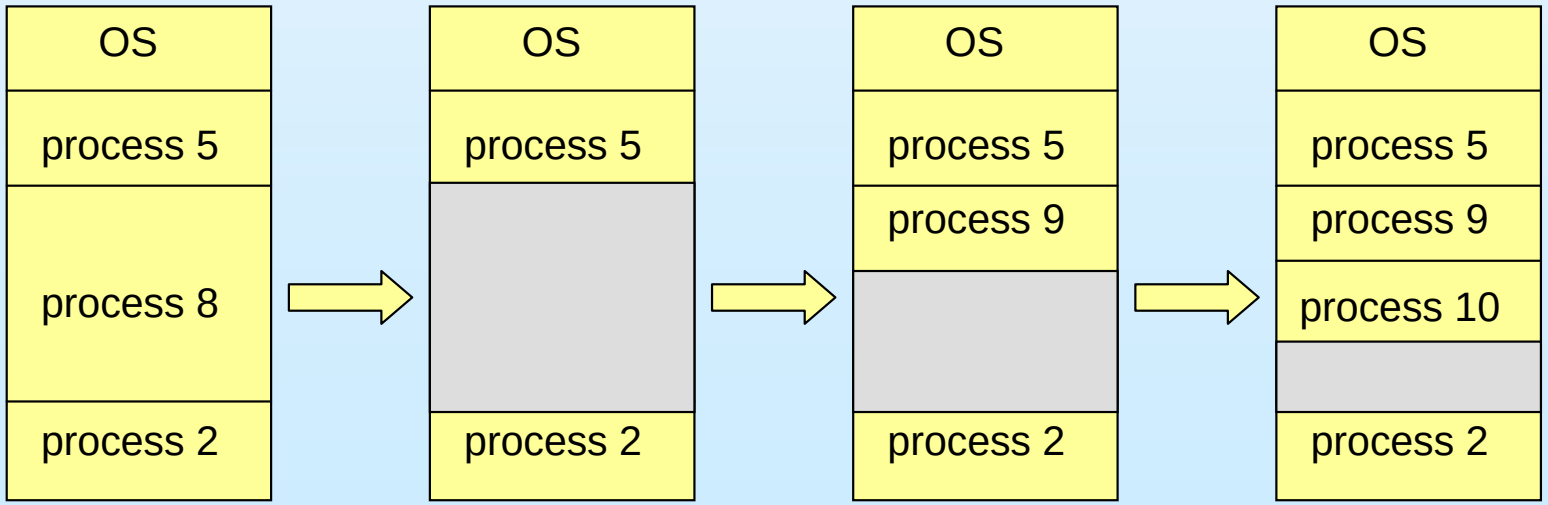
HW support for relocation and limit registers

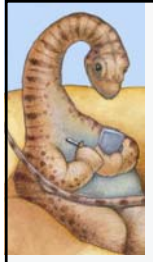




Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - *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:
    a) allocated partitions b) free partitions (hole)



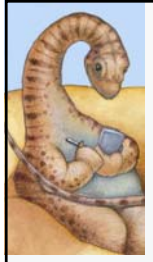


Memory Allocation

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

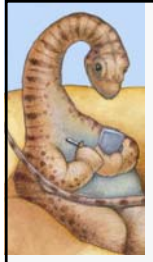
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization



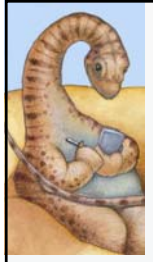
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - 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



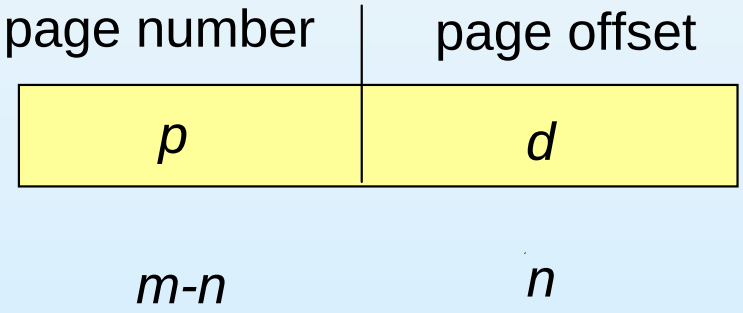
Paging

- Logical address space of 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**.
- 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
- Internal fragmentation

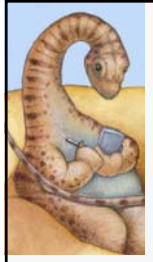


Address Translation Scheme

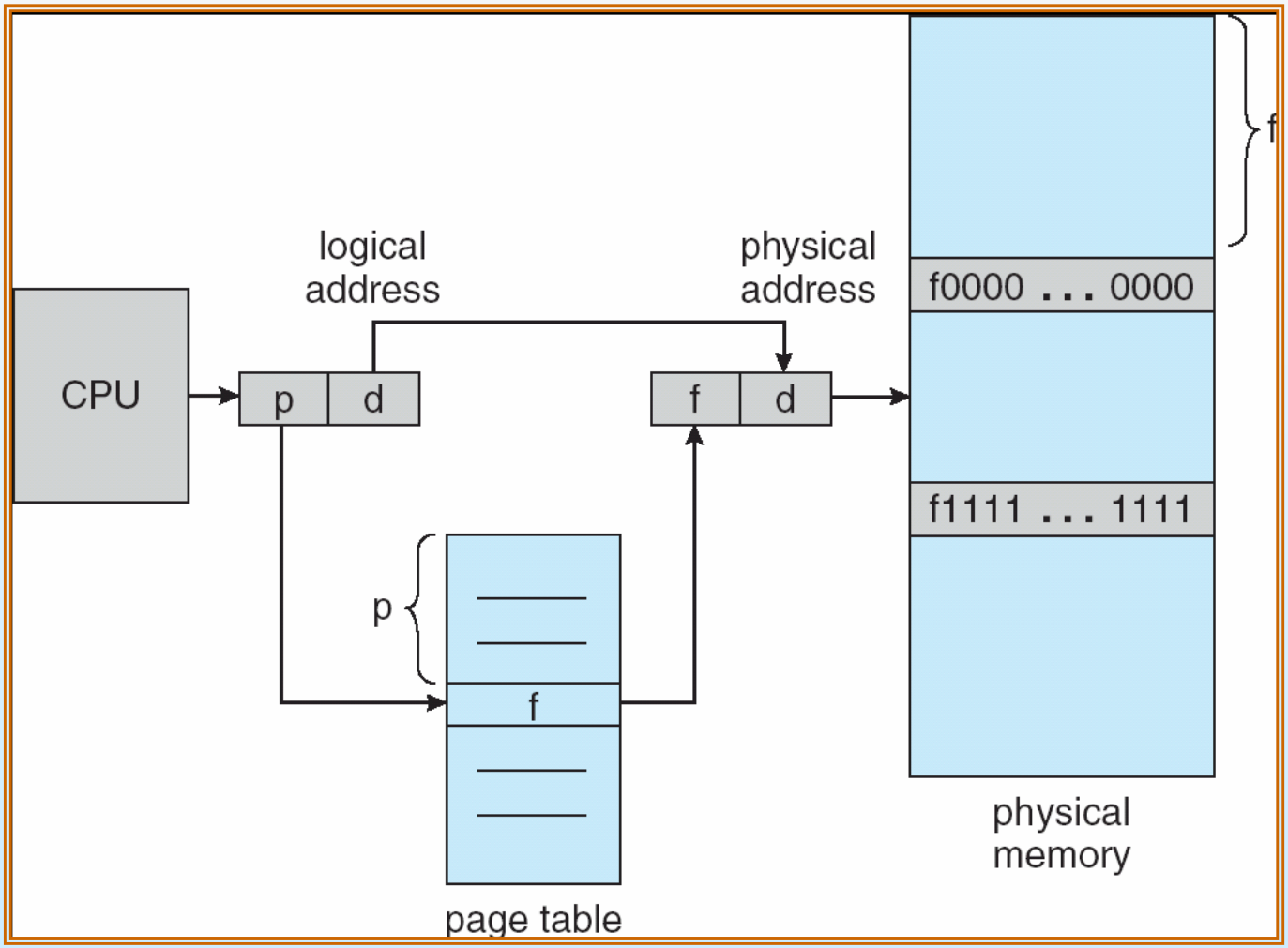
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.

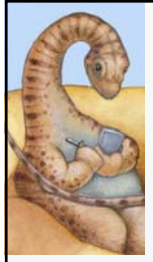


  - Logical address space size = 2^m
  - page size = 2^n
  - 2^{m-n} different page numbers

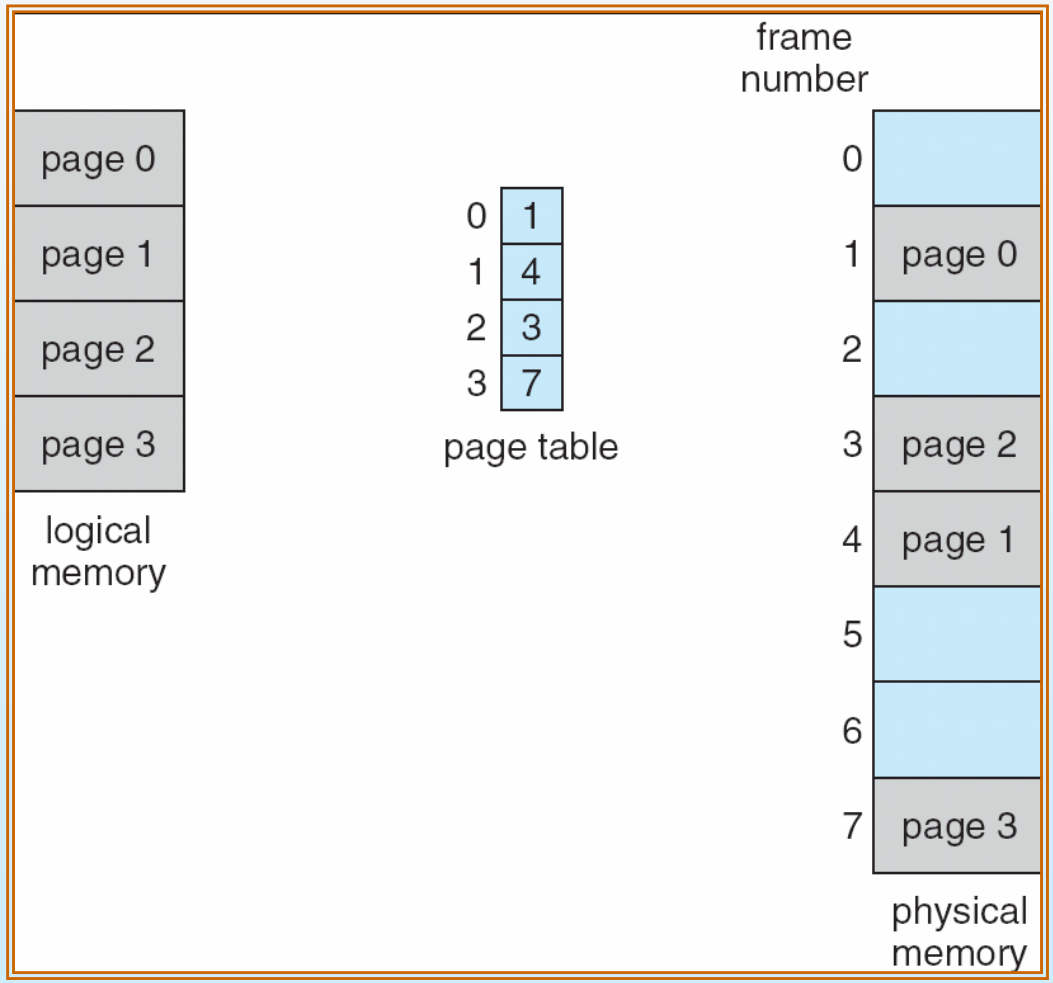


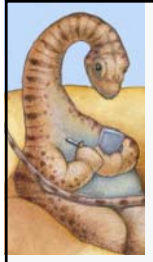
Paging hardware





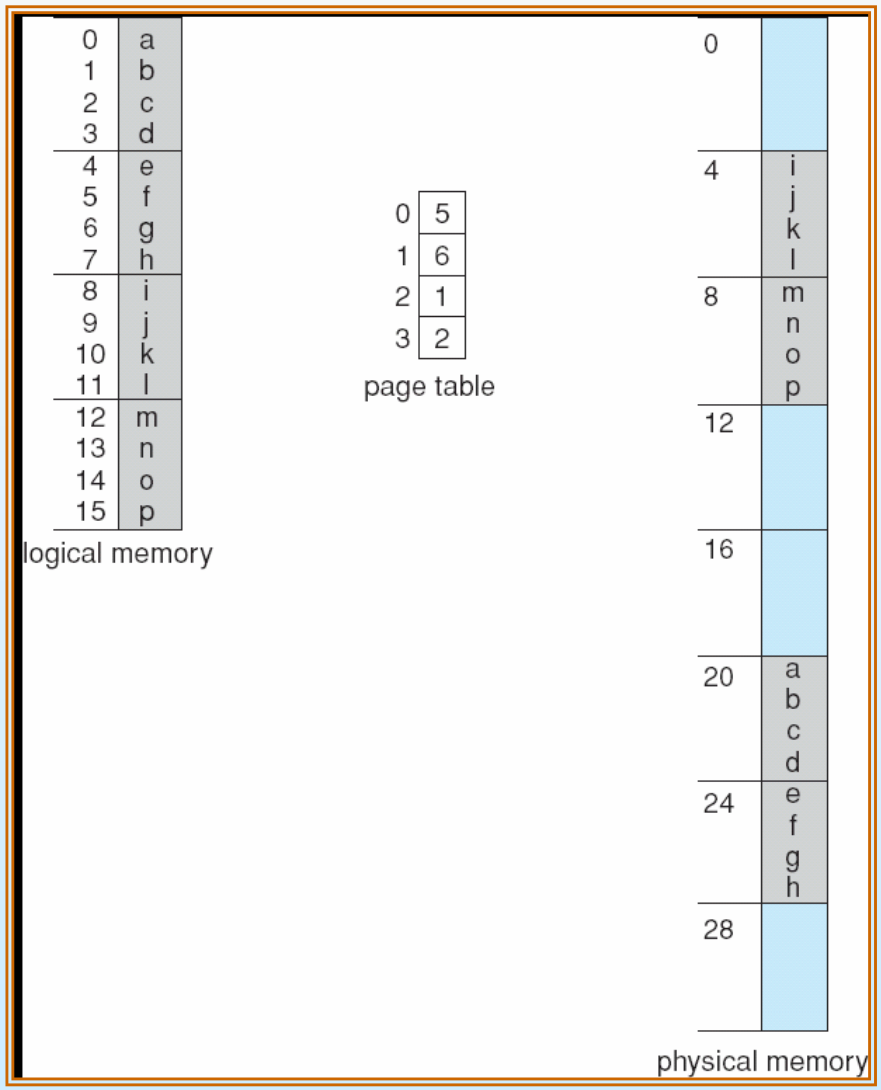
Paging Model of logical and physical memory

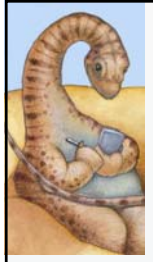




Paging Example

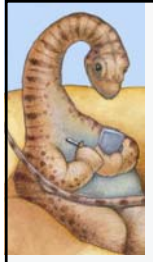
page size = 4 bytes





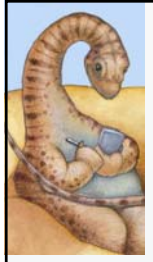
Fragmentation

- No external fragmentation
- Internal fragmentation exists
  - Page size = *2048* bytes;
  - A process of *72766* bytes will need *35* pages plus *1086* bytes
    - Total 36 frames allocated and resulting in an internal fragmentation of *2048-1086=962* bytes.
- The worst case
  - *n* pages + *1* byte $\rightarrow$ *n+1* frames allocated

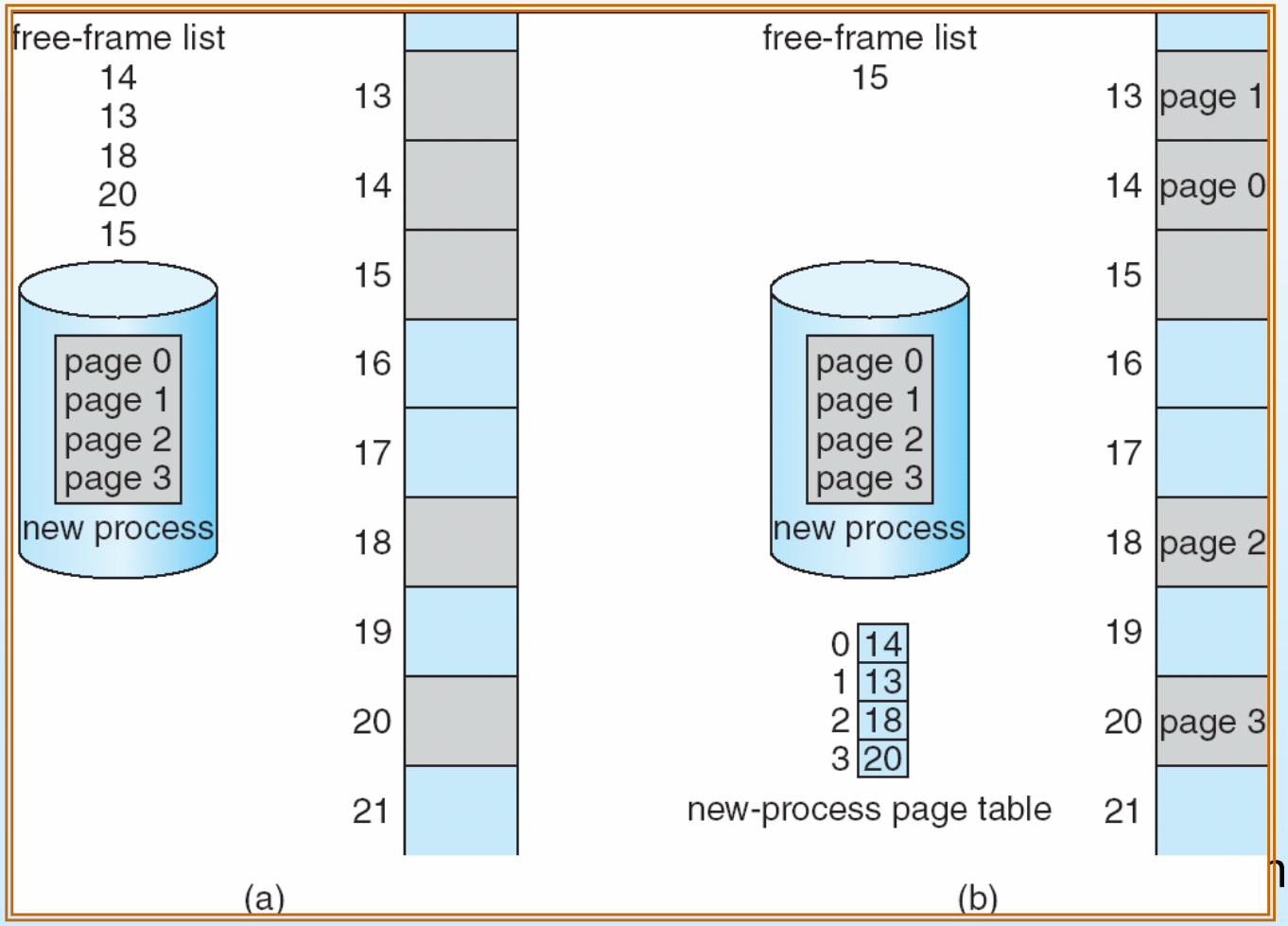


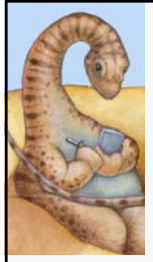
Page and Page Table Entry Size

- Pages typically are between 4 KB and 8 KB
  - Some support multiple page sizes
  - Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages.
- Usually, each page table entry is 4 bytes long
  - 32-bit can address 2^{32} frames
  - If frame size is 4 KB, then it can address 2^{44} bytes (16 TB) of physical memory



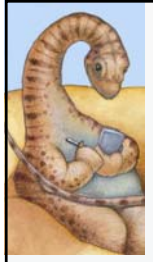
Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)



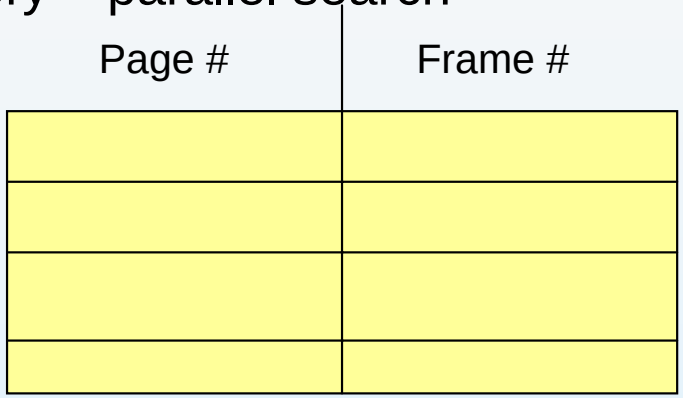
Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *Page-table length register* (PRLR) 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.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**



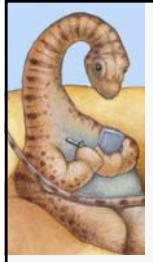
Associative Memory

- Associative memory – parallel search

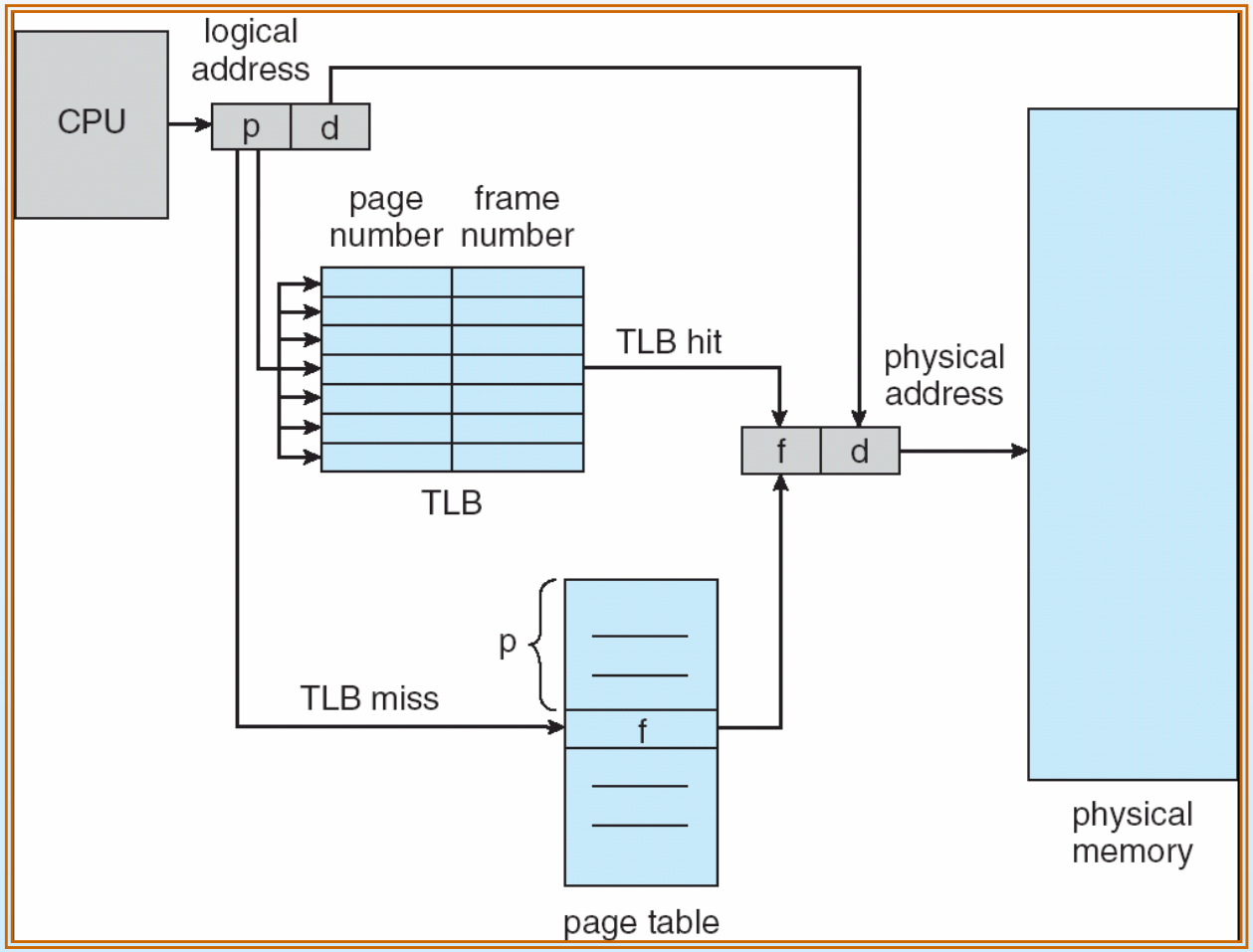


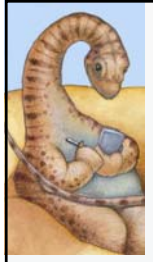
Address translation (A´, A´´)

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



Paging Hardware With TLB





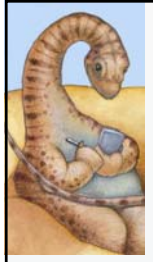
Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is *t* time unit
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$$

$$= 2t - \alpha t + \varepsilon$$

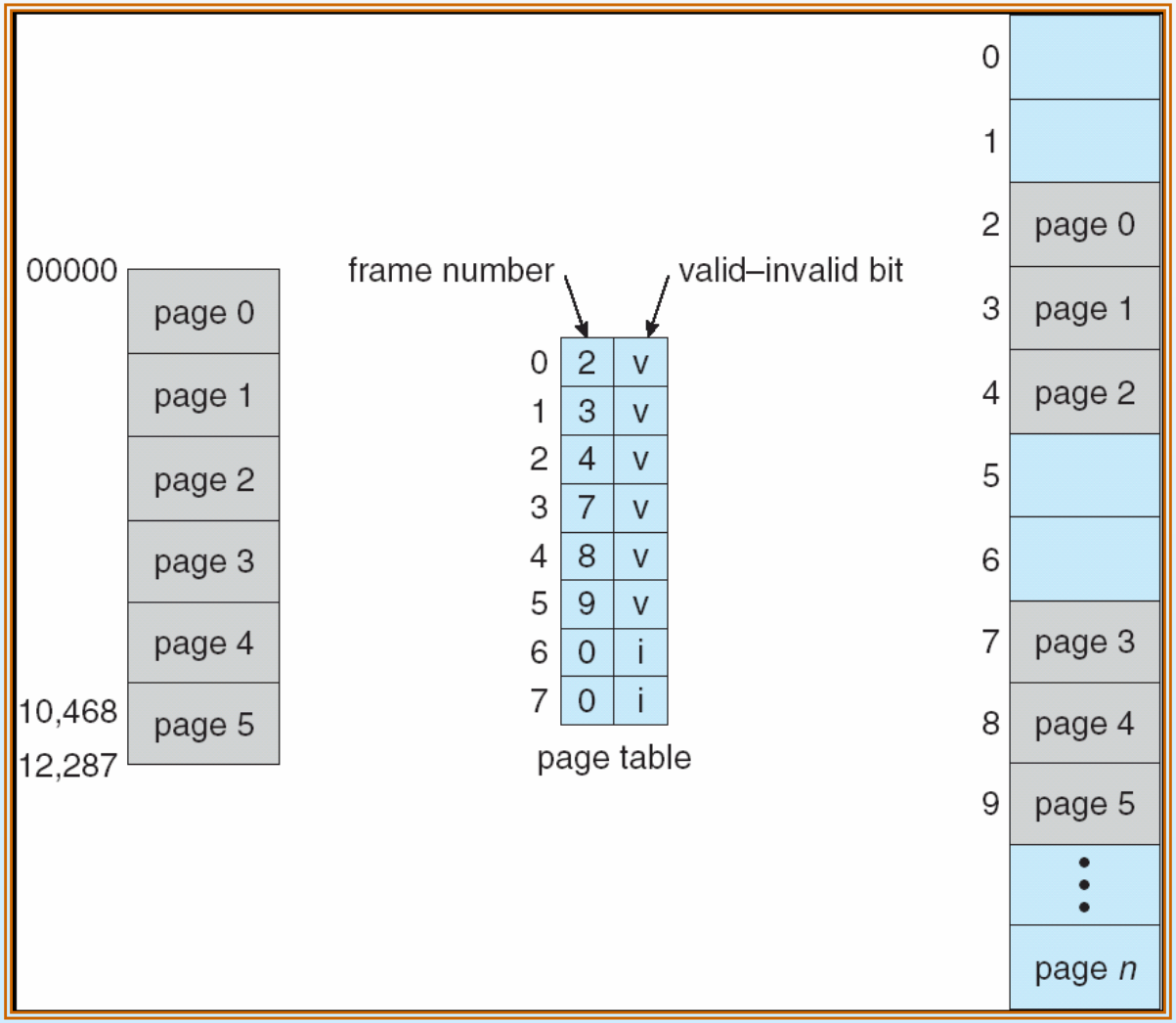
- t = 100 ns and ε = 20 ns
  - 80% hit ratio $\rightarrow$ EAT = 140 ns
  - 98% hit ratio $\rightarrow$ EAT = 122 ns

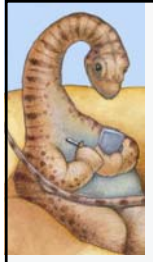


Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space

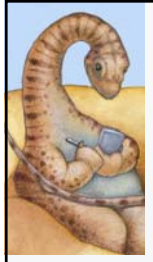
Valid (v) or Invalid (i) Bit In A Page Table



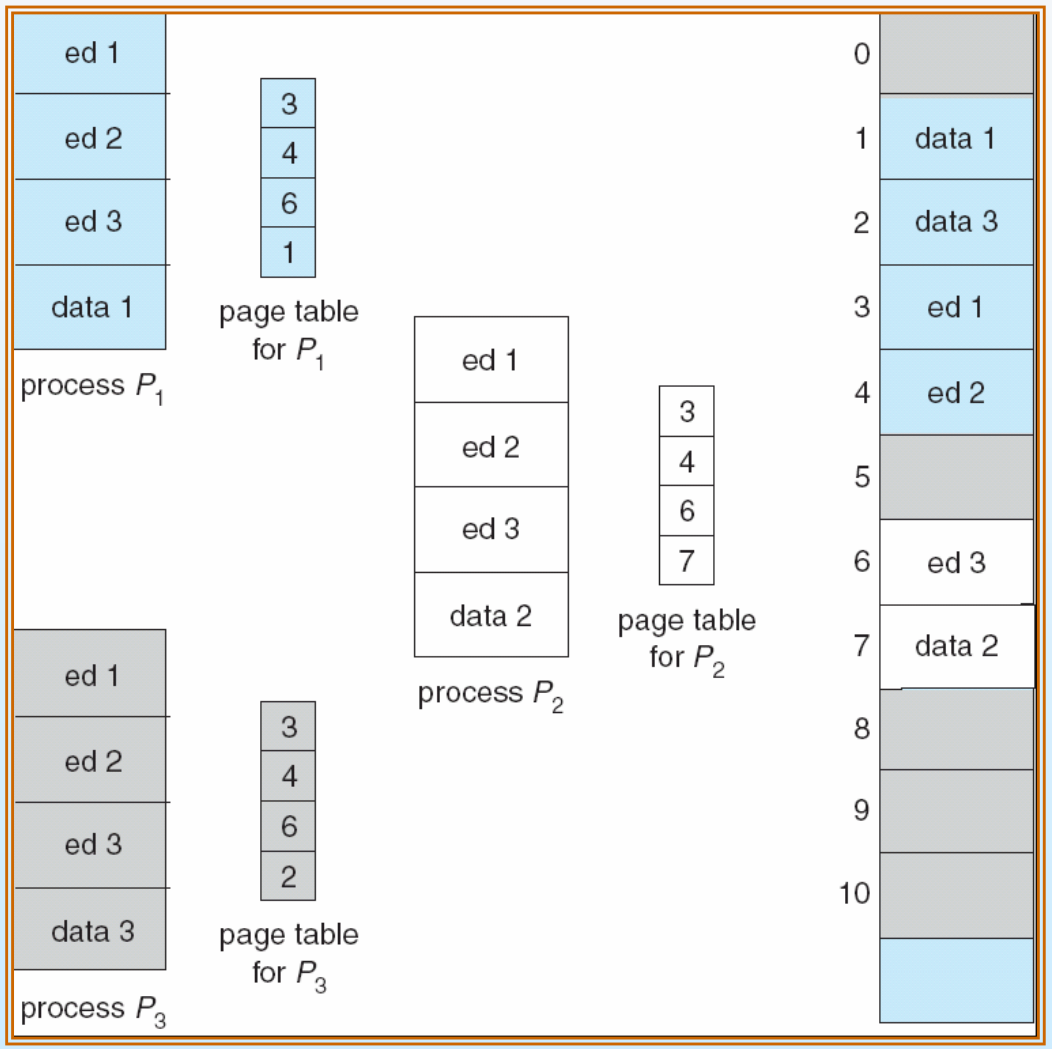


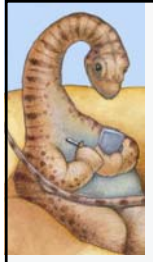
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
- Other heavily used programs can also be shared
  - Compilers, window systems, run-time libraries, etc.
  - To be sharable, the code must be reentrant.



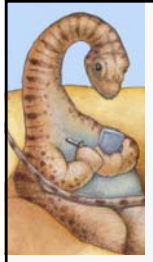
Sharing of code in a paging environment





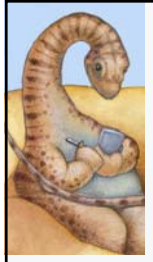
Structure of Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

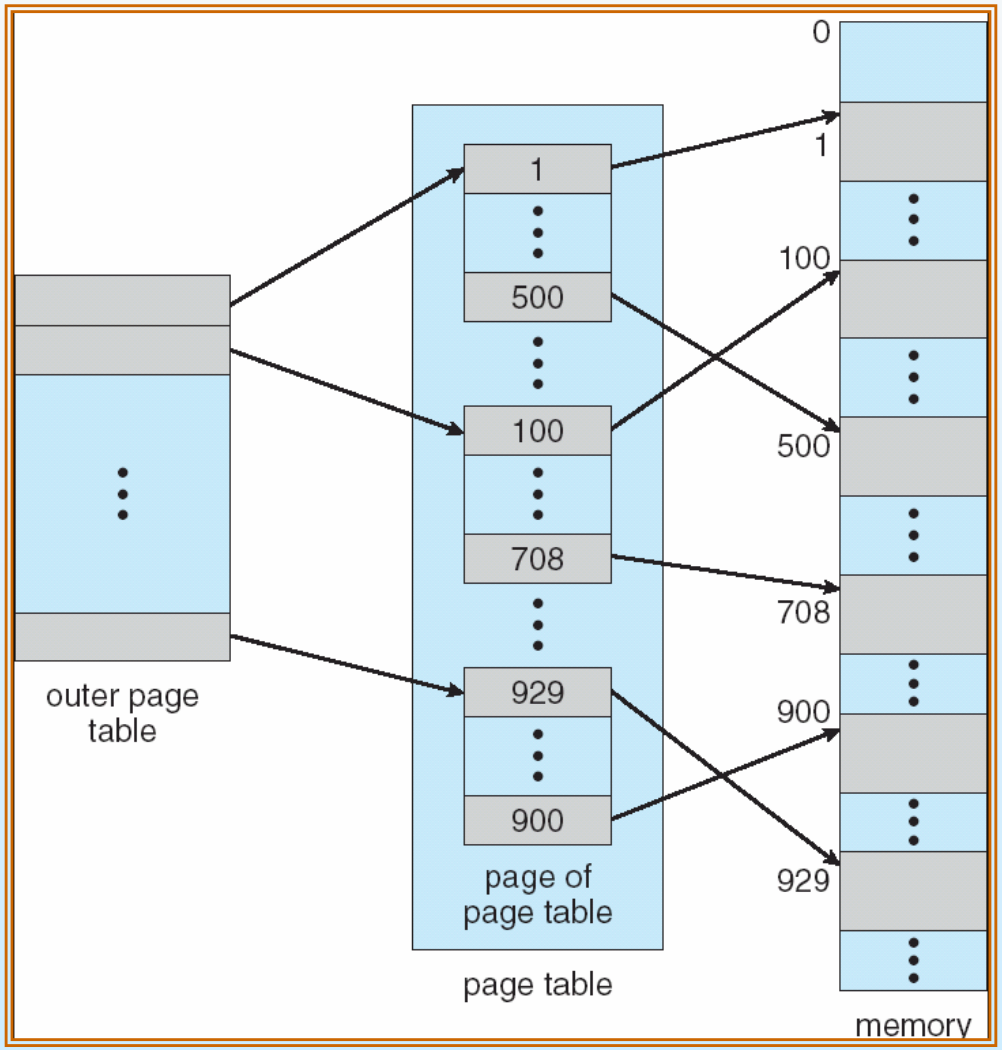


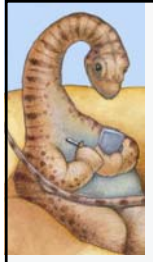
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
    - may contain 1 million entries (2^{20})
    - If size of each entry is 4 bytes, page table along takes up 4 MB of physical address space
  - a page offset consisting of 12 bits
- A simple technique is a two-level page table



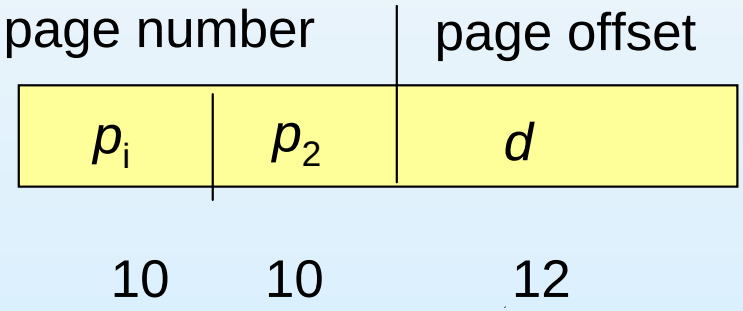
Two-Level Page-Table Scheme



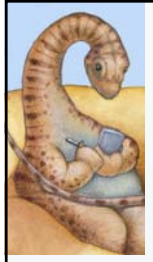


Two-Level Paging Example, cont.

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

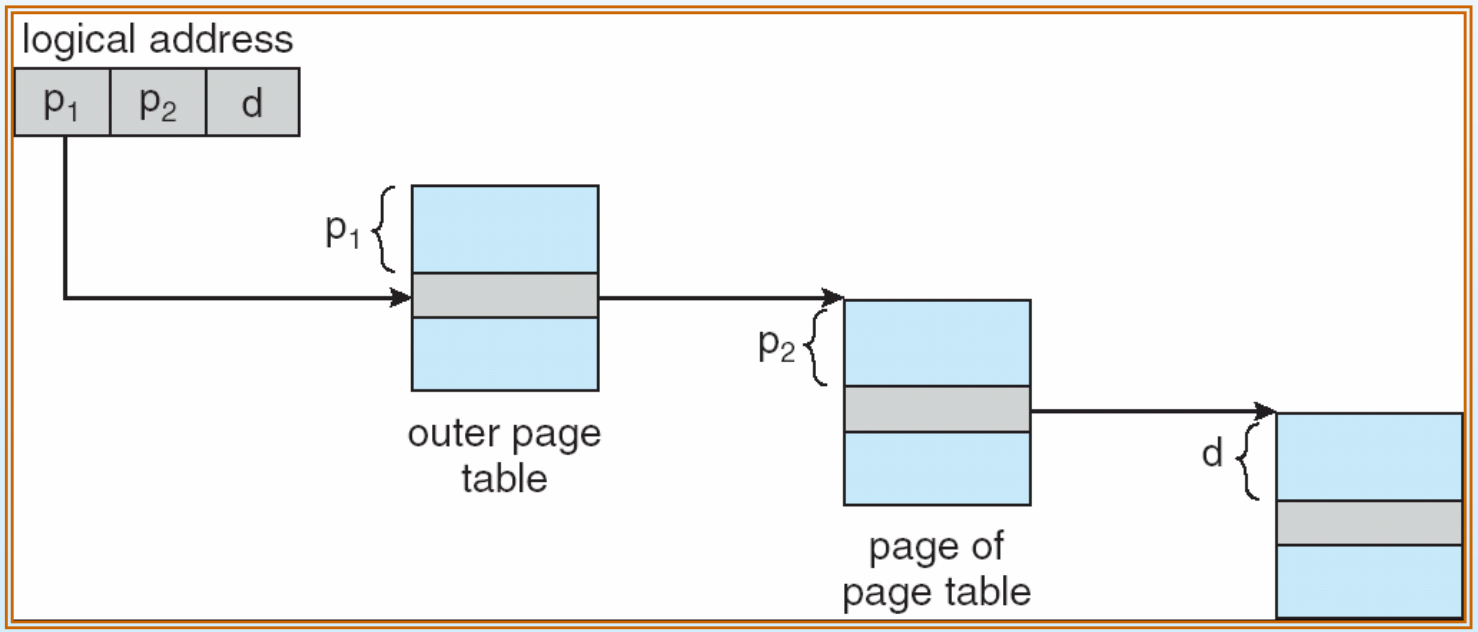


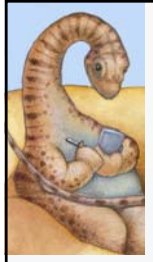
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table



Address-Translation Scheme

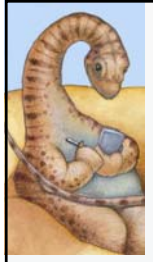
- Address-translation scheme for a two-level 32-bit paging architecture



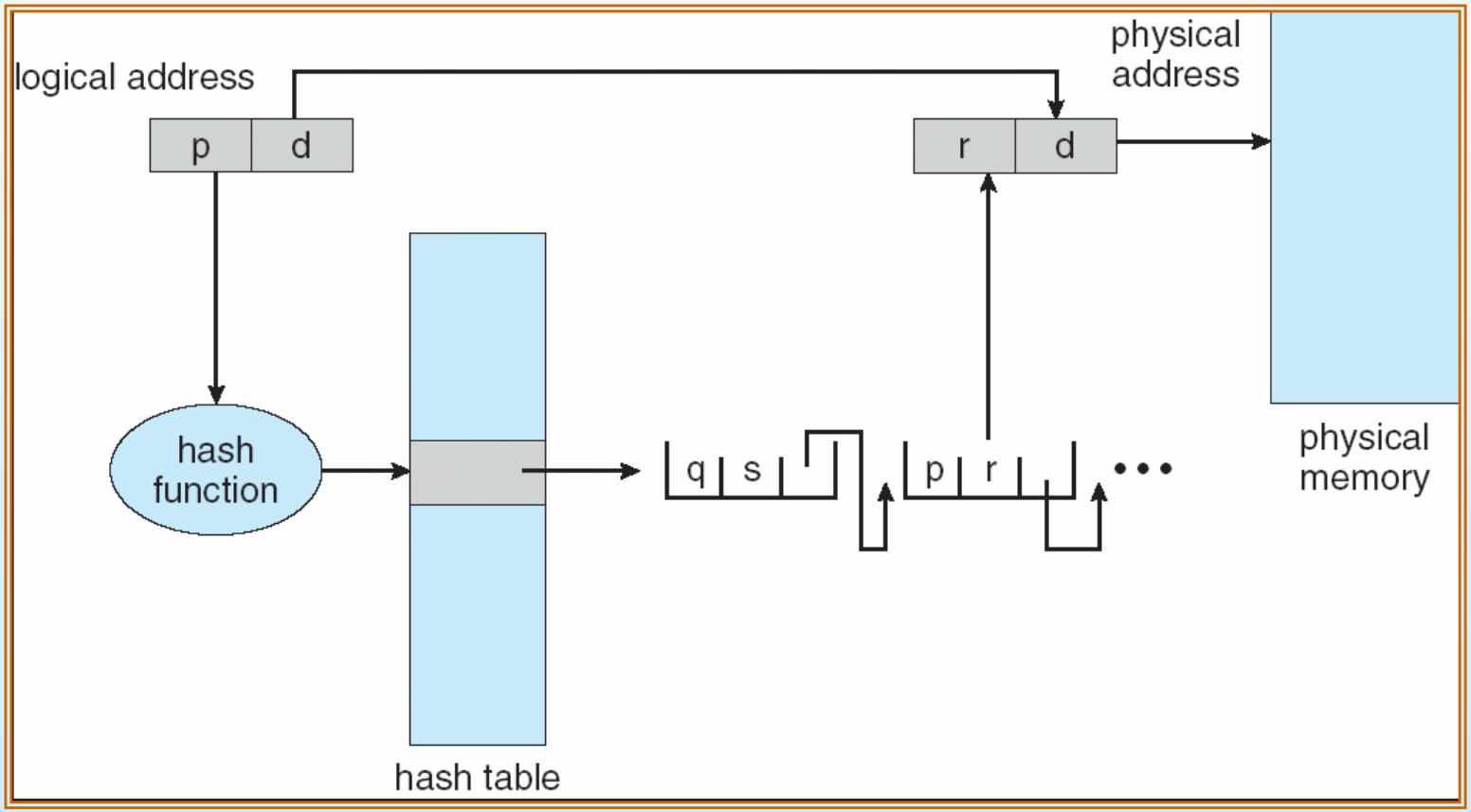


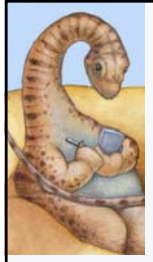
Hashed Page Tables

- Common in 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.



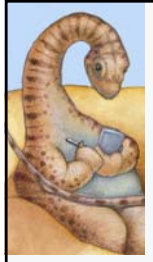
Hashed Page Table



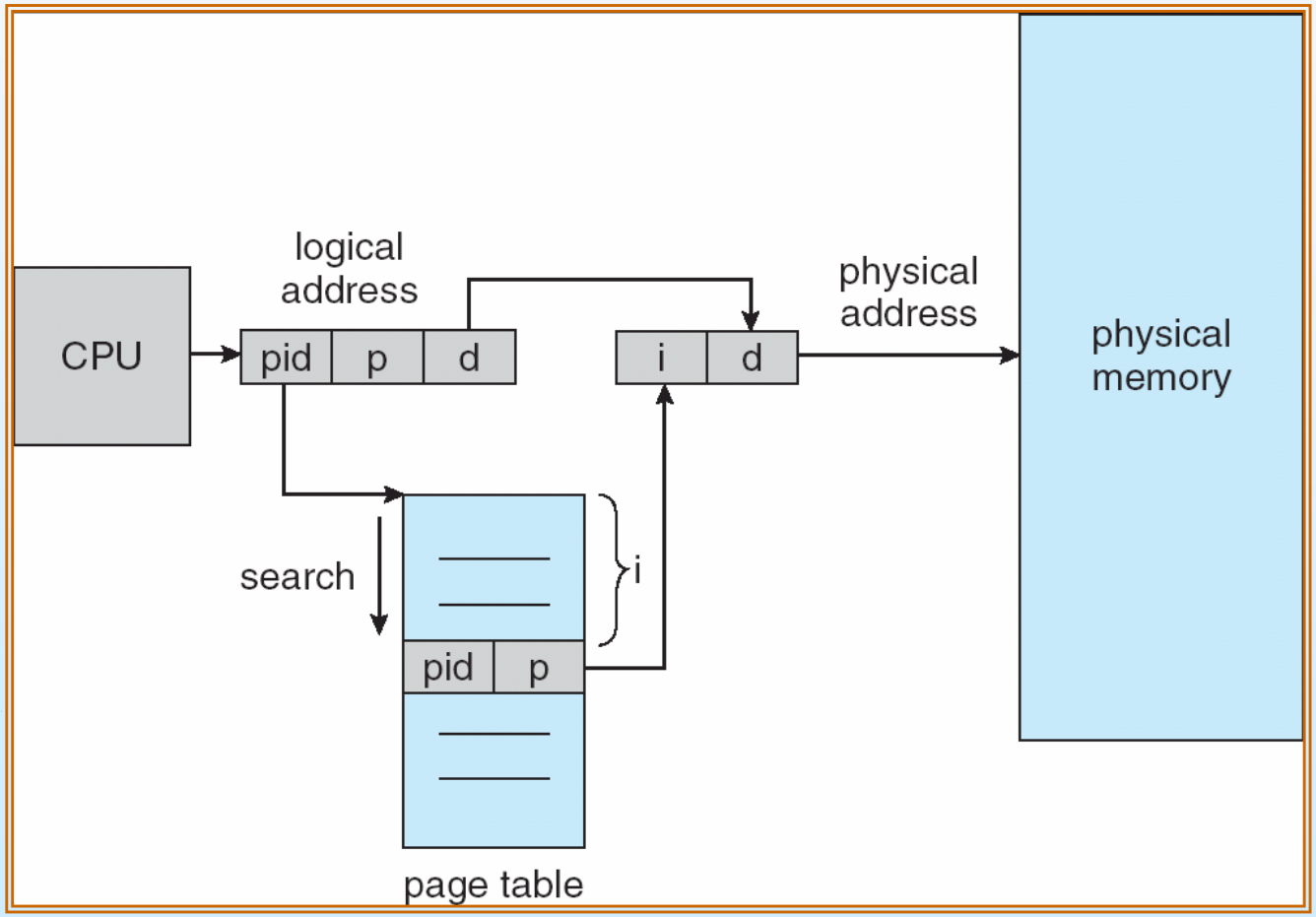


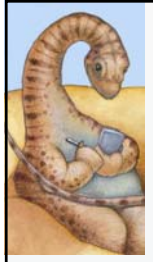
Inverted Page Table

- 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
  - TLB + hash for performance improvement
- Difficult to implement shared memory



Inverted Page Table Architecture





Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  main program,

  procedure,

  function,

  method,

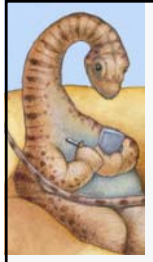
  object,

  local variables, global variables,

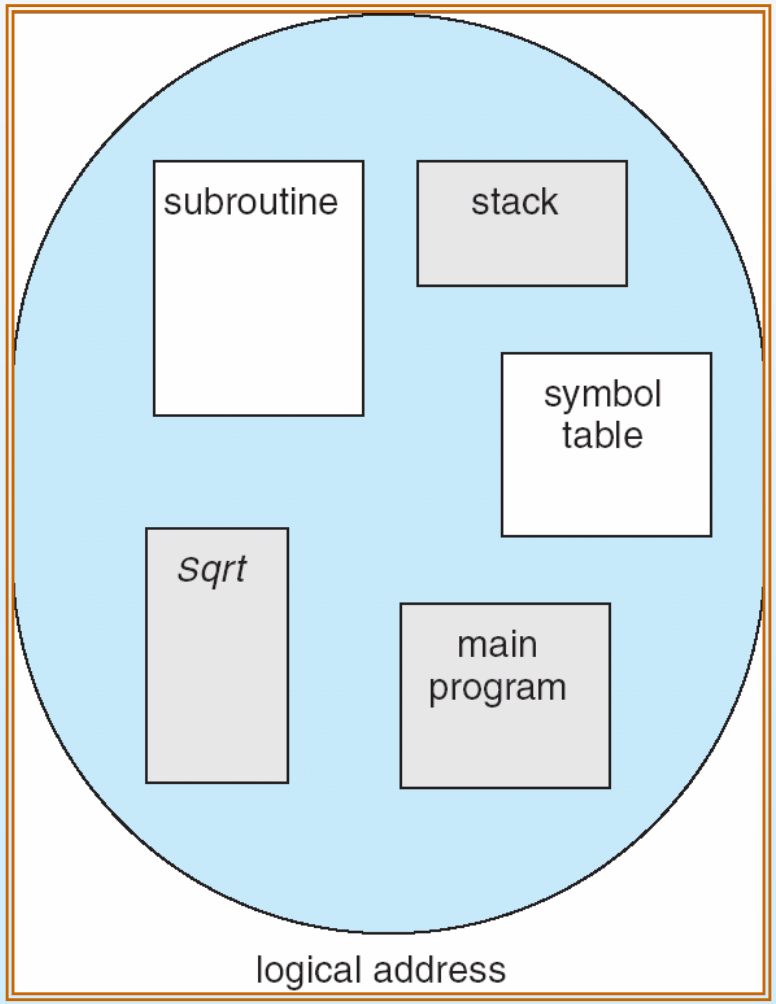
  common block,

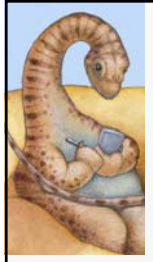
  stack,

  symbol table, arrays

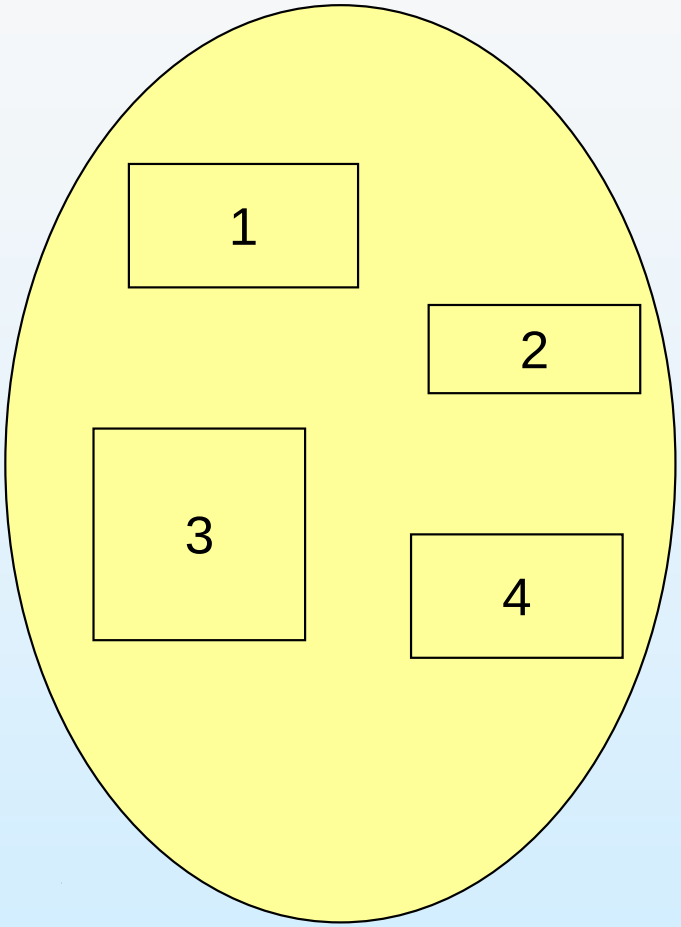


User's View of a Program

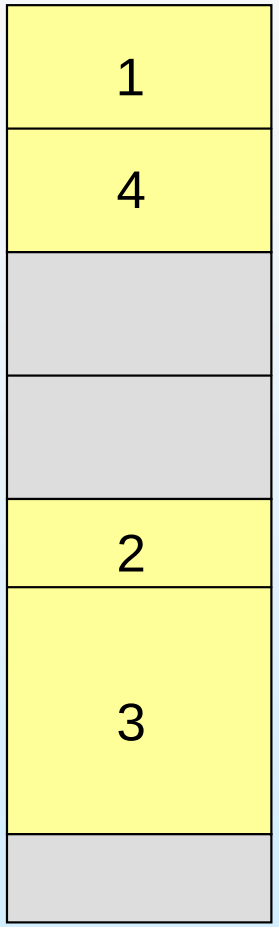




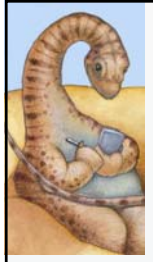
Logical View of Segmentation



user space



physical memory space



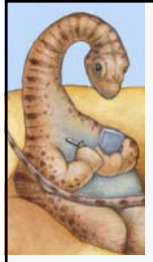
Hardware

- Logical address consists of a two tuple:

  <segment-number, offset>,

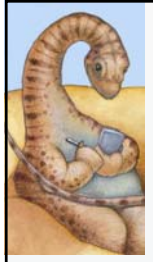
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - *limit* – specifies the length of the segment
- *Segment-table base register (STBR)* points to the segment table's location in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

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



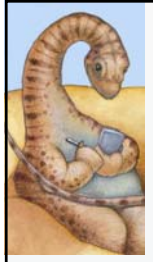
Hardware (Cont.)

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

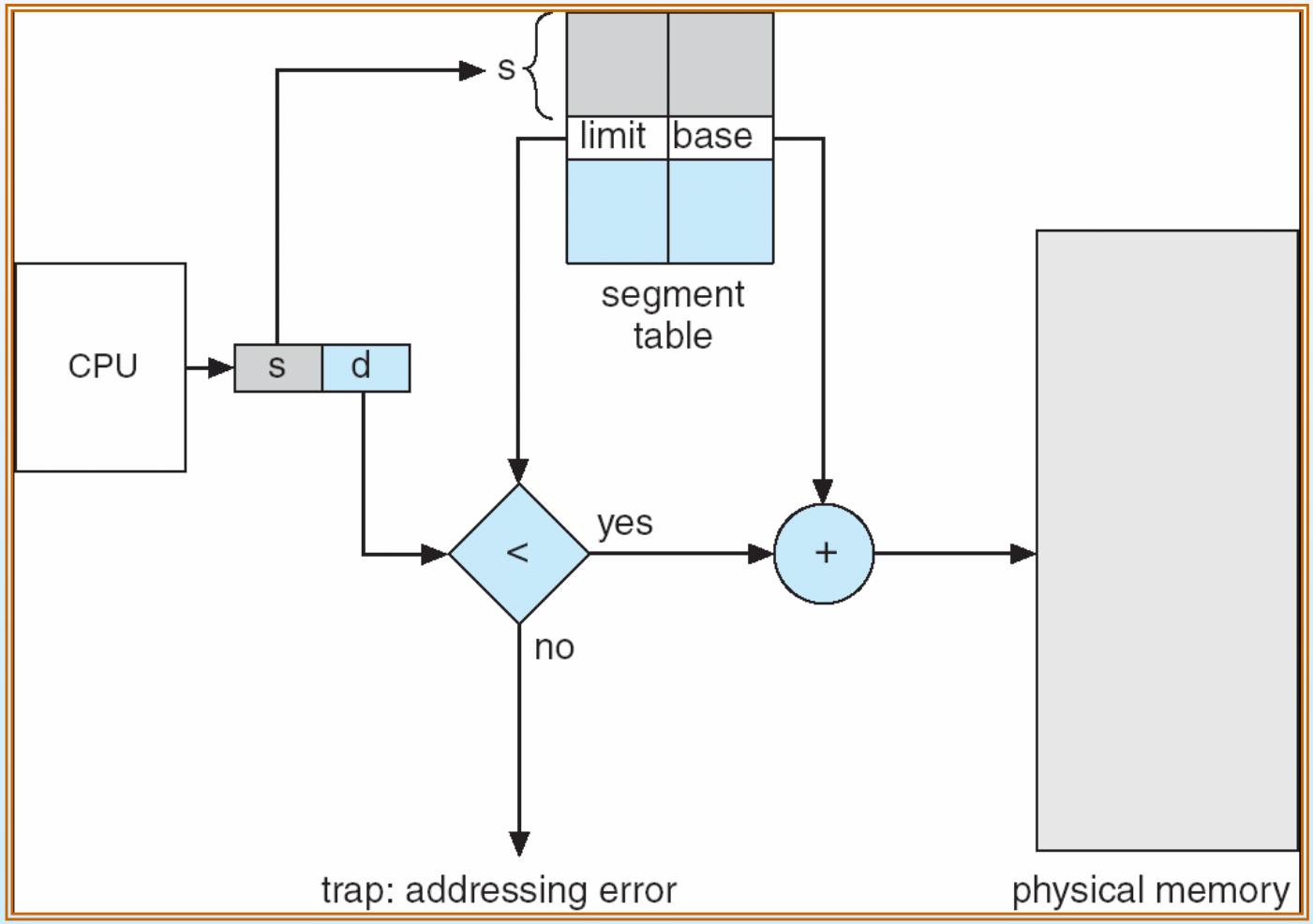


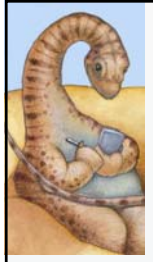
Hardware (Cont.)

- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

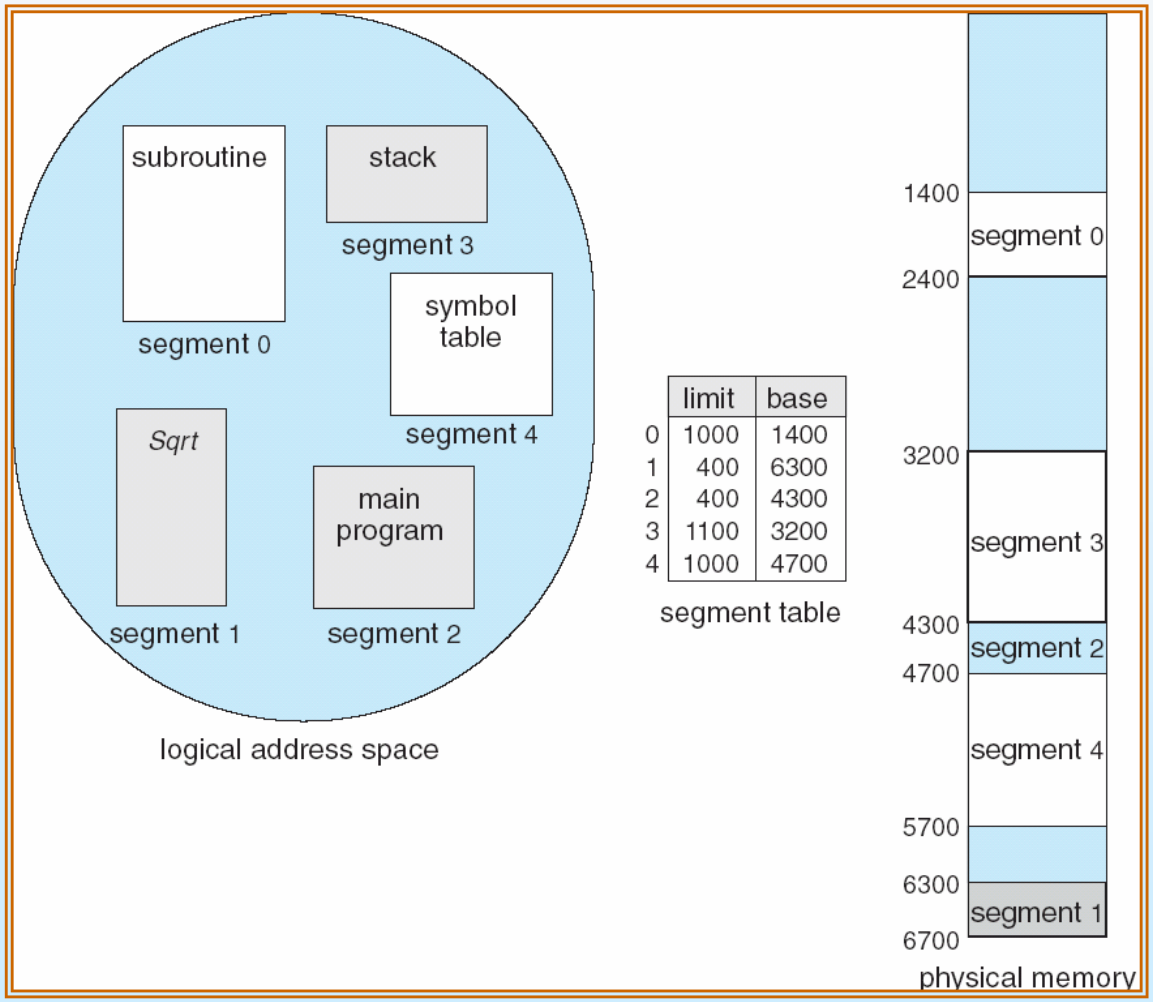


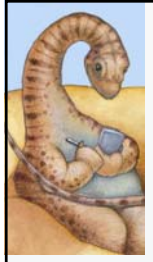
Segmentation hardware



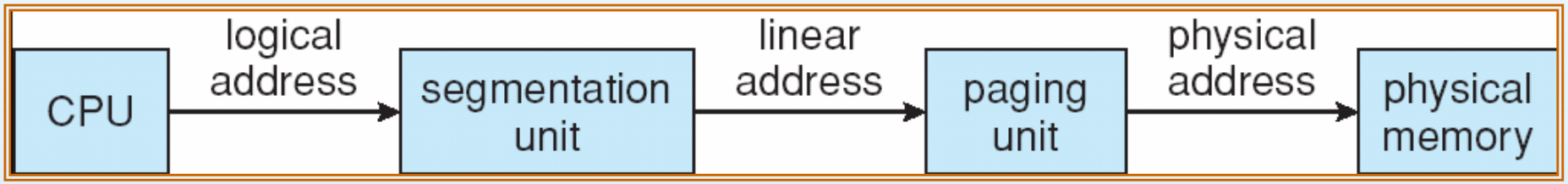


Example of Segmentation





Address Translation in Pentium



End of Chapter 8