

Chapter 8: Memory Management

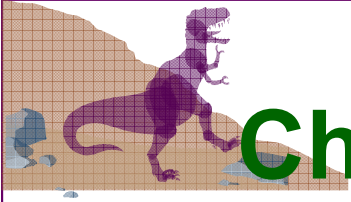
肖 卿 俊

办公室：计算机楼532室

电邮： csqjxiao@seu.edu.cn

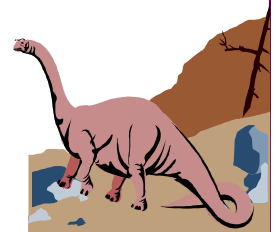
主页： <http://cse.seu.edu.cn/PersonalPage/csqjxiao>

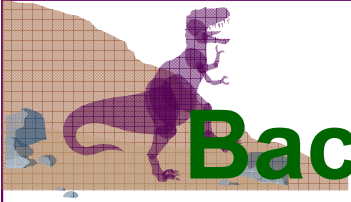
电话： 025-52091023



Chapter 8: Memory Management

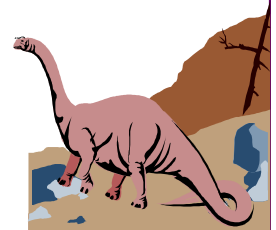
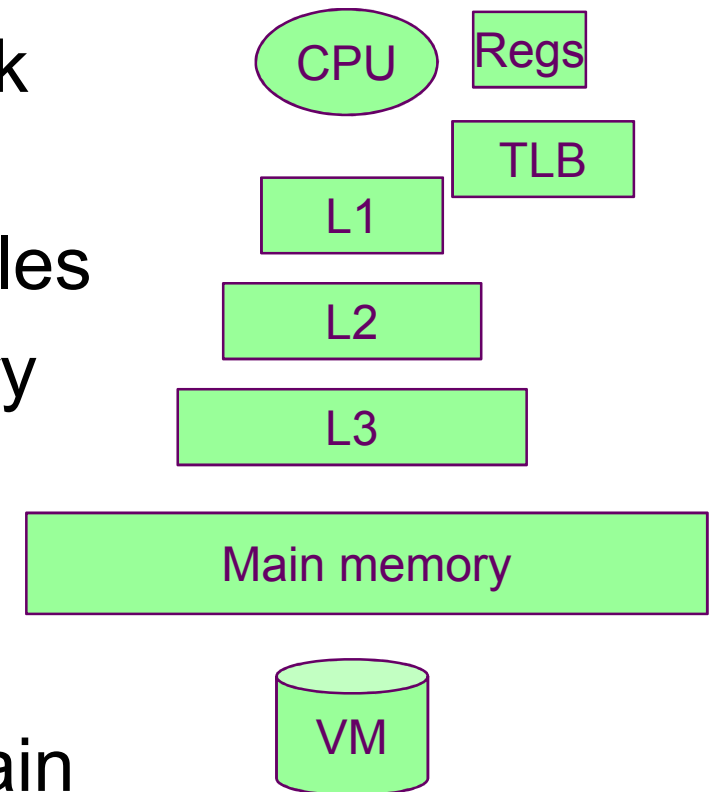
- Background
- Swapping
- Contiguous Allocation
- Paging
- Structure of Page Table
- Segmentation

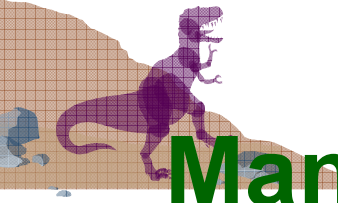




Background for Memory Hierarchy

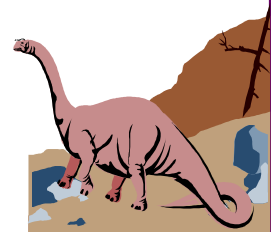
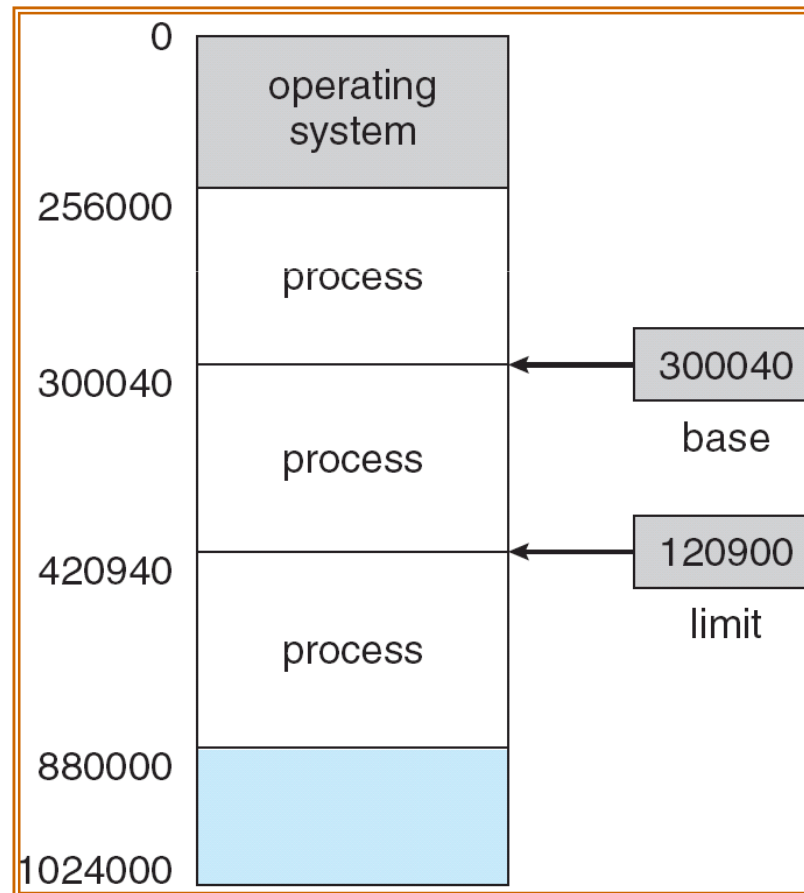
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- Program must be brought into main memory and placed within a process for it to be run.

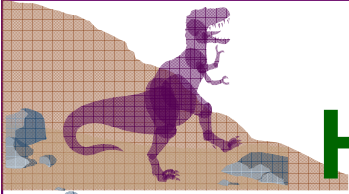




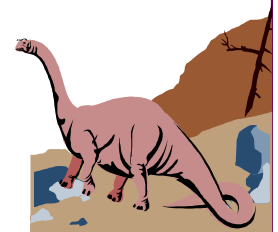
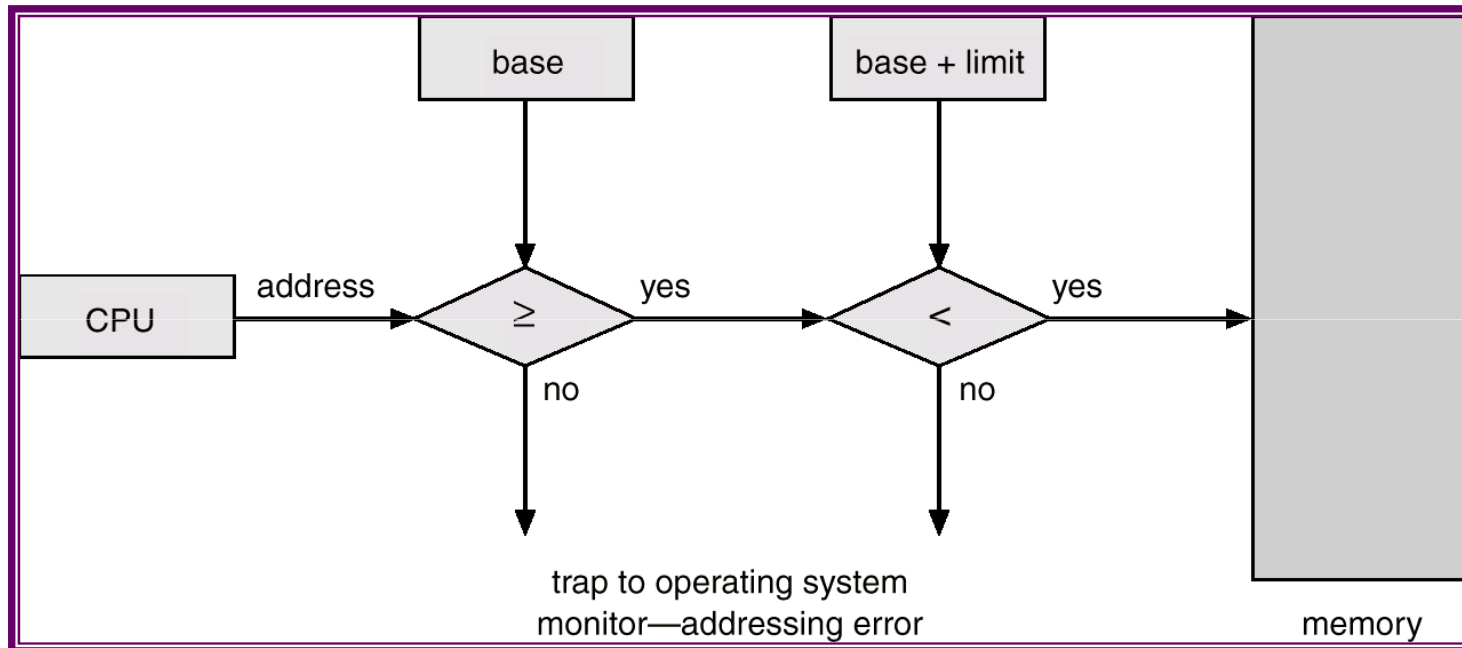
Revisit the Simple Memory Management: Base + Limit Registers

- A pair of **base** and **limit** registers define the logical address space

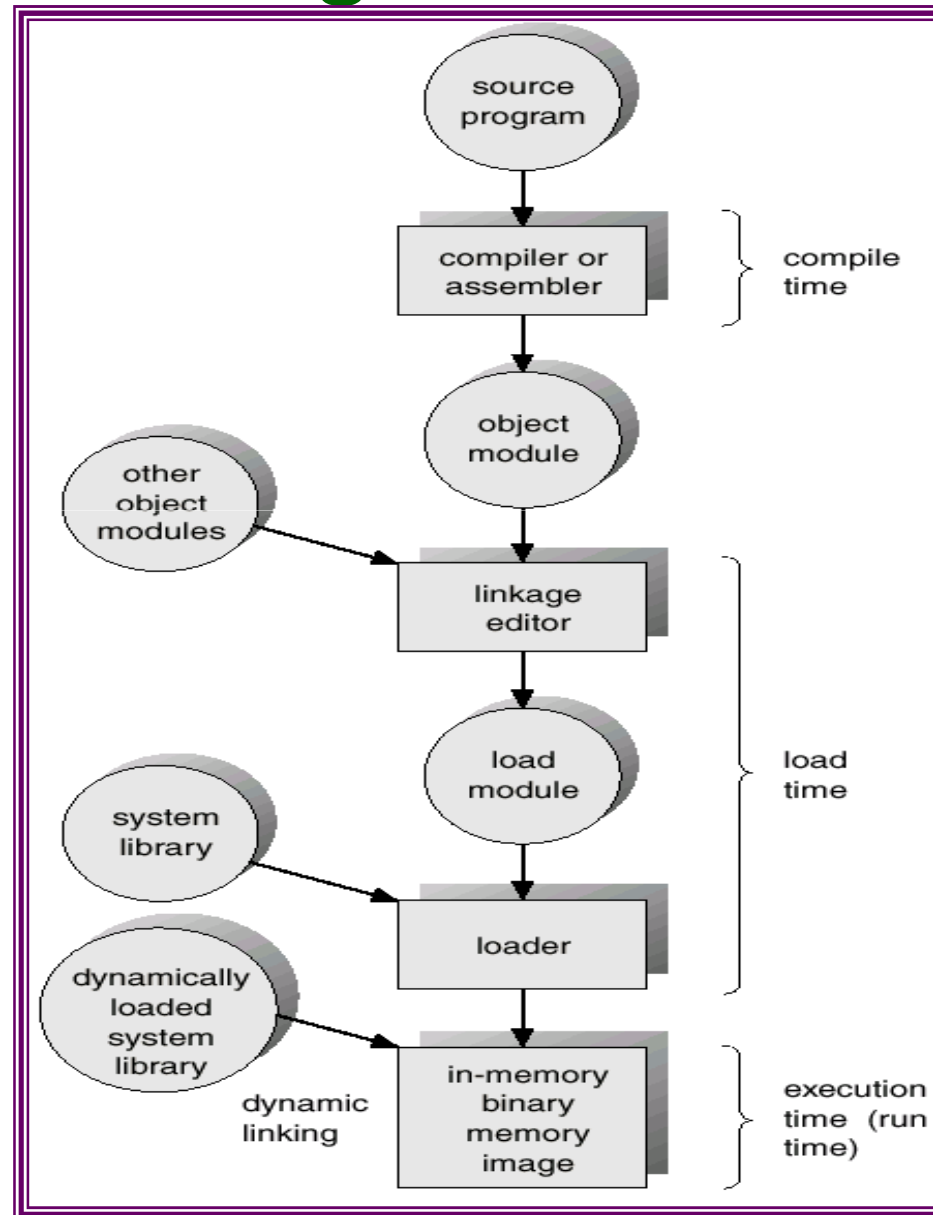


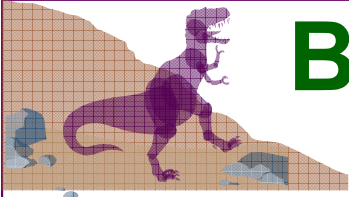


Hardware Address Protection



Background on Multistep Processing of a User Program





Binding of Instructions and Data to Memory

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

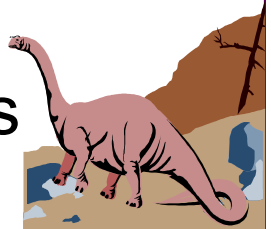
- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time:** Must generate *relocatable* code if memory location is not known at compile time.
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps

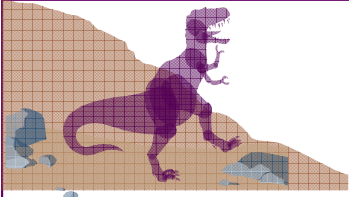




Dynamic Linking

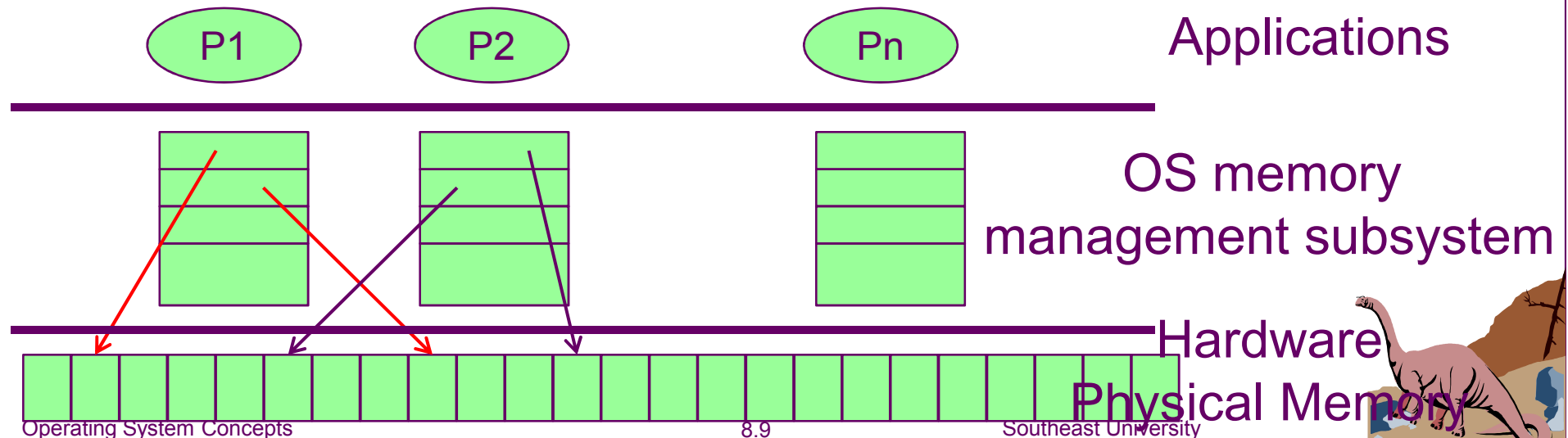
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** – linking postponed until run time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
 - ◆ If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries

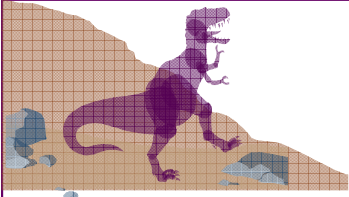




Logical vs. Physical Address Space

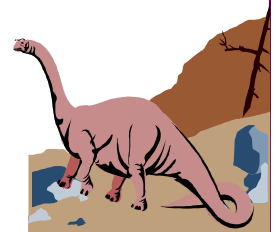
- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
 - ◆ *Logical address* – generated by the CPU; also referred to as *virtual address*.
 - ◆ *Physical address* – address seen by memory unit.

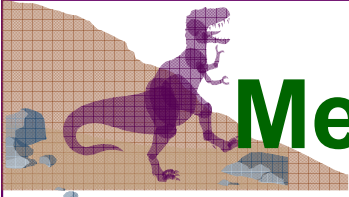




Logical vs. Physical Address Space (Cont.)

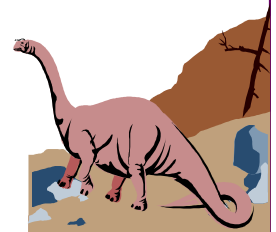
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme.
 - ◆ In this case, logical address is also referred to as virtual address. (Logical = Virtual in this course)



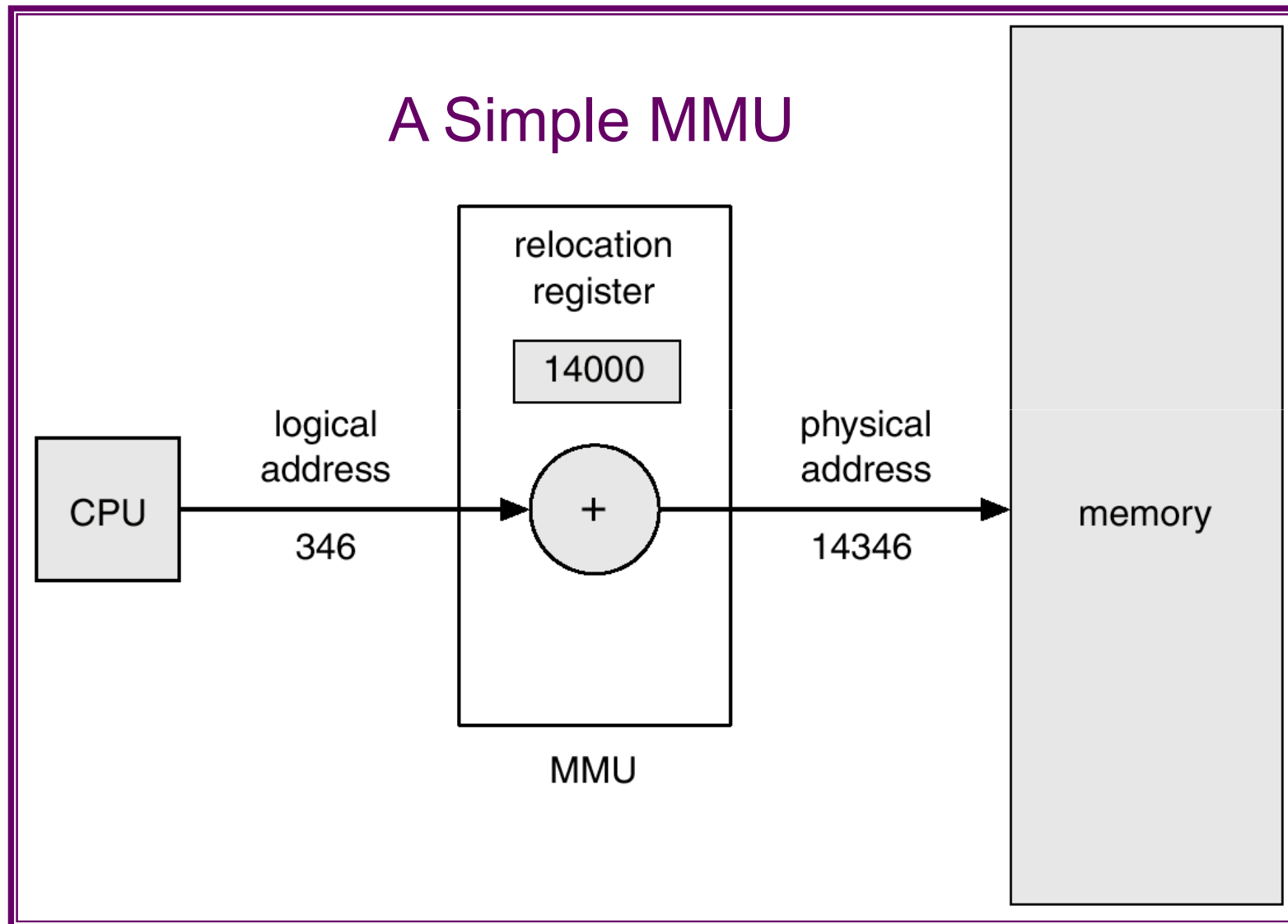


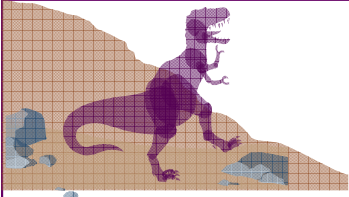
Memory-Management Unit (MMU)

- 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.



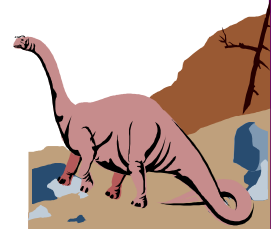
Dynamic Relocation using a Relocation Register





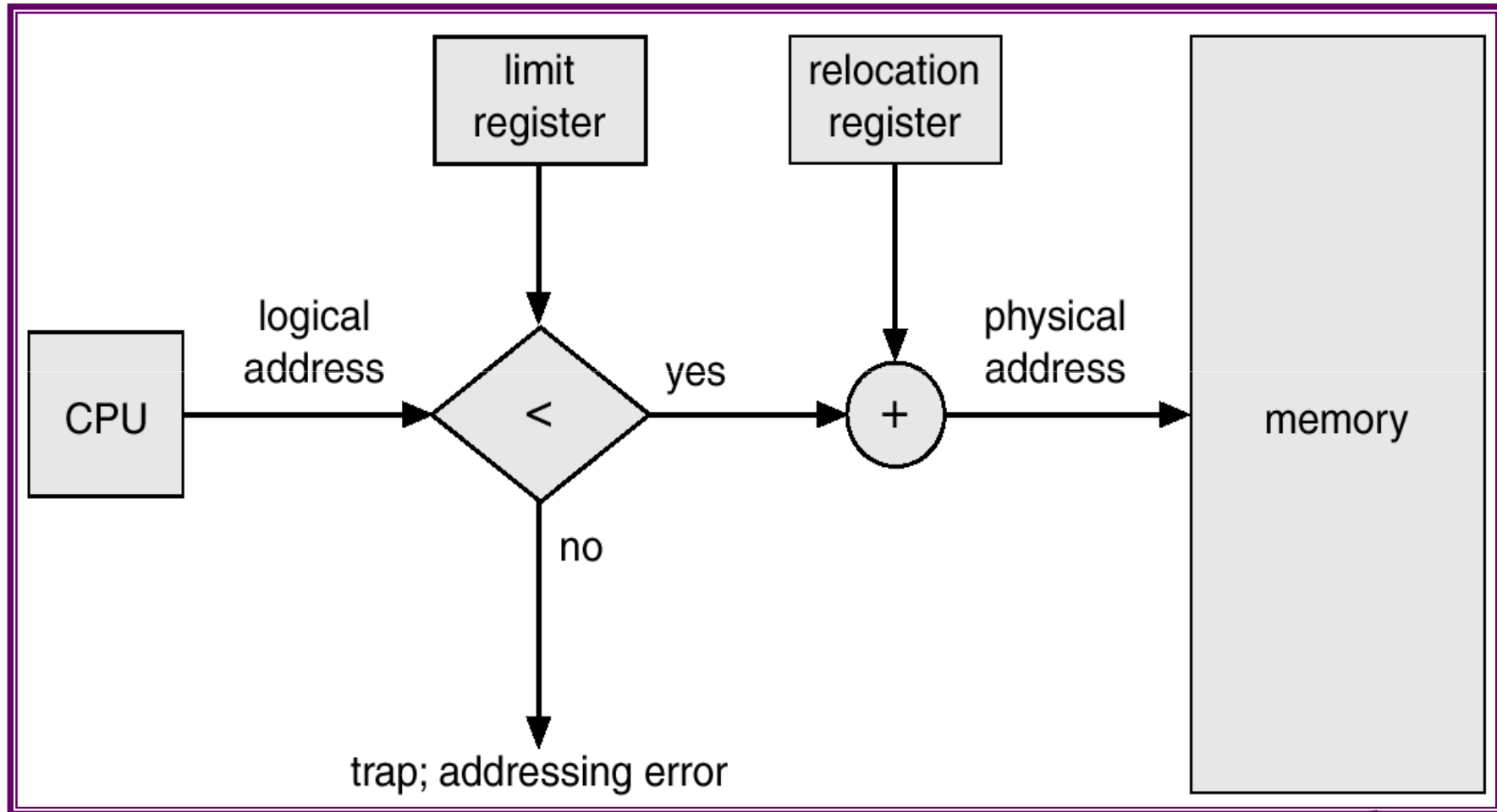
Memory Protection

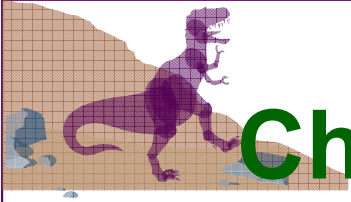
- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
- Relocation register contains value of the smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register.





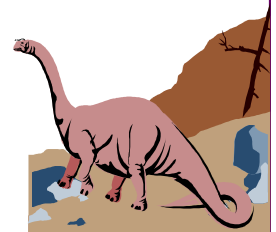
Hardware Support for Relocation and Limit Registers

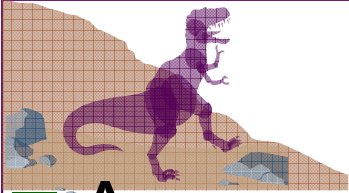




Chapter 9: Memory Management

- Background
- **Swapping**
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging





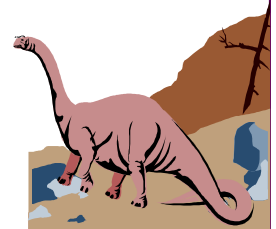
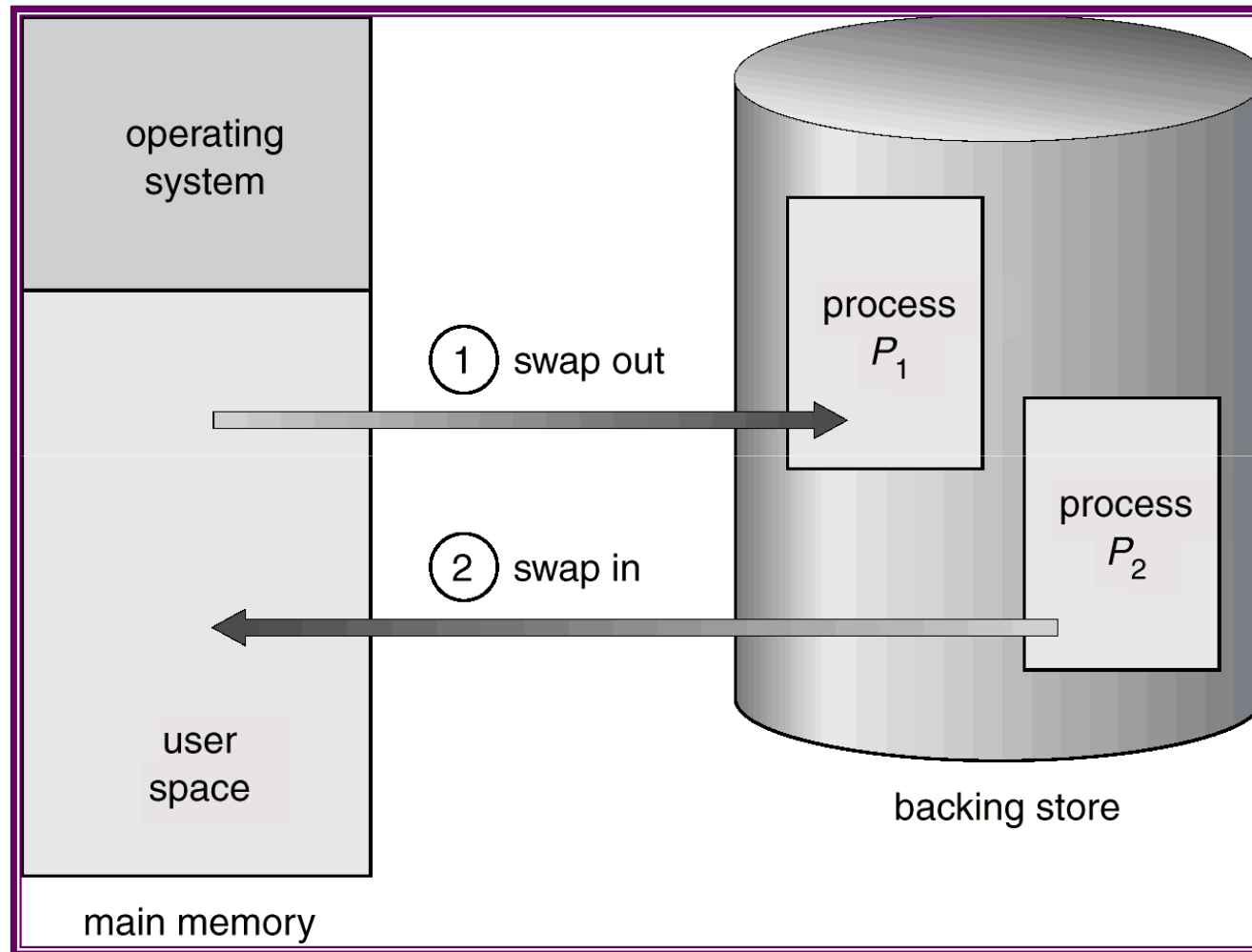
Swapping

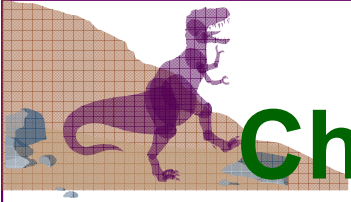
- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to hold copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – 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.





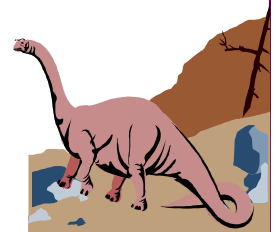
Schematic View of Swapping

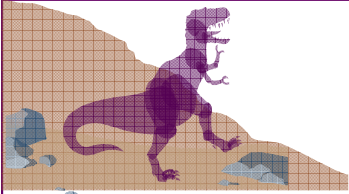




Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

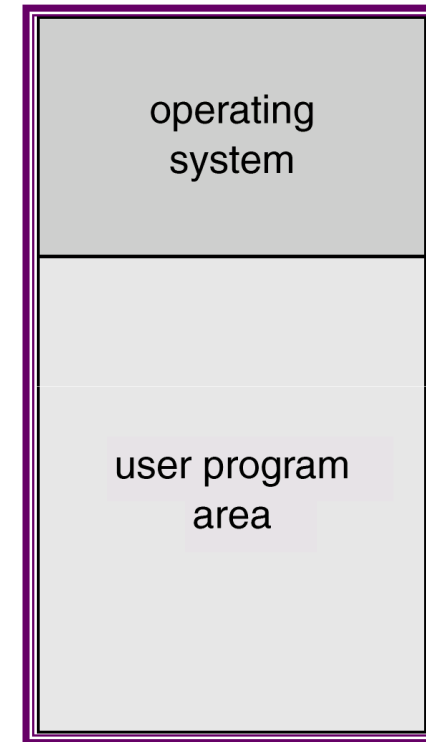




Contiguous Allocation

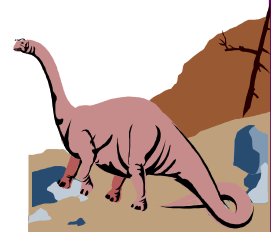
■ Monoprogramming systems usually have two partitions:

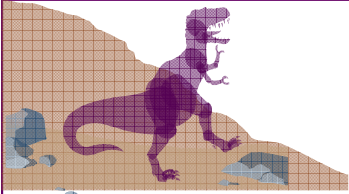
- ◆ Resident operating system, usually held in low memory with interrupt vector.
- ◆ User processes then held in high memory.



■ Multiprogramming Systems:

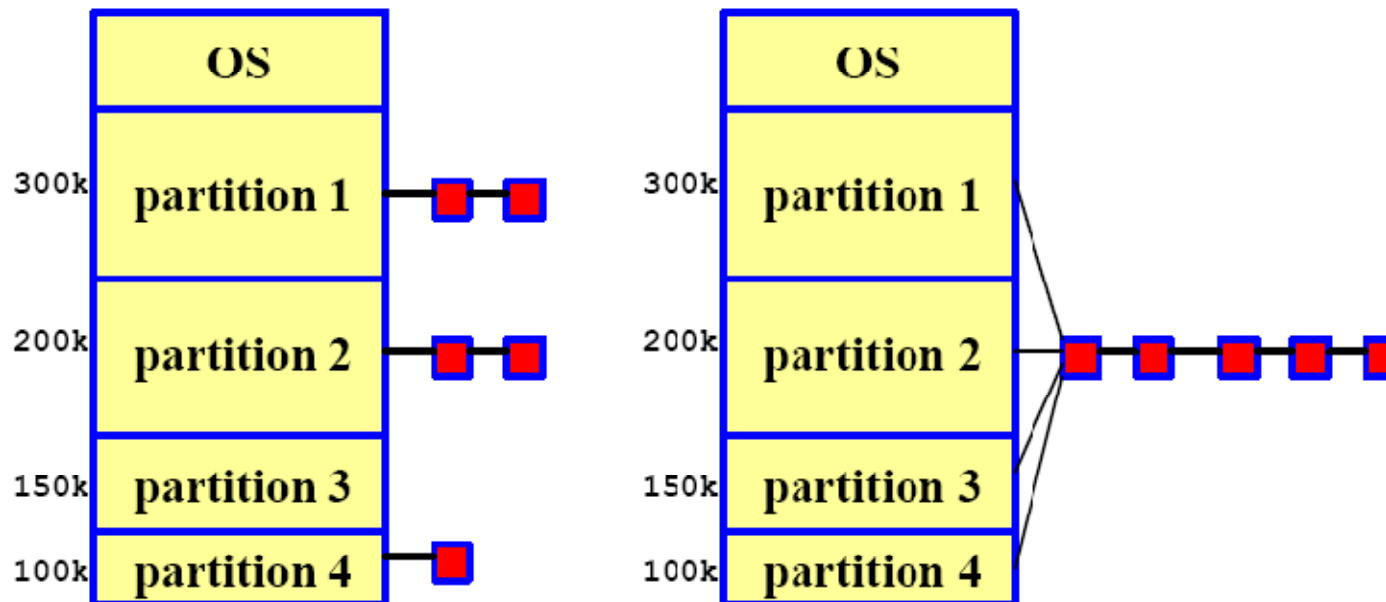
- ◆ Fixed partitions
- ◆ Variable partitions

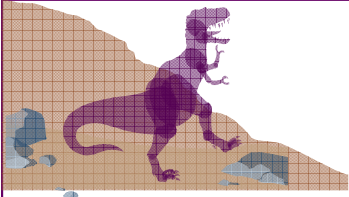




Fixed Partitions

- Main memory is divided into n partitions.
- Partitioning can be done at the startup time and altered later on.
- Each partition may have a job queue. Or, all partitions share the same job queue.

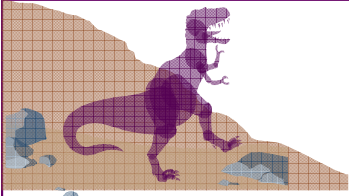




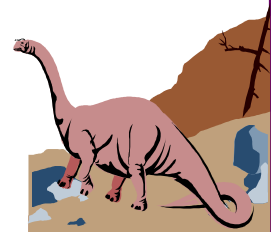
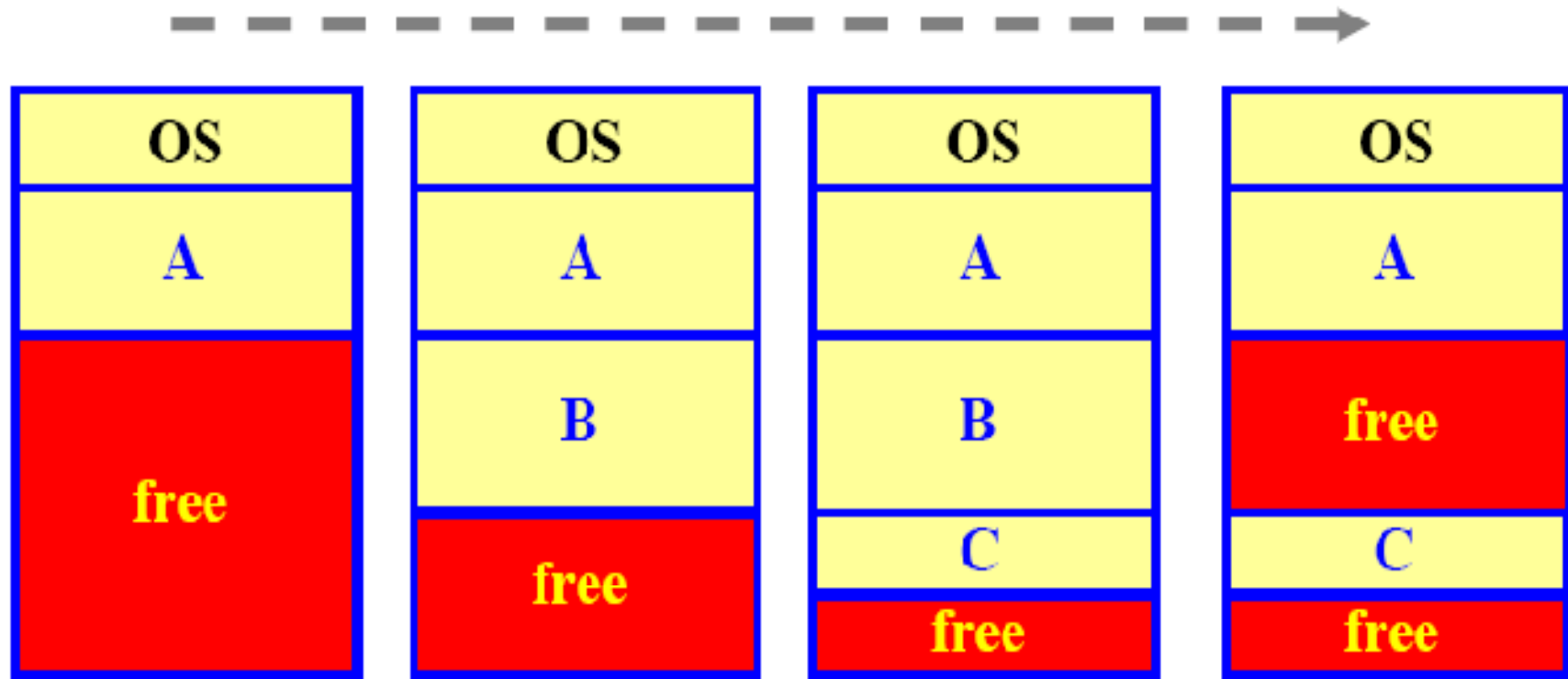
Variable Partitions

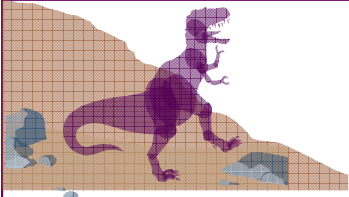
- **Hole** – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Thus, partition sizes are not fixed, The number of partitions also varies.
- Operating system maintains information about:
 - a) allocated partitions b) free partitions (hole)





Variable Partitions(Cont.)

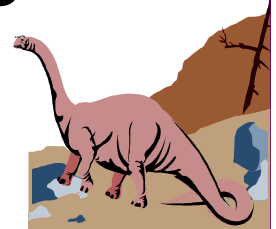


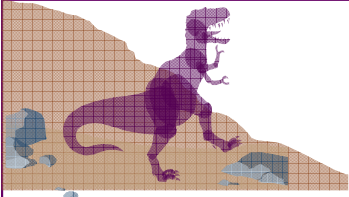


Dynamic Storage-Allocation Problem

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

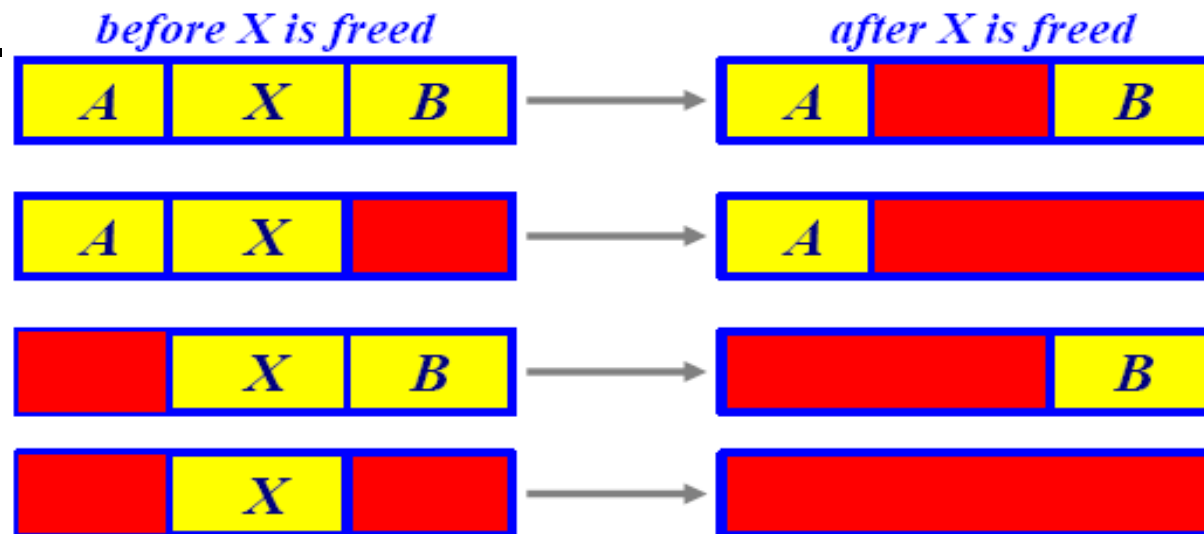
- **First-fit (首次适配):** Allocate the *first* hole that is big enough.
- **Best-fit (最佳适配):** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit (最差适配):** Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

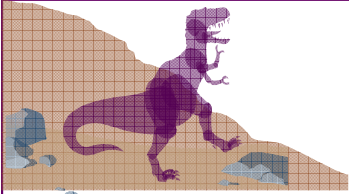




Dynamic Storage-Allocation Problem

- If the hole is larger than the requested size, it is cut into two. The one of the requested size is given to the process, the remaining one becomes a *new* hole.
- When a process returns a memory block, it becomes a hole and must be combined with its neighbors.

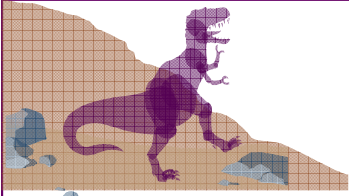




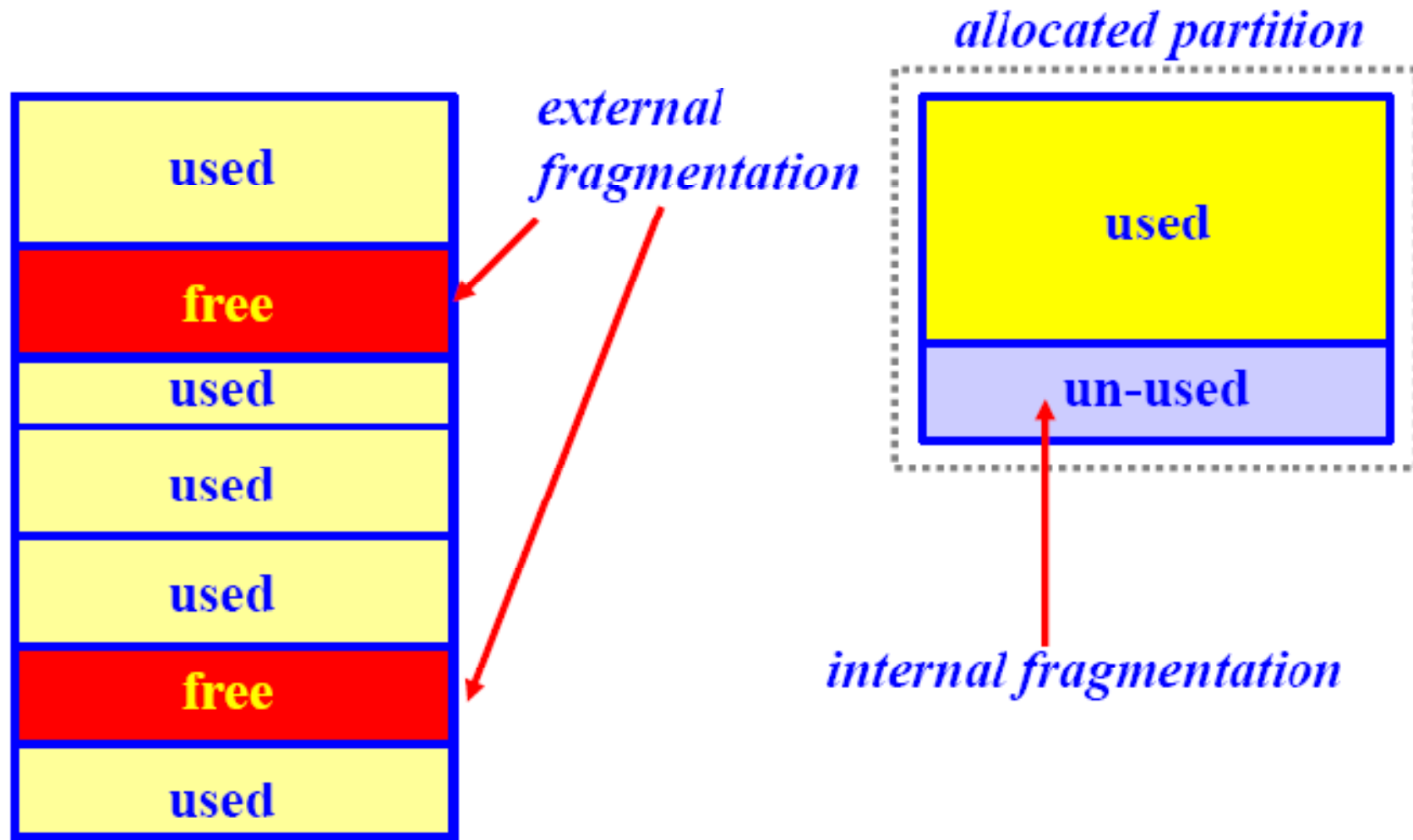
Fragmentation (碎片)

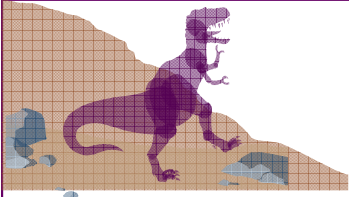
- Processes are loaded and removed from memory, eventually the memory will be cut into small holes that are not large enough to run any incoming process.
- Free memory holes between allocated ones are called *external fragmentation*.
- It is unwise to allocate exactly the requested amount of memory to a process, because of the minimum requirement for memory management.
- Thus, memory that is allocated to a partition, but is not used, are called *internal fragmentation*.





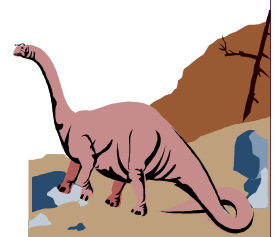
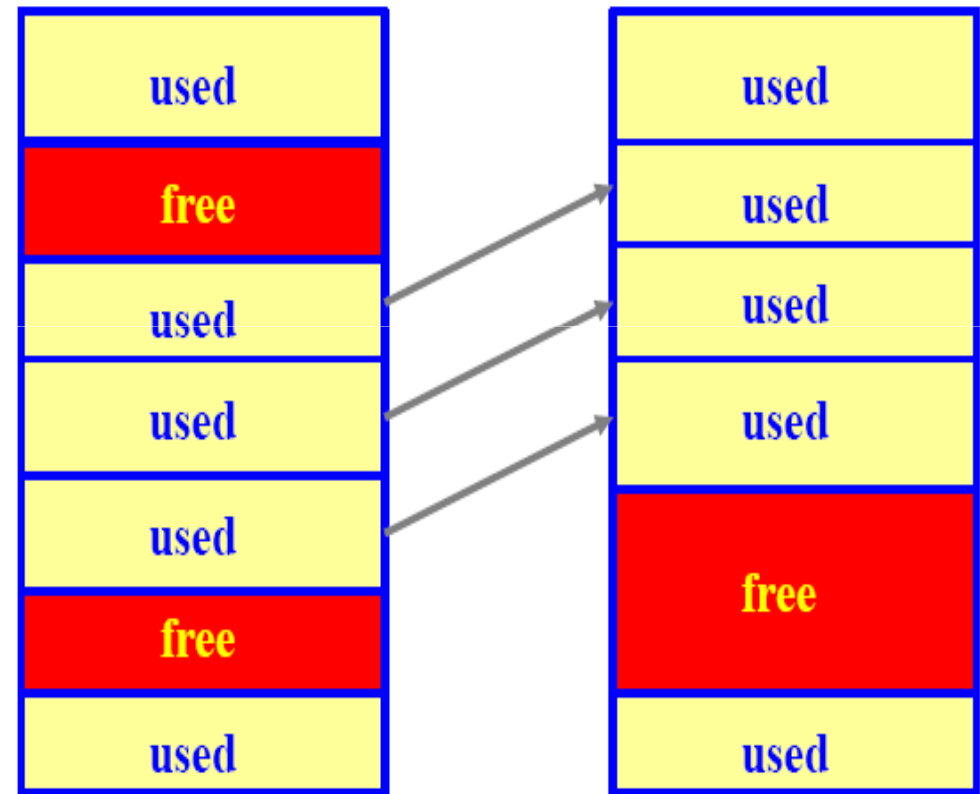
Fragment (Cont.)

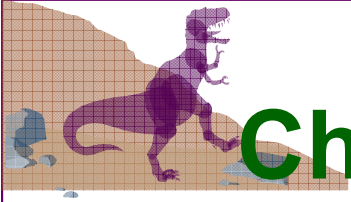




Compaction for External Fragmentation

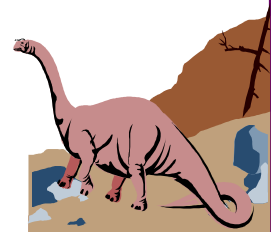
- Shuffle memory contents to place all free memory together in one large block.
- Compaction is possible *only* if program relocation is dynamic, and is done at execution time.
- Compaction scheme can be expensive

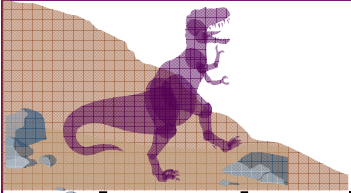




Chapter 9: Memory Management

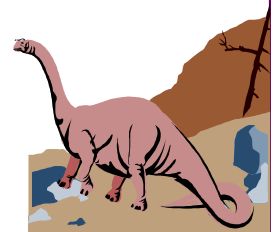
- Background
- Swapping
- Contiguous Allocation
- **Paging**
- Segmentation
- Segmentation with Paging

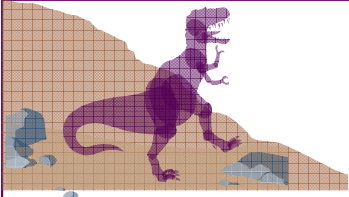




Paging

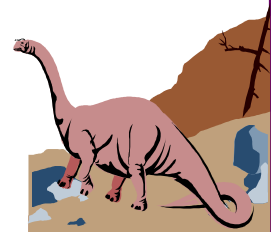
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (帧) (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same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

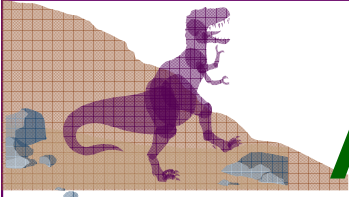




Paging (Cont.)

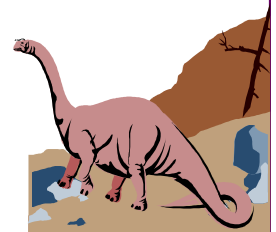
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

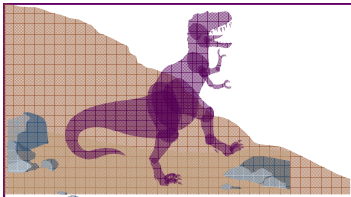




Address Translation Scheme

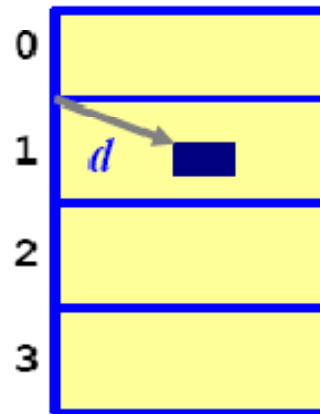
- Address generated by CPU is divided into:
 - ◆ *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
 - ◆ *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.





p d
page # offset within the page

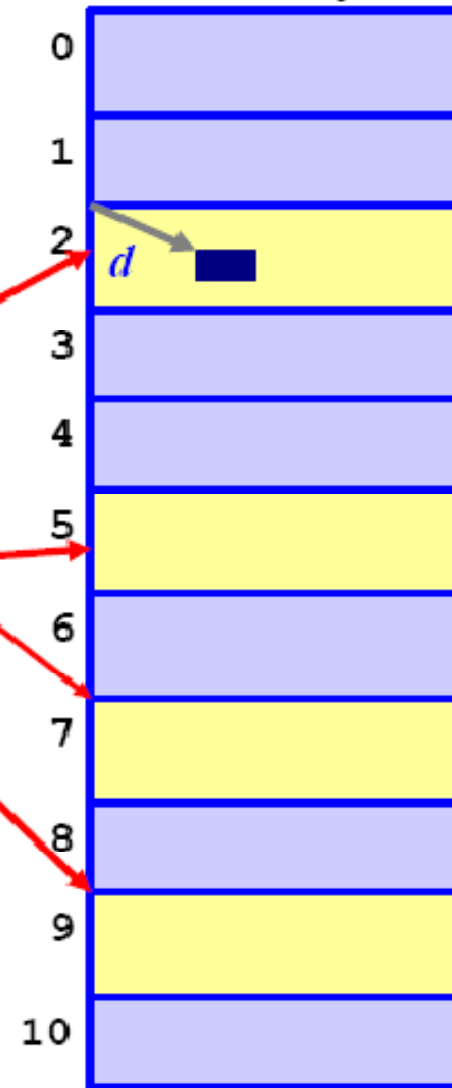
*logical
memory*



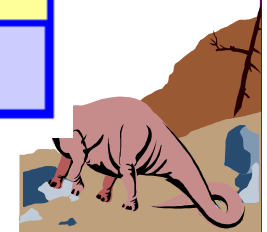
page table

0	7
1	2
2	9
3	5

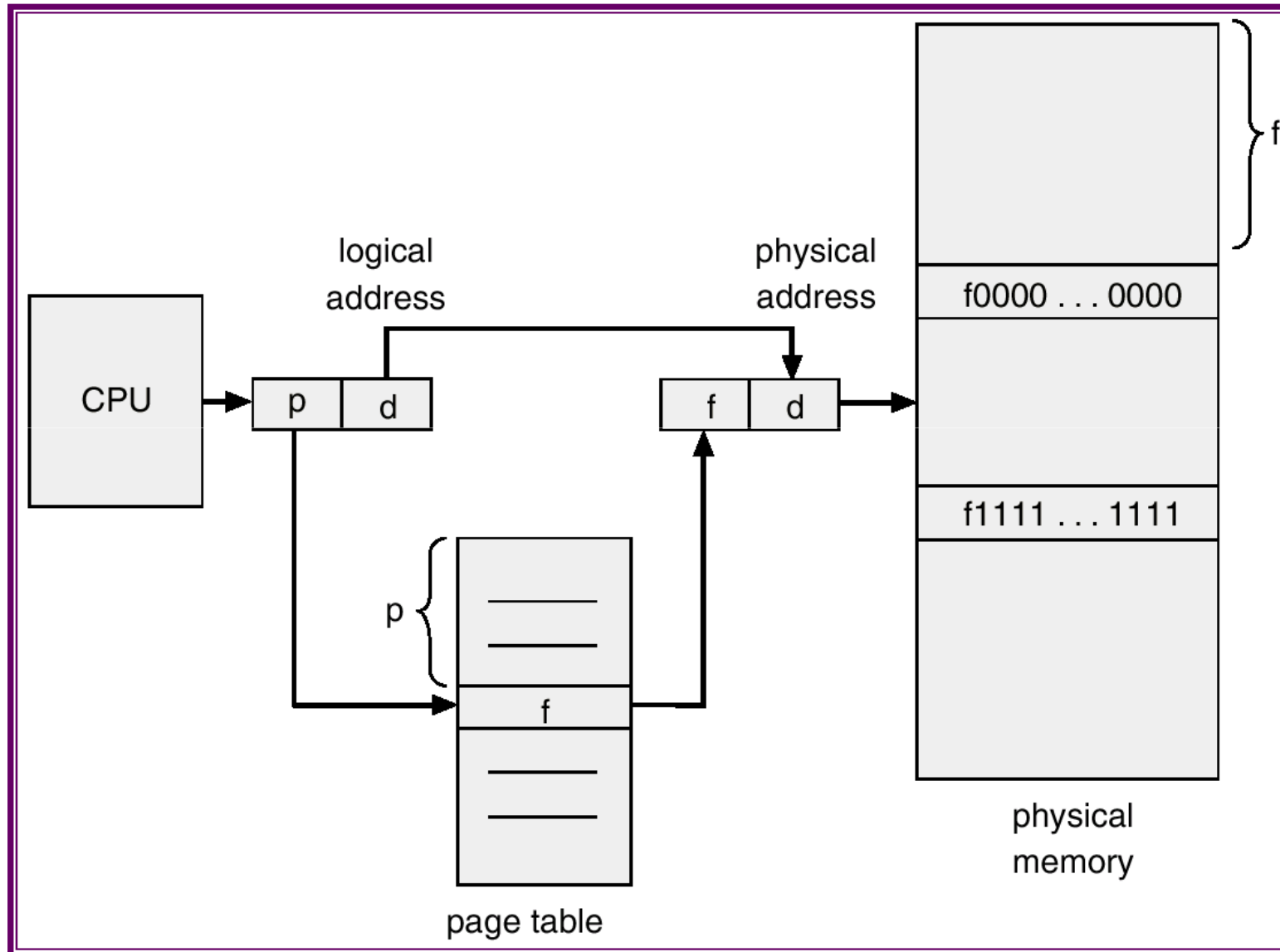
memory

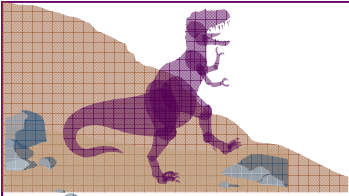


*logical address $\langle 1, d \rangle$ translates to
physical address $\langle 2, d \rangle$*

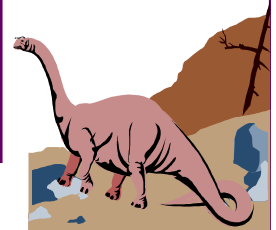
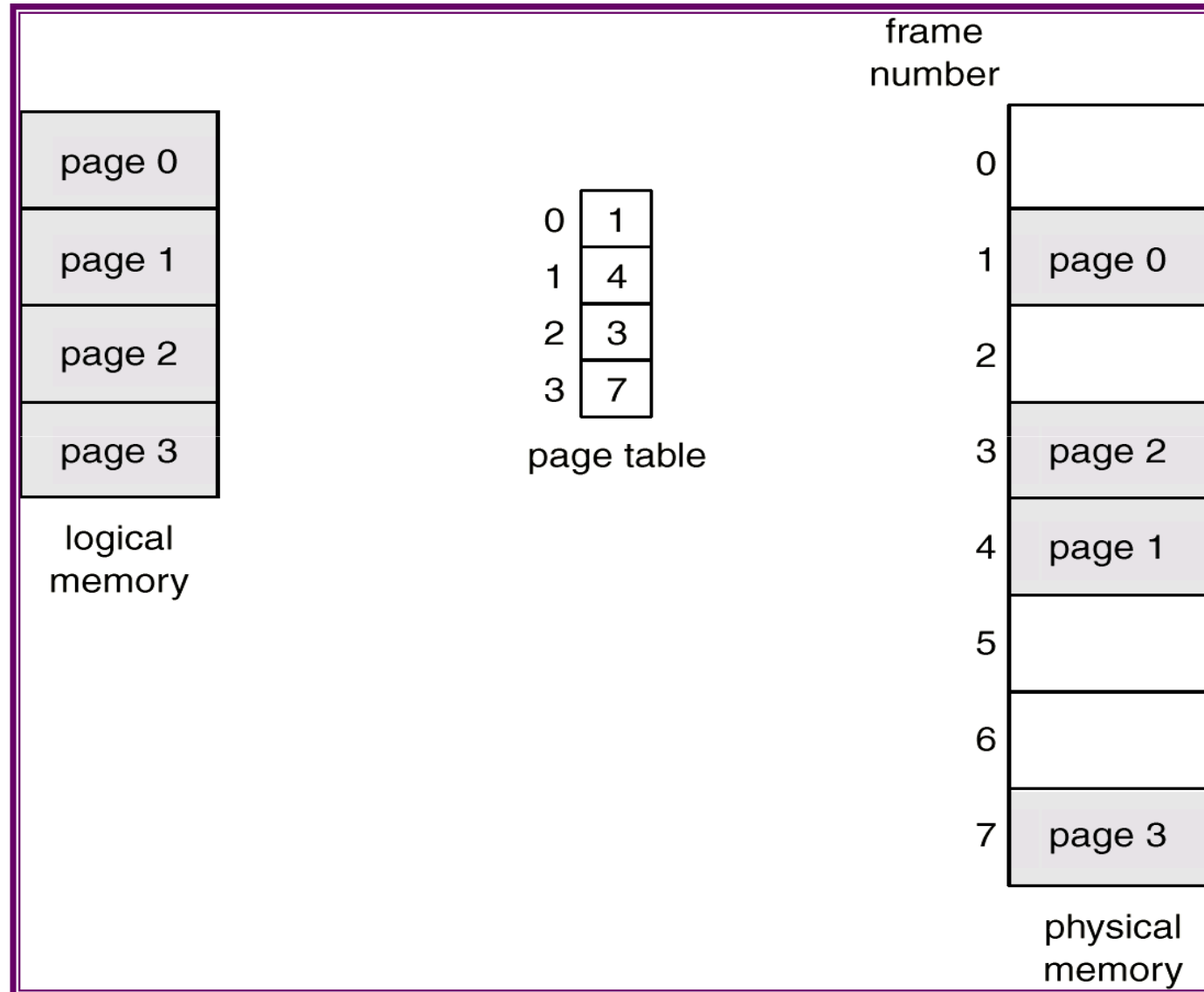


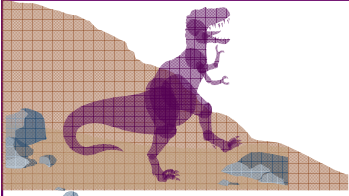
Address Translation Architecture





Paging Example





Paging Example

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

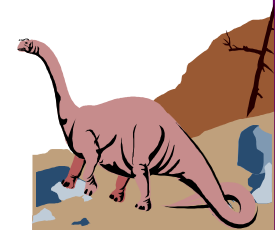
logical memory

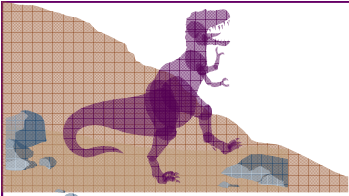
0	5
1	6
2	1
3	2

page table

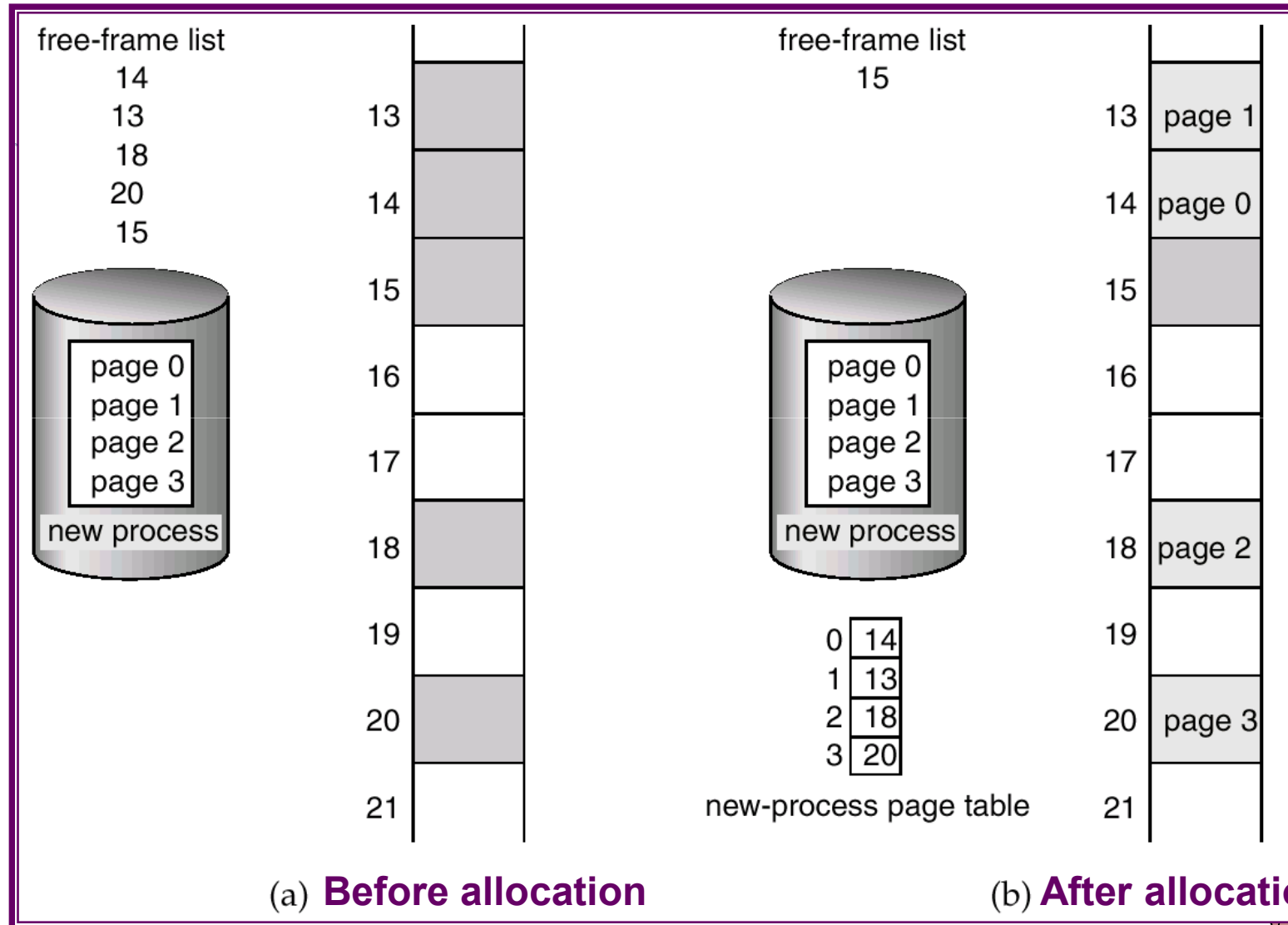
0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

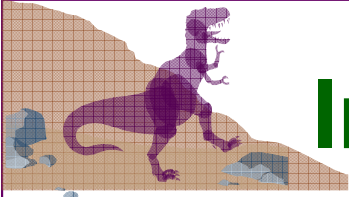
physical memory





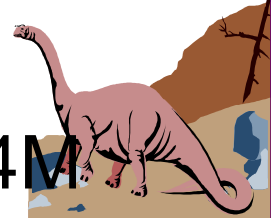
Free Frames

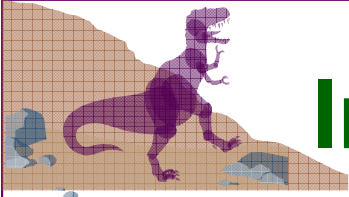




Implementation of Page Table

- **Page table must be kept in main memory.**
- **Question:** Why is a page table hard to entirely fit into L2 cache? What will be the size of a page table, if assuming 32 bits virtual address, 4GB physical memory and 4KB page/frame size?
 - ◆ **20 bits required for frame number.**
 - ✓ 4 GB of Physical Memory = 2^{32} bytes.
 - ✓ 2^{32} bytes of memory / 2^{12} bytes per frame = 2^{20} frames
 - ◆ So each page table entry is approximately **4 bytes**.
(20 bits frame number is roughly 3 bytes and access control contributes 1 byte)
 - ◆ Page table size = 2^{20} entries * 4 bytes/entry = 4M

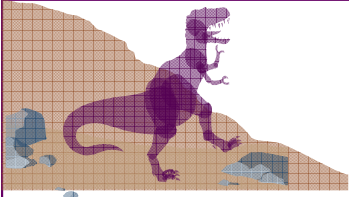




Implementation of Page Table

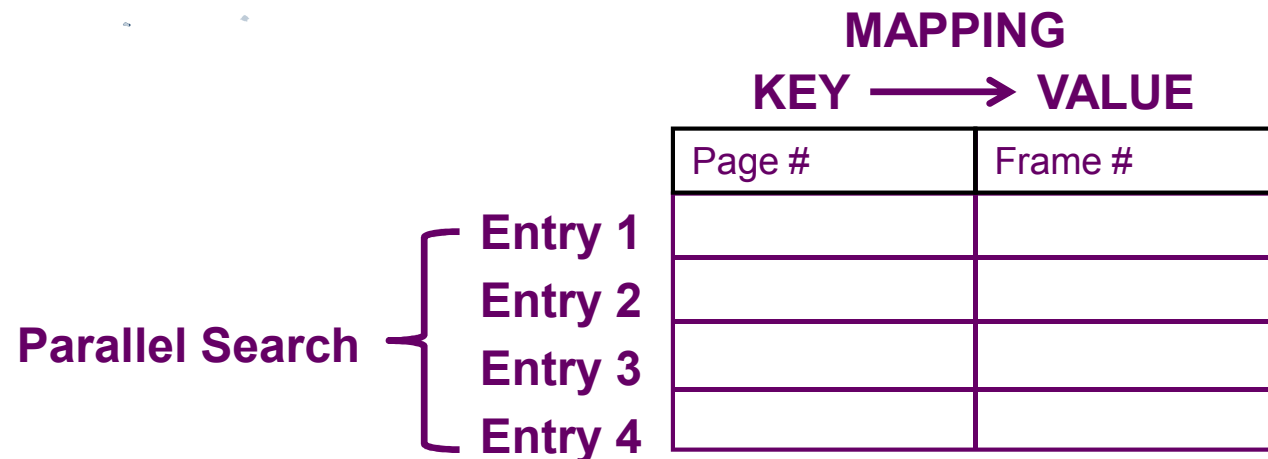
- *Page-table base register (PTBR)* points to the page table existing in main memory.
- In this scheme every data/instruction access requires two memory accesses: One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*





Associative Memory

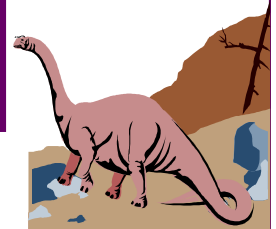
■ Associative memory – parallel search



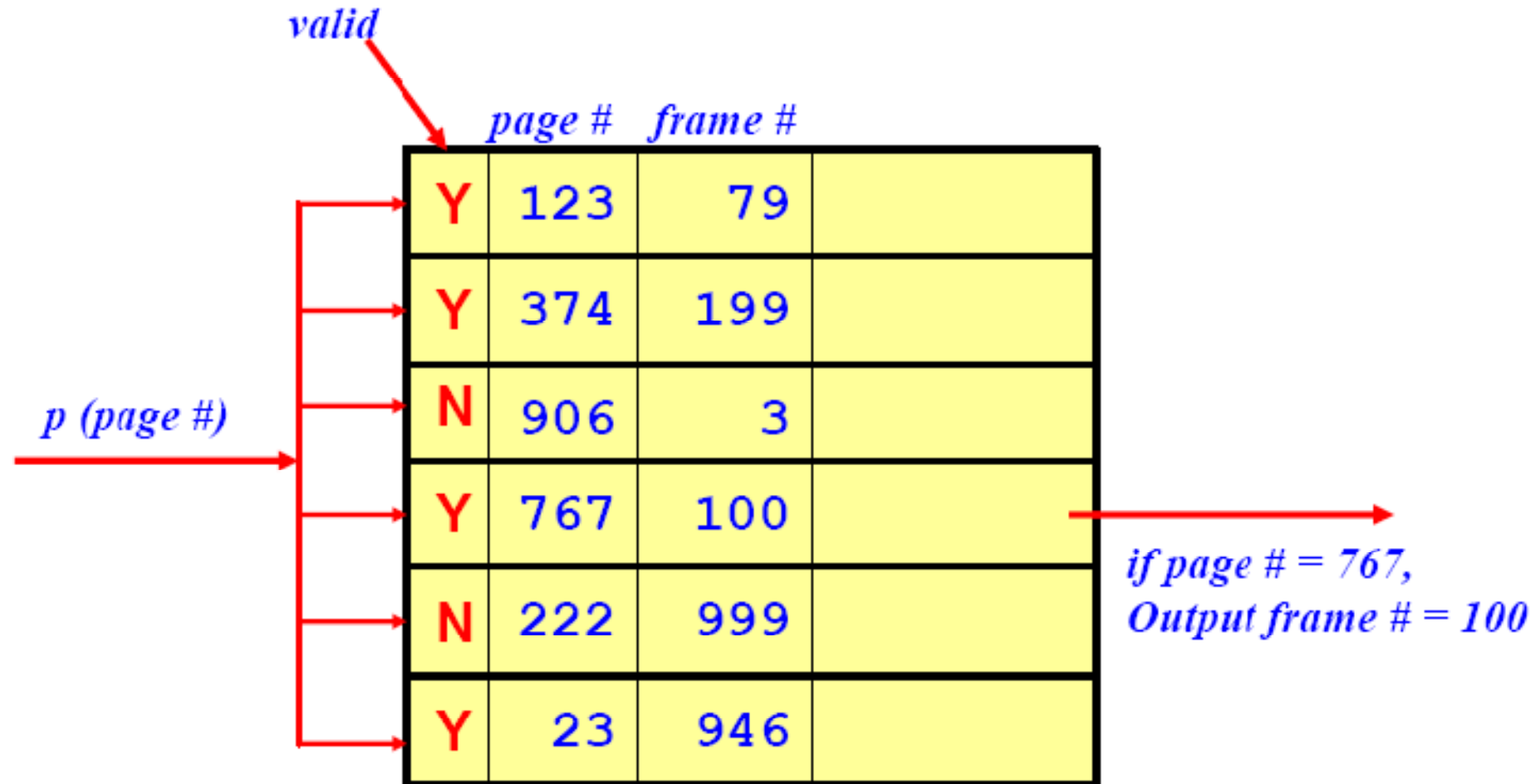
Address translation (A' , A'')

- ◆ If an entry with the key A' can be found in the associative memory, returns the value of frame #
- ◆ Otherwise, get the frame # value from the page table that exists in memory

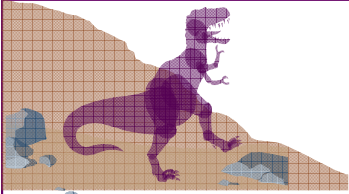




Translation Look-Aside Buffer



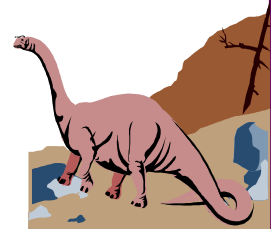
If the TLB reports no hit, then we go for a page table look up!

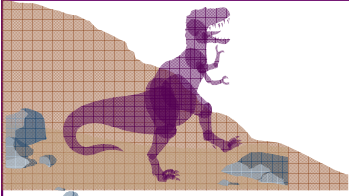


Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 time unit
- Hit ratio – percentage of times that a page number is found in the associative registers
- Hit ratio is related to the number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

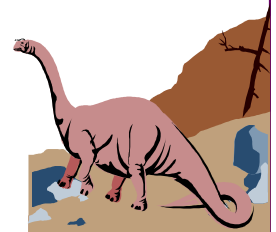
$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

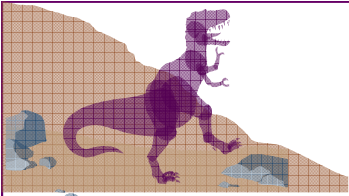




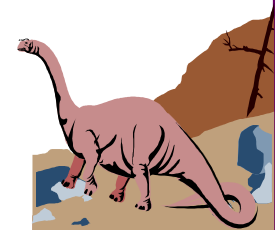
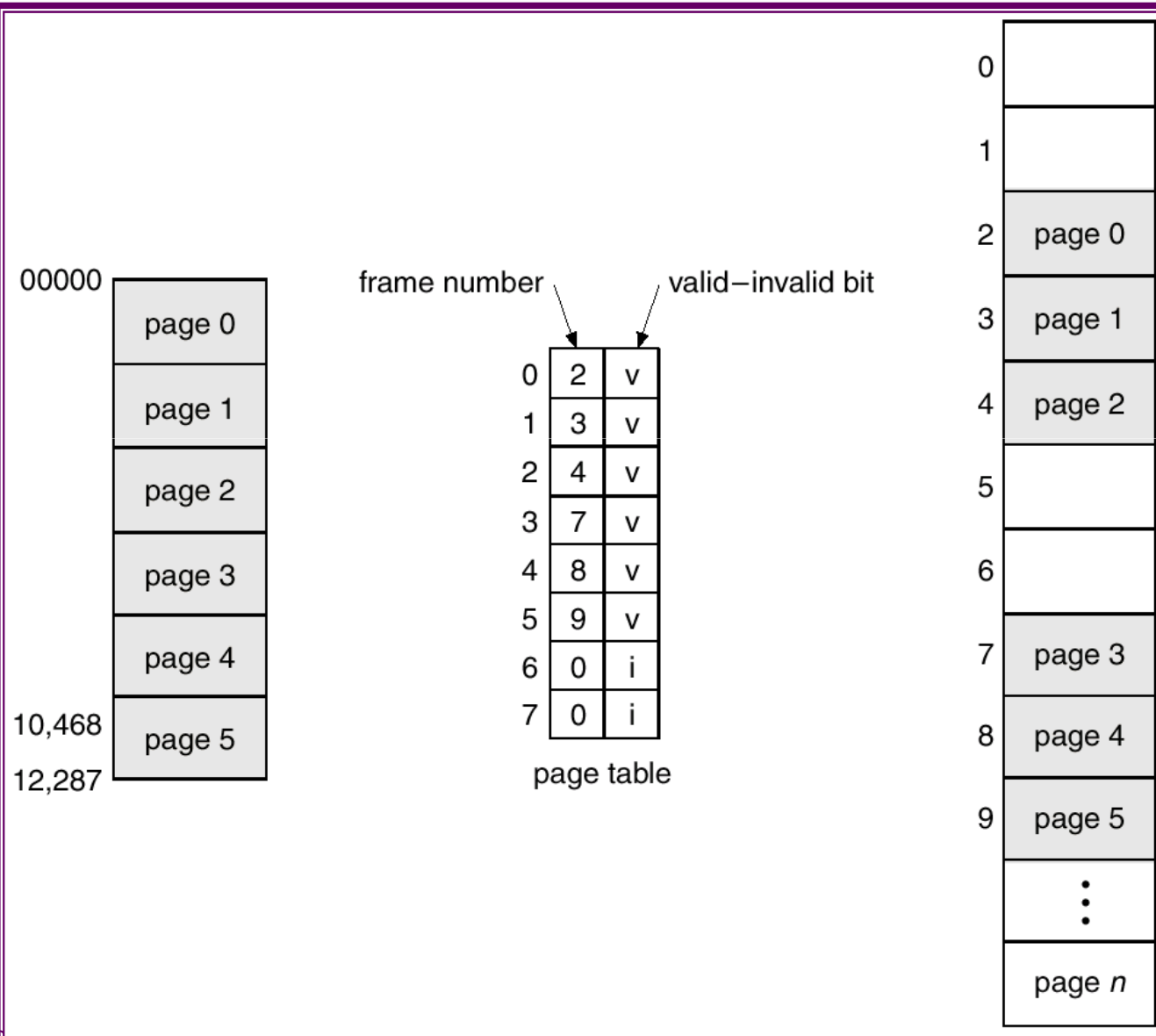
Memory Protection

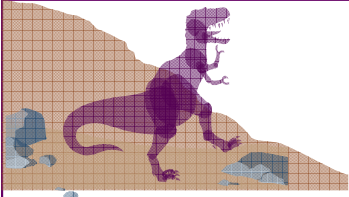
- Memory protection implemented by associating protection bit with each frame.
- *Valid-invalid* bit attached to each entry in the page table:
 - ◆ “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
 - ◆ “invalid” indicates that the page is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





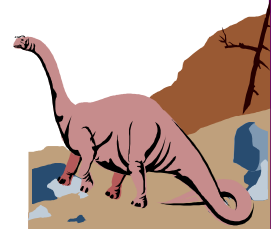
Valid (v) or Invalid (i) Bit in a Page Table Entry

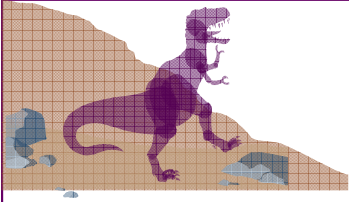




Memory Protection (Cont.)

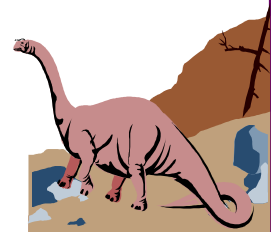
- We can use a *page table length register (PTLR)* that stores the length of a process's page table. In this way, a process cannot access the memory beyond its region.
Compare this with the base/limit register pair.
- We can also add read-only, read-write, or execute bits in page table to enforce **r-w-e** permission.

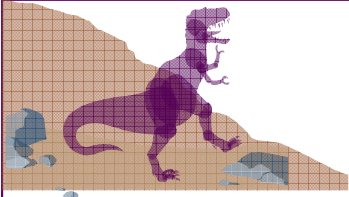




Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

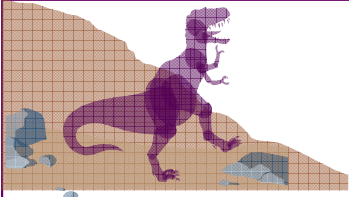




Hierarchical Page Tables

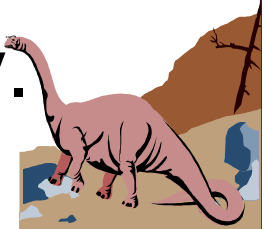
- Why it needs the multiple-level page table?
- **Answer:** A single-level page table may become too big to fit into the physical memory of a commodity machine.
 - ◆ Assume we have a 64-bit computer (which means **64 bit virtual address space**), which has **4KB pages** and **4 GB** of physical memory
 - ◆ In the single-level page table, 2^{64} addressable bytes / 2^{12} bytes per page = 2^{52} page entries
 - ◆ One page table entry contains: Access control bits (like Page present, RW) + Physical page #

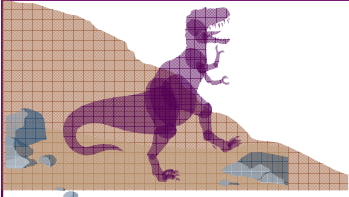




Hierarchical Page Tables

- ◆ **20 bits required for physical page number.**
 - ✓ 4 GB of Physical Memory = 2^{32} bytes.
 - ✓ 2^{32} bytes of memory / 2^{12} bytes per page = 2^{20} physical pages
- ◆ So each page table entry is approximately **4 bytes**. (20 bits physical page number is roughly 3 bytes and access control contributes 1 byte)
- ◆ Now page table size = $2^{52} * 4$ bytes = 2^{54} bytes
- Hence, the size of single-level page table is **2^{54} bytes (16 petabytes) per process**, which is a very huge amount of memory.

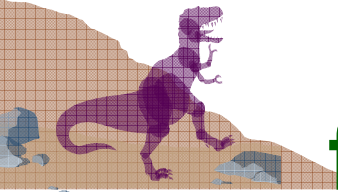




Hierarchical Page Tables

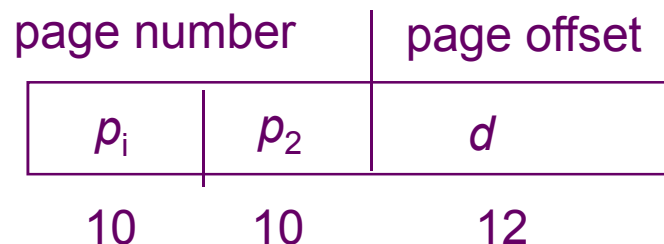
- **Solution:** Break up the logical address space into multiple page tables.
- If we page the page table too, we can magically bring down the memory required
 - ◆ In the first-level page table, **2^{52} page entries**, and 4 KB page / 4 bytes per page table entry = **1024 entries**, i.e., 10 bits of address space required
 - ◆ The second-level page table needs **2^{42} entries**
 - ◆
 - ◆ The fifth-level page table only needs **2^{12} page entries**, as low as four pages, just 16 KB memory



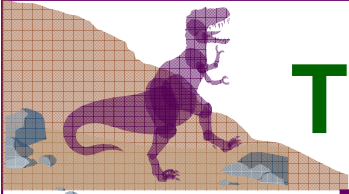


Two-Level Paging Example for 32-bit Operating Systems

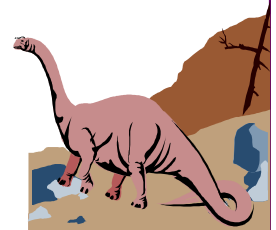
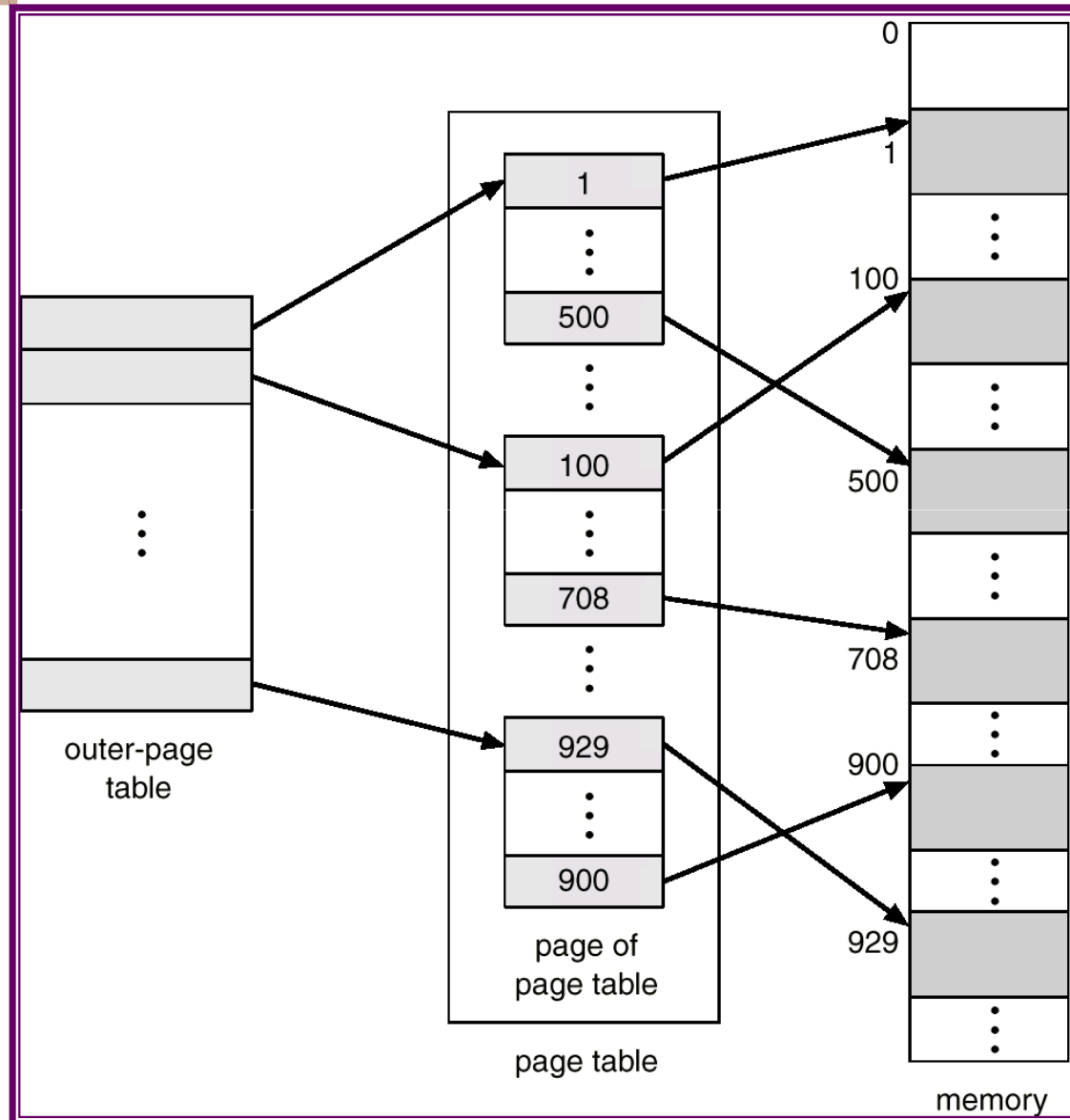
- A logical address (on 32-bit machine with 4K page size) is divided into:
 - ◆ a page number consisting of 20 bits, and
 - ◆ a page offset consisting of 12 bits.
- Since the page table itself is also paged, the page number is further divided into:
 - ◆ a 10-bit page number, and
 - ◆ a 10-bit page offset.
- 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 outer page table.



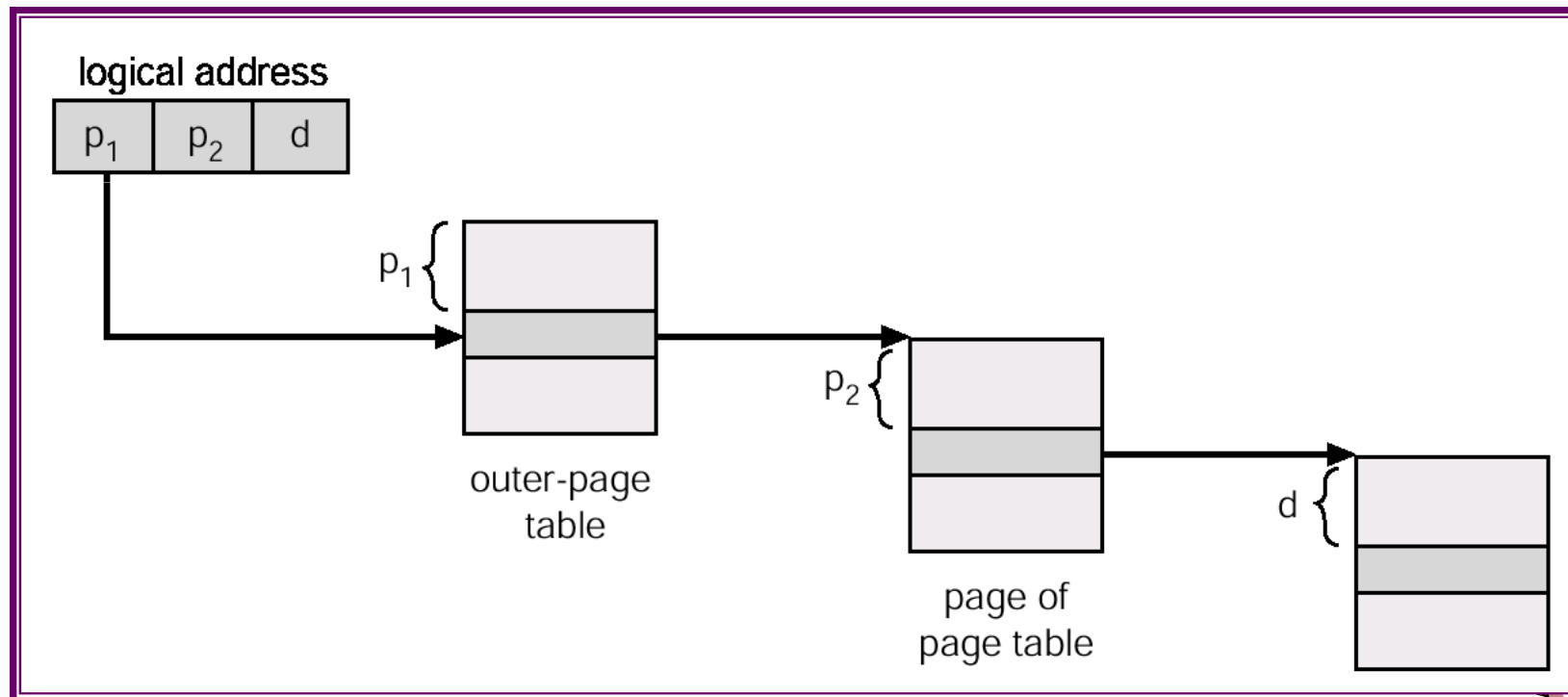
Two-Level Page-Table Scheme

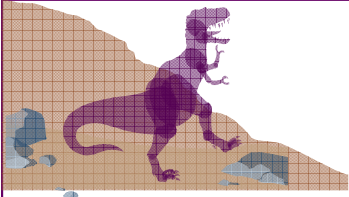




Address-Translation Scheme

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





Three-level Paging Scheme for 64-bit Operating System

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- The 2nd outer page table still needs 16GB memory, since 2^{32} number of table entries X 4 bytes per table entry = 2^{34} bytes memory





Hashed Page Tables

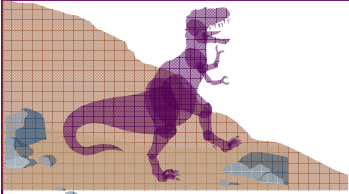
- Common in address spaces > 32 bits.

- **Motivation**

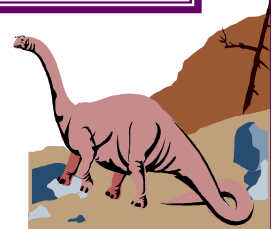
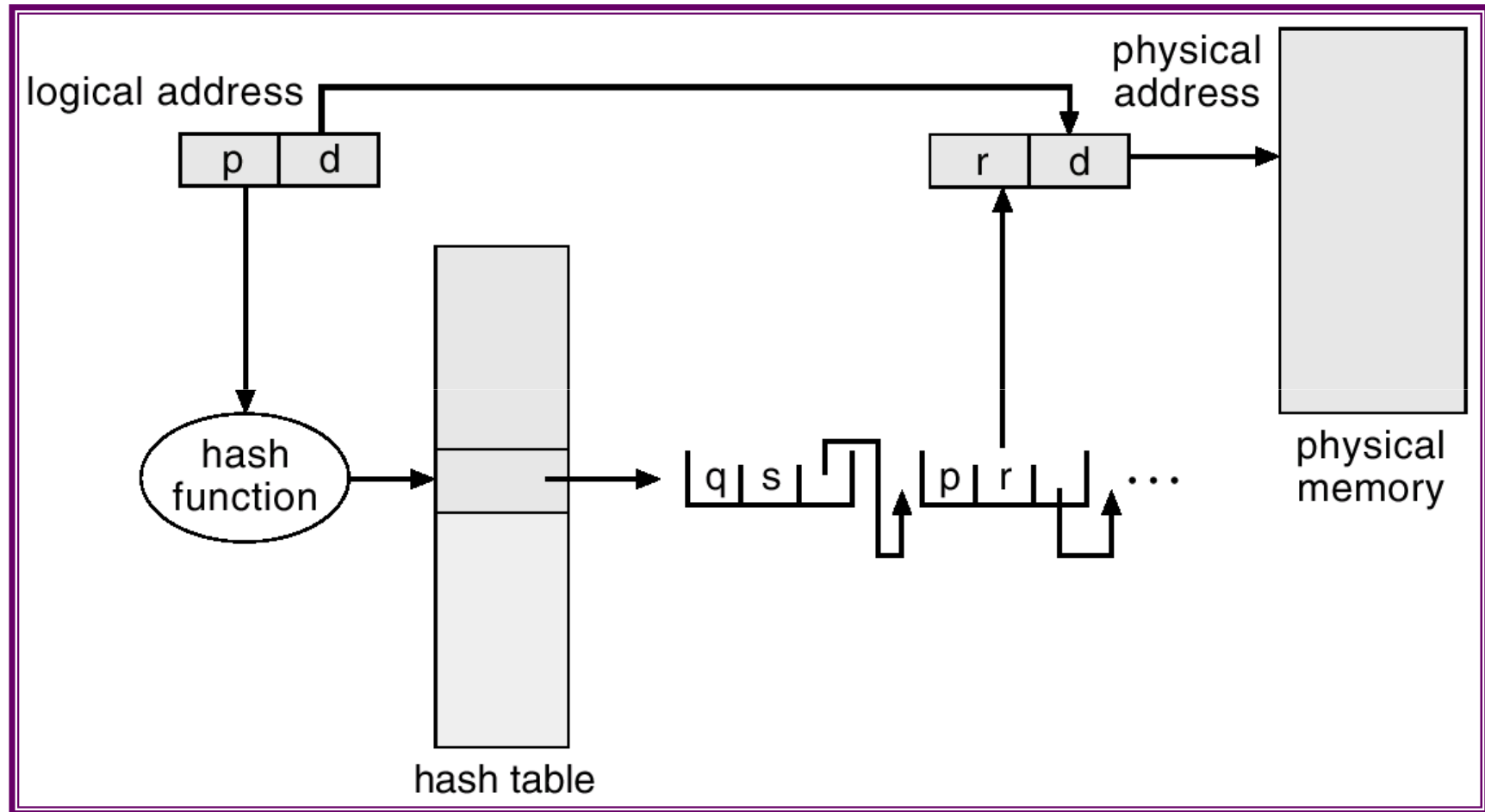
- ◆ On a 64-bit operating system, the third-level page table is still too large to fit in main memory
- ◆ In a 32-bit or 64-bit address space of a process, most part of it is unused

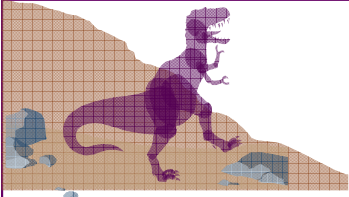
- **Solution** base on Chained Hash Table

- ◆ 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.



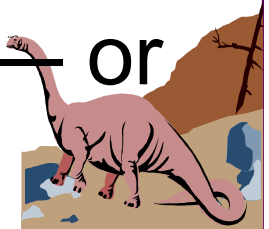
Hashed Page Tables



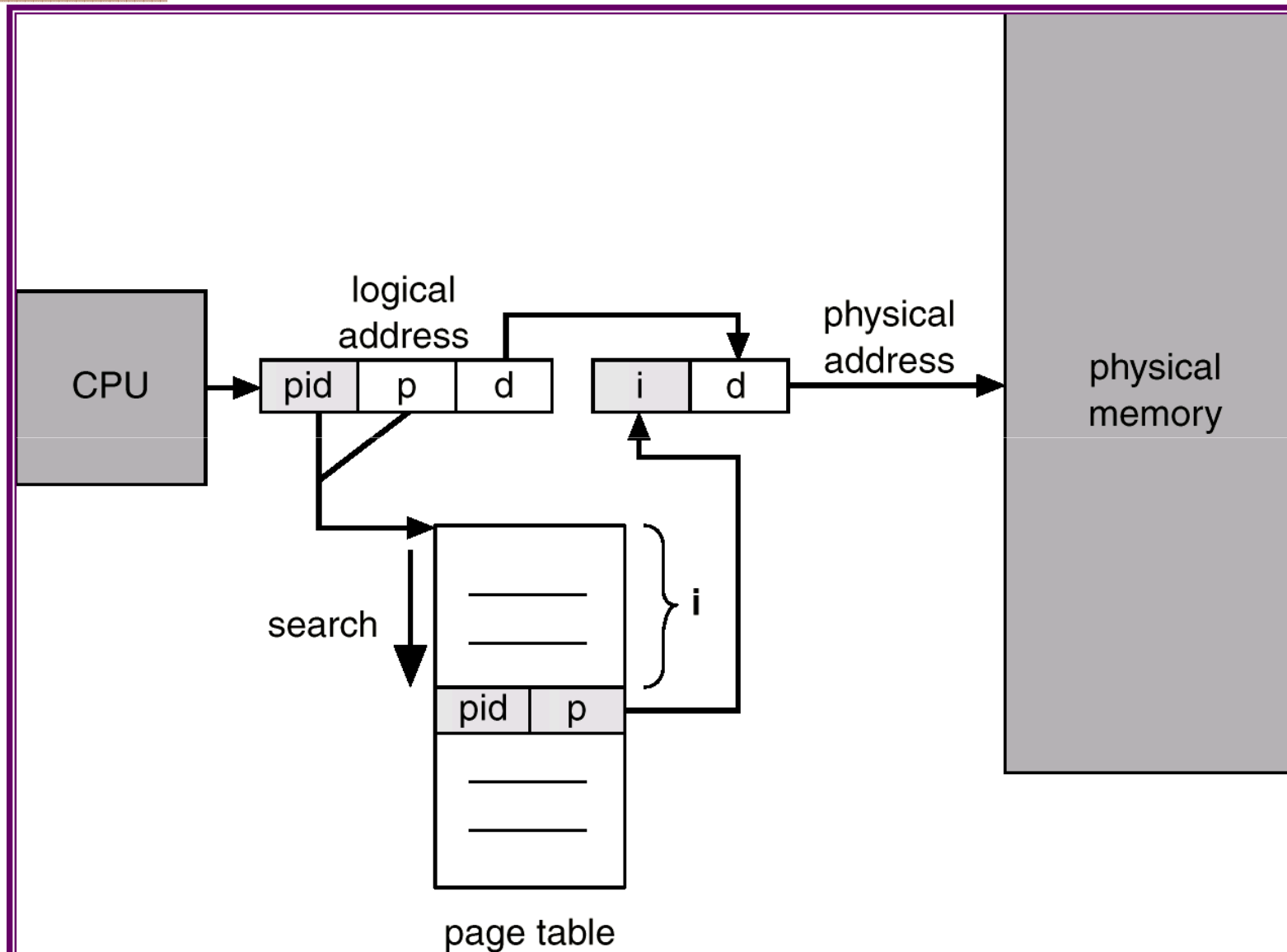


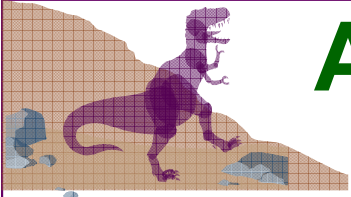
Inverted Page Table

- **Motivation:** All the previous schemes need to maintain a page table for each process.
- One entry for each real page of memory (frame)
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process owning 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 Architecture





Advantage of Paging Method: Shared Pages

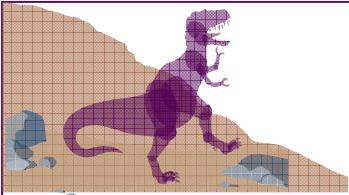
■ Shared code

- ◆ One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

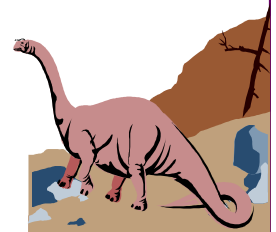
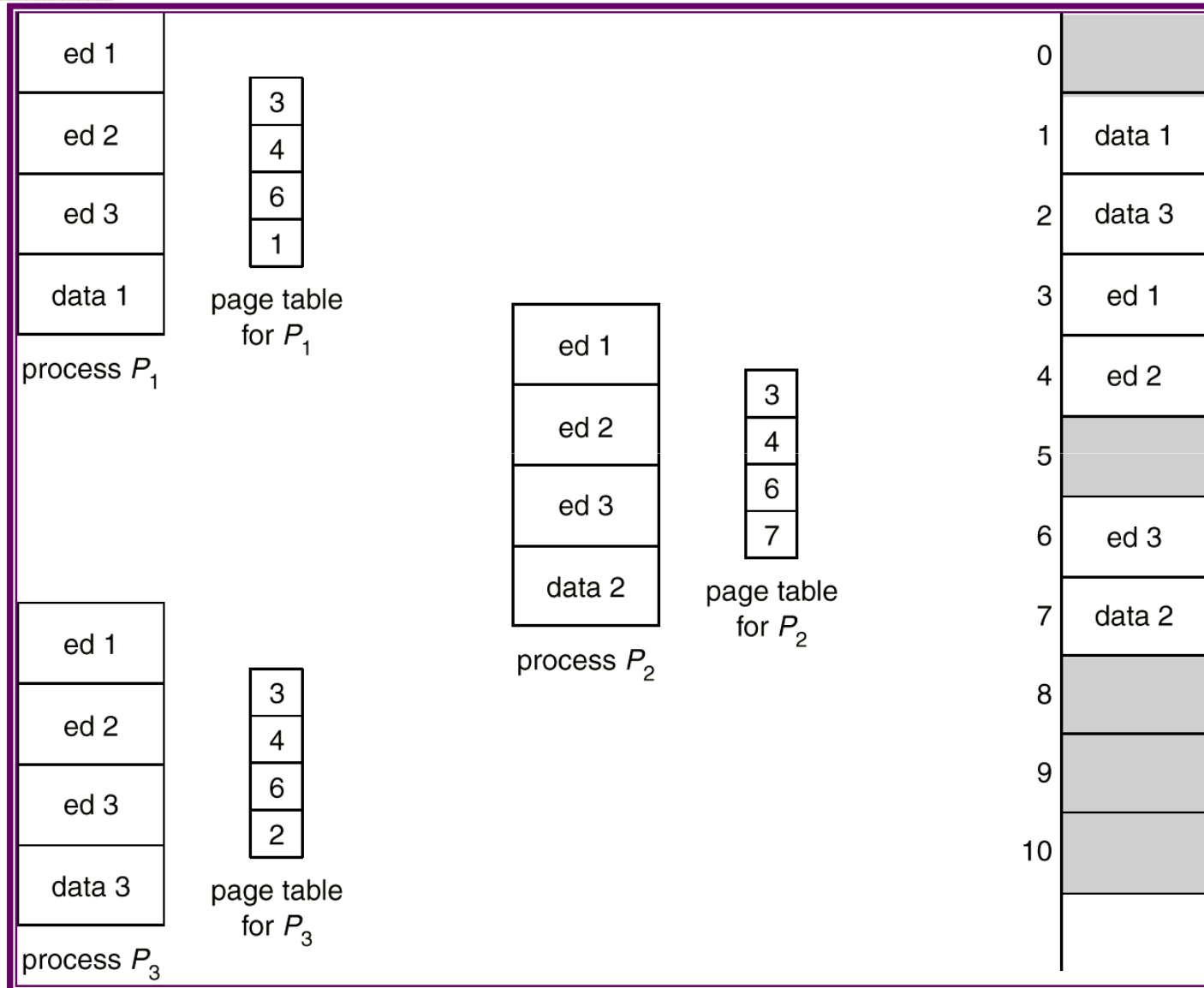
■ Private code and data

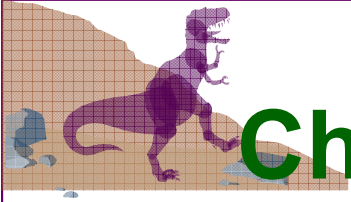
- ◆ Each process keeps a separate copy of the code and data.
- ◆ The pages for the private code and data can appear anywhere in the logical address space.





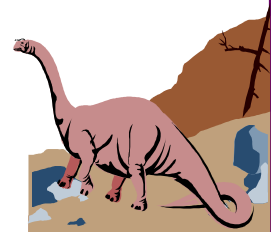
Shared Pages Example

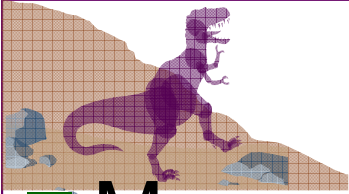




Chapter 9: Memory Management

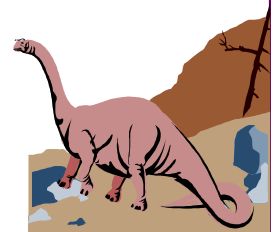
- Background
- Swapping
- Contiguous Allocation
- Paging
- **Segmentation**
- Segmentation with Paging

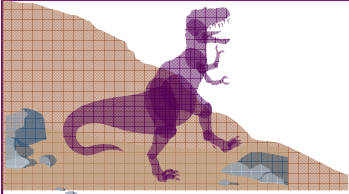




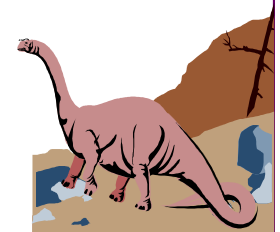
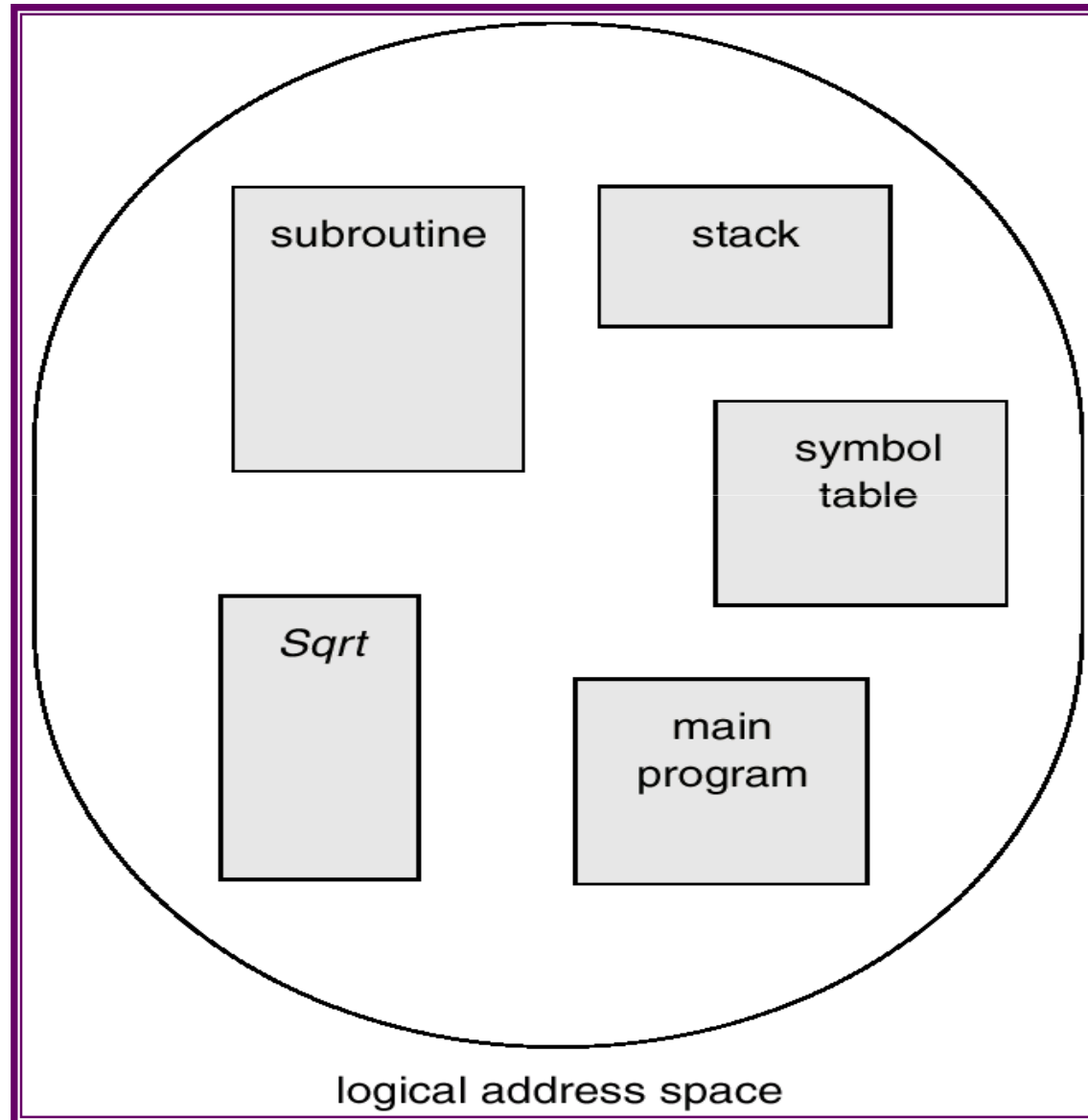
Segmentation

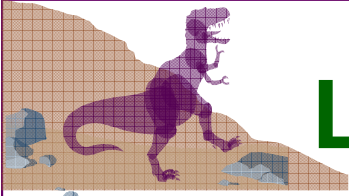
- Memory-management scheme that supports **user view of memory**.
- A program is a collection of segments. A segment is a logical unit such as:
 - main program,
 - procedure,
 - function,
 - method,
 - object,
 - local variables, global variables,
 - common block,
 - stack,
 - symbol table, arrays



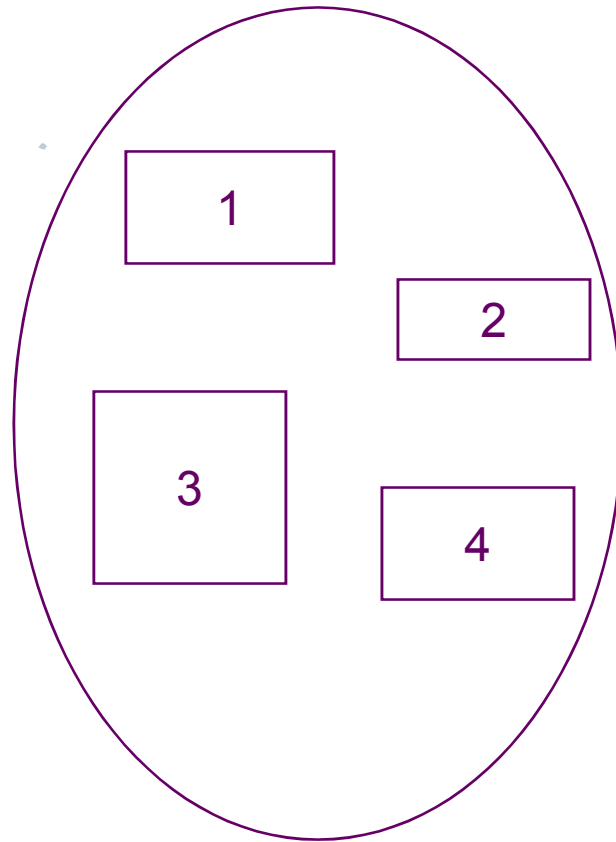


User's View of a Program

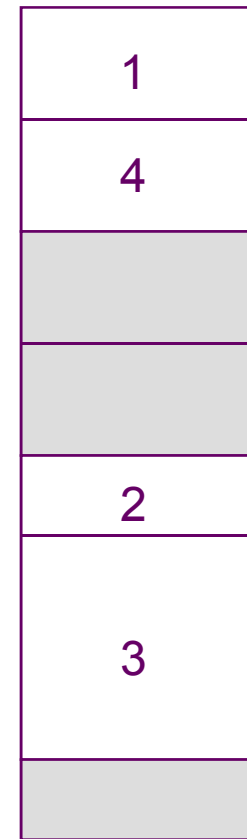




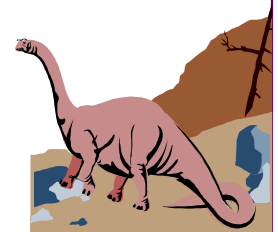
Logical View of Segmentation

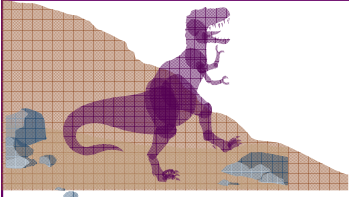


user space



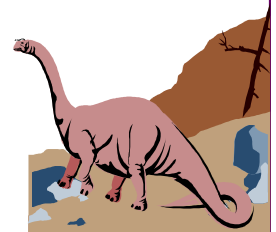
physical memory space



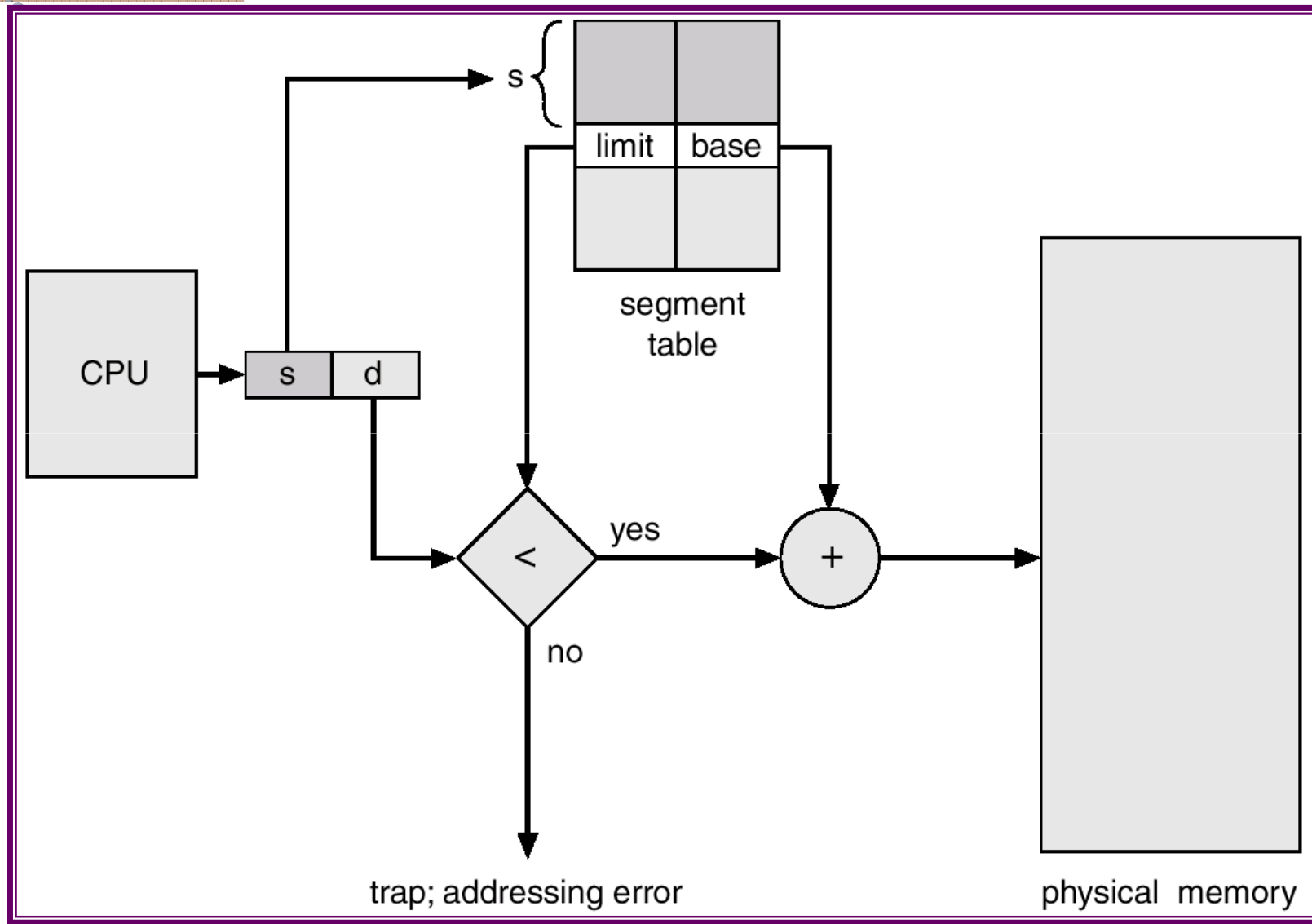


Segmentation Architecture

- 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.



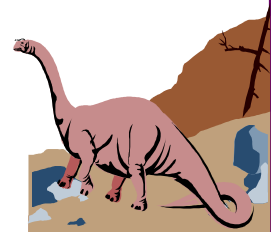
Segmentation Hardware

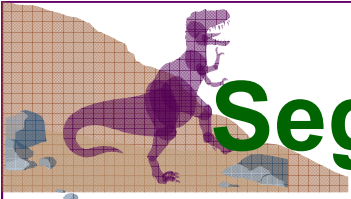




Segmentation Architecture (Cont.)

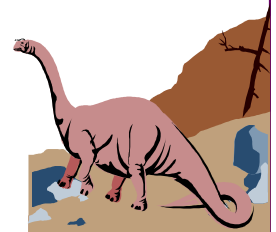
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates the number of segments used by a program; segment number s is legal if $s < \text{STLR}$.



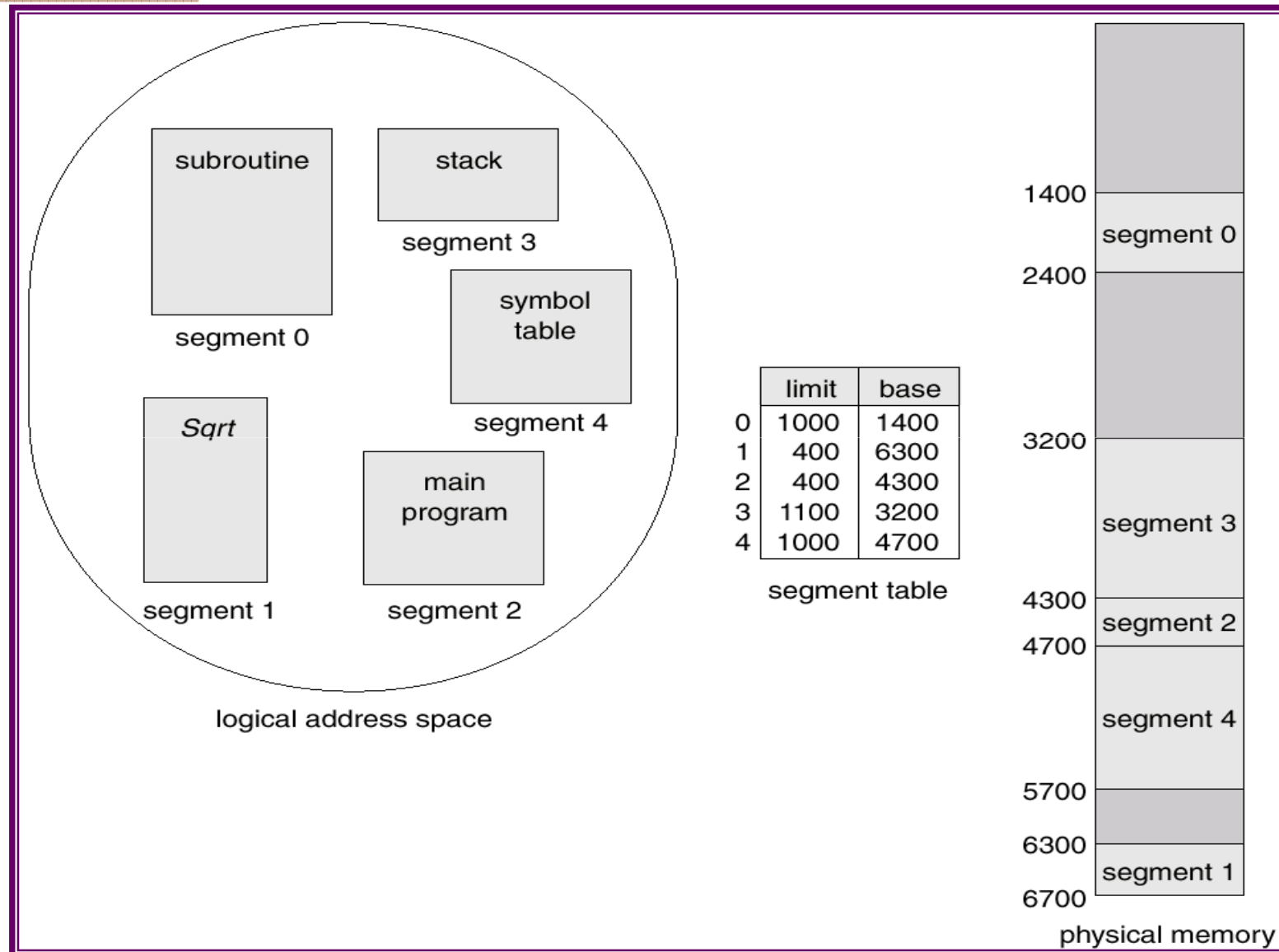


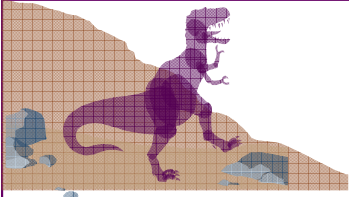
Segmentation Architecture (Cont.)

- Protection. With each entry in segment table, associate:
 - ◆ validation bit = 0 \Rightarrow 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



Example of Segmentation

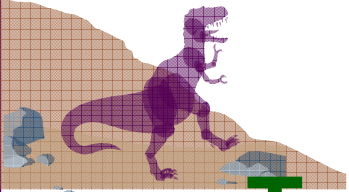




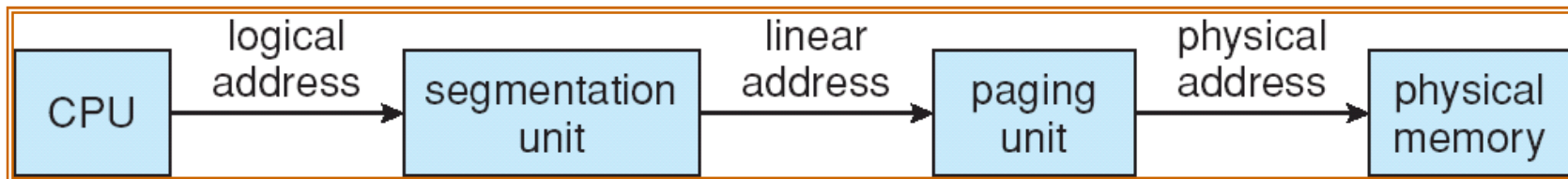
Example: The Intel Pentium

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

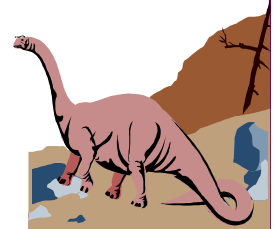


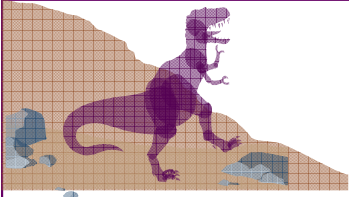


Logical to Physical Address Translation in IA-32 Architecture



page number		page offset
p_1	p_2	d
10	10	12





The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
- CPU generates logical address
 - ◆ Selector given to segmentation unit
 - ✓ Which produces linear addresses
 - ✓ Up to 16K segments per process

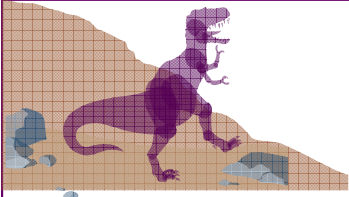
s	g	p
13	1	2

s: segment number
g: whether in GDT or LDT
p: protection bits

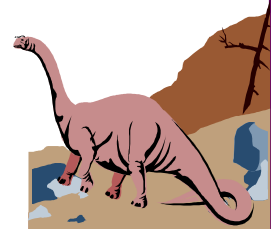
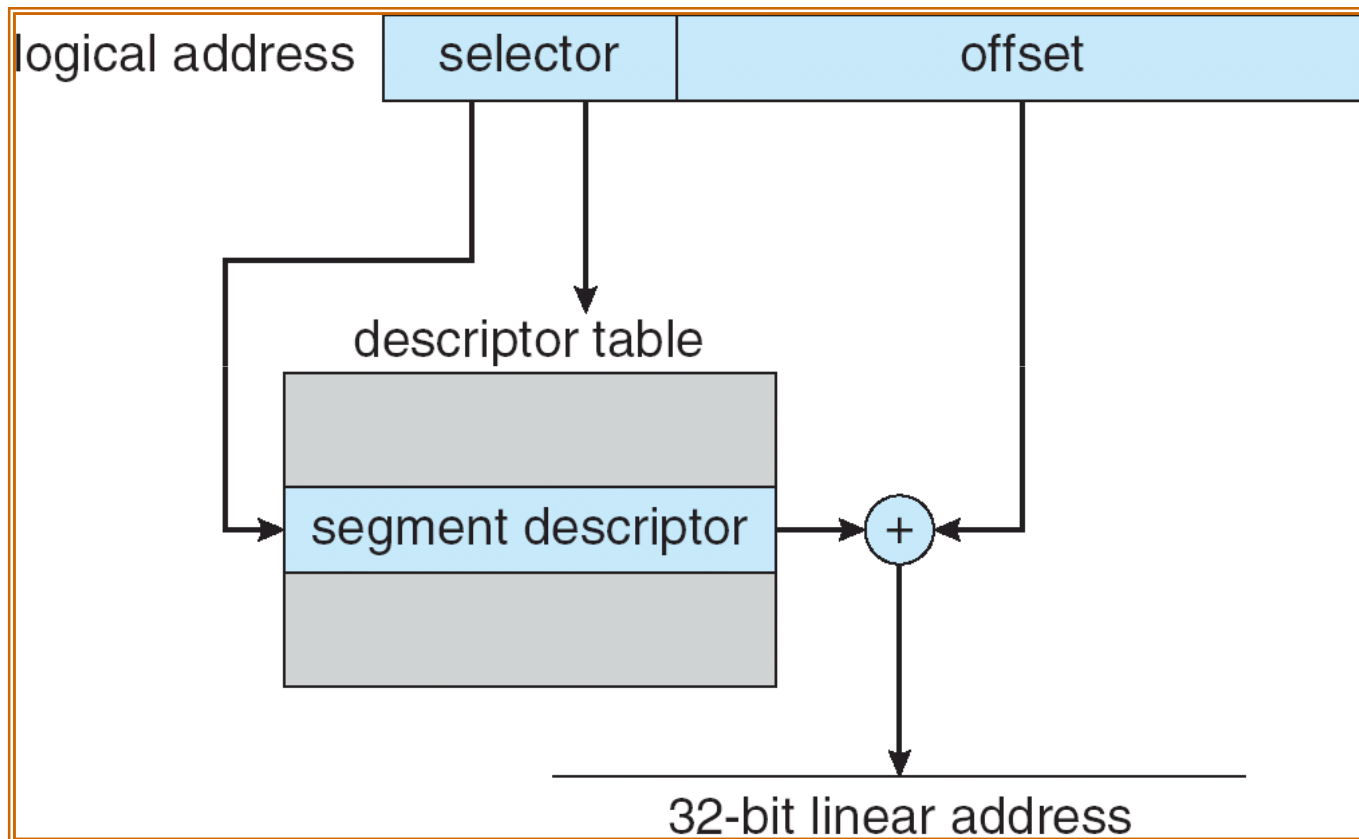
- ◆ Linear address given to paging unit
 - ✓ Which generates physical address in main memory
 - ✓ Paging units form equivalent of MMU

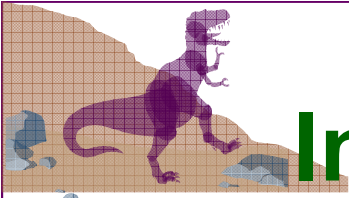


Segmentation with Paging



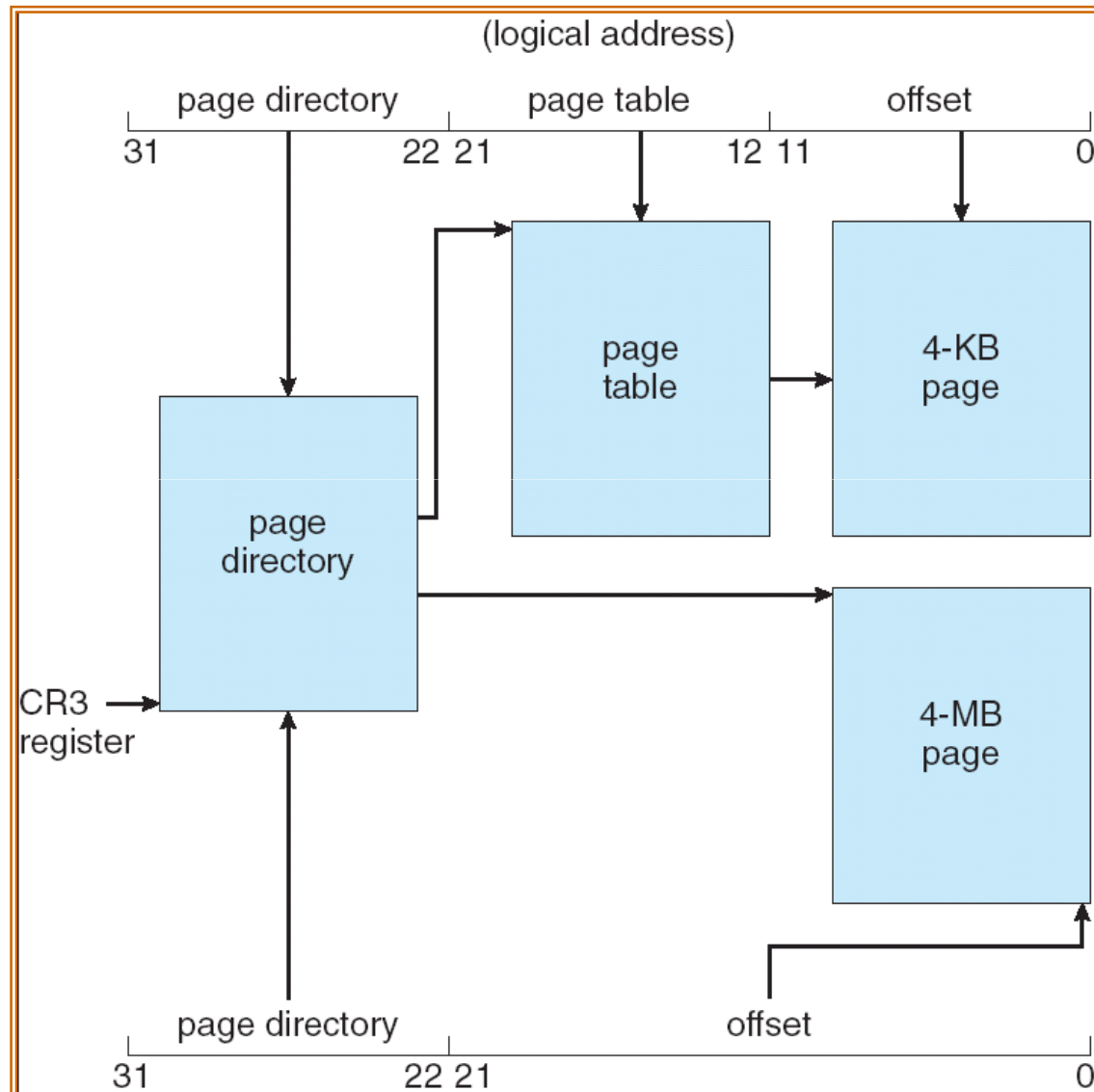
Intel IA-32 Segmentation

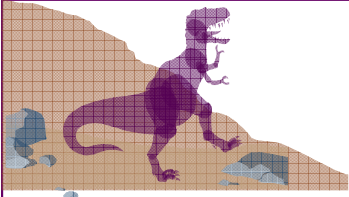




Intel IA-32 Paging Architecture

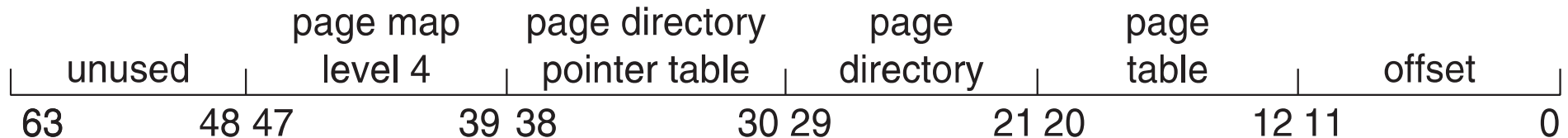
Pages sizes can be either 4 KB or 4 MB





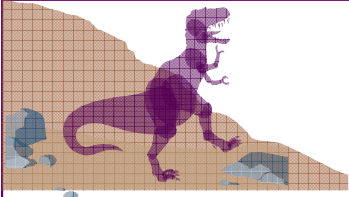
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
 - ◆ Page sizes of 4 KB, 2 MB, 1 GB
 - ◆ Four levels of paging hierarchy



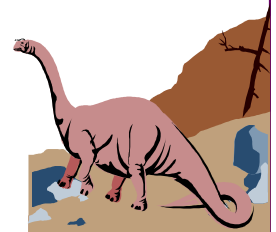
- Can also use PAE (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits



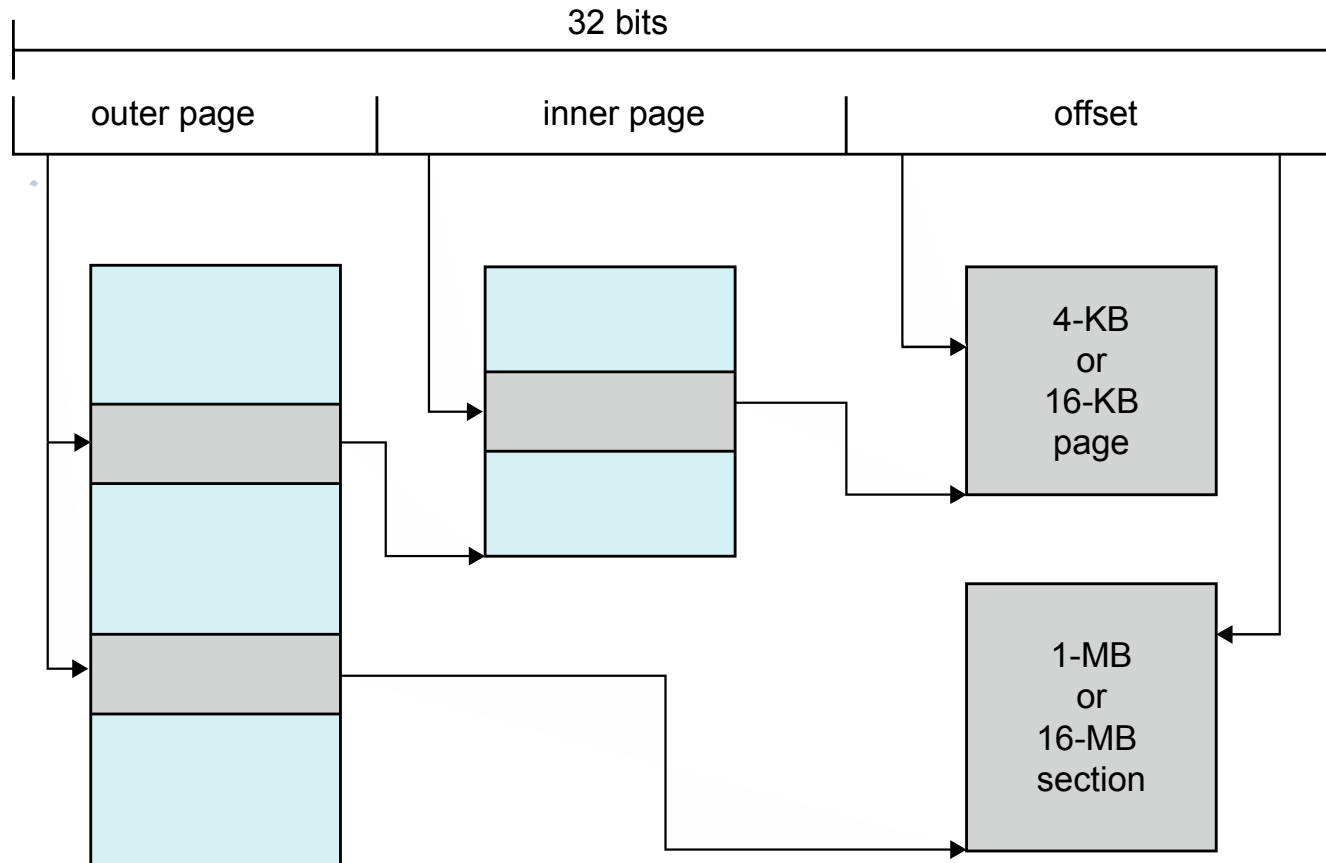


Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages



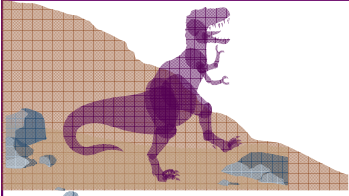
Example: ARM Architecture (Cont.)



■ Two levels of TLBs

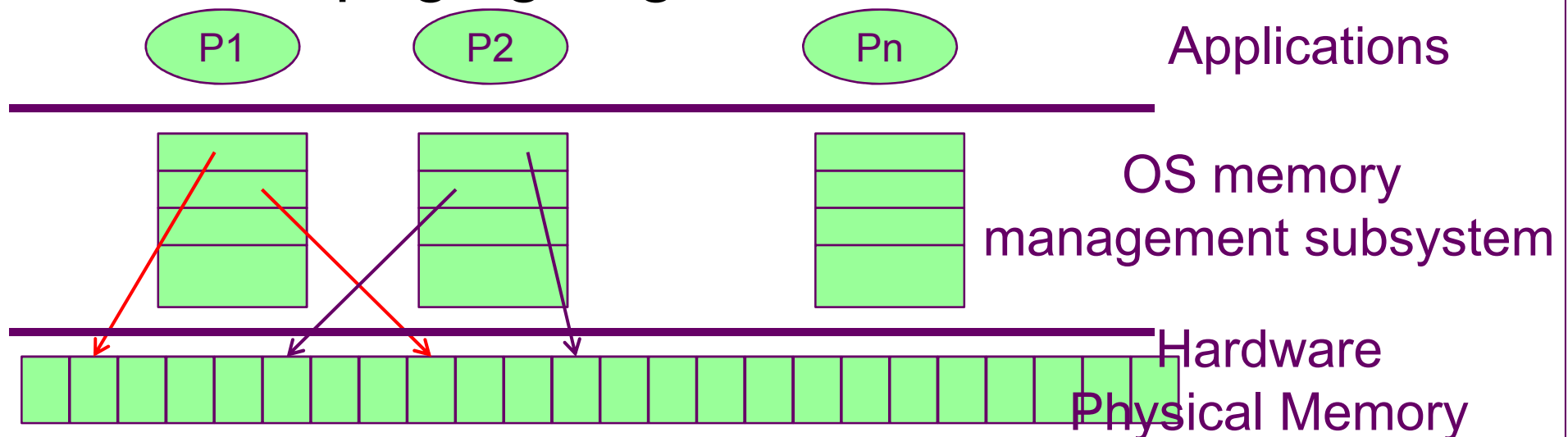
- ◆ Outer level has two micro TLBs (one data, one instruction)
- ◆ Inner is single main TLB
- ◆ First inner is checked, on miss outers are checked, and on miss





Concluding Marks

- OS creates, for each process, an illusion of continuous memory address space, based on the paging/segmentation mechanism



- **Question:** Why the mainstream CPU chips organize the page#-to-frame# mapping table in a hierarchical way, instead of using hashed page table or inverted page table?

