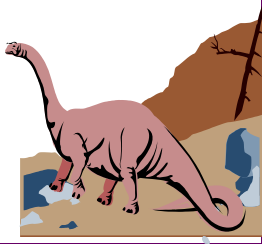
Chapter 8: Memory Management

肖 卿 俊
办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页： https://csqjxiao.github.io/PersonalPage
电话：025-52091022

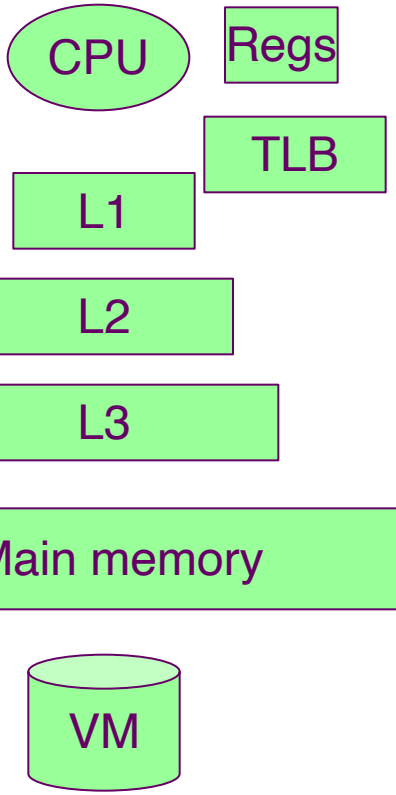
Chapter 8: Memory Management

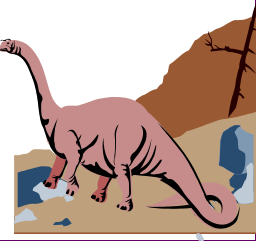
- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging



Background for Memory Hierarchy

- Main memory and registers are the only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **L1/L2/L3 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.





什么是RAM?

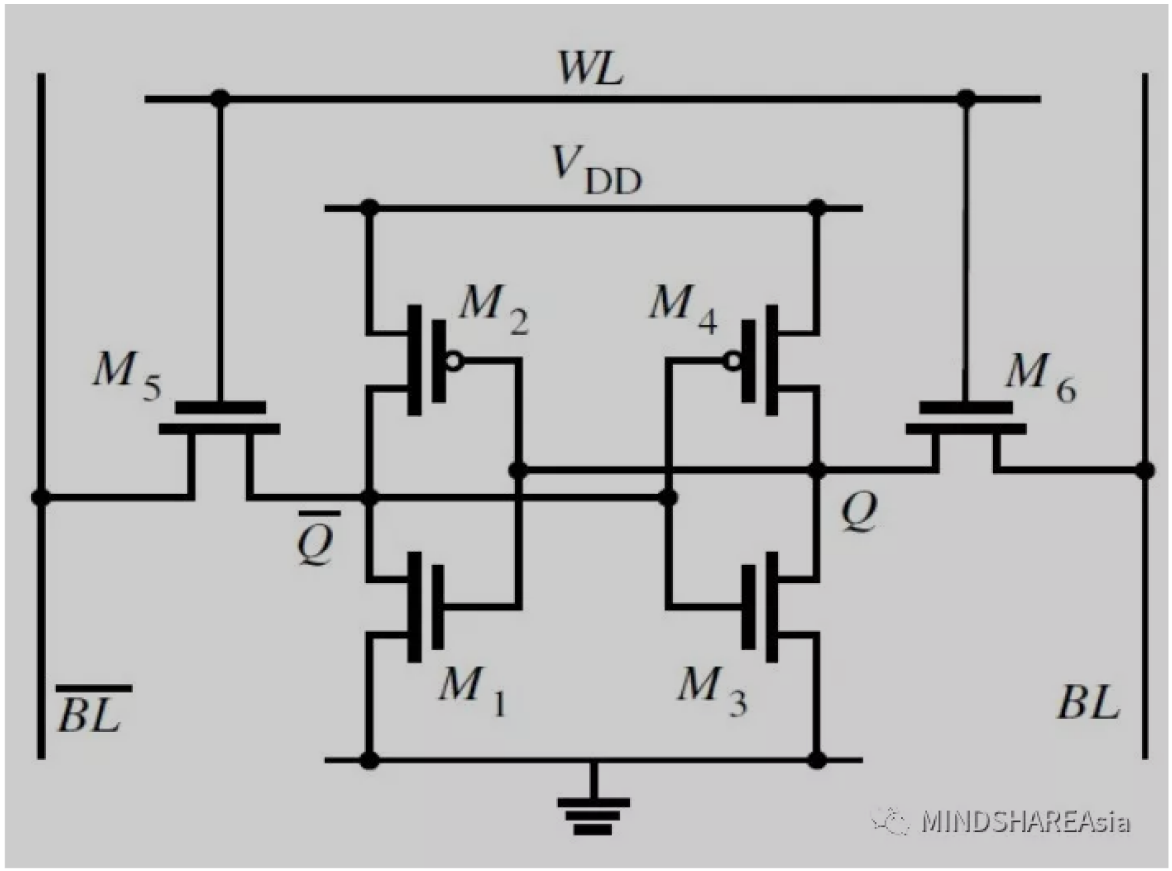
- RAM(Random Access Memory) 中文是随机存取存储器。为什么要强调随机存储呢？因为在此之前，大部分的存储器都是顺序存储（Direct-Access)，比较常见的如硬盘，光碟，老式的磁带，磁鼓存储器等等。随机存取存储器的特点是其访问数据的时间与数据存放在存储器中的物理位置无关。

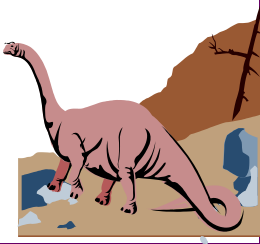
- RAM的另一个特点是易失性（Volatile),虽然业界也有非易失(non-volatile）的RAM，例如，利用电池来维持RAM中的数据等方法。

- RAM主要的两种类别是SRAM（Static RAM）和DRAM（Dynamic RAM）。SRAM的S是Static的缩写，全称是静态随机存取存储器。而DRAM的D是Dynamic的缩写，全称是动态随机存取存储器。

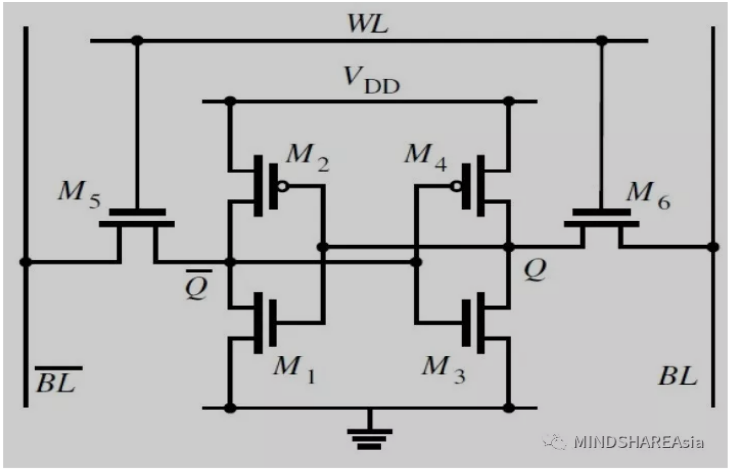
SRAM

- SRAM结构：6场效应管组成一个存储bit单元





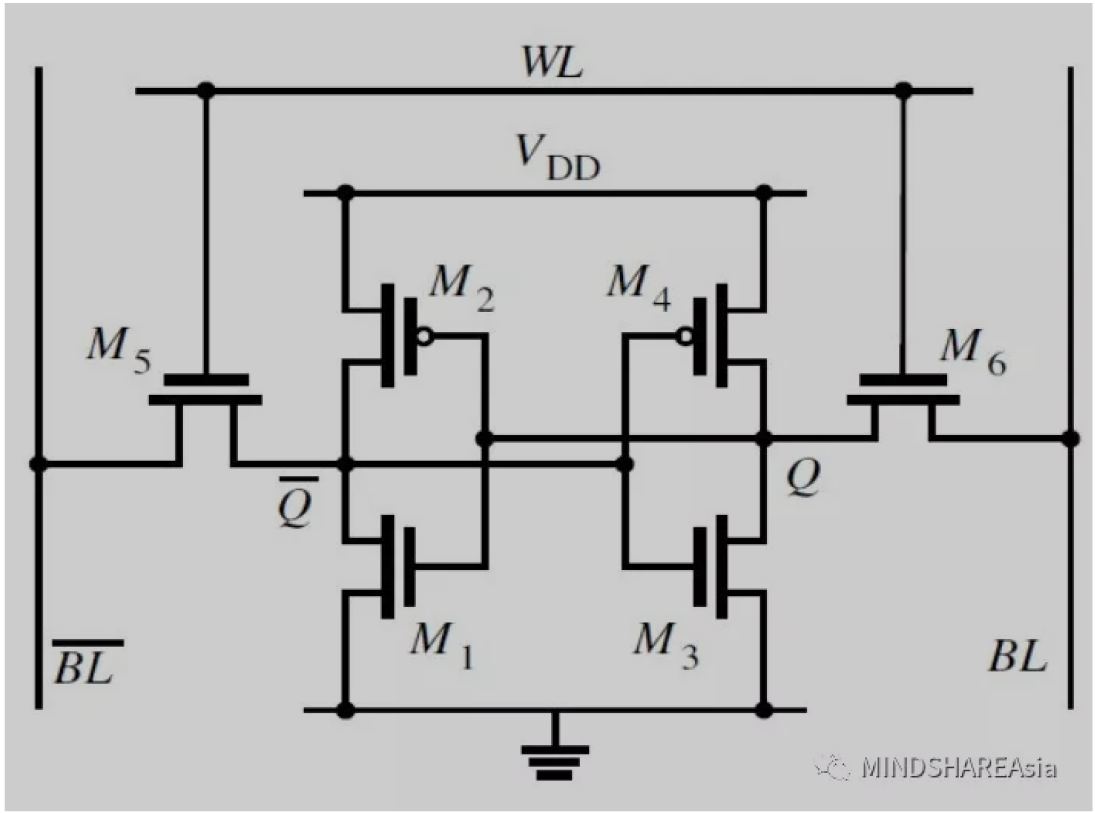
SRAM写操作(以写0为例)

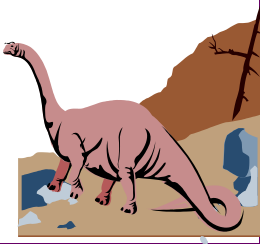


- 首先将BL输入0电平，(~ BL)输入1电平。
- 然后，相应的Word Line(WL)选通，则M5和M6将会被打开。
- 0电平输入到M1和M2的G极控制端
- 1电平输入到M3和M4的G极控制端
- 因为M2是P型管，高电平截止，低电平导通。而M1则相反，高电平导通，低电平截止。
- 所以在0电平的作用下，M1将截止，M2将打开。(~ Q)点将会稳定在高电平。
- 同样，M3和M4的控制端将会输入高电平，因NP管不同，M3将会导通，而M4将会截止。Q点将会稳定在低电平0。
- 最后，关闭M5和M6，内部M1,M2,M3和M4处在稳定状态，一个bit为0的数据就被锁存住了。
- 此时，在外部VDD不断电的情况下，这个内容将会一直保持

SRAM读操作

- 读操作相对比较简单，只需要预充BL和（~BL）到某一高电平，然后打开M5和M6，再通过差分放大器就能够读出其中锁存的内容。

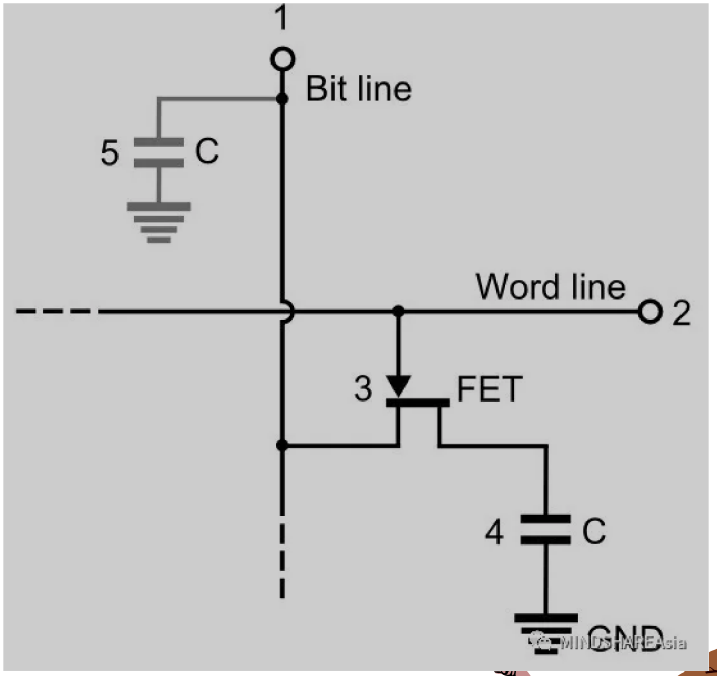




DRAM

- DRAM（Dynamic RAM）是指动态随机存取存储器。与SRAM最大的不同是，DRAM需要通过刷新操作来保持其存储的内容。右下角为其一个bit存储单元（Cell）的结构图。

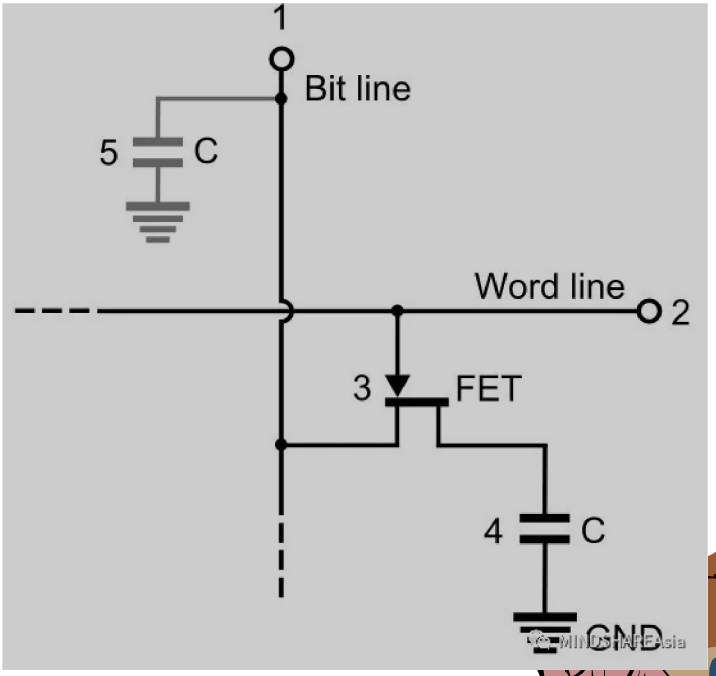
- 其核心部件是4号位的电容C，这个电容大小在pF级别，用来存储0和1的内容。由于电容会慢慢放电，其保存的内容将会随时间推移而慢慢消失。为了保证其内容的完整性，我们需要把里面的内容定期读出来再填写回去。这个操作称为刷新操作（Refresh）。



DRAM写操作

- 写1时，先将BL（Bit Line）输入高电平1，然后选中对应的Word Line（同一时间将只有一根WL被选中），打开相应的MOS管，如图中所示3号位。此时，外部驱动能力很强，通过一定的时间，4号位的电容将会被充满。此时，关闭3号位的MOS管。内容1将在一定时间内被保存在4号位的电容中。

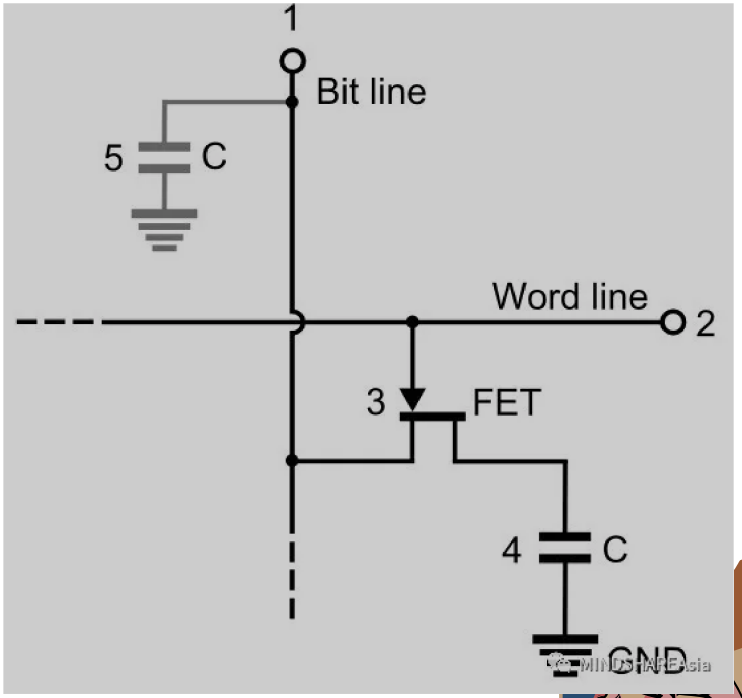
- 写0的操作与之相反，不同的是将4号位电容中的电荷通过Bit Line放光。然后关闭3号位的MOS管，锁存相应数据。



DRAM读操作

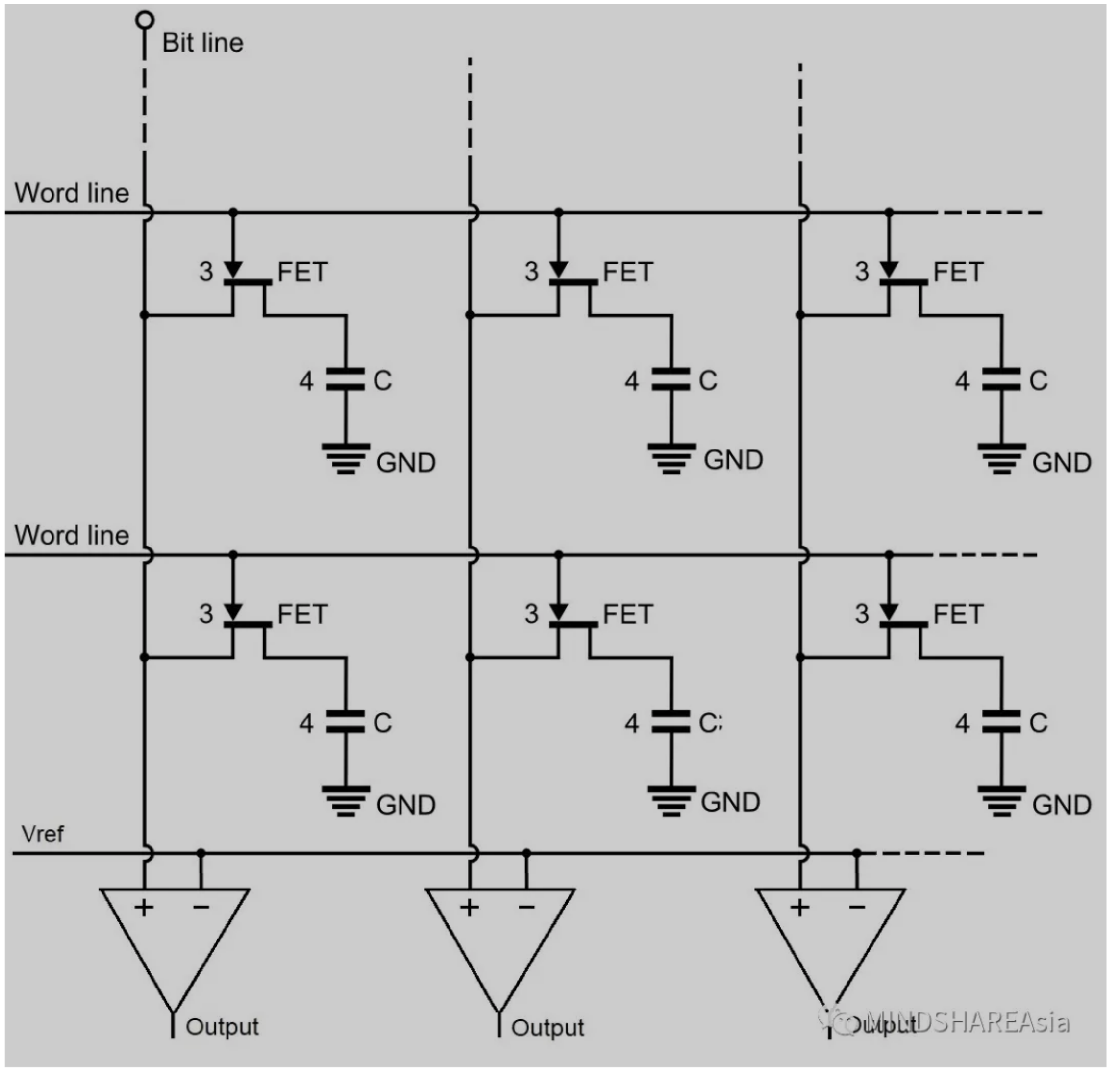
- DRAM读操作相对来说，较为复杂。我们可以观察到4号位电容非常小，只有Pf级别，而Bit Line往往都很长，上面挂了非常多个存储单元（cell），我们可以通过5号位的电容来表示。所以当我们直接把3号位的MOS管打开，Bit Line上将基本看不到什么变化。

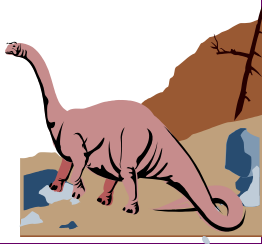
- 于是有人提出是否能够采用放大器来放大4号位电容的效果。



DRAM读操作

- 采用放大器来放大4号位电容的效果，其结构图如下图所示:

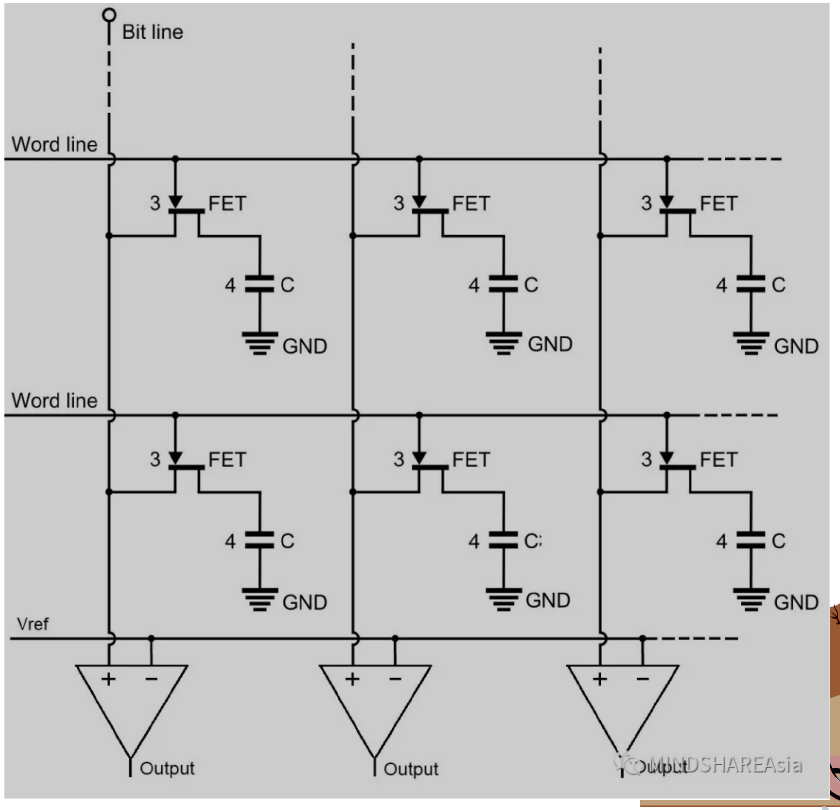




DRAM读操作

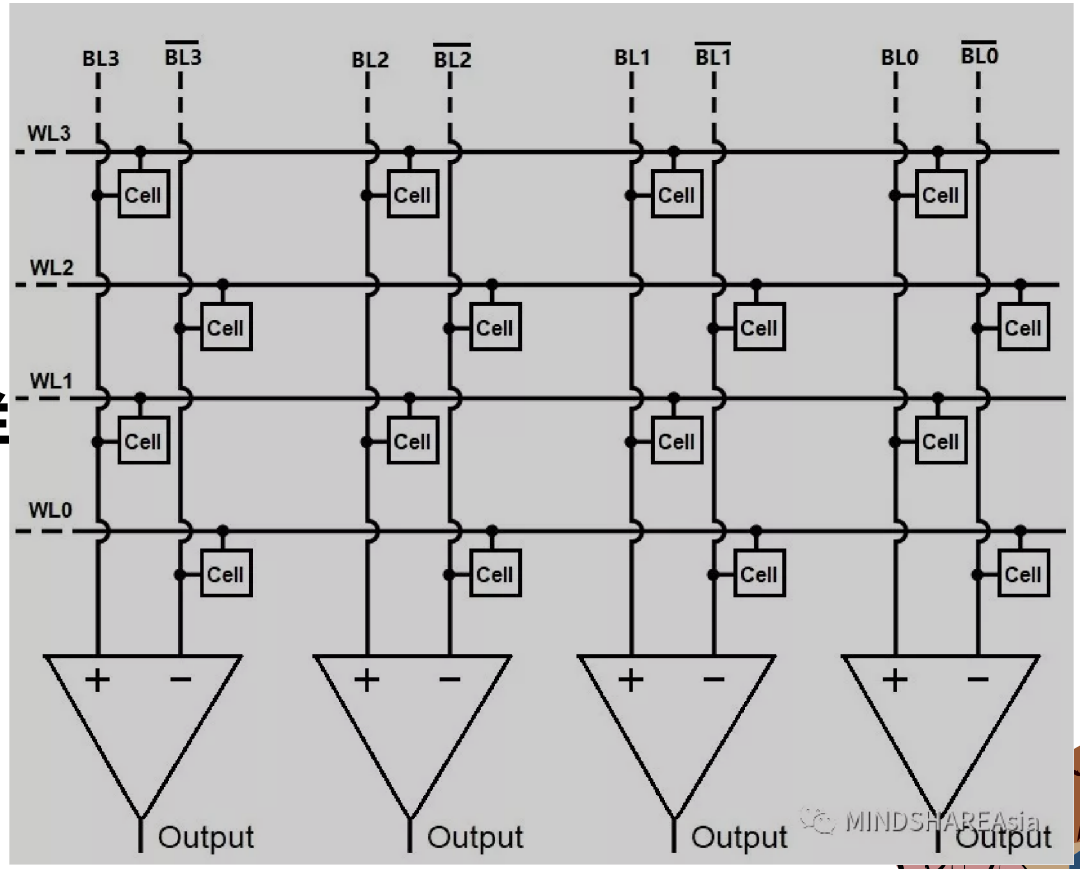
- 我们可以定Vref为1/2的VDD电压，在读取电容里数据之前，我们先将所有Bit Line预充1/2 VDD的电压。然后，打开Word Line让选中的电容连接到Bit Line上面，如果原本的内容是1，则Bit line的总电压将会小幅攀升。否则，则会小幅下降。再通过差分放大器，将结果放大从而实现读操作。

- 这套方案是可以工作的，但Bit Line的数量不能太大。否则会导致距离Vref供电处较远的放大器Vref的值偏低，而导致差分放大器工作异常。同时，对于所谓的1/2 VDD预充，也存在不准的情况。



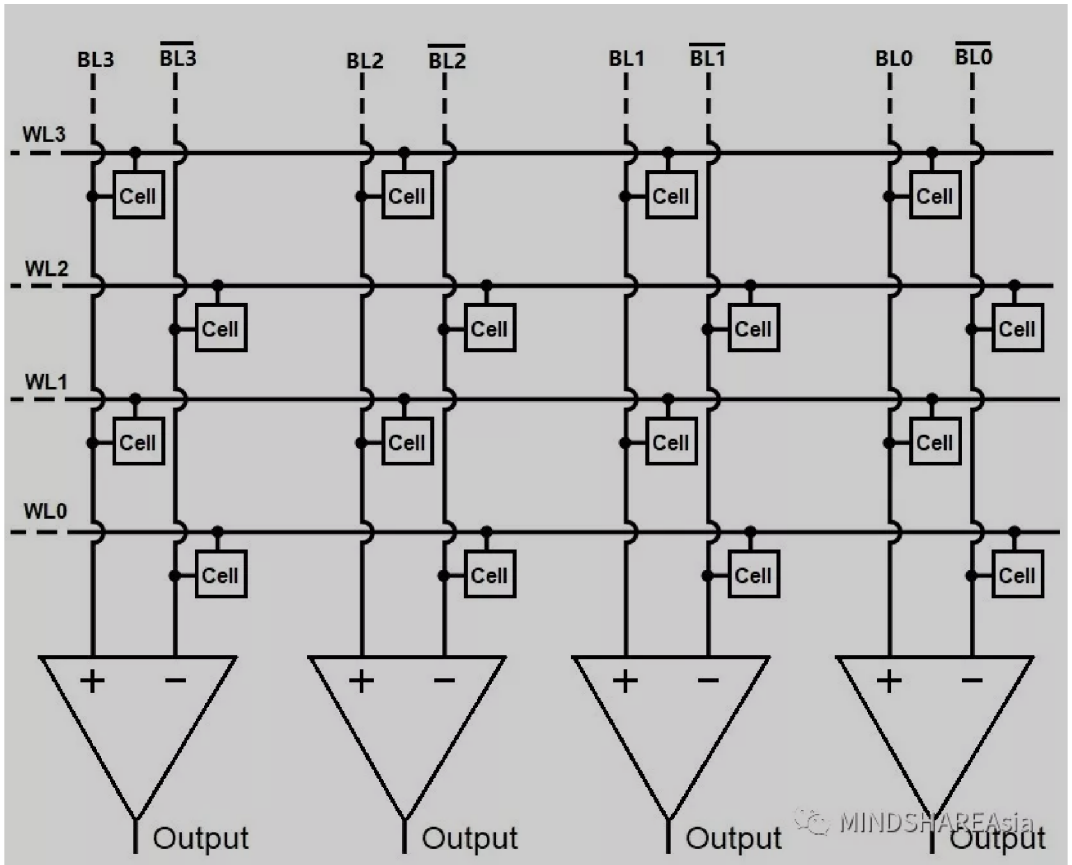
DRAM读操作

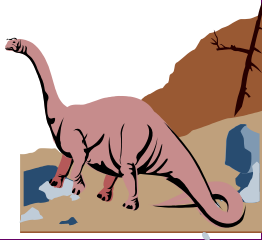
- 为了解决这个问题，有人提出，不如将原来的一根Bit Line设计成一对Bit Line，当其中一根Bit Line上的Cell被选中时，另一根Bit Line将不会有Cell被选中。从而没有Cell被选中的Bit Line可以充当放大器的Vref输入，其长度，负载以及寄生参数将会和另一根Bit Line十分一致，这样一来，放大器的工作就更加稳定了。结构如右图所示。



DRAM读操作

- 当读操作之前，我们先将1/2 VDD电压同时注入到BL和（~BL）上，这个动作被称为（pre-charge 预充电）然后其中一根作为参考，来观察另一根Bit Line在某个Cell导通后的变化。





SRAM与DRAM的区别

- SRAM成本比较高（6 个场效应管组成一个存储单元）

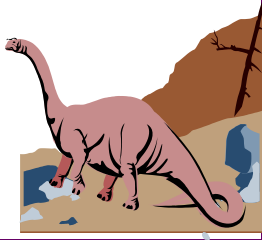
  DRAM成本较低（1个场效应管加一个电容）

- SRAM存取速度比较快

  DRAM存取速度较慢（电容充放电时间）

- SRAM一般用在高速缓存中

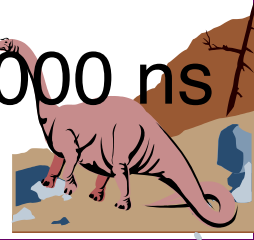
  DRAM一般用在内存条里



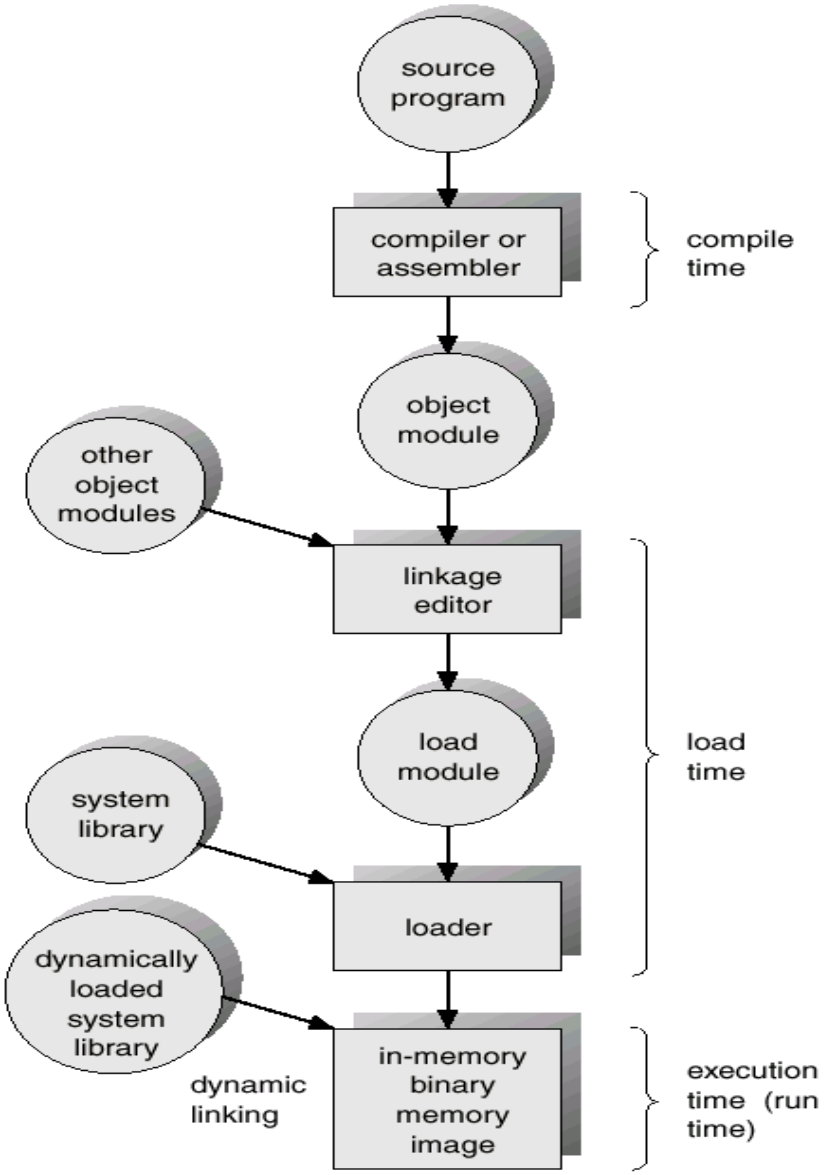
Some numbers everyone should know (from the legendary Jeff Dean)

- L1 cache reference 0.5 ns (instruction & data cache)
- L2 cache reference 7 ns
- Main memory reference/RAM (cache miss) 100 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

http://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/stanford-295-talk.pdf



Background on Multistep Processing of a User Program



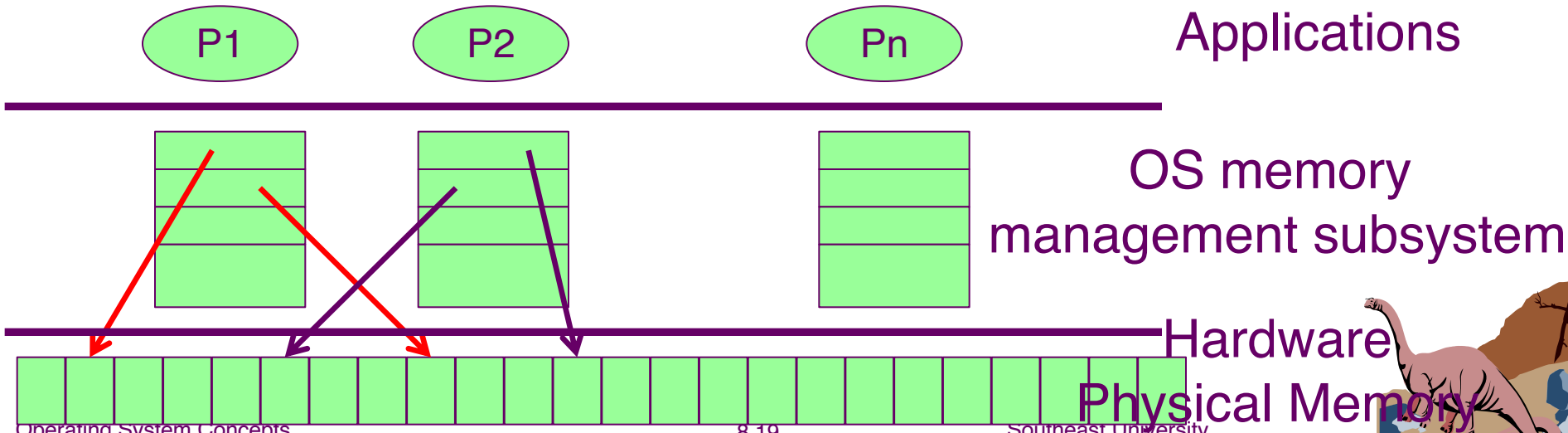
Binding of Instructions and Data to Physical Memory Addresses

- **Compile time**
  - If memory location of running a program is known a priori, absolute code can be generated by compiler; 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 mappings.

Most general-purpose operating systems use the execution-time address binding

Logical Address Space vs. Physical Address Space

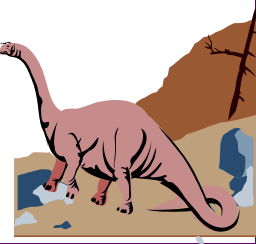
- The concept of a *logical address space* that is bound to a separate *physical address space* is central to the proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by memory unit.



Logical Address Space vs. Physical Address Space (cont.)

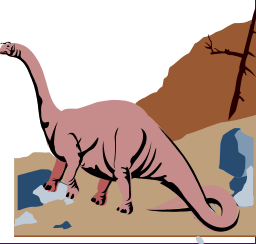
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes

- 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)



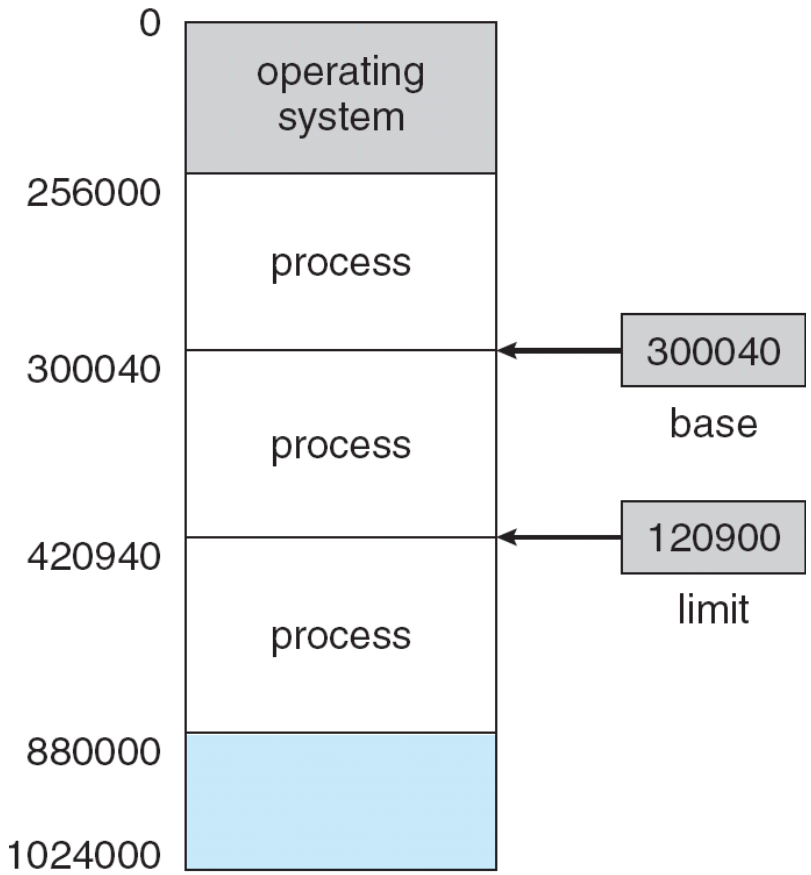
Memory-Management Unit (MMU)

- 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.

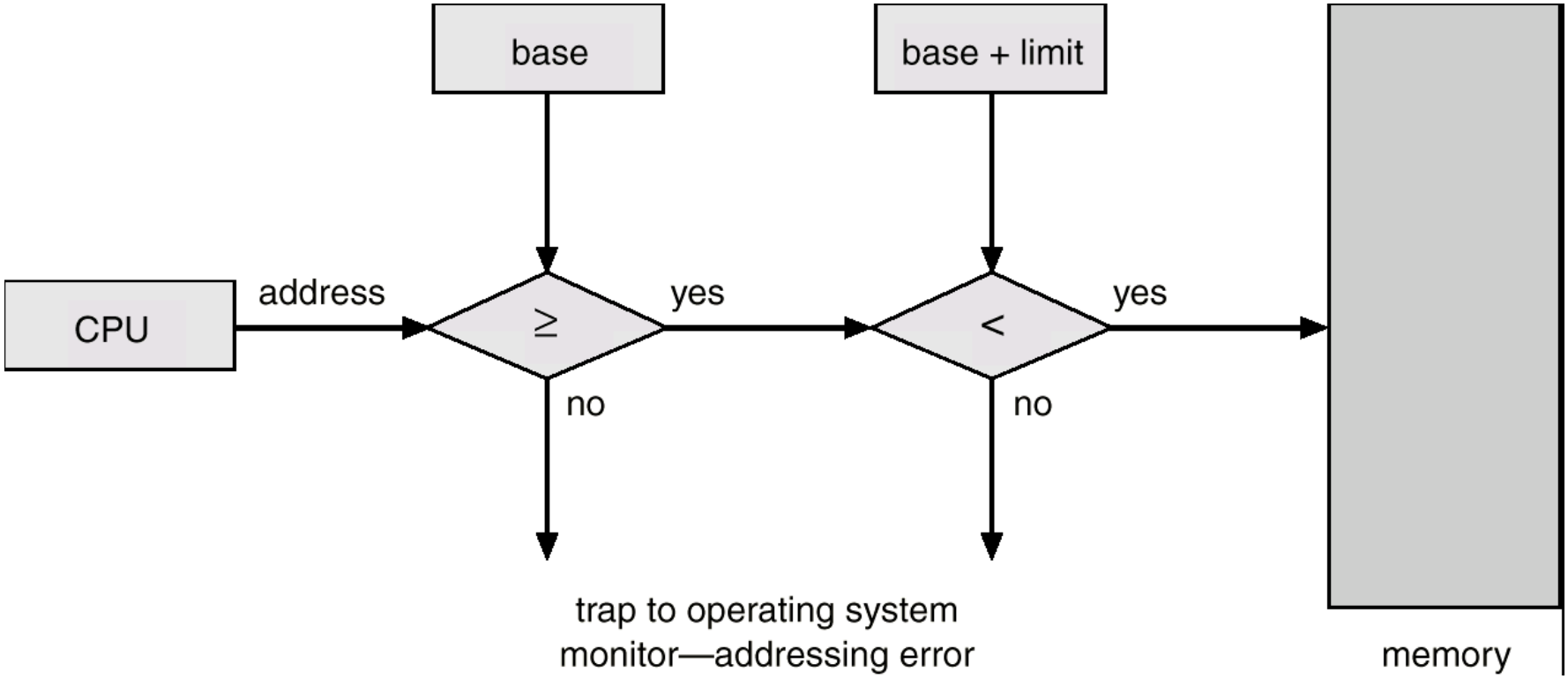


Revisit the Simple Memory Management: Base + Limit Registers

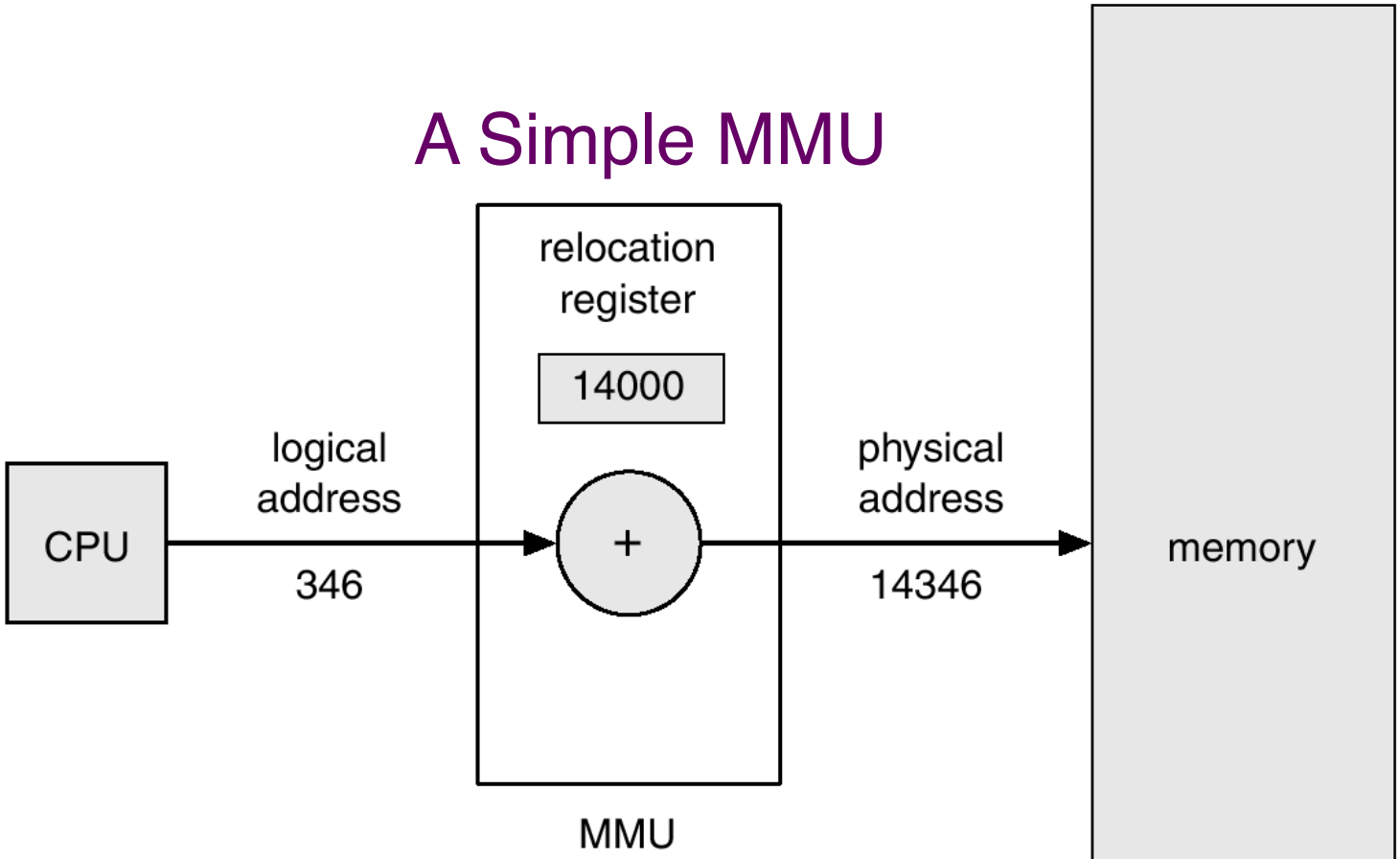
- A pair of **base** and **limit** registers define the logical address space



Hardware Address Protection

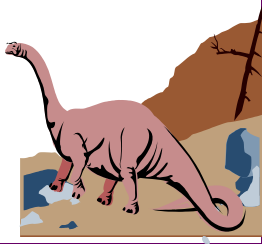


Applications only know logical addresses, so use a relocation register instead of a base register

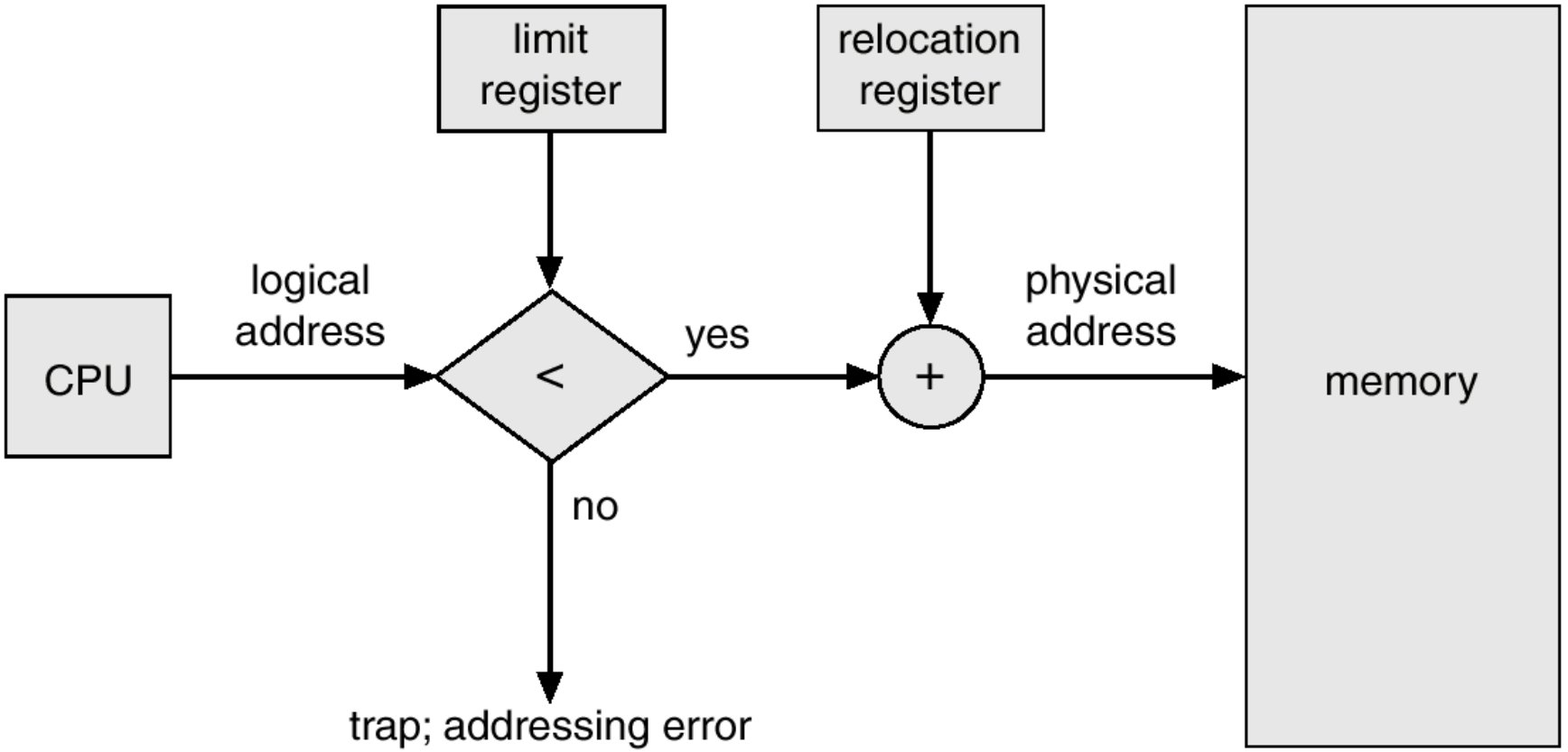


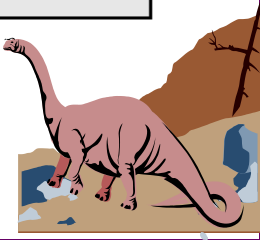
Memory Protection

- 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.



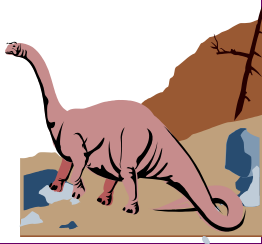
Hardware Support for Relocation and Limit Registers





Chapter 9: Memory Management

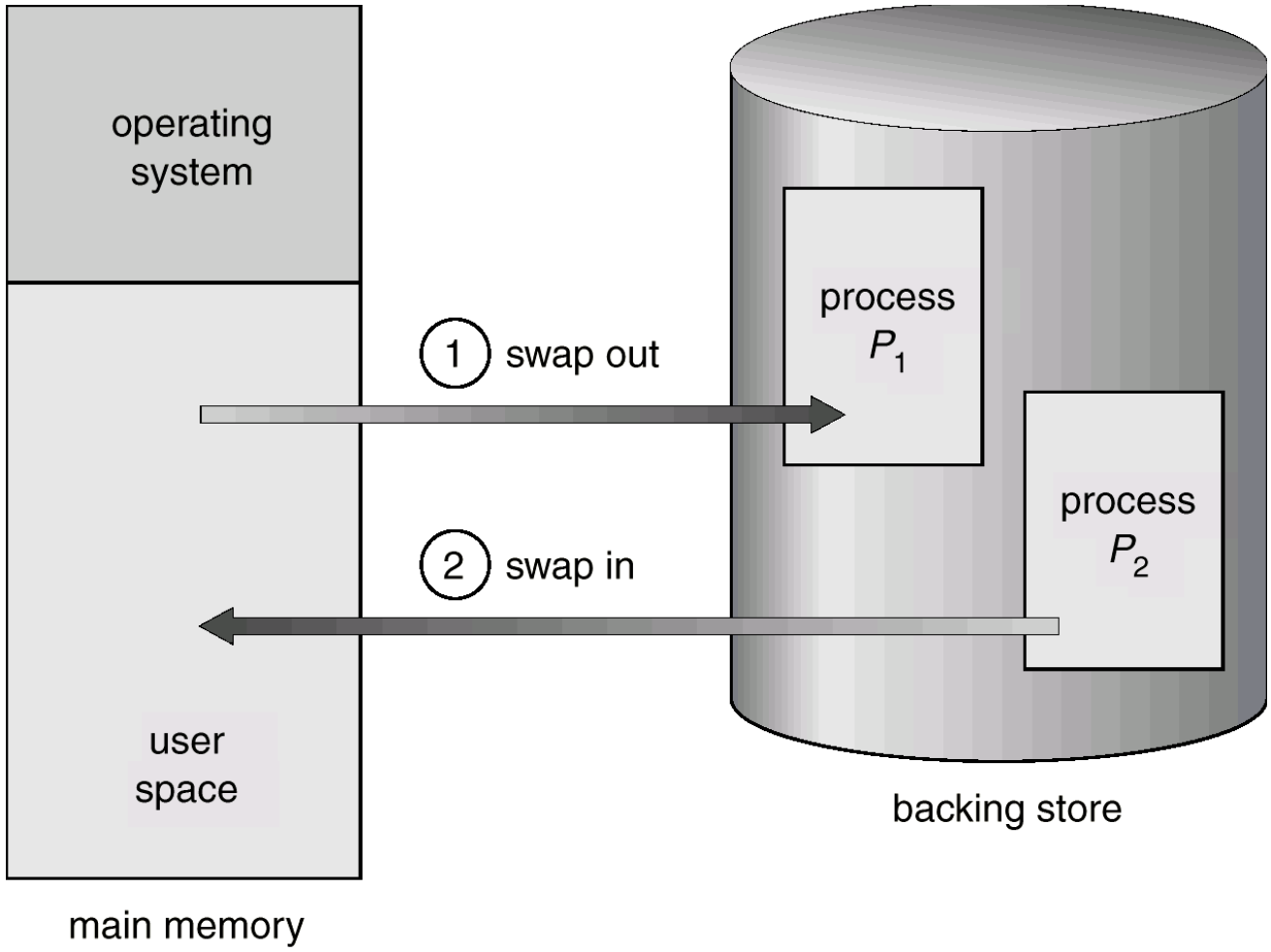
- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging



Swapping

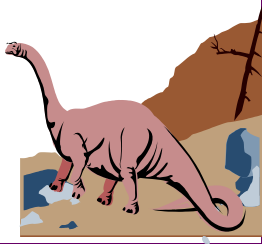
- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to hold copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in*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



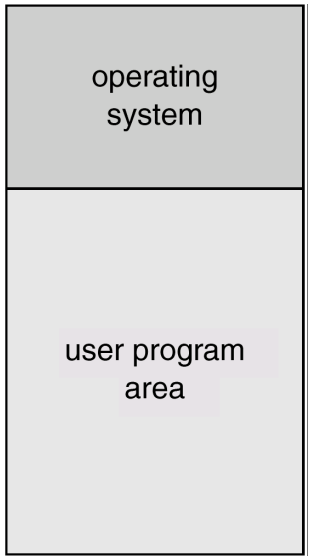
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging

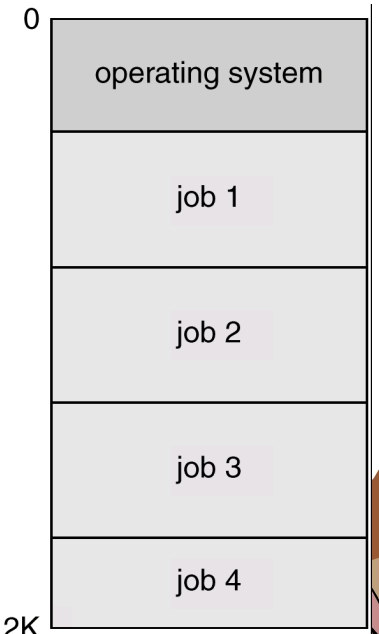


Contiguous Allocation

- Monoprogramming systems usually have two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.

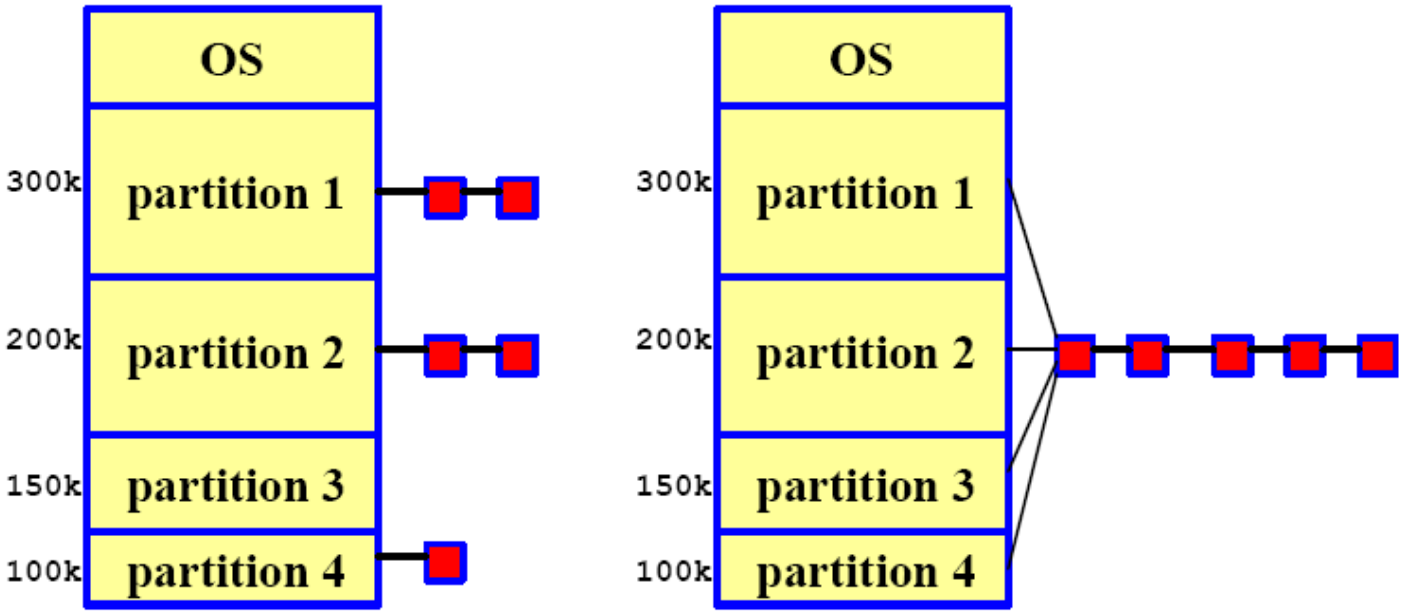


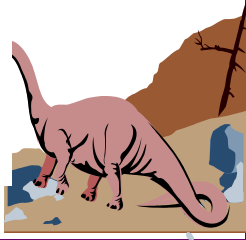
- Multiprogramming Systems:
  - Fixed partitions
  - Variable partitions



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.

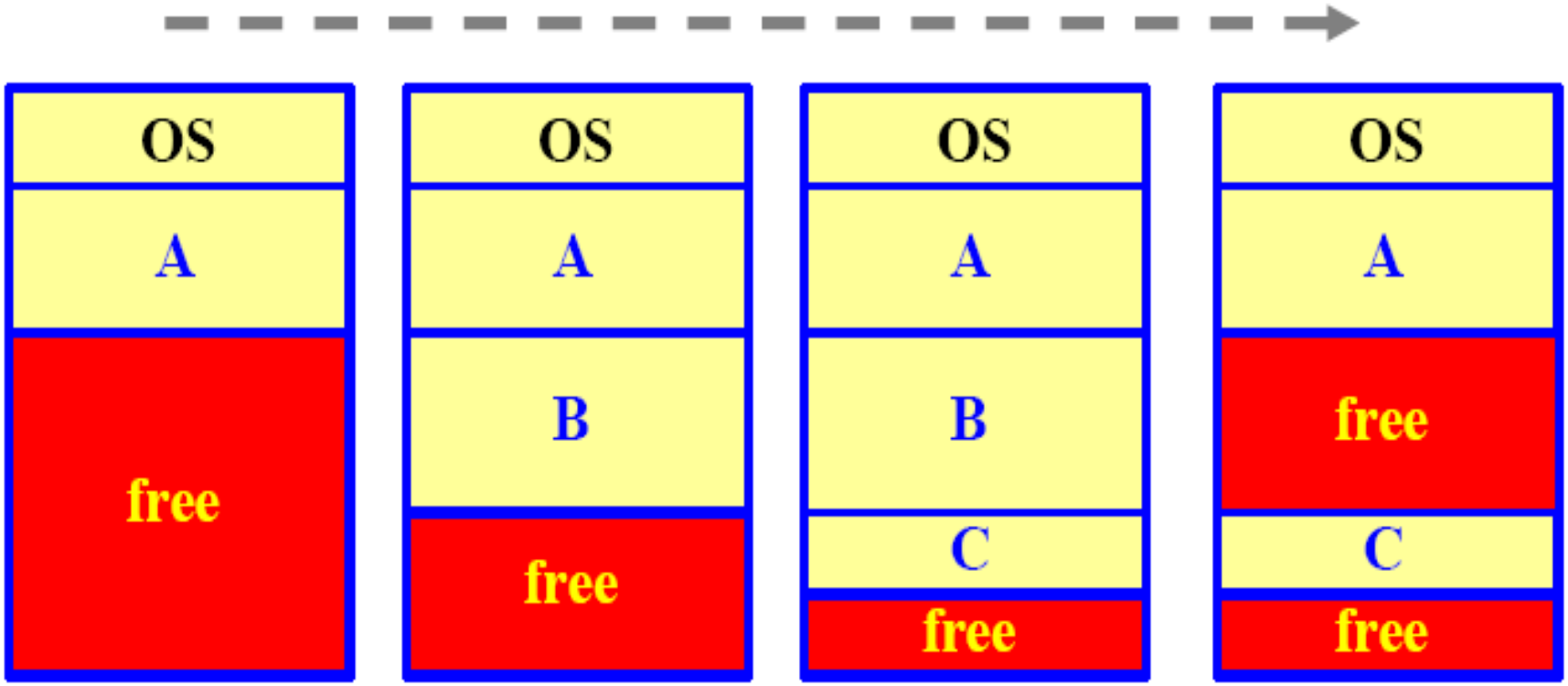


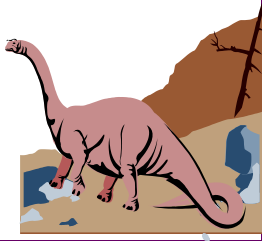


Variable Partitions

- *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.)





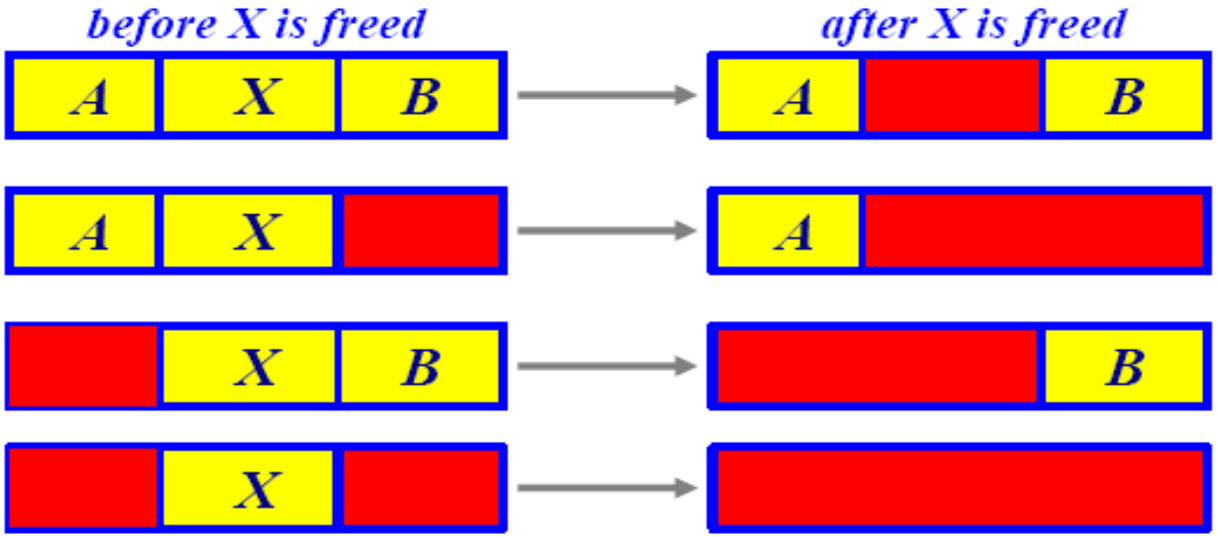
Dynamic Storage-Allocation Problem

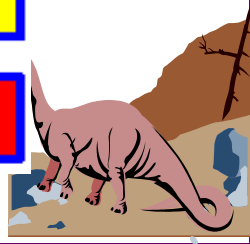
How to satisfy a request of size *n* from a list of free holes.

- **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, unless ordered by size. 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.

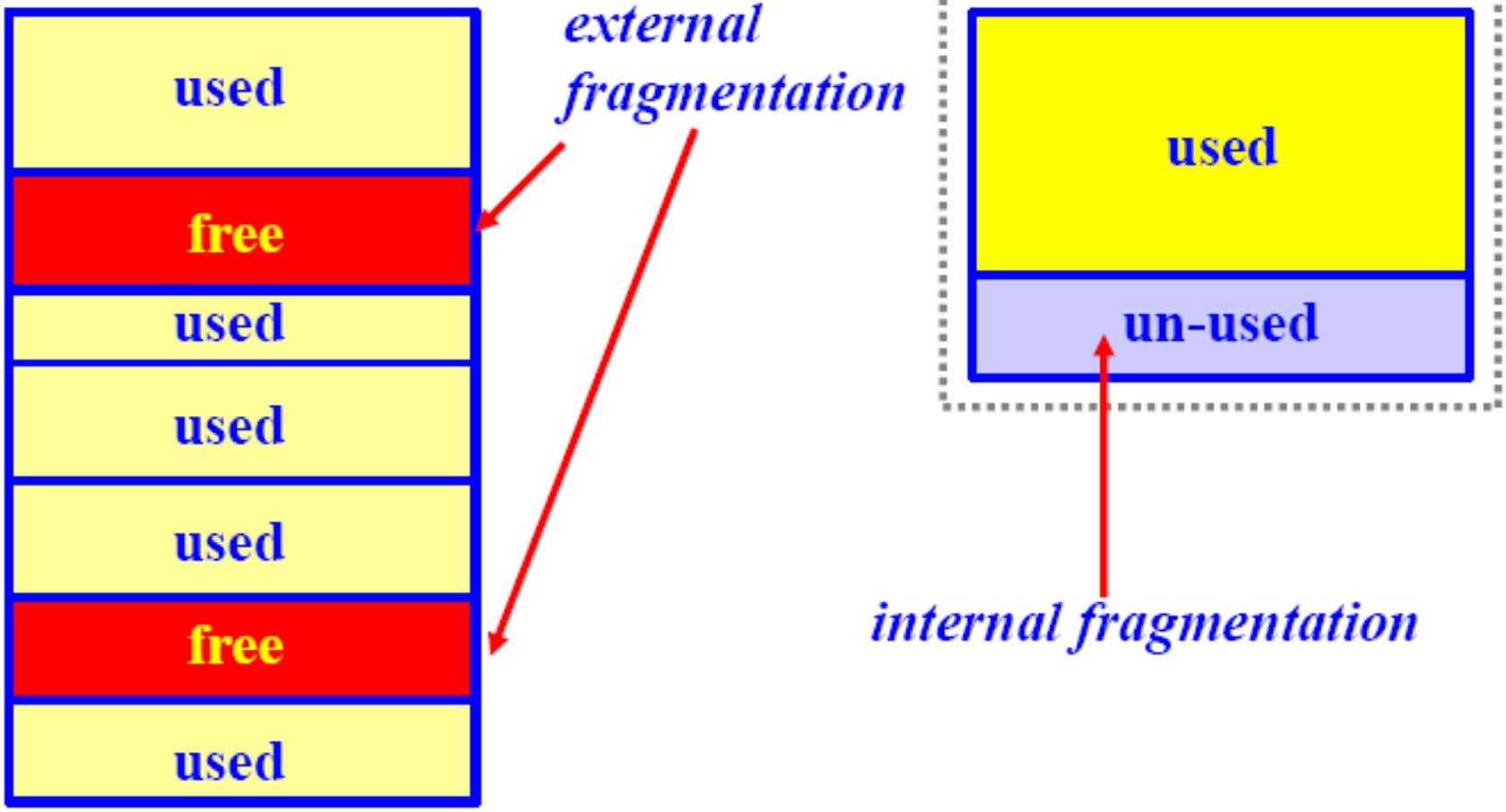




Fragmentation (内存碎片)

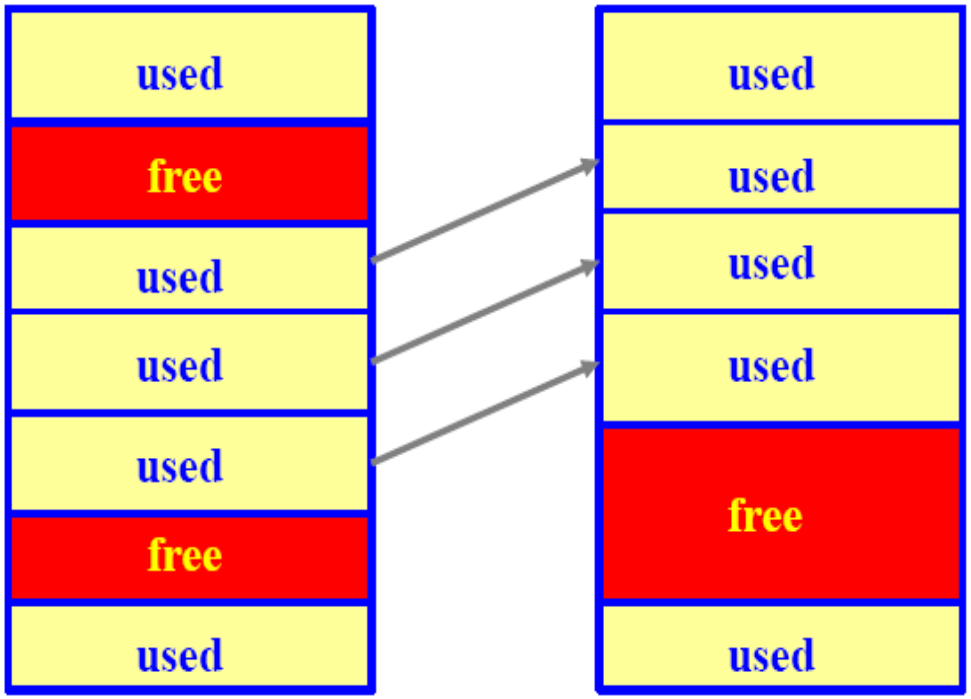
- Processes are loaded and removed from memory, eventually the memory will be cut into small holes that are not large enough to run any incoming process.
- Free memory holes between allocated ones are called *external fragmentation*.
- It is unwise to allocate exactly the requested amount of memory to 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*.

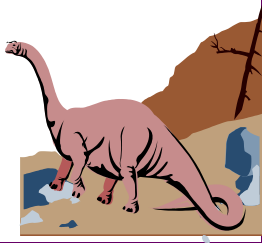
Fragment (Cont.)



Compaction for External Fragmentation

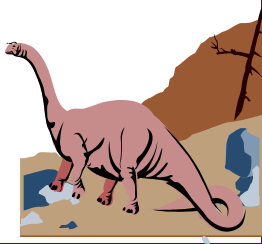
- Shuffle memory contents to place all free memory together in one large block.
- Compaction is possible *only* if program relocation is dynamic and is done at execution time.
- Compaction scheme can be expensive





Chapter 9: Memory Management

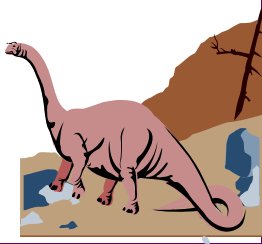
- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging



Segmentation

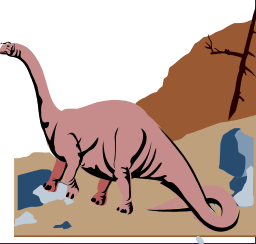
- Memory-management scheme that supports **user view of memory**.
- A program is a collection of segments. A segment is a logical unit such as:

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

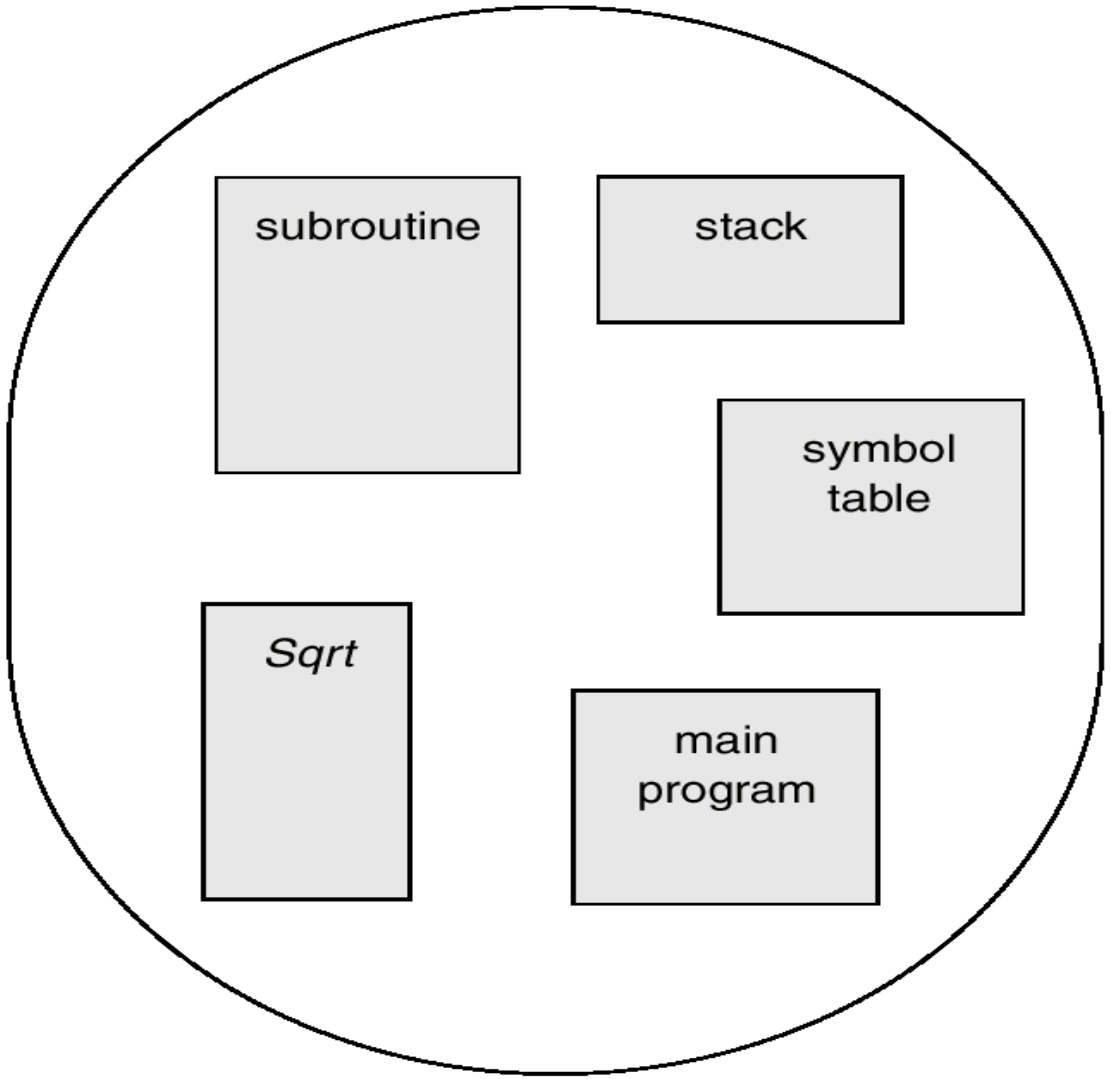


A Previously Used Example

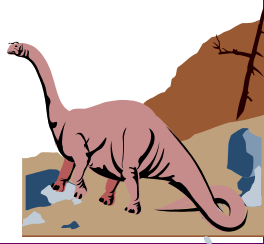
```
//main.cpp
int a = 0;          ←—— 数据段，全局变量
char *p1;           ←—— 数据段，全局变量
main()
{   int b;          ←—— 栈段，局部变量
    char s[] = "abc";   ←—— 栈段，局部变量
    char *p2;       ←—— 栈段，局部变量
    char *p3 = "123456";   ←—— 栈段，局部变量
    p1 = (char *)malloc(10);   ←—— 堆段
    p2 = (char *)malloc(20);   ←—— 堆段
}
```



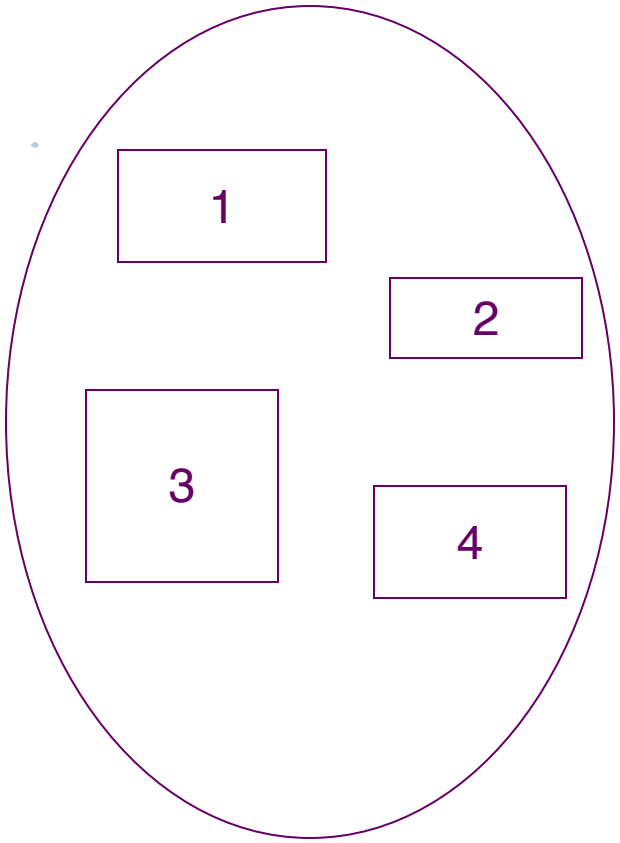
User's View of a Program



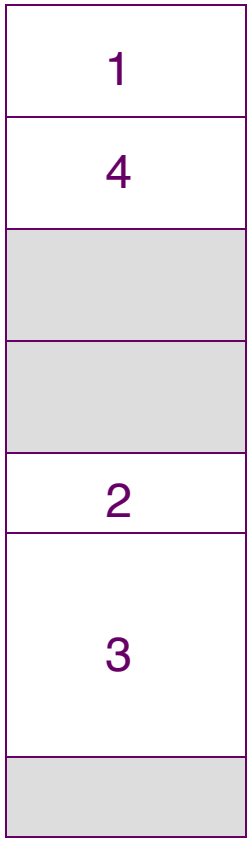
logical address space



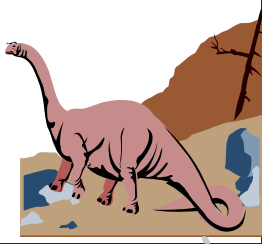
Logical View of Segmentation



user space



physical memory space

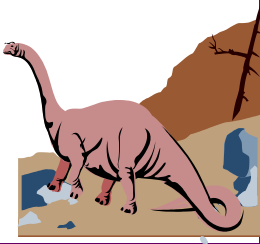


Segmentation Architecture

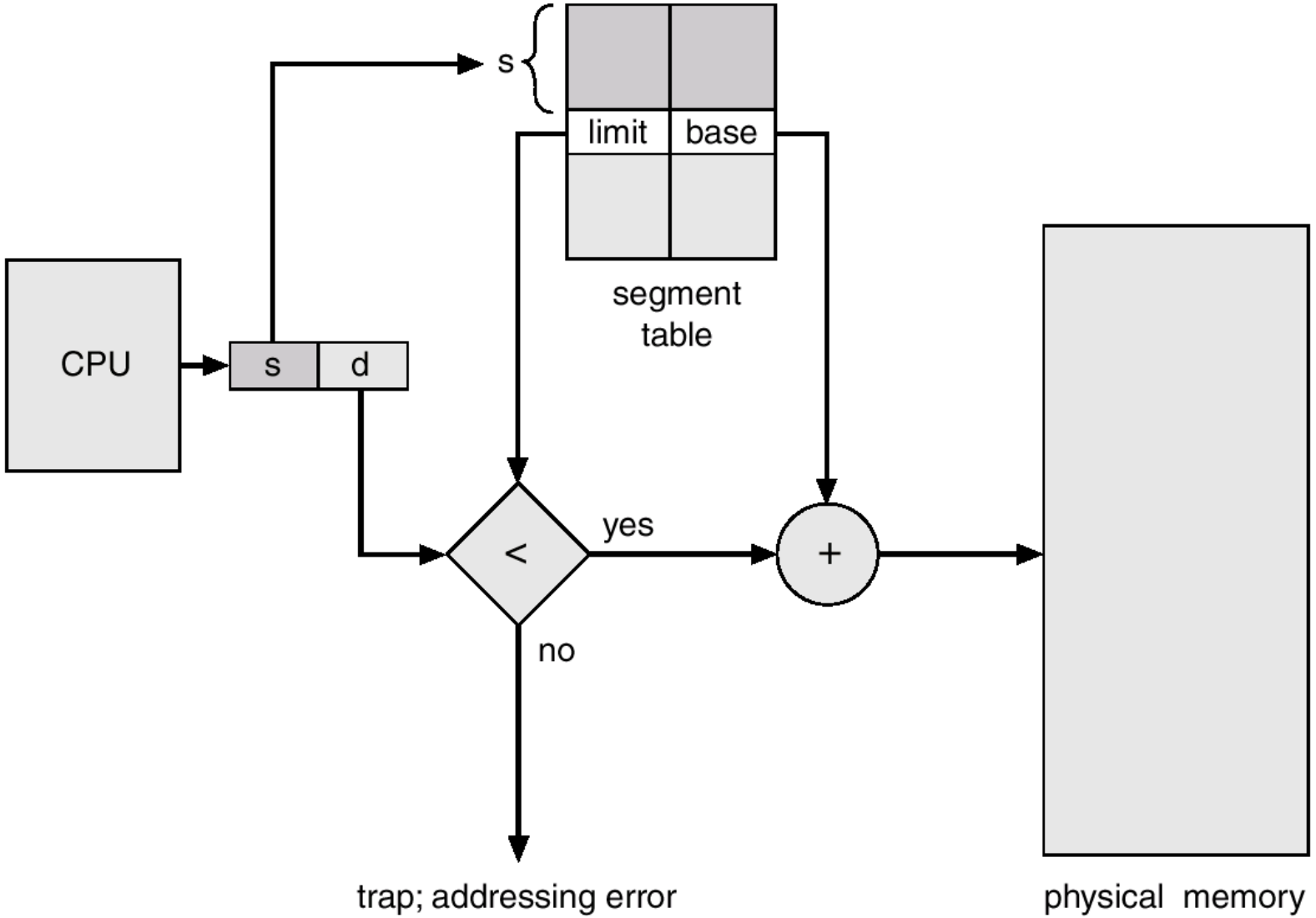
- Logical address consists of a two tuple:

  <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

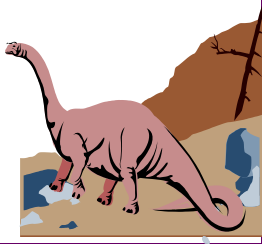


Segmentation Architecture (Cont.)

- *Segment-table base register (STBR)* points to the segment table's location in memory.

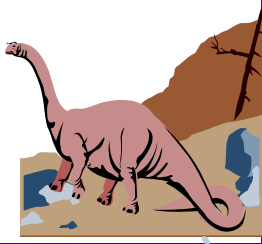
- *Segment-table length register (STLR)* indicates the number of segments used by a program;

  segment number s is legal if $s < \text{STLR}$.

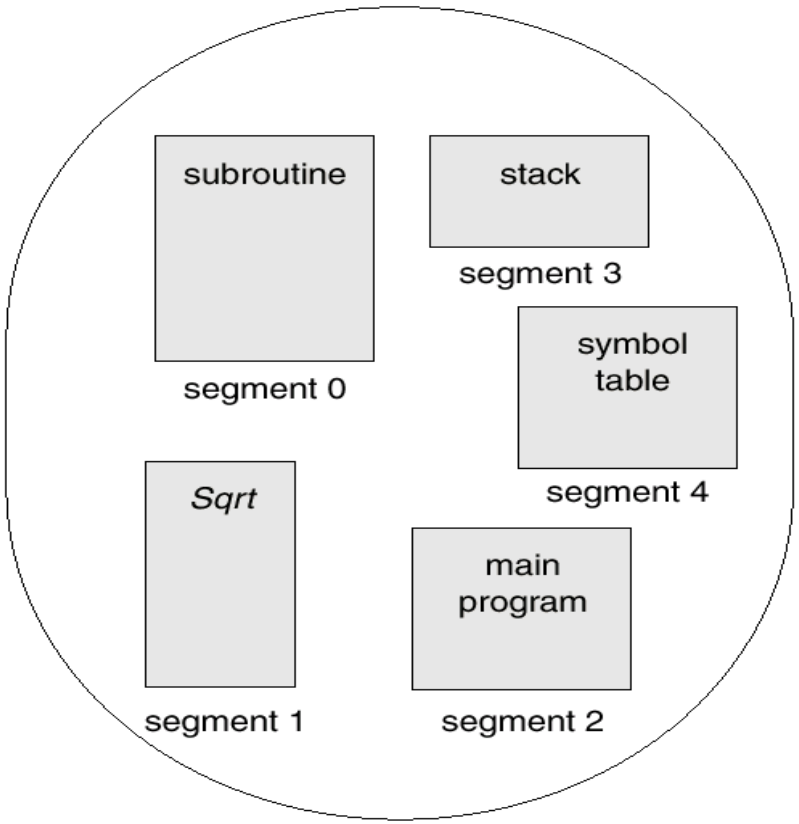


Segmentation Architecture (Cont.)

- 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

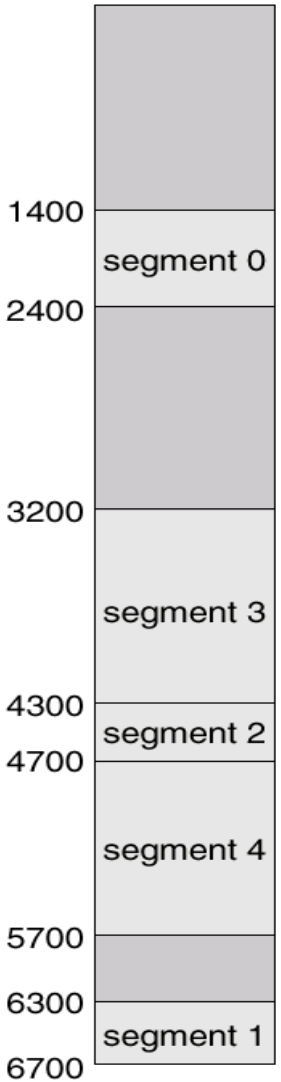


Example of Segmentation



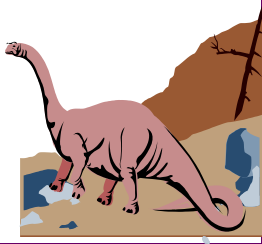
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging

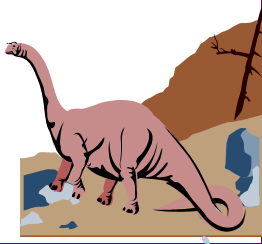


Paging

- Contiguous memory allocation method suffers from the external fragmentation problem
- Paging method allows logical address space of a process to be noncontiguous; a process is allocated physical memory whenever the latter is available
- How?
  - 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)**.

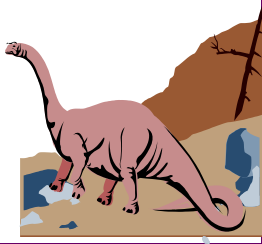
Paging (Cont.)

- 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.



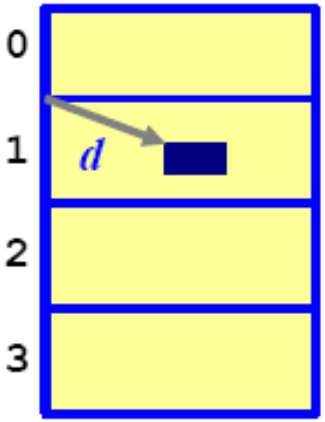
Address Translation Scheme

- 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.

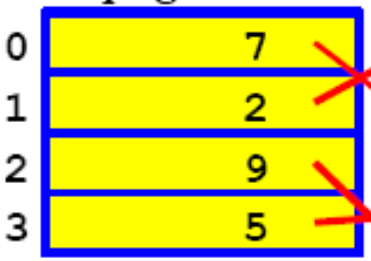


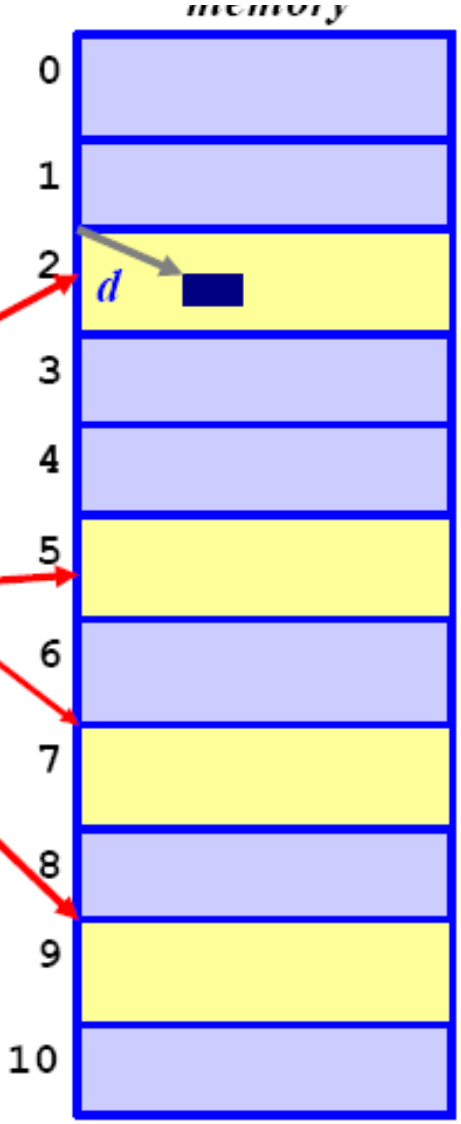
page # offset within the page

logical memory



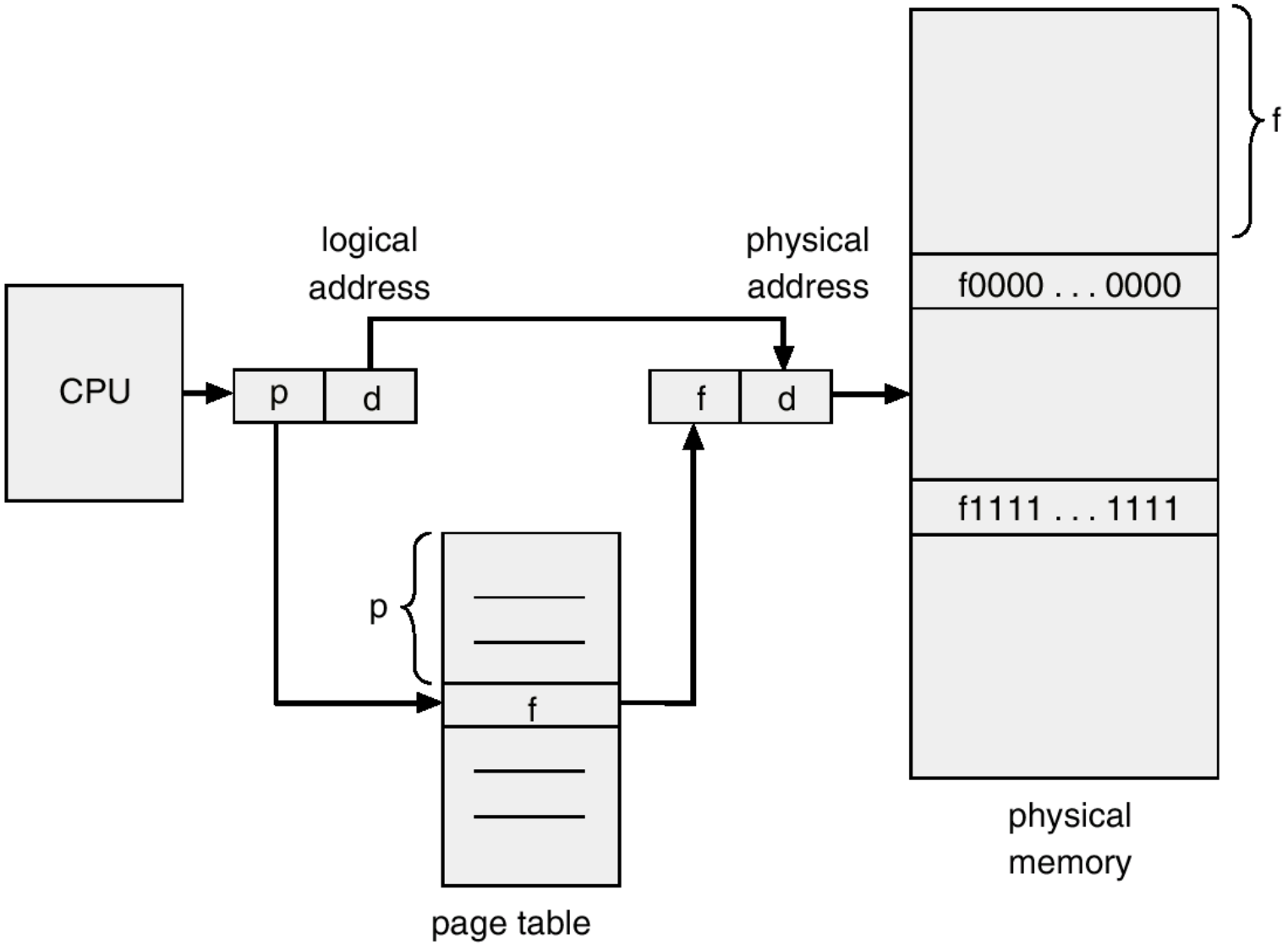
page table



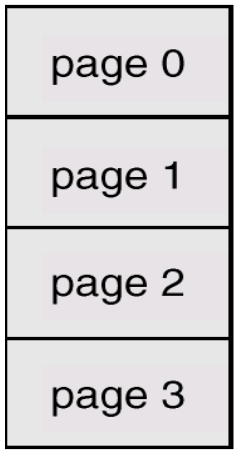


logical address <1, d> translates to physical address <2,d>

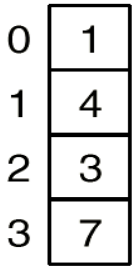
Address Translation Architecture



Paging Example

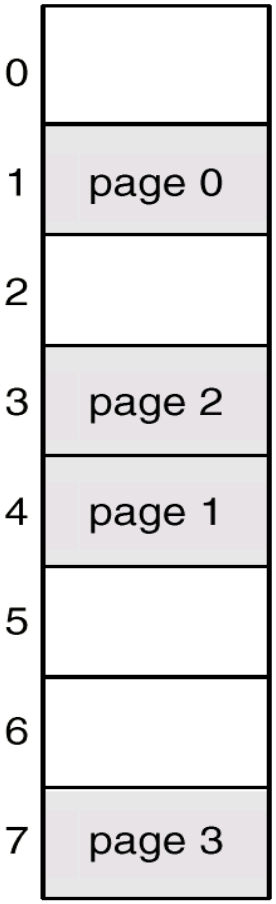


logical memory

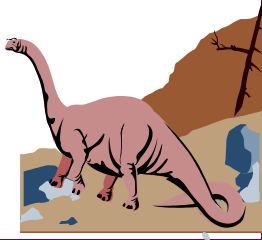


page table

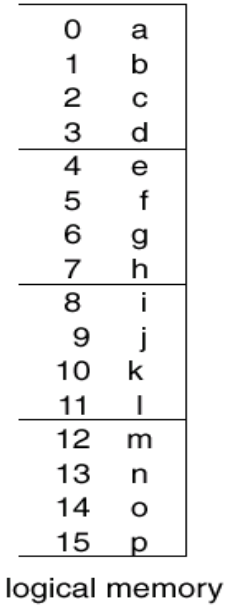
frame number



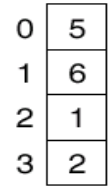
physical memory



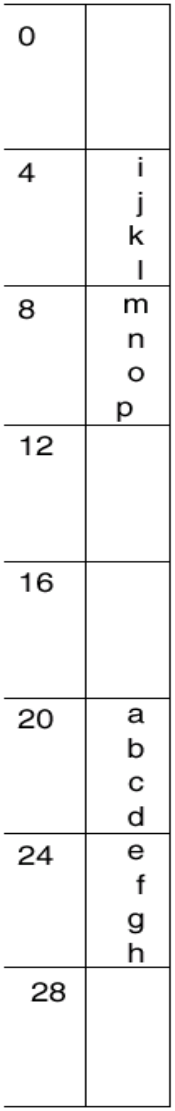
Paging Example



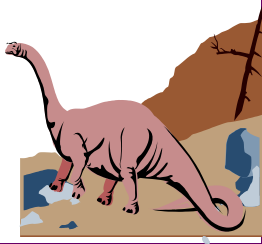
logical memory



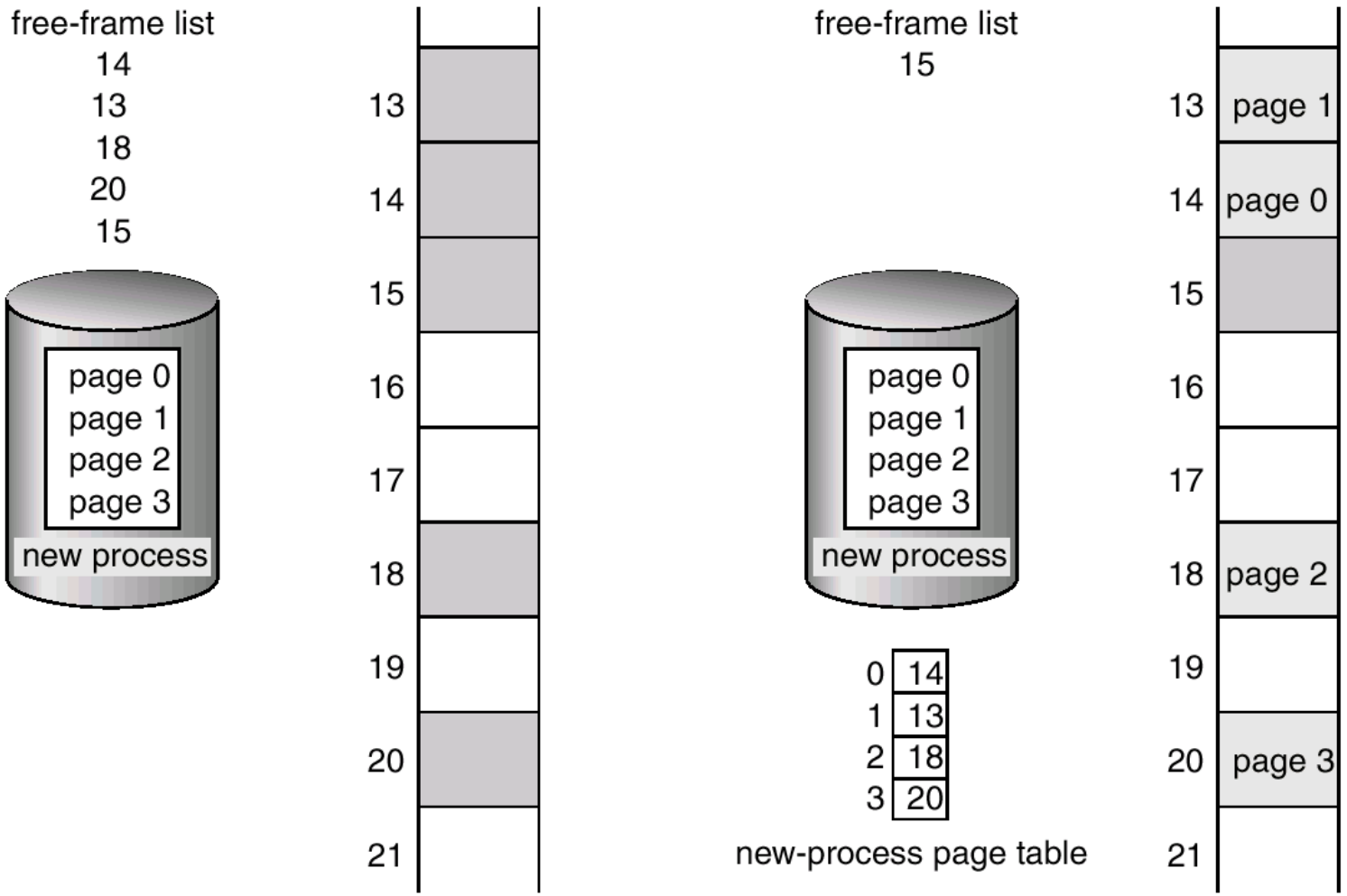
page table



physical memory



Free Frames

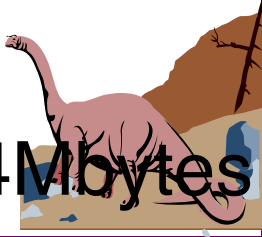


(a) **Before allocation**

(b) **After allocation**

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*4bytes/entry = 4Mbytes

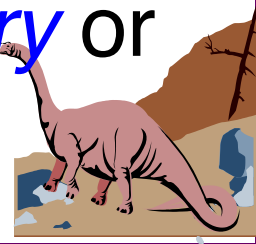


A Quiz

- **Q1**: What will be the size of a page table, if assuming 32 bits virtual address, 8GB physical memory, 8KB page size, and 4KB frame size?
- **Q2**: What if the page size is increased to 2MB?
- **Q3**: What are the pros & cons of larger page size

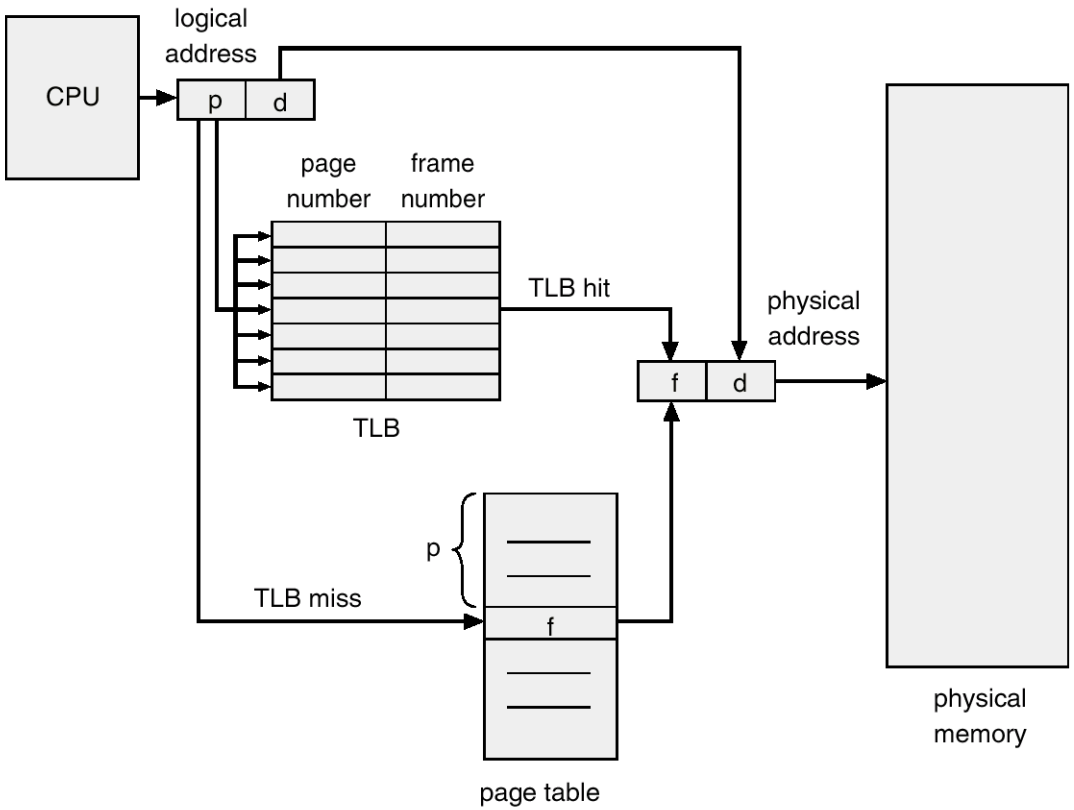
Implementation of Page Table

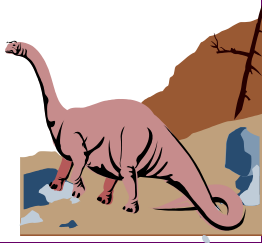
- *Page-table base register (*PTBR) points to the page table existing in main memory.
- In this scheme every data/instruction access requires two memory accesses: One for the page table and one for the data/instruction.
- The two-memory-access problem can be solved by the use of a special fast-lookup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



Paging Hardware With TLB

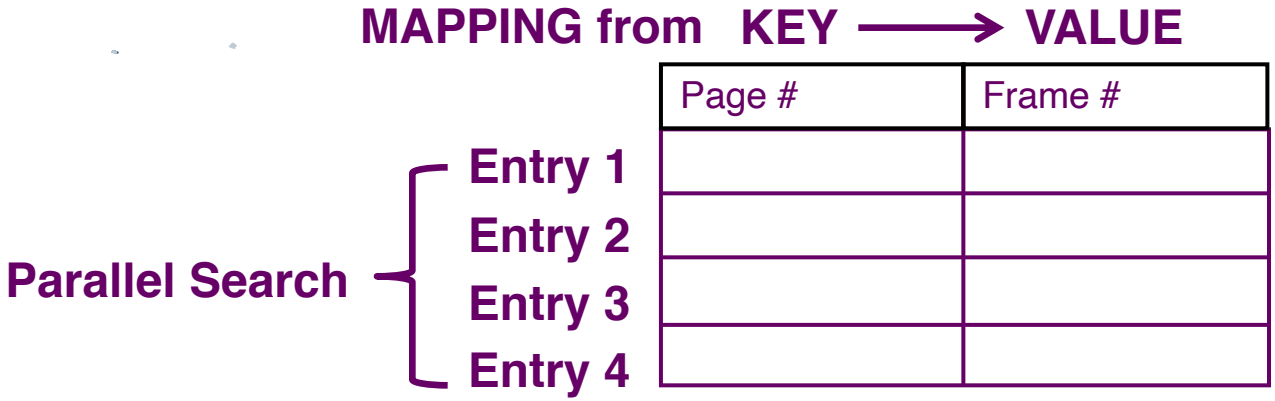
- Translation of virtual address (*p*, *d*)
  - If an entry with the key *p* can be found in the TLB or associative memory, returns the value of frame #
  - Otherwise, get the frame # value from the page table that exists in memory



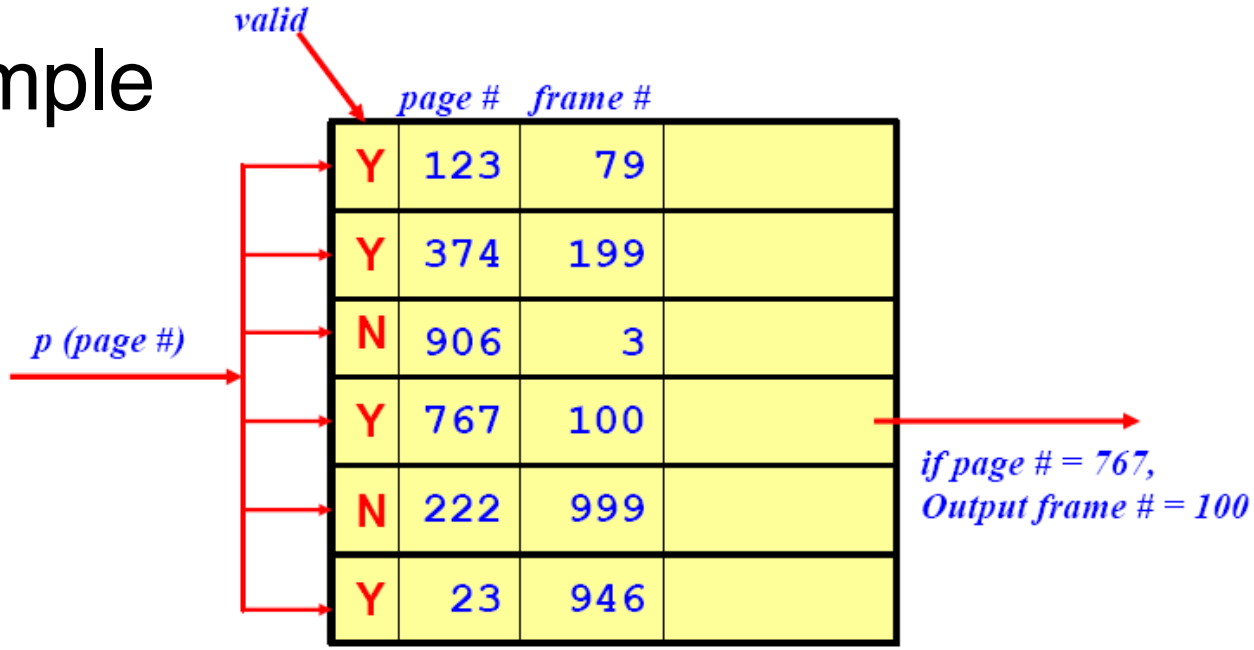


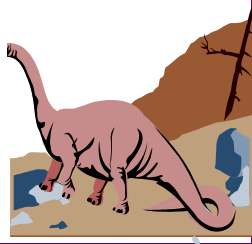
TLB based on Associative Memory

- Associative memory – parallel search



- An Example



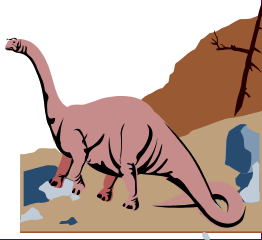


Effective Access Time

- 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)

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

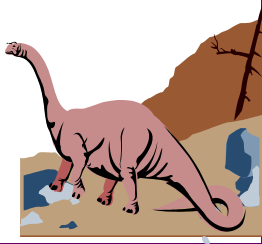
$$= 2 + \varepsilon - \alpha$$



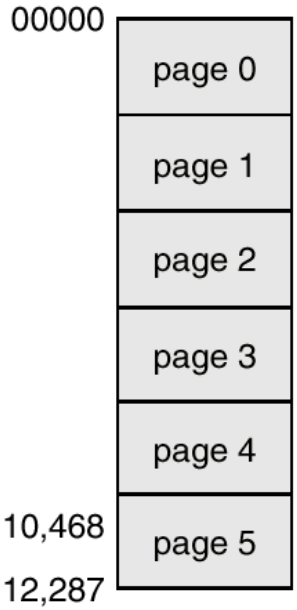
Memory Protection

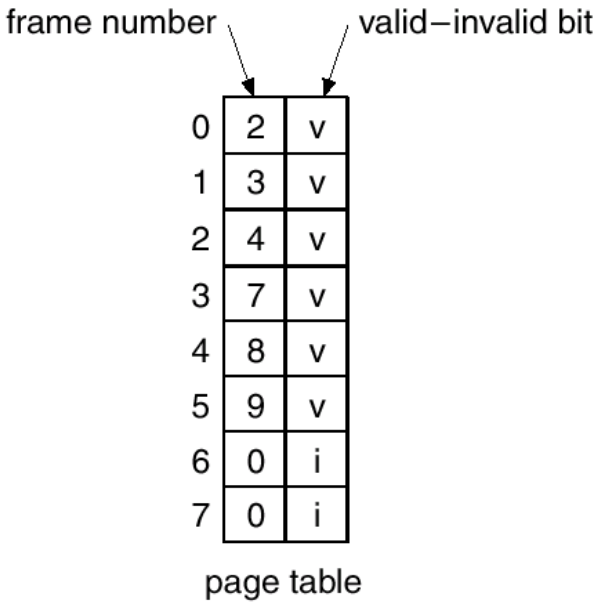
- Memory protection implemented by associating protection bit with each frame.

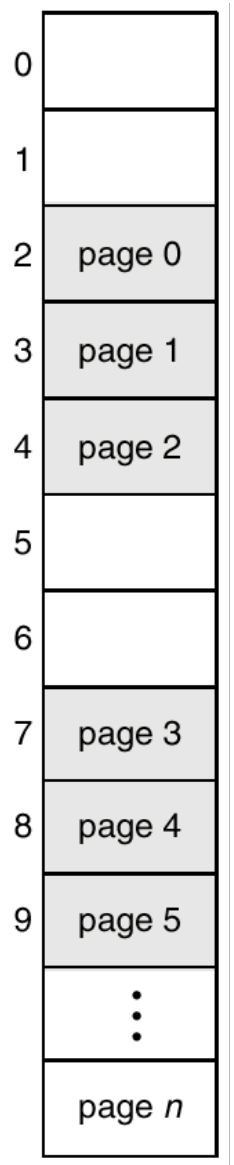
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.



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



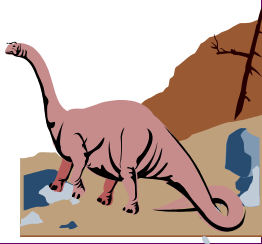




Memory Protection (Cont.)

- 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.

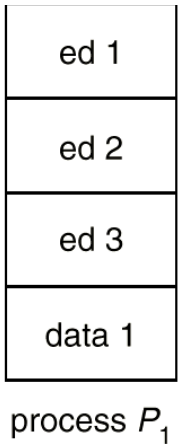
- We can also add read-only, read-write, or execute bits in page table to enforce r-w-e permission.



Advantage of Paging Method: Shared Pages

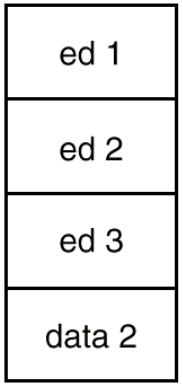
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example



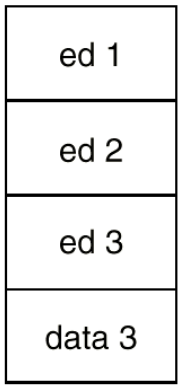
process P_1

page table for P_1



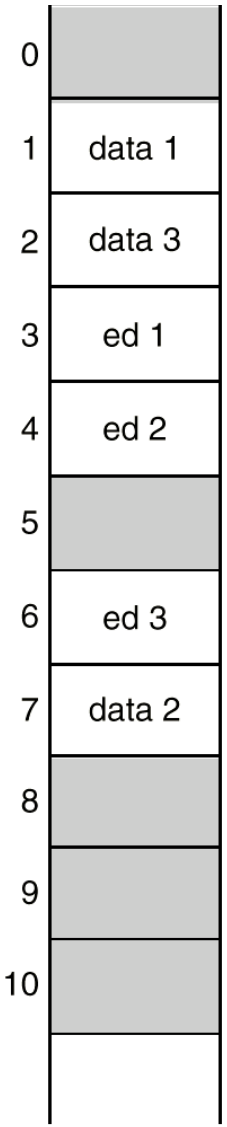
process P_2

page table for P_2



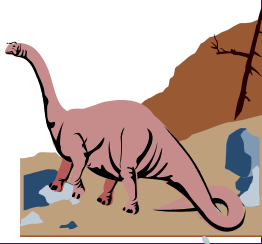
process P_3

page table for P_3



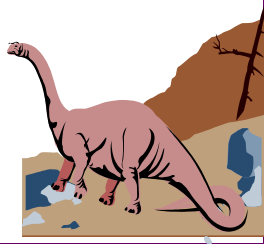
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging



Advanced Page Table Structure

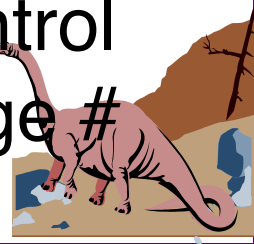
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables



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 #

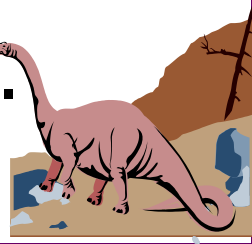
http://en.wikipedia.org/wiki/Page_table#Multilevel_page_table



Hierarchical Page Tables

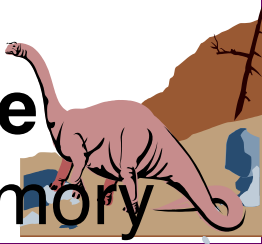
- **20 bits required for physical page number.**
  - ✓4 GB of Physical Memory = **2^{32} bytes**.
  - ✓2^{32} bytes of memory/2^{12} bytes per page = **2^{20} physical pages**
- So each page table entry is approximately **4 bytes**. (20 bits physical page number is roughly 3 bytes and access control 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.

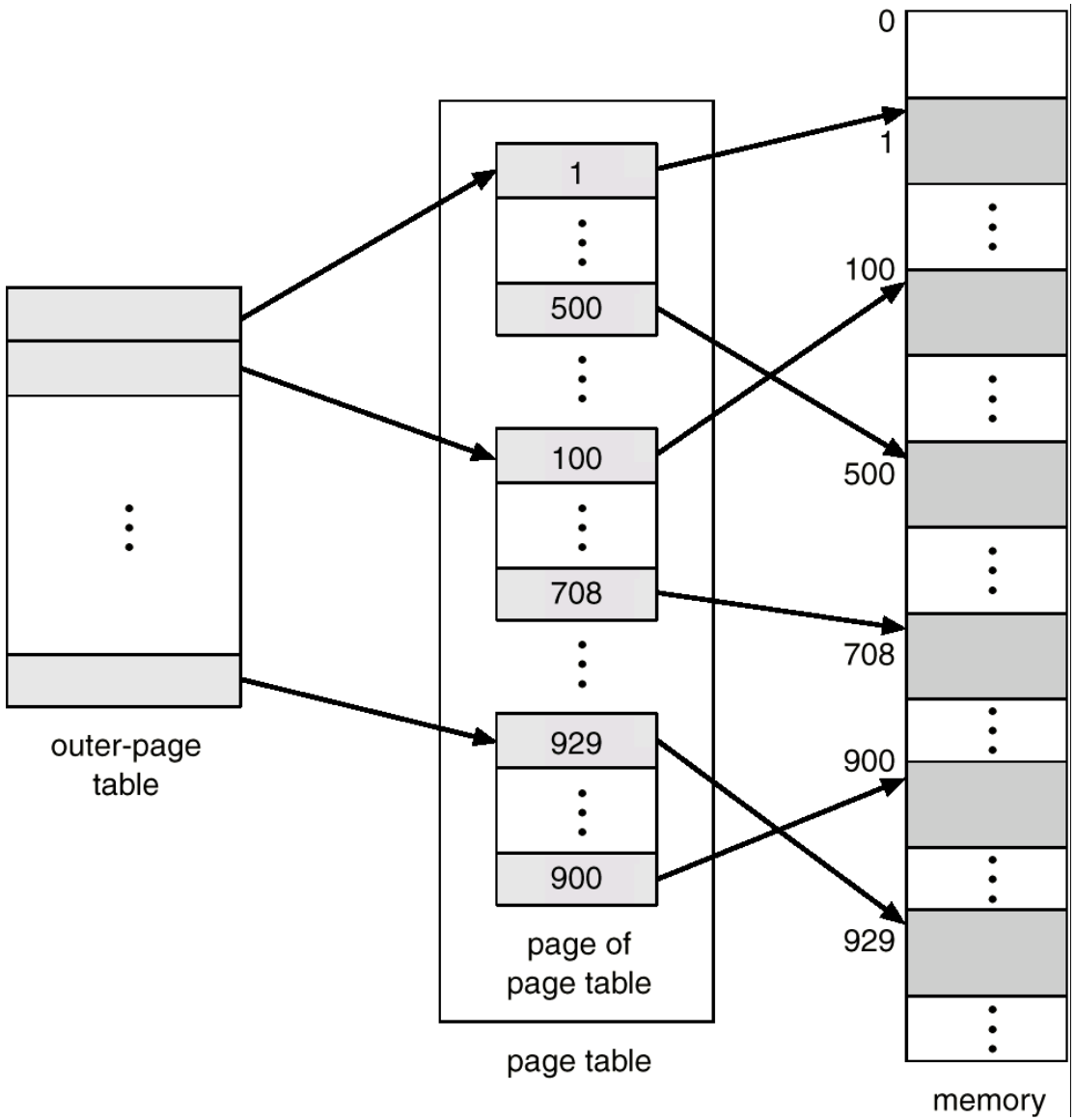


Hierarchical Page Tables

- **A 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
  - The first-level page table contains **2^{52} page entries**
  - If we page the first-level page table, then one page contains 4KB / 4 bytes per entry = 1024=**2^{10} entries**
  - So the first-level page table is divided into **2^{42} pages**
  - So 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

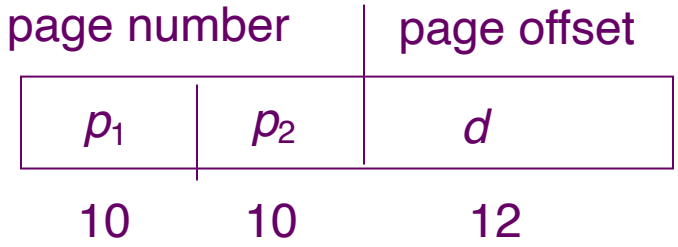


Two-Level Page-Table Scheme

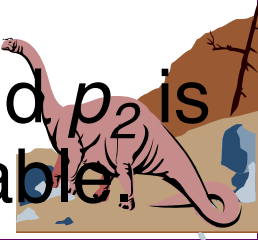


Two-Level Paging Example for 32-bit Operating Systems

- 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:

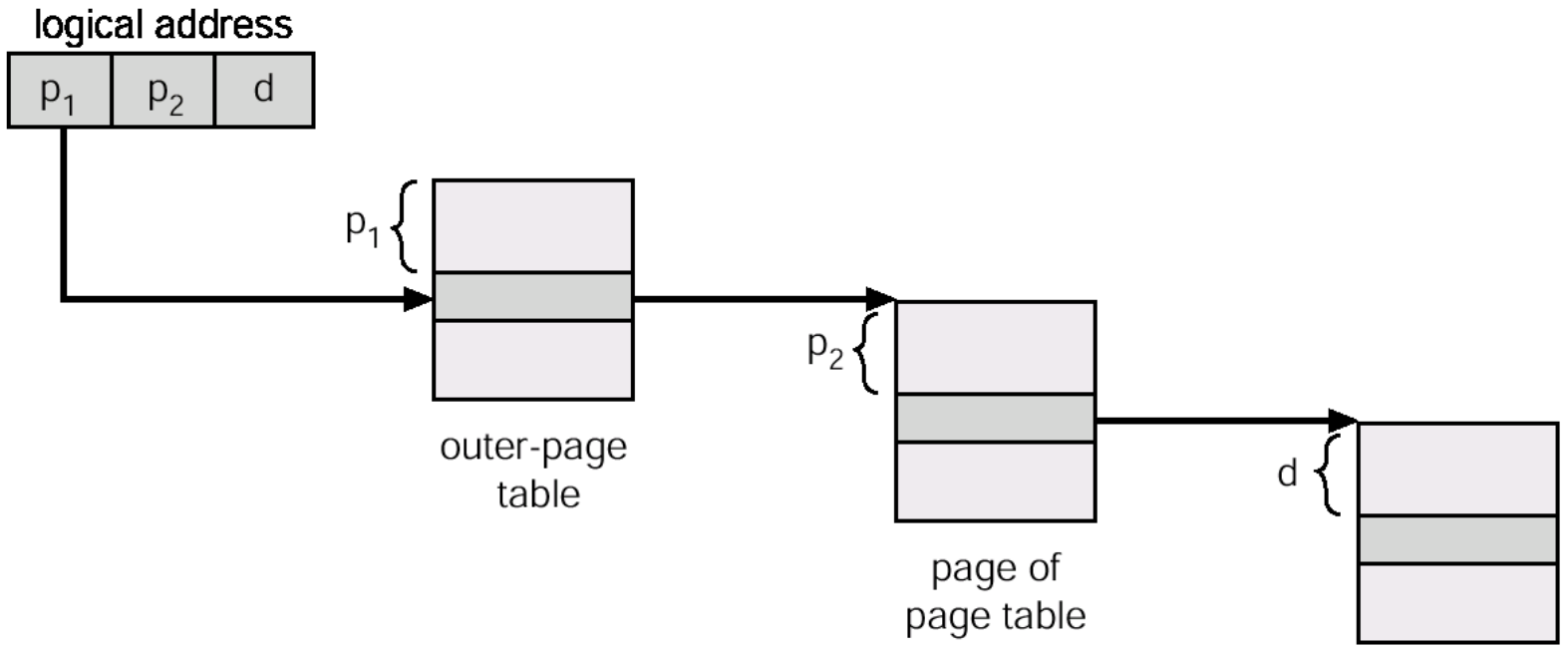


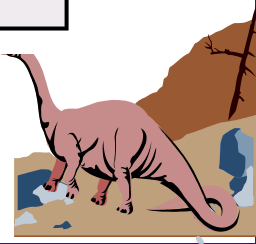
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of outer page table.



Address-Translation Scheme

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





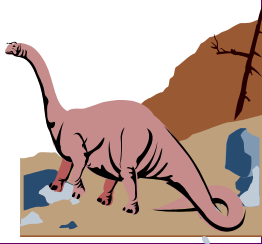
Quiz of Two-level Page Table

- 某计算机采用二级页表的分页存储管理方式，按字节编制，页大小为2^{10}字节，页表项大小为2字节。逻辑地址结构为：页目录号、页号、页内偏移量，逻辑地址空间大小为2^{16}页，则表示整个逻辑地址空间的页目录表中包含表项的个数是（　）

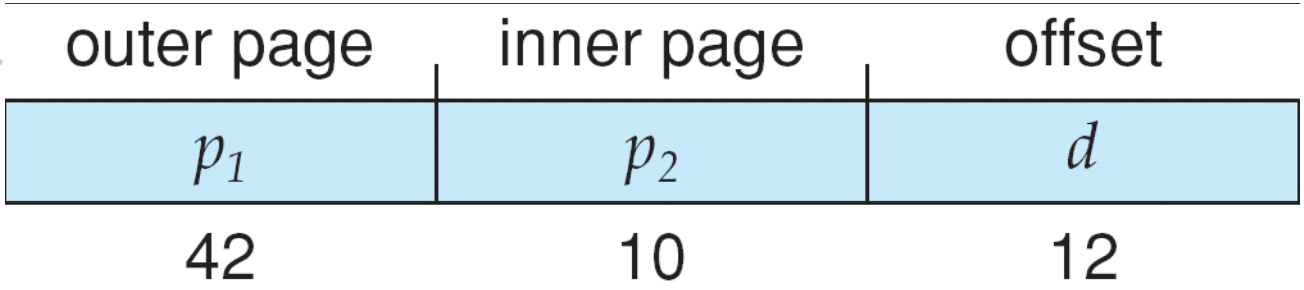
  A、64　　B、128　　C、256　　D、512

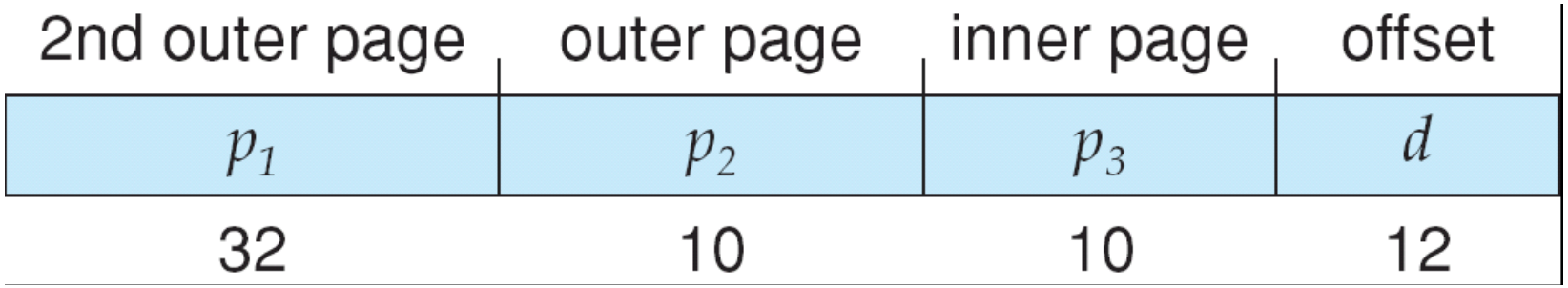
- 答案：B

  2^{16} / (2^{10} bytes/2 bytes) = 2^{7}=128



Three-level Paging Scheme for 64-bit Operating System

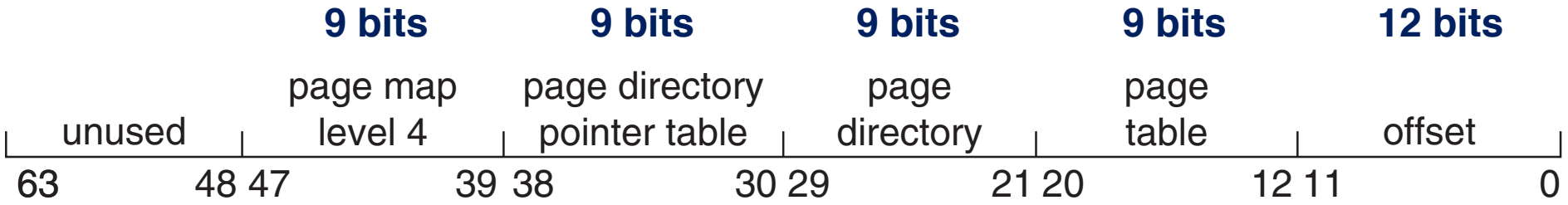




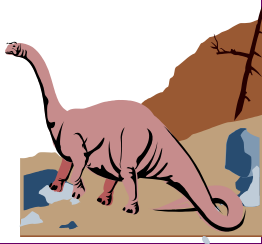
- The 2nd outer page table still needs 16GB memory, since 2^{32} number of table entries $\times$ 4 bytes per table entry = 2^{34} bytes memory

Intel x86-64

- Current generation Intel x86 architecture
- 64 bits address space is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Four levels of paging hierarchy



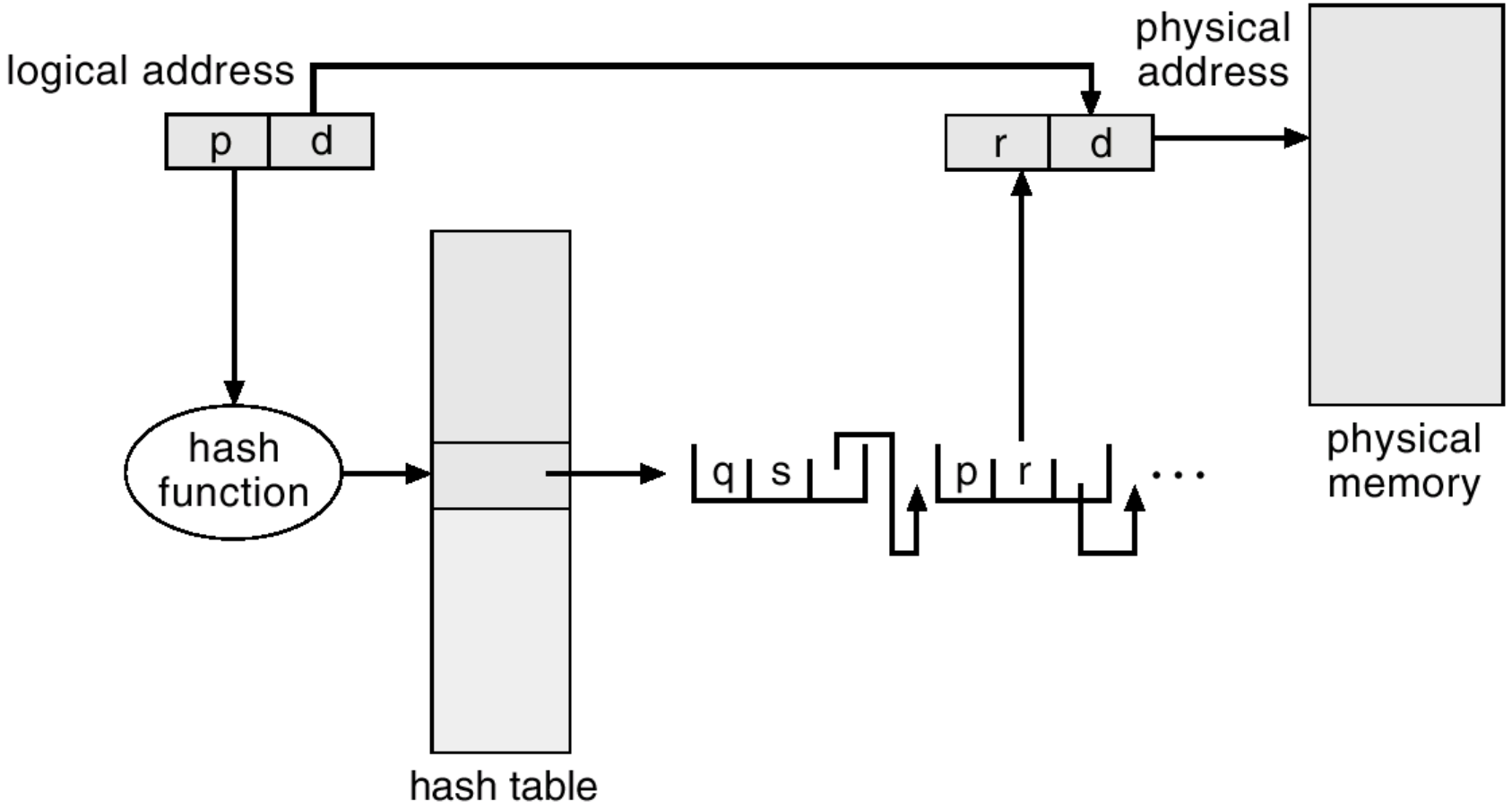
  - Multiple page sizes of 4 KB, 2 MB, 1 GB

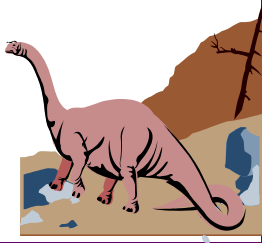


Hashed Page Tables

- 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.

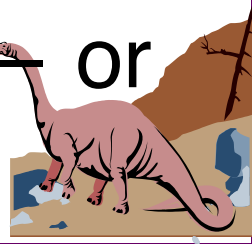
Hashed Page Tables



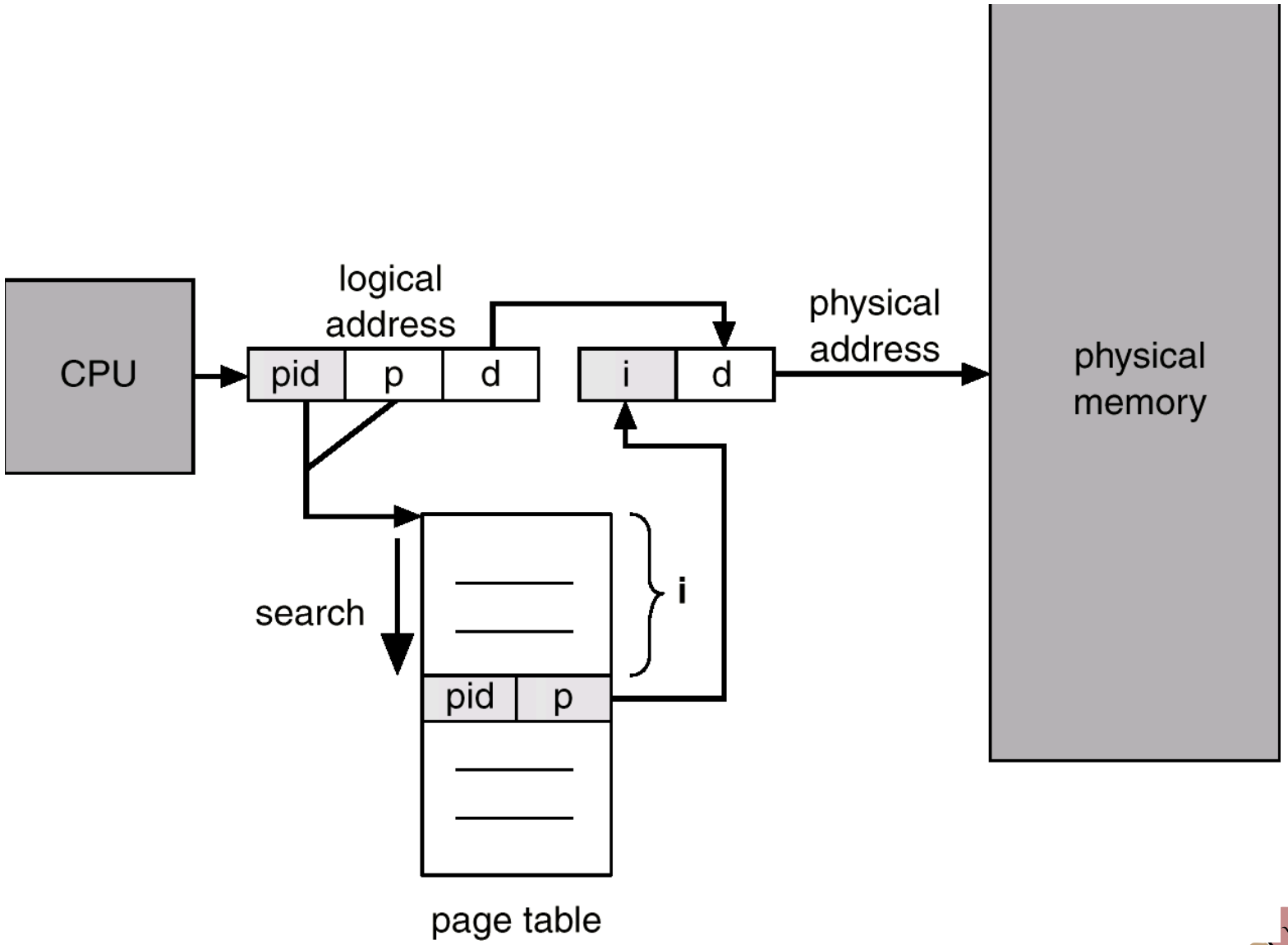


Inverted Page Table

- **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.

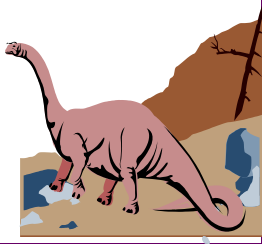


Inverted Page Table Architecture



Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging

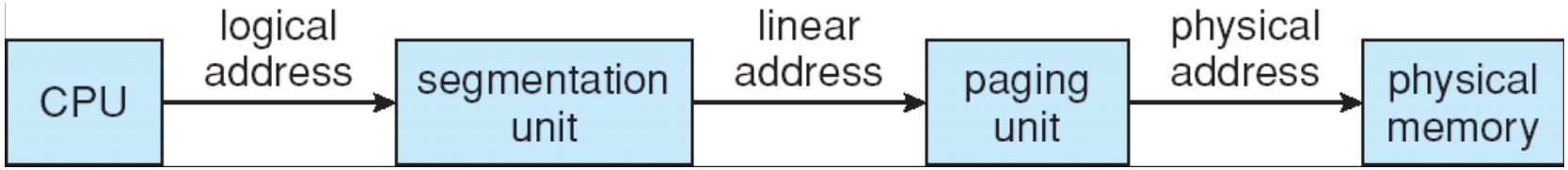


Example: The Intel Pentium

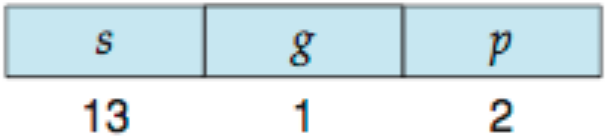
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

The Intel IA-32 Architecture

- Supports segmentation with paging



- CPU generates logical address
  - Selector given to segmentation unit
    - ✓Which produces linear addresses
    - ✓Up to 16K segments per process

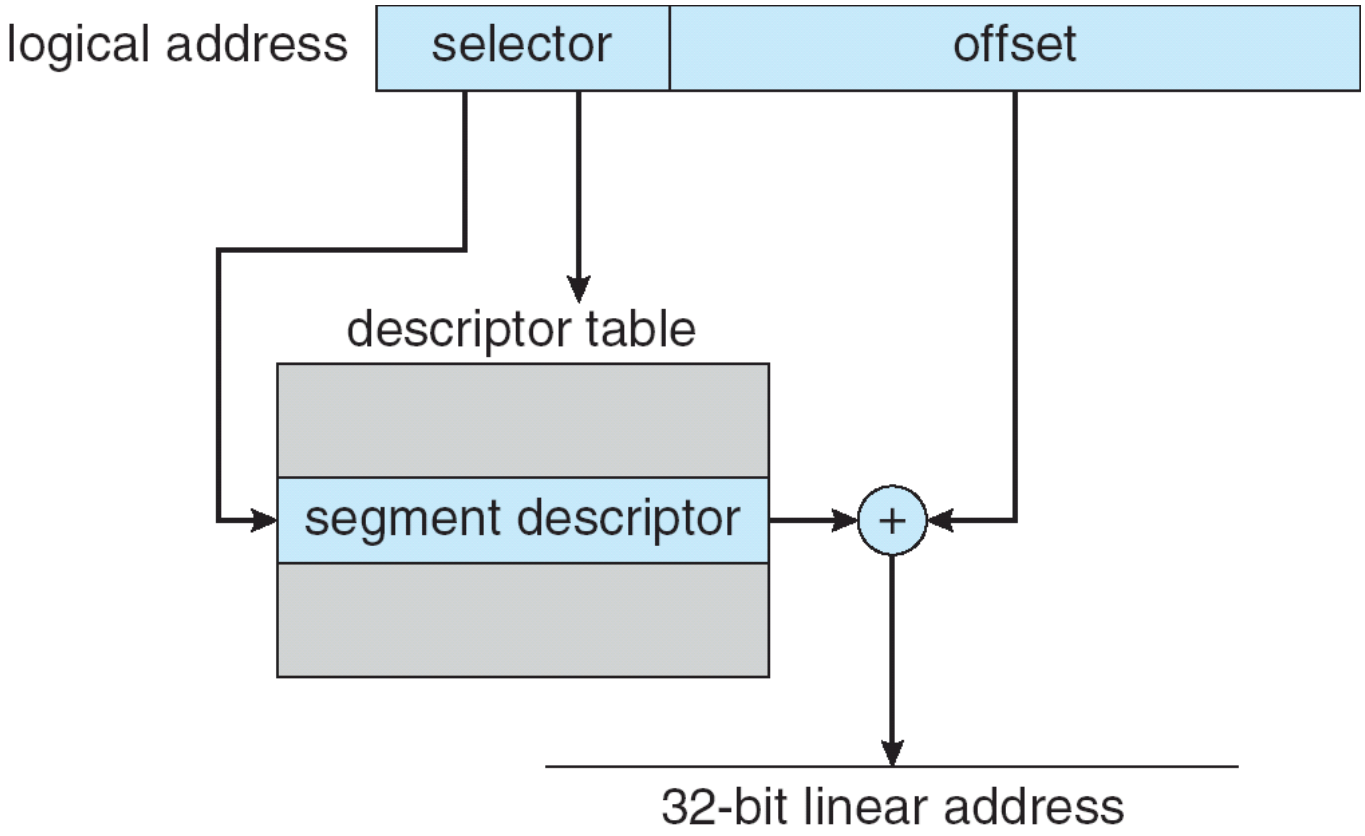


s: segment number
g: whether in GDT or LDT
p: protection bits

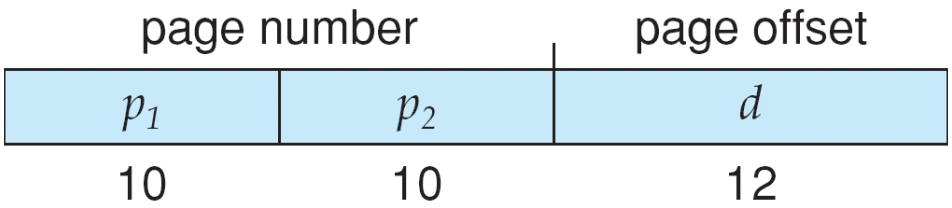
  - Linear address given to paging unit
    - ✓Which generates physical address in main memory
    - ✓Paging units form equivalent of MMU

http://en.wikipedia.org/wiki/Memory_segmentation#Segmentation_with_paging

Intel IA-32 Segmentation with Paging

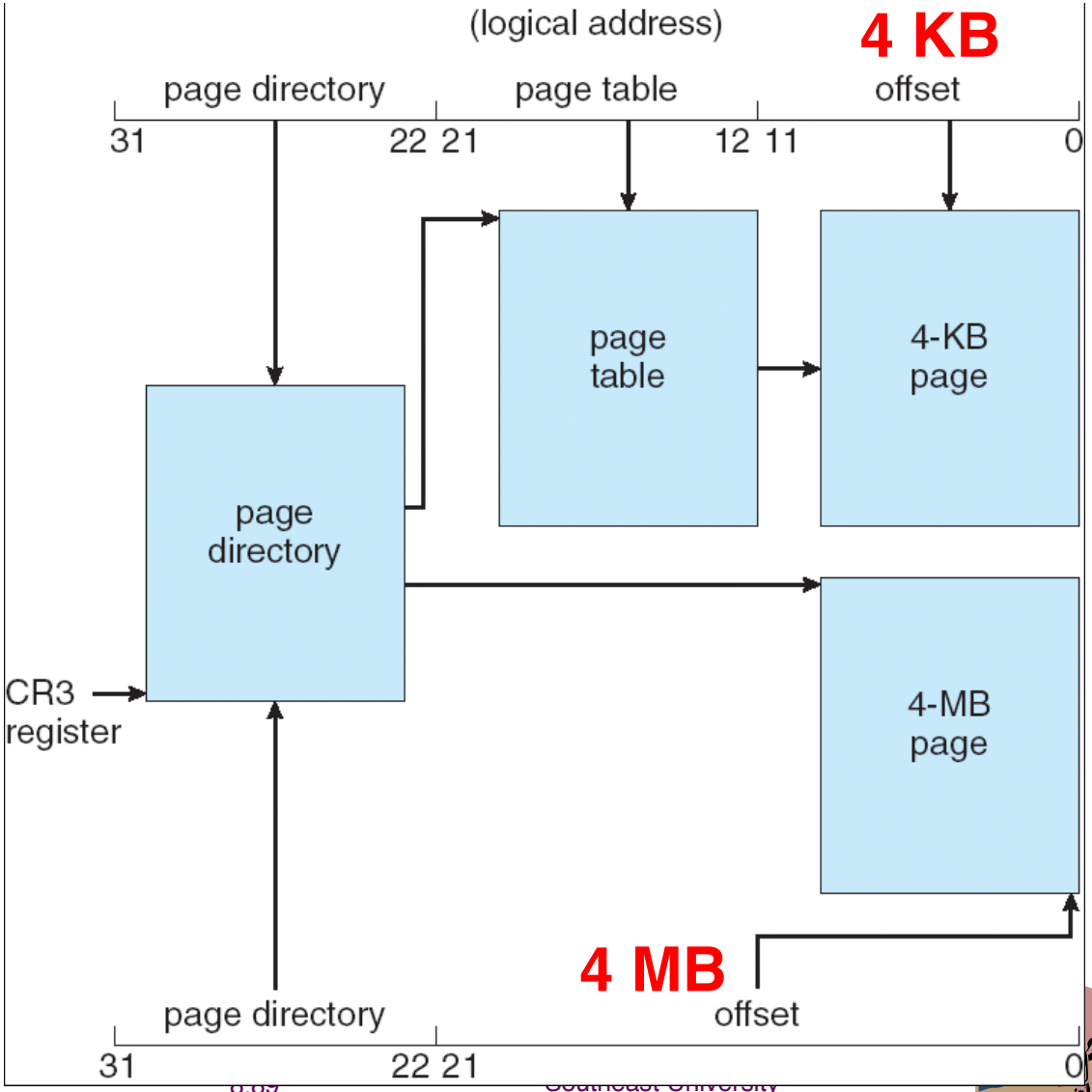


Support two-level page table



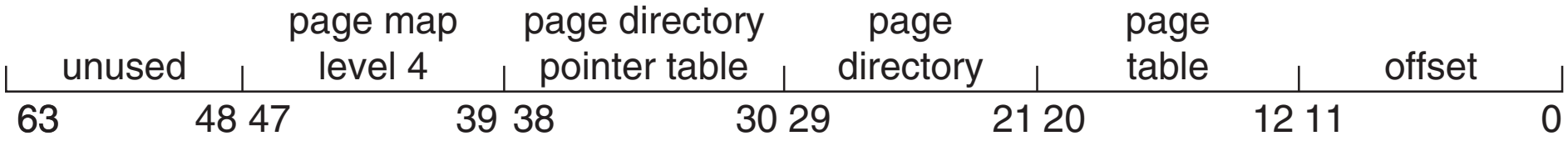
Intel IA-32 Architecture Supports Multiple Page Size

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



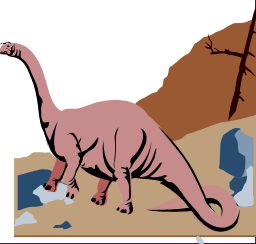
- Can also use PAE (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits

Example: ARM Architecture

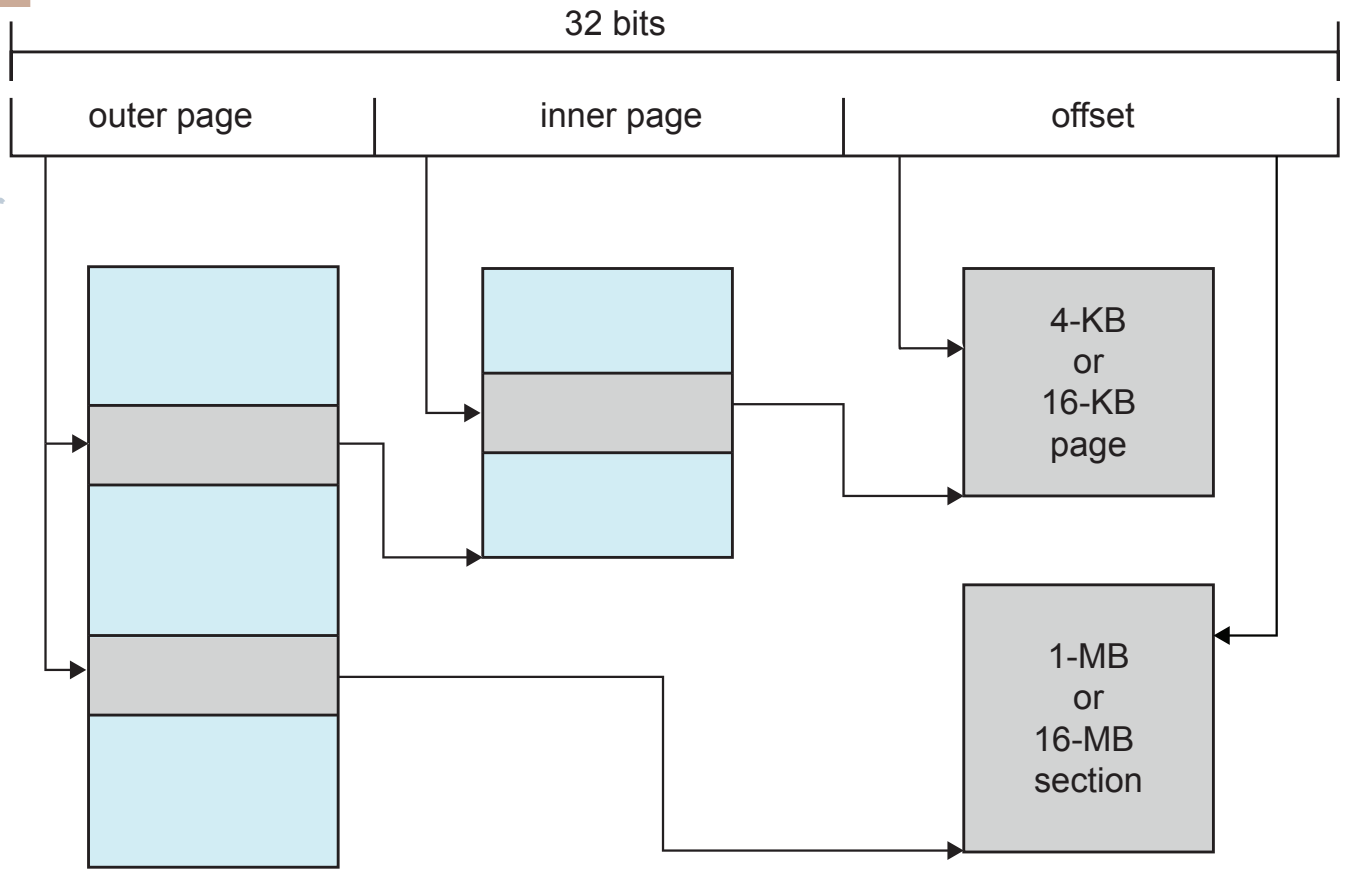
- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU

- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)

- One-level paging for sections, two-level for smaller pages



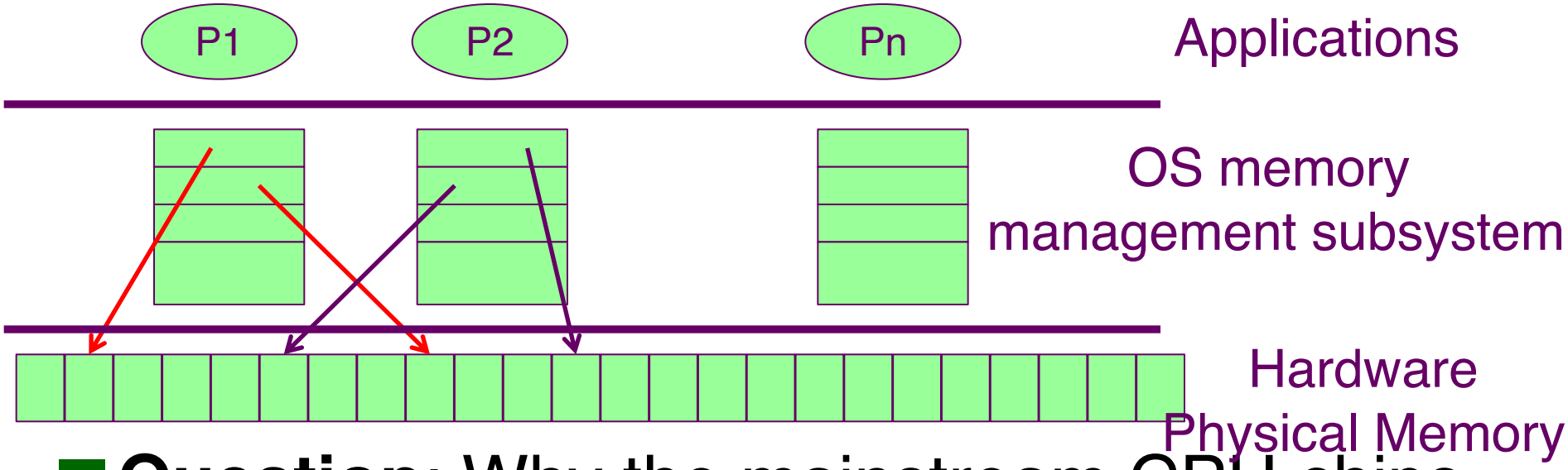
Example: ARM Architecture (Cont.)



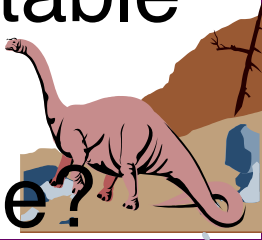
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - Firstly, inner is checked, on miss outers are checked, and on miss page table walk performed by CPU

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?



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
  - Versioning may be needed