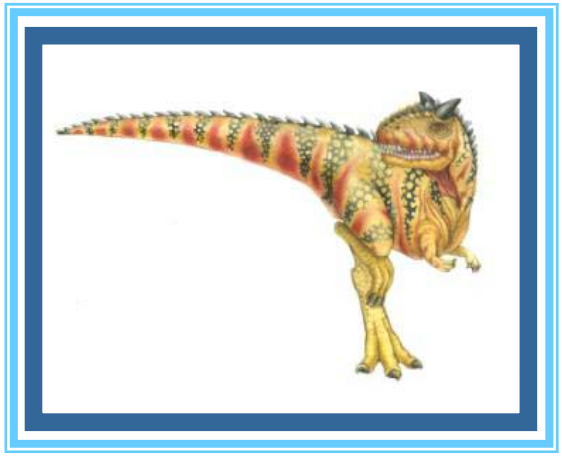
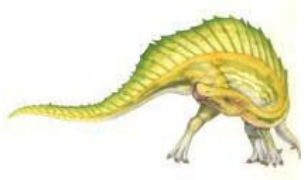
Topic 3: Memory Management



- Chapter 8: Main Memory
- Chapter 9: Virtual Memory

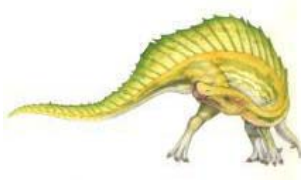
Contents

- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy



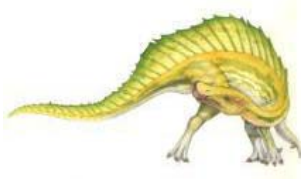
Contents

- Background and Memory hierarchy
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy



Background

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory Management in charge of:
  - allocating memory to ensure a reasonable supply of ready processes to consume available processor time
  - How? Different strategies for it
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests



Goals of Memory Management

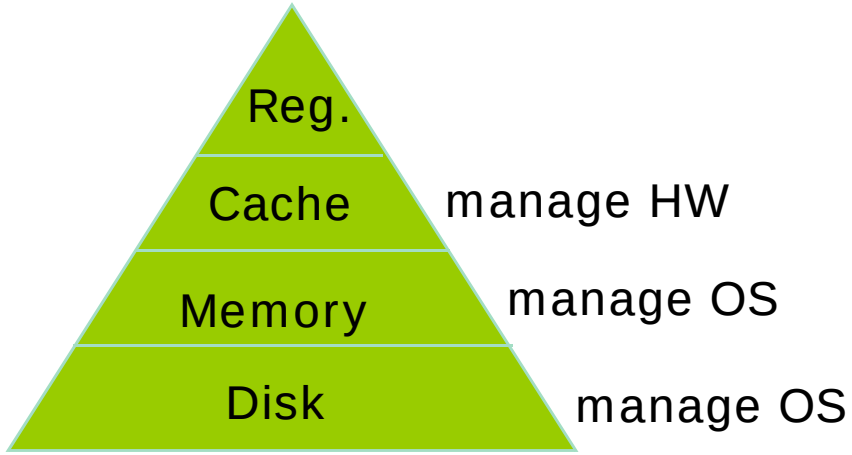
- O.S. multiplexes resources between processes
  - each process thinks all the machine is for its own
  - process management: distribute processor
  - memory management: distribute memory
- Memory Management goals:
  - provide each process with own logical space
  - offer protection between processes
  - allow processes to share memory
  - maximize level of multiprogramming
  - offer processes very large memory spaces
  - simultaneously: speed and low cost

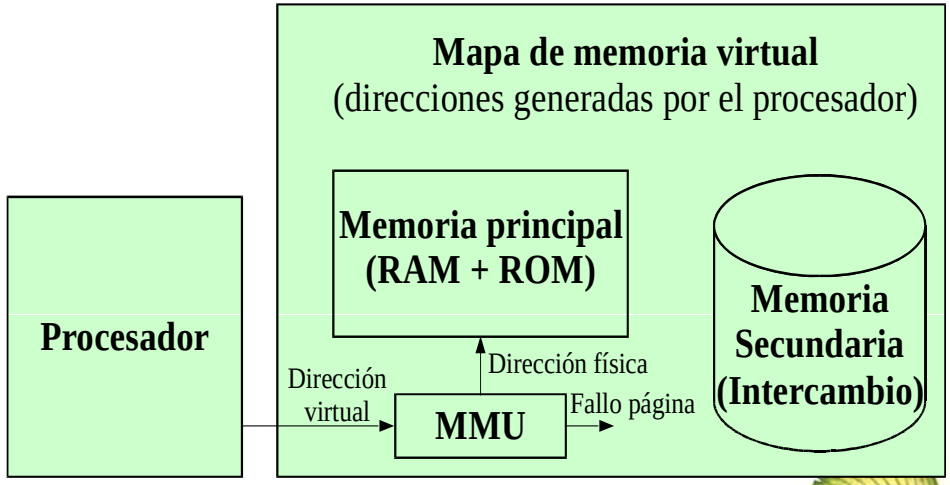
Virtual Memory

- Virtual Memory technique fulfills previous goals
- Virtual memory:
  - involves swapping blocks of data from secondary storage
  - the CPU generates virtual addresses
  - part of memory map is in Memory and part on HD
  - the MMU translates virtual addr. into physical addresses
  - page fault if the address is not in main memory
  - the OS brings one page to Mem (possibly replacing in Mem)

Memory Hierarchy

- Main memory and registers are only storage CPU can access directly
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Virtual memory adds to memory hierarchy

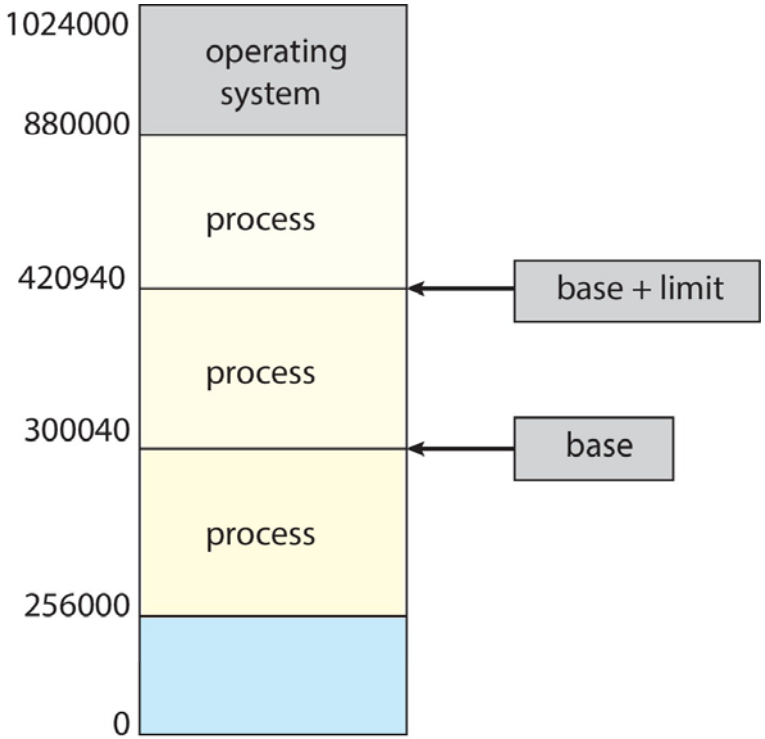




source: Sistemas Operativos. Una visión aplicada. Fig 1.14.

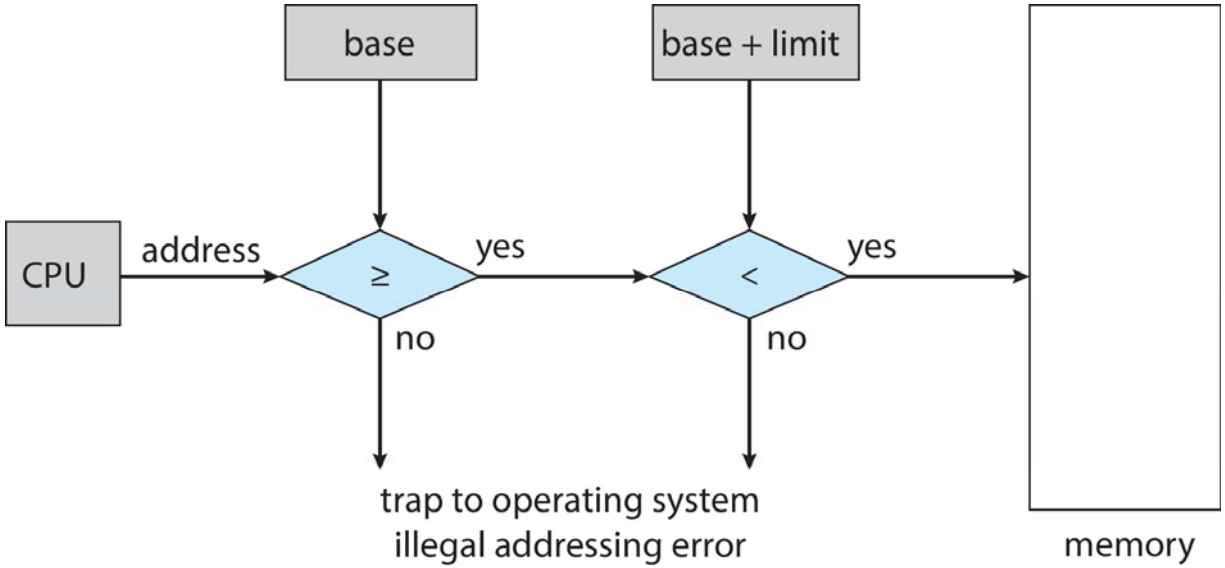
Protection

- Need to ensure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

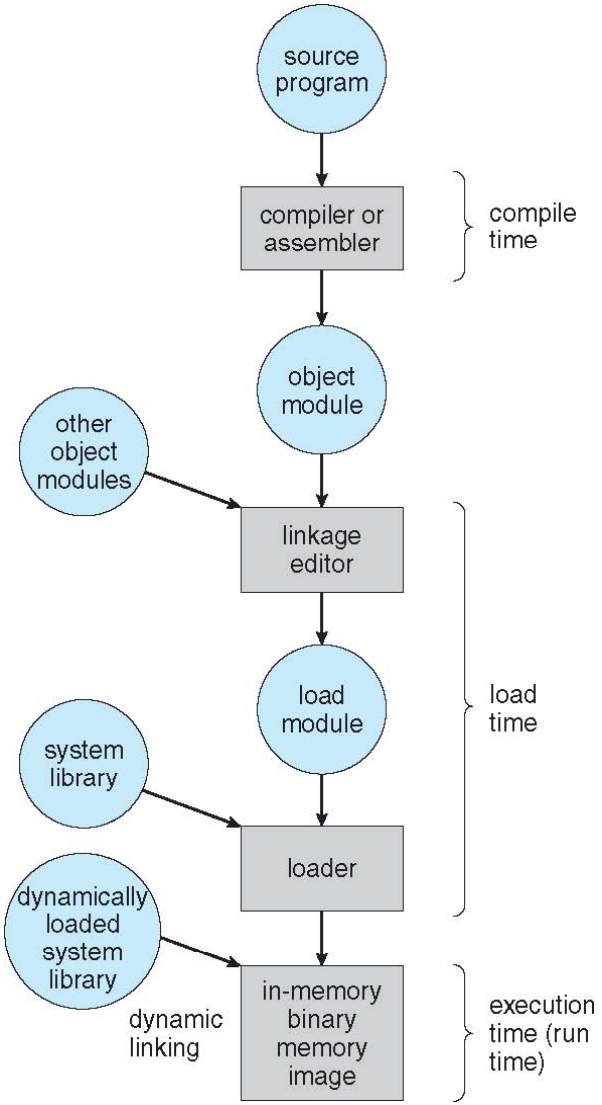


- the instructions to loading the base and limit registers are privileged

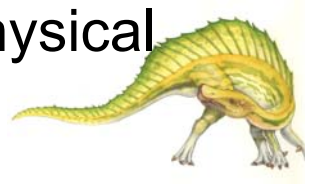
Contents

- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy

Multistep Processing of a User Program



- From source code to process
  - Compiling
    - separates code from data
  - Linking
    - solves cross references between modules
  - Loading
    - allocates initial addresses to partitions of the program
  - Execution
    - translates from logical to physical



Address Binding and Relocation

- The programmer does not know where the program will be placed in memory when it is executed
  - must be loaded into address 0000?
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
- Relocation occurs

Relocation

- *Relocatability* - the ability to move process around in memory without it affecting its execution
- OS manages memory, not programmer, and processes may be moved around in memory
- MMM must convert program's logical addresses into physical addresses
- Process's first address is stored as virtual address 0
- *Static Relocation* - Program must be relocated before or during loading of process into memory. Program must always be loaded into same address space in memory, or relocator must be run again
- *Dynamic Relocation* - Process can be freely moved around in memory. Virtual-to-physical address space mapping is done at run-time

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

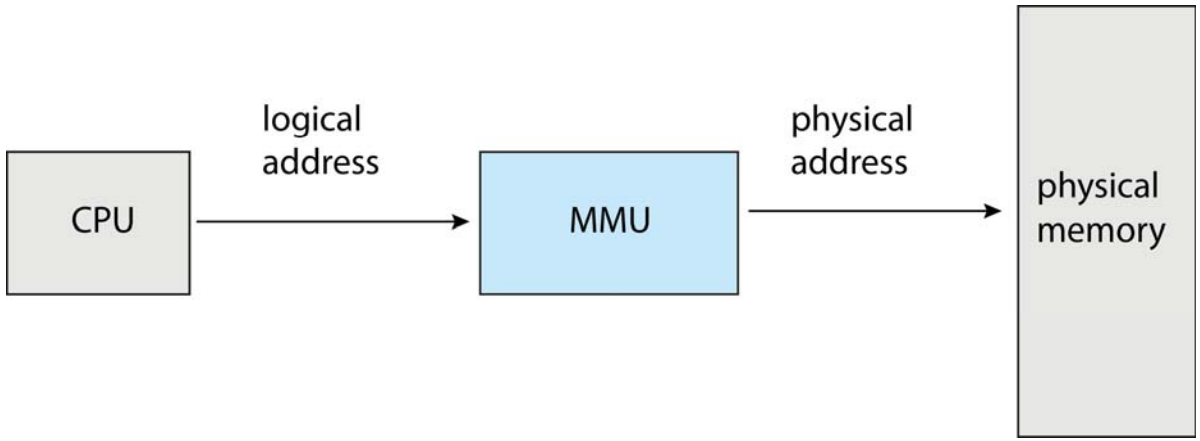
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory

Logical vs. Physical Address Space

- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program
- Logical and physical addresses are:
  - the same in compile-time and load-time address-binding schemes;
  - logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit

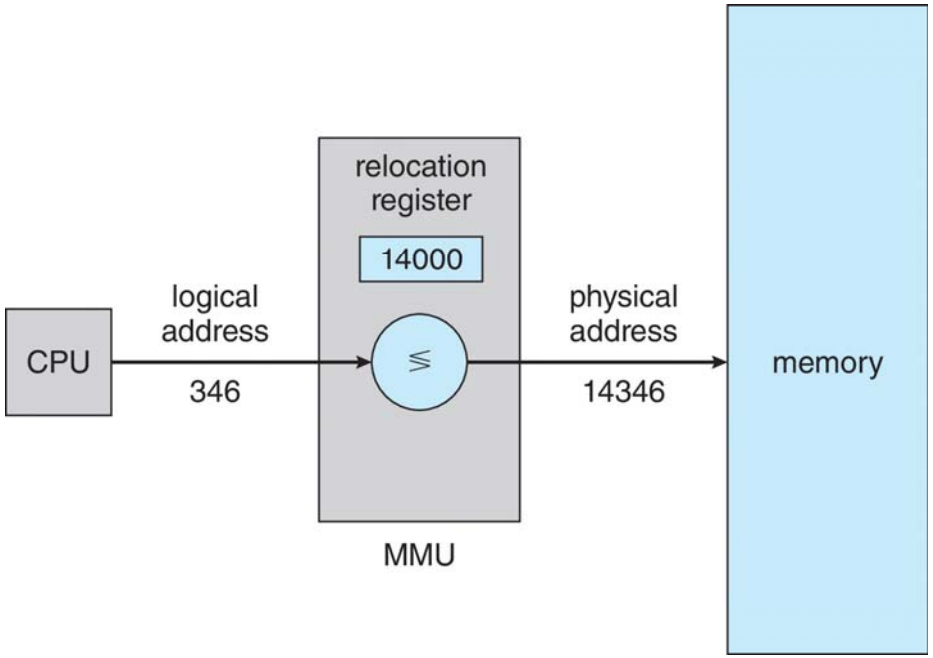
- Memory management unit (MMU)
  - Hardware device that at run time maps virtual (logical) to physical address



- MMU implementation depends on memory management strategy (memory allocation later on slides)

Memory-Management Unit

- Consider simple scheme which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Dynamic Loading

- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system 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

Contents

- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy

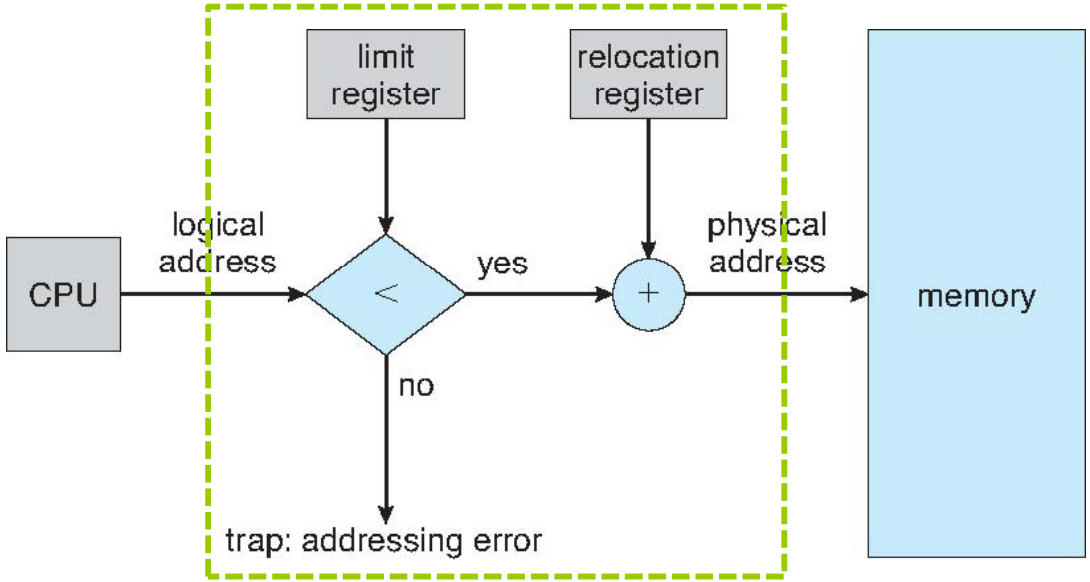
Memory Allocation Methods

- In a multiprogramming system, OS has to provide memory to processes, how?
- Main memory must support both OS and user processes
- Main memory is a limited resource, must be allocated efficiently
- Memory allocation methods:
  - Contiguous Allocation
    - Fixed and Variable Partitioning
    - Dynamic Allocation
  - Non-Contiguous Allocation
    - Paging
    - Segmentation
    - Virtual memory

Contiguous Allocation

- Contiguous allocation is one early method
  - a (whole) process is contained in a single contiguous section of memory
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- HW support for protection:

MMU:

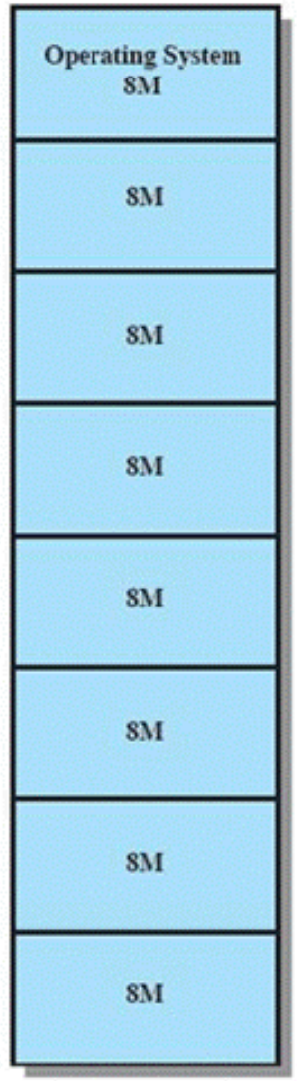


Contiguous Allocation (Cont.)

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

Fixed partitioning

- OS divides memory into partitions with fixed size (equal-size partitions)
  - new process → assign a partition ≥ size needed by the process
  - management: table with as many entries as partitions
  - OS can swap a process out of a partition
- Problems:
  - A program may not fit in a partition
  - Any program, no matter how small, occupies an entire partition
  - This is results in ***internal fragmentation***

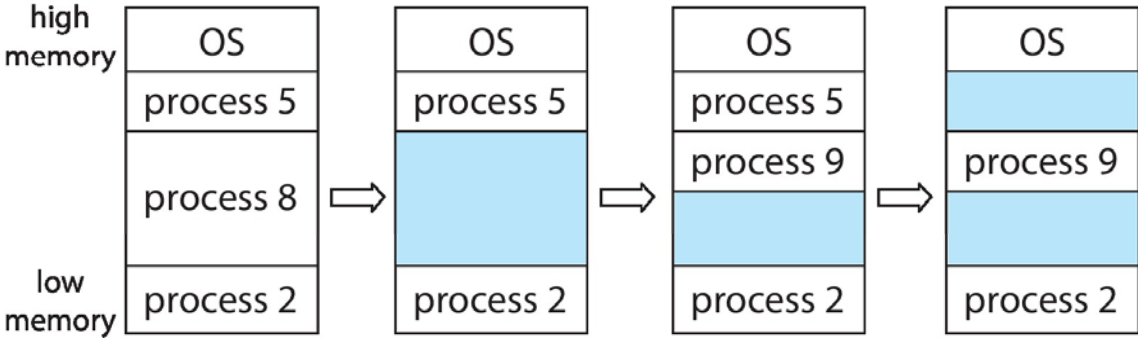


(a) Equal-size partitions

source: Operating Systems: Internals and Design principles. William Stallings

Variable Partitioning

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



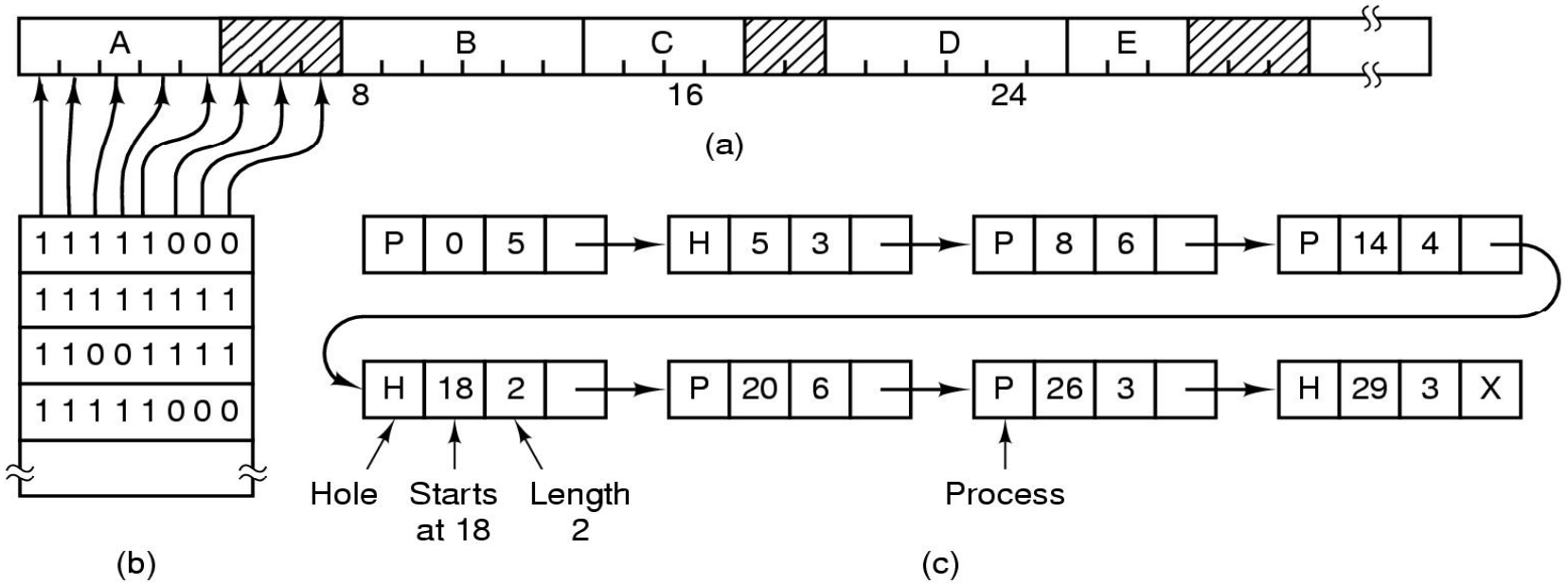
Partitions management

- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)
- Management of free and allocated partitions (blocks)
  - linked list or bitmap
- Bitmap: divide memory into small blocks (clicks)
  - a bitmap indicates free clicks with a zero
  - allocate consecutive clicks to a process
  - internal fragmentation may occur
- Linked lists: each element refers to a memory segment
  - two lists: one for free segments one for occupied ones
  - several free segment lists for different sizes

Partitions management

- Examples of space management
  - with bitmaps
  - with linked lists

memory segment with 5 Processes and 3 free Holes



Dynamic Storage-Allocation Problem

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

Allocation strategies:

- **First-fit**: Allocate the ***first*** hole that is big enough; must search entire list from the beginning and chooses the first available block that is large enough
- **Next-fit**: Allocate the ***next*** hole that is big enough. ; must search entire list from the last allocation and chooses the first available block that is large enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

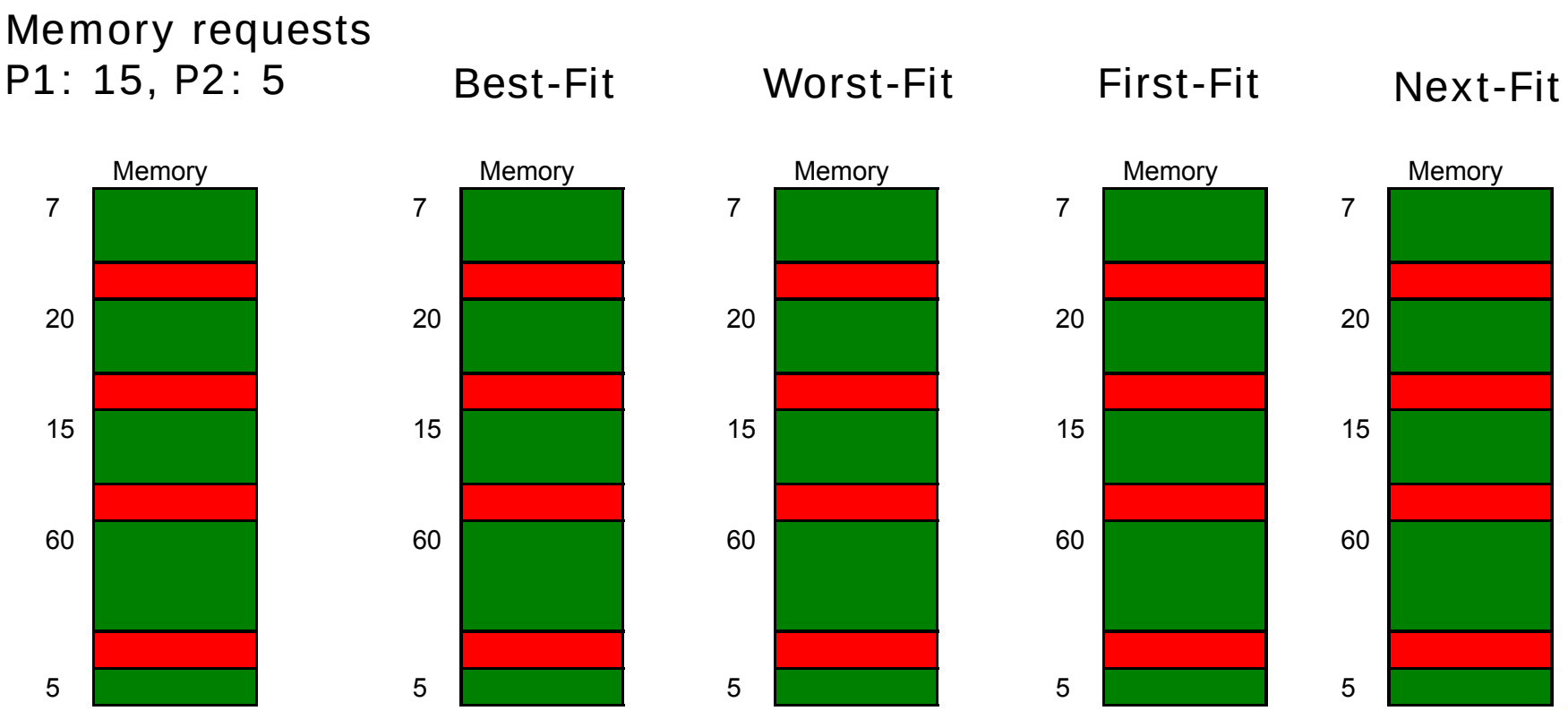
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

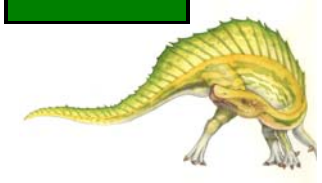
Dynamic Storage-Allocation Exercise

Green: free (available) memory - holes (size in the leftmost column)

Red: allocated memory

Arrives processes P1 and P2 (in this order)



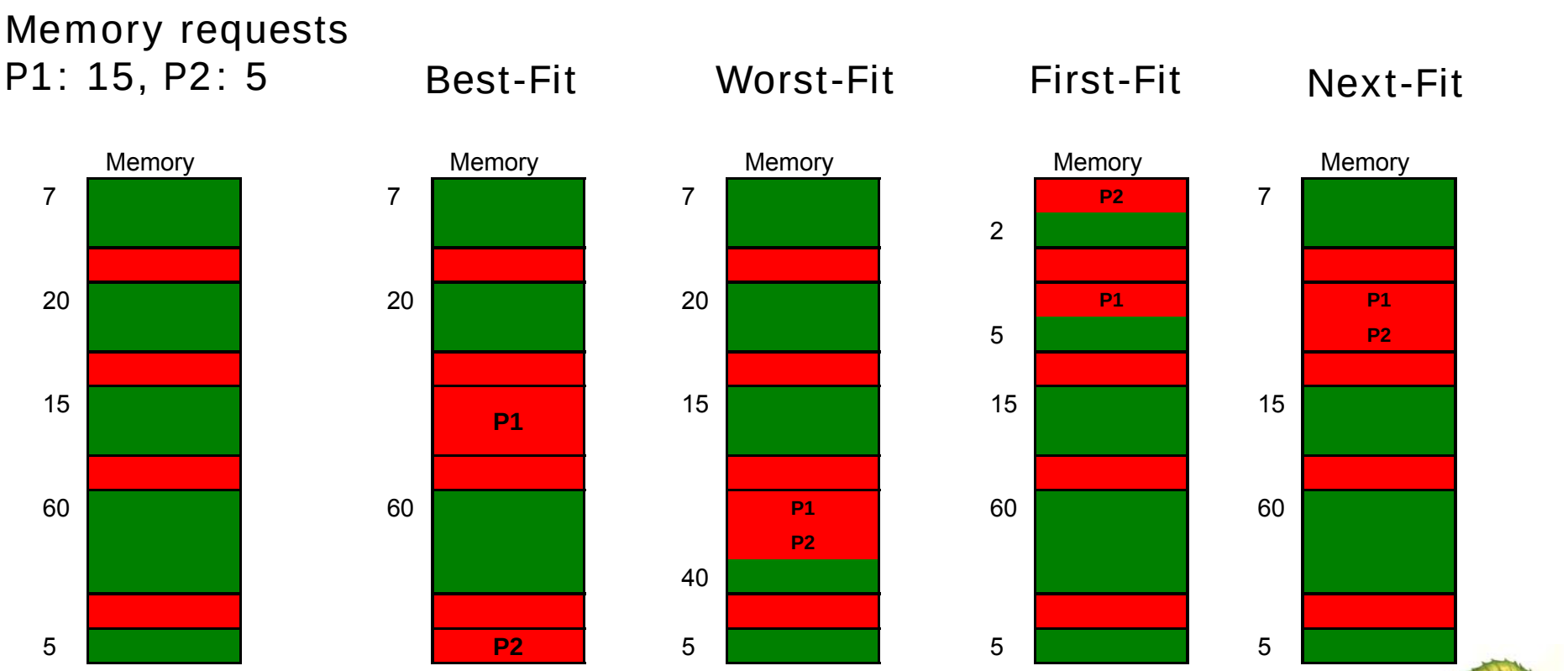


Dynamic Storage-Allocation Solution

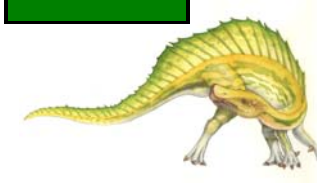
Green: free (available) memory - holes (size in the leftmost column)

Red: allocated memory

Arrives processes P1 and P2 (in this order)



Initial Memory

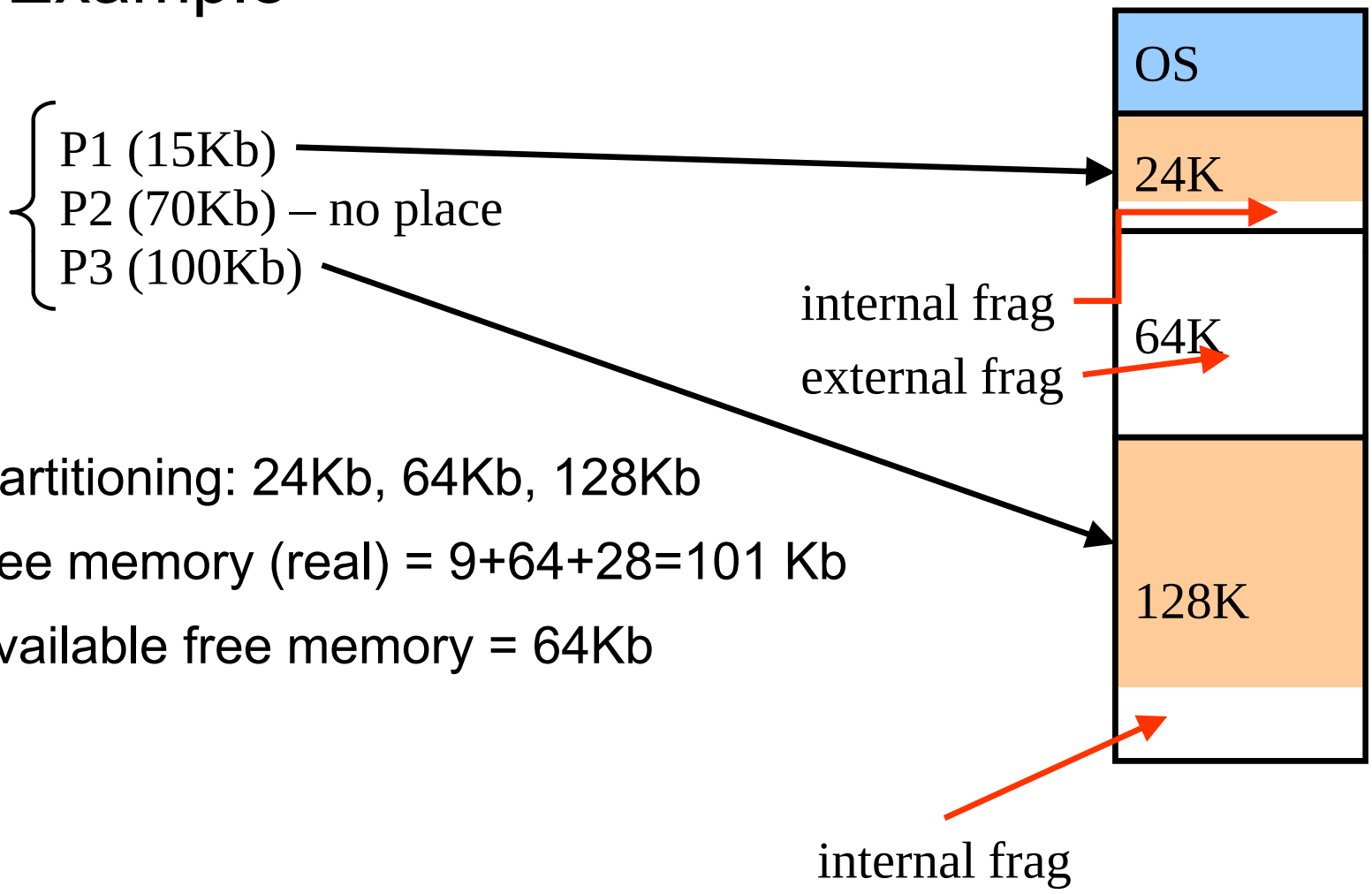


Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Example)

- Example



- Partitioning: 24Kb, 64Kb, 128Kb
- free memory (real) = 9+64+28=101 Kb
- available free memory = 64Kb

Solutions to Fragmentation

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Backing store (HD, flash memory, SSD,…) has same fragmentation problems (will be considered in Topic 4)

Contents

- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy

Motivation Non-Contiguous Allocation

Contiguous allocation model easy??

- Drawback of contiguous model:
  - finding partitions to keep the contiguous process in memory
  - if there is no sufficiently large hole:
    - Swap-out other processes : practical?
    - compaction of memory, relocating mode: high cost (time)
    - don’t create the process: low flexibility

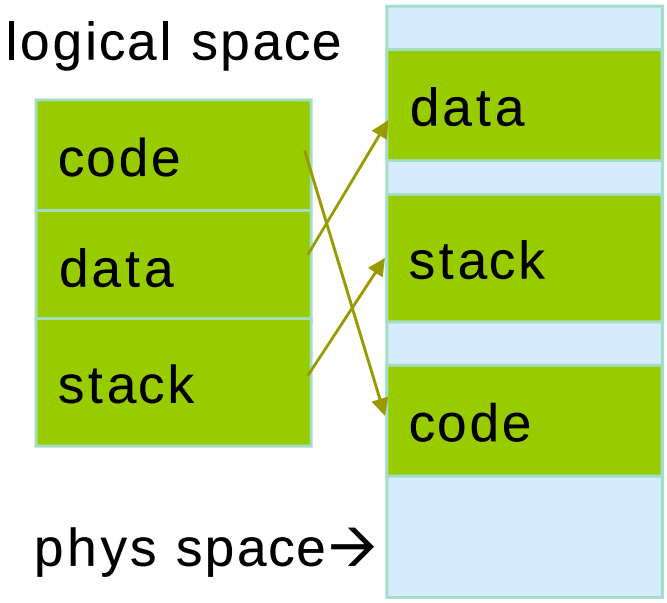
Motivation Non-Contiguous Allocation

Why not to follow a non-contiguous model?

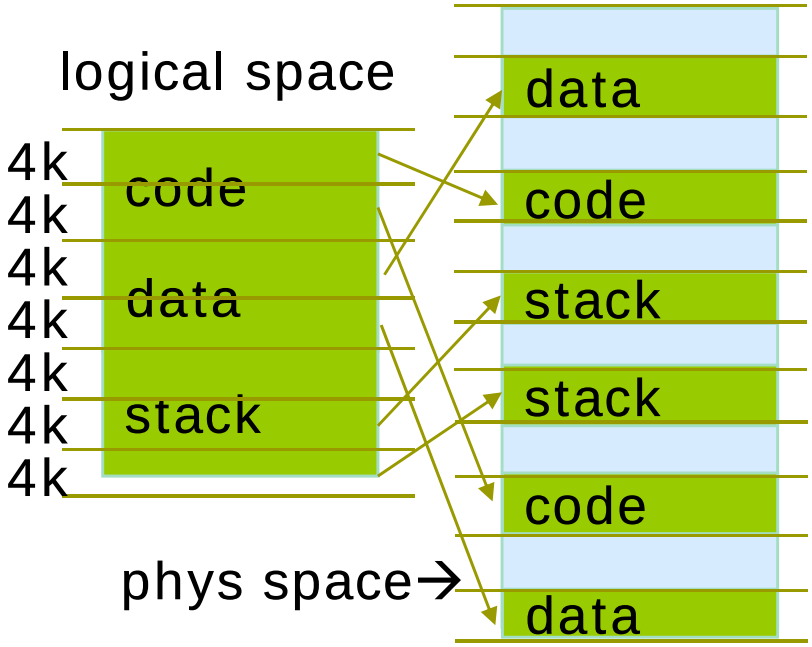
- Solution: fragmenting the logical process space
  - Map logical space onto physical space in fragments
- Strategies:
  - Paging
    - Implementation: design criteria
    - Optimization: multilevel paging inverted page table, TLB
  - Segmentation
  - Paged segmentation

Non-Contiguous Allocation

- Non-Contiguous allocation
  - no worry about using consecutive physical space
  - process can be scattered over separated blocks in memory (non-contiguous sections)
  - logical space still contiguous, physical one not
- Solutions:



Segmentation



Paging

Contents

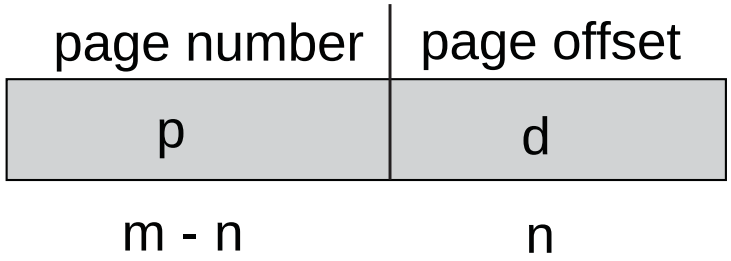
- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide **physical memory** into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide **logical memory** into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

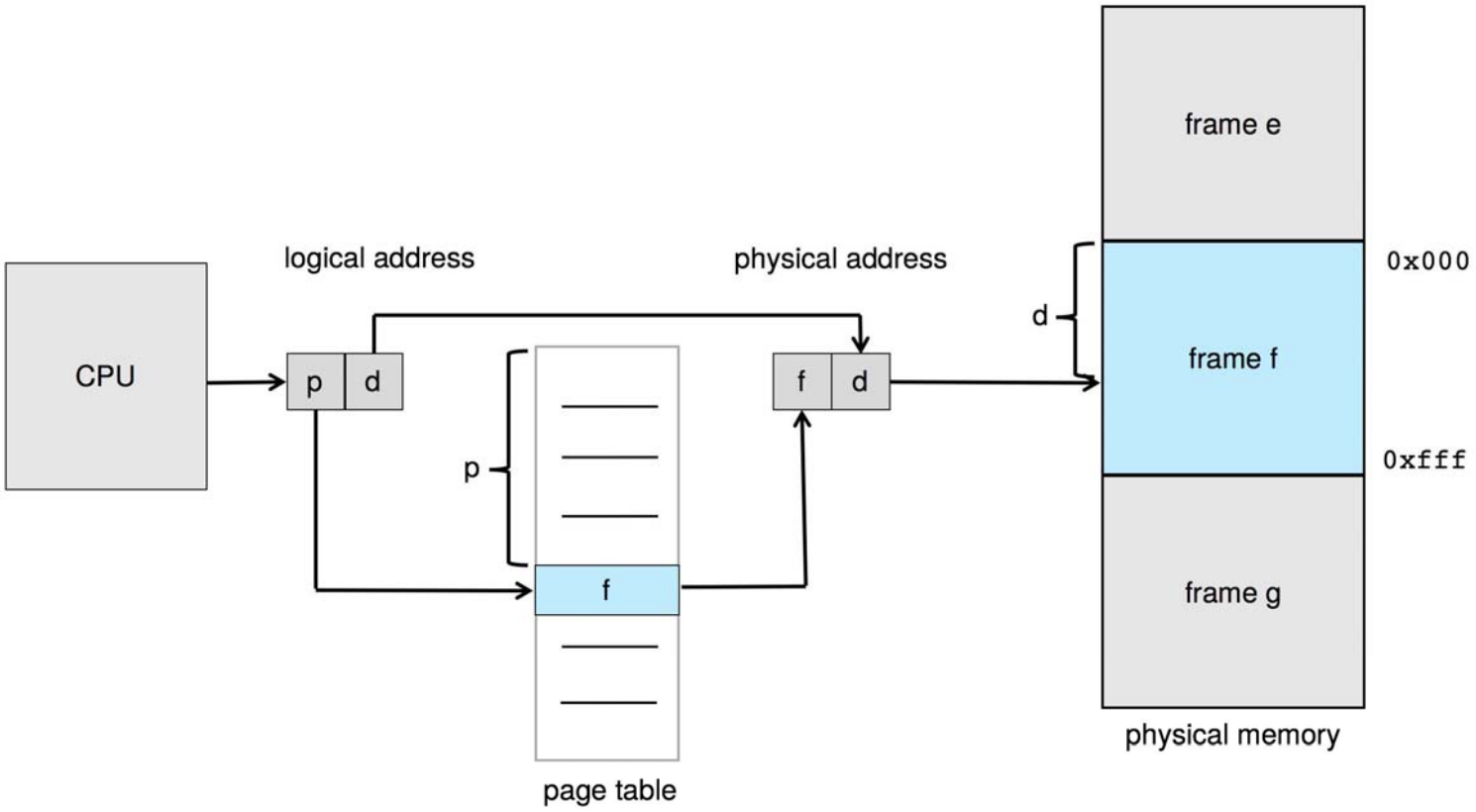
Address Translation Scheme

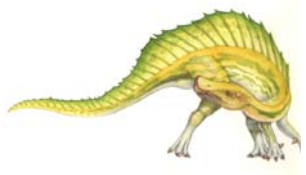
- Address generated by CPU (virtual/logical) is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent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



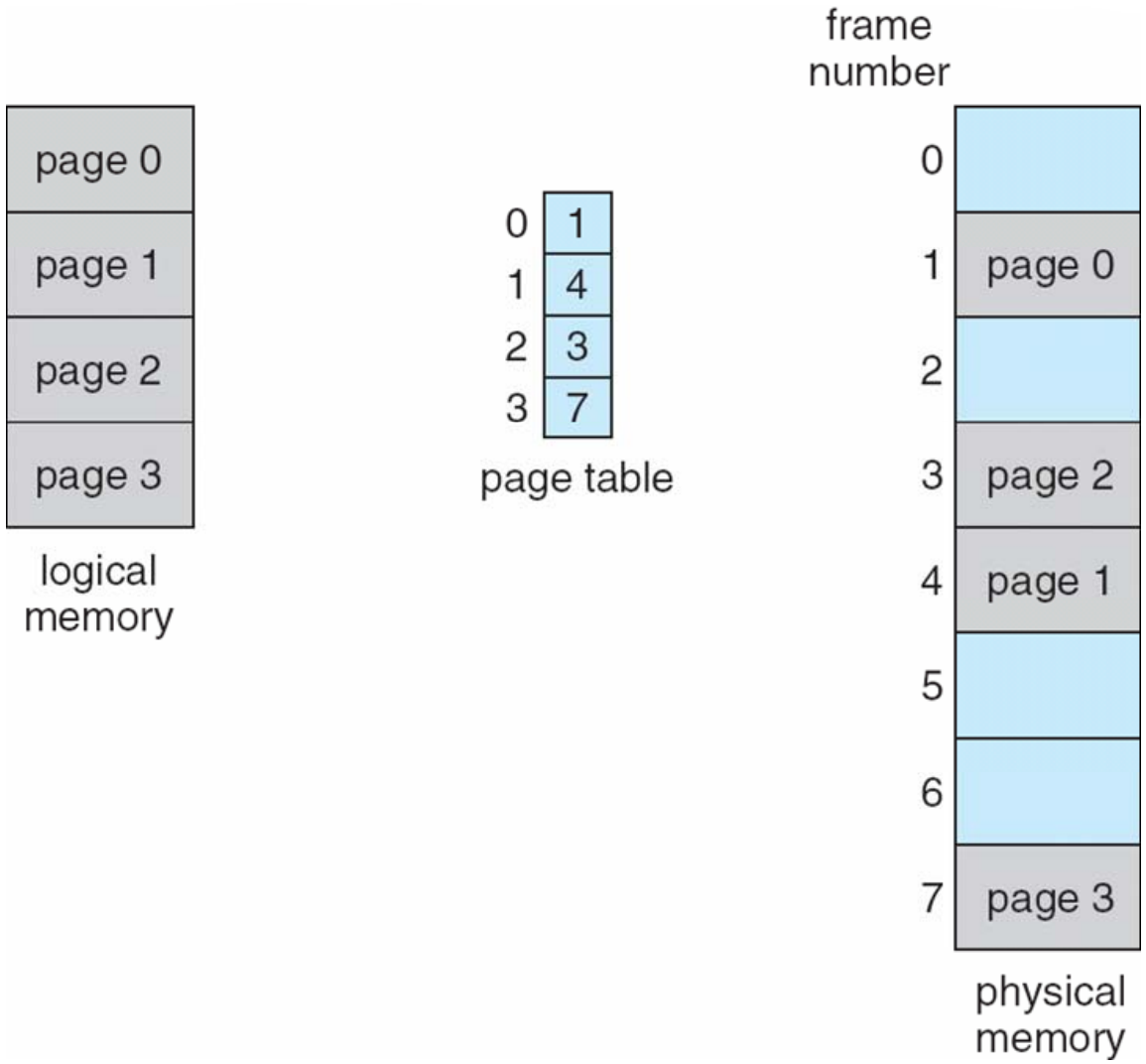
  - For given logical address space 2^m and page size 2^n

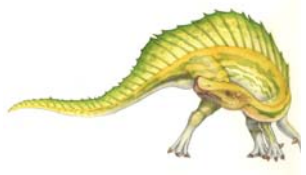
Paging Hardware





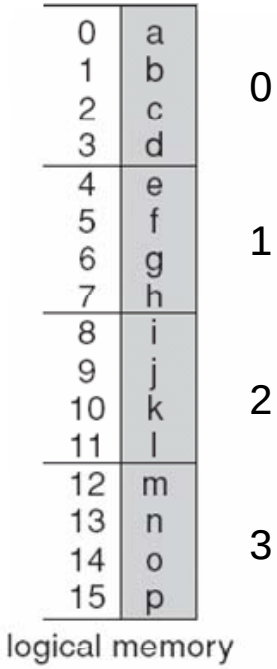
Paging Model of Logical and Physical Memory

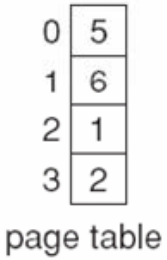


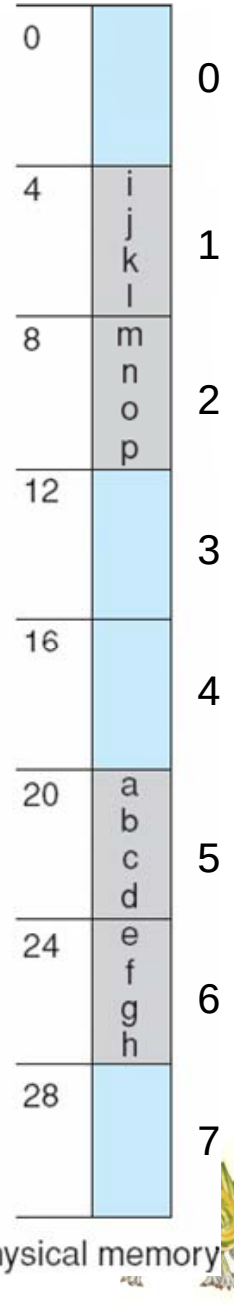


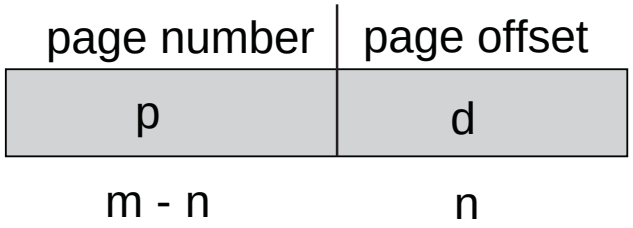
Toy Paging Example

- For a given logical address space 2^m and page size 2^n bytes
  - logical address: m = 4 bits
  - using a page size of 4 bytes (2^2 -> n=2)
  - and a physical memory of 32 bytes (8 pages)









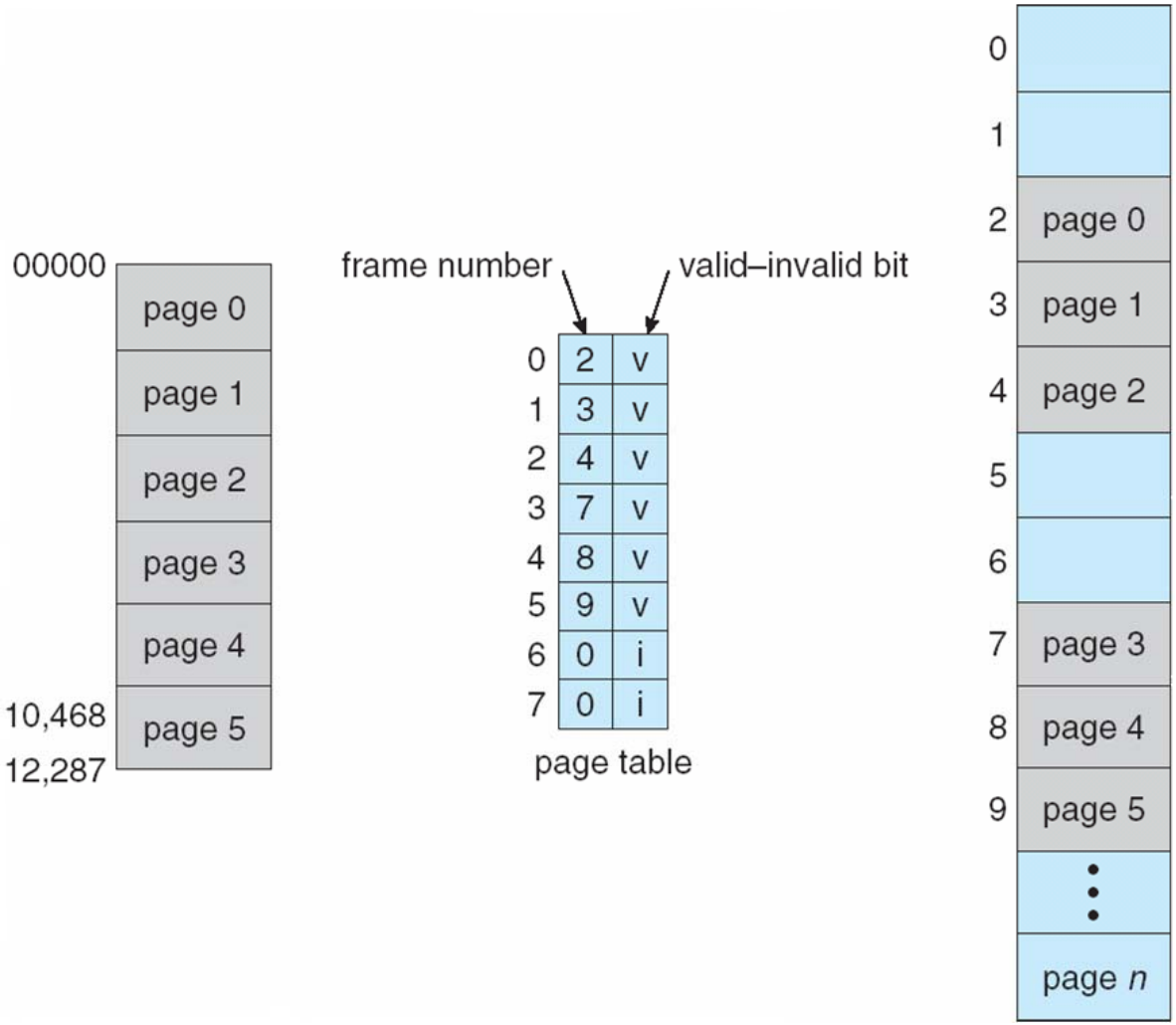
Paging -- Calculating internal fragmentation

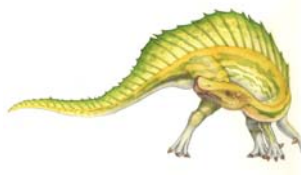
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Memory Protection

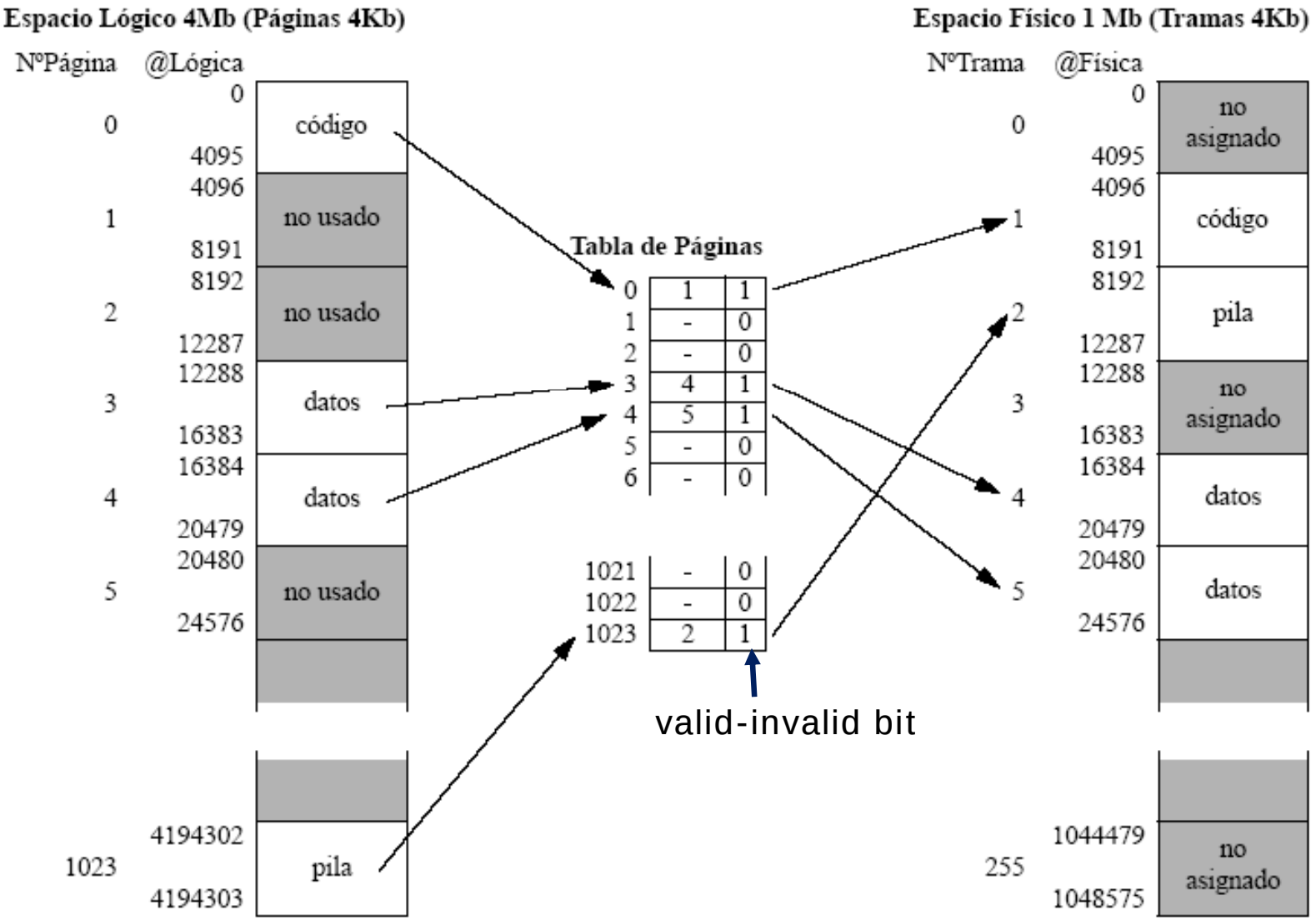
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated 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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

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





Paging Example



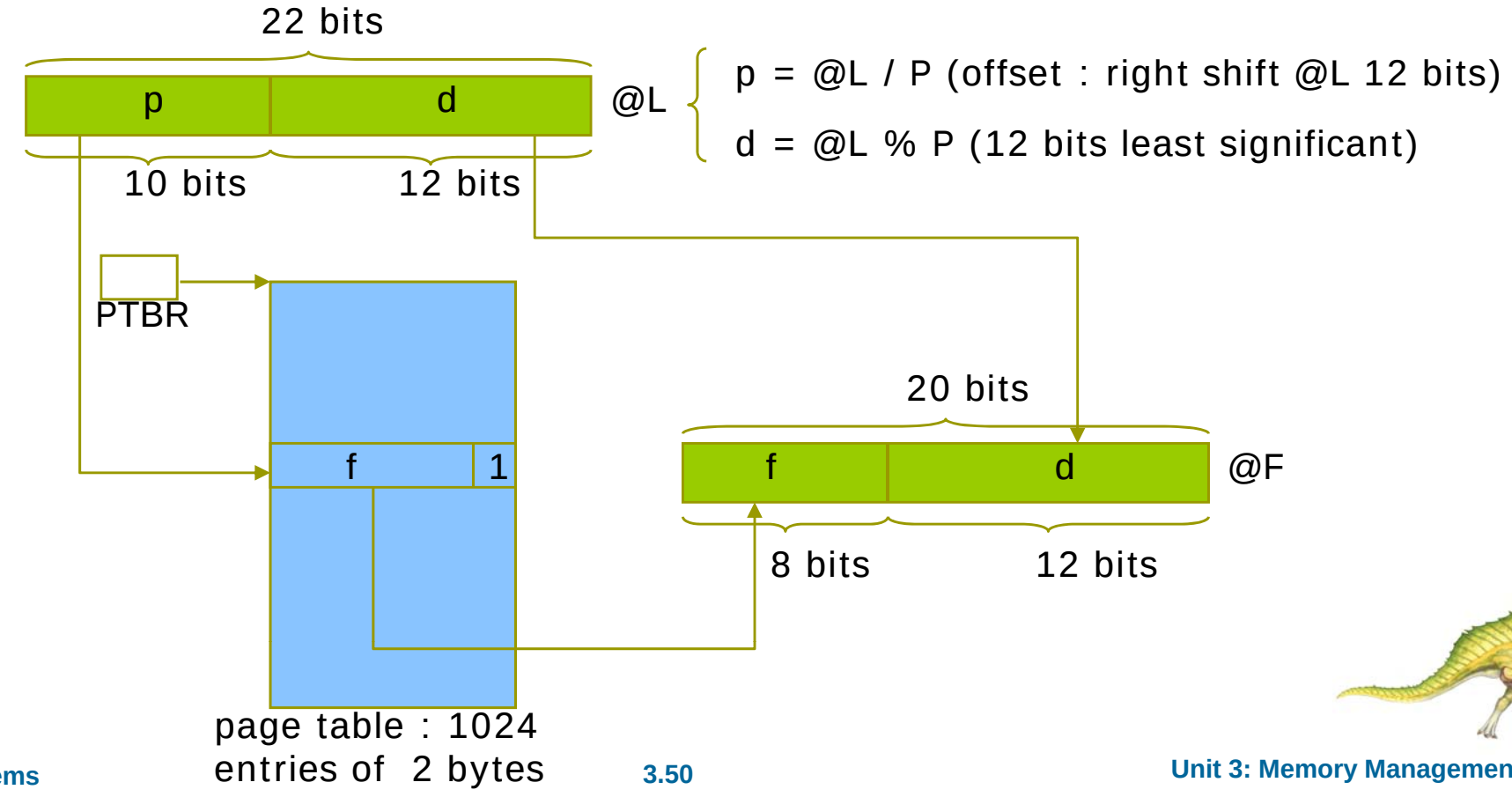
Address Translation Example

- Useful notation:
  - @L: logical address @F: physical address
  - d: offset (byte within the page, **d**isplacement)
  - p: page number
  - f: frame number
  - P: page size PT: page Table
  - PTBR: Page Table Base Register
    - starting point in PT of the process (stored in memory)
  - bit range in the addresses:
    - [M..N] represents bit M up to including bit N
    - example:

    @F[19..0] represents bits 19 to 0 of the physical address

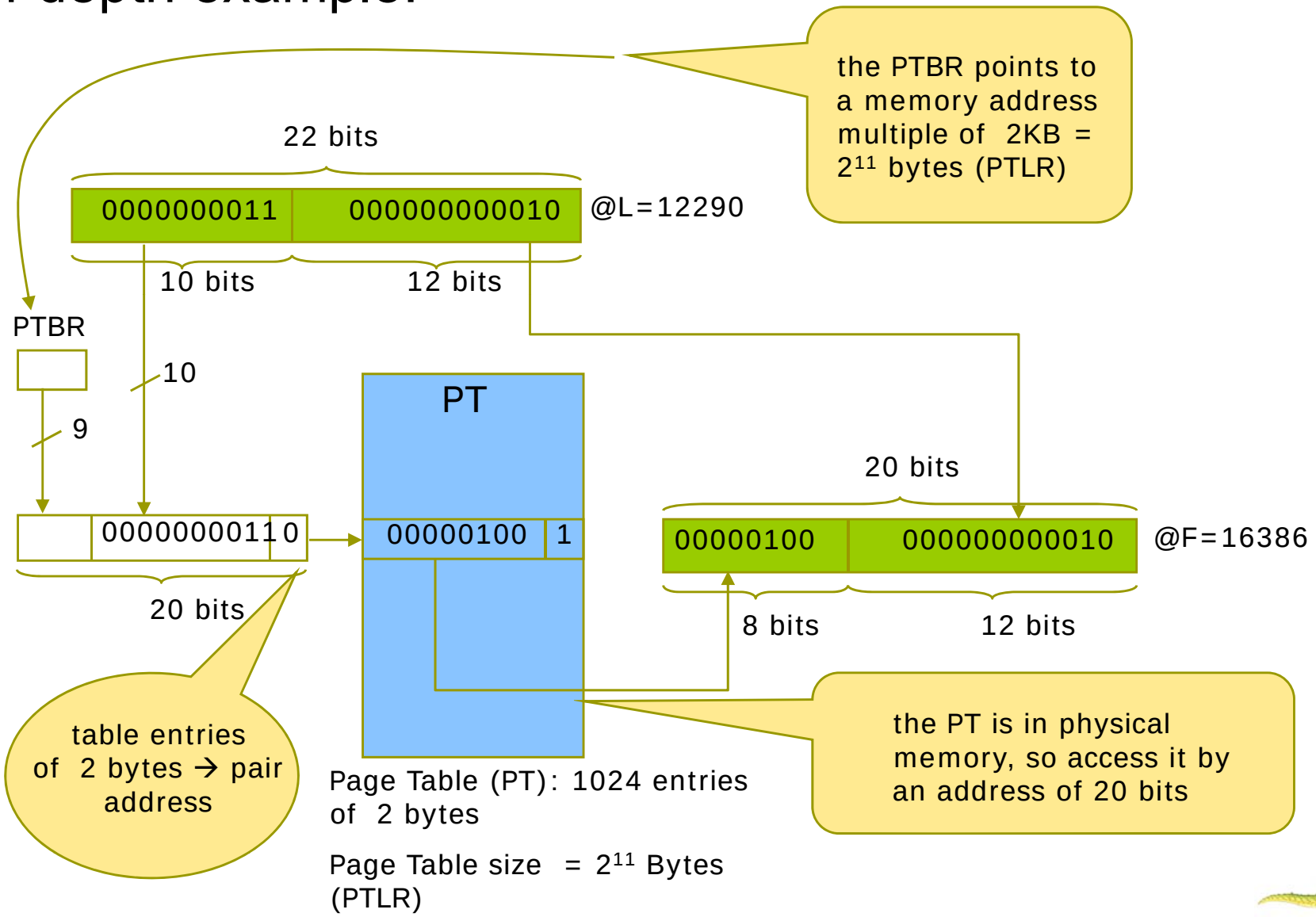
Address Translation Example

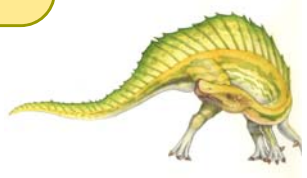
- Log. space 4MB (2^{22} bytes). → @L of 22 bits (@L[21..0])
- Phys. space 1MB (2^{20} bytes) → @F of 20 bits (@F[19..0])
- P = 4KB = 4096bytes = 2^{12} bytes



Address Translation Example

- In-depth example:





Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Size of page table (PT)
  - Number of entries = number of pages log space
    - Number of entries = size logical space / page size
    - larger pages:
      - less entries, more internal fragmentation in the processes
    - smaller pages:
      - more entries, less internal fragmentation

Implementation of Page Table

- Size of page table (PT)
  - Number of entries = size logical space / page size
- Bytes for each entry **(power of two)**
  - field address of frame (f) of width= $\log_2$(size phys space/P)
  - validity bit (V) : 1 if page has associated frame
  - access permission: R (Read), W (Write), X (eXecute)
  - access bit (A): 1 if page is accessed
  - modify bit (D): 1 if one writes on the page (Dirty)
  - presence bit (P): used in virtual memory

Implementation of Page Table

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

Translation Look-Aside Buffer

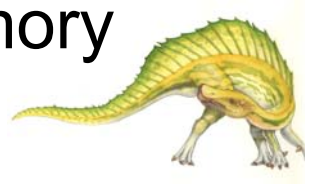
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Hardware

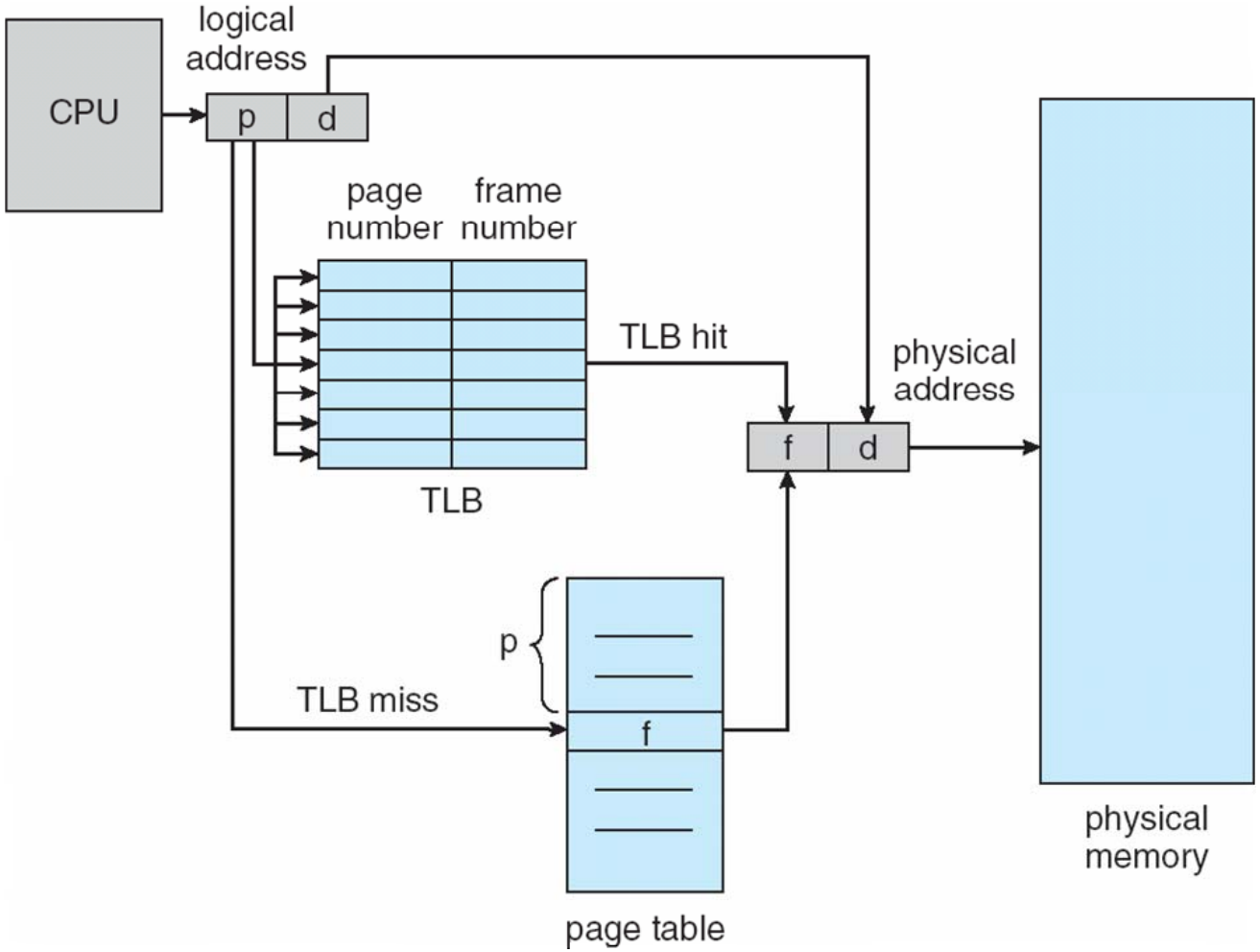
- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory



Paging Hardware With TLB



Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

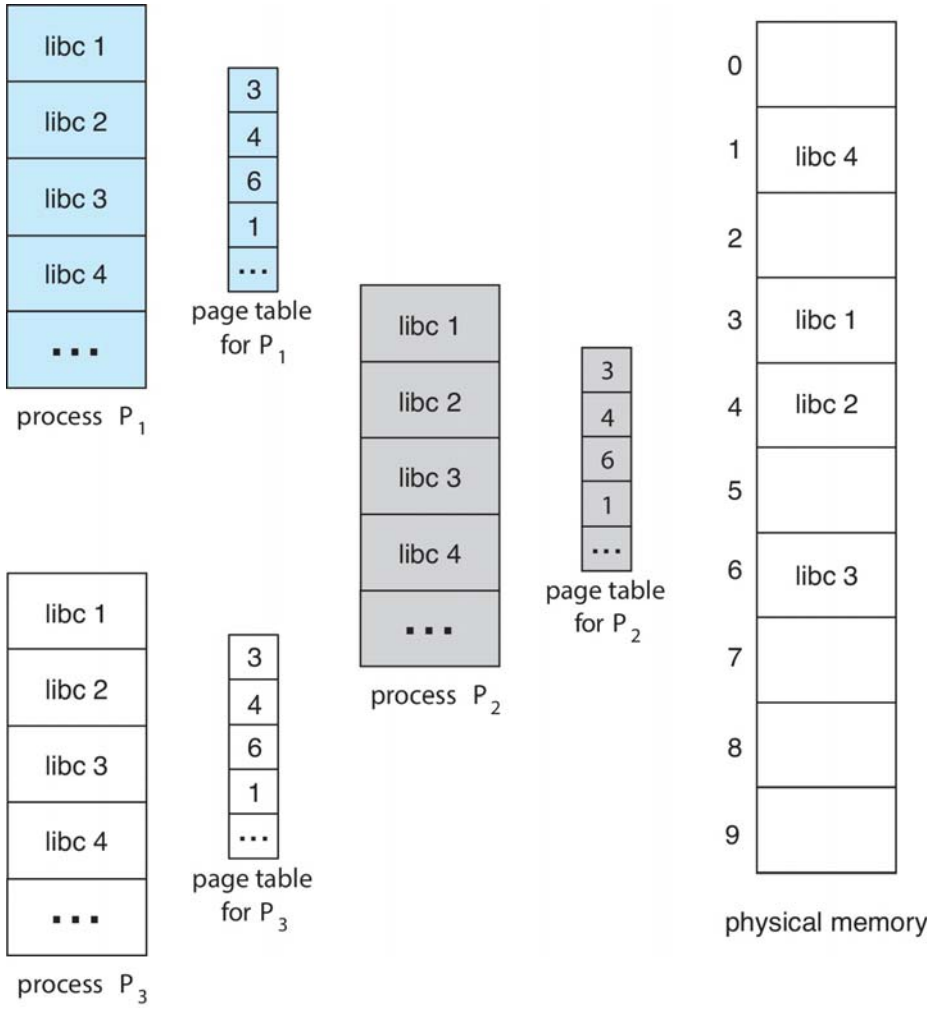
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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



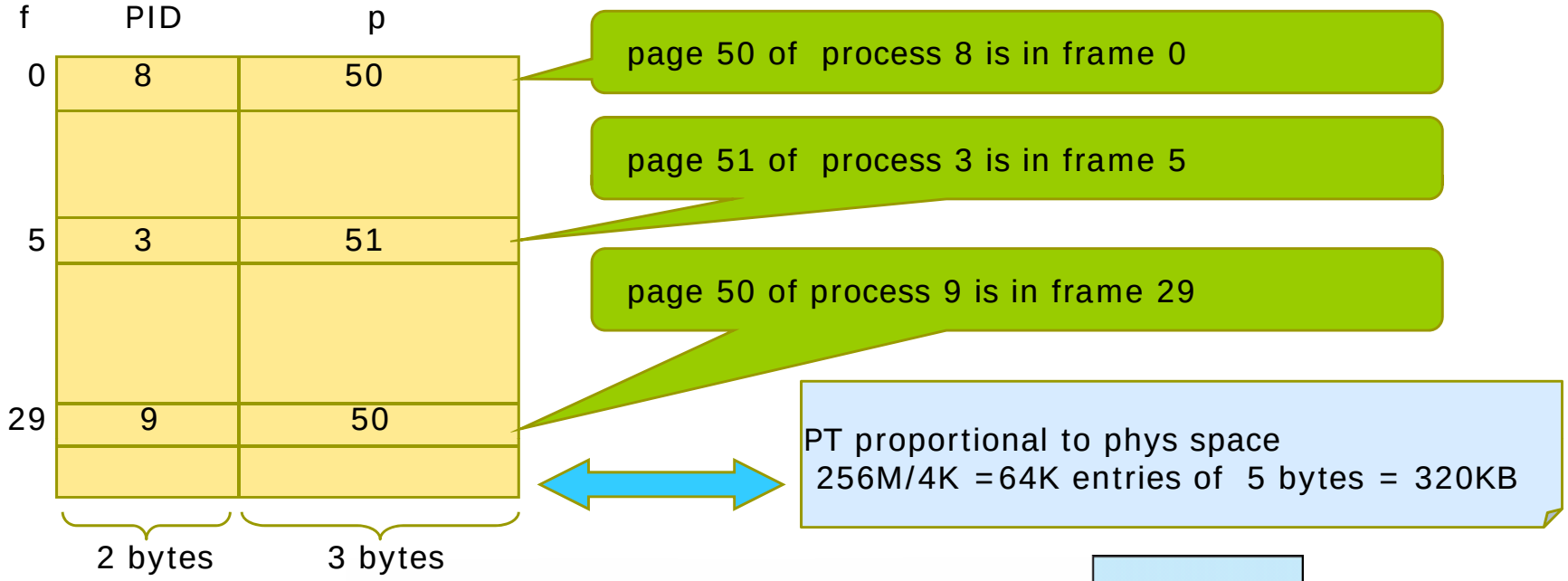
Structure of the Page Table

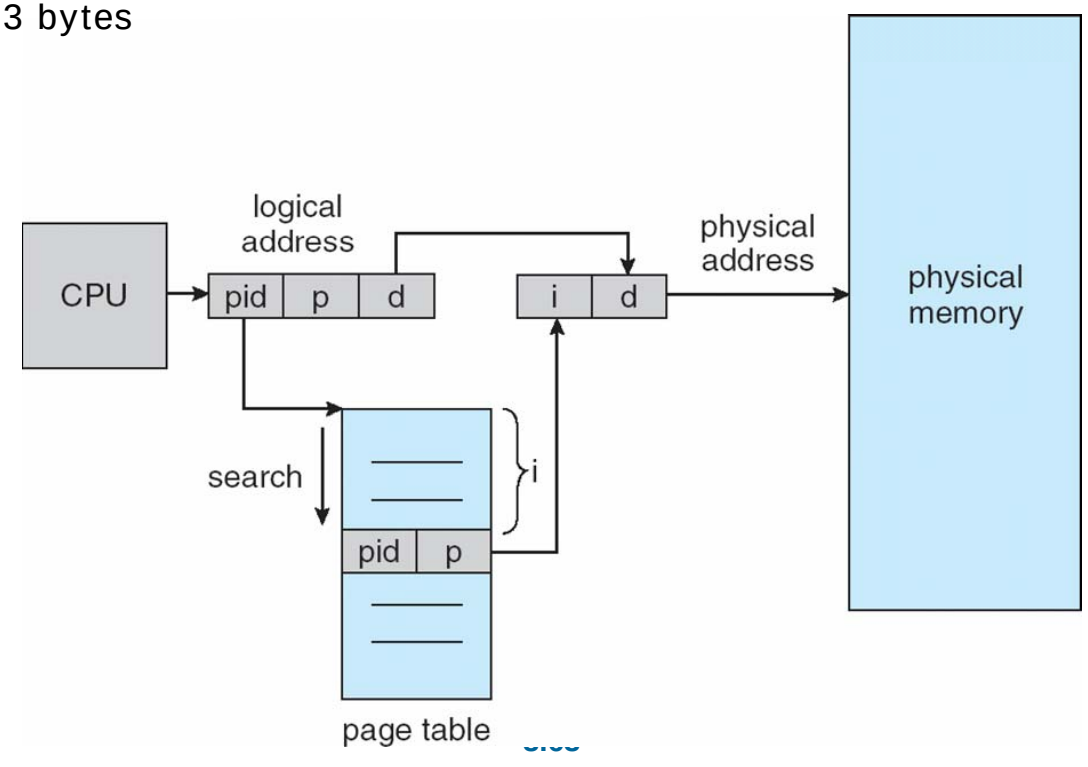
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - **Hierarchical Paging**
    - Hashed Page Tables
    - **Inverted Page Tables**

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

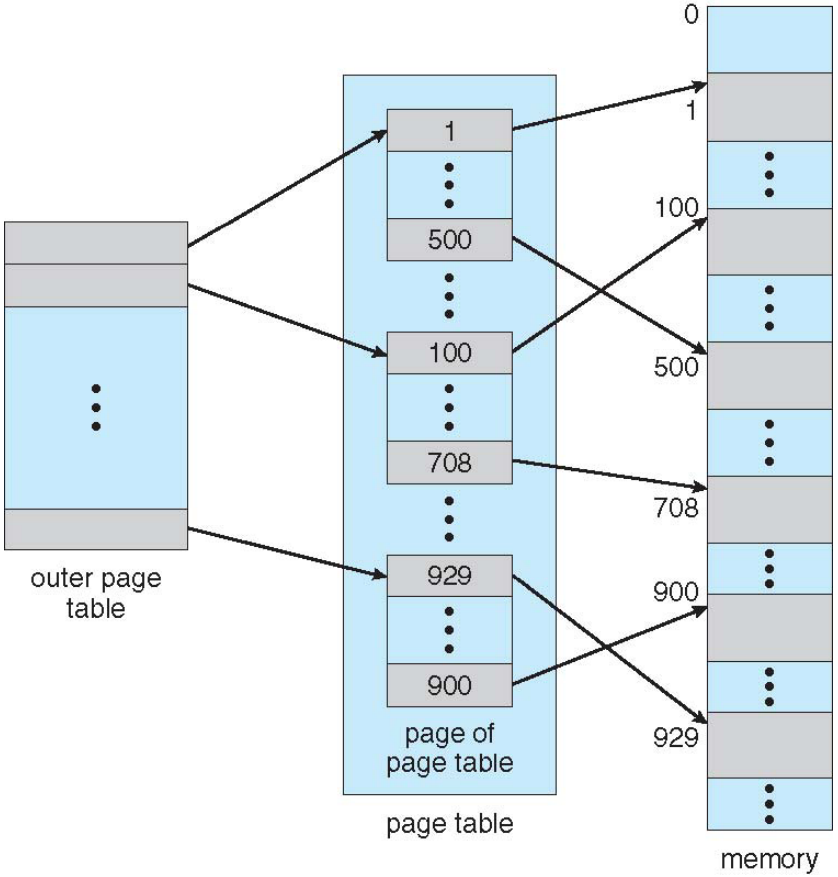
Inverted Page Table Architecture



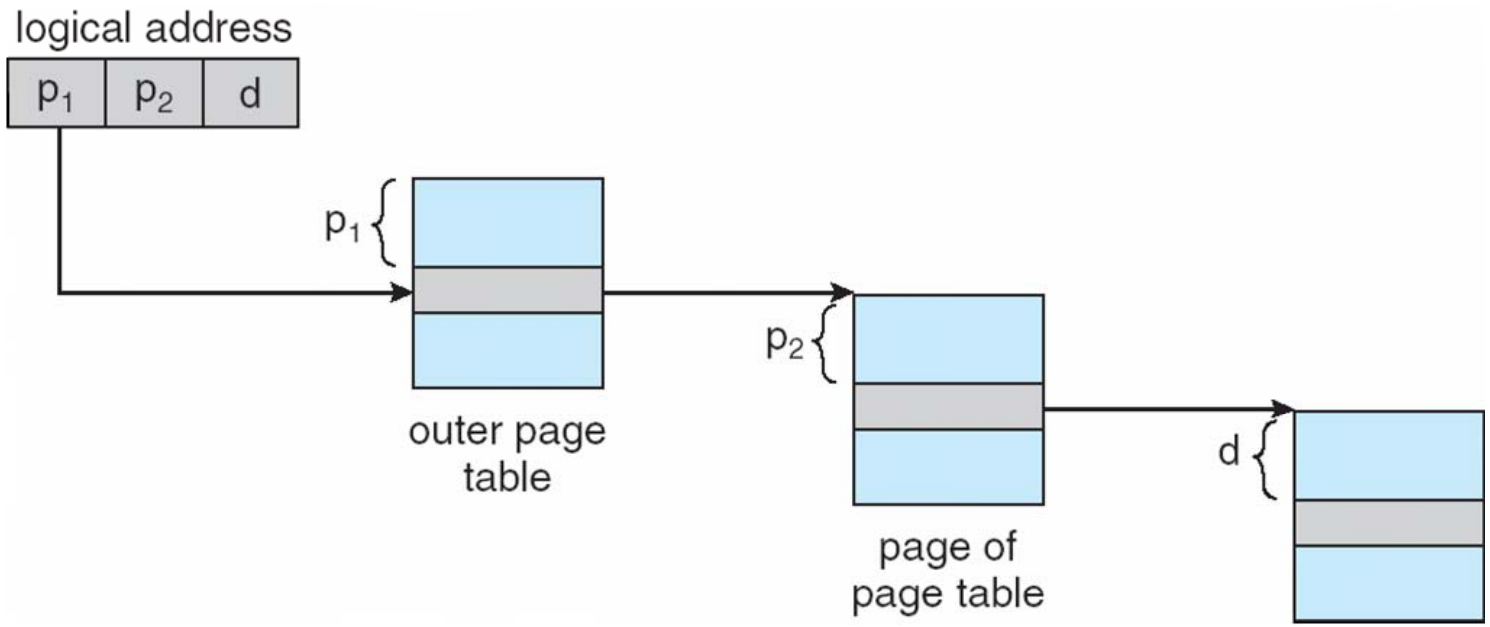


Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

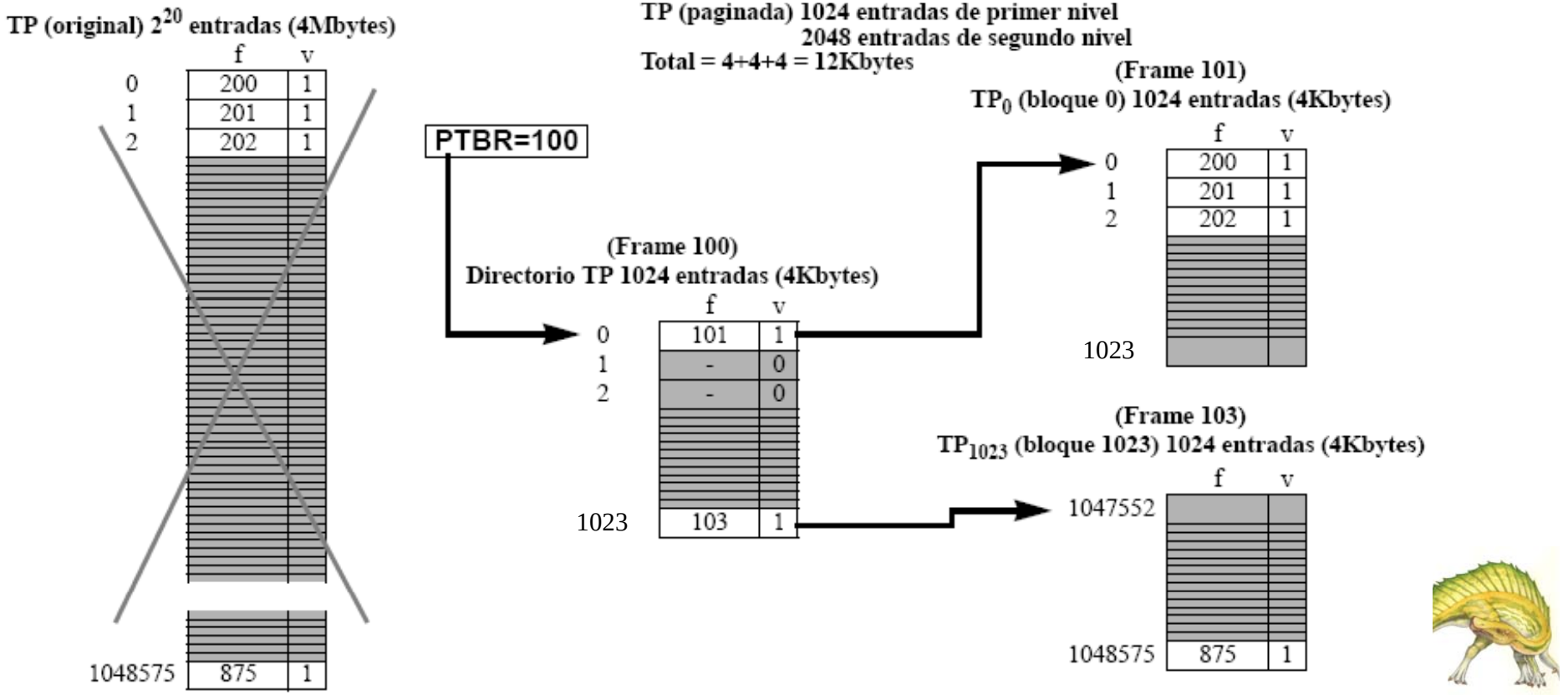


Address-Translation Scheme



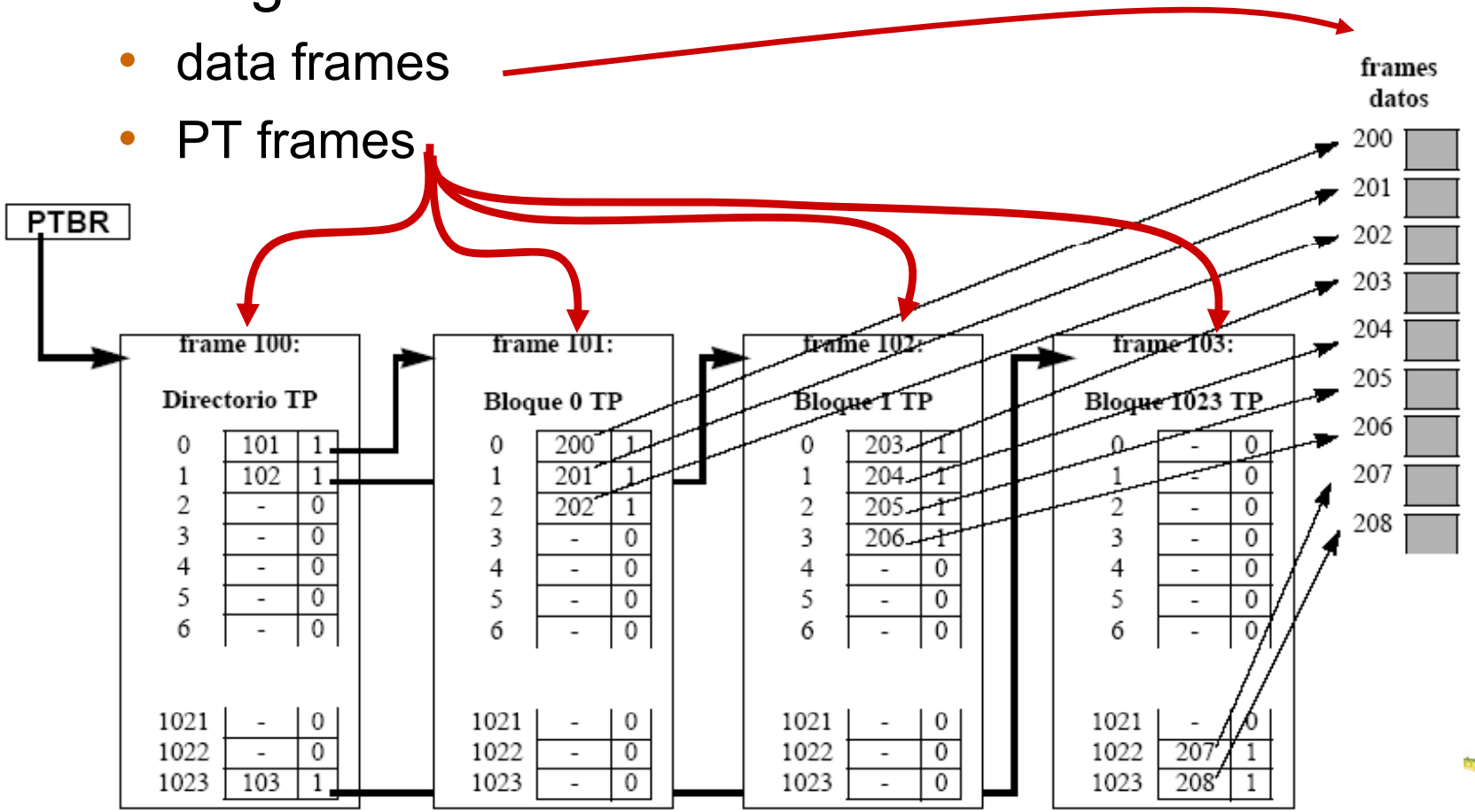
Two-Level Paging Example

- Paging on two levels:
  - Idea: paging the page table
  - Divide the original PT into blocks of one page size
    - if no valid entry in the block → block not valid.



Two-Level Paging Example

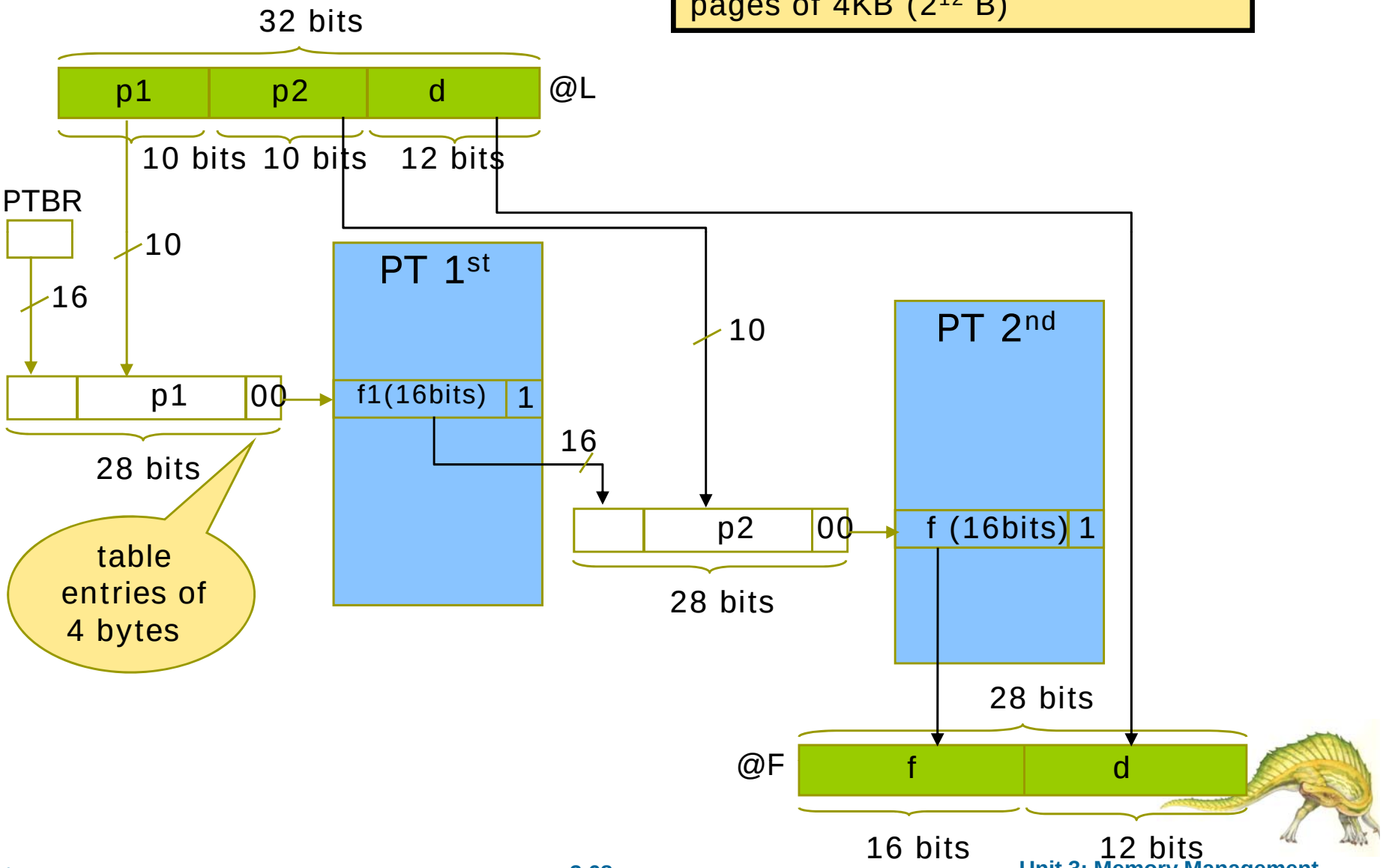
- Saving space in 2nd level PT
  - 1st level PT has to be in memory
  - only valid 2nd level PT occupy memory
- Distinguish between
  - data frames
  - PT frames



Two-Level Paging Example

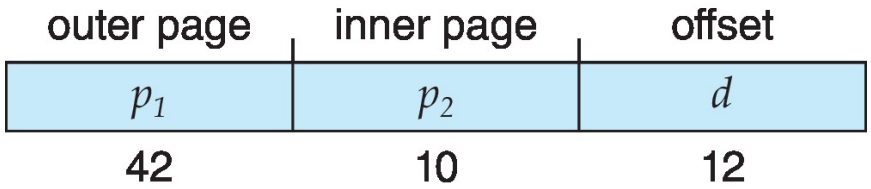
- In-depth translation example:

Consider logical space 4GB (2^{32} B),
physical sp 256Mb (2^{28} B),
pages of 4KB (2^{12} B)

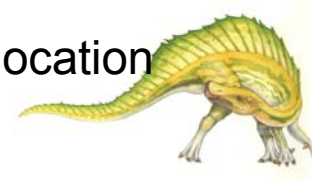


64-bit Logical Address Space

- Even two-level paging scheme not sufficient if large logical space
  - 64-bit logical address $\Rightarrow 2^{64}$ bytes = 16 Exabytes!!!! Logical space
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries (and assume 4-byte entries)
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries (and 2^{44} bytes size!!!)
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size (16 GB!!!)
    - And possibly 4 memory access to get to one physical memory location



Three-level Paging Scheme

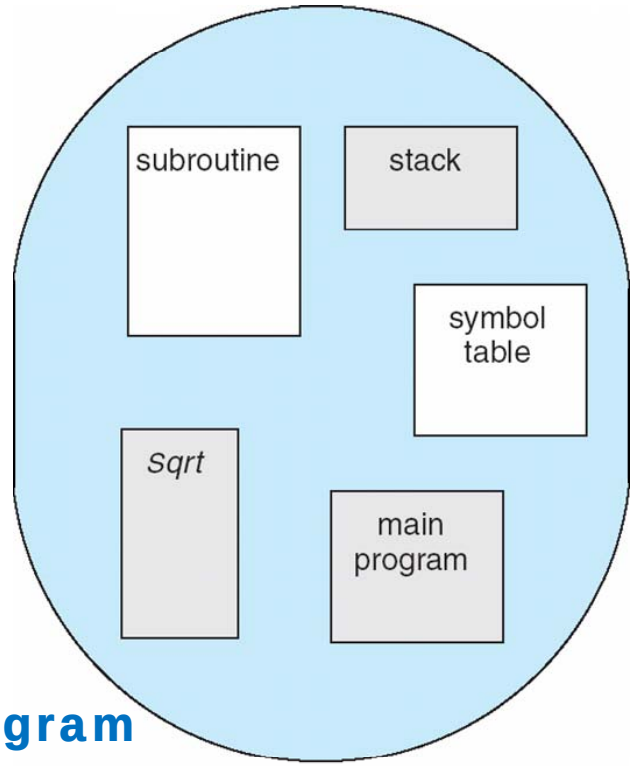
outer page	inner page	offset
p_1	p_2	d
42	10	12

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

Segmentation

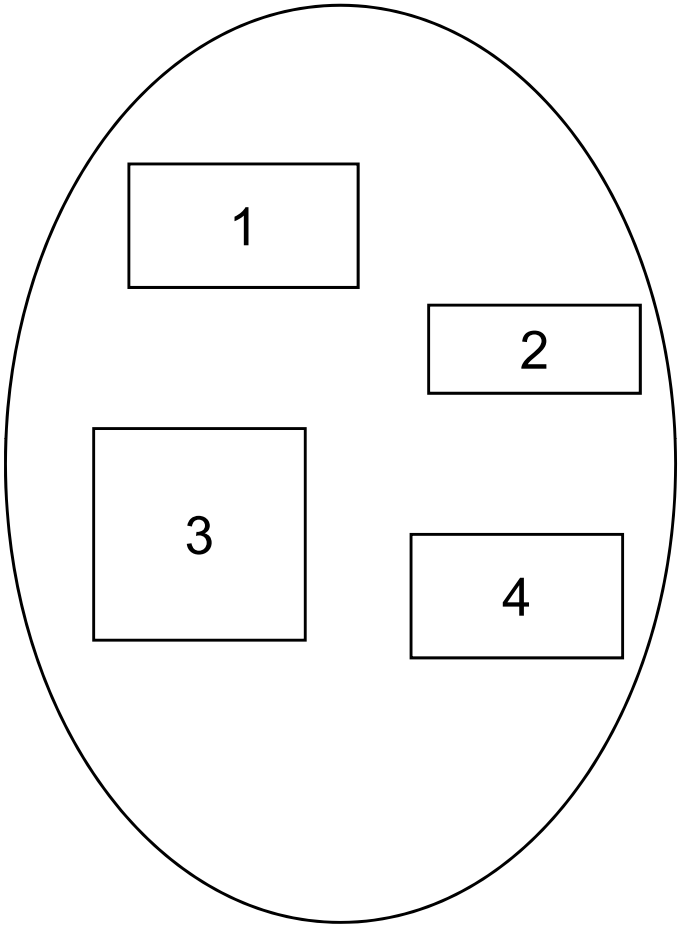
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

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



User's view program

Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>

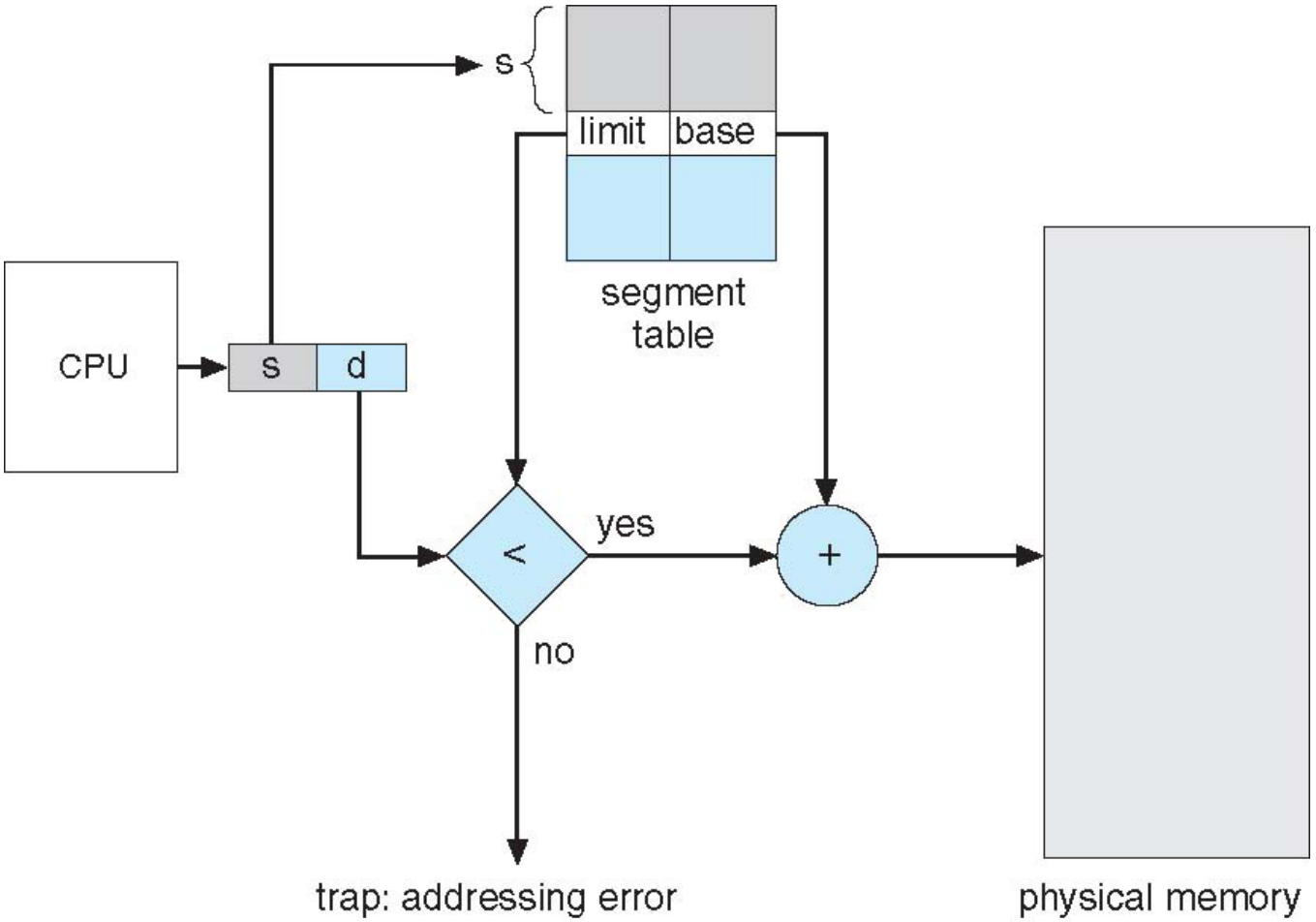
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

Segmentation Architecture (Cont.)

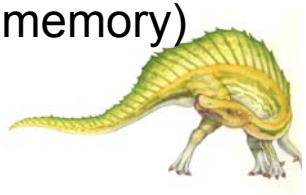
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation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

Segmentation Hardware



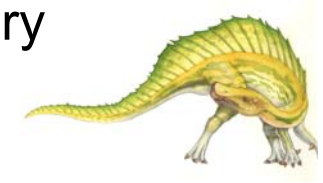
Paging vs. Segmentation

- Size of address base
  - Pag. : only page number (f). combine with offset (d)
  - Seg.: complete physical address SUM to d
- Overlap (shared memory)
  - Pag.: complete pages may overlap
  - Seg.: partial overlap of physical segments may occur
- Memory allocation
  - Pag.: simple:
    - process requires n pages → look for n holes (free frames)
    - No external frag. , yes internal frag. (average P/2 per process)
  - Seg: more complicated:
    - process requires n segments, each one in size → search n holes of appropriate size which may imply garbage collection (compaction of memory)
    - No internal frag. yes external fragmentation

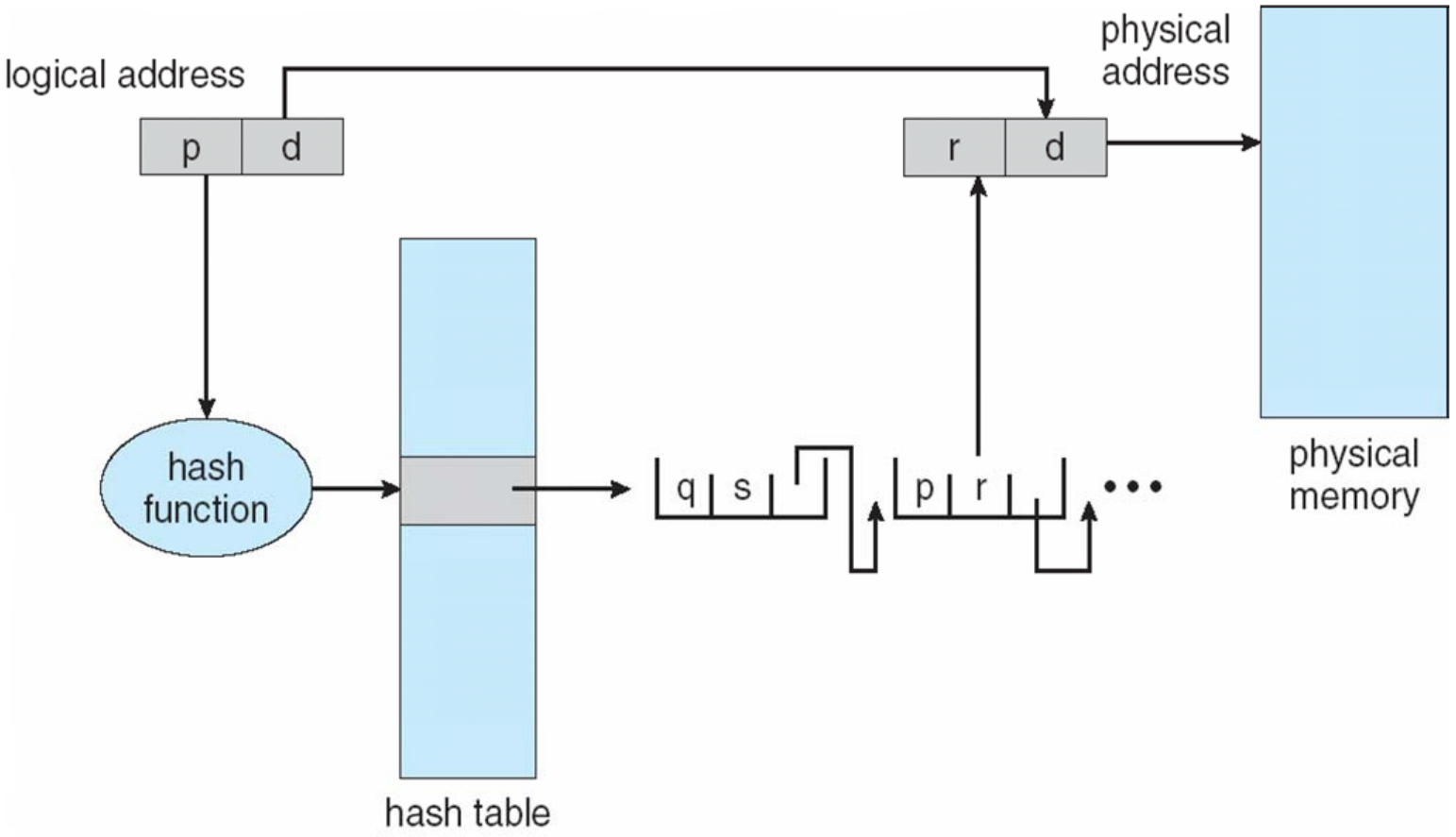


Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



Hashed Page Table



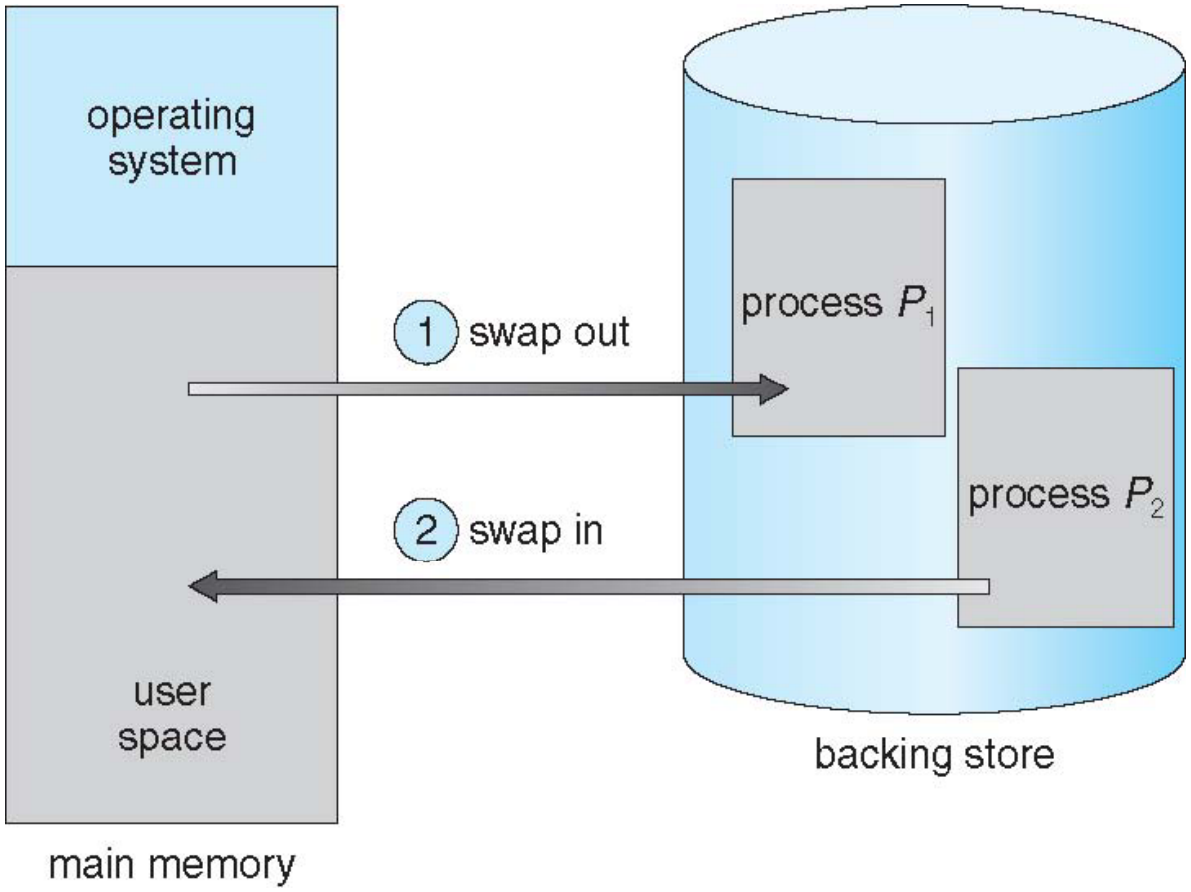
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Swapping with Paging