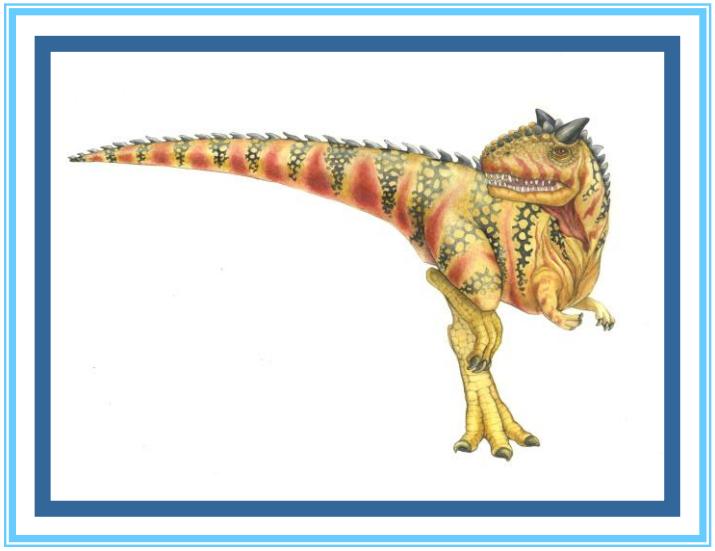
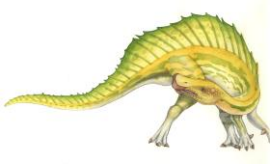
Chapter 8: Main Memory



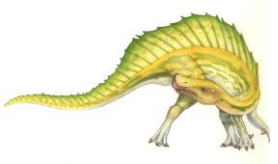
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Examples:
  - The Intel 32 and 64-bit Architectures
  - ARM Architecture



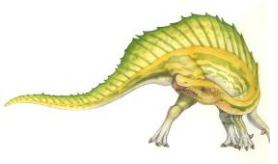
Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging



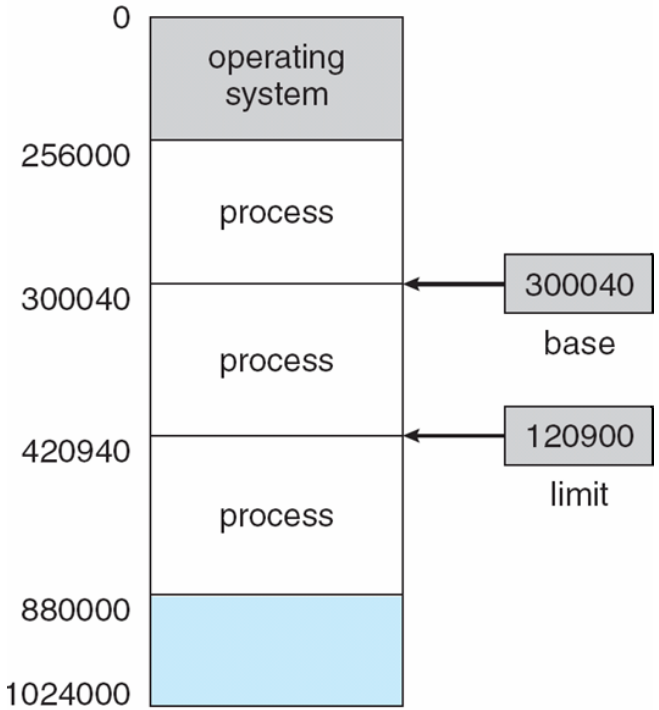
Background

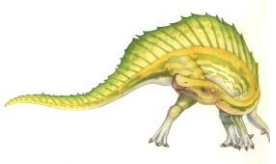
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
  - Memory unit only sees a stream of addresses + read requests, or address + data and write requests
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



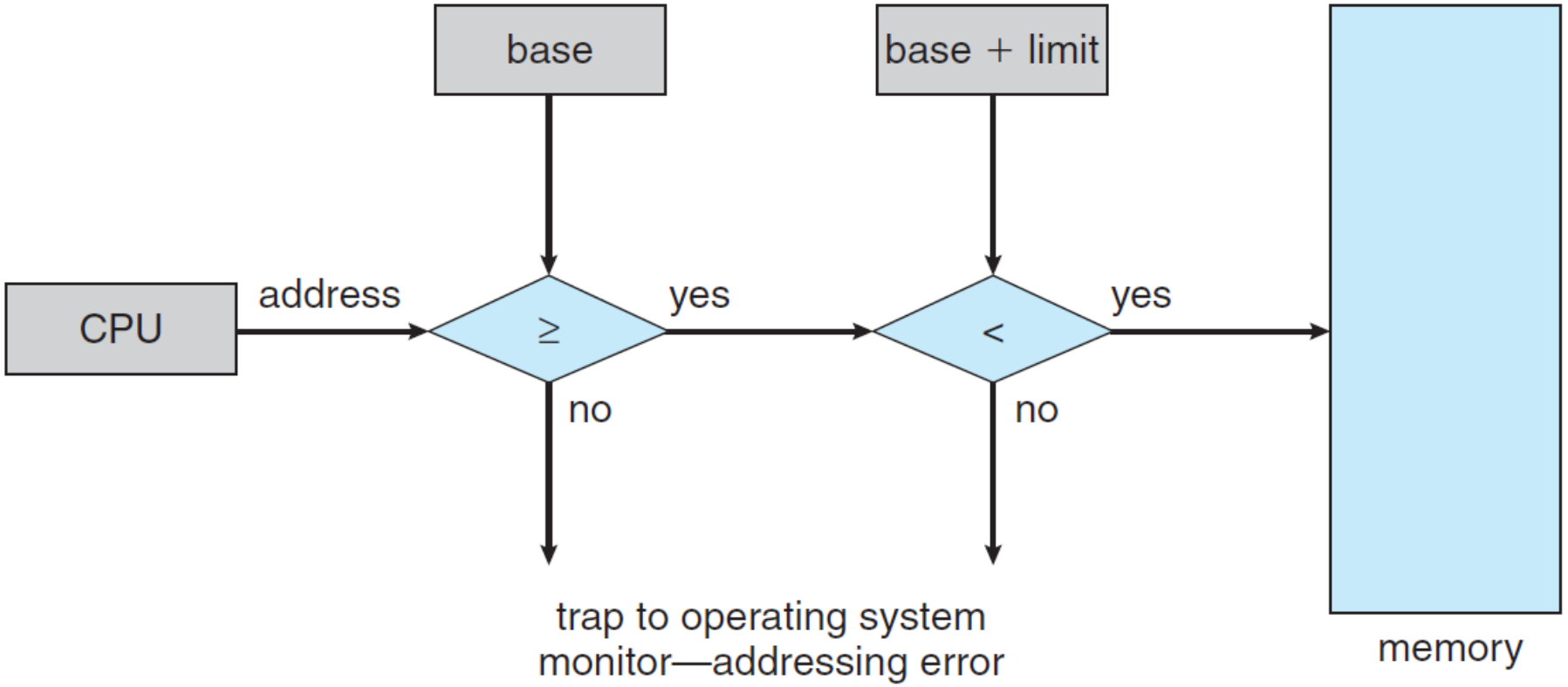
Base and Limit Registers

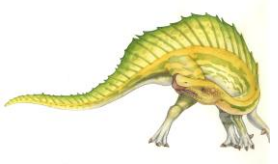
- A pair of **base** (relocation) and **limit registers** define the logical address space
- CPU must check, every memory access generated in user mode, to be sure it is between base and limit for that user





Hardware Address Protection with Base and Limit Registers





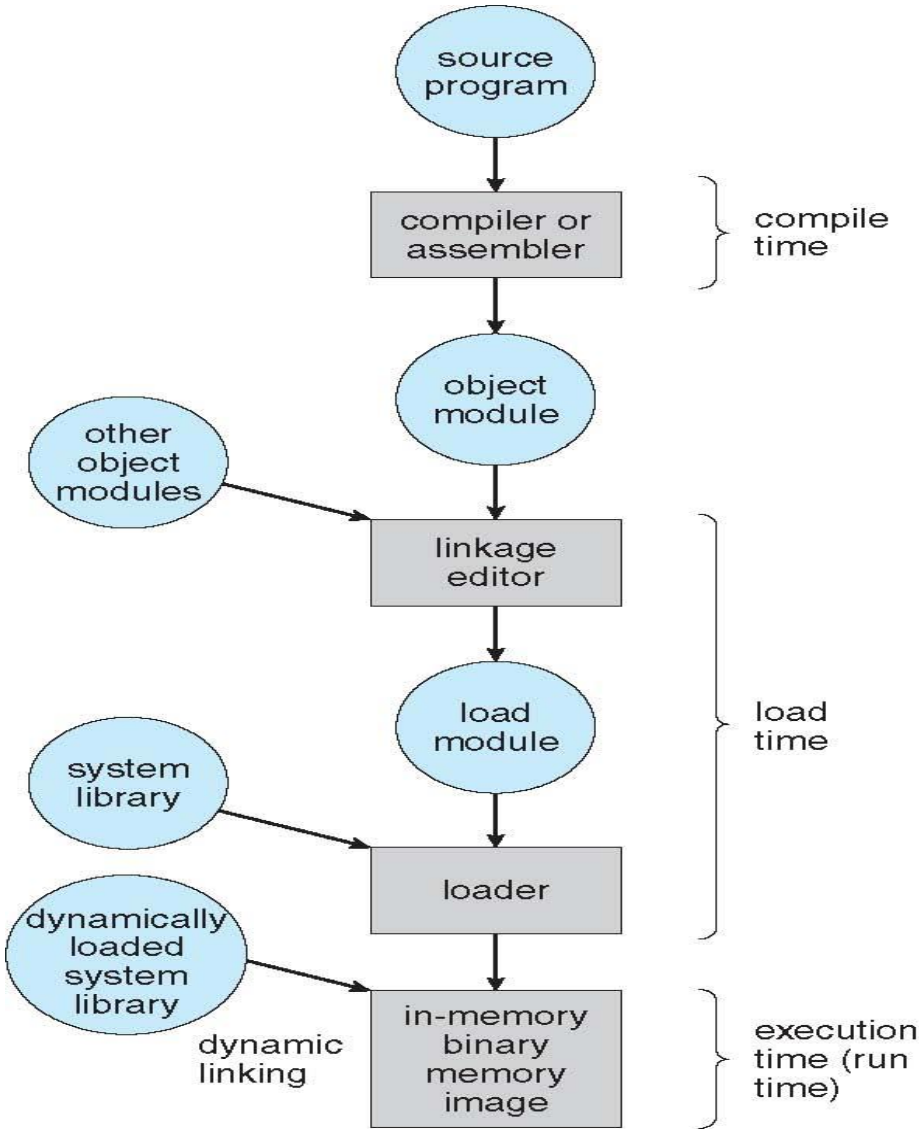
Address Binding

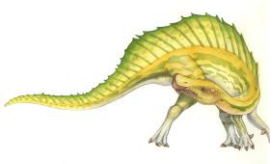
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different 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)
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

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

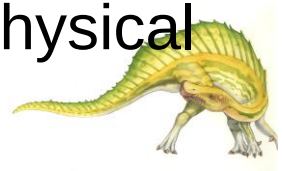
Multistep Processing of a User Program





Logical vs. Physical Address Space

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

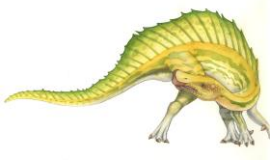


Memory-Management Unit (MMU)

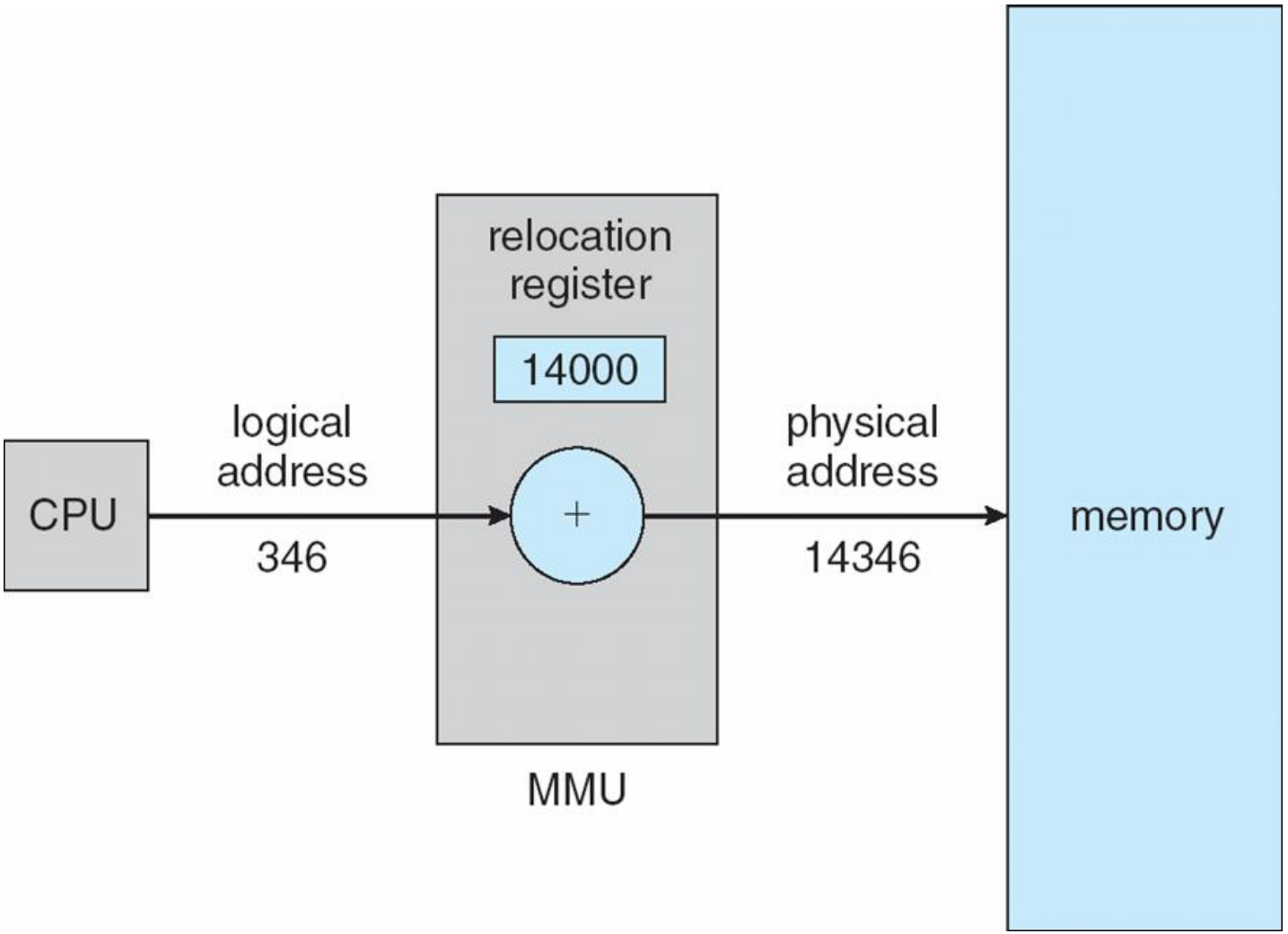
- Hardware device that at run time maps logical (virtual) to physical address
  - Many methods possible, covered in rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- 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

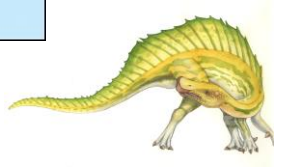
Dynamic Relocation Using a Relocation Register

- 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 Relocation Using a Relocation Register



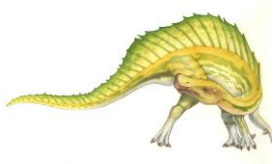


Dynamic Linking

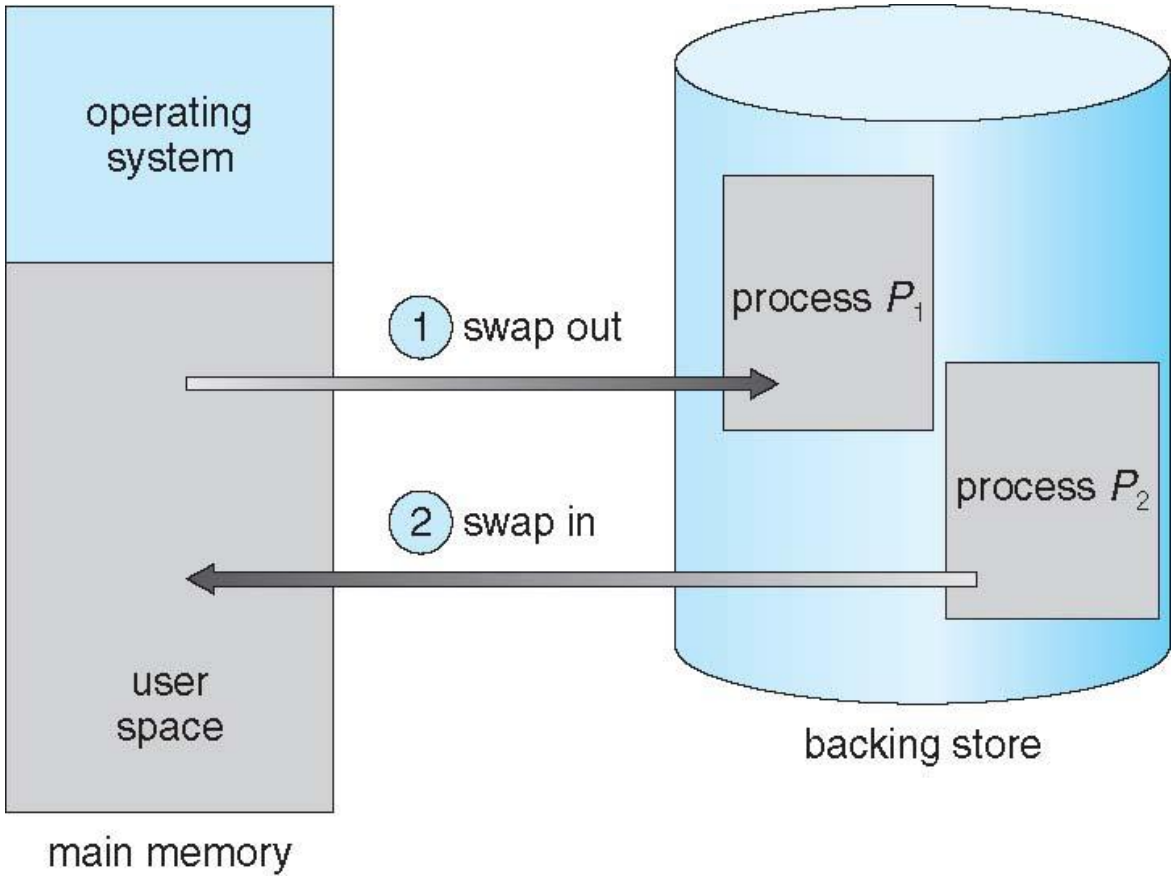
- **Static linking** – system libraries combined by the loader into the binary program image
- Dynamic linking – linking of system libraries postponed until execution time
  - Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the 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
  - This system is also known as **shared libraries**
  - This feature can be extended to library updates (such as bug fixes) that may be replaced by a new version

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



Schematic View of Swapping

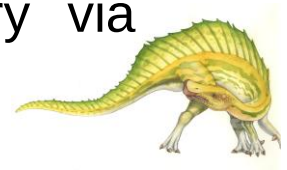


Swapping

- 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** consisting of all ready-to-run processes whose memory images are on backing store or in memory
- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled and started only if more than threshold amount of memory allocated
  - Then disabled again once memory demand reduced below threshold

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
  - 100 MB process swapping to hard disk with transfer rate of 50 MB/sec
    - Swap out time of 2000 ms
    - Plus swap in of same sized process
    - Total context switch swapping component time of 4000 ms
  - Can reduce time through decreasing size of memory swapped – by knowing how much memory really being used
    - System calls to inform OS size of used memory via `request_memory()` and `release_memory()`

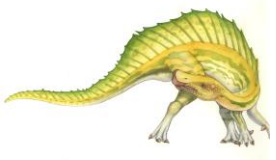


Swapping

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, but adds overhead
- Standard swapping not used in modern OSs
  - But modified version common
  - Swap only when free memory extremely low
- Swapping not supported on mobile systems
  - Instead use other methods to free memory if low space
    - Android terminates apps if low free memory
    - iOS ***asks*** apps to voluntarily release allocated memory

Contiguous Allocation

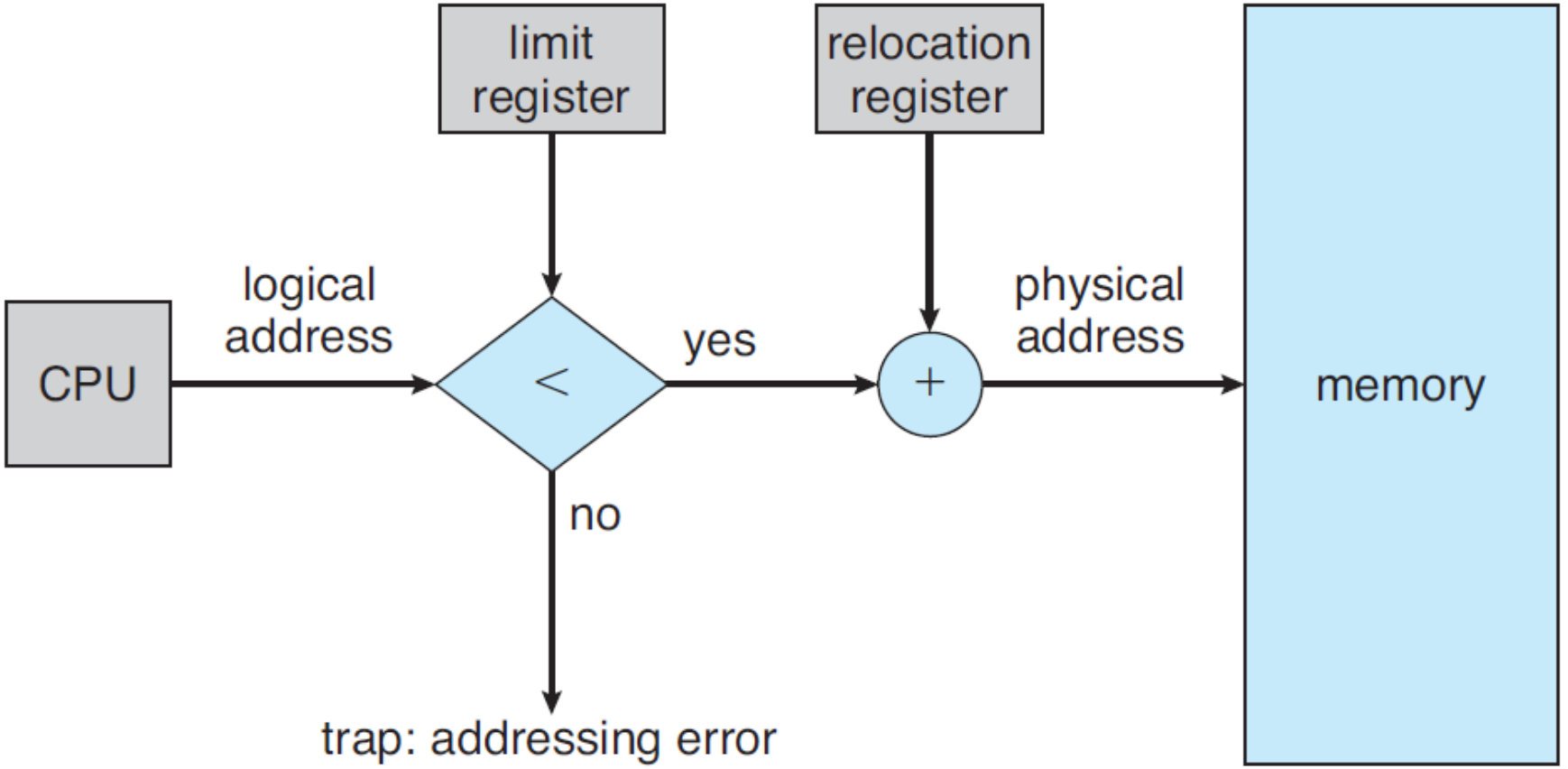
- Main memory must accommodate both OS and various user processes
  - Must to allocate main memory in most efficient way
  - Contiguous allocation is one early method
- Main memory usually divided into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in a single contiguous section of memory

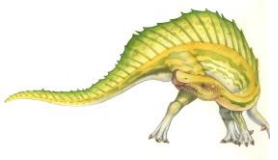


Contiguous Allocation

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base 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
  - MMU maps logical address *dynamically* by adding the value in the base register
  - As part of context switch, dispatcher loads base and limit registers values of a process under execution
- This scheme provides an effective way to allow operating system's size to change dynamically.
  - Suitable if an OS service is not commonly used
  - Such services code is called **transient** code

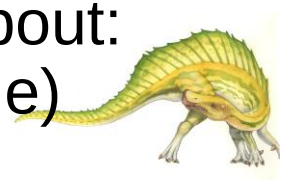
Hardware Support for Relocation and Limit Registers



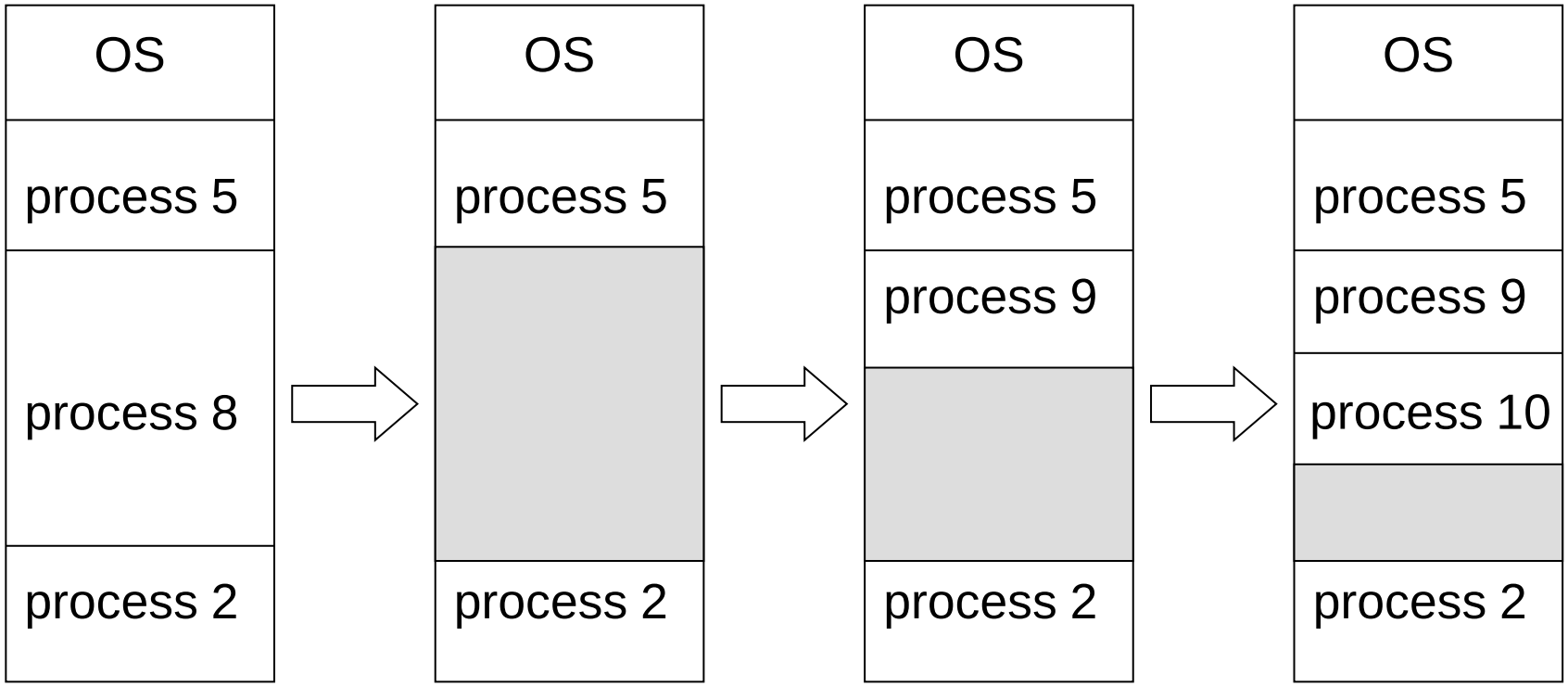


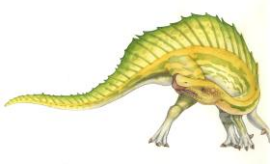
Contiguous Allocation

- Memory is divide into several partitions – **multiple-partition method**
  - 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 and adjacent free partitions combined
  - Operating system maintains information about: a) allocated partitions b) free partitions (hole)



Contiguous Allocation



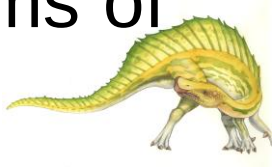


Dynamic Storage-Allocation Problem

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

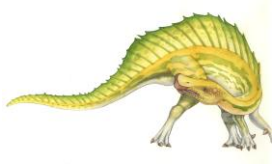
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list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

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



Fragmentation

- **External Fragmentation** – total memory space is enough to satisfy a request, but it is not contiguous
  - Storage is fragmented into a large number of small holes
  - First-fit and best-fit strategies for memory allocation suffer from external fragmentation
  - First fit analysis reveals that given *N* blocks allocated, 1/2 *N* blocks lost to fragmentation
    - 1/3 of memory may be unusable ⇒ **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used



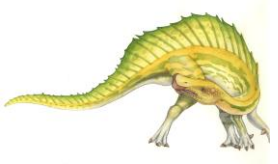
Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
    - Relocation requires just moving the process and then changing the base register to reflect the new base address
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

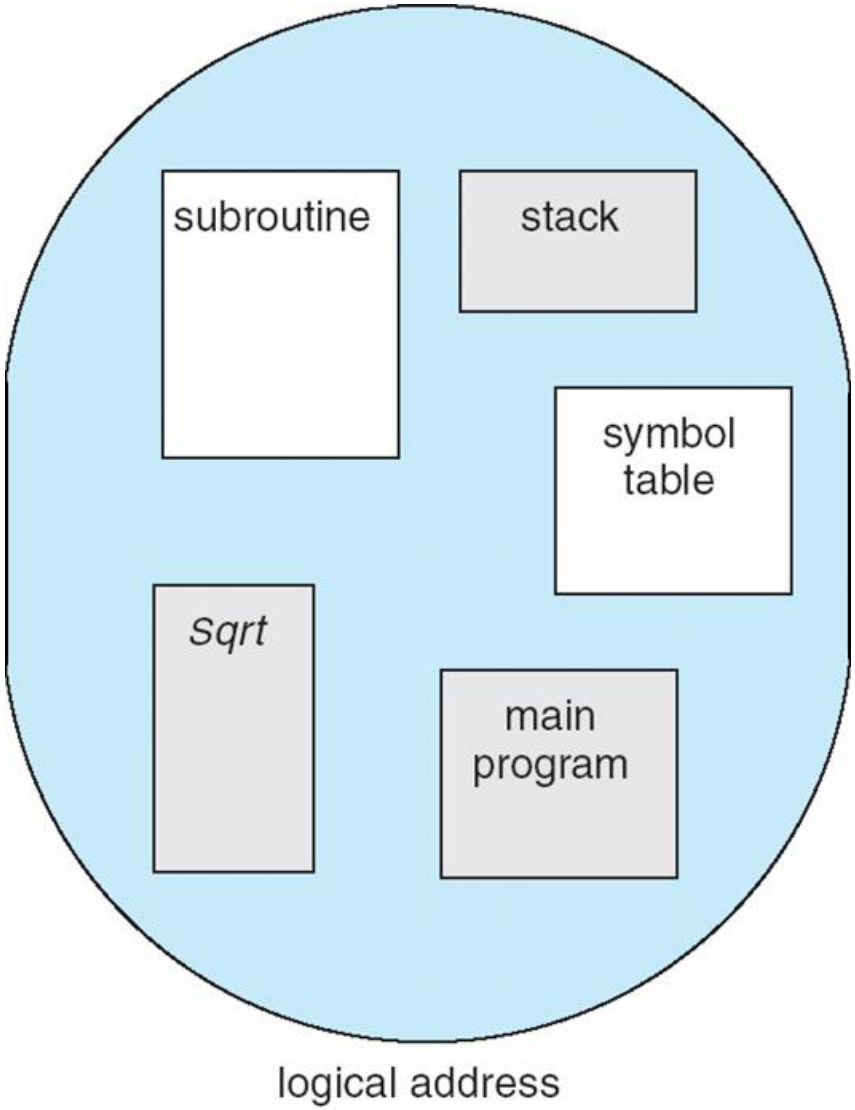
Segmentation

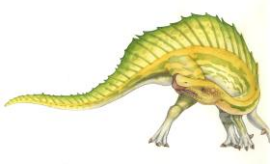
- Memory-management scheme that mapped the programmer's view of memory
- A program is a collection of segments and each segment is a logical unit, such as:

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

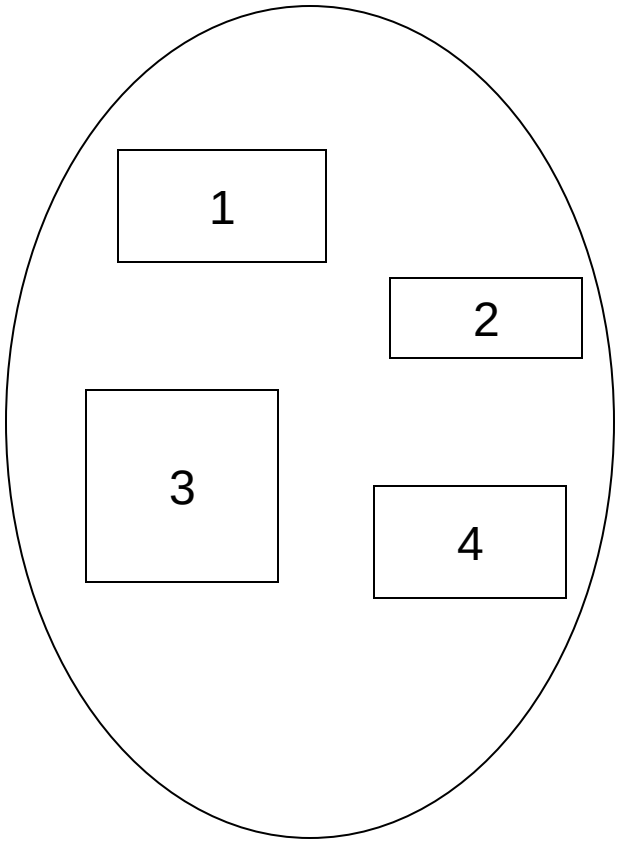


Programmer's View of a Program





Logical View of Segmentation



Logical address space

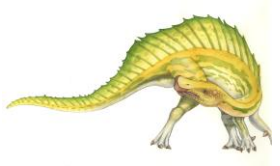
Physical memory space

Segmentation Architecture

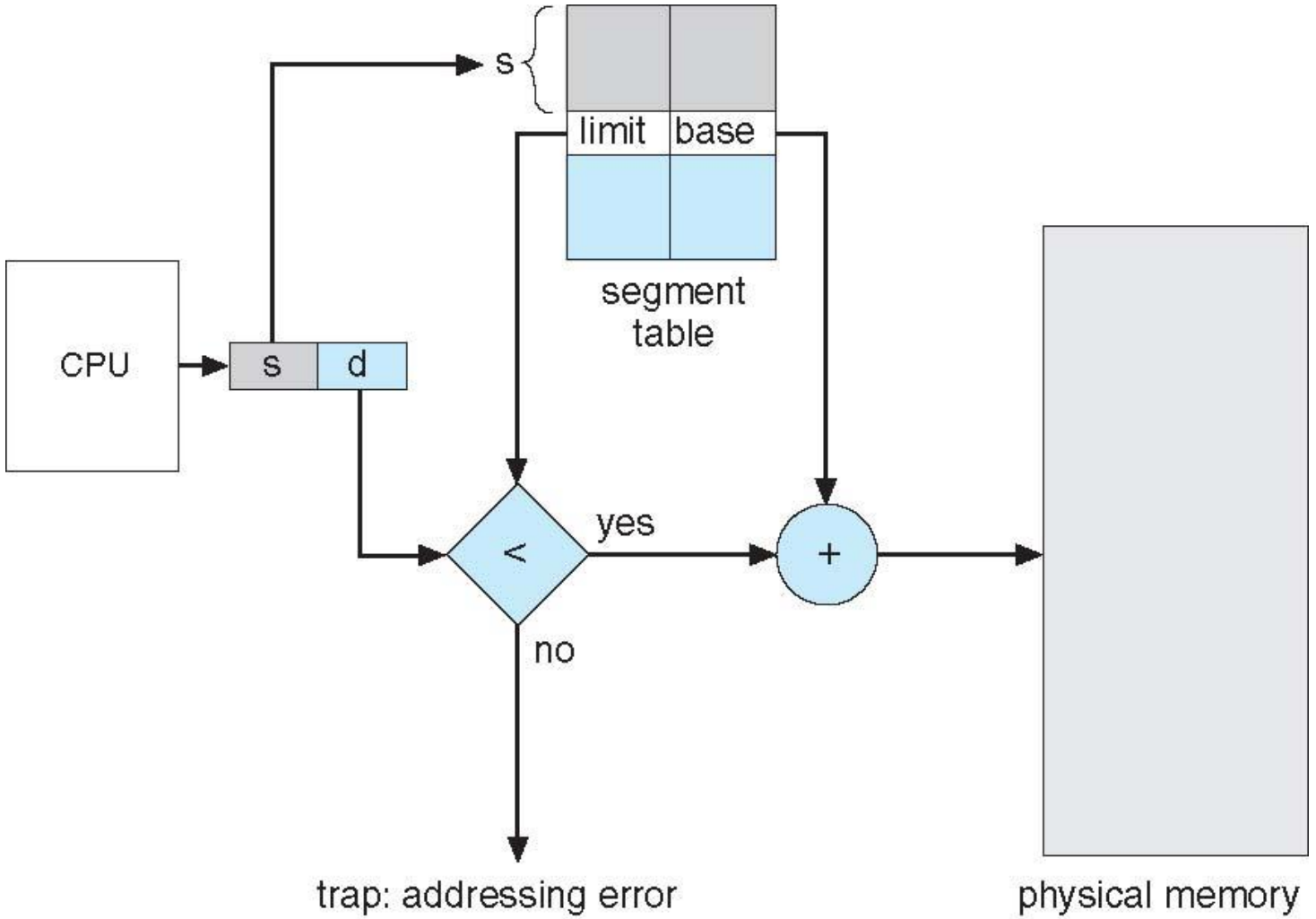
- Logical address consists of a two tuple:

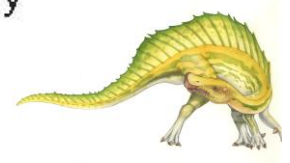
  <segment-number, offset>

- A **Segment table** is used to map the two-dimensional logical addresses of each process into one-dimensional physical addresses
  - Each entry in the segment table has:
    - **base** – contains the starting physical address where the segment reside in memory
    - **limit** – specifies the length of the segment
- Hardware support:
  - **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 process;
    - segment number ***s*** is legal if ***s*** < **STLR**



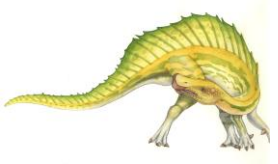
Segmentation Hardware



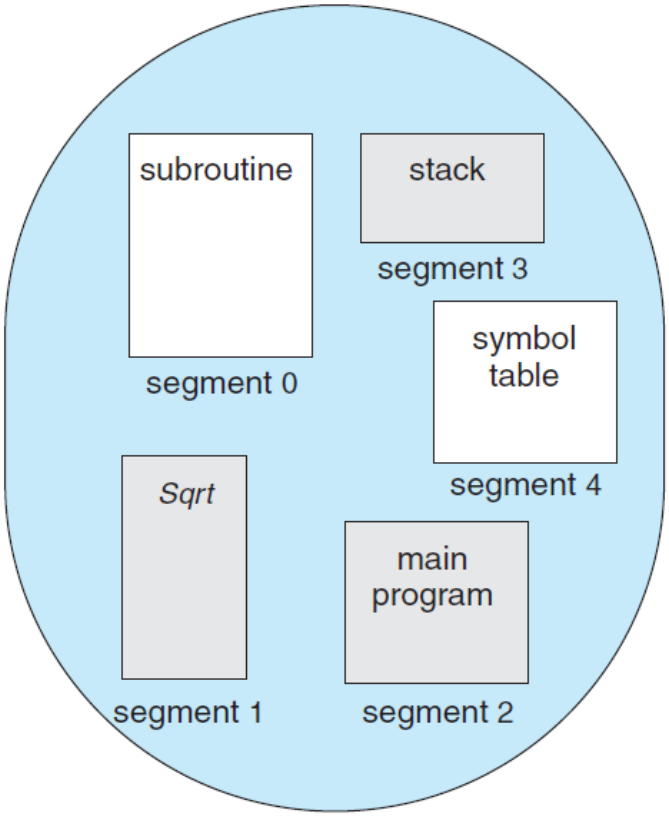


Segmentation Architecture

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

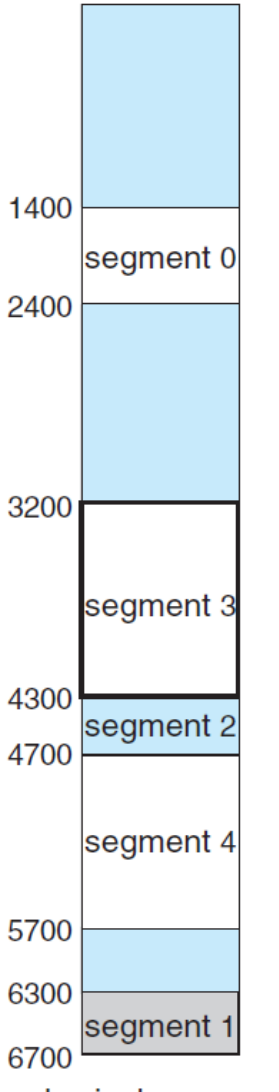


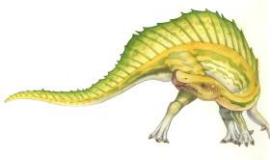
Example of Segmentation



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

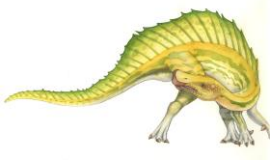
segment table





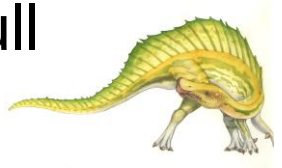
Paging

- 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
  - Normally, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**



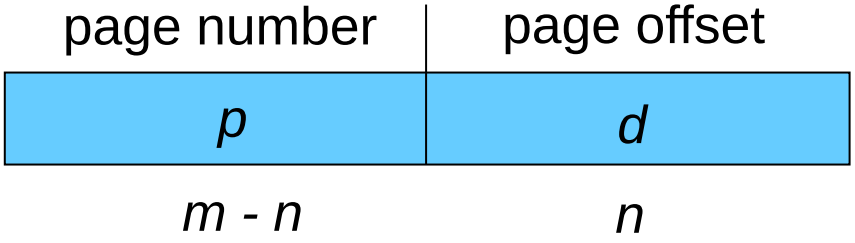
Paging

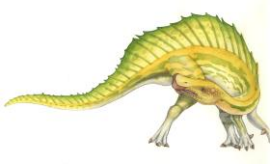
- Keep track of all free frames
- To execute a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical addresses to physical addresses
  - Map the corresponding physical frame for each logical page
- Backing store likewise split into pages
- Still have Internal fragmentation
  - Last frame allocated may not be completely full



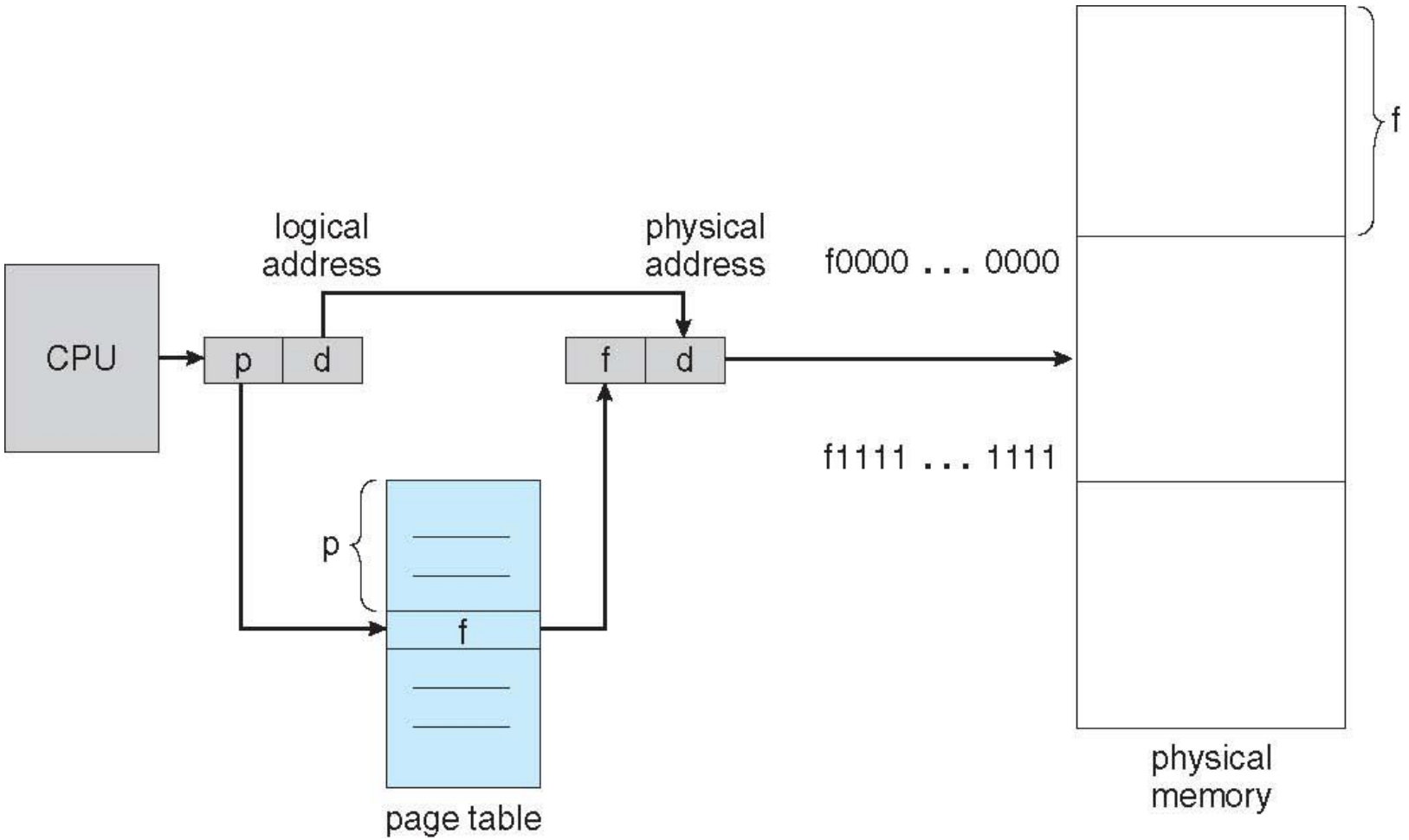
Address Translation Scheme

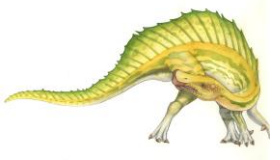
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space 2^m and page size 2^n:



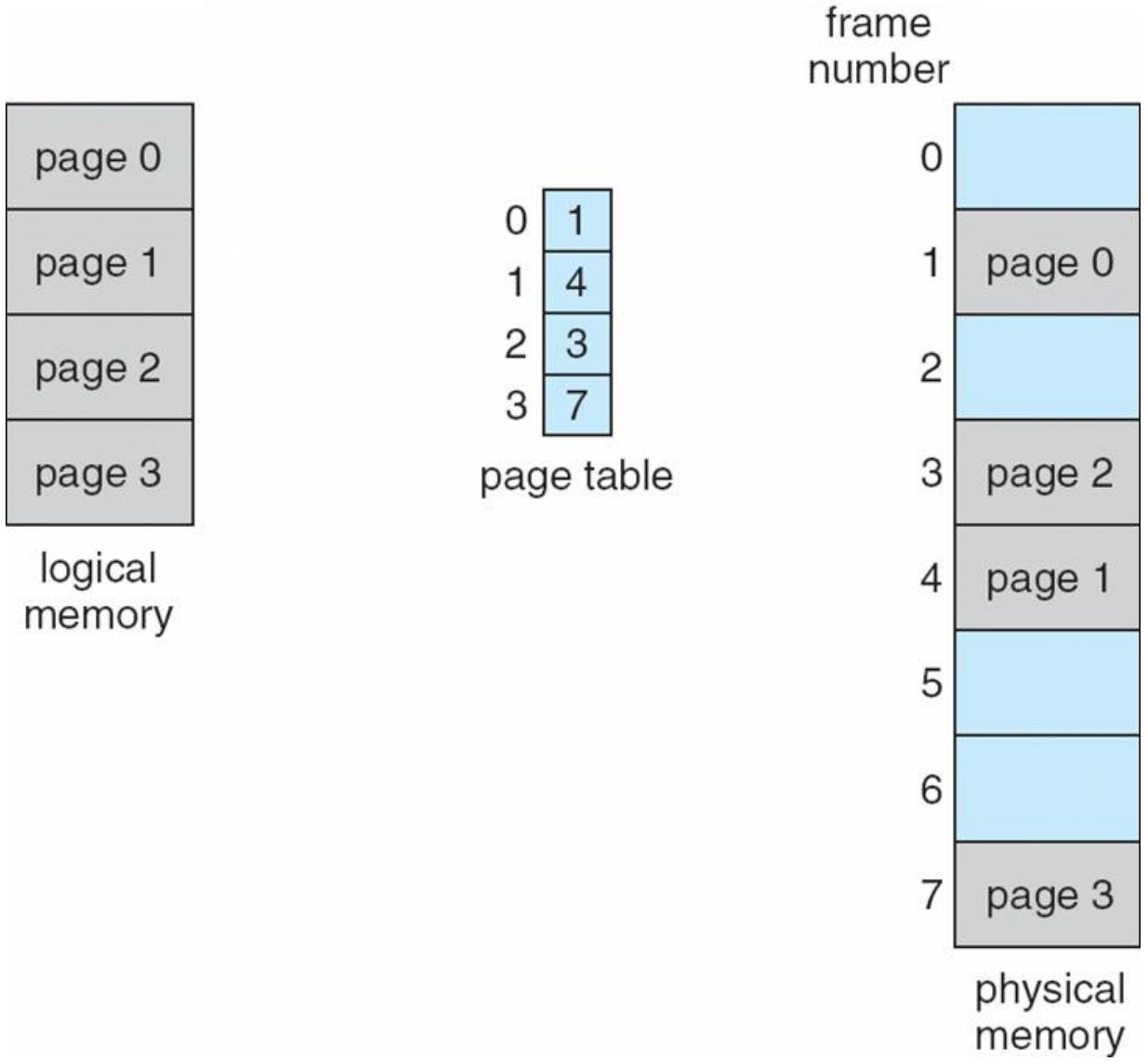


Paging Hardware

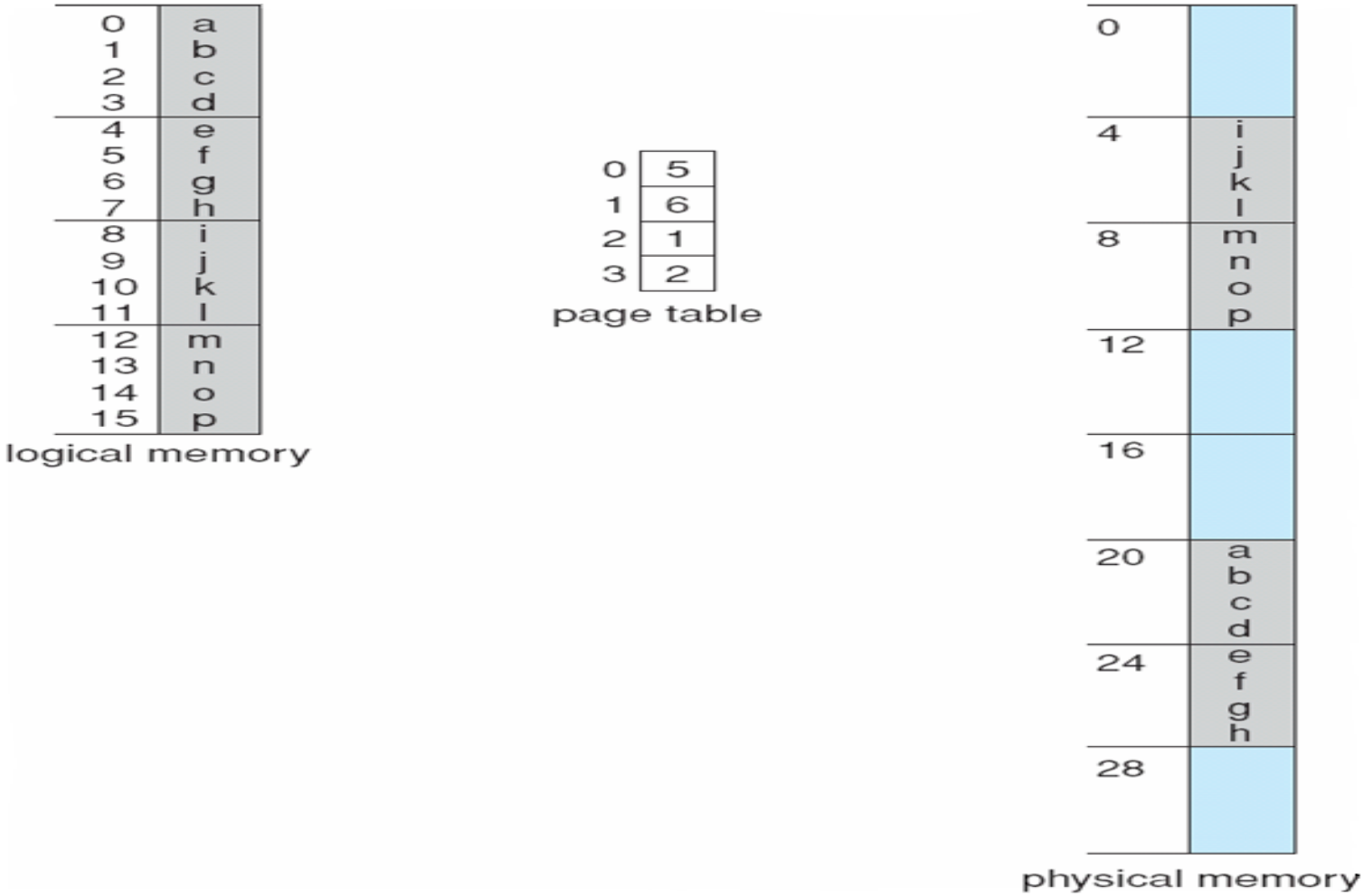




Paging Model of Logical and Physical Memory



Paging Example

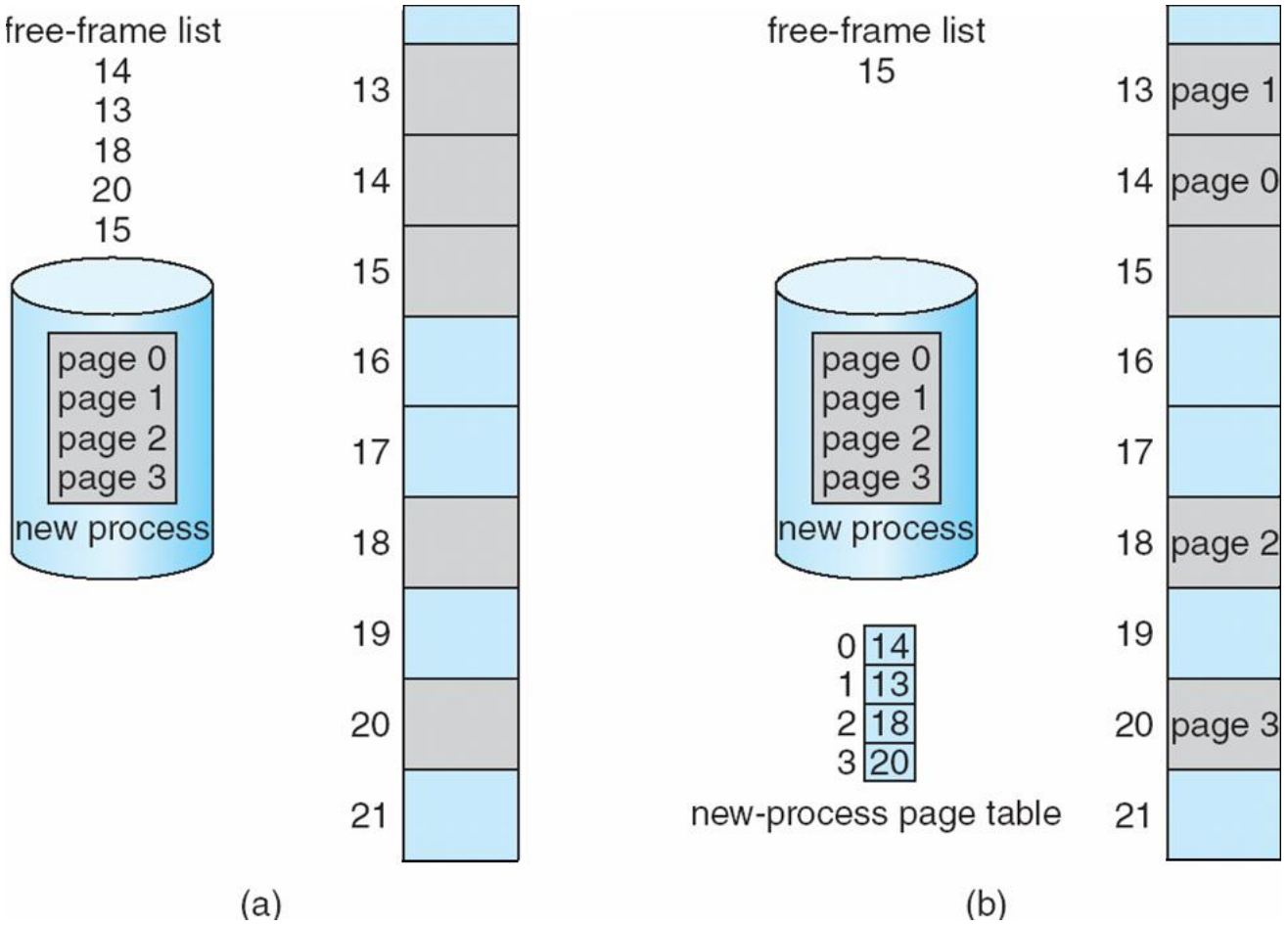


A 32-byte memory with 4-byte pages (*n*=2 and *m*=4)

Paging

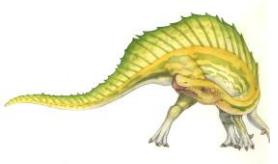
- Process view and physical memory now very different
- By implementation, process can only access its own memory
- Calculating internal fragmentation:
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst 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

Free Frames



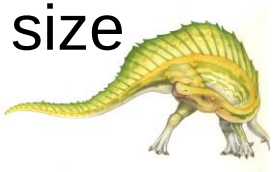
Before allocation

After allocation



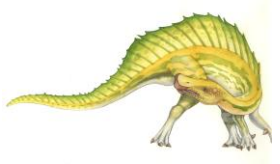
Implementation of Page Table

- Dedicated registers are used for implementing the page table if the table is reasonably small
  - In DEC PDP-11, the address consists of 16 bits, and the page size is 8 KB.
  - The page table thus consists of eight entries that are kept in fast registers.
- Most computers allow the page table to be very large; therefore, the 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



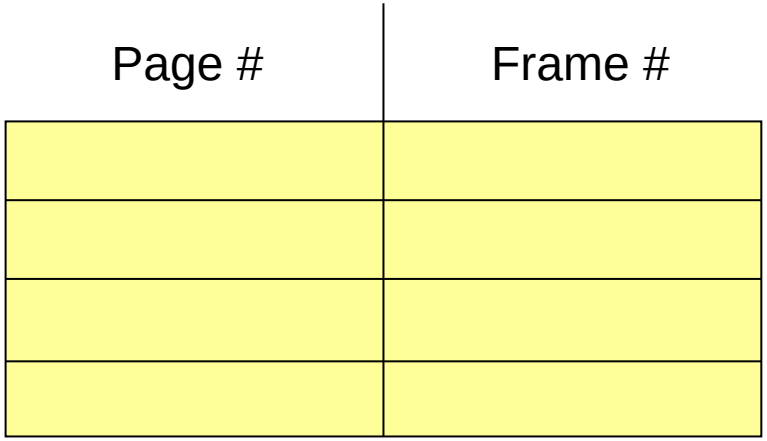
Implementation of Page Table

- In the last 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 using a special fast lookup memory cache called **translation look-aside buffer** (**TLB**)
  - Each entry in TLB consists of two parts: a page no. as a key and a corresponding frame value
  - An item is compared with all keys simultaneously
  - If a page number is not in the TLB (a **TLB miss**), its frame is added for faster access next time
    - Replacement policies must be considered

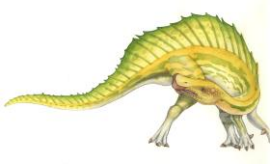


Associative Memory

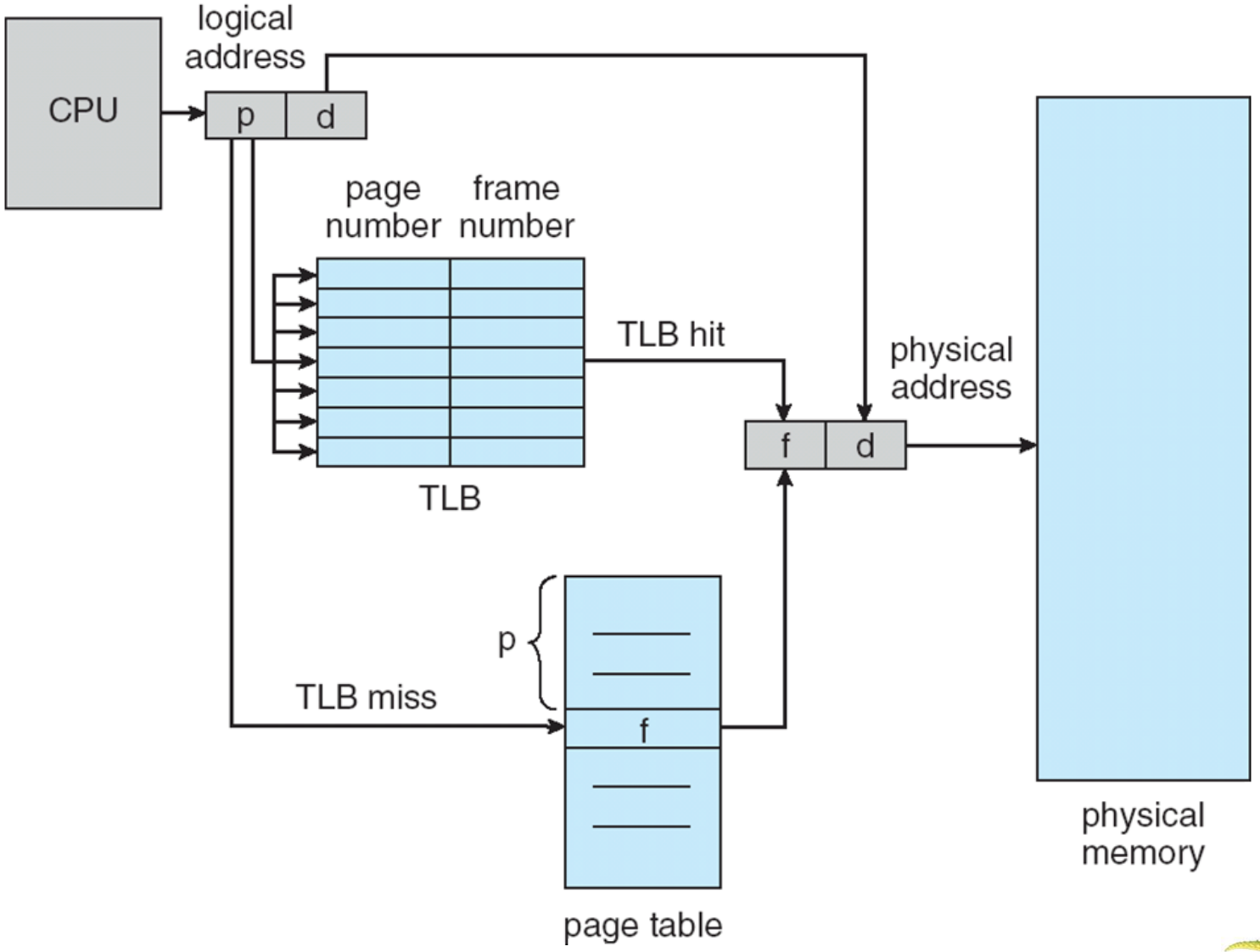
- Associative memory – parallel search



- 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

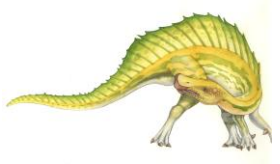


Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (**EAT**)

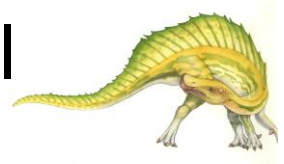
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

  - Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
  - Consider more realistic hit ratio α = 99%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

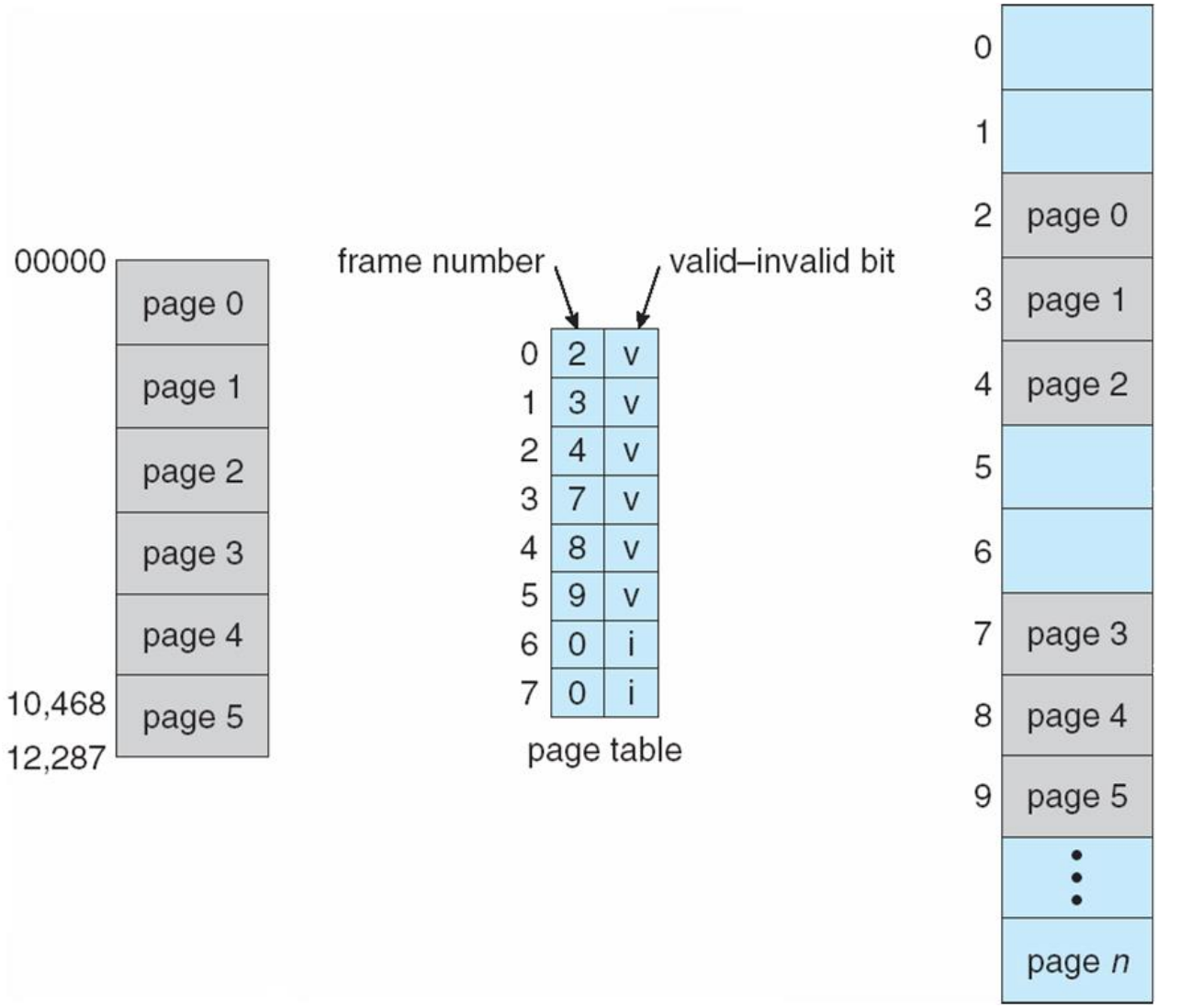


Memory Protection

- 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 it is 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

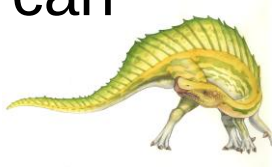


Valid (v) or Invalid (i) Bit In A Page Table

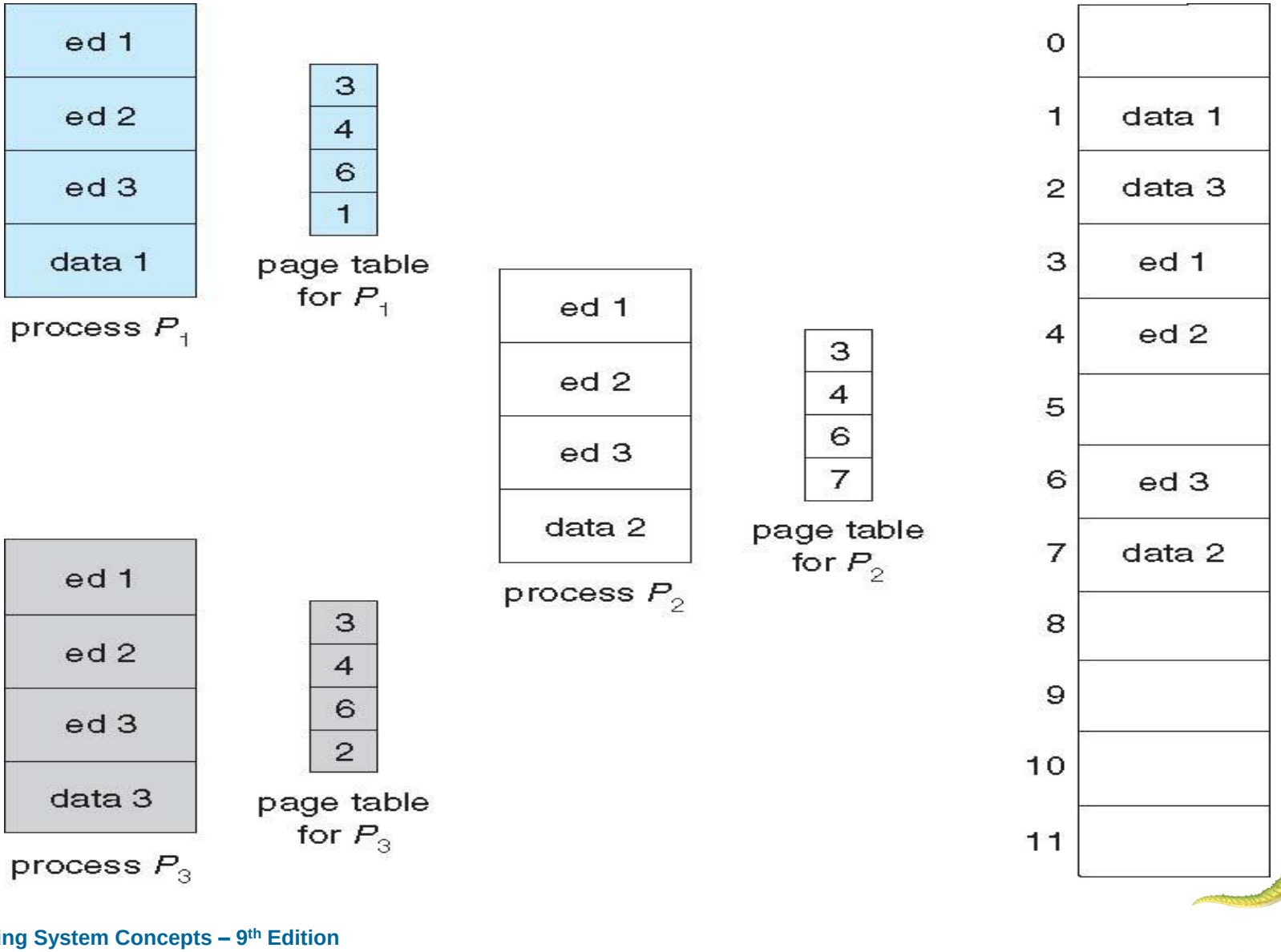


Shared Pages

- **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

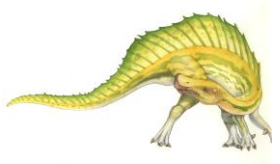


Shared Pages Example



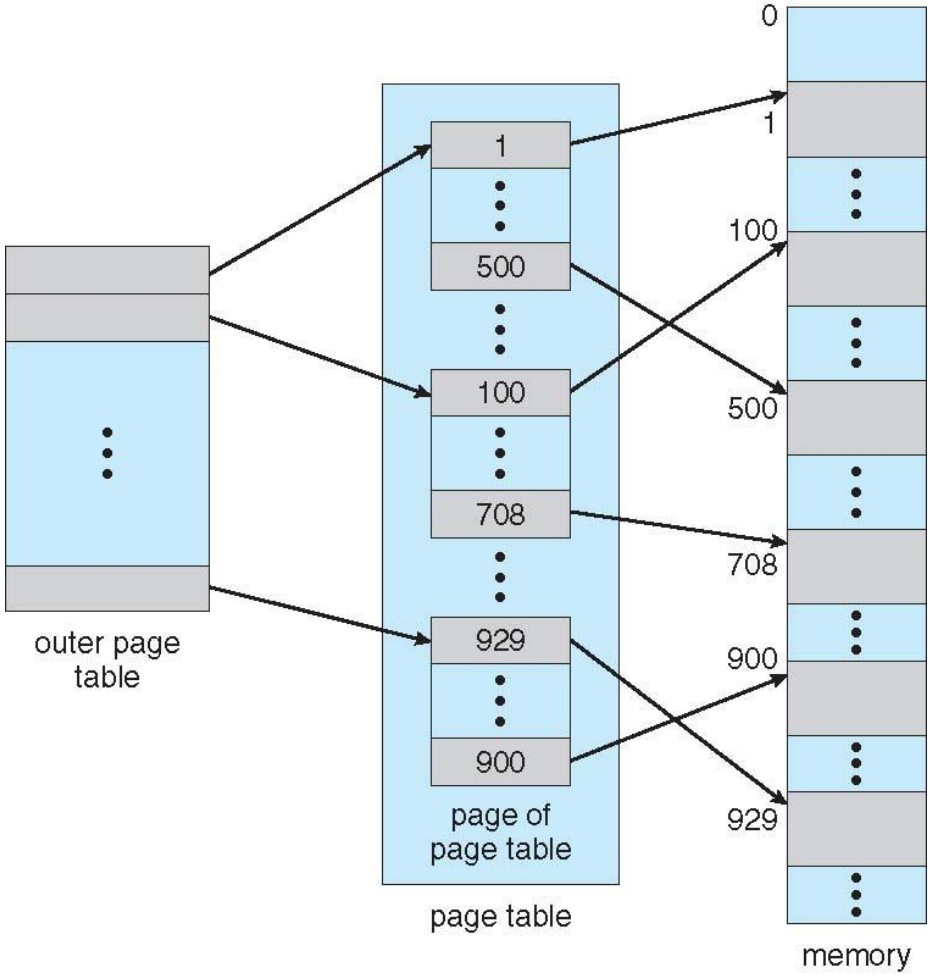
Structure of the Page Table

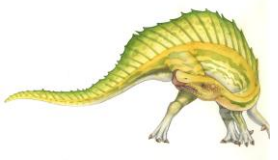
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes => 4 MB of physical address memory space for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Alternative methods:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



Hierarchical Page Tables

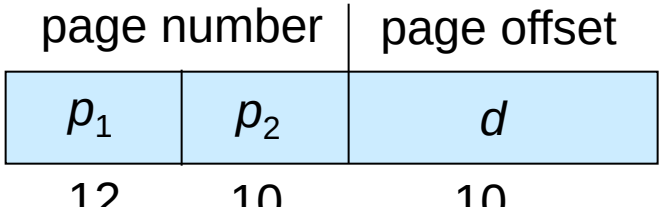
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

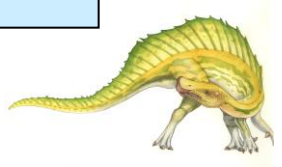




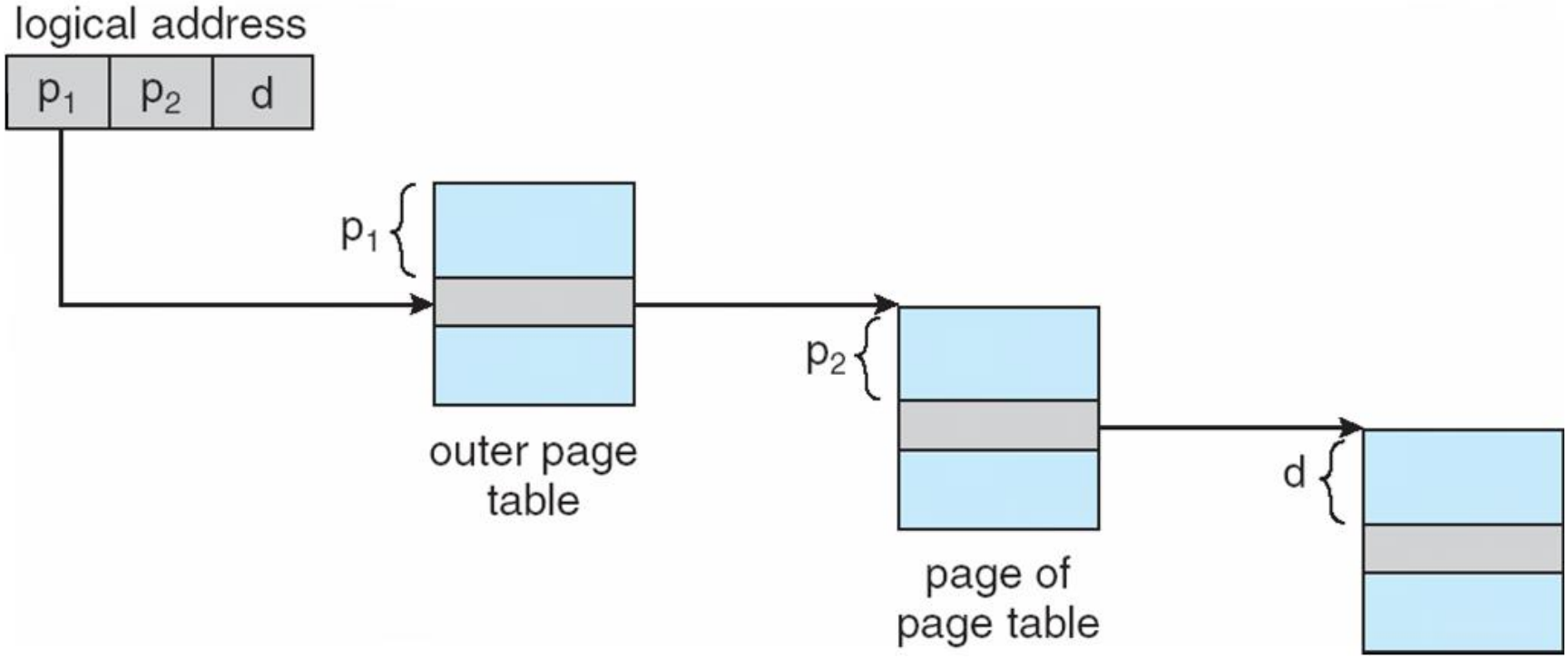
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, then the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:
  - where p_1 is an index into outer page table, and p_2 is the displacement within the page of inner page table





Address-Translation Scheme

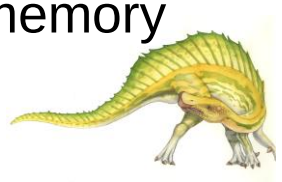


64-bit Logical Address Space

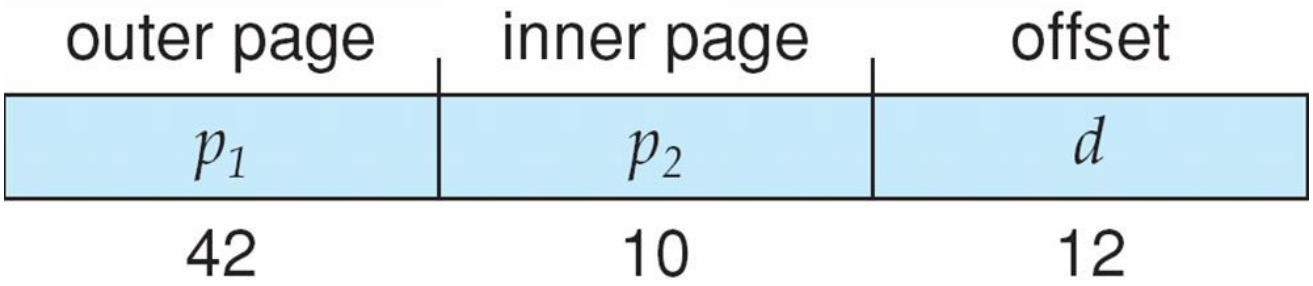
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like:

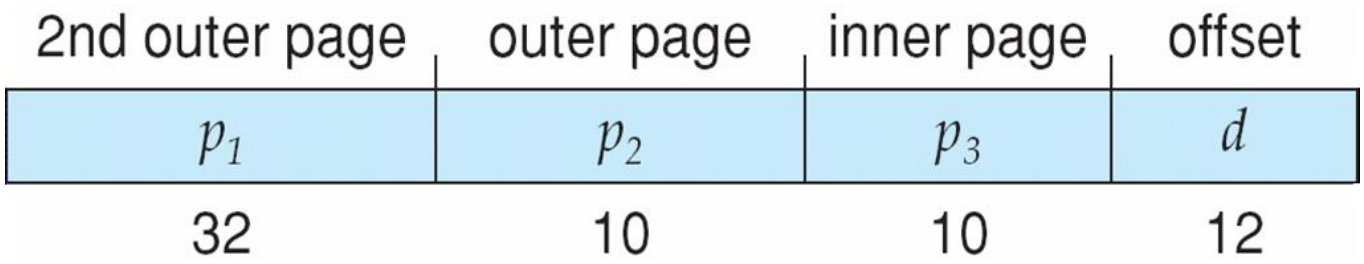
outer page	inner page	page offset
p_1	p_2	d
42	10	12

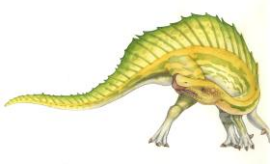
  - Outer page table has 2^{42} entries
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{32} bytes in size
    - And possibly 4 memory access to get to one physical memory location



Three-level Paging Scheme



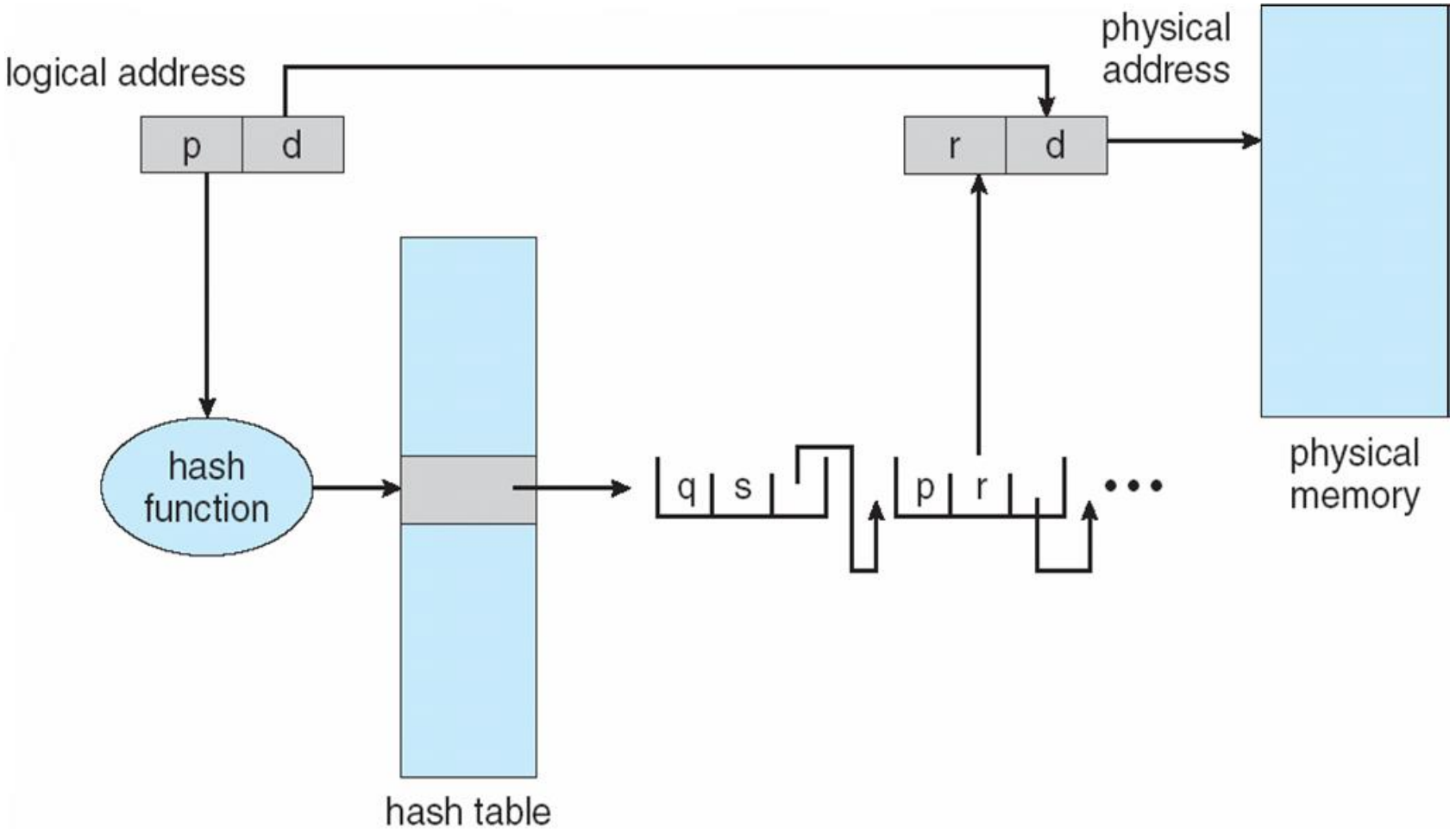


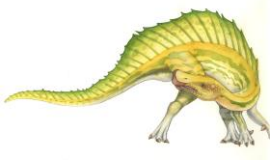


Hashed Page Tables

- 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
  - When a match is found, the corresponding physical frame is extracted

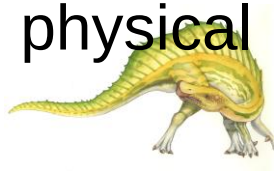
Hashed Page Table



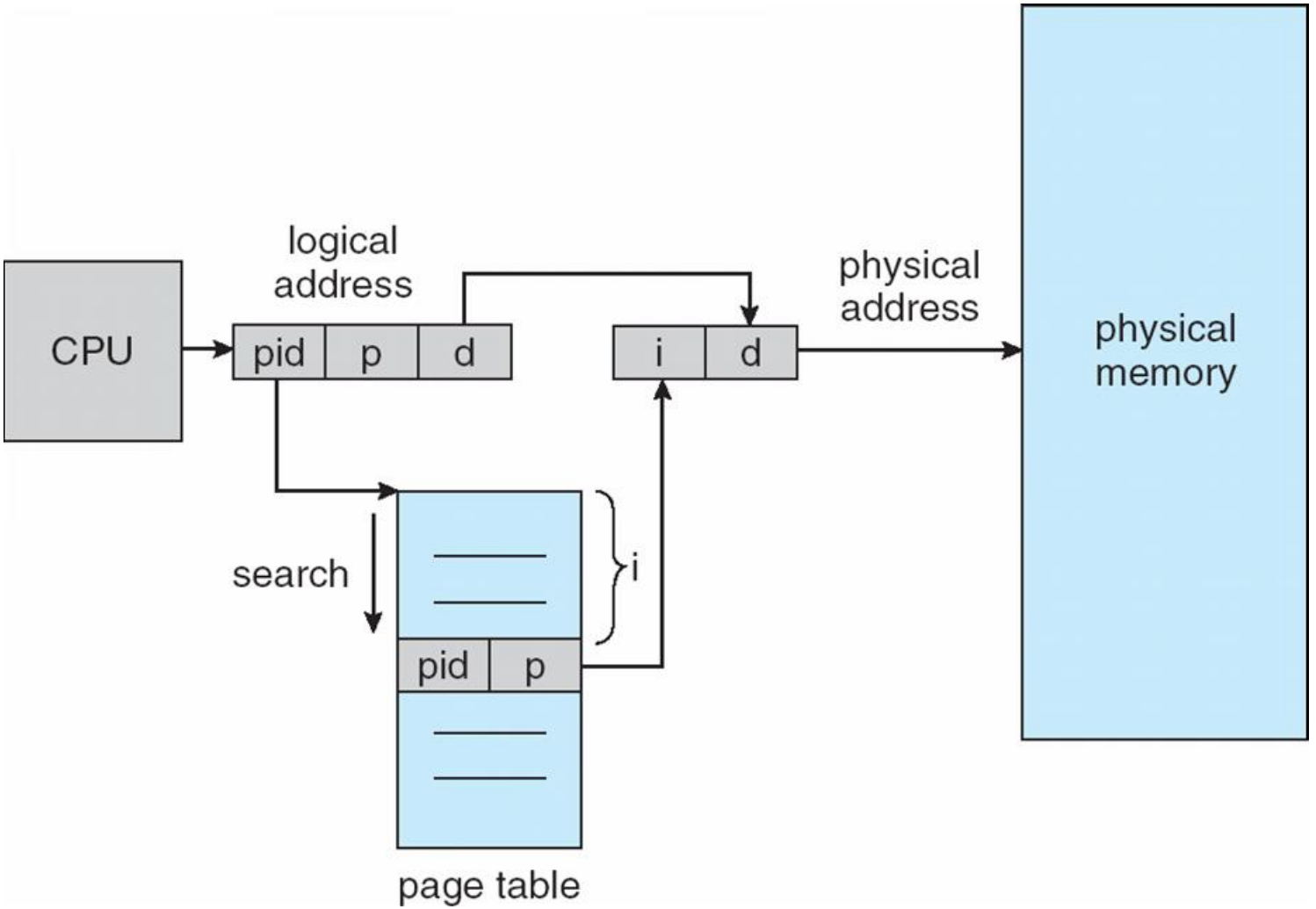


Inverted Page Table

- 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 (frame) of memory
  - Each 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 search to one, or at most a few, page-table entries and use TLB to accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address



Inverted Page Table Architecture



End of Chapter 8