

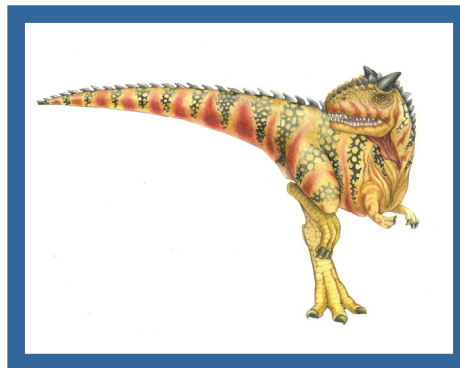
Operating System Concepts

Tenth Edition

Silberschatz, Galvin and Gagne

Chapter 9

Main Memory





Chapter 9: Memory Management

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





Chapter 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





Chapter 9: Background

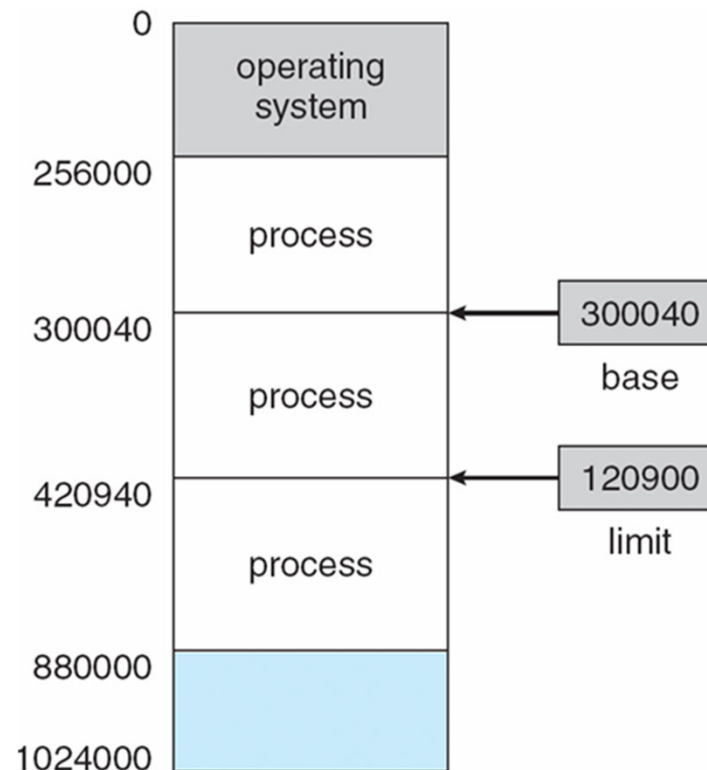
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or 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





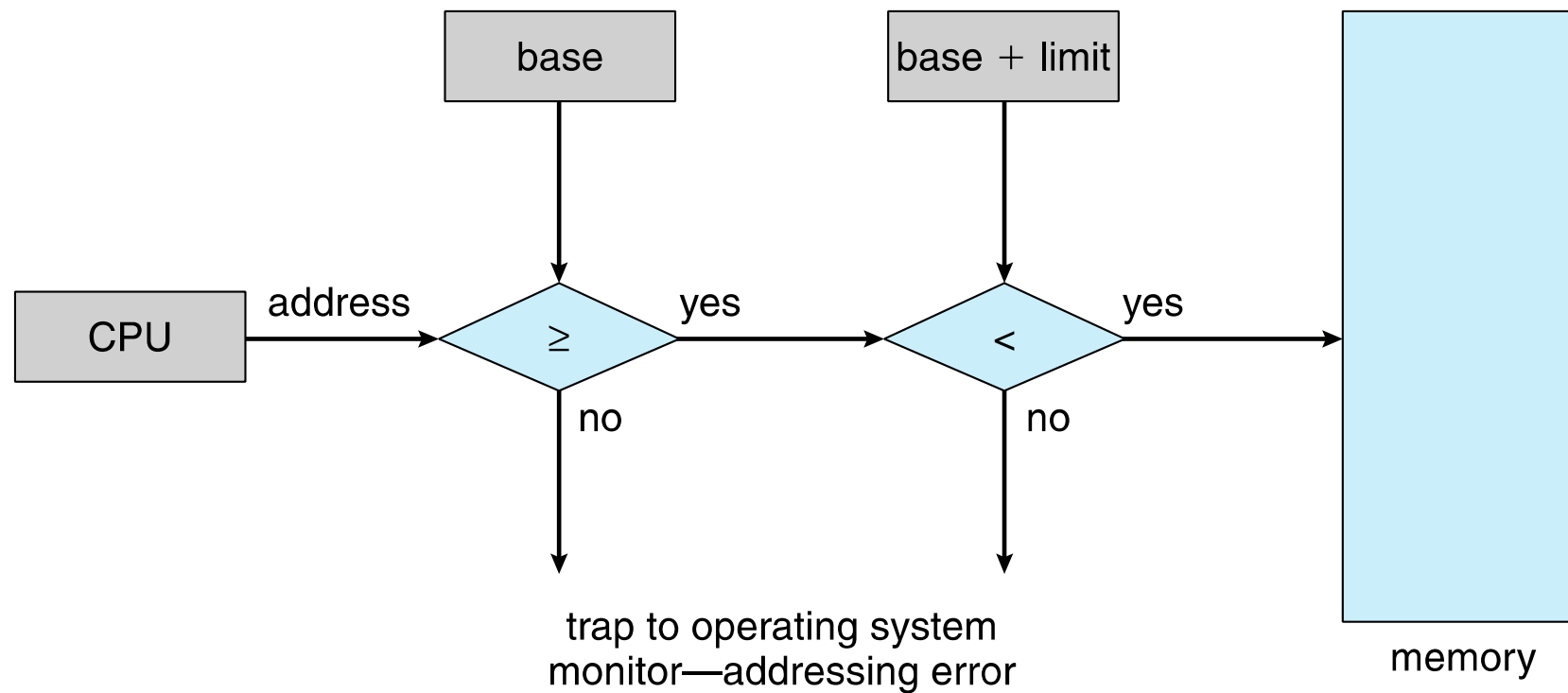
Base and Limit Registers

- A pair of **base** 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





Address Binding ¹

- 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”





Address Binding ₂

- Linker or loader will bind relocatable addresses to absolute addresses
 - i.e. 74014
- Each binding maps one address space to another





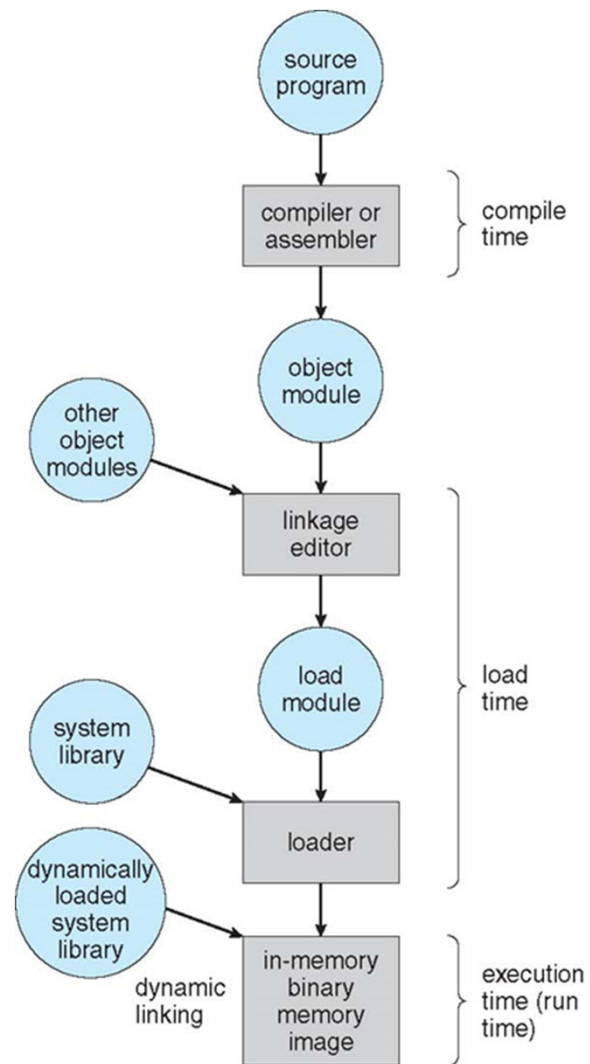
Binding of Instructions and Data to Memory

- 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 the 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)

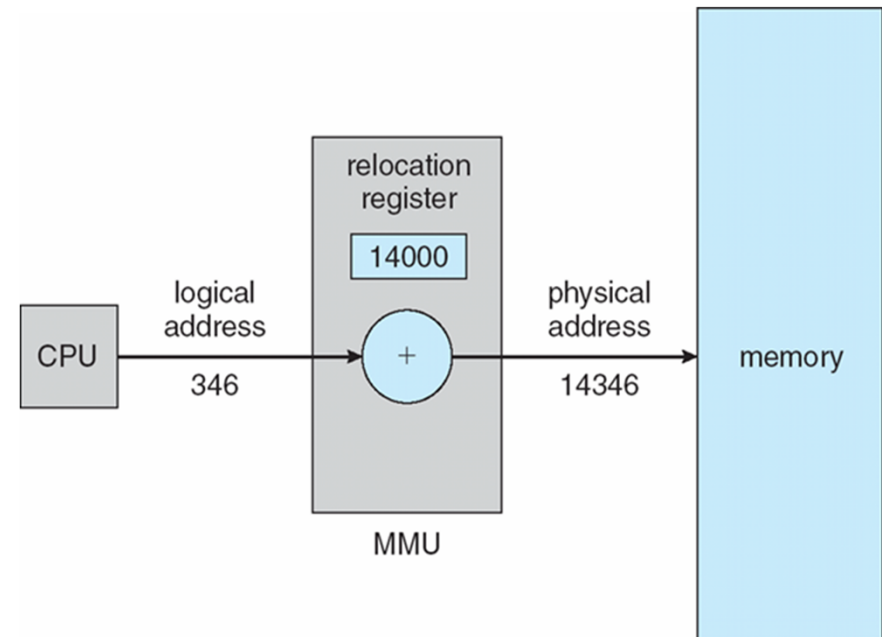
- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base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





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 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





Swapping ¹

- 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 transferring 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





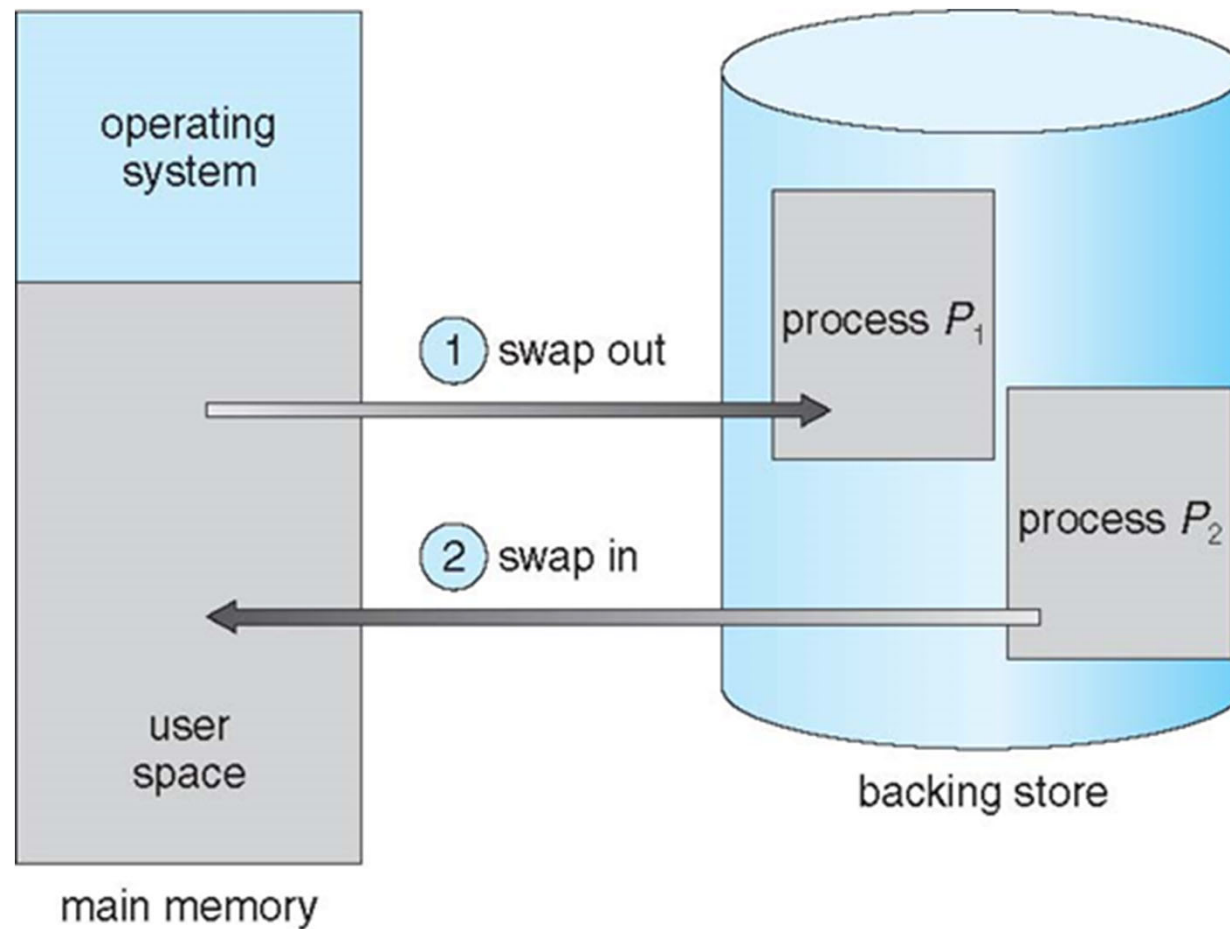
Swapping ²

- Does the swapped-out process need to swap back into 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 including Swapping ²

- 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 in Modern Systems

- Standard swapping not used in modern operating systems
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - Swapping normally disabled
 - Started if memory demand exceeds a certain threshold
 - Disabled again once memory demand drops below threshold





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





Swapping on Mobile Systems ₂

- Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
- Both OSes support paging as discussed below





Contiguous Allocation ₁

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- 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





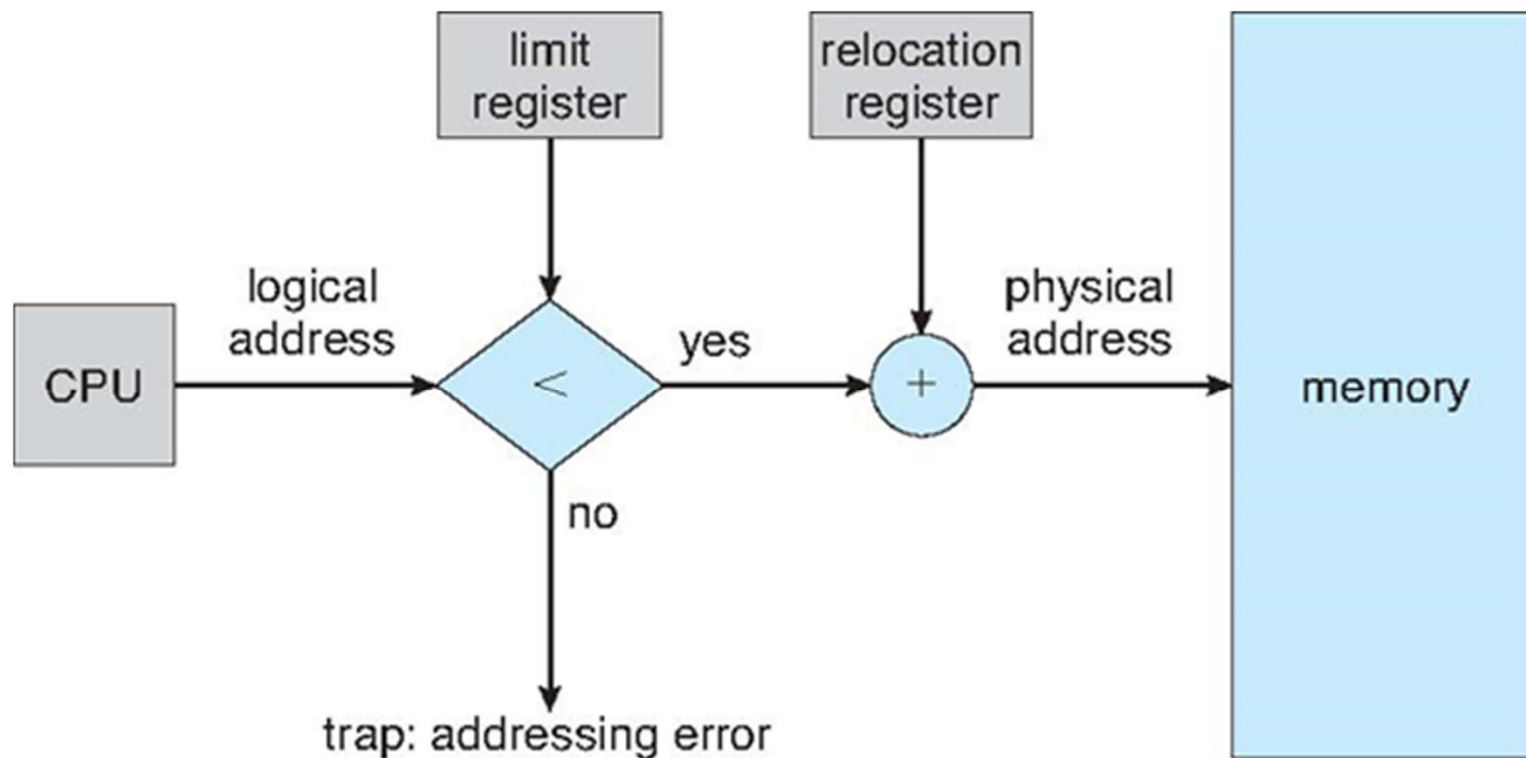
Contiguous Allocation ₂

- 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*
 - Can then allow actions such as kernel code being **transient** and kernel changing size





Hardware Support for Relocation and Limit Registers





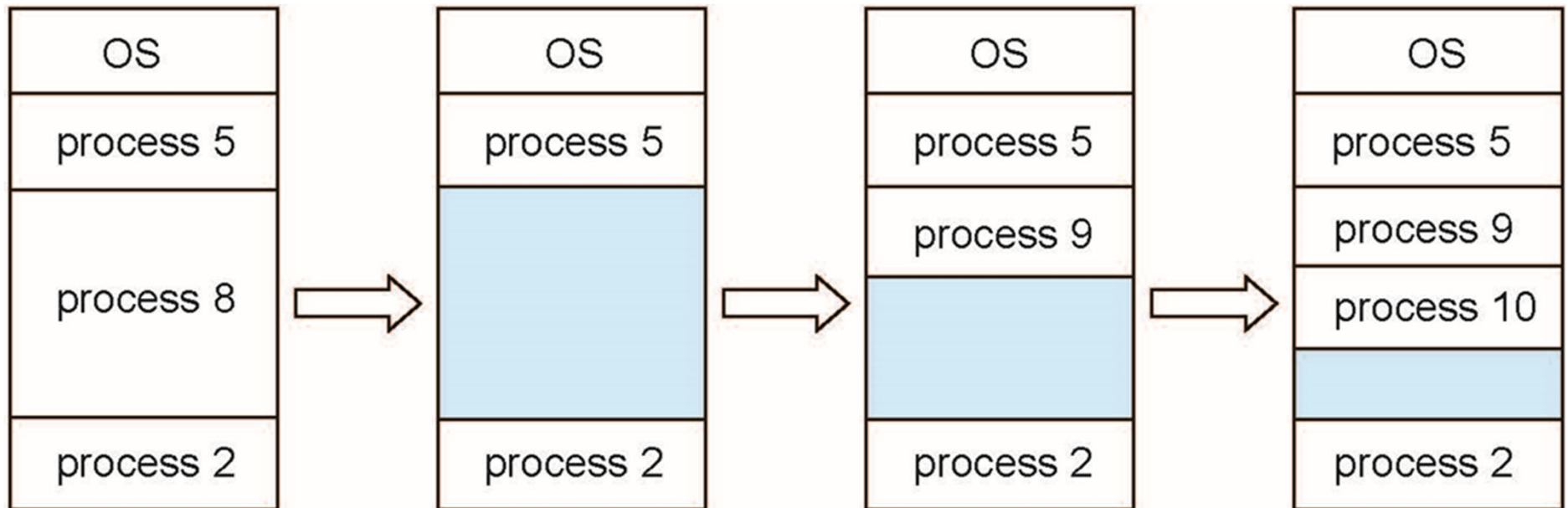
Multiple-partition allocation

- 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)





Multiple-partition allocation ₂





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 ₁

- **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 ₂

- 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
- Now consider that backing store has same fragmentation problems





Segmentation ₁

- 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





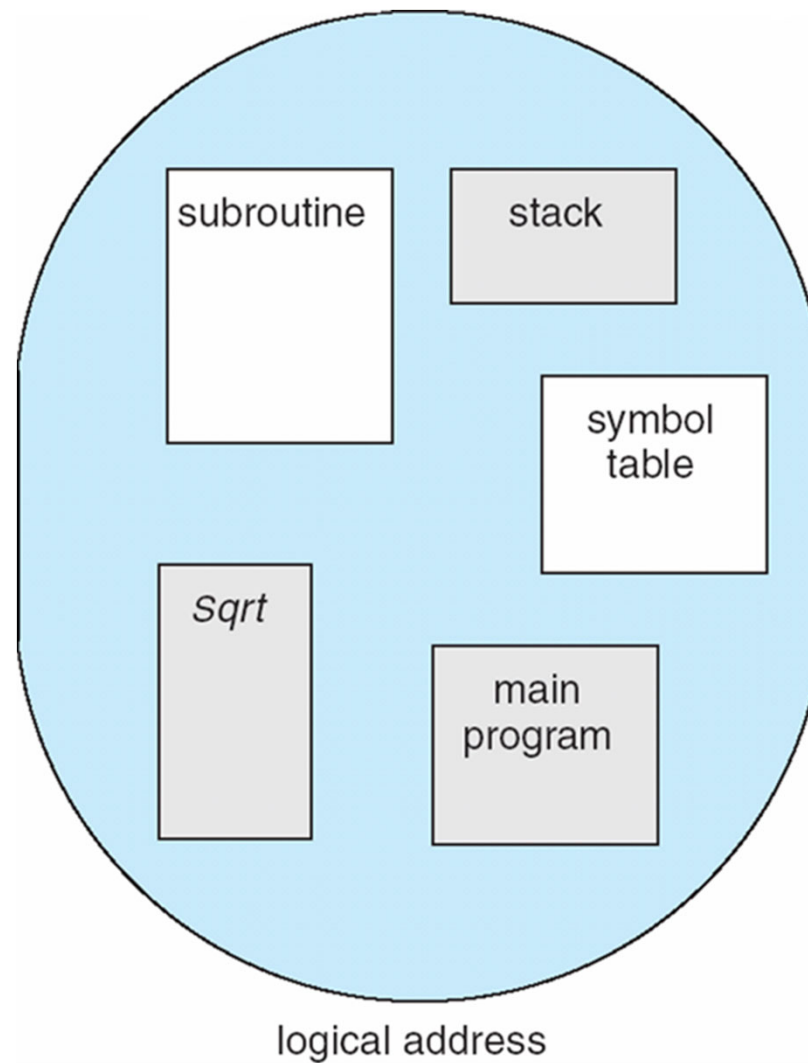
Segmentation ₂

- common block
- stack
- symbol table
- arrays
- A compiler might create separate segments for the following:
 - The code
 - Global Variables
 - The heap (dynamic memory allocation)
 - Stacks used by threads
 - Language Library





User's View of a Program





Segmentation Architecture ₁

- 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**





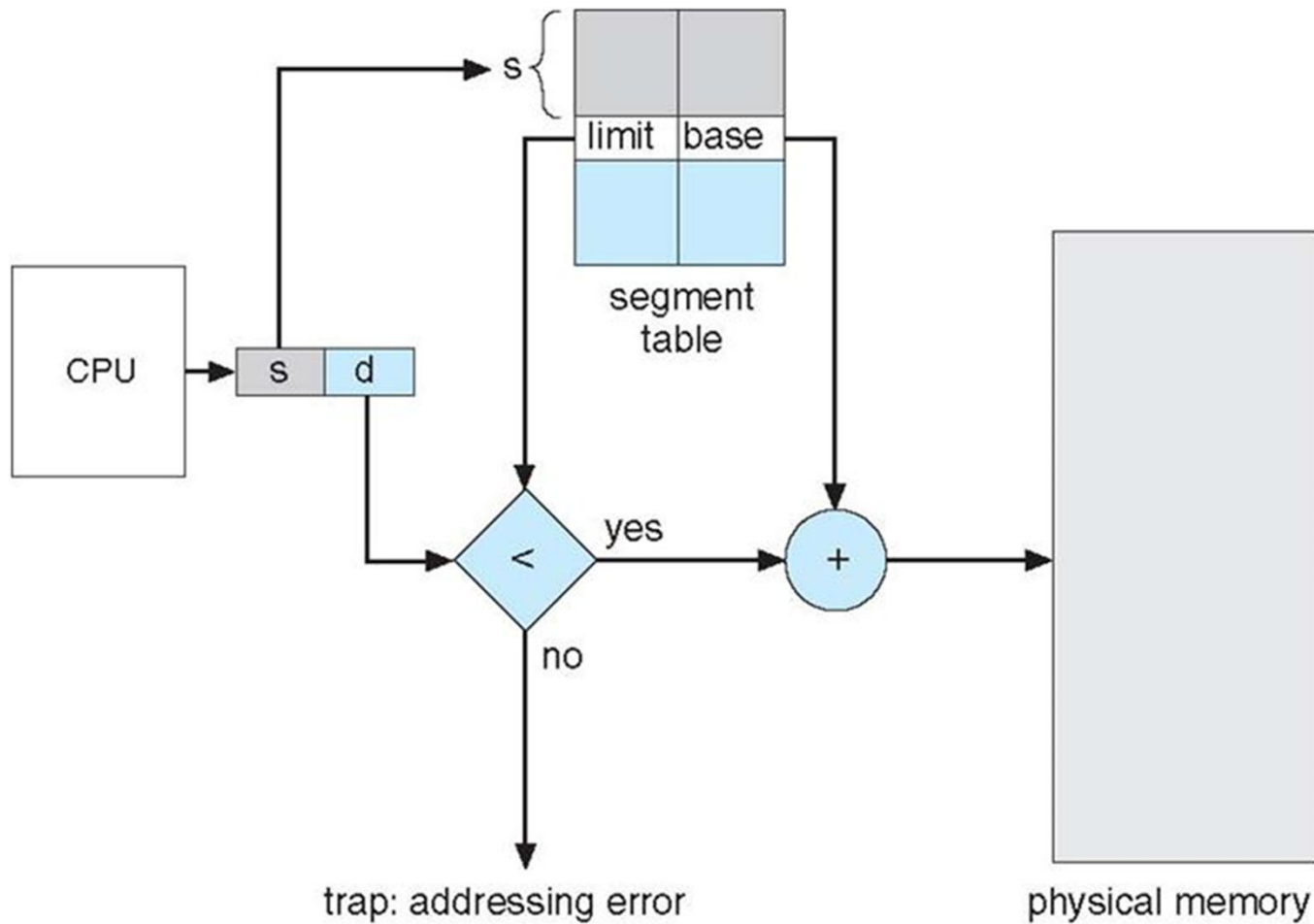
Segmentation Architecture ₂

- 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





Segmentation Hardware





Paging ¹

- 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





Paging₂

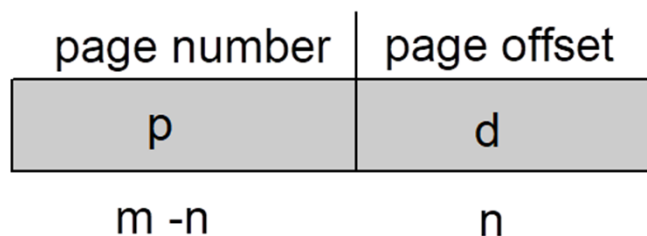
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation





Address Translation Scheme

- Address generated by CPU 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

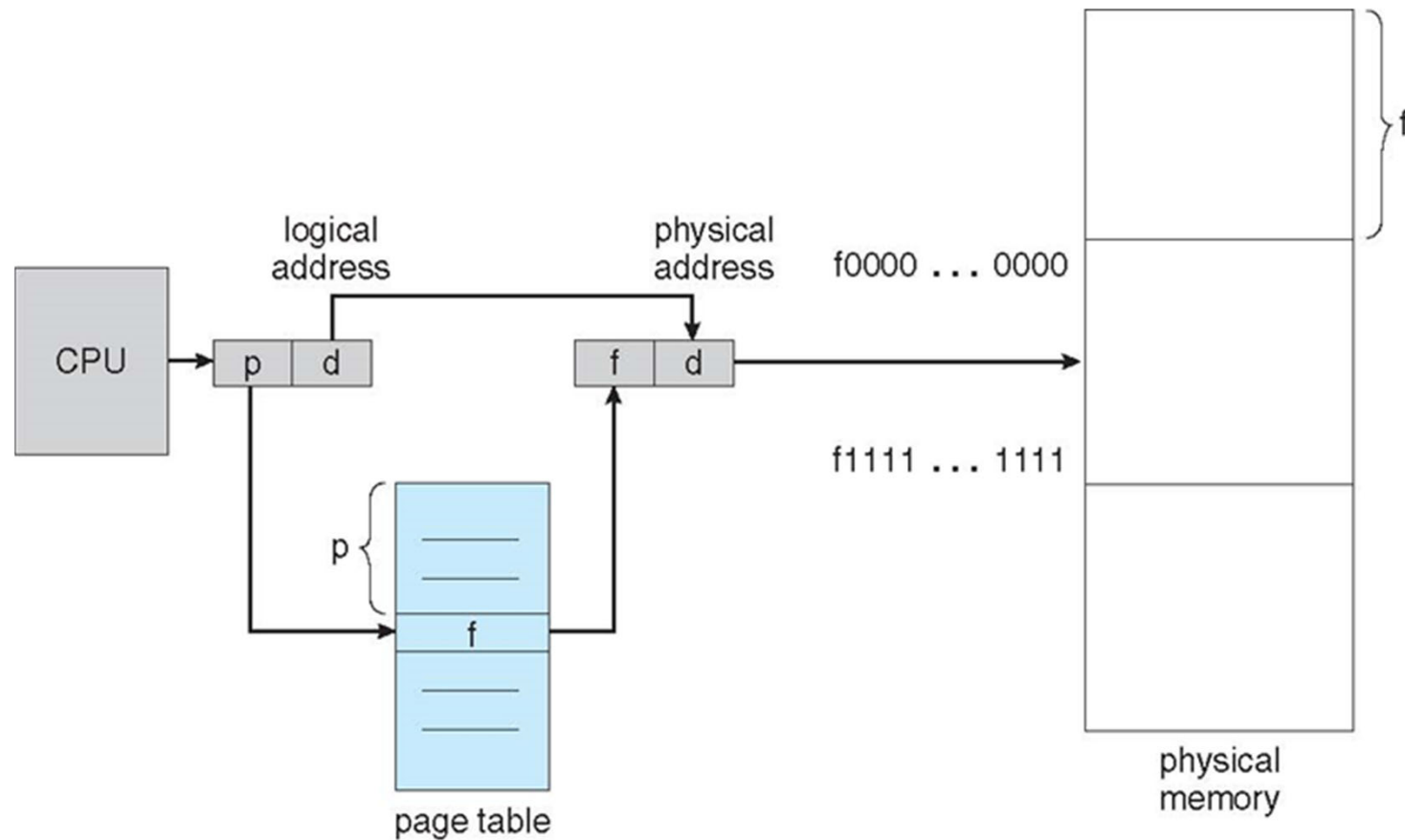


- For given logical address space 2^m and page size 2^n



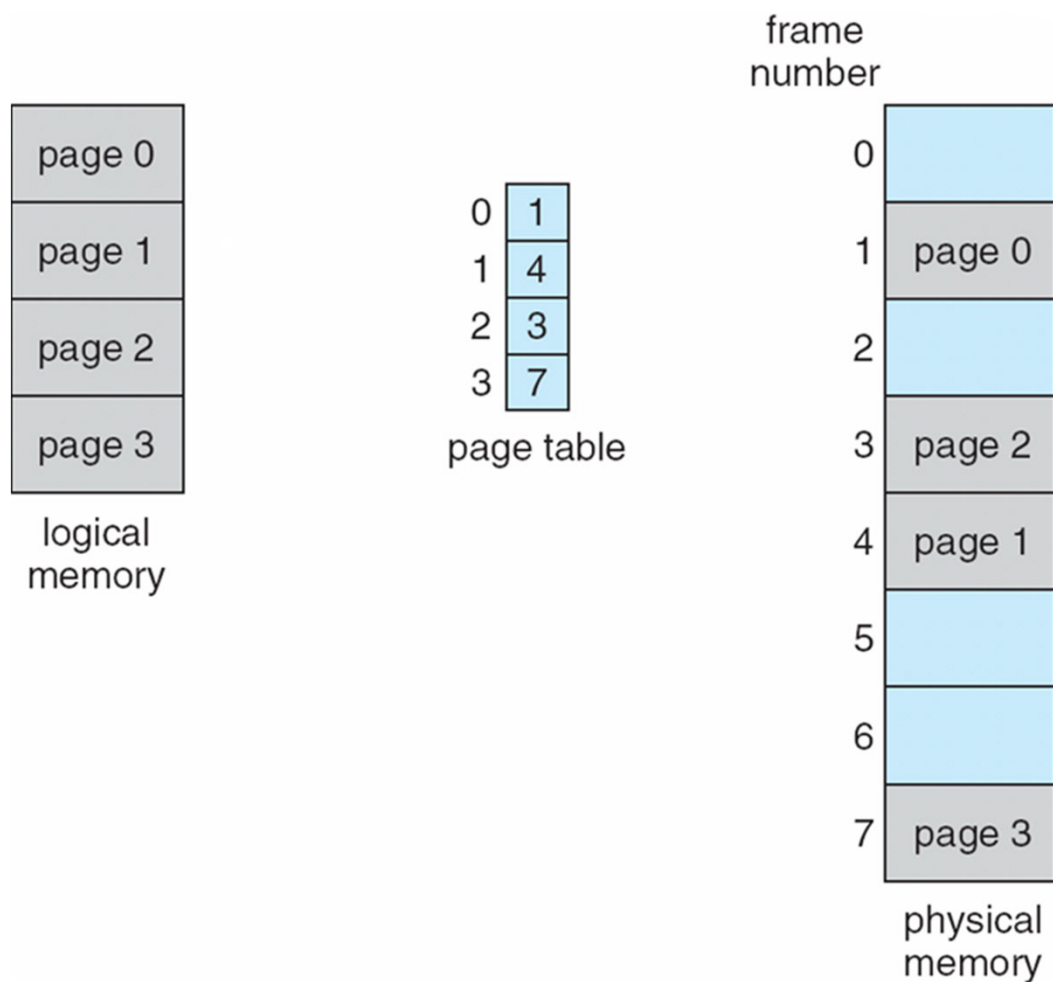


Paging Hardware



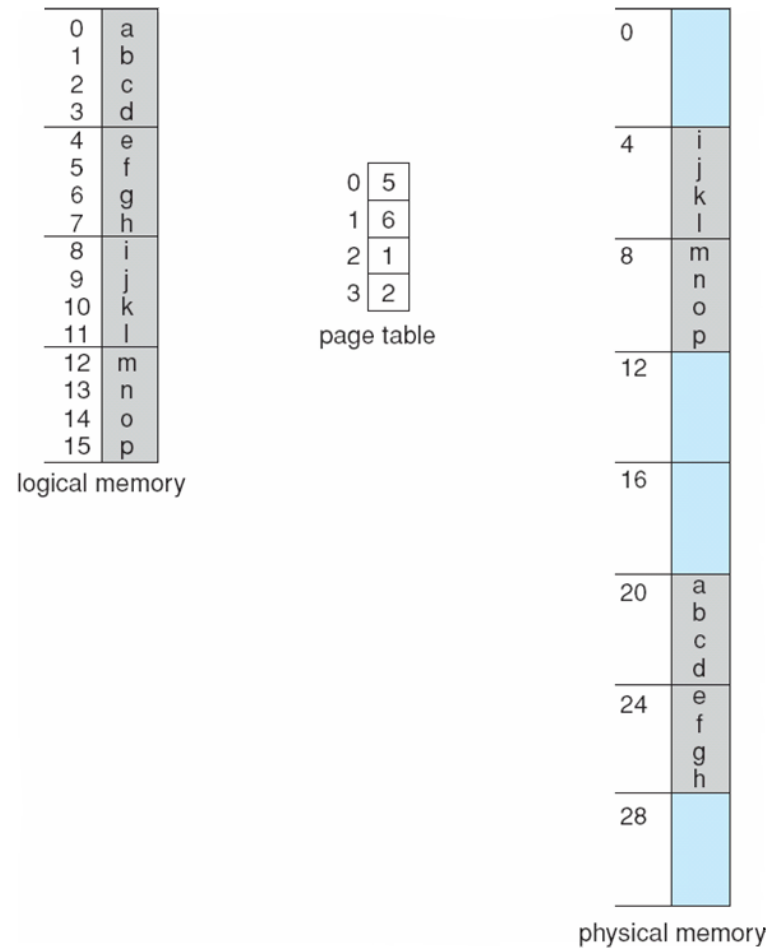


Paging Model of Logical and Physical Memory





Paging Example ₁



$n=2$ and $m=4$ 32-byte memory and 4-byte pages





Paging Example ₂

- 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





Paging Example ₃

- Page sizes growing over time
 - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory





Addressing

- How much memory can we address with 4-byte page table entries?
 - 32-bit entry can address up to 2^{32} page frames
 - If the frame size is 4KB (2^{12}), then we can address 2^{44} bytes (16 TB)
- Actual size will be smaller, because other information will have to be kept in a page table entry
- Process view and physical memory now very different
 - Process view is contiguous, while it is actually scattered throughout physical memory
- By implementation, process can only access its own memory
 - A process has no way of addressing memory outside of its page table
 - Its page table includes only the frames that the process owns

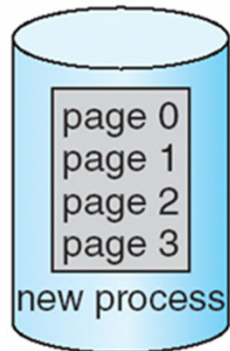




Free Frames

free-frame list

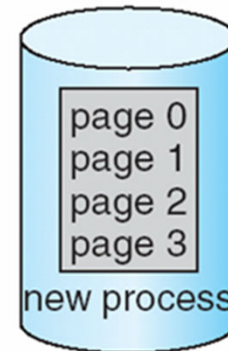
14
13
18
20
15



(a)

Before allocation

free-frame list
15



(b)

new-process page table

0	14
1	13
2	18
3	20

After allocation





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 **associative memory** or **translation look-aside buffers (TLBs)**





Implementation of Page Table ₂

- 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





Associative Memory

- Associative memory – parallel search

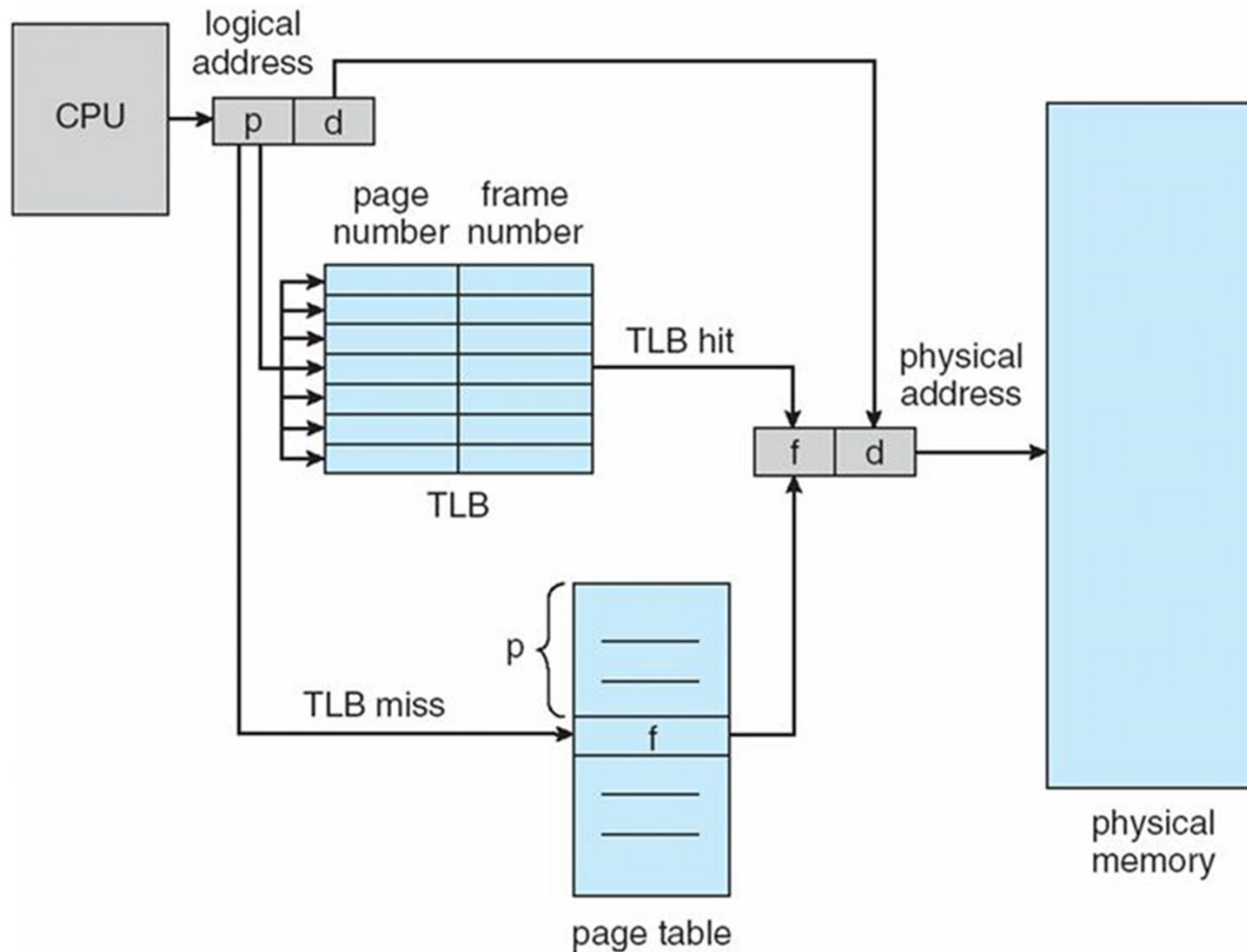
Page #	Frame #

- 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

- 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
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, $M = 100\text{ns}$ for memory access





Effective Access Time ₂

- **Effective Access Time (EAT)**

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

- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, $M = 100\text{ns}$ for memory access
 - $\text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$
- Consider more realistic hit ratio $\rightarrow \alpha = 99\%$, $\varepsilon = 2\text{ns}$ for TLB search, $M = 100\text{ns}$ for memory access
 - $\text{EAT} = 0.99 \times 102 + 0.01 \times 202 = 103\text{ns}$





Multiple-Choice Question

- Which of the following statement is correct?
 - A) Base register holds the size of a process.
 - B) Limit register holds the size of a process.
 - C) Base and limit registers can be loaded by the standard load instructions in the instruction set.
 - D) Any attempt by a user program to access memory at an address higher than the base register value results in a trap to the operating system.





Multiple-Choice Question ²

- _____ is the method of binding instructions and data to memory performed by most general-purpose operating systems.
 - A) Interrupt binding
 - B) Compile time binding
 - C) Execution time binding
 - D) Load-time binding





Multiple-Choice Question ³

- Given the logical address 0xAEF9 (in hexadecimal) with a page size of 256 bytes, what is the page number?
 - A) 0xAE
 - B) 0xF9
 - C) 0xA
 - D) 0x00F9





Essay Questions

- Distinguish between internal and external fragmentation.
- Describe how a transaction look-aside buffer (TLB) assists in the translation of a logical address to a physical address.
- What is the context switch time, associated with swapping, if a disk drive with a transfer rate of 2 MB/s is used to swap out part of a process that is 200 KB in size? The average latency is 15 ms. The time should reflect only the amount of time necessary to swap out the process.

