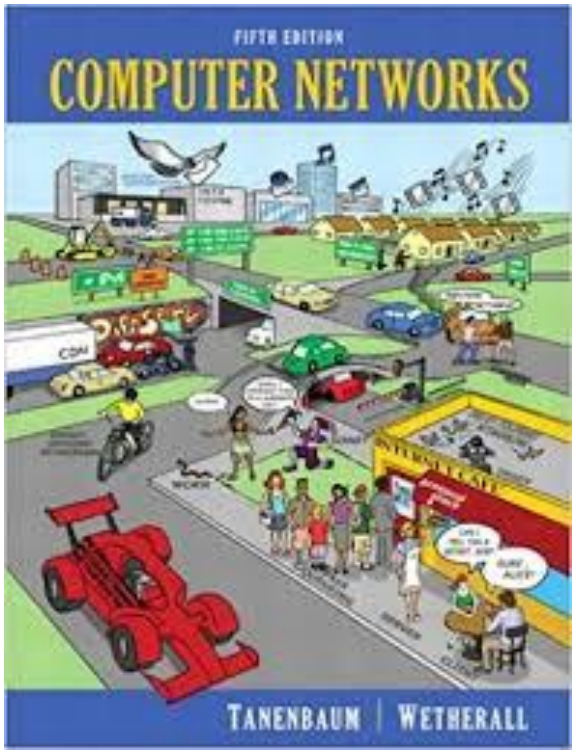
Part Three: Storage Management

- The programs are in the Main memory
  - We need a memory management scheme to manage several processes in the main memory
- Main memory is very small
  - Secondary storage is used to support main memory
- We also need file management for on-line storage of access to both data and programs that reside on disk

Announcements

- Topics to be covered
  - memory management (2 classes),
  - virtual memory (2 classes),
  - Disk scheduling and file management (1 class),
  - Protection, security, media management (1 class)
  - overview of computer networks (6 classes).
- Procure the following book.



Memory Management

- Memory is shared by multiple processes
- We discuss various ways to manage memory.
- Outline
  - Background
    - Address binding
    - Dynamic loading and linking
    - Overlays
    - Swapping
  - Memory management approaches
    - Contiguous Allocation
    - Paging
    - Segmentation
    - Segmentation with Paging

Background

- Memory is a central part of modern computer systems
- Large array of bytes and words each with its own address.
- CPU fetches and executes instructions
  - After the operation, the result is stored back into memory
- Memory unit sees only stream of addresses.

- Program resides on the disk.
- Program must be brought into memory and placed within a process for it to be run.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.

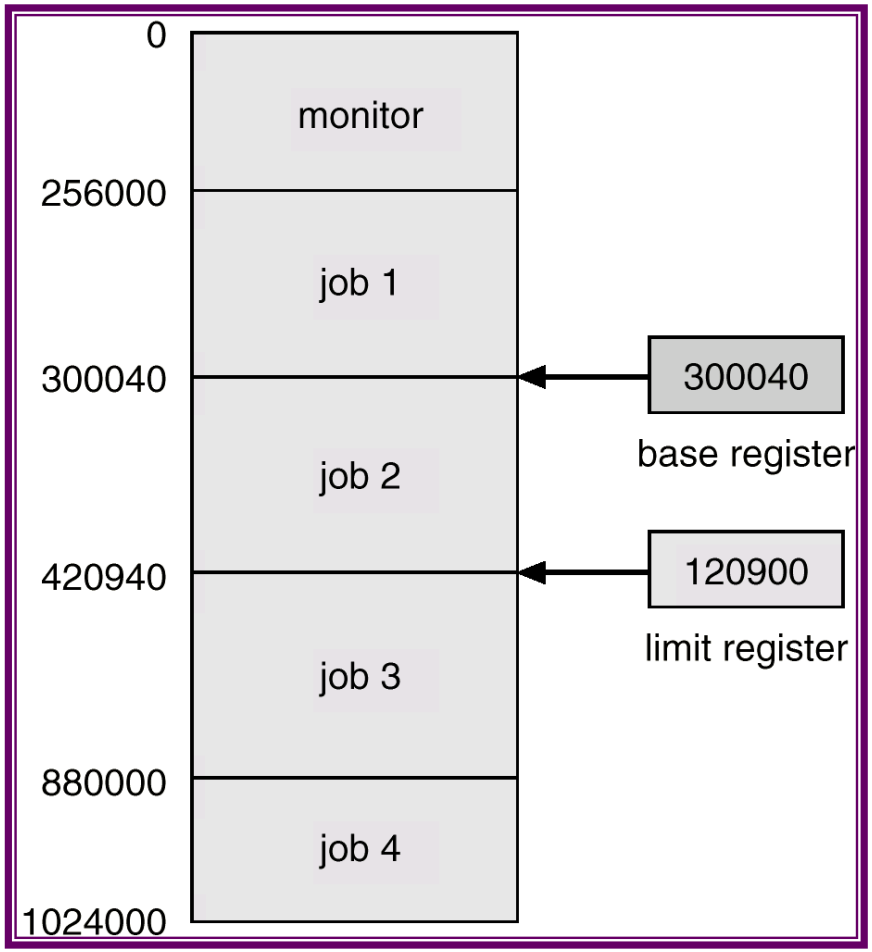
Basic hardware

- Ensuring correct operation
  - Each process has a separate memory space
- Base and limit registers
  - Every address should be compared.

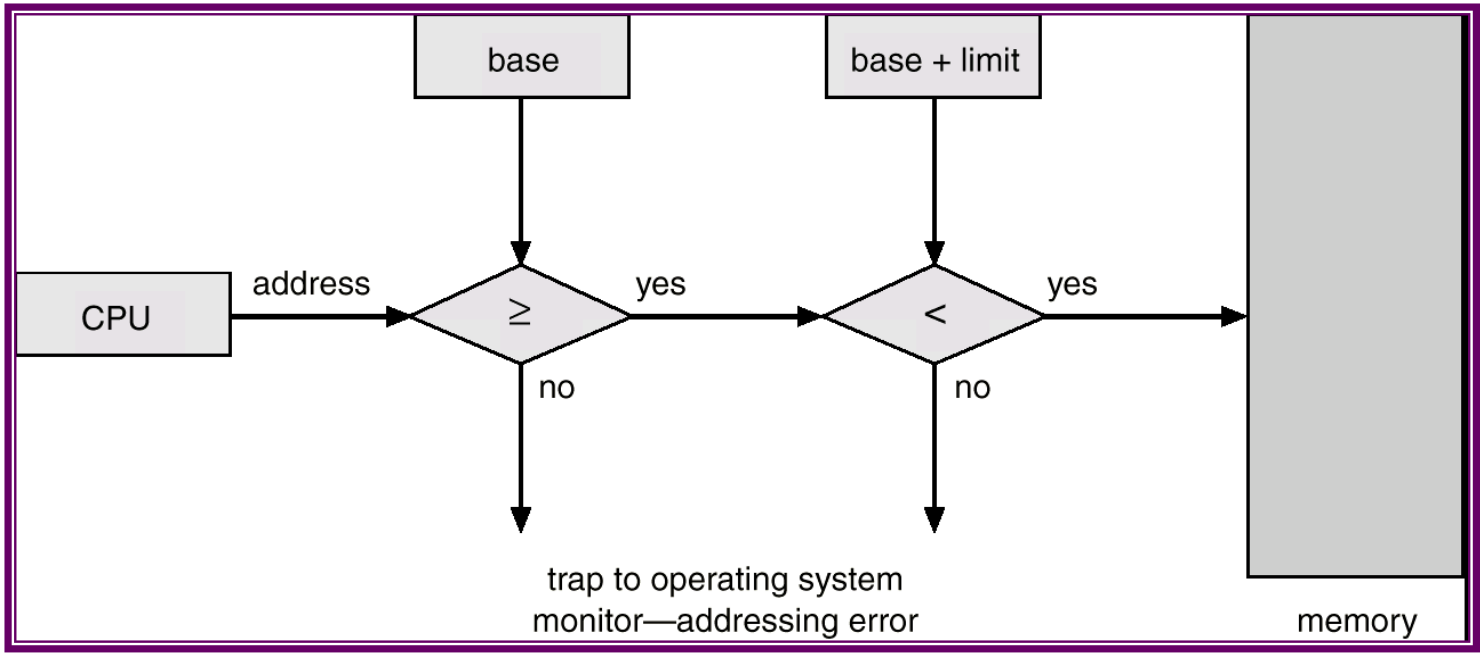
Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** – holds the smallest legal physical memory address.
  - **Limit register** – contains the size of the range
- Memory outside the defined range is protected.

Use of A Base and Limit Register



Hardware Address Protection



Memory Protection

- When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory.
- The load instructions for the *base* and *limit* registers are privileged instructions.

Address binding

- Most systems allow user process to reside in any part of the physical memory.
- Address space starts at 00000; However, the first address of user process may not start at 00000.
- So user program will go through several steps.
- Addresses in the source program are symbolic.
  - Example: Count
- A compiler will bind these addresses to re-locatable addresses.
  - Example: 14 bytes from the beginning of the module.
- A linkage editor or a loader will bind these re-locatable addresses to absolute addresses.
- A binding is a mapping from one address space to another.

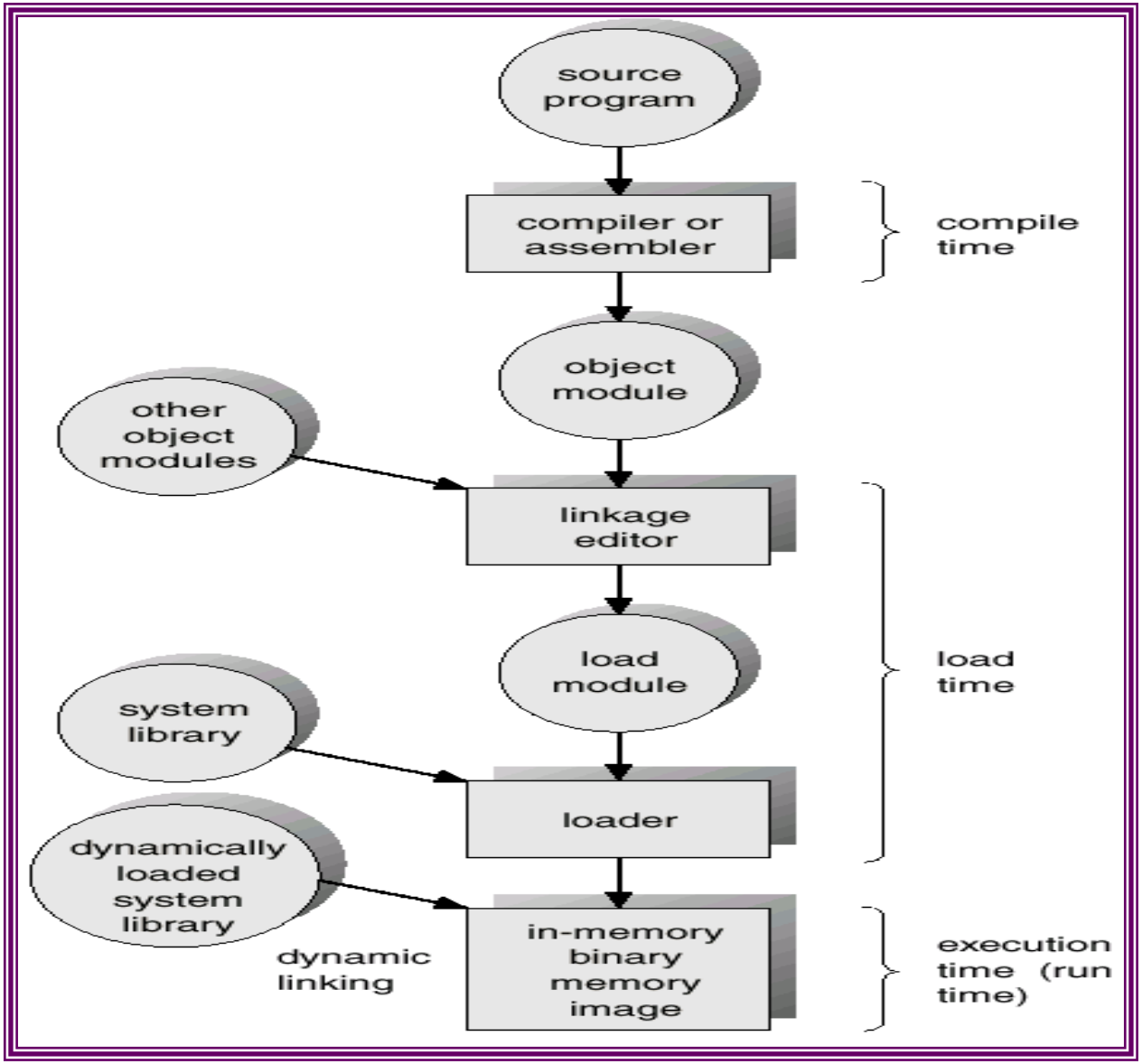
Binding of Instructions and Data to Memory

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.
  - MSDOS.COM-format programs are absolute code bound
- **Load time**: Compiler must generate *relocatable* code if memory location is not known at compile time.
  - Final binding is delayed until load time.
  - If the starting address changes, we need only to reload the user code to incorporate the changed value.
- **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*).
  - Most OSs use this method.

In this chapter we study about various binding schemes and corresponding hardware support.

Multi-step Processing of a User Program



Dynamic Loading

- Dynamic loading is used to obtain better memory-space utilization
- Routine is not loaded until it is called
- All routines are kept on disk in a re-locatable load format
  - Better memory-space utilization; unused routine is never loaded.
  - 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.

Dynamic Linking

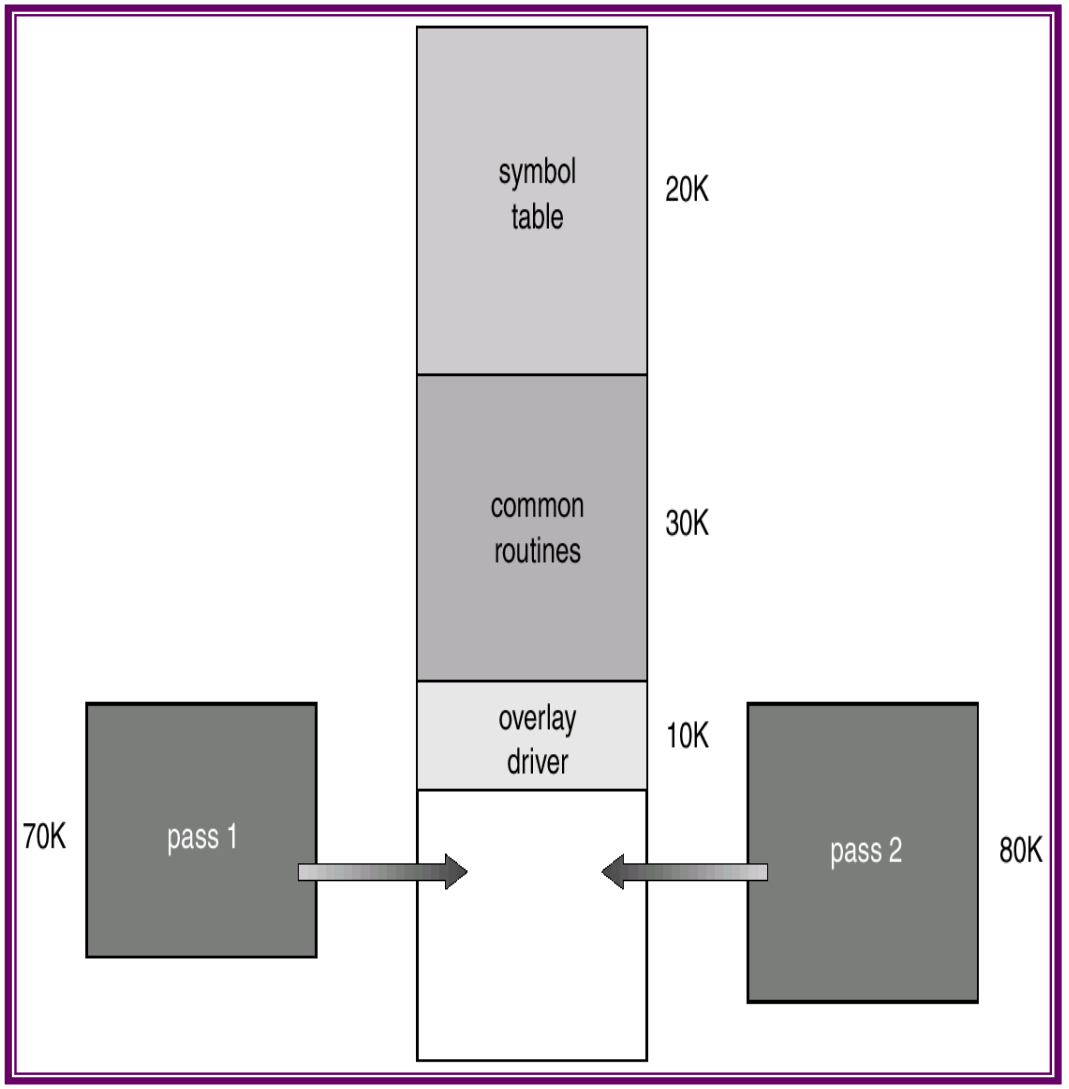
- Some OSs support static linking.
  - System language libraries are treated like any other object module and loader combines them into binary program image.
- In dynamic linking, the linking is postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.
  - Library updates; different versions
- Needs support from OS.

Overlays

- To allow a process to be larger than main memory Overlays can be used.
- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.
- Example: assembler
  - Pass1: 70K
  - Pass2: 80K
  - Symbol table: 20 K
  - Common routines: 30 K
  - To load everything we need 200K of memory.
  - With 150 K we can not run the program.
  - Two overlays are defined:
    - Overlay A: Symbol table, common routine and pass1
    - Overlay B: Symbol table, common routines and pass2.

Overlays for a Two-Pass Assembler

- After finishing Pass1, the overlay driver reads overlay B, overwriting overlay A, and transfers control to pass 2.
- Special loading and linking algorithms are needed to construct overlays.
- Writing of overlay driver requires complete knowledge of the program structure.
- Limited to microcomputers and embedded systems
  - Lack hardware support for memory management.



Logical vs. Physical Address Space

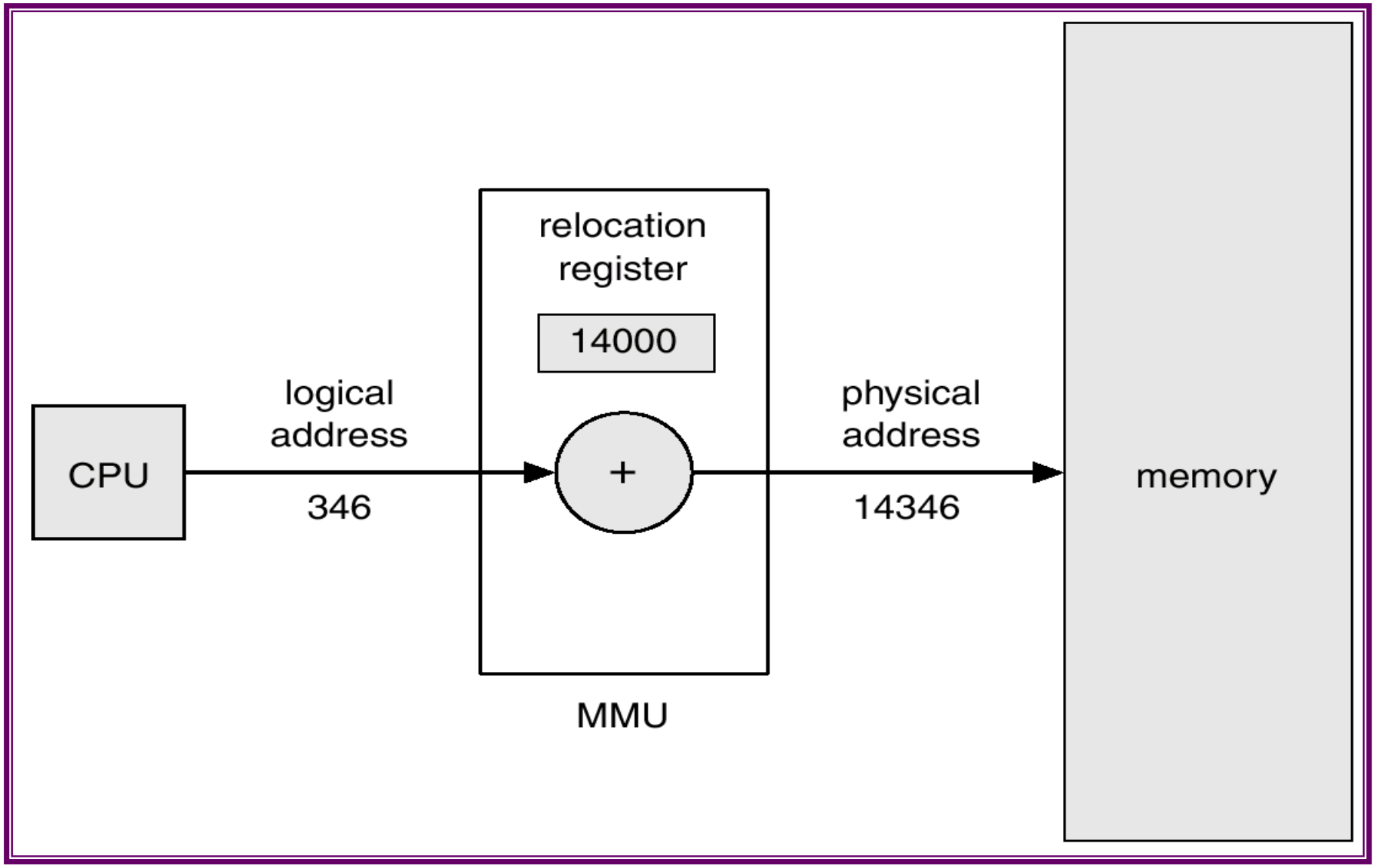
- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
    - Set of logical addresses generated by a program is called logical address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

- 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 (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

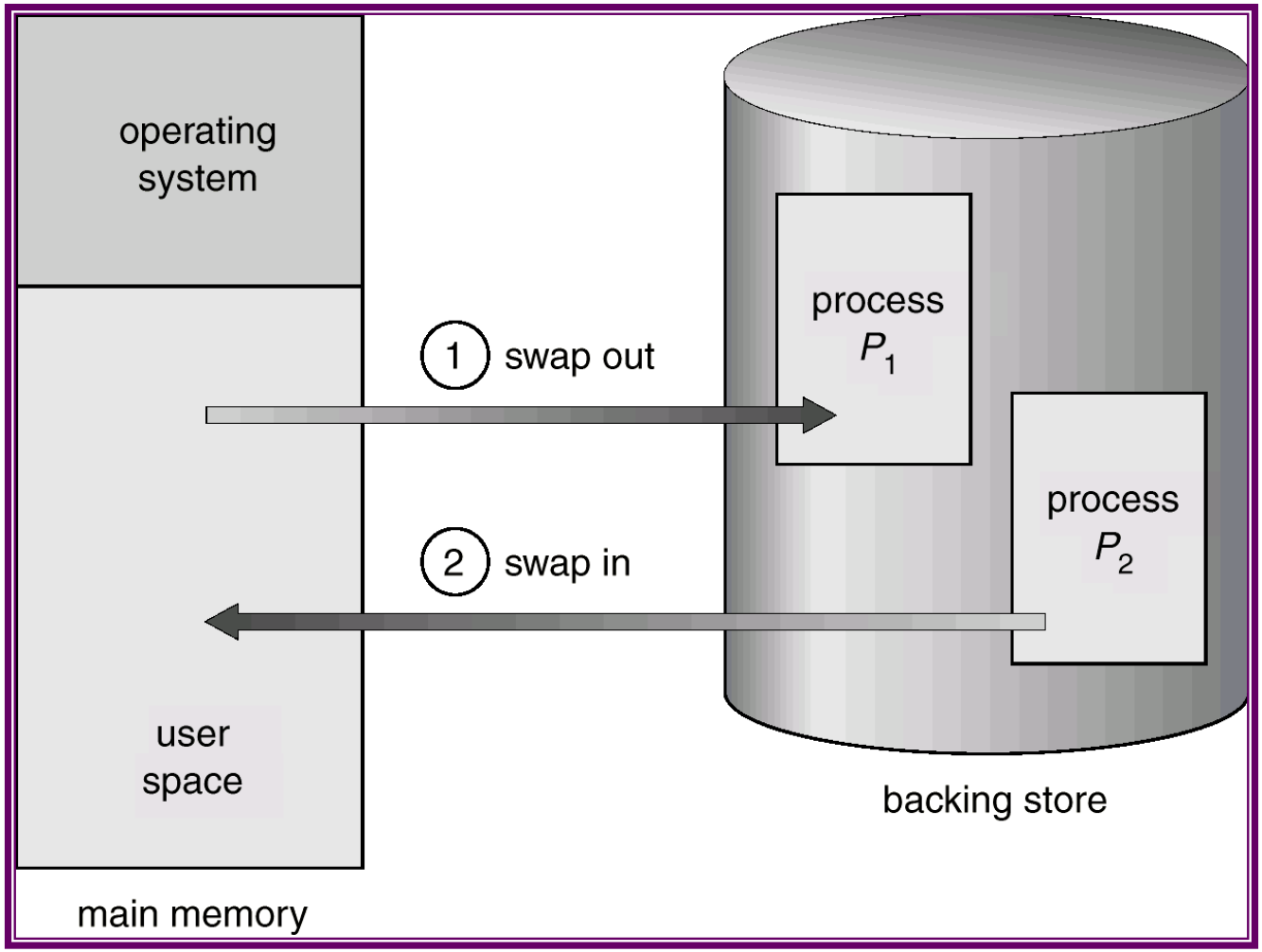
Dynamic relocation using a relocation register



Swapping

- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks whether the process is in main memory.
- If the process is not, and there is no free memory, the dispatcher swaps out a process currently in main memory and swaps in the desired process.
- 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.
- When a process which is swapped-out swaps-in
  - Binding during assembly and loading
    - Process can not be moved to different locations.
  - Binding during execution
    - Process can be moved to different locations
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

Schematic View of Swapping



Swapping…

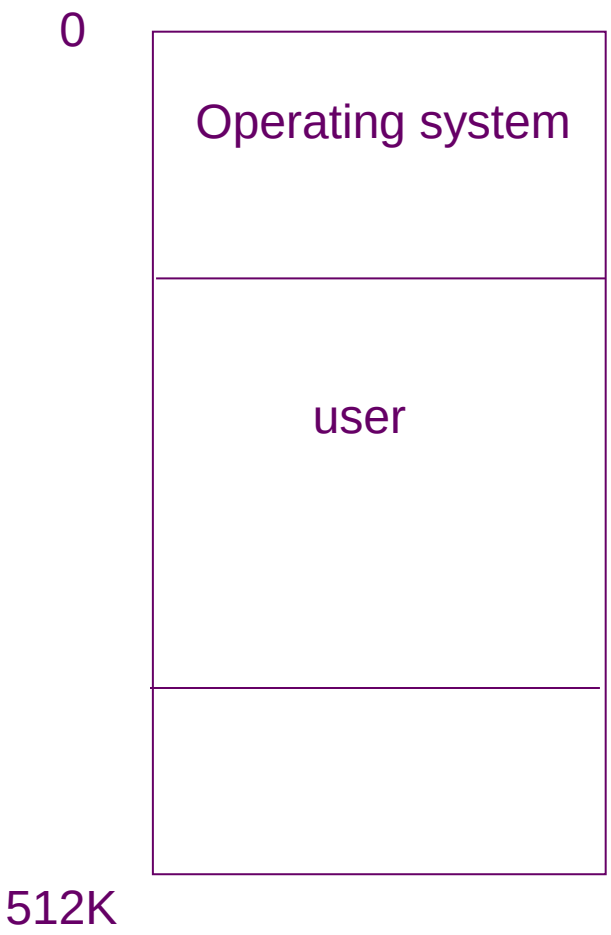
- A process should be idle to swap.
  - It should not have any pending I/Os.
- Solutions
  - Never swap a process with pending I/Os
    - Difficult if the process is accessing asynchronously.
  - Execute I/O operations into OS buffers.
    - Transfers between OS buffers to process occur only when the process is swapped in.
- UNIX
  - Swapping is normally disabled.
  - However, swapping starts if many processes are running and using threshold amount of memory.
  - Swapping is halted when the load is reduced.
- WINDOWS
  - OS does not provide full swapping.
  - The scheduler decides when the process should be swapped out.
  - When the user selects the process, the process is swapped-in.

Outline

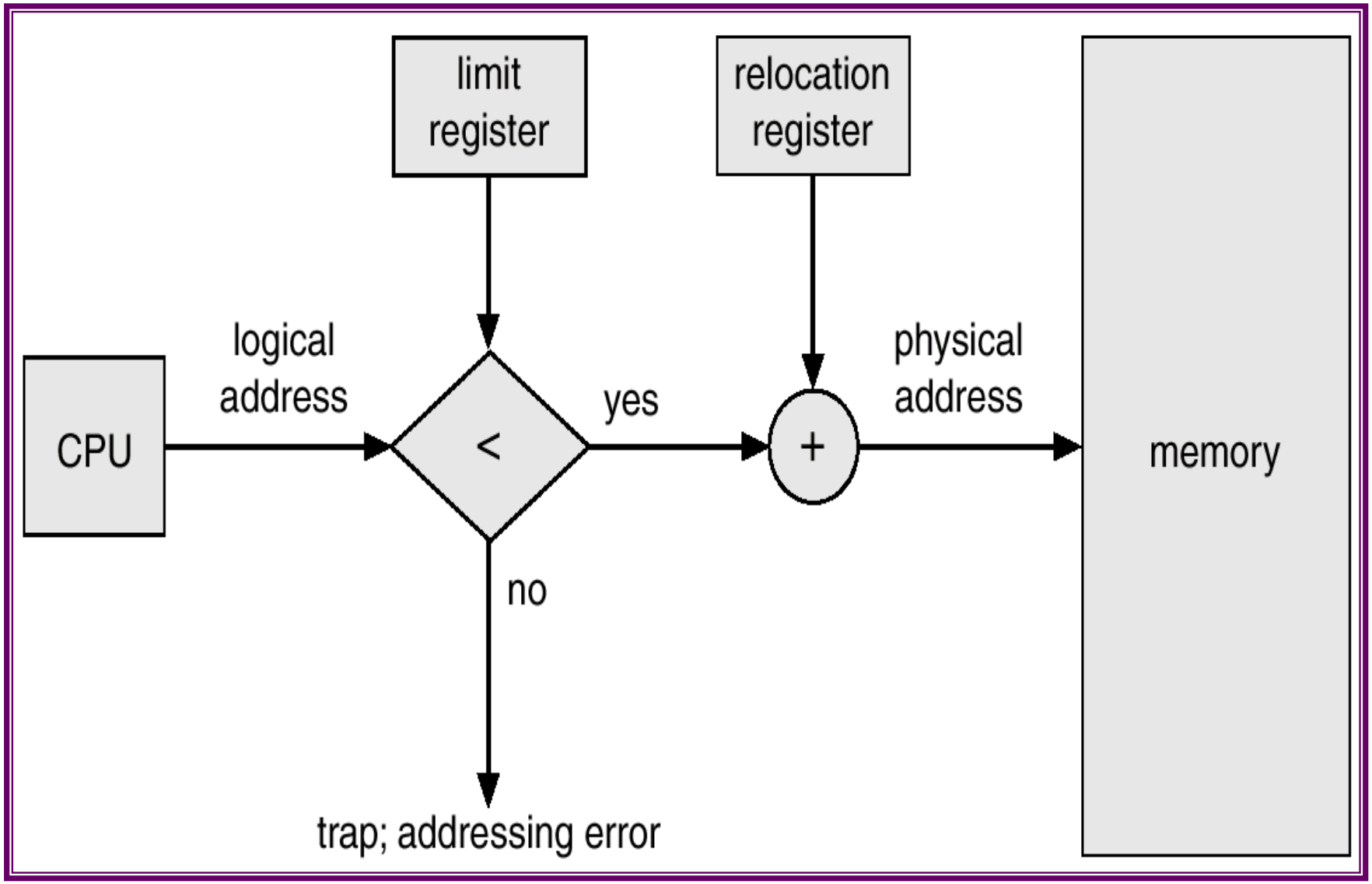
- Background
  - Address binding
  - Dynamic loading and linking
  - Overlays
  - Swapping
- **Memory management approaches**
  - **Contiguous Allocation**
  - **Paging**
  - **Segmentation**
  - **Segmentation with Paging**

Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.
- When a CPU scheduler selects a process from execution, the dispatcher loads the relocation and limit registers with correct values as a part of context switch.



Hardware Support for Relocation and Limit Registers

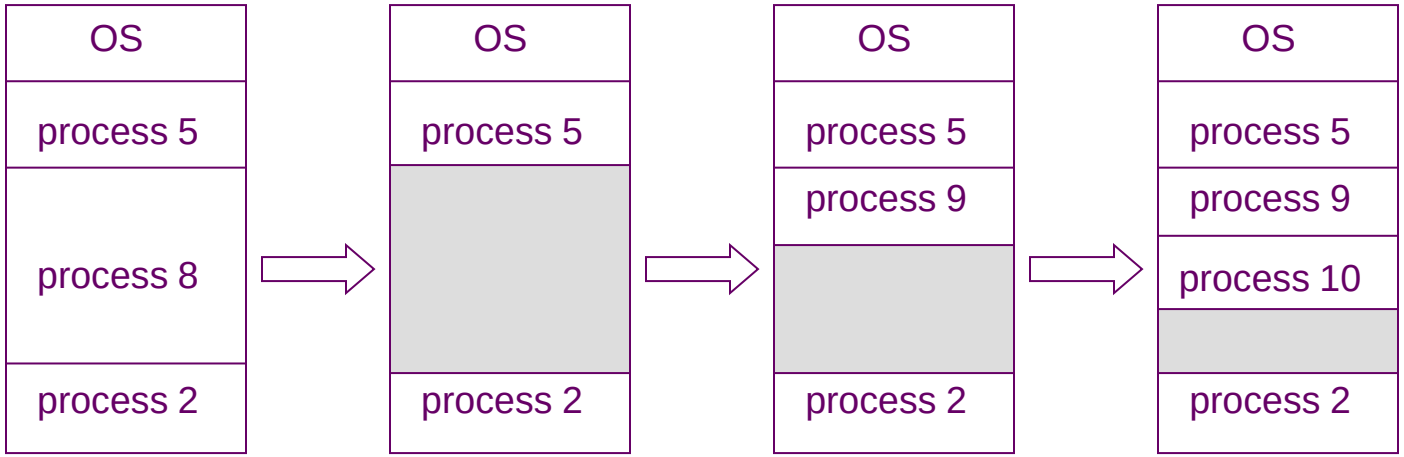


Contiguous Allocation (Cont.)

- Multiple-partition allocation
- How to allocate available main-memory to the various processes ?
- Simple solution (fixed partition solution): partition the memory
  - Number of programs= Number of processes.
  - Not efficient
- Improved solution:
  - Generalization of fixed partition solution.
- OS keeps a table indicating which parts of memory are available and which are occupied.
- Initially all memory available for a user process.
- *Free partition or Hole* – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from free partitions large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

Contiguous Allocation (Cont.)

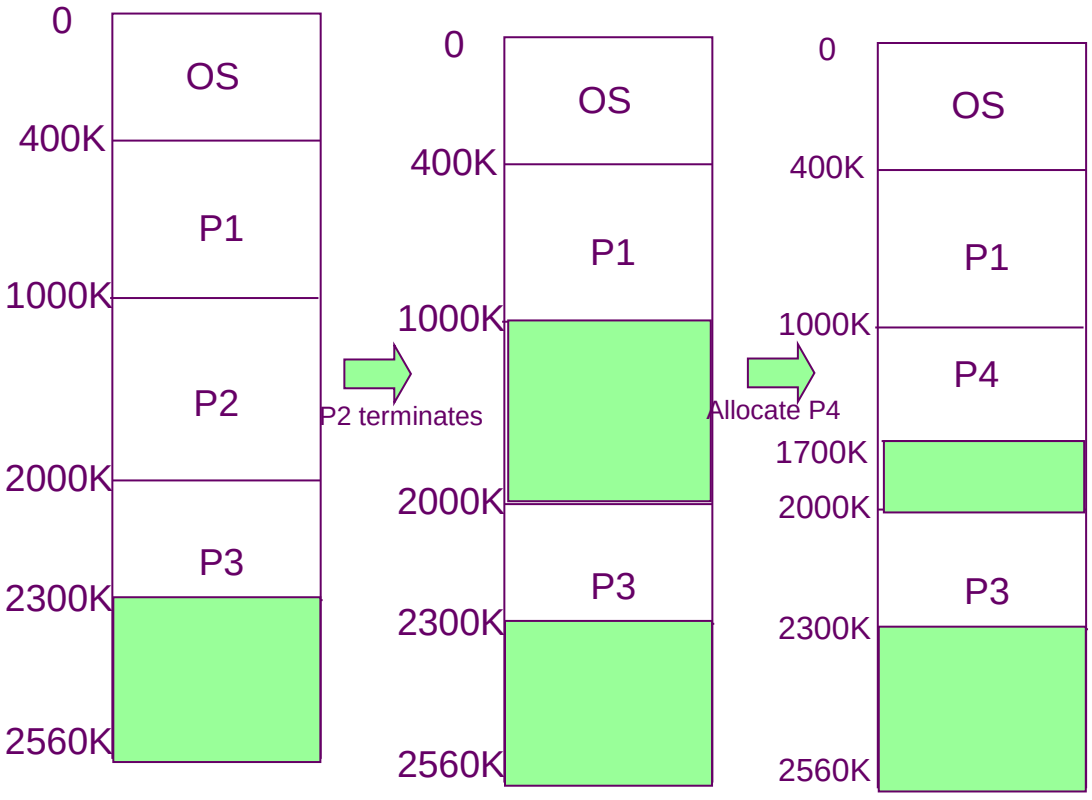
- At any time a set of holes of various sizes scattered throughout the memory
- A free partition large enough for the requesting process is searched.
- If the free partition is too large it is split into two: one is allocated to the arriving process and the other is returned to the set of partitions.
- When a process terminates, it releases its block of memory, which is then placed back in the set of free partitions.
- If the new hole is adjacent to other partitions, we merge these partitions to form a larger hole.
- Example:

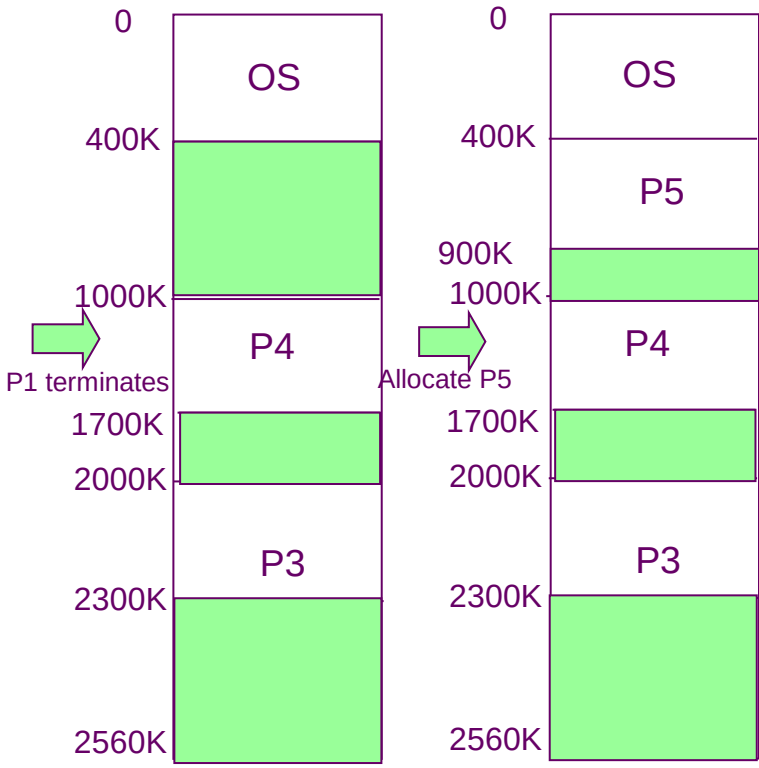


Contiguous Allocation (Cont.)

- Example: 2560K of memory available and a resident OS of 400K. Allocate memory to processes P1…P4 following FCFS.
- Shaded regions are holes
- Initially P1, P2, P3 create first memory map.

Process	Memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15





Dynamic Storage-Allocation Problem

Algorithms to search for a free partition of size n.

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

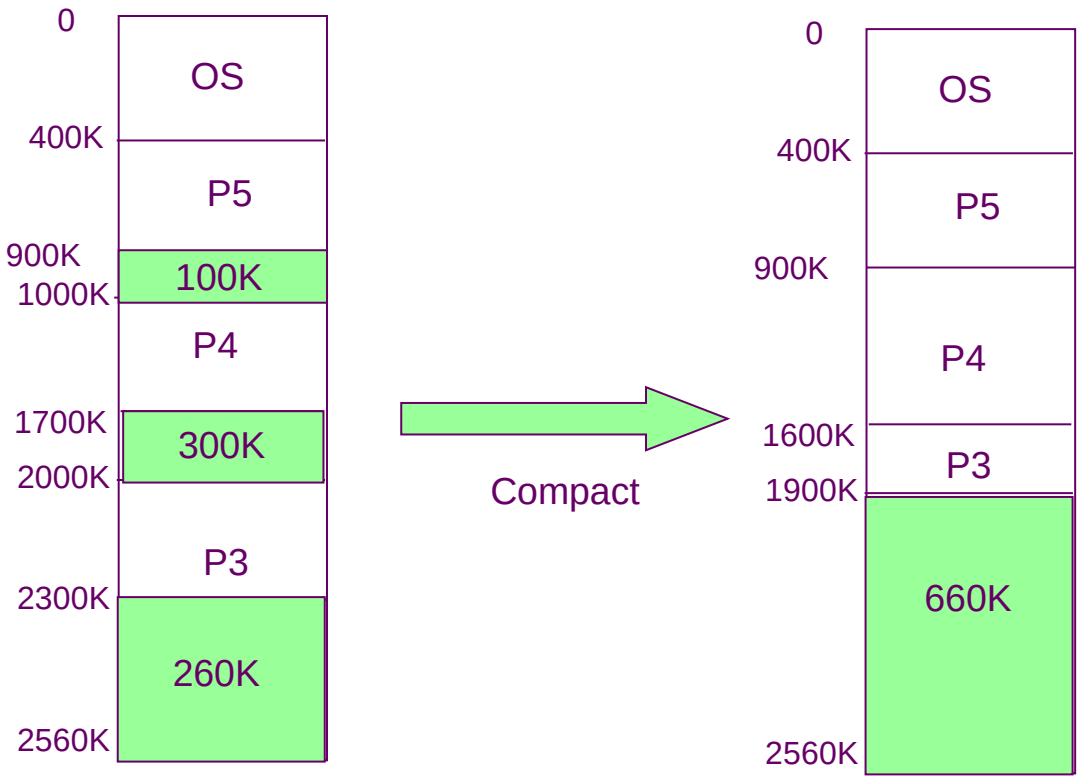
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
  - 50 % rule: Given N allocated blocks 0.5 blocks will be lost due to fragmentation.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
  - Consider the hole of 18,464 bytes and process requires 18462 bytes.
  - If we allocate exactly the required block, we are left with a hole of 2 bytes.
  - The overhead to keep track of this free partition will be substantially larger than the hole itself.
  - Solution: allocate very small free partition as a part of the larger request.

Solution to fragmentation

- Compaction
- Paging
- Segmentation

Compaction

- 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.
  - Compaction depends on cost.

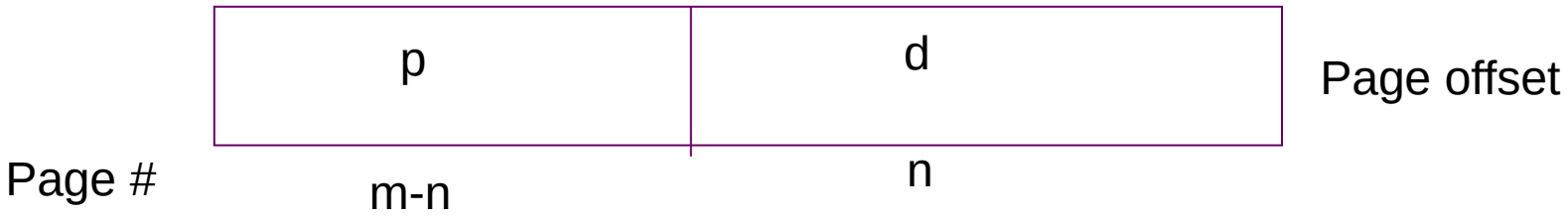


Paging

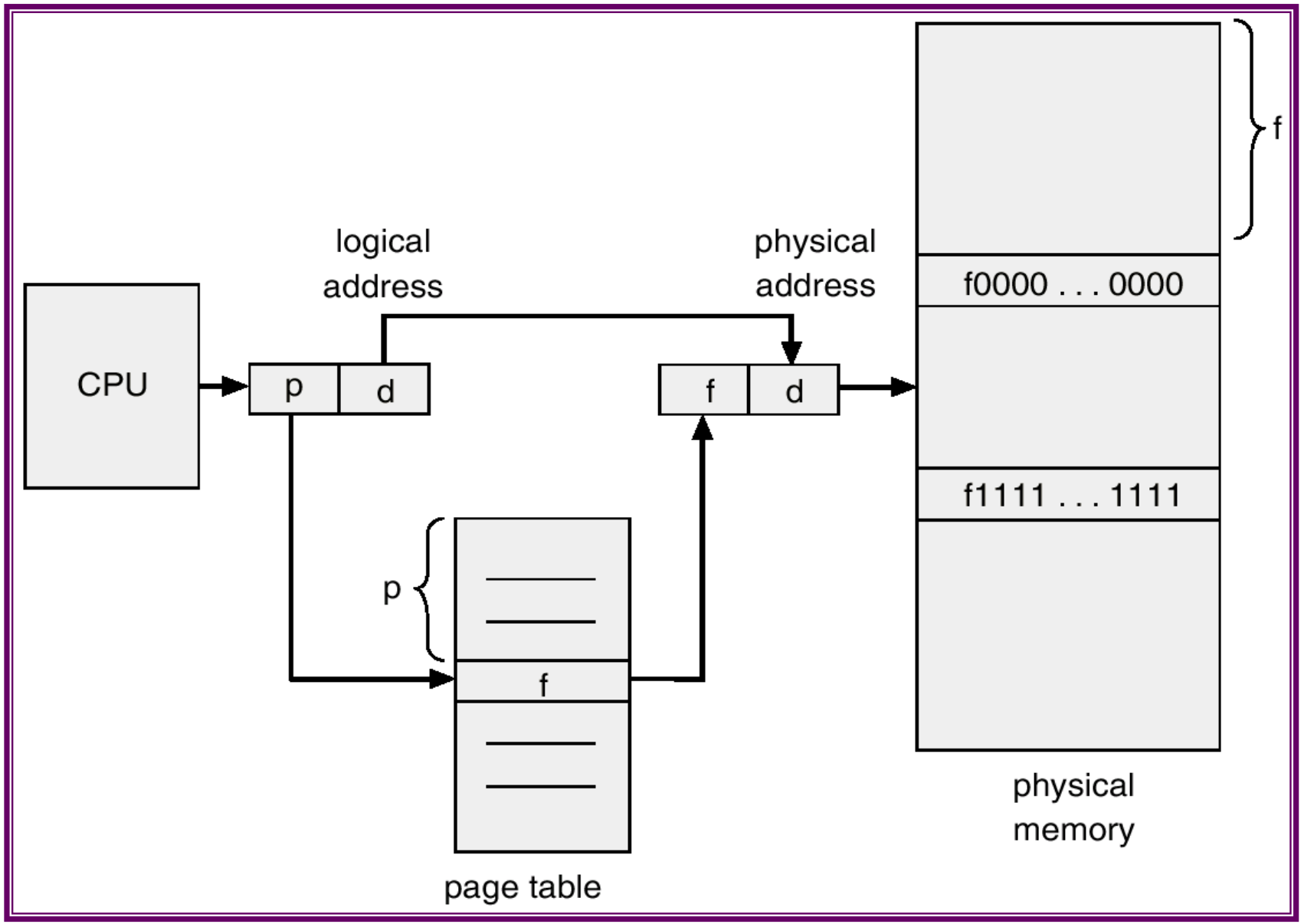
- Solution to external fragmentation
  - Permit the logical address space of the process to be non contiguous, allowing a process to be allocated physical memory whenever the later is available.
  - Paging is a solution.
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
  - Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
  - 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.
- Internal fragmentation.

Address Translation Scheme

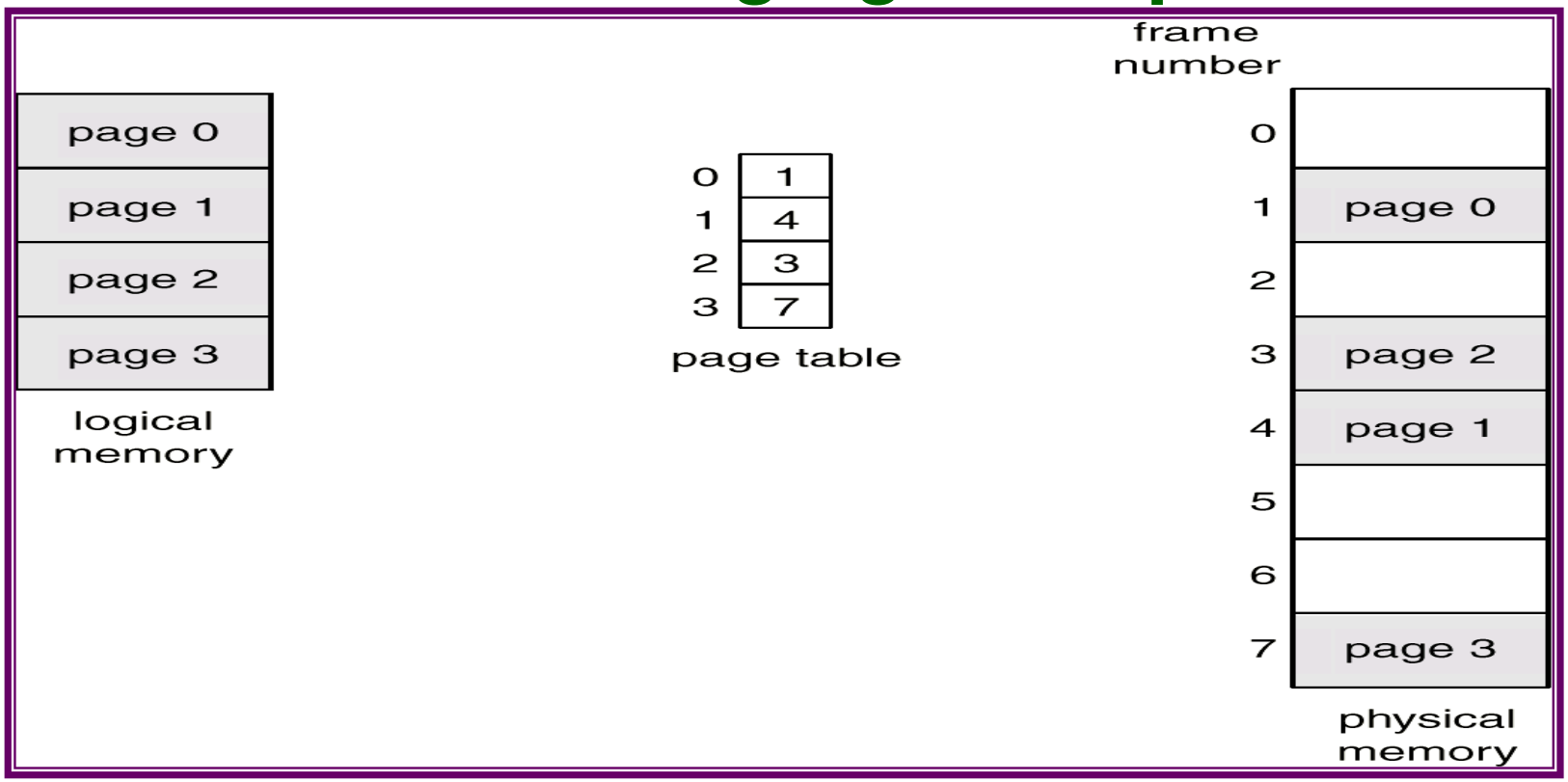
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.
- Page number is an index to the page table.
- The page table contains base address of each page in physical memory.
- The base address is combined with the page offset to define the physical address that is sent to the memory unit.
- The size of a page is typically a power of 2.
  - 512 –8192 bytes per page.
- The size of logical address space is 2^m and page size is 2^n address units.
- Higher m-n bits designate the page number
- n lower order bits indicate the page offset.



Address Translation Architecture



Paging Example

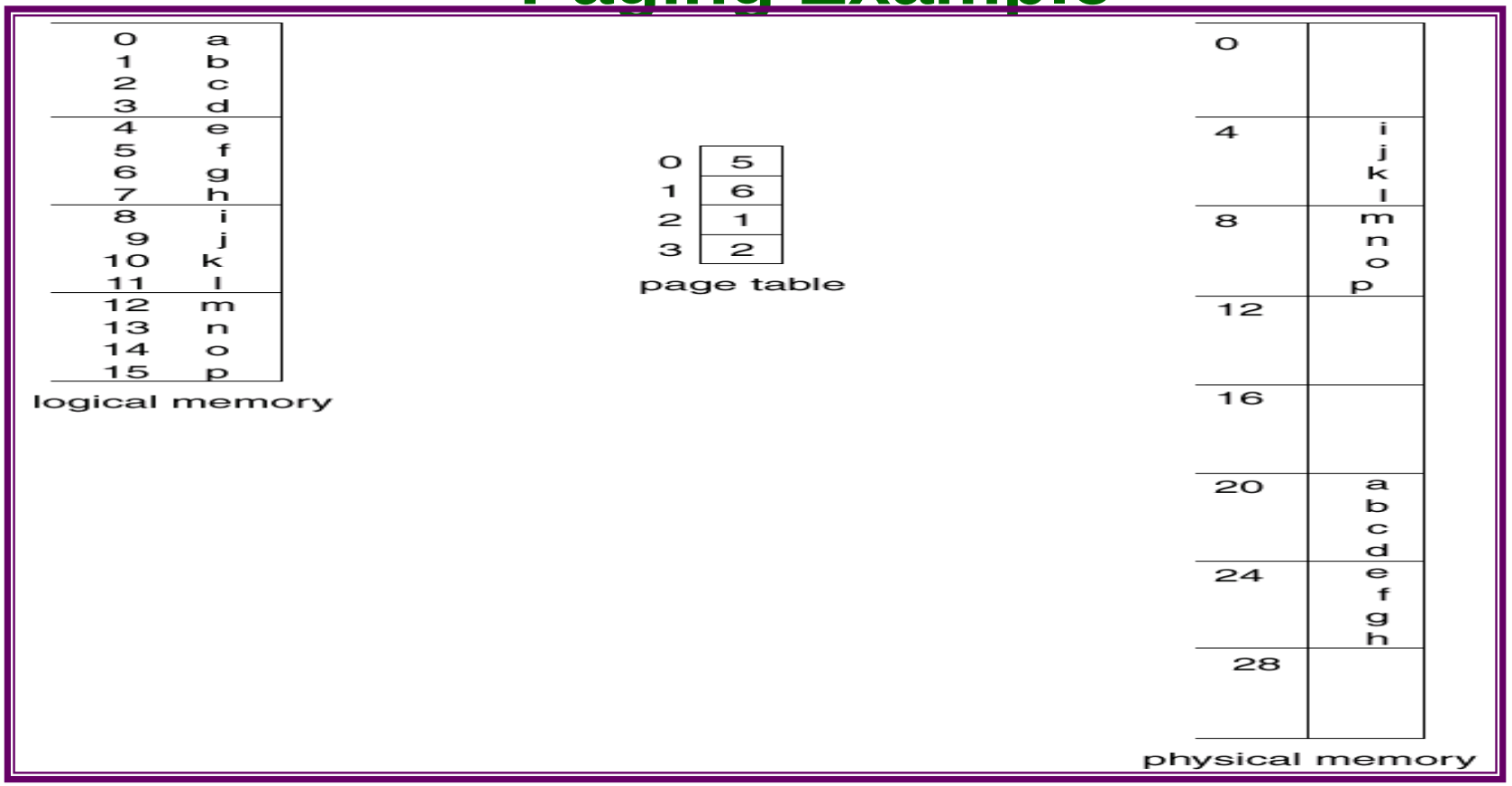


- Page size= 4 bytes; Physical memory=32 bytes (8 pages)
- Logical address “x” maps to

(frame number of the corresponding page*page size)+x

- Logical address 0 maps 1*4+0=4
- Logical address 3 maps to= 1*4+3=7
- Logical address 4 maps to =4*4+0=16
- Logical address 13 maps to= 7*4+1=29.

Paging Example



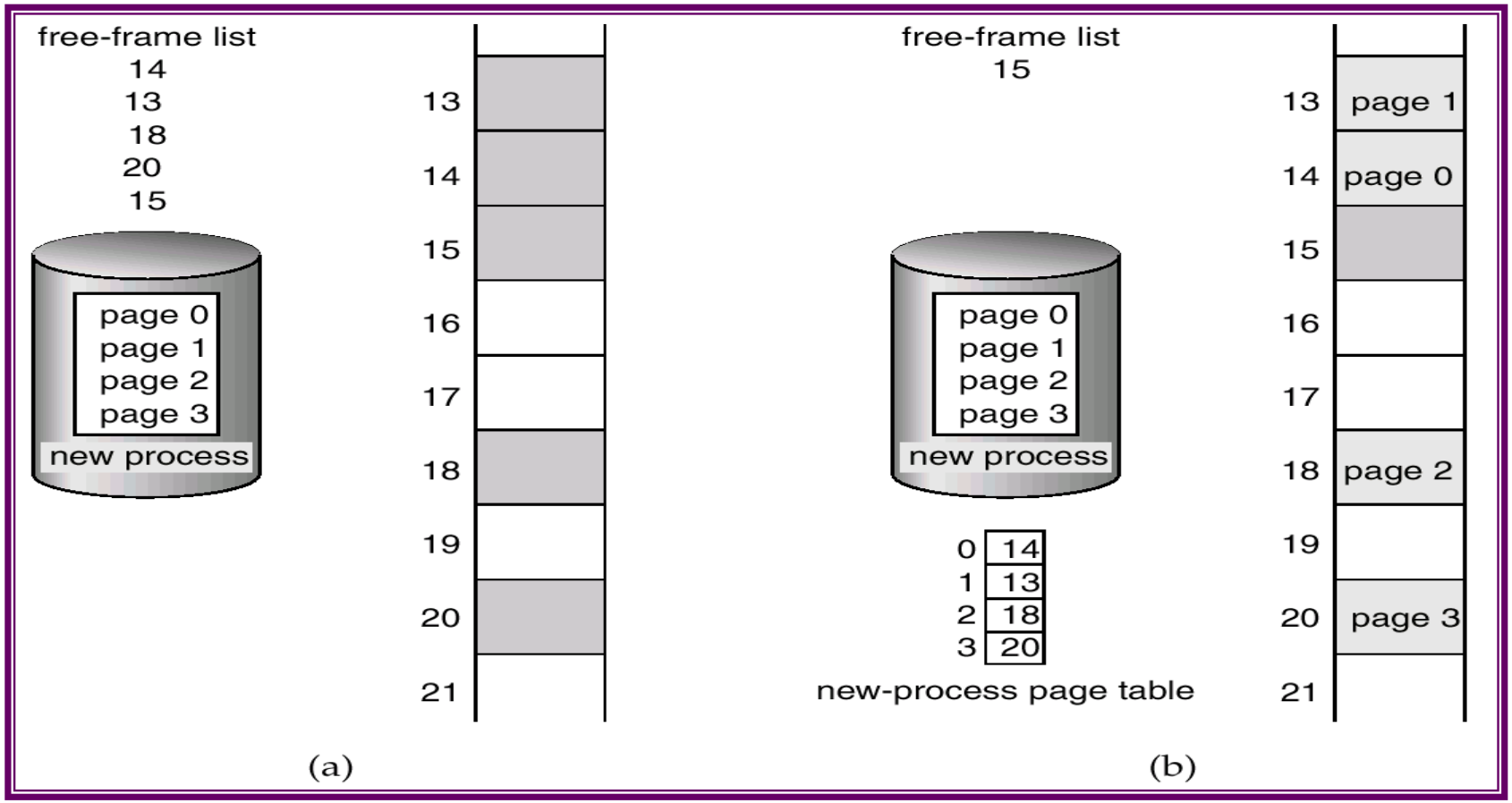
- Page size= 4 bytes; Physical memory=32 bytes (8 pages)
- Logical address “x” maps to

(frame number of the corresponding page*page size)+x

- Logical address 0 maps 5*4+0=20
- Logical address 3 maps to= 5*4+3=23
- Logical address 4 maps to =6*4+0=24
- Logical address 13 maps to= 2*4+1=9.

Free Frames

- When a process arrives the size in pages is examined
- Each page of process needs one frame.
- If n frames are available these are allocated, and page table is updated with frame number.



Before allocation

After allocation

More about Paging

- Paging separates user's view of memory and the actual physical memory.
- User program views memory as single contiguous space.
- In fact, user program is scattered throughout physical (main) memory.
- OS maintains the copy of the page table for each process. This copy is used to translate logical addresses to physical addresses.

- With paging we have no external fragmentation.
- We may have some internal fragmentation.
- In general, page sizes of 2K or 4K bytes.

Two issues

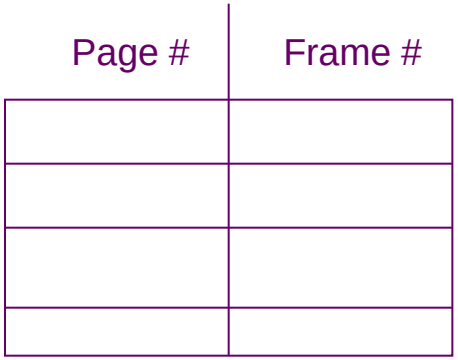
- Speed
- Space

Implementation of Page Table

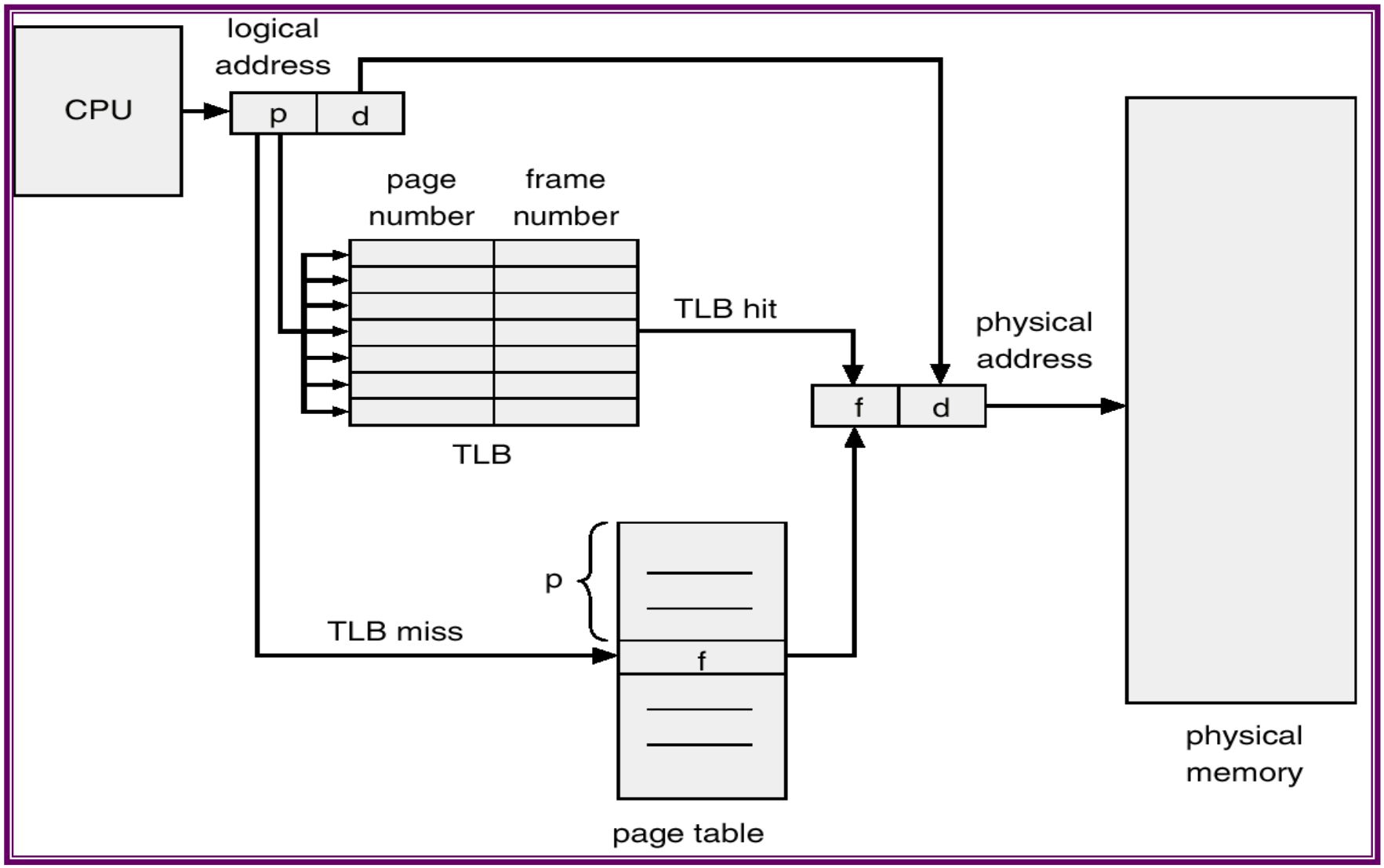
- Two options: Page table can be kept in registers or main memory
- Page table is kept in main memory due to bigger size.
  - Ex: address space = 2^{32} to 2^{64}
  - Page size= 2^{12}
  - Page table= 2^{32} / 2^{12}=2^{20}
  - If each entry consists of 4 bytes, the page table size = 4MB.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- PTBR, and PRLR are maintained in the registers.
- Context switch means changing the contents of PTBR and PRLR.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
  - Memory access is slowed by a factor of 2.
  - **Swapping might be better !**
- 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)*

Translation look-aside buffer (TLB) to address Speed issue

- TLB is an associative memory – parallel search
- A set of associative registers is built of especially high-speed memory.
- Each register consists of two parts: key and value
- When associative registers are presented with an item, it is compared with all keys simultaneously.
- If corresponding field is found, corresponding field is output.
- Associative registers contain only few of page table entries.
  - When a logical address is generated it is presented to a set of associative registers.
  - If yes, the page is returned.
  - Otherwise memory reference to page table mode. Then that page # is added to associative registers.
- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory
- It may take 10 % longer than normal time.
- % of time the page # is found in the associative registers is called hit ratio.



Paging Hardware With TLB



Address Space Identifiers (ASIDs)

- Some TLBs store ASIDs in each TLB entry.
- ASID uniquely identifies each process.
- ASID mechanism allows the TLB to contain entries for several different processes simultaneously.
- If TLB does not support separate ASIDs, every time new page table is selected, the TLB must be flushed (erased) to ensure that next executing process does not use wrong translation information.

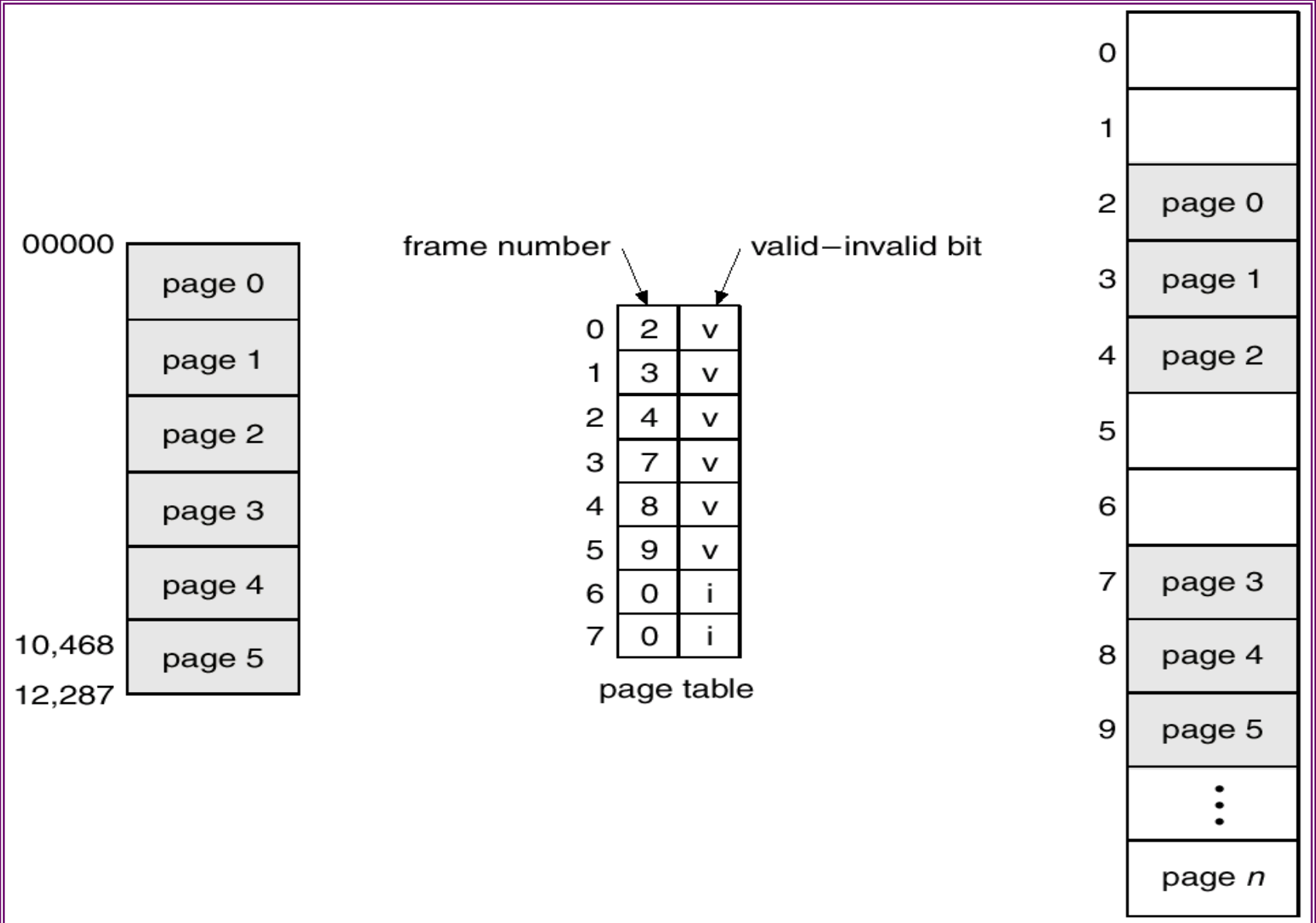
Effective Access Time

- If it takes 20 nsec to search the associative registers and 100 nsec to access memory and hit ratio is 80 %, then,
  - Effective access time=hit ratio*Associate memory access time +miss ratio* memory access time.
  - 0.80 * 120+0.20*220=140 nsec.
    - 40 % slowdown.
- For 98-percent hit ratio, we have
  - Effective access time= 98*120+0.02*220
    = 122 nanoseconds
    = 22 % slowdown.

Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- *One bit can be assigned to indicate read and write or read-only*
  - *An attempt to write read-only page causes hardware trap to OS.*
- *More bits can be added to provide read-only, read-write or execute-only protection.*
- *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.
  - Illegal addresses can be trapped with valid/invalid bit.
  - Some systems implement page table length register (PTLR) in case of internal fragmentation.
    - PTLR is used to check whether address is in valid range or not.

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



Page Table: Space Issue

- Page table:
  - Each process has a page table associated with it.
  - The page table has a slot for each logical address regardless of its validity.
  - Since table is sorted by virtual address, OS calculates the value directly.
  - **However, it consumes more space.**
- Example regarding space issue: Modern computer systems support a very large address space: 2^{32} to 2^{64}.
  - Consider a system with 32-bit logical address space. If the page size is 4K, then the page table size is 2^{12} entries.
  - If each entry consists of 4 bytes, then each process may need 4 mega bytes for a page table.

Page Table Structure

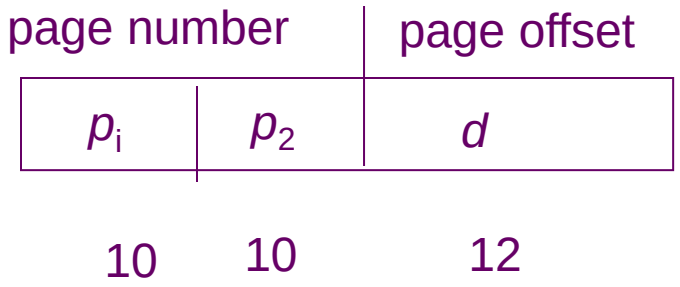
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Modern computer systems support a very large address space: 2^{32} to 2^{64}.
  - Consider a system with 32-bit logical address space. If the page size is 4K, then the page table size is 2^{12} entries.
  - If each entry consists of 4 bytes, then each process may need 4 mega bytes.
- Solution: Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.
  - Page table itself is paged.

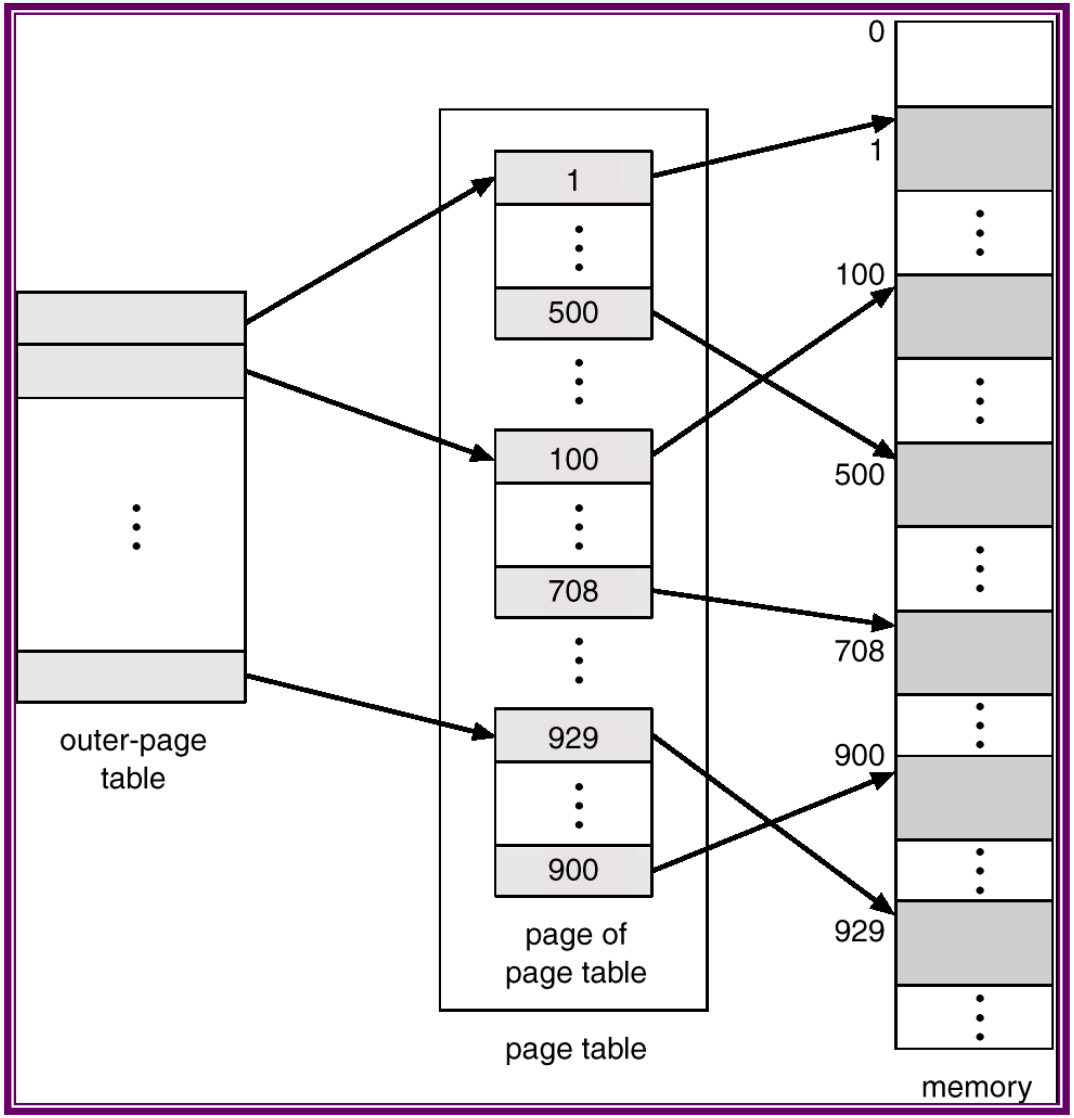
Two-Level Paging Example

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



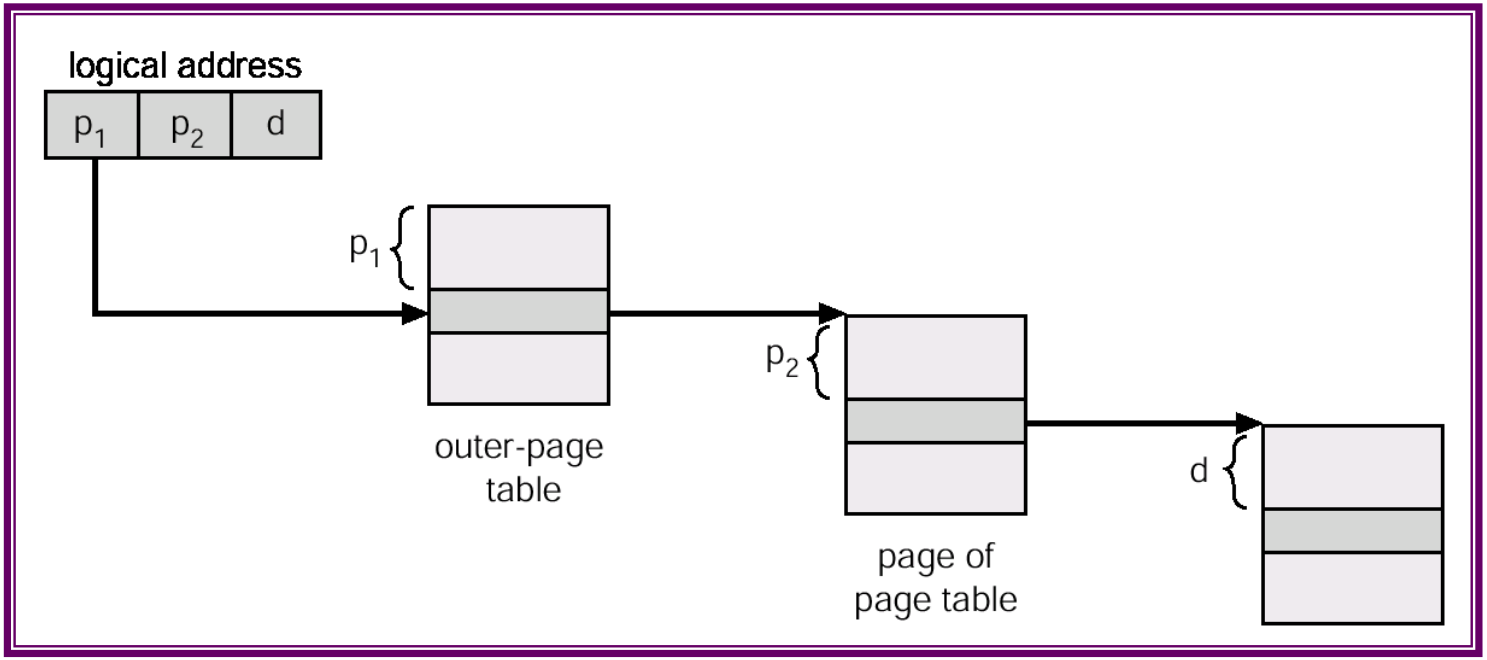
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme



Address-Translation Scheme

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



Address-Translation Scheme

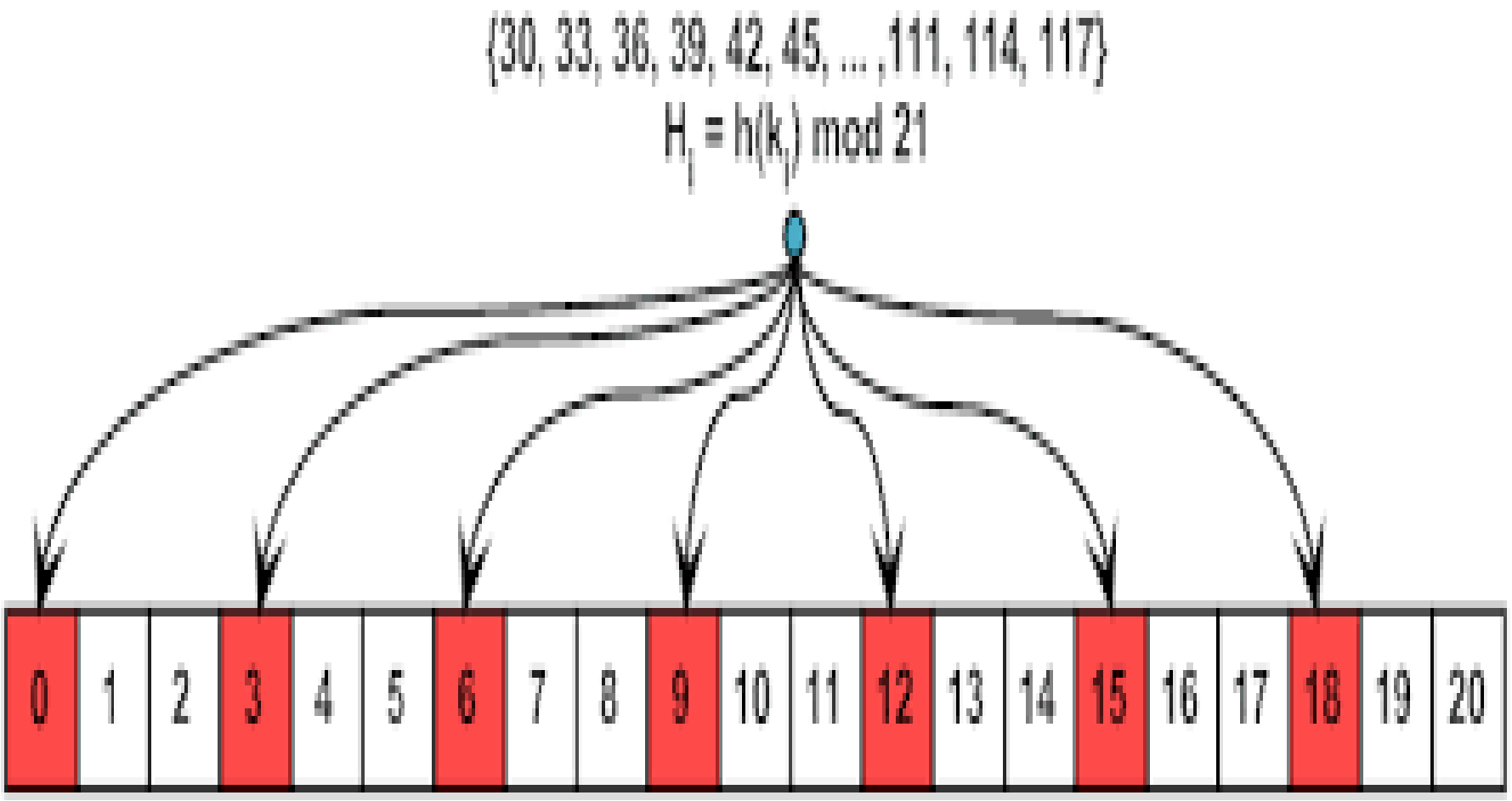
- VAX 32-bit architecture supports a variation of two-level paging scheme
- SPARC 32 bit architecture supports 3-level paging scheme.
- 32-bit Motorola 68030 supports a 4-level paging scheme.
- Tradeoff: size versus speed
  - For 64-bit architectures hierarchical page tables are inappropriate.
  - For example, the 64-bit UltraSPARC would require seven levels of paging.

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.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

About Hash Tables

- A hash function takes a search key as an argument and computes an integer range 0 to B-1, where B is a number of buckets.
  - A bucket array, which is an array indexed from 0 to B-1, holds headers of B linked lists, one for each bucket of the array.
  - If the record has a search key K, we store the record by linking it to the bucket list for the bucket numbered h(K), where h is the hash function.
- Common hash function
  - Remainder of K/B


{30, 33, 36, 39, 42, 45, ... ,111, 114, 117}
$H_i = h(k_i) \bmod 21$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Secondary Storage Hash Tables

- Bucket contains sequence of blocks
- First block can be found given “i”.
- If a bucket overflows, a chain of overflow blocks are added.

Bucket

- Bucket is a logical concept
  - It is the concept of hashing
  - Physically implemented as a sequence of blocks
  - Ideal= 1 block

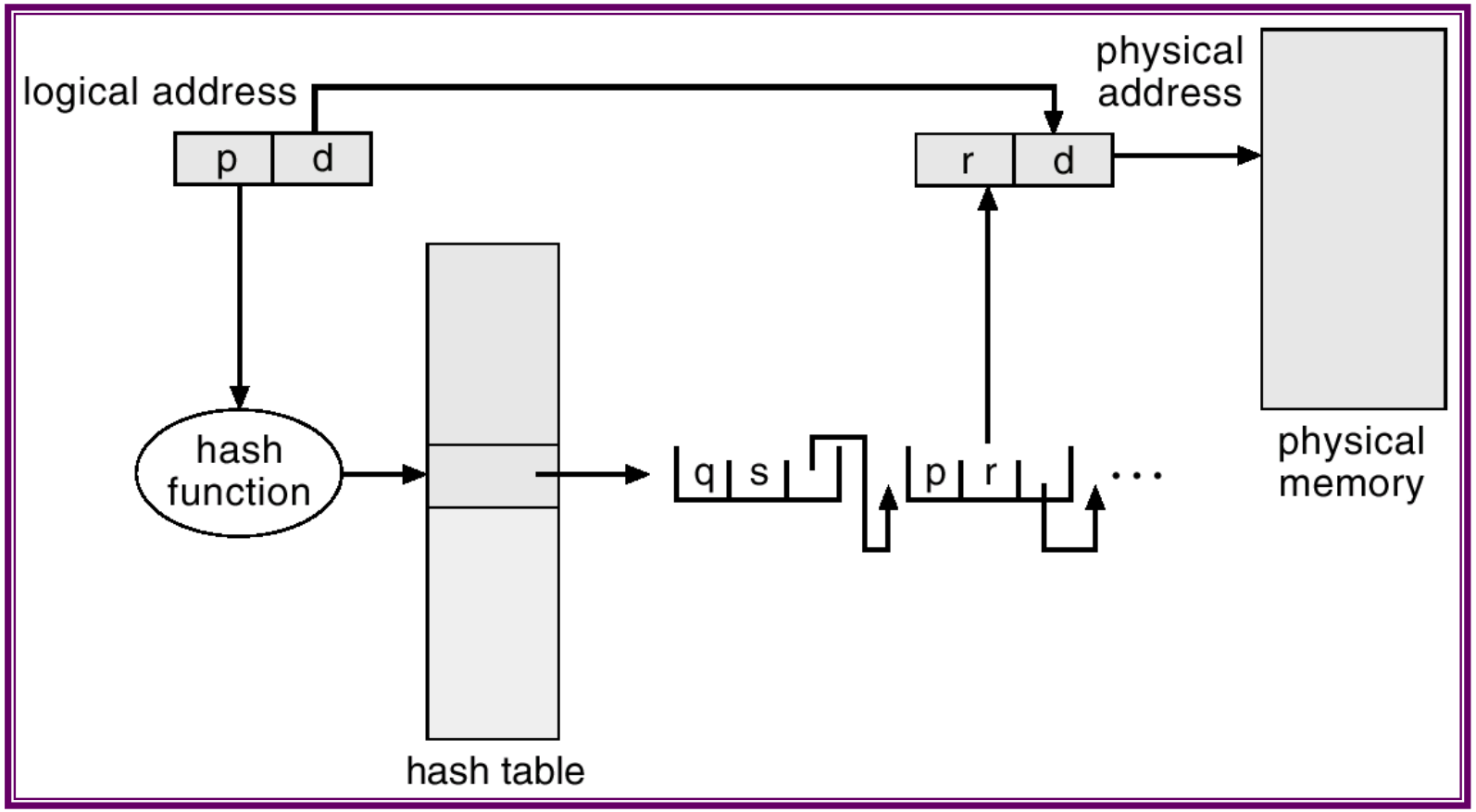
Insertion into a Hash Table

- When a new record with search key K must be inserted, we compute h(K).
- If the bucket number h(K) has the space, we insert the record into the block for this bucket or into one of the chain of blocks.
- If there is no space, we add extra block.

Hash-table deletion

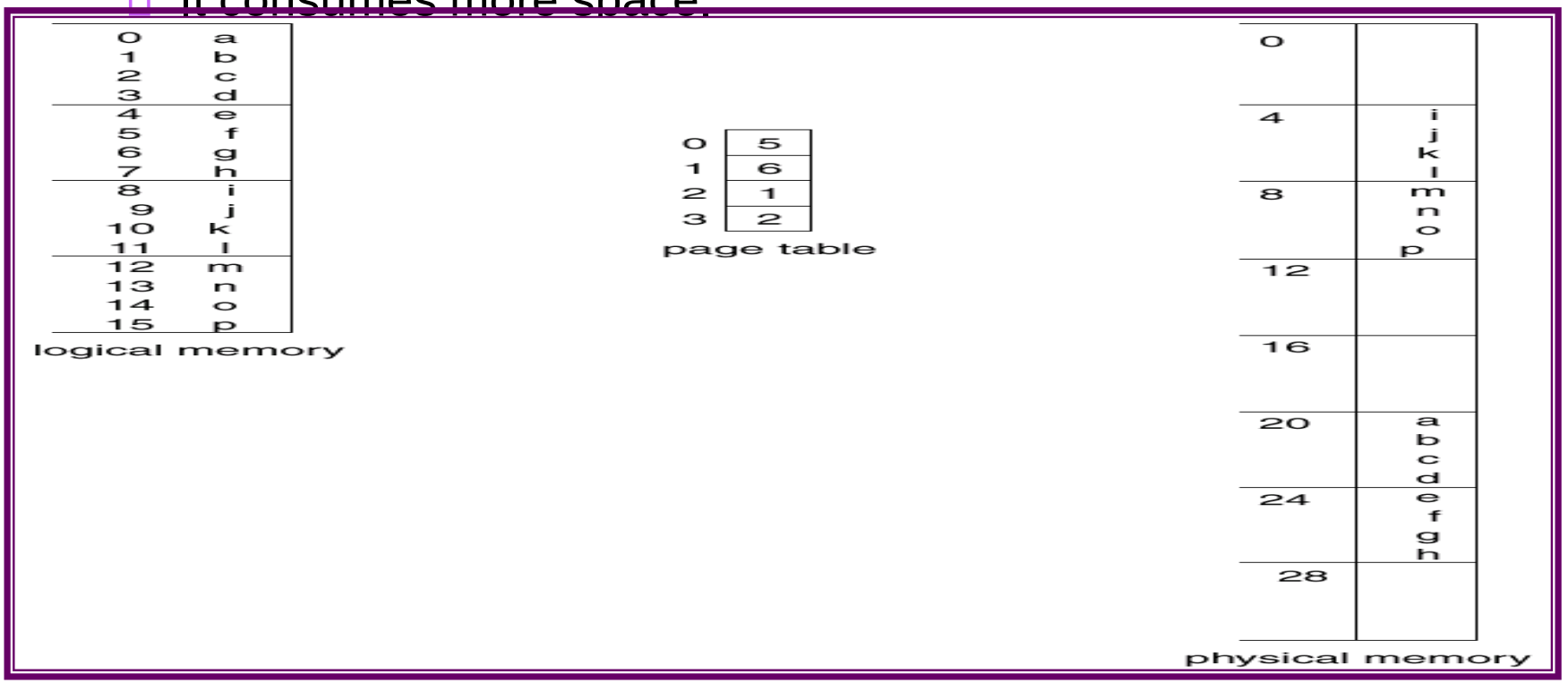
- Go to the bucket number h(K) and search for records with that search key. Delete if we find any data.
  - Consolidate (optional)

Hashed Page Table



Inverted Page Table

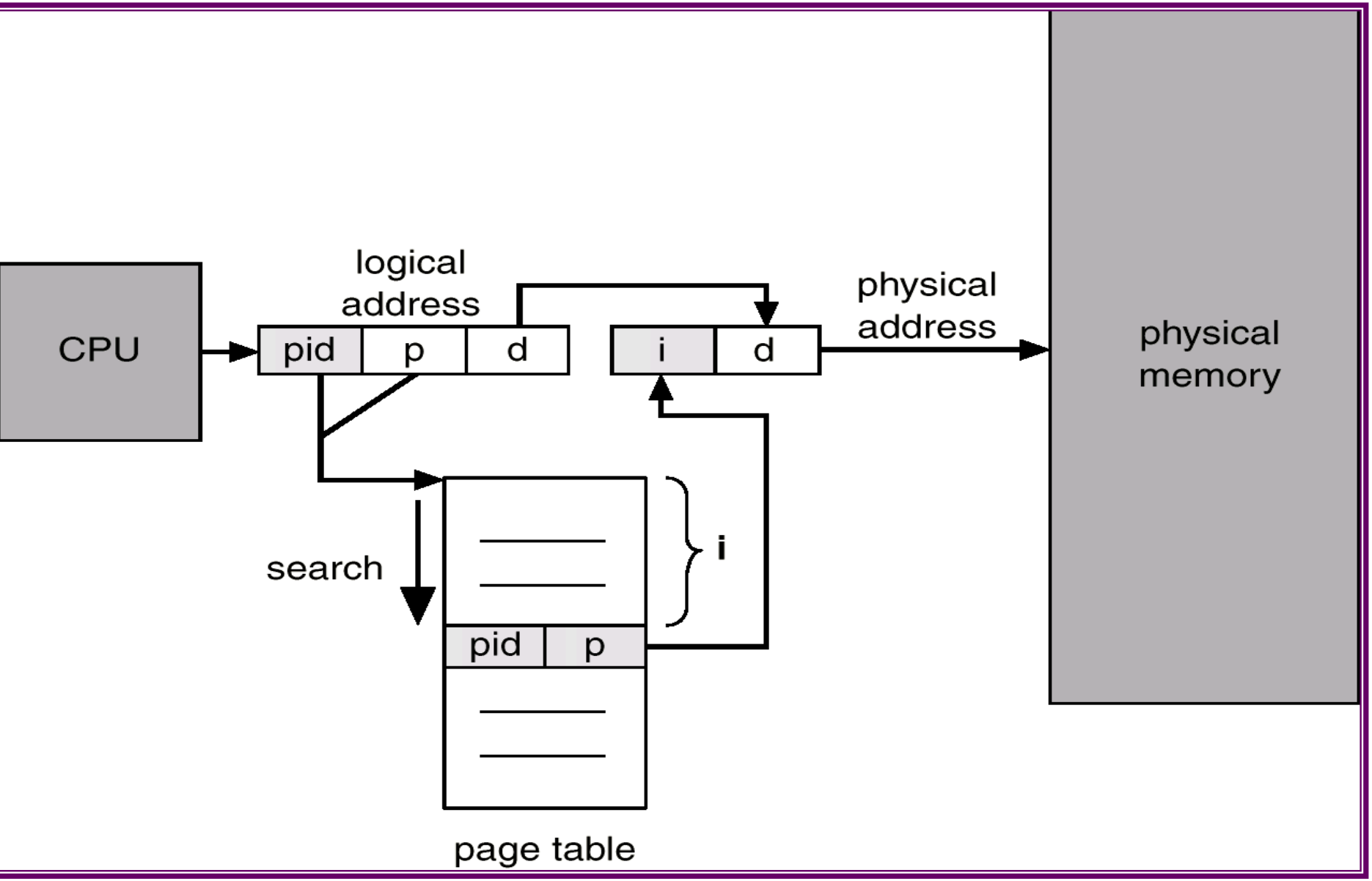
- Page table:
  - Each process has a page table associated with it.
  - The page table has a slot for each logical address regardless of its validity.
  - Since table is sorted by virtual address, OS calculates the value directly.
  - It consumes more space.



Inverted Page Table

- Solution: Inverted page table
- 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.
- There is only one page table in the system.
- Each virtual address space in the the system consists of a triple <Process-id, page #, offset>
- When a memory reference occurs part of <process-id, page#> is presented to memory subsystem.
- The inverted page table is searched for a match.
- Use hash table to limit the search to one — or at most a few — page-table entries.
- It is implemented in 64-bit UltraSPARC and PowerPC.
- Decreases amount of memory but increases time needed to search the table.

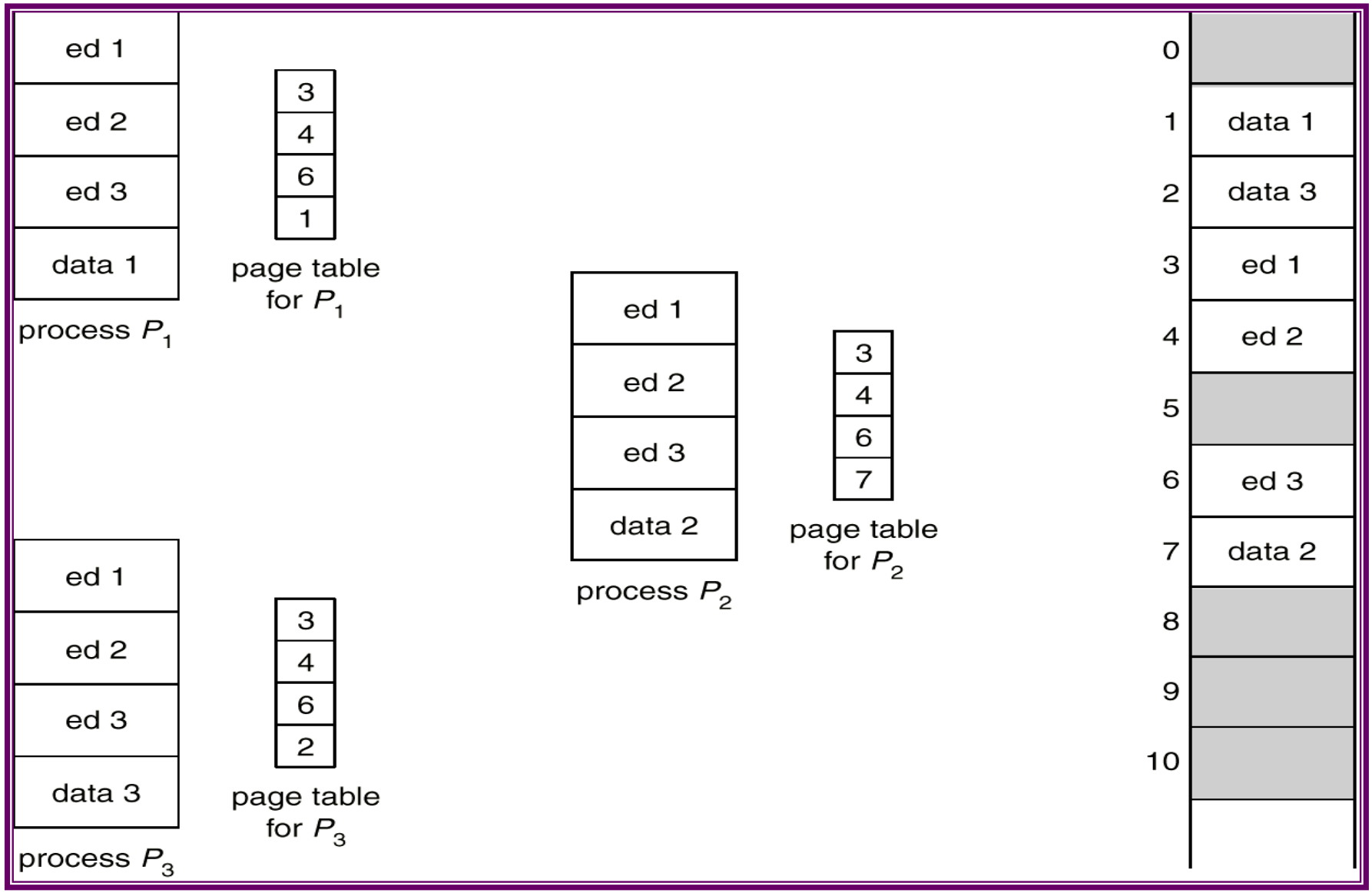
Inverted Page Table Architecture



Shared Pages

- Advantage of paging
  - Paging allows sharing of common code.
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in the same location in the logical address space of all processes.
- 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.
- For example if a system supports text editor and supports 40 users. If the text editor consists of 150K and 50K of data space we need 8000K for the 40 users.
- If the code reentrant it can be shared.
  - Reentrant code is non-self modifying code
  - It never changes during execution
  - Ex: compilers, window system, DBS and so on can be shared.

Shared Pages Example

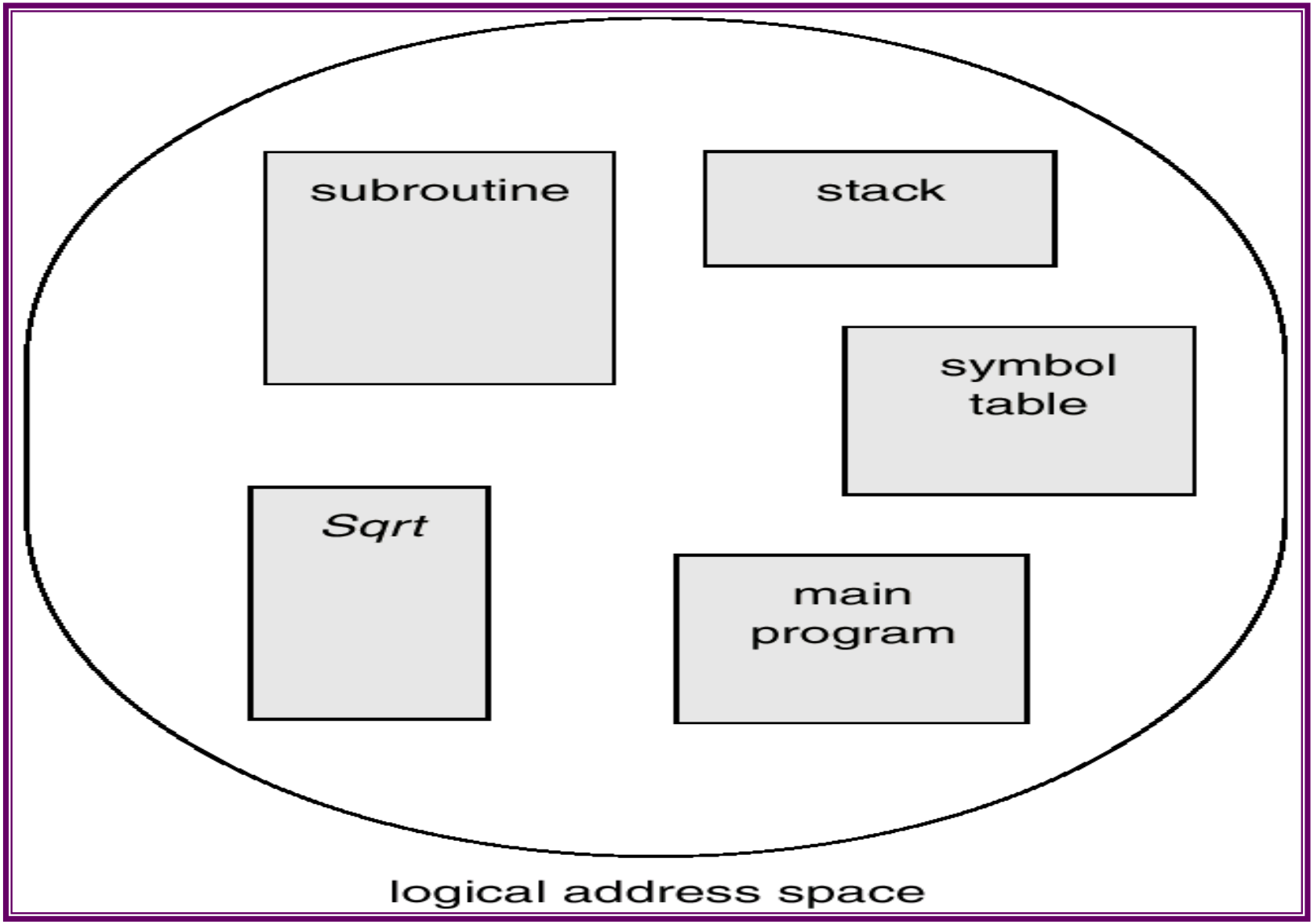


Segmentation

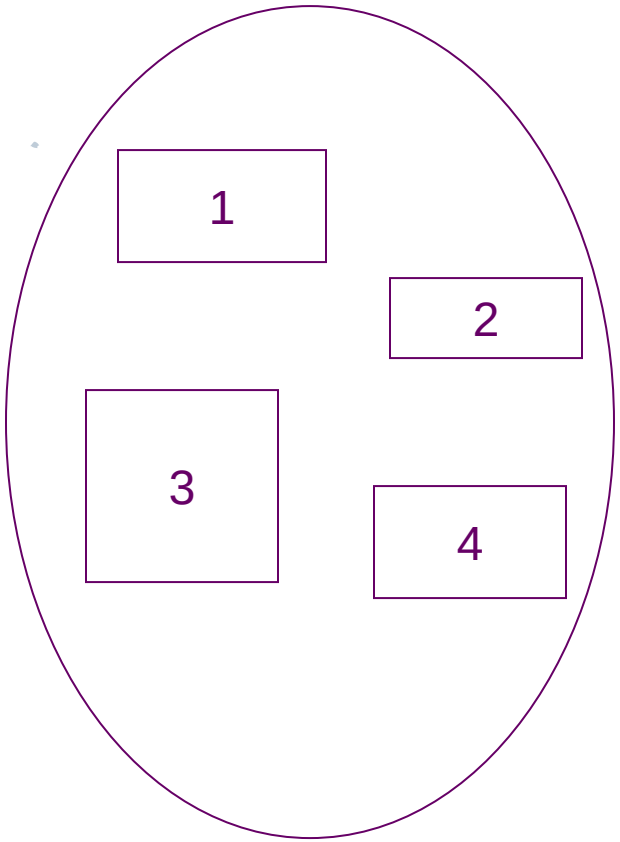
- Paging issues
  - Space: Size of page table
  - Searching time
- Segmentation is a memory-management scheme that supports user view of memory.
- Users prefer to view memory as a collection of variable-sized segments without any ordering.
- Logical address space is a collection of segments.
  - Each segment has name and length.
  - The address specify segment name and length.
- A program is a collection of segments. A segment is a logical unit such as:

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

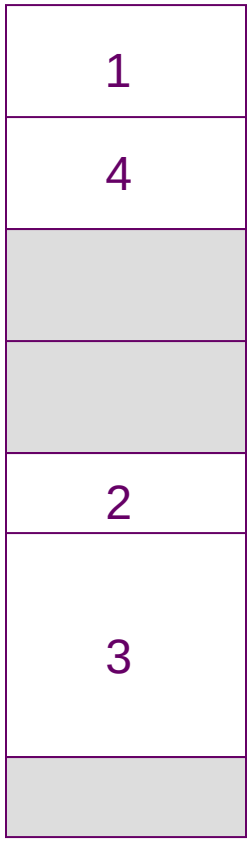
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *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.

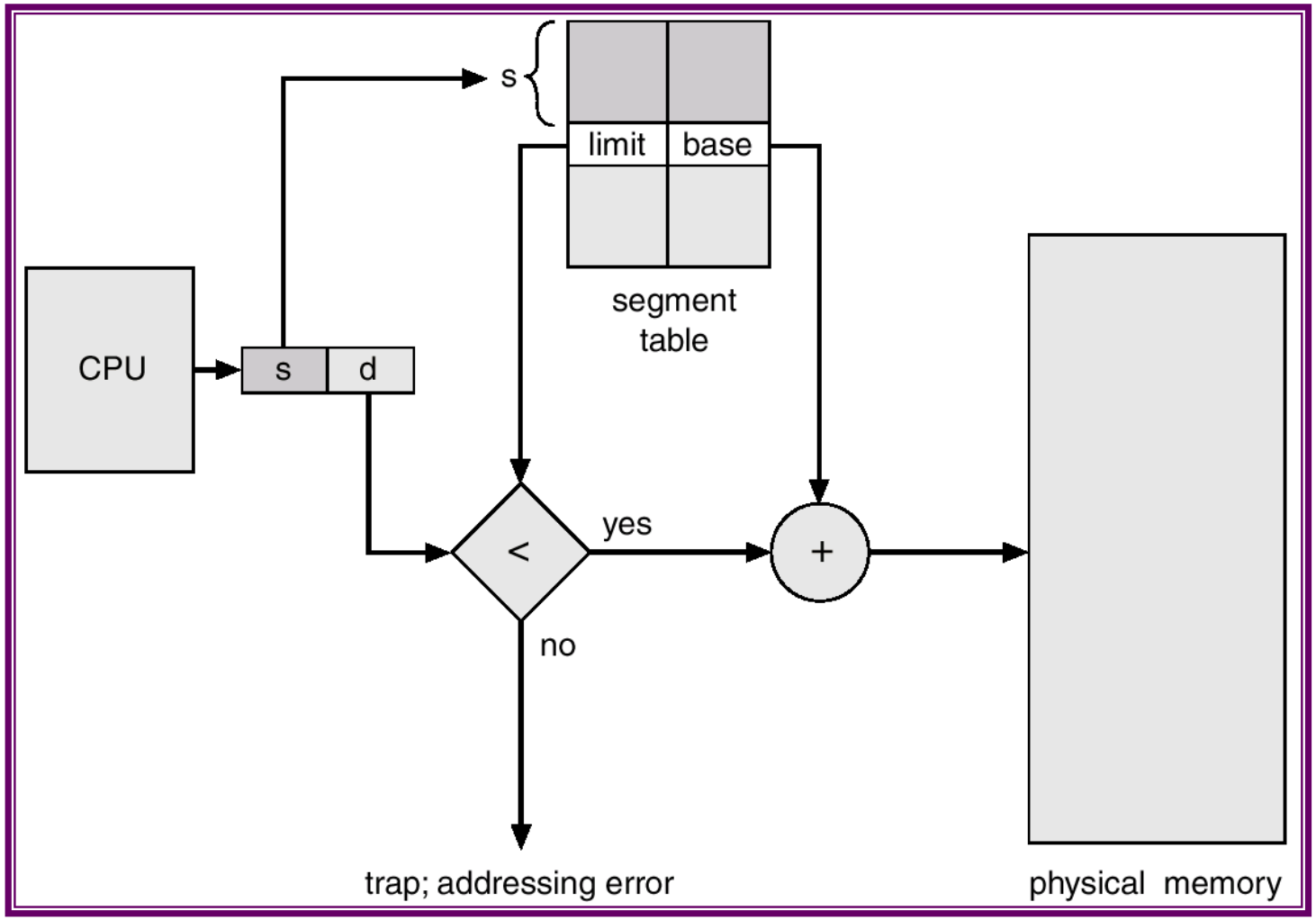
Segmentation Architecture (Cont.)

- Relocation.
  - dynamic
  - by segment table
- Sharing.
  - shared segments
  - same segment number
- Allocation.
  - first fit/best fit
  - external fragmentation

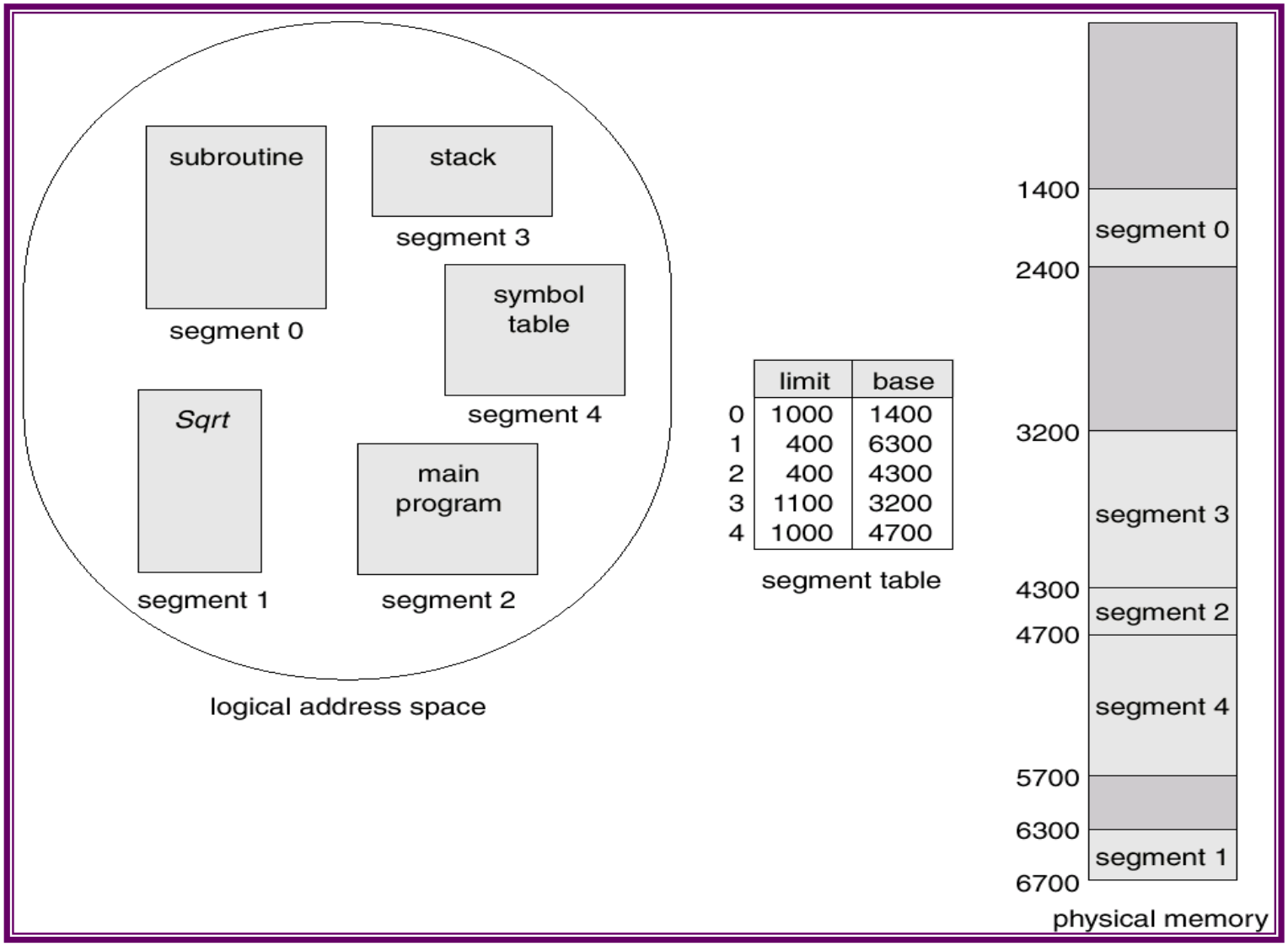
Segmentation Architecture (Cont.)

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

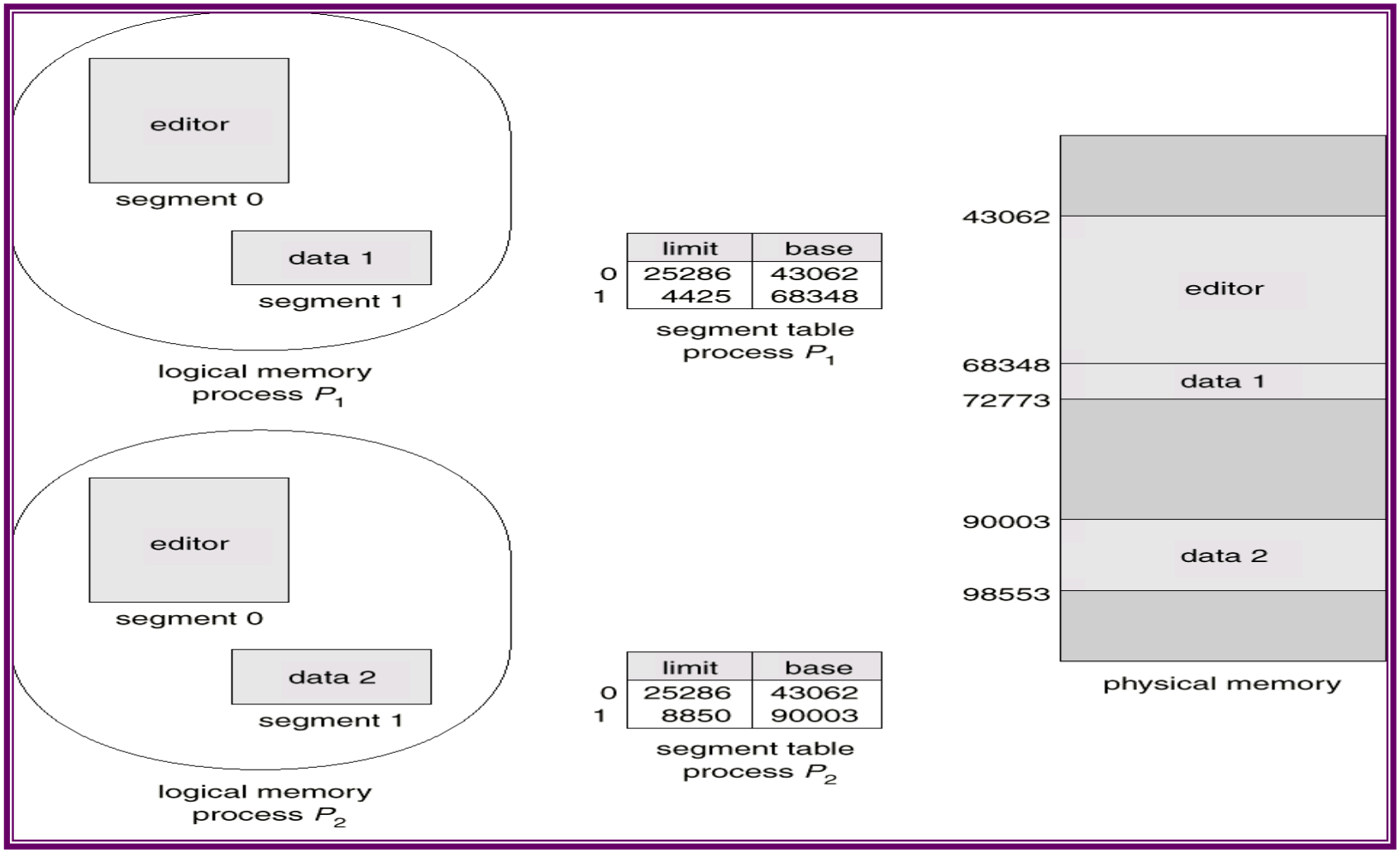
Segmentation Hardware



Example of Segmentation



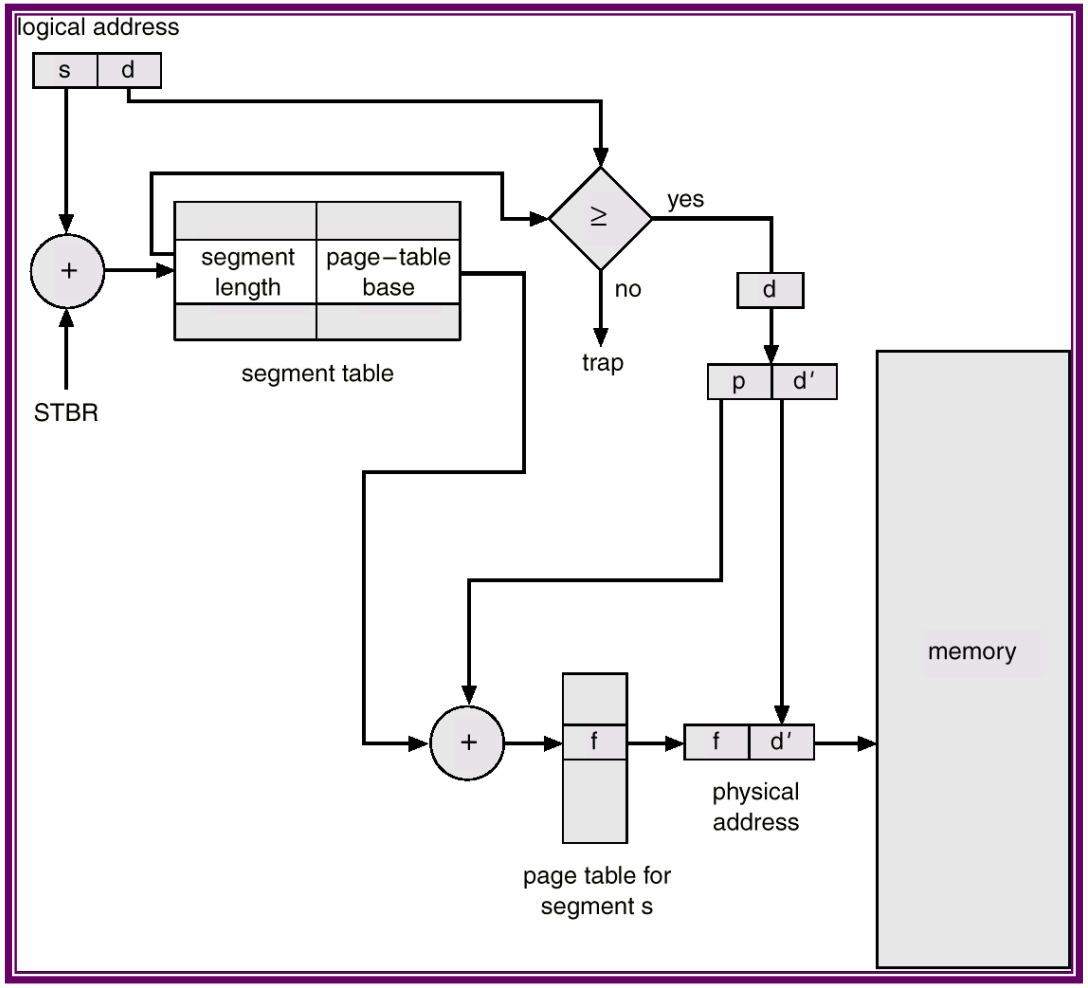
Sharing of Segments



Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

MULTICS Address Translation Scheme



Segmentation with Paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Intel 30386 Address Translation