Operating Systems (CSE531)
Lecture # 15

Manish Shrivastava

LTRC, IIIT Hyderabad

Memory Management

Manish Shrivastava

Why?

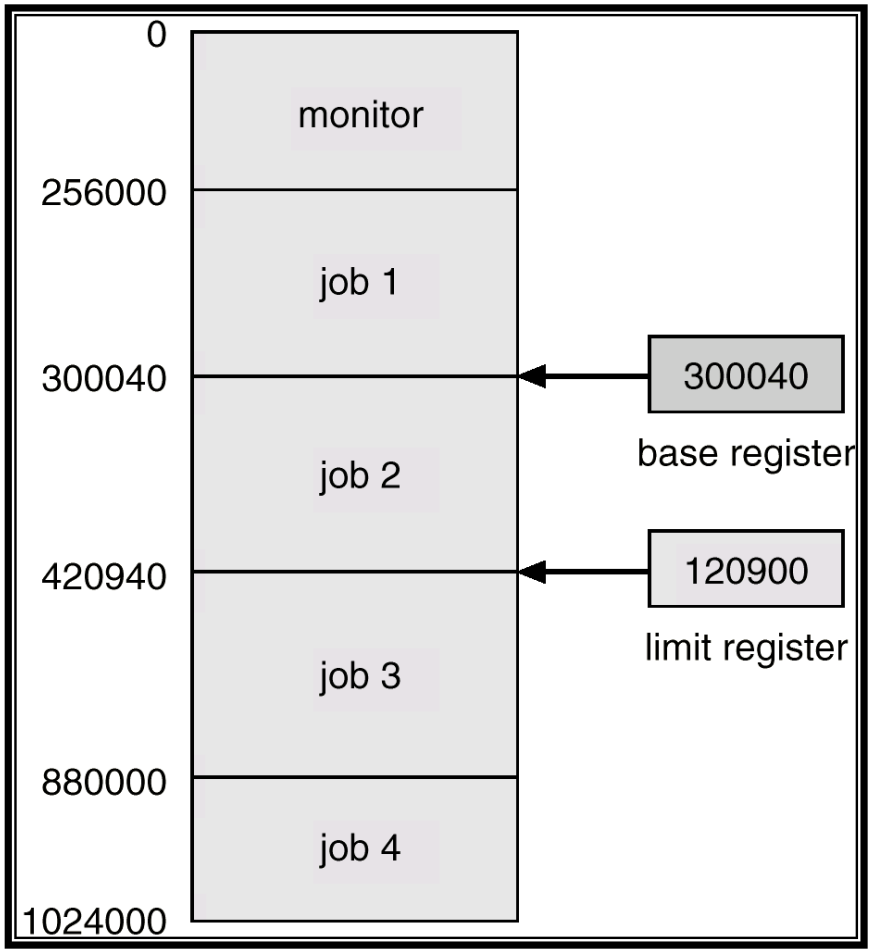
- The programs are in the Main memory
  - We need a memory management scheme to manage several processes in the main memory
- Main memory is very small
  - Secondary storage is used to support main memory
- We also need file management for on-line storage of access to both data and programs that reside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

Memory Protection

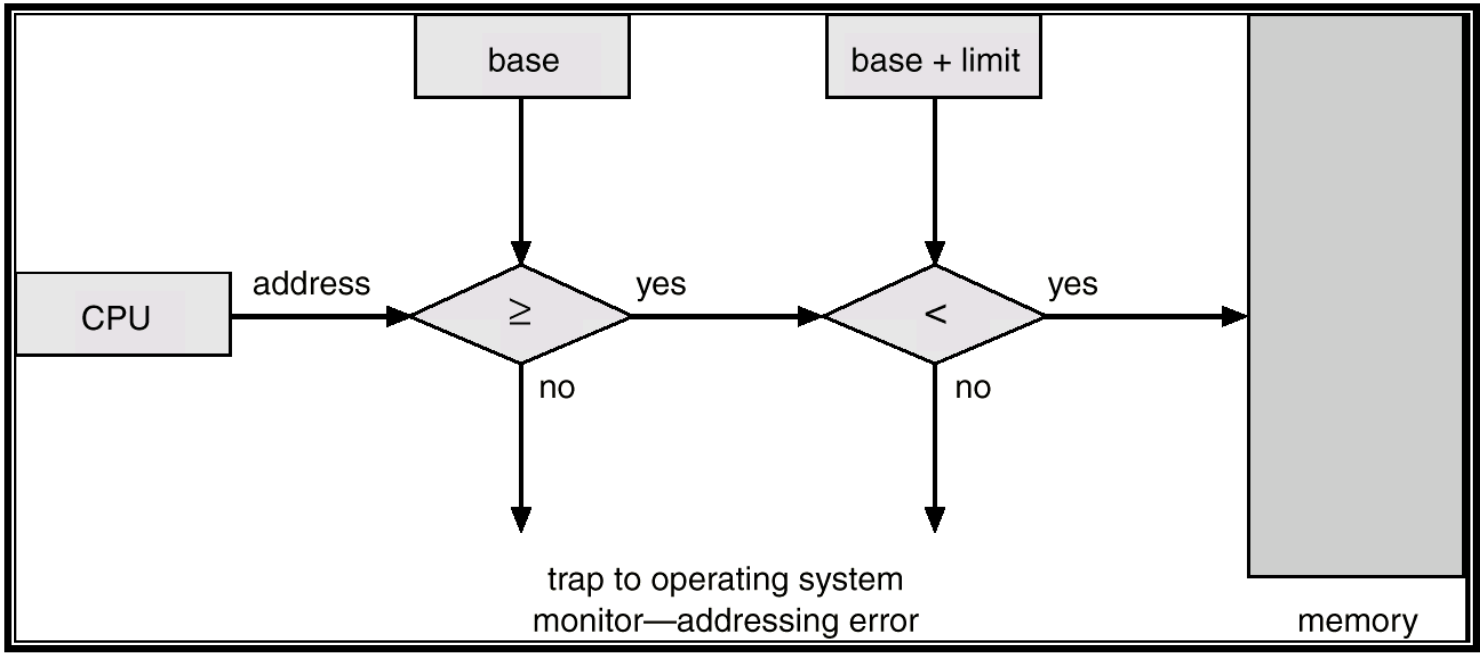
- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** – holds the smallest legal physical memory address.
 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

- 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.

Use of A Base and Limit Register



Hardware Address Protection



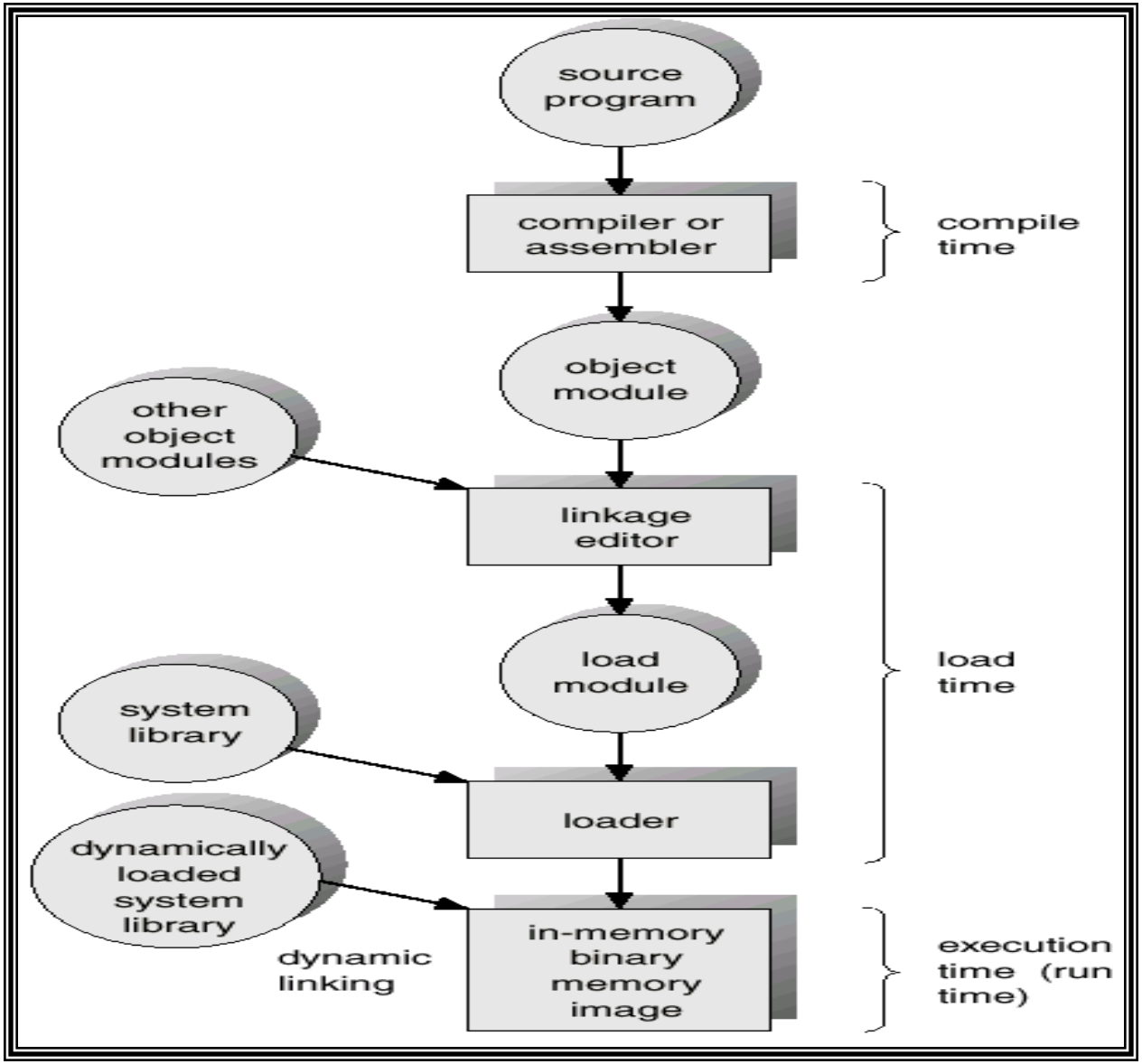
Memory Protection

- 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.

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.

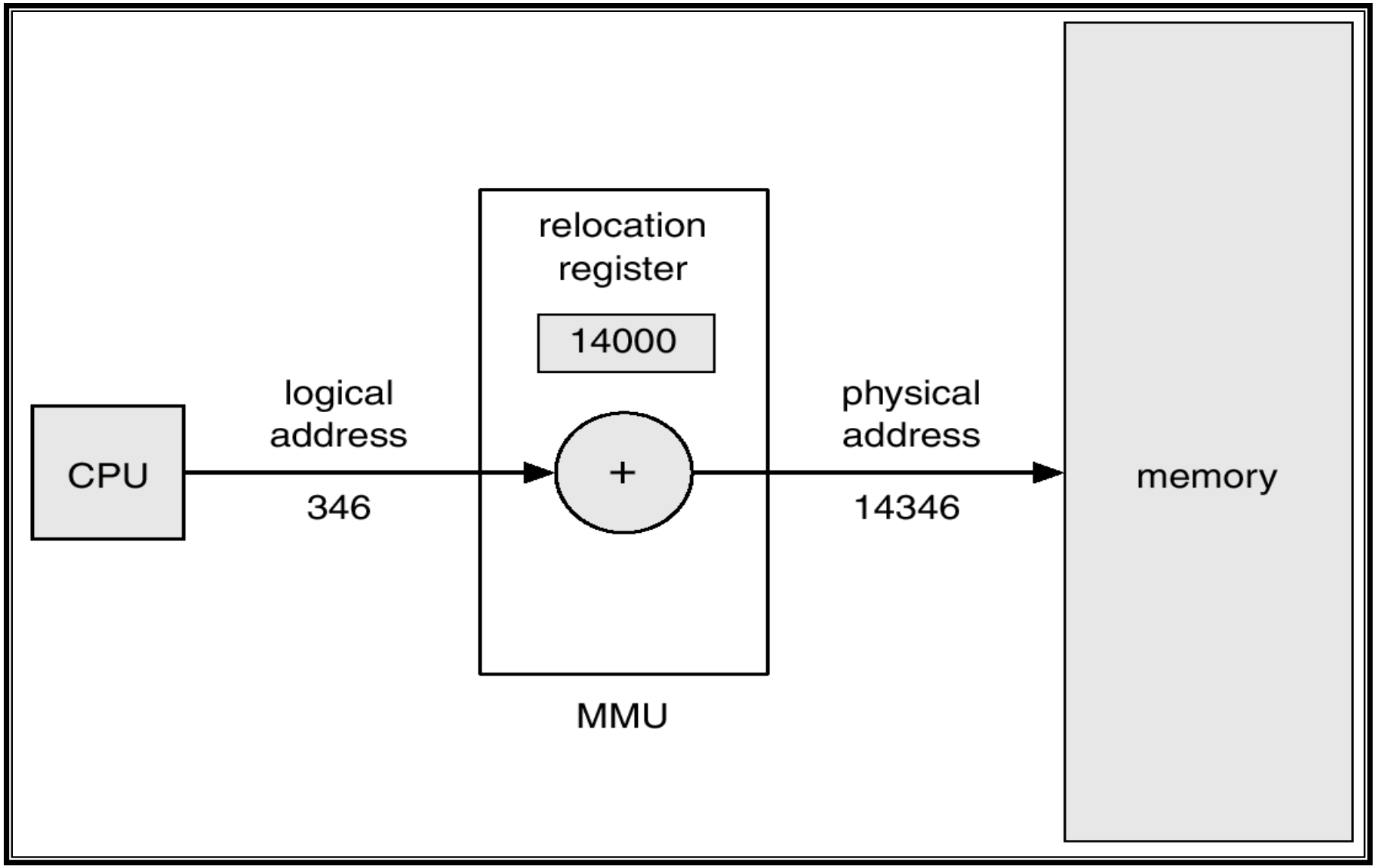
Logical vs. Physical Address Space

- 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.

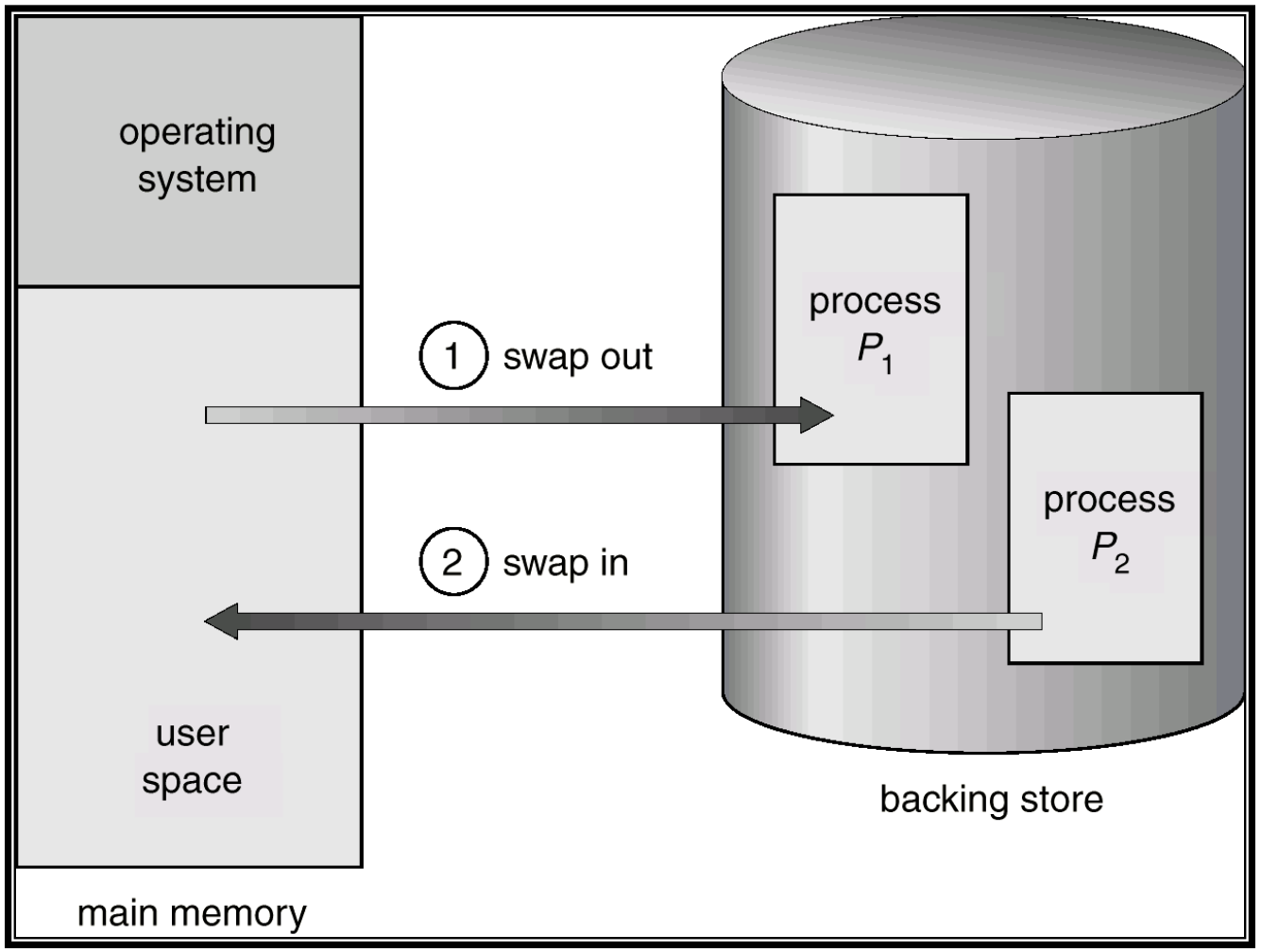
Dynamic relocation using a relocation register



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

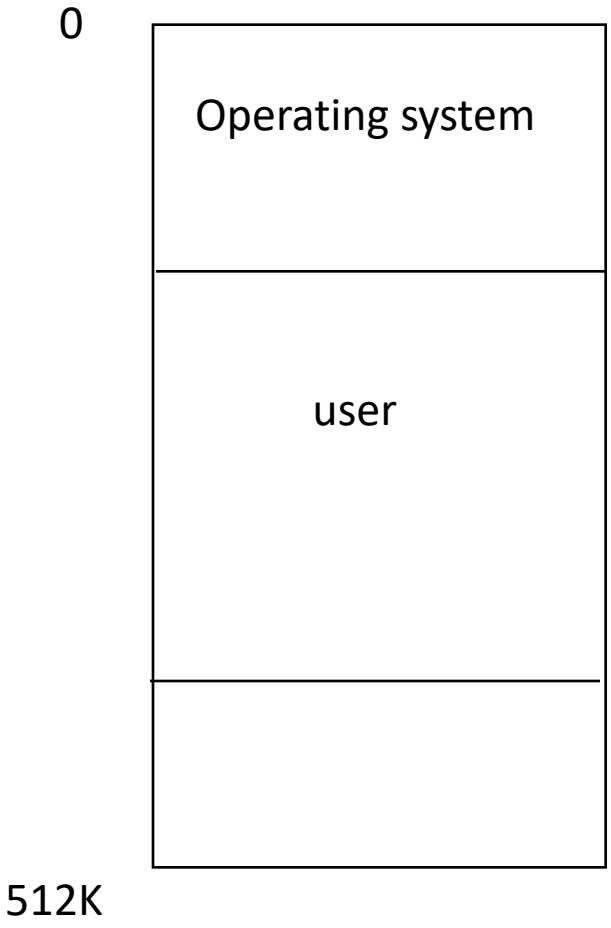


Swapping…

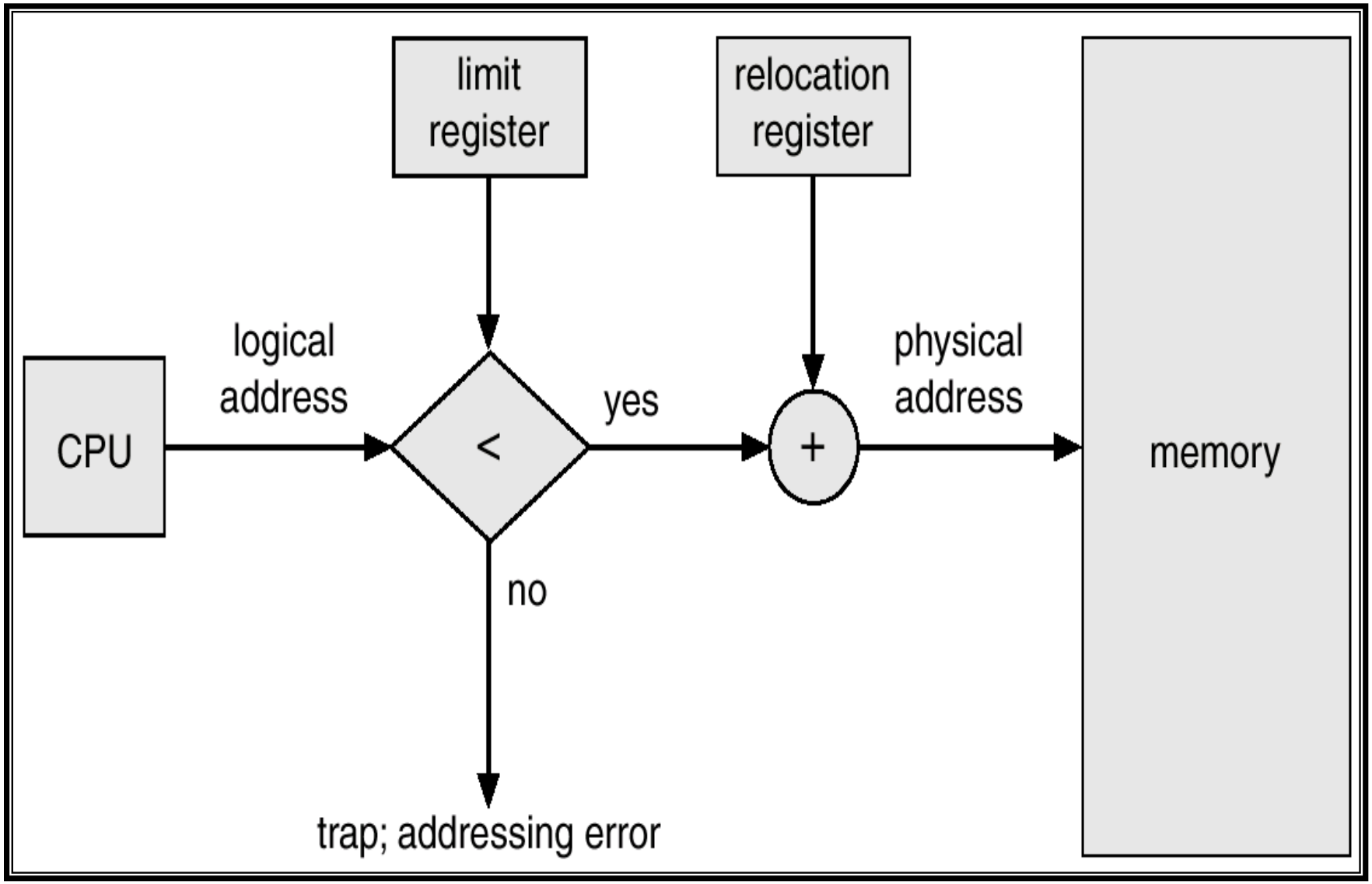
- A process should be idle to swap.
  - It should not have any pending I/Os.
- Solutions
  - Never swap a process with pending I/Os
    - Difficult if the process is accessing asynchronously.
  - Execute I/O operations into OS buffers.
    - Transfers between OS buffers to process occur only when the 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.

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

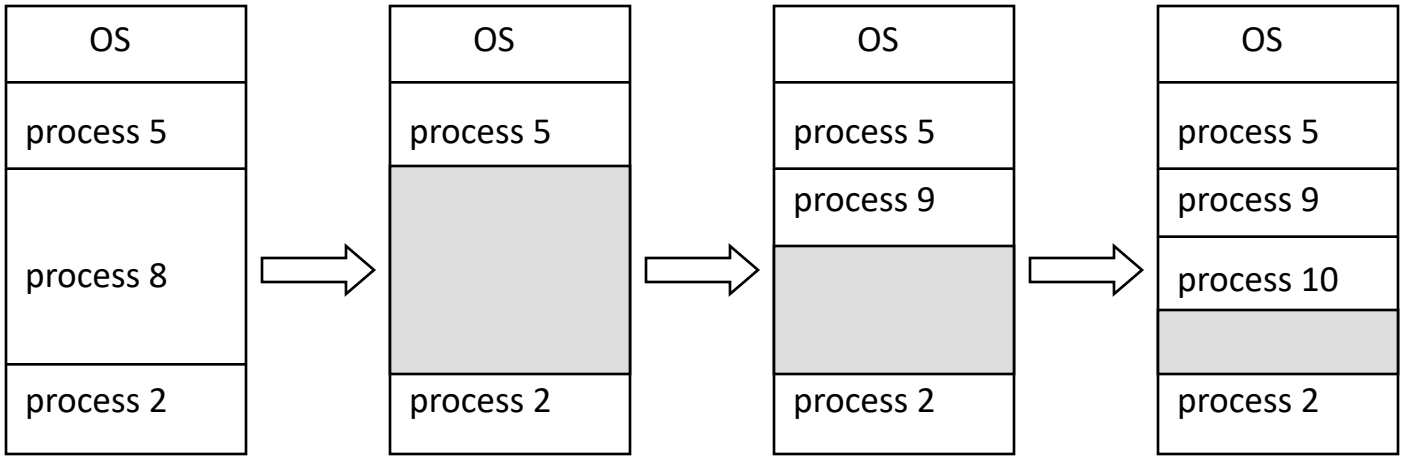


Contiguous Allocation (Cont.)

- 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.)

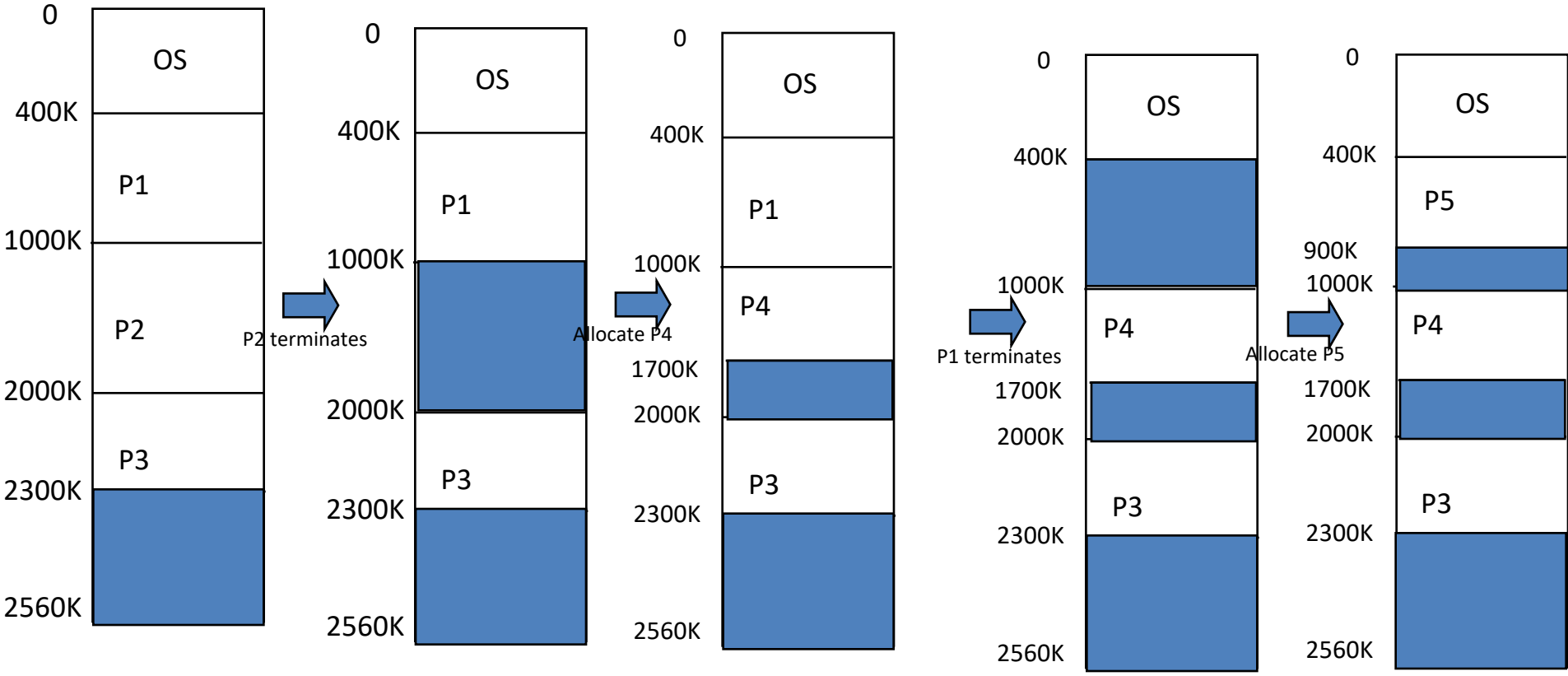
- At any time a set of holes of various sizes scattered throughout the memory
- A free partition large enough for the requesting process is searched.
- If the free partition is too large it is split into two: one is allocated to the arriving process and the other is returned to the set of partitions.
- 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

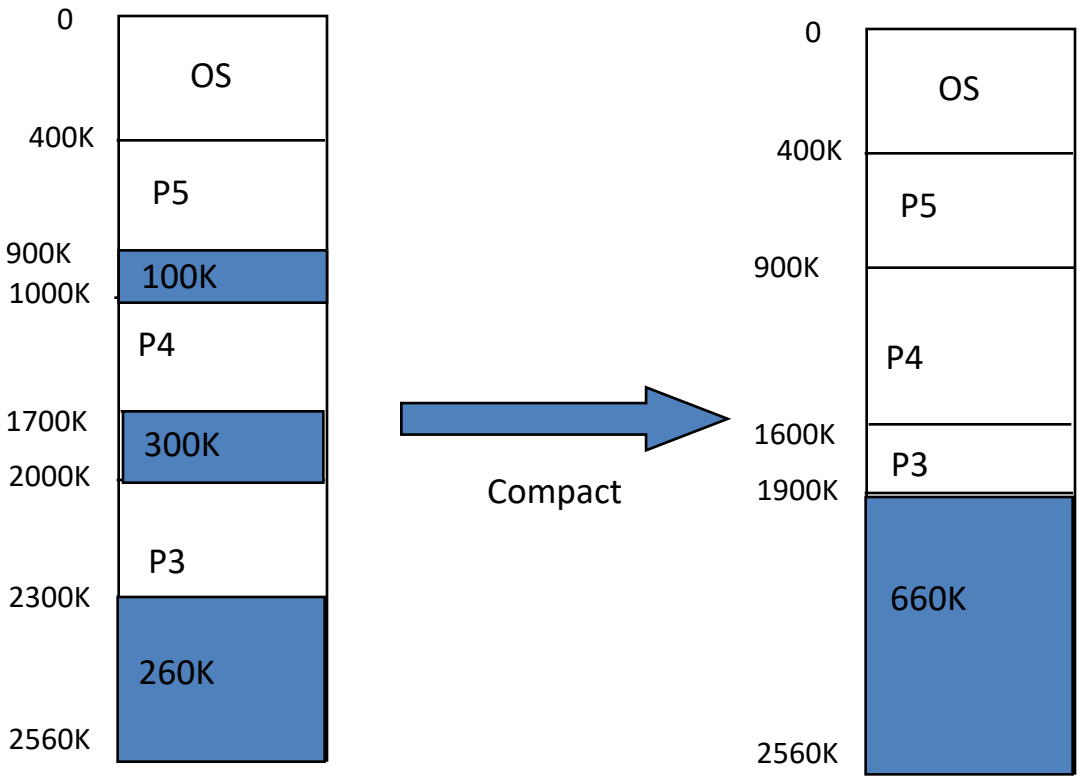
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
  - 50 % rule: Given N allocated blocks 0.5 blocks will be lost due to fragmentation.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference 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.

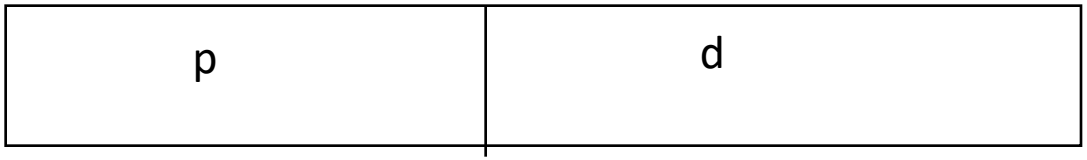


Paging

- 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.



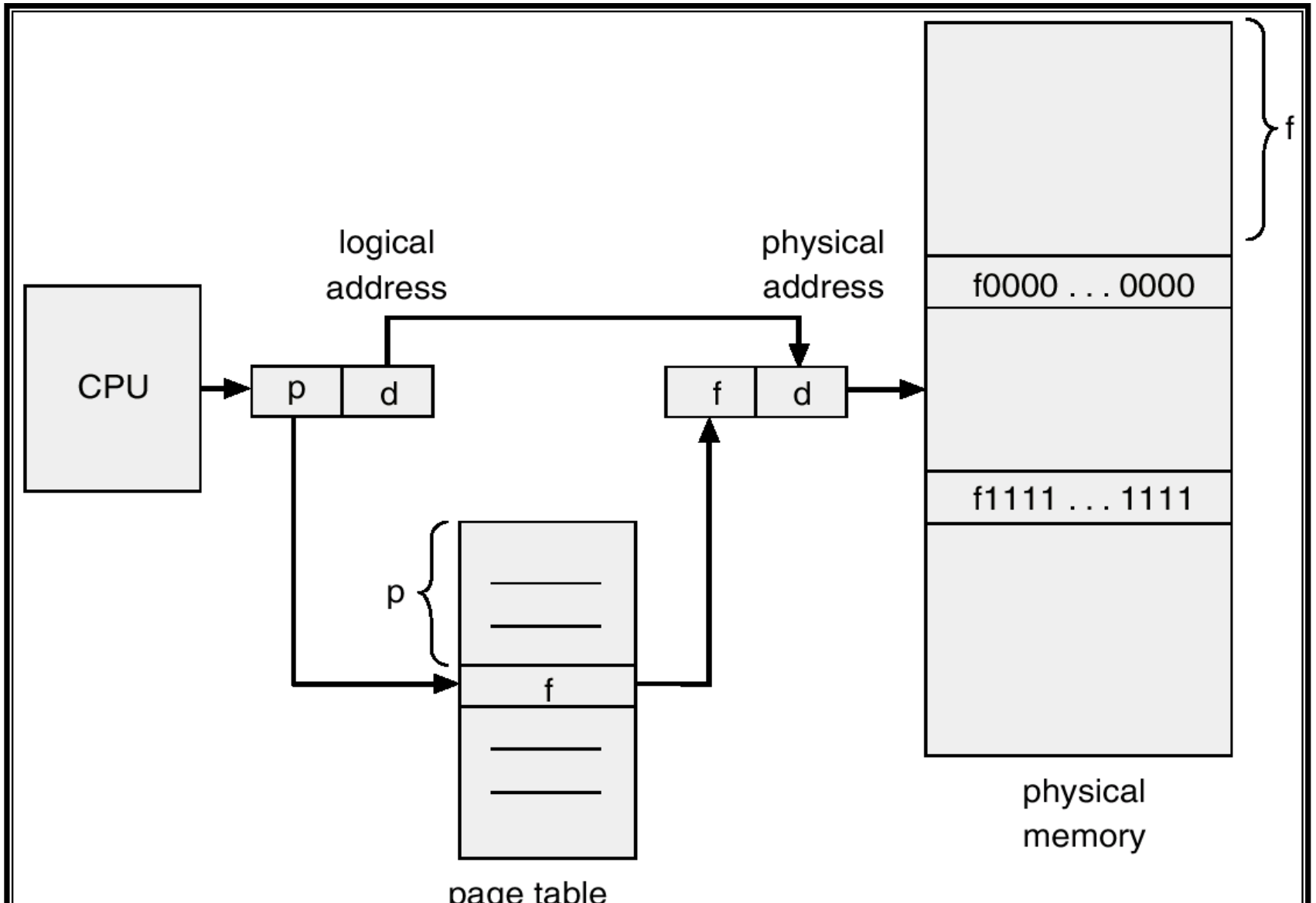
Page offset

Page #

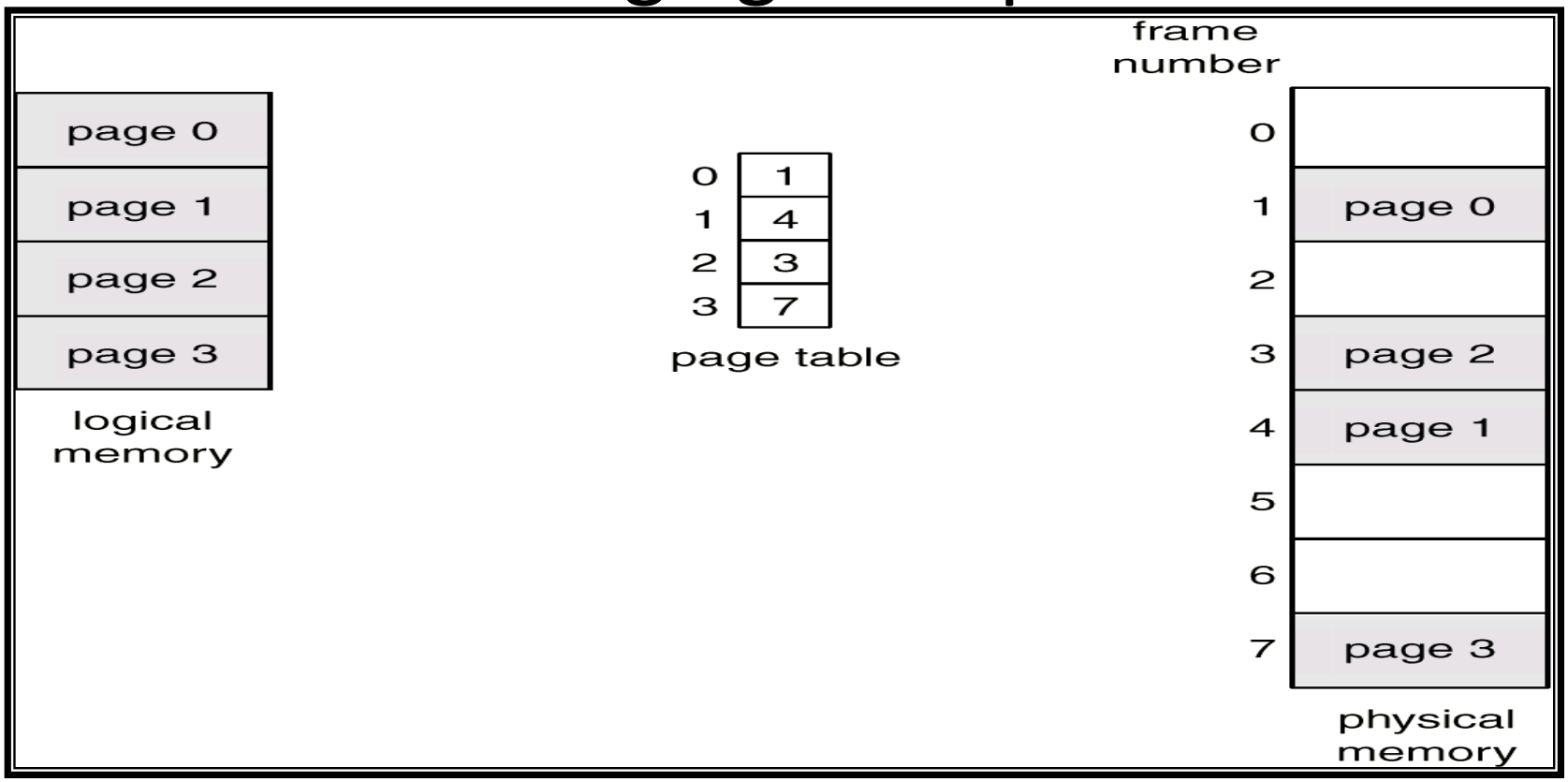
m-n

n

Address Translation Architecture

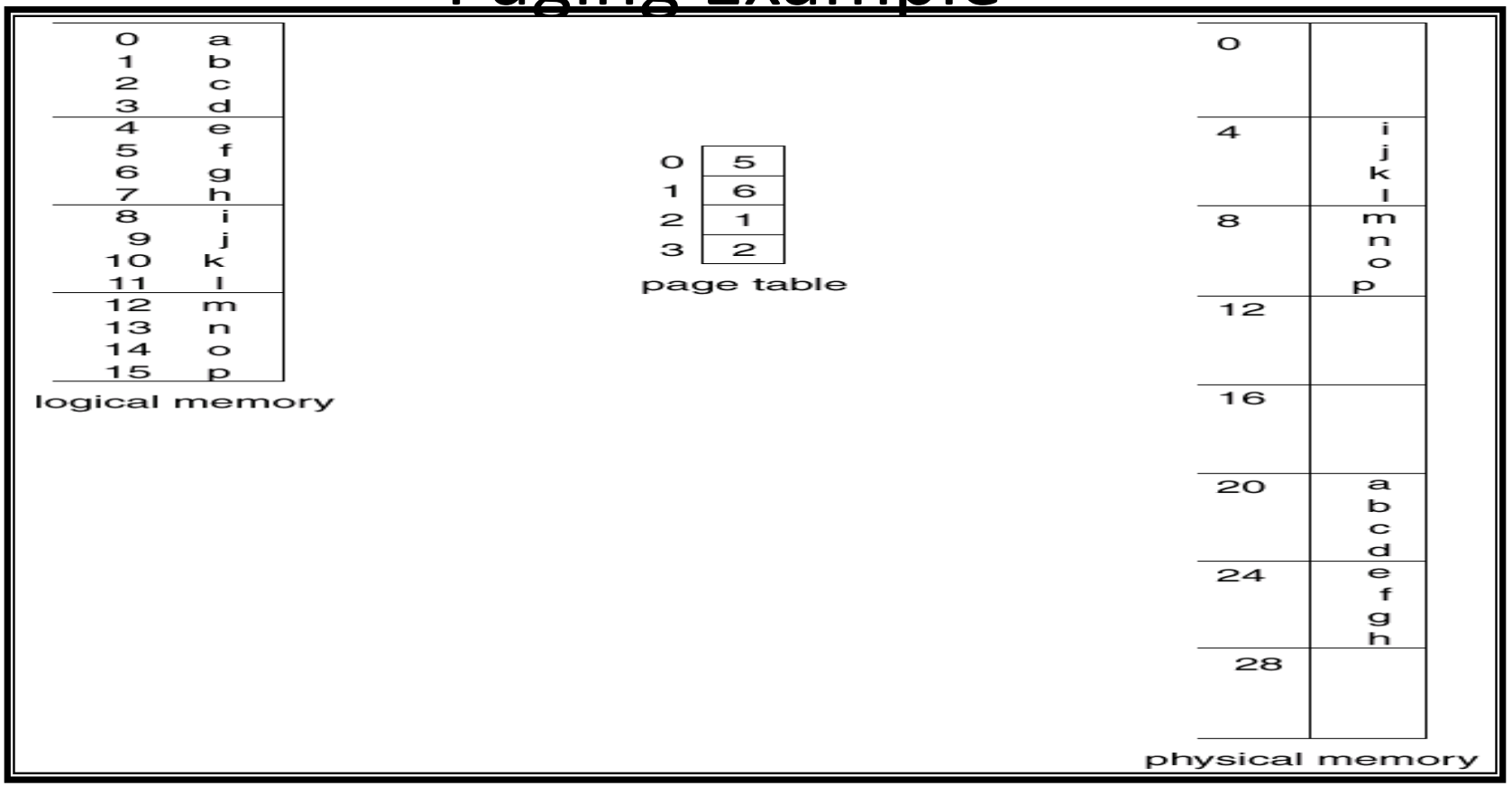


Paging Example



- Page size= 4 bytes; Physical memory=32 bytes (8 pages)
- 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.

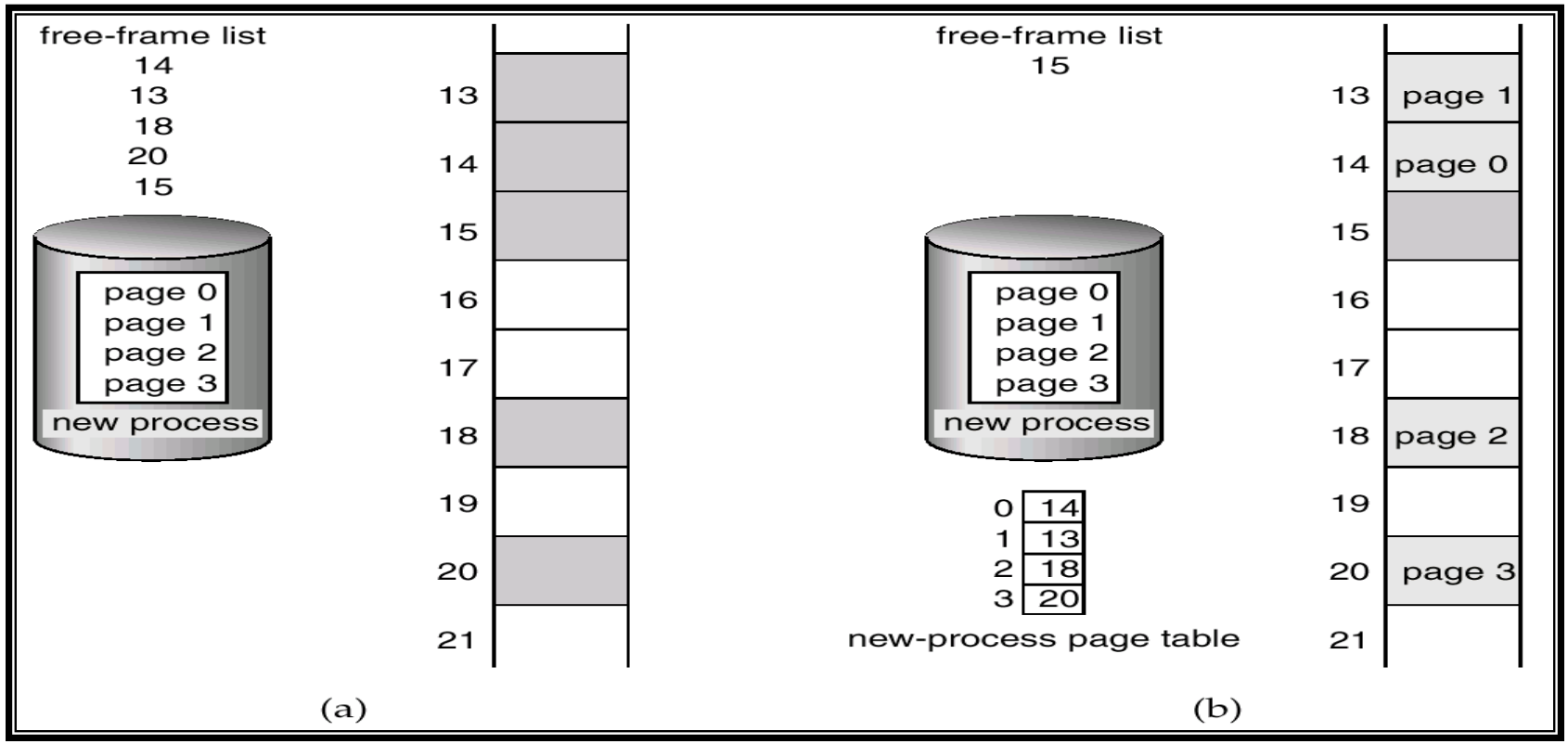
Paging Example



- Page size= 4 bytes; Physical memory=32 bytes (8 pages)
- Logical address 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 memory.
- OS maintains 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

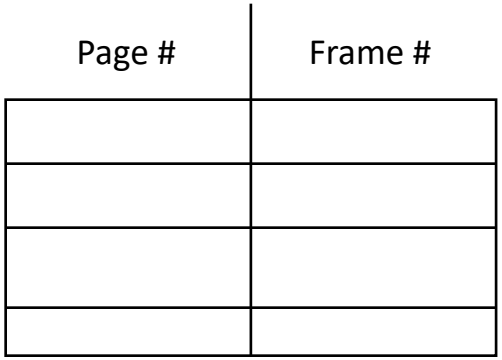
- Speed
- Space

Implementation of Page Table

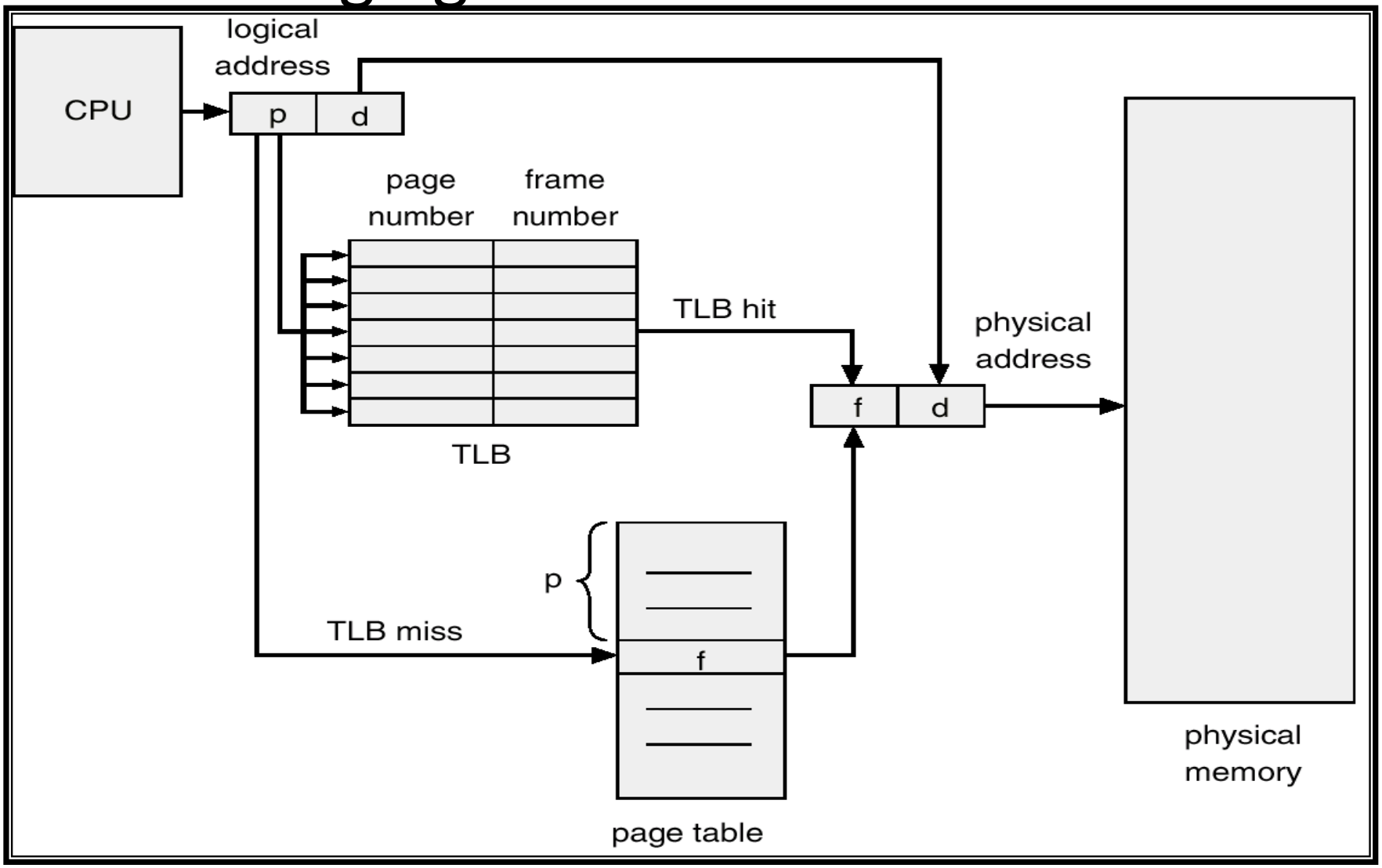
- Two options: Page table can be kept in registers or main memory
- Page table is kept in main memory due to bigger size.
  - Ex: address space = 2^{32} to 2^{64}
  - Page size= 2^{12}
  - Page table= 2^{32} / 2^{12}=2^{20}
  - If each entry consists of 4 bytes, the page table size = 4MB.
- *Page-table 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)

- 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



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

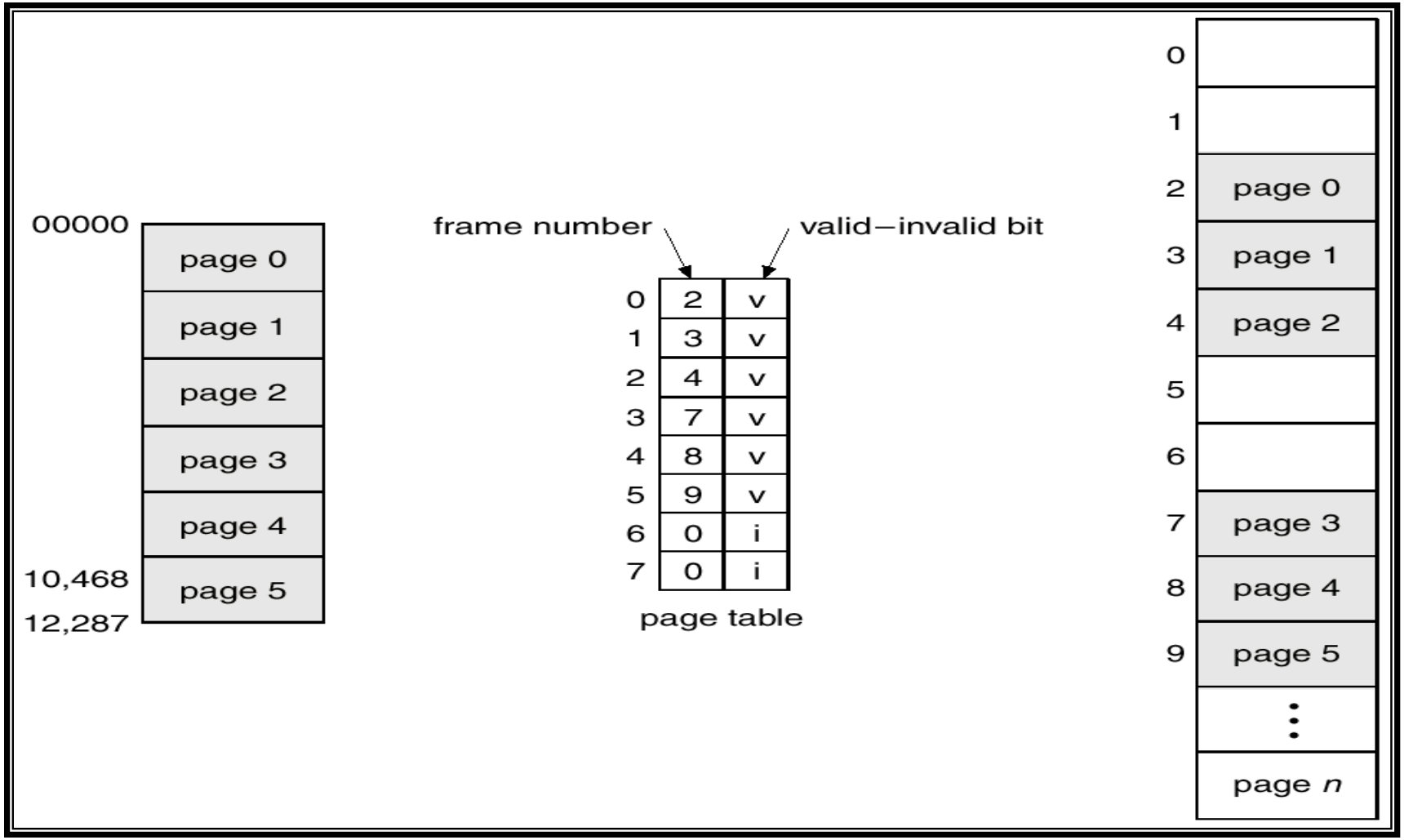
    = 122 nanoseconds

    = 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

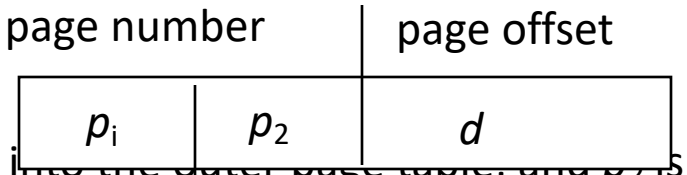
Hierarchical Page Tables

- 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.

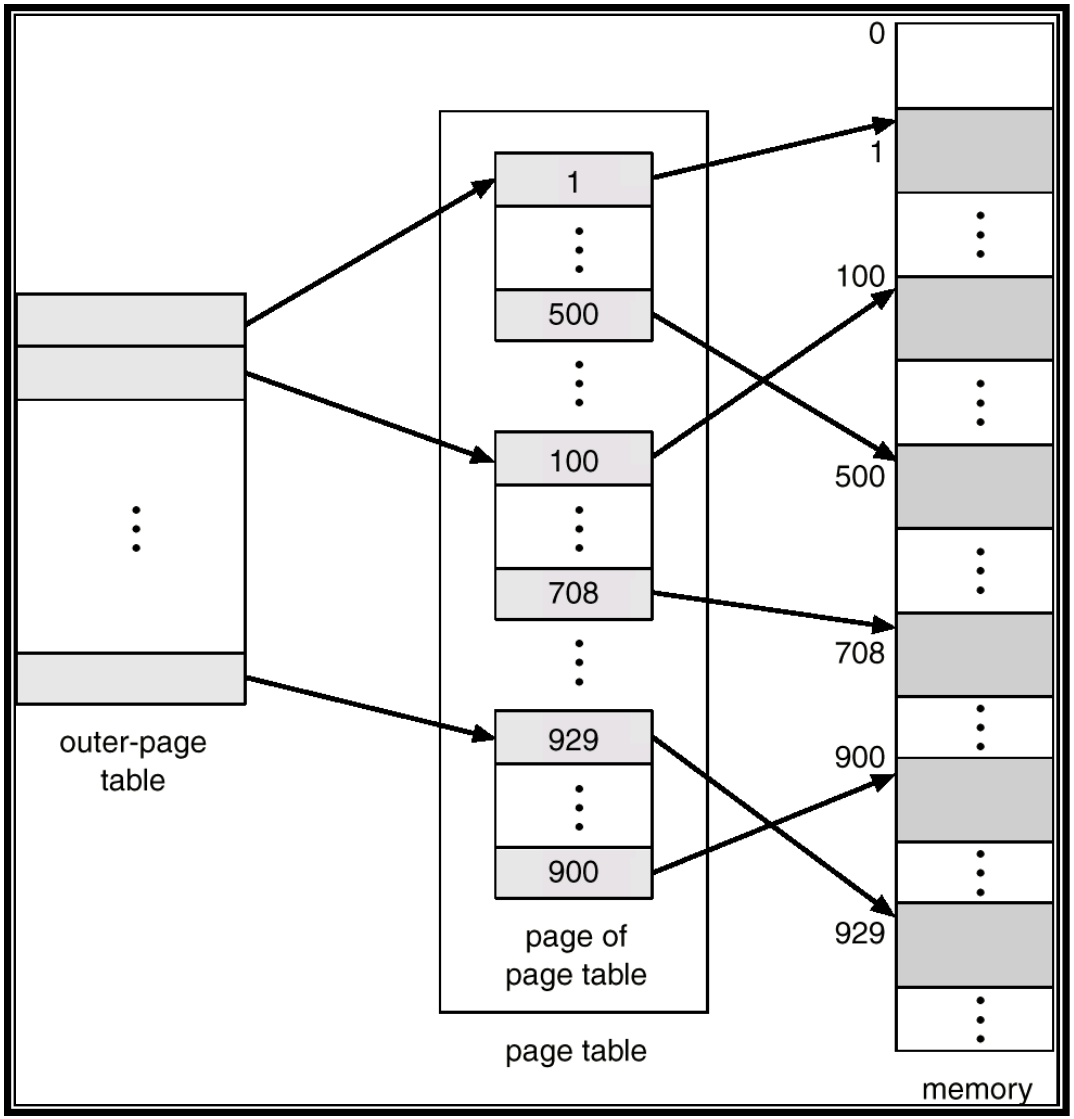
Two-Level Paging Example

- 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:



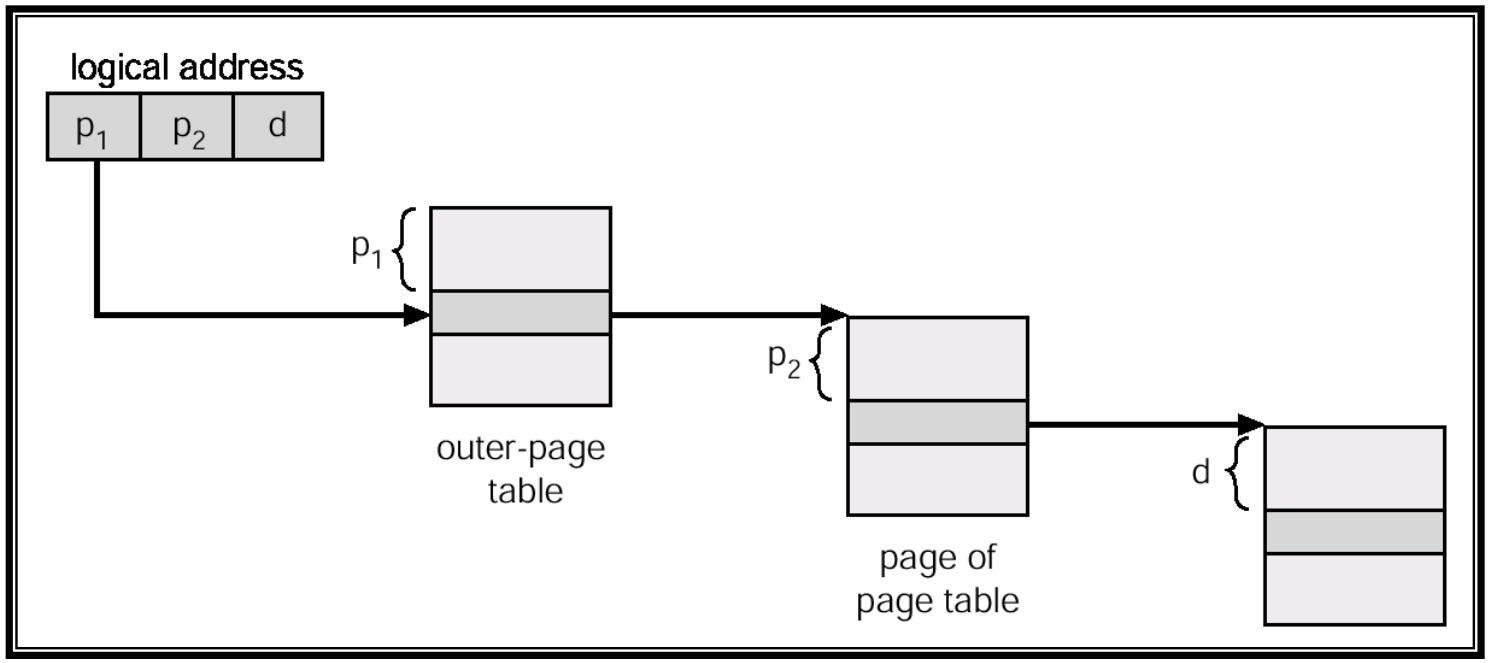
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme



Address-Translation Scheme

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



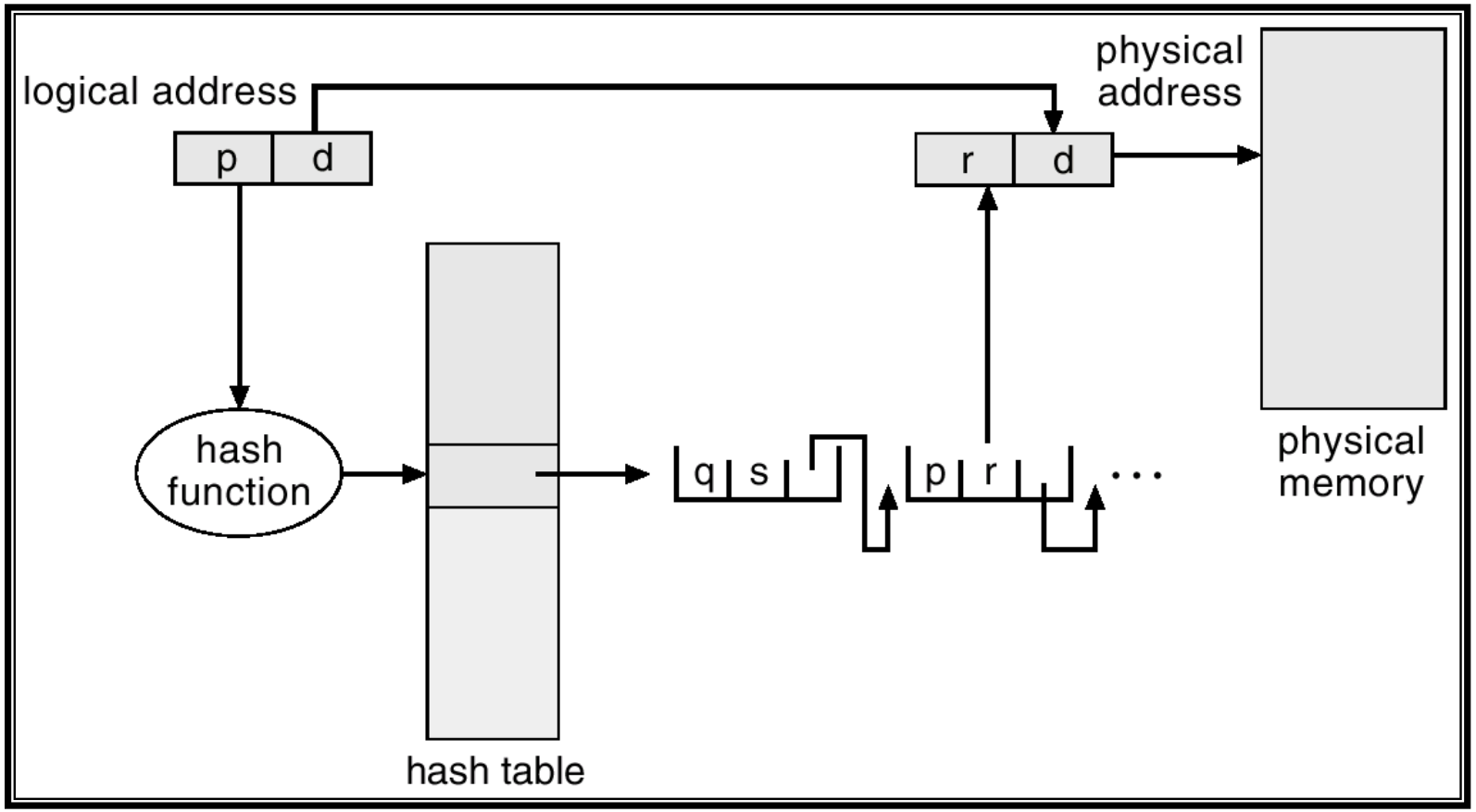
Address-Translation Scheme

- 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

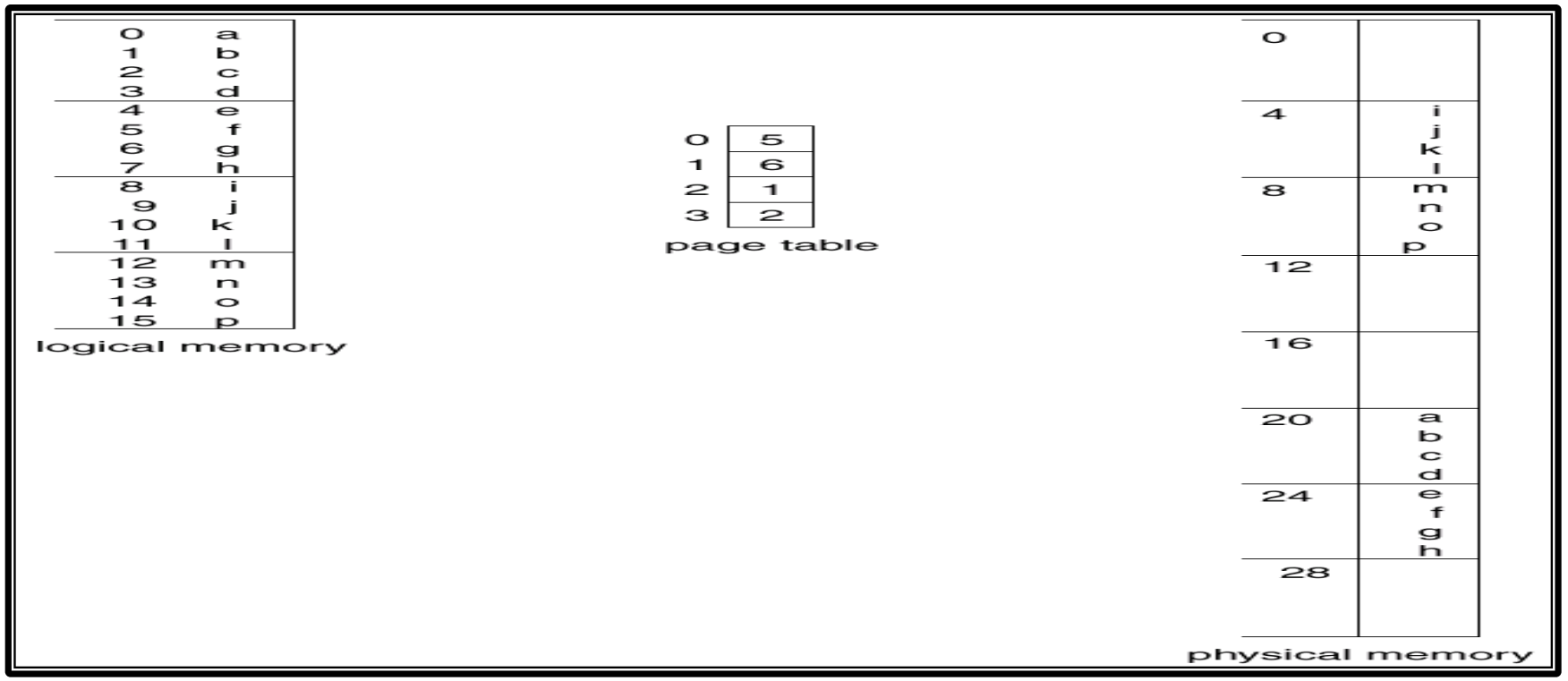
- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Table



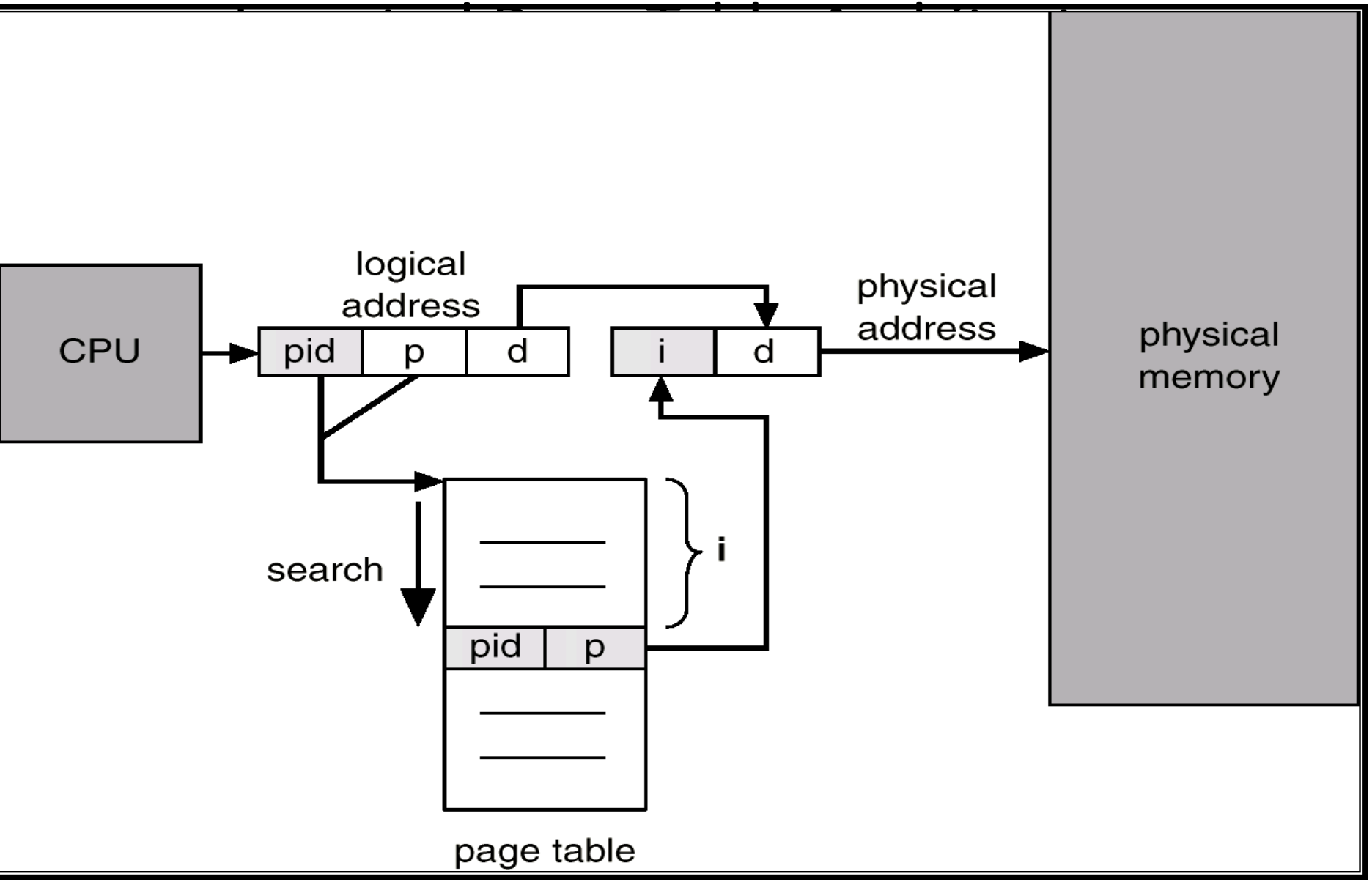
Inverted Page Table

- 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.
- Solution: Inverted page table



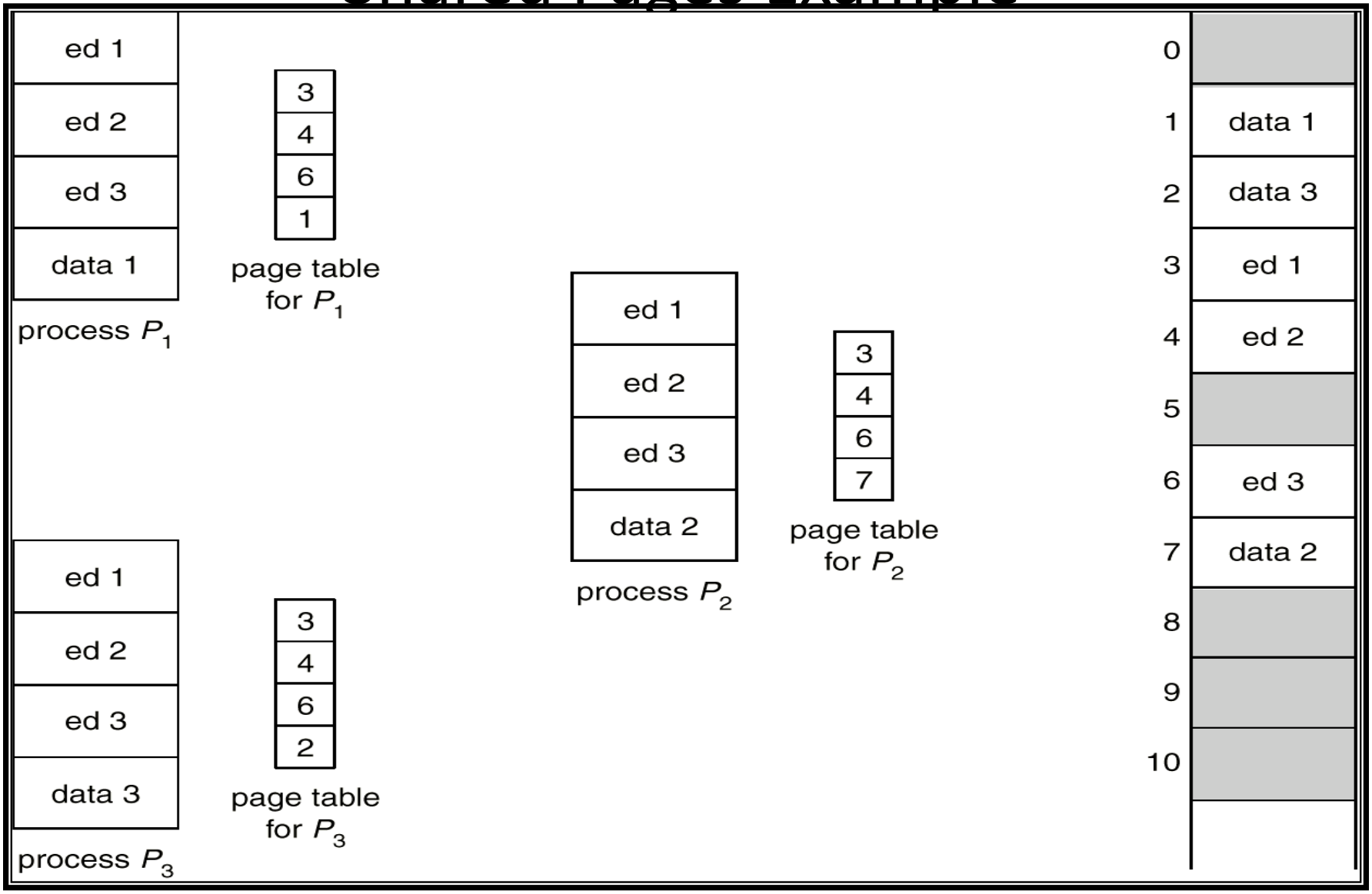
Inverted Page Table

- 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.


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

- Advantage of paging **Shared Pages**
  - 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

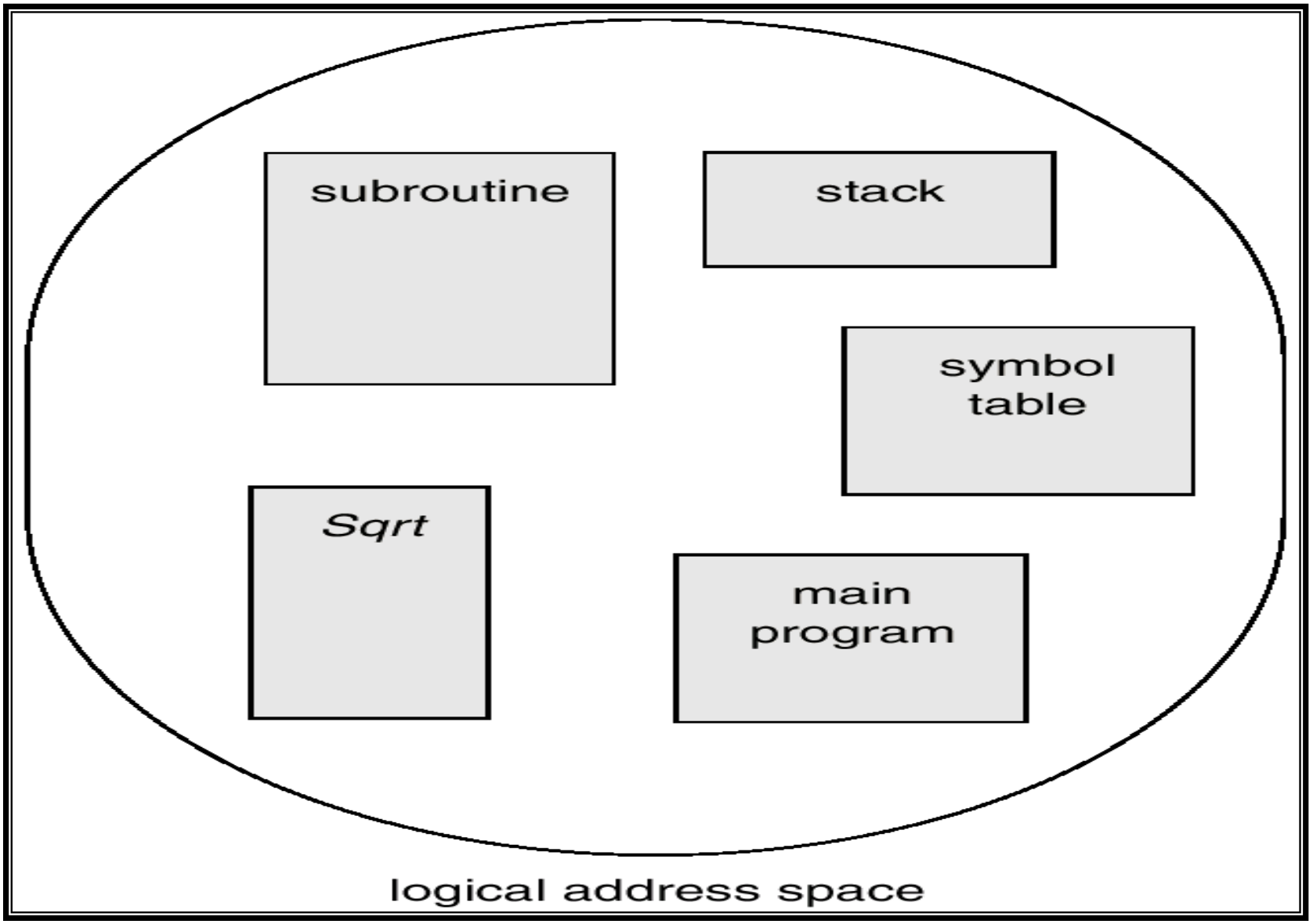


Segmentation

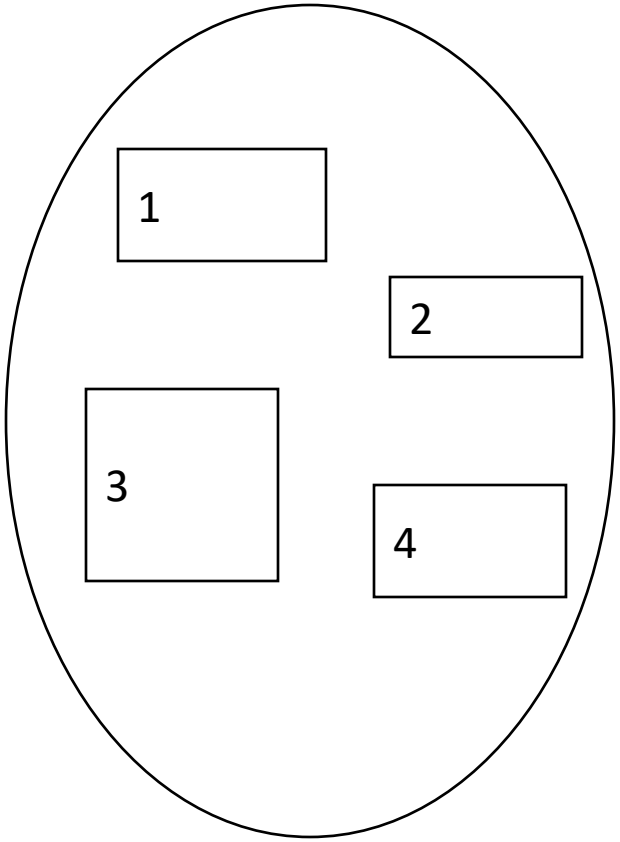
- Paging issues
  - Space: Size of page table
  - Searching time
- Segmentation is a memory-management scheme that supports user view of memory.
- Users prefer to view memory as a collection of variable-sized segments without any ordering.
- Logical address space is a collection of segments.
  - Each 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.

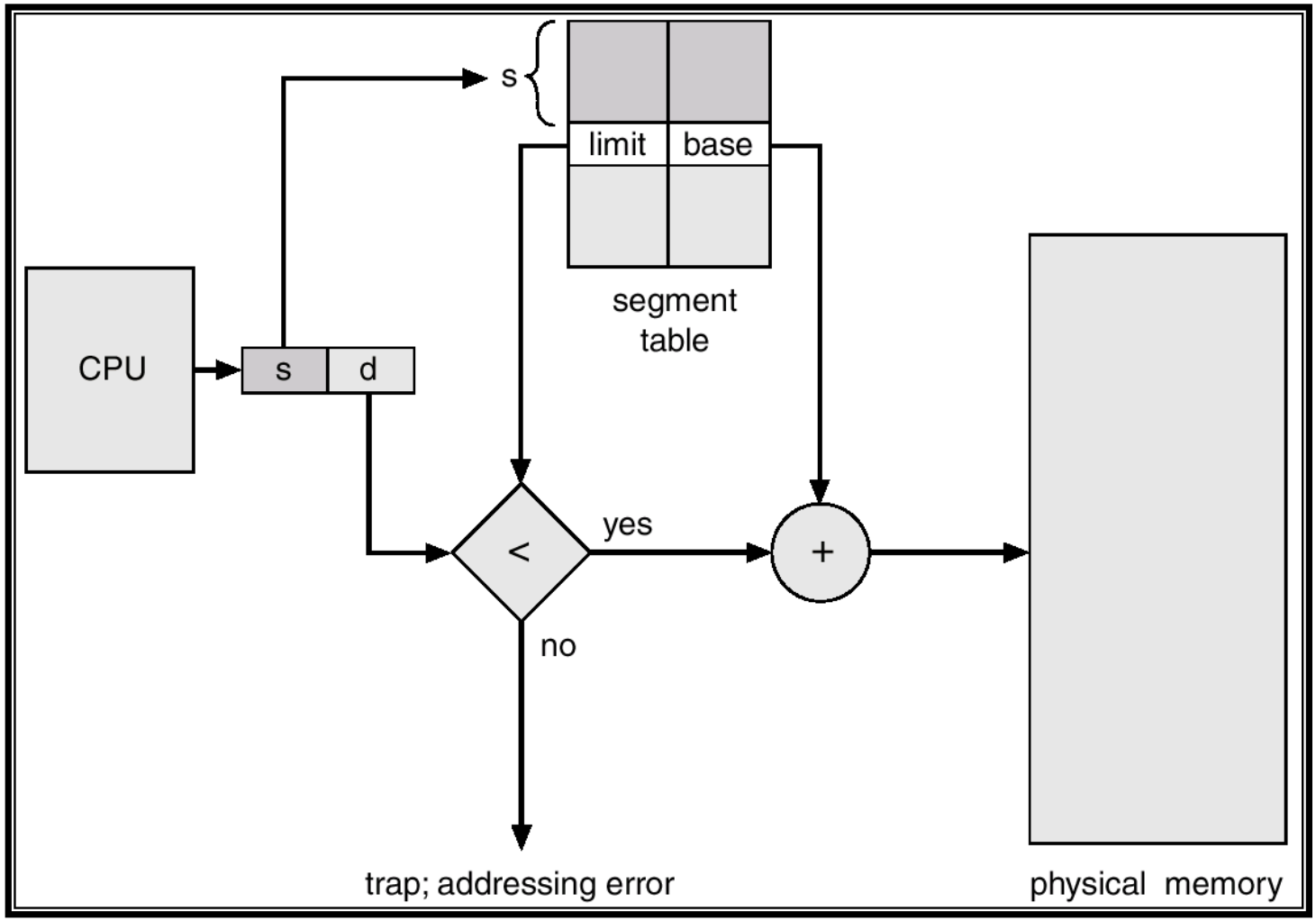
Segmentation Architecture (Cont.)

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

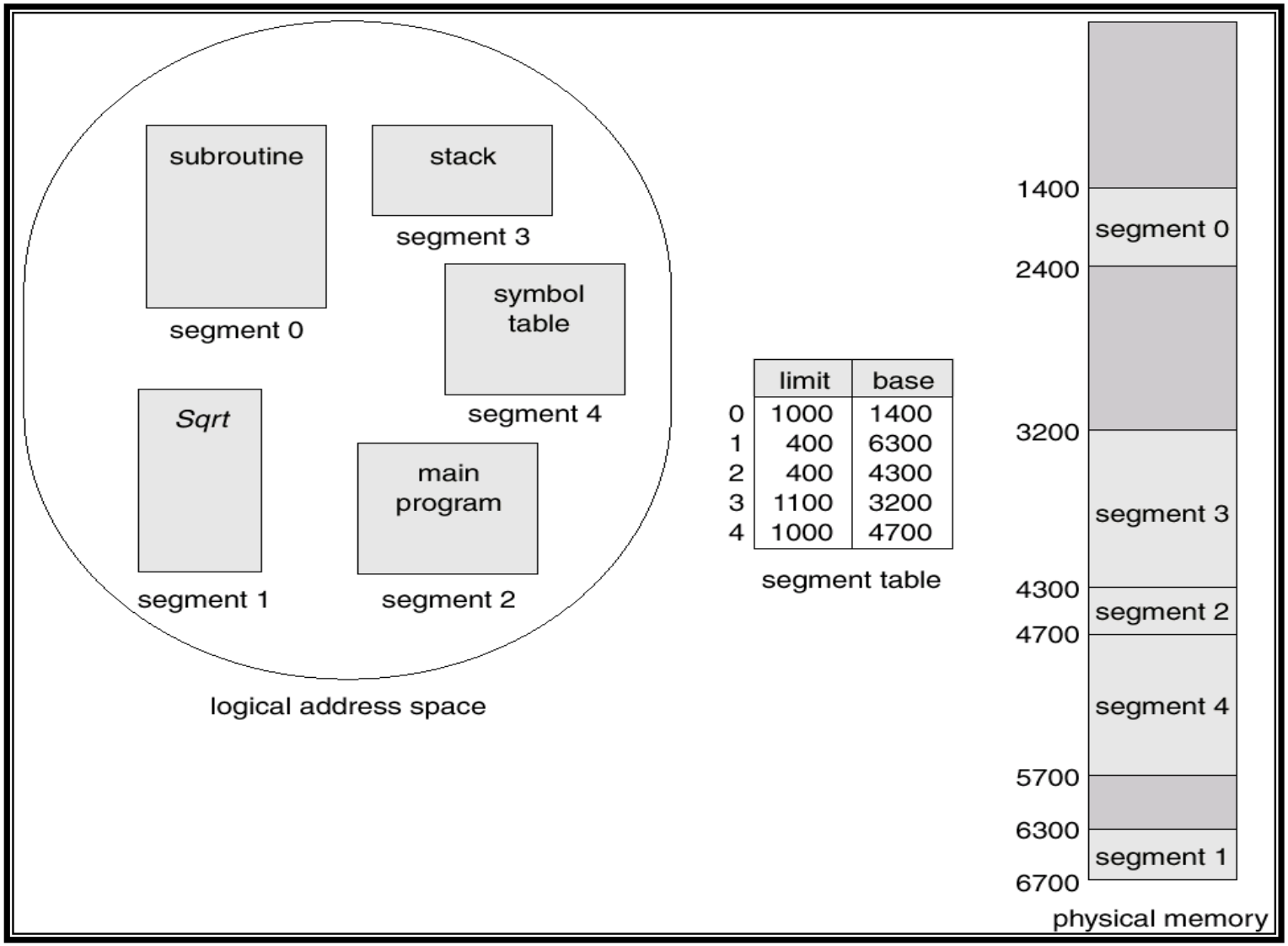
Segmentation Architecture (Cont.)

- 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

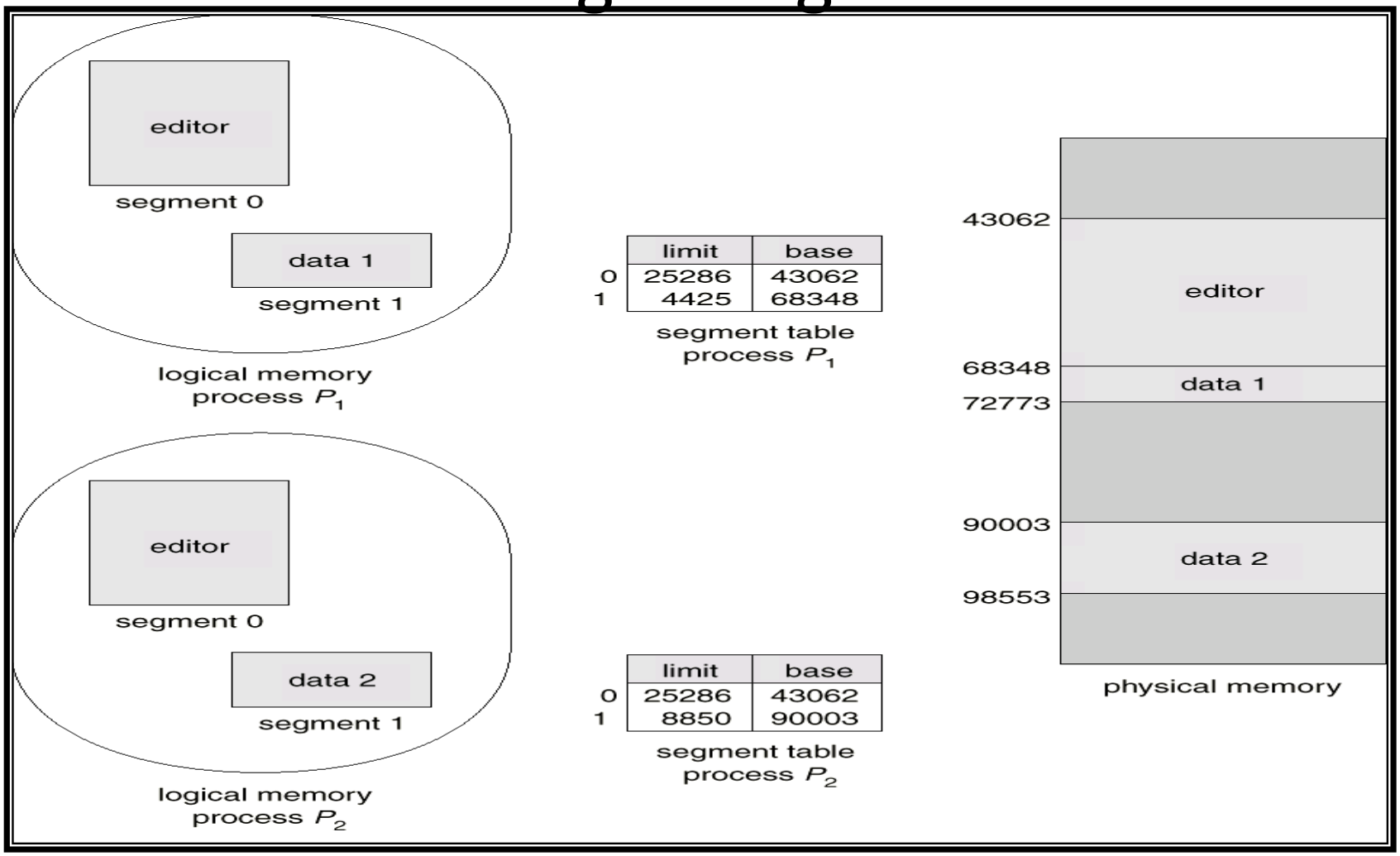
Segmentation Hardware



Example of Segmentation



Sharing of Segments

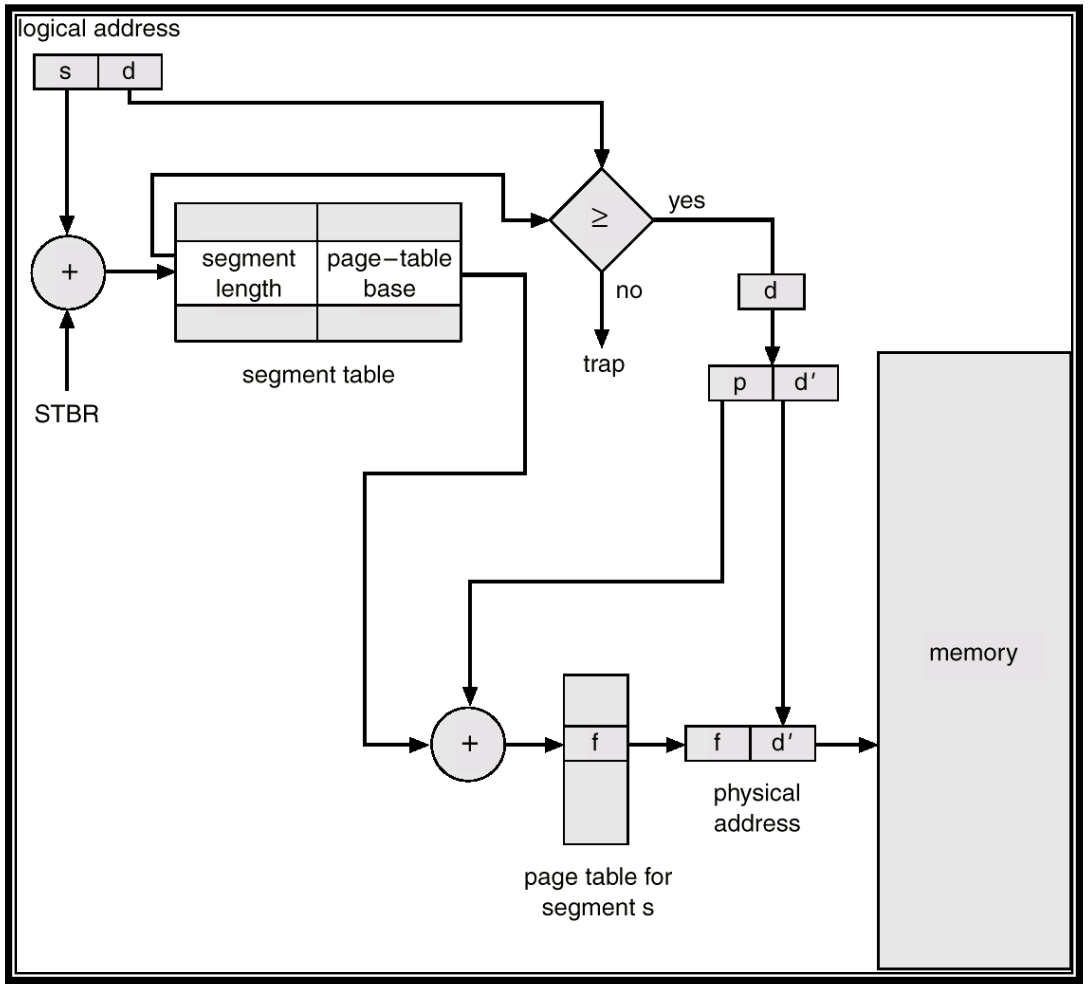


Segmentation with Paging – MULTICS

- 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.

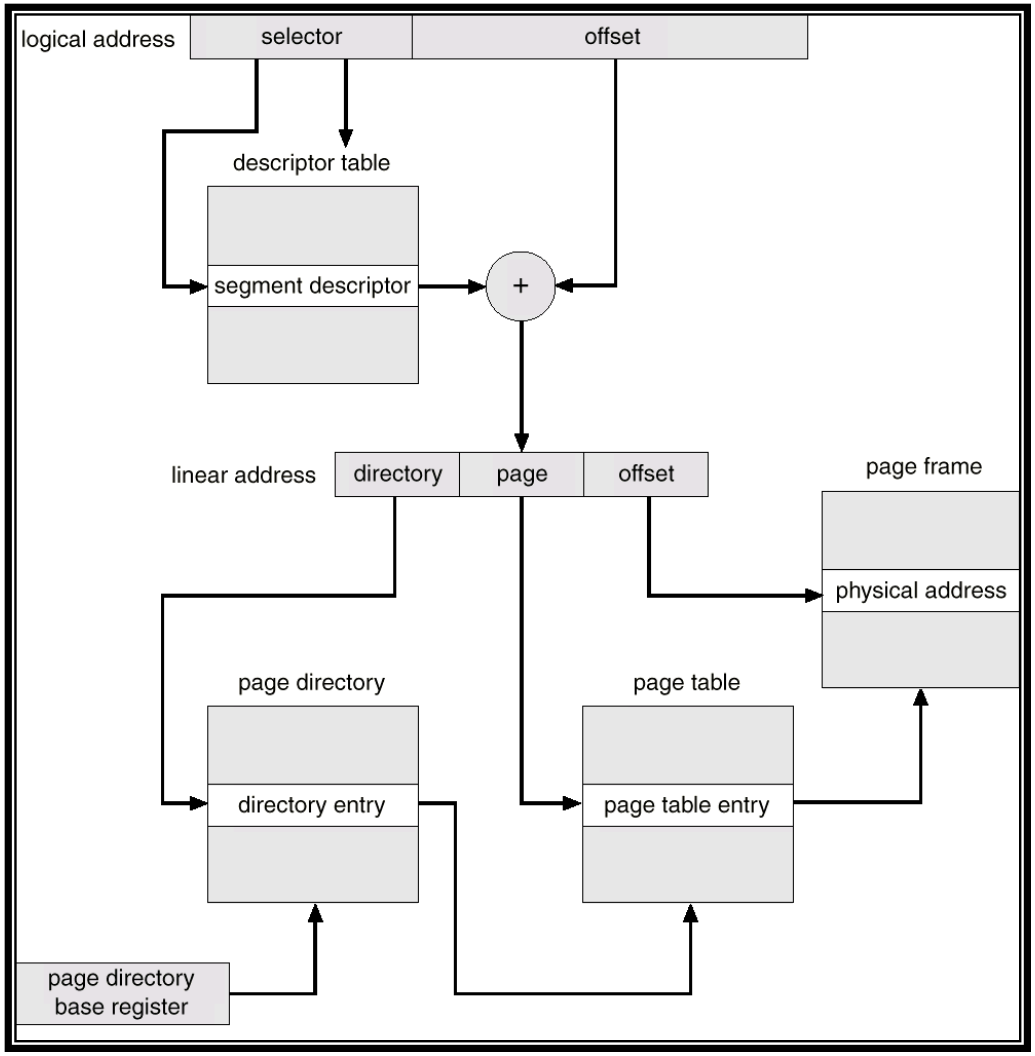
MULTICS Address Translation Scheme



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



Page Frame Management

- Page frames are 4KB in Linux.
- The kernel must keep track of the current status of each frame.
  - Are page frames allocated or free?
  - If allocated, do they contain process or kernel pages?
  - Linux maintains an array of page frame descriptors (one for each frame) of type **`struct page`**.
- NOTE: see the **`mm`** directory for memory management, especially **`page_alloc.c`**.

Page Frame Descriptors

- Each descriptor has several fields, including:
  - **`count`** - equals 0 if frame is free, >0 otherwise.
  - **`flags`** - an array of 32 bits for frame status.
    - Example flag values:
      - **`PG_locked`** - page cannot be swapped out.
      - **`PG_reserved`** - page frame reserved for kernel code or unusable.
      - **`PG_Slab`** - included in a slab (more later).

The `mem_map` Array

- All page frame descriptors are stored in the **`mem_map`** array.
- Descriptors are less than 64 bytes. Therefore, **`mem_map`** requires about 4 page frames for each MB of RAM.
- The **`MAP_NR`** macro computes the number of the page frame whose address is passed as a parameter:
  - **`#define MAP_NR(addr) (__pa(addr) >> PAGE_SHIFT)`**
  - **`__pa`** macro converts logical address to physical.

Requesting Page Frames

- Main routine for requesting page frames is:
  `__get_free_pages(gfp_mask,order)`
- Request $2^{\texttt{order}}$ contiguous page frames.
- **`gfp_mask`** specifies how to look for free frames. It is a bitwise OR of several flags, including:
  - **`__GFP_WAIT`** – Allows kernel to discard page frame contents to satisfy request.
  - **`__GFP_IO`** – Allows kernel to write pages to disk to free page frames for new request.
  - **`__GFP_HIGH/MED/LOW`** – Request priority. Usually user requests are low priority while kernel requests are higher.
    - eg., **`GFP_ATOMIC=__GFP_HIGH;`**
      `GFP_USER=__GFP_WAIT=1|__GFP_IO=1|__GFP_LOW;`

Releasing Page Frames

- Main routine for freeing pages is: **`Free_pages(addr,order)`**
  - Check frame at physical address **`addr`**.
    - If not reserved, decrement descriptor's **`count`** field.
      - If **`count==0`**, free $2^{\texttt{order}}$ contiguous frames.
        - » **`free_pages_ok()`** inserts page frame descriptor of 1st free page in list of free page frames.

External Fragmentation

- *External fragmentation* is a problem when small blocks of free page frames are scattered between allocated page frames.
  - Becomes impossible to allocate large blocks of *contiguous* page frames.
- Solution:
  - Use paging h/w to group non-contiguous page frames into contiguous linear (virtual) addresses.
  - Track free blocks of contiguous frames & attempt to avoid splitting *large* free blocks to satisfy requests.
    - DMA controllers, which bypass the paging hardware, sometimes need contiguous page frames for buffers.
    - Contiguous frame allocation can leave page tables unchanged – TLB contents don't need to be flushed, so memory access times are reduced.

The Buddy System

- Free page frames are grouped into lists of blocks containing 2^n contiguous page frames.
  - Linux has 10 lists of 1,2,4,…,512 contiguous page frames.
  - Physical address of 1^{st} frame in a block is a multiple of the group size e.g., multiple of $16*2^{12}$ for a 16-page-frame block.

Buddy Allocation

- Example: Need to allocate 65 contiguous page frames.
  - Look in list of free 128-page-frame blocks.
  - If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
  - If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
  - If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

- Question: What is the worst-case *internal* fragmentation?

Buddy De-Allocation

- When blocks of page frames are released the kernel tries to merge pairs of “buddy” blocks of size ***b*** into blocks of size ***2b***.
- Two blocks are buddies if:
  - They have equal size ***b***.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b*2^{12}$.
- The process repeats by attempting to merge buddies of size ***2b***, ***4b***, ***8b*** etc…

Buddy Data Structures

- An array of 10* elements (one for each group size) of type **`free_area_struct`**.
  - **`free_area[0]`** points to array for non-ISA DMA buddy system.
  - **`free_area[1]`** points to array for ISA DMA buddy system.
  - Linux 2.4.x has a 3rd buddy system for high physical memory! Makes dynamic memory mgt fast!
- A group of binary arrays (**`bitmaps`**) for each group size in each buddy system.

Example Buddy Memory Mgt

- 128MB of RAM for non-ISA DMA.
  - **`free_area[0][k]`** consists of ***n*** bits, one for each pair of blocks of size 2^K page frames.
    - Each bit in a bitmap is 0 if a pair of buddy blocks are ***both*** free or allocated, else 1.
  - **`free_area[0][0]`** consists of ***16384*** bits, one for each pair of the ***32768*** page frames.
  - **`free_area[0][9]`** consists of ***32*** bits, one for each pair of blocks of ***512*** contiguous page frames.

Memory Area Management

- *Memory areas* are contiguous physical addresses of arbitrary length e.g., from a few bytes to several KBs.
- Could use a buddy system for allocating memory in blocks of size 2^K within pages, and then another for allocating blocks in power-of-2 multiples of pages.
- Linux uses a ***slab allocator*** for arbitrary memory areas.
  - Memory is viewed as a collection of related objects.
    - Objects are cached when released so that they can be allocated quickly for new requests.
    - Memory objects of the same type are repeatedly used e.g., process descriptors for new/terminating processes.
  - Can have memory allocator for commonly used objects of known size and buddy system for other cases.

Linux Slab Allocator

- Objects of same type are grouped into ***caches***.
  - Can view caches by reading /proc/slabinfo.
  - Caches are divided into ***slabs***, with >=1 page frames containing both allocated & free objects.
  - See mm/slab.c for more details.
- A newly created cache does not contain any slab or free objects.
- Slabs are assigned to a cache when:
  - A request for allocating a new object occurs.
  - Cache does not already have a free object.
    - Buddy system is invoked to get new pages for a slab.

Slab De-allocation

- A slab is released only if the following conditions hold:
  - Buddy system is unable to satisfy a new request for a group of page frames.
  - Slab is empty – all objects in it are free.
- ***Destructor methods**** on objects in an empty slab are invoked.
- Contiguous page frames in slab are returned to buddy system.

- NOTE: when objects are created, they also have corresponding ***constructor methods*** (possibly NULL valued) that can initialize objects.

Object Alignment

- Memory accesses usually faster if objects are word aligned e.g., on 4-byte boundaries with 32-bit Intel architecture.
- Linux does not align objects if space is continually wasted as a result.
  - Linux rounds up object size to a factor of cache size if internal fragmentation does not exceed a threshold.
    - Idea is to trade fragmentation for aligning object of larger size.

General Purpose Objects

- Linux maintains a list of general purpose objects, having geometrically distributed sizes from 32 to 131072 bytes.
- These objects are allocated using:
  `void *kmalloc(size_t size, int flags);`
  - **`flags`** is same as for **`get_free_pages()`**.
    - e.g. **`GFP_ATOMIC`**, **`GFP_KERNEL`**.
- Gen purpose objects are freed using **`kfree()`**.