Main Memory

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

Objectives

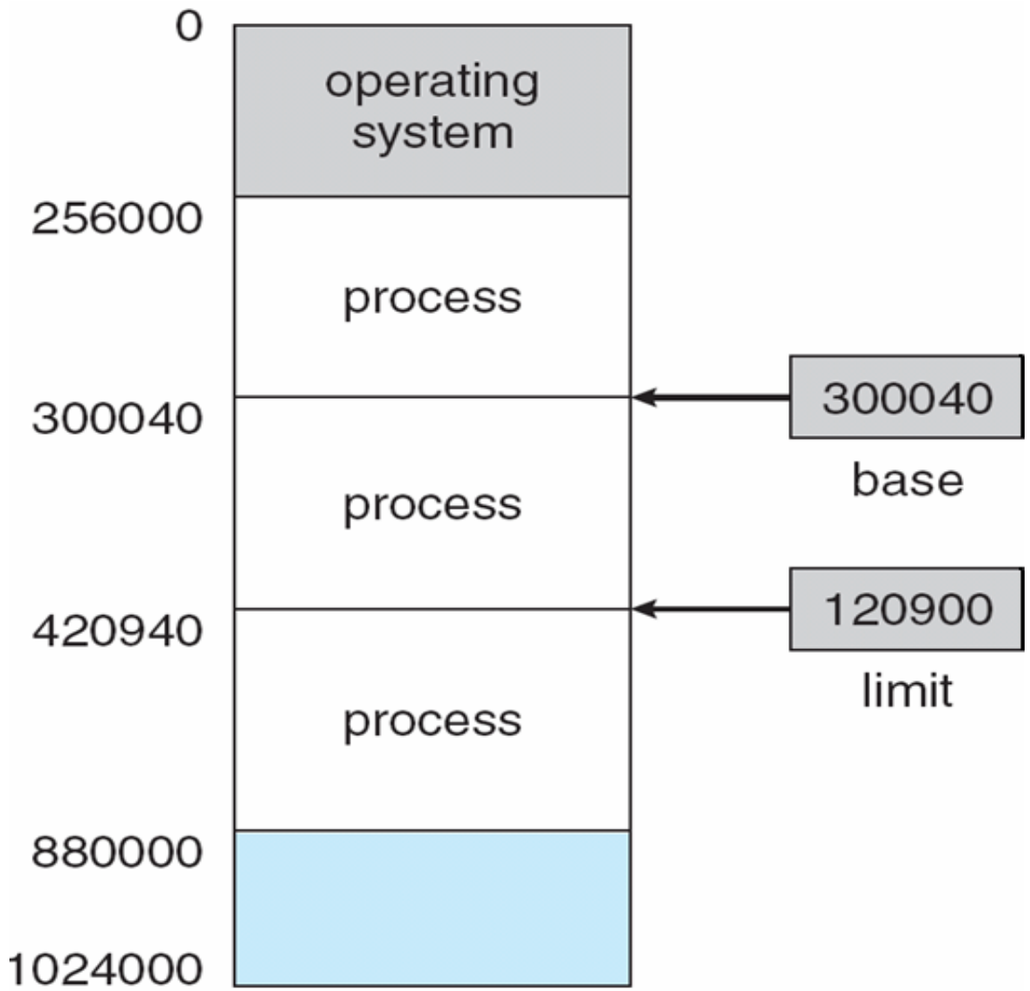
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

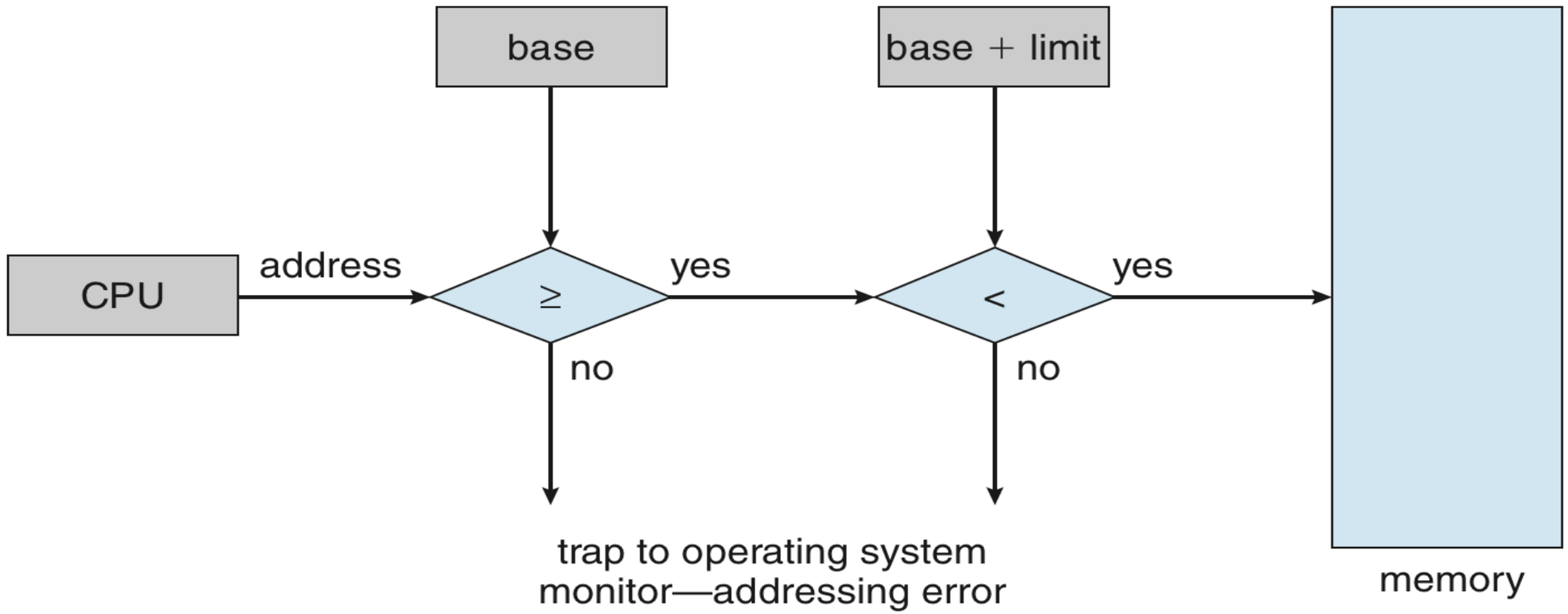
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Machine Instructions takes memory addresses as arguments but not the disk addresses.
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles. In such cases, the processor normally needs to stall the instruction until the data is accessed from the main memory.
- The Remedy is to keep the fast memory between CPU and Main Memory, I.e, Cache
- **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



Hardware Address Protection with Base and Limit Registers



Base and Limit Registers can be loaded only by OS

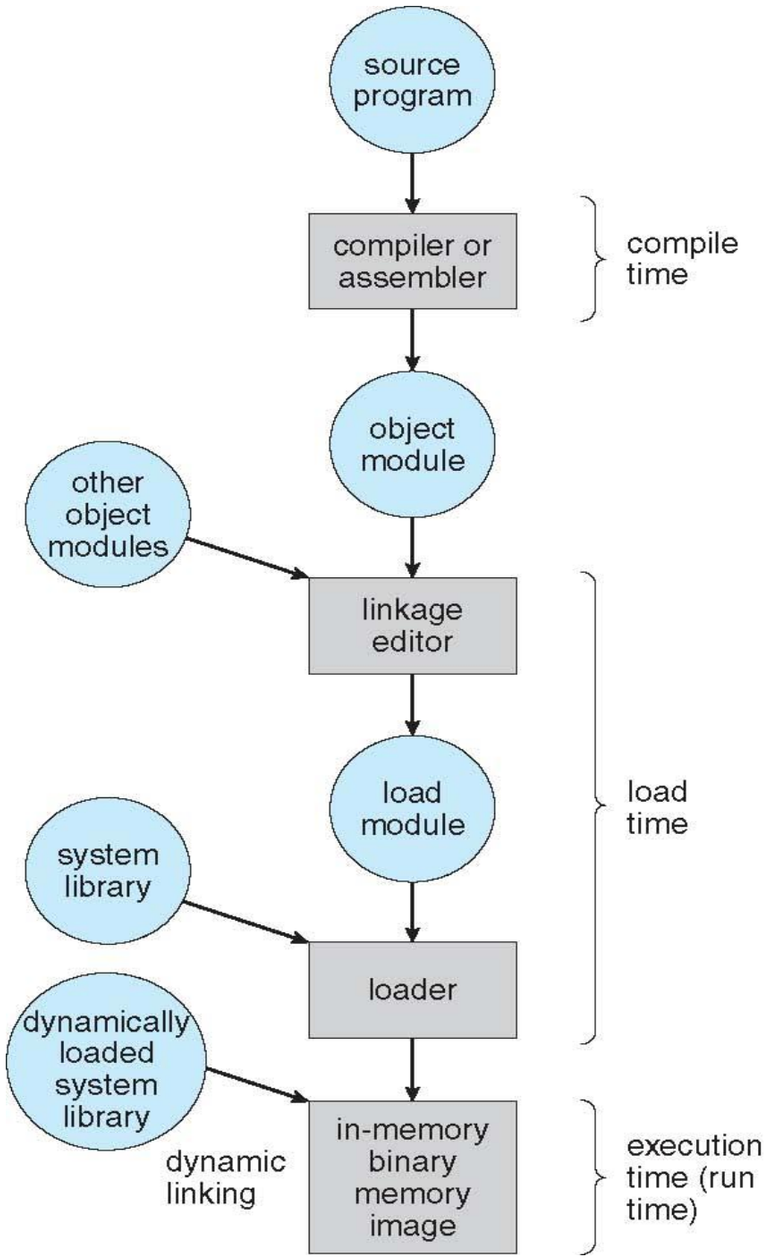
Address Binding

- 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(Depends on Symbols)
  - Compiler simply **BIND these Symbolic addresses to Relocatable addresses**. Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time. Final Binding is delayed until Load 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 corresponding to the logical addresses generated by a program

Memory-Management Unit (MMU)

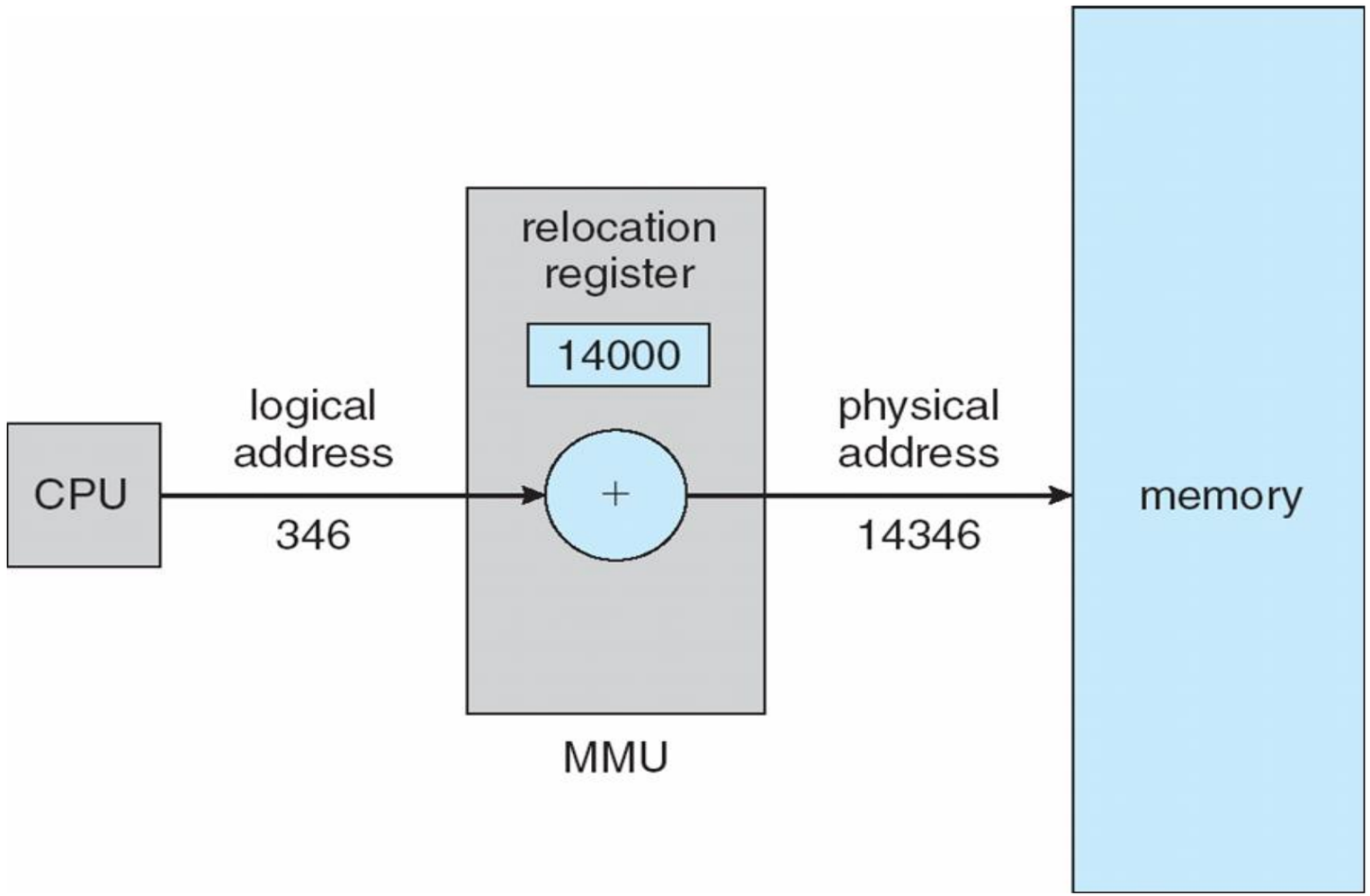
- Hardware device that at run time maps virtual to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 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

Memory-Management Unit (MMU)

- We have 2 different types of address:
  - **Logical address**
    - In the range of **0 to max**
  - **Physical address**
    - In the range of **R+0 to R+max**, where R is the base register value.

- The user generates only Logical addresses and thinks that the process runs in locations 0 to max.
- Logical addresses must be converted to Physical addresses before they are used.

Dynamic relocation using a relocation register



Dynamic Loading

- For executing a process, it is required to keep entire program in physical memory. Thus the size of the program is restricted to size of Physical Memory.
- Dynamic Loading provides the solution for above specified problem. 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
- With Dynamic Linking, a stub is included in the image for each library routine reference.
- Stub is a small piece of code that indicates how to locate the appropriate memory resident library routine or how to load the library if the routine is not already present.
- When the stub is executed, it checks whether the routine is already in memory. If it is not, the program loads the routine into the memory.
- Stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that particular routine is executed directly, incurring no cost of dynamic linking.

Dynamic Linking

- 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

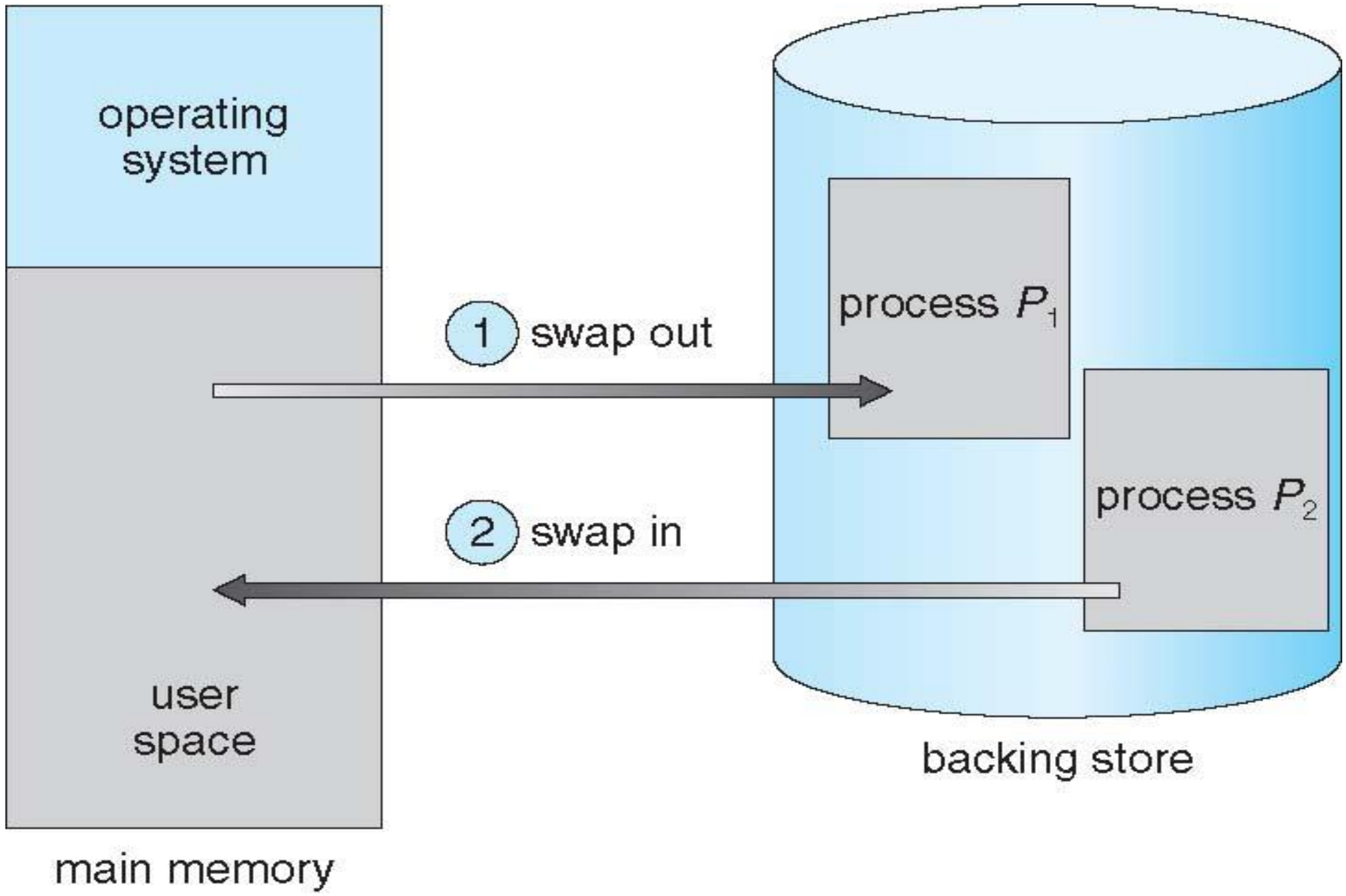
Swapping

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

Swapping

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
  - If binding is done at assembly or load time, then the process can not be easily moved to a different location.
  - If Execution-time binding is used then the process can be swapped back into a different location because the physical addresses are calculated at run time.
- 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



Swapping

- Whenever the CPU scheduler decides to execute a process, it calls the Dispatcher.
- The Dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- Reloads the Registers and transfers the control to the selected new process.
- The Context switch time in such a swapping system is fairly high.

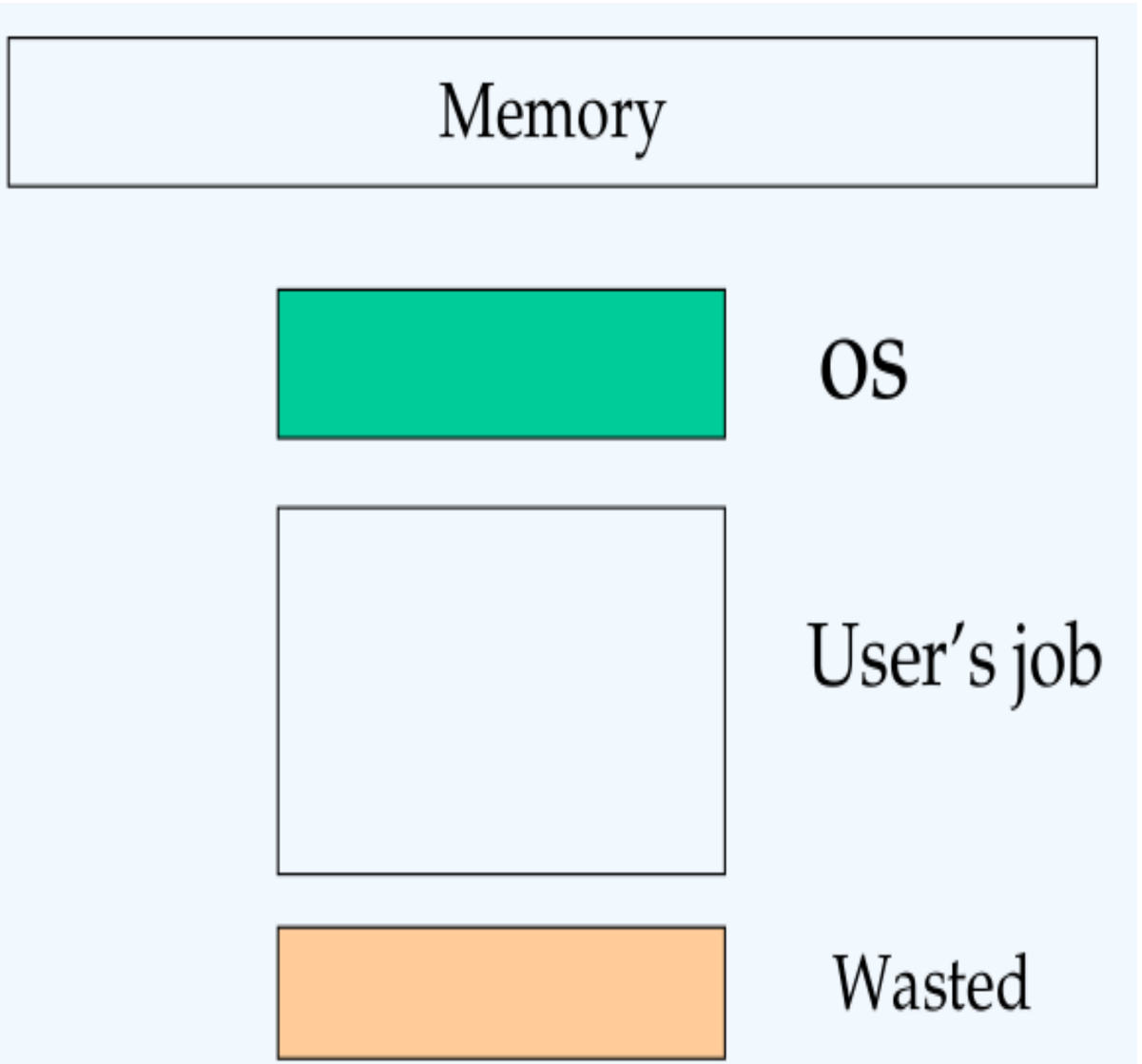
Context Switch Time including Swapping

- 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
  - Plus disk latency of 8 ms
  - Swap out time of 2008 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4016ms (> 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 in case of Dynamic Memory allocation.

Memory Management Schemes

- Single Contiguous Allocation
- Partitioned Allocation
  - Fixed Partitioned Allocation
  - Variable Partitioned Allocation
- Relocatable Partitioned Allocation
- Simple Paged Allocation
- Demand Paging
- Segmentation

Single Contiguous Allocation



In single contiguous allocation, the user program is given complete control of the CPU until completion or an error occurs.

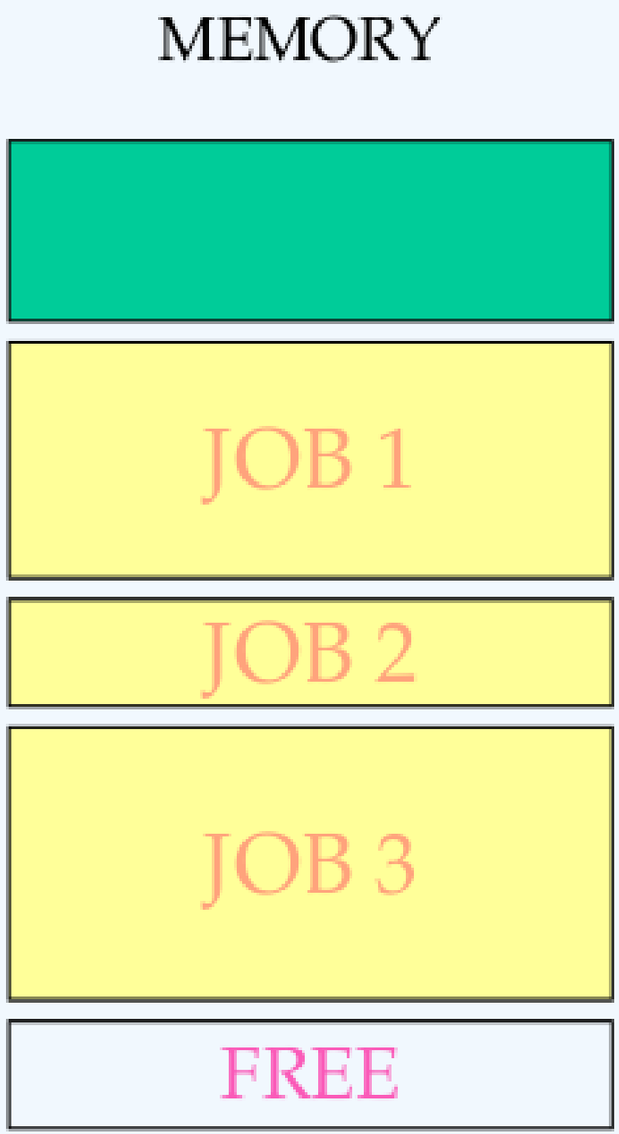
Advantages:

- Very simple to implement

Disadvantages:

- Leads to uniprogramming
- Leads to wastage of space
- Leads to wastage of time (During any I/O operation CPU has to wait till I/O is finished)

Fixed Partitioned Allocation



Here, the memory is divided into fixed partitions as shown

Advantage:

•Leads to Multiprogramming (CPU utilization is increased).

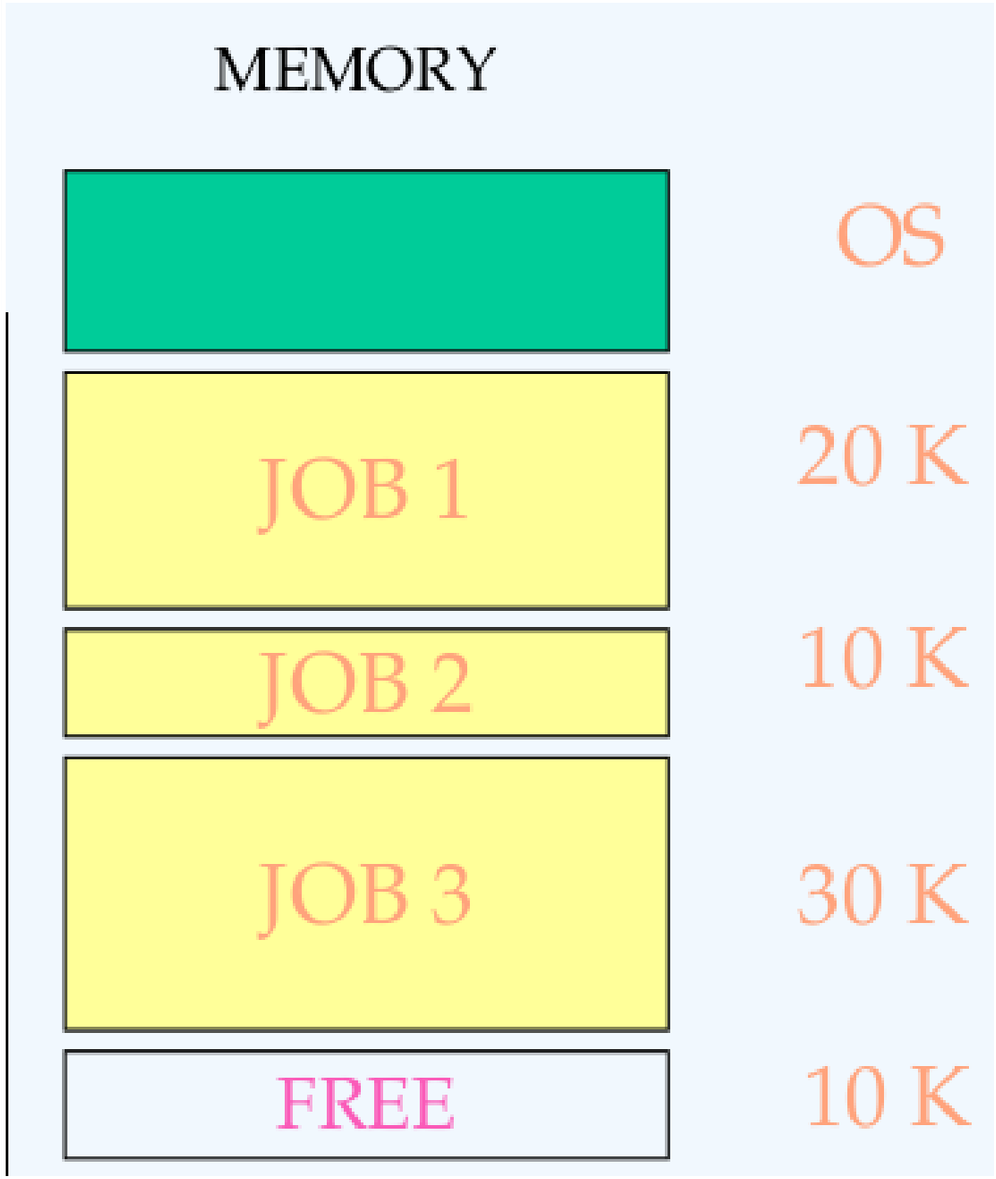
Disadvantages:

•Leads to Internal Fragmentation (Explained in the next slide)

Solution:

- •Relocatable partition
- •Paged allocation

Internal Fragmentation

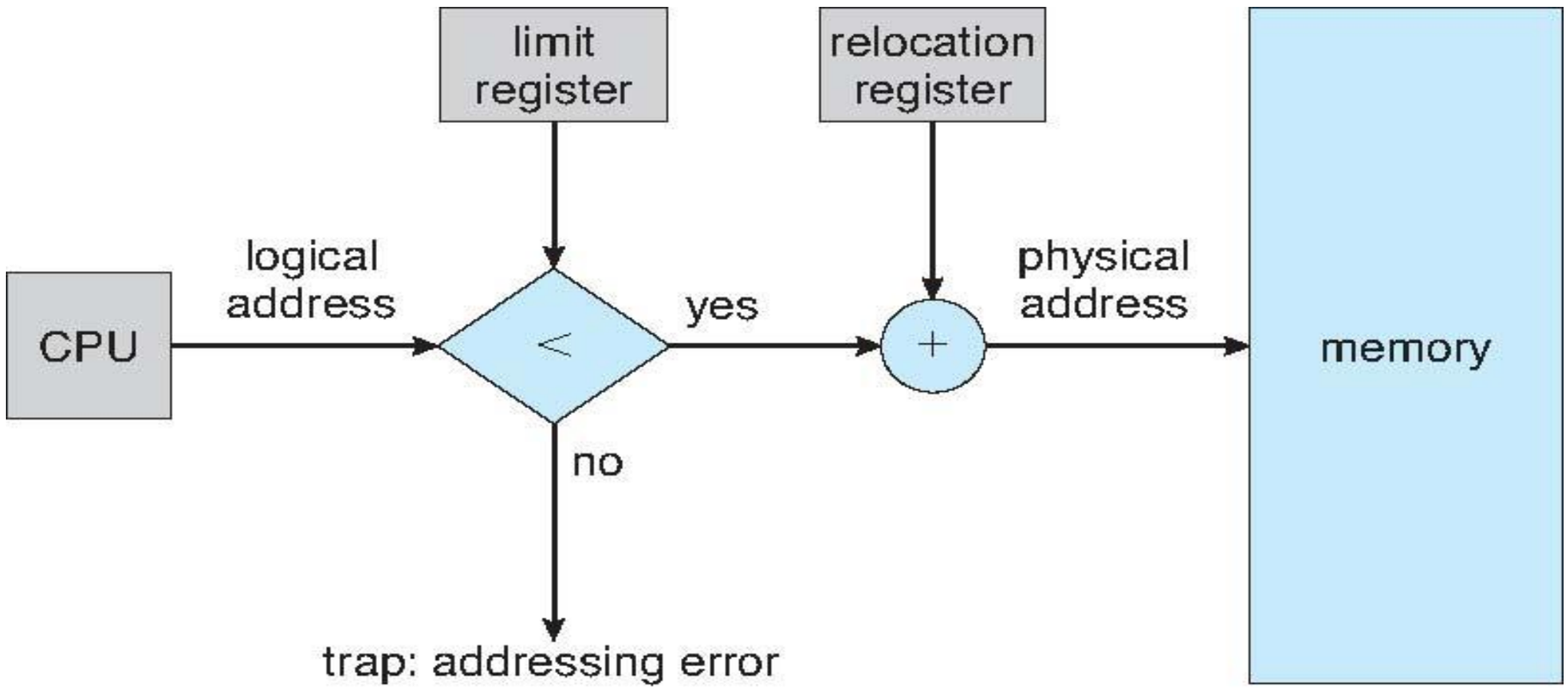


- Example:
  - If a job of 25K has to be executed, it has to go into 30K slot resulting in wastage of 5K. This occurrence of free space within the active process space is called as Internal Fragmentation.
- If the Required memory is less than the allocated memory then it is called as Internal Fragmentation.
- Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Contiguous Allocation

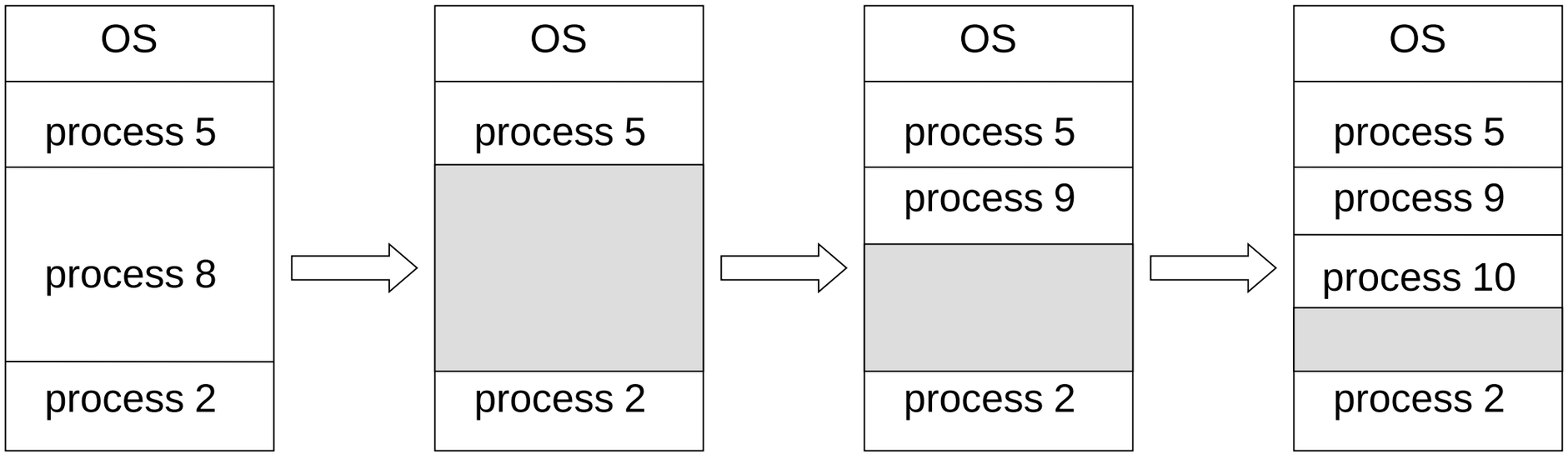
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, 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



Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - 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)



Variable Partitioned Allocation

Variable Partitioned Allocation

- No predetermined partitioning of memory – allocates the exact amount of memory space needed for each process as and when required
- Processes are loaded into consecutive areas until the memory is filled or remaining space is too small to accommodate a new process.
- Here, the partitions are not fixed. As and when the jobs come, they take up consecutive space in memory.
- Disadvantage
  - External Fragmentation

External Fragmentation

- When a process terminates, the space that was occupied by it is freed and these free spaces are called ‘holes’
- When the holes are formed between active (running) processes, even though the total free space may be sufficient to hold a new process, there may not be a single large enough hole to accommodate the incoming process.
- This kind of wastage which occurs outside the space allocated for an active process is called External Fragmentation.
- Definition: **Total memory space exists to satisfy a request, but it is not contiguous**

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

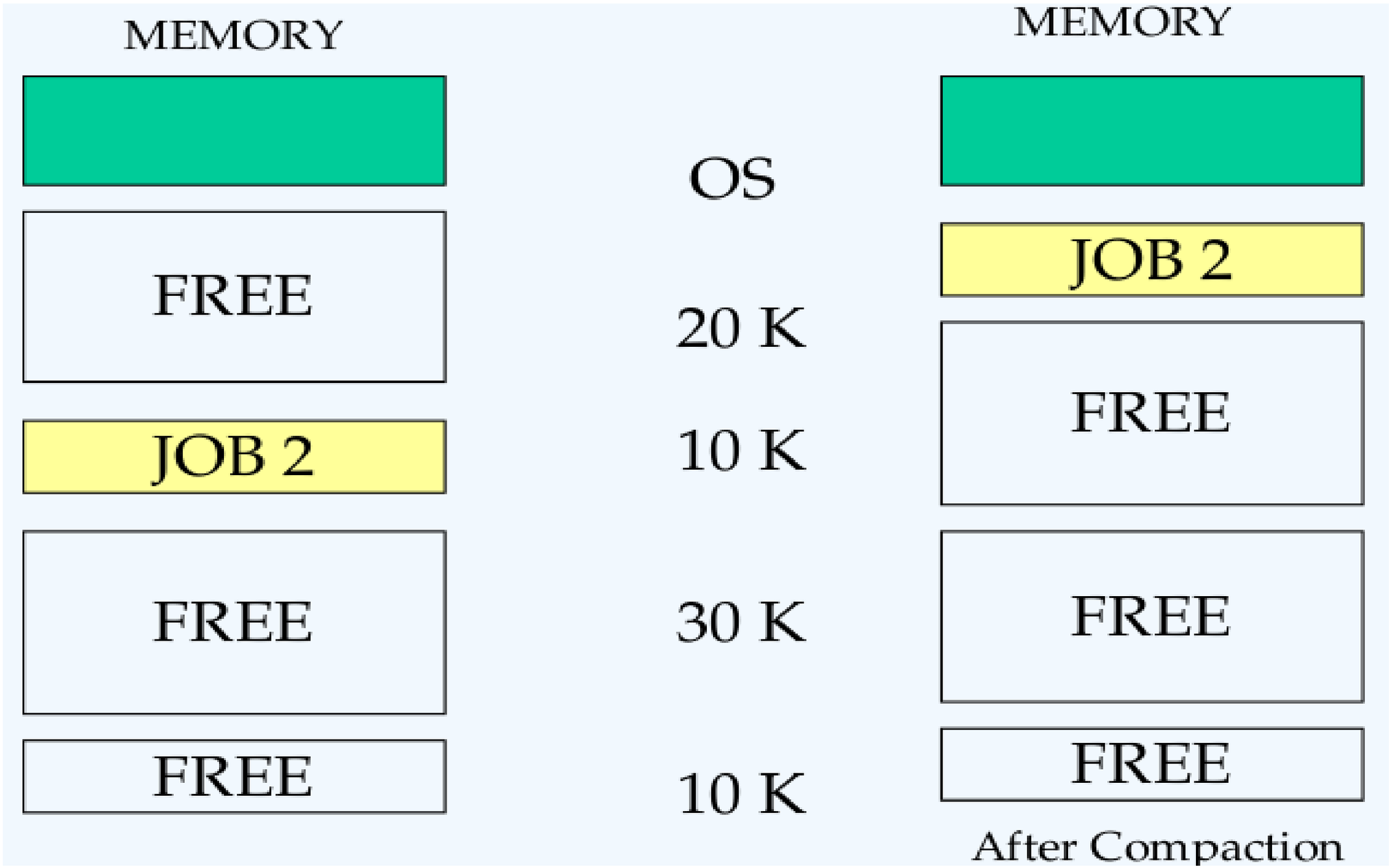
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

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

Relocatable Partitioned Allocation(1)

- Space wasted due to fragmentation can be used by doing compaction – running jobs are squeezed together by relocating them and clubbing the rest of the free space into one large block.
- Simple to implement
- The active processes shifts to one end, leaving the holes to combine to get a much larger space than before.
- Compaction is not always possible, however, if relocation is static and is done at assembly or Load time, Compaction Cannot be done.
- Compaction is possible only if Relocation is Dynamic and is done at execution time.

Relocatable Partitioned Allocation(2)



Relocatable Partitioned Allocation(3)

- The diagram above shows that JOB2 has been moved upward leaving 50 K of contiguous free space. A new job of 50 K can be run now.
- Disadvantage:
  - Relocating the running jobs afresh leads to problems that are address dependent
- Solution:
  - Reload & start from beginning every programs that needs to be relocated which is very expensive and at times is an irreversible action
  - Relative addressing mechanism wherein the job is run independent of any program location.
  - The disadvantage of Relative Addressing is that an extra overhead would be incurred because of a separate index register and addressing through the index registers.

Paging(1)

- 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 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size *N* pages, need to find *N* free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages

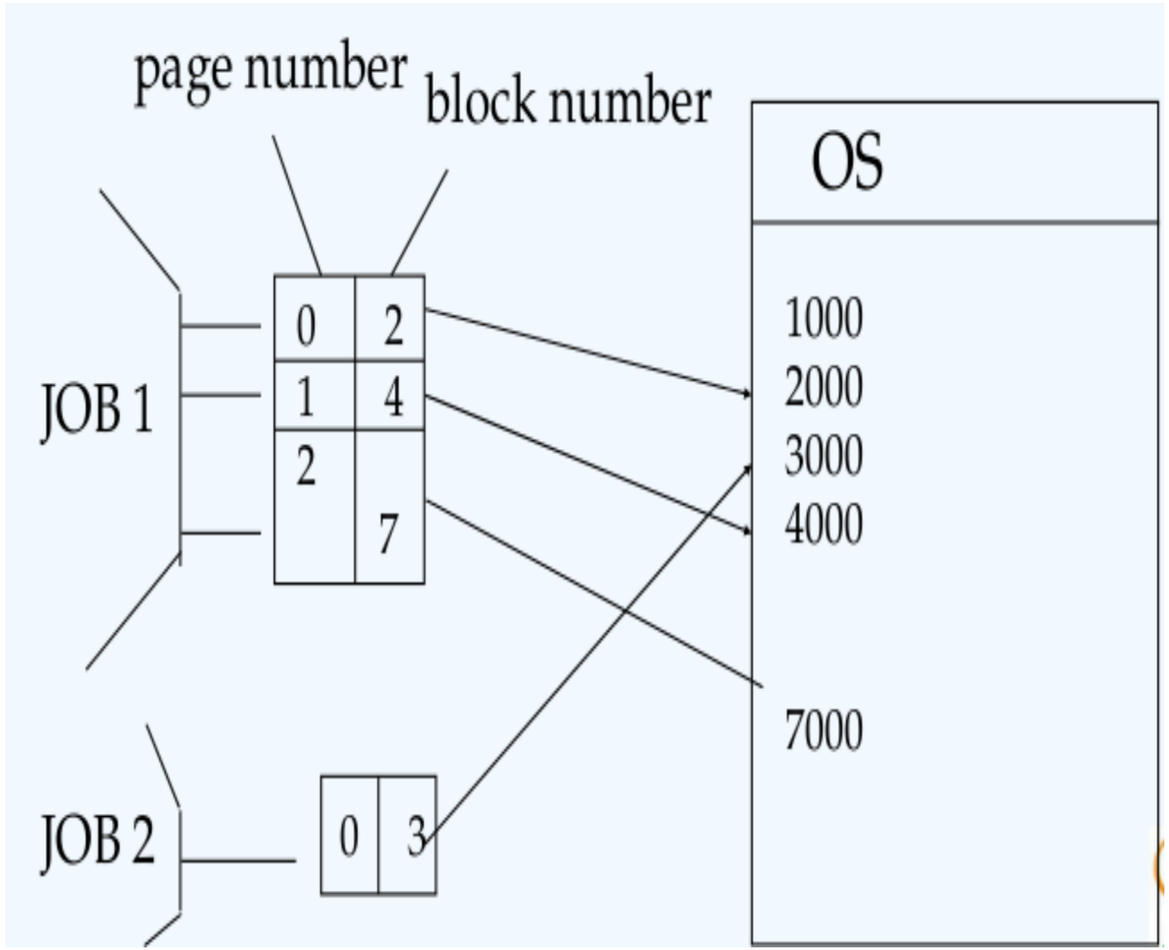
Paging(2)

- Simple paged allocation is a solution for External fragmentation.
- Divides the jobs address space into pages of the same size(4K)
- Divides the main memory address space into blocks/frames(4K)
- Pages are units of memory that are swapped in and out of primary memory
- Pages are grouped together for a given user job and are placed in page tables

Paging(3)

- Advantage:
  - As each page is separately allocated, the users job need not be contiguous.
- Disadvantages:
  - Extra memory required for storing page tables
  - Considerable amount of hardware support is required for address transformations etc.
  - All pages of entire job must be in memory
  - Still have Internal fragmentation

Simple Paging Example



The example in the slide shows a page table with 2 columns viz., page number and block (frame) number which essentially shows the mapping between page number and block number.

JOB1 has 3 pages viz., 0,1,2. Page 0 maps to block 2 in the RAM, page 1 to block 4 and page 2 to block 7 in the RAM.

JOB 2 has 1 page i.e. page 0 which maps to block 3 in the OS.

Thus, we can see that pages of a job need not be located contiguously in the memory.

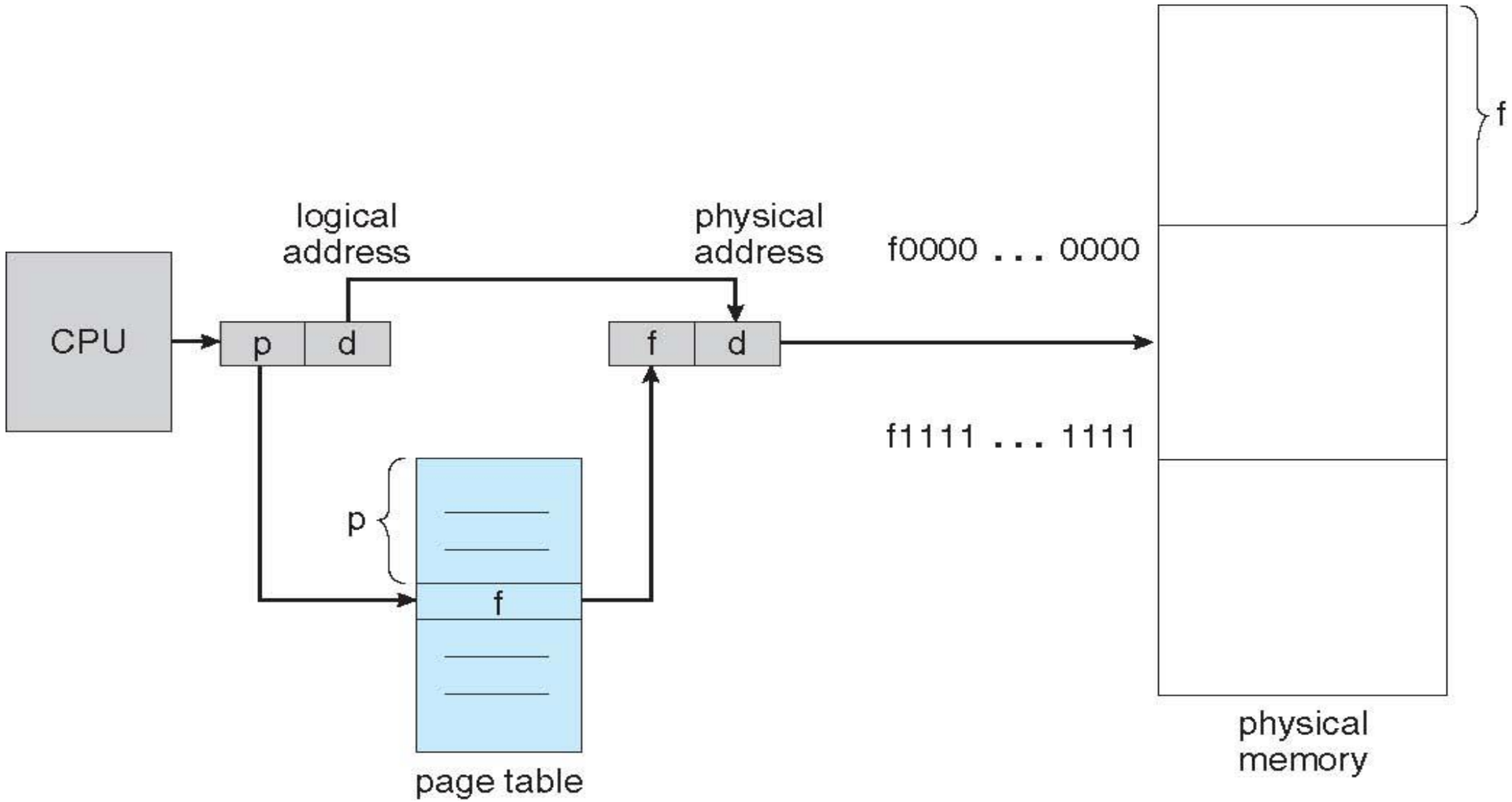
Address Translation Scheme

- 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

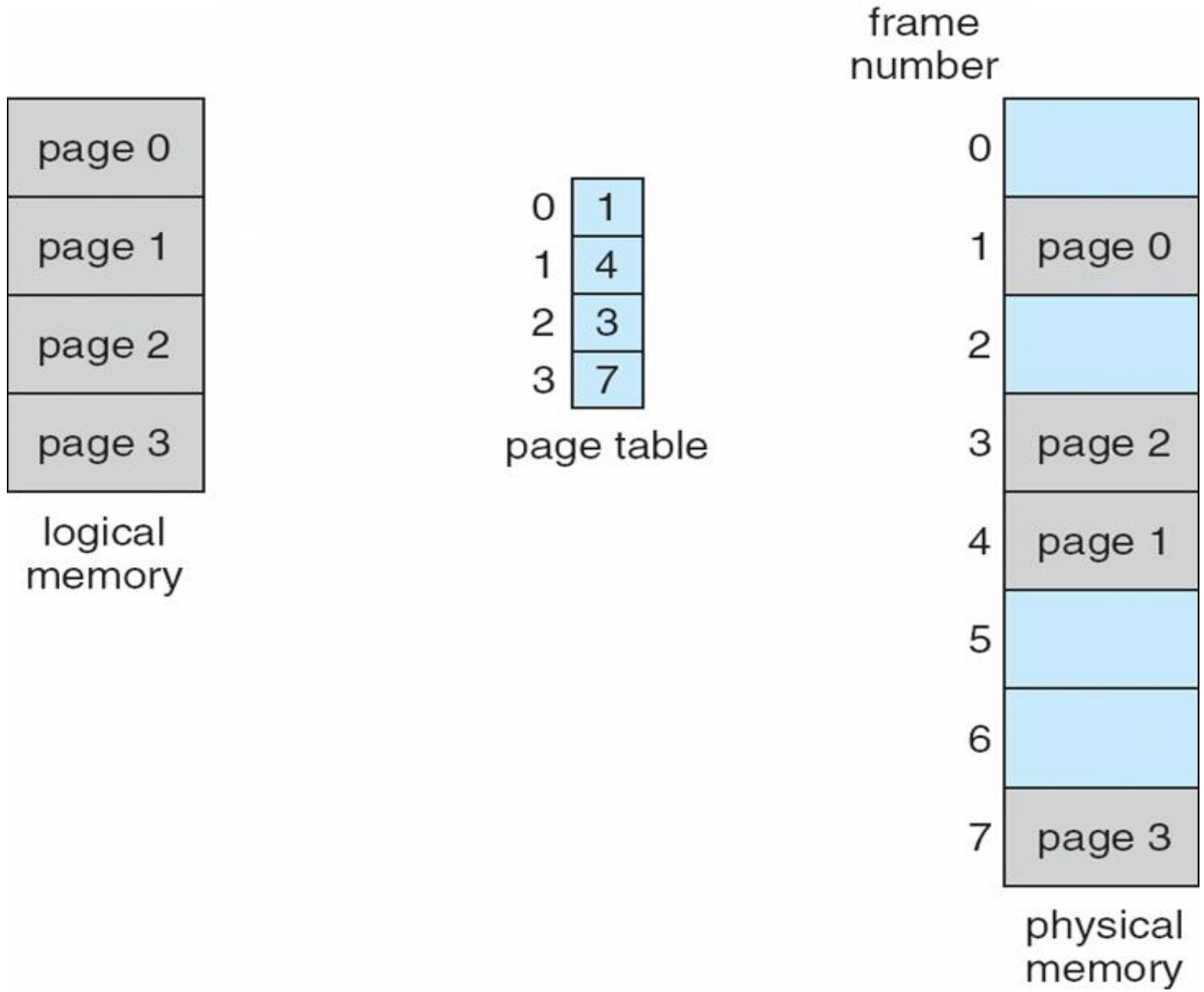
page number	page offset
p	*d*
m - n	*n*

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

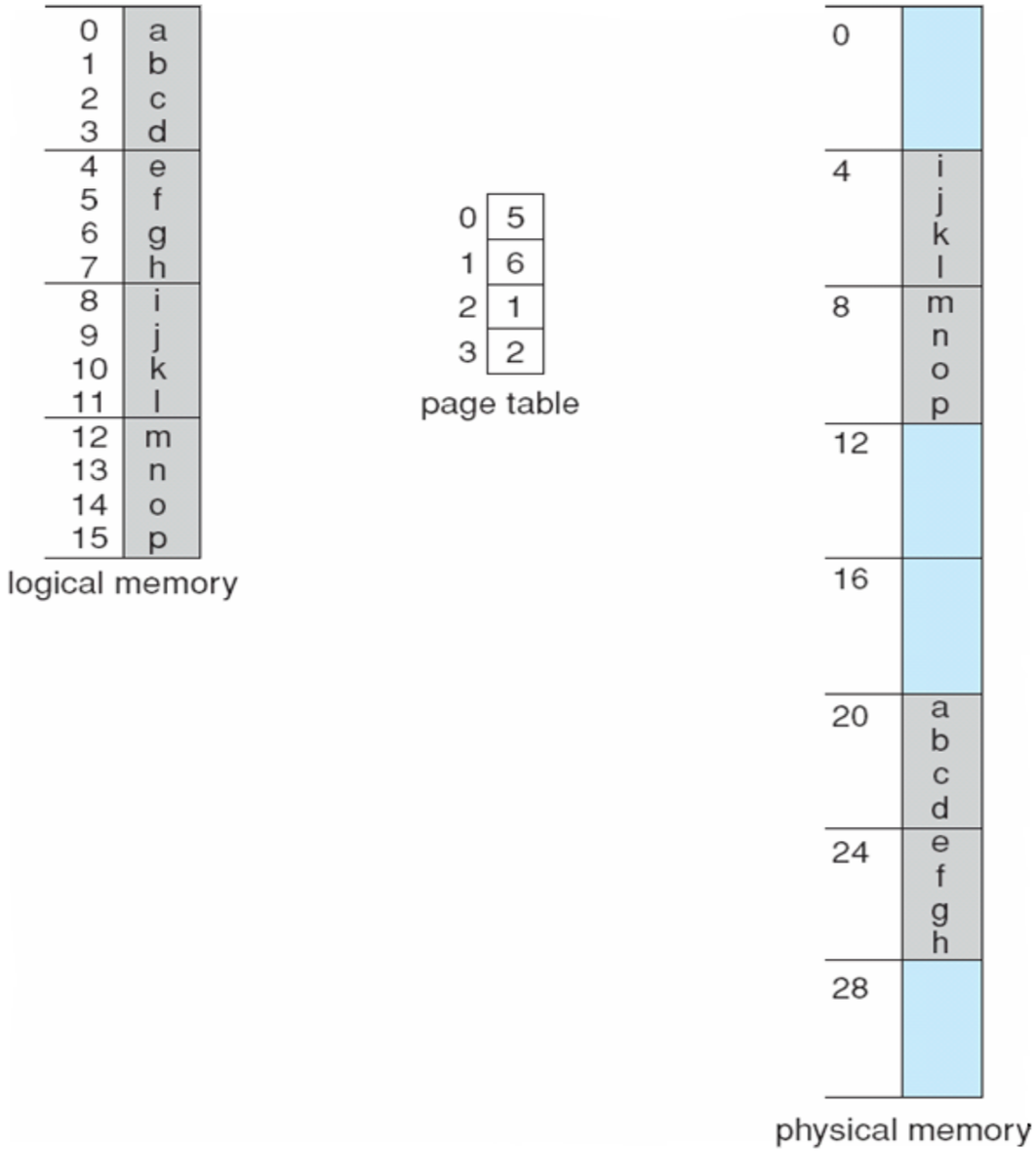
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example

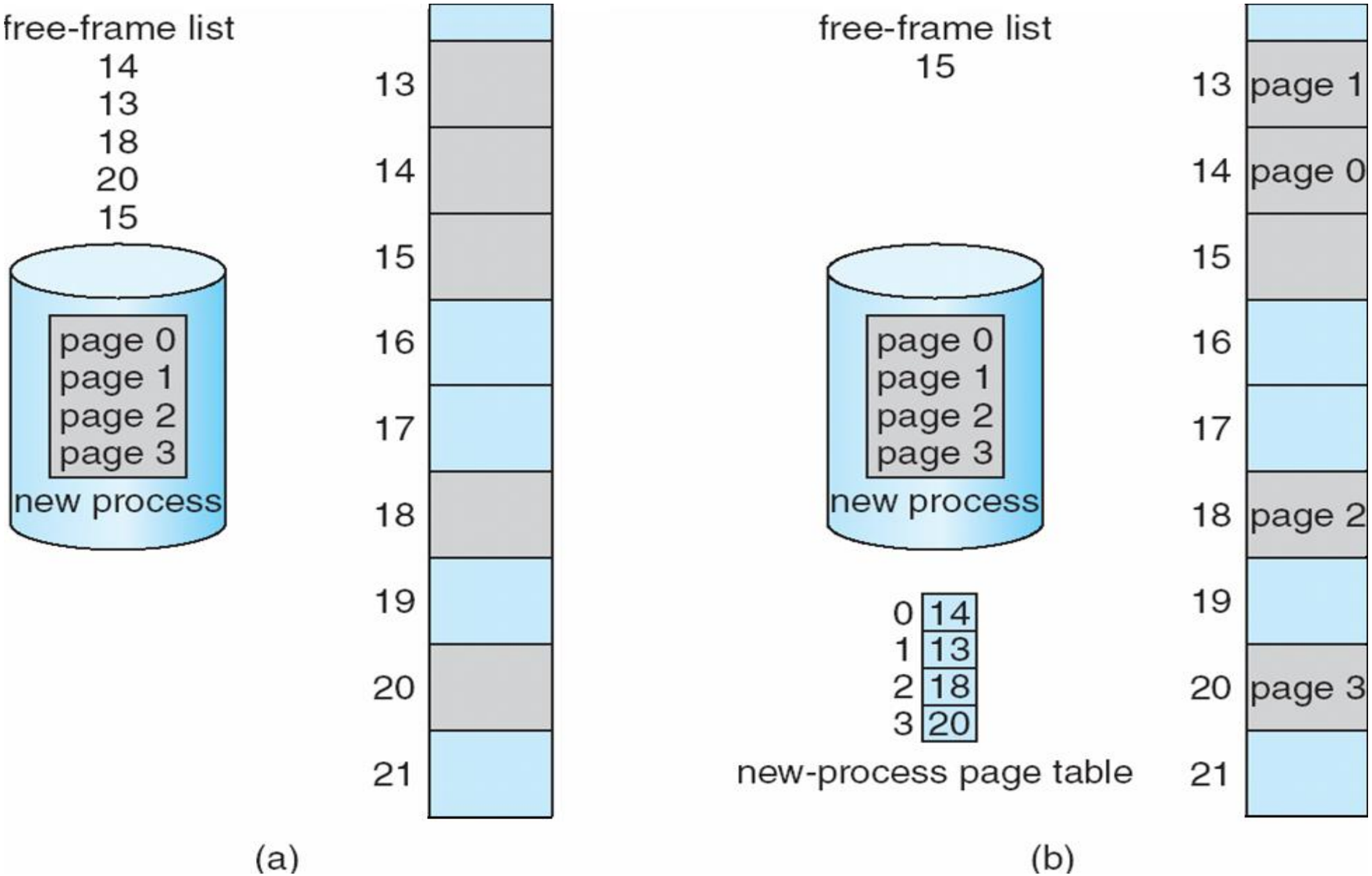


n=2 and *m*=4 32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

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
  - For the page table using the value in PTBR offset by the page number. It provides us the Frame number and requires a memory access
  - By using this Frame number and the offset, physical address can be calculated and access the desired byte in Memory (i.e, 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)**

TLB(1)

- Each entry in TLB consists of 2 parts
  - Key (or Tag)
  - Value
- The number of **entries in a TLB** is small, often numbering between **64 and 1024.**
- The TLB contains only a few of the page table entries.
- When a Logical address is generated by the CPU, its page number is given to TLB.
  - If the Page number is found, its Frame number is returned immediately and it can be used to access memory.
  - If the Page Number is not found in TLB(Called as TLB miss), a memory reference is made to the Page Table. When Frame number is obtained, we can use it to access memory.
  - In addition, we add the page number and Frame number to the TLB. If the TLB is already full of entries, the OS must select one for replacement.
  - Some TLBs allow some entries to be "Wired down", meaning that they cannot be removed from TLB. TLB entries for Kernel code are wired down.

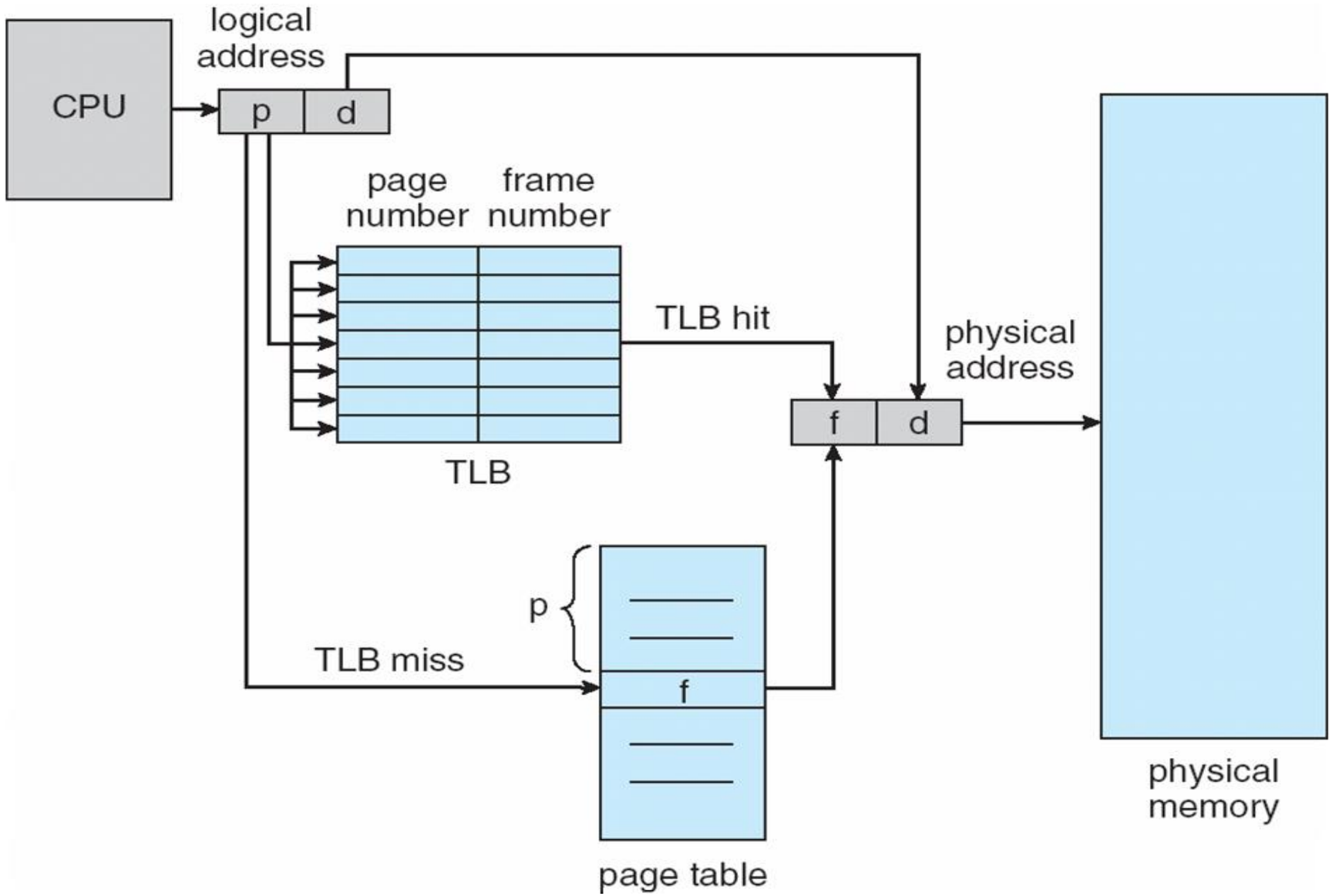
Associative Memory

- 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



TLB(2)

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
- TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently executing process matches the ASID associated with the Virtual page. If it doesn't match then it's a TLB miss.
- In addition to providing address space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.
  - Otherwise need to flush at every context switch.

TLB(3)

- The percentage of times that a particular page number is found in the TLB is called as "hit ratio".
- Example:

hit ratio = 80%

If it takes 20ns to search in TLB and 100ns to access memory then mapped-memory access takes 120 ns, if the page is available in TLB.

If page is not found in TLB, 20ns to search in TLB, 100ns for accessing page table and 100ns for memory access. So total of 220ns.

Effective access time = 0.80 *120 + 0.20*220 = 140ns.

So 40% slowdown in memory access time.

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 α = 80%, ε = 20ns for TLB search, 100ns for memory access

- **Effective Access Time (EAT)**

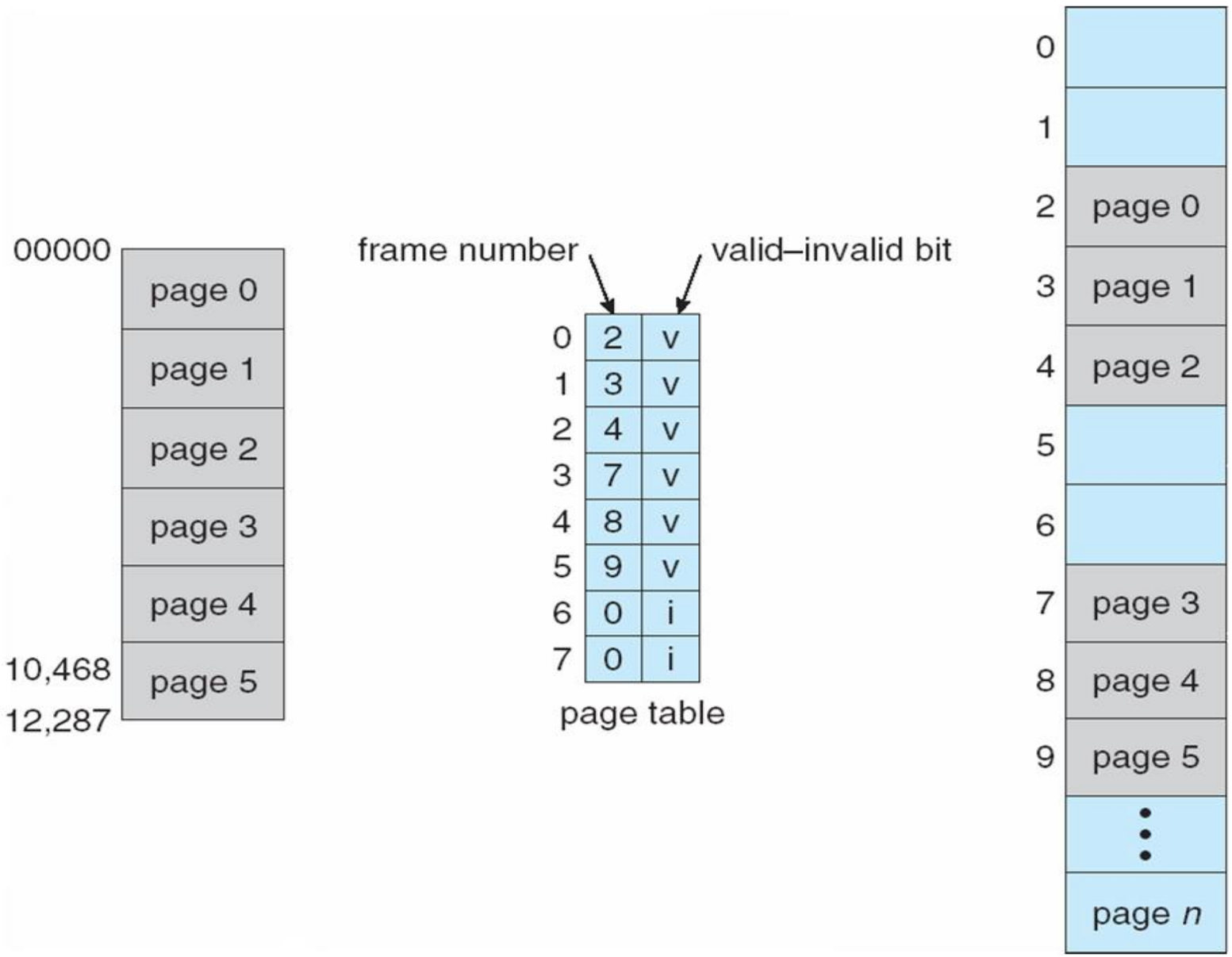
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider slower memory but better hit ratio -> α = 98%, ε = 20ns for TLB search, 140ns for memory access
  - EAT = 0.98 x 160 + 0.02 x 300 = 162.8ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use PTLR
- Any violations result in a trap to the kernel

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

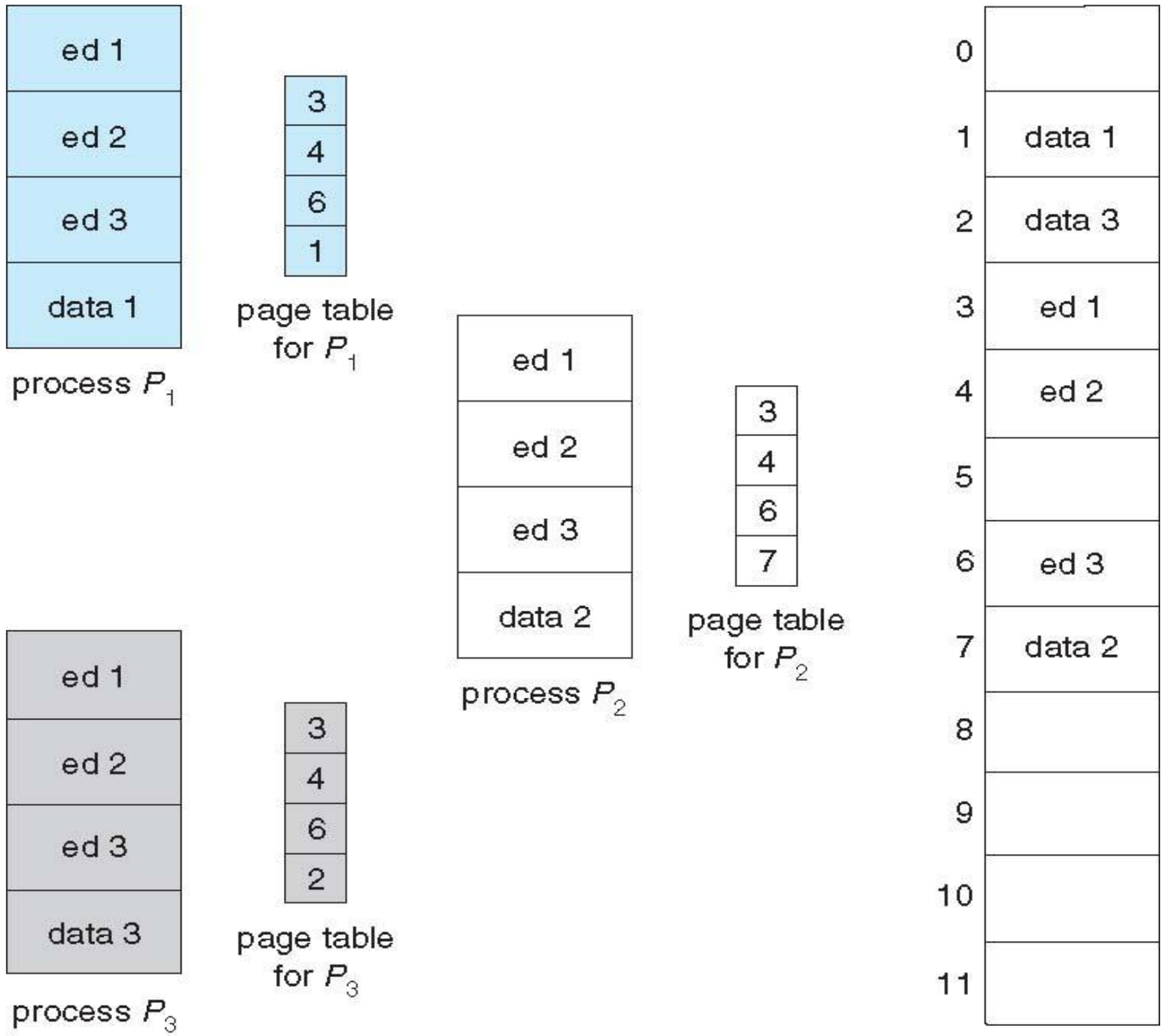


Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant: means it never changes during execution**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter process communication if sharing of read-write pages is allowed

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Structure of the Page Table

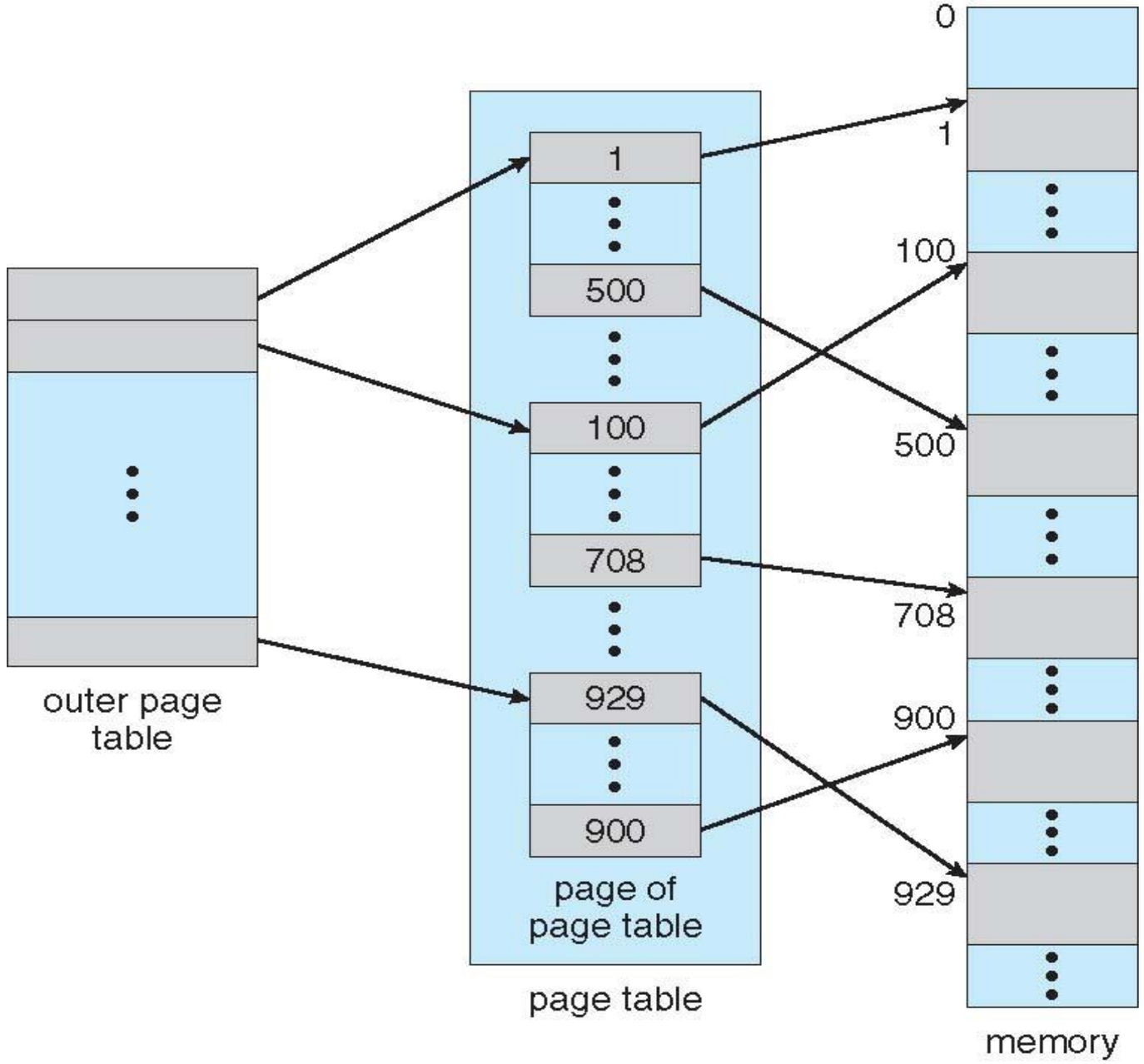
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / $2^{12)}$
  - If each entry is 4 bytes -> 4 MB of physical address space /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

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

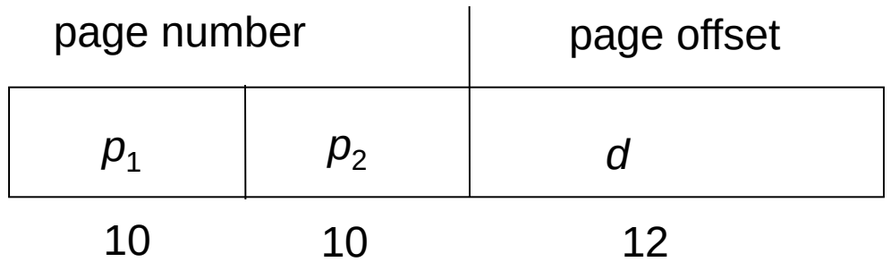
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



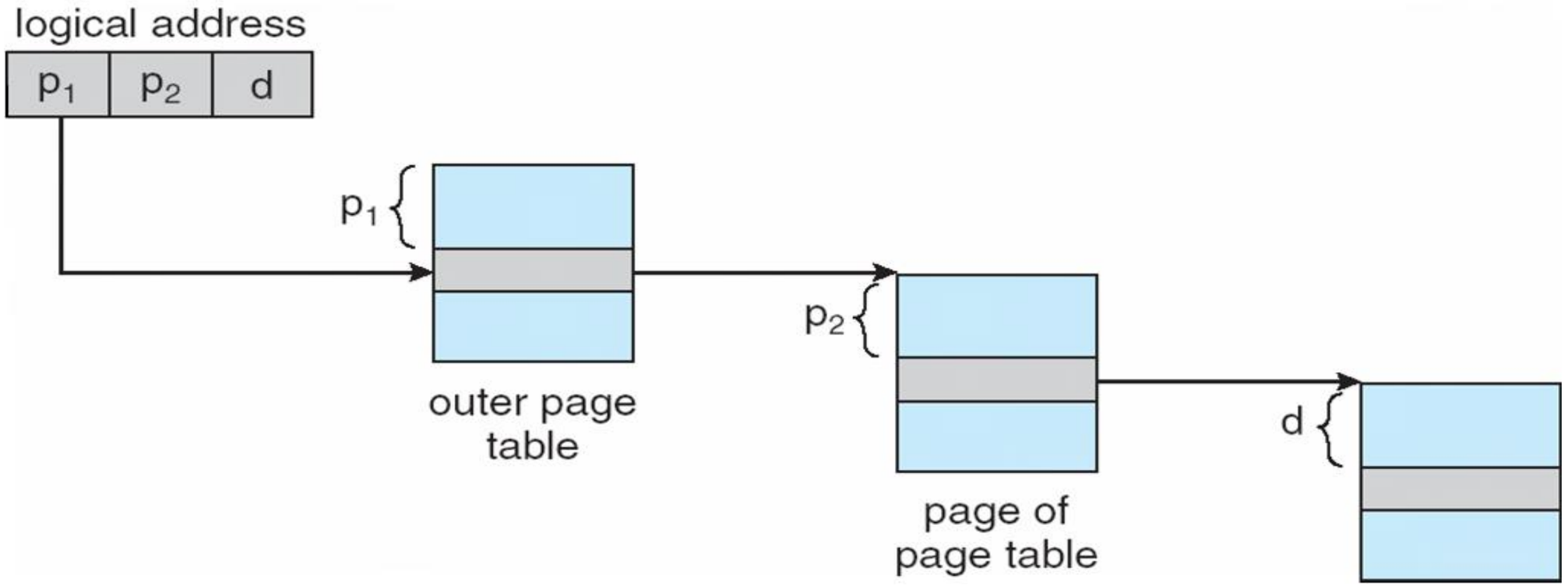
Two-Level Paging Example

- 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 12-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



Multilevel Page Tables

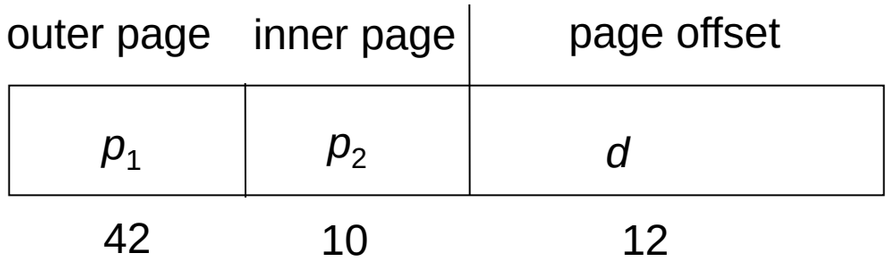
- Since the page table can be very large, one solution is to page the page table
- Divide the page number into
  - Index into page table of 2nd level page tables
  - Page within a 2nd level page table
- Advantage
  - No need to keep all the page tables in memory all the time
  - Only recently accessed memory's mapping need to be kept in memory, the rest can be fetched on demand
  - System can store in noncontiguous locations in main memory those portions of process's page table that the process is using
- Can reduce memory overhead compared to direct-mapping system

TLB

- Compromise between cost and performance
  - Most PTEs are stored in direct-mapped tables in main memory
  - Most-recently-used PTEs are stored in high-speed set-associative cache memory called a Translation Lookaside Buffer (TLB)
    - If PTE is not found in TLB, the DAT mechanism searches the table in main memory
  - Can yield high performance with relatively small TLB due to locality
    - **A page referenced by a process recently is likely to be referenced again soon**

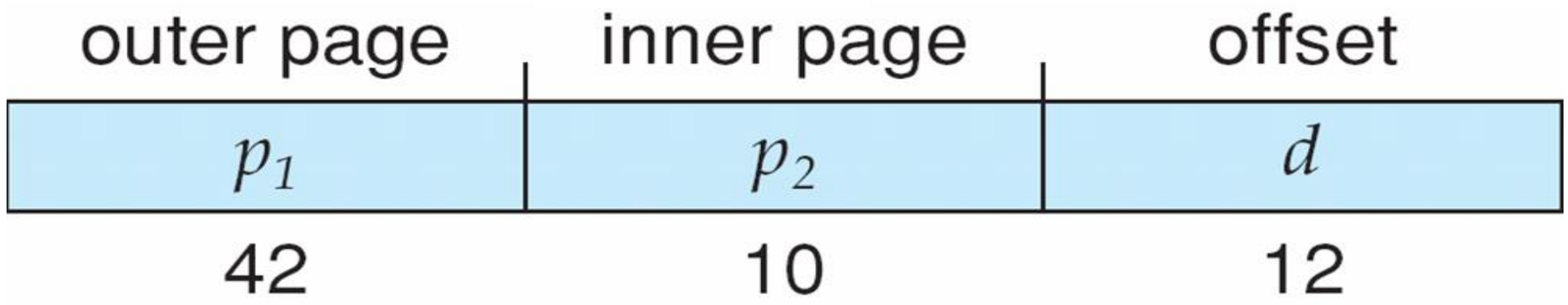
64-bit Logical Address Space

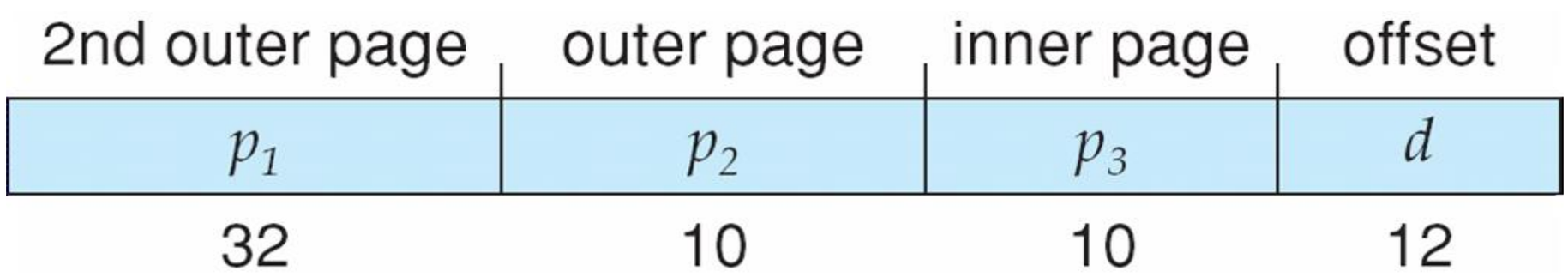
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



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

Three-level Paging Scheme

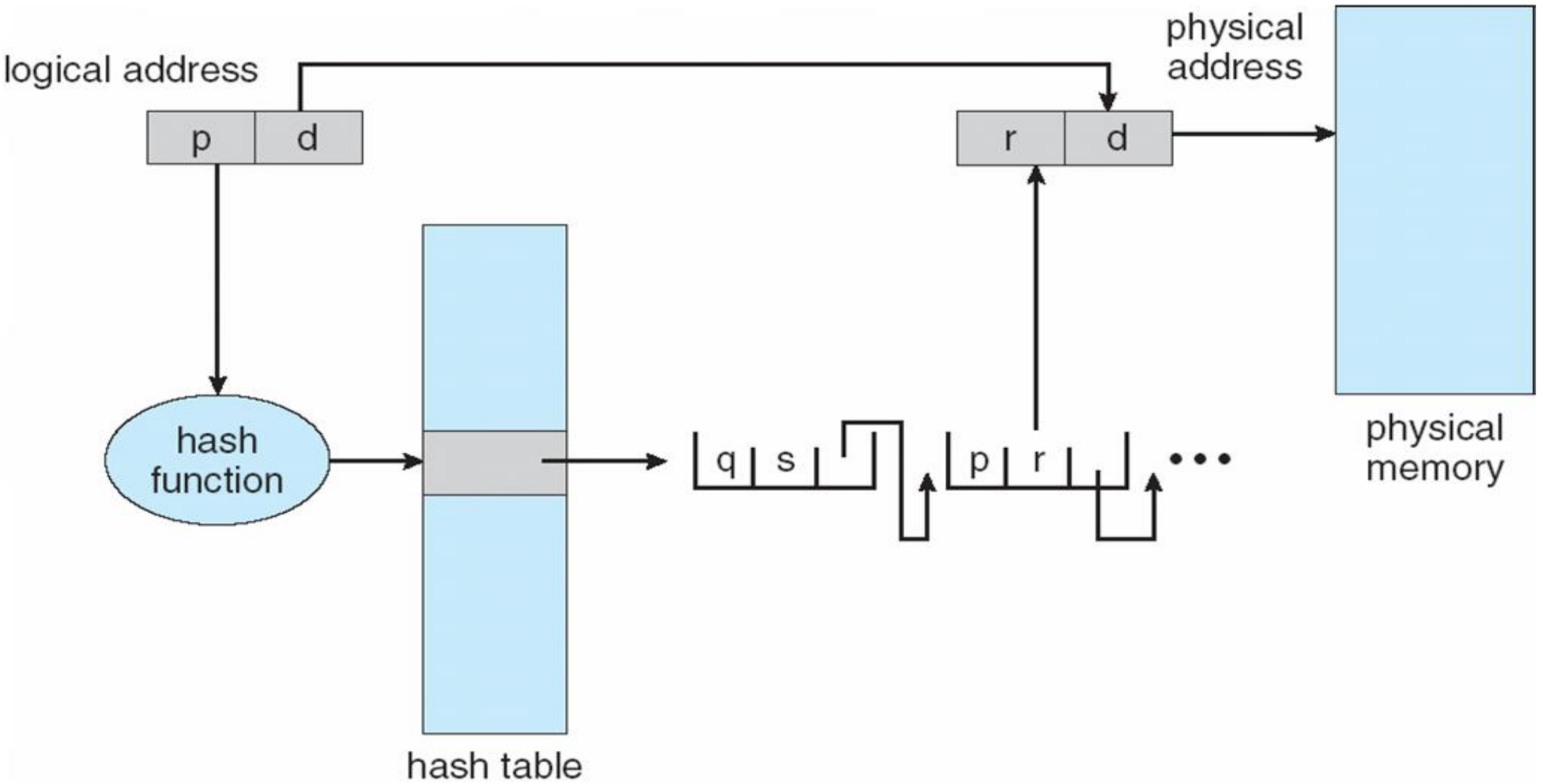




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location(Separate Chaining)
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

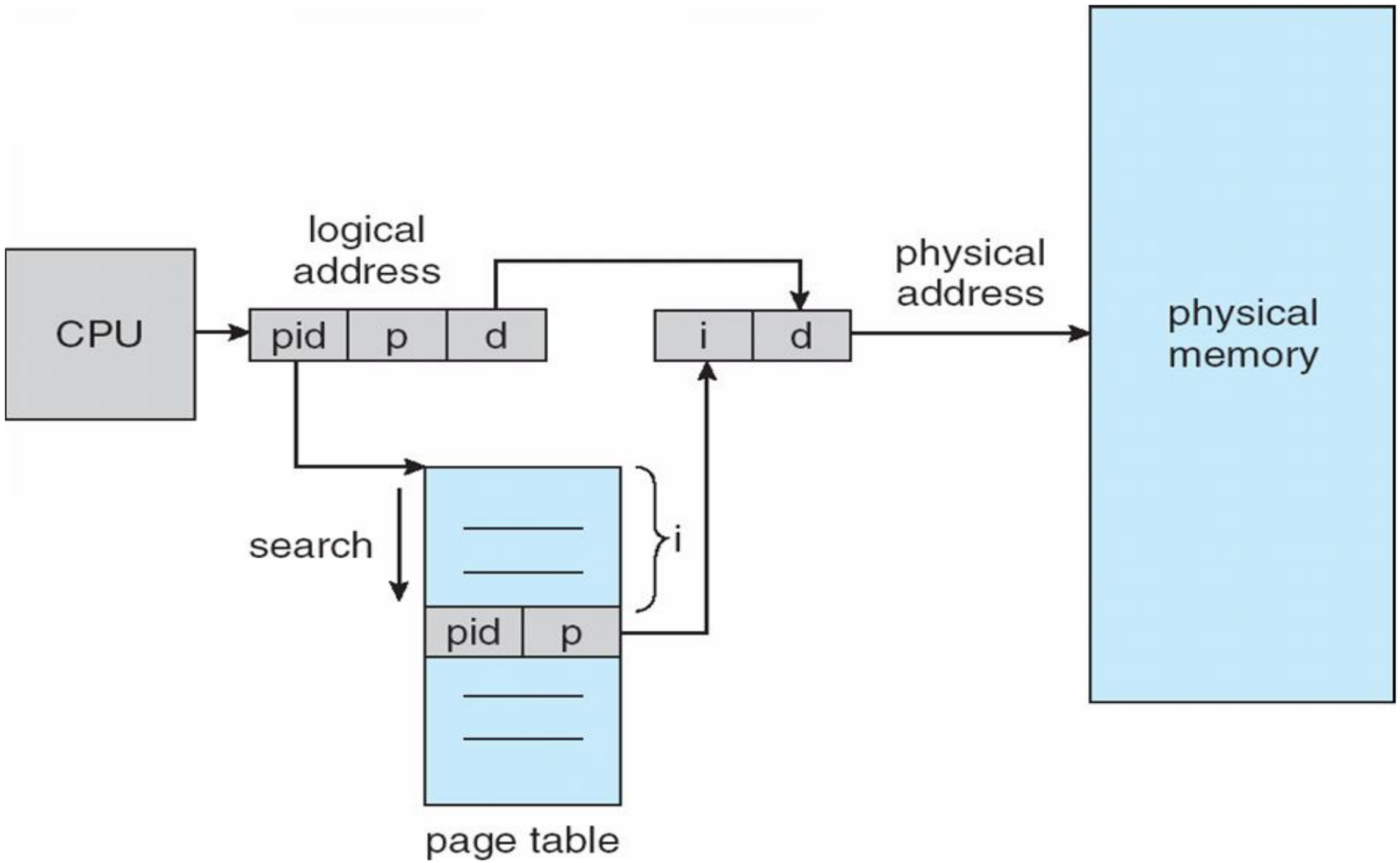
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page 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
  - <<pid, page num, offset>>
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
  - Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take long time.

Inverted Page Table Architecture



Segmentation

- Memory-management scheme that supports user view of memory
- Logical Address space is a collection of segments. Each segment has a name and length.
- Segment
  - Block of program's data and/or instructions
  - Contains meaningful portion of the program (e.g. procedure, array, stack)
  - Consists of contiguous locations
  - Segments need not be the same size nor must they be adjacent to one another in main memory
- A process may execute while its current instructions and referenced data are in segments in main memory

Segmentation

- Addresses specify both segment number and offset with in the segment. Thus a logical address consists of a 2 tuple: **<<segment-number, offset>>**
- 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

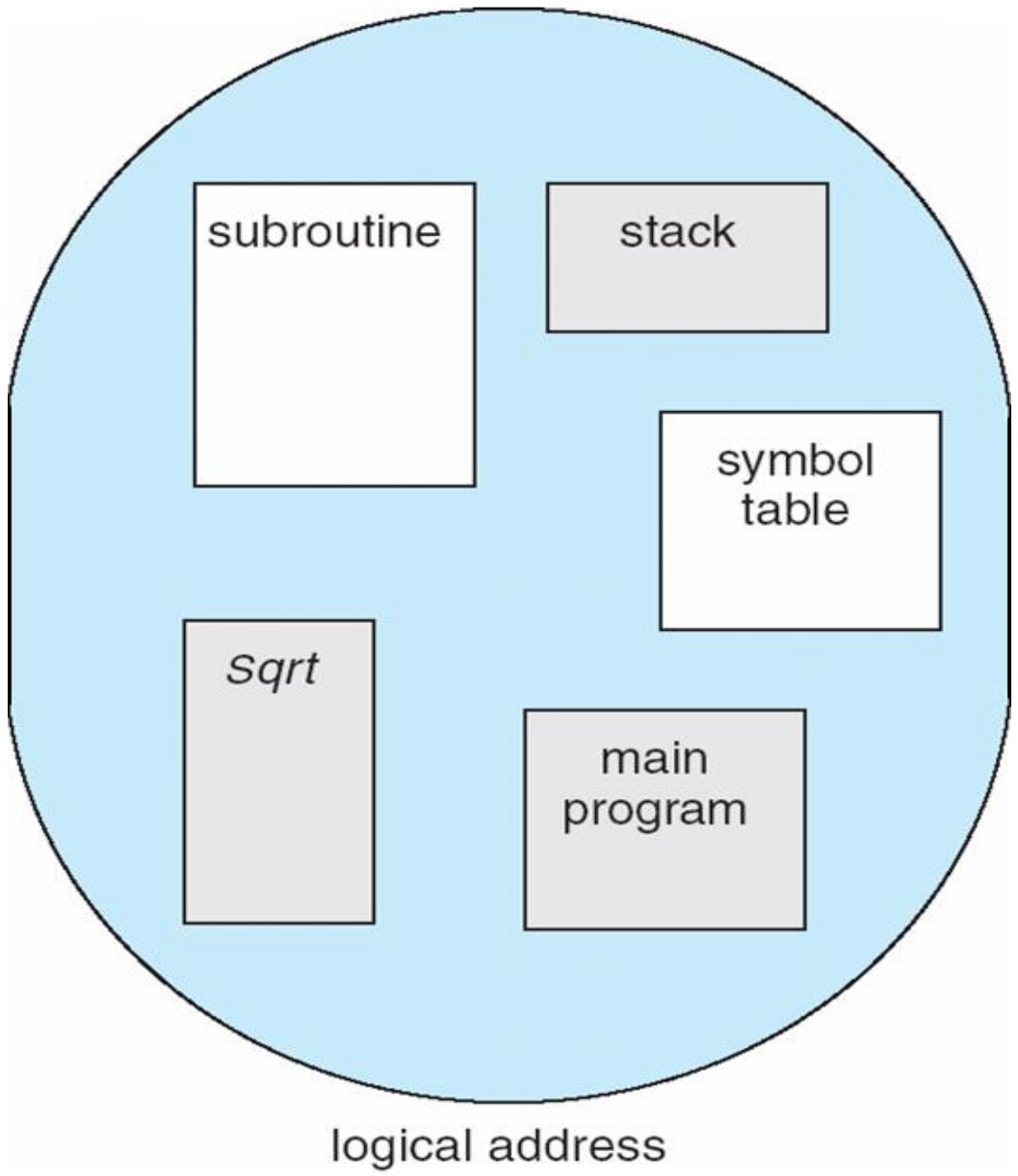
    common block

    stack

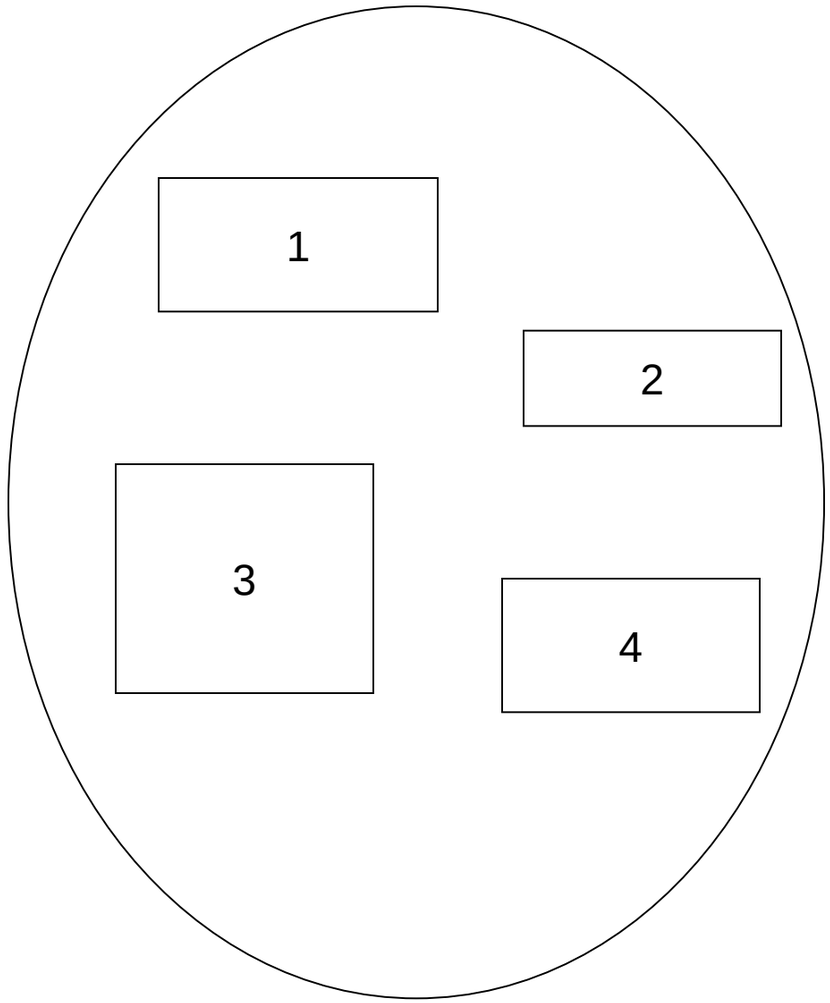
    symbol table

    arrays

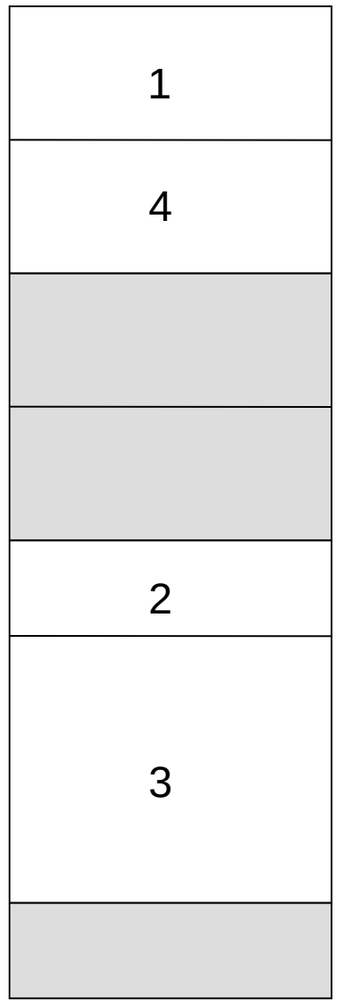
User's View of a Program



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.)

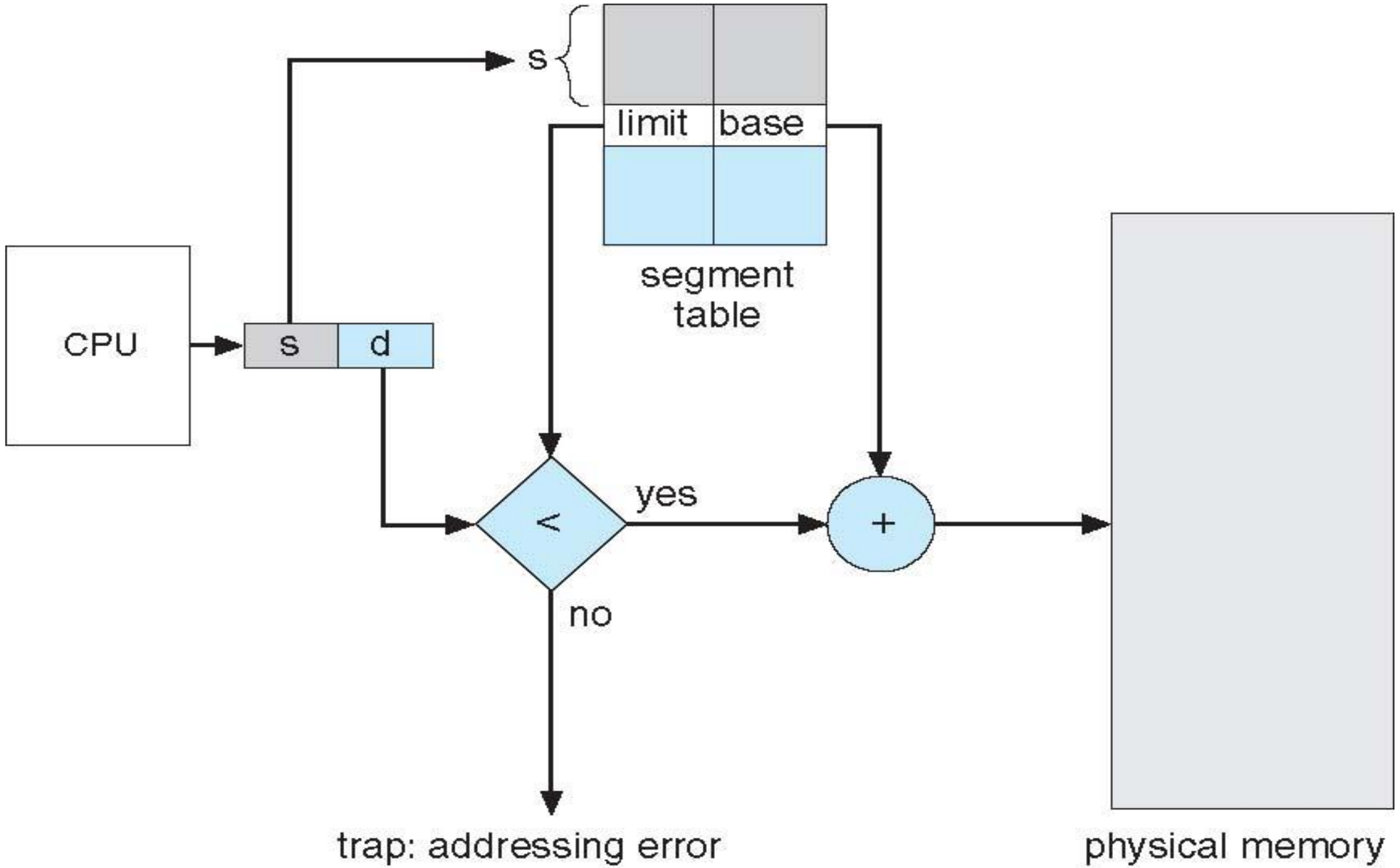
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

- Protection bits associated with segments; **code sharing occurs at segment level**

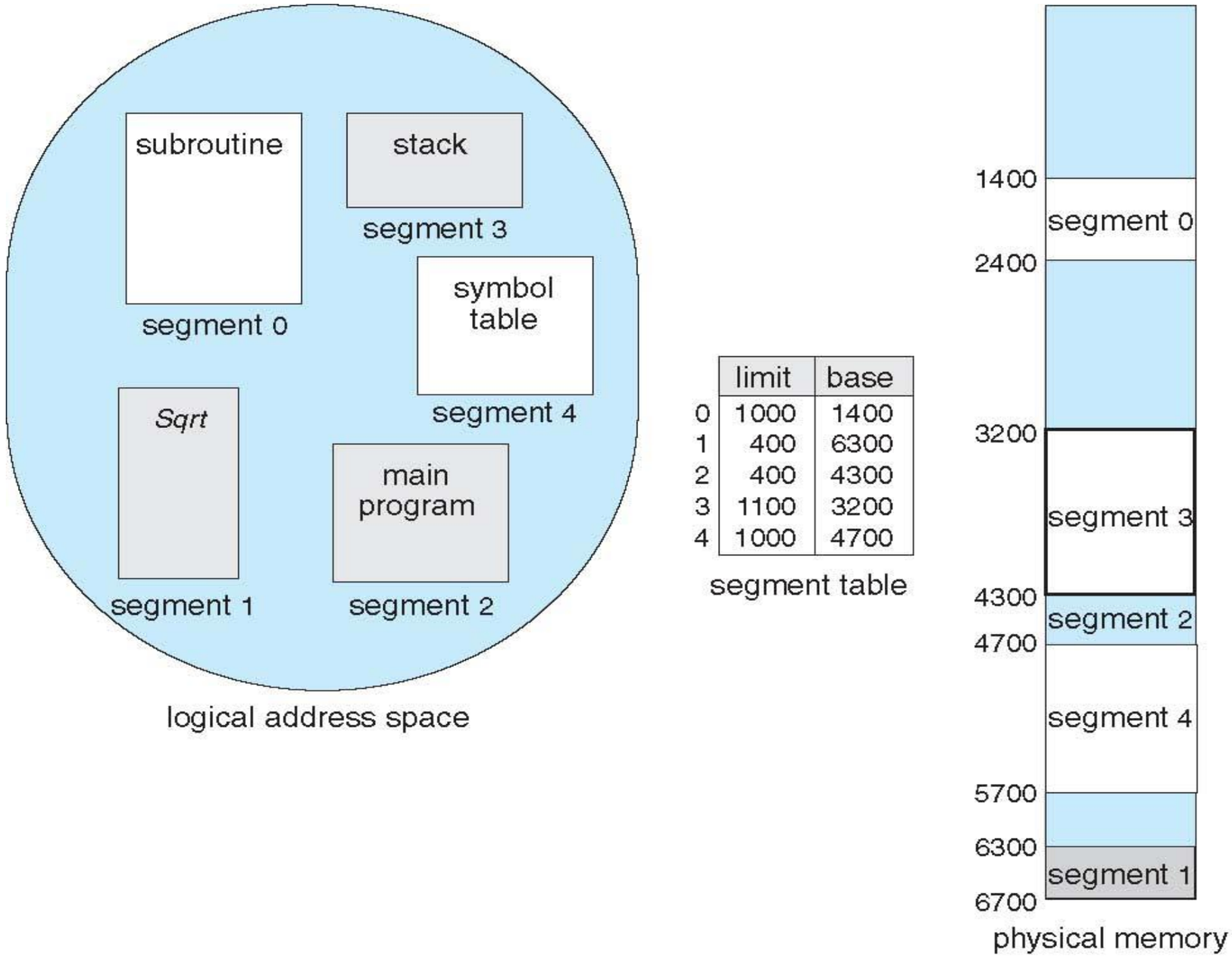
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



Thank You