Operating Systems: Main Memory

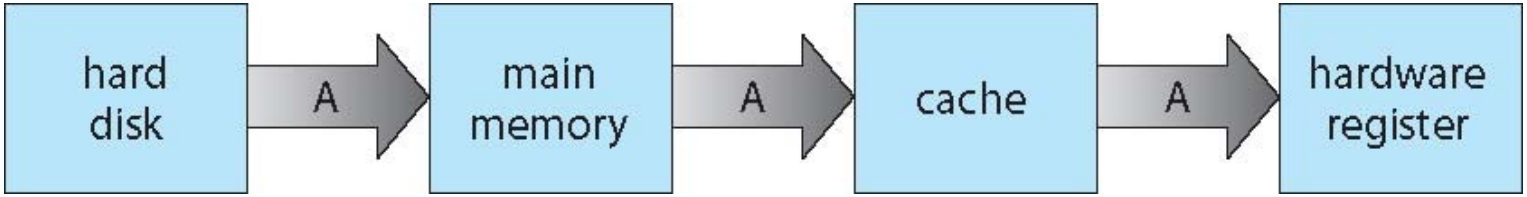
Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 9)

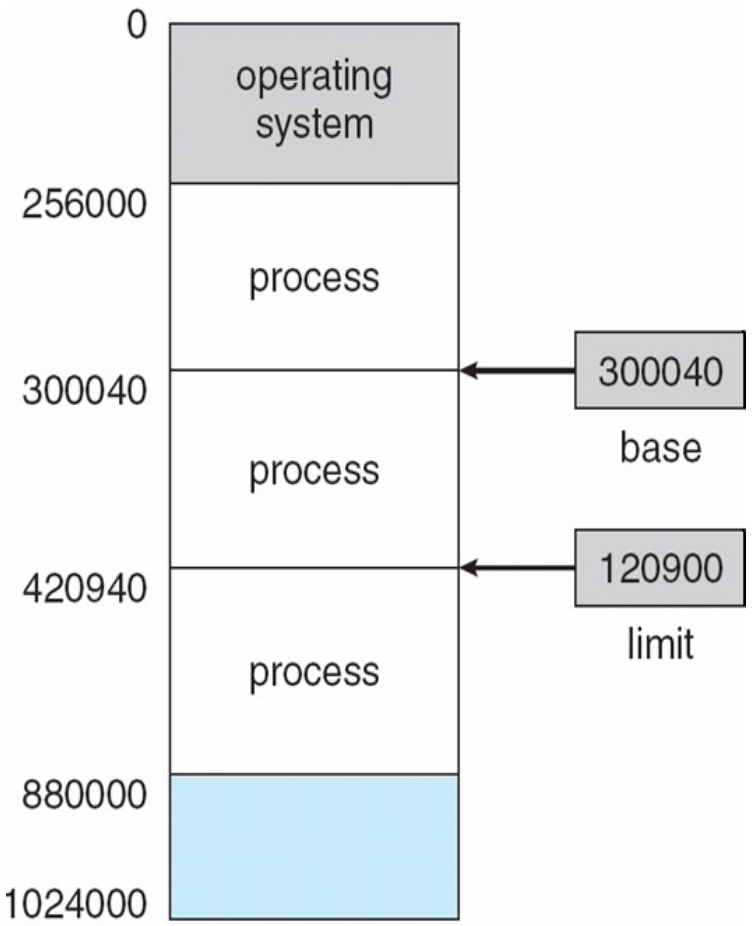
Background

- **Main memory is simply an array of bytes.**
- Main memory, Cache and registers are **only storage CPU can access** directly
- Program must be brought (from disk) into main memory for execution



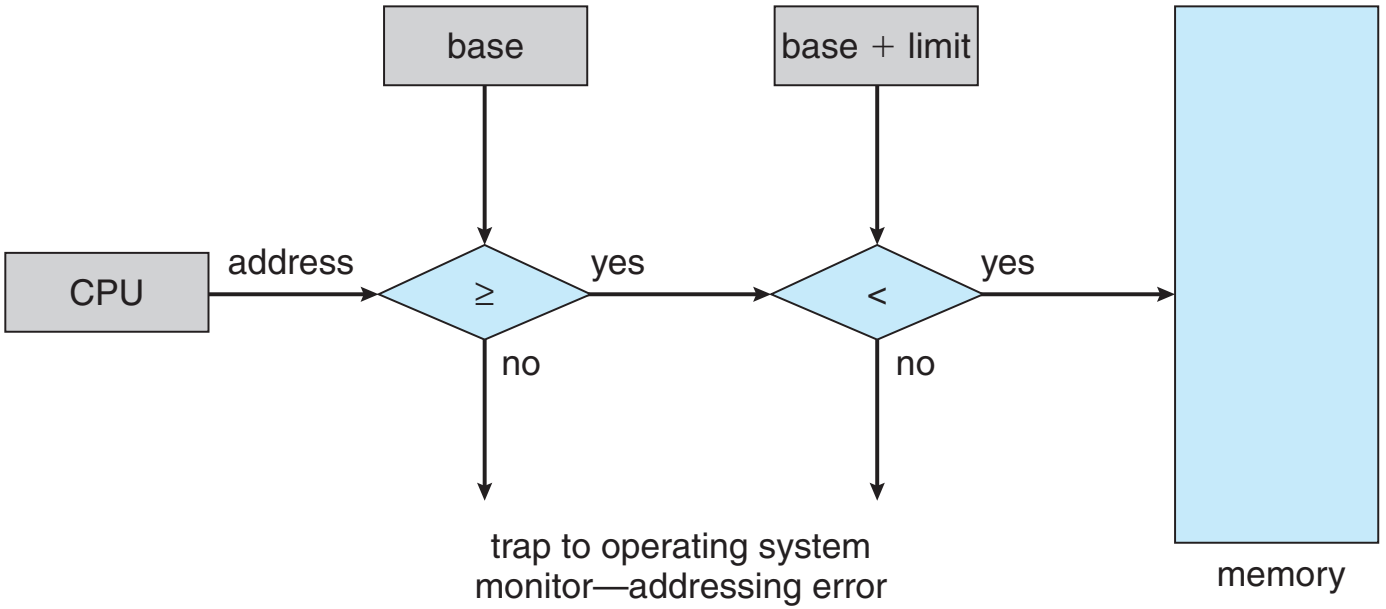
Memory protection - Multi-programming environment

- In a multi-programming environment, *many processes are in main memory* each having its own memory space.
- Protection of processes' memory is required to ensure correct operation
- This protection is provided by the below **hardware.**
  - **base registers** - smallest legal physical memory address, and
  - **limit registers** - size of the address space of the program.



Hardware Address Protection with Base and Limit registers

- Program generates addresses from 0. However, instructions are not stored in memory at address 0.
- Hence, during execution to access instructions CPU hardware compares every address generated by the program with the base and limit registers
- Lets it access the memory address only if its with in the legal range
- Base and limit register values **loaded only by the operating system**



Logical vs. Physical Address Space

- **Logical address** – addresses generated by the CPU
  - Also referred to as **virtual address**
  - Range from 0 to *max* = size of the address space of the program
- The user program generates only logical addresses and thinks that the process runs in locations 0 to *max.*
- However, these logical addresses must be **mapped** to physical addresses before they are used.
- **Physical address** – address in main memory and seen by the memory unit
  - Range from *R* + 0 to *R* + *max,* where R = base value.
- **Logical address space** is the **set of all logical** addresses generated by a program
- **Physical address space** is the **set of all physical** addresses assigned to the program

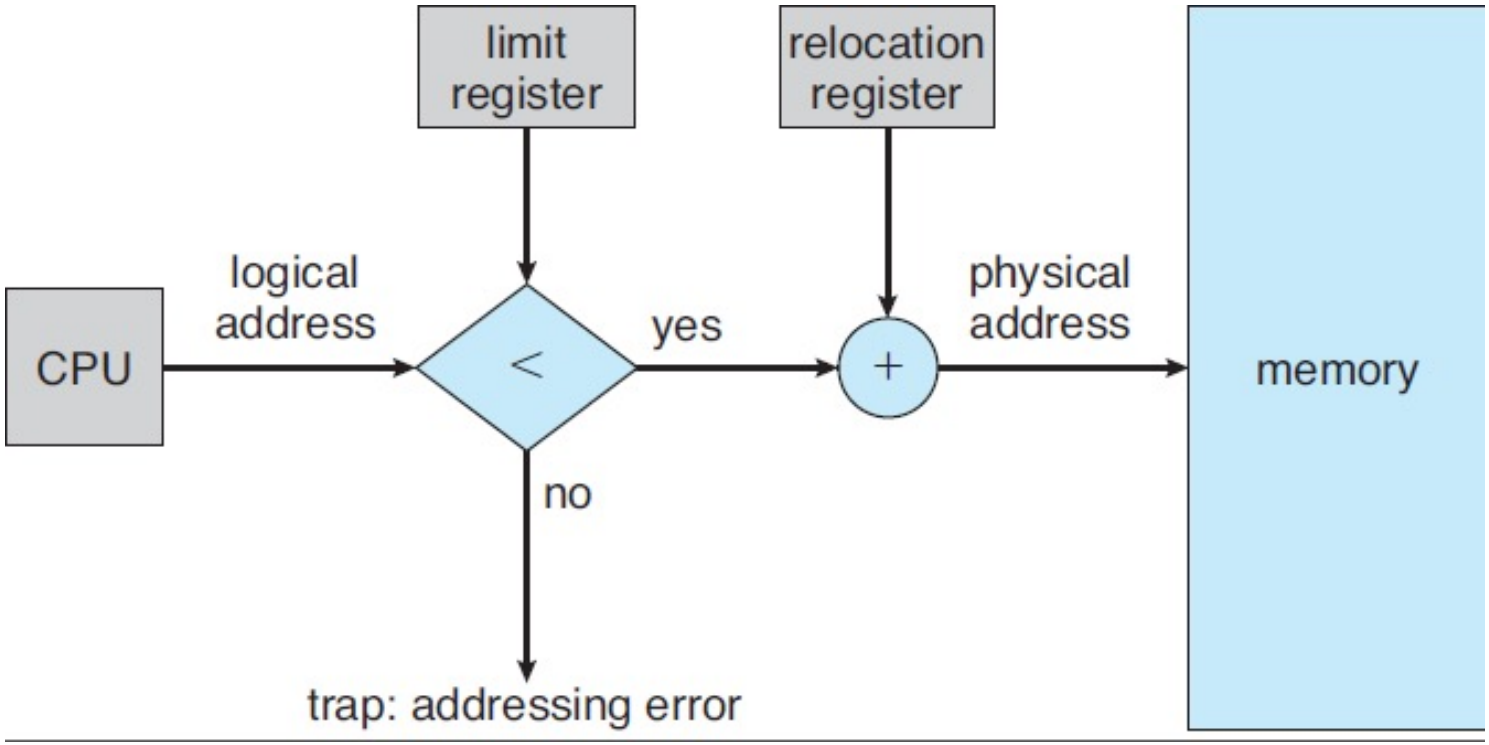
Memory-Management Unit (MMU)

- **Memory-Management Unit (MMU)** – is a **hardware device** that at run time maps/translates logical/virtual addresses to physical addresses
- This mapping based on below schemes
  - contiguous memory allocation (obsolete)
  - segmentation
  - paging

Contiguous Memory Allocation

- Main memory usually divided into two **partitions**:
  - Operating system held in low memory with interrupt vector
  - User processes held in high memory
    - Each process contained in single **contiguous** section of memory.
    - Memory protection in this scheme is achieved using *Relocation (base) and limit registers*.

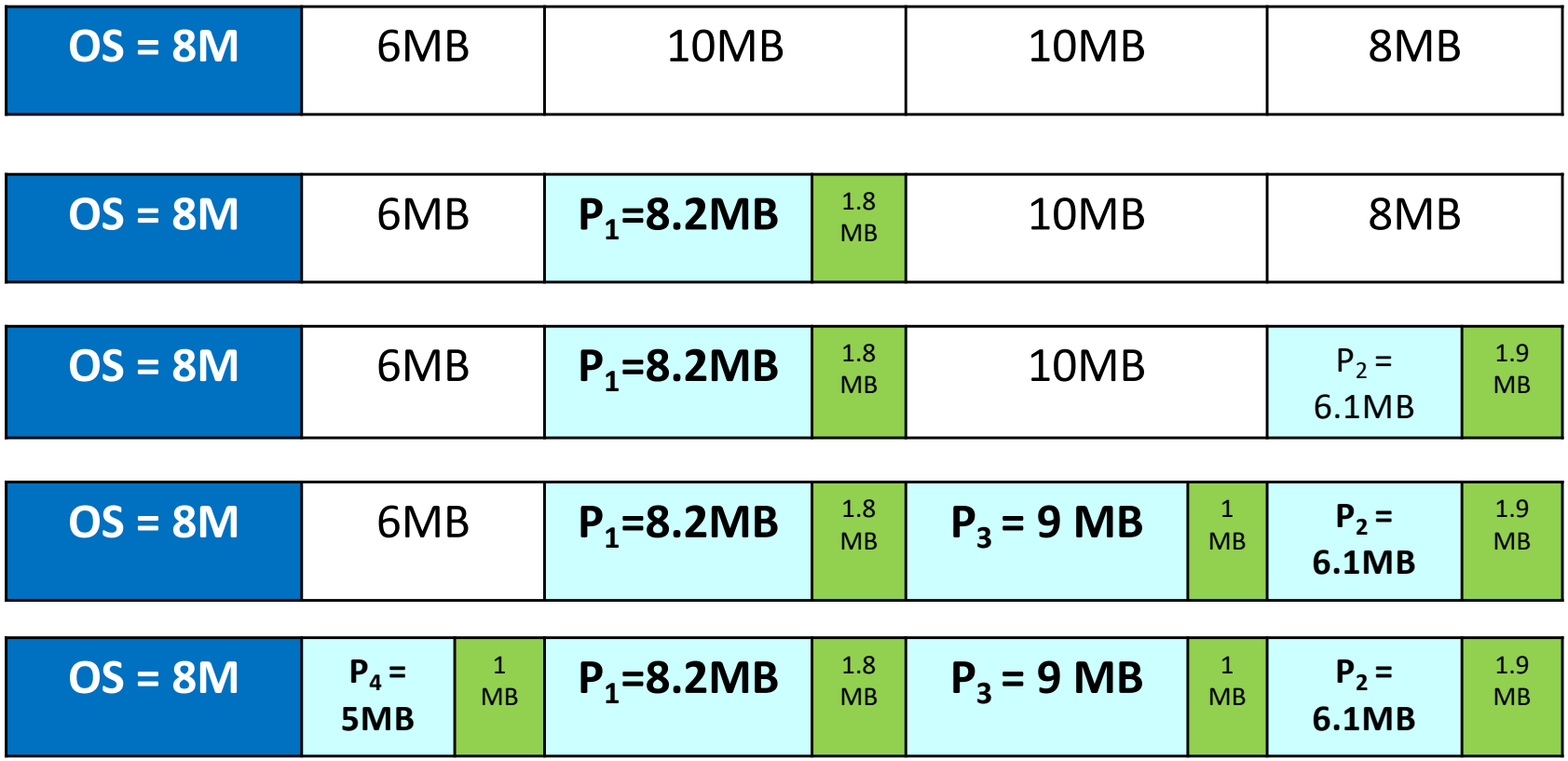
Hardware support for relocation and limit registers



Contiguous Allocation - Partitioning

- Contiguous memory allocation can be achieved in two ways:
  - Fixed partitioning
  - Variable or dynamic partitioning
- **Fixed partitioning:** Main memory is divided into a number of **static partitions** (possibly of **different sizes**) at system generation time.
- A process may be loaded into a partition of equal or greater size.
- Maximum number of active processes is fixed.
- Simple to implement
- Suffers from internal fragmentation
  - **Internal Fragmentation** – Fragmentation that is internal to a partition.
    - Allocated memory may be slightly larger than requested memory; this size difference in memory is internal to a partition, but not being used

Fixed partitioning example



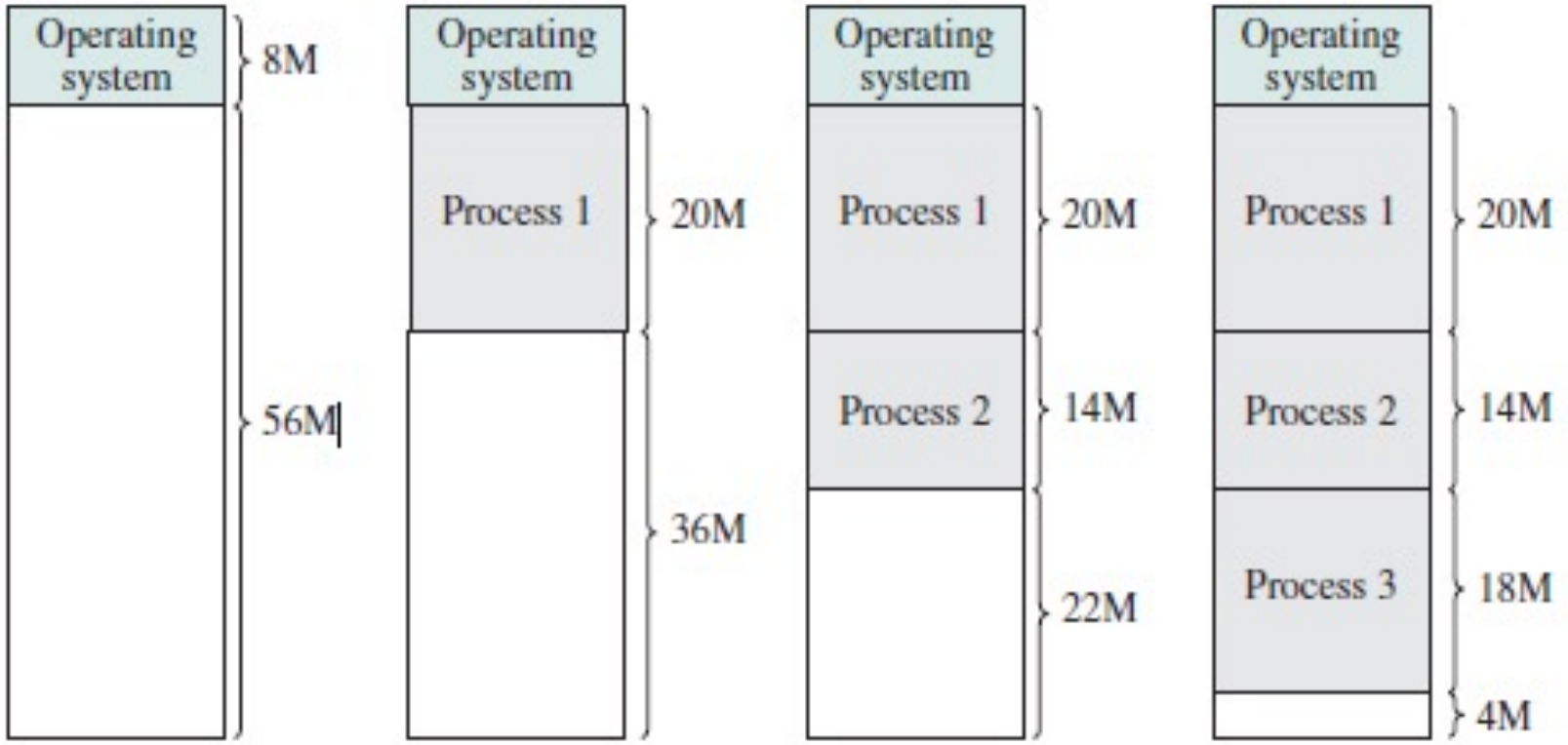
- Maximum number of possible active processes = ?
- Can process P_5 = 3MB be brought into main memory?

Contiguous Allocation – Variable Partitioning

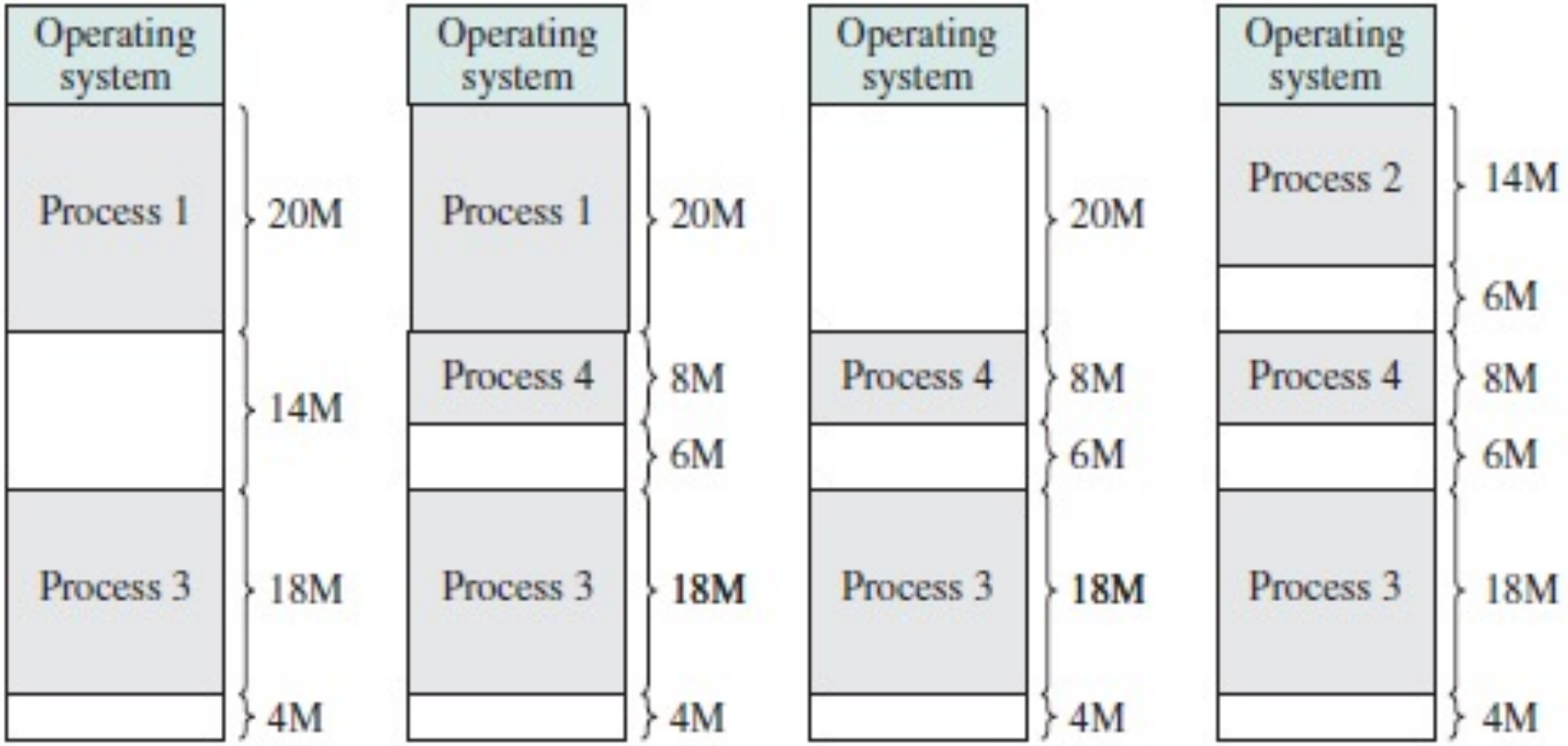
- **Variable or dynamic partitioning:**
  - The operating system keeps a table indicating which parts of memory are available and which are occupied.
  - Initially, all user memory is available for user processes as one large block of available memory (**hole**).
  - Eventually, as processes move in and out of memory, the main memory contains a set of holes of various sizes.
  - No internal fragmentation but suffers from external fragmentation.
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- External fragmentation problem can be solved by compaction.
- **Compaction** - Shuffle memory contents to place all free memory together in one large block.

Dynamic partitioning example

Memory = 64M



Dynamic partitioning example Cont…



- Can process P_5 = 10 MB be brought into main memory?
- Can process P_6 = 17MB be brought into main memory?

Dynamic Storage-Allocation Problem

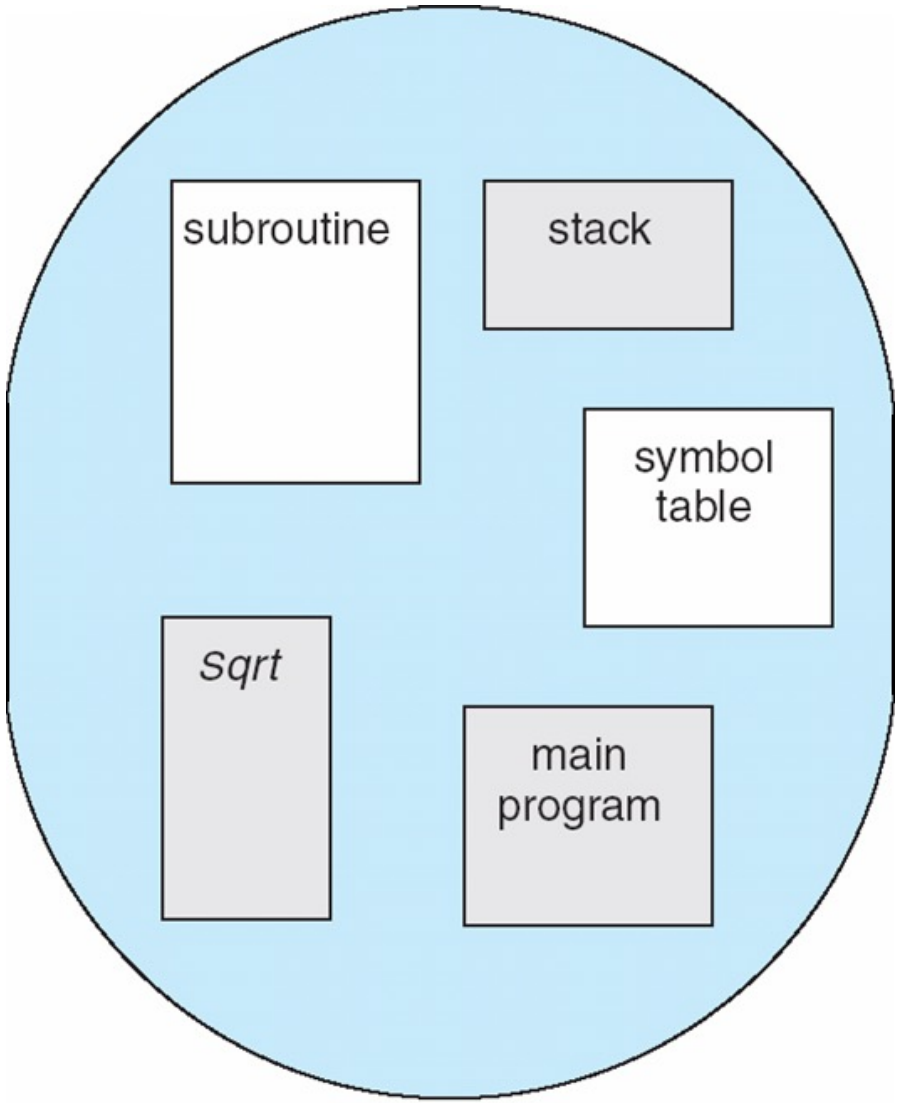
How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough
  - must also search entire list
- **Worst-fit**: Allocate the ***largest*** hole available
  - must also search entire list
- Simulations indicate that First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Segmentation

- **Segmentation** - partitions logical address space of a program into segments (related data units).
- For example, module, procedure, stack, data, file, etc.
- Each segment instead of the entire program is loaded into an available block of memory
- As a result, programs can reside in *non-contiguous* memory locations.
- Segmentation suffers from external fragmentation and the need for compaction.

Programmer's View of a Program



Segmentation Architecture

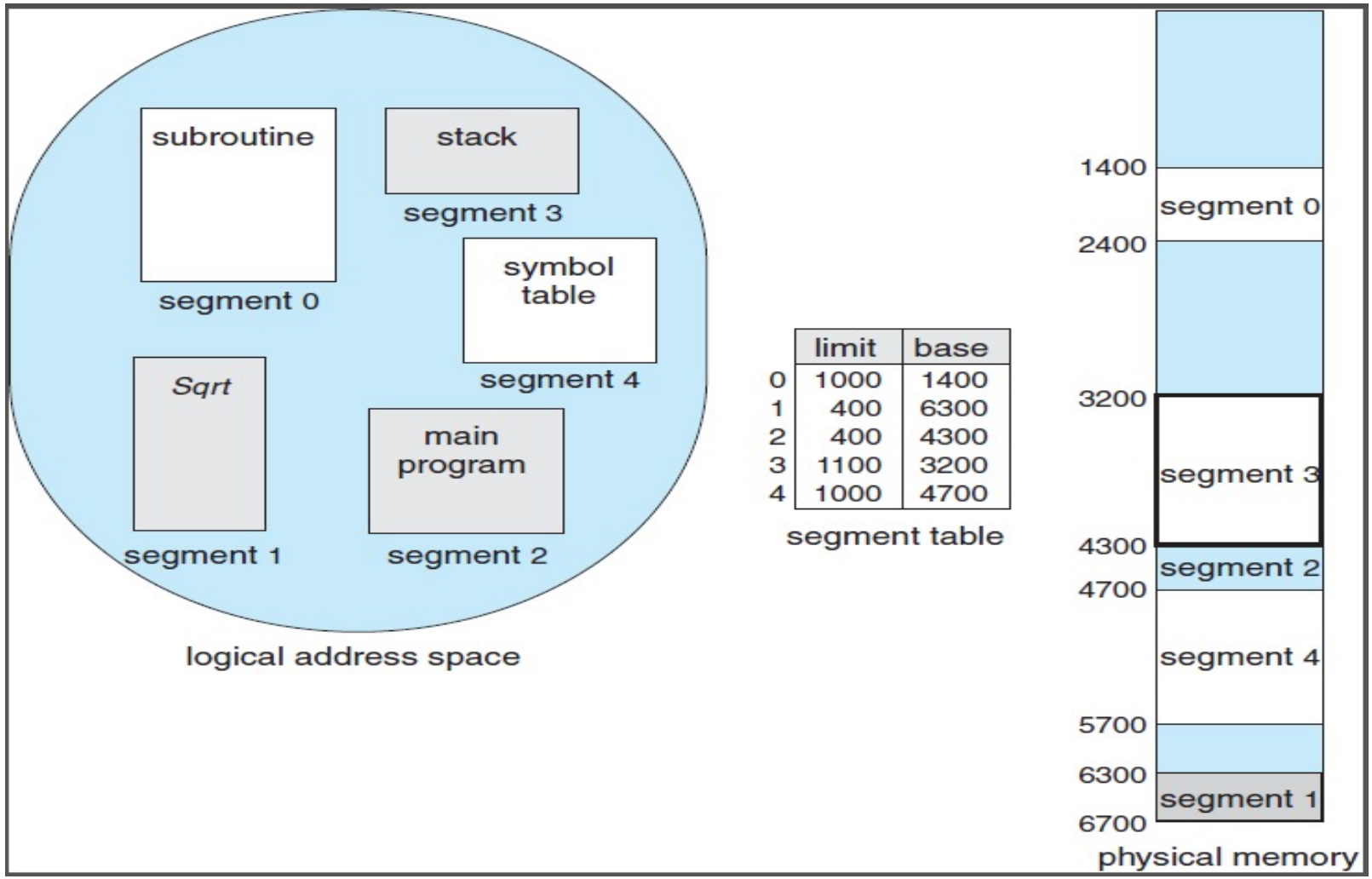
- Logical address generated by the CPU, consists of a two tuple:

  <segment-number, offset>,

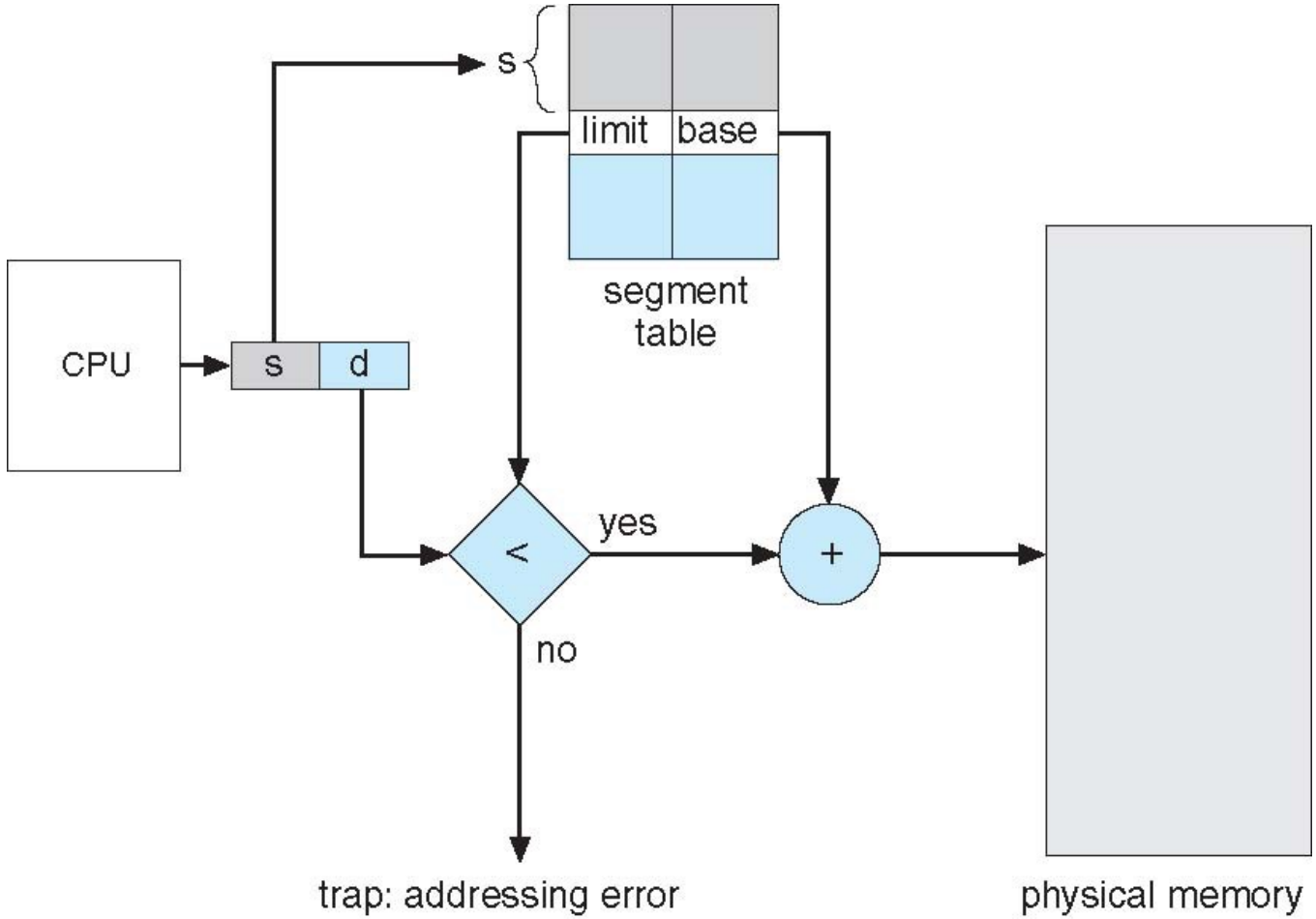
- We now must map two-dimensional user-defined addresses into one-dimensional physical memory addresses.
  - Achieved through hardware support.
- **Segment table** – Indexed by segment number and consists of base–limit register pairs for each segment.
  - **base** – contains the starting physical address where the segment resides in memory
  - **limit** – specifies the length of the segment
- Multiple base/limit pairs, one per segment (segment table)

Logical View of Segmentation

- CPU generates <2, 53>, that is CPU is referencing byte 53 of Segment 2.
- This is mapped on the location 4300 + 53 = 4353 of physical memory.



Segmentation Hardware



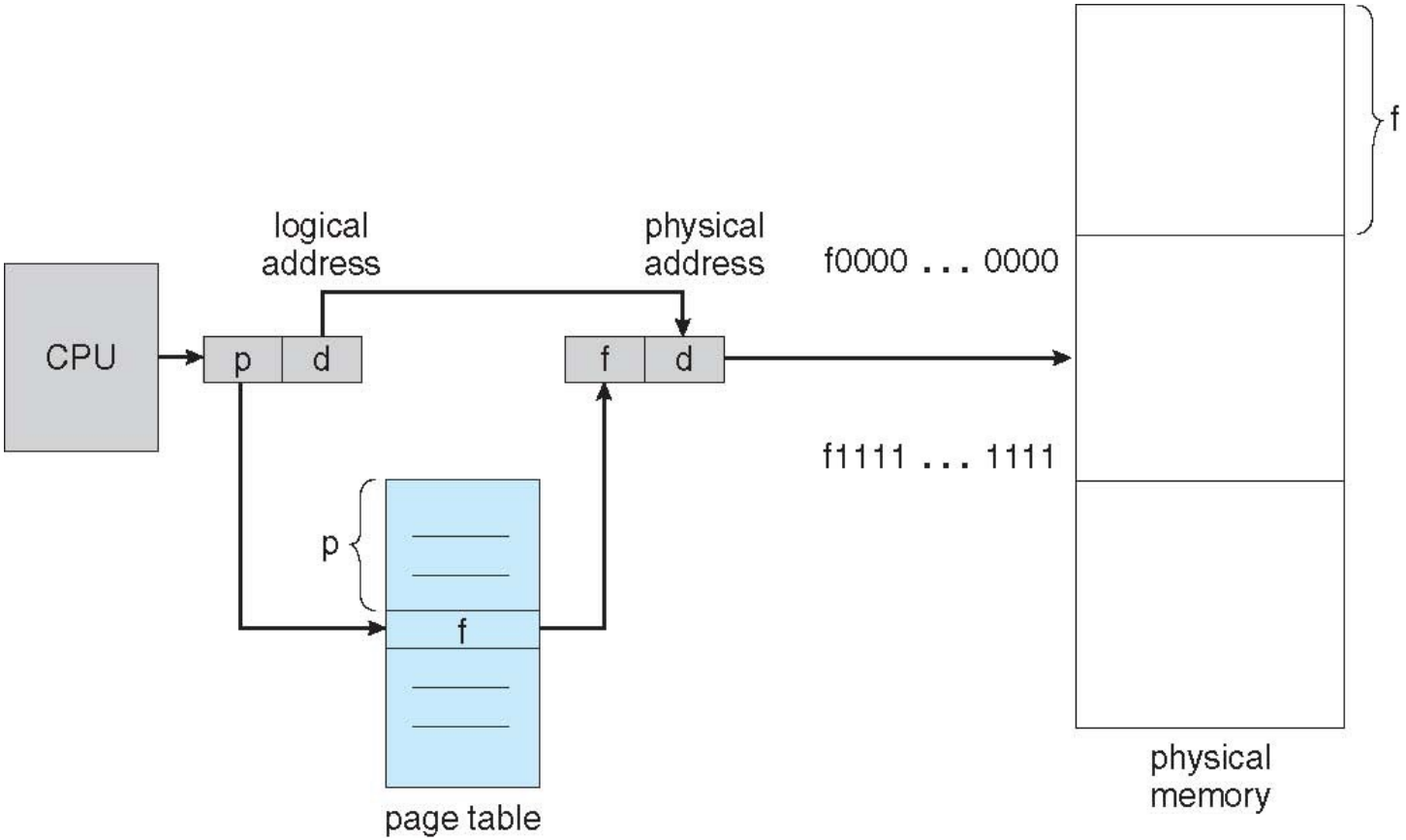
Paging

- **Paging** - divides logical address space of a program into fixed blocks called **pages** and physical address space into fixed-sized blocks called **frames**.
  - **Frame size = page size.**
- Paging is used in most OS ranging from PC, mainframes to smart phones.
- Avoids external fragmentation but suffers from internal fragmentation.

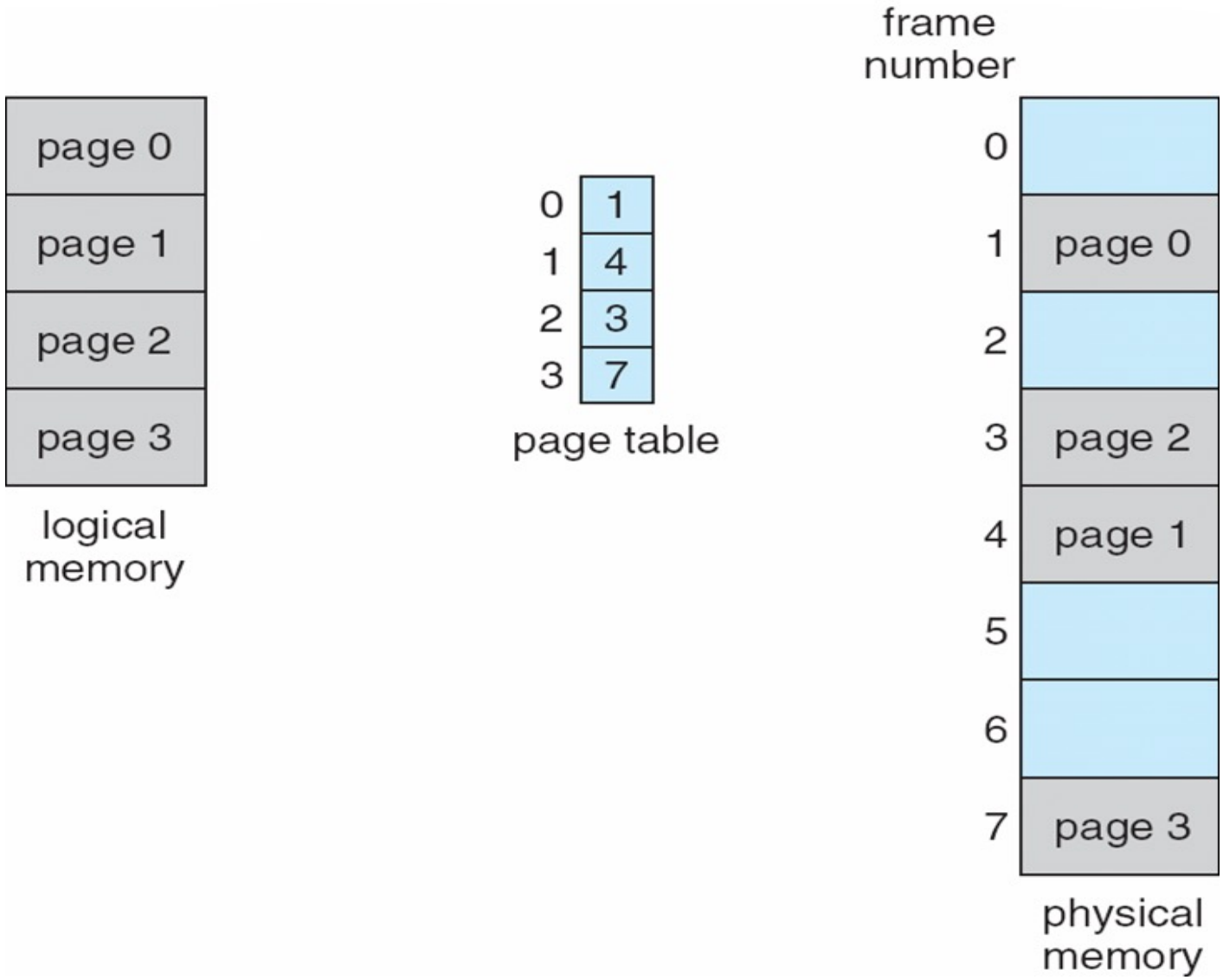
Address Translation Scheme

- Address generated by CPU (virtual address) consists of two parts:
  - **Page number** (***p***) – used as an index into a **page table** which contains the frame number of a page loaded in physical memory.
  - **Page offset** (***d***) – is the offset within the page.
- page size = frame size => page offset = frame offset.
- The frame number is combined with the page offset to obtain the physical memory address.
- The page offset ranges from 0 to page size -1
- The page size (like the frame size) is defined by the hardware.

Paging Hardware



Paging Model of Logical and Physical Memory



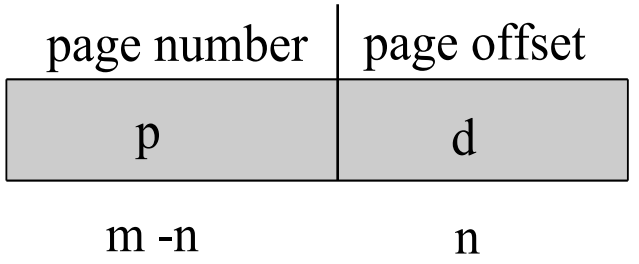
Address Translation Scheme

- Page Size is usually a power of 2 (512 bytes to 1GB), as it makes address translation particularly easy
- Consider a logical address space size of 2^m
  - What is the range of addresses in this logical space?
    - 0.. 2^m -1
  - How many bits do you need to represent an address in this space?
    - You need m bits to represent an address

Address Translation Scheme Cont…

For given logical address space size = 2^m, page size = 2^n and physical address space = 2^r

- Number of bits to represent logical address = m
- Number of bits to represent physical address = r
- Number of bits to represent offset = n
- Number of pages = 2^{m-n}
- Number of bits to represent a page number = m-n
- Number of frames = 2^{r-n}
- Number of bits to represent a frame number = r-n



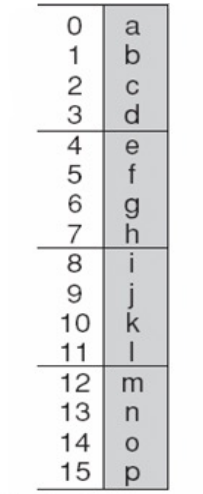
Paging and Address Translation Example

- *Logical address space = 2^4 = 16 bytes*
- Number of bits to represent logical address = 4
- *Page size = 2^2 = 4 bytes*
- Number of bits to represent offset = 2
- Number of bits to represent a page number = 2
- *Total # of pages = $2^{(4-2)} = 2^2$*
- *Memory/Physical address size = 2^5 = 32 bytes*
- Number of bits to represent physical address = 5
- *Total # of frames = $2^5 - 2^2 = 2^3$*
- Number of bits to represent a frame number = 3

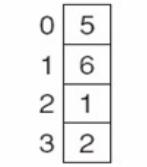
Logical addresses	P number	d (offset)
0	00	00
1	00	01
2	00	10
3	00	11
4	01	00
5	01	01
6	01	10
7	01	11
8	10	00
9	10	01
10	10	10
11	10	11
12	11	00
13	11	01
14	11	10
15	11	11

Paging and Address Translation Example

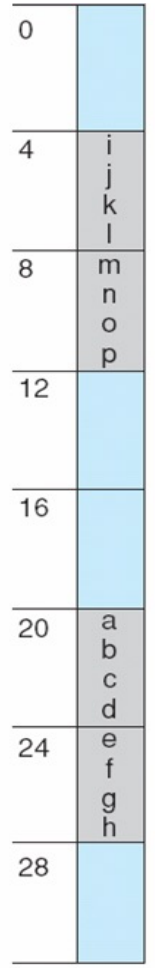
- *Logical address space = 2^4*
- *Page size = 2^2 = 4 bytes*
- *Memory size = 2^5 = 32 bytes*
- *Total # of pages = $2^{(4-2)} = 2^2$*
- *Total # of frames = $2^5 - 2^2 = 2^3$*
- *Logical address 3 (d) is in page 0,*
- *and page 0 resides in the 5^{th} frame*
- *Logical address 3 maps to physical address = (5*4 + 3) =20 + 3 = 23.*



logical memory



page table

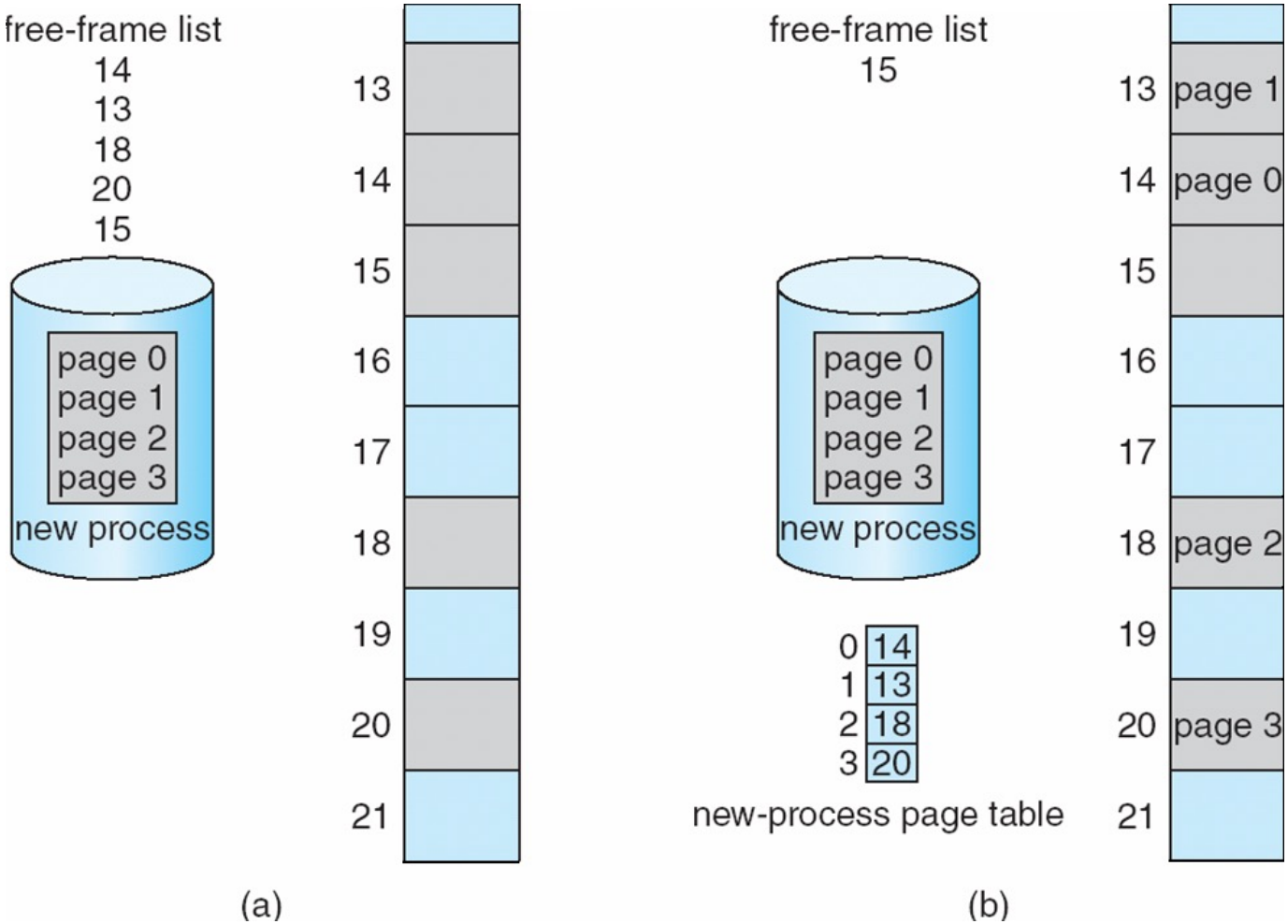


physical memory

Frames in Memory

- Address translation is hidden from users, and
- The operating system manages physical memory.
  - Needs to know which frames are allocated, which frames are available, how many total frames there are, etc.
  - To answer the above questions OS maintains a data structure called a **frame table**.
  - The frame table has one entry per frame.

Allocating Frames



Before allocation

After allocation

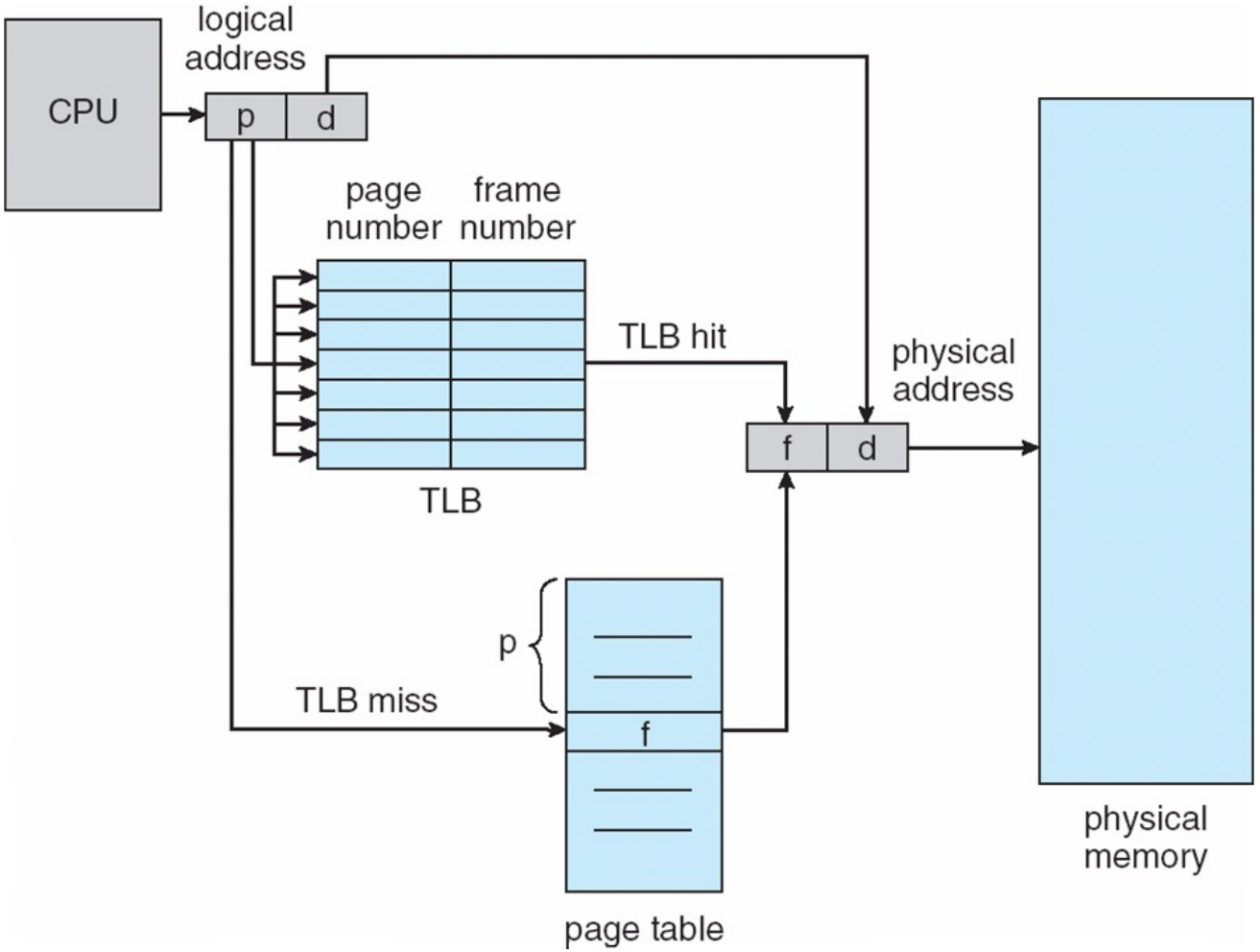
Implementation of Page Table

- For small size page tables – provide dedicated registers.
- For large page tables – **page table kept in main memory**
  - **Page-table base register** (**PTBR**) points to the starting address of the page table in main memory.
  - In this scheme every data/instruction access requires two memory accesses
    - One for the page table and one for the data / instruction
  - To mitigate the two memory access problem current machines, use a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**)
    - TLB is associative, high-speed memory.
    - TLBs typically are small (64 to 1,024 entries)

Translation Look-aside Buffer (TLB)

- When a logical address is generated by the CPU, its page number is presented as a data item to TLB.
- This data is checked with all the entries in TLB **simultaneously**.
- If page is found (called a **TLB Hit**) its corresponding frame# is returned.
- If page is not found in TLB (called a **TLB Miss**) page table is searched for the page and its corresponding frame# is retuned and entry for the page added in TLB.
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch

Paging Hardware With TLB



TLB - Effective Access Time

- **Hit ratio (α):** Hit ratio – percentage of times a page number of interest is found in TLB.
- Memory access time = ***m***
- ***TLB access time = t***
- EAT = (***m + t***) α + (2***m + t***)(1 – α)
- **Effective Access Time** (**EAT**):

$$EAT = (m)\ \alpha + (2m)(1 - \alpha)$$

We ignore the time taken to search TLB as we assume that the data to search is checked with all the entries in TLB simultaneously.

TLB - Effective Access Time Example

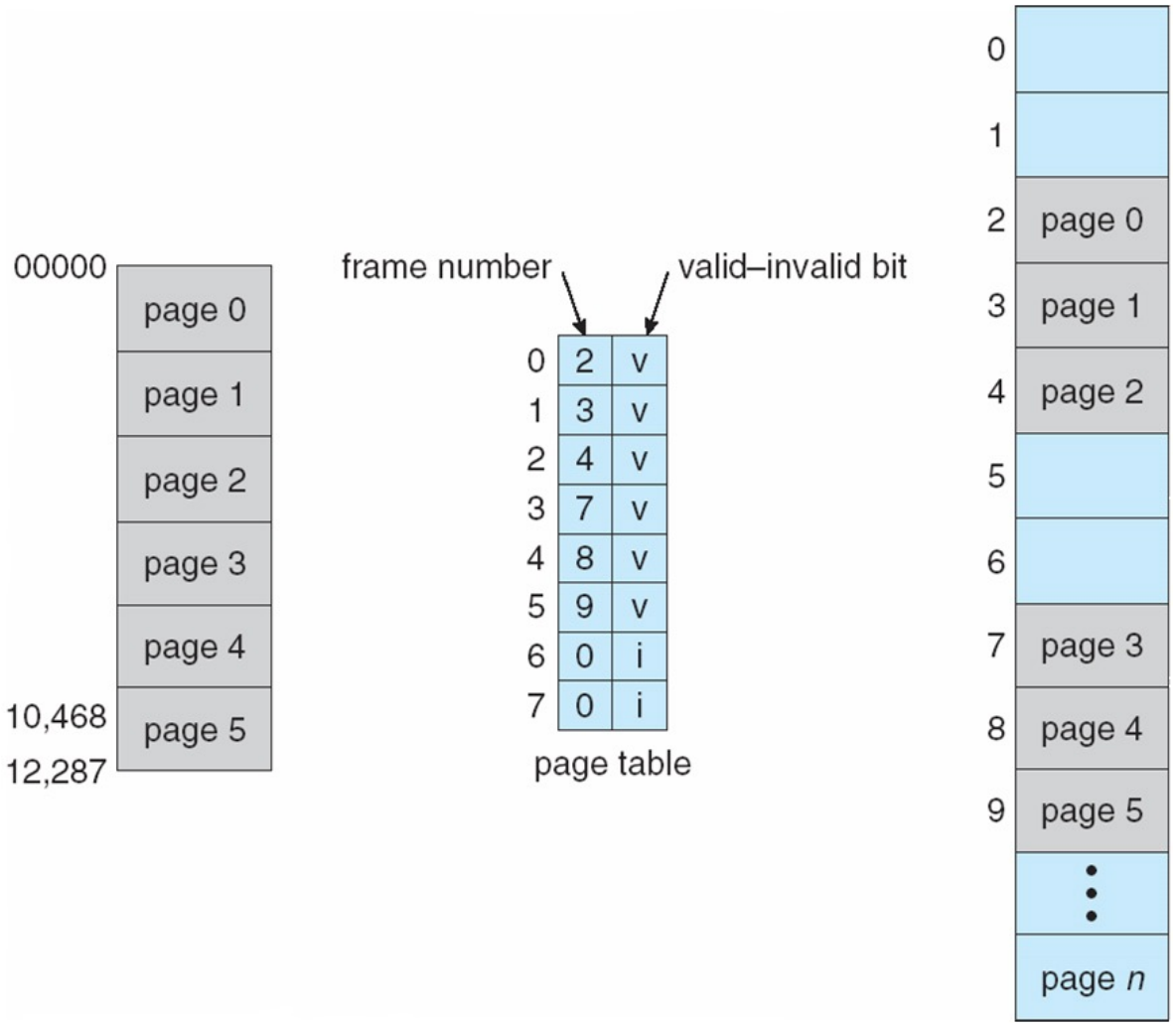
- **Suppose memory access time (*m*)** = 100ns
- If we fail to find the page in TLB, then EAT = 200ns.

  (first access for the frame# in page table, and the second to access the desired byte in memory)
- If hit-ratio (α) = 80%
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns!

Memory Protection

- In paging a program cannot access memory that is not its own.
- A bit or bits can be added to the page table to classify the access type of the page.
  - Eg. read-write, read-only, read-write-execute etc.
  - Each memory reference is checked to ensure memory is accessed in the appropriate mode.
- **Valid / invalid bits** can be added to "mask off" entries in the page table that are not in use by the current process.
  - “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

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



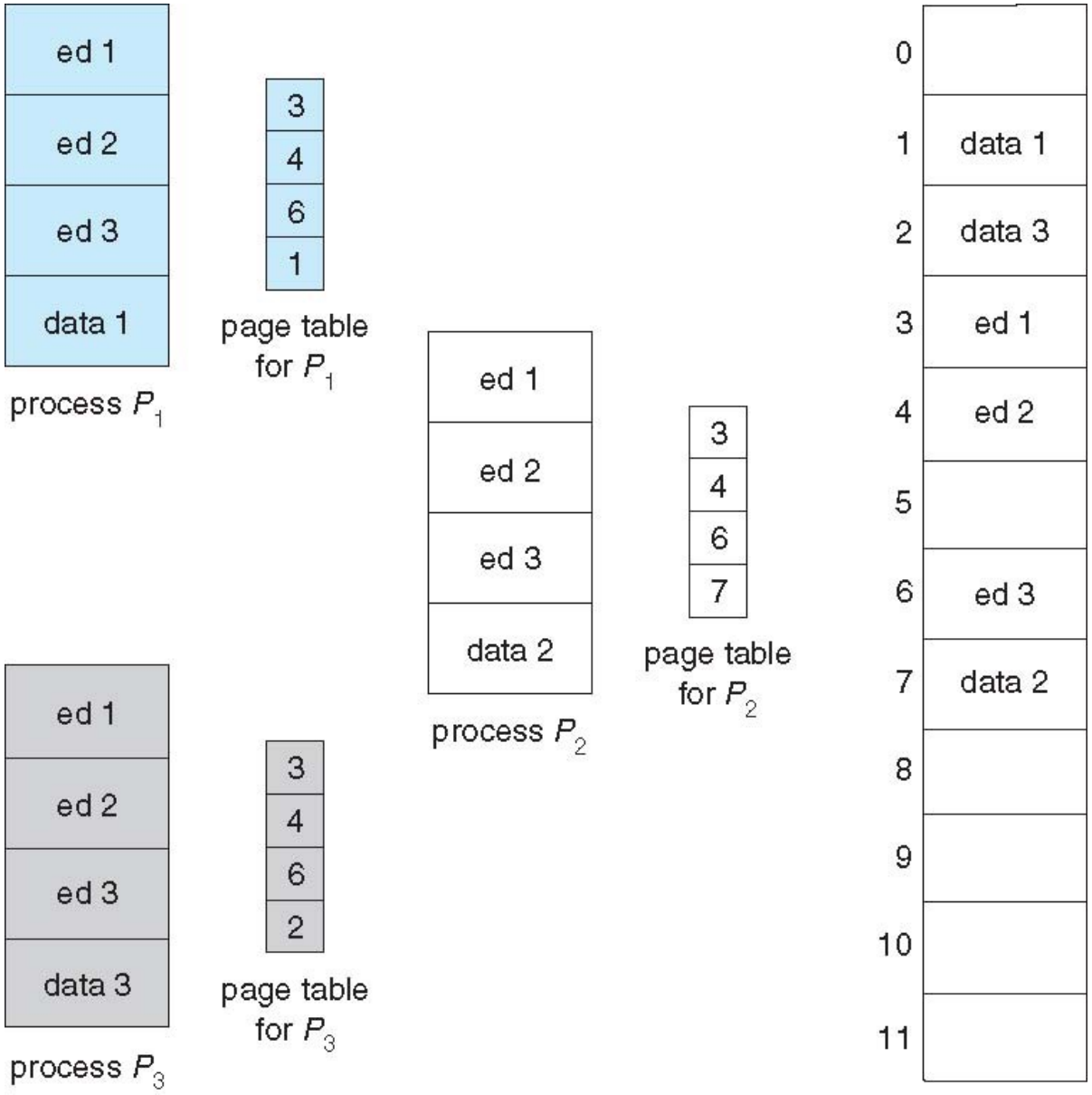
Memory Protection Cont…

- Most processes use only a small fraction of the Logical address range available to them.
- It is wasteful in these cases to create a page table with entries for every page in the address range.
- Some systems provide the **page-table length register** (**PTLR**) – which stores the length of page table.
  - Any access greater than this length => trap to the kernel

Shared Pages

- Paging systems can make it very easy to **share blocks of memory.**
- **Reentrant code** is non-self-modifying code: it never changes during execution.
- Sharing is possible as follows:
  - Have just one copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers etc.)
  - Maintain multiple copies of private data; that is, each process keeps a separate copy of the code and data
- Some operating systems implement shared memory using shared pages.

Example Shared Pages



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, and page size of 4 KB (2^{12})

Page number | Page offset

20 bits	12 bits

  - No of entries in a single level page table = 2^{20} entries
  - If each page entry is 4 bytes:
    - size of page table = $4*2^{20}$ = 4 MB!
    - Don't want to allocate that contiguously in main memory

Structure of the Page Table Cont…

- Common techniques for structuring a page table:
  - Hierarchical paging
  - Hashed page table (used for logical address spaces > 2^{32})
  - Inverted page table

Multi- Level/Hierarchical Paging

- Technique in which the page table is paged; that is, broken down into multiple page tables of the same size as the page.
- From the example previously shown the size of a single level page table = $4*2^{20}$ bytes.
- Number of pages of the page table =
- Size of the single level page table/size of a page
- $= 4*2^{20}/2^{12} = 2^{10}$
- To track these pages of the page table we need a **second level page table** (or outer page table)!
- Size of this outer page table
  - = Number of pages of the page table * page entry size
  - = $2^{10}(2^2) = 2^{12}$ bytes = size of one page.

Multi- Level/Hierarchical Paging Cont…

- Also, note that a single page of a page table contains

  = size of the page/size of a page entry

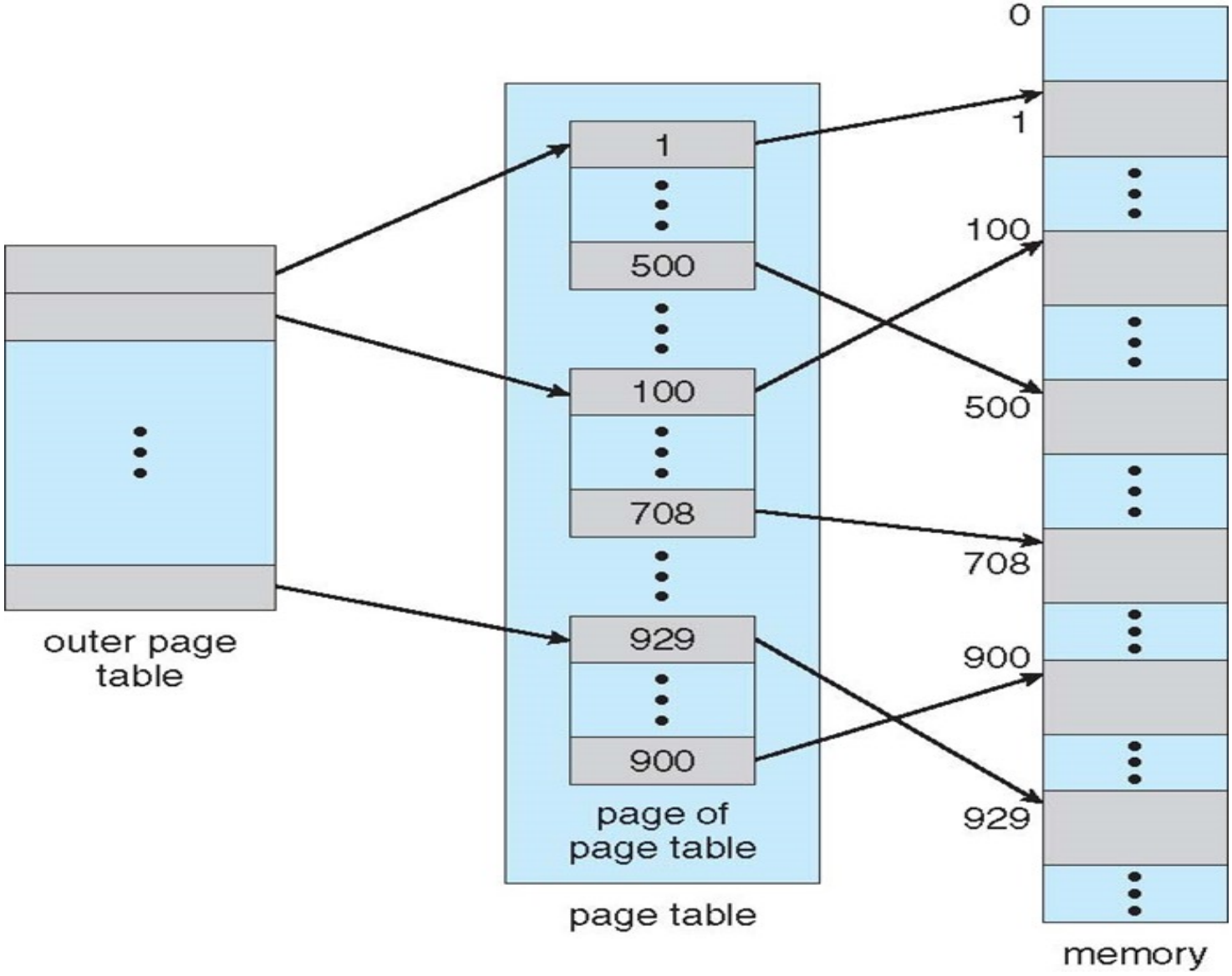
  = $2^{12}/4 = 2^{10}$ entries.

- Since there are two levels of page tables, this above technique is called **two-level paging**.

- With varying sizes of the logical address space and pages, we could have a very large outer page table.

  - In this case, we repeat this process by paging the outer page table (thus introducing another layer of page table).

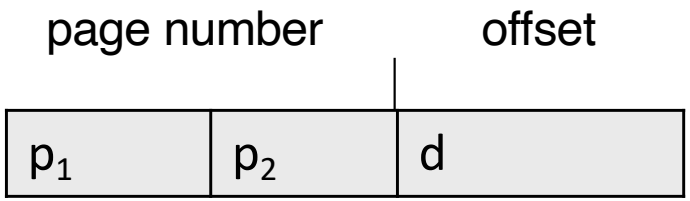
Two-Level Page-Table Scheme



Address Translation in two level paging scheme

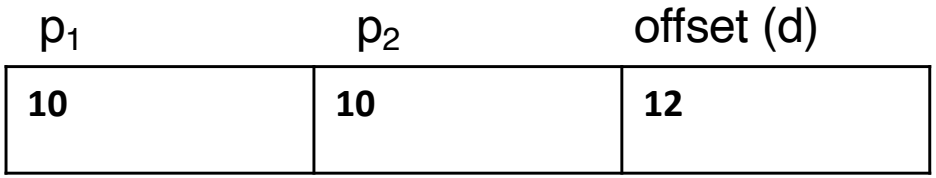
- Since the page table is paged, the page number is further divided into:
  - p_1 is an index into the inner page table, and
  - p_2 is the displacement within the page of the inner page table

- Thus, a logical address is as follows:

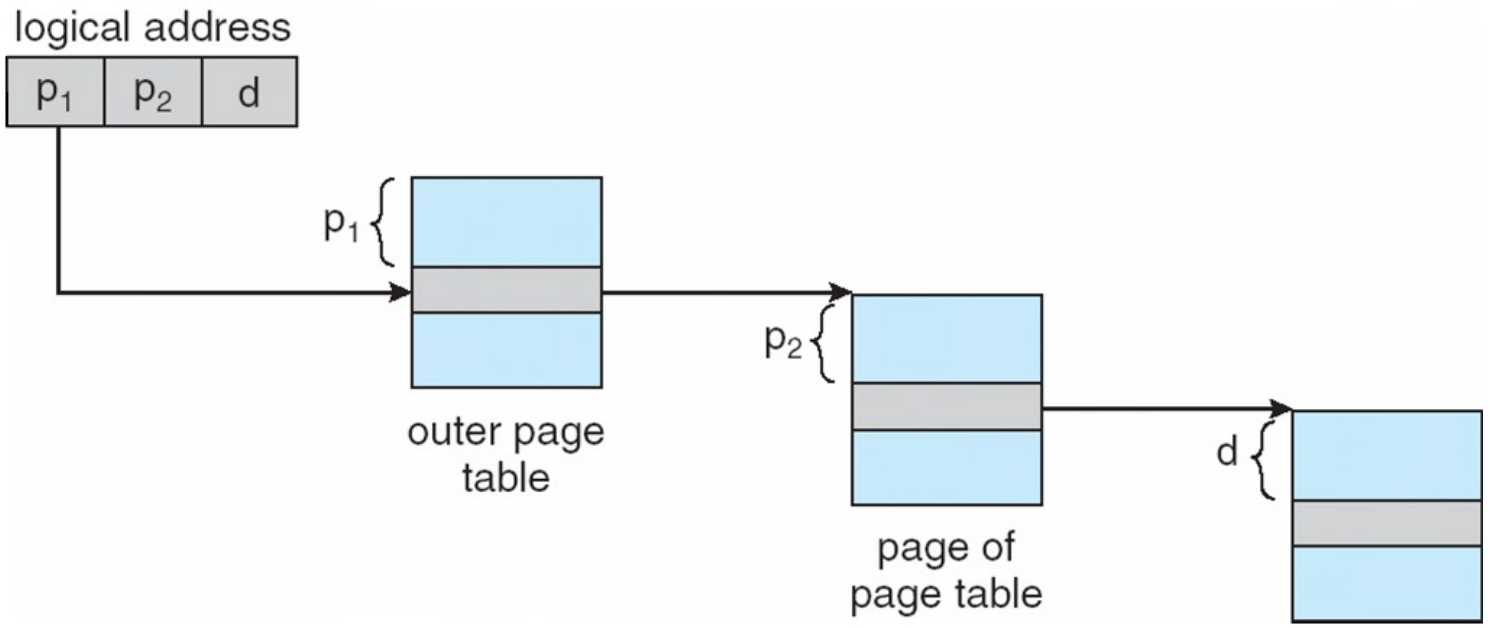


On a 32-bit machine with 4K page size

The distribution of no. of bits to represent each level of the multi-level page table, and the page offset is as follows:



Address Translation in two level paging scheme



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all **physical pages (that is frames)**
- Inverted page table has one entry for each **frame** 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.

- **Example:** Consider a computer system with a 32-bit logical address, 4-KB page size, and 24-bit physical address.
- A conventional, single-level page table contains 2^{20} entries.
- An inverted page table contains **2^{12} entries**!