Operating Systems

Dr Ali Sayyed

Department of Computer Science
National University of Computer & Emerging Sciences

Operating Systems

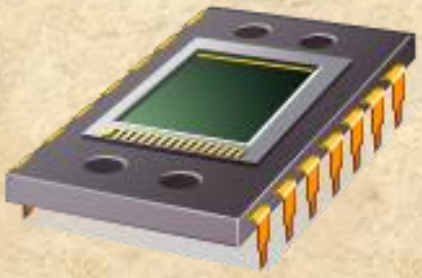
Lecture 10
Memory Management

Memory Management

- In a uniprogramming system, main memory is divided into two parts:
  - For OS
  - For program currently being executed.
- In a multiprogramming system, the "user" part is further subdivided to accommodate multiple processes.
- The task of subdivision is carried out dynamically by the operating system and is known as **memory management**
- Memory needs to be **efficiently allocated** to ensure a reasonable supply of ready processes to consume available processor time.

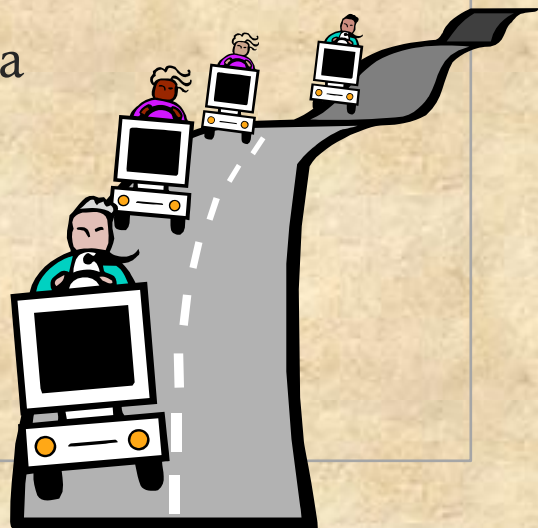
Memory Management Requirements

- Memory management is intended to satisfy the following requirements (fundamental design goals for any memory management system)
  - Relocation
  - Protection
  - Sharing
  - Logical organization
  - Physical organization



Relocation

- **Programmers** typically don't know in advance which other programs will remain in main memory at the time of execution of their program
- Over time, active processes are **swapped out** to the disk and later **swapped back** into RAM in order to maximize processor utilization
- When a program is swapped back in, **it may not (and often cannot) be placed in the exact same physical memory location** it occupied before.
- The system must be able to "relocate" the program to a different area of memory without crashing it.



- OS need to know the **location of process control information** and the entry point to begin execution of this process.
- Since the OS is managing memory and has brought this process into RAM, these addresses are easy to come by.
- However, the **OS must deal with memory references** within the program.
- **Branch instructions** contain an address to the instruction to be executed next.
- **Data reference** contain the address of data to be fetched.
- Somehow, the hardware and OS **must be able to translate these memory references into actual physical memory addresses**

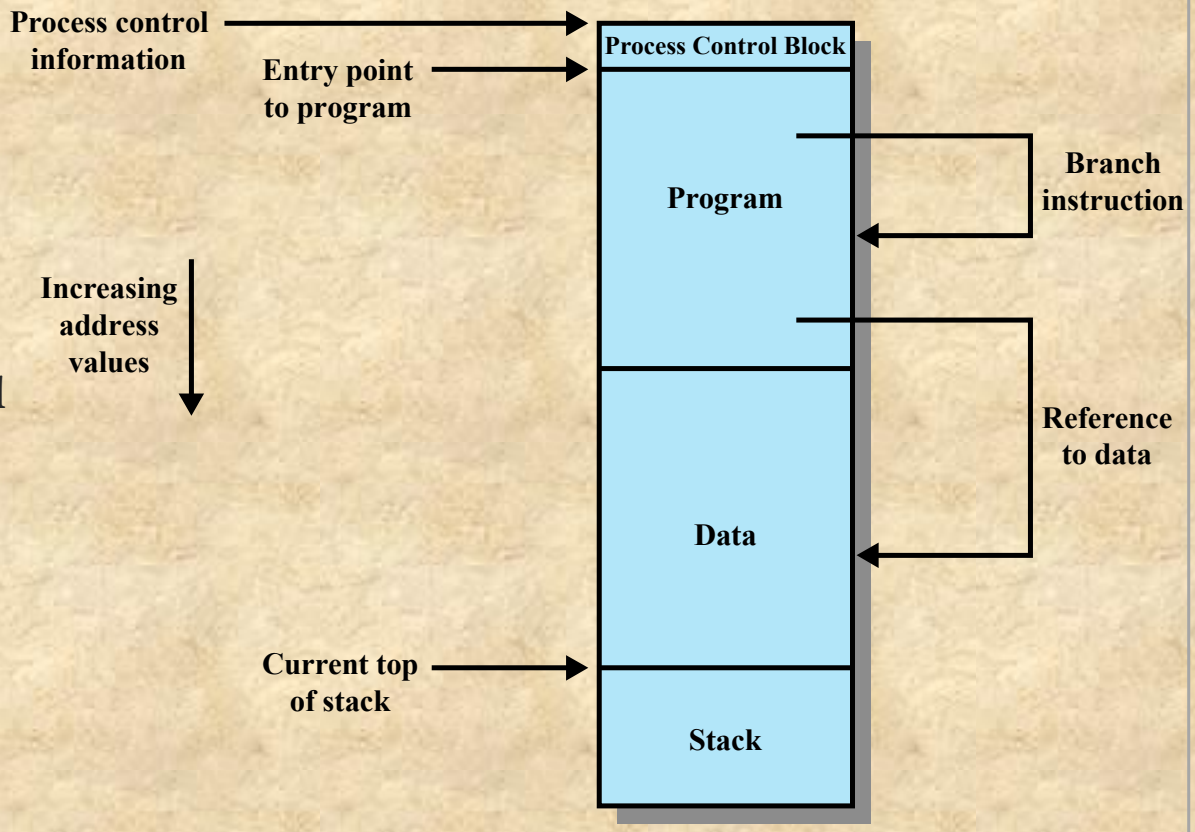


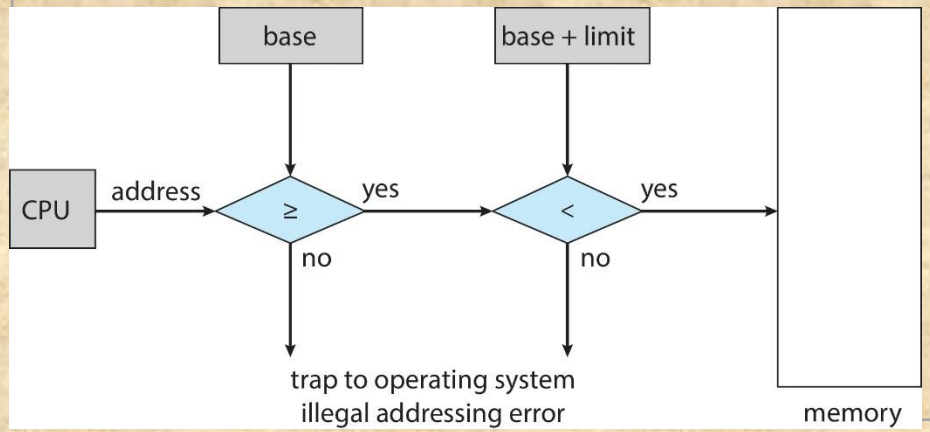
Figure 7.1 Addressing Requirements for a Process

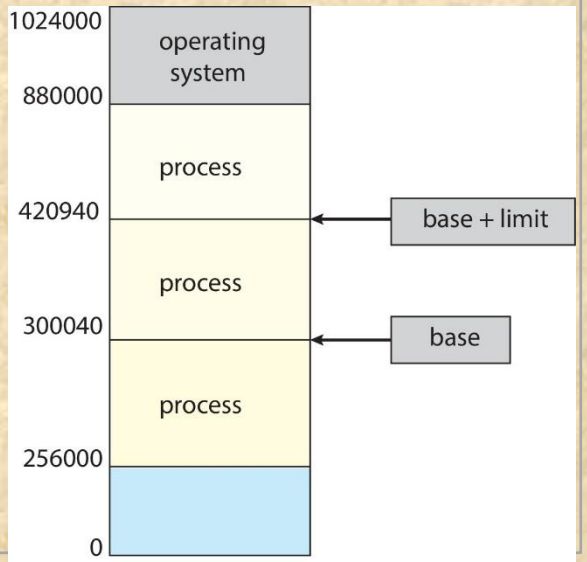
Protection

- Processes must **not be able to interfere** with each other.
- **Every memory reference** generated by a process **must be checked** at **runtime** to ensure it falls within that process's allocated address space.
- A user process **cannot access** any portion of the **operating system**, neither program nor data.
- Without special arrangement, a **program in one process cannot access the data area of another process**. The processor must be able to abort such instructions at the point of execution.
- **Memory references** generated by a process **must be checked at run time**

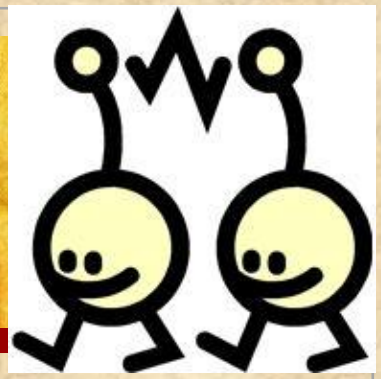
Protection

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user





Sharing



- While protection is about isolation, sharing is about controlled cooperation.
- Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory.
- For example, if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy.
- Memory management must allow controlled access to shared areas of memory without compromising protection
- Mechanisms used to support relocation support sharing capabilities

Logical Organization

- This requirement relates to how the programmer or the user views memory, which is different from how the hardware views it.
- Memory is organized as linear array of bytes (0 to max)

Programs are written in modules

- programmers see memory as a collection of modules (functions, stacks, data arrays, and libraries) written and compiled independently
- different degrees of protection given to modules (read-only, execute-only)
- Memory management system should support this modular view to make programming, linking, and protection easier.

- Segmentation is the tool that most readily satisfies this requirements

Physical Organization

- Hierarchy of the **system memory** is organized into at **least two levels**.
- RAM is fast, expensive, and volatile. Secondary storage (Disk/SSD) is slow, cheap, and permanent.
- The flow of data between these two levels must be managed efficiently.
- The programmer should not have to worry about running out of RAM or manually moving data to the disk.
- So, the task of moving data between the two levels should be a system responsibility. This task is the essence of memory management.

Cannot leave the programmer with the responsibility to manage memory

Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

Addresses

Logical

- reference to a memory location independent of the current assignment (physical address) of data to memory

Relative

- address is expressed as a location relative to some known point

Physical or Absolute

- actual location in main memory

Binding of Instructions and Data to Memory

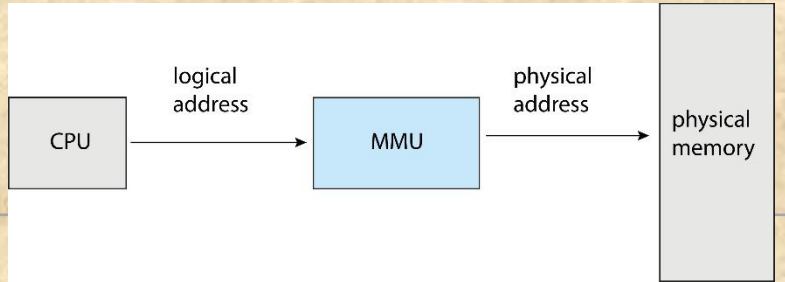
- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile Time Binding**: This is the most rigid form of address binding.
  - If you know exactly where the process will reside in memory before you even run it, the compiler can generate **Absolute Code**
  - It is inflexible. If you want to run this program on a different computer, or if starting location changes you must change the source code and recompile the entire program.
  - **Use Case**: Very simple embedded systems or old operating systems (like MS-DOS programs) where only one program ran at a time.

Binding of Instructions and Data to Memory

- **Load Time Binding**: This offers slightly more flexibility
- At compile time, the specific memory location is unknown. So, the compiler generates **Relocatable Code**. The code uses relative addresses (e.g., "start at 0 ").
- When a program is started, the OS Loader finds a free block of RAM (say, starting at address 5000). It then adds 5000 to every address in the code as it loads it into memory.
- However, once the program is loaded, it cannot move. If the program is paused and swapped out to the disk, it must be swapped back into the exact same memory location (5000) later.

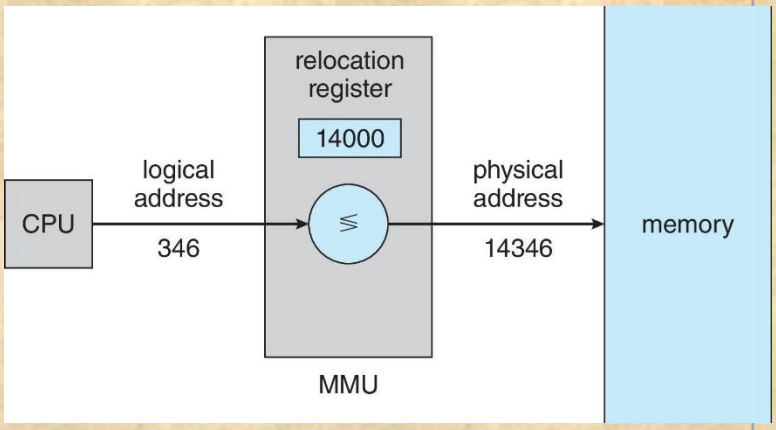
Binding of Instructions and Data to Memory

- **Execution Time Binding (Run Time)**: It provides maximum flexibility
- The program is loaded into memory, but the addresses remains logical (virtual). The translation to a physical address happens dynamically for every single instruction while the program is running.
- The process can be moved (relocated) in memory and the program won't crash because the address is calculated on the fly.
- Because doing this calculation via software for every instruction would be incredibly slow, special hardware is required:
  - **Memory Management Unit:** Hardware device that at run time maps virtual to physical address
  - **Base Register**
  - **Limit Register**



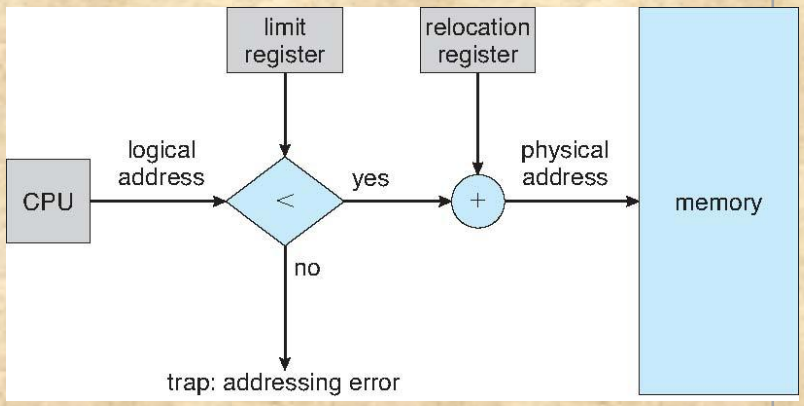
Memory-Management Unit

- A simple implementation of the Execution Time Binding. The **base register** now called **relocation register**
- 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 mapping from logical to physical happens via simple addition.
- 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



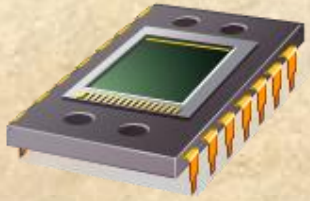
Memory-Management Unit

- A simple implementation of the Execution Time Binding. The **base register** now called **relocation register**
- 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 mapping from logical to physical happens via simple addition.
- 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 Partitioning

- Memory Partitioning is the method by which the Operating System divides the main memory (RAM) into distinct sections to handle multiple processes simultaneously (Multiprogramming).
- There are two primary techniques for doing this, each with its own trade-offs regarding how they waste memory (fragmentation).



- **Fixed (Static) Partitioning**
- This is the oldest and simplest technique. The RAM is divided into fixed-sized partitions *before* any process is executed.
- If you have 100MB of RAM, the OS might divide it into four fixed blocks of 25MB each.
- One process = One partition.
- With **equal-size partitions**, as long as there is a partition available, a process can be loaded into it. Because all partitions are of equal size, it does not matter which partition is used.
- With **unequal-size partitions**, the simplest way is to assign each process to the smallest partition within which it will fit.

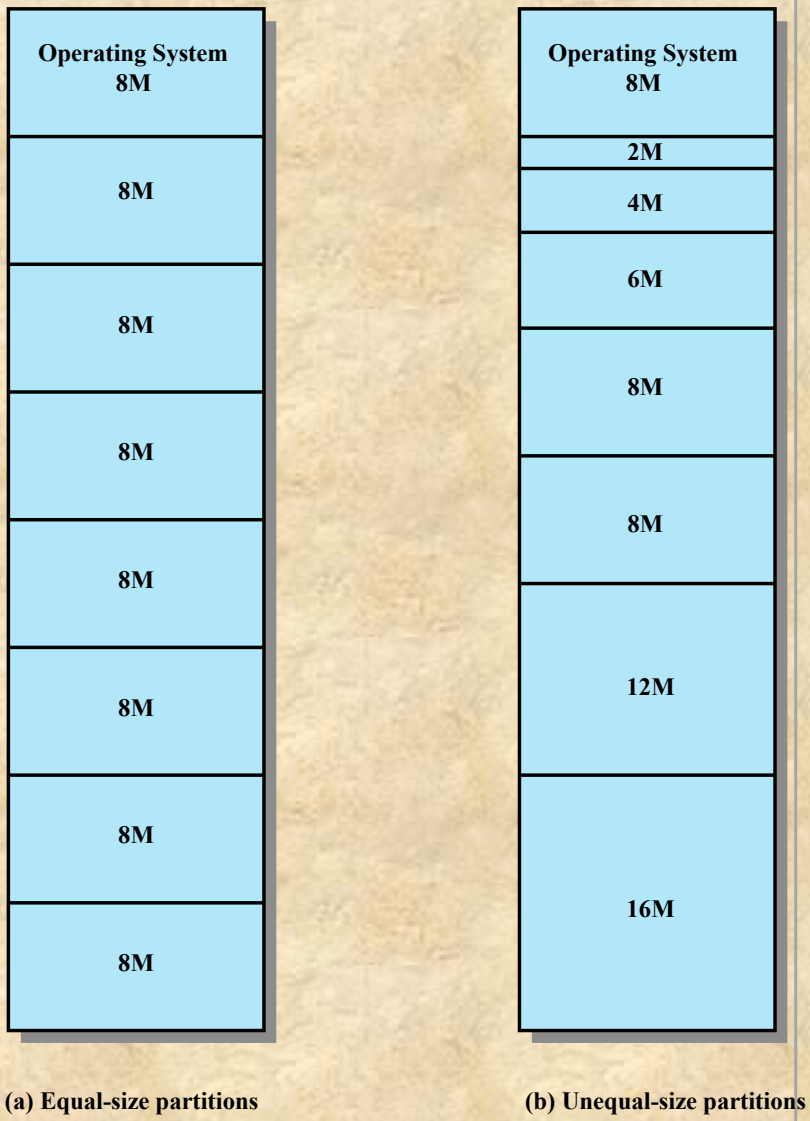


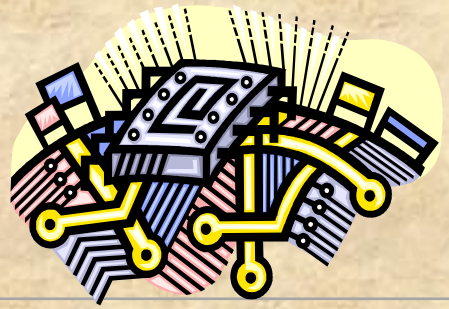
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Disadvantages

- A program may be too big to fit in a partition
  - program needs to be designed with the use of modules
- The number of partitions limits the number of active processes in the system
- Main memory utilization is inefficient
  - any program, regardless of size, occupies an entire partition
  - **Internal Fragmentation**
    - wasted space due to the block of data loaded being smaller than the partition
    - If a process of size 2MB is put into a partition of size 6MB, the remaining 4MB inside that partition is wasted. It cannot be used by anyone else.

Dynamic Partitioning

- Memory is not divided ahead of time. When a process arrives, the OS allocates exactly the amount of memory it requests.
- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM's mainframe operating system, OS/MVT



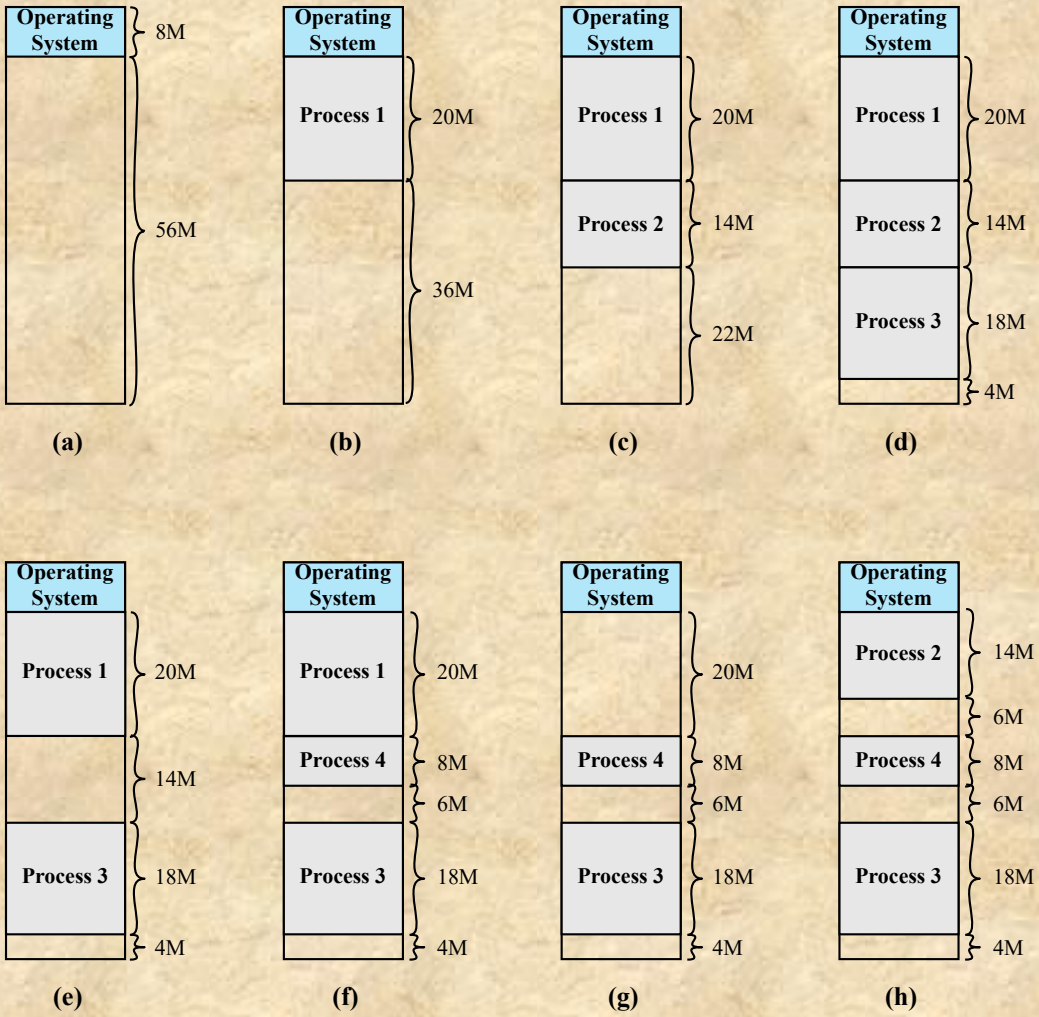


Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning

External Fragmentation

- memory becomes more and more fragmented
- memory utilization declines

Compaction

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time

Placement Algorithms

Because memory compaction is time consuming, the OS designer must be clever in deciding how to assign processes to memory

Best-fit	First-fit	Next-fit
• chooses the block that is closest in size to the request	• begins to scan memory from the beginning and chooses the first available block that is large enough	• begins to scan memory from the location of the last placement and chooses the next available block that is large enough

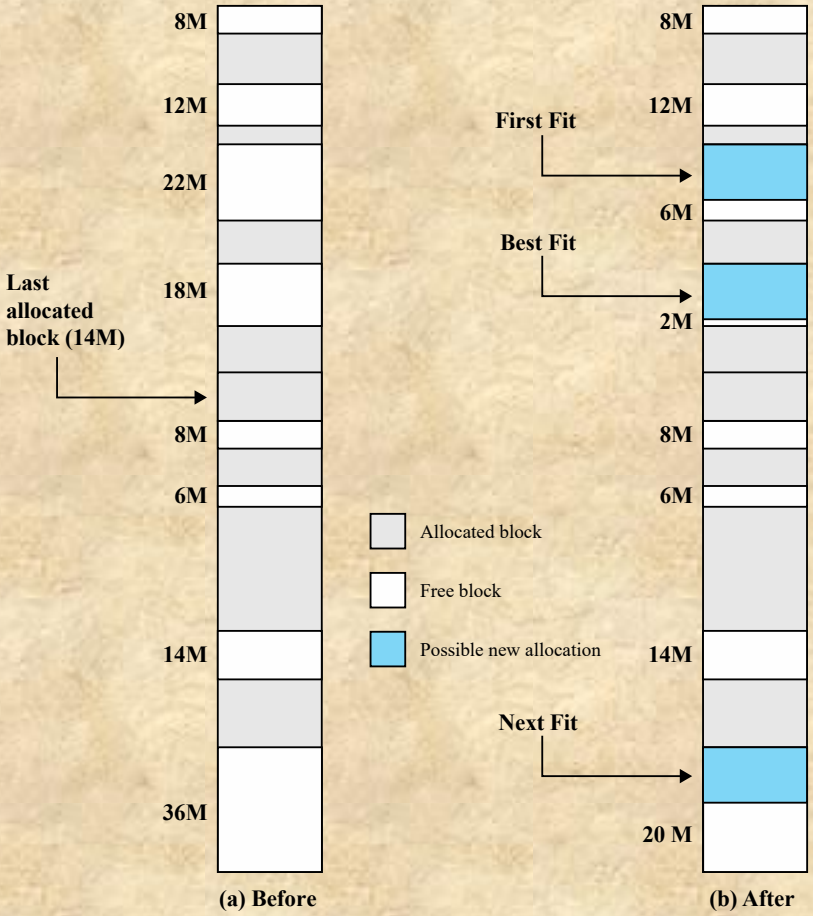


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

- The **Buddy System** is a memory allocation algorithm that acts as a compromise between Fixed Partitioning and Variable Partitioning.
- Memory is always allocated in blocks that are powers of 2 (e.g., 4KB, 8KB, 16KB, 32KB…)
- Memory blocks are available of size 2^K *words,* $L \leq K \leq U$, where
  - 2^L = smallest size block that is allocated
  - 2^U = largest size block that is allocated
  - generally 2^U is the size of the entire memory available for allocation
  - Request: 3KB → Allocated: 4KB block.
  - Request: 14KB →Allocated: 16KB block.

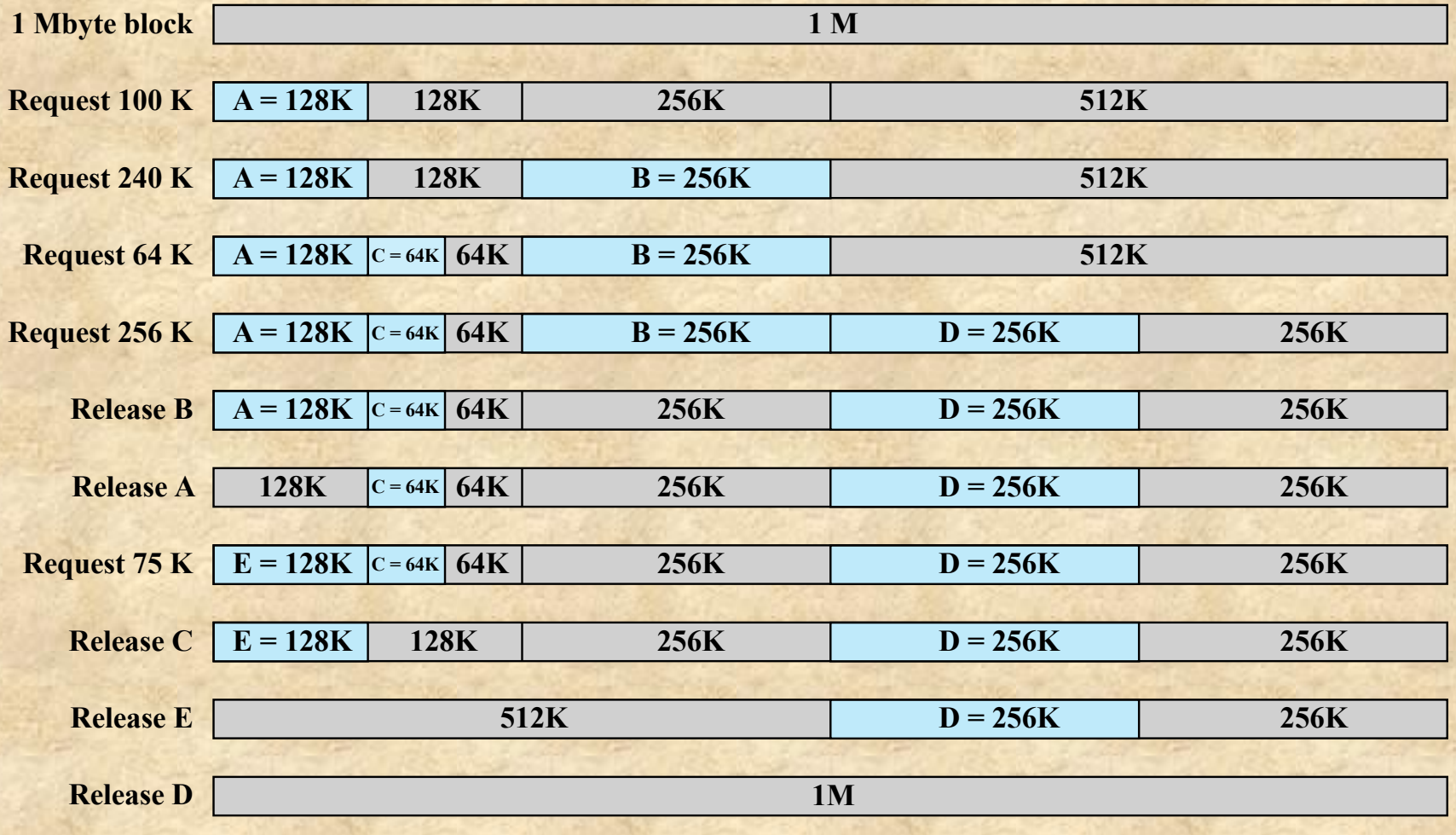
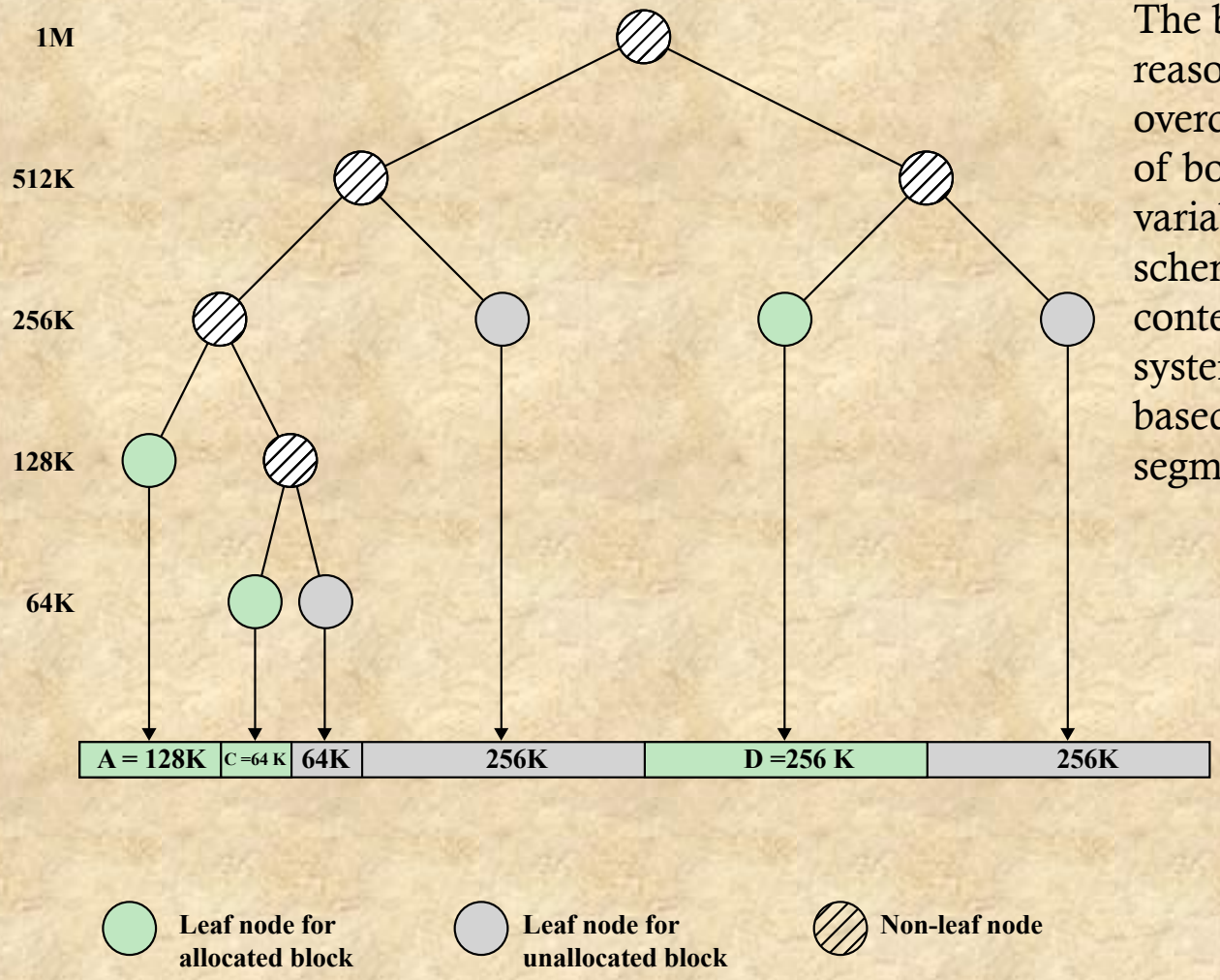


Figure 7.6 Example of Buddy System



The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes, but in contemporary operating systems, virtual memory based on paging and segmentation is superior.

Figure 7.7 Tree Representation of Buddy System

Paging

- Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in internal fragmentation, the latter in external fragmentation.
- The physical memory is divided into fixed-size blocks called **Frames**, and the process is divided into blocks of the same size called **Pages**.
- When a process executes, its pages are loaded into any available frames in memory

Pages	Frames
• chunks of a process	• available chunks of memory

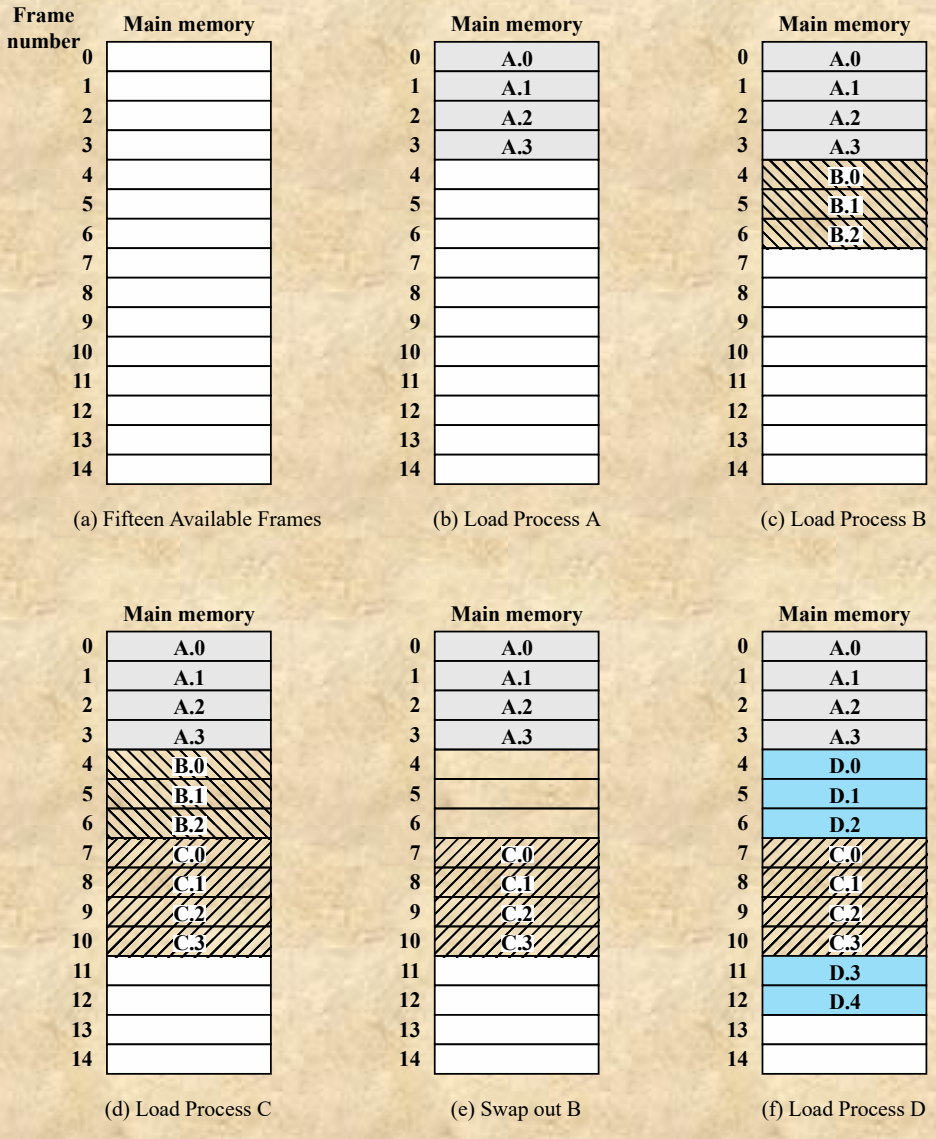
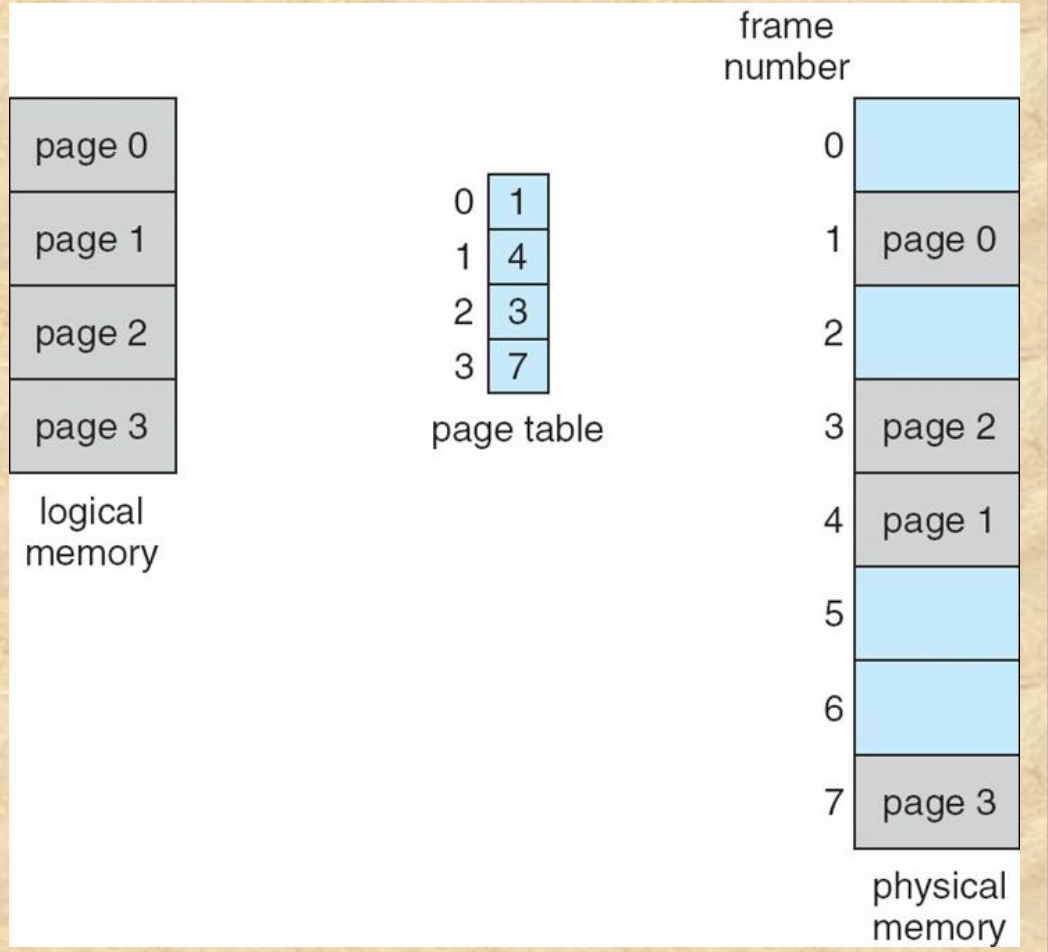


Figure 7.9 Assignment of Process Pages to Free Frames

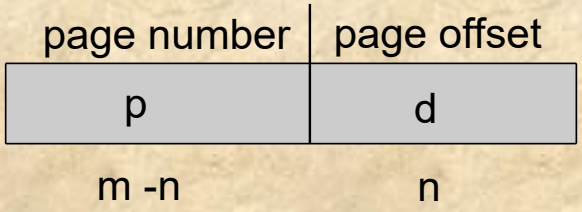
Page Table

- **Page Table:** A data structure kept by the OS for each process to map "Page Number" to "Frame Number.“
- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Used by processor to produce a physical address

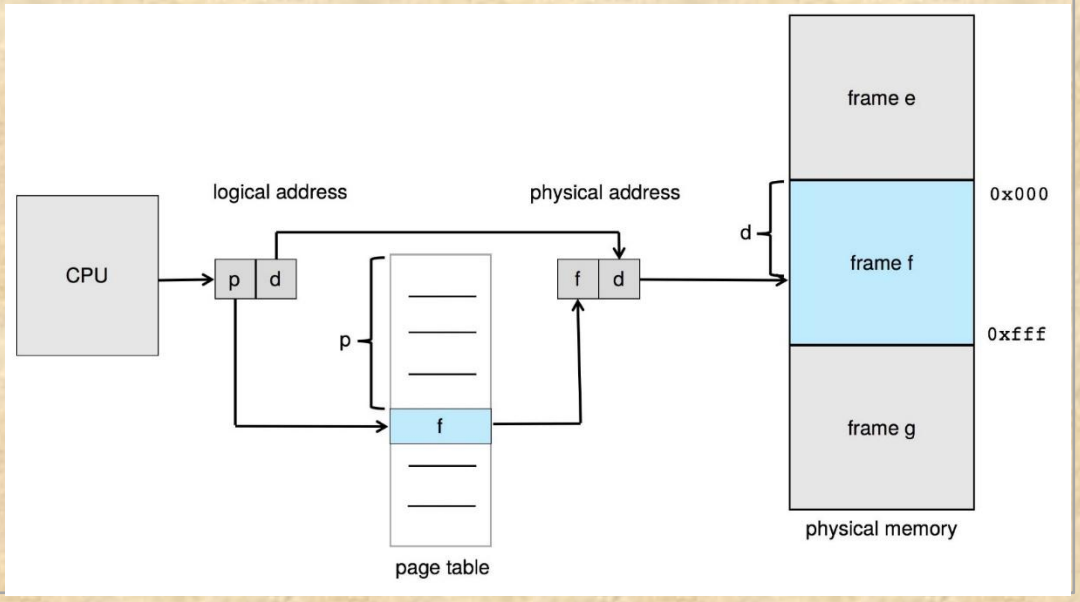


Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit



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



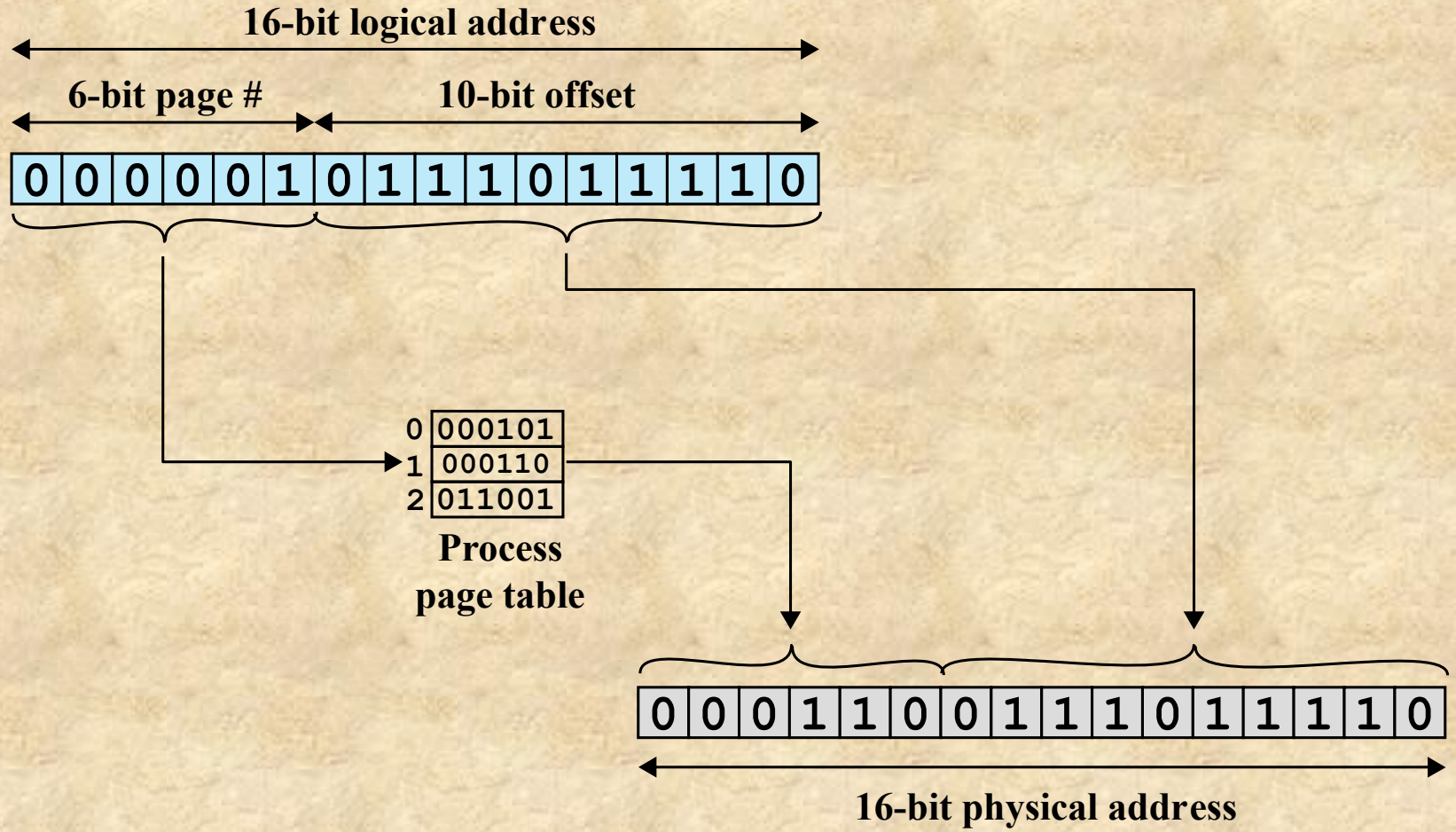


Figure 7.12 Examples of Logical-to-Physical Address Translation

Implementation of Page Table

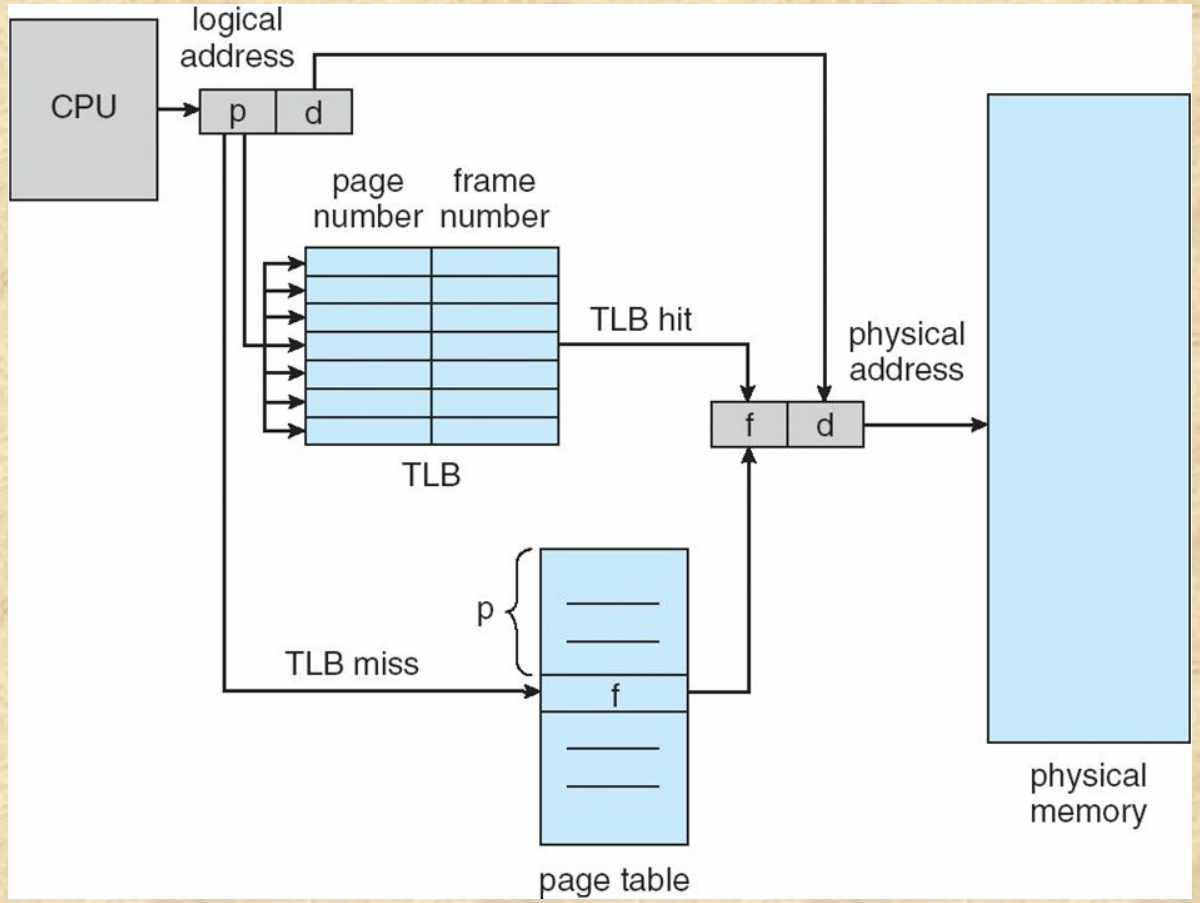
- Page table is kept in main memory
  - **Page-table base register** (**PTBR**): where the Page Table starts
  - **Page-table length register** (**PTLR**) indicates size of the page table
- It prevents the CPU from accessing memory outside the page table (a **protection mechanism**).
- In this scheme every data/instruction access requires two memory accesses
  - **Access 1 (The Lookup):** The CPU goes to the Page Table and find the frame number.
  - **Access 2 (The Fetch):** The CPU goes back to RAM to actually get the data.
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**) inside CPU.

Translation Look-Aside Buffer

- TLBs typically small (64 to 1,024 entries), very fast but expensive
- It searches all its entries simultaneously (in parallel).
- **TLB Hit**: When the CPU generates a virtual address, it checks the TLB first. If the page number is found, the physical address is returned immediately. No need to go to the Page Table in RAM.
- **TLB Miss**: If the page number is not in the TLB, the CPU must do it the hard way: go to the Page Table in RAM (Access 1), get the address, and then update the TLB so it's ready for next time.
  - Replacement policies must be considered

Paging Hardware With TLB

- The Page Table contains the mapping for every single page. It is massive and slow.
- The TLB only holds the mappings for the active pages. It lives directly inside the CPU and is very fast.



Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

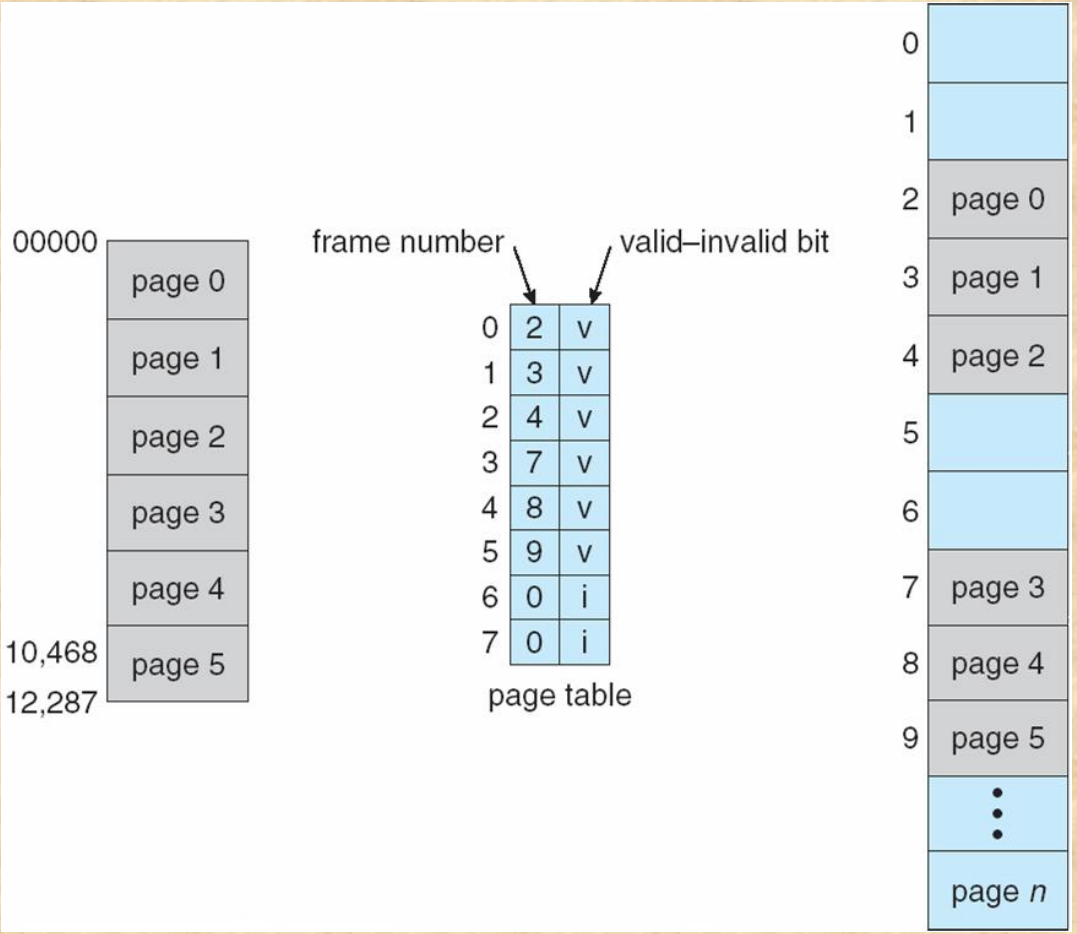
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Memory Protection

- Memory protection implemented by adding an extra column (protection bit) to every entry in the Page Table. If 0, Read-Only; If 1, can modify
  - Can also add more bits to indicate page execute-only, and so on
- A process might theoretically have a logical address space of 4 GB, but it may only be using 10 MB of actual memory. We need a way to distinguish "owned" memory from "empty" space.
- **Valid-invalid** bit attached to each entry in the page table:
  - **Valid (v):** This page is mapped to a physical frame. It is legal for the process to access it.
  - **Invalid (i):** This page is not in the process's logical address space.

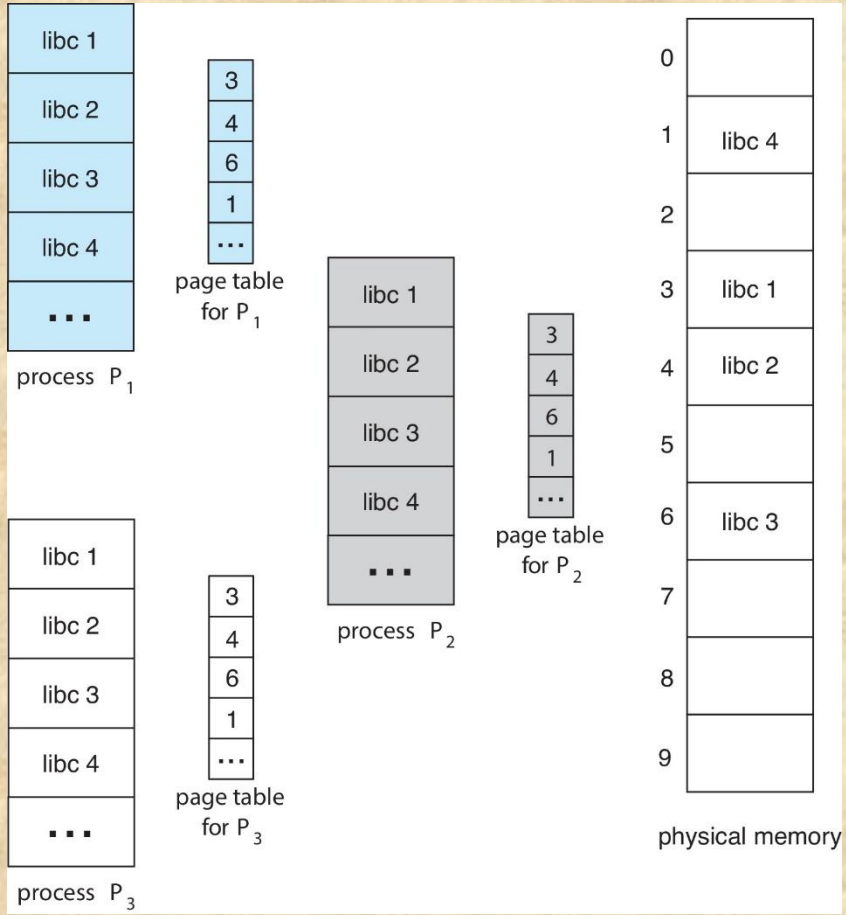
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **The ability to save massive amounts of RAM by sharing common resources between different running programs**
  - Imagine you open 5 separate windows of a text editor (like Notepad)
  - If the program code is 100 MB, you need 500 MB of RAM. With Shared Pages, you only load the code once, and all 5 users look at that same single physical copy.
- **Shared code**
  - **Reentrant Code (Shared Code)**: This means the code instructions do not change while they run. It is Read-Only.
  - Both processes page table maps the code to the same physical frame.
- **Private code and data**
  - While the instructions (the code) can be shared, the data (what the user is actually doing) cannot.
  - Separate Copies: Each process gets its own physical frames for variables, the stack, and the data segment.

Shared Pages Example

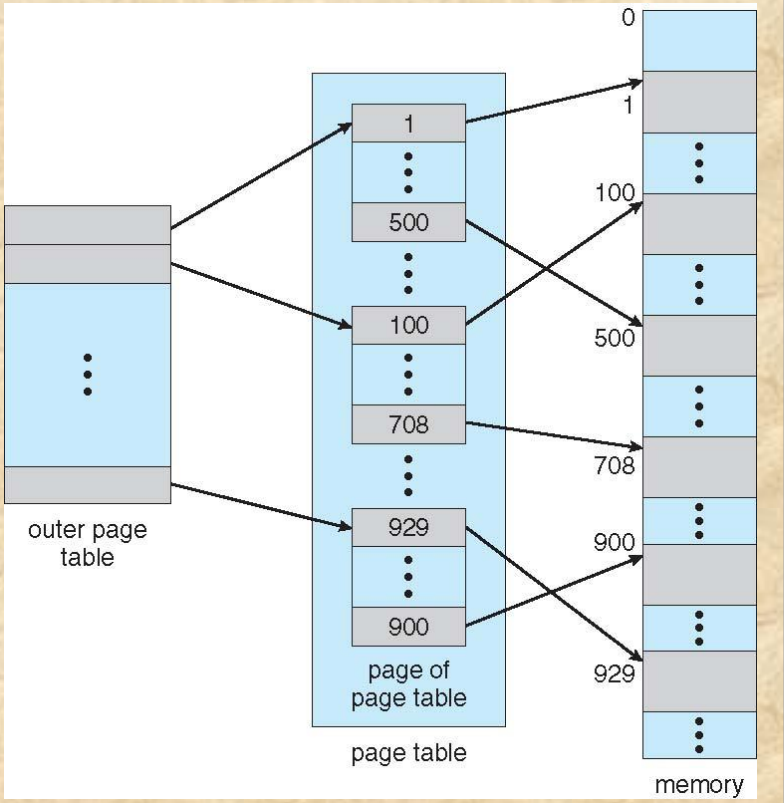


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods. Page Tables are too big to fit efficiently in memory.
  - Consider a 32-bit logical address space 2^{32} bytes (4 GB).
  - Page size of 4 KB (2^{12} bytes)
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes ➔ each process 4 MB of space for the page table alone
    - 4 MB sounds small today, but in OS, allocating 4 MB of contiguous (unbroken) memory is very difficult.
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

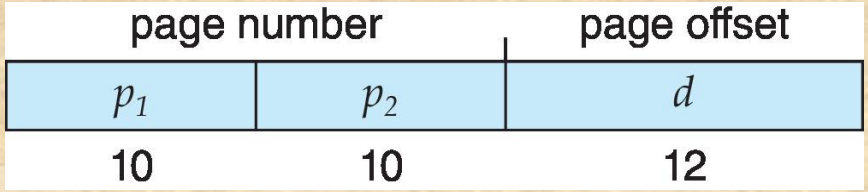
Hierarchical Page Tables

- 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
- **Outer Page Table**: Keeps track of where the inner tables are located. This table is small and fits easily in one memory frame.
- **Inner Page Tables**: These hold the actual map to physical memory.



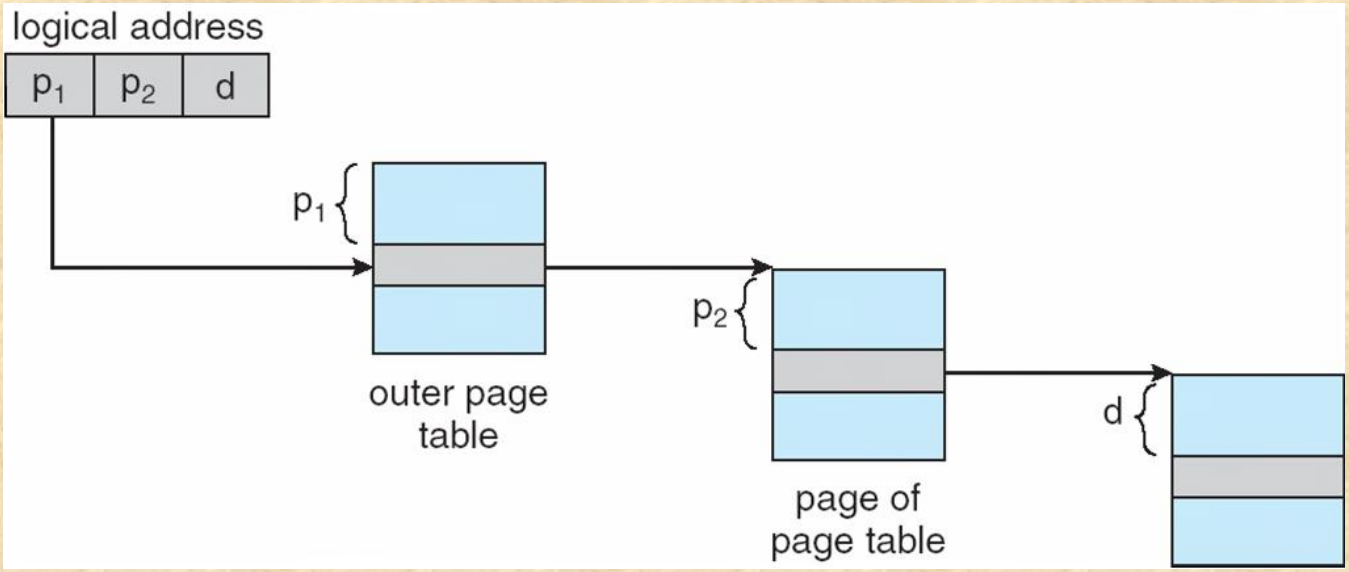
Two-Level Paging Example

- 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_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



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	inner page	offset
p_1	p_2	d
42	10	12

  - 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

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Virtual Memory

- **Virtual memory** is a technique that gives the illusion that the system has more RAM than it physically does. It allows the execution of processes that are not completely in memory.
- **Demand Paging**: Pages are loaded into RAM only when they are needed (demanded) during execution, not all at once.
- **Swapping**: The OS moves idle processes or pages from RAM to the hard disk (swap space) to free up memory for active processes.
- **Page Fault**: An interrupt that occurs when a program tries to access a page that is not currently in RAM. The OS must then fetch it from the disk.

Segmentation

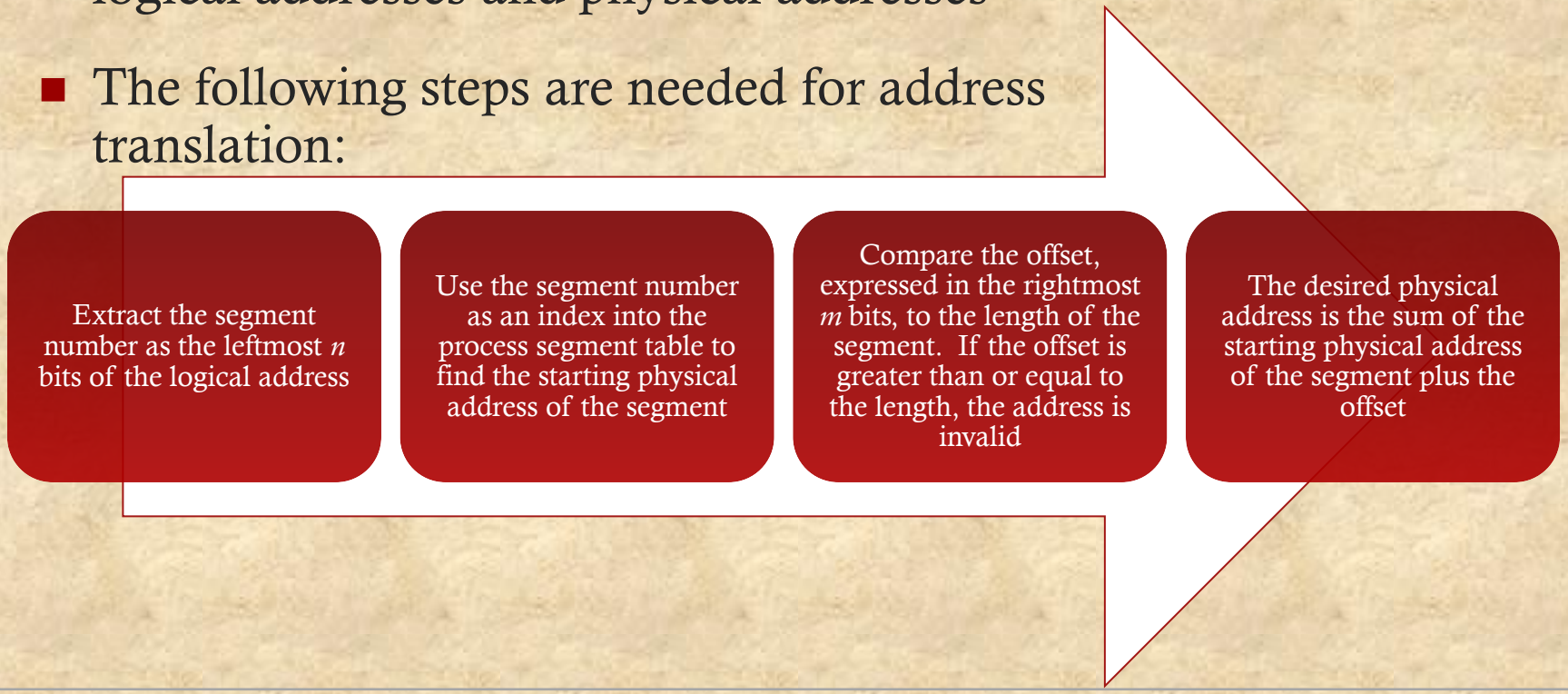
- In another mechanism, a program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments.
  - may vary in length
  - there is a maximum length
- As with paging, addressing consists of two parts:
  - segment number
  - an offset
- Similar to the idea of dynamic partitioning

Segmentation

- Whereas paging is invisible to the programmer, segmentation is usually visible
- Provided as a convenience for organizing programs and data
- Typically the programmer will assign programs and data to different segments
- For purposes of modular programming the program or data may be further broken down into multiple segments
  - the principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation

Address Translation

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:



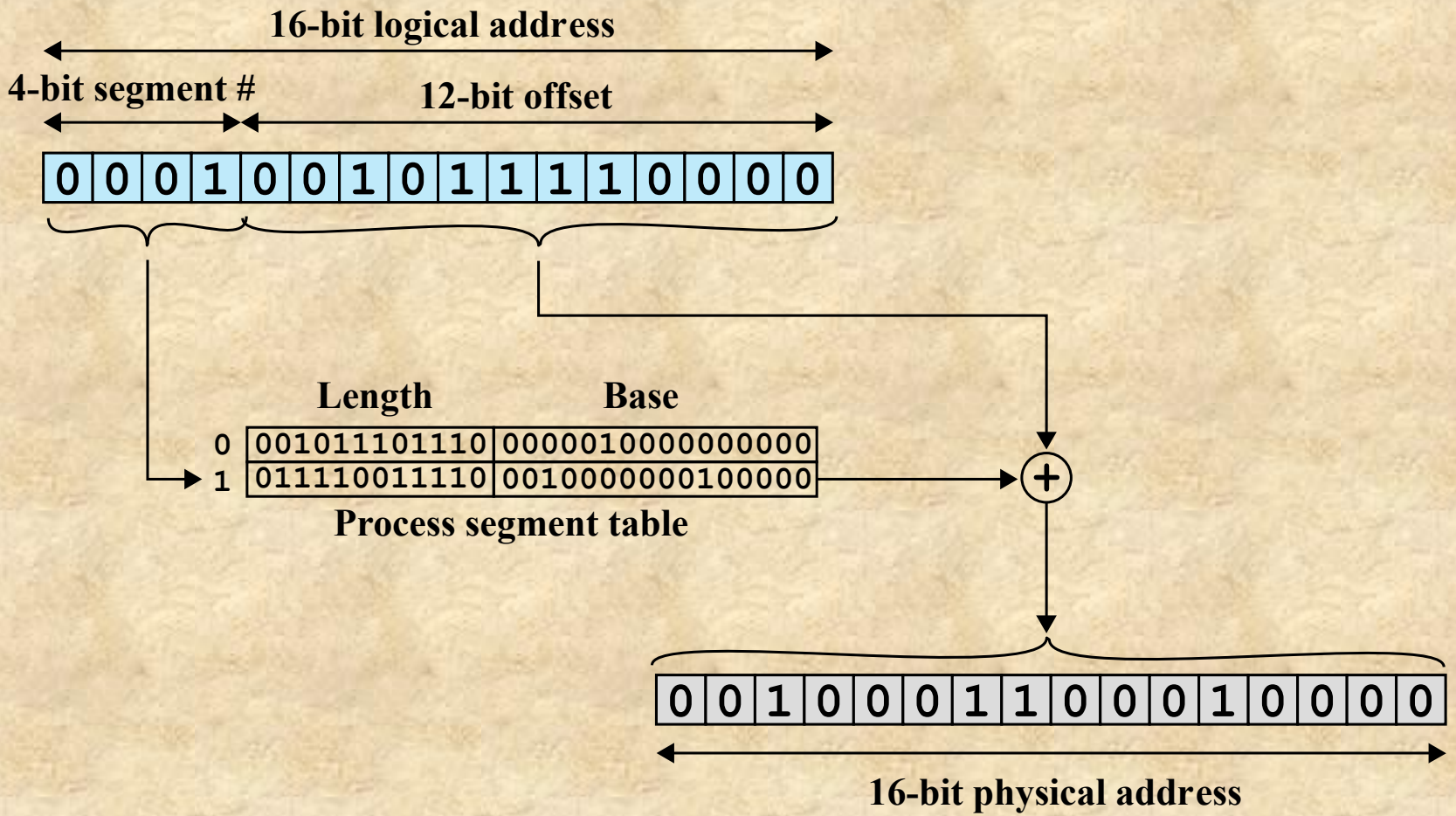


Figure 7.12 Examples of Logical-to-Physical Address Translation

Thank You