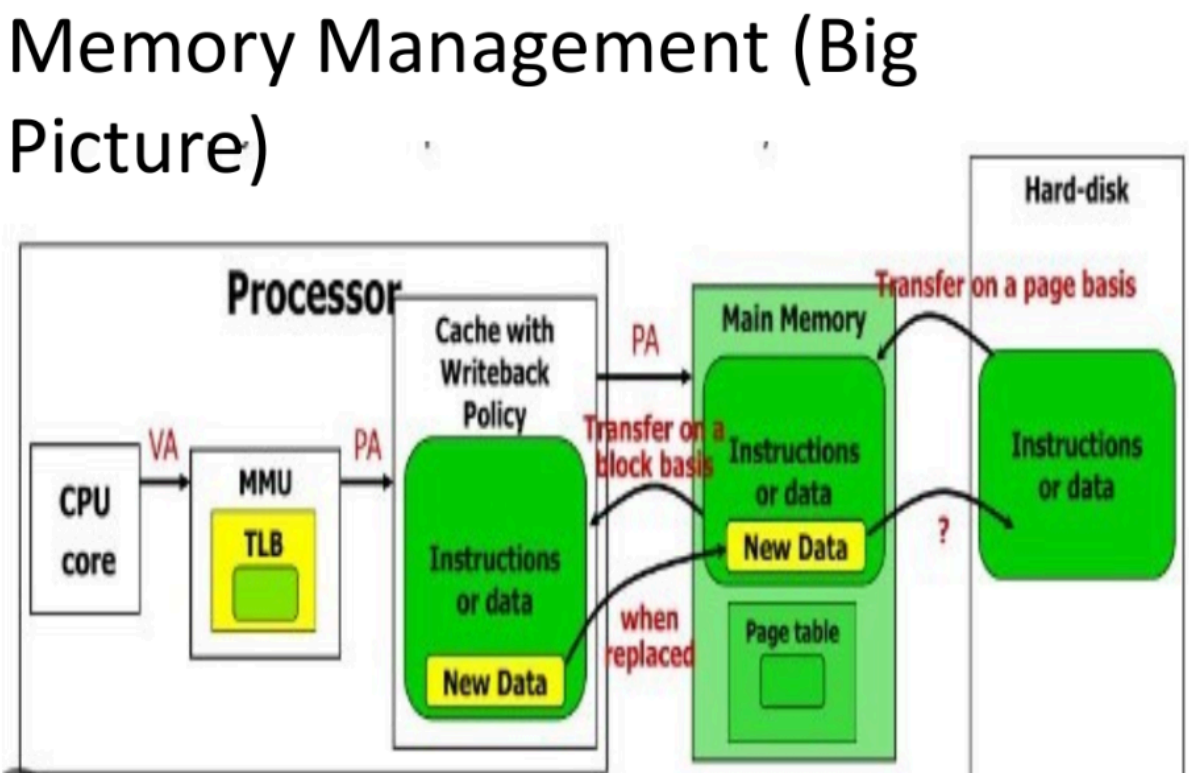
CHAPTER 9: MEMORY MANAGEMENT



This diagram, titled "Memory Management (Big Picture)," illustrates the flow of data within a computer system's memory hierarchy and highlights how different components interact. Here's an explanation of its key elements:

1. CPU Core:
    - The CPU generates a virtual address (VA) as it processes instructions or data.
2. Memory Management Unit (MMU):
    - The MMU is responsible for translating the virtual address into a physical address (PA).
    - It uses the Translation Lookaside Buffer (TLB) to speed up this address translation process.
3. Processor Cache:
    - A high-speed memory unit that stores frequently accessed data for rapid access.
    - When replaced, data in the cache is written back to main memory on a block basis.
4. Main Memory:
    - Acts as the system's primary storage for instructions and data.
    - Includes a page table that manages the organization of memory pages.
    - Transfers data to/from the processor cache and hard disk depending on system needs.

5. Hard Disk:
    - Stores data on a more permanent basis.
    - Data is transferred between main memory and the hard disk on a page basis, typically involving slower access speeds compared to memory.

Key Flows in the Diagram:

- Processor Cache to Main Memory: Data is moved in blocks when the cache is full or needs replacement.
- Main Memory to Hard Disk: Data is moved in pages, particularly during processes like swapping or paging.

ADDRESS BINDING:

Flexible Placement in Memory:

- A user process can reside in any part of the physical memory.
- The first address of the user process doesn't need to start at 00000, thanks to the operating system's role in memory allocation.

Steps of Address Translation:

- Symbolic Addresses:
    - In the source code, variables or addresses are represented symbolically (e.g., count for a variable).
- Relocatable Addresses:
    - The compiler converts symbolic addresses into relocatable ones, defined relative to a module (e.g., "14 bytes from the start").
- Absolute Addresses:
    - The linker or loader further maps relocatable addresses to physical, absolute addresses (e.g., 74014).

Address Binding as a Mapping Process:

- Each translation step binds addresses from one form to another, moving through symbolic, relocatable, and absolute forms.

Variables and Addressing:

- It notes that variables' addresses are typically represented as base + offset from the data segment. This emphasizes efficient memory organization.

3 ways of address binding:

1. **Compile Time:**

If the memory location of the process is known during compilation, absolute addresses are generated.

For example, if the starting address of a user process is predetermined (e.g., location R), the compiled code will reference that specific location.

Changing the starting location later requires recompiling the code.

2. **Load Time:**

If the exact memory location is unknown at compile time, the compiler generates relocatable code.

Final address binding happens when the program is loaded into memory.

Reloading the code is sufficient if the starting location changes.

3. **Execution Time:**

If a process can move between memory segments during its execution, address binding is delayed until runtime.

This method requires special hardware support, as discussed in future sections of the material.

Most modern operating systems rely on execution-time binding due to its flexibility.

How do programs generate instruction and data addresses?

- **Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS does nothing.

- **Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory.

- **Execution time:** Compiler generates an address, and OS can place it any where it wants in memory.

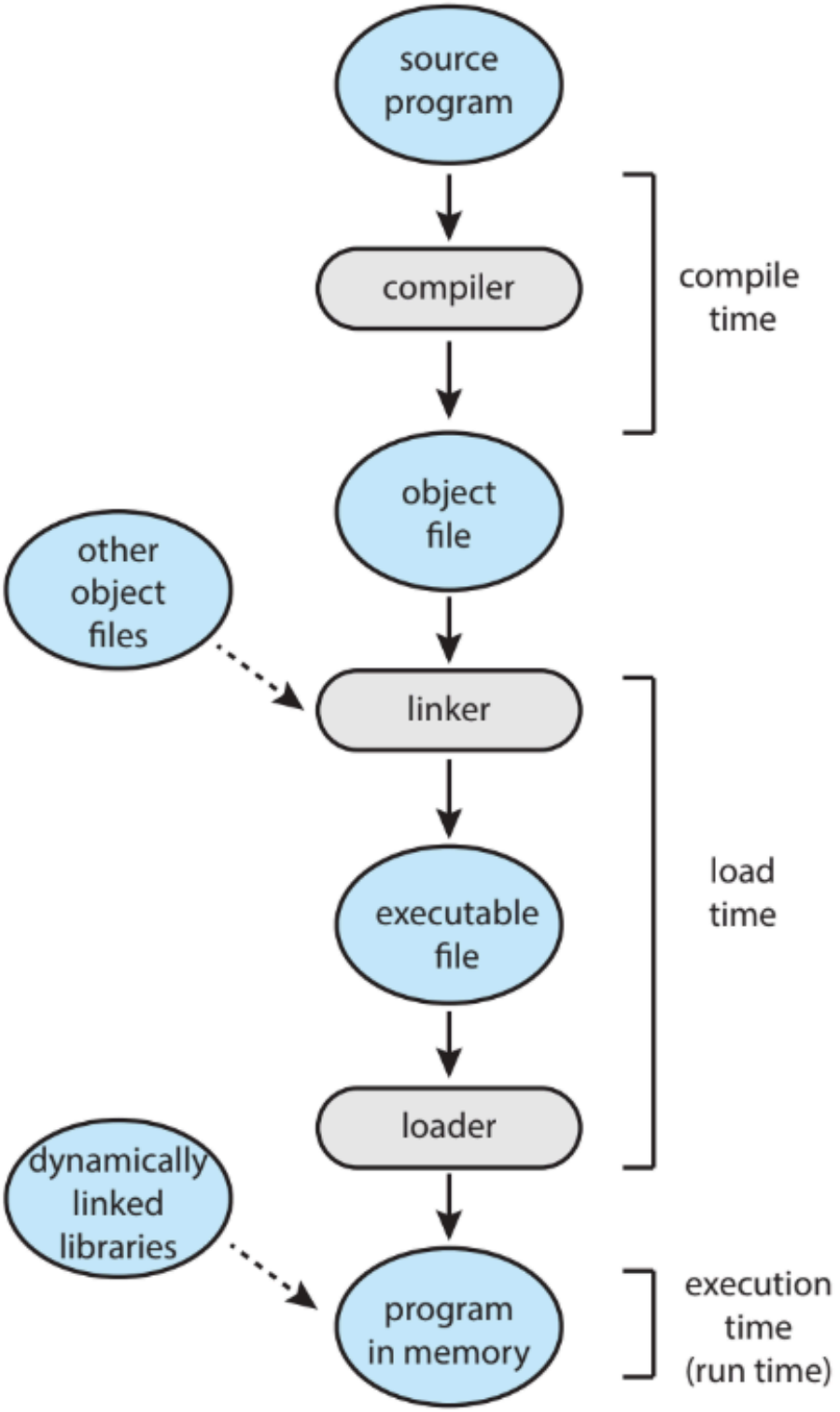


Figure 9.3 Multistep processing of a user program.

Logical Versus Physical Address Space

- Logical Address:
  - Generated by the CPU during program execution.
  - Also known as the virtual address in systems using execution-time address-binding schemes.
  - Represents the location as perceived by the user program, ranging from 0 to max.
- Physical Address:
  - The actual memory location in the system.
  - Stored in the memory-address register.
  - Represents the system's hardware memory allocation.

Binding and Spaces:

- When binding happens at compile time or load time, logical and physical addresses are identical.
- When execution-time address binding is applied, logical (virtual) and physical addresses differ. These two address spaces:
  - Logical Address Space: Set of all logical addresses generated by a program.
  - Physical Address Space: Set of physical addresses mapped to those logical addresses.

Memory Management Unit (MMU):

- The MMU is the hardware device responsible for converting logical addresses to physical addresses.
- A relocation register in the MMU is used for dynamic relocation:
  - A base value (e.g., 14000) is added to each logical address.
  - Example: Logical address 346 becomes physical address 14346.

Key Takeaway: The separation of logical and physical address spaces enhances system flexibility and proper memory management, allowing user programs to function as if they are accessing contiguous memory, while in reality, memory allocation is managed dynamically.

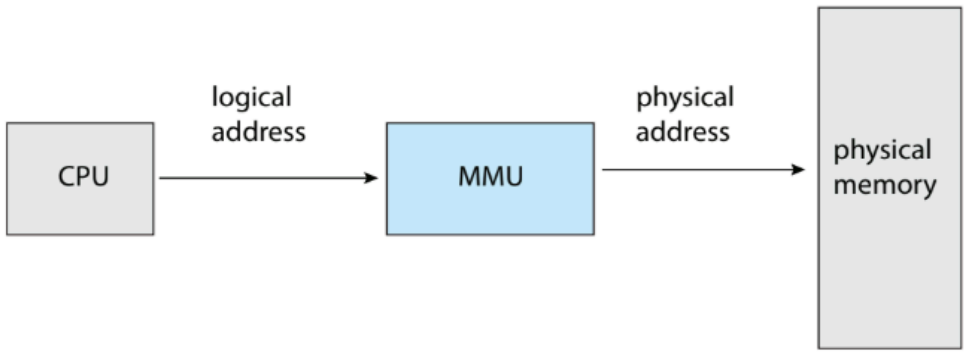


Figure 9.4 Memory management unit (MMU).

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

PROTECTION OF PROCESS ADDRESS SPACE (1):

This image illustrates the concept of memory protection in operating systems, which ensures processes can run concurrently without interfering with each other. The key idea is that every process has a separate memory space, defined by two registers:

- Base Register: Holds the smallest legal memory address for a process.
- Limit Register: Defines the size of the memory range the process is allowed to access.

For example, if the base register is 300040 and the limit register is 120090, the process can legally access memory addresses from 300040 to 420039 (inclusive).

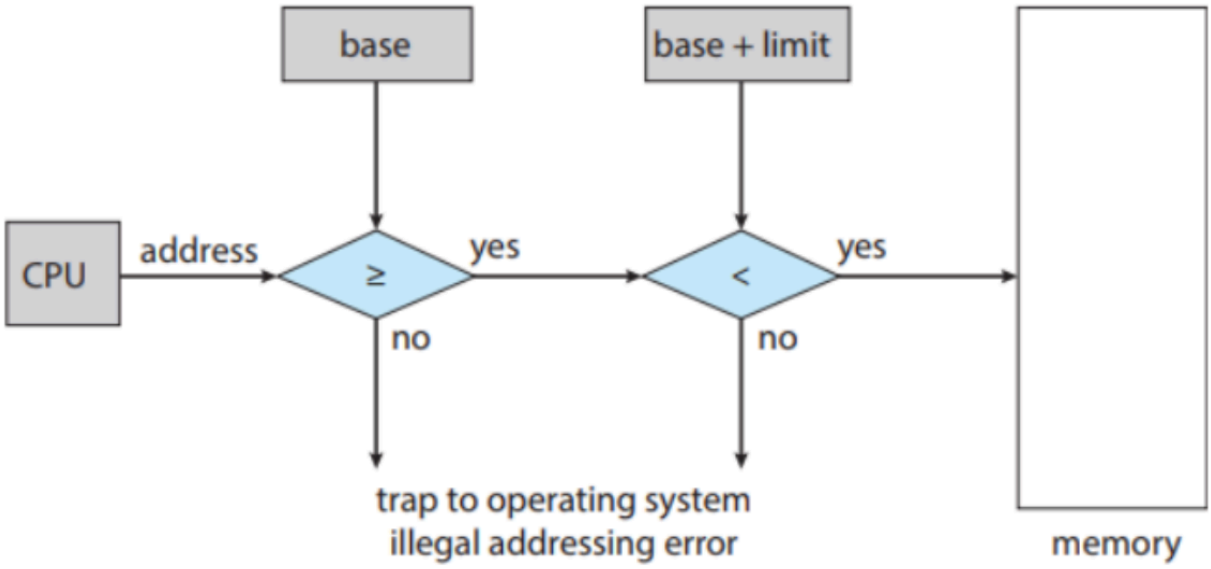
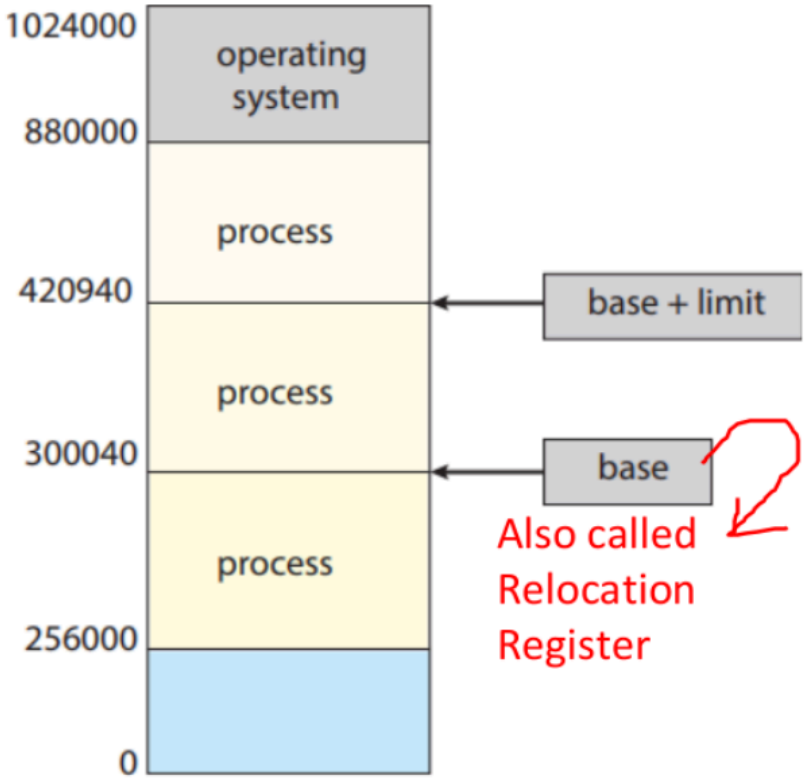


Figure 9.2 Hardware address protection with base and limit registers.



PROTECTION OF PROCESS ADDRESS SPACE (2)

1. **Protection Using Hardware**: The CPU compares every memory address generated by a program (when it runs in user mode) against specific base and limit registers. This ensures that:
    - User programs only access their own memory area.
    - Any violation triggers a "trap" (an error sent to the operating system), preventing the program from proceeding.
2. **Privileged Control of Base and Limit Registers**: These registers are critical for memory protection and can only be updated by the operating system. This is enforced through:
    - Special privileged instructions that require **kernel mode** to execute.
    - Programs running in user mode are blocked from accessing or modifying these registers.
3. **Kernel's Role in Memory Access**: When the operating system runs in **kernel mode**, it has unrestricted access to:
    - All system memory (operating system and user memory alike).
    - This unrestricted access allows the kernel to load programs, handle errors, and perform essential tasks like system calls and I/O operations.
4. **Context Switching in Multiprocessing**: In systems running multiple processes, the operating system manages "context switching," where it switches from one process to another, ensuring memory protection for each process.

PROTECTION OF PROCESS ADDRESS SPACE (3)

Why Memory Protection Matters: It's essential to ensure that a process only accesses its own allocated memory and doesn't tamper with the memory of other processes or the operating system.

Relocation and Limit Registers:

- **Relocation Register**: Stores the smallest physical address a process can use.
- **Limit Register**: Specifies the size or range of a process's memory. Example: If the relocation register is 100040 and the limit register is 74600, the logical address range for the process is valid from 100040 to 174640 (after mapping).

How It Works: Every logical memory address generated by the process is checked against the **limit register** to confirm it falls within its permissible range. The system then **maps** this address using the relocation register to calculate the actual physical address in memory.

When Registers Are Set: During a **context switch**, when the operating system schedules a process, it loads these registers with values specific to the process being executed. This ensures memory isolation and protection.

Dynamic Operating System Size: The relocation-register scheme allows flexibility in adjusting the size of the operating system dynamically. For example:

- Drivers or other OS components can be loaded into memory only when needed.
- Unused memory can be reassigned for other purposes.

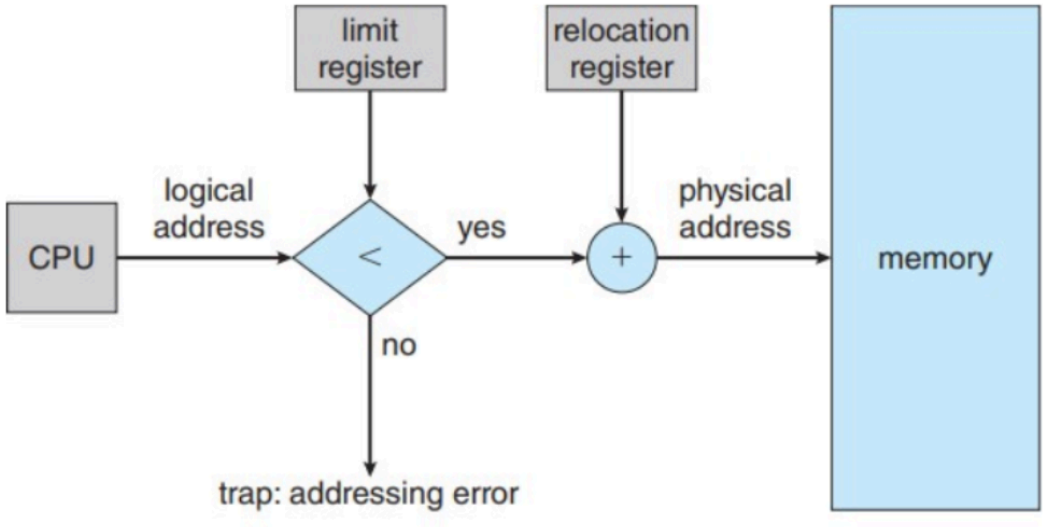
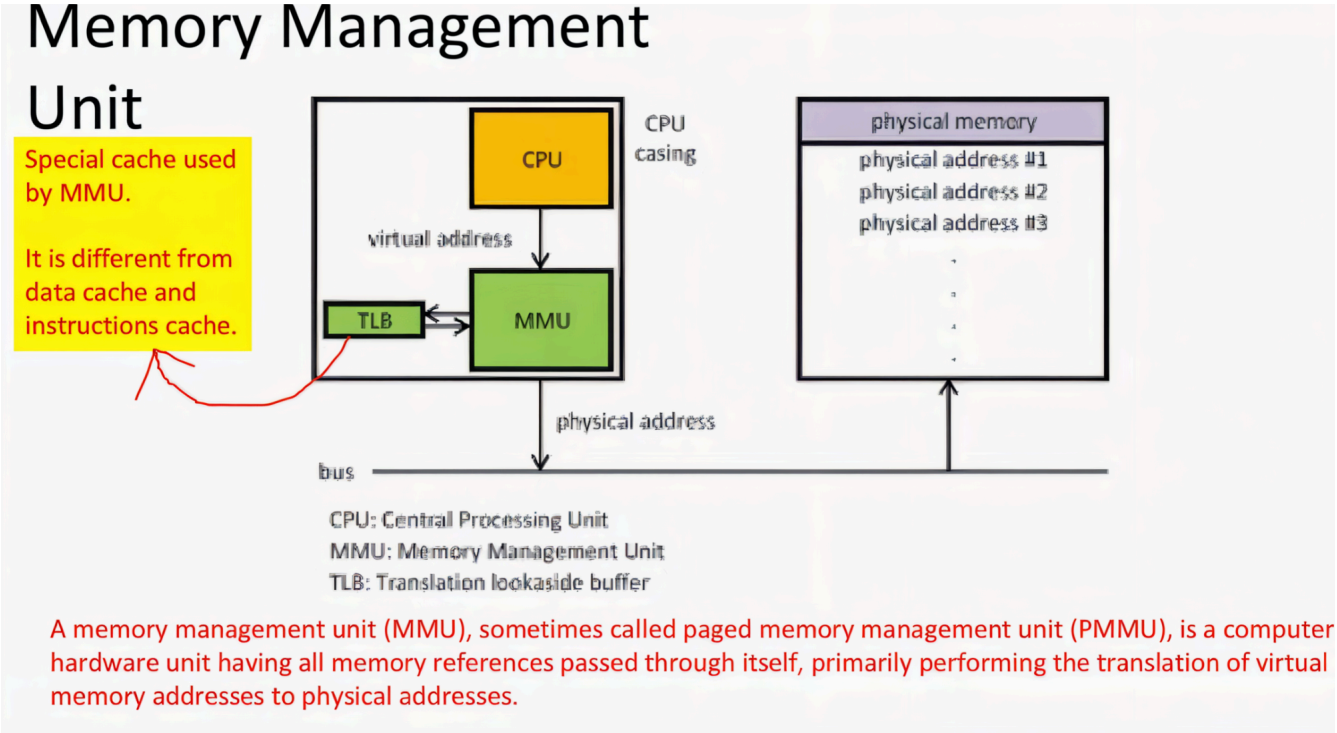


Figure 9.6 Hardware support for relocation and limit registers.

MEMORY MANAGEMENT UNIT:

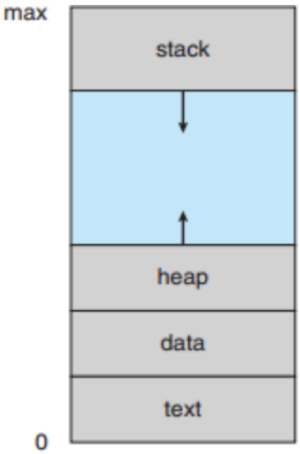


User Programs and Logical Addresses: User programs always work with **logical/virtual addresses**, never physical addresses. These logical addresses are independent of the actual memory locations.

Role of MMU (Memory Management Unit): The MMU dynamically converts logical addresses to **physical addresses** at runtime. This process, known as **execution-time binding**, ensures that the final physical address is determined only when memory is accessed.

Pointer Operations: Programs can create and manipulate logical pointers (e.g., location 346) without concern for their physical counterpart. The actual physical location is resolved using the base register when memory is accessed.

Dynamic Binding Advantage: Execution-time binding provides flexibility, allowing dynamic relocation of processes in memory while maintaining memory isolation and protection.



Layout of a program in memory

DYNAMIC RELOCATION:

- The base register is now called a **relocation register**.
- **The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory** (see Figure 9.5).
  - For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

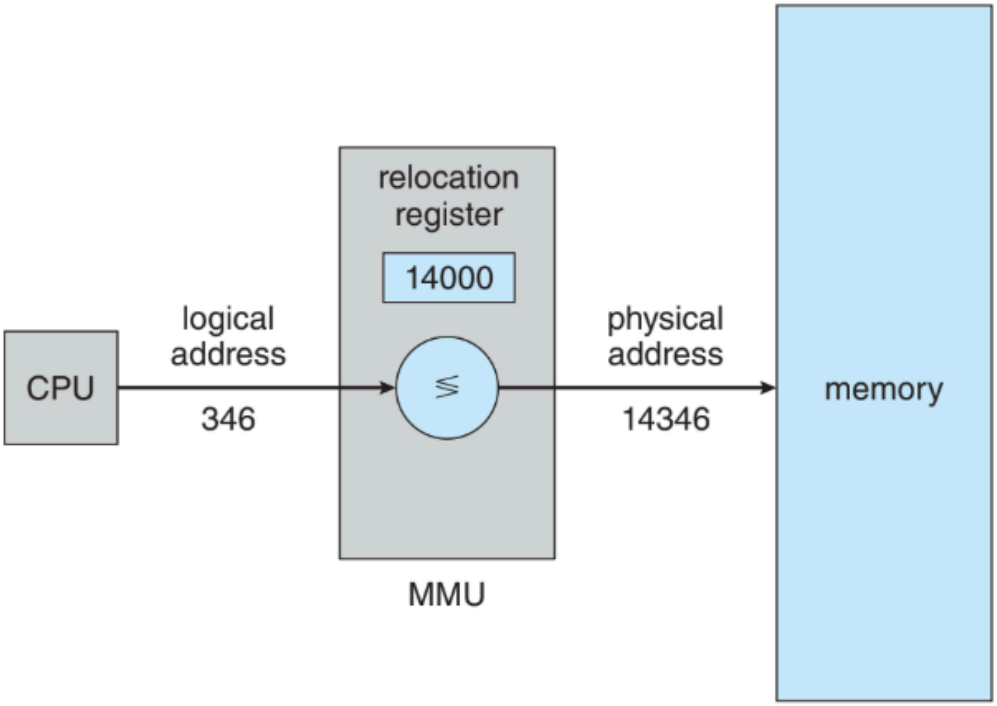


Figure 9.5 Dynamic relocation using a relocation register.

Advantages:

1. Process Movement During Execution:
   - The OS can relocate a process in memory even while it's running, offering flexibility in managing memory resources.
2. Dynamic Process Expansion:
   - Processes can grow in memory size as needed during execution, supporting modern and dynamic application requirements.

3. Simple and Efficient Hardware:
    - Only minimal hardware—two special registers, a single addition, and a comparison operation—is required to implement this scheme.

Disadvantages:

1. Increased Hardware Overhead:
    - Performing an addition for every memory reference incurs a performance cost, slowing down overall hardware efficiency.
2. Memory Sharing Limitation:
    - It does not allow processes to share memory (e.g., shared program text), which can lead to inefficient memory usage.
3. Dependence on Physical Memory Size:
    - A process cannot exceed the total size of available physical memory, limiting its scalability.
4. Restricted Multiprogramming:
    - The ability to run multiple programs simultaneously is limited, as all processes' memory must fit into the available physical memory.
5. Complex Memory Management:
    - Managing such systems increases the complexity of the operating system's memory handling.

What is Dynamic Loading?

- Instead of loading an entire program and all its data into memory, only the main program is initially loaded. Routines are only loaded into memory when they are called or needed.
- This helps to handle larger programs effectively without wasting memory space on unused code.

Advantages of Dynamic Loading:

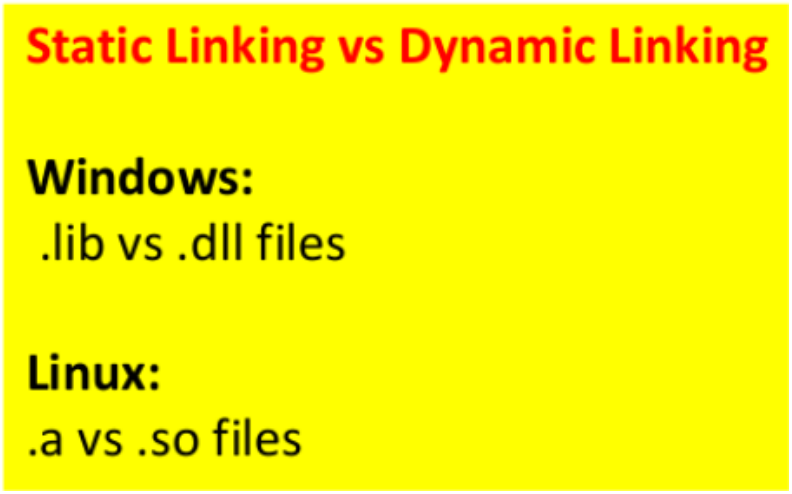
- It ensures better memory-space utilization, as only the required portions of code are loaded.
- Particularly useful for rare scenarios, like error handling, where large amounts of code might not always be needed.

User Responsibility:

- The operating system doesn't necessarily need special features for dynamic loading. It's up to programmers to design their programs to benefit from it. Some operating systems, however, provide library routines to simplify this.

Static vs Dynamic Linking:

- Static linking incorporates all the code directly into the program's executable file, whereas dynamic linking uses shared libraries (.dll files in Windows and .so files in Linux). These shared libraries are loaded dynamically at runtime, keeping the executable file size smaller.



Dynamic Linking:

- Unlike static linking, where libraries are included in the program's binary at compile time, dynamic linking attaches libraries to a program during runtime. This keeps the program's executable file smaller and more efficient.
- Dynamic linking loads only the required library routines when the program executes, making it flexible and resource-efficient.

Advantages:

- **Smaller Executable Size**: Since the libraries aren't hardcoded into the executable file, the program's size remains compact.
- **Memory Savings**: Shared libraries are loaded into memory once, and multiple programs can reference them simultaneously, avoiding duplication and saving memory.

Usage in Modern Systems:

- Both **Linux** (using `.so` files) and **Windows** (using `.dll` files) extensively utilize dynamically linked libraries for system-level and application-level programs.
- When a program calls a routine from a shared library, it uses system mechanisms to resolve and load the routine dynamically during runtime.

CONTIGUOUS MEMORY ALLOCATION:

This method involves assigning memory to processes in continuous blocks. Each process gets a single, unbroken partition in memory.

- The **variable-partition scheme** allows partitions to differ in size, and the operating system keeps track of both available (holes) and occupied memory segments.
- Initially, all memory is considered a single large block, referred to as a "hole." Over time, as processes are allocated and removed, memory splits into fragments or smaller holes of different sizes.

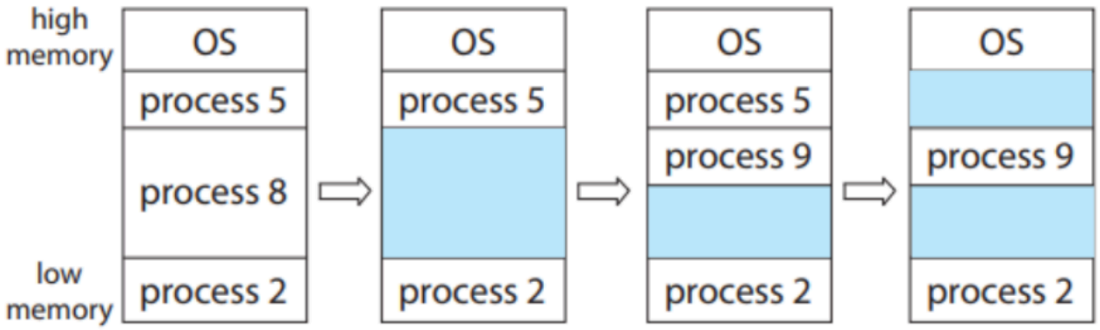


Figure 9.7 Variable partition.

STRATEGIES:

First Fit:

- Searches for the first hole that can accommodate the requested size.
- It's faster because the search ends as soon as a suitable hole is found.
- May leave larger holes fragmented, which might be less ideal for future allocations.

Best Fit:

- Allocates the smallest hole that meets the requested size, ensuring the leftover is minimal.
- Requires scanning the entire list unless it's sorted by size.
- While it minimizes wasted space, it can cause small fragments that are harder to reuse.

Worst Fit:

- Allocates the largest hole available, leaving behind the biggest leftover.
- Also requires scanning the list unless sorted.
- This strategy aims to preserve large blocks, but simulations show it performs worse than the other two.

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

FRAGMENTATION:

1. External Fragmentation:

- Occurs when free memory is divided into non-contiguous blocks due to frequent allocation and deallocation of processes.
- Even if there's sufficient total free memory, processes may fail to allocate memory because it's not contiguous.
- The **50-percent rule**: Simulations show that for every 2N allocated blocks, about N blocks are lost to fragmentation. This means around **one-third of memory can be wasted**.
- The goal is to design allocation policies that minimize this wasted space.

2. Internal Fragmentation:

- Happens when memory is allocated in blocks of fixed sizes, and a process doesn't fully utilize the allocated block.
- For example, if a process requires 8846 bytes and is allocated a block of 8848 bytes, the extra 2 bytes represent internal fragmentation.
- Keeping track of very small unused portions might be inefficient, so they're considered wasted.

Solution 1 - Compaction

One solution to the problem of external fragmentation is compaction.

The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Compaction is not always possible.

If relocation is static and is done at assembly or load time, compaction cannot be done.

It is possible only if relocation is dynamic and is done at execution time.

If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.

When compaction is possible, we must determine its cost.

The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory.

This scheme can be expensive.

Solution 2 - Paging

Paging permits the logical address space of processes to be noncontiguous. It allows a process to be allocated physical memory wherever such memory is available.

Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices.

Paging is implemented through cooperation between the operating system and the computer hardware.

90/10 Rule: Processes spend 90% of their time accessing 10% of their memory.
Optimization Tip: Keep only the actively used parts of a process in memory.
Paging Advantages:

- Pages simplify the hole-fitting problem.
- Logical memory appears contiguous, but physical memory allocation can be non-contiguous.
- Dividing memory into fixed-size pages removes **external fragmentation**.
- **Internal fragmentation** still exists (average of half a page wasted per process).

9.3.1 BASIC METHOD

Physical memory is divided into fixed-size blocks called **frames**.

Logical memory is divided into blocks of the same size called **pages**.

When executing a process:

- Pages are loaded into any available frames from a **backing store** (disk or file system).
- The backing store is also divided into fixed-size blocks (same size as frames or clusters).

Key Advantage:

- Logical address space is **independent** of physical address space.
- Example: A process can have a **64-bit logical address space** even if physical memory is much smaller.

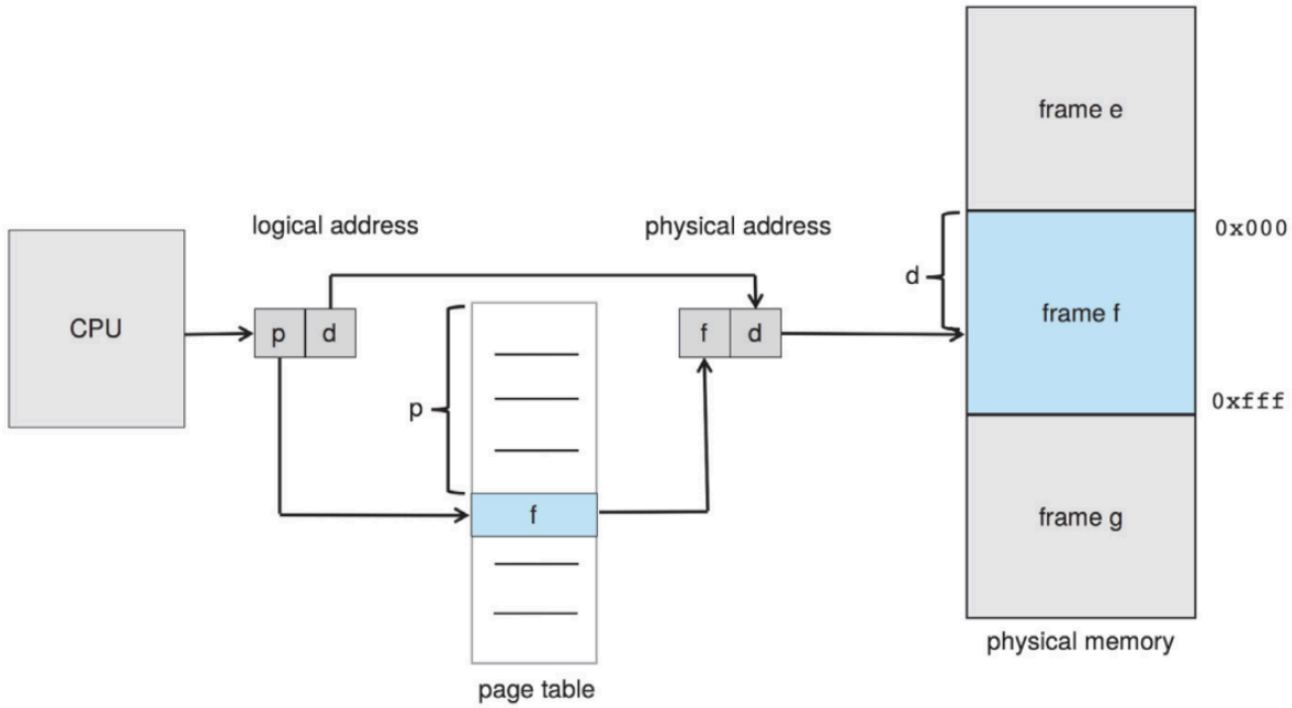


Figure 9.8 Paging hardware.

The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f.

If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m−n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

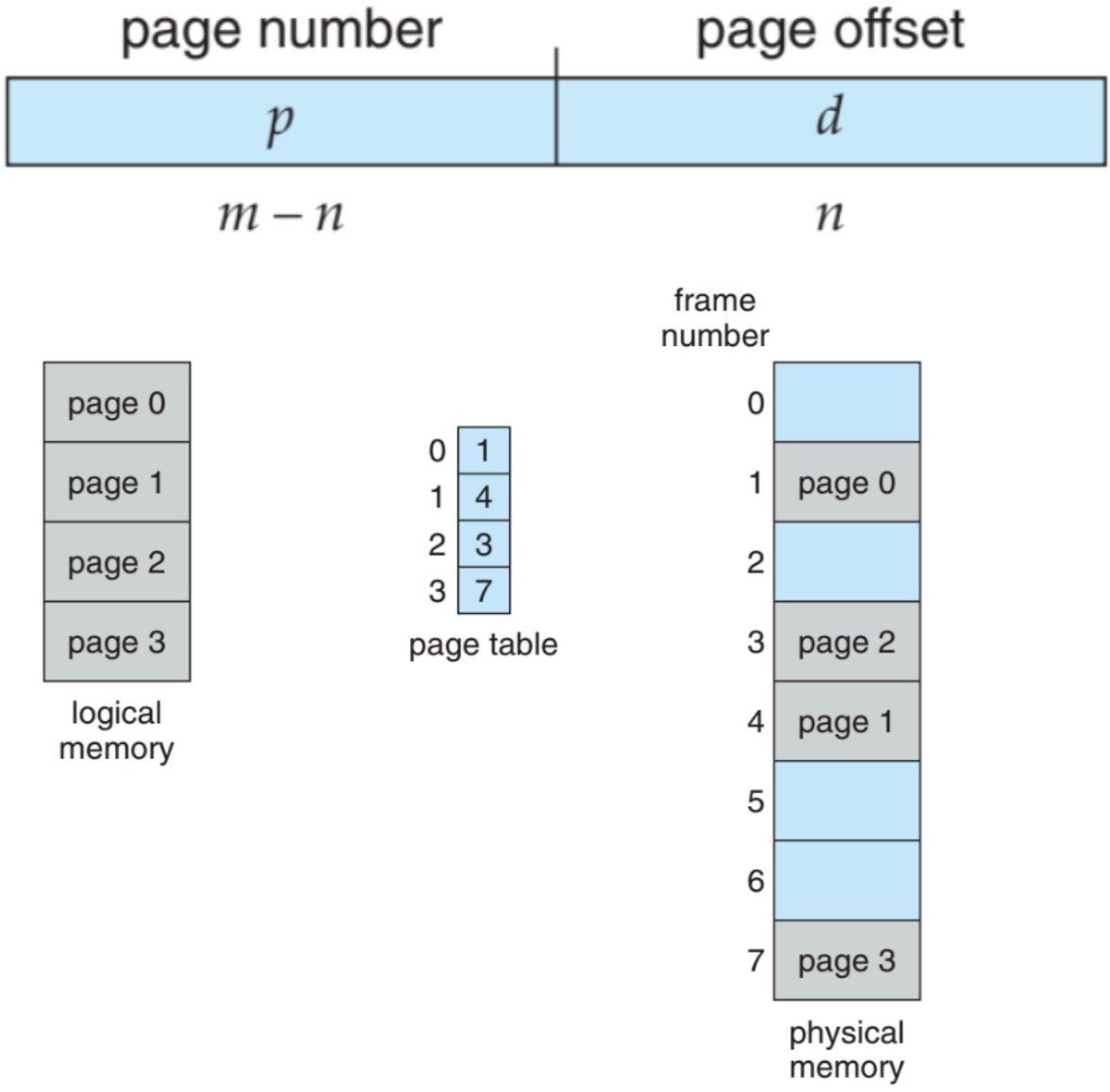


Figure 9.9 Paging model of logical and physical memory.

Logical address is broken into page number + offset.

Page table maps page number → frame number.
Physical address = (frame number × page size) + offset.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page bound- aries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.

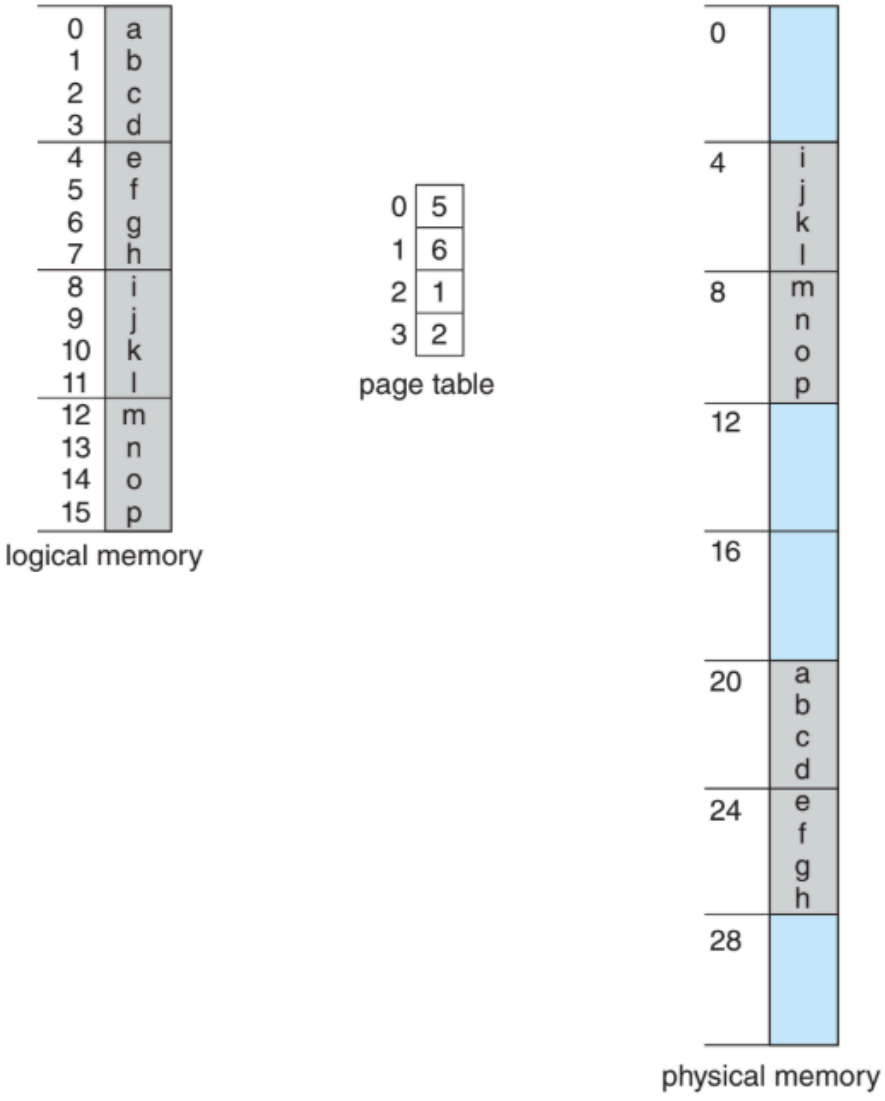


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

Here, in the logical address, n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

Problem Without TLB:

- Suppose the CPU wants to access **logical address i**.
- Step 1: Look up **page number** in the **Page Table** → find the **frame number** (1 memory access).
- Step 2: Use the **frame number** to access **physical memory** (2nd memory access).

➡️ **Total = 2 memory accesses per data access → double memory delay**

TLB (Translation Look-Aside Buffer) Solution:

- **TLB** is a **small, fast hardware cache** inside CPU.
- **Stores** some (not all) **page number → frame number** mappings.
- When CPU generates a logical address:
    1. First **check TLB** for page number.
        - **If found (TLB hit)** → directly get frame number → **1 memory access** ✅
        - **If not found (TLB miss)** → do normal page table lookup → **2 memory accesses** (slower).
    2. After a miss and lookup, **update the TLB** with the new page–frame mapping for future use.

Effective Memory Access Time (EMAT) Formula:

$$\text{EMAT} = (\text{Hit Ratio}) \times (\text{TLB Access Time}) + (\text{Miss Ratio}) \times (\text{TLB Miss Access Time})$$

- **Hit Ratio** = % of time we find the page in TLB.
- **Miss Ratio** = 1−Hit Ratio.
- **TLB Access Time** = Time for memory access when TLB hits (1 memory access).
- **TLB Miss Access Time** = Time for memory access when TLB misses (2 memory accesses — one for page table, one for data).

🔍 **Example 1 (80% Hit Ratio):**

- **Memory Access Time** = 10 ns
- **TLB Hit Time** = 10 ns

- **TLB Miss Time** = 10 ns (page table) + 10 ns (data access) = 20 ns

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

➡️ **20% slowdown** compared to the ideal 10 ns.

🔍 **Example 2 (99% Hit Ratio):**

- **Memory Access Time** = 10 ns
- **TLB Hit Time** = 10 ns
- **TLB Miss Time** = 20 ns (same as above)

$$\text{effective access time} = 0.99 \times 10 + 0.01 \times 20 = 10.1 \text{ nanoseconds}$$

➡️ **Only 1% slowdown**, almost negligible!

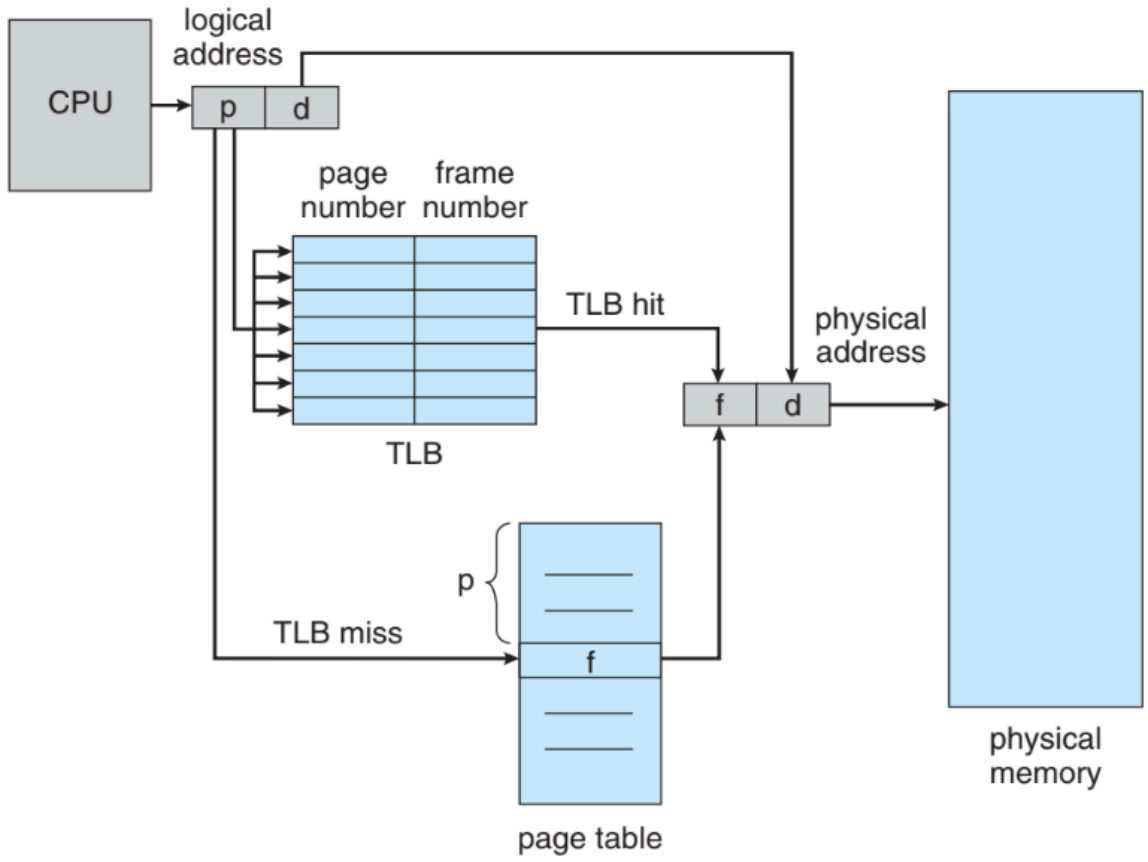
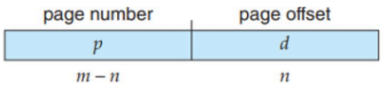


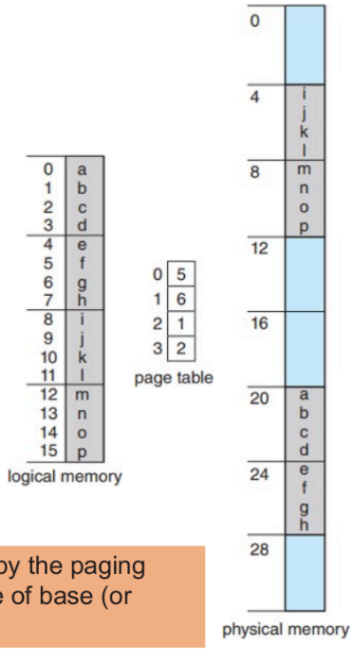
Figure 9.12 Paging hardware with TLB.

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages

Example. Logical address, $n = 2$ and $m = 4$. Page size = 4 bytes and a physical memory of 32 bytes (8 pages). Show how the programmer's view of memory can be mapped into physical memory.



2^m 2^n



1. Logical address 0 is page 0, offset 0.
2. Indexing into the page table, we find that page 0 is in frame 5.
3. Logical address 0 maps to physical address 20 [= (5 × 4) + 0].
4. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
5. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
6. Logical address 13 maps to physical address 9.

Paging is a form of dynamic relocation. Every logical address is translated by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

Paging: No External and may have some internal Fragmentation

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.

- For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, **resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes**.
- In the worst case, a process would need n pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.

ALLOCATION OF FRAMES AT PROCESS CREATION:

1. **Process Size in Pages**: Each process is divided into fixed-size pages.
2. **Frame Requirement**: For a process requiring n pages, at least n free **frames** (fixed-size blocks in physical memory) are needed.
3. **Allocation**: If enough frames are available, they're allocated to the process.
4. **Page Loading**: Each page of the process is loaded into a separate allocated frame.
5. **Page Table Update**: The **page table** of the process maps each page to the frame it is loaded into.

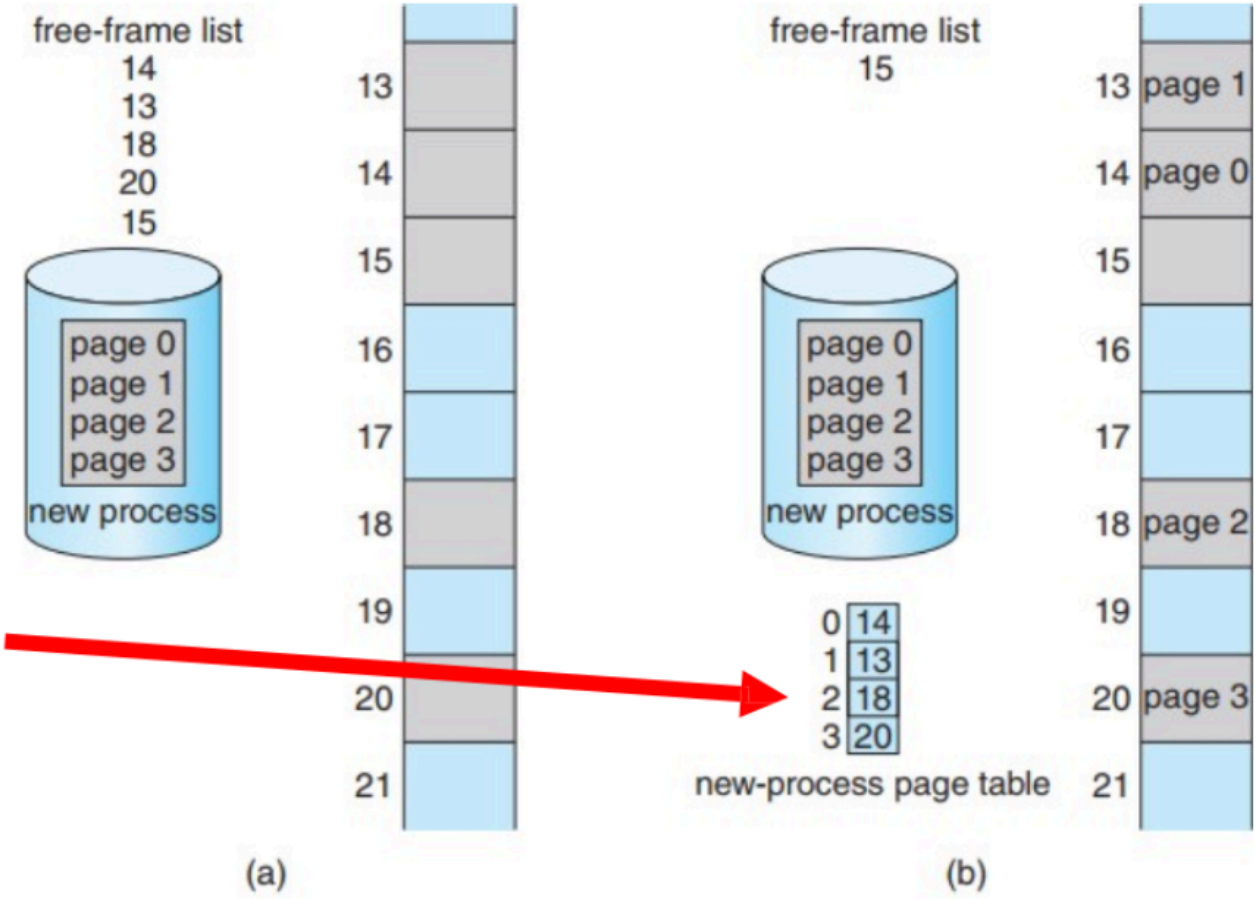


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

PROGRAMMER'S VS PAGING VIEW OF MEMORY

Programmer's View

- The programmer sees memory as a **single continuous space**.
- This gives the illusion that their program runs in isolation, starting from address 0.

Paging View (OS View)

- In reality, memory is divided into **frames**, and a program is divided into **pages**.
- Pages of the program are **scattered throughout physical memory**, not stored contiguously.

Frame Table

- A **system-wide** structure.
- One entry per **physical page frame**.
- Tracks whether a frame is **free or allocated**.

Page Table (per process)

- Maps **logical (virtual) pages** to **physical frames**.
- Stored with the **Process Control Block (PCB)**.
- When the CPU switches to a new process, it uses the **page-table base register (PTBR)** to load the new page table.

Physical Address = Base Address + Offset

Context-Switching Overhead: Changing the page table increases the **context-switch time**, as the PTBR must be updated.

PROTECTION: Each entry in the **frame table** (and often in the page table as well) may include:

✅ Valid Bit (V)

- **1**: The page is valid (i.e., it's in physical memory and mapped properly).
- **0**: The page is invalid (i.e., it may not be in memory, or the mapping is not yet created).
- **Used to prevent** accessing pages that are not currently in memory.
- **If invalid**: causes a **page fault**, prompting the OS to:
    - Load the page from disk (demand paging), or
    - Terminate the process (e.g., segmentation fault).

Protection Bits

- Typically include:
    - **Read (R)**
    - **Write (W)**
    - **Execute (X)**

- **Purpose**: Enforce access control over pages.
    - Prevents writing to read-only code pages.
    - Stops execution from data-only pages.
- These bits are checked on **every memory access** by the **Memory Management Unit (MMU)**.

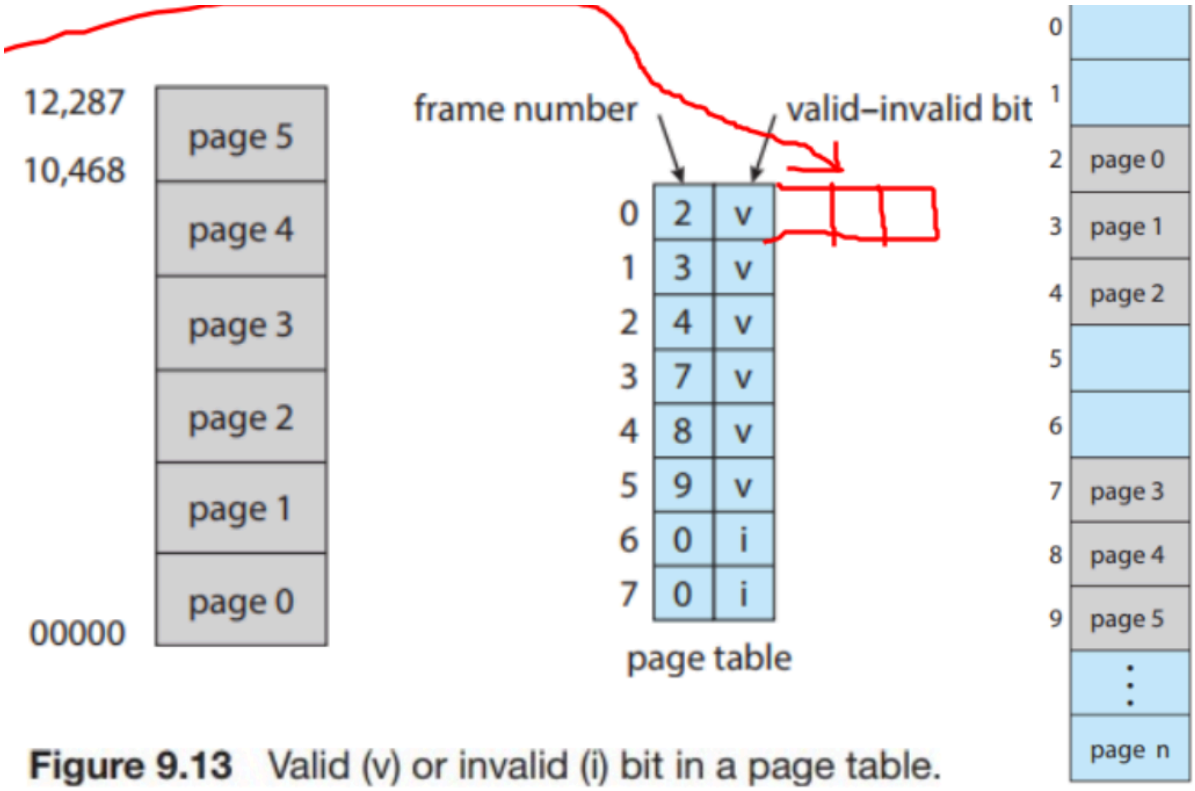


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

1. Address Space and Page Size

- The **address space is 14 bits** ⇒ ranges from **0 to 16,383 (2^{14} - 1)**.
- **Page size = 2 KB (2,048 bytes)** ⇒ so total number of pages = 163842048=8\frac{16384}{2048} = 8204816384=8 pages (0 through 7).

2. Program Usage

- The program only uses memory from **address 0 to 10,468**.
- This spans over **pages 0 to 5**:
    - Page 0: 0–2047
    - Page 1: 2048–4095
    - Page 2: 4096–6143
    - Page 3: 6144–8191
    - Page 4: 8192–10239
    - Page 5: 10240–12287 ✅

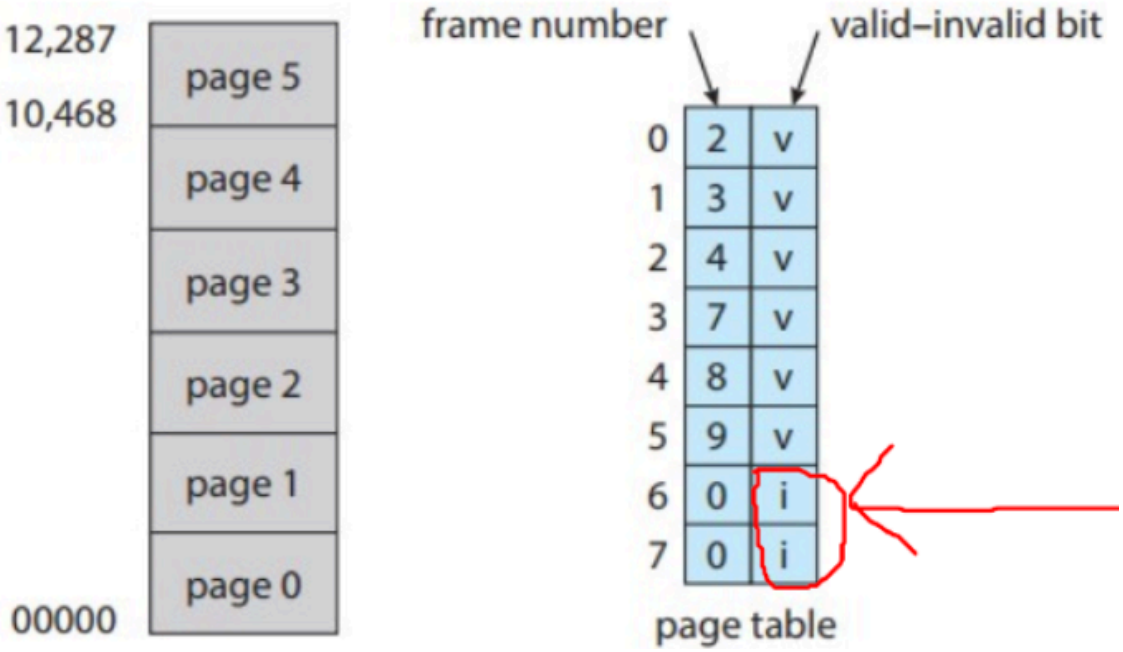
So page 5 is marked **valid**, even though only part of it (up to 10,468) is used.

3. Valid/Invalid Bit Use

- **Pages 0–5**: marked **valid (V)** → accessible
- **Pages 6–7**: marked **invalid (I)** → access will trap to OS (invalid page reference)

4. Internal Fragmentation

- The last **allocated page (page 5)** holds **from 10,240 to 12,287**, but:
  - Only **10,240 to 10,468** is used (just 228 bytes).
  - The rest (12,287 – 10,468 = **1,819 bytes**) is **wasted**, leading to **internal fragmentation**.



What Are Shared Pages? Shared pages are pages in memory (typically part of the code or library) that are mapped into the virtual address space of multiple processes. Instead of giving each process a separate physical copy, all processes point to the same physical page.

Why Share Pages?

1. **Avoid Duplication:** If 100 processes use the same 2 MB library, storing 100 copies wastes memory.
2. **Efficient Memory Use:** Only one physical copy of the page is stored.
3. **Faster Process Creation:** Sharing allows quick creation via fork() since most code pages don't need to be copied.

4. **Consistency:** All processes use the same code; updates (like library patches) are reflected universally.

Which Pages Can Be Shared?

- ✅ **Reentrant (pure) code**: e.g., shared libraries like libc.
- ✅ **Read-only data**: like constant tables.
- ❌ **Writable data (e.g., stack/heap)**: usually **not shared**, unless intentionally done using **shared memory**

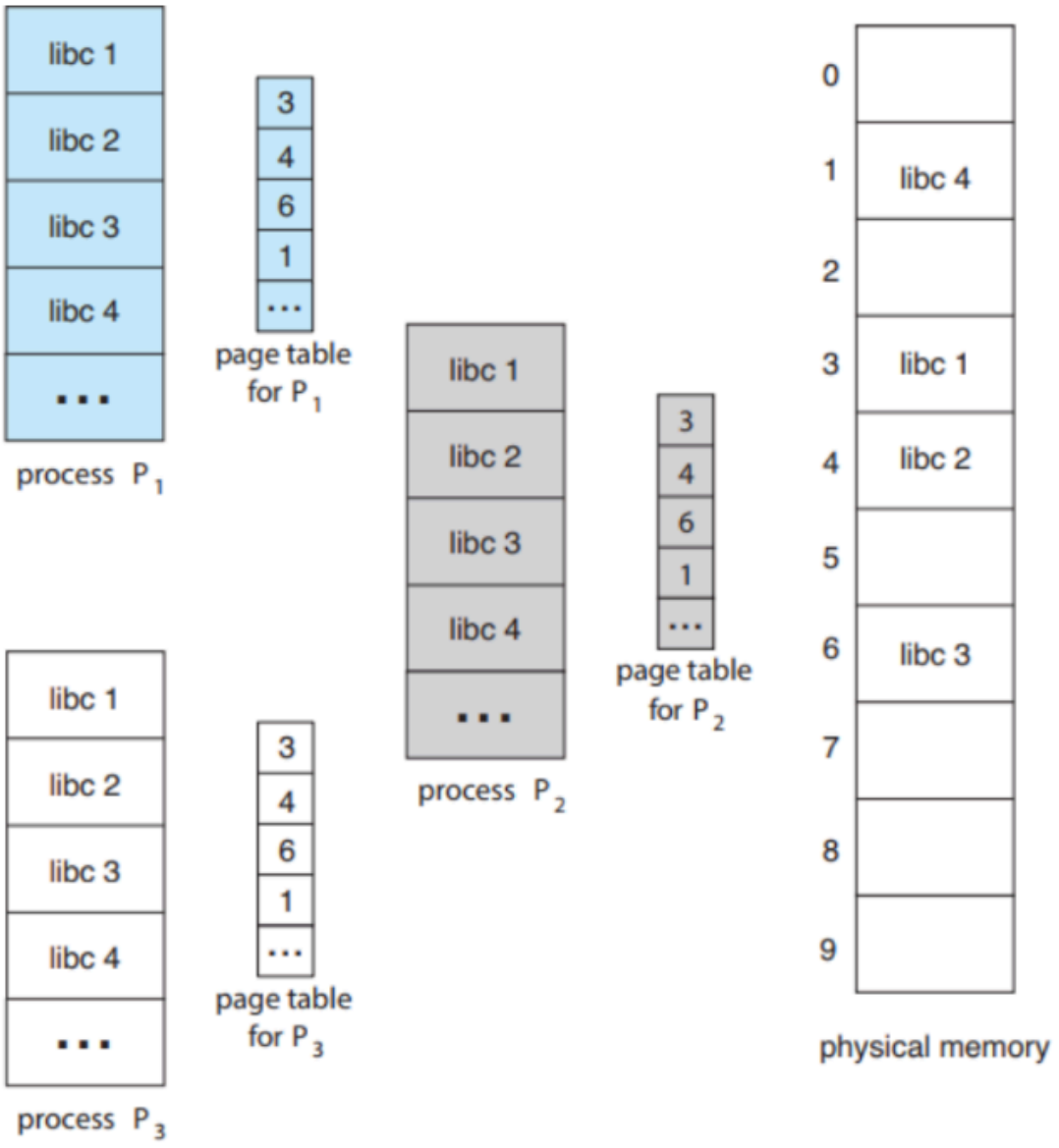


Figure 9.14 Sharing of standard C library in a paging environment.

STRUCTURE OF THE PAGE TABLE:

9.4.1 Hierarchical Paging

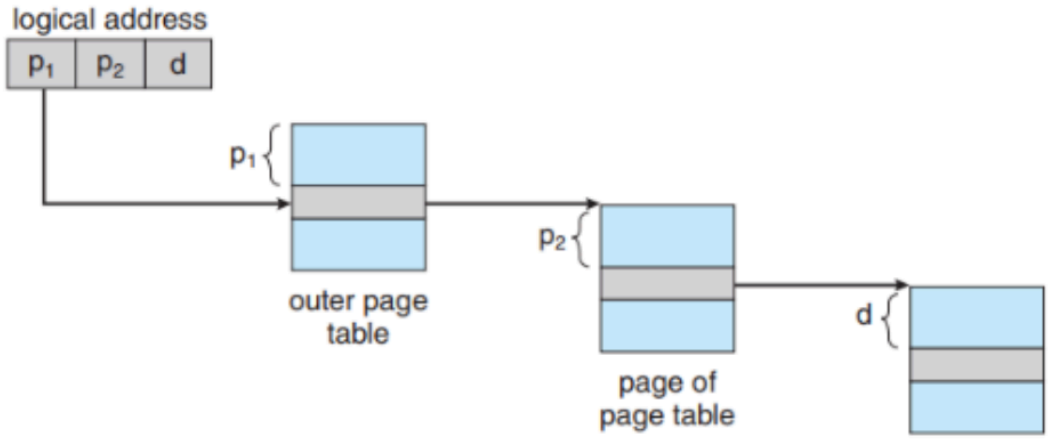


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

Hierarchical Page Table:

- **Structure:** Organized into multiple levels, typically two or three. The top-level table indexes the second-level tables, and so on, until the final level indexes the actual pages.
- **Memory Access:** Requires multiple memory accesses to translate a virtual address to a physical address, traversing through the levels of the page table.
- **Advantages:**
  - Efficient memory usage, especially for sparse address spaces.
  - Provides a balance between memory overhead and access time.
- **Disadvantages:**
  - May incur overhead due to multiple memory accesses.
  - Complexity increases with the number of levels.

9.4.2 Hashed Page Tables

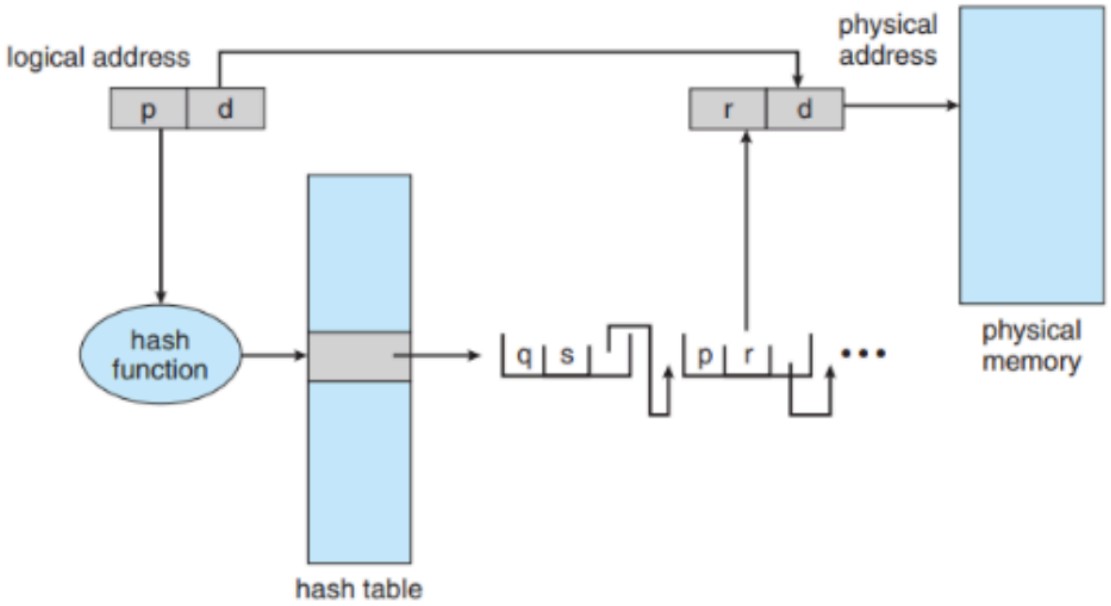


Figure 9.17 Hashed page table.

Hashed Page Table:

- **Structure:** Employs a hash function to map virtual page numbers to physical frame numbers
- **Memory Access:** Directly calculates the physical frame number using the hash function, resulting in a single memory access.
- **Advantages:**
  - Constant-time memory access, regardless of address space size.
  - Effective for handling collisions in the hash table.
- **Disadvantages:**
  - May suffer from collisions, requiring collision resolution techniques such as chaining or open addressing.
  - Hash function overhead and potential for uneven distribution of pages.

9.4.3 Inverted Page Tables

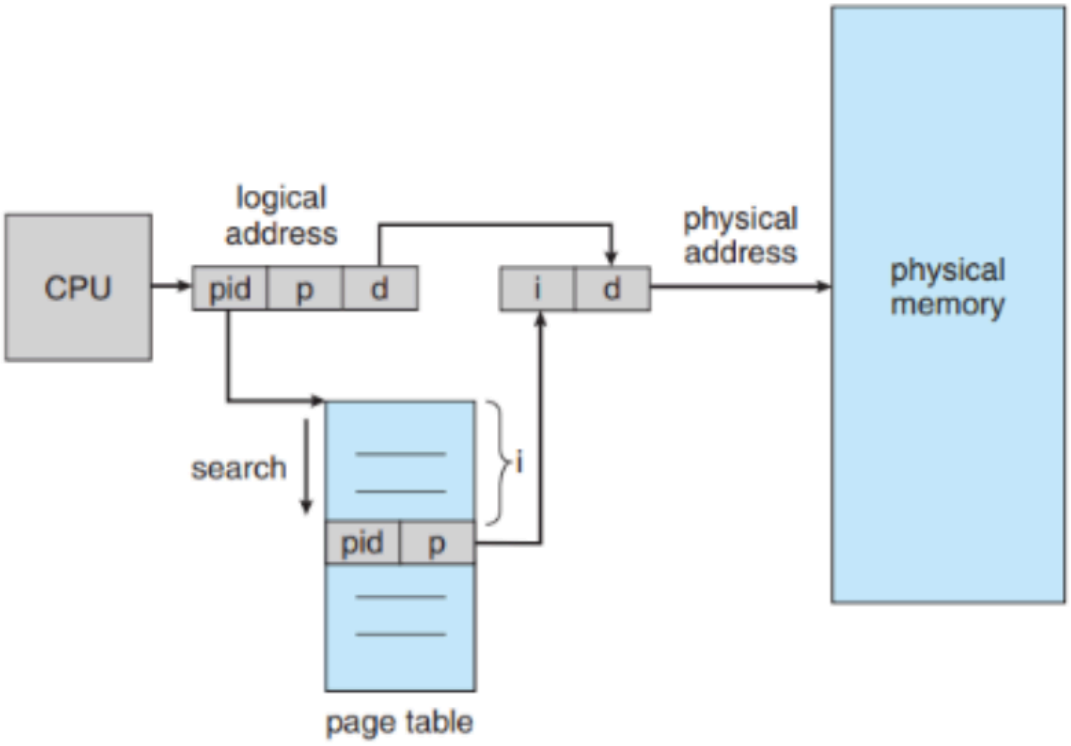


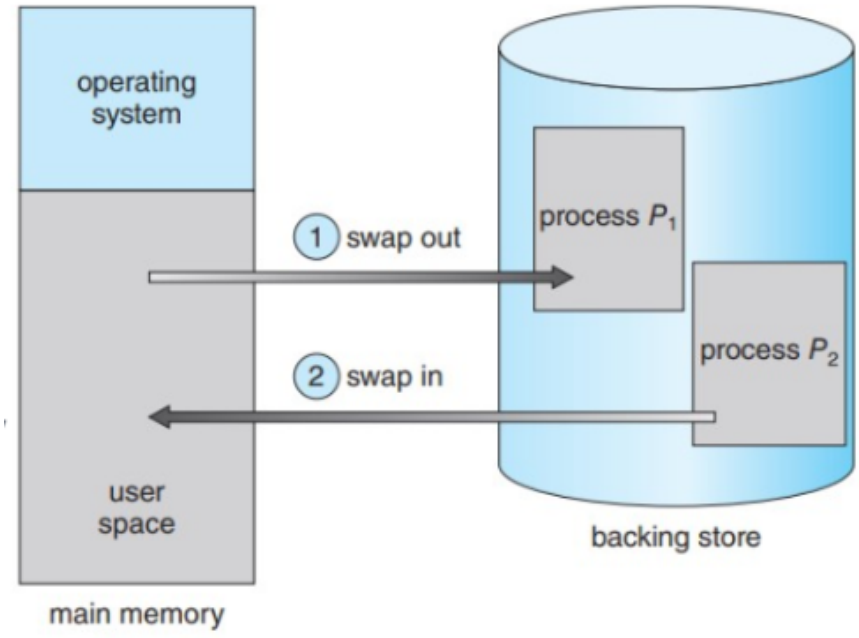
Figure 9.18 Inverted page table.

Inverted Page Table:

- **Structure:** A single table where each entry corresponds to a physical page frame, containing the process ID and the virtual page number it maps.
- **Memory Access:** Requires searching the entire table to find the mapping for a given virtual page number.
- **Advantages:**
  - Efficient memory usage, especially for systems with large physical memory but small address spaces per process.
  - Avoids duplication of identical pages across multiple page tables.
- **Disadvantages:**
  - Search time increases linearly with the number of physical pages.
  - May require additional data structures or algorithms to optimize search time.

SWAPPING: In standard swapping, the entire process is moved in and out of main memory (RAM) to and from a backing store (usually disk).

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.



Swapping with Paging:

1. Memory full? → OS selects a victim page (via LRU, FIFO, etc.).
2. The victim page is written to disk → *page out*.
3. The required page is loaded into memory → *page in*.
4. The page table is updated: valid/invalid bits, physical location, dirty bit, etc.

Assume:

- Process P has 5 pages.
- Physical memory has room for 3 pages.
- When P runs, 3 pages are loaded.
- If page 4 is accessed → **page fault** → one page is **swapped out**, and page 4 is **swapped in**.

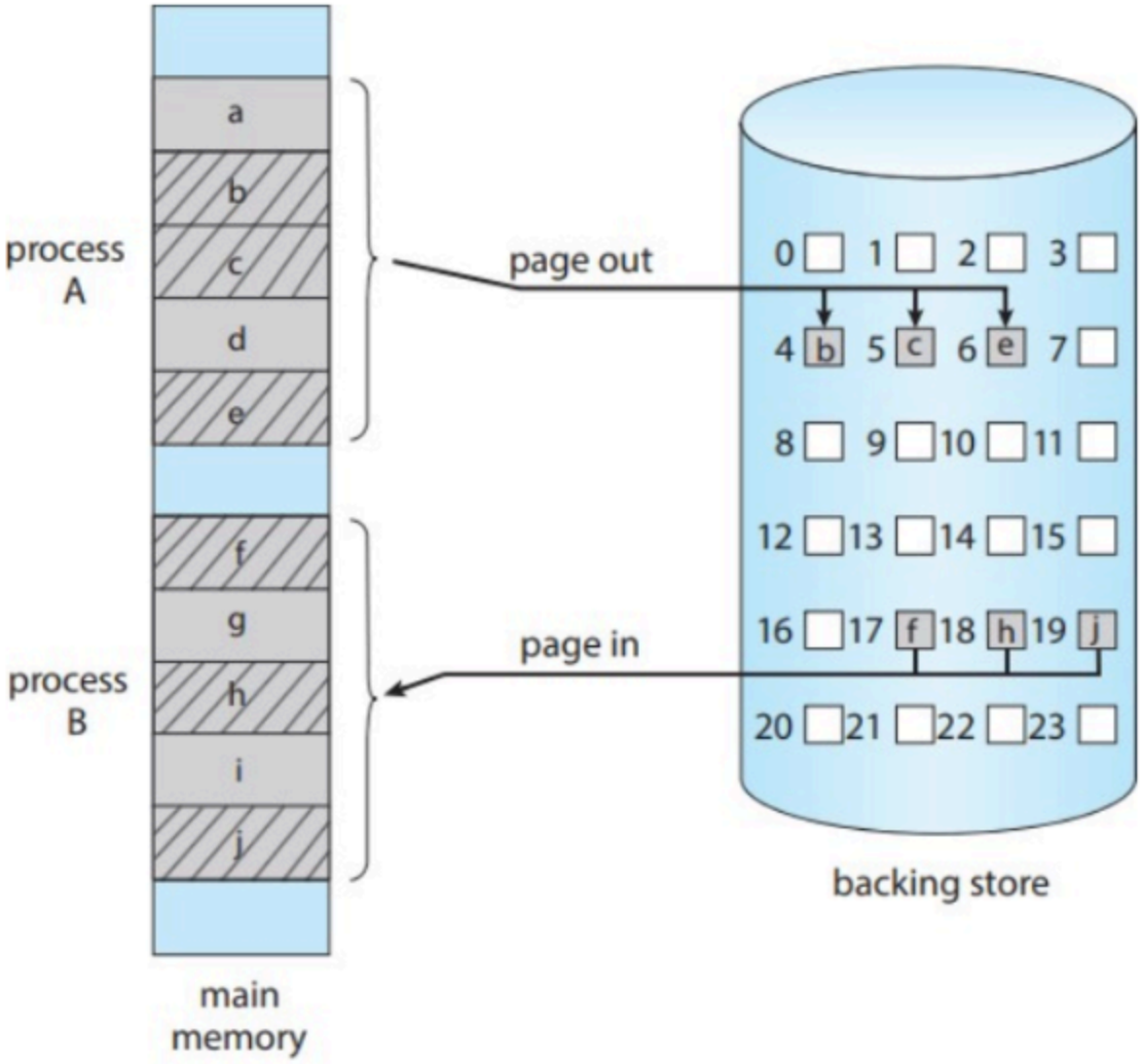


Figure 9.20 Swapping with paging.

9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

(a) How many bits are in the logical address?

A logical address consists of:

- **Page number** bits + **Offset within page** bits.
- **Page number**:
  64 pages → To uniquely identify a page, you need log2(64)=6\log_2(64) = 6log2(64)=6 bits.
- **Offset within a page**:
  Each page has 1,024 words → To uniquely identify a word within a page, you need log2(1024)=10\log_2(1024) = 10log2(1024)=10 bits.

Thus, **Logical address size = 6 (page bits) + 10 (offset bits) = 16 bits.**

✅ So your answer of **16 bits** for (a) is correct.

(b) How many bits are in the physical address?

A physical address consists of:

- **Frame number** bits + **Offset within frame** bits.
- **Frame number**:
  32 frames → To uniquely identify a frame, you need log2(32)=5\log_2(32) = 5log2(32)=5 bits.
- **Offset within a frame**:
  Each frame holds 1,024 words → So again, log2(1024)=10\log_2(1024) = 10log2(1024)=10 bits.

Thus, **Physical address size = 5 (frame bits) + 10 (offset bits) = 15 bits.**

✅ So your answer of **15 bits** for (b) is also correct.

9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

a. **First fit**:

b. 115 KB is put in 300-KB partition, leaving 185 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB

c. 500 KB is put in 600-KB partition, leaving 185 KB, 100 KB, 350 KB, 200 KB, 750 KB, 125 KB

d. 358 KB is put in 750-KB partition, leaving 185 KB, 100 KB, 350 KB, 200 KB, 392 KB, 125 KB

e. 200 KB is put in 350-KB partition, leaving 185 KB, 100 KB, 150 KB, 200 KB, 392 KB, 125 KB

f. 375 KB is put in 392-KB partition, leaving 185 KB, 100 KB, 150 KB, 200 KB, 17 KB, 125 KB

g. **Best fit**:

h. 115 KB is put in 125-KB partition, leaving 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB

i. 500 KB is put in 600-KB partition, leaving 300 KB, 100 KB, 350 KB, 200 KB, 750 KB, 10 KB

j. 358 KB is put in 750-KB partition, leaving 300 KB, 100 KB, 350 KB, 200 KB, 392 KB, 10 KB

k. 200 KB is put in 200-KB partition, leaving 300 KB, 100 KB, 350 KB, 0 KB, 392 KB, 10 KB

l. 375 KB is put in 392-KB partition, leaving 300 KB, 100 KB, 350 KB, 0 KB, 17 KB, 10 KB

m. **Worst fit**:

n. 115 KB is put in 750-KB partition, leaving 300 KB, 600 KB, 350 KB, 200 KB, 635 KB, 125 KB

o. 500 KB is put in 635-KB partition, leaving 300 KB, 600 KB, 350 KB, 200 KB, 135 KB, 125 KB

p. 358 KB is put in 600-KB partition, leaving 300 KB, 242 KB, 350 KB, 200 KB, 135 KB, 125 KB

q. 200 KB is put in 350-KB partition, leaving 300 KB, 242 KB, 150 KB, 200 KB, 135 KB, 125 KB

r. 375 KB must wait

9.7 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

(a) 3085

- Page number = 3085 ÷ 1024 = 3 (integer part)
- Offset = 3085 mod 1024 = 3085 - (3 × 1024) = 3085 - 3072 = **13**

✅ **Page = 3, Offset = 13** (Correct)

(b) 42095

- Page number = 42095 ÷ 1024 = 41 (integer part)
- Offset = 42095 mod 1024 = 42095 - (41 × 1024) = 42095 - 41984 = **111**

✅ **Page = 41, Offset = 111** (Correct)

(c) 215201

- Page number = 215201 ÷ 1024 = 210 (integer part)

- Offset = 215201 mod 1024 = 215201 - (210 × 1024) = 215201 - 215040 = **161**

✅ **Page = 210, Offset = 161** (Correct)

(d) 650000

- Page number = 650000 ÷ 1024 ≈ 634 (integer part)
- Offset = 650000 mod 1024 = 650000 - (634 × 1024) = 650000 - 649216 = **784**

✅ **Page = 634, Offset = 784** (Correct)

(e) 2000001

- Page number = 2000001 ÷ 1024 ≈ 1953 (integer part)
- Offset = 2000001 mod 1024 = 2000001 - (1953 × 1024) = 2000001 - 1999872 = **129**

✅ **Page = 1953, Offset = 129** (Correct)

9.9 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

(a) A logical address = **page number bits** + **offset within page bits**.

- **Page number bits**:
  256 pages → 8log2(256)=8 bits.
- **Offset bits**:
  4 KB = 4096 bytes → log2(4096) = 12log2(4096)=12 bits.

Thus: Logical address size = 8 (page bits) + 12 (offset bits) = **20 bits**.

✅ So your answer for (a) is **correct: 20 bits**.

(b) A physical address = **frame number bits** + **offset within frame bits**.

- **Frame number bits**:
  64 frames → log2(64) = 6log2(64)=6 bits.
- **Offset bits**:
  Same 4 KB page/frame size → log2(4096)= 12log2(4096)=12 bits.

Thus: Physical address size = 6 (frame bits) + 12 (offset bits) = **18 bits**.

✅ So your answer for (b) is also **correct: 18 bits**.

9.10 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

a. In a **conventional page table**, **each page** of the logical address space has **one page table entry**.

- **Page size = 4 KB = 2^12 bytes** → so **offset = 12 bits**.
- Thus, the number of **page numbers** = 2^{32-12} = 2^20 pages.

✅ Therefore, **the page table must have 2^20 entries**.

Answer for (a): 2^{20} entries ✅

b. An **inverted page table** has **one entry for each physical frame** (not for each logical page). Now:

- **Physical memory = 512 MB** = 512×2^20= 2^29 bytes.
- **Page size = 4 KB = 2^{12}** bytes.

Thus, number of **frames** = 2^29/2^12

= **131,072 frames** = **128K frames** (since 1K = 1024).

✅ Therefore, **the inverted page table has 128K entries**.

9.8 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

Conventional, single-level page table will have 2^10 = 1024 entries. Inverted page table will have 2^5 = 32 entries. The maximum amount of physical memory is 2^16 = 65536 (or 64 KB.)

BACKGROUND:

- Memory management's goal: Ensuring executable instructions are in physical memory.
- Challenge in multitasking: Allocating memory efficiently across multiple programs.
- Fragmentation solutions: Traditional compaction versus paging, which eliminates external fragmentation and enables noncontiguous memory allocation.

Do we need the entire program in memory?

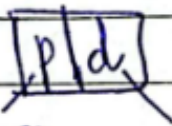
- Rarely executed error-handling code: Programs often include routines for handling unusual errors, but these errors occur so infrequently that the code is seldom executed.
- Over-allocated memory structures: Arrays, lists, and tables are often declared larger than needed. For example, an array may be allocated as 100×100 elements, even though it rarely exceeds 10×10 elements in practice.
- Seldom-used features: Certain program options and functionalities may be accessed rarely. A humorous example given is a U.S. government budget-balancing routine, which has not been used in years.

Benefits of executing programs that are only partially in memory

- Virtual address space expansion: Programs are no longer constrained by physical memory, allowing for large address spaces and simplifying programming.
- Higher CPU utilization & throughput: With less physical memory required per program, more programs can run concurrently without increasing response time.
- Reduced I/O overhead: Less swapping and loading means programs run faster overall.

Chapter 9

9.4 logical address space of 64 pages of 1024 words each, mapped onto physical memory of 32 frames.

Logical address 

p: pages, 64 pages = 2^6

d: 1024 words, $2^{10} = 1024$

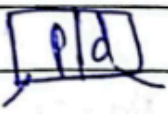
Logical address size = 6 (page bits) + 10 (offset bits)
= 16 bits

Physical address 

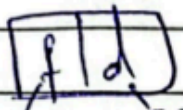
32 frames = 2^5

offset within a frame :-
Each frame holds 1024 words
$2^{10} = 1024$

Physical address size = 5 (frame bits) + 10 (offset bits)
= 15 bits



# of pages — size of page



# of frames — size of frame

9.7 Page size = 1-KB. Addresses provided.

a) 3085

1 KB = 2^{10} = Page size

| p | d |

$2^x + 2^{10} = 3085$

Page number = 3085 / 1024 = 3

Offset = 3085 mod 1024

= 3085 - (3 × 1024) = 13

Page = 3, offset = 13

b) 42095

Page number = 42095 / 1024 = 41

offset = 42095 mod 1024

= 111

Page = 41, offset = 111

c) 215201

Page number = 215201 / 1024 = 210

Offset = 215201 mod 1024

= 161

9.7 logical address space of 256 pages
4KB page size,
physical memory of 64 frames.

a.

| p | d |

256 = 28 = 8 bits

4KB = $2^2 \times 2^{10} = 2^{2+10} = 2^{12}$ = 12 bits

bits required in logical address :-

= 8 + 12 = **20 bits**

b.

| f | d |

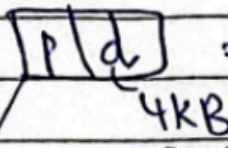
64 = 2^6

4KB = $2^2 \times 2^{10} = 2^{2+10} = 2^{12}$

12 + 6 = **18 bits** required in Physical address size.

9.10 4KB page size, 32-bit logical address. 512 MB of physical memory.

a. A conventional, single-level page table.



$= 2^{32}$

$4KB = 2^{10} \times 2^2 = 2^{10+2} = 2^{12}$

Pages $= 2^{32-12} = 2^{20}$

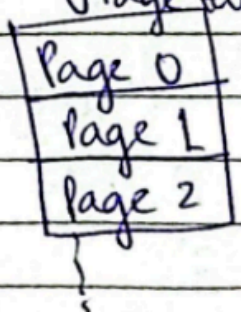
Number of pages $= 2^{20}$

The page table must have 2^{20} enteries.

In conventional, single level page table.

# of pages = # of enteries in page table.

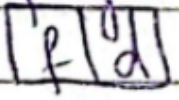
Indexing on the basis of pages.



b. Inverted page table.

Indexing on the basis of frames.

An inverted page table has one entry for each physical frame.



512 MB $= 2^9 \times 2^{20} \times 2^{10} = 2^{10+10+9} = 2^{29}$

$512 = 2^9$, $1MB = 2^{20}$ bytes

512 MB = 2^{29}

Page size = 4KB = 2^{12}

f	d
	12

= 29

2^{29-12} = 2^{17} frames

2^{17} = 131,072 frames. So, inverted page table has 131,072 entries.

9.8 21-bit virtual address, 16-bit physical address. 2-KB page size.

a. A conventional, single-level page table.

p	d

= 21

Page size = 2KB = $2^1 \times 2^{10}$

= 2^{11}

2^{21-11} = 2^{10} pages

Page table will have 2^{10} = 1024 entries.

b. Inverted page table.

f	d

= 16

2^5 = 32

Page size = 2KB = 2^{11}

2^{16-11} = 2^5 entries

2^5 = 32 entries in the inverted page table.

- Maximum amount of physical memory = 2^{16}

9.2 Effect of allowing two entries in a page table to point to the same page frame in memory?

- Allows memory sharing between processes or within the same process.
- Used to save memory by sharing common code (e.g. text editors, compilers) and data.
- optimizes large memory copies.
  - Instead of physically copying the data, the OS maps new virtual pages to existing physical frames (fast, constant time).
  - Technique is called Copy-On-Write (COW).
- On write access, OS creates a separate physical copy only when needed to maintain data integrity.
- Without COW, any update one one page is reflected in the other, since both point to the same physical memory.
- Read-only sharing is used to avoid accidental modifications when sharing memory.

9.3 Advantages & disadvantages of code & data separation.

Advantages:

- Code & data separation allows for better

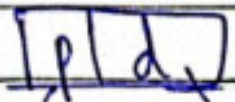
protection. Instruction segments is read-only by design - prevents accidental or malicious code modification.

- Code sharing is easier.
  - A single copy of program (e.g. editor or compiler) can be shared across multiple users / processes.
  - Saves memory & improves efficiency.
- Improved security & stability.
  - Read-only instructions segments reduce risk of code corruption.

Disadvantages :-

- Increased hardware complexity.
  - Requires two base-limit register pairs & logic to distinguish between instruction fetches & data accesses.
- Limits flexibility
  - Some advanced technique may be harder or impossible to implement.
- Fragmentation risks
  - Code & data must be managed in separate segments, which may lead to inefficient memory usage.

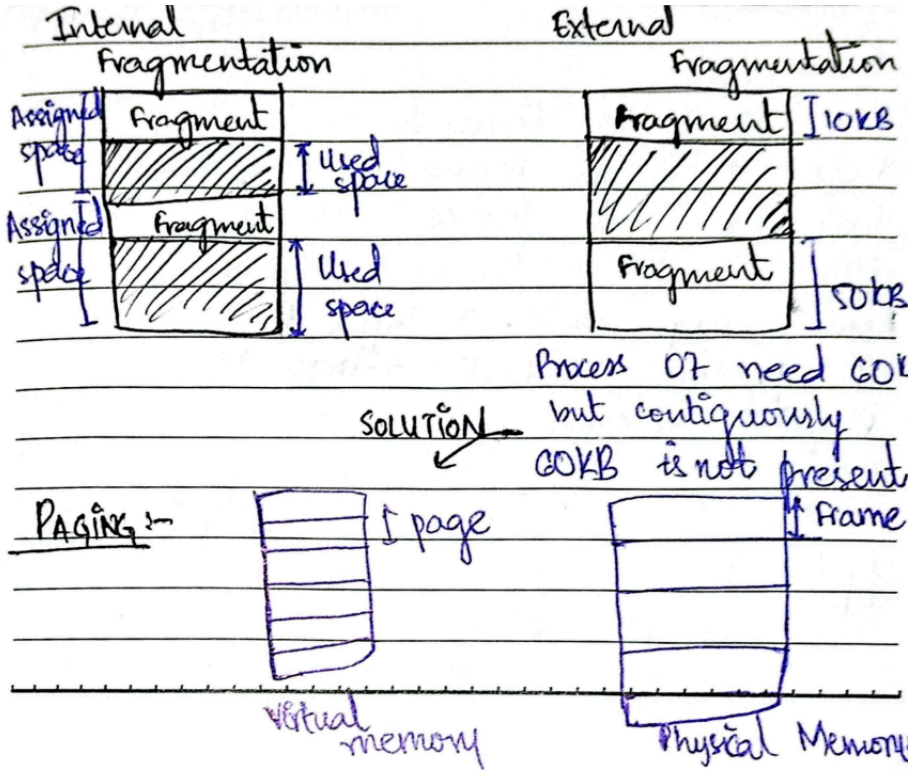
Q.2 Why are page sizes in powers of 2?



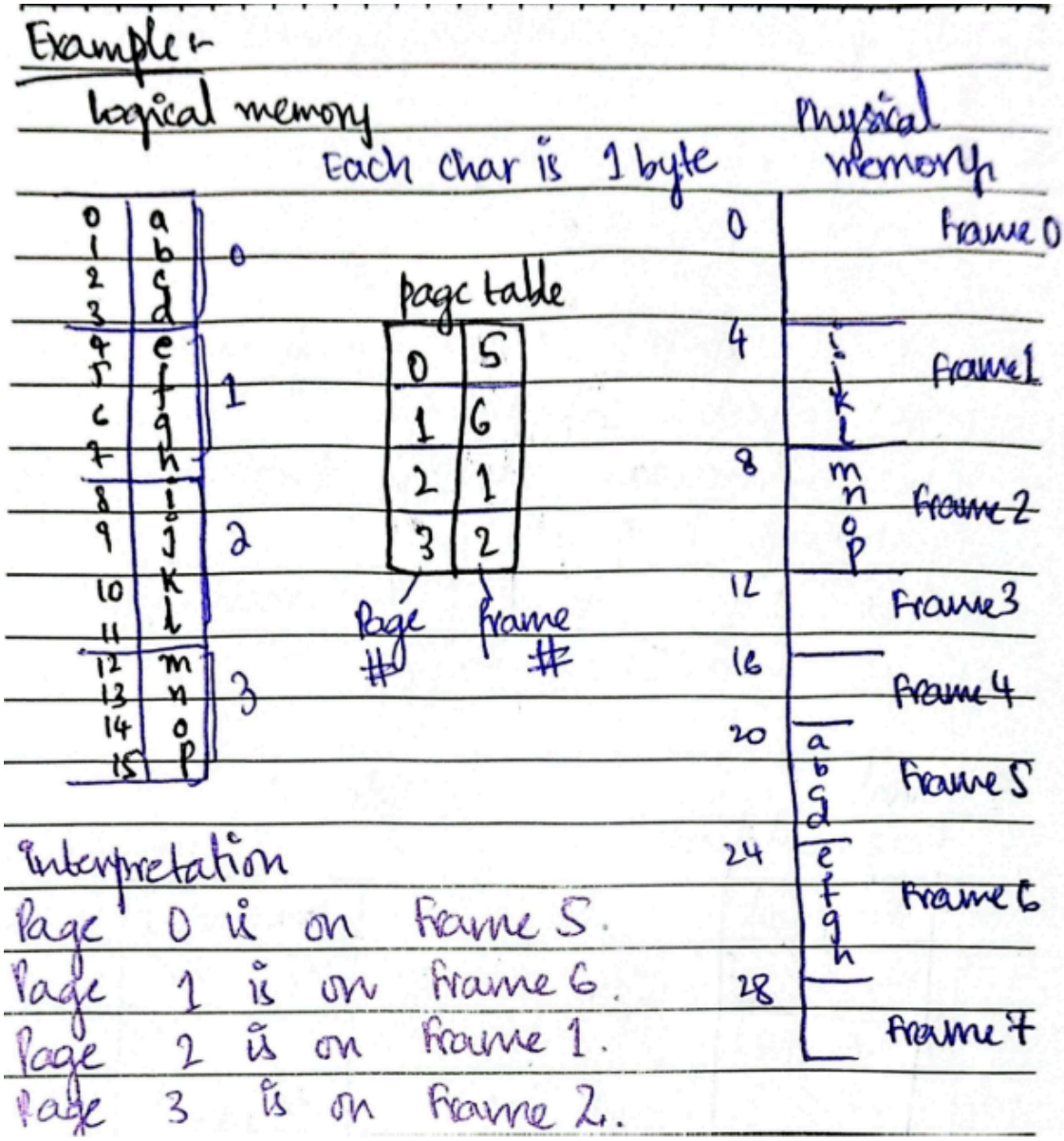
$= m = 2^m$

page size $= 2^n$

- powers of simplifies the page number & page offset calculations.
- Makes hardware design for MMU more straightforward & faster.
- Pages align naturally in memory without additional logic.



Example:-



Interpretation

Page 0 is on Frame 5.

Page 1 is on Frame 6.

Page 2 is on Frame 1.

Page 3 is on Frame 2.