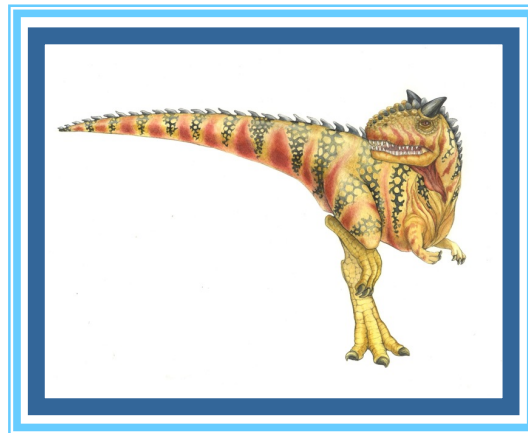


Chapter 9: Memory Management





Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Fragmentation
- Segmentation
- Paging
- Translation Lookaside Buffers
- Structure of the Page Table
- Implementation of the Page Table





Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation





Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation





Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program





Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic
 - Compiled code addresses **bind** to relocatable addresses
 - ▶ i.e. "14 bytes from beginning of this module"
 - Linker or loader will bind relocatable addresses to absolute addresses
 - ▶ i.e. 74014
 - Each binding maps one address space to another





Binding of Instructions and Data to Memory

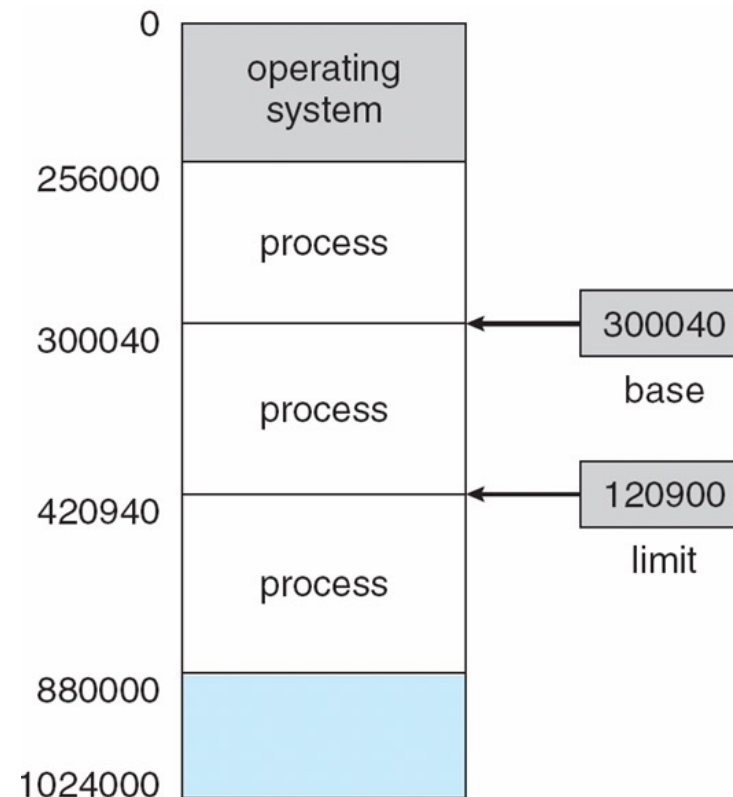
- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - ▶ Need hardware support for address maps (e.g., base and limit registers)





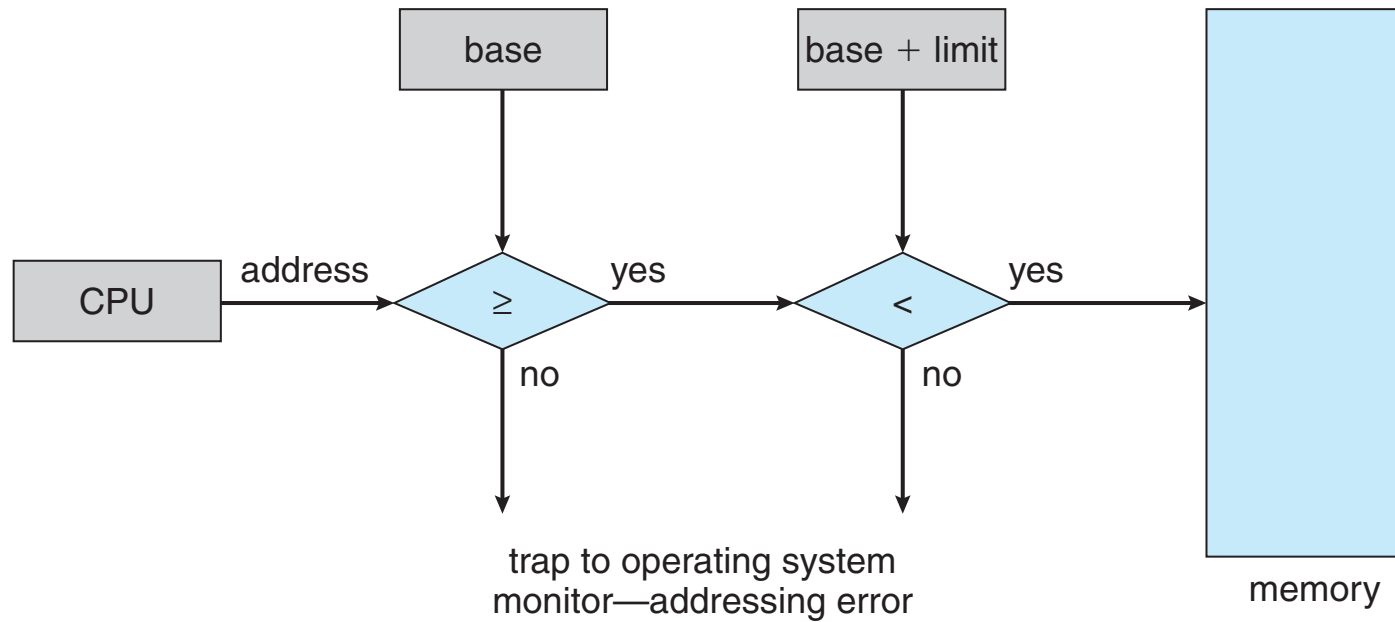
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user





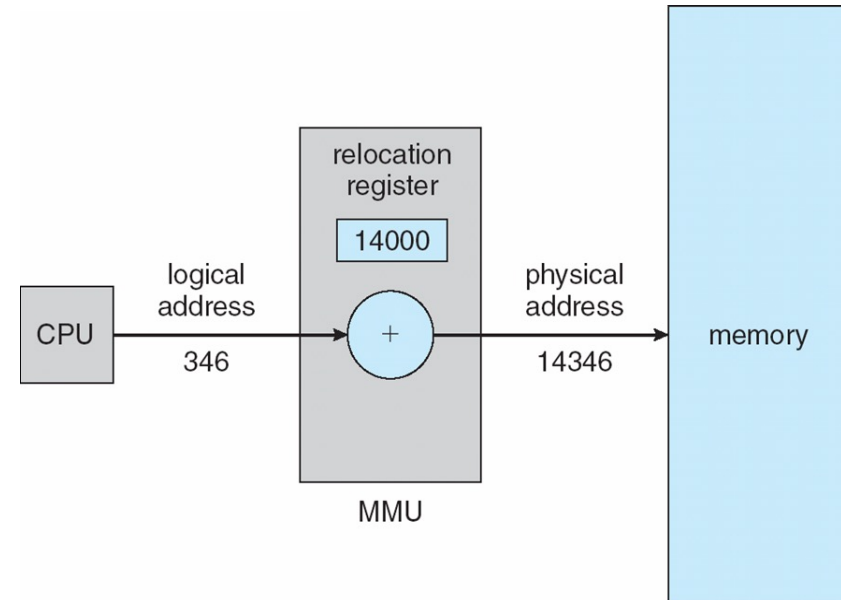
Hardware Address Protection





Dynamic Relocation Using a Relocation Register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading





Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory





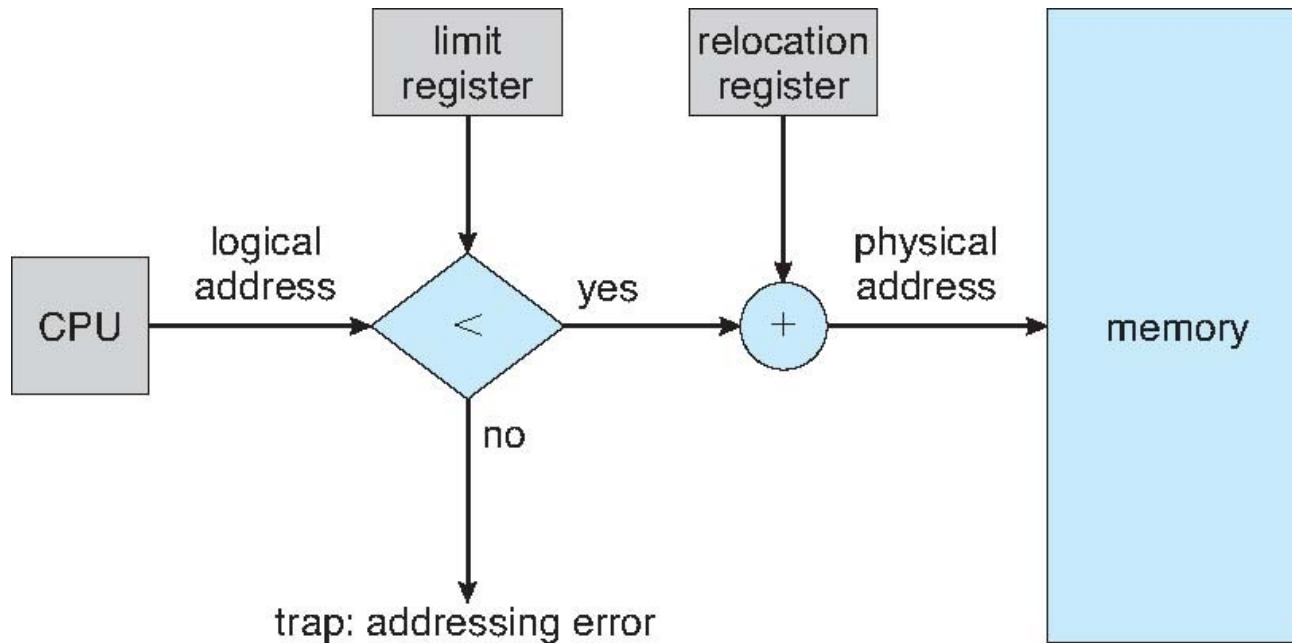
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size





Hardware Support for Relocation and Limit Registers

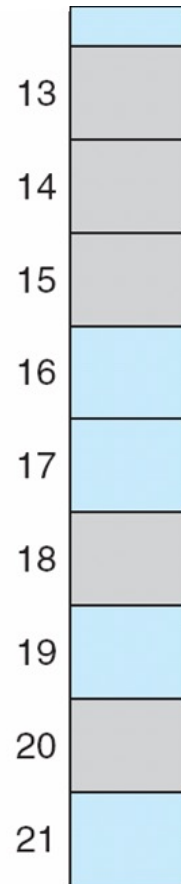
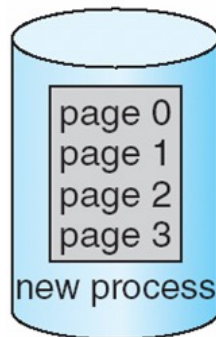




Free Frames

free-frame list

14
13
18
20
15

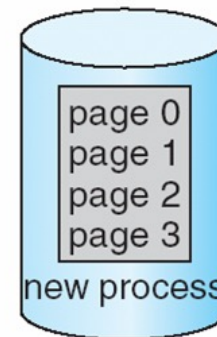


(a)

Before allocation

free-frame list

15

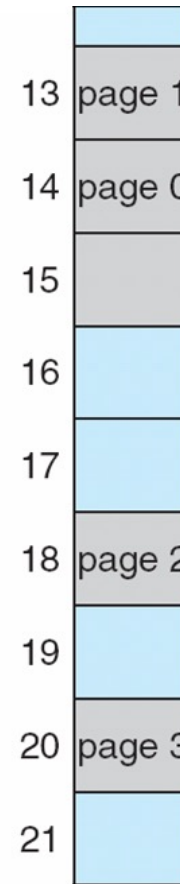


0	14
1	13
2	18
3	20

new-process page table

(b)

After allocation

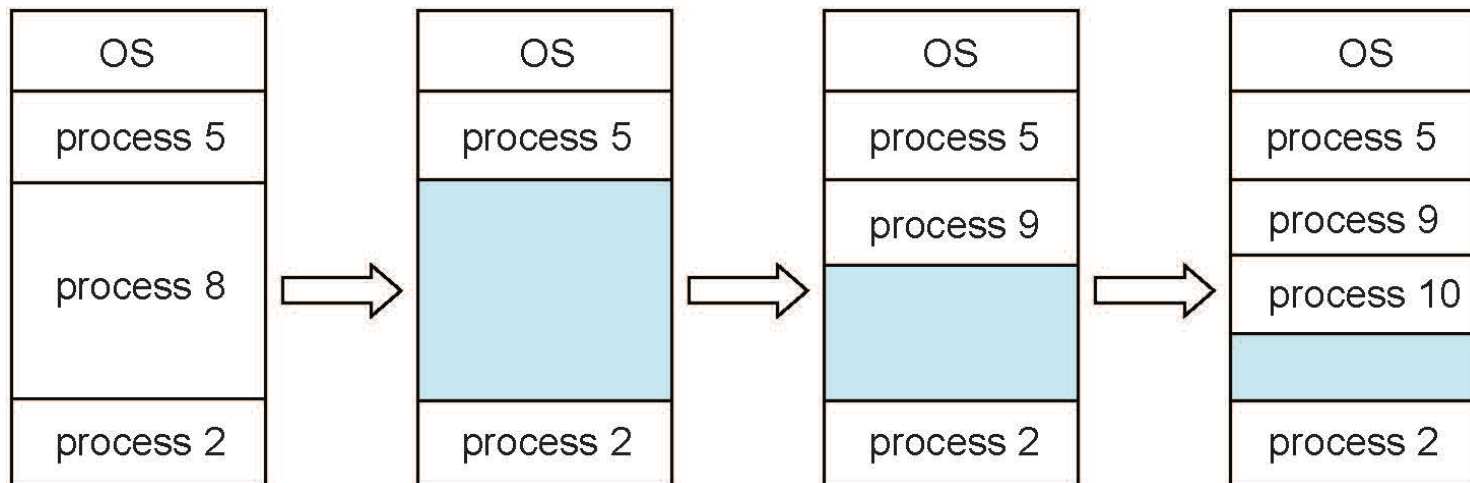




Multiple-partition allocation

■ Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
a) allocated partitions b) free partitions (hole)





Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
 - Produces the largest leftover hole

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





Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - $1/3$ may be unusable -> **50-percent rule**





Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - I/O problem
 - ▶ Latch job in memory while it is involved in I/O
 - ▶ Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems





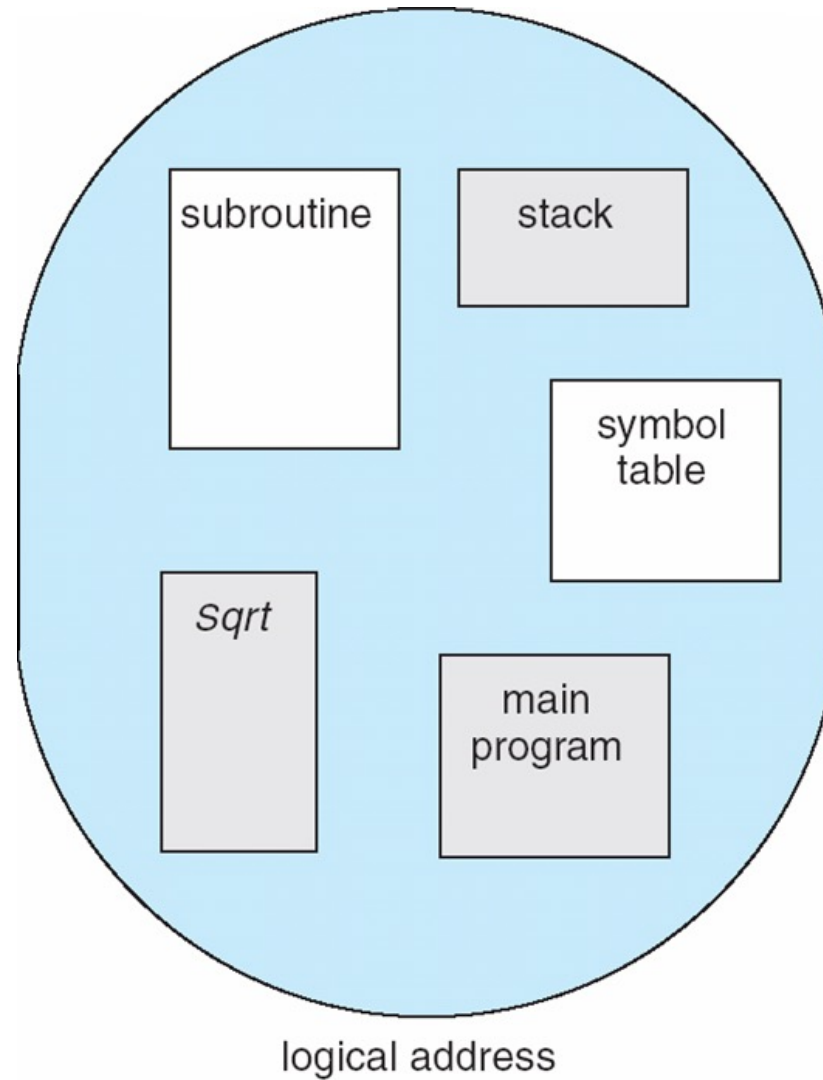
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays



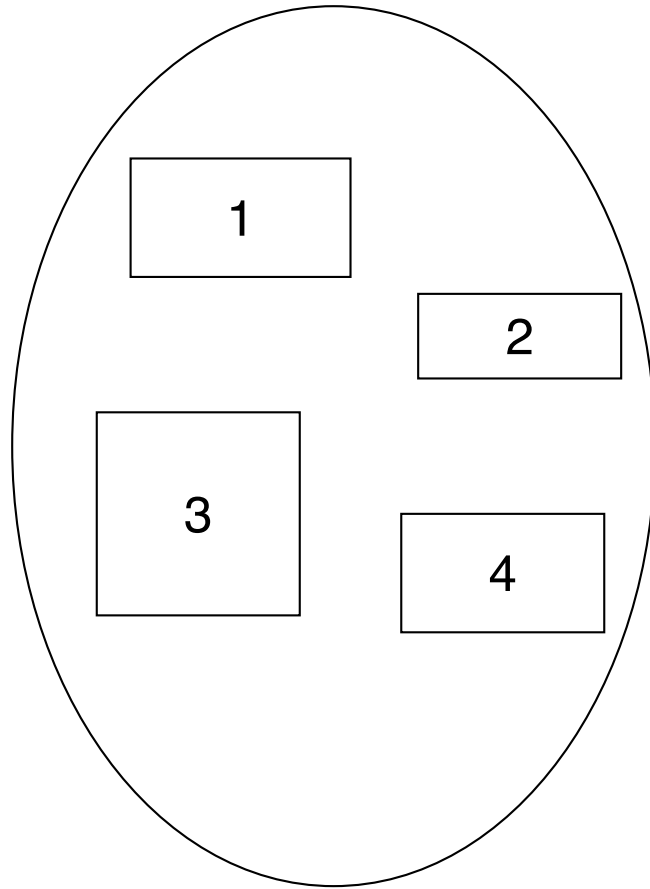


User's View of a Program

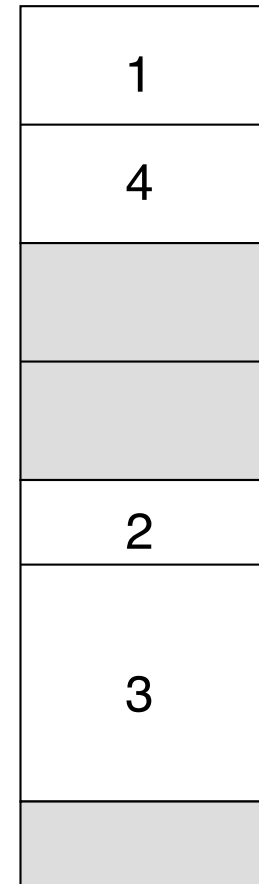




Logical View of Segmentation



user space



physical memory space





Segmentation Architecture

- Logical address consists of a two tuple:
 <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - **base** – contains the starting physical address where the segments reside in memory
 - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
 segment number **s** is legal if **s** < **STLR**





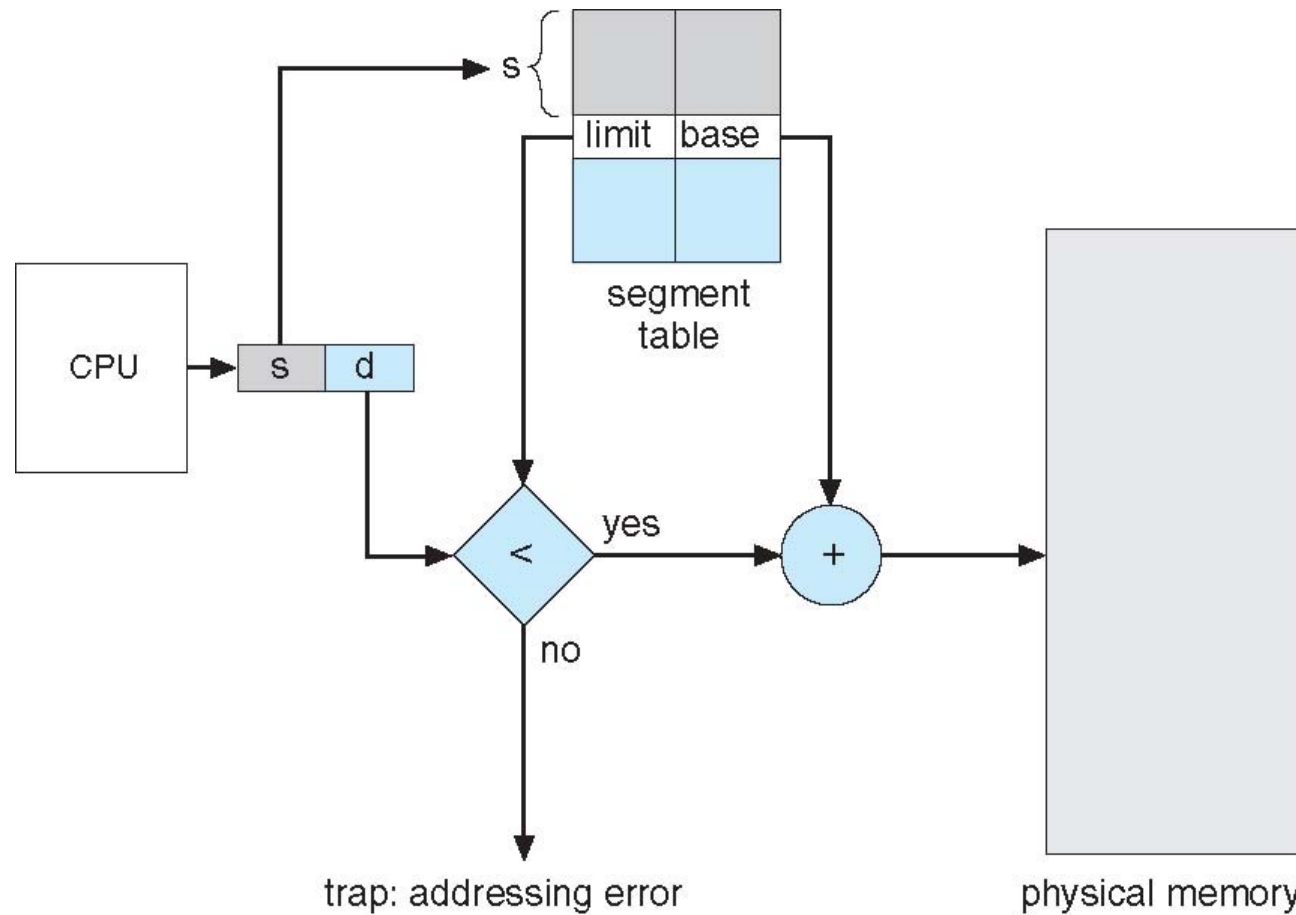
Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - ▶ validation bit = 0 \Rightarrow illegal segment
 - ▶ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram





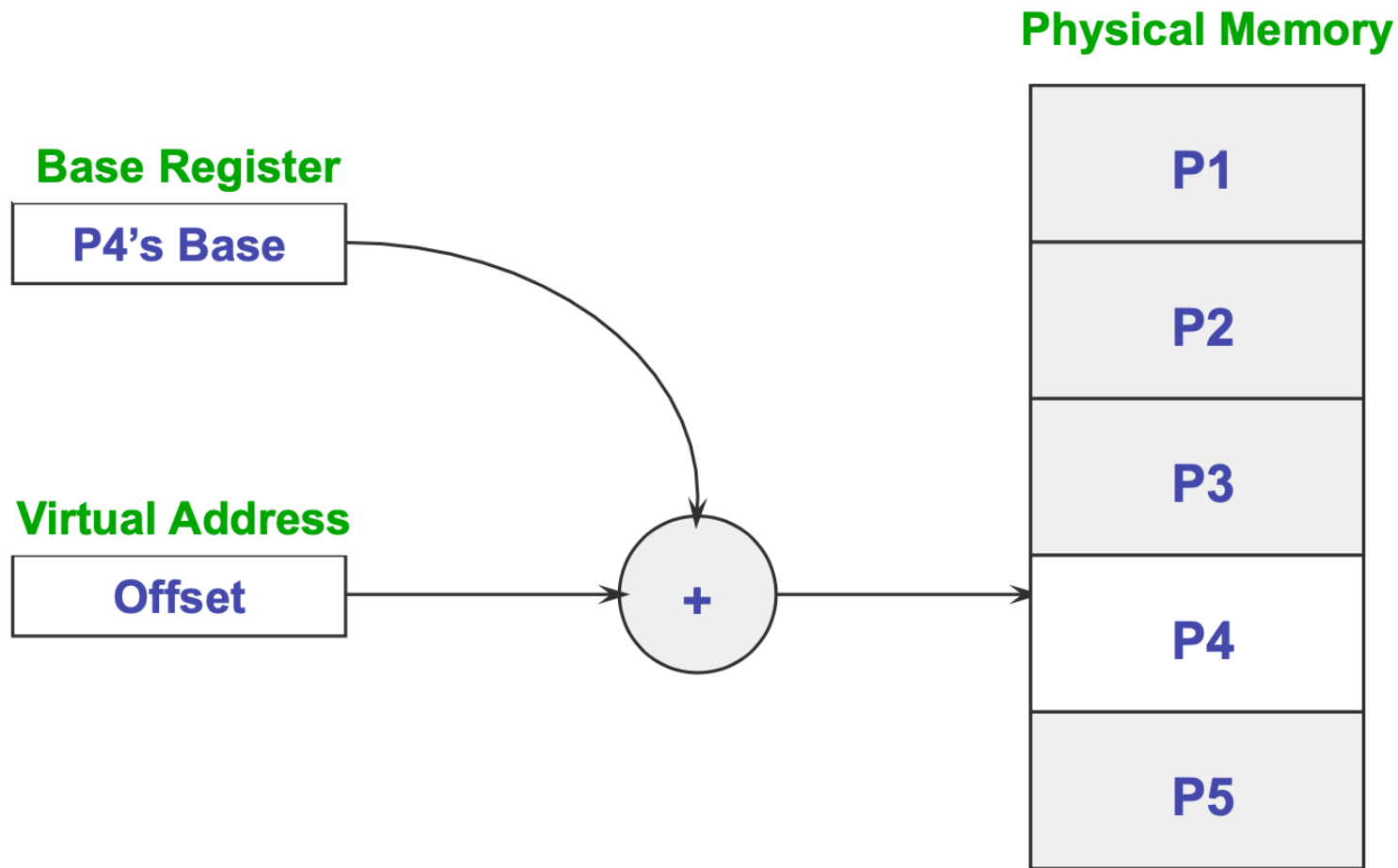
Segmentation Hardware





Fixed Partitions (I)

■ Fix Partitions





Fixed Partitions (II)

- Physical memory is broken up into fixed partitions
 - Hardware requirements: base register
 - Physical address = virtual address + base register
 - Base register loaded by OS when it switches to a process
 - Size of each partition is the same and fixed
- Advantages
 - Easy to implement, fast context switch
- Problems
 - Internal fragmentation: memory in a partition not used by a process is not available to other processes
 - Partition size: one size does not fit all (very large processes?)





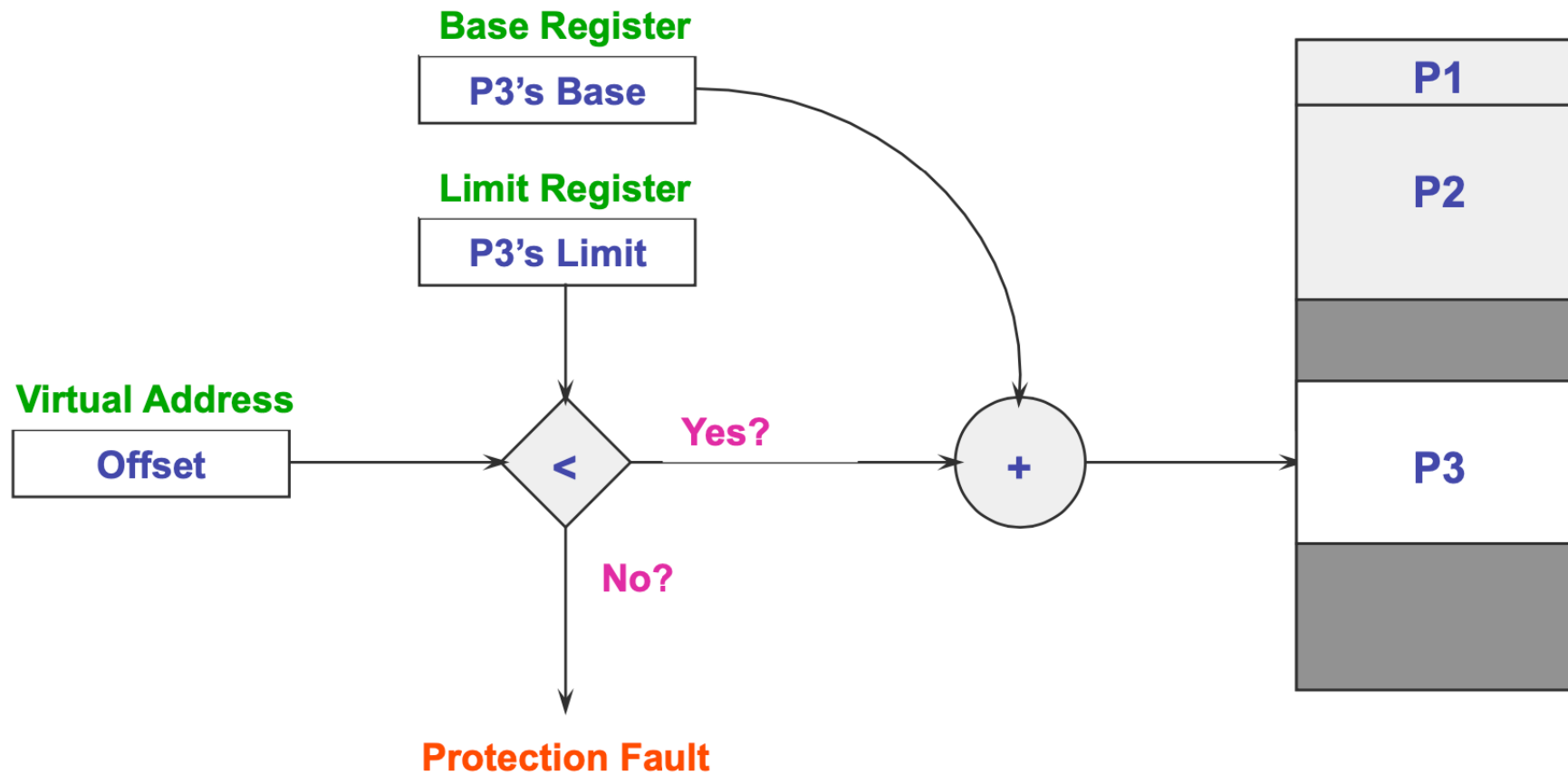
Variable Partitions (I)

- Natural extension -- physical memory is broken up into variable sized partitions
 - Hardware requirements: base register and limit register
 - Physical address = virtual address + base register
 - Why do we need the limit register? Protection
 - If (physical address > base + limit) then exception fault
- Advantages
 - No internal fragmentation: allocate just enough for process
- Problems
 - External fragmentation: job loading and unloading produces empty holes scattered throughout memory





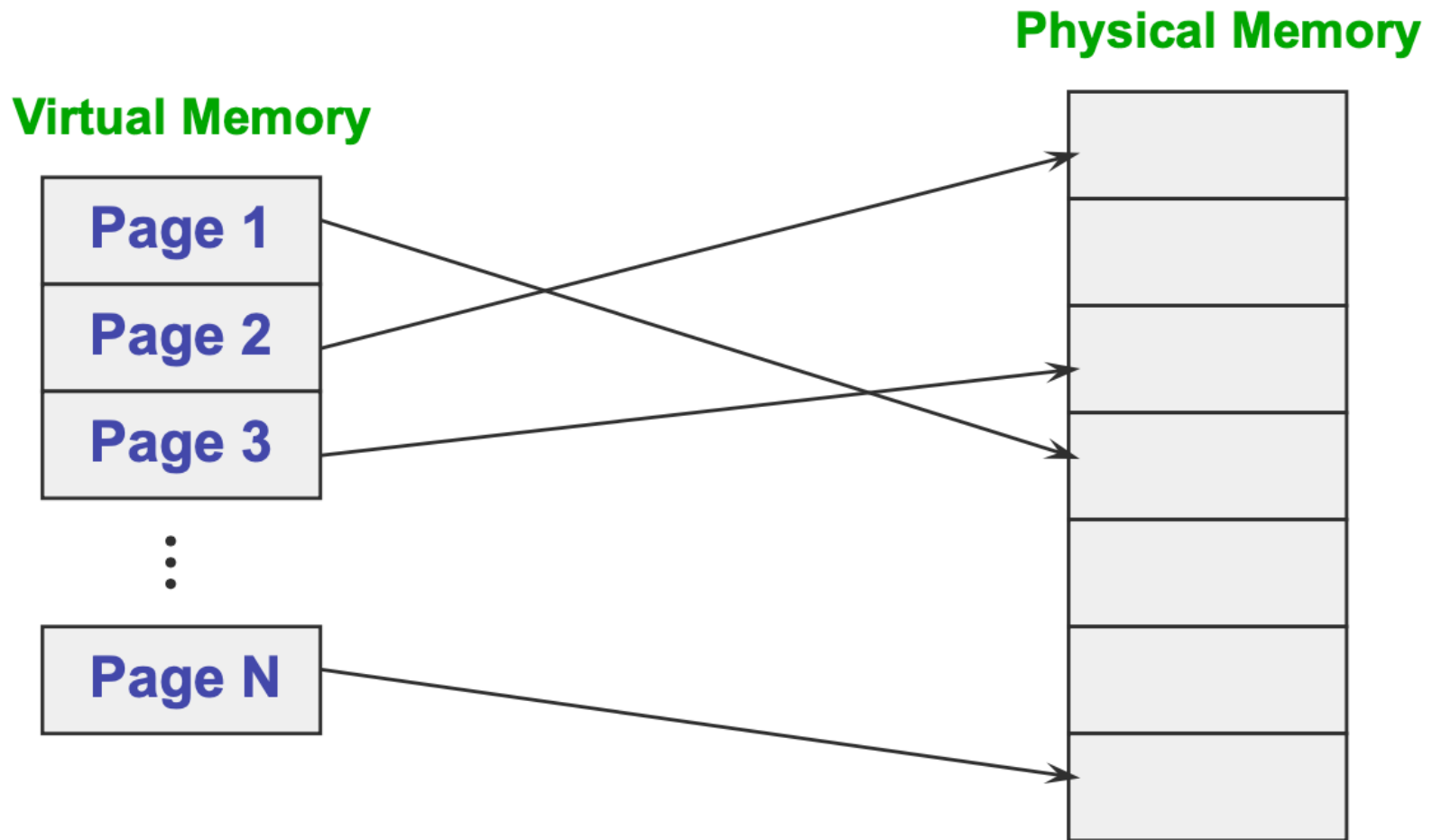
Variable Partitions (II)





Paging

- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory





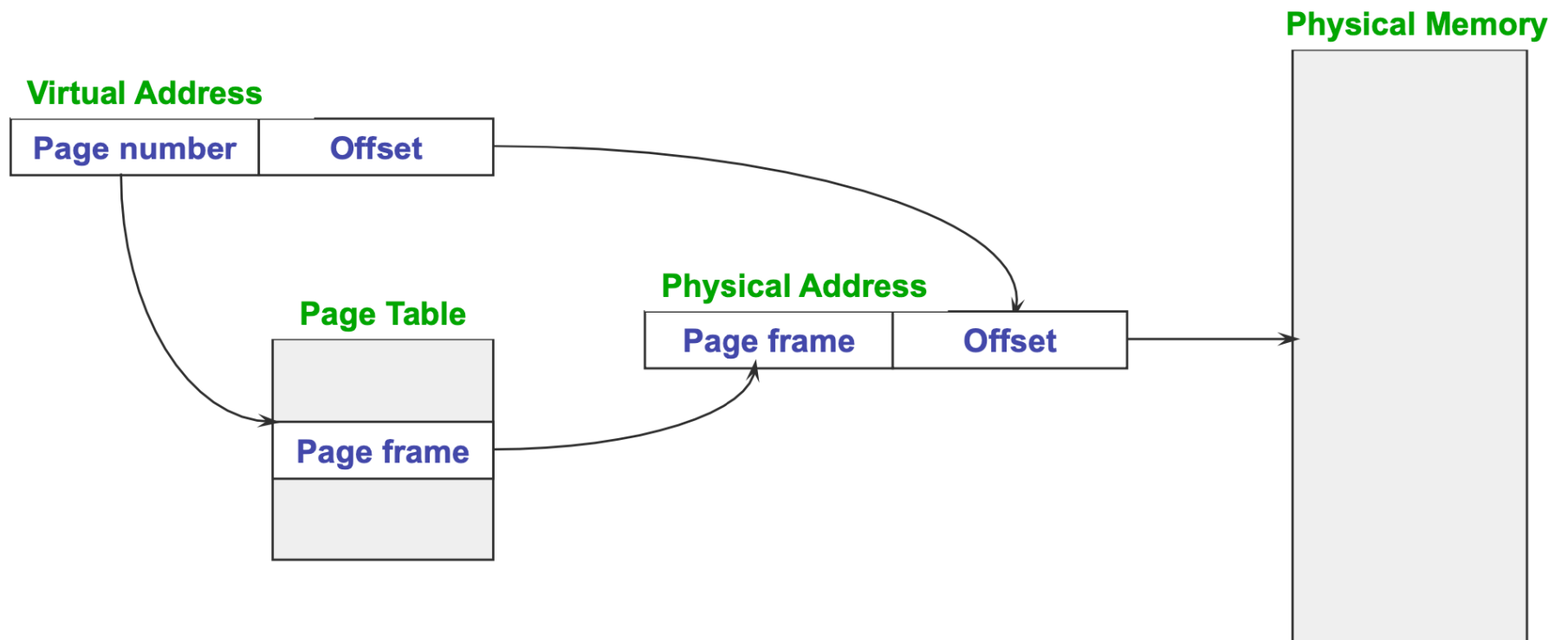
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation





Page Lookups





Address Translation Scheme

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

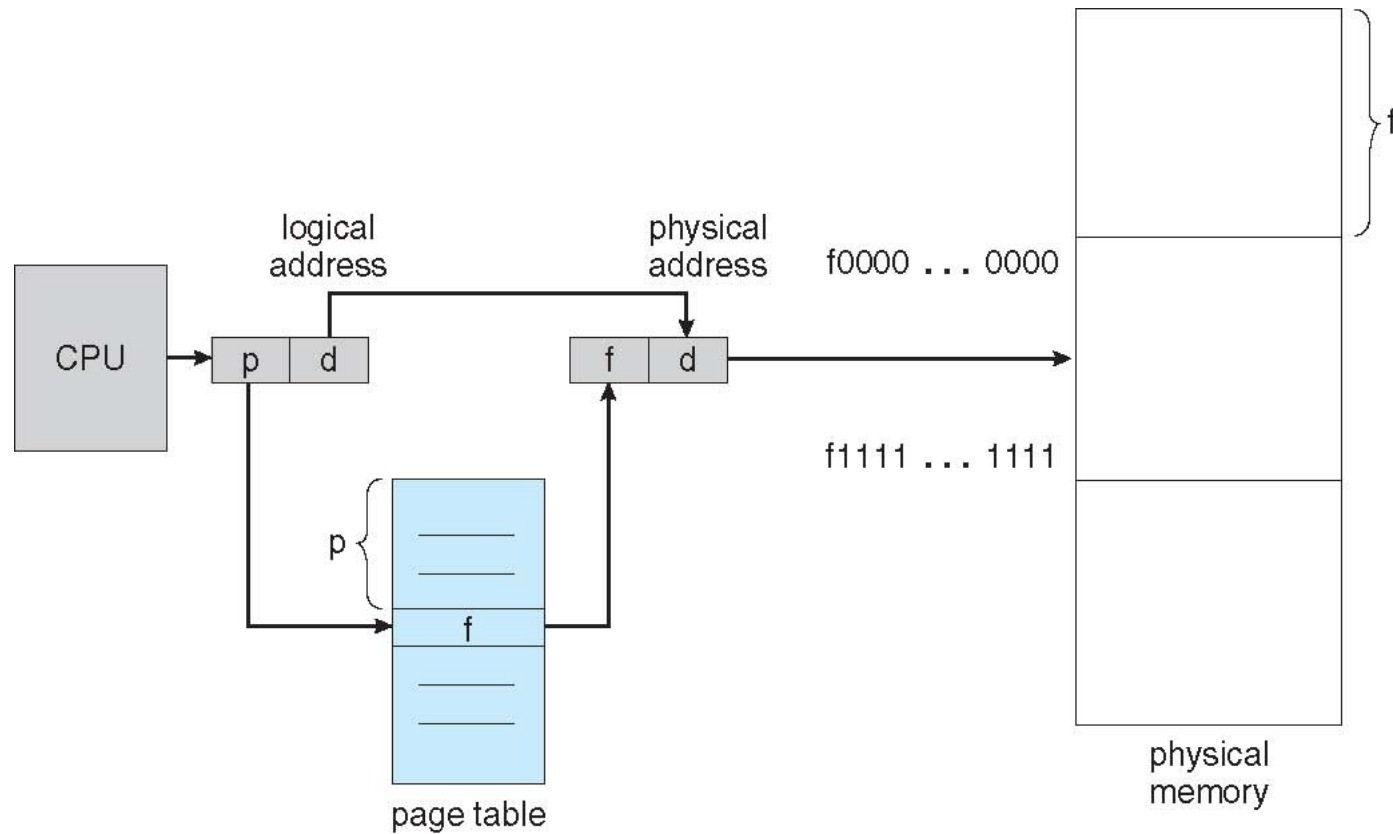
page number	page offset
p	d
$m - n$	n

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



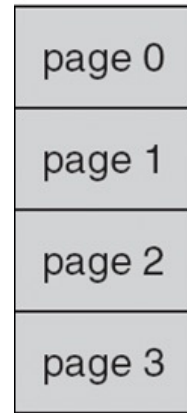


Paging Hardware





Paging Model of Logical and Physical Memory

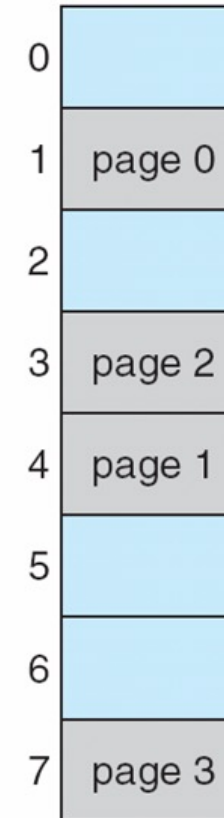


logical
memory

0	1
1	4
2	3
3	7

page table

frame
number

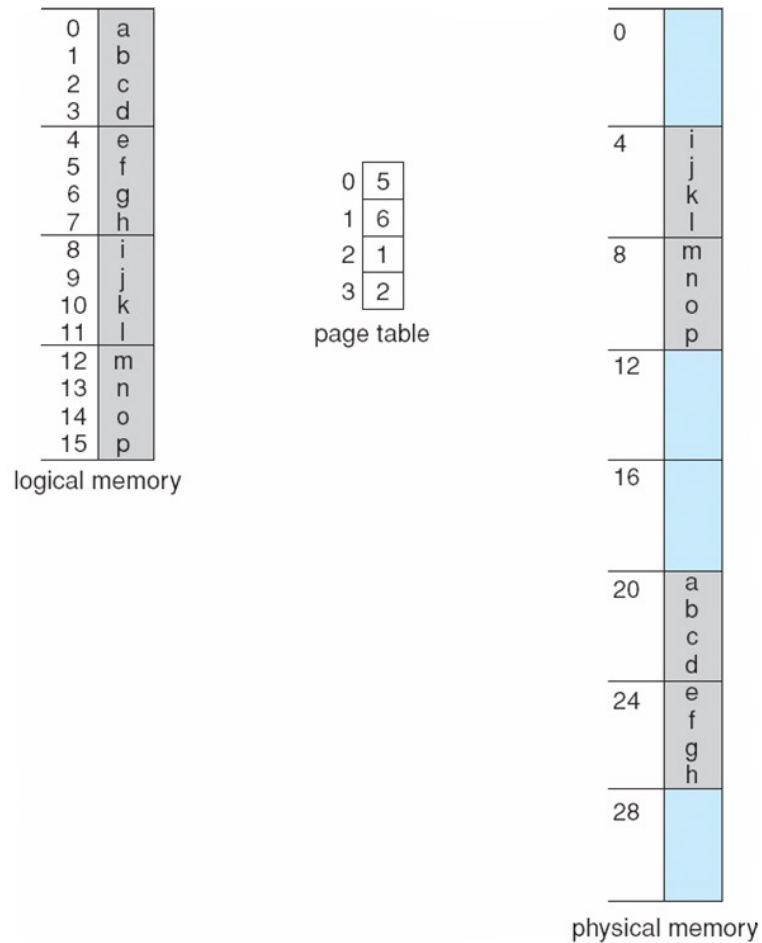


physical
memory





Paging Example



32-byte memory and 4-byte pages





Paging (Cont.)

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





Paging Advantages

- Easy to allocate memory
 - Memory comes from a free list of fixed size chunks
 - Allocating a page is just removing it from the list
 - External fragmentation not a problem
- Easy to swap out chunks of a program
 - All chunks are the same size
 - Pages are a convenient multiple of the disk block size
 - How do we know if a page is in memory or not?





Paging Limitations

- Can still have internal fragmentation
 - Process may not use memory in multiples of a page
- Memory reference overhead
 - 2 references per address lookup (page table, then memory)
 - Solution – use a hardware cache of lookups (more later)
- Memory required to hold page table can be significant
 - Need one PTE per page
 - 32 bit address space w/ 4KB pages = 2^{20} PTEs
 - 4 bytes/PTE = 4MB/page table
 - 25 processes = 100MB just for page tables!
 - Solution – page the page tables





Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**





Page Table Entries (PTE)

1	1	1	2	20
M	R	V	Prot	Page Frame Number

- Page table entries control mapping
 - The Modify bit says whether or not the page has been written » It is set when a write to the page occurs
 - The Reference bit says whether the page has been accessed » It is set when a read or write to the page occurs
 - The Valid bit says whether or not the PTE can be used » It is checked each time the virtual address is used
 - The Protection bits say what operations are allowed on page » Read, write, execute
 - The page frame number (PFN) determines physical page





Implementation of Page Table (Cont.)

- 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
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered





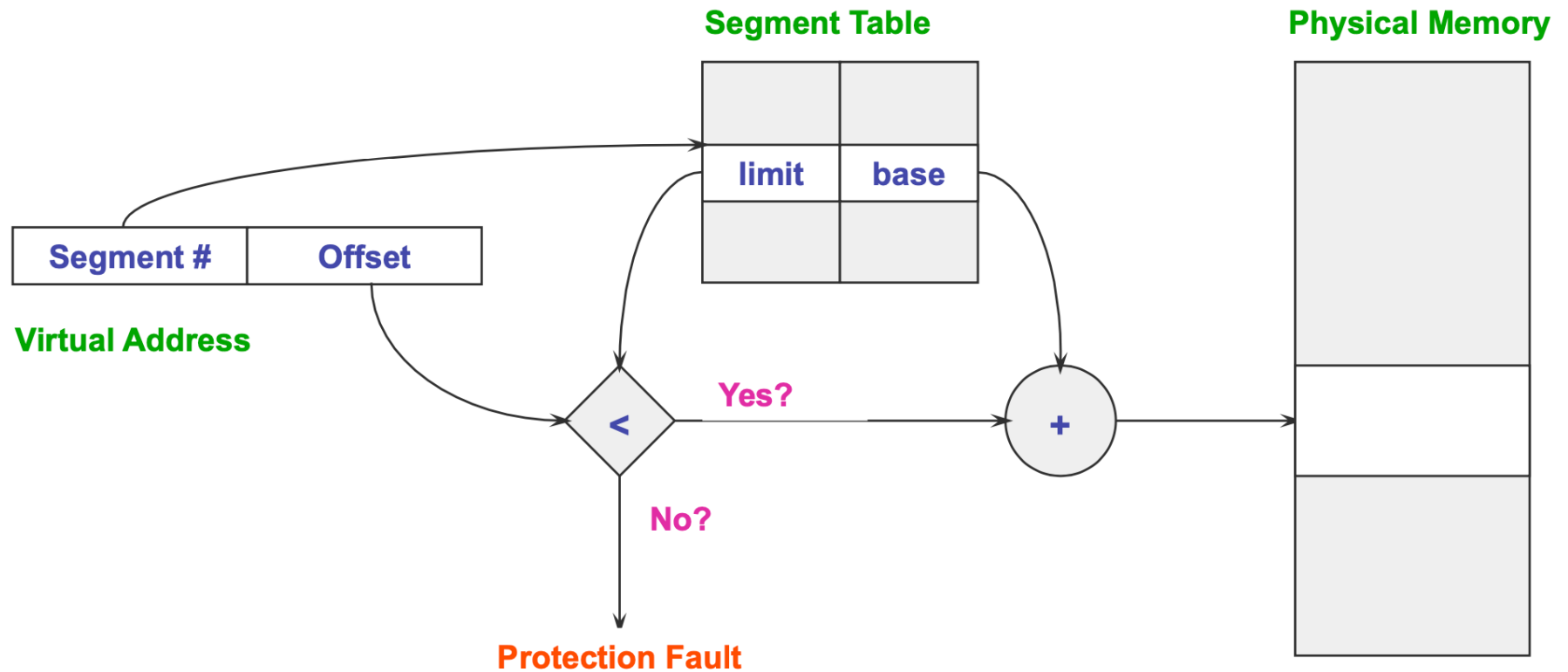
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
 - Module, procedure, stack, data, file, etc.
 - Virtual addresses become <segment #, offset>
 - Units of memory from user's perspective
- Natural extension of variable-sized partitions
 - Variable-sized partitions = 1 segment/process
 - Segmentation = many segments/process
- Hardware support
 - Multiple base/limit pairs, one per segment (segment table)
 - Segments named by #, used to index into table





Segment Lookup





Segmentation and Paging

- Can combine segmentation and paging
 - The x86 supports segments and paging
- Use segments to manage logically related units
 - Module, procedure, stack, file, data, etc.
 - Segments vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed size chunks
 - Makes segments easier to manage within physical memory
 - Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
 - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated





Summary

- Fixed partitions – easy to use, but internal fragmentation
- Variable partitions – more efficient, but external fragmentation
- Paging – use small, fixed size chunks, efficient for OS
- Segmentation – manage in chunks from user's perspective
- Combine paging and segmentation to get benefits of both





Memory Protection

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





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

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page <i>n</i>





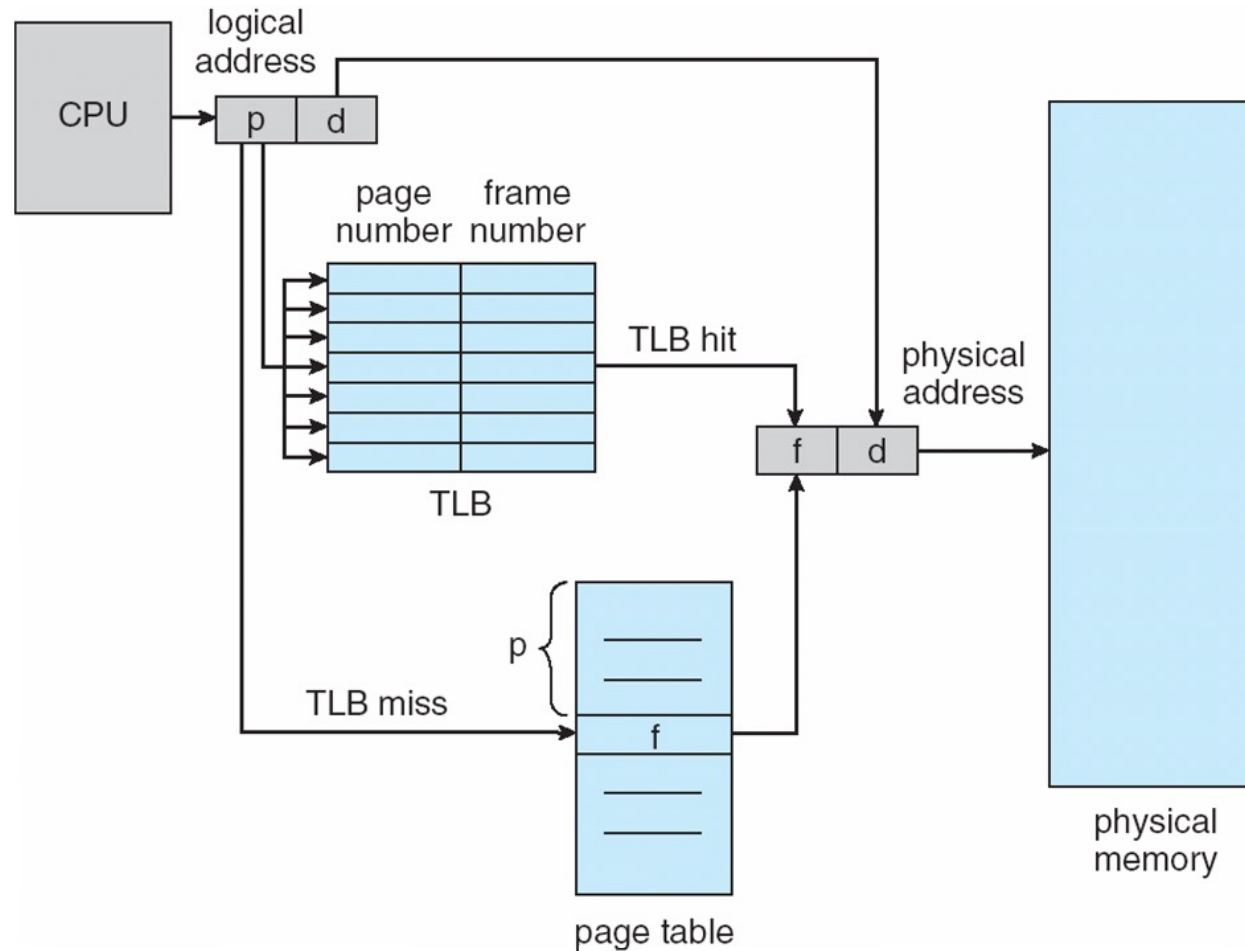
Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
 - One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost!
 - Two lookups into the page tables, a third to fetch the data
 - And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
 - Cache translations in hardware
 - Translation Lookaside Buffer (TLB)
 - TLB managed by Memory Management Unit (MMU)





Paging Hardware With TLB





Translation Lookaside Buffers (TLBs)

- TLBs
 - Translate virtual page #s into PTEs (not physical addresses)
 - Can be done in a single machine cycle
- TLBs implemented in hardware
 - Fully associative cache (all entries looked up in parallel)
 - Cache tags are virtual page numbers
 - Cache values are PTEs (entries from page tables)
 - With PTE + offset, can directly calculate physical address
- TLBs exploit locality
 - Processes only use a handful of pages at a time
 - ▶ 16-48 entries/pages (64-192K)
 - ▶ Only need those pages to be “mapped”
 - Hit rates are therefore very important





Loading TLBs

- Most address translations are handled using the TLB
 - >99% of translations, but there are misses (TLB miss)...
- Who places translations into the TLB (loads the TLB)?
 - Software loaded TLB (OS)
 - ▶ TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
 - ▶ Must be fast (but still 20-200 cycles)
 - ▶ CPU ISA has instructions for manipulating TLB
 - ▶ Tables can be in any format convenient for OS (flexible)
 - Hardware (Memory Management Unit)
 - ▶ Must know where page tables are in main memory
 - ▶ OS maintains tables, HW accesses them directly
 - ▶ Tables have to be in HW-defined format (inflexible)





Structure of the Page Table

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





Hierarchical Page Tables

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





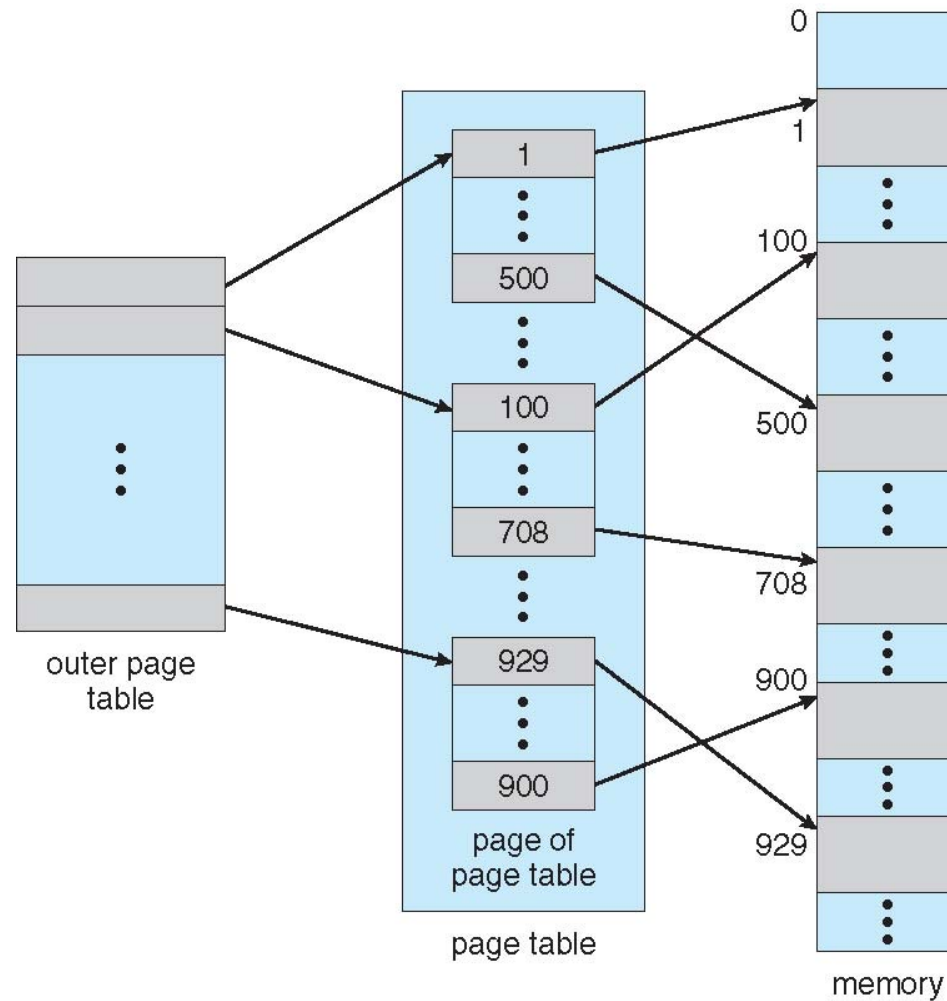
Two-Level Page Tables

- Originally, virtual addresses (VAs) had two parts
 - Page number (which mapped to frame) and an offset
- Now VAs have three parts:
 - Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
 - We'd like a manageable master page size
- Secondary table maps page number to physical page
 - Determines which physical frame the address resides in
- Offset indicates which byte in physical page
 - Final system page/frame size is still the same, so offset length stays the same





Two-Level Page-Table Scheme





Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
 - a page number consisting of 22 bits
 - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
 - a 12-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

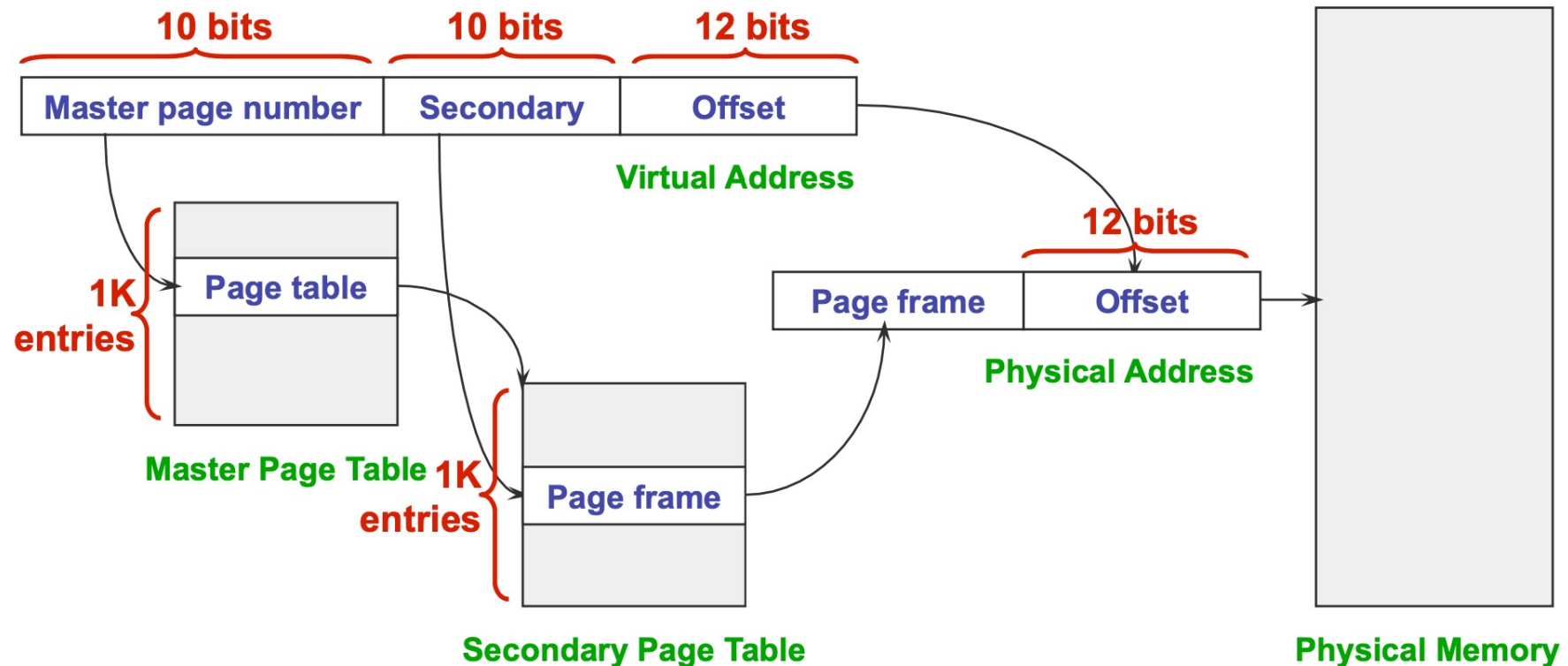
page number		page offset
p_1	p_2	d
12	10	10

- 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**





An Example w/ 4 byte PTEs

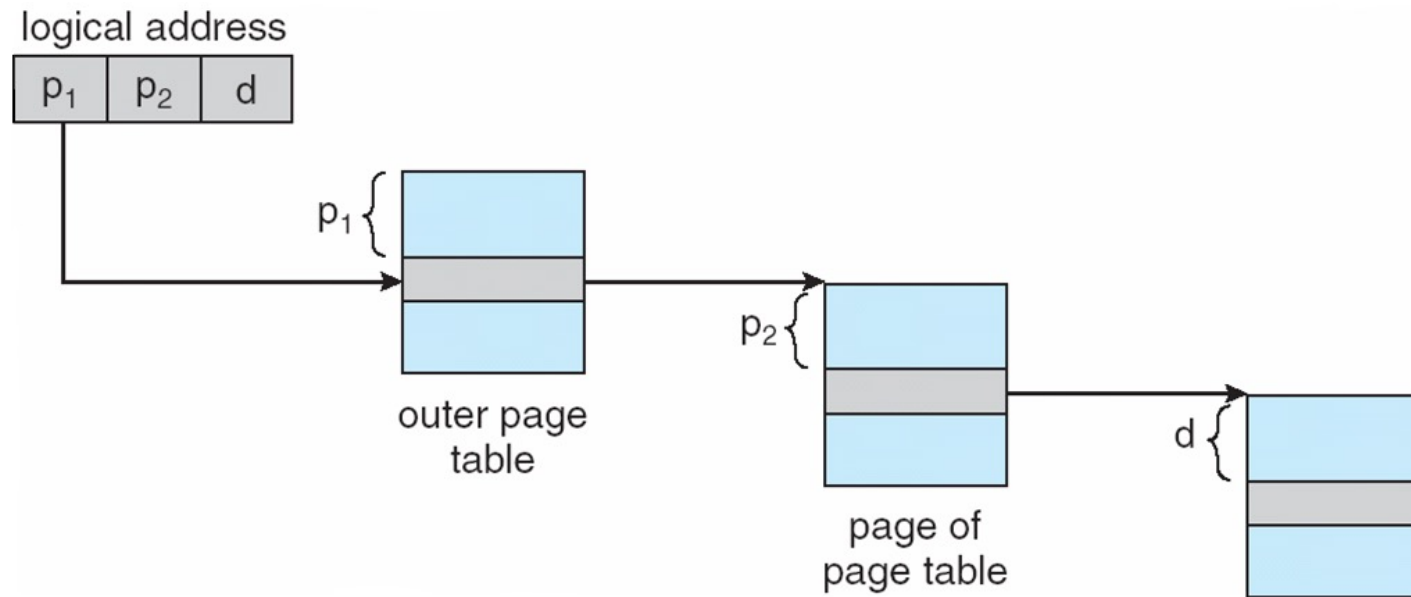


4K pages = 12-bit offset; 1 master page = $4K/4 = 1K$ entries = 10bits
Secondary page table size = $32 - 12 - 10 = 10$ bits = $1K$ entries * 4 = 4K





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	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





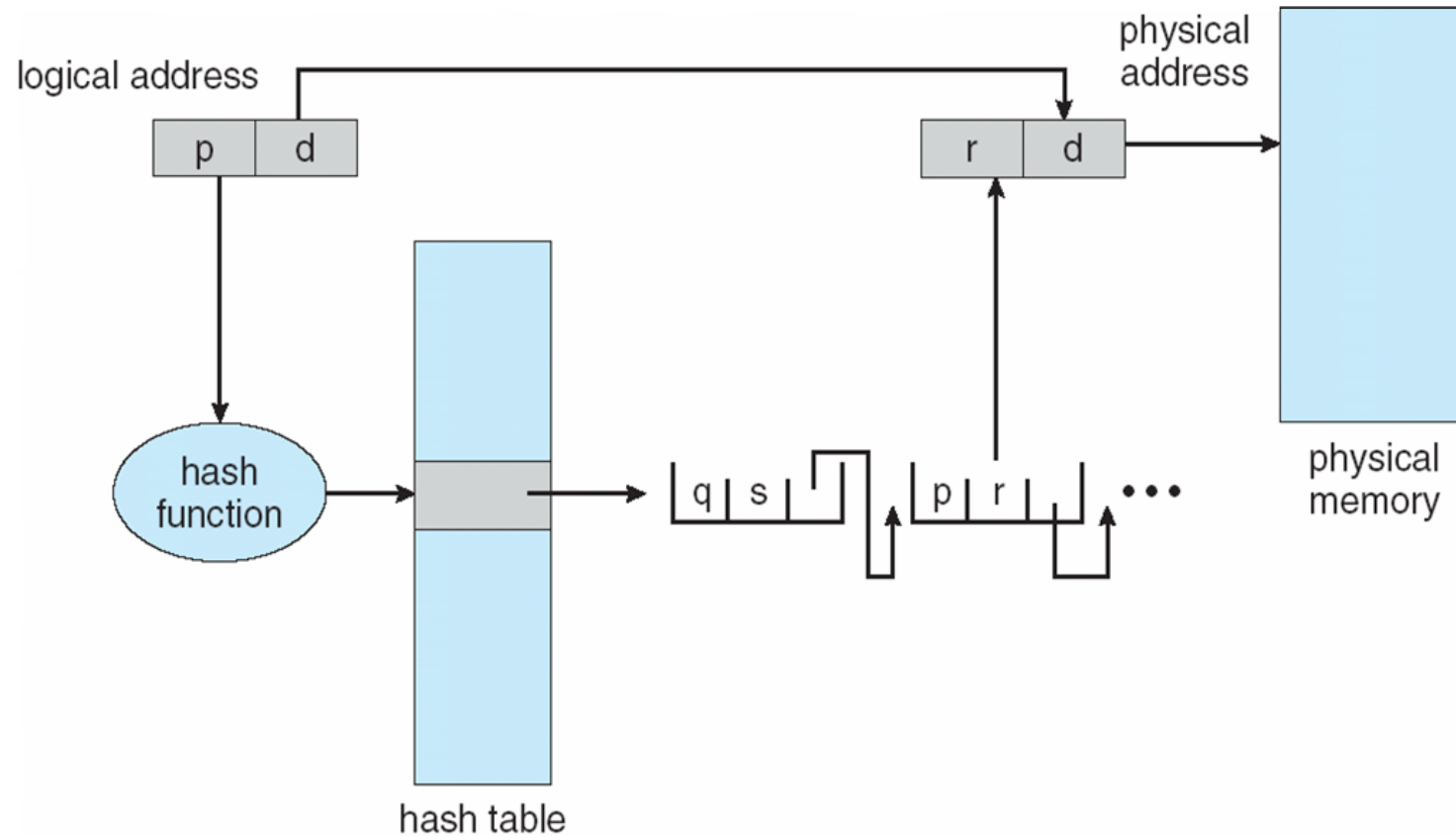
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)





Hashed Page Table





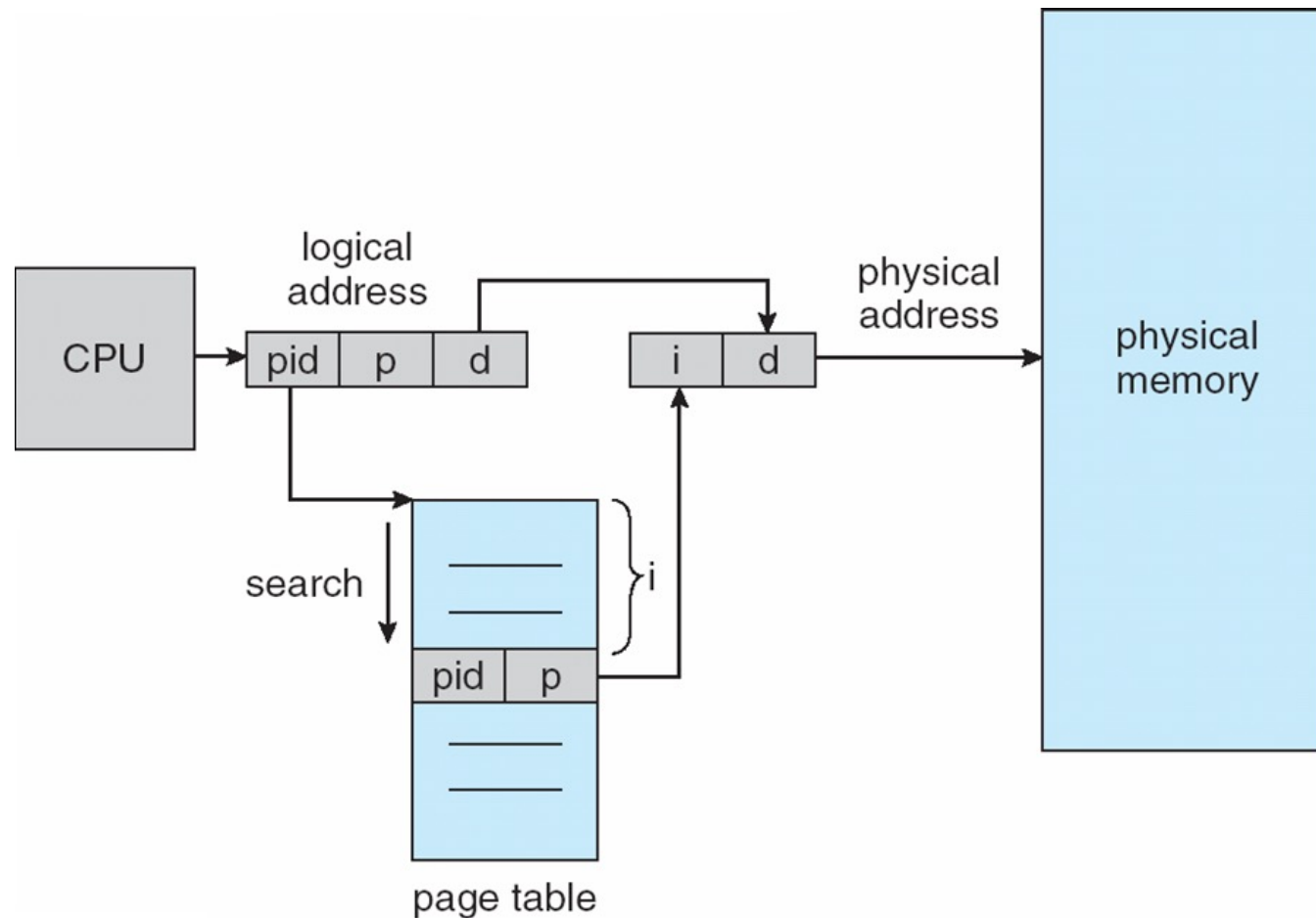
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address





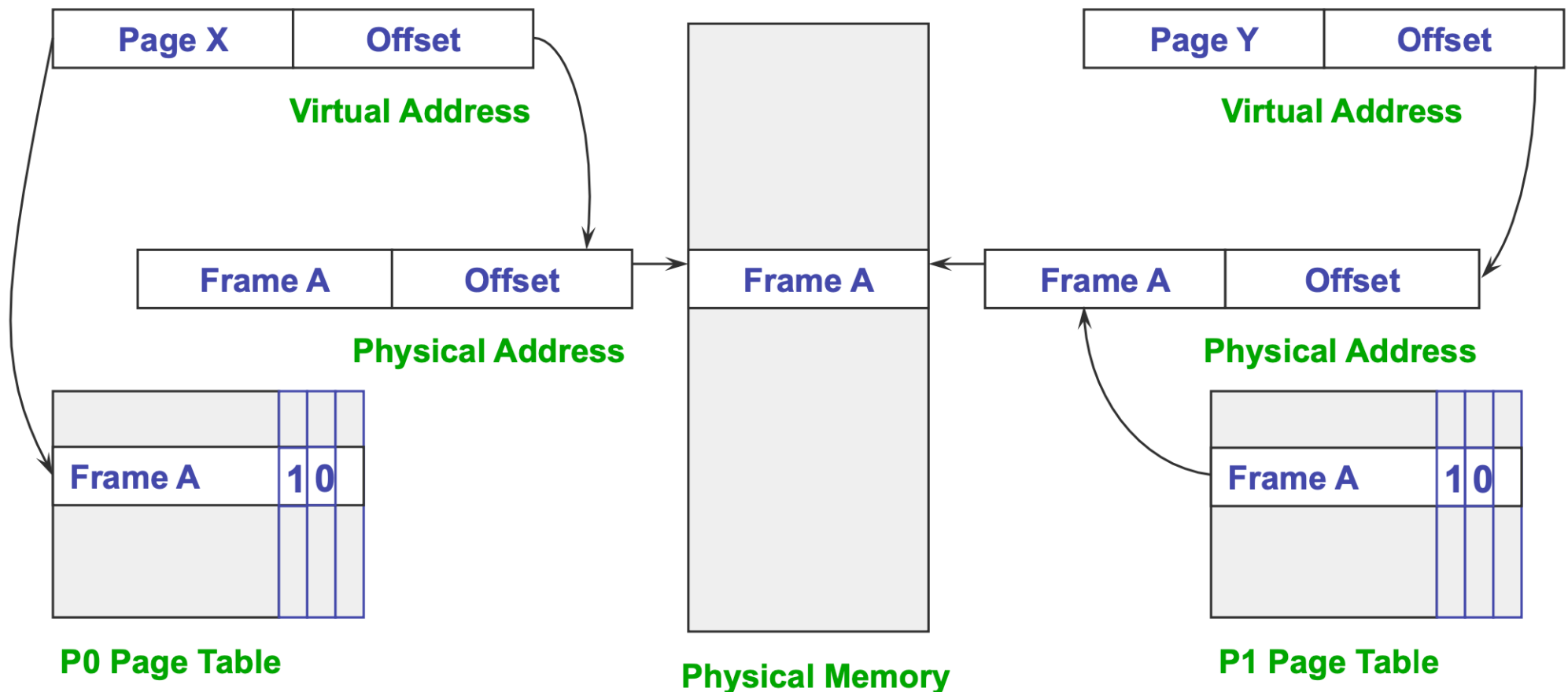
Inverted Page Table Architecture





Page-Level Sharing

- We can use shared memory to allow processes to share data using direct memory references
- Both processes see updates to the shared memory segment » Process B can immediately read an update by process A





Shared Pages

■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

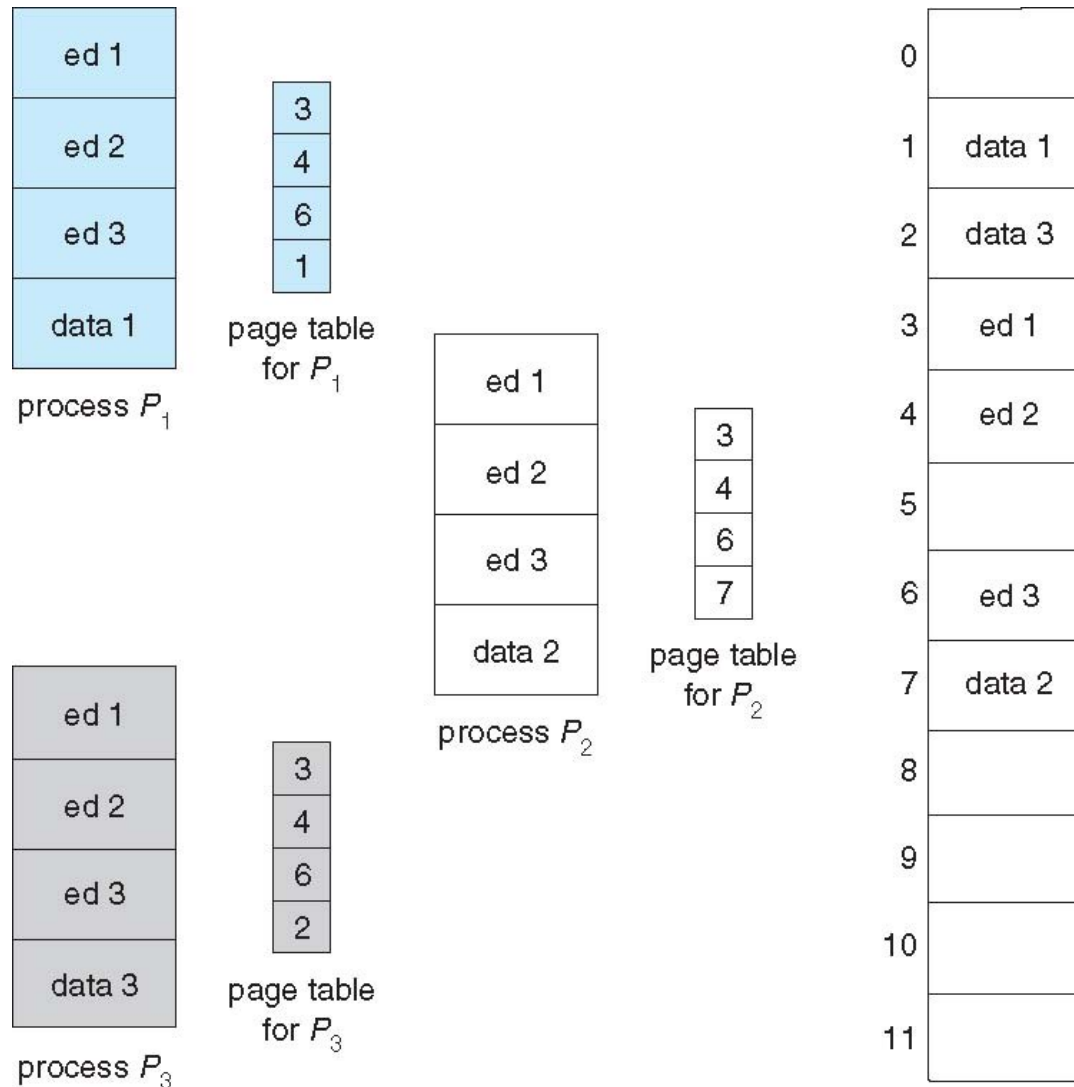
■ Private code and data

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





Shared Pages Example



End of Chapter 9

