CSC139 Operating System Principles

Fall 2020, Part 3-1

Instructor: Dr. Yuan Cheng

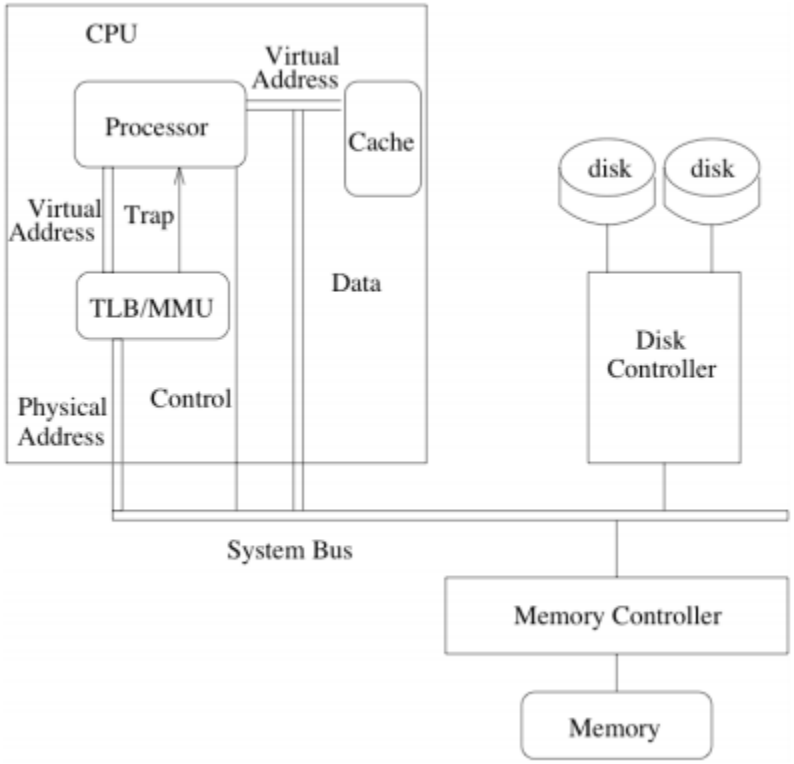
Where we are in the course

- Discussed
  - Processes & Threads
  - CPU Scheduling
  - Synchronization & Deadlock
- Next up
  - Memory Management
- Yet to come
  - File Systems & I/O Storage
  - Security and Protection

Memory Management

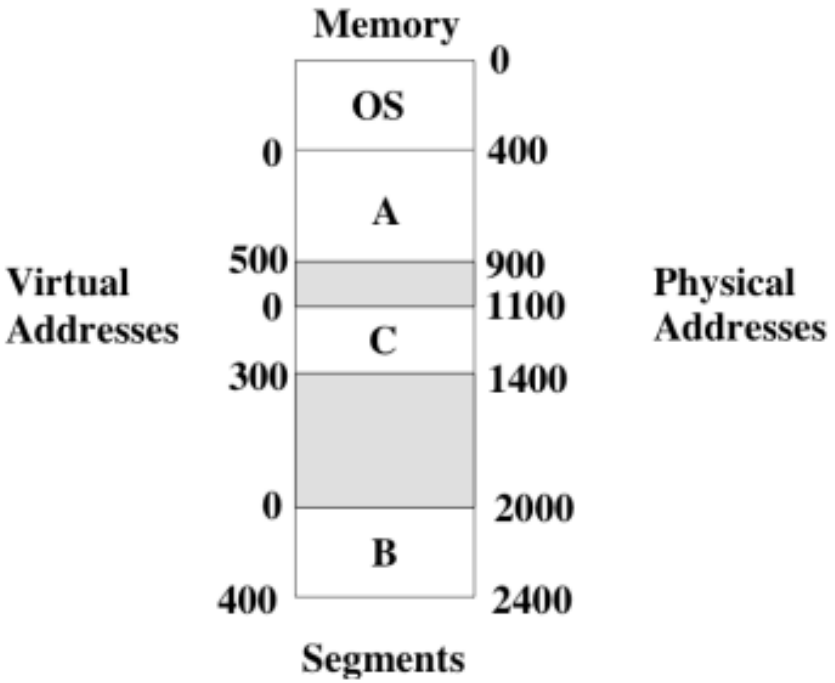
- Where is the executing process?
- How do we allow multiple processes to use main memory simultaneously?
- What is an address and how is one interpreted?

Background



- Program executable starts out on disk
- The OS loads the program into memory
- CPU fetches instructions and data from memory while executing the program

Terminology



- Segment: a chunk of memory assigned to a process
- Physical Address: a real address in memory
- Virtual Address: an address relative to the start of a process's address space

Where do addresses come from?

- 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: The compiler generates an address, but at load time the OS determines the process's starting position. Once the process loads, it does not move in memory.
  - Execution time: The compiler generates an address, and the OS can place it anywhere it wants in memory.

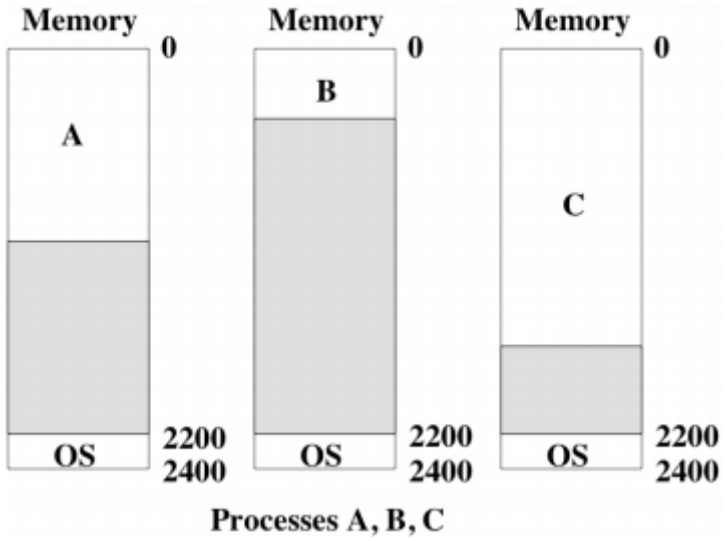
Logical vs. Physical Address Space

- 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

Uni-programming

- OS gets a fixed part of memory (highest memory in DOS).
- One process executes at a time.
- Process is always loaded starting at address 0.
- Process executes in a contiguous section of memory.
- Compiler can generate physical addresses.
- Maximum address = Memory Size - OS Size
- OS is protected from process by checking addresses used by process.

Uni-programming



=> Simple, but does not allow for overlap of I/O and computation

Multiple Programs Share Memory

- Transparency:
  - We want multiple processes to coexist in memory.
  - No process should be aware that memory is shared.
  - Processes should not care what physical portion of memory they are assigned to.
- Safety:
  - Processes must not be able to corrupt each other.
  - Processes must not be able to corrupt the OS.
- Efficiency:
  - Performance of CPU and memory should not be degraded badly due to sharing.

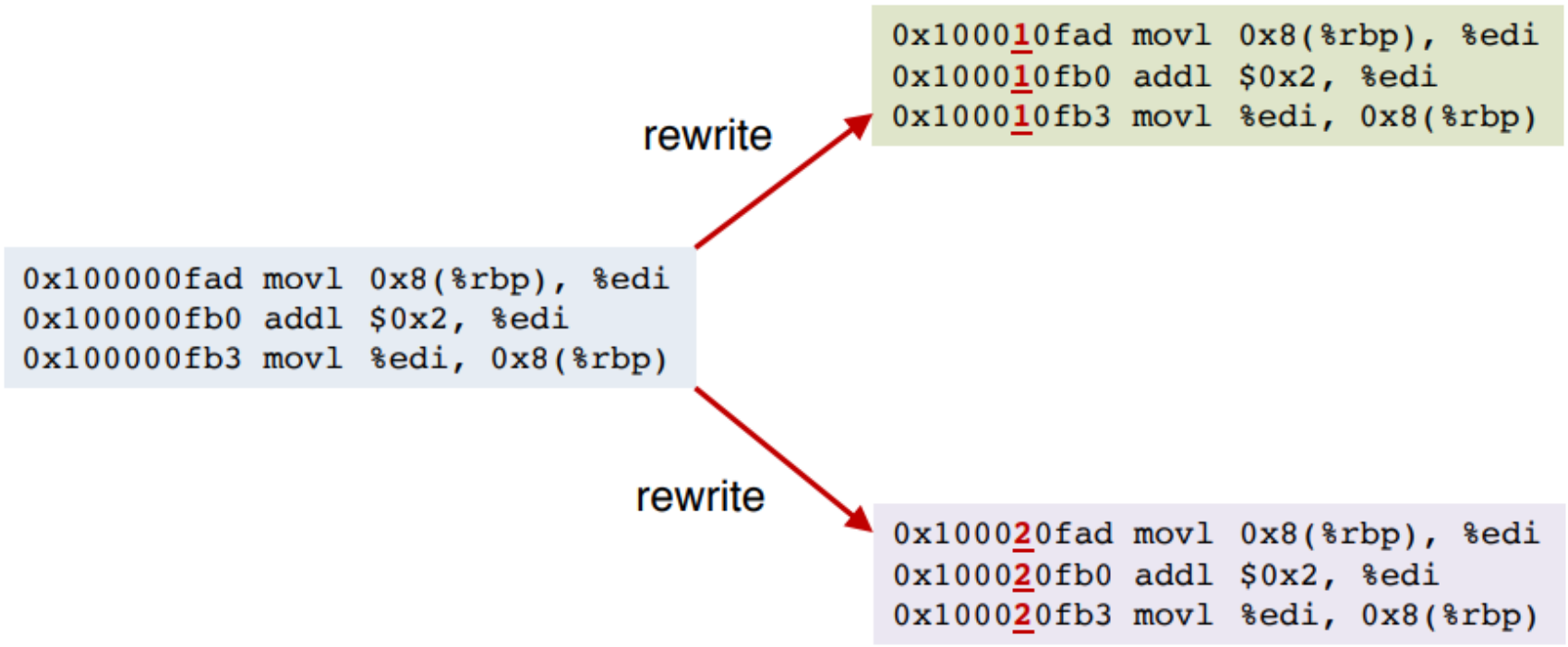
How to Run Multiple Programs?

- Approaches:
  - Static relocation
  - Dynamic relocation
  - Segmentation

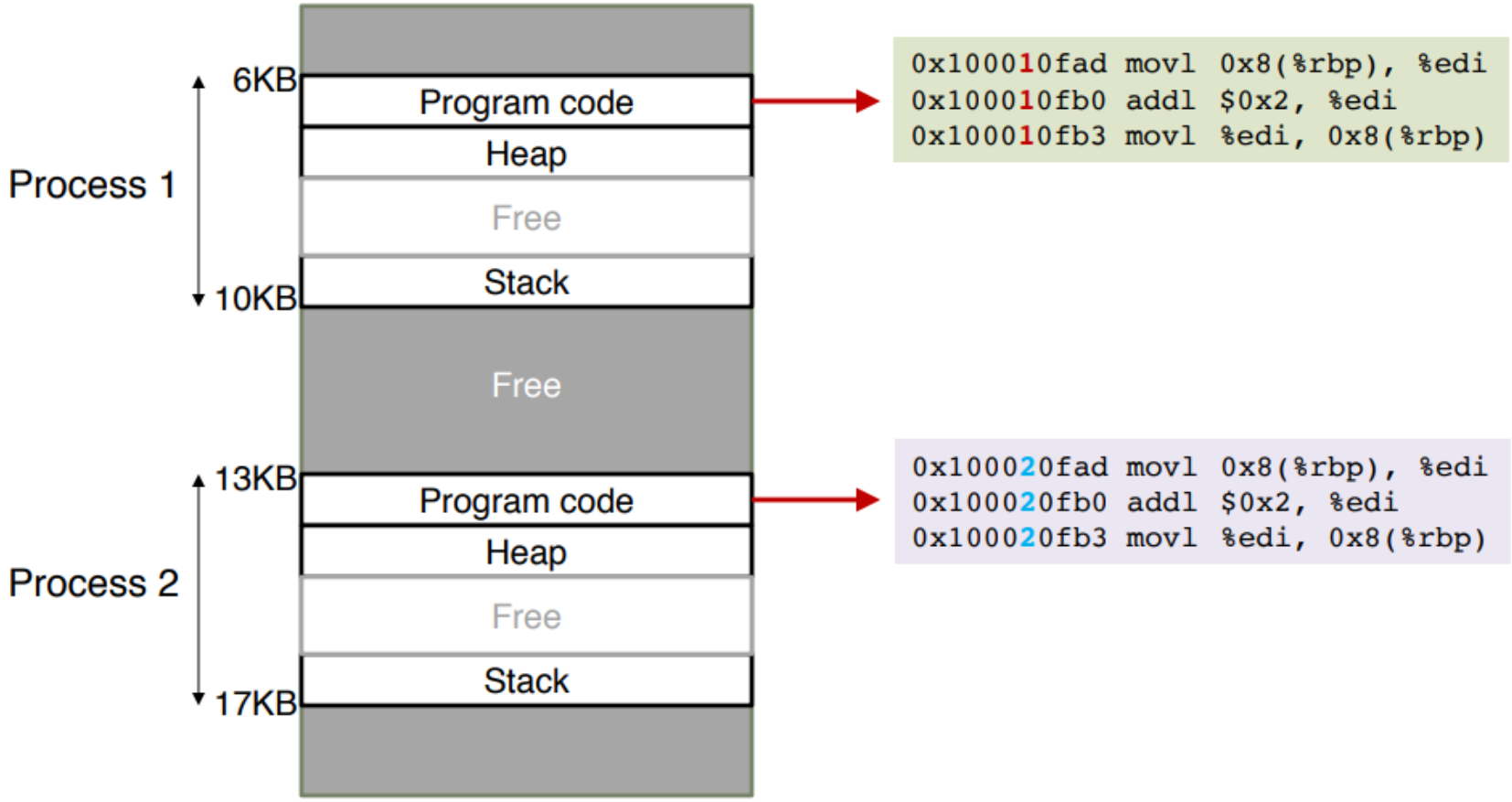
Static Relocation

- At load time, the OS adjusts the addresses in a process to reflect its position in memory.
  - Each rewrite uses different addresses and pointers
  - Change jumps, loads, etc.
- Once a process is assigned a place in memory and starts executing it, the OS cannot move it. (Why?)

Rewrite for Each New Process



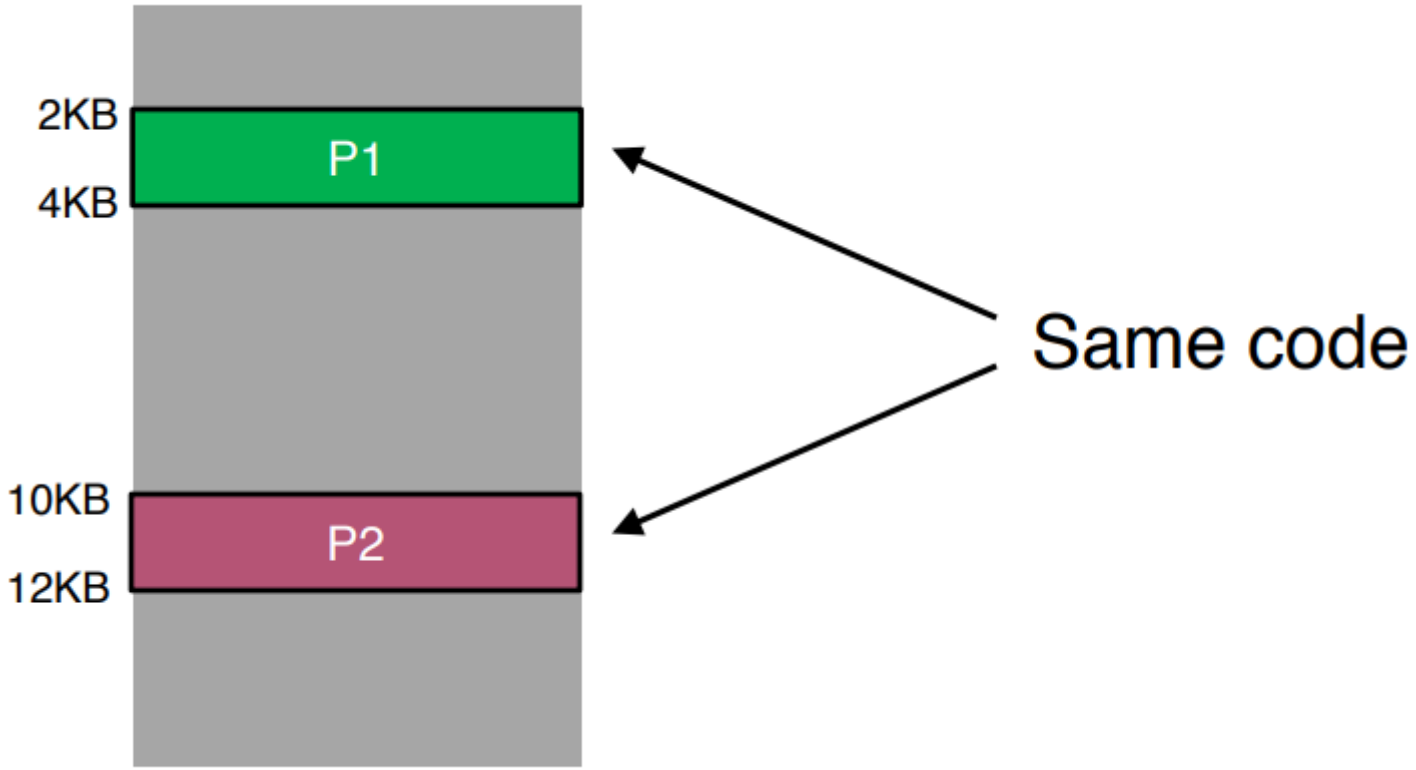
Rewrite for Each New Process



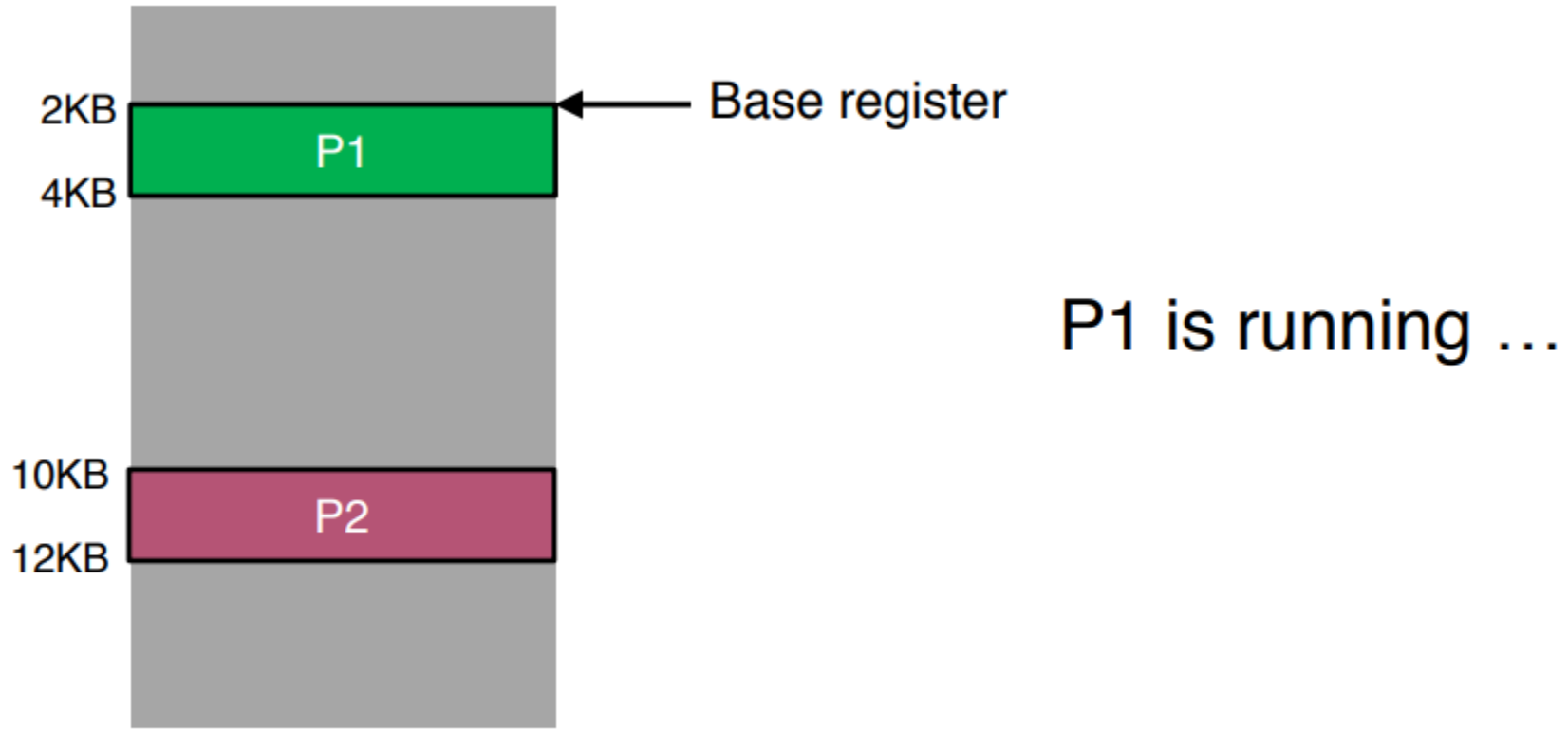
Dynamic Relocation

- Hardware adds relocation register (base) to virtual address to get a physical address;
  - Each process has a different value in the base register when running
- Hardware compares address with limit register (address must be less than limit).
- If test fails, the processor takes an address trap and ignores the physical address.

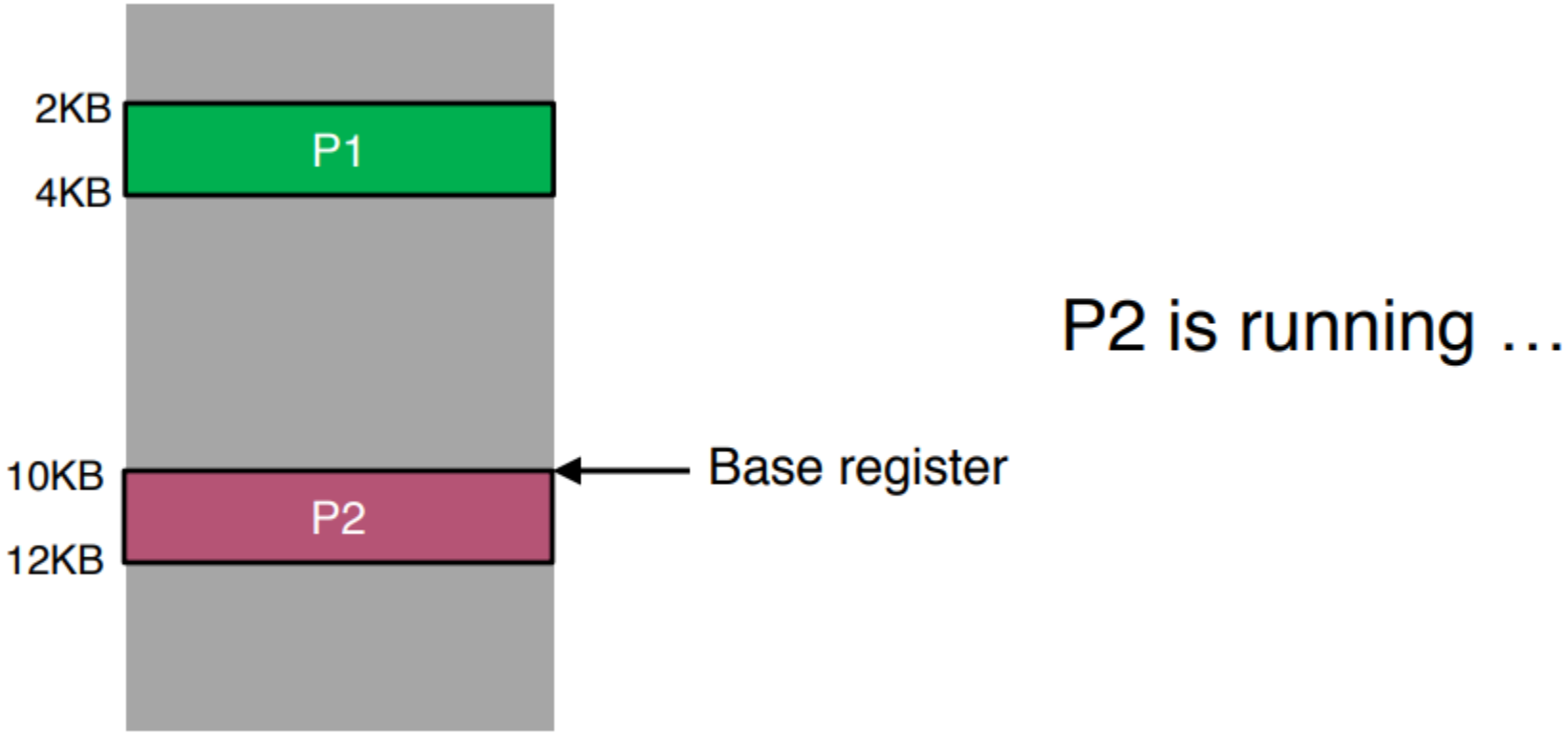
Base Relocation



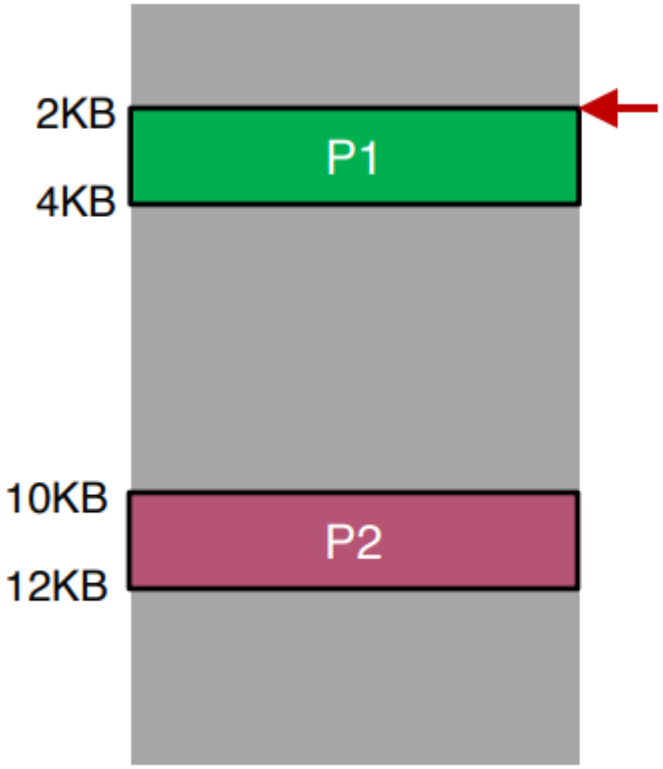
Base Relocation



Base Relocation

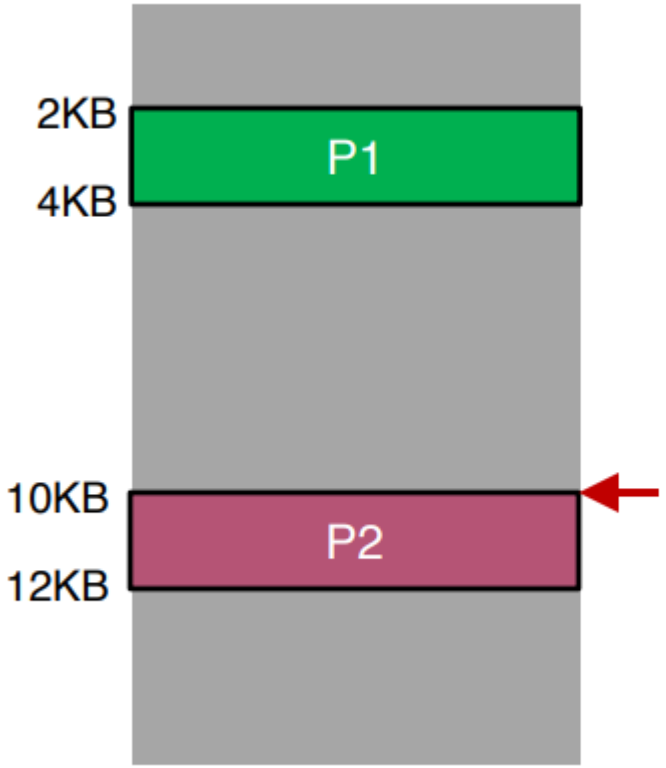


Base Relocation



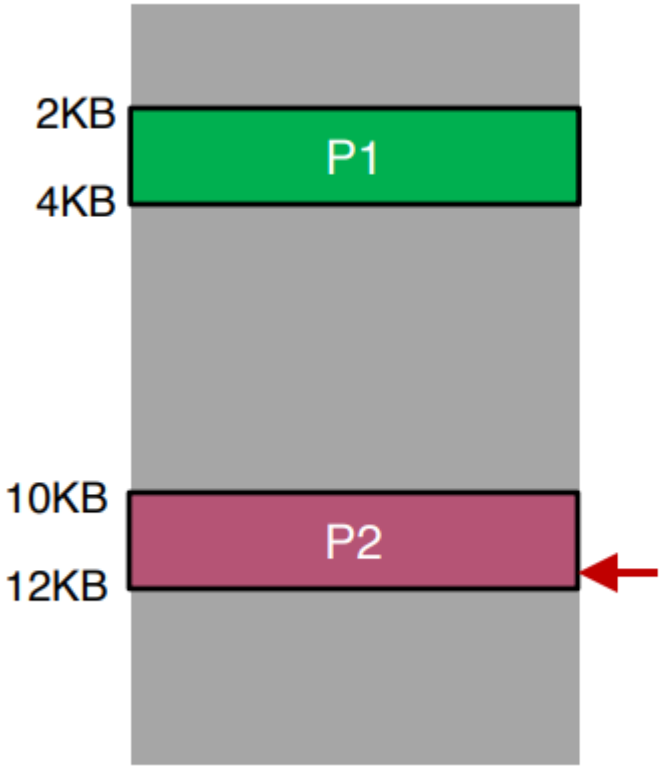
Virtual	Physical
P1: load 100, R1	load 2148, R1

Base Relocation



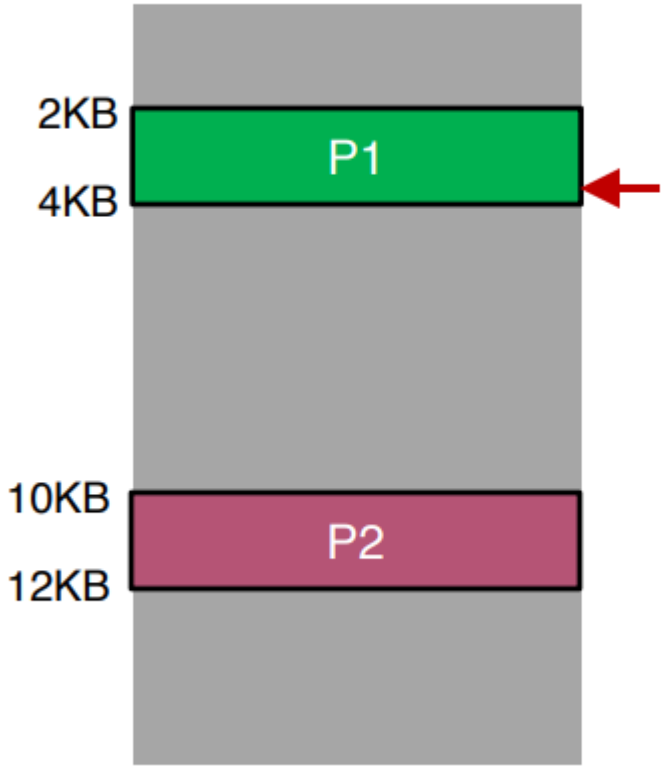
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1

Base Relocation



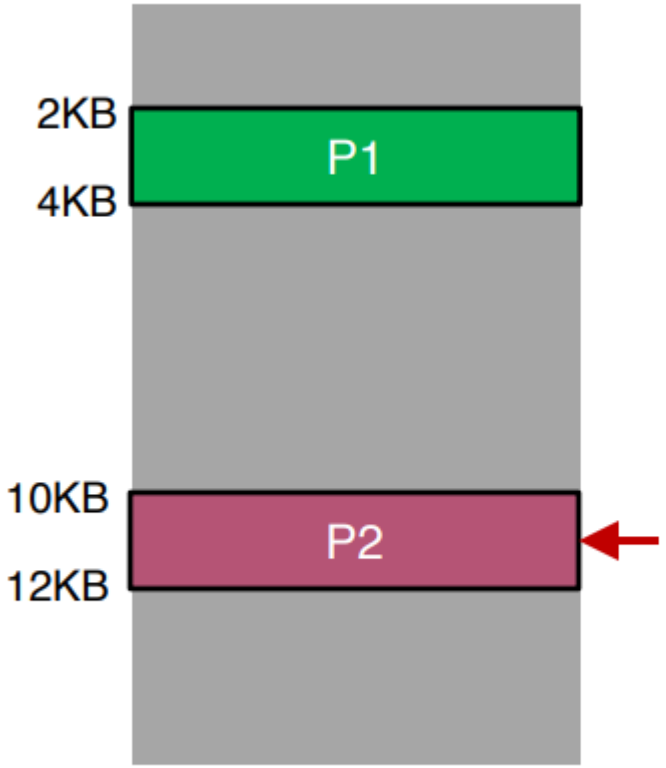
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1

Base Relocation



Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1

Base Relocation

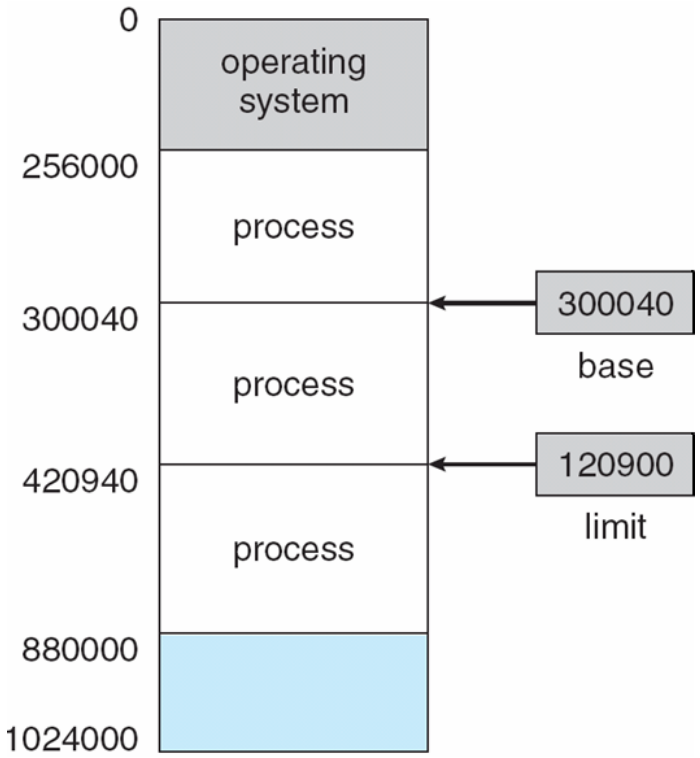


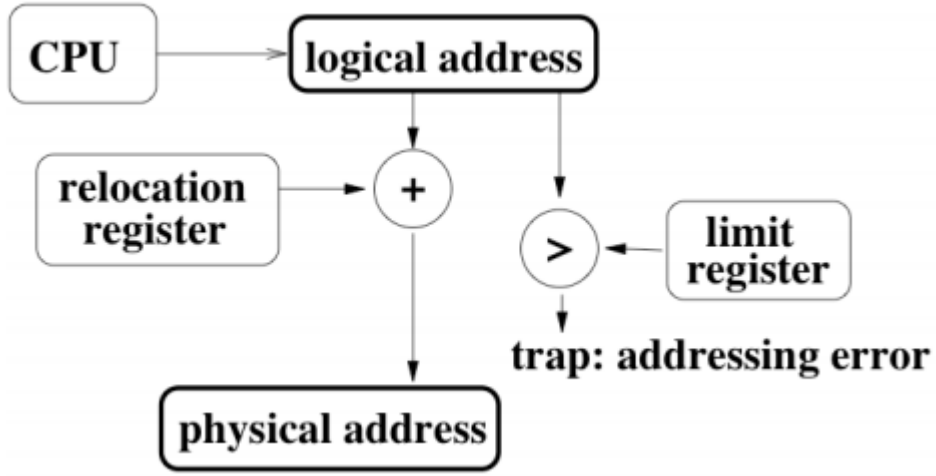
Virtual	Physical
P1: load 100, R1	load 2148, R1
P2: load 100, R1	load 10340, R1
P2: load 2000, R1	load 12240, R1
P1: load 2000, R1	load 4048, R1
P1: store 9241, R1	**store 11289, R1**

Can P1 hurt P2?
Can P2 hurt P1?

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





Dynamic Relocation

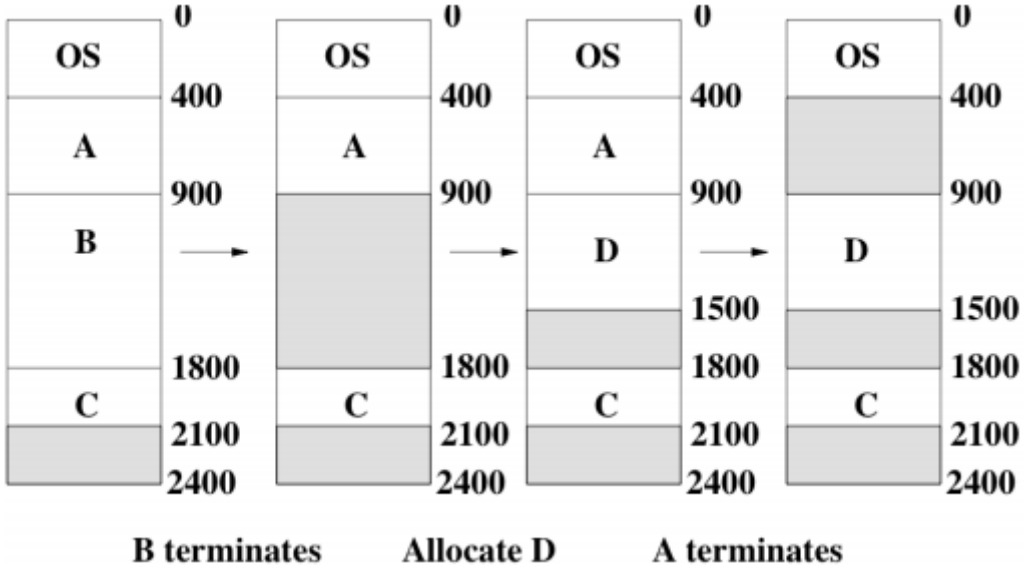
- Advantages:
  - OS can easily move a process during execution.
  - OS can allow a process to grow over time.
  - Simple, fast hardware: two special registers, an add, and a compare.
- Disadvantages:
  - Slows down hardware due to the add on every memory reference.
  - Can't share memory (such as program text) between processes.
  - Process is still limited to physical memory size.
  - Degree of multiprogramming is very limited since all memory of all active processes must fit in memory.
  - Complicates memory management.

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

Memory Allocation

- As processes enter the system, grow, and terminate, the OS must keep track of which memory is available and utilized



  - Holes: pieces of free memory (shaded area in the above figure)
  - Given a new process, the OS must decide which hole to use for the process

Memory Allocation Policies

- First-Fit: allocate the first one in the list in which the process fits. The search can start with the first hole, or where the previous first-fit search ended.
- Best-Fit: Allocate the smallest hole that is big enough to hold the process. The OS must search the entire list or store the list sorted by size hole list.
- Worst-Fit: Allocate the largest hole to the process. Again the OS must search the entire list or keep the list sorted.
- Simulations show first-fit and best-fit usually yield better storage utilization than worst-fit; first-fit is generally faster than best-fit

External Fragmentation

- Frequent loading and unloading programs causes free space to be broken into little pieces
- External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous
- 50-percent rule: Simulations show that for every 2N allocated blocks, N blocks are lost due to fragmentation (i.e., 1/3 of memory space is wasted)
- We want an allocation policy that minimizes wasted space.

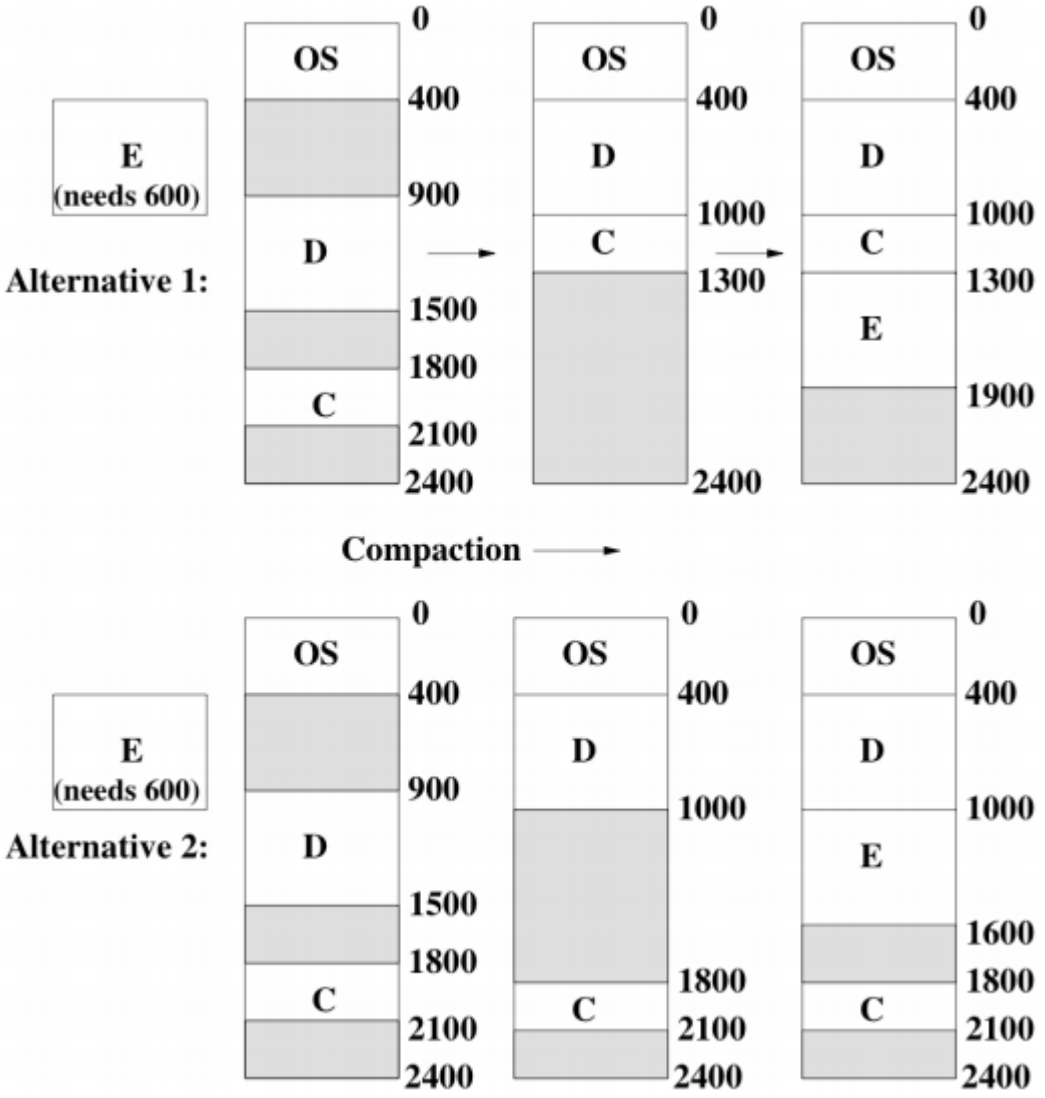
Internal Fragmentation

- Consider a process of size 8846 bytes and a block of size 8848 bytes ⇒it is more efficient to allocate the process the entire 8848 block than it is to keep track of 2 free bytes
- Internal fragmentation exists when memory internal to a partition that is wasted

Compaction

- 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

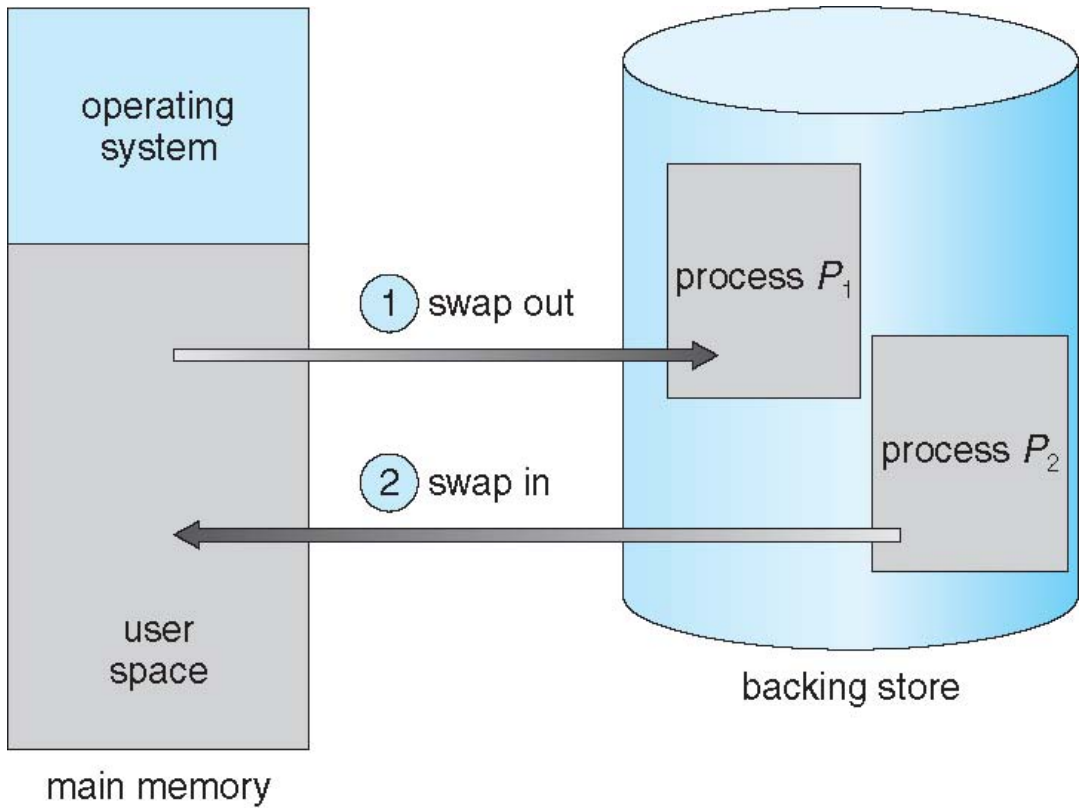
- How much memory is moved?
- How big a block is created?
- Any other choices?



Swapping

- Roll out a process to disk, releasing all the memory it holds.
- When process becomes active again, the OS must reload it in memory.
  - With static relocation, the process must be put in the same position.
  - With dynamic relocation, the OS finds a new position in memory for the process and updates the relocation and limit registers.
- If swapping is part of the system, compaction is easy to add.
- How could or should swapping interact with CPU scheduling?

Systematic View of Swapping



Problems

- Fragmentation
  - Frequent compaction needed
- Contiguous allocation
  - Difficult to grow or shrink process memory
- Requirement that process resides entirely in memory
  - Swapping helps but not perfect

Session Plan

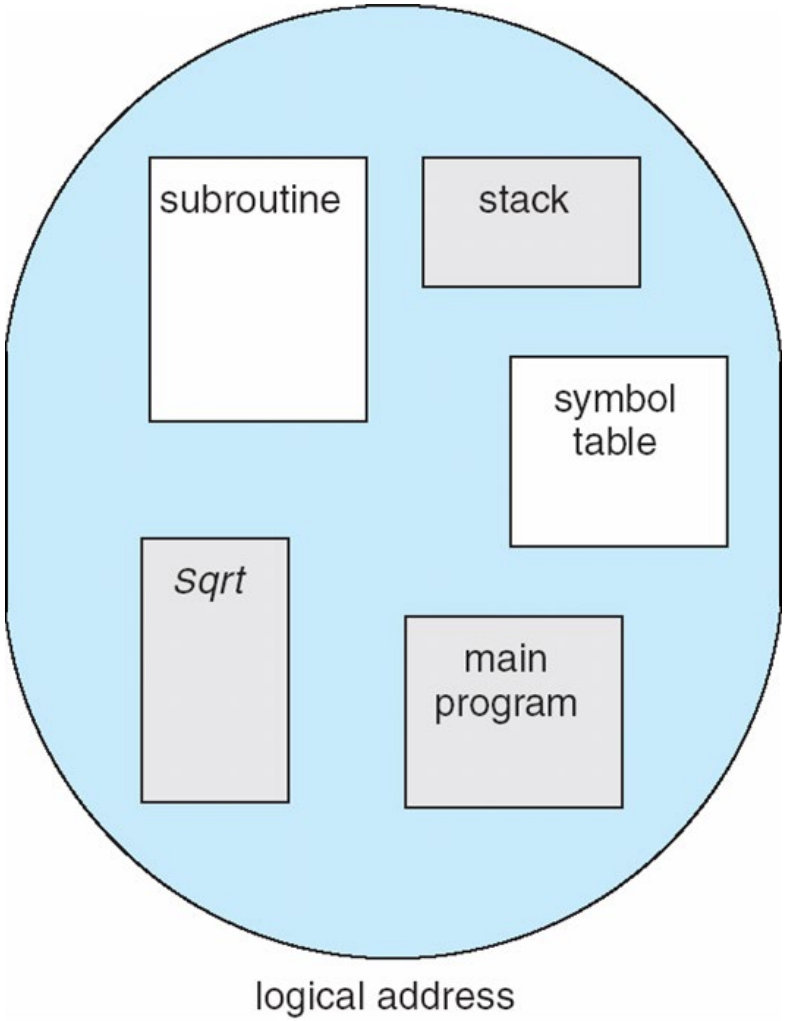
- Segmentation
- Paging

Segmentation

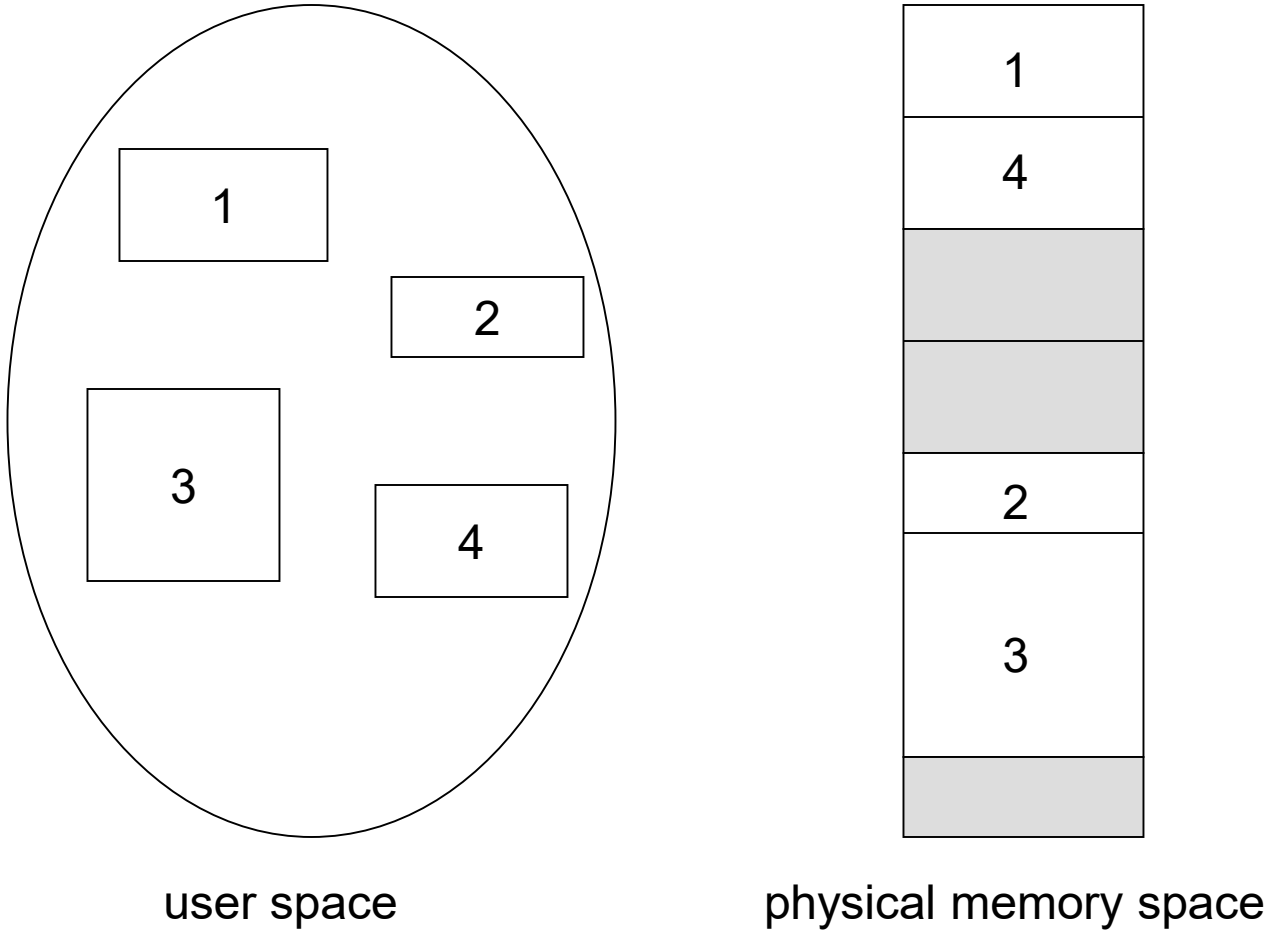
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    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



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

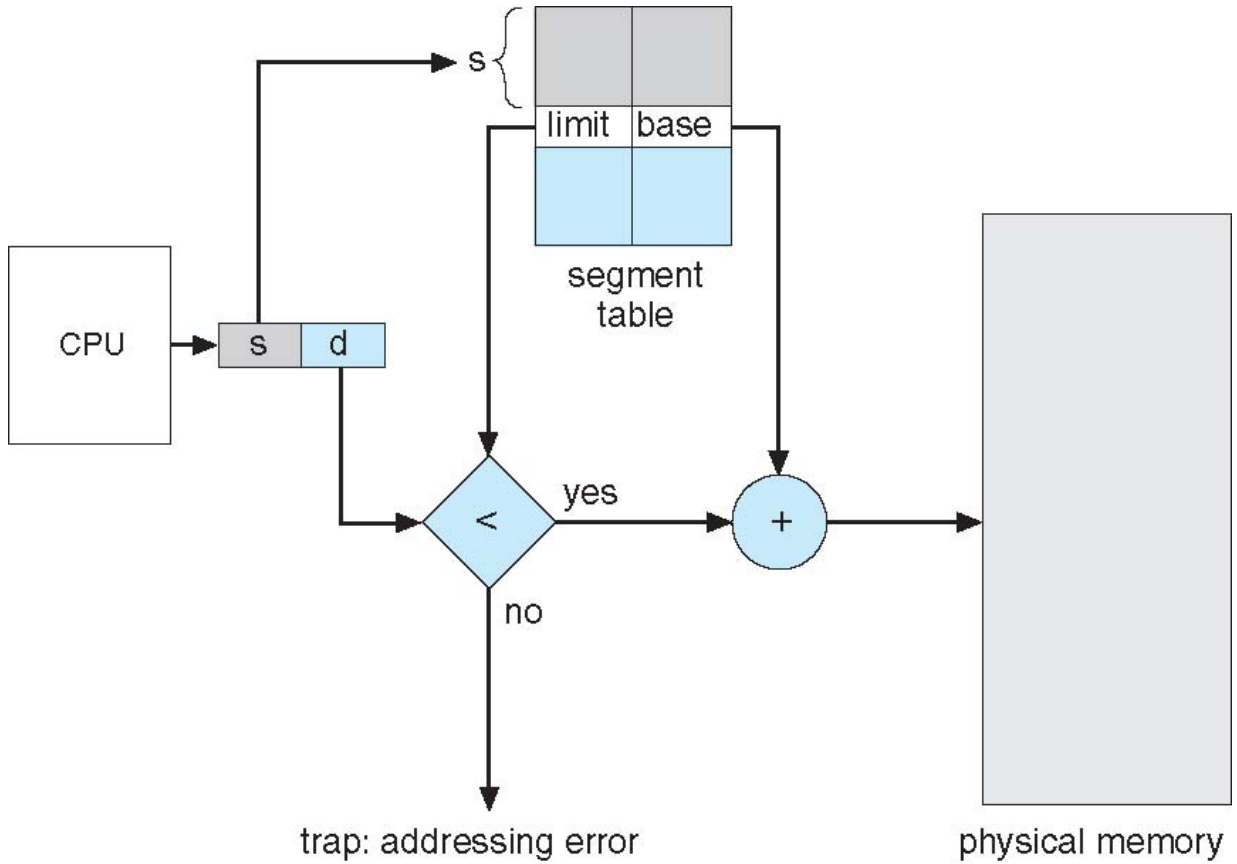
- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- 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



Paging: Motivation & Features

- 90/10 rule: Processes spend 90% of their time accessing 10% of their space in memory

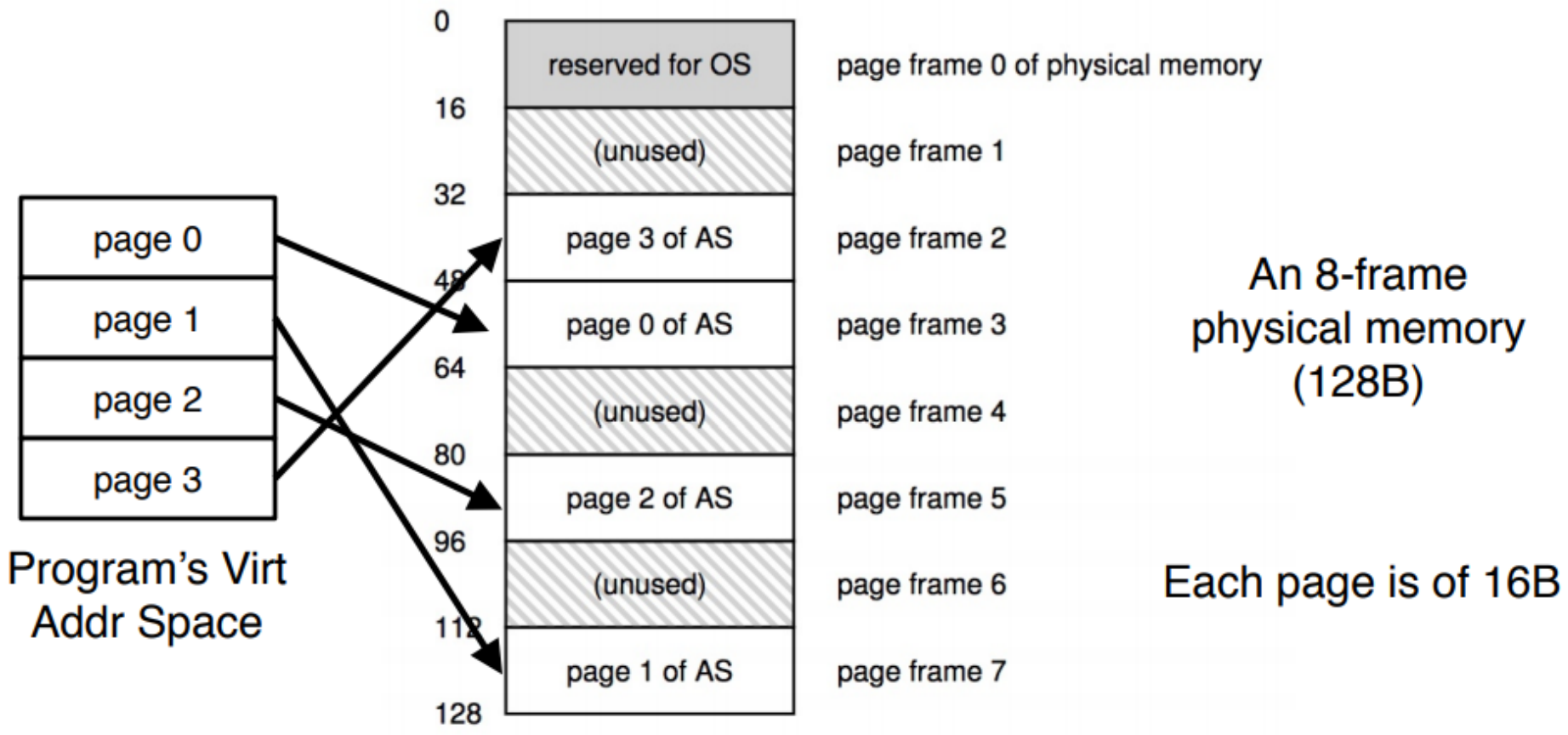
=> Keep only those parts of a process in memory that are actually being used

- Pages greatly simplify the hole fitting problem
- The logical memory of the process is contiguous, but pages need not be allocated contiguously in memory
- By dividing memory into fixed size pages, we can eliminate external fragmentation
- Paging does not eliminate internal fragmentation

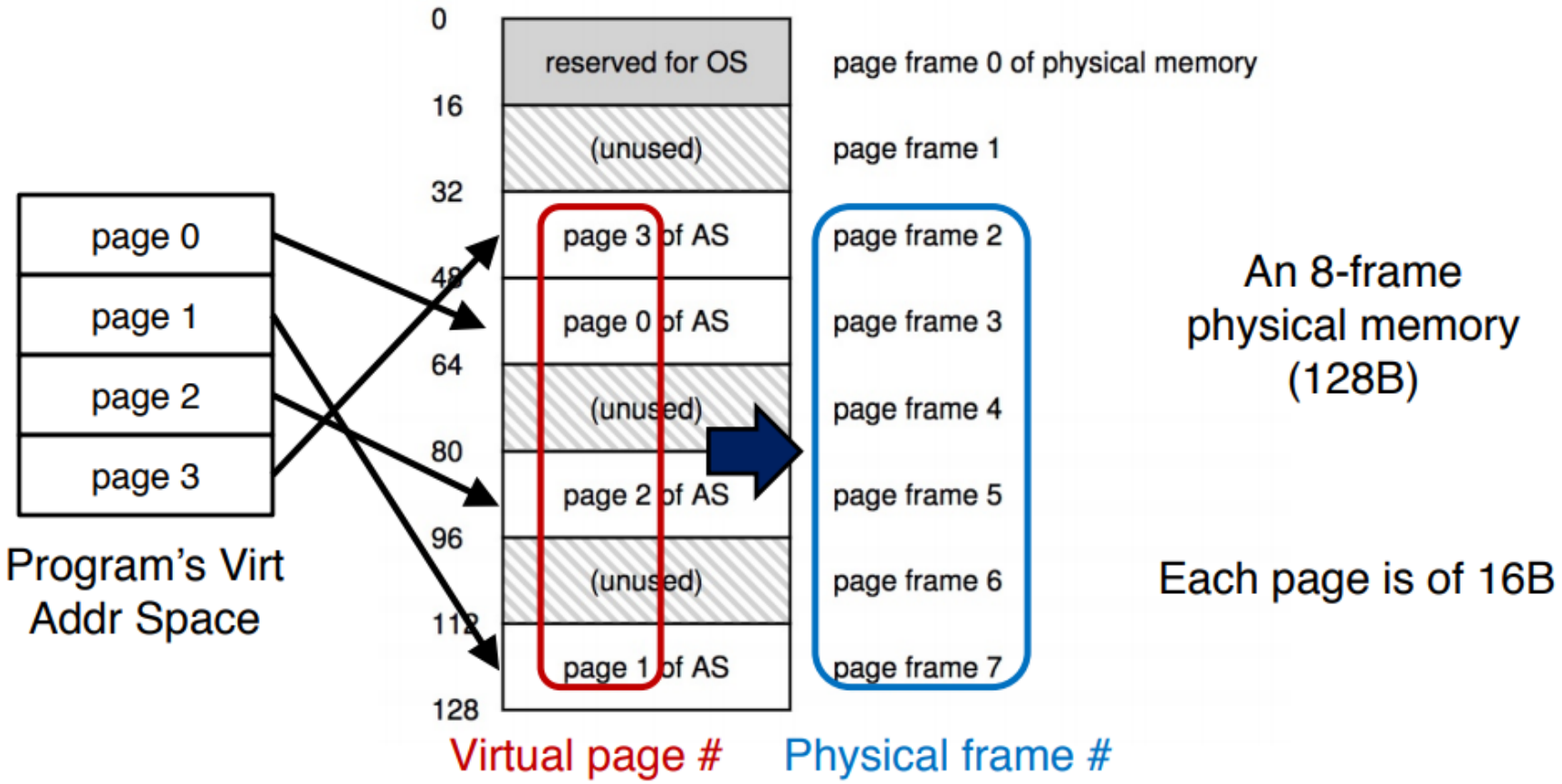
Paging

- A memory management scheme that allows the physical address space of a process to be non-contiguous
- Divide physical memory into fixed-sized blocks called frames
- Divide logical memory into blocks of same size called pages
- Flexible mapping: Any page can go to any free frame
- Scalability: To run a program of size *n* pages, need to find *n* free frames and load program

A Simple Example

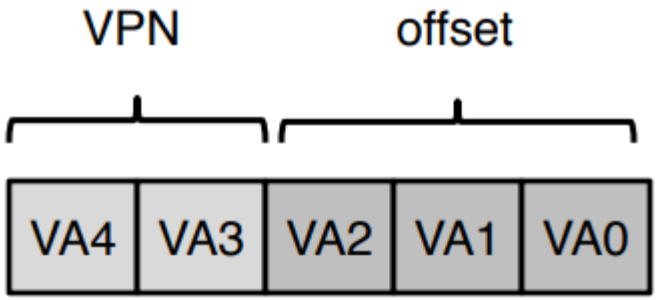


A Simple Example (cont.)



Addressing Basics

- For segmentation
  - High bits => segment #
  - Low bits => offset
- For paging
  - High bits => page #
  - Low bits => offset



Q: How many offset bits do we need?
A: log(page_size)

Address Examples

Page size	Offset
16 Bytes	
2 KB	
4 MB	
256 Bytes	
16 KB	

Address Examples

Page size	Offset
16 Bytes	4
2 KB	11
4 MB	22
256 Bytes	8
16 KB	14

Address Examples

Page size	Offset	Virtual Address Bits	VPN
16 Bytes	4	10	
2 KB	11	20	
4 MB	22	32	
256 Bytes	8	16	
16 KB	14	64	

Address Examples

Page size	Offset	Virtual Address Bits	VPN
16 Bytes	4	10	6
2 KB	11	20	9
4 MB	22	32	10
256 Bytes	8	16	8
16 KB	14	64	50

Address Examples

Page size	Offset	Virtual Address Bits	VPN	# of Virtual Pages
16 Bytes	4	10	6	
2 KB	11	20	9	
4 MB	22	32	10	
256 Bytes	8	16	8	
16 KB	14	64	50	

Address Examples

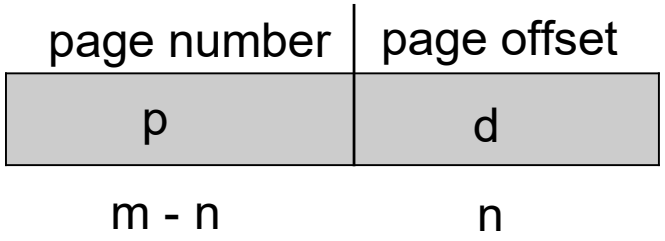
Page size	Offset	Virtual Address Bits	VPN	# of Virtual Pages
16 Bytes	4	10	6	64
2 KB	11	20	9	512
4 MB	22	32	10	1K
256 Bytes	8	16	8	256
16 KB	14	64	50	2^{50}

Page Table

- A per-process data structure used to keep track of virtual page to physical frame mapping
- Major role: store address translation

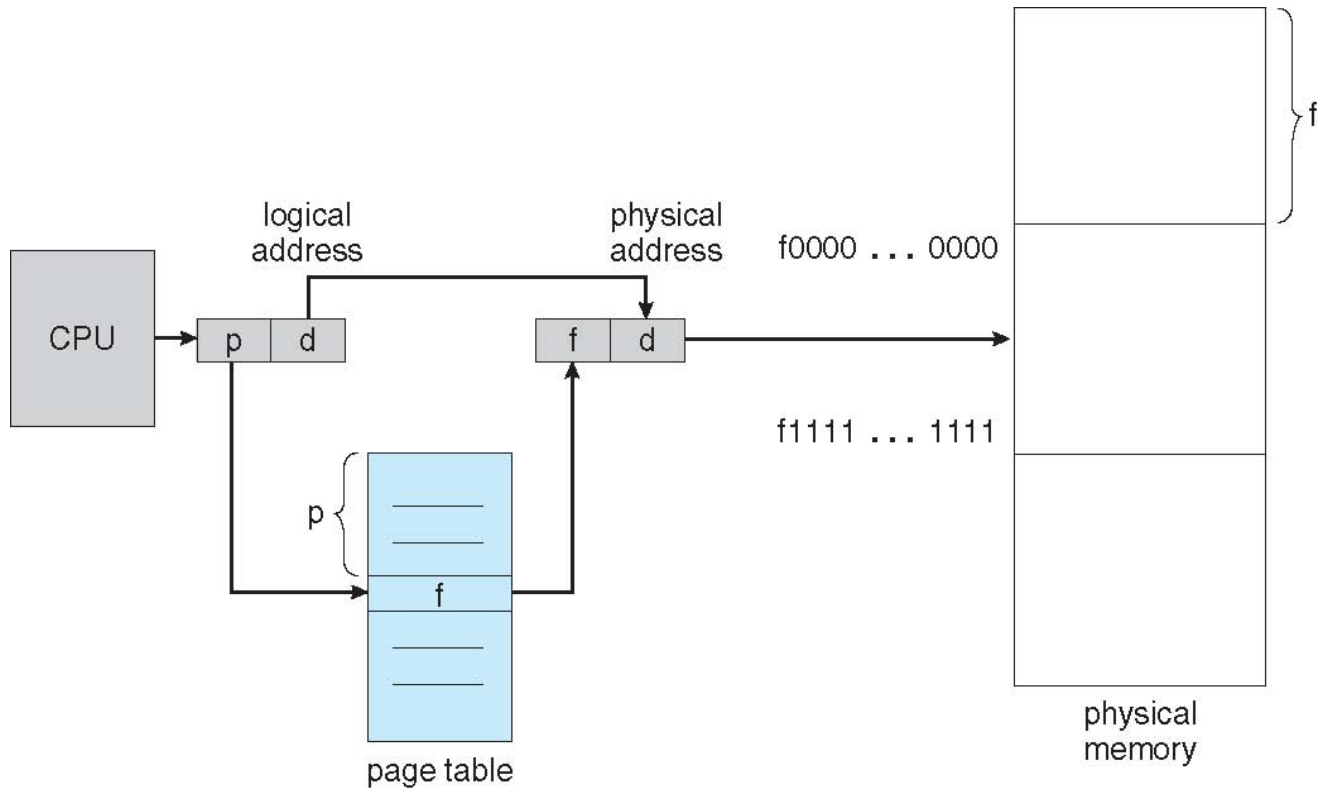
Address Translation Scheme

- m-bit virtual 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

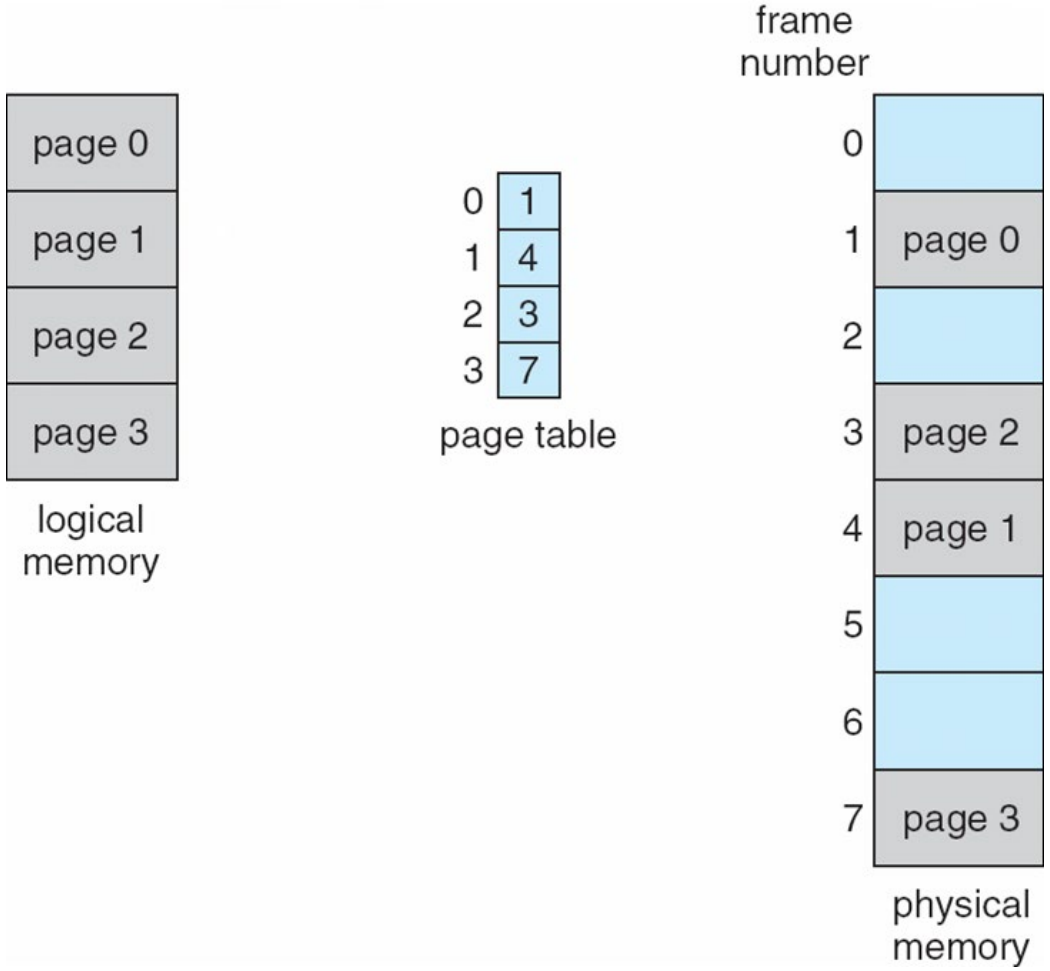
Paging Hardware



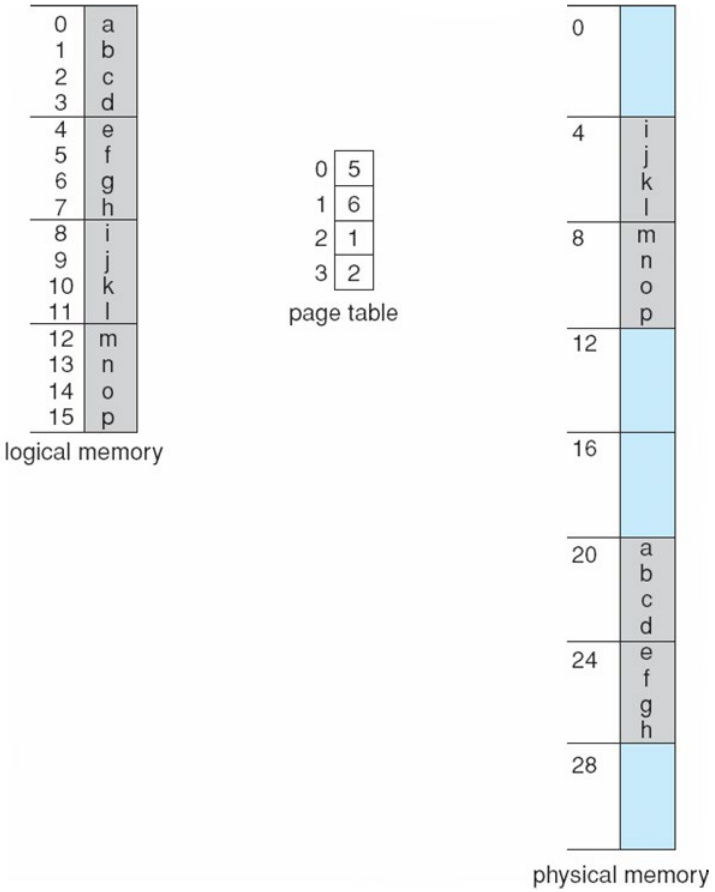
Paging Hardware

- Paging is a form of dynamic relocation, where each virtual address is bound by the paging hardware to a physical address.
- Think of the page table as a set of relocation registers, one for each frame.
- Mapping is invisible to the process; the OS maintains the mapping and the hardware does the translation.
- Protection is provided with the same mechanisms as used in dynamic relocation.

Paging Model of Logical and Physical Memory



Paging Example

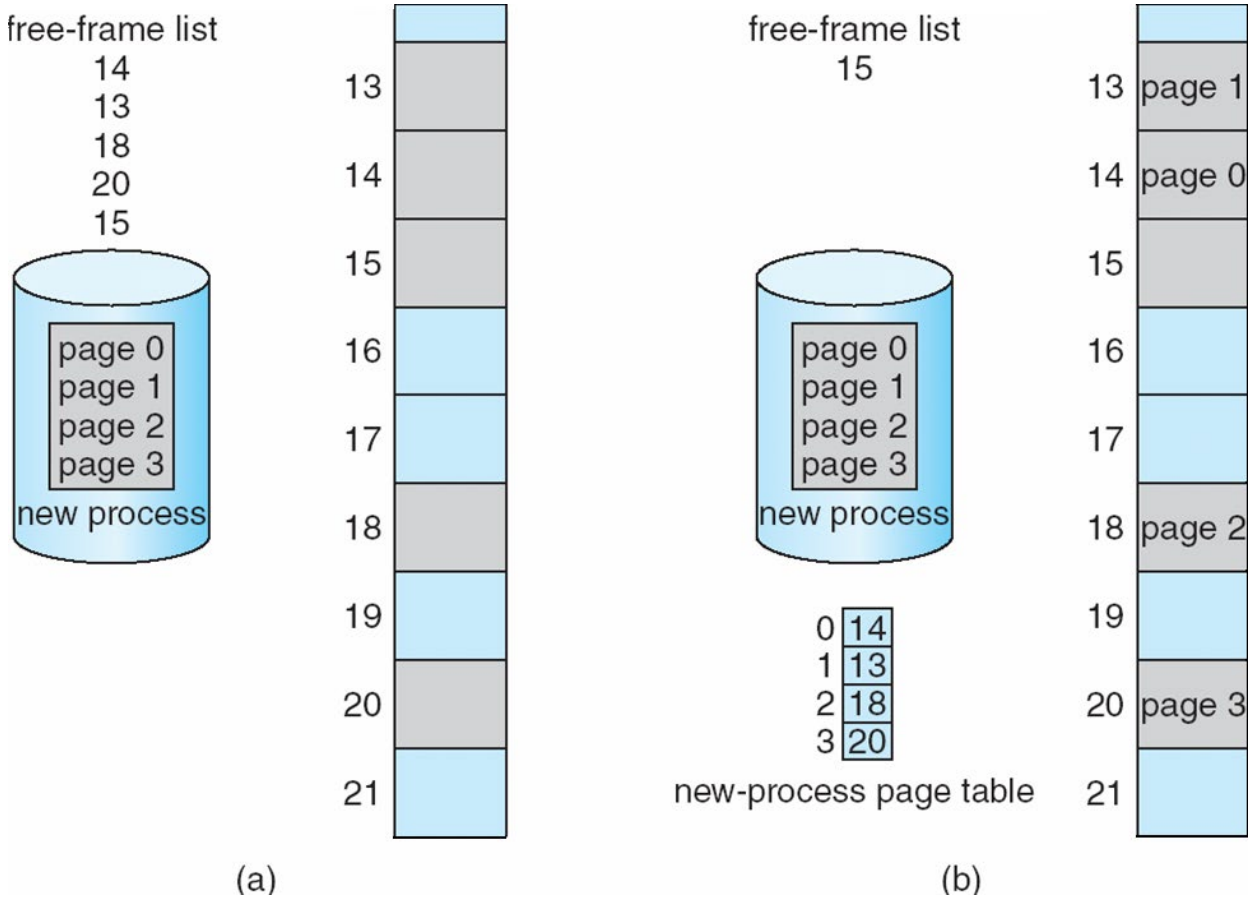


n=2 and *m*=4 32-byte memory and 4-byte pages

Paging (cont.)

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

Free Frames



Before allocation

After allocation

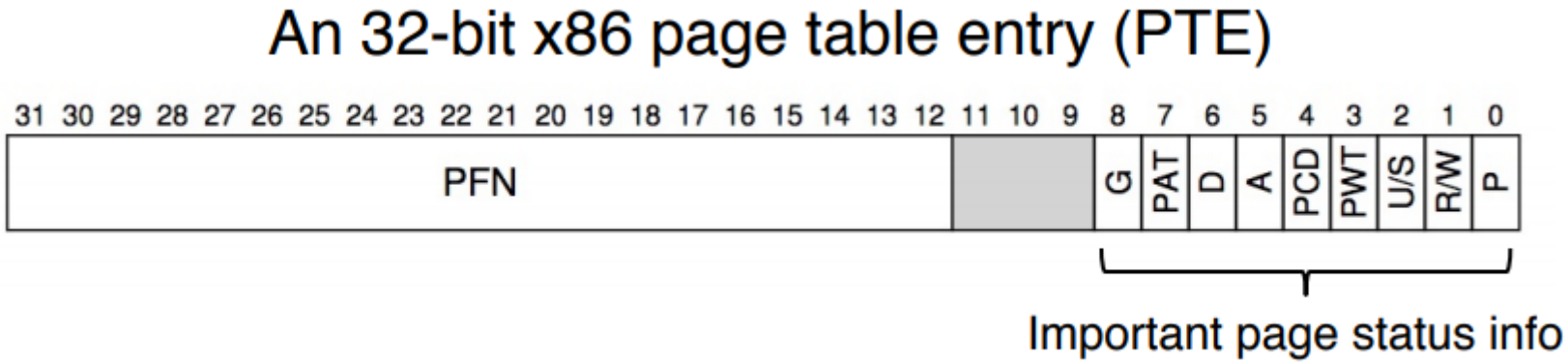
Implementation of Page Table

- Page table is kept in main memory
- Each page table entry (PTE) holds

  <physical translation + other info>
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table

Page Table Entry (PTE)

- The simplest form of a page table is a linear page table
  - Array data structure
  - OS indexes the array by virtual page number (VPN)
  - To find the desired physical frame number (PFN)



Paging Problems

- Page tables are too slow
- Page tables are too big

Translation Lookaside Buffer (TLB)

Performance Problems of Paging

- A basic memory access protocol
  1. Fetch the translation from in-memory page table
  2. Explicit load/store access on a memory address
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table
  - And one for the data/instruction
- Too much performance overhead!

Speeding Up Translation

- 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)
- A TLB is part of the memory-management unit (MMU)
- A TLB is a hardware cache
- Algorithm sketch
  - For each virtual memory reference, hardware first checks the TLB to see if the desired translation is held therein

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a TLB hit – extract PFN from the TLB entry, concatenate it onto the offset to form the PA
4. If it is a TLB miss – access page table to get the translation, update the TLB entry with the translation

How Many TLB Lookups?

- Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

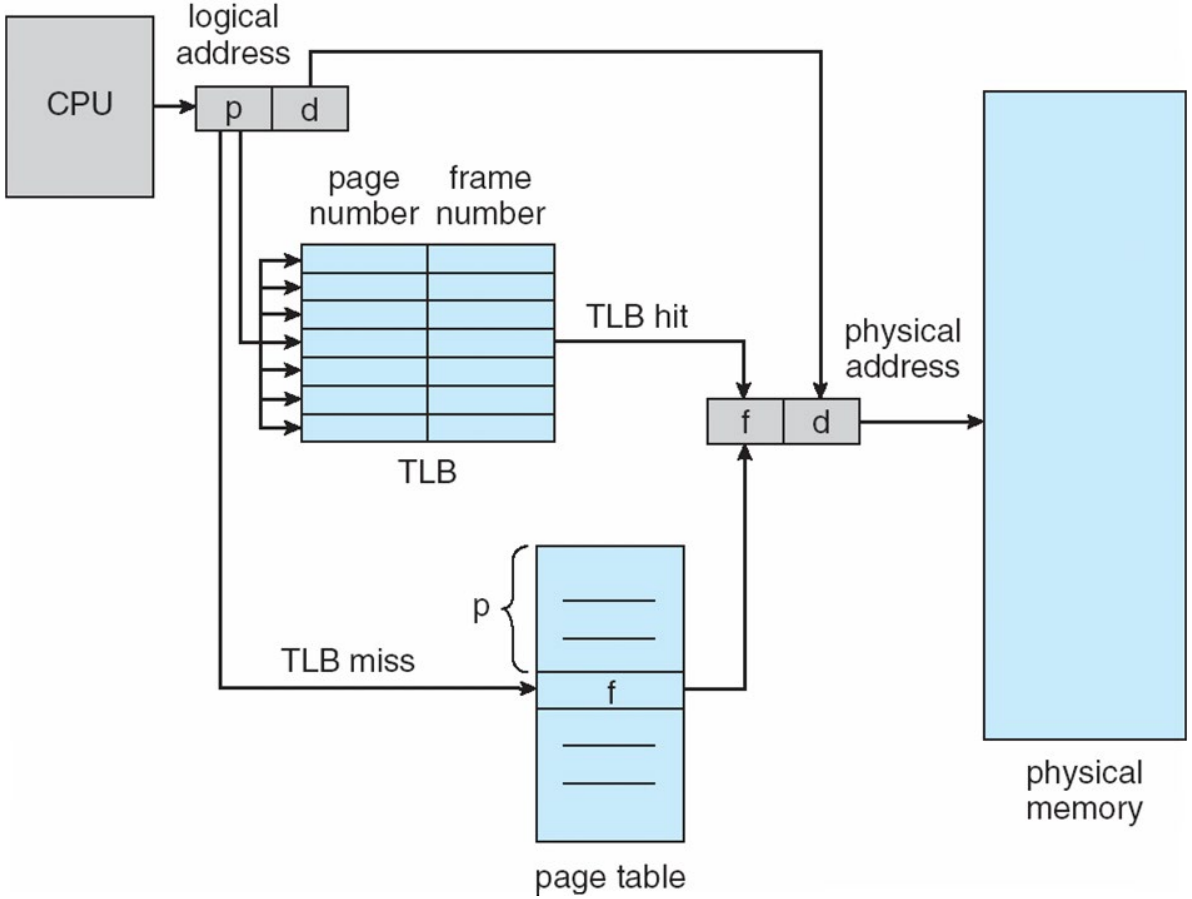
How Many TLB Lookups?

- Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

- Array a has 1024 items, each item is 4 bytes: size(a) = 4096 bytes
- Number of TLB miss = 4096/4096 = 1
- TLB miss rate = 1/1024 = 0.09%

Paging Hardware with TLB



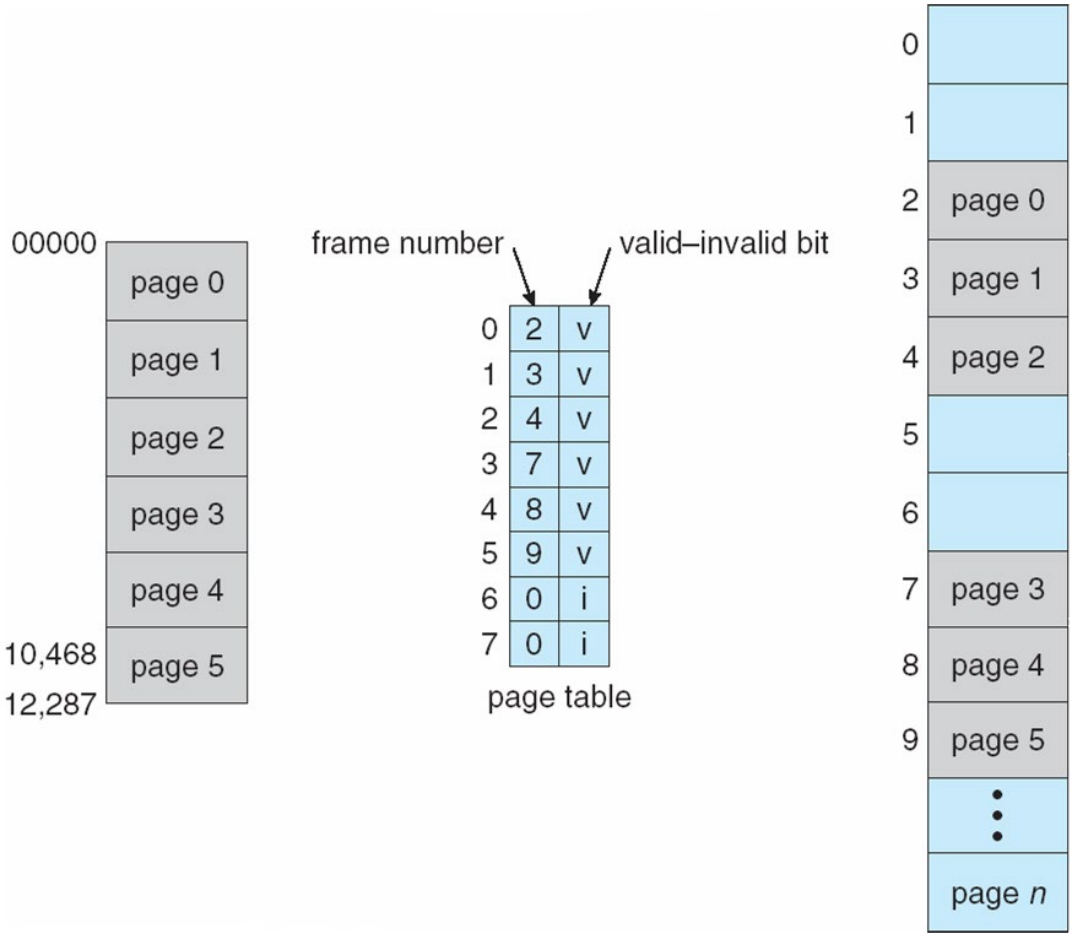
Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 100ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε = 100ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

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



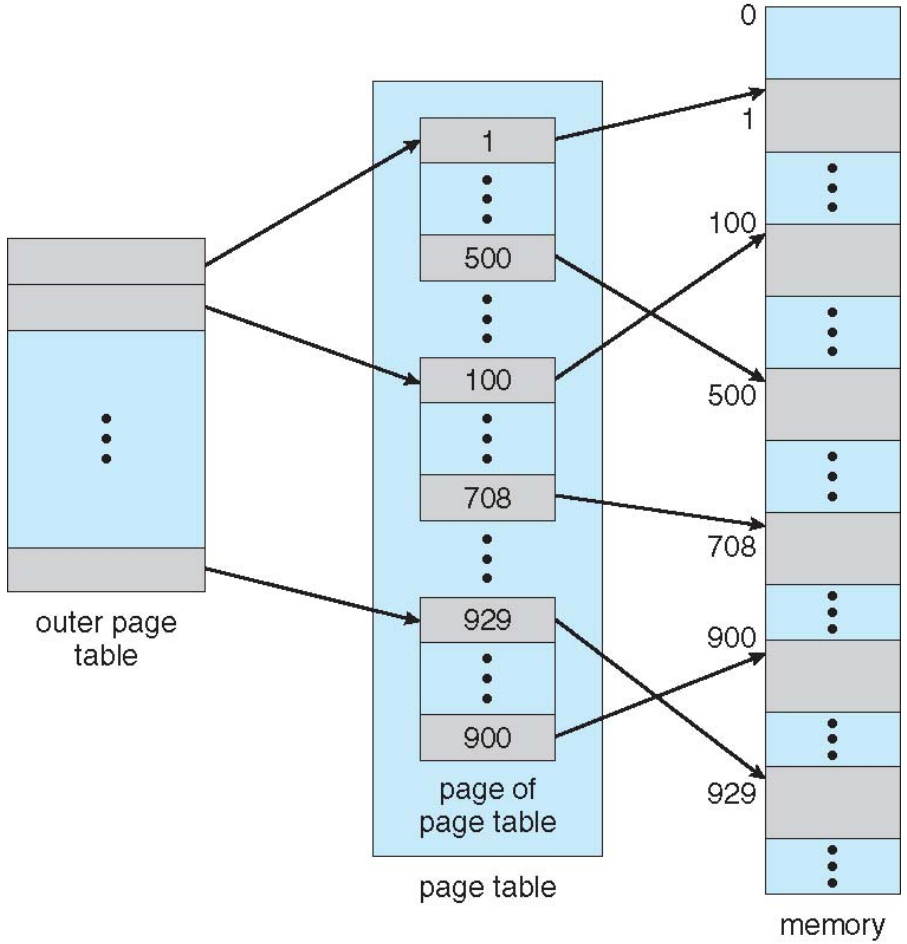
Structure of the Page Table

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

Hierarchical Page Tables

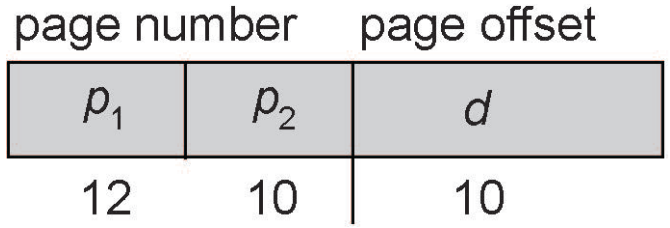
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



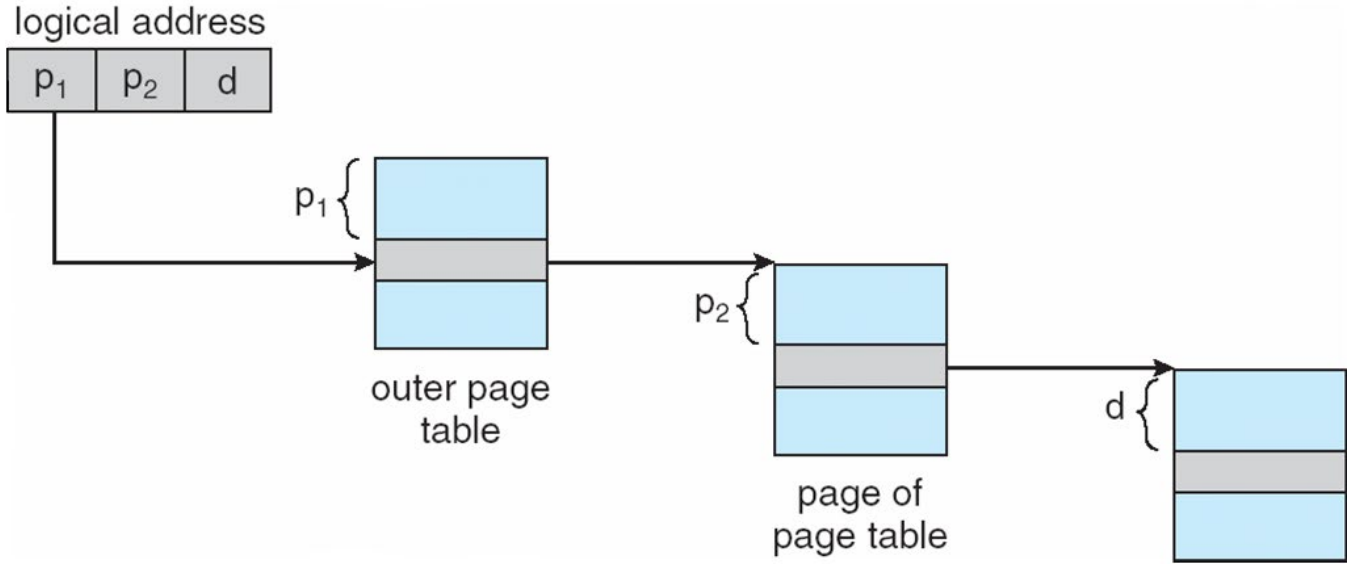
Two-Level Paging Example

- A logical address (on 32-bit machine with 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:



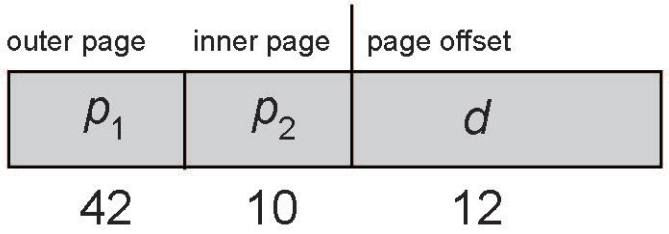
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as forward-mapped page table

Address-Translation Scheme



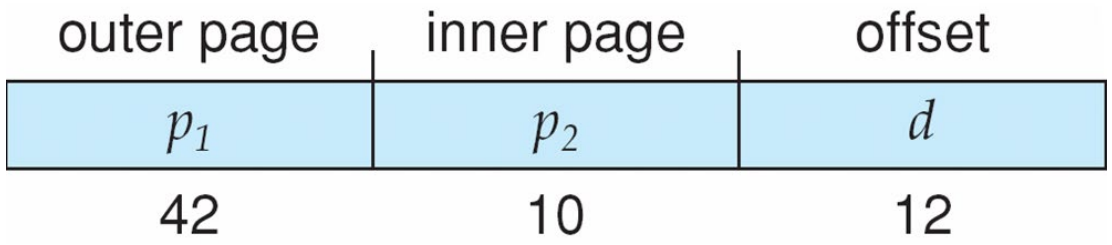
64-bit Logical Address Space

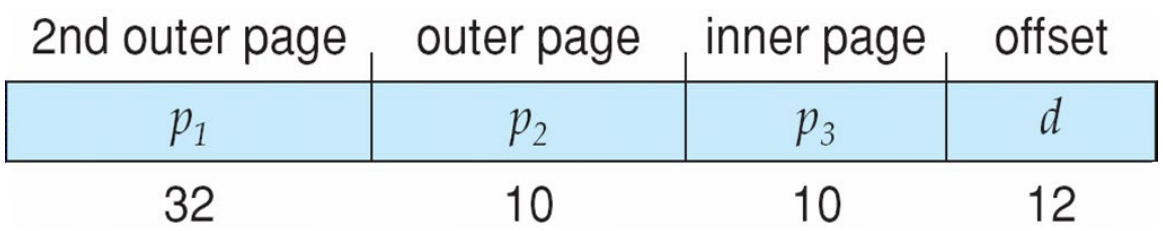
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



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

Three-Level Paging Scheme

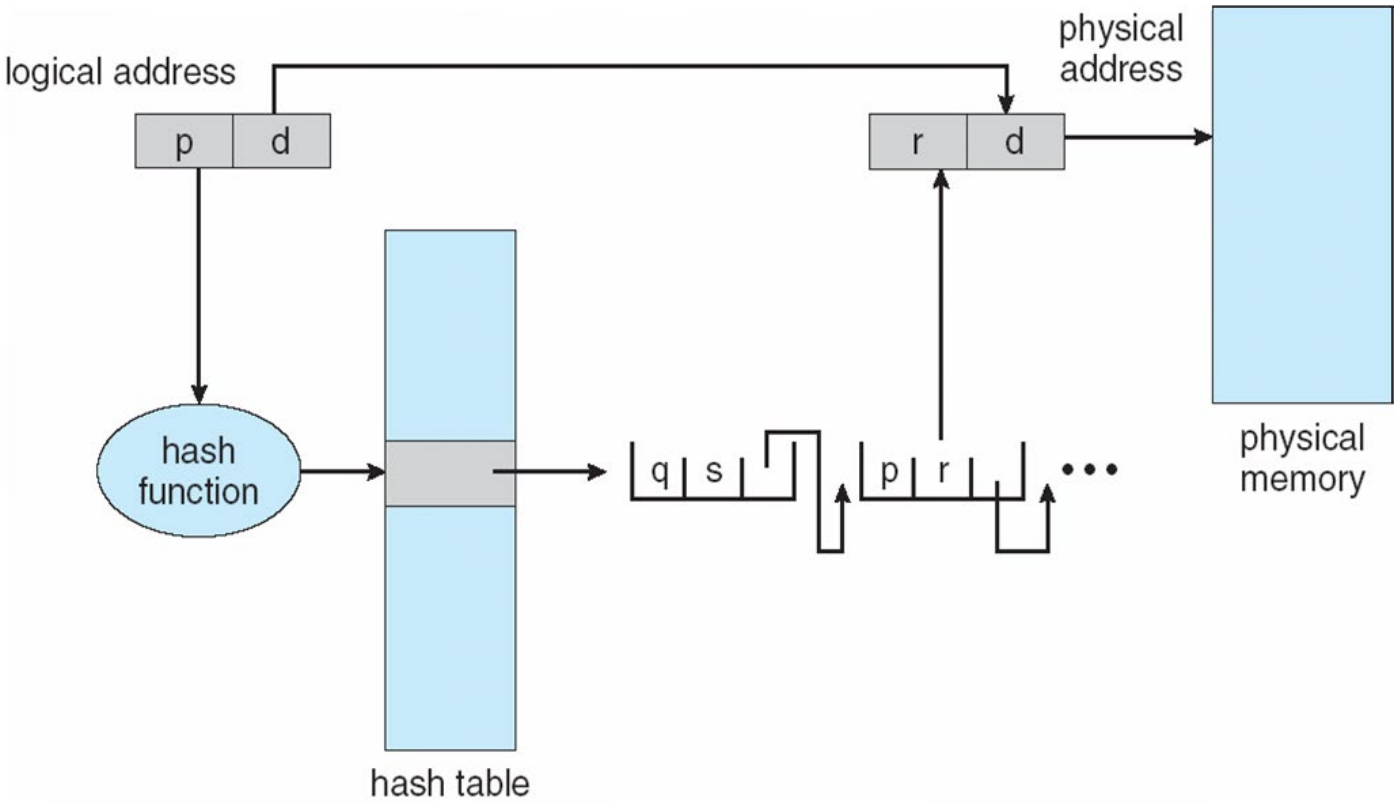




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- 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)

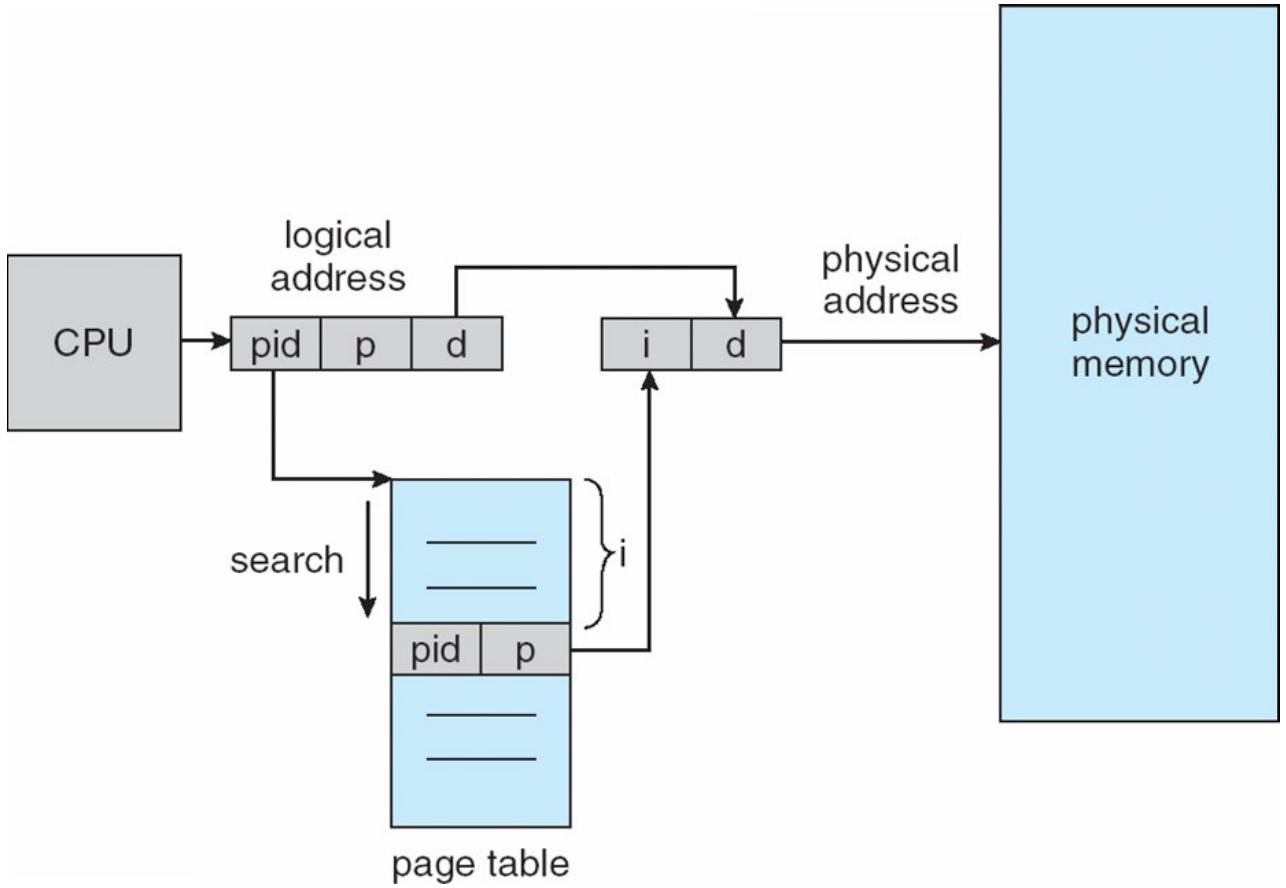
Hashed Page Table



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



Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn’t understand

Next class

- We will continue to discuss:
  - Virtual Memory
- Reading assignment:
  - SGG: Ch. 10

Acknowledgment

- The slides are partially based on the ones from
  - The book site of *Operating System Concepts (Tenth Edition)*: http://os-book.com/