

Main Memory: Part 2

COMP 3659 – Operating Systems

Wednesday, November 10, 2021

Text: Chapter 9 (9.3-9.4)

Announcements and Course Administration

- Programming project 2:
 - Questions?

Overview

- Last lecture:
 - Main memory: background; contiguous memory allocation (text sections 9.1-9.2)
- Continuing this week:
 - A two-week unit on memory management: textbook chapters 9-10
- Today's focuses:
 - Paging 9.3
 - Structure of the page table 9.4

Last Lecture Recap:

- A modern kernel must allocate space to ≥ 1 processes (and itself)
 - While providing memory protection
 - Raises allocation strategy and fragmentation issues
- One “simple” solution: contiguous allocation with base & limit registers
- A program’s instructions & data must be bound to the (logical) address range provided by the kernel
 - Can only be done at compile time if absolute (logical) addresses known a priori
 - Must be done at load and/or execute time otherwise; code must be relocatable
- A hardware MMU supports relocation
 - And better-distinguishes the processes *logical address space* from physical addresses



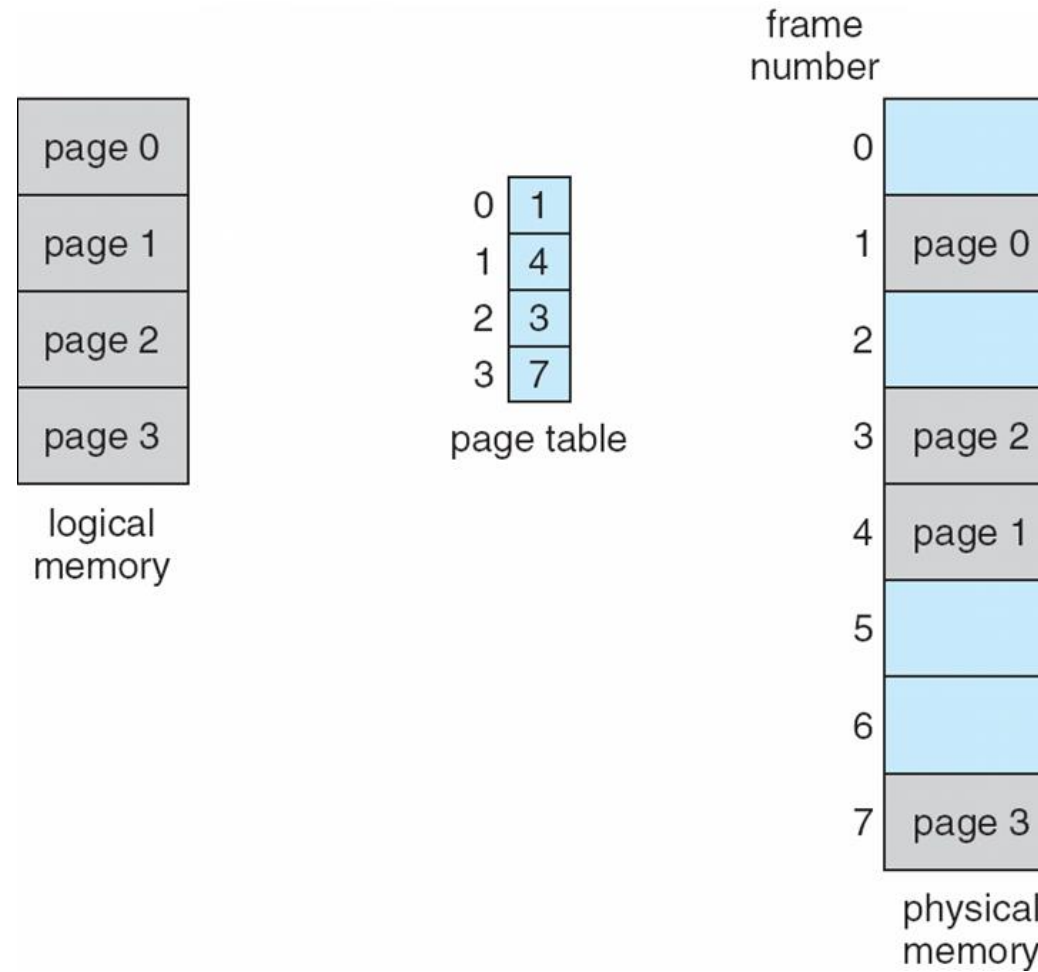
Paging

- 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



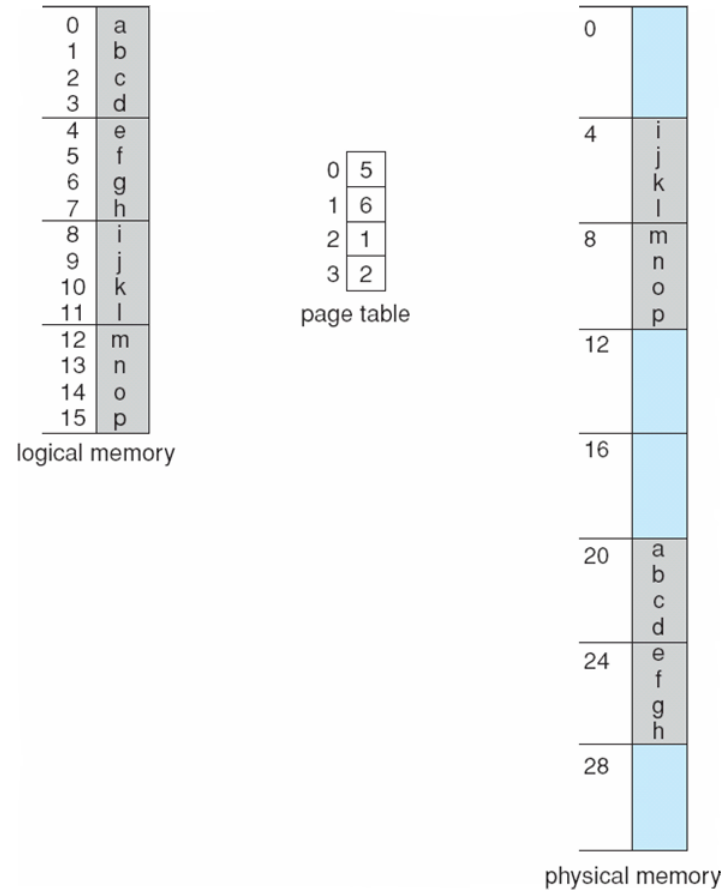


Paging Model of Logical and Physical Memory





Paging Example

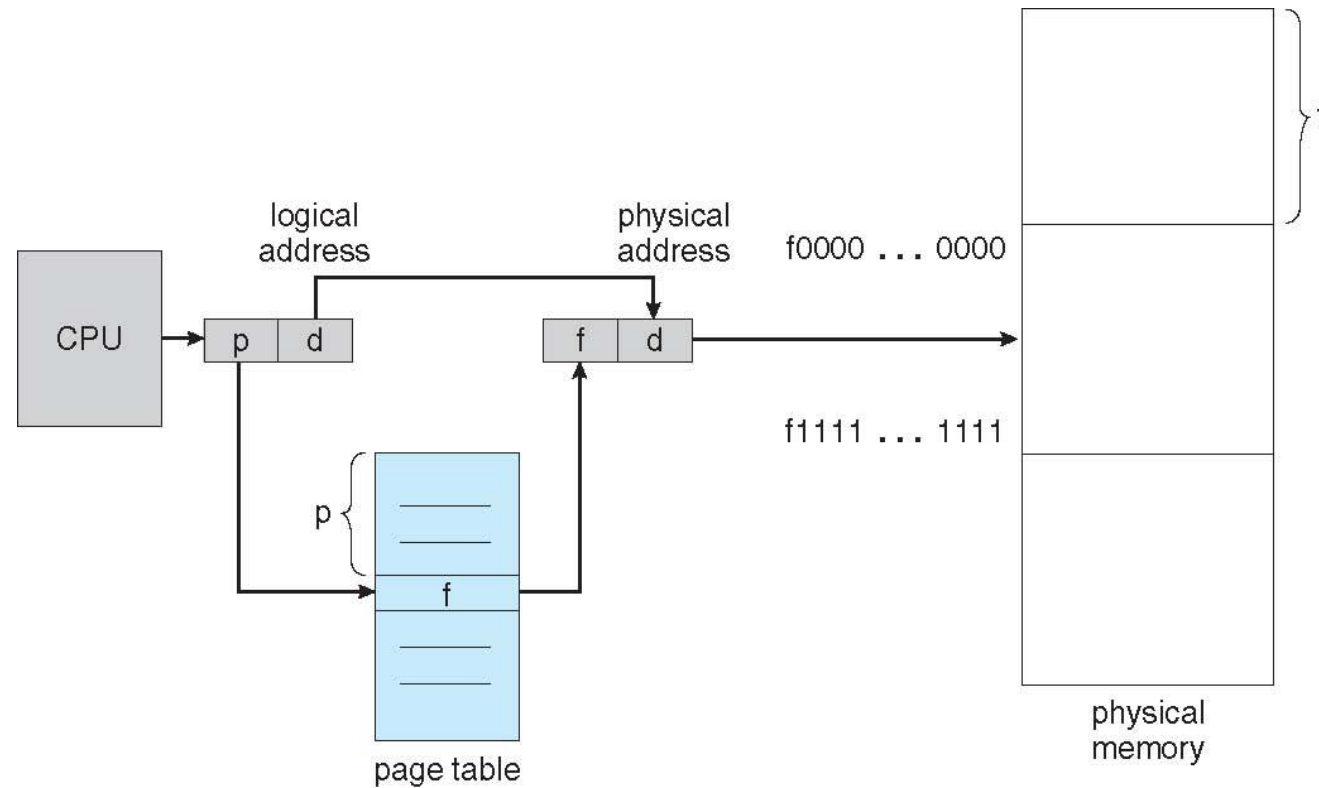


$n=2$ and $m=4$ 32-byte memory and 4-byte pages





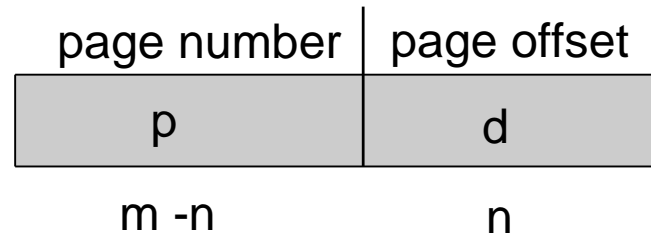
Paging Hardware





Address Translation Scheme

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



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





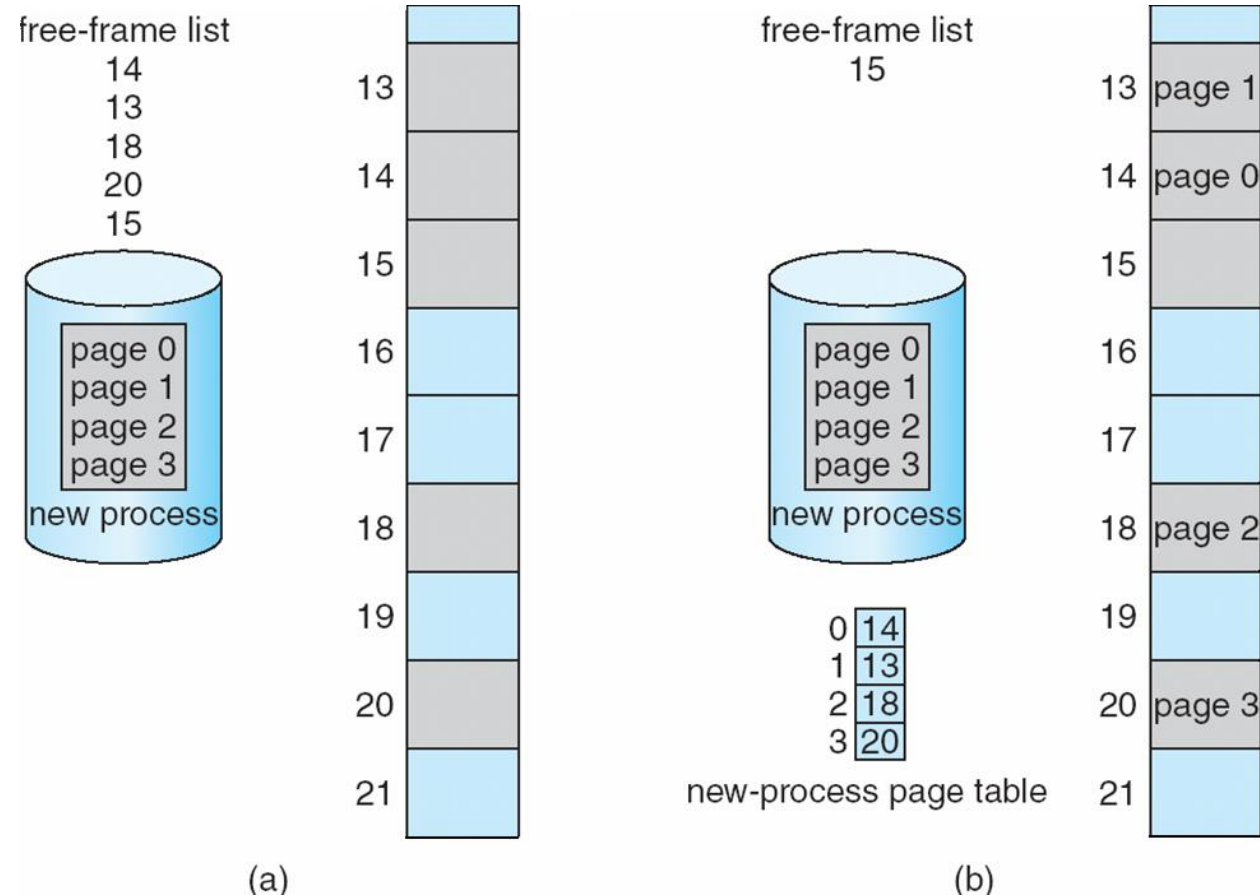
Paging (Cont.)

- 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





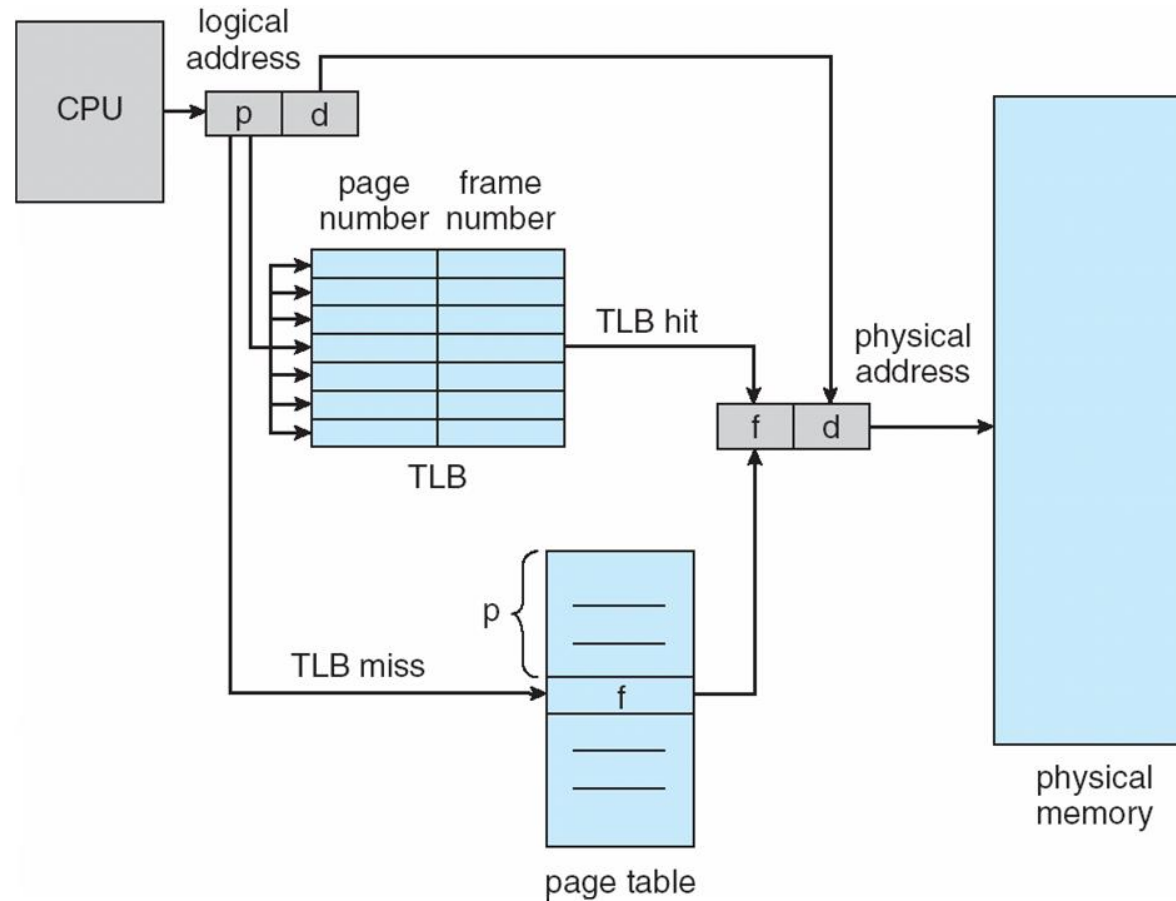
Implementation of Page Table

- Page table is kept in main memory
- **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)**





Paging Hardware With TLB





Implementation of Page Table (Cont.)

- 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
 - Some entries can be **wired down** for permanent fast access





Associative Memory

- Associative memory – parallel search

Page #	Frame #

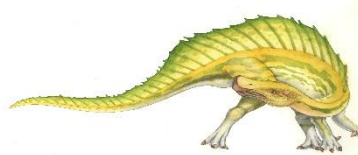
- Address translation (p, d)
 - If p is in associative register, get frame # out
 - Otherwise get frame # from page table in memory





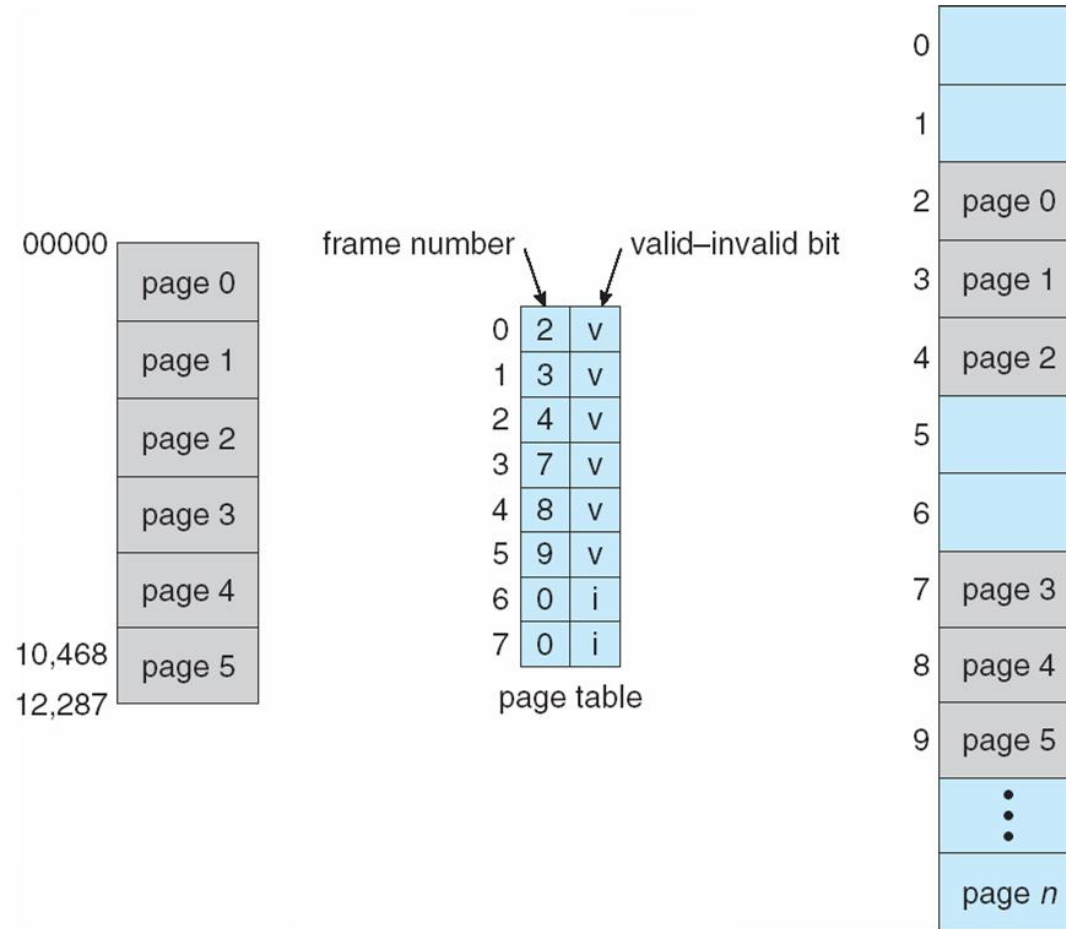
Memory Protection

- 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





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

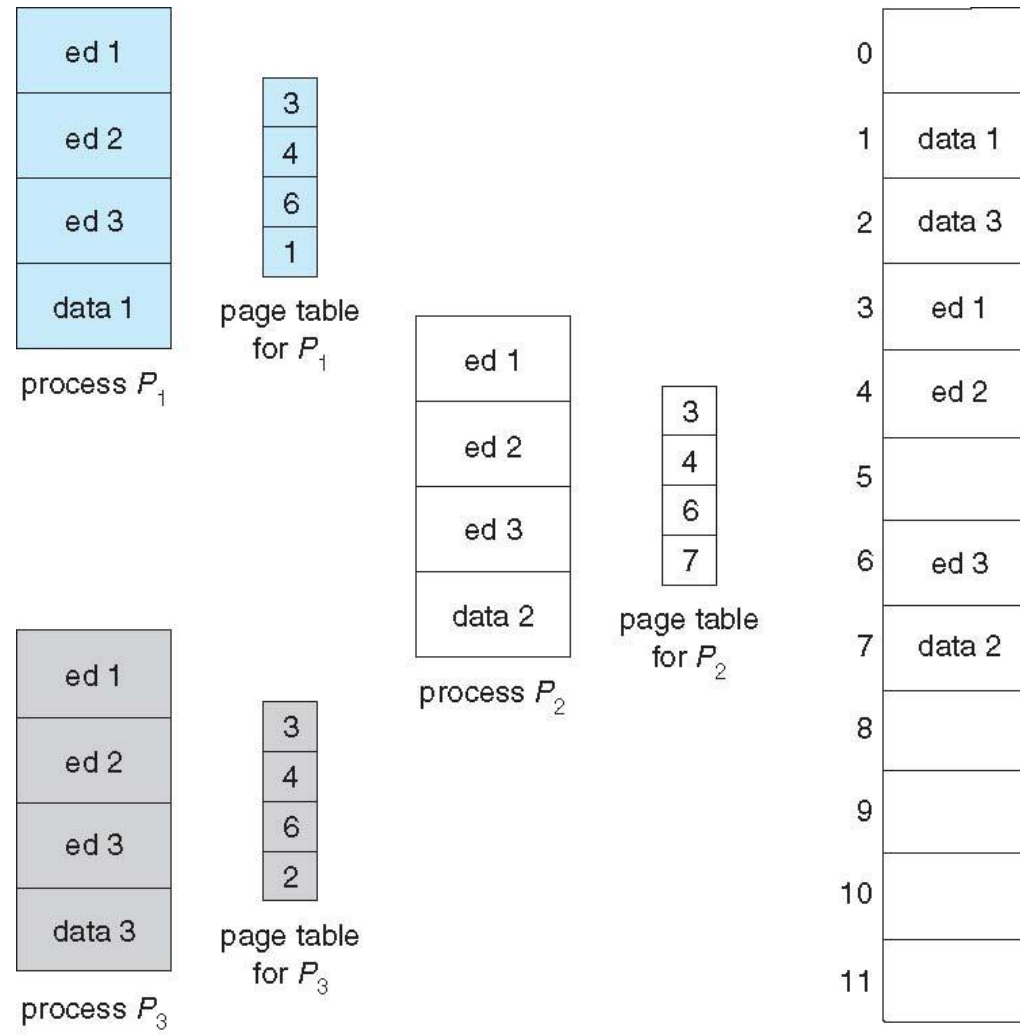
■ Private code and data

- 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





Structure of the Page Table

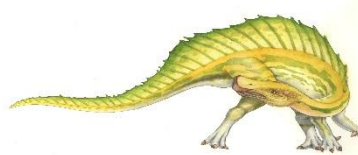
- Memory structures for paging can get huge using straightforward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes -> 4 MB of physical address space / memory 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





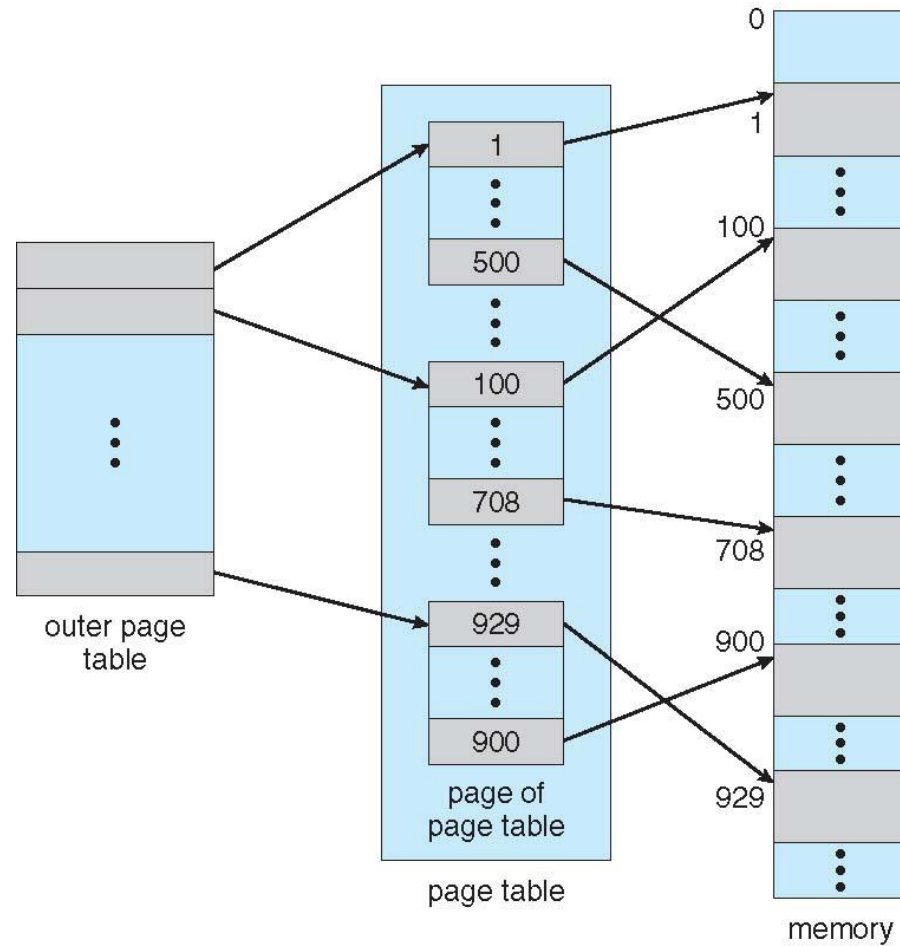
Hierarchical Page Tables

- 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:

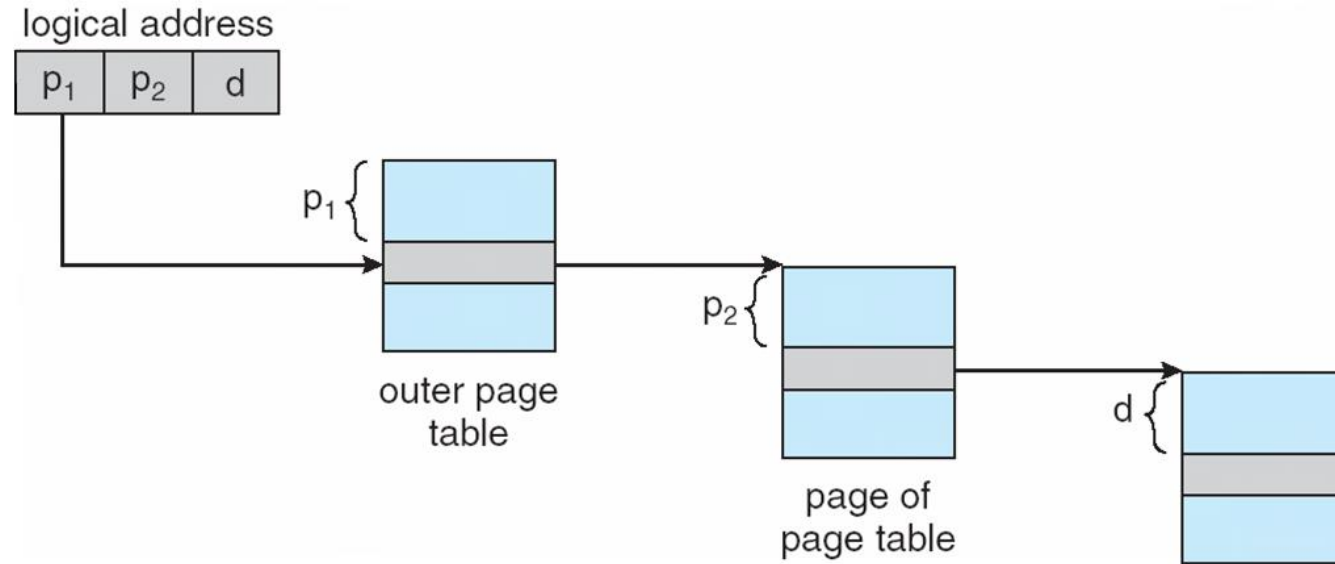
page number		page offset
p_1	p_2	d
12	10	10

- 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**





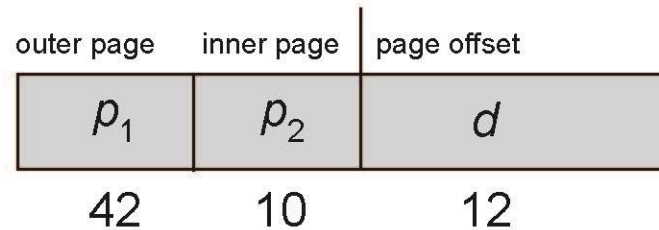
Address-Translation Scheme





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 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location

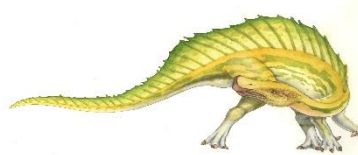




Three-level Paging Scheme

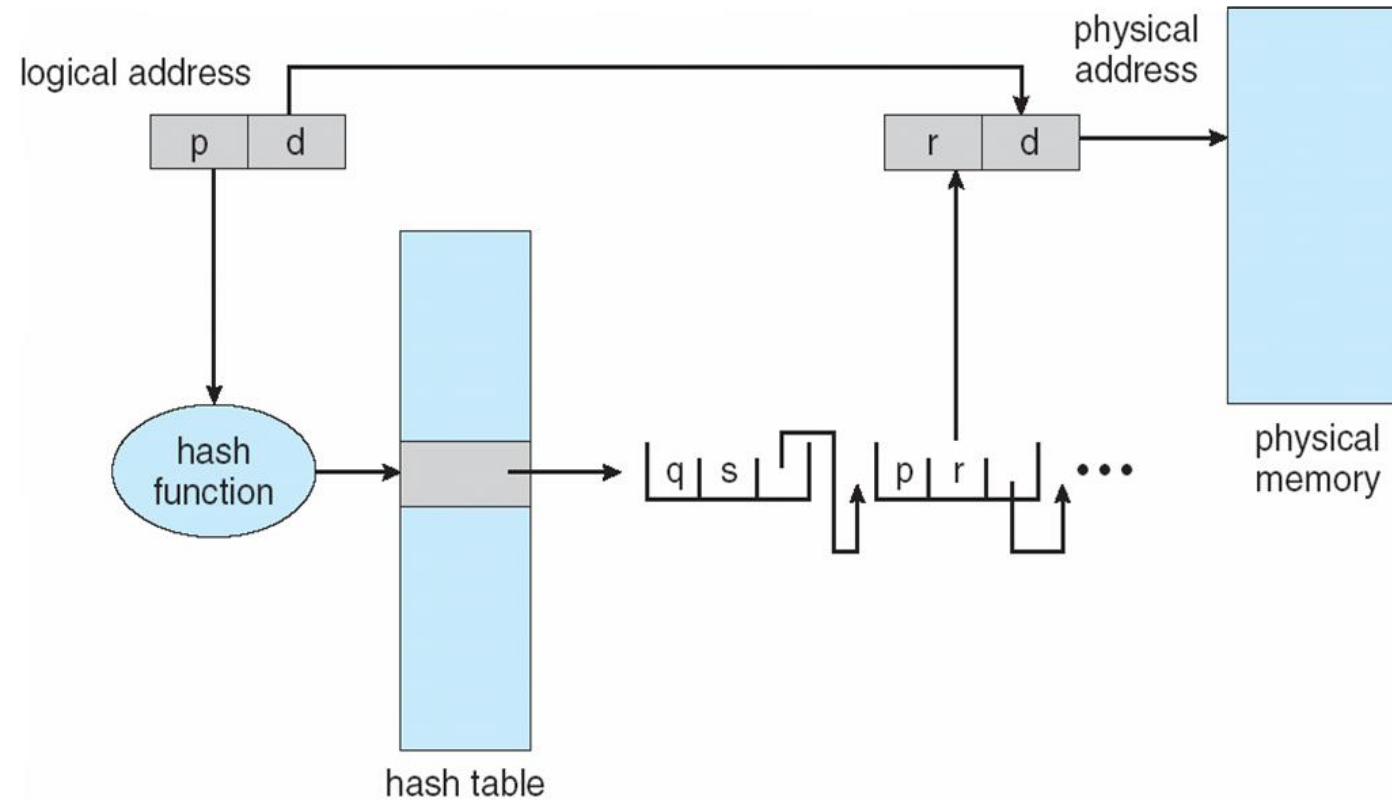
outer page	inner page	offset
p_1	p_2	d
42	10	12

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





Hashed Page Table





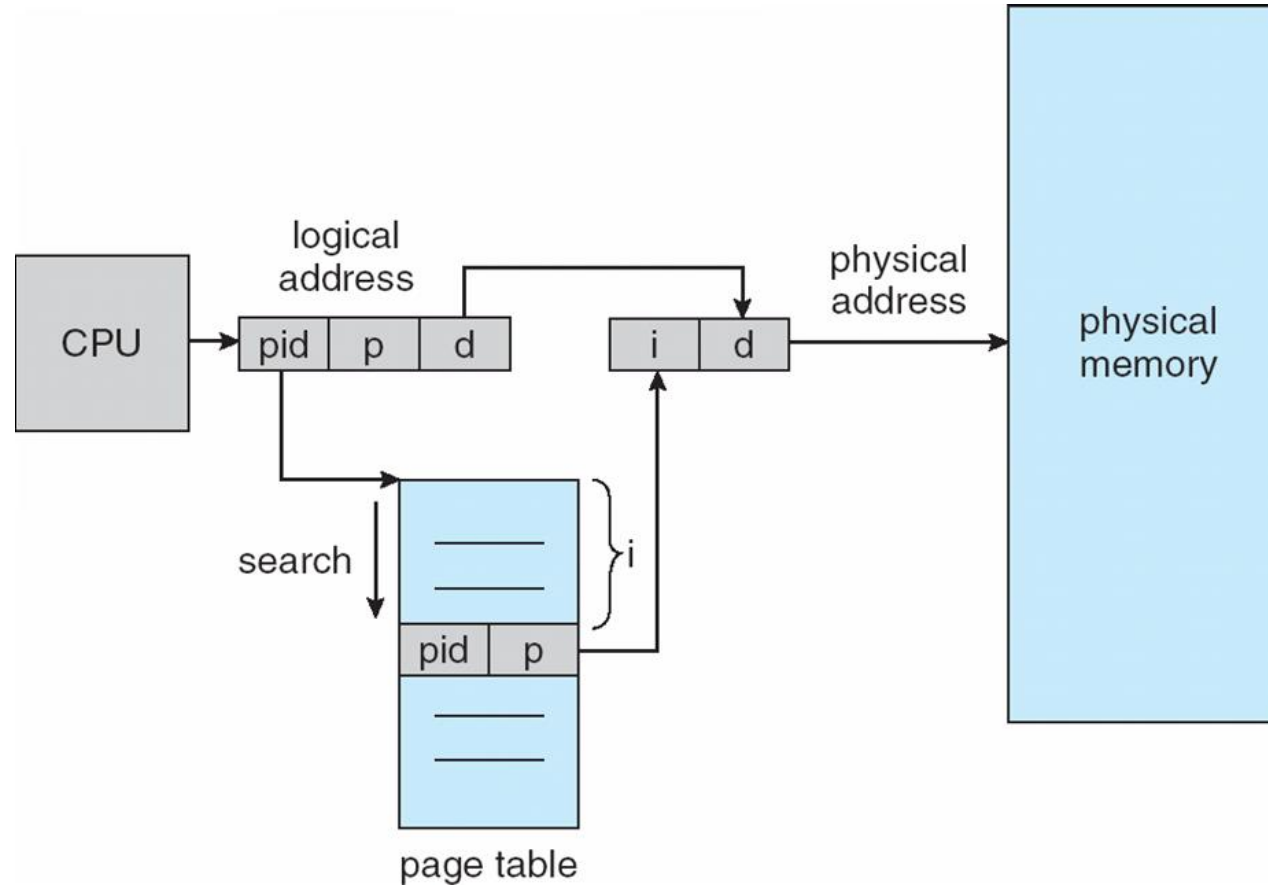
Hashed Page Tables

- 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)





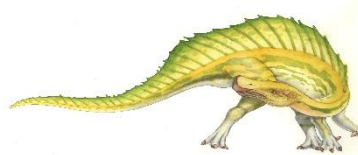
Inverted Page Table Architecture





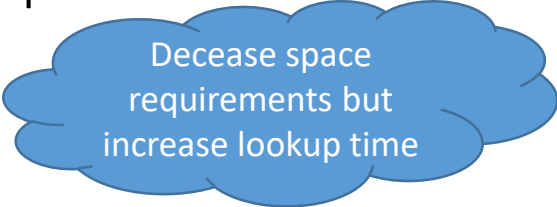
Inverted Page Table

- 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



Conclusion

- Page tables are stored in main memory (by the kernel, per process)
 - By default, would require *two* main memory accesses for *each* CPU-attempted access
 - A TLB is a hardware cache dedicated to fast page → frame lookup
- Page table entries often include additional bits:
 - Valid/invalid page bit (permits more efficient use of physical memory)
 - For enforcing memory protection, e.g., read-only page bit
- Page tables can become unpractically large, e.g., on 32-bit and 64-bit architectures.
Solutions include:
 - Hierarchical page tables (2-, 3-, and 4-level schemes)
 - Hashed page tables
 - Inverted page tables
- Next:
 - Virtual memory: background 10.1
 - Demand paging 10.2
 - Copy-on-write 10.3



Decrease space
requirements but
increase lookup time

Recommended Post-class Studying

- Review these slides and your lecture notes
 - Ideally within 24-48 hours
 - Then review text sections 9.3-9.4 as necessary
- Complete all assigned (pre-)readings for chapter 9
 - If you haven't already
- Review chapter summary section 9.8
- Answer the review questions (“practice exercises”) on pp. 385-386
- As always, please let me know if you have any questions