Module 5.1
Paging

Outline

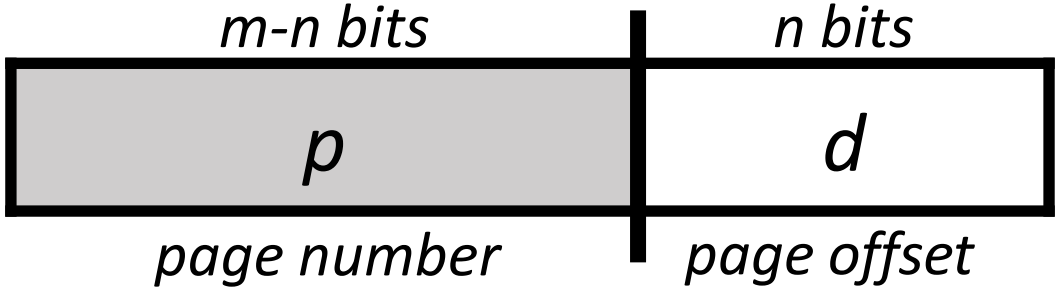
- Non-contiguous allocation
- Paging implementation
- Page table structure

Non-contiguous allocation

- How can we enable the physical address space of a process to be non-contiguous?
  - Allows physical memory to be allocated whenever available
  - Avoids external fragmentation and the problem of varying sized memory chunks
  - Still have internal fragmentation though
- Paging
  - Divide physical memory into **frames**, fixed-size (power of two) blocks from 512 bytes to 1GB
  - Divide logical memory into **pages**, blocks of the same fixed size
  - Build a **page table** to map between pages and frames
- Running a program that needs *N* pages then requires
  - Find *N* free frames
  - Create entries in page table to map each page to a frame
  - Load the program

Address translation

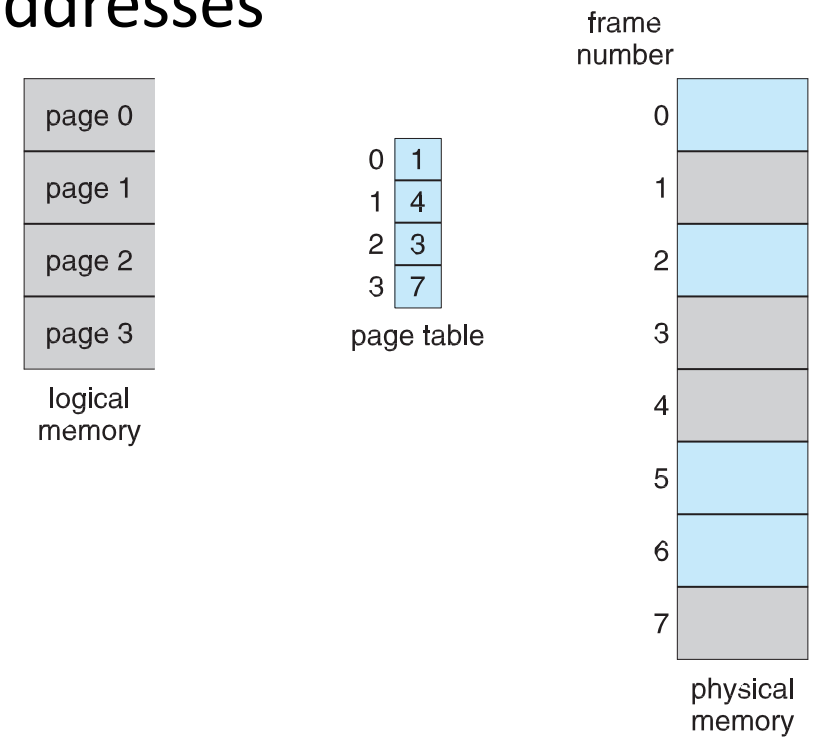
- Divide each logical address generated by the CPU into:
  - **Page number** (p) used as an index into a page table which contains base address of each page in physical memory
  - **Page offset** (d) is combined with base address to define the physical memory address that is sent to the memory unit



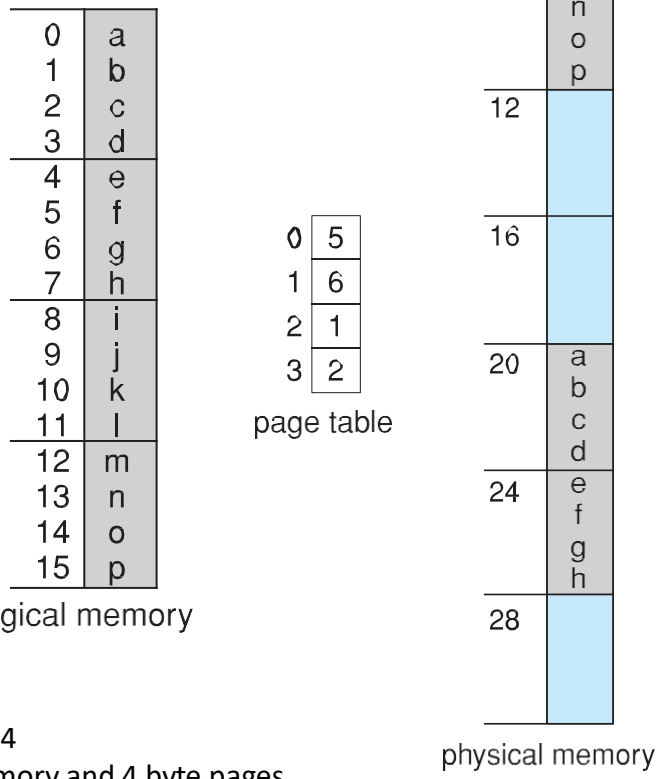
- For given logical address space 2^m and page size 2^n

Paging model

- **Page Table** stores **Page Table Entries** (**PTEs**) that map between logical and physical addresses



- For example,



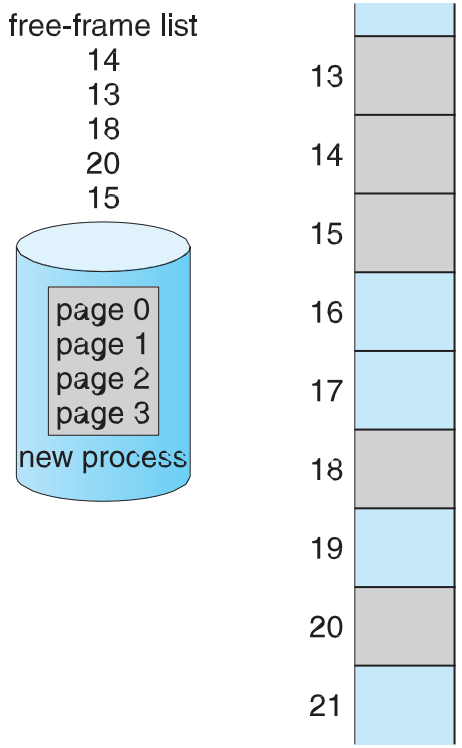
n=2 and m=4
32 byte memory and 4 byte pages

Pros and cons

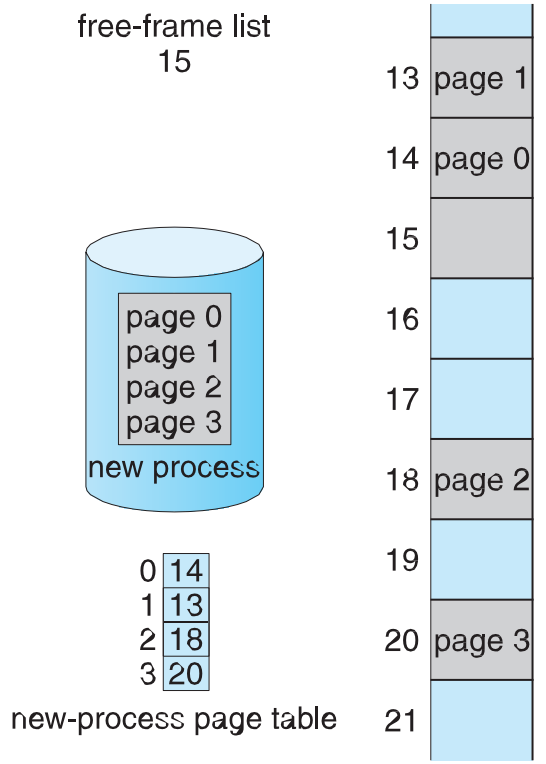
- No external fragmentation but still have internal fragmentation, e.g.,
  - Page size 2048 bytes, process size 72,766 bytes, so process requires 35 pages plus 1086 bytes, so internal fragmentation is 2048 – 1086 = 962 bytes
- On average, fragmentation is ½ frame per process
  - So small frame sizes desirable to waste less
  - But each page table entry takes memory to track so page table grows
- Process view and physical memory now very different
  - OS controls the mapping so user process can only access its own memory
  - OS must track the free frames
  - OS must remap the page table on every context switch – adds overhead

Free frames

- Before allocation, OS has several frames on the free frame list

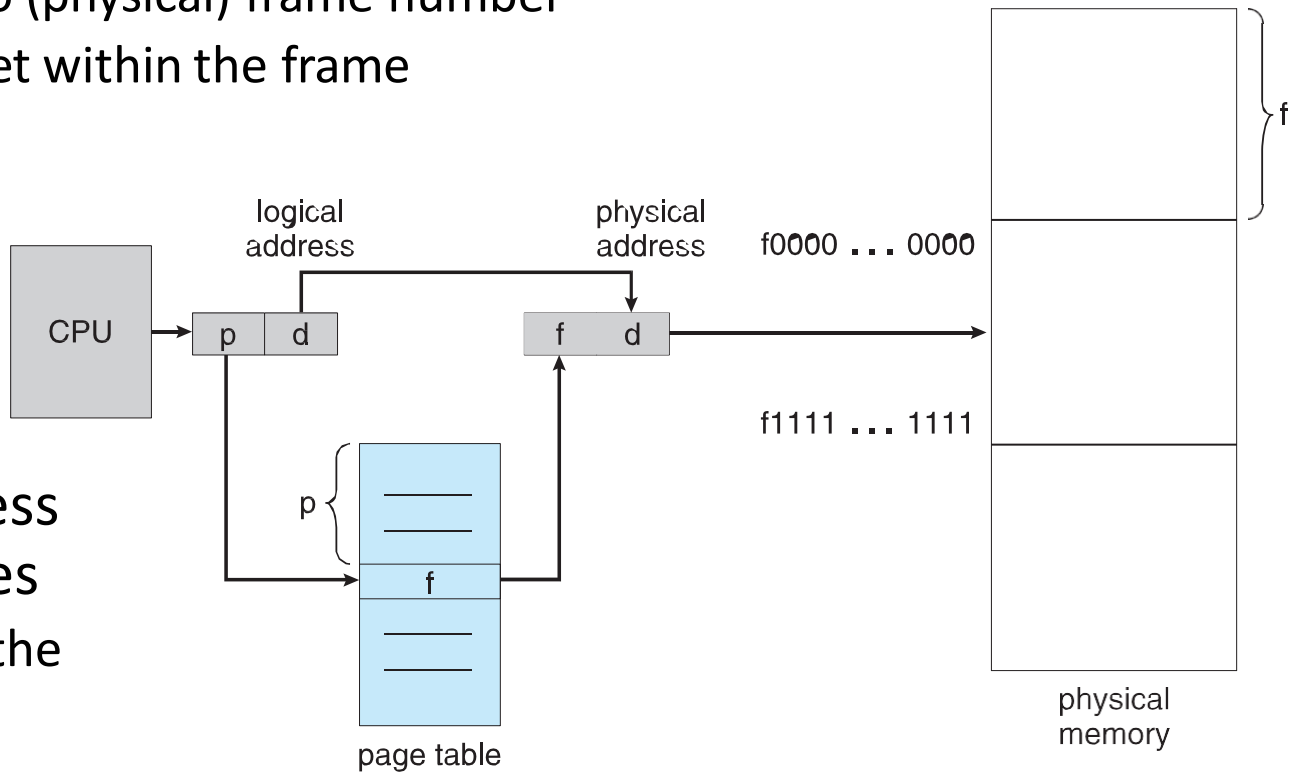


- After allocation, page table entries created and frames no longer in free-frame list



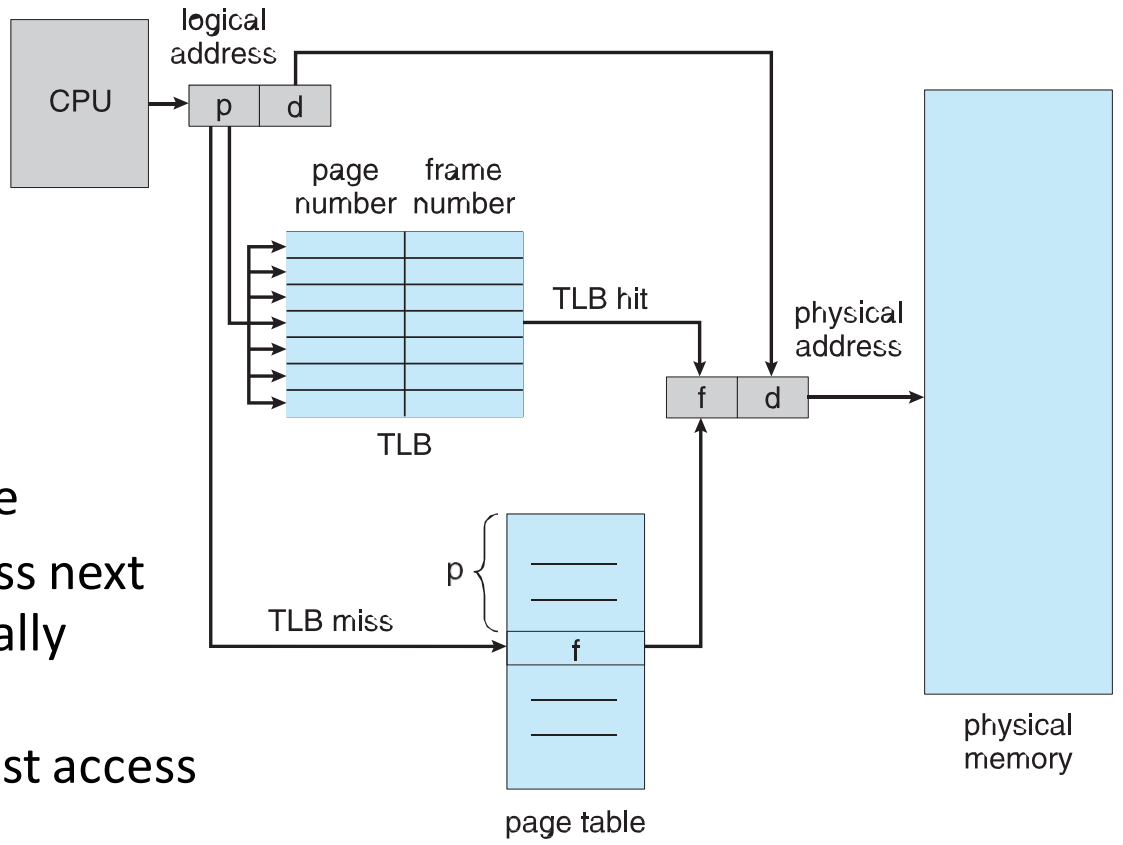
Page table implementation

- Hardware support required for performance
  - Translates (logical) page number into (physical) frame number
  - Offset within a page is then the offset within the frame
- Page table sits in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Means every data/instruction access now requires two memory accesses
  - One for the page table plus one for the data/instruction
  - Dramatically reduces performance



Translation Lookaside Buffer (TLB)

- Resolves the performance issue of two memory accesses
  - Effectively a special hardware cache using associative memory
  - Typically fairly small, 64—1024 entries
- Operation
  - If translation is in the TLB, use it
  - Else we have a **TLB miss** so do the slow two-memory-access lookup in the page table
  - Also add the entry to the TLB for faster access next time subject to replacement policies – typically **Least Recently Used** (**LRU**)
  - Can sometimes pin entries for permanent fast access



TLB performance

- Performance is measured in terms of **hit ratio**, the proportion of time a PTE is found in TLB, e.g., assume
  - TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%
- If one memory reference is required for lookup, what is the **effective memory access time**?
  - 0.8 × 120ns + 0.2 × 220ns = 140ns
- If the hit ratio increases to 98%, what is the new effective access time?
  - 0.98 × 120ns + 0.2 × 220ns = 122ns
  - That is, it only gives a 13% improvement
  - (Intel 80486 had 32 registers and claimed a 98% hit ratio)
- TLB also adds context switch overhead as need to flush the TLB each time
  - Can store address-space identifiers (ASIDs) in each entry to avoid this

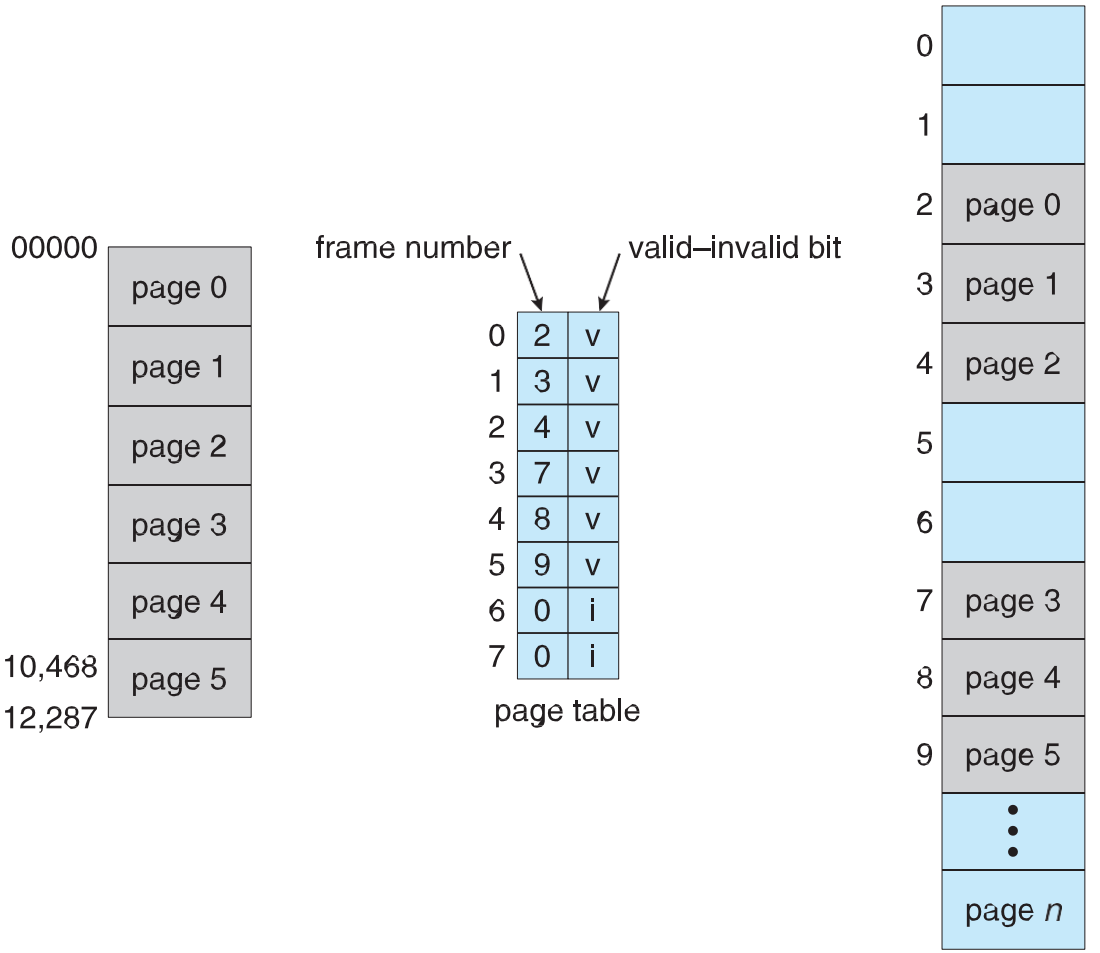
Protection

- Associate **protection bits** with each page, in the Page Table Entry (PTE), e.g.,
  - Accessible in kernel mode only, or user mode
  - Read/Write/Execute to page permitted
  - Valid/Invalid

- As the address goes through the page hardware, protection bits are checked
  - Note this only gives page granularity protection, not byte granularity protection
- Attempts to violate protection cause a hardware trap to the OS
  - TLB entry has the valid/invalid bit indicating whether the page is mapped
  - If invalid, trap to the OS handler to map the page
- Can do lots of interesting things here, particularly with regard to sharing and virtualization

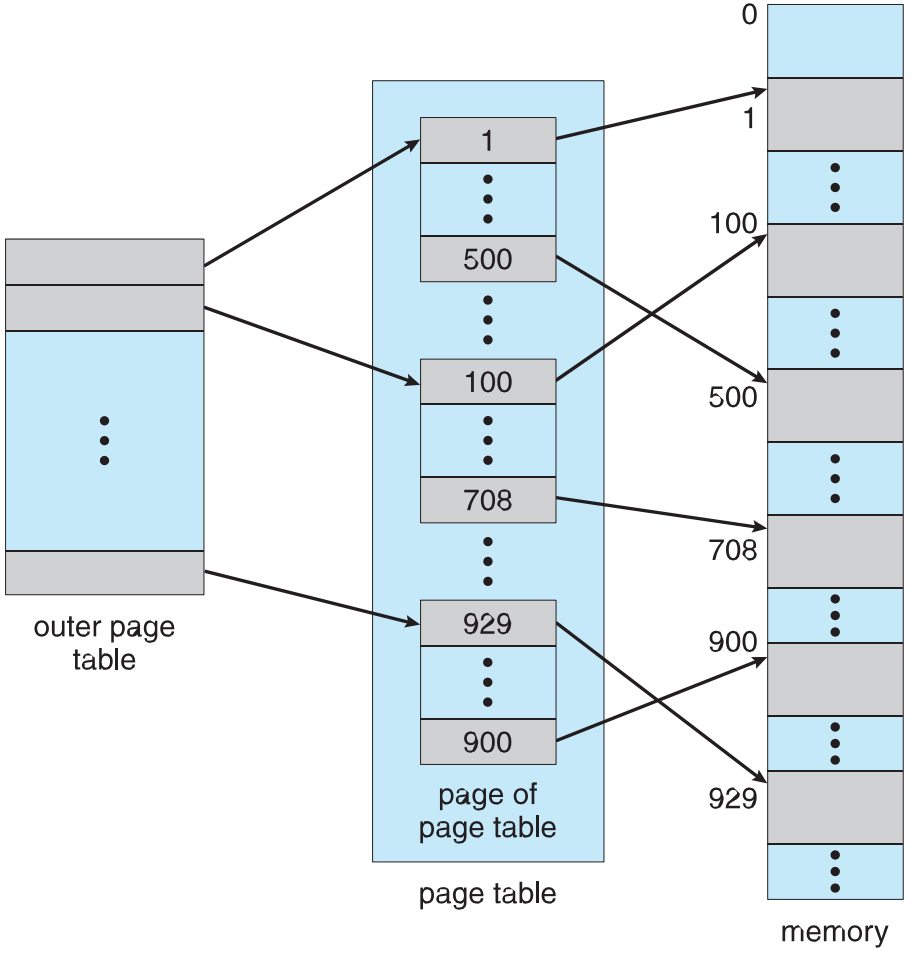
Sharing pages

- Shared code
  - Keep just one copy of read-only (reentrant) code shared among processes
  - Similar to multiple threads sharing the same process space
  - Can also be useful for IPC if read-write pages can be shared
- Private code and data
  - Each process keeps its own copy of private code and data
  - Pages for which can appear anywhere in the logical address space



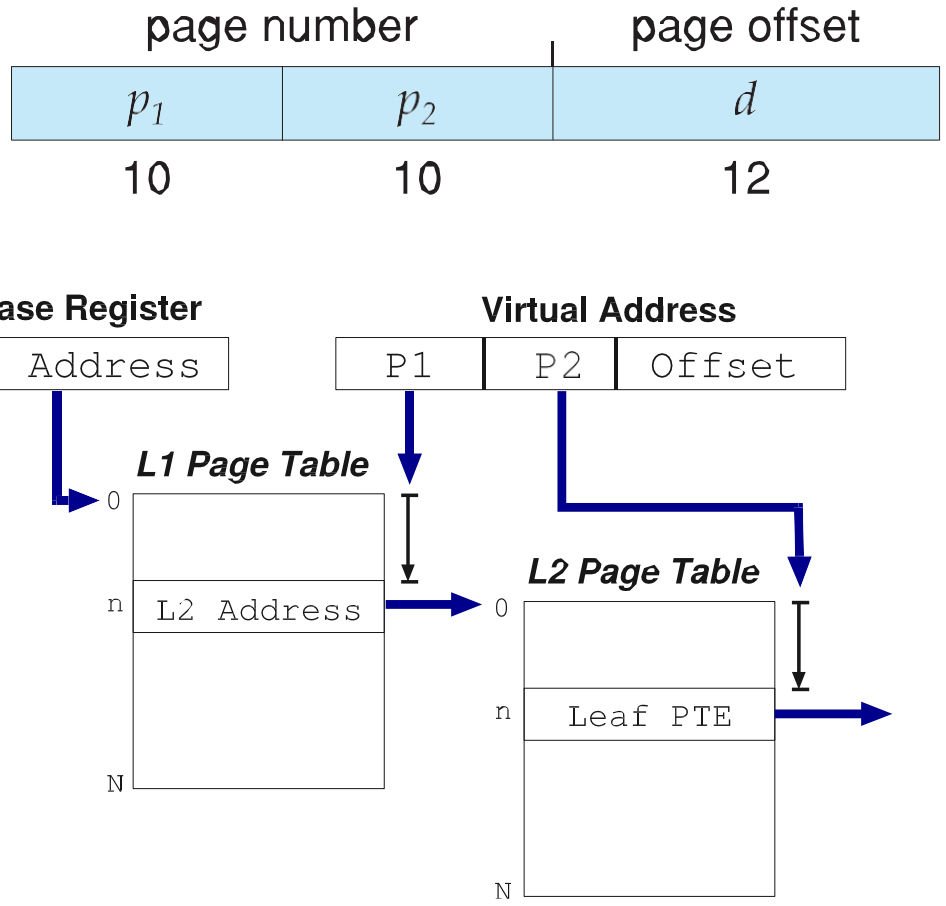
Page table structure

- Page tables can get huge using straight-forward methods
  - E.g., for a 32-bit logical address space and page size of 4 KB (2^{12}), page table would have 1 million entries ($2^{32} / 2^{12} = 2^{20}$)
  - If each entry is 4 bytes that means 4 MB of physical memory for page table – don't want to contiguously allocate that
- Instead, split the page table into multiple levels and page out all but the outermost level



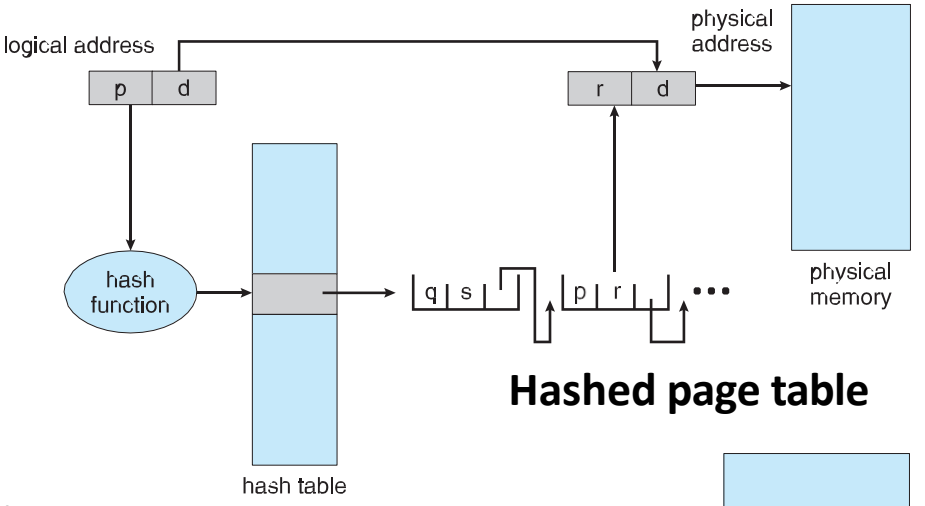
Two-level paging

- For example, given a 20 bit page number and a 12 bit page offset, split the page number into two equal sized parts of 10 bits each
  - NB. A 12 bit offset implies 2^{12} = 4096 byte pages
  - There is no requirement that the two (or more) parts be equal sized
- The PTBR then points to the address of the outermost L1 page table and lookup proceeds by
  - The 10 bit p_1 value indexes into the L1 page table to obtain the address of the relevant page of the L2 page table
  - The 10 bit p_2 value then indexes into the L2 page table to obtain the address of the mapped frame
  - Finally the page offset *d* then indexes into the frame to obtain the intended byte
- This is a **forward mapped** page table

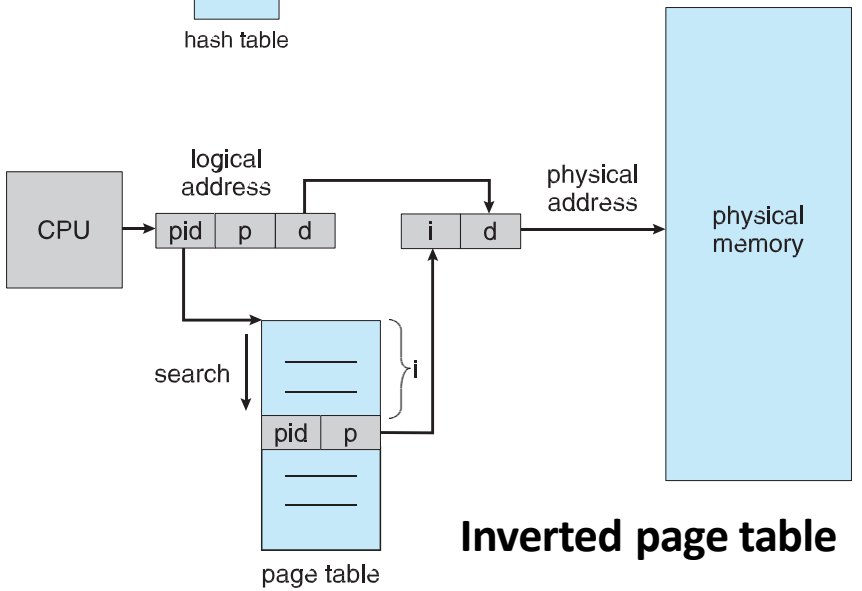


Larger address spaces

- For large address spaces – e.g., 64 bit – simple hierarchy is impractical
  - Either one or more layers remains too large,
  - Or the number of accesses to get to the target address becomes too large
- **Non-examinable** alternatives include
  - **Hashed page tables**, where the page number is hashed into a table and the chain followed until the specific entry is found
  - **Inverted page tables**, with an entry for each frame and a hash-table used to limit the search to one or a few entries, trading size for lookup latency
- Three **non-examinable** practical examples follow: Intel IA-32, Intel x86-64, and ARM



Hashed page table



Inverted page table