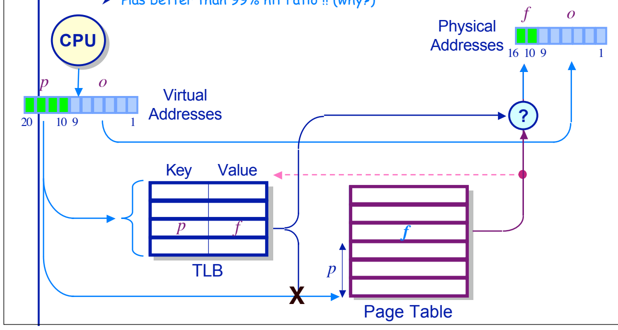
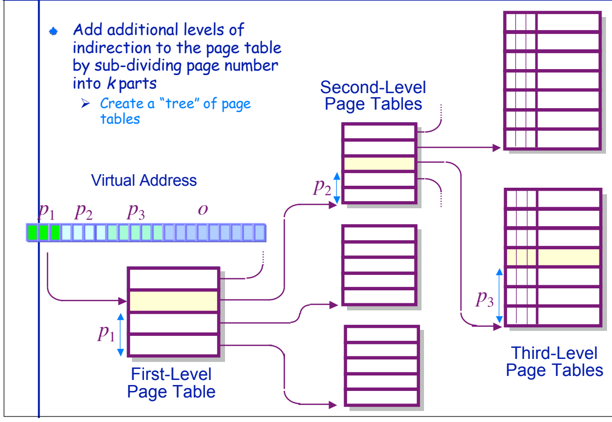
Virtual Address Translation
Using TLBs to Speedup Address Translation

- Cache recently accessed page-to-frame translations in a TLB
  - For TLB hit, physical page number obtained in 1 cycle
  - For TLB miss, translation is updated in TLB
  - Has better than 99% hit ratio !! (why?)



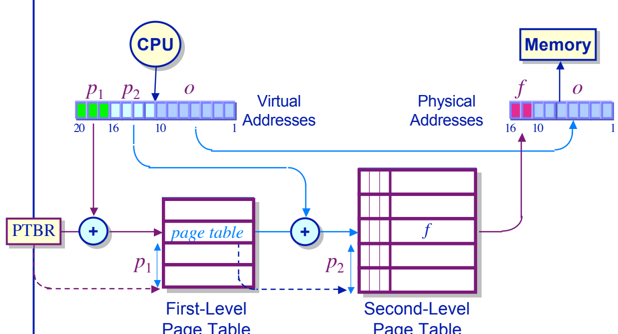
Dealing With Large Page Tables
Multi-level paging

- Add additional levels of indirection to the page table by sub-dividing page number into *k* parts
  - Create a "tree" of page tables



Dealing With Large Page Tables
Multi-level paging

- Example: Two-level paging



Virtual Address Translation
Using Page Registers (aka Inverted Page Tables)

- Each frame is associated with a register containing
  - Residence bit: whether or not the frame is occupied
  - Occupier: page number of the page occupying frame
  - Protection bits
- Page registers: an example
  - Physical memory size: 16 MB
  - Page size: 4096 bytes
  - Number of frames: 4096
  - Space used for page registers (assuming 8 bytes/register): 32 Kbytes
  - Percentage overhead introduced by page registers: 0.2%
  - Size of virtual memory: irrelevant

Page Registers
Tradeoffs

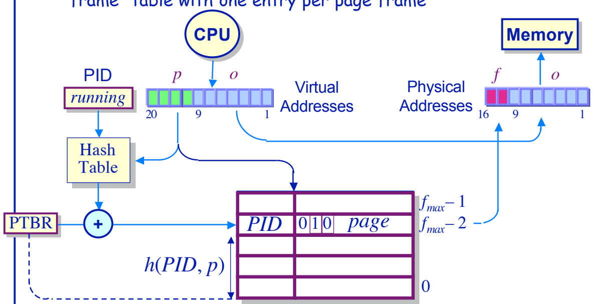
- Advantages:
  - Size of translation table occupies a very small fraction of physical memory
  - Size of translation table is independent of VM size
- Disadvantages:
  - We have reverse of the information that we need….
  - How do we perform translation ?
  - Search the translation table for the desired page number

Inverted Page Tables
Searching for a Virtual Page

- If the number of frames is small, the page registers can be placed in an associative memory
- Virtual page number looked up in associative memory
  - Hit: frame number is extracted
  - Miss: results in page fault
- Limitations:
  - Large associative memories are expensive
  - Memory expansion is non-trivial

Tables
Using Hash Tables

- Use a proven fast search technique: Hash Tables
- Hash page numbers to find corresponding frame numbers in a "frame" table with one entry per page frame



Inverted Page Table

Searching Inverted Page Tables
Using Hash Tables

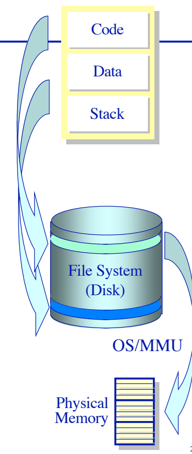
-
- Page registers are placed in an array
- Page *i* is placed in frame *f(i) where f* is an agreed-upon "hashing function"
- To lookup page *i*, perform the following:
  - Compute *f(i)* and use it as an index into the table of page registers
  - Extract the corresponding page register
  - Check if the register contains *i*, if so, we have a hit
  - Otherwise, we have a miss

Searching the Inverted Page Table
Using Hash Tables (Cont'd.)

- Minor complication
  - Since the number of pages is usually larger than the number of slots in a hash table, two or more items *may* hash to the same location
- Two different entries that map to same location are said to collide
- Many standard techniques for dealing with collisions
  - Use a linked list of items that hash to a particular table entry
  - Rehash index until the key is found or an empty table entry is reached
  - …

Virtual Memory (Paging)
The bigger picture

- A process's VAS is its context
  - Contains its code, data, and stack
- Code pages are stored in a user's file on disk
  - Some are currently residing in memory; most are not
- Data and stack pages are also stored in a file
  - Although this file is typically not visible to users
  - File only exists while a program is executing
- OS determines which portions of a process's VAS are mapped in memory at any one time

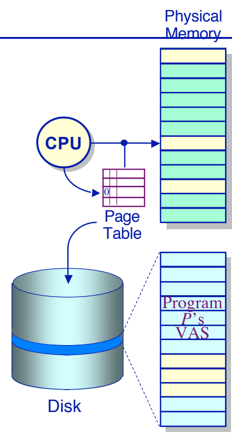


Virtual Memory
Page fault handling

- References to non-mapped pages generate a *page fault*

Page fault handling steps:

Service the fault
Block the running process
Read in the unmapped page
Resume/initiate some other process
Map the missing page into memory
Restart the faulting process



Virtual Memory Performance
Page fault handling analysis

- To understand the overhead of paging, compute the *effective memory access time* (*EAT*)
  - *EAT* = *memory access time* × *probability of a page hit* + *page fault service time* × *probability of a page fault*
- Example:
  - Memory access time: 20 *ns*
  - Disk access time: 25 *ms*
  - Let p = the probability of a page fault
  - $EAT = 20(1-p) + 25{,}000{,}000p$
- To realize an *EAT* within 5% of minimum, what is the largest value of p we can tolerate?

Virtual Memory
Summary

- Physical and virtual memory partitioned into equal size units
- Size of VAS unrelated to size of physical memory
- Virtual *pages* are mapped to physical *frames*
- Simple placement strategy
- There is no external fragmentation
- Key to good performance is minimizing page faults