

Virtual Memory

Paging (cont'd)
Inverted Page Tables

Topics

- Inverted Page Tables

Review

- Single-level paging
 - Single page table contains one entry per virtual page per process
- Multilevel paging
 - Several tables
 - Higher level tables contain pointers to page tables (Pentium III – Page Directory Table)
 - Lowest level tables contain virtual to physical address mapping (Pentium III – Page Tables)
 - Advantage: Don't need to allocate large chunk of memory for whole page table (if implemented with single level), if only a few pages are actually active. Only need to allocate enough chunks of memory to hold the segments of page table that are active.

Inverted Page Table - Motivation

- Example:
 - 64-bit virtual address space
 - 4 KB page size
 - 512 MB physical memory
- How much space (memory) needed for a single level page table?

Single Level Page Table

- One entry per virtual page
 - 2^{64} addressable bytes / 2^{12} bytes per page = 2^{52} page table entries
- Page table entry size
 - One page table entry contains:
 - Access control bits + Physical page number
 - 512 MB physical memory = 2^{29} bytes
 - 2^{29} bytes of memory / 2^{12} bytes per page = 2^{17} physical pages
 - 17 bits needed for physical page number
 - Page table entry = ~4 bytes
 - 17 bit physical page number = ~3 bytes
 - Access control bits = ~1 byte
- Page table size
 - 2^{52} page table entries * 2² bytes = 2^{54} bytes (16 petabytes)

A lot of memory! (kilo-, mega-, giga-,tera-, peta-) PER PROCESS

64-bit address space
4 KB page size
512 MB physical mem

How About Multilevel Paging?

- How many levels are needed to ensure that any page table requires only a single page (4 KB)?
 - Assume page table entry is 4 bytes
 - 4 KB page / 4 bytes per page table entry = 1024 entries
 - 10 bits of address space needed
 - ceiling(52/10) = 6 levels needed

6 levels → 6 memory accesses → slow

2^{52} page table entries (single-level)

Observation

- 512 MB physical memory
 - 2^{29} bytes / 2^{12} byte pages = 2^{17} physical pages
- Is there a way to store a single page table entry per **physical page**?
 - Can reduce the page table size to **2 MB**
 - Assuming 16 byte entries
 - 16 bytes because need to map **physical address to virtual address**
 - (page table entry contains virtual page number instead of physical page number)
 - $2^{17} * 2^4 = 2^{21}$ bytes = 2 MB

Inverted Page Table Only need one global page table

Inverted Page Table

- **Linear inverted page table**
 - Page table entry contains
 - **Process ID** (table shared between all processes)
 - **Virtual** page number
 - Table indexed by **physical page number**
- **Example:**
 - Assume 16 bits for process ID, 52 bit virtual page number, 12 bits of access information
 - 80 bits = 10 bytes
 - 10 bytes * 2^{17} bytes ≈ 1.3 MB

Accessing Inverted Page Table

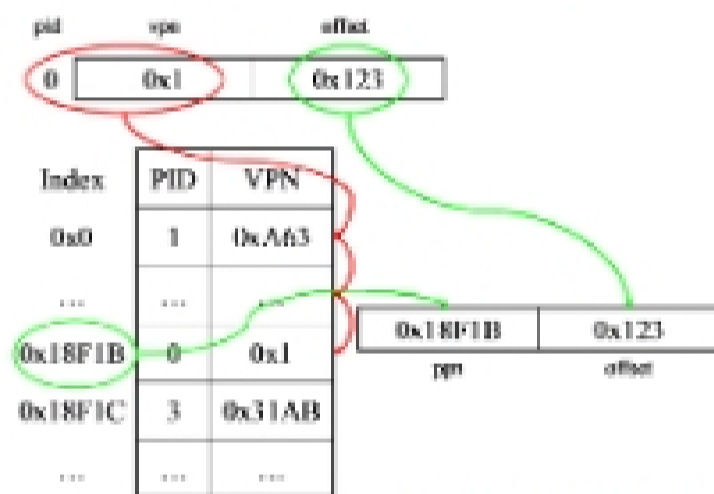


Image from <http://www.cs.berkeley.edu>

Accessing Inverted Page Table

- For each entry in the inverted page table, compare process ID and virtual page number in entry to the requested process ID and virtual page number
- **Linear time.** Up to 2^{17} memory accesses possible for our example
 - SLOW!
 - Expect on average about half this number of accesses

Hashed Inverted Page Tables

- Linear inverted page tables require too many memory accesses.
- Keep another level before actual inverted page table (**hash anchor table**)
 - Contains a mapping of process ID and virtual page number to page table entries
 - Use separate chaining for collisions
- Lookup in hash anchor table for page table entry
 - Compare process ID and virtual page number
 - if match, then found
 - if not match, check the next pointer for another page table entry and check again

Accessing Hashed Inverted Page Table

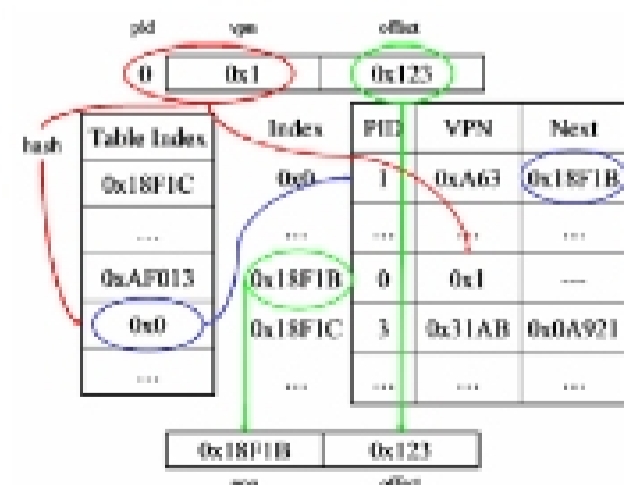


Image from <http://www.cs.berkeley.edu>