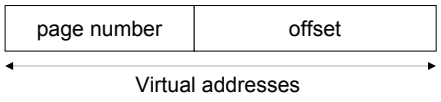
Memory Management -3

CS 342 – Operating Systems

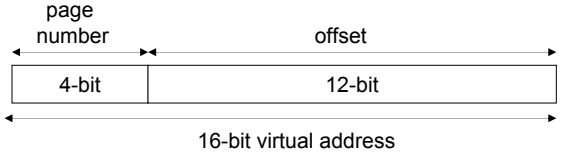
Ibrahim Korpeoglu
Bilkent University
Department of Computer Engineering

Page Tables

- Mapping of virtual addresses to physical addresses
  - Split virtual address into two parts
    - Virtual page number (high-order) bits
    - Offset (low-order bits)



Example:



Page Table

- *Virtual page number* is used an index into the page table.
  - The entry at that index contain information about the virtual page
    - Physical page frame number
    - Bit indicating whether the page is in memory or not.
    - …
  - The physical address is obtained by attaching *offset* to the *page frame number*.

Page Table - Issues

- Two important issues that need to be considered
  1. The page table can be extremely large
  2. The mapping must be fast
- First Issue:
  - Modern computer have virtual addresses that are at least 32 bits long.
    - Assume page size of 4K. Then we need $2^{32}/2^{12} = 220 =$ 1 million page table entries.
    - Each processes need its own page table (every process has its own virtual address space).

Page Table - Issues

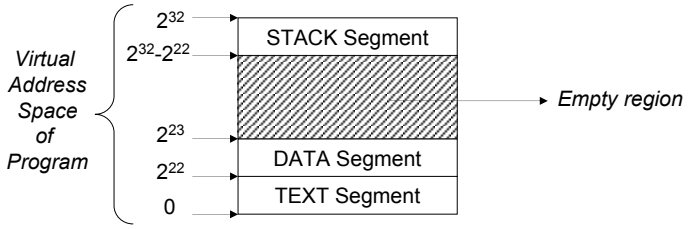
- Second Issue
  - Virtual to physical address mapping needs to be done at every memory reference.
  - Therefore its has to be very *fast*.
  - Some *approaches*:
    1. Have the page table in registers during program execution.
       - Requires loading of registers from main memory with every context switch.
    2. Have the page table entirely in main memory.
       - Requires more memory accesses per instruction.

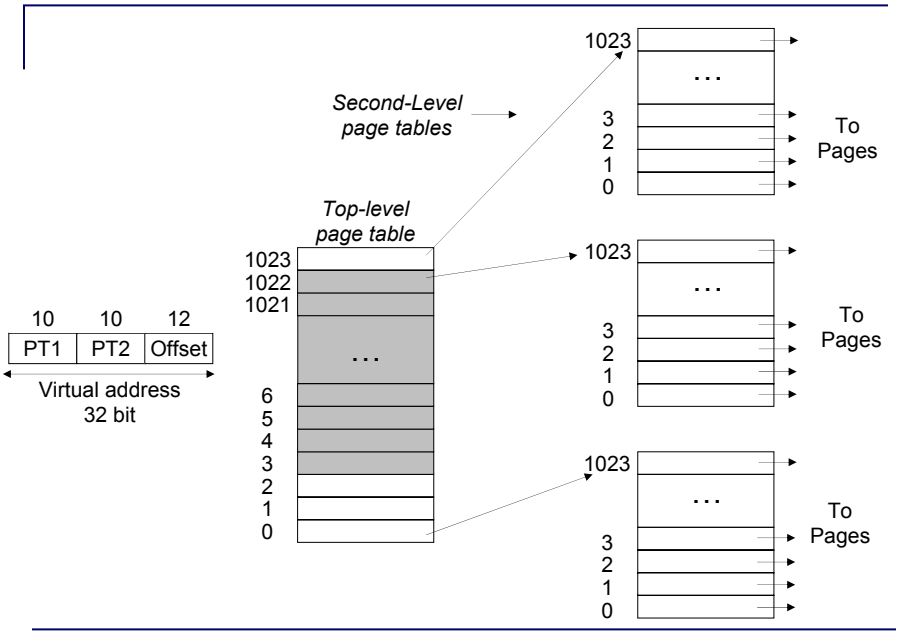
Multi-Level Page Tables

- An approach to have huge page table size.
  - Page table is not stored in memory all the time.
- Divide the virtual addresses into more than 2 parts
  - 1st level page table part (PT1)
  - 2nd level page table part (PT2)
  - Offset

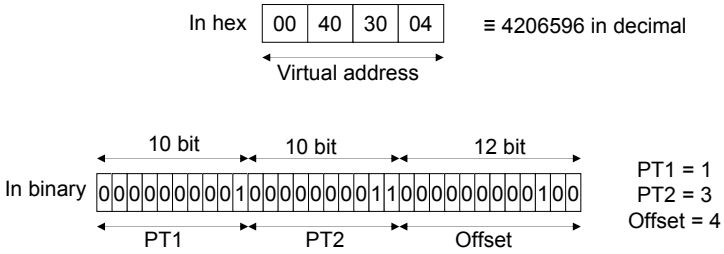
Example

- Assume we have a program that consists of 3 parts
  - Text (needs 4 MB = 2^{22} bytes)
  - Data (needs 4 MB)
  - Stack (needs 4 MB)
- The layout of the program inside its address space is shown as below:



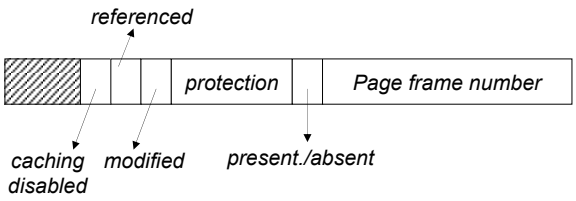


Example – virtual address is 0x00403004



-First look to the top-level page entry. Look to entry 1, get the address of the corresponding secondary page table
-Look to the secondary page table. Look to entry 3, get the frame number (address) of the memory location where the actual data is stored.
-Go to that location in memory and retrieve the value (data).

Structure of a page table entry



- ***Protection***: Tells what kind of access is permitted (read-only, read/write, etc)
- ***Modified***: When page needs to be reclaimed (removed from memory), then if modified bit is set, it is written back to disk, otherwise there is no need to write it back.
- ***Referenced***: When a page is accessed, this bit set. It is used in page replacement algorithms
- ***Present/absent***: Tells if the page is in the memory or not. If not in memory, a page fault occurs.
- ***Caching disabled****:* doe not allow the content of the page to be cached in L2 or L1 cache. Useful for memory mapped I/O
- ***Page frame number***: The physical page frame number corresponding to a location in physical main memory.

TLB – Translation Lookaside Buffer

- Accessing memory is slow compared to CPU speed.
  - Therefore if the page table kept in memory, an instruction will require several memory access during its execution time.
- Solution to cache some of the page table entries in a fast memory called *TLB* (or *associative memory*)
  - TLB is located usually in MMU.
  - Contains number of entries that can range from 8 to 64.
- Most programs make large number of references to a small number of pages.
  - Locality of reference.

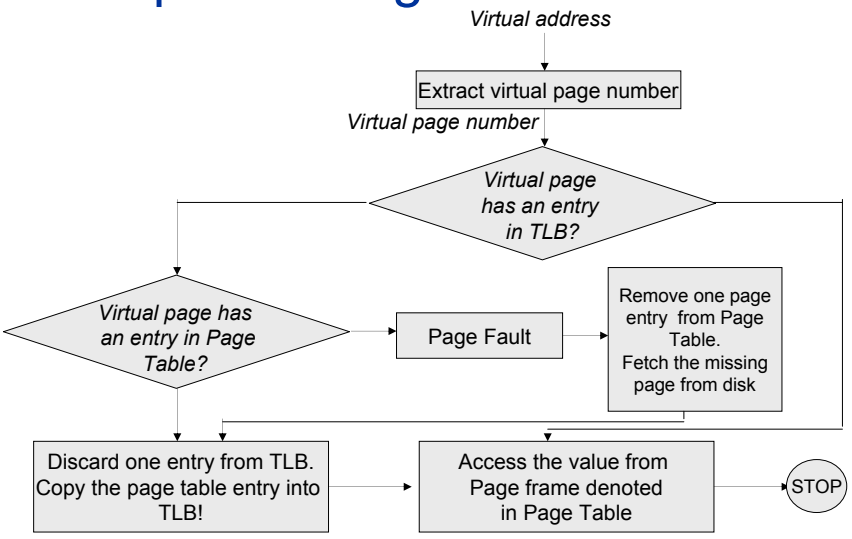
TLB

- A TLB entry contains info about
  - Virtual page number
  - Page frame number
  - A bit saying if page is modified
  - Protection code
  - A bit saying if entry is valid or not.

TLB

Valid	Virtual Page	Modified	Protection	Page Frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TBL processing in MMU

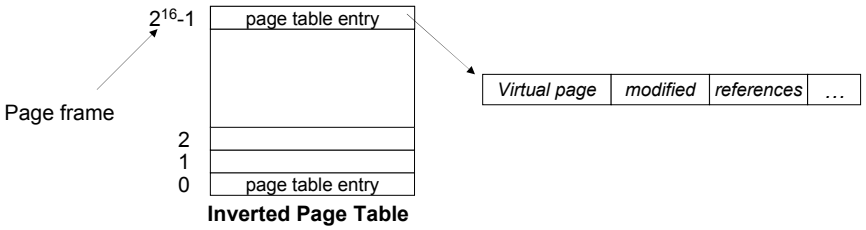


Inverted Page Tables

- For large address spaces (64-bit machines), page table will have enormous number of entries.
  - Assume 4KB page size.
  - We need $2^{64}/2^{12} = 2^{52}$ entries.
  - Assume each entry is 8 bytes
    - We need 2^{55} bytes of storage, which is over 30 GBs.
- A solution is use of *inverted page tables*.
  - We have one entry per page frame.
  - The table is indexed by the *page frame number.*

Inverted Page Tables

- Assume
  - address space of 2^{64}
  - page size of 4KB.
  - Physical memory size of 256 MB.
- Number of entries required = $(256x2^{20})/4x2^{10}$ = 64K



Inverted Page Tables

- Issue is, how to search for a virtual page
- Solution: use hash table to store the entries of page table.
  - Implement the table as a hash table.
- Hash based on the virtual page number
- If more than one virtual page number map to the same hash table position, then use separate chaining collision resolution technique.

Inverted Page Tables – implemented as a hash table