Preview

- Virtual Memory with Paging
  - Page Table with Hardware Support
    - Translation Look-Aside Buffer
  - Page Table Structure
    - Shared Pages
    - Multilevel Page Table
    - Hashed Page Table
    - Inverted Page table

Page Table with Hardware Support*

- OS maintains a page table per a process. A pointer to the page table is stored in the process table (process control block)
- When short term scheduler selects a process for execution, its page table must be loaded.
- The hardware implementation of the page table can be done in several ways.
  - The page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch – it can be applied with small size page table only.
  - The page-table based register is used to save a pointer to a page table which is kept in main memory. Changing page table requires only change the content of this register

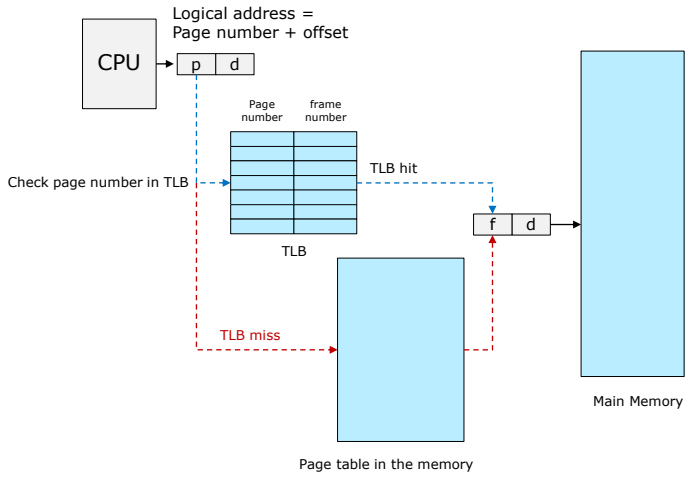
Page Table with Hardware Support
(Translation Look-Aside Buffer)

- Maintaining page table in memory can result in slower memory access times.
  - To access a instruction located in virtual address i, system need two memory access times
    - Access a memory to get page frame number from page table in the memory
    - Now calculate physical address by combining page frame number + offset
  - Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances.
  - The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a translation look-aside buffer (TLB). Each entries in TLB consists of two parts: a key (or tag) and a value.
  - The TLB is used with page table by saving a few of page table entries.

Page Table with Hardware Support
(Translation Look-Aside Buffer)

- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB. If the page number is found, its frame number is immediately available (hit) and MMU can map physical address which is used to access memory.
- If the page number is not in the TLB (known as a TLB miss), address translation proceeds following the steps.
  - Search the page table and get a page frame number to map physical address.
  - The search result will be added to the TLB, so that they will be used quickly on the next reference.
  - If the TLB is already full of entries, an existing entry must be selected for replacement based on replacement policies(Least Recently Used (LRU), Round-Robin or random)

Page Table with Hardware Support
(Translation Look-Aside Buffer)



Page Table with Hardware Support
(Translation Look-Aside Buffer)

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio. An 80% hit ratio means that we find the desired page number in the TLB 80 % of the time.
- If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds with hit.
- If it miss to find out desired pages in TLB, then need two memory access: access memory for page table frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds)
- Effective access time with 80% hit ratio

  = 0.8 x 10 + 0.2 x 20 = 12 nanoseconds

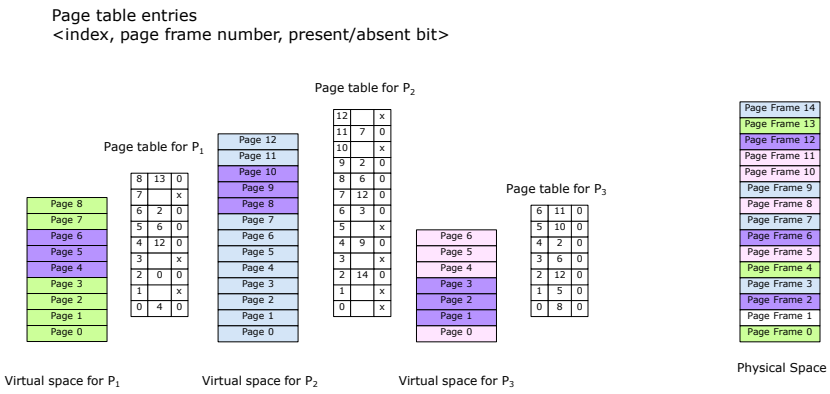
Page Table with Hardware Support
(Translation Look-Aside Buffer)

- CPUs today may provide multiple levels of TLBs.
- the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB.
- A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

Page Table
(Shared Pages)

- An advantage of paging is the possibility of sharing common code, a consideration that is particularly important in an environment with multiple processes.
  - The Standard C library provides a portion of system call interface for many version of UNIX (Linux).
  - On a typical Linux system, most user processes require the standard C library libc. If code in a library is reentrant code, it can be shared by multiple processes. (Reentrant code is non-self-modifying code: it never changes during execution) In stead of each process load its own copy of libc into its address space, one copy can be shared by multiple process.
  - Also, heavily used programs can also be shared—compilers, window systems, database systems, and so on.

Page Table
(Shared Pages)



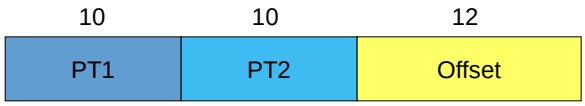
One copy of frames in Physical space are shared by P_1, P_2 and P_3

Structure of the Page Table
(Multilevel Page Table)

- The basic idea of multilevel page table method is to avoid keeping all the page tables in memory all the time!!!

Ex) A system is using second level page tables

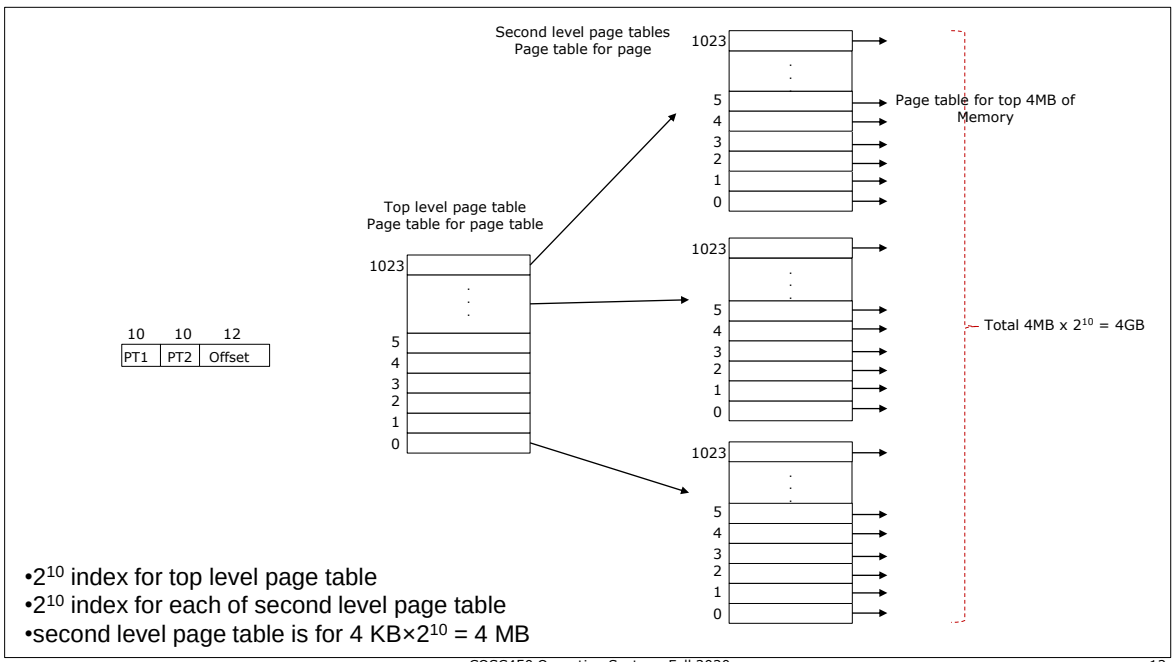
- The system generates 32 bit virtual address with page size 4KB.
- We have a 32 virtual address that is partitioned into 10-bit PT1 field and 10-bit PT2 field and a 12-bit offset field (for page size 4KB).
- Since offset are 12 bits, pages are 4KB and there are total 2^{20} of them.



Structure of the Page Table
(Multilevel Page Table)

Example continue)

- First level page table is the page table for page table
  - First level page table has 2^{10} entries, there are 2^{10} page tables
  - Present/absent bit is used in the first level page entries used to save whether a page table is in the memory or not
- The second level page tables are page tables for pages
  - Each second level page table has 2^{10} entries, and a page size is 4KB, each second level page table support $2^{10} \times 4$ KB = 4 MB virtual space.
  - There are 2^{10} second level page table, this example system support 4MB $\times$ 2^{10} = 4GB virtual memory!



Structure of the Page Table
(Multilevel Page Table)

Ex) virtual address 403004_{16}
($=00000000010000000011000000000100_2 = 4206596_{10}$)

- PT1 = $0000000001_2 = 1_{10}$
- PT2 = $0000000011_2 = 3_{10}$
- Offset = $000000000100_2 = 4_{10}$

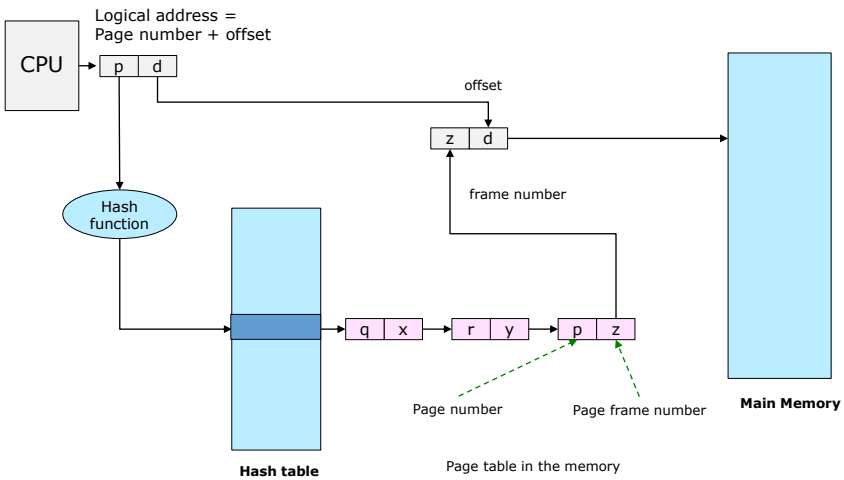
Structure of the Page Table
(Multilevel Page Table)

- The MMU uses PT1 to index into the top-level page table and obtain entry 1 (0000000001)
  - (physical address between 4M ~ 8 M).
- Then the MMU uses PT2 to index into the second-level page table and exact entry 3 (0000000011)
  - (Corresponds to address 12 KB ~ 16 KB within 4M chunk = 12288 ~ 16383) = (obsolete address between 4M + 12K ~ 4M + 16K –1 = 4206592 ~ 4210687)
- If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4 =000000000100) to construct a physical address.

Structure of the Page Table
(Hashed Page Table)

- One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
- Each element consists of three fields:
  - the virtual page number,
  - the value of the mapped page frame,
  - a pointer to the next element in the linked list.

Structure of the Page Table
(Hashed Page Table)



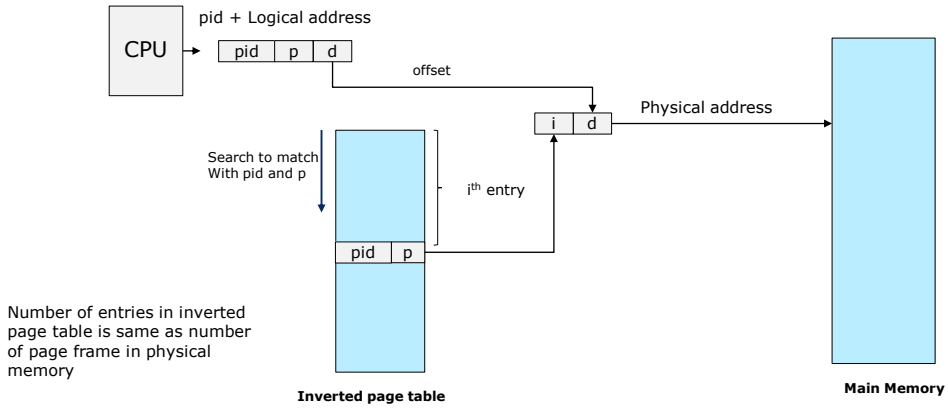
Structure of the Page Table
(Inverted Page Table)

- Inverted Page Table structure consists of one-page table entry for every frame of the main memory.
  - The number of page table entries in the Inverted Page Table is the number of page frames in physical memory.
  - A single page table is used to represent the paging information of all the processes.
  - With the inverted page table, the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together.

Structure of the Page Table
(Inverted Page Table)

- Each entry in the page table contains the following fields
  - Virtual page number
  - Process id
  - Control bits
- Given a process ID and a virtual page number, search the inverted page table for a match.
- If the match is found at the i^{th} entry then this page is located in page frame number i. Then, calculate physical address to access memory.

Structure of the Page Table
(Inverted Page Table)



Page table in the memory

Structure of the Page Table
(Inverted Page Table)

- Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
- Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found.
- To alleviate this problem, we can use a hash table. But, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table

Structure of the Page Table
(Inverted Page Table)

- One issue with inverted page tables involves shared memory.
- With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address.
- Since there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time.
- A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.