Virtual Memory

• Main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks — this technique is called virtual memory. Two major motivations for virtual memory were
i) to allow efficient and safe sharing of memory among multiple programs, such as for the memory needed by multiple virtual machines for cloud computing, and
ii) to remove the programming burdens of a small, limited amount of main memory.

< First reason is still more important today)

• To allow virtual m/cs to share the same memory, one must protect the virtual machines from each other (so that a program can read and write the portions of main memory that have been assigned to it). Main memory needs to contain only the active portions of the virtual machines (just as a cache contains only the active portion of one program). Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to efficiently share the processor as well as the main memory.

• One cannot know which VMs will share the memory with other virtual machines (VMs) when we compile them. In fact, the VMs sharing the memory change dynamically while the VMs are running. Due to this dynamic interaction, one wishes to compile each program into its own address space, which is a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to physical addresses. This translation process enforces protection of a program's address from other virtual machines.

• The second motivation for virtual memory is to allow a single user program to exceed the size of primary memory. Earlier, if a program became too large (to fit into) main memory, programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These overlays were loaded or unloaded under user program control during execution, with the programmer ensuring

that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying (replacement) of one module with another. Virtual memory (which was invented to relieve programmers of this difficulty), automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called physical memory to distinguish it from virtual memory) and secondary storage

- Although the concepts applicable to virtual memory and caches are similar, there is a difference in terminology. A virtual memory block is called a page, and a virtual memory miss is called a page fault. With virtual memory, the processor produces a virtual address, which is translated by a combination of hardware and software to a physical address, which in turn can be used to access main memory.

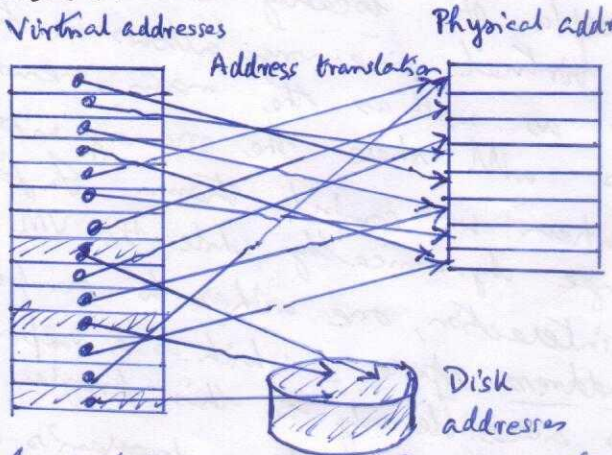

Fig: In virtual memory, blocks of memory (called pages) are mapped from one set of addresses (called virtual addresses) to another set (called physical addresses)

- Figure (on the left) shows the virtually addressed memory with pages mapped to main memory. This process is called address mapping or address translation.
- At present, the memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices (PMDs) and DRAMs and magnetic disks in servers.

- Virtual memory also simplifies loading the program for execution by providing relocation. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory. Moreover, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thus eliminating the need to find a contiguous block of memory.

Virtual Memory (cont.)

to allocate to a program; instead, the operating system need only find an adequate number of free pages in main memory.

<Referring to the figure on the flip side of page 2/19, the processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. {Note that it is possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk}. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code>.

- In virtual memory, the address is broken into a virtual page number and a page offset. Figure below shows translation of the virtual page number to a physical page number. The physical page number constitutes the upper portion of the physical address, while the page offset (which is not changed) makes up the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

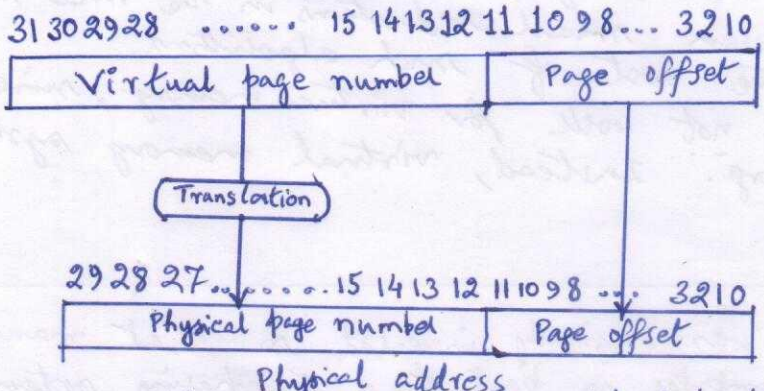

Figure: Mapping from a virtual to a physical address.

- The page size is 2^{12} = 4 Kilobytes. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. Thus, the main memory can have at most 1 Gigabytes (2^{30}), while the virtual address space is 4 Gigabytes (2^{32}).

P.T.O.

• Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. As is evident from the Table below, the latency of main memory (DRAM) is about 100,000 times quicker than disk. This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

Memory technology	Typical Access Time
SRAM semiconductor memory	0.5 – 2.5 ns
DRAM Semiconductor memory	50 – 70 ns
Flash Semiconductor memory	5,000 – 50,000 ns
Magnetic disk	$5 \times 10^6 - 20 \times 10^6$ ns

1) Pages should be large enough to try to amortize the high access time. Sizes from 4 Kilobytes to 16 Kilobytes are typical today. New desktop and server systems are being developed to support 32 Kilobytes and 64 Kilobytes pages, but new embedded systems are going in the other direction (to 1 Kilobyte pages)

2) Organizations that reduce the page fault rate are attractive. The primary technique used here is to allow fully associative placement of pages in memory.

3) Page faults can be handled in software because the overhead will be small compared to disk acces time. Also, the software can use clever algorithms for choosing how to place pages because even small reductions in the miss rate will compensate for the cost of such algorithms

4) Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

Note: 1) Virtual memory was originally invented so that many programs could share a computer as part of a timesharing system, although virtual machines (sharing the same memory) has been presented above as motivation for virtual memory

2) For servers and PCs, 32-bit address processors are problematic. Although one normally thinks of virtual addresses as much larger than physical addresses, the opposite can occur when processor address size is small relative to the state of the memory technology. No single ~~state~~

Why 32-bit address processes are problematic for servers PCs (cont.)

programs or virtual machine can benefit, but a collection of programs or VMs running at the same time can benefit from not having to be swapped to memory or by running on parallel processors.

3) Note that paging uses fixed-size blocks. There also exists a variable-size block scheme called segmentation. In segmentation, an address has two parts: a segment number and a segment offset. The segment number is mapped to a physical address, and the offset is added to find the actual physical address. As the segment can vary in size, a bounds (limit) check is also needed to ensure that the offset is within the segment.

The major use of segmentation is to support more powerful methods of protection and sharing in an address space. Most operating system textbooks extensively discuss segmentation (as compared to paging) and the use of segmentation to logically share the address space. Major disadvantage of segmentation: it splits the address space into logically separate pieces that must be manipulated as a two-part address: the segment number and the offset. In contrast, paging makes the boundary between page number and page offset invisible to programmers and compilers.

• Many architectures divide the address space into large fixed-size blocks that simplify protection between the operating system and user programs, and increase the efficiency of implementing paging. Although these divisions are often called "segments", this mechanism is much simpler than variable-block size segmentation, and is not visible to user programs.

Placing a Page and Finding it Again

- Due to the extremely high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If one allows a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs. For example, the OS can use a sophisticated algorithm and complex data structures that track page usage to try to choose a page that will not be needed for a long time. Ability to use a clever and flexible replacement scheme reduces the page fault rate and simplifies the use of fully associative placement of pages.

- As mentioned earlier, the difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical. In virtual memory systems, one locates pages by using a table that indexes the memory; this structure is called a page table, and it resides in memory. A page table is indexed with the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory. As may be seen later, the page table may contain entries for pages not present in memory. To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; one calls this the page table register. Assume for now that the page table is in a fixed and contiguous area of memory

Hardware / Software Interface

- The page table, together with the program counter and the registers, specifies the state of a virtual machine. If one allows another virtual machine to use the processor, one must save this state. Later, after restoring this state, the VM can continue execution. One often refers to this state as a process. The process is considered active when it is in possession of the processor; else, it is considered inactive. The operating system can make a process active by loading the state of the process, including the program counter, which will initiate execution at the value of the saved program counter. The address space of the process, and hence all the data that it can access in memory, is defined by its page table, which resides in memory. Rather than saving the entire page table, the operating system simply loads the page table register, to point to the page table of the process which it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The operating system is responsible for allocating the physical memory and updating the page tables, so that the virtual address spaces of different processes do not collide. As may be seen shortly afterward, the use of separate page tables also provides protection of one process from another.
- Figure (on the flip side) uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry. (As was done in a cache). If the bit is off, the page is not present in main memory, and a page fault occurs. If the bit is on, the page is in memory, and the entry contains the physical page number.
- As the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index, that is used to access the page table, consists of the full block address, which is the virtual page number.

P.T.O.

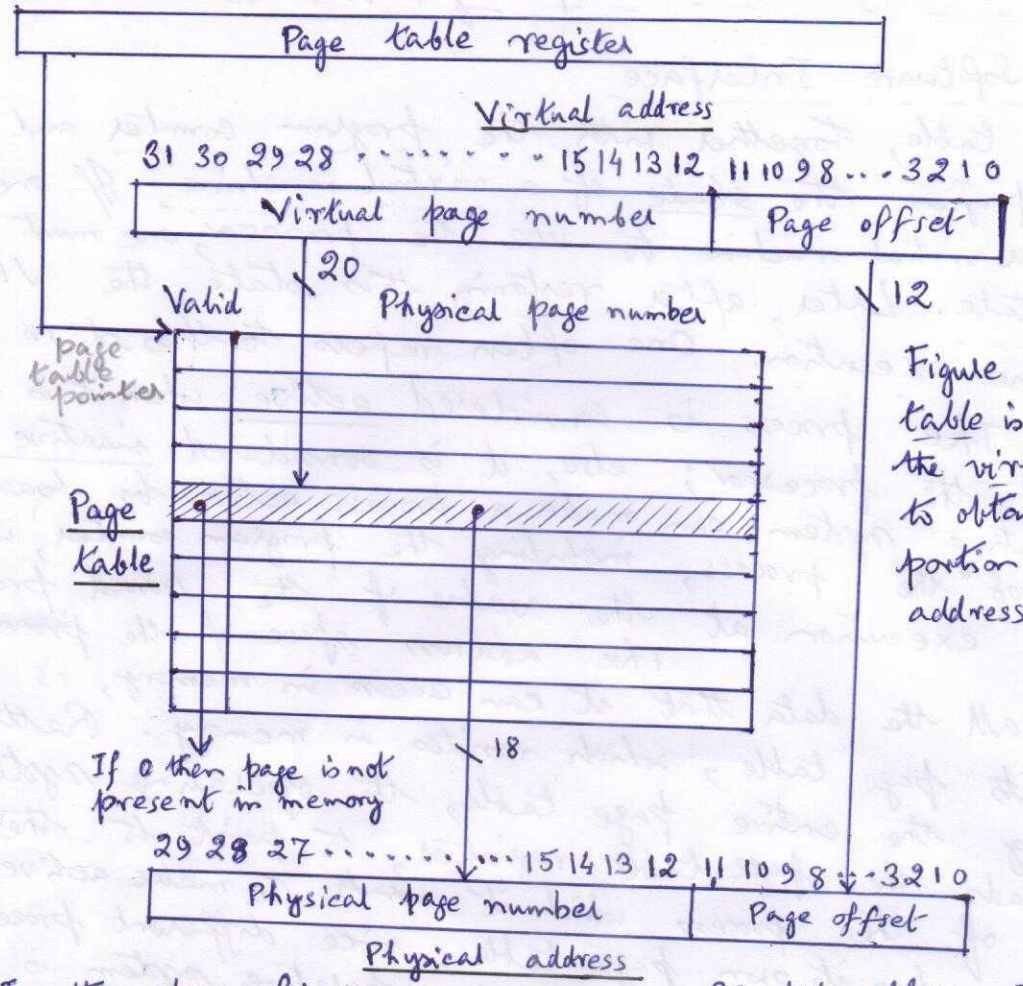

Figure : the page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.

- In the above figure, one assumes a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 Kilobytes (4096 bytes). The virtual address space is 2^{32} bytes, or 4 Gigabytes, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 Gigabytes. The number of entries in the page table is 2^{20}, or 1 million entries. The valid bit for each entry indicates whether the mapping is legitimate. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

Page Faults:

• If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. Once the O.S. gets control (this transfer of control is done with the exception mechanism), it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in main memory.

• The virtual address alone does not immediately tell us where the page is on disk. In a virtual memory system, one must keep track of the location on disk of each page in virtual address space.

• As one does not know in advance when a page in memory will be replaced, the OS usually creates the space on flash memory or disk for all the pages of a process, when it creates the process. This space is called the swap space. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. Figure (on the flip side) shows the organization when a single table holds either the physical page number or the disk address.

• The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the OS must choose a page to replace. As one wants to minimize the number of page faults, most operating systems try to choose a page that they hypothesize will not be needed in the near future. Using the past to predict the future, an OS normally follows the least recently used (LRU) replacement scheme. < It is assumed that a page that has not been used for a long time is less likely to be needed than a more recently accessed page >. The replaced pages are written to swap space on the disk. [Swap space : is the space on the disk reserved for the full virtual memory space of a process.]

P.T.O.

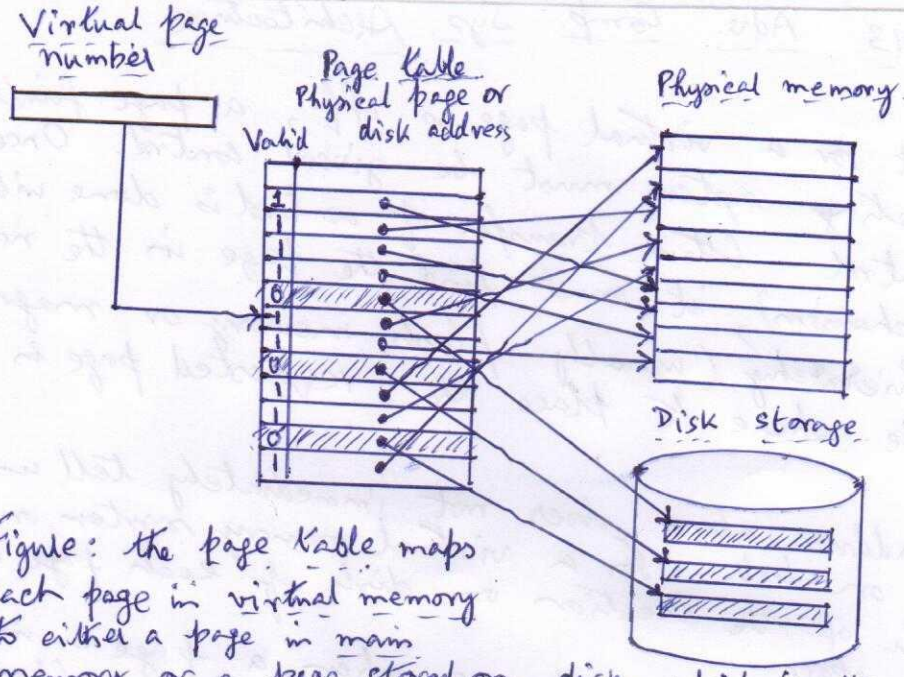

Figure: the page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e. the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part, as one must keep the disk addresses of all the pages, even if they are currently in main memory. Recall that the pages in main memory and the pages on disk are the same size

- Implementing an absolutely accurate LRV (least recently used) scheme is too costly, as it requires updating a data structure on every memory reference. Instead, most operating systems approximate LRV by keeping track of which pages have and which pages have not been recently used. To help the OS estimate the LRV pages, some computers provide a reference bit or use bit, which is set (to '1'), whenever a page is accessed. The OS periodically clears the reference bits and later records them so it can determine which pages were touched during a particular time period. Based on this usage information, the OS can select a page that is among the least recently referenced (detected by having its reference bit off). If this bit is not provided by the hardware, the operating system must find another way to estimate which pages have been accessed.

How to Manage "Writes" in Virtual Memory Systems

- Difference between the access time to the cache and main memory is tens to hundreds of cycles, and write-through schemes can be used, although one needs a write buffer to hide the latency of the write from the processor. In a virtual memory system, writes to the next level of the hierarchy (disk) can take millions of processor clock cycles (so, building a write buffer to allow the system to "write through" to disk would be highly impractical). Instead, virtual memory systems must use write-back, performing the individual writes into the page in memory, and copying the page back to disk when it is replaced in the memory.

- Major advantage of a write-back scheme in a virtual memory system: as the disk transfer time is small as compared with its access time, copying back an entire page is much more efficient than writing individual words back to the disk. A write-back operation, though more efficient than transferring individual words, is still costly. Thus one would like to know if a page needs to be copied back when one chooses to replace it. To track whether a page has been written since it was read into the memory, a dirty bit is added to the page table. The dirty bit is set when any word in a page is written. If the operating system chooses to replace the page, the dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page. Hence, a modified page is often referred to as a dirty page.

P.T.O.

Making Address Translation Fast - the TLB

- Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address, and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality.
- Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a translation-lookaside buffer (TLB), although it would be more accurate to call it a translation cache. The TLB (rather than the page table) will prove to be useful on every reference.

Figure (on page 2/25) indicates that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. As TLB is accessed on every reference, it needs to include other status bits, such as dirty and reference bits.

On every reference, one looks up the virtual page number in the TLB. If one gets a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor performs a write, a dirty bit is also turned on. If a miss in TLB occurs, one must find out if it is a page fault, or only a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor handles TLB miss by loading the translation from the page table into the TLB, and then trying the reference again. If the page is not present in memory, then TLB miss indicates a true page fault. In this case, the processor invokes the OS using an exception. As the TLB has many fewer entries than the number of pages in main memory, the TLB misses will be much more frequent than true page faults.

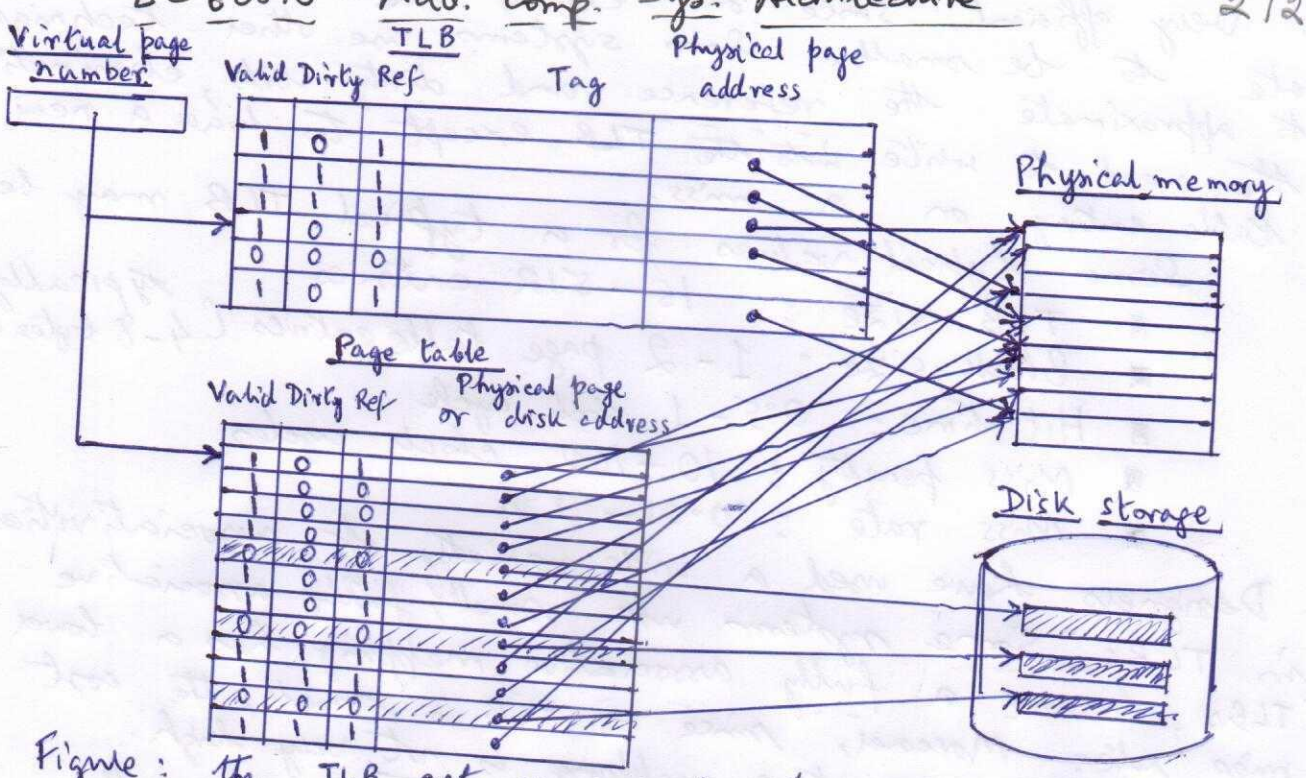

Figure: the TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. As the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed. In other words, unlike a TLB, a page table is not a cache.

~~than true page faults.~~ • TLB misses can be handled either in hardware or in software. In practice, there can be little performance difference between the two approaches, as the basic operations are the same in both cases.

After a TLB miss occurs, and the missing translation has been retrieved from the page table, one needs to select a TLB entry to replace. As the reference and dirty bits are contained in the TLB entry, one needs to copy these bits back to the page table entry when one replaces an entry. These bits are the only portion of the TLB entry that can be changed. Using write-back, that is, copying these entries back at miss time rather than when they are written-

is very efficient, since one expects the TLB ~~miss~~ miss rate to be small. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

- Some typical values for a typical TLB may be
  - TLB size : 16 – 512 entries
  - Block size : 1 – 2 page table entries (typically 4 – 8 bytes each)
  - Hit time : 0·5 – 1 clock cycle
  - Miss penalty : 10 – 100 clock cycles
  - Miss rate : 0·01 – 1%

Designers have used a wide variety of associativities in TLBs. Some systems use small fully associative TLBs, because a fully associative mapping has a lower miss rate. Moreover, since the TLB is small, the cost of a fully associative mapping is not very high.

Other systems use large TLBs, often with small associativity. For a fully associative mapping, choosing the entry to replace becomes tricky as implementing a hardware LRU scheme is very expensive. Also, since TLB misses are much more frequent than page faults and this must be handled more cheaply, one cannot afford an expensive software algorithm, as one can for page faults. As a result, many systems provide some support for randomly choosing an entry to replace.