

Chapter 6 Objectives

- Master the concepts of **hierarchical** memory organization.
- Understand how each level of memory contributes to system performance, and how the performance is measured.
- Master the concepts behind **cache** memory, **virtual** memory, memory **segmentation**, **paging** and address **translation**.

2

6.1 Introduction

- Memory lies at the heart of the stored-program computer. Its access speed affects the overall system performance.
- In previous chapters, we studied the components (D flip-flops) from which memory is built and the ways in which memory is accessed by various ISAs (direct, indirect, ...).
- In this chapter, we focus on memory organization. A clear understanding of these ideas is essential for the analysis of system performance.

3

6.2 Types of Memory

- There are two kinds of memory: *RAM and ROM*. *RAM* (SRAM (**static RAM**) and DRAM (**dynamic RAM**)) are read-write memory. ROM is read-only memory (**some can be written to**).
- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory, but it is very expensive. It is used to build cache memory, which we will discuss in detail later.
- DRAM consists of capacitors that slowly leak their charge over time. Thus they must be refreshed every few milliseconds to prevent data loss.

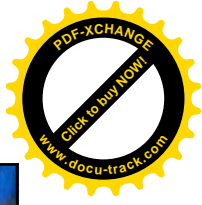
4



6.2 Types of Memory

- DRAM is slower than SRAM, because it is “cheap” owing to its simple design. It is much denser, uses less power, generates less heat than SRAM.
- The main memory is built of DRAM (e.g. EDO DRAM, SDRAM, DDR SDRAM, DR DRAM, ...).
- ROM is not volatile and always retains its data.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off (e.g. system settings, boot program, ...). The control store (**the miro-program for micro-instructions**) is also built of ROM.
- Types of ROM include ROM, PROM (**programmable**), EPROM (**erasable programmable**), EEPROM (**electronically-erasable programmable**), and flash memory.

5



6.3 The Memory Hierarchy

- One problem with the computer system is that the memory is much slower than CPU, causing CPU to wait for the memory.
- Though SRAM is faster, it is much more expensive than DRAM (main memory).
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.

6

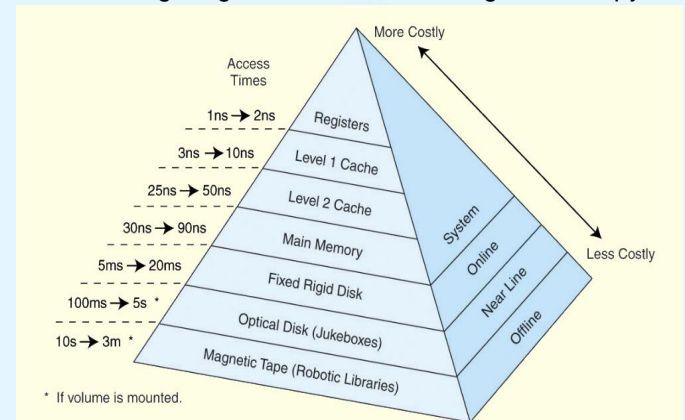
6.3 The Memory Hierarchy

- Small, fast storage elements (cache) are kept in the CPU; larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.
- This memory hierarchy provides a speed as fast as the small cache memory and a space as big as the large hard disk at a reasonable price.

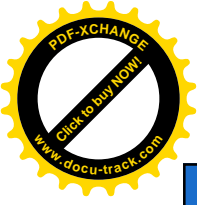
7

6.3 The Memory Hierarchy

- This storage organization can be thought of as a pyramid:



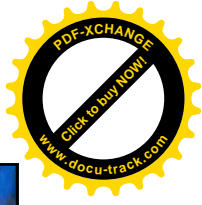
8



6.3 The Memory Hierarchy

- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
- Once the data is located, then the data, and a number of its nearby data elements are fetched into cache memory.

9



6.3 The Memory Hierarchy

- This leads us to some definitions.
 - A *hit* is when data is found at a given memory level.
 - A *miss* is when it is not found.
 - The *hit rate* is the percentage of time data is found at a given memory level.
 - The *miss rate* is the percentage of time it is not.
 - Miss rate = 1 – hit rate.
 - The *hit time* is the time required to access data at a given memory level.
 - The *miss penalty* is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor.

10

6.3 The Memory Hierarchy

- An entire block of data is copied to cache after a miss because the *principle of locality* tells us that once a byte is accessed, it is likely that a nearby data element will be needed soon.
- There are three forms of locality:
 - *Temporal locality*- Recently-accessed data elements tend to be accessed again.
 - *Spatial locality* - Accesses tend to cluster – **fields that are close together tend to be accessed sooner.**
 - *Sequential locality* - Instructions tend to be accessed sequentially.
- The locality principle allows a small but very fast memory to hold most of the instructions and data that CPU will need.

11

6.4 Cache Memory

- The purpose of cache memory is to speed up accesses by storing recently used data closer to the CPU, instead of storing it in main memory.
- Although cache is much smaller than main memory, its access time is a fraction of that of main memory.
- Unlike main memory, which is accessed by address, **cache is accessed using different techniques, depending upon which type of cache you are using – this will be discussed.**
- **All of the notes here assume that we are dealing with word-addressable data.**

12

6.4 Cache Memory

- Memory (RAM) address (more bits) does not match cache address (less bits).
- The “content” that is addressed by the content addressable cache memory is a subset of the bits of a main memory address called a *tag*, and is saved.
- The main memory address is divided into fields which provide a many-to-one mapping between larger main memory and the smaller cache memory.
- Main memory and cache are divided into many blocks. A *tag* field in the cache block distinguishes one cached memory block from another.

13

6.4 Cache Memory

- The simplest cache mapping scheme is **direct mapped** cache (see Figure 06.02 on next page).
- A direct mapped cache consists of N total blocks of cache, block X of main memory maps to cache block $Y = X \text{ mod } N$, *i.e., X/N , & the remainder is the value we use – e.g., assuming we have a total of 10 blocks of cache memory, $Y = \text{RAM block } 17 / \text{mod } 10 = 7$, so it writes to block 7 in cache.*
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.
- Once a block of memory is copied into its slot in cache, a *valid* bit is set for the cache block to let the system know that the block contains valid data.

14

6.4 Cache Memory

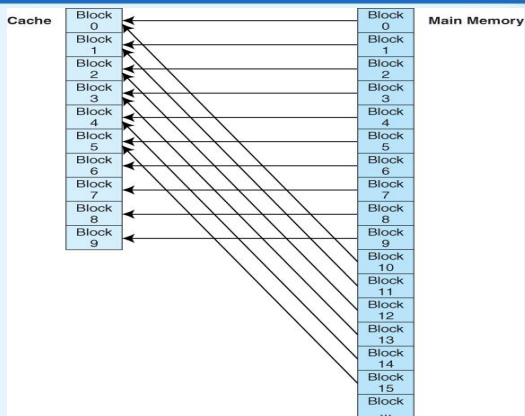


Figure 06.02
Direct Mapping of Main Memory Blocks to Cache Blocks

15

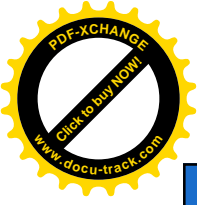
6.4 Cache Memory

- The diagram below is a schematic of what cache looks like.

Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	-----		0
3	-----		0

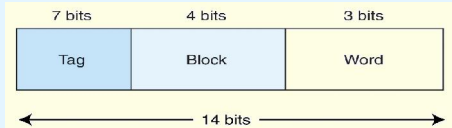
- Block 0 contains multiple words from main memory, identified with the tag 00000000. Block 1 contains words identified with the tag 11110101. **At this point the tag is meaningless to us.**
- The other two blocks are not valid, *i.e., has no RAM currently cached.*

16



6.4 Cache Memory

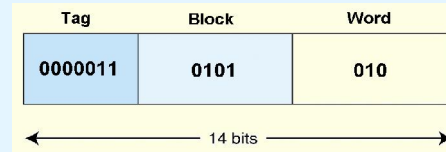
- The size of each field into which a RAM memory address is divided depends on the size of the cache.
- Suppose our RAM memory consists of $16K = 2^{14}$ words; cache has $16 = 2^4$ blocks, and each block holds $8 = 2^3$ words. **And suppose that the whole cache has $16 \times 8 = 128 = 2^7$ words – (16 blocks, 8 words per block).**
 - Thus RAM memory is divided into $2^{14} / 2^3 = 2^{11}$ (2K) blocks.
- For our field sizes, we know we need 4 bits for the block. **Since we have 16 cache blocks, 0-15**, 3 bits for the word **(since we have 8 words per block)**, and the tag is what's left over:



17

6.4 Cache Memory

- As an example, suppose a program generates the address **1AA**. In 14-bit binary, this number is: **00000110101010**.
- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.

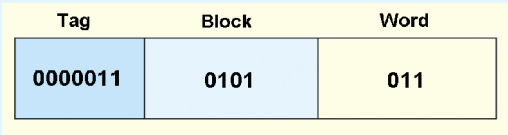


- So CPU will first check the data in cache memory location block 5 (0101_2) and word 2 (010_2) – **which is really the 3rd word, as we start from word 000.**

18

6.4 Cache Memory

- If subsequently the program generates the address **1AB**, it will find the data it is looking for in cache block **0101**, word **011**.



- However, if the program generates the address, **3AB** (tag=0000111), instead, the block loaded for address **1AB** would be evicted from the cache **(since the block # is the same)**, **copied back to RAM**, and replaced by the block associated with the **3AB** reference.

19

6.4 Cache Memory

- Suppose a program generates a series of memory references such as: **1AB, 3AB, 1AB, 3AB, ...**. The cache will continually evict and replace blocks **(since each of these addresses refer to the same block)**, though empty cache blocks are available.
- The theoretical advantage offered by the cache is lost in this extreme case.
- This is the main disadvantage of direct mapped cache.
- Other cache mapping schemes are designed to prevent this kind of thrashing.

20

6.4 Cache Memory

- Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- This is how **fully associative** cache works.
- A memory address is partitioned into only two fields: the tag and the word.

21

6.4 Cache Memory

- Suppose, as before, we have 14-bit memory addresses, **i.e., 16k words**, and a cache with 16 blocks, each block of size 8 words. The field format of a memory reference is:



- When the cache is searched, all tags are searched in parallel, **i.e., concurrently**, to retrieve the data (block) quickly.
- This requires special, costly hardware (associative₂₂ memory).

6.4 Cache Memory

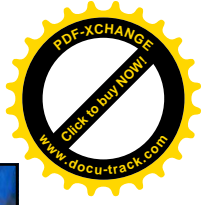
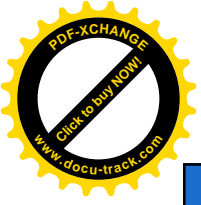
- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- Fully associative cache does not need to evict a block as long as there is an empty block available.
- When all blocks are used in a fully associative cache, we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.
- There are a number of ways to pick a victim; we will discuss them shortly.

23

6.4 Cache Memory

- **Set associative** cache combines the ideas of direct mapped cache (simple) and fully associative cache (no block conflict).
- An *N*-way (**2 or 4 or 8**) set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular block (set) in the cache.
- Unlike direct mapped cache, a memory reference is not mapped to one block, but a set of cache blocks, similar to the way in which fully associative cache works, **e.g., if 2-way, the RAM block has 2 cache blocks it can be mapped to.**
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

24



6.4 Cache Memory

- The number of cache blocks per set in set associative cache varies according to overall system design.
- For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
- Each set contains two different memory blocks.

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, ...	1	-----		0
1	11110101	Words L, M, N, ...	1	-----		0
2	-----		0	10111011	P, Q, R, ...	1
3	-----		0	11111100	T, U, V, ...	1

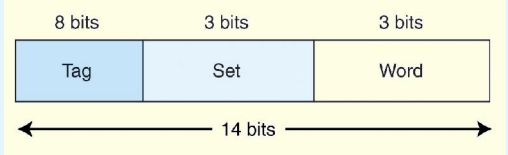
6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and word, as shown below.
- As with direct-mapped cache, the word field chooses the word within the cache block, and the tag field uniquely identifies the memory block.
- The set field determines the set to which the memory block maps.



6.4 Cache Memory

- Suppose we have a main memory of 2^{14} words.
- This memory is mapped to a 2-way set associative cache having 16 blocks where each block contains 8 words.
- Since this is a 2-way cache, each set consists of 2 blocks, and there are 8 sets.
- Thus, we need 3 bits for the set (since there are 8 sets), 3 bits for the word, giving 8 leftover bits for the tag:



6.4 Cache Memory

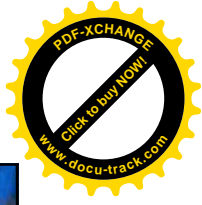
- The set field of a main memory address specifies a unique set in cache for the given main memory block. All blocks in that cache set are then searched for a matching tag. An associative search must be performed, but the search is restricted to the specified set instead of the entire cache. This significantly reduces the cost of the specialized hardware. For example, in a 2-way set associative cache, the hardware searches only 2 blocks in parallel.



6.4 Cache Memory

- With fully associative and set associative cache, a *replacement policy* is invoked when it becomes necessary to evict a block from cache.
- An *optimal* replacement policy would be able to look into the future to see which blocks won't be needed for the longest period of time.
- Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

29



6.4 Cache Memory

- The replacement policy that we choose depends upon the locality that we are trying to optimize – usually, we are interested in temporal locality – **recently accessed items tend to be accessed again in the near future.**
- A *least recently used* (LRU) algorithm keeps track of the last time that a block was assessed and evicts the block that has not been used for the longest period of time.
- The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.

30

6.4 Cache Memory

- *First-in, first-out* (FIFO) is a popular cache replacement policy.
- In FIFO, the block that has been in the cache the longest, regardless of when it was last used, will be evicted.
- A *random* replacement policy does what its name implies: It picks a block at random and replaces it with a new block.
- Random replacement can certainly evict a block that will be needed often or needed soon, but it never thrashes (move data in and out repeatedly) as LRU and FIFO do.

31

6.4 Cache Memory

- The performance of hierarchical memory is measured by its *effective access time* (EAT).
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.
- The EAT for a two-level (**cache & main memory**) memory is given by:

$$EAT = H \times Access_C + (1-H) \times Access_{MM}$$

where H is the cache hit rate, (so 1-H = miss rate), and Access_C and Access_{MM} are the access times for cache and main memory, respectively. Access_{MM} includes both the time to read a block of data from main memory to the cache and the time to read the required data from cache to CPU.

32

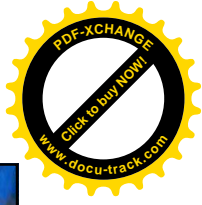


6.4 Cache Memory

- For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.
- The EAT is:
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$

Miss rate of 1%
↓
- The EAT is mostly dependent on the hit rate.
- In this case, the memory system is almost as fast as the cache and is as big as the main memory.
- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

33



6.4 Cache Memory

- Cache replacement policies must also take into account *dirty blocks*, those blocks that have been updated while they were in the cache.
- Dirty blocks must be written back to memory. A *write policy* (**controlled by hardware**) determines how this will be done.
- There are two types of write policies, *write through* and *write back*.
- Write through updates cache and main memory simultaneously on every write.

34

6.4 Cache Memory

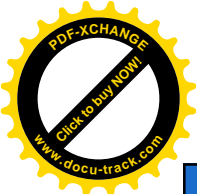
- Write back (also called *copyback*) updates memory only when the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because the majority of accesses tend to be reads, not writes.
- The advantage of write back is that memory traffic is minimized, but its disadvantage is that memory does not always agree with the value in cache, causing problems in systems with many concurrent users.

35

6.4 Cache Memory

- The cache we have been discussing is called a *unified* or *integrated* cache where both instructions and data are cached.
- Many modern systems employ separate caches for data and instructions.
 - This is called a *Harvard* cache.
- The separation of data from instructions provides better locality, performance at the cost of greater complexity.
 - Simply making the cache larger (more expensive) provides about the same performance improvement (but less efficiency) without the complexity.

36



6.4 Cache Memory

- Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.
 - This is called a *victim cache*.
- A trace cache is a variant of an instruction cache **that holds instructions which have already been decoded, so if the instructions are needed again, no decoding is required.**

37



6.4 Cache Memory

- Most of today's small systems employ multilevel cache hierarchies.
- The levels of cache form their own small memory hierarchy.
- Level1 cache (8KB to 64KB) is situated on the processor itself.
 - Access time is typically about 4ns.
- Level 2 cache (64KB to 2MB) may be on the motherboard, or on an expansion card.
 - Access time is usually around 15 - 20ns.

38

6.4 Cache Memory

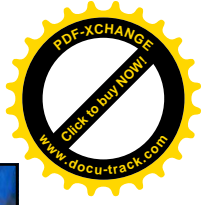
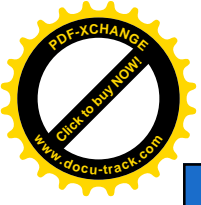
- In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns)
- Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.
- Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.

39

6.4 Cache Memory

- If the cache system used an *inclusive* cache, the same data may be present at multiple levels of cache **concurrently.**
- *Strictly inclusive* caches guarantee that all data in a smaller cache also exists at the next higher level.
- *Exclusive* caches permit only one copy of the data.
- The tradeoffs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity.

40



6.5 Virtual Memory

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- Most common virtual memory implementation uses **paging**, where main memory/programs are divided into fixed-size blocks called *frames/pages*.
- If the size of program is larger than RAM, some program pages will reside in the hard disk.
- Virtual memory functions similarly to cache memory.

6.5 Virtual Memory

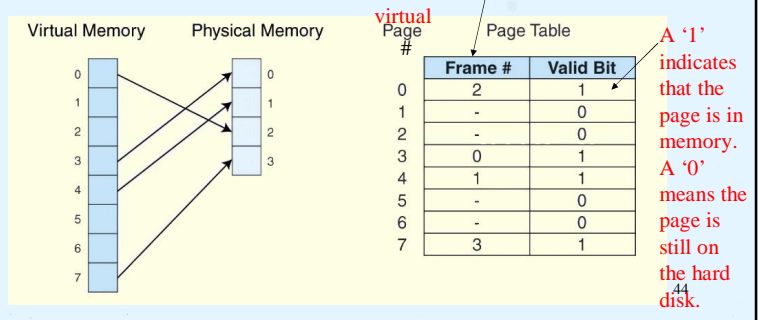
- A *physical address* is the actual memory address of physical memory.
- Programs use *virtual (logical) addresses* that are *mapped* to physical addresses by the memory manager. The virtual address space is larger than the physical address space.
- *Page faults* occur when a logical address requires that a page be brought into the RAM from disk.
- *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses (e.g., the last page of the program, when in RAM, will likely not fully use the full page).

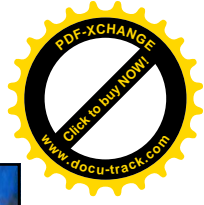
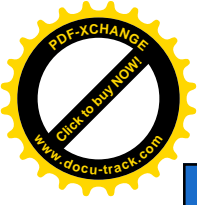
6.5 Virtual Memory

- Main memory and virtual memory are divided into equal sized pages.
- The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- Further, the pages allocated to a process do not need to be stored contiguously-- either on disk or in memory.
- In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

6.5 Virtual Memory

- Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a *page table* (shown below).
- There is one page table for each active process, e.g., a program.





6.5 Virtual Memory

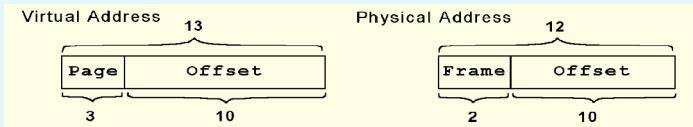
- When a process generates a virtual address, the operating system translates it into a physical memory address.
- To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field.
- The page field determines the page location of the address, and the offset indicates the location of the address within the page.
- The logical page number is translated into a physical page frame number through a lookup in the page table.

6.5 Virtual Memory

- If the valid bit is zero in the page table entry for the logical address, this means that the page is not in memory and must be fetched from disk.
 - This is a page fault.
 - If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1.
- If the valid bit is 1, the virtual page number is replaced by the physical frame number.
- The data is then accessed by adding the offset to the physical frame number.

6.5 Virtual Memory

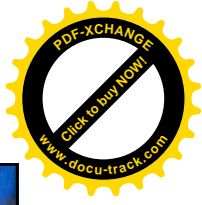
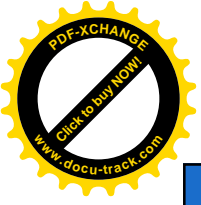
- As an example, suppose a system has a virtual address space of 8K (2^{13}), a physical address space of 4K (2^{12}) and 1K (2^{10}) page size, and the system uses byte addressing. We have $2^{13}/2^{10} = 2^3 = 8$ virtual pages, and $2^{12}/2^{10} = 2^2 = 4$ physical frames.
- A virtual address has 13 bits with 3 bits for the page field (since there are only 8 virtual pages), and 10 for the offset within the page.
- A physical memory address has 12 bits, the first two bits for the page frame (4 available) and the trailing 10 bits for the offset.



6.5 Virtual Memory

- Suppose we have the page table shown below.
- What happens when CPU generates address $5459_{10} = 10101010011_2$? (the first 3 bits represent frame #5)

Virtual Page #	Physical Frame	Valid Bit	Addresses
0	-	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	-	0	3 : 3072 - 4095
4	-	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	-	0	7 : 7168 - 8191



6.5 Virtual Memory

- The address 1010101010011_2 is converted to physical address 010101010011 because the page field

Virtual Page #	Physical Frame	Valid Bit	Addresses
Page 0	-	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	-	0	3 : 3072 - 4095
4	-	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	-	0	7 : 7168 - 8191

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2 ?

Frame # 4

Page Table	Frame	Valid Bit	Addresses
Page 0	-	0	Page 0 : 0 - 1023
1	3	1	1 : 1024 - 2047
2	0	1	2 : 2048 - 3071
3	-	0	3 : 3072 - 4095
4	-	0	4 : 4096 - 5119
5	1	1	5 : 5120 - 6143
6	2	1	6 : 6144 - 7167
7	-	0	7 : 7168 - 8191

Generates a page fault, because the valid bit = 0. So a page has to be fetched from disk. But here, in our example, we only have 4 pages of real memory (all currently used), so one of the pages have to be paged out to disk, & this one will replace it.

6.5 Virtual Memory

- We said earlier that effective access time (EAT) takes all levels of memory into consideration.
- Thus, virtual memory is also a factor in the calculation, and we also have to consider page table access time.
- Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. We have:
- $EAT = 0.99(200ns + 200ns) + 0.01(10ms) = 100,396ns$
 - To access the data in ram
 - To access the page table in RAM
- Virtual memory causes a lot of delay in access time:
 - $100,396 / 200 = 501.98$ or perhaps
 - $100,396 / (200 + 200) = 250.99$
- How can we speed up the effective access time ?

6.5 Virtual Memory

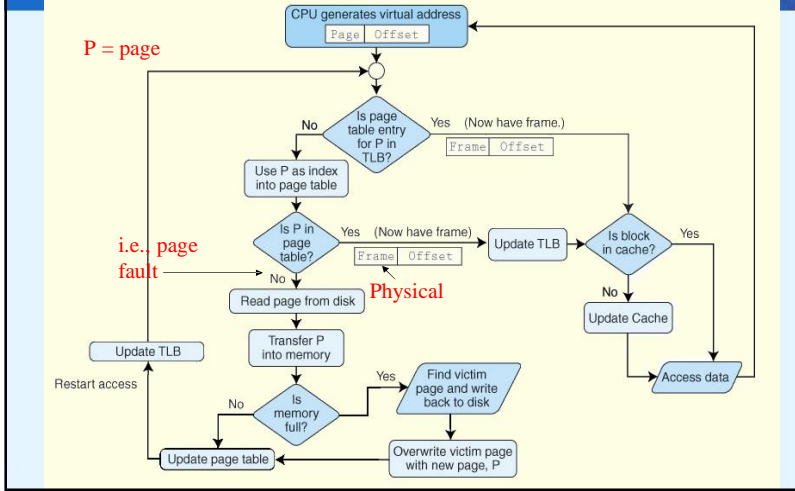
- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because page tables are read constantly, it makes sense to keep them in a special cache called a *translation look-aside buffer (TLB)*.
- TLB is a special associative cache that stores the mapping of virtual pages to physical pages. Usually only part of the mapping information in the page table are stored in TLB.

The next slide shows how all the pieces fit together.

6.5 Virtual Memory

- What are the advantages of using physical memory and paging? Programs are no longer restricted by the amount of physical memory that is available. Virtual memory permits us to run individual programs whose virtual address space is larger than physical memory. In effect, this allows one program to share memory with itself. This makes it much easier to write programs, because the programmer no longer has to worry about the physical address space limitations.

6.5 Virtual Memory

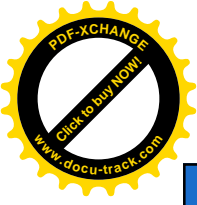


6.5 Virtual Memory

- In addition to paging, another approach to virtual memory is the use of *segmentation*.
- Instead of dividing memory into equal-sized pages, virtual address space is divided into variable-length segments, often under the control of the programmer.
- Segments may be stored in contiguous memory space. A segment is located through its entry in a segment table, which contains the segment's memory location and a bounds limit that indicates its size.
- After a segment fault, the operating system searches for a location in memory (RAM) large enough to hold the segment that is retrieved from disk.

6.5 Virtual Memory

- Both paging and segmentation have advantages and disadvantages.
- Paging is subject to *internal* fragmentation because a process may not need the entire range of addresses contained within the last page **i.e., the last page of the program.**
- Segmentation is subject to *external* fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over the time.
- Paging, with fixed-size pages, is easier to manage for the system.



6.5 Virtual Memory

- Paging uses large page tables, which are cumbersome and slow. But with its uniform memory mapping, page operations are fast. Segmentation allows easier access to the small segment table, but segment loading is labor-intensive.
- Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

57

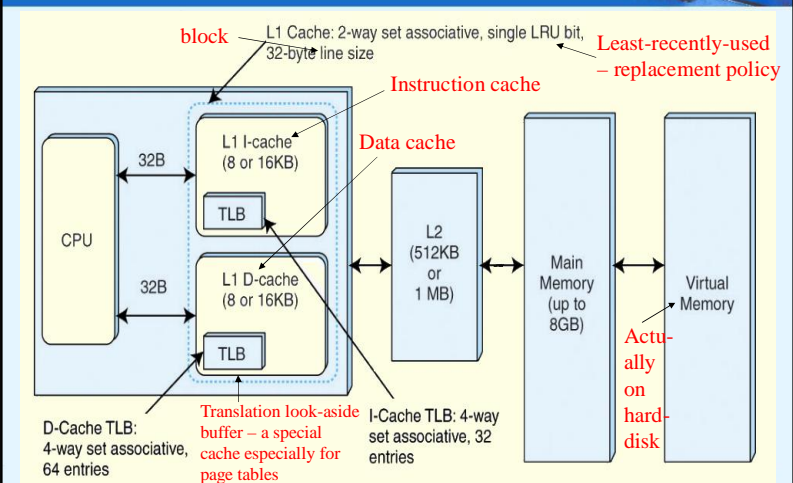
6.6 A Real-World Example

- The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpagged unsegmented, segmented unpagged, and unsegmented pagged.
- The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).

The next slide shows this organization schematically.

58

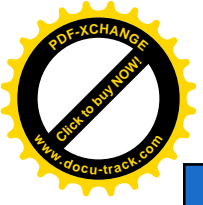
6.6 A Real-World Example



60

Chapter 6 Conclusion

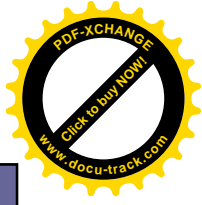
- Computer memory is organized in a hierarchy, with the smallest, fastest memory at the top and the largest, slowest memory at the bottom.
- Cache memory gives faster access to main memory, while virtual memory uses disk storage to give the illusion of having a large main memory.
- Cache maps blocks of main memory to blocks of cache memory. Virtual memory maps page frames to virtual pages.
- There are three general types of cache: Direct mapped, fully associative and set associative.



Chapter 6 Conclusion

- With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- Replacement policies include LRU, FIFO, or LFU. These policies must also take into account what to do with dirty blocks.
- All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.

61



62