

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.
- In previous chapters, we studied the components from which memory is built and the ways in which memory is accessed by various ISAs.
- 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 main memory: *random access memory, RAM, and read-only-memory, ROM.*
- There are two types of RAM, dynamic RAM (DRAM) and static RAM (SRAM).
- Dynamic RAM consists of capacitors that slowly leak their charge over time. Thus they must be refreshed every few milliseconds to prevent data loss.
- DRAM is “cheap” memory owing to its simple design.

4

6.2 Types of Memory

- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory, which we will discuss in detail later.
- ROM also does not need to be refreshed, either. In fact, it needs very little charge to retain its memory.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off.

5

6.3 The Memory Hierarchy

- Generally speaking, faster memory is more expensive than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- Small, fast storage elements 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.

6

6.3 The Memory Hierarchy

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



7

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.

8

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.

9

6.3 The Memory Hierarchy

- An entire blocks of data is copied after a hit 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.
 - Sequential locality* - Instructions tend to be accessed sequentially.

10

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 typically accessed by content; hence, it is often called *content addressable memory*.
- Because of this, a single large cache memory isn't always desirable-- it takes longer to search.

11

6.4 Cache Memory

- The "content" that is addressed in content addressable cache memory is a subset of the bits of a main memory address called a *field*.
- The fields into which a memory address is divided provide a many-to-one mapping between larger main memory and the smaller cache memory.
- Many blocks of main memory map to a single block of cache. A *tag* field in the cache block distinguishes one cached memory block from another.

12

6.4 Cache Memory

- The simplest cache mapping scheme is *direct mapped cache*.
- In a direct mapped cache consisting of N blocks of cache, block X of main memory maps to cache block $Y = X \bmod N$.
- 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.

What could happen without having a valid bit?

13

6.4 Cache Memory

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

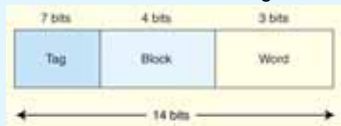
Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	0
3	0

- Block 0 contains multiple words from main memory, identified with the tag 00000000. Block 1 contains words identified with the tag 11110101.
- The other two blocks are not valid.

14

6.4 Cache Memory

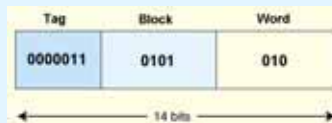
- The size of each field into which a memory address is divided depends on the size of the cache.
- Suppose our memory consists of 2^{14} words, cache has $16 = 2^4$ blocks, and each block holds 8 words.
 - Thus memory is divided into $2^{14} / 2^8 = 2^{11}$ blocks.
- For our field sizes, we know we need 4 bits for the block, 3 bits for the word, and the tag is what's left over:



15

6.4 Cache Memory

- As an example, suppose a program generates the address **1AA**. In 14-bit binary, this number is: 0000110101010.
- 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.



16

6.4 Cache Memory

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

Tag	Block	Word
0000011	0101	010

- However, if the program generates the address, **3AB**, instead, the block loaded for address **1AA** would be evicted from the cache, and replaced by the blocks associated with the **3AB** reference.

17

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.
- 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.

18

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.

19

6.4 Cache Memory

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

The diagram shows a horizontal rectangle representing a 14-bit memory reference. It is divided into two sections: a larger left section labeled 'Tag' with '11 bits' written above it, and a smaller right section labeled 'Word' with '3 bits' written above it.

- When the cache is searched, all tags are searched in parallel to retrieve the data quickly.
- This requires special, costly hardware.

20

6.4 Cache Memory

- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus 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.

21

6.4 Cache Memory

- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An *N*-way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of several cache blocks, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

22

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	Words P, Q, R, ...	1
3	-----	-----	0	11111100	Words T, U, V, ...	1

23

6.4 Cache Memory

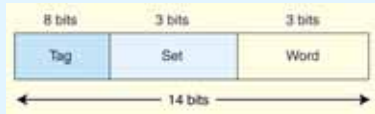
- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and word, as shown below.
- As with direct-mapped cache, the word field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set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

The diagram shows a horizontal rectangle representing a memory reference divided into three equal-width sections labeled 'Tag', 'Set', and 'Word' from left to right.

24

6.4 Cache Memory

- Suppose we have a main memory of 2^{14} bytes.
- 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, 3 bits for the word, giving 8 leftover bits for the tag:



25

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.

26

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.
- A *least recently used* (LRU) algorithm keeps track of the last time that a block was assessed and evicts the block that has been unused 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.

27

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.
- 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.

28

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 memory is given by:
$$\text{EAT} = H \times \text{Access}_C + (1-H) \times \text{Access}_{MM}$$
where H is the cache hit rate and Access_C and Access_{MM} are the access times for cache and main memory, respectively.

29

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:
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$
- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

30

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* 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.

31

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.

32

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.
- If a system uses paging, virtual memory partitions main memory into individually managed *page frames*, that are written (*or paged*) to disk when they are not immediately needed.

33

6.5 Virtual Memory

- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

34

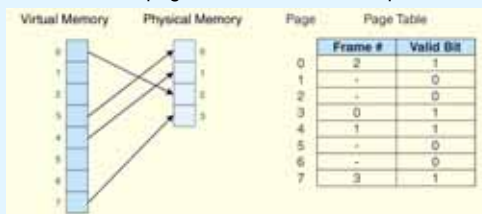
6.5 Virtual Memory

- 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.

35

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.



36

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 through a lookup in the page table.

37

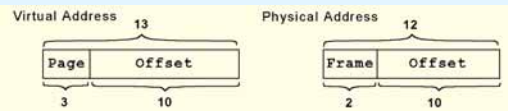
6.5 Virtual Memory

- 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.

38

6.5 Virtual Memory

- As an example, suppose a system has a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing.
 - We have $2^{13}/2^{10} = 2^3$ virtual pages.
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.



39

6.5 Virtual Memory

- Suppose we have the page table shown below.
- What happens when CPU generates address $5459_{10} = 1010101010011_2$?

	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

40

6.5 Virtual Memory

- The address 1010101010011_2 is converted to physical address 010101010011 because the page field 101 is replaced by frame number 01 through a lookup in the 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

41

6.5 Virtual Memory

- What happens when the CPU generates address 100000000100_2 ?

	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

42

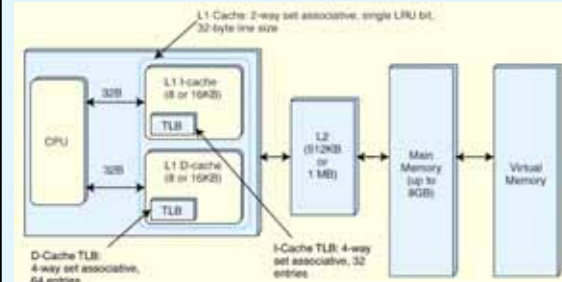
6.6 A Real-World Example

- 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.

49

6.6 A Real-World Example



50

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.

51

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.

52

53