CHAPTER 6

Memory

6.1 Memory 281

- In this chapter we examine the various types of memory and how each is part of memory hierarchy system
- We then look at **cache memory** (a special high-speed memory) and a method that utilizes memory to its fullest by means of **virtual memory** implemented via **paging**.

6.2 Types of Memory 281

- 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**).
- DRAM
- 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.
    - It is used to build **main memory**.
- SRAM
    - SRAM consists of circuits similar to the **D** flip-flop.
    - SRAM is very fast memory and it doesn't need to be refreshed like DRAM does.
    - It is used to build **cache memory**.
- ROM
    - 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.
- Types of DRAM (Basic operations of all DRAM memories are the same)
    - MDRAM (Multibank DRAM)
    - FPM RAM (Fast-Page Mode DRAM)
    - SDRAM (Synchronous DRAM)
    - DDR SDRAM (Double Data Rate Synchronous DRAM)
- Types of ROMs
    - ROM (Read-Only Memory)
    - PROM (Programmable Read-Only Memory)
    - EPROM (Erasable PROM)
    - EEPROM (Electrically Erasable PROM)

6.3 The Memory Hierarchy 283

- 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.

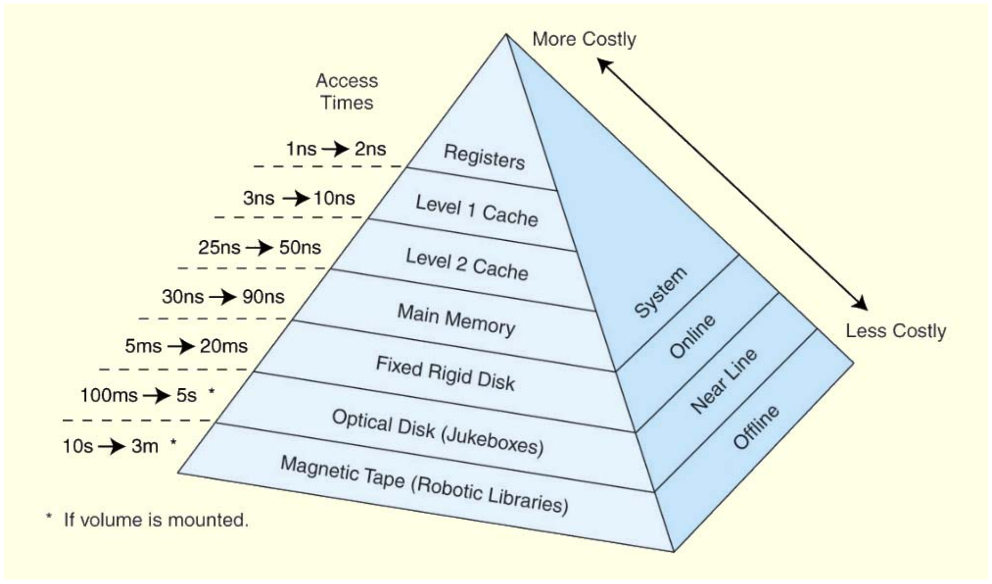


FIGURE 6.1 The Memory Hierarchy

6.3.1 Locality of Reference 285

- 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** in the **near future**.
  - *Spatial locality* - Accesses tend to be **clustered** in the address space (for example, as in **array** or **loops**).
  - *Sequential locality* - **Instructions** tend to be accessed **sequentially**.

6.4 Cache Memory 285

- 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.
- The computer uses the locality principle and transfers **an entire block** from main memory into cache whenever it has to make a main memory access.
- Unlike main memory, which is accessed by address, cache is typically accessed by **content**; hence, it is often called *content addressable memory or CAM*.
- A single large cache memory **isn't** always desirable-- it takes longer to search.

6.4.1 Cache Mapping Schemes 287

- The CPU uses a specific mapping scheme that "converts" the main memory address into a cache location.
- Many blocks of main memory map to a single block of cache.
- 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.

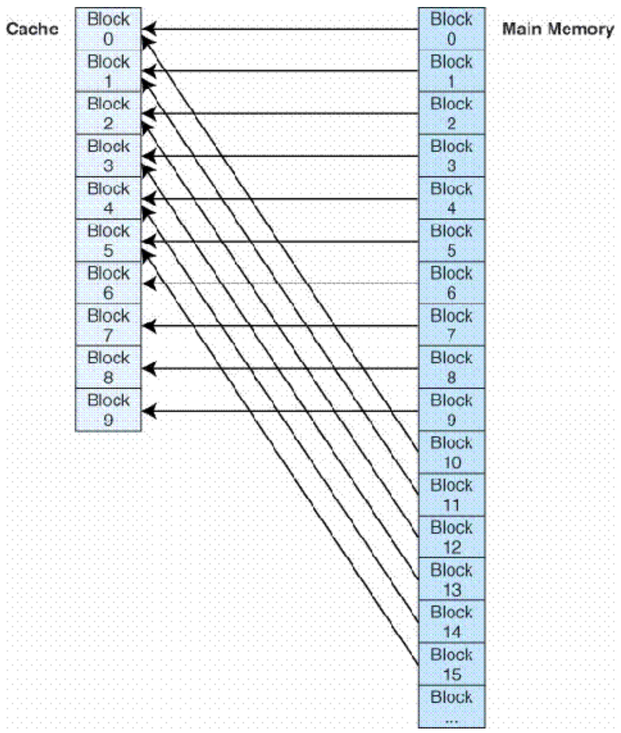
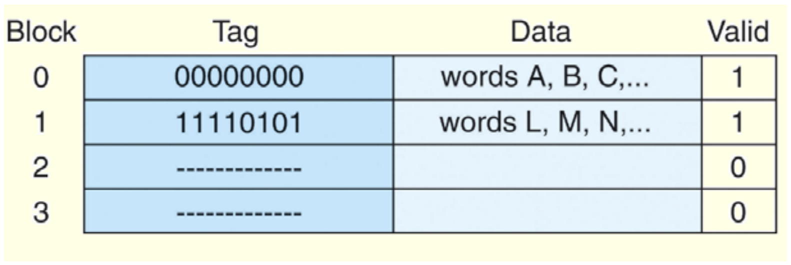


FIGURE 6.2 Direct Mapping of Main Memory Blocks to Cache Blocks

- A *tag* field in the cache block distinguishes one cached memory block from another.

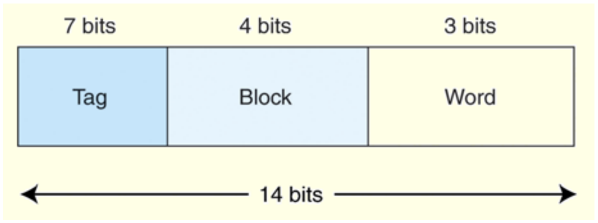


Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	-------------		0
3	-------------		0

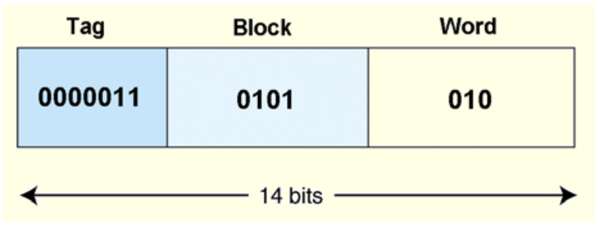
FIGURE 6.3 A Closer Look at Cache

- 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.

- The size of each field into which a memory address is divided depends on the size of the cache.
- Suppose our memory consists of $\mathbf{2^{14}}$ words, cache has **16** = 2^4 blocks, and each block holds **8** words.
- Thus memory is divided into $2^{14} / 2^3 = 2^{11}$ 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:



- 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



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

    - However, if the program generates the address, **3AB**, instead, the block loaded for address **1AA** would be evicted from the cache, and replaced.
- Fully Associative Cache
    - 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.
    - A memory address is partitioned into only two fields: the tag and the word.
    - 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:

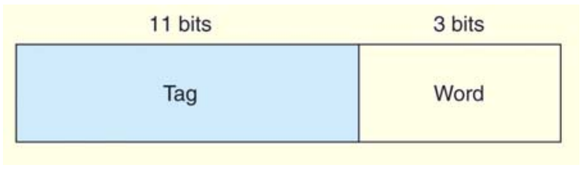
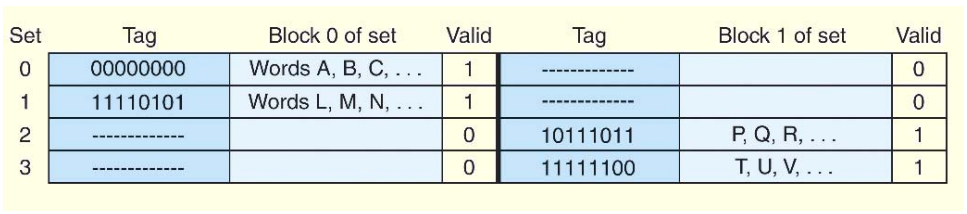


FIGURE 6.8 The Main Memory Address Format for Associative Mapping

    - The tag must be stored with **each block** in cache.
    - When the cache is searched, all tags are searched in parallel to retrieve the data quickly. This requires special, costly **hardware**.
    - 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***.
- Set Associative Cache
    - 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.
    - 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

FIGURE 6.9 A Two-Way Set Associative Cache

- 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:

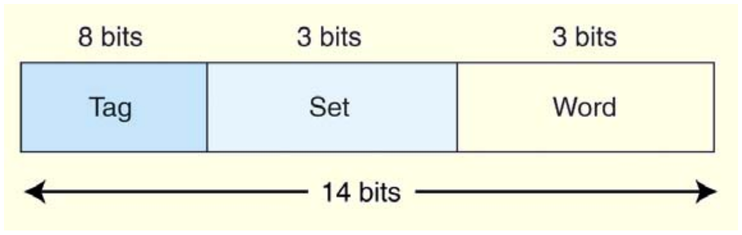


FIGURE 6.10 Format for Set Associative Mapping

6.4.2 Replacement Policies 295

- The existing block is kicked out of cache to make roon for the new block. This process is called replacement.
- With direct mapping, there is **no** need for a replacement policy.
- LRU
    - 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.
- FIFO
    - ***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.
- Random
    - 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** (constantly throw out a block, then bring it back, then throw out it out, then bring it back, repeatedly).

6.4.3 Effective Access Time and Hit Ratio 296

- 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:

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

where H is the cache hit rate and $Access_C$ and $Access_{MM}$ are the access times for cache and main memory, respectively.

- 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(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11ns.

6.4.4 When Does Caching Break Down? 297

- In particular, **object-oriented programming** can cause programs to exhibit less than optimal locality.
- Another example of bad locality can be seen in two dimensional array access. Arrays are typically **stored in row-major order**.
  - 5 X 4 array. If a program accesses the array in **row-major order**. So 5 X 4 array would produce 5 misses and 15 hits over 20 accesses.
  - If a program accesses the array in **column-major order**. So 5 X 4 array would produce 20 misses on 20 accesses.
- A third example would be a program that **loops** through a linear array that does **not fit in cache**. There would be a significant reduction in the locality when memory is used in this fashion.

6.4.5 Cache Write Policies 297

- 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.
  - 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**.

6.4.6 Instruction and Data Caches 300

- The cache we have been discussing is called a **unified** or **integrated cache** where both instructions **and** data are cached.
- Harvard cache: Many modern systems employ **separate** caches for data and instructions.
- The separation of data from instructions provides better **locality**, at the cost of greater complexity.
- 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 decoded instructions for program branches, giving the illusion that noncontiguous instructions are really contiguous.

6.4.7 Levels of Cache 301

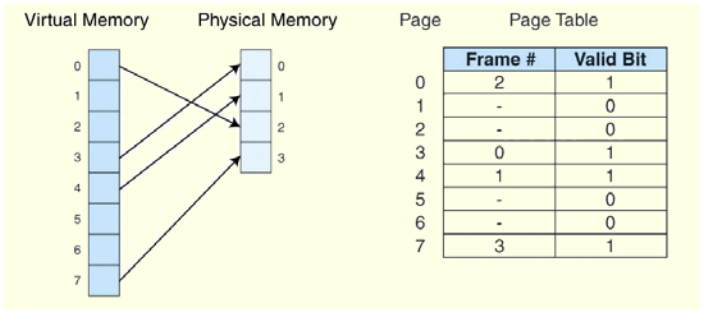
- Most of today's small systems employ **multilevel** cache hierarchies.
- The levels of cache form their own small memory hierarchy.
- Level1 cache (8KB to 64KB) is situated on the **processor itself**.
  - Access time is typically about **4ns**.
- Level 2 cache (64KB to 2MB) is typically located **external** to the processor, may be on the motherboard, or on an expansion card.
  - Access time is usually around **15 - 20ns**.
- 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.
- If the cache system used an **inclusive cache**, the same data may be present at multiple levels of cache. For example, in the Intel Pentium family, data **found** in L1 may **also** exist in L2
- **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.
- Separate data and instruction caches: For example, the **Intel Celeron** uses two separate L1 caches, one for instructions and one for data.

6.5 Virtual Memory 302

- 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.
- Virtual address: The **logical or program address** that the process uses. Whenever the **CPU** generates an address, it is always in terms of virtual address space.
- Physical address: The **real address** in physical memory.
- Page frames: The **equal-size** chunks or blocks into which main memory (physical memory) is divided.
- Pages: The chunks or blocks into which **virtual memory** (the logical address) is divided, each **equal in size** to a page frame.
- Paging: The process of copying a virtual page from disk to a page frame in main memory.
- Fragmentation: Memory that becomes **unusable**.
- Page fault: An event that occurs when a requested page is **not** in main memory and must be copied int memory from disk.
- **We need not have all of the process in main memory at once.** 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.

6.5.1 Paging 303

- The basic idea behind paging is quite simple: Allocate physical memory to processes in fixed size chucks (page frames) and keep track of where various pages of the process reside by recording information in a **page table**.
- **Every process** has its own **page table** that typically resides in **main memory**.
- If the page is in main memory the valid bit is set to **1**.
- Process memory is divided into these fixed size pages, resulting in potential **internal fragmentation** when the last page is copied into memory.
- The **operating system** must dynamically translate this virtual address into the physical address in memory at which the data actually resides.
- The newly retrieved page can be placed in **any** of those free frames.
- For example,
    - Suppose we have a virtual address space of 2^8 words for a given process, and physical memory of 4 page frames.
    - Assume also that pages are 32 words in length.



Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

FIGURE 6.11 Current State Using Paging and the Associated Page

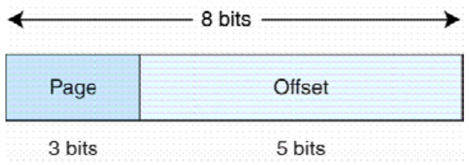


FIGURE 6.12 Format for an 8-Bit Virtual Address with $2^5 = 32$ Word Page Size.

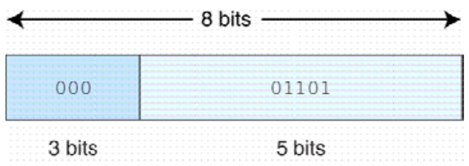


FIGURE 6.13 Format for Virtual Address $00001101_2 = 13_{10}$

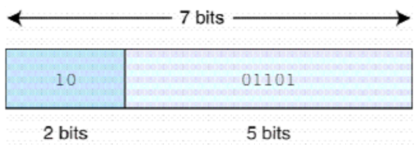


FIGURE 6.14 Format for Physical address $1001101_2 = 77_{10}$

6.5.2 Effective Access Time Using Paging 310

- 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

- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the **page table** (the page itself is stored in **main memory**), 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).
- Typically, the TLB is a special **associative cache** that stores the mapping of virtual pages to physical pages.

Virtual Page Number	Physical Page Number
-	-
5	1
2	0
-	-
-	-
1	3
6	2

TABLE 6.2 Current State of the TLB for Figure 6.16

6.5.3 Putting It All Together: Using Cache, TLBs, and Paging 311

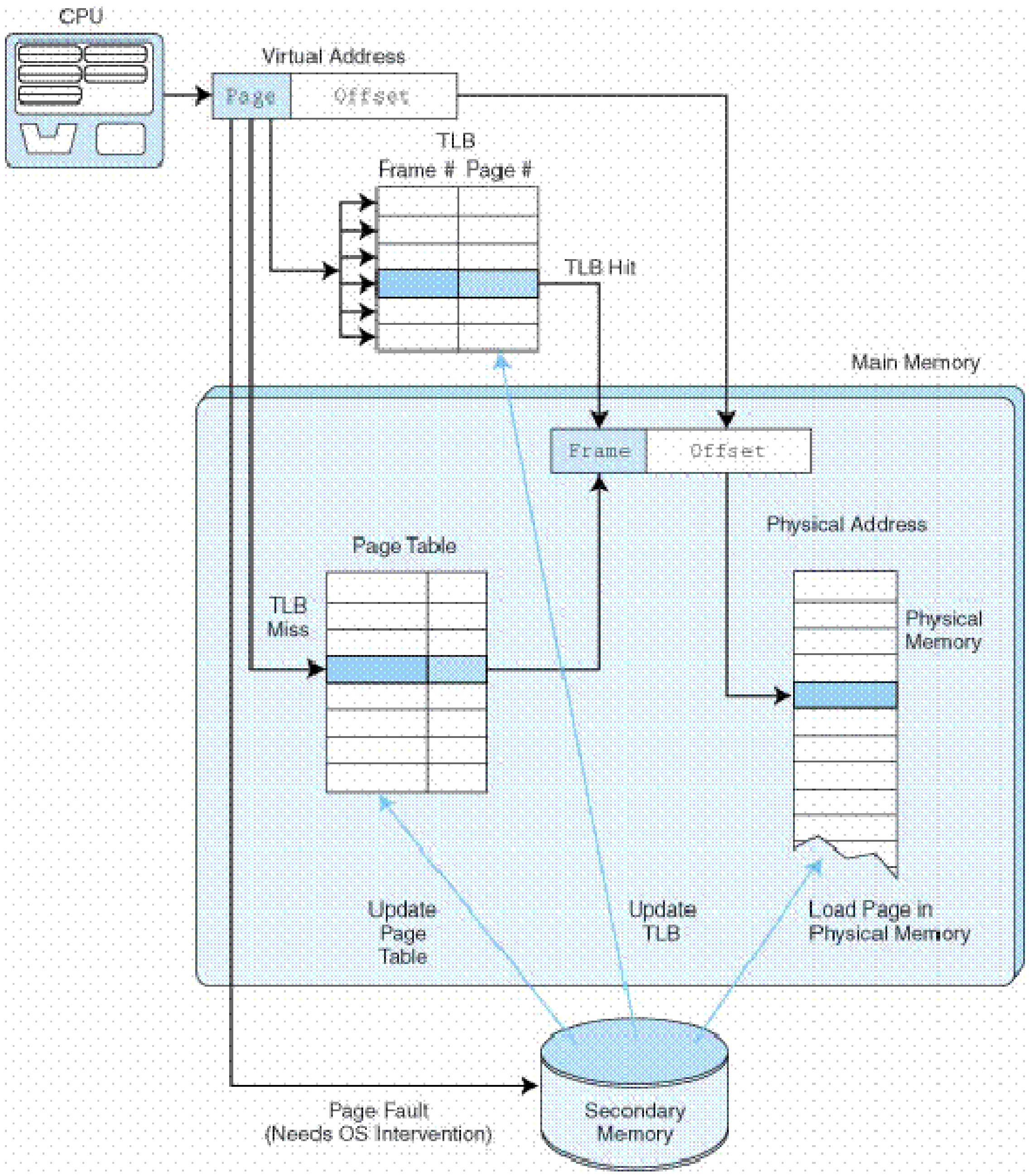


FIGURE 6.17 Using the TLB

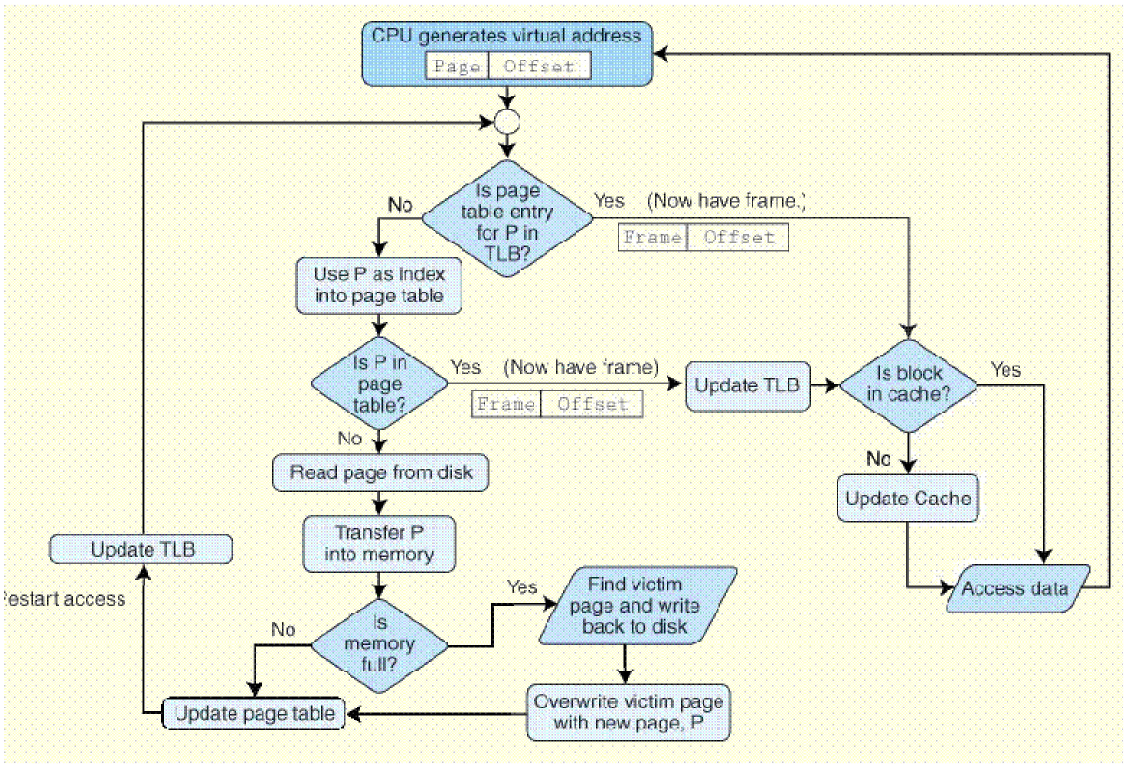


FIGURE 6.18 Putting it All Together: The TLB, Page Table, Cache and Main Memory

6.5.4 Advantages and Disadvantages of Paging and Virtual Memory 313

- Programs are **no longer** restricted by the amount of physical memory that is available. Virtual address space is **larger** than physical memory.
- Because each program requires less physical memory, virtual memory also permits us to **run more programs** at the same time.
- Increase in CPU utilization and system throughput.

6.5.5 Segmentation 314

- Instead of dividing memory into equal-sized pages, virtual address space is divided into **variable-length segments**, often under the control of the programmer.
- 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.
- This segment table is simply a collection of the **base/bounds pairs** for each segment.
- Both paging and segmentation can cause fragmentation.
  - Paging is subject to ***internal*** fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
  - Segmentation is subject to ***external*** fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

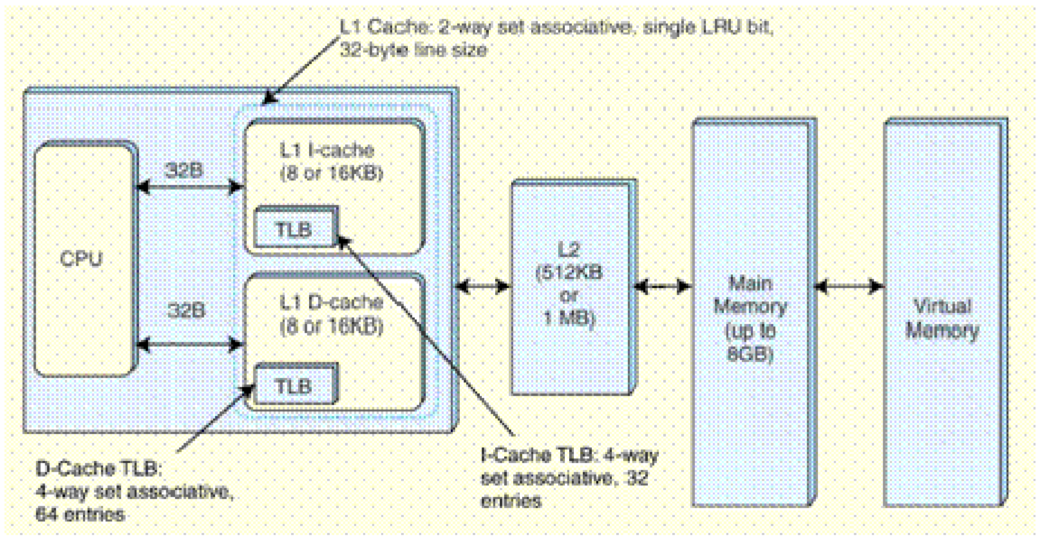
    - To combat external fragmentation, system use some sort of **garbage collection**.

6.5.6 Paging Combined with Segmentation 315

- Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- Pages are typically smaller than segments.
- **Each segment** has a page table, which means every program has multiple page tables. This means that a memory address will have three fields:
    - The first field is the **segment** field, which points the system to the appropriate page table.
    - The second field is the **page number**, which is used as an offset into this page table.
    - The third field is the **offset** within the page

6.6 A Real-World Example of Memory Management 316

- The Pentium architecture allows for **32-bit** virtual addresses and 32-bit physical addresses.
- It uses either **4KB or 4MB** page size, when using paging.
- The Pentium architecture supports **both** paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, and unsegmented paged.
- 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).


L1 Cache: 2-way set associative, single LRU bit, 32-byte line size
CPU
32B
32B
L1 I-cache (8 or 16KB)
TLB
L1 D-cache (8 or 16KB)
TLB
L2 (512KB or 1 MB)
Main Memory (up to 8GB)
Virtual Memory
D-Cache TLB: 4-way set associative, 64 entries
I-Cache TLB: 4-way set associative, 32 entries
FIGURE 6.19 Pentium Memory Hierarchy

Chapter Summary 317

- 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**. With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.
- Replacement policies include LRU, FIFO, or some other placement policy to determine the block to remove from cache to make room for a new block, if cache is full.
- Virtual memory allows us to run programs whose virtual address is **larger than** physical memory. It also allows **more processes** to run concurrently.
- TLB is a cache
- Cache improves the **effective access time** to main memory whereas paging **extends the size** of memory.