Chapter 6 – Memory

CS 271 Computer Architecture
Purdue University Fort Wayne

1

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

Outline

- Types of memory and the memory hierarchy
- Cache memory
- Virtual memory

4

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).
- DRAM 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.

5

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.

6

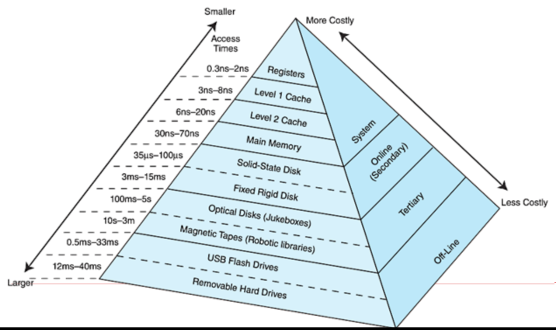
6.3 The Memory Hierarchy

- 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.

7

6.3 The Memory Hierarchy

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



8

6.3 The Memory Hierarchy

- We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.
- Registers are storage locations available on the processor itself.
- Virtual memory is typically implemented using a hard drive; it extends the address space from RAM to the hard drive.
- Virtual memory provides more space: Cache memory provides speed.

9

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.

10

Outline

- Types of memory and the memory hierarchy
- Cache memory
- Virtual memory

13

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.
- Three types of cache:
  - Direct mapped cache
  - Fully associative cache
  - Set associative cache

14

6.4 Cache Memory

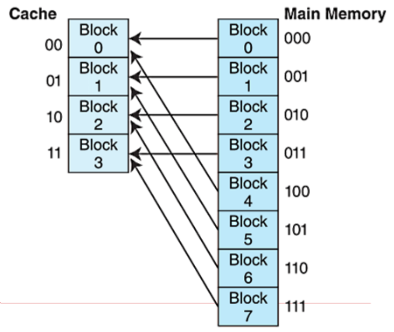
- The simplest cache mapping scheme is *direct mapped cache*.
- In a direct mapped cache consisting of *N* blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod *N*.
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.

The next slide illustrates this mapping.

15

6.4 Cache Memory

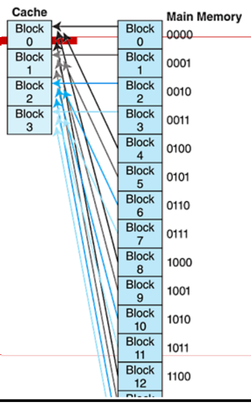
- With direct mapped cache consisting of *N* blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod *N*.



16

6.4 Cache Memory

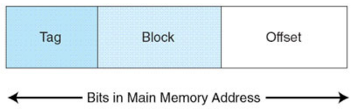
- A larger example.



17

6.4 Cache Memory

- To perform direct mapping, the binary main memory address is partitioned into the *fields* shown below.
  - The *offset* field uniquely identifies an address within a specific block.
  - The *block* field selects a unique block of cache.
  - The *tag* field is whatever is left over.



- The sizes of these fields are determined by characteristics of both memory and cache.

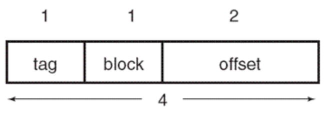
18

6.4 Cache Memory

- EXAMPLE 6.1 Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.
- This means Block 0 and 2 of main memory map to Block 0 of cache, and Blocks 1 and 3 of main memory map to Block 1 of cache.
- Using the tag, block, and offset fields, we can see how main memory maps to cache as follows.

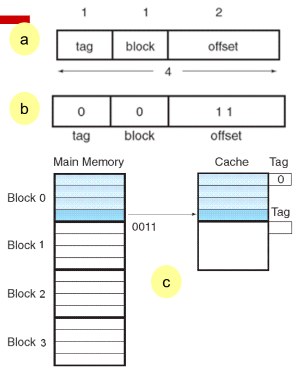
6.4 Cache Memory

- EXAMPLE 6.1 Cont'd Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.
  - First, we need to determine the address format for mapping. Each block is 4 bytes, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (as a main memory address has 4 bits because there are a total of 2^4=16 bytes).



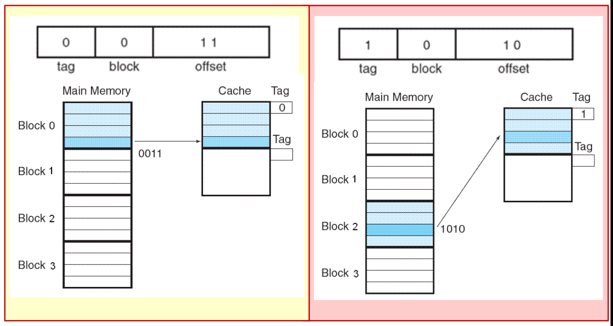
6.4 Cache Memory

- EXAMPLE 6.1 Cont'd
  - Suppose we need to access main memory address 3_{16} (0x0011 in binary). If we partition 0x0011 using the address format from Figure a, we get Figure b.
  - Thus, the main memory address 0x0011 maps to cache block 0.
  - Figure c shows this mapping, along with the tag that is also stored with the data.



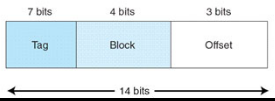
The next slide illustrates another mapping.

6.4 Cache Memory



6.4 Cache Memory

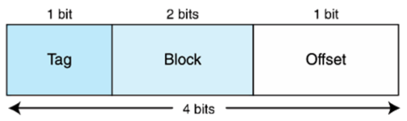
- EXAMPLE 6.2 Assume a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes.
  - The number of memory blocks are: $\frac{2^{14}}{2^{3}} = 2^{11}$
  - Each main memory address requires14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field
  - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
  - The remaining 7 bits make up the tag field.



23

6.4 Cache Memory

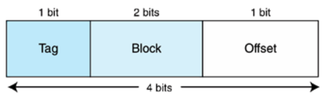
- EXAMPLE 6.3 Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks. We know:
  - A memory address has 4 bits.
  - The 4-bit memory address is divided into the fields below.



24

6.4 Cache Memory

- EXAMPLE 6.3 Cont'd The mapping for memory *references is shown below:*

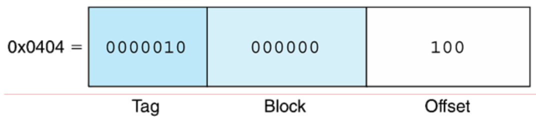


Main Memory	Maps To	Cache
(000) Block 0 (addresses 0x0, 0x1)	$\longrightarrow$	Block 0 (00)
(001) Block 1 (addresses 0x2, 0x3)	$\longrightarrow$	Block 1 (01)
(010) Block 2 (addresses 0x4, 0x5)	$\longrightarrow$	Block 2 (10)
(011) Block 3 (addresses 0x6, 0x7)	$\longrightarrow$	Block 3 (11)
(100) Block 4 (addresses 0x8, 0x9)	$\longrightarrow$	Block 0 (00)
(101) Block 5 (addresses 0xA, 0xB)	$\longrightarrow$	Block 1 (01)
(110) Block 6 (addresses 0xC, 0xD)	$\longrightarrow$	Block 2 (10)
(111) Block 7 (addresses 0xE, 0xF)	$\longrightarrow$	Block 3 (11)

25

6.4 Cache Memory

- EXAMPLE 6.4 Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. We have:
  - 3 bits for the offset
  - 6 bits for the block
  - 7 bits for the tag.
- A memory reference for 0x0404 maps as follows:



26

6.4 Cache Memory

- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:
- The number of bits in the main memory address (how many addresses exist in main memory)
- The number of blocks are in cache (which determines the size of the block field)
- How many addresses (either bytes or words) are in a block (which determines the size of the offset field)

27

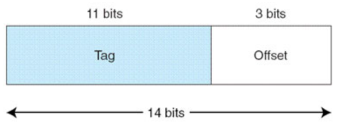
6.4 Cache Memory

- Suppose 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.

28

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:



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

29

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.

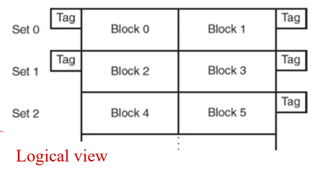
30

6.4 Cache Memory

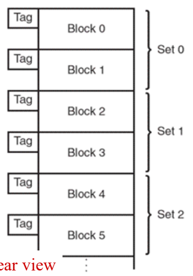
- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An *N*-way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of several 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.

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.



Logical view



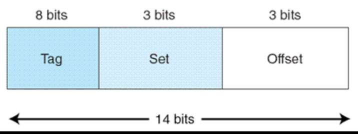
Linear view

6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and offset.
- As with direct-mapped cache, the offset 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.

6.4 Cache Memory

- EXAMPLE 6.5 Suppose we are using 2-way set associative mapping with a word-addressable main memory of 2^{14} words and a cache with 16 blocks, where each block contains 8 words.
  - Cache has a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache.
  - Thus, the set field is 3 bits, the offset field is 3 bits, and the tag field is 8 bits.



Outline

- Types of memory and the memory hierarchy
- Cache memory
- Virtual memory

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.

6.5 Virtual Memory

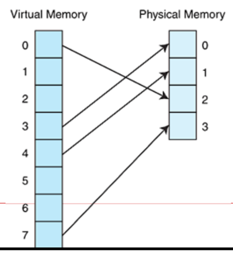
- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when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

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.



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

Page Table

59

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.

60

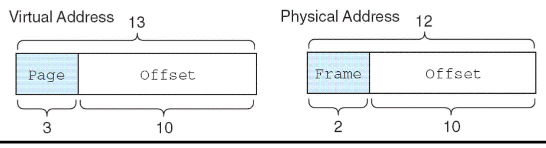
6.5 Virtual Memory

- 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.

61

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.



62

6.5 Virtual Memory

- Suppose we have the page table shown below.
- What happens when CPU generates address $5459_{10} = 1010101010011_2 = 0x1553$?

Page Table

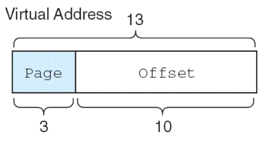
Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Addresses

Page	Base 10	Base 16
0 :	0 – 1023	0 – 3FF
1 :	1024 – 2047	400 – 7FF
2 :	2048 – 3071	800 – BFF
3 :	3072 – 4095	C00 – FFF
4 :	4096 – 5119	1000 – 13FF
5 :	5120 – 6143	1400 – 17FF
6 :	6144 – 7167	1800 – 1BFF
7 :	7168 – 8191	1C00 – 1FFF

6.5 Virtual Memory

- What happens when CPU generates address $5459_{10} = 1010101010011_2 = 0x1553$?



The high-order 3 bits of the virtual address, 101 (5_{10}), provide the page number in the page table.

64

6.5 Virtual Memory

- The address 1010101010011_2 is converted to physical address $010101010011_2 = 0x553$ because the page field 101 is replaced by frame number 01 through a lookup in the page table.

Page Table

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

Addresses

Page	Base 10	Base 16
0 :	0 – 1023	0 – 3FF
1 :	1024 – 2047	400 – 7FF
2 :	2048 – 3071	800 – BFF
3 :	3072 – 4095	C00 – FFF
4 :	4096 – 5119	1000 – 13FF
5 :	5120 – 6143	1400 – 17FF
6 :	6144 – 7167	1800 – 1BFF
7 :	7168 – 8191	1C00 – 1FFF

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2?

Page Table

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

Addresses

Page	Base 10	Base 16
0 :	0 – 1023	0 – 3FF
1 :	1024 – 2047	400 – 7FF
2 :	2048 – 3071	800 – BFF
3 :	3072 – 4095	C00 – FFF
4 :	4096 – 5119	1000 – 13FF
5 :	5120 – 6143	1400 – 17FF
6 :	6144 – 7167	1800 – 1BFF
7 :	7168 – 8191	1C00 – 1FFF

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) = 100.396us.

67

6.5 Virtual Memory

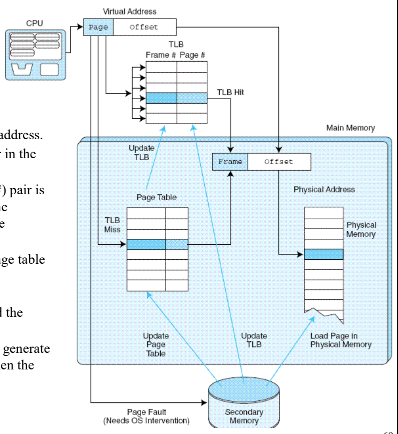
- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because page tables are read constantly, it makes sense to keep them in a special cache called a *translation look-aside buffer* (TLB).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

The next slide shows address lookup steps when a TLB is involved.

68

TLB lookup process

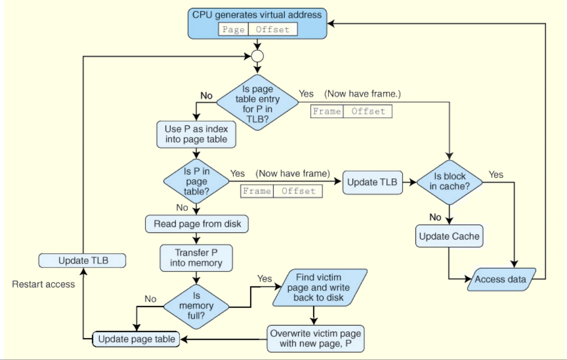
1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB.
4. If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number.
If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.
6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.



69

Putting it all together:
The TLB, Page Table,
and Main Memory

6.5 Virtual Memory



6.5 Virtual Memory

- 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.
- 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 page fault, the operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

71

6.5 Virtual Memory

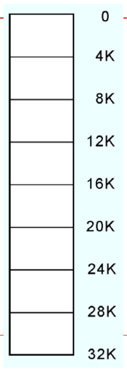
- Both paging and segmentation can cause fragmentation.
- Paging is subject to *internal* fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
- Segmentation is subject to *external* fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

The next slides illustrate internal and external fragmentation.

72

6.5 Virtual Memory

- Consider a small computer having 32K of memory.
- The 32K memory is divided into 8 page frames of 4K each.
- A schematic of this configuration is shown at the right.
- The numbers at the right are memory frame addresses.



73

6.5 Virtual Memory

- Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table.
- We observe that these processes require 31K of memory.

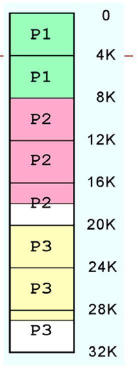
Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

74

6.5 Virtual Memory

- When the first three processes are loaded, memory looks like this:
- All of the frames are occupied by three of the processes.

P1	8K
P2	10K
P3	9K
P4	4K

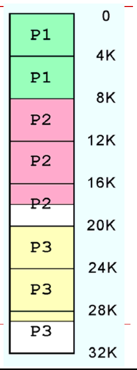


75

6.5 Virtual Memory

- Despite the fact that there are enough free bytes in memory to load the fourth process, P4 has to wait for one of the other three to terminate, because there are no unallocated frames.
- This is an example of *internal fragmentation*.

P1	8K
P2	10K
P3	9K
P4	4K



76

6.5 Virtual Memory

- Suppose that instead of frames, our 32K system uses segmentation.
- The memory segments of two processes is shown in the table at the right.
- The segments can be allocated anywhere in memory.

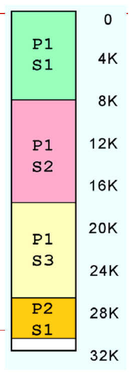
Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

77

6.5 Virtual Memory

- All of the segments of P1 and one of the segments of P2 are loaded as shown at the right.
- Segment S2 of process P2 requires 11K of memory, and there is only 1K free, so it waits.

P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

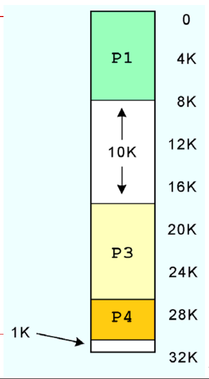


78

6.5 Virtual Memory

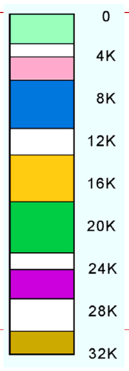
- Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded giving 11K of free memory.
- But Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.

P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



6.5 Virtual Memory

- Over time, the problem gets worse, resulting in small unusable blocks scattered throughout physical memory.
- This is an example of *external fragmentation*.
- Eventually, this memory is recovered through compaction, and the process starts over.

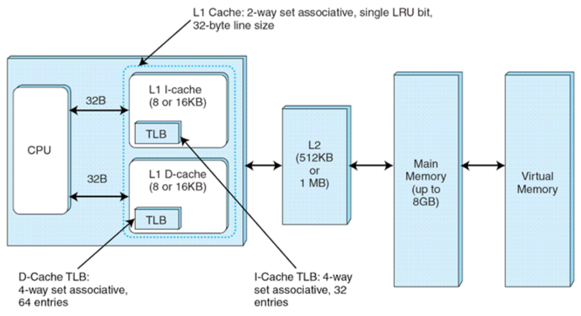


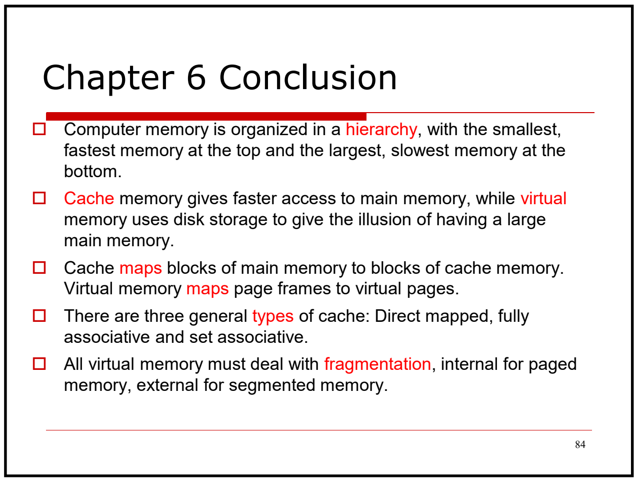
6.6 A Real-World Example

- 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).

The next slide shows this organization schematically.

6.6 A Real-World Example


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



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.
- All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.

84