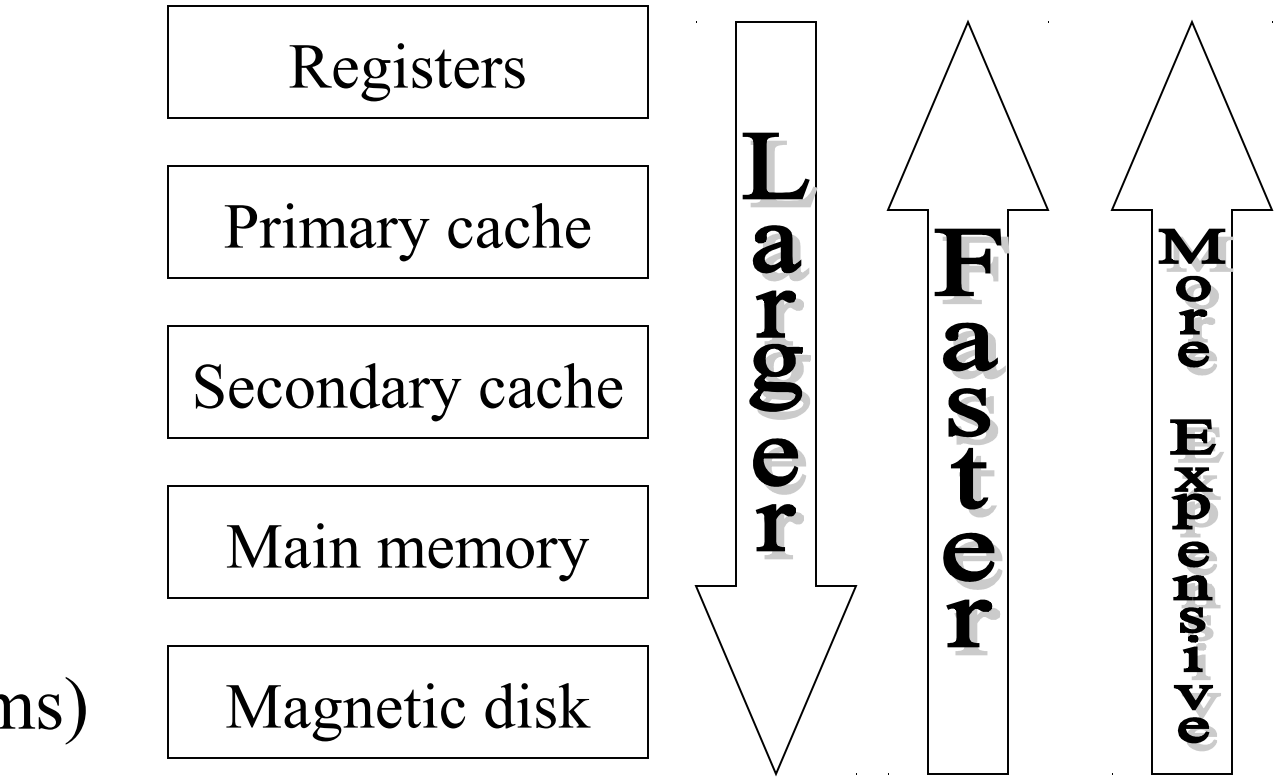
CACHE MEMORY

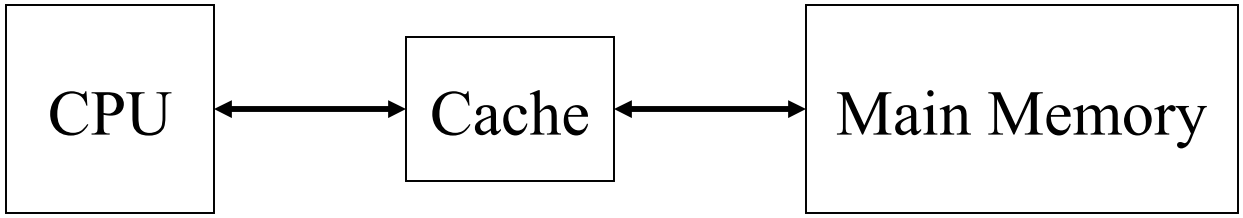
- SATHISH.V
- VELAMMAL INSTITUTE OF TECNOLOGY
- ECE- 2010-2014

The Memory Hierarchy

- ~2ns
- ~4-5ns
- ~30ns
- ~220ns+
- >1ms (~6ms)



Cache & Locality



- Cache sits between the CPU and main memory
  - Invisible to the CPU
- Only useful if recently used items are used again
- Fortunately, this happens a lot. We call this property *locality of reference.*

Locality of reference

- *Temporal locality*
  - Recently accessed data/instructions are likely to be accessed again.
    - Most program time is spent in loops
    - Arrays are often scanned multiple times
- *Spatial locality*
  - If I access memory address *n*, I am likely to then access another address close to *n* (usually *n*+1, *n*+2, or *n*+4)
    - Linear execution of code
    - Linear access of arrays

How a cache exploits locality

- Temporal – When an item is accessed from memory it is brought into the cache
  - If it is accessed again soon, it comes from the cache and not main memory
- Spatial – When we access a memory word, we also fetch the next few words of memory into the cache
  - The number of words fetched is the *cache line* size, or the *cache block size* for the machine

Cache write policies

- As long as we are only doing READ operations, the cache is an exact copy of a small part of the main memory
- When we write, should we write to cache or memory?
- Write through cache – write to both cache and main memory. Cache and memory are always consistent
- Write back cache – write only to cache and set a "dirty bit". When the block gets replaced from the cache, write it out to memory.

When might the write-back policy be dangerous?

Cache mapping

- Direct mapped – each memory block can occupy one and only one cache block
- Example:
  - Cache block size: 16 words
  - Memory = 64K (4K blocks)
  - Cache = 2K (128 blocks)

Direct Mapped Cache

- Memory block *n* occupies cache block (*n* mod 128)
- Consider address $2EF4

  `001011101111 0100`

  block: $2EF = 751 word: 4
- Cache:

  `00101 1101111 0100`

  tag: 5 block: 111 word: 4

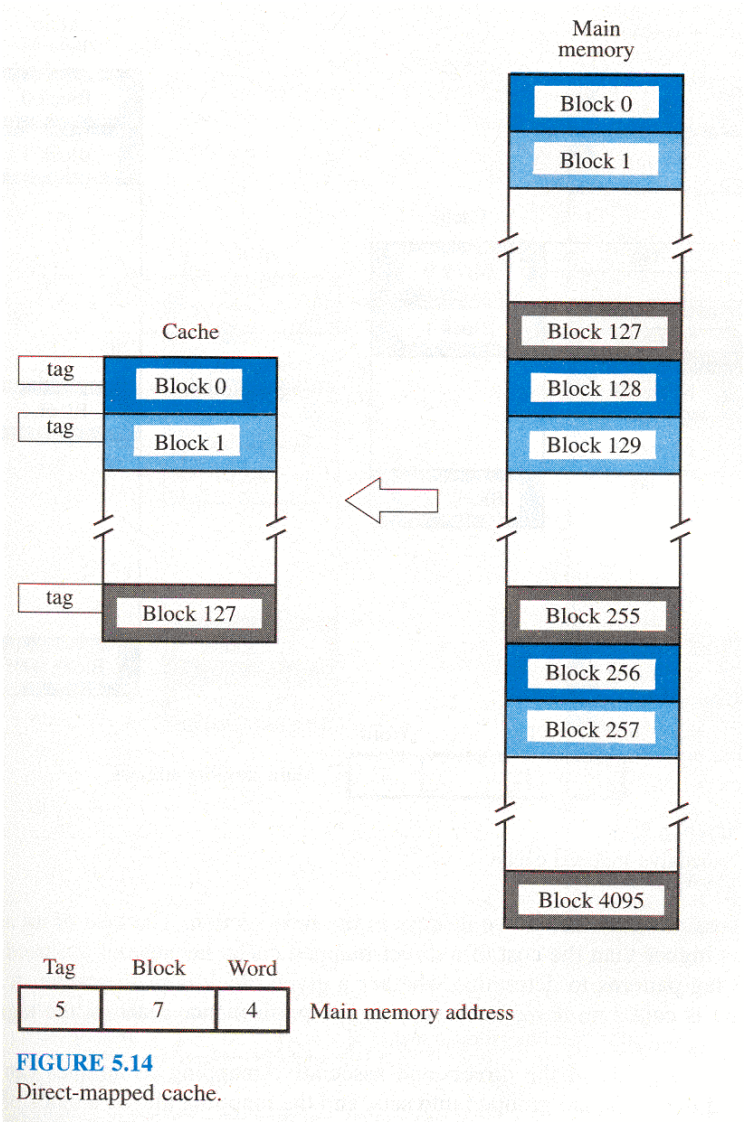


FIGURE 5.14
Direct-mapped cache.

Fully Associative Cache

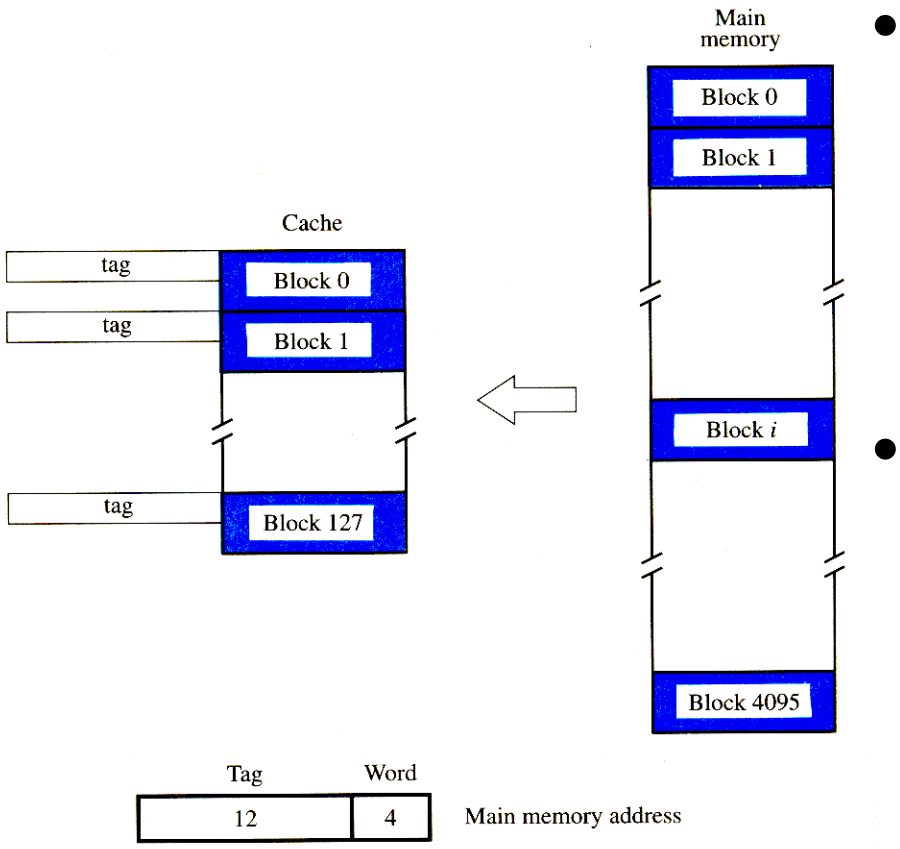


FIGURE 5.15
Associative-mapped cache.

- More efficient cache utilization
  - No wasted cache space
- Slower cache search
  - Must check the tag of every entry in the cache to see if the block we want is there.

Set-associative mapping

- Blocks are grouped into sets
- Each memory block can occupy any block in its set
- This example is 2-way set-associative
- *Which of the two blocks do we replace?*

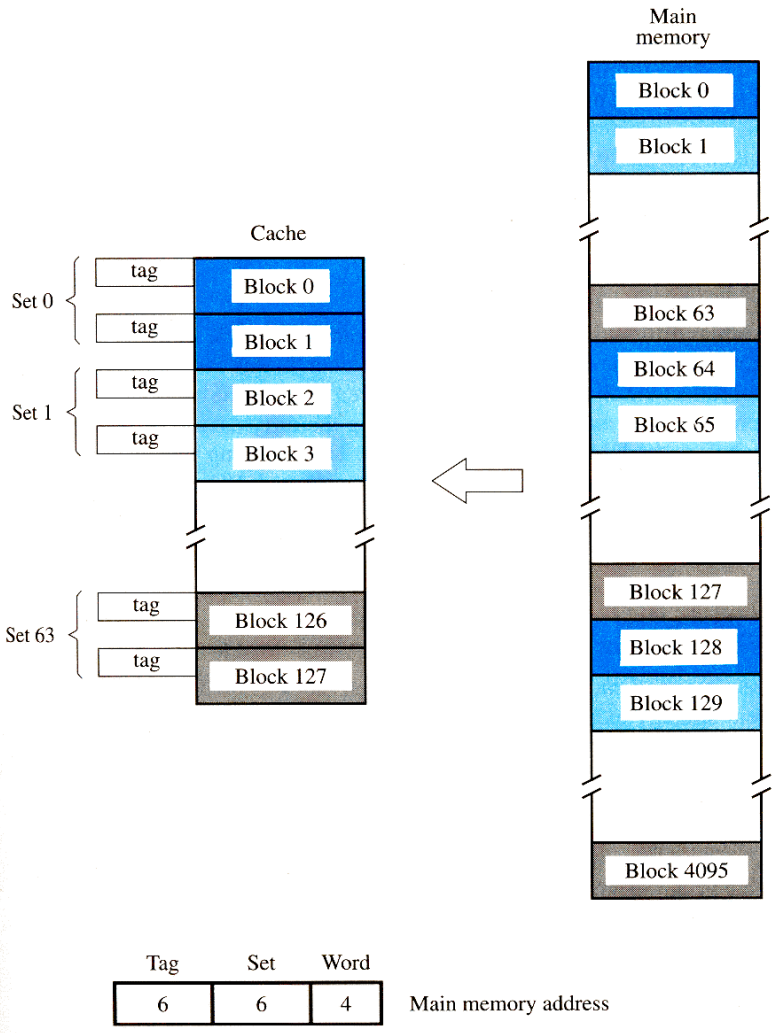


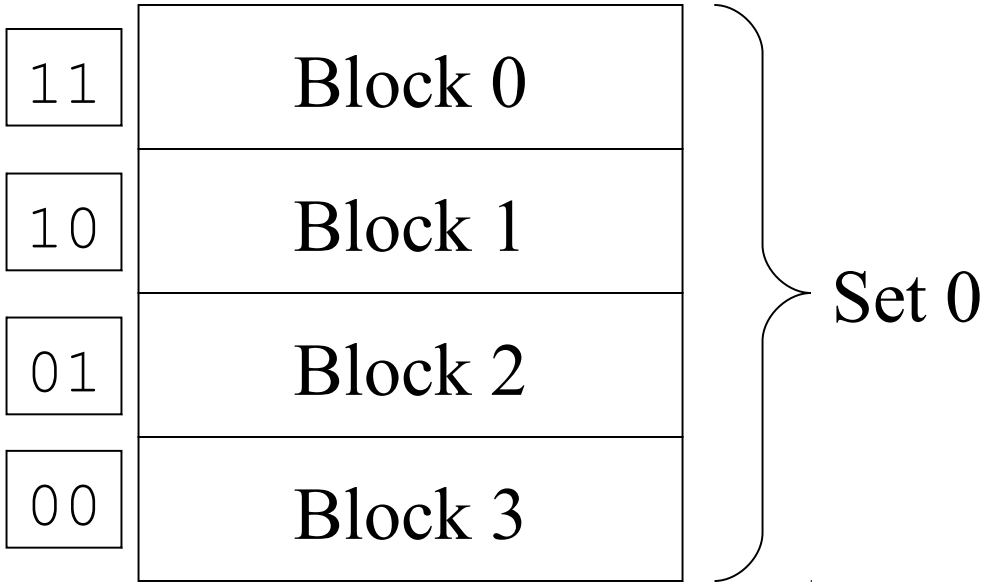
FIGURE 5.16
Set-associative-mapped cache with two blocks per set.

Replacement algorithms

- Random
- Oldest first
- Least accesses
- Least recently used (LRU): replace the block that has gone the longest time without being referenced.
  - This is the most commonly-used replacement algorithm
  - Easy to implement with a small counter…

Implementing LRU replacement

- Suppose we have a 4-way set associative cache…
- Hit:
  - Increment lower counters
  - Reset counter to `00`
- Miss
  - Replace the `11`
  - Set to `00`
  - Increment all other counters



Interactions with DMA

- If we have a write-back cache, DMA must check the cache before acting.
  - Many systems simply flush the cache before DMA write operations
- What if we have a memory word in the cache, and the DMA controller changes that word?
  - *Stale data*
- We keep a *valid* bit for each cache line. When DMA changes memory, it must also set the *valid* bit to 0 for that cache line.
  - *"Cache coherence"*

Typical Modern Cache Architecture

- L0 cache
  - On chip
  - Split 16 KB data/16 KB instructions
- L1 cache
  - On chip
  - 64 KB unified
- L2 cache
  - Off chip
  - 128 KB to 16+ MB

Memory Interleaving

- Memory is organized into *modules* or *banks*
  - Within a bank, capacitors must be recharged after each read operation
  - Successive reads to the same bank are slow

<u>Non-interleaved</u>

Module	Byte	
0000	0100	0200
0002	0102	0202
0004	0104	0204
0006	0106	0206
...	...	...
00F8	01F8	02F8
00FA	01FA	02FA
00FC	01FC	02FC
00FE	01FE	02FE

<u>Interleaved</u>

Byte		Module
0000	0002	0004
0006	0008	000A
000C	000E	0010
0012	0014	0016
...	...	...
02E8	02EA	02EC
02EE	02F0	02F2
02F4	02F6	02F8
02FA	02FC	02FE

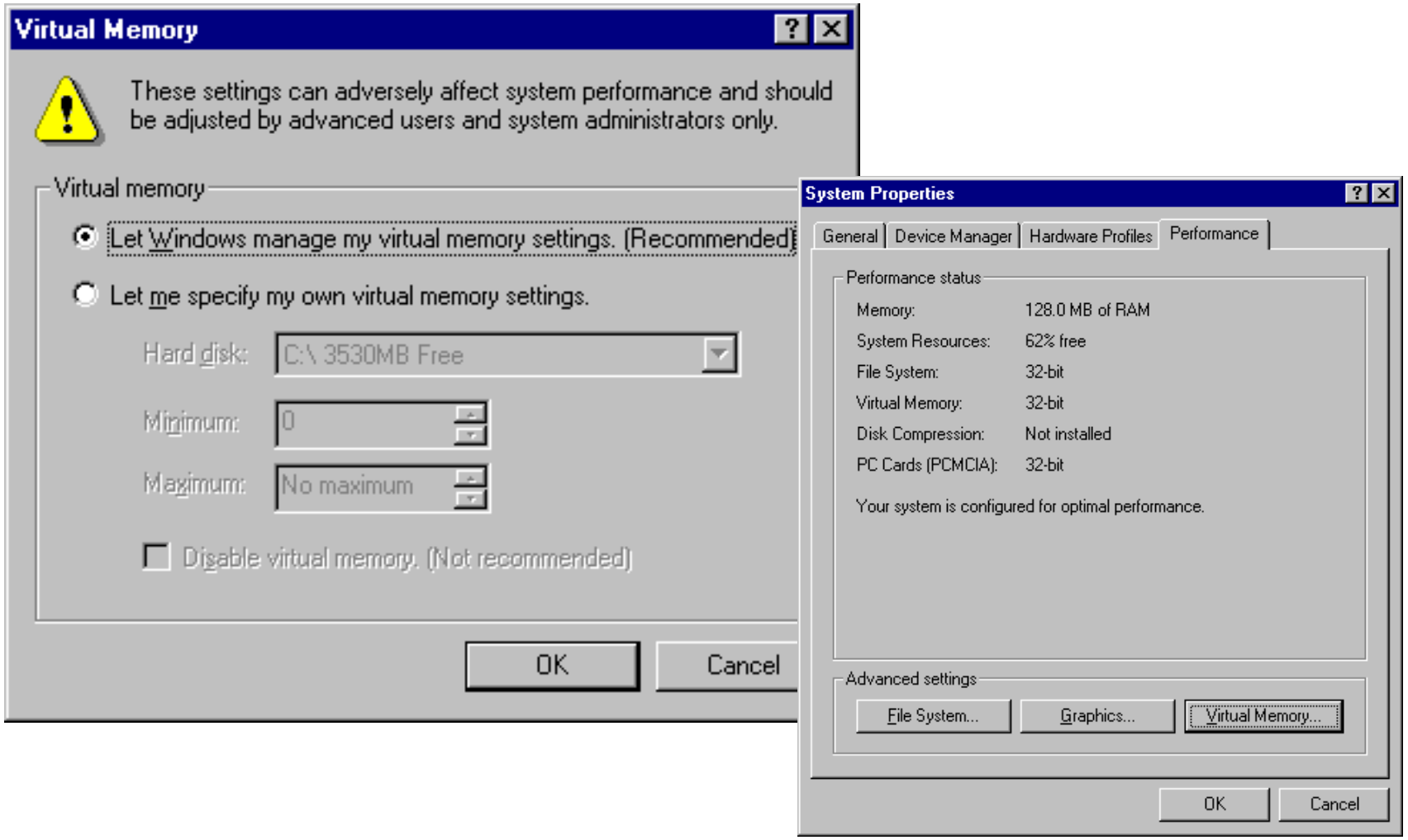
Measuring Cache Performance

- No cache: Often about 10 cycles per memory access
- Simple cache:
  - $t_{ave} = h\text{C} + (1\text{-}h)M$
  - C is often 1 clock cycle
  - Assume M is 17 cycles (to load an entire cache line)
  - Assume h is about 90%
  - $t_{ave} = .9\ (1) + (.1)17 = 2.6$ cycles/access
  - *What happens when h is 95%?*

Multi-level cache performance

- $t_{ave} = h_1 C_1 + (1-h_1)\, h_2 C_2 + (1-h_1)\,(1-h_2)\, M$
  - h_1 = hit rate in primary cache
  - h_2 = hit rate in secondary cache
  - C_1 = time to access primary cache
  - C_2 = time to access secondary cache
  - M = miss penalty (time to load an entire cache line from main memory)

Virtual Memory



Virtual Memory: Introduction

- Motorola 68000 has 24-bit memory addressing and is byte addressable
  - Can address 2^{24} bytes of memory – 16 MB
- Intel Pentiums have 32-bit memory addressing and are byte addressable
  - Can address 2^{32} bytes of memory – 4 GB
- What good is all that address space if you only have 256MB of main memory (RAM)?

Virtual Memory: Introduction

- Unreal Tournament (full install) uses 2.4 gigabytes on your hard disk.
- You may only have 256 megabytes of RAM (main memory).
- How can you play the game if you can't fit it all into main memory?
- Other Examples:
  - Neverwinter Nights – 2 gigabytes
  - Microsoft Office – 243 megabytes
  - Quicken Basic 2000 – 28 megabytes

Working Set

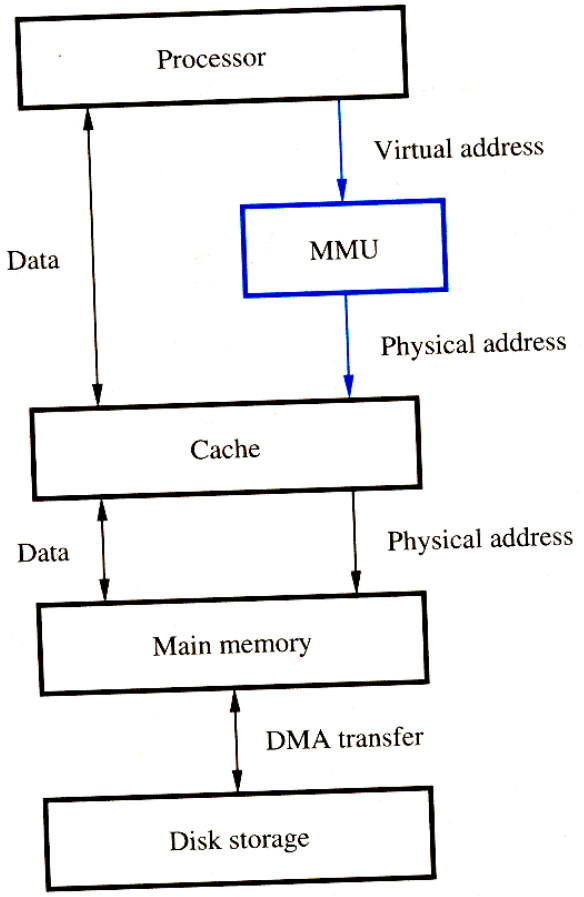
- Not all of a program needs to be in memory while you are executing it:
  - Error handling routines are not called very often.
  - Character creation generally only happens at the start of the game… why keep that code easily available?
- *Working set* is the memory that is consumed at any moment by a program while it is running.
  - Includes stack, allocated memory, active instructions, etc.
- Examples:
  - Unreal Tournament – 100MB (requires 128MB)
  - Internet Explorer – 20MB

Virtual Memory

- In modern computers it is possible to use more memory than the amount physically available in the system
- Memory not currently being used is temporarily stored on magnetic disk
- Essentially, the *main memory acts as a* ***cache*** for the virtual memory on disk.

Memory Management Unit (MMU)

- Virtual memory must be invisible to the CPU
  - Appears as one large address space



- The MMU sits between the CPU and the memory system
- Translates virtual addresses to real (physical) addresses

Paged Memory

- Memory is divided into *pages*
  - Conceptually like a cache block
  - Typically 2K to 16K in size
  - A page can live anywhere in main memory, or on the disk
- Like cache, we need some way of knowing what page is where in memory
  - Page table instead of tags
  - Page table base register stores the address of the page table

Page lookup in the page table

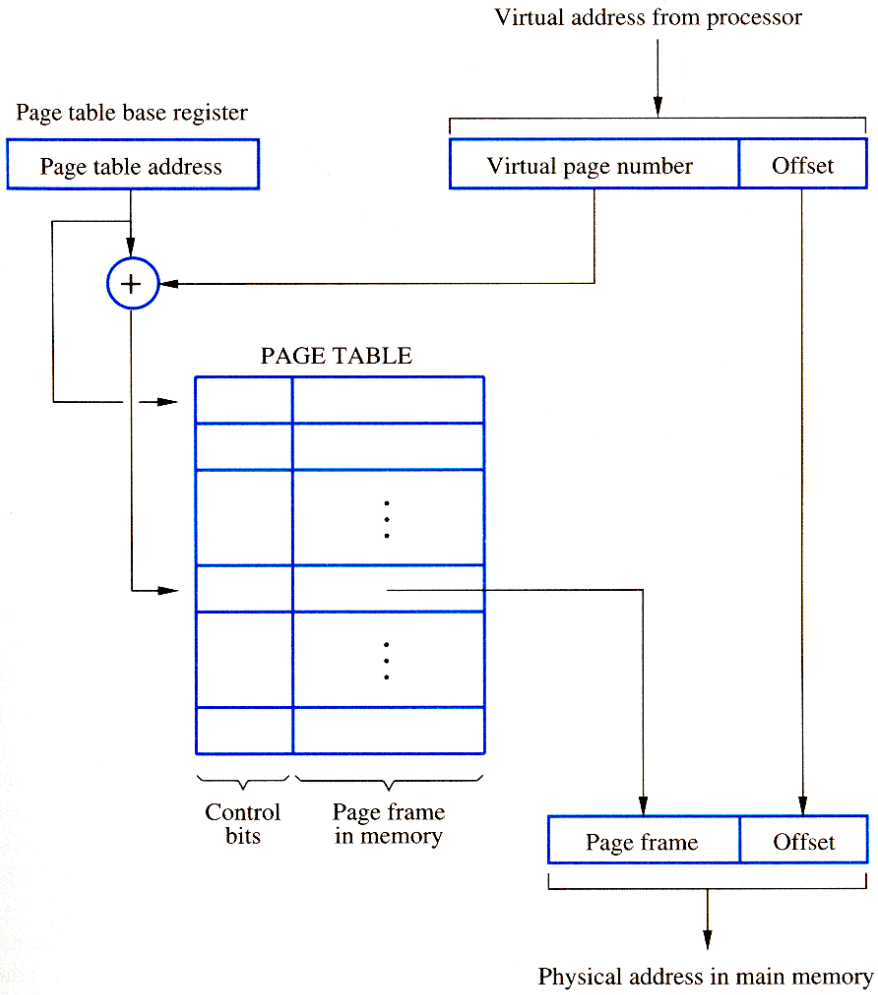


FIGURE 5.26
Virtual-memory address translation.

- Control bits:
  - *Valid*
  - *Modified*
  - *Accessed*
    - Occasionally cleared by the OS
    - Used for LRU replacement
- TLB: A cache for the page table

The TLB

- A special, fully-associative cache for the page table is used to speed page lookup
- This cache is called the *Translation Lookaside Buffer* or *TLB*.

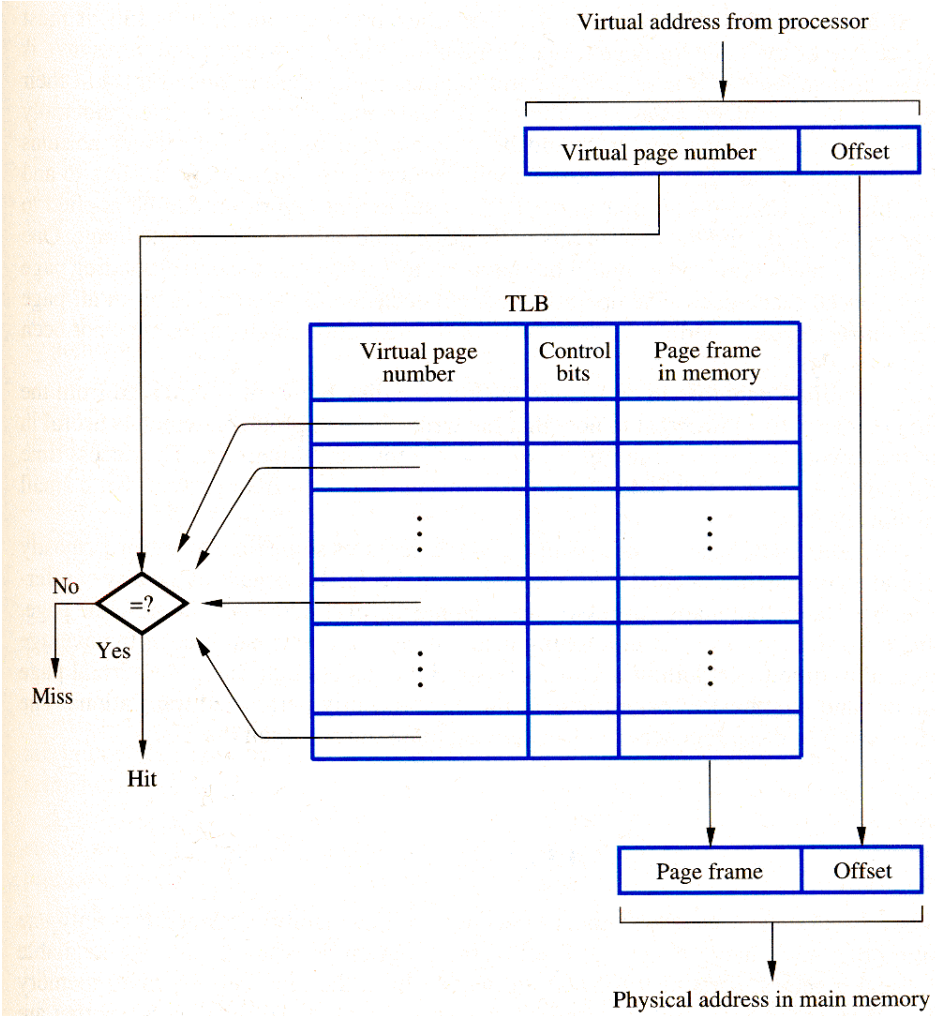


FIGURE 5.27
Use of an associative-mapped TLB.

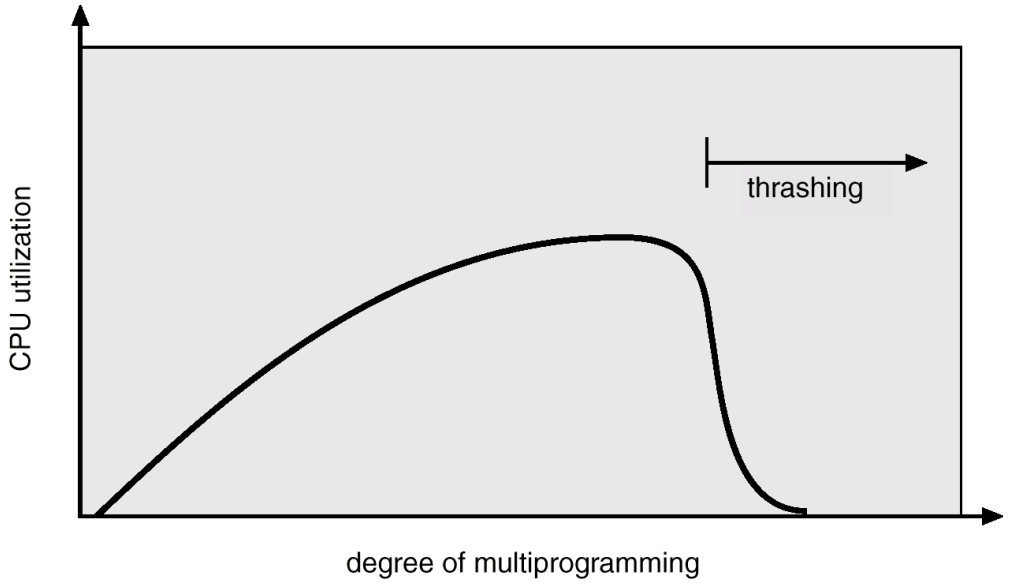
Page Faults

- If the memory address desired cannot be found in the TLB, then a *page fault* has occurred, and the page containing that memory must be loaded from the disk into main memory.
- Page faults are costly because moving 2K – 16K (page size) can take a while.
- What if you have too many page faults?

Thrashing

- If a process does not have enough frames to hold its current working set, the page-fault rate is very high
- Thrashing
  - a process is thrashing when it spends more time paging than executing
  - w/ local replacement algorithms, a process may thrash even though memory is available
  - w/ global replacement algorithms, the entire system may thrash
    - Less thrashing in general, but is it fair?

Thrashing Diagram



- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size
- What should we do?
  - suspend one or more processes!

Program Structure

- How should we arrange memory references to large arrays?
  - Is the array stored in row-major or column-major order?
- Example:
  - Array A[1024, 1024] of type integer
  - Page size = 1K
    - *Each row is stored in one page*
  - System has one frame

  - Program 1

```
for i := 1 to 1024 do
for j := 1 to 1024 do
    A[i,j] := 0;
```

    1024 page faults

  - Program 2

```
for j := 1 to 1024 do
for i := 1 to 1024 do
    A[i,j] := 0;
```

    1024 x 1024 page faults