Virtual Memory

Ref: Chapter 7

Disk caching

- Use main memory as a *cache* for magnetic disk.
- We can do this for a number of reasons:
  - speed up disk access
  - pretend we have more main memory than we really have.
  - support multiple programs easily (each can pretend it has all the memory).

Our focus

- We will focus on using the disk as a storage area for chunks of main memory that are not being used.
- The basic concepts are similar to providing a cache for main memory, although we now view part of the hard disk as *being the memory*.

Virtual memory

Consider a machine with a 32 bit address space:

- it probably doesn't have 2^{32} = 4 GB of main memory!
- How do we write programs without knowing how much memory is really available ahead of time?
- Why not *pretend* we always have 4GB, and if we use more than we really have, store some blocks on the hard disk.
  - this must happen automatically to be useful.

Motivation

- Pretend we have 4GB, we really have only 64MB.
- At any time, the processor needs only a small portion of the 4GB memory.
  - only a few programs are active
  - an active program might not need all the memory that has been reserved by the program.
- We just keep the stuff needed in the main memory, and store the rest on disk.

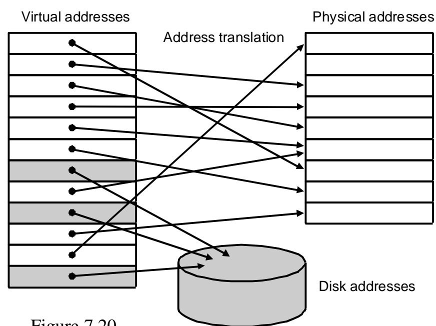


Figure 7.20

A Program's view of memory

- We can write programs that address the *virtual memory*.
- There is hardware that translates these virtual addresses to physical addresses.
- The operating system is responsible for managing the movement of memory between disk and main memory, and for keeping the address translation table accurate.

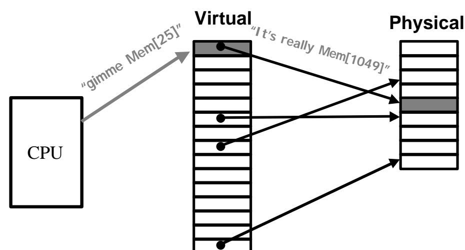
Terminology

- *page*: The unit of memory transferred between disk and the main memory.
- *page fault*: when a program accesses a virtual memory location that is not currently in the main memory.
- *address translation*: the process of finding the physical address that corresponds to a virtual address.

block

miss

Virtual Memory and Address Translation



Translation and Pages

- Only the page number need be translated.
- The offset within the page stays constant.

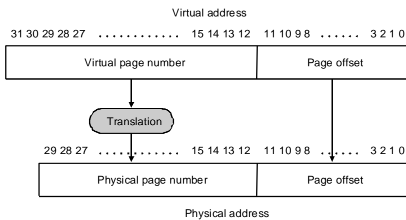


Figure 7.21

CPU and address translation

- The CPU doesn't need to worry about address translation – this is handled by the memory system.
- As far as the CPU is concerned, it *is* using physical addresses.

Advantages

- A program can be written (linked) to use whatever addresses it wants to! It doesn't matter where it is physically loaded!
- When a program is loaded, it doesn't need to be placed in continuous memory locations
  – any group of physical memory *pages* will do fine.

Design Issue

- A Page Fault is a disaster!
  - disk is very, very, very slow compared to memory – millions of cycles!
- Minimization of faults is the primary design consideration for virtual memory systems.
- This page is important! You can tell by the number of exclamation points!!!

Did he say millions?
Yes, I believe he did!

CompOrg Fall 2000 - Memory part 2 13

Minimizing faults

- Pages should be big enough to make a transfer from disk worthwhile. 4KB-64KB are typical sizes.
- Fully associative placement is the most flexible (will reduce the rate of faults).
  - software handles the placement of pages, and unless we let CS1 students write this code, it won't take millions of cycles.

CompOrg Fall 2000 - Memory part 2 14

What about rights writes?

- Write through is not practical for a virtual memory system (writes to disk are way to slow).
- Write back is always used.
  - write the entire page to disk only when kicked out of the main memory and placed on disk.

CompOrg Fall 2000 - Memory part 2 15

The *dirty* bit

- It would be wasteful to *always* write an entire page to disk if nothing in the page has changed.
- A flag is used to keep track of which pages have been changed in main memory (if not change happens, no need to write the page to disk).
- The flag is called the *dirty bit*.

Address Translation

- Address translation must be fast (it happens to every memory access).
- We need a fully associative placement policy.
- We can’t afford to go looking at every virtual page to find the right one
  - we don’t use the *tag bits* approach

Page Table

- We need a large table that holds the physical address for each virtual page.
- Want virtual page 1234? Look at row 1234 in the table.
  - the page table is a big array indexed by virtual page number.
- The table will be huge! 2^{32}/page size.

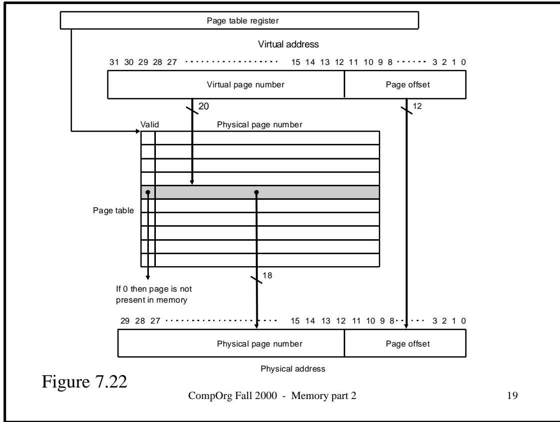


Figure 7.22

Processes and Page Tables

- Each process has it's own page table!
  - each program can pretend it is loaded and running at the same address.
- One page table is huge, now we need to worry about lots of page tables.
- We can't possible included dedicated hardware that holds all these page tables.

Page Tables memory needs

- Assume 32 bit virtual address space.
- Assume 16K Byte page size.
  - each page table needs $2^{32}/2^{14} = 2^{18}$ elements.
- We would like to support 256 different processes.
- We need $2^8 * 2^{18} = 2^{26}$ page table elements, assume each is 1 word wide.
- Total needed is 256 MBytes!

Page Table Elements

- Each element in the page table needs to include:
  - a valid bit.
  - if the page is in memory, the physical address.
  - if the page is on disk, some indication of where on the disk

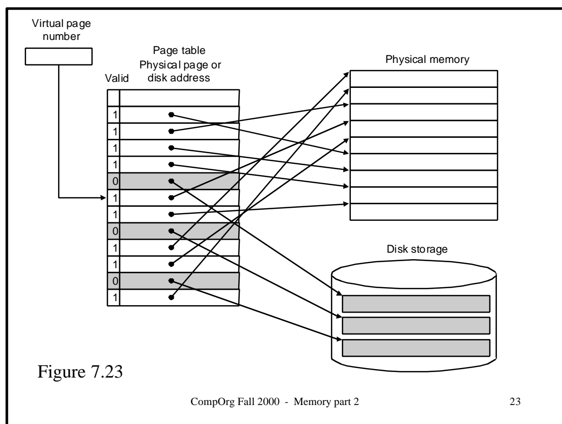


Figure 7.23

I need to go buy more memory!

- Page tables are stored in main memory.
- Most programs are small, so we don't need to actually create the entire page table for each process.
  - just enough to cover the actual pages that have been reserved for use by the program.
  - this number will be quite small (a few thousand pages is enough for a large program).

Speed of address translation

- Page tables are in memory.
- We need to access an element of the page table every time a translation is needed.
- A translation is needed on every memory access!
- Every memory access really requires 2 memory accesses!

Making address translation fast

- We can create a dedicated cache that holds the most recently used page table entries.
  - the same page table entry is used for all memory locations in the page. Spatial Locality.
- This cache is called a *Translation Lookaside Buffer* (TLB).

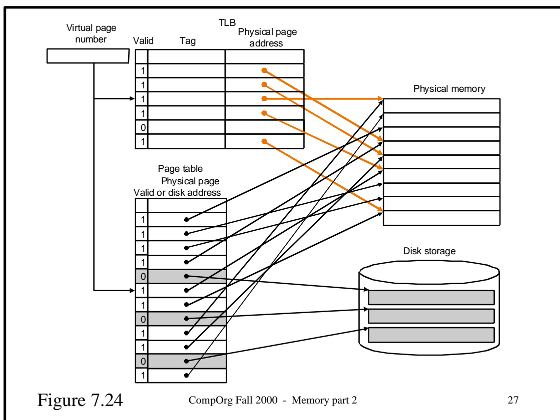


Figure 7.24

DecStation 3100 TLB

- 32 bit address space
- 4KB Page size
  - virtual page address is 20 bits.
- TLB has 64 slots
  - each has 20 bit tag, 20 bit physical page address, a valid bit and a dirty bit.
  - fully associative.

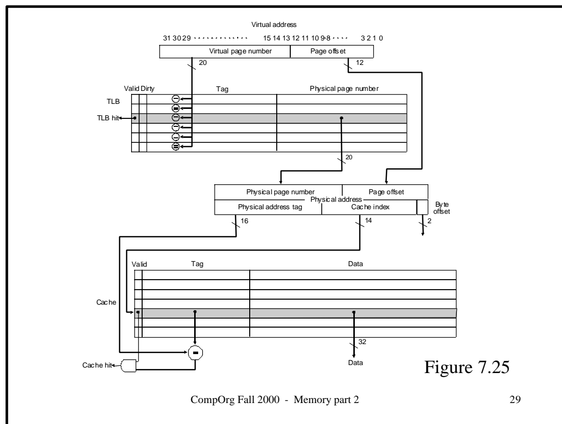


Figure 7.25

Cache + Virtual Memory

- The Decstation 3100 does address translation *before* the cache.
- The cache operates on physical memory addresses.
- It is also possible to cache virtual memory, although there are some problems.
  - if programs can share pages, a single word from physical memory could end up in the cache twice! (the same physical location could have 2 different virtual addresses).

Protection

- Virtual memory allows multiple processes to share the same physical memory.
- What if my process tries to write to your process's memory?
  - we don't want this to be possible!
  - we don't even want it to be able to read!

Independent Page Tables

- Each process has it's own page table.
- All page tables are created by the operating system – your program can't change it's own page table.
- Supporting virtual memory requires a combination of hardware and software.

Common Issues

- There are a number of issues that are common to both cache and virtual memory system design:
  - block placement policy.
  - how is a block found?
  - block replacement policy.
  - write policy.

Block Placement Options

- Direct-Mapped
  - cheap, easy to implement, relatively high miss rate.
- Set Associative
  - middle ground
- Fully Associative
  - expensive (lots of hardware or software), minimizes miss rate.

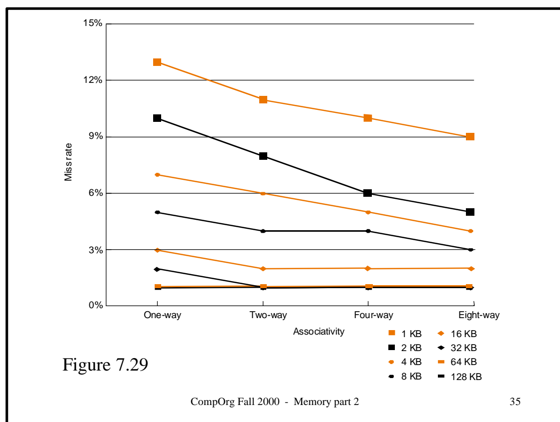


Figure 7.29

How is a block found?

This depends on placement policy.

- Direct Mapped: uses an index.
- Set Associative: index selects a set, and we need to look at all set elements.
- Fully Associative: need to look at all elements.

Replacement Policies

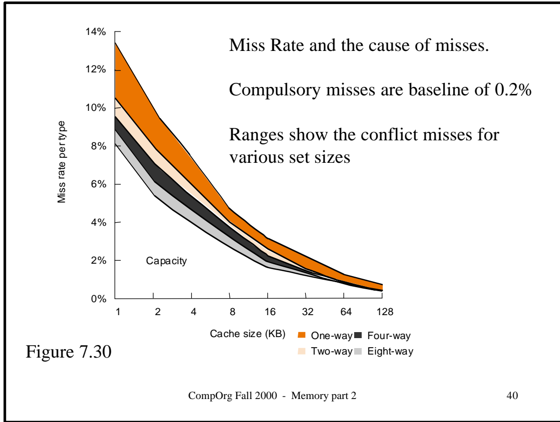
- Direct-Mapped: not an issue.
- Set and fully associative
  - LRU (*least recently used*) hard to implement in hardware for large sets, often approximated.
  - random easy to implement, does nearly as well as LRU approximations.
- LRU is always used (or approximated) for virtual memory.

Write Policies

- Write-Through: update the cache and lower level memory.
- Write-Back: update the cache only. When block/page is booted from the cache - write to lower-level memory if any changes.

Where do Mrs. misses come from?

- Compulsory misses: the first access is always a miss. Can't avoid these.
- Capacity misses: cache can't hold all the blocks needed.
- Conflict misses: multiple blocks compete for the same cache slot(s) and collide.



Miss Rate and the cause of misses.

Compulsory misses are baseline of 0.2%

Ranges show the conflict misses for various set sizes

Figure 7.30

Cache friendly code

(a great name for a band!)

- There are sometimes things you can do to your program to take advantage of the cache.
  - usually it's not necessary to know much about the specific architecture of the cache on which a program is run.
- The patterns of array element access is one good example.

Matrix Multiplication

```
for (i=0; i!=500; i++)
  for (j=0;j!=500; j++)
    for (k=0;k!=500; k++)
      x[i][j] = x[i][j] + y[i][k]*z[k][j];
```

almost twice as fast on SGI Mips R4000

```
for (k=0; k!=500; k++)
  for (j=0;j!=500; j++)
    for (i=0;i!=500; i++)
      x[i][j] = x[i][j] + y[i][k]*z[k][j];
```