

19. Virtual memory

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan in Ann Arbor, USA

© T. Austin & M. Papaefthymiou, 2009

The material in this presentation cannot be
copied in any form without our written permission

This week in 370

- ❑ Today: Virtual memory
- ❑ Friday March 27: Pr3 due
- ❑ Tuesday March 31: Exam review in class
- ❑ **Thursday April 2: Second Midterm in class**

Class Problem 5

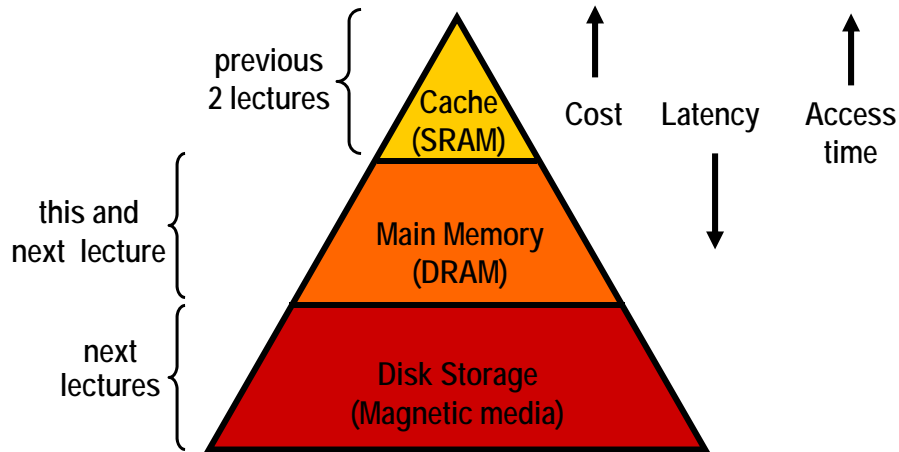
Given the following: instruction breakdown:
LW 25%, Add 20%, BEQ 20%, SW, 15%, Nand 20%;
branches are taken 60% of the time, LW followed by
immediate use occurs 20% of the time. The instruction
cache hit rate is 90% and the data cache hit rate is 80%.
Assume each cache miss requires 10 stall cycles.
Data hazards – detect and forward
Control hazards – predict not taken and squash

Calculate the CPI

Lectures on memory

1. Introduction to memory systems
2. Basic cache design
3. Write-back and Write-through caches, Associativity
4. Cache performance, cache misses, cache integration
5. Virtual Memory
6. Making VM faster: TLBs
7. Disk storage

Storage Hierarchy



The issues(s)

- ❑ We run many programs on a same machine
 - Each of them may require GBs of storage
 - Unrelated programs should not have access to each other's storage

- ❑ DRAM is too expensive to buy 100s GB, but disk space is not...
 - We want our system to work even if it requires more DRAM than we bought.
 - We also don't want a program that works on a machine with 2048 MB of DRAM to stop working if we try to run it on a machine with only 512 MB of main memory.

- ❑ And, it would be nice to be able to enforce different policies on different portions of the memory (e.g.: read-only, etc)

Solution 1: User control

- ❑ Leave the problem to the programmer
 - Assume the programmer knows the exact configuration of the machine.
 - Programmer must either make sure the program fits in memory, or break the program up into pieces that do fit and load each other off the disk when necessary

- ❑ Not a bad solution in some domains
 - The hardware desing is simple
 - Playstation 2, cell phones, etc.

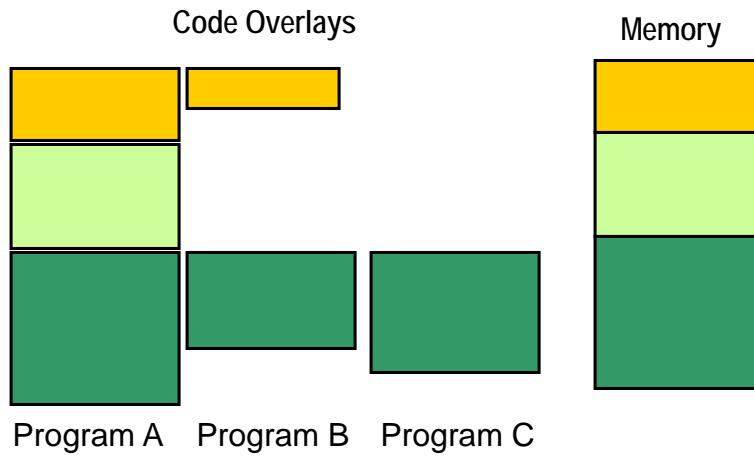
Solution 2: Overlays

- ❑ A little automation to help the programmer
 - build the application in **overlays**
 - Two pieces of code/data may be overlaid iff
 - They are not active at the same time
 - They are placed in the same memory region

- ❑ Managing overlays is performed by the compiler
 - Good compilers may determine overlay regions
 - Compiler adds code to read the required overlay memory off the disk when necessary

- ❑ The hardware design is still simple (most of the time)

Overlay example



Solution 3: Virtual memory

- Build new hardware and software that automatically translates each memory reference from a

virtual address

(which the programmer sees as an array of bytes)

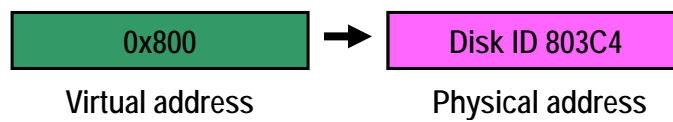
to a

physical address

(which the hardware uses to either index DRAM or identify where the storage resides on disk)

Basics of Virtual Memory

- ❑ Any time you see the word virtual in computer science and architecture it means “using a level of indirection”
- ❑ Virtual memory hardware changes the virtual address the programmer sees into the physical one the memory chips see.



Virtual Memory View

- ❑ Virtual memory lets the programmer “see” a memory array **larger** than the DRAM available on a particular computer system.
- ❑ Virtual memory enables multiple programs to share the physical memory without:
 - Knowing other programs exist (**transparency**).
 - Worrying about one program modifying the data contents of another (**protection**).

Managing virtual memory

- ❑ Managed by hardware logic *and* operating system software.
 - Hardware for speed
 - Software for flexibility and because disk storage is controlled by the operating system.

- ❑ The hardware must be designed to support VM

Virtual Memory

- ❑ Treat main memory like a cache
 - Misses go to the disk
- ❑ How do we minimize disk accesses?
 - Buy lots of memory.

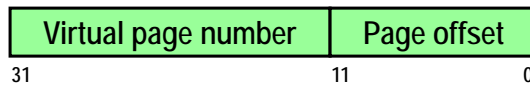
 - Exploit temporal locality
 - Fully associative? Set associative? Direct mapped?

 - Exploit spatial locality
 - How big should a block be?

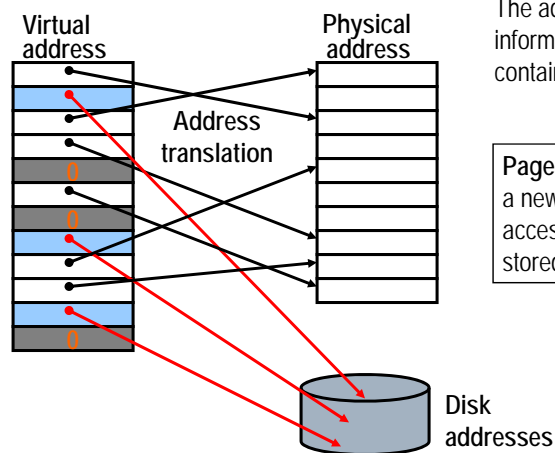
 - Write-back or write-through?

Virtual memory terminology

- ❑ Blocks are called **Pages**
 - A virtual address consists of
 - A virtual page number
 - A page offset field (low order bits of the address)
- ❑ Misses are call **Page faults**
 - and they are generally handled as an exception



Address Translation

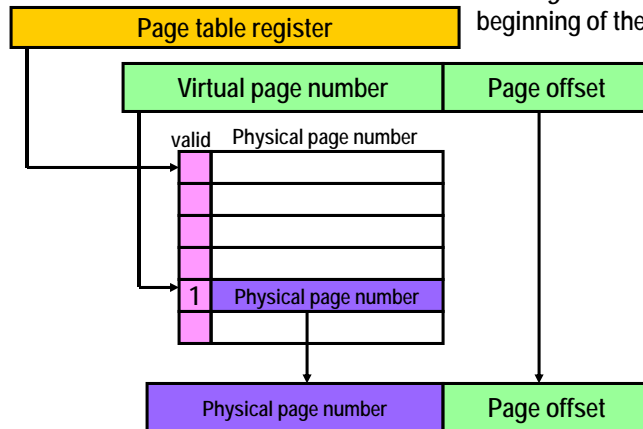


The address translation information of the program is contained in the **Page Table**

Page Table:
a new data structure
accessed by the OS and HW
stored in memory

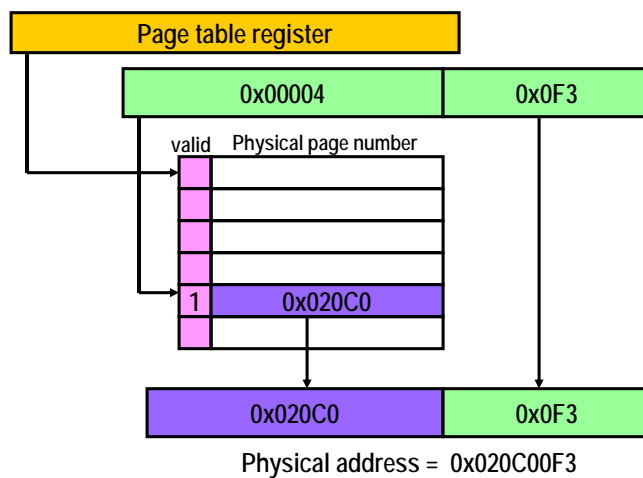
Page table components

The *Page table register* points to the beginning of the page table

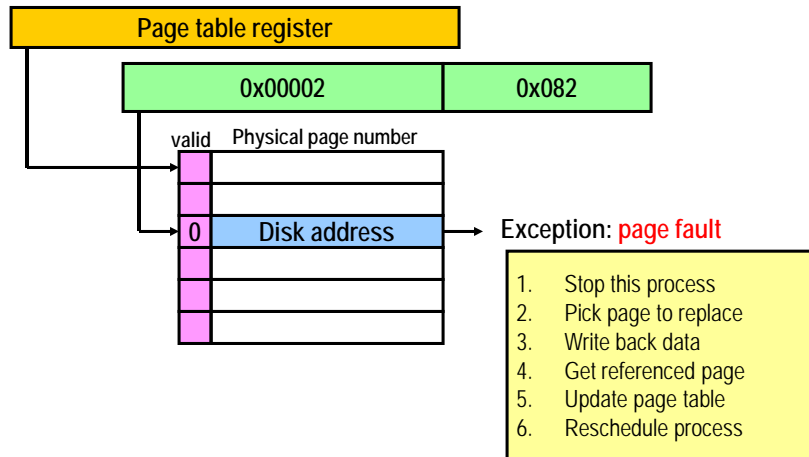


Page table components - Example

Virtual address = 0x000040F3



Page faults



How do we find it on disk?

- ❑ That is not a hardware problem! 😊
- ❑ Most operating systems partition the disk into logical devices (C: , D: , /home, etc.)
- ❑ They also have a hidden partition to support the disk portion of virtual memory
 - **Swap partition** on UNIX machines
 - You then index into the correct page in the swap partition.

Class Problem

- ❑ Given the following
 - 4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.
 - The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.
- ❑ Fill in the table on the next slide for each reference

Class Problem (continued)

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C			
0x01F0C			
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

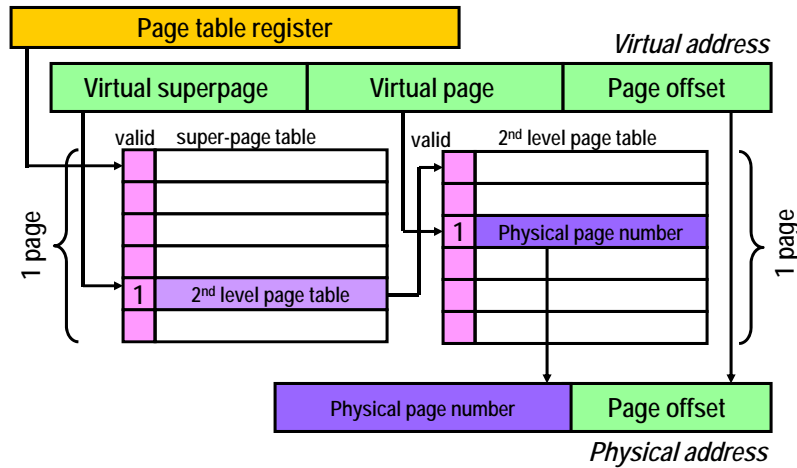
Size of the page table

- ❑ How big is a page table entry?
 - For MIPS the virtual address is 32 bits
 - If the machine can support $1\text{GB} = 2^{30}$ bytes of physical memory and we use pages of size $4\text{KB} = 2^{12}$, then the physical page number is $30 - 12 = 18$ bits.
Plus another **valid bit** + other useful stuff (**read only, dirty, etc.**)
 - Let say about 3 bytes.
- ❑ How many entries in the page table?
 - MIPS virtual address is 32 bits – 12 bit page offset = 2^{20} or ~1,000,000 entries
- ❑ Total size of page table: $2^{20} \times 3$ bytes ~ 3 MB

How can you organize the page table?

1. Continuous 3MB region of physical memory
 - Cons: Always takes 3MB regardless of how much physical memory is used
2. Use a hash function instead of an array
 - Slower, but less memory
3. Use a multi-level page table! (Build a hierarchical page table, keep in physical memory only the translations used)
 - Super page table in physical memory
 - Second (and maybe third) level page tables only if needed
 - **Size is proportional to the amount of memory used**

Hierarchical page table – Possible structure



Hierarchical page table – Possible structure (cont'd)

- | | | |
|--------------------------|--------------------------------------------------------|----------------------|
| <input type="checkbox"/> | How many bits in the virtual superpage field? | <i>Example</i>
10 |
| <input type="checkbox"/> | How many bits in the virtual page field? | 10 |
| <input type="checkbox"/> | How many bits in the page offset? | 12 |
| <input type="checkbox"/> | How many pages in the super page table? | 1024 |
| <input type="checkbox"/> | How many bytes for each entry in the super page table? | 4 |
| <input type="checkbox"/> | How many pages in the 2nd level of the page table? | 1024 |
| <input type="checkbox"/> | How many bytes for each VPN in a 2nd level table? | 4 |
| <input type="checkbox"/> | What is the total size of the page table? | $4K+n*4K$ |

(this example is how 32bit x86 works)

Putting it all together

- ❑ Loading your program in memory
 - Ask operating system to create a new process
 - Construct a page table for this process
 - Mark all page table entries as invalid with a pointer to the disk image of the program
 - That is, point to the executable file containing the binary.
 - Run the program and get an immediate page fault on the first instruction.

Page replacement strategies

- ❑ Page table indirection enables a fully associative mapping between virtual and physical pages.
- ❑ How do we implement LRU?
 - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
 - Reference bit on page, cleared occasionally by operating system. Then pick any “unreferenced” page to evict.

Other VM translation functions

- ❑ Page data location
 - Physical memory, disk, uninitialized data
- ❑ Access permissions
 - Read only pages for instructions
- ❑ Gathering access information
 - Identifying dirty pages by tracking stores
 - Identifying accesses to help determine LRU candidate

Placing caches in a VM system

- ❑ VM systems give us two different addresses: virtual and physical

- ❑ Which address should we use to access the data cache?
 - Virtual address (before VM translation)
 - Faster access? More complex?
 - Physical address (after VM translations)
 - Delayed access?