Lecture 19: Virtual Memory

COS 471a, COS 471b / ELE 375

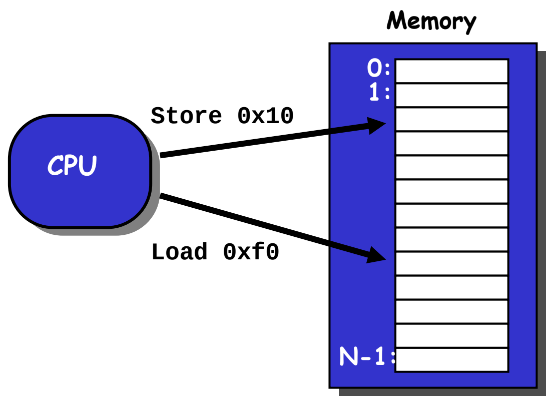
Computer Architecture and Organization

Princeton University
Fall 2004

Prof. David August

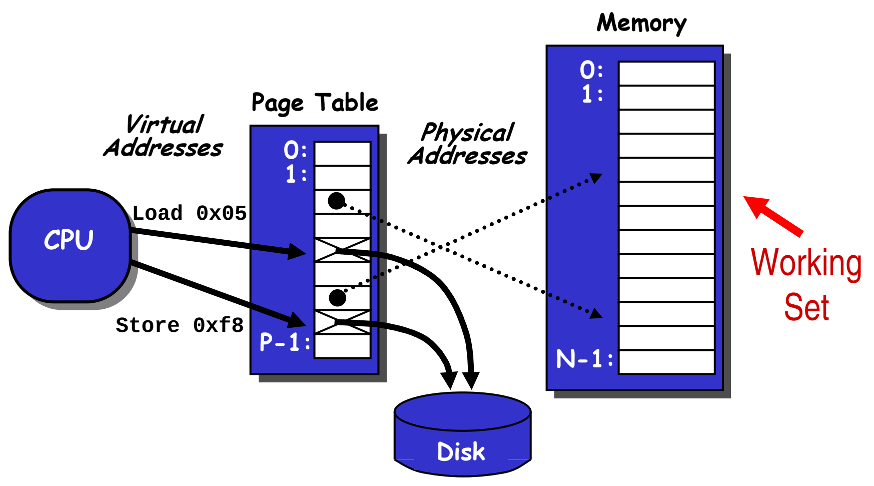
A System with Physical Memory

Examples: Most Cray machines, early PCs, nearly all current embedded systems, etc.

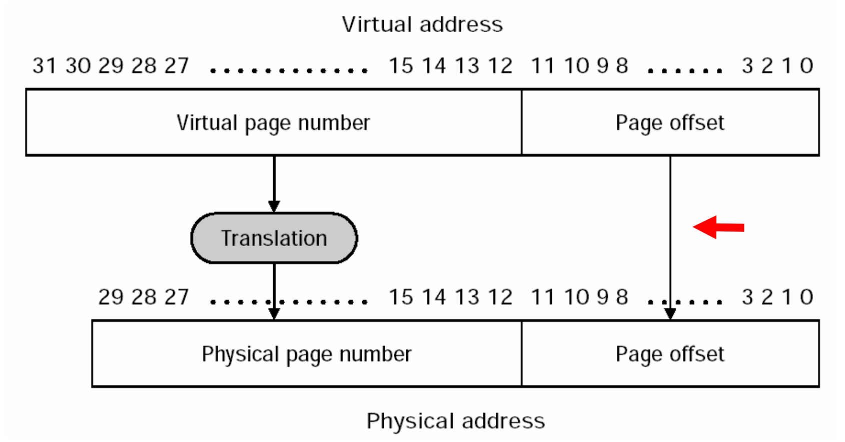


CPU's load or store addresses used directly to access memory.

A System with Virtual Memory



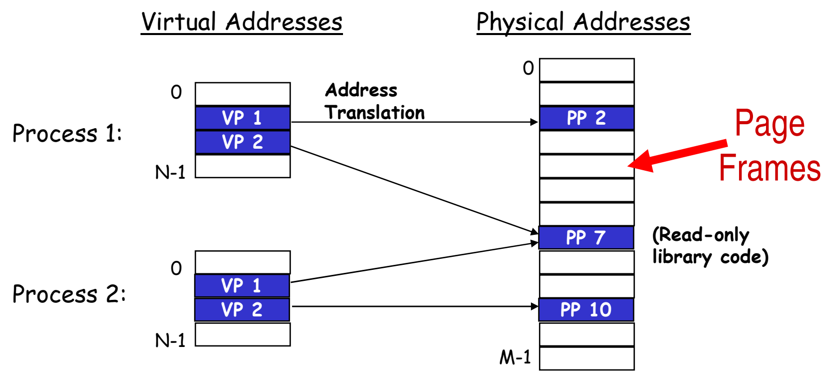
Page Tables



Each process gets its own page table, why?

Separate Virtual Address Spaces

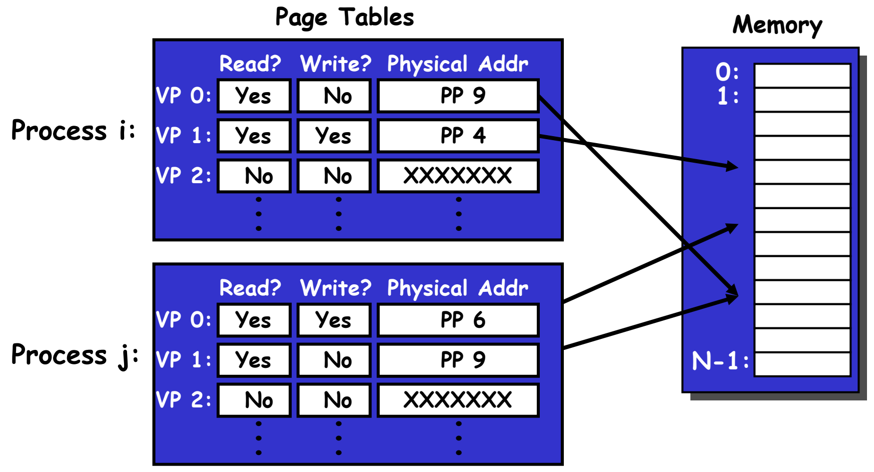
- Each process has its own virtual address space
- OS controls how virtual is assigned to physical memory



Process Protection

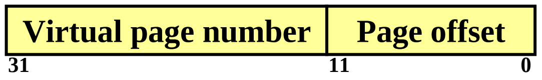
Page table entry contains access rights information

- Hardware enforces this protection (trap into OS if violation occurs)



Virtual Memory Lingo

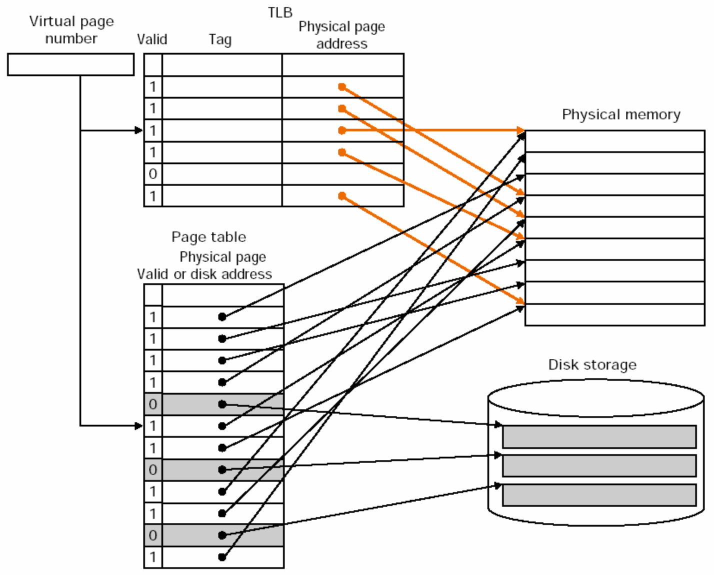
Blocks are called Pages



Misses are called Page faults (handled as an exception)

- Retrieve data from disk
- Huge miss penalty, pages are fairly large (4-8K)
- Reducing page faults is important
- Can handle the faults in software instead of hardware
- Using write-through is too expensive, use writeback

Making Translation Faster: The TLB
Translation Look-Aside Buffer

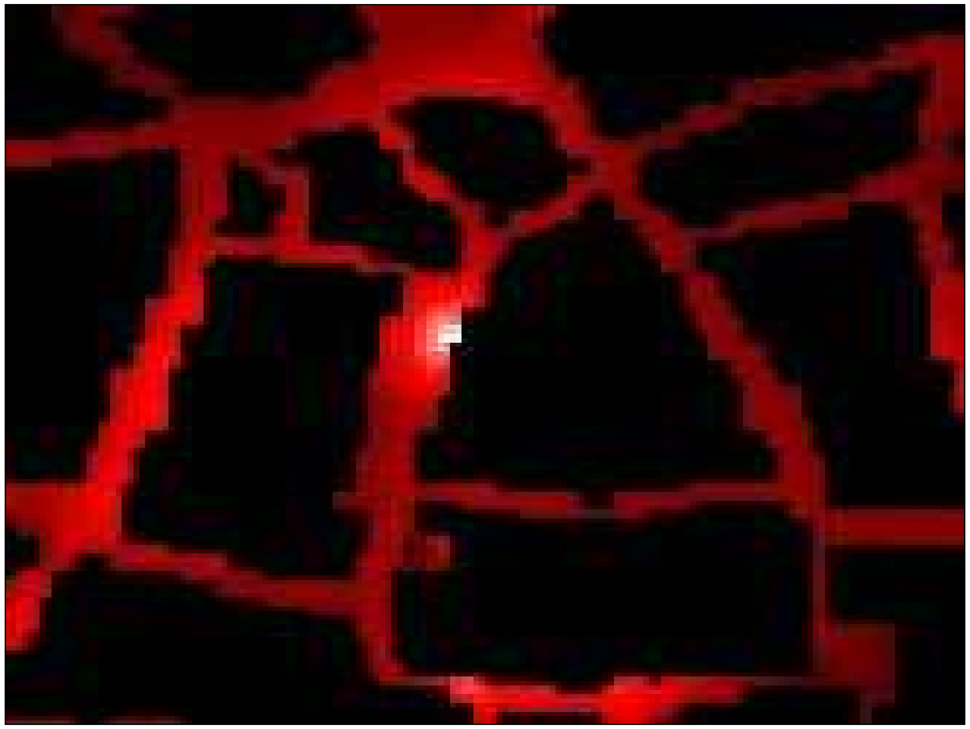


Virtual Memory Summary

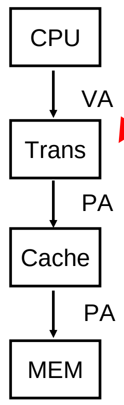
Virtual memory provides
- Protection and sharing
- Illusion of large main memory
- Speed/Caching (when viewed from disk perspective)

- Virtual Memory requires twice as many memory accesses, so cache page table entries in the TLB.
- Three things can go wrong on a memory access
  - TLB miss
  - Page fault
  - Cache miss

Caches and virtual memory?



Virtually Memory and Caches: 3 Options
1. Physically Addressed Cache



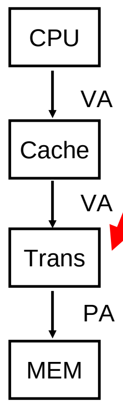
Translate before accessing cache

Cache is:
- Physically Indexed
- Physically Tagged

- Allows multiple processes to have blocks in cache at same time, share pages, etc.
- Access rights checked as part of translation

- Speed?

Virtually Memory and Caches: 3 Options
2. Virtually Addressed Cache



Translate after accessing cache

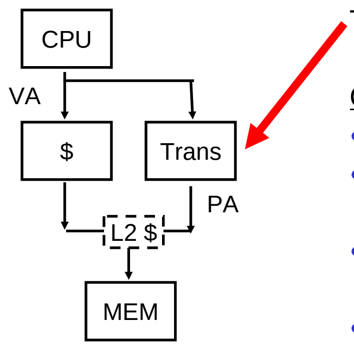
Cache is:
- Virtually Indexed
- Virtually Tagged

- Translate Only on Miss!

- The synonym/alias problem
- How would you make this work?

Virtually Memory and Caches: 3 Options

3. Virtually Indexed, Physically Tagged



Translate during cache access

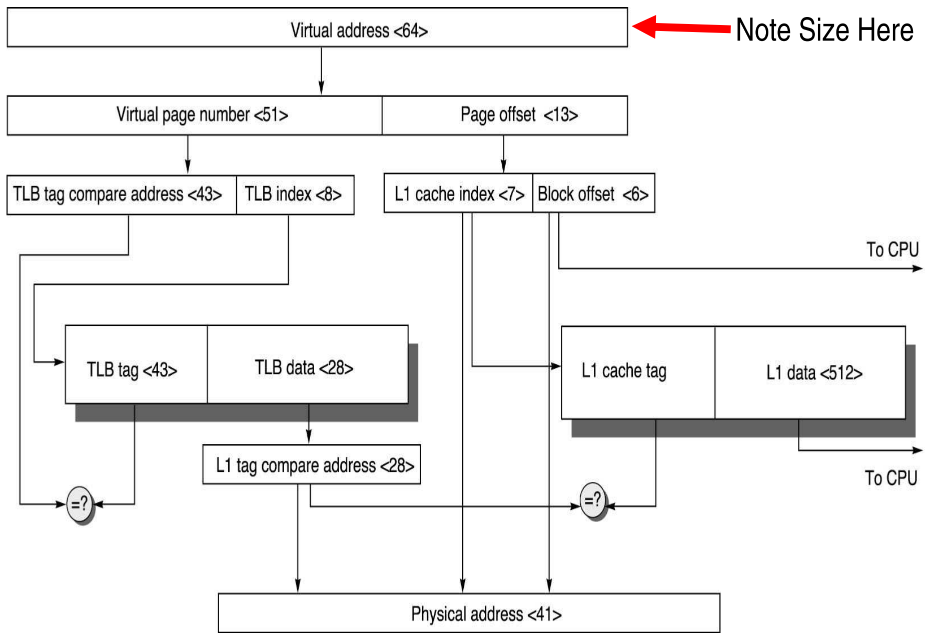
Cache is:

- Physically/Virtually Indexed
- Physically Tagged

- Excellent performance

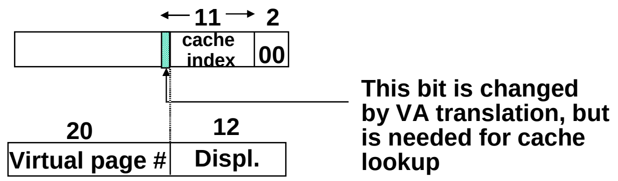
- Requires cache index to remain invariant across translation. How?

Virtually Indexed, Physically Tagged Example

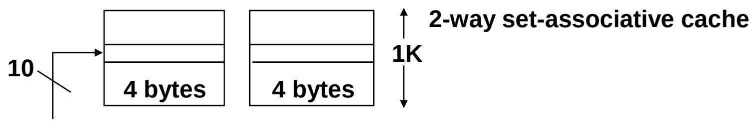


Issues With Overlapped TLB Access

- Limits cache parameters: small caches, large page sizes, or high n-way set-associative caches
- Example: Suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K


11
2
cache index
00
This bit is changed by VA translation, but is needed for cache lookup
20
12
Virtual page #
Displ.

Solutions:
Go to 8K byte page sizes;
Go to 2-way set-associative cache; or
SW guarantee VA[13]=PA[13]



Some Page Table Math

# of page table entries on 64-bit machine with 4K pages:

$2^{64} / 2^{12}$ = (only) 2^{52} entries

Size of page table:

2^{52} * 8 bytes per table entry = 2^{55} bytes
(only 32 petabytes)

kilo- 2^{10}, mega- 2^{20}, giga- 2^{30}, tera- 2^{40},
peta- 2^{50},
exa- 2^{60}, zetta- 2^{70}, yotta- 2^{80}

Some Page Table Math

Size of page table:

2^{52} * 8 bytes per table entry = 2^{55} bytes
(only 32 petabytes)

Oh, by the way, that's per process…

Solutions
1. Limit Page Table Size

- Keep a limit
- Check limit before going to page

If more entries needed (process needs more memory):
1. Up the limit
2. Add the entries

Good way to do this:
- Double page table size at each step:
- Limit is: 0…01…1 (number 0 à 2^{n-1})

Also, can grow bi-directionally (stack/heap)

Solutions
2. Inverted Page Table

!! These things are UGLY !!

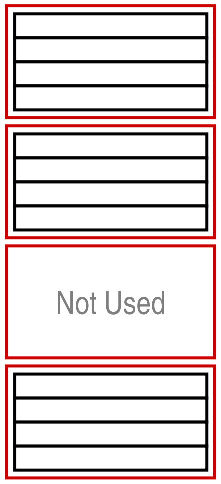
Each Physical Frame has an entry.

Inverted page table size:
Physical memory size = 8 Gigabytes = 2^{33} bytes
Page frame size = 4K = 2^{12} bytes
2^{33} / 2^{12} = 2^{21} entries
2^{21} entries * 8 bytes per entry (incl. PID) = 2^{24} bytes
16MB, not too bad (not per process!)

Solutions
3. Multilevel Page Tables

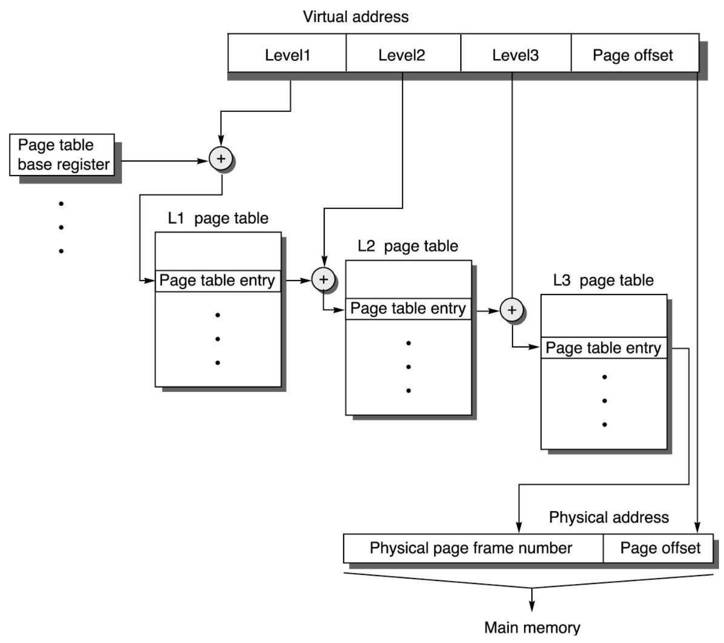
Key Idea: Take advantage of sparse use of virtual memory
Create a hierarchy of pages:

Create a red page table to describe very large pages (coarse cut of virtual address space)



Create a black page table for each red page table entry used (finer cut of superpage)

Solution 3: Multi-Level Page Tables Example



Solutions
4. Page The Page Table

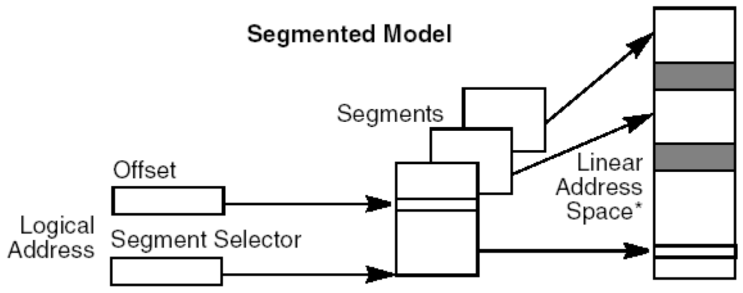
- Compatible with other methods
- Tricky to get right
- Need to have page portion that refers to rest of page table always in memory

Segmentation
Real Stuff (x86 IA32)

- Segments: Variable-sized pages
- Virtual address are segment number + offset
- Generally 2 quantities
  - Segment register
  - Offset is address
- Bounds checking
- Nice in some ways:
  - Program fits in one segment - set ReadOnly/Executable
  - Data in another - set ReadWrite/NonExecutable

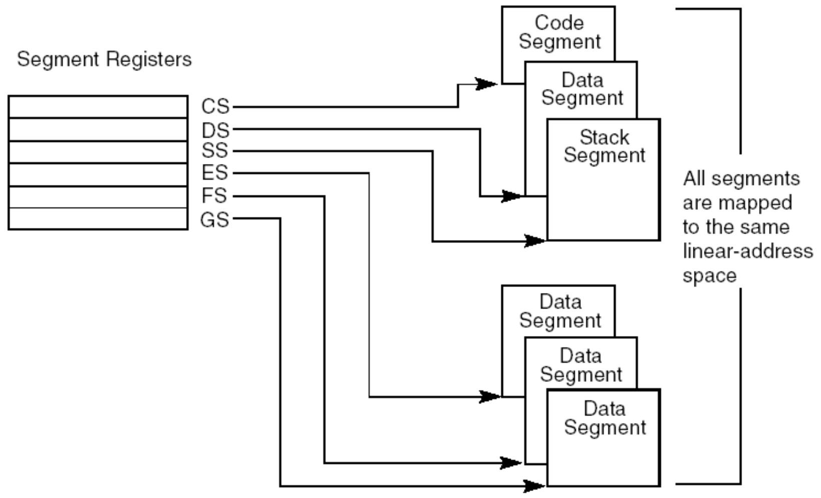
x86: Segmentation

(From: IA-32 Intel® Architecture Software Developers Manual)



x86: Segment Registers

(From: IA-32 Intel® Architecture Software Developers Manual)



Pages and Segments Can Co-exist!

Relating to the MIPS Pipeline

MIPS R3000 Pipeline

Inst Fetch	Dcd/ Reg	ALU / E.A.	Memory	Write Reg
TLB, I-Cache	RF	Operation	▨	WB
		E.A., TLB	D-Cache	

Summary

- Real/Virtual Tag/Index Cache
- Multi Level Page Tables
- Segments
- Pipeline Interaction
- Read book for more real stuff