Topic 18 (updated): Virtual Memory

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August

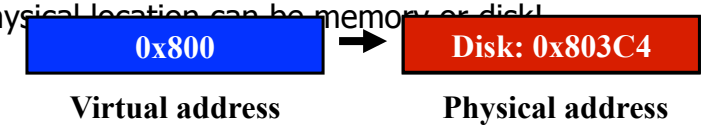
Virtual Memory

Any time you see virtual, think "using a level of indirection"

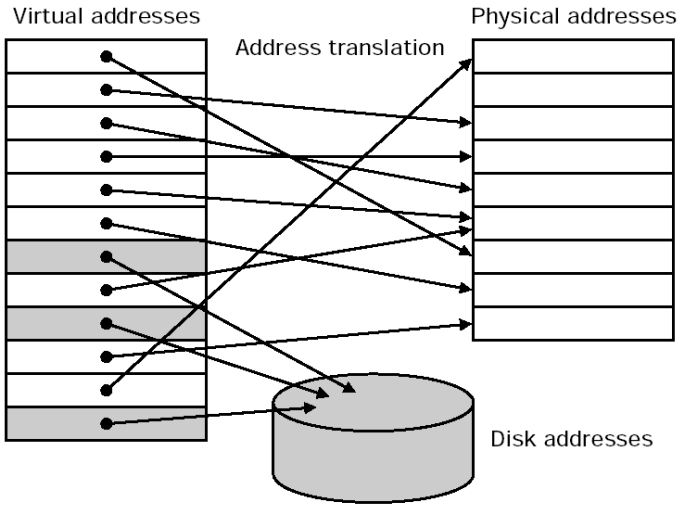
Virtual memory: level of indirection to physical memory

- Program uses virtual memory addresses
- Virtual address is converted to a physical address
- Physical address indicates physical location of data

- Physical location can be memory or disk!



Virtual Memory



Virtual Memory: Take 1

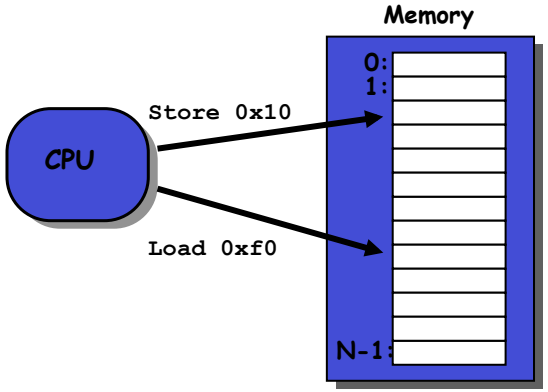
Main memory may not be large enough for a task

- Programmers turn to overlays and disk
- Many programs would have to do this
- Programmers should have to worry about main memory size across machines

Use virtual memory to make memory look bigger for all programs

A System with Only Physical Memory

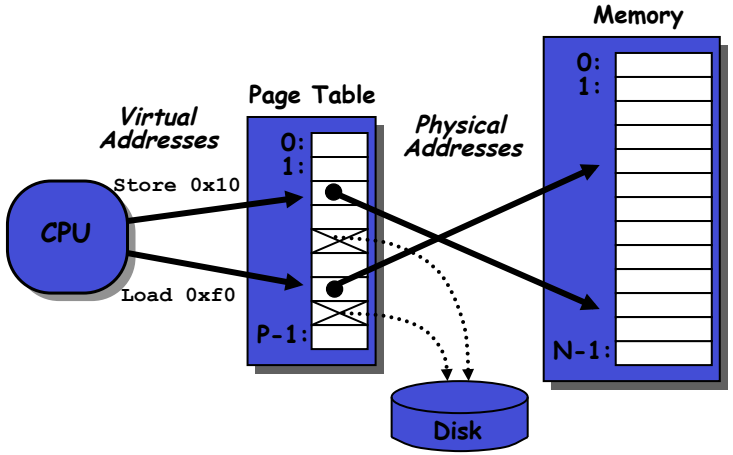
Examples: Most Cray machines, early PCs, nearly all current embedded systems, etc.



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

A System with Virtual Memory

Examples: modern workstations, servers, PCs, etc.

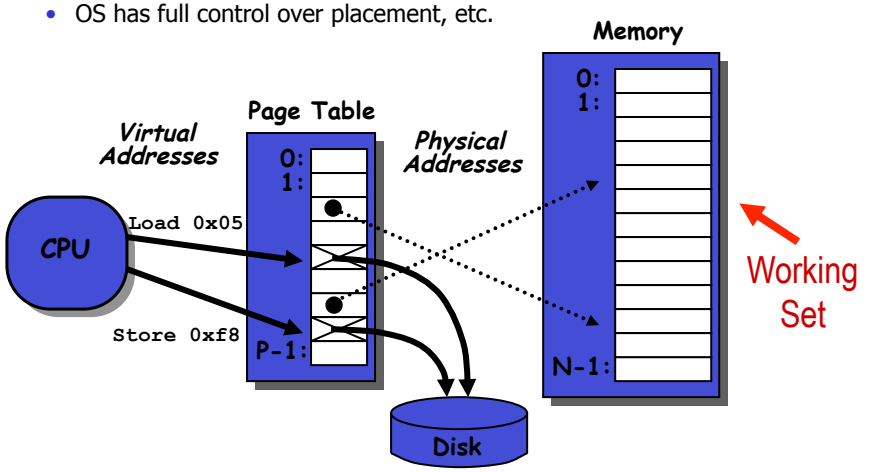


Address Translation: the hardware converts *virtual addresses* into *physical addresses* via an OS-managed lookup table (*page table*)

Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?

1. Page table indicates that the virtual address is not in memory
2. OS trap handler is invoked, moving data from disk into memory
   - Current process suspends, others can resume
   - OS has full control over placement, etc.



Virtual Memory: Take 2

Concurrently executing programs will interfere in memory

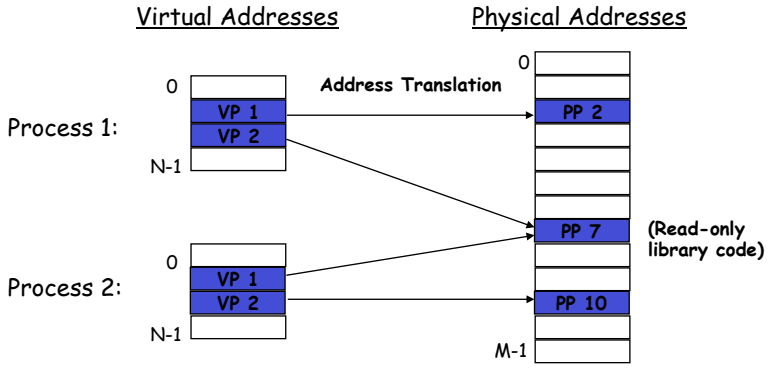
- At some point, programs assume addresses
- These addresses may conflict if we don't manage them.
- Which programs will execute concurrently?
  - Don't know
  - Manage dynamically

- Programs can maliciously interfere with each other!
- They need protection from one another

Use virtual memory to avoid/manage conflict between programs

Separate Virtual Address Spaces

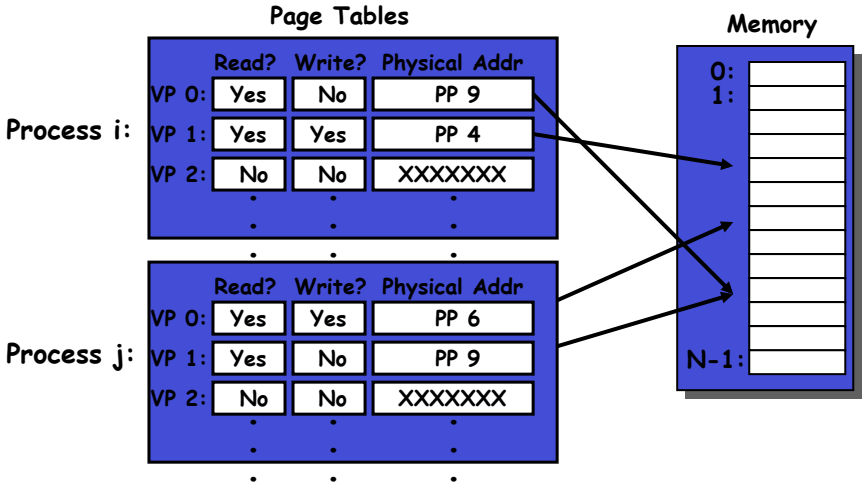
- Each process has its own virtual address space
- OS controls how virtual is assigned to physical memory



Motivation: Process Protection

Page table entry contains access rights information

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



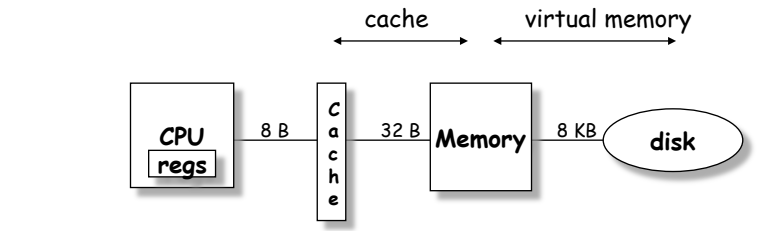
Virtual Memory: Take 3

Programs and data exist on disk

- Registers, caches, and memory just make using the data on disk faster
- Locality at different granularities

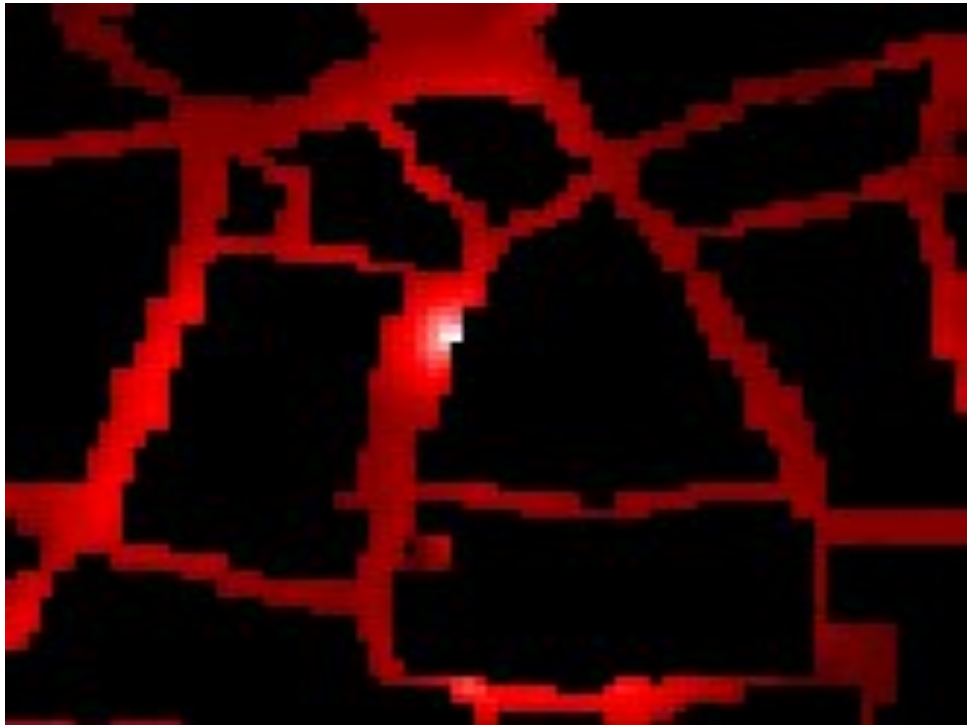
Use virtual memory to improve performance, hide physical location from program

Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	200 B	32 KB / 4MB	128 MB	20 GB
speed:	3 ns	6 ns	60 ns	8 ms
$/Mbyte:		$100/MB	$1.50/MB	$0.05/MB
block size:	8 B	32 B	8 KB	

larger, slower, cheaper



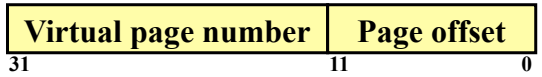
Virtual Memory

Just like caches, but origins are different

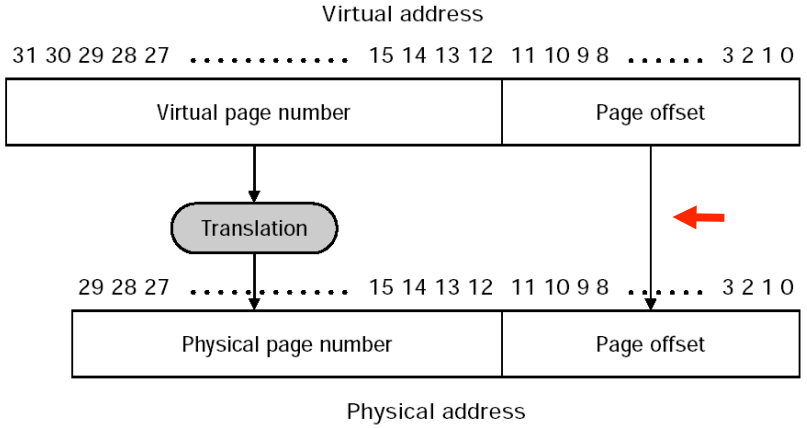
- Cache - performance goals
- Virtual Memory - programmability/multiprogram goals

Blocks are called Pages

- A virtual address consists of
  - A virtual page number
  - A page offset field (low order bits of the address)

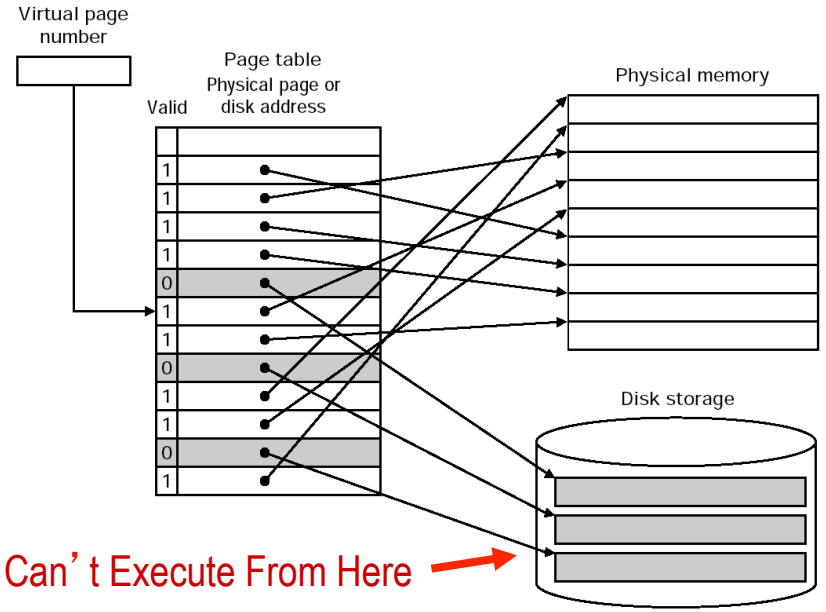


Page Tables



Each process gets its own page table, why?

Page Tables



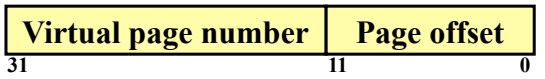
Page Faults

Blocks are called Pages

Page Tables translate virtual to physical page numbers

Misses are call Page faults (handled as an exception)

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

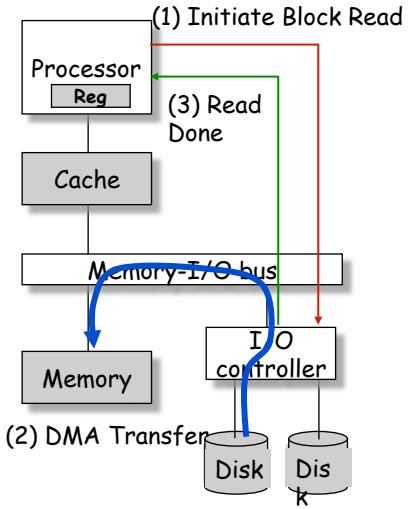


Servicing a Page Fault

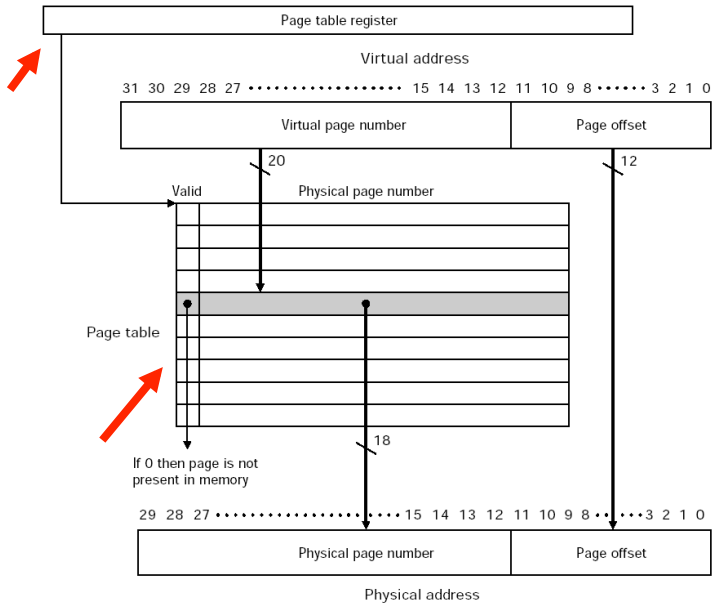
1. Make space in memory by writing physical page to disk
   - Page Frames
   - Replacement policy?
2. Load page
   - Loading pages could waste processor time, use DMA
   - DMA allows processor to do something else
3. OS updates the process's page table
   - Desired data is in memory for process to resume

Servicing a Page Fault

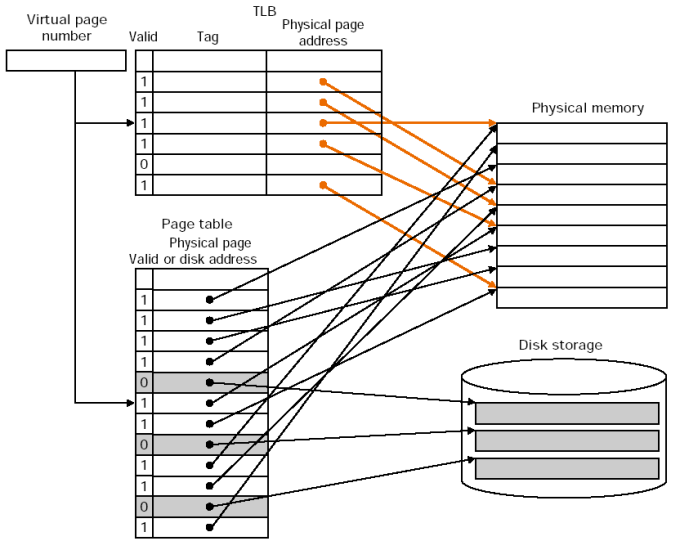
1. Processor Signals Controller
   "Read block of length P starting at disk address X and store starting at memory address Y"
2. DMA Read Occurs
3. I / O Controller Signals Completion
   - Interrupts processor
   - Can resume suspended process



Where Is the Page Table?



Making Translation Faster: The TLB
Translation Look-Aside Buffer



TLB Design Issues

Accessed frequently: speed is important

TLB Miss Involves (not to be confused with page fault):

1. Stall pipeline
2. Invoke Operating System (what about OS pages?)
3. Read Page Table
4. Write entry in TLB (evicting old entry?)
5. Return to user code
6. Restart at reference

MIPS.
Another HW Option?

TLB Design Issues

Clearly, we want to minimize TLB misses:

- Can be fully-associative, set-associative, or direct mapped
- Often fully associative, can be set associative

- Sized to maximize hits, but make timing
- Usually not more than 128, 256 entries (associtivity)

Loading Your Program: A Neat Trick

1. Ask operating system to create a new process
2. Construct a page table for this process
3. Mark all page table entries as invalid with a pointer to the disk image of the program
4. Run the program and get an immediate page fault on the first instruction.

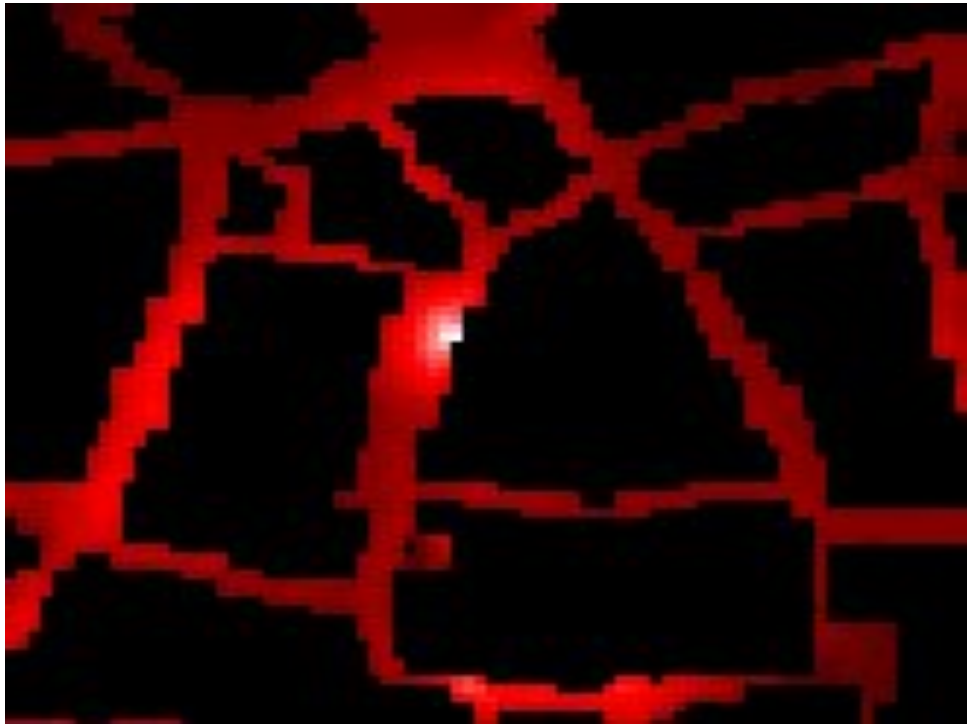
Process Interactions

Virtual Addresses are per Process
Context Switch: Save "state": regs, PC, page table (PTBR)

TLB?

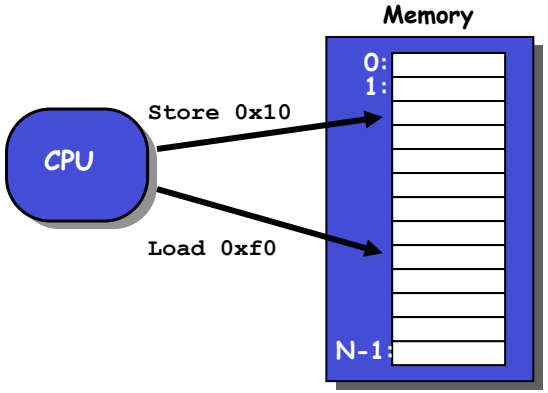
- Could Flush TLB
  - Every time perform context switch
  - Refill for new process by series of TLB misses
  - ~100 clock cycles each

- Could Include Process ID Tag with TLB Entry
  - Identifies which address space being accessed
  - OK even when sharing physical pages



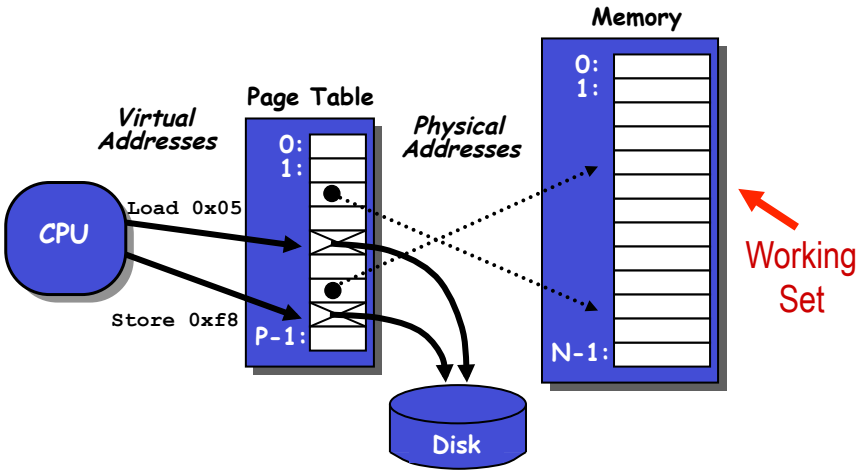
A System with Physical Memory

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

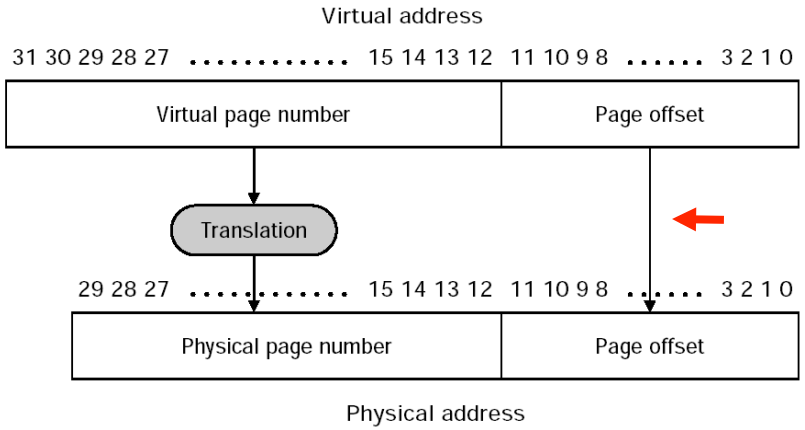


CPU's load or store addresses used directly to access memory.

A System with Virtual Memory



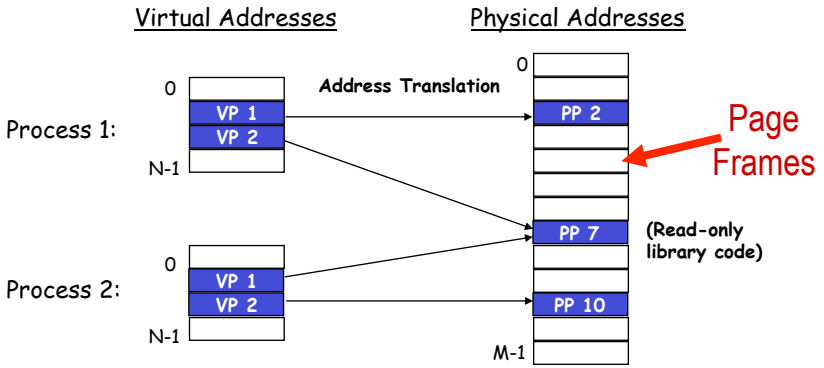
Page Tables



Each process gets its own page table, why?

Separate Virtual Address Spaces

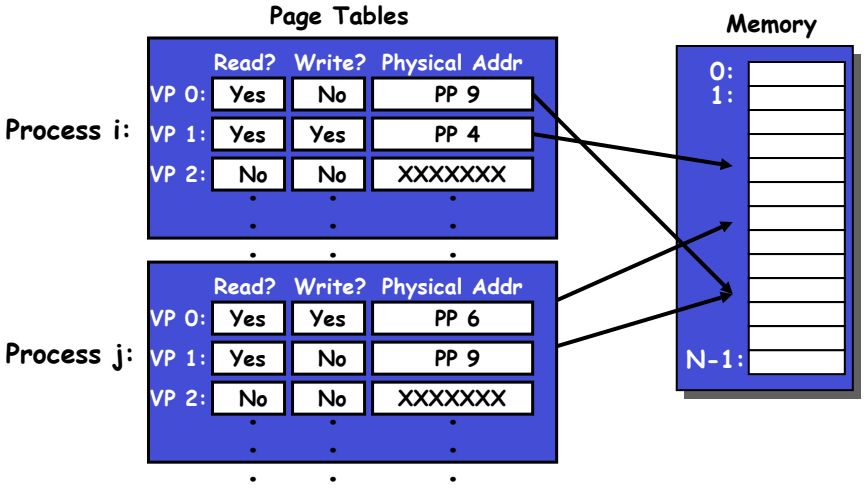
- Each process has its own virtual address space
- OS controls how virtual is assigned to physical memory



Process Protection

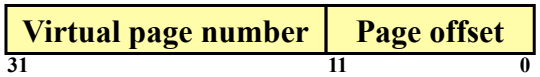
Page table entry contains access rights information

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



Virtual Memory Lingo

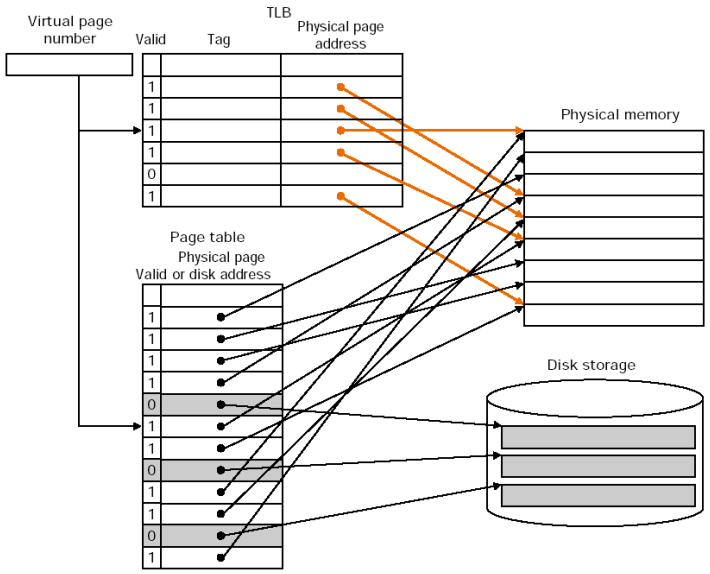
Blocks are called Pages



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

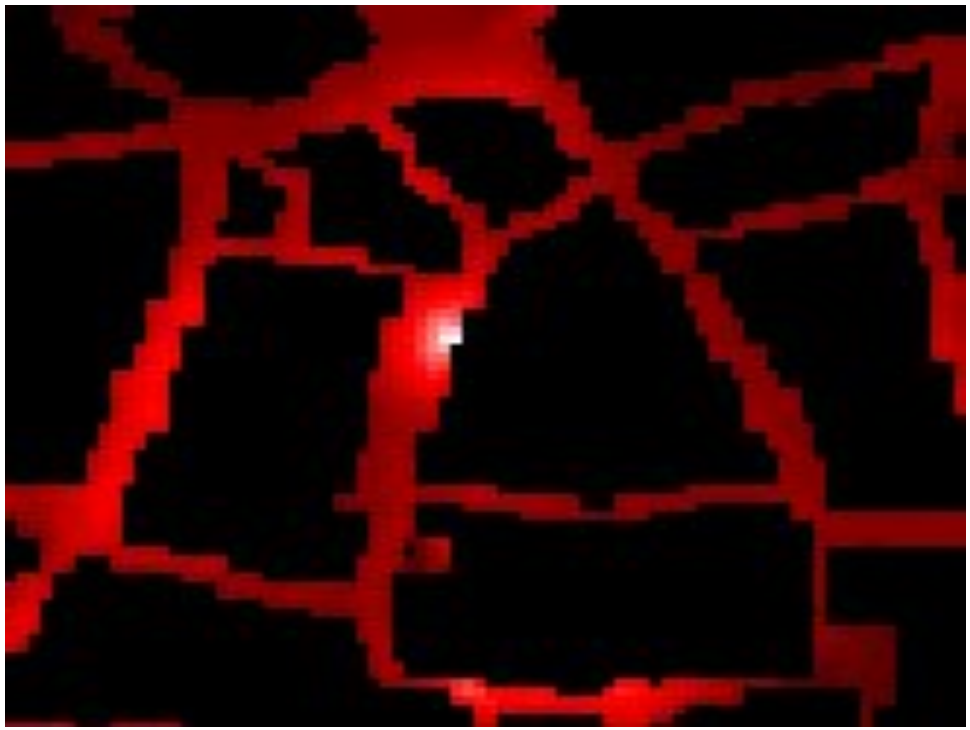
Virtual memory provides

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

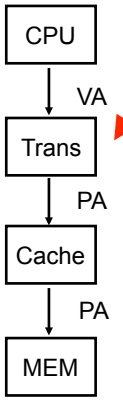
- Virtual Memory requires twice as many memory accesses, so cache page table entries in the TLB.

- 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

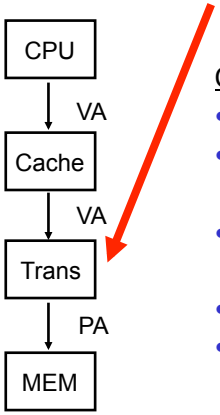
Cache is:

- 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

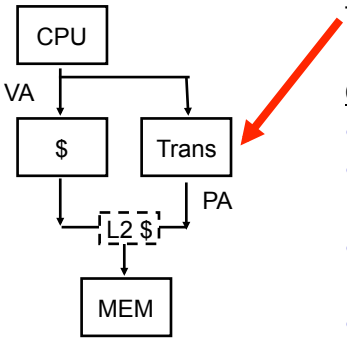


Translate after accessing cache

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

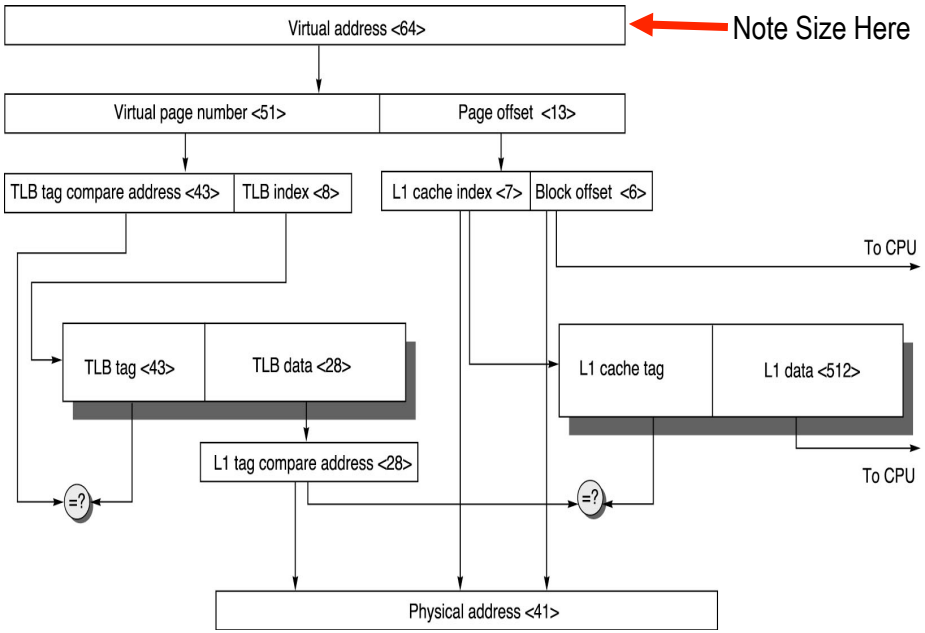


Translate during cache access

Cache is:

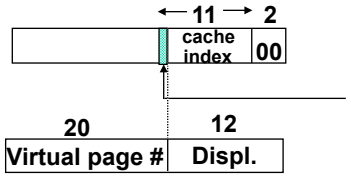
- Physically/Virtually Indexed
- Physically Tagged
- Excellent performance
- Requires cache index to remain invariant across translation. How?

Virtually Indexed, Physically Tagged Example



Issues With Overlapped TLB Access

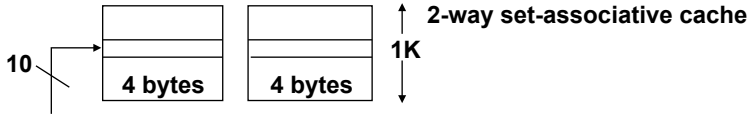
- Limits cache parameters: small caches, large page sizes, or high n-way set-associative caches
- Example: Suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K

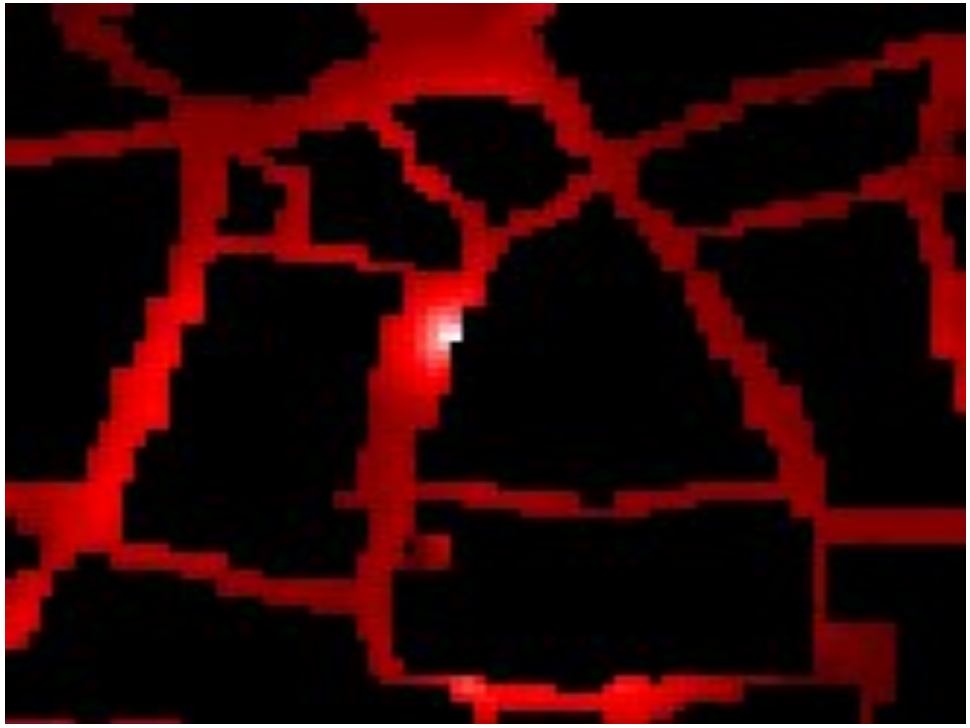


This bit is changed by VA translation, but is needed for cache lookup

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 → 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

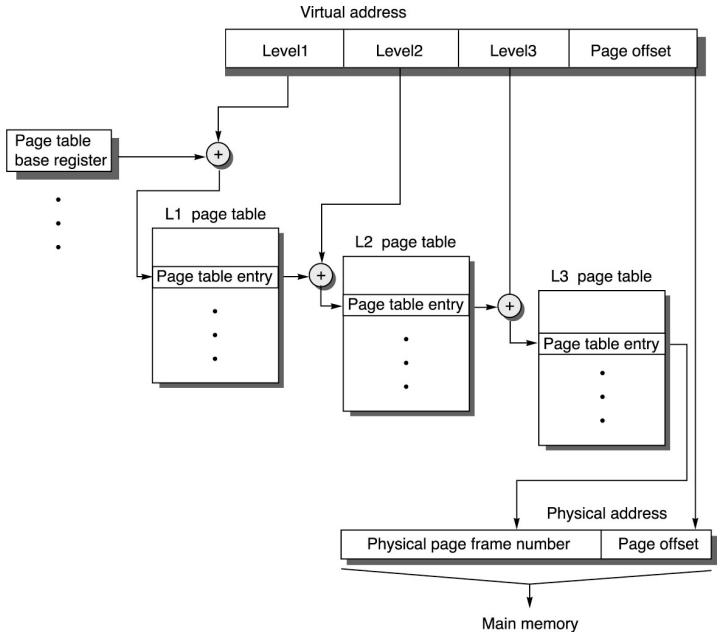
Key Idea: Take advantage of sparse use of virtual memory

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



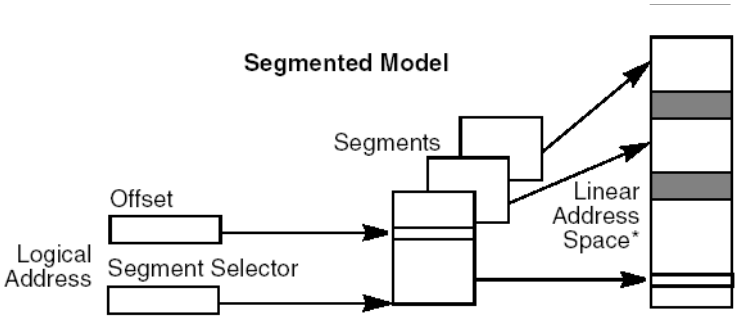
Solutions
4. Page The Page Table

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

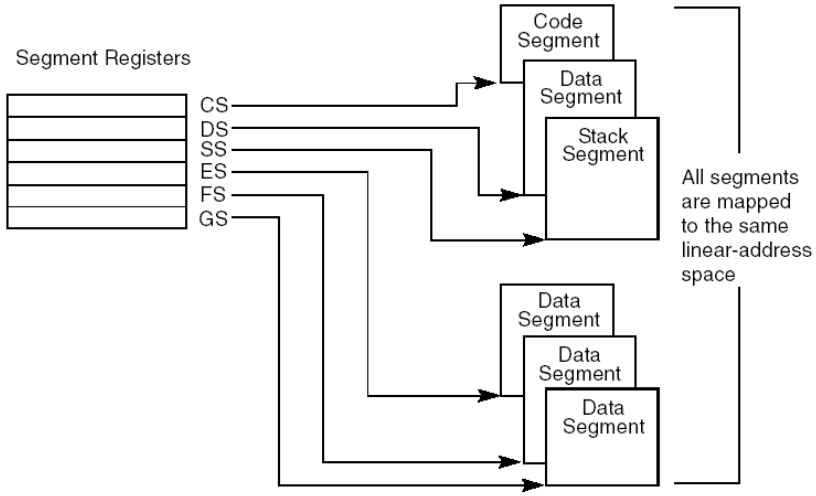
Segmentation
Real Stuff (x86 IA32)

- 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