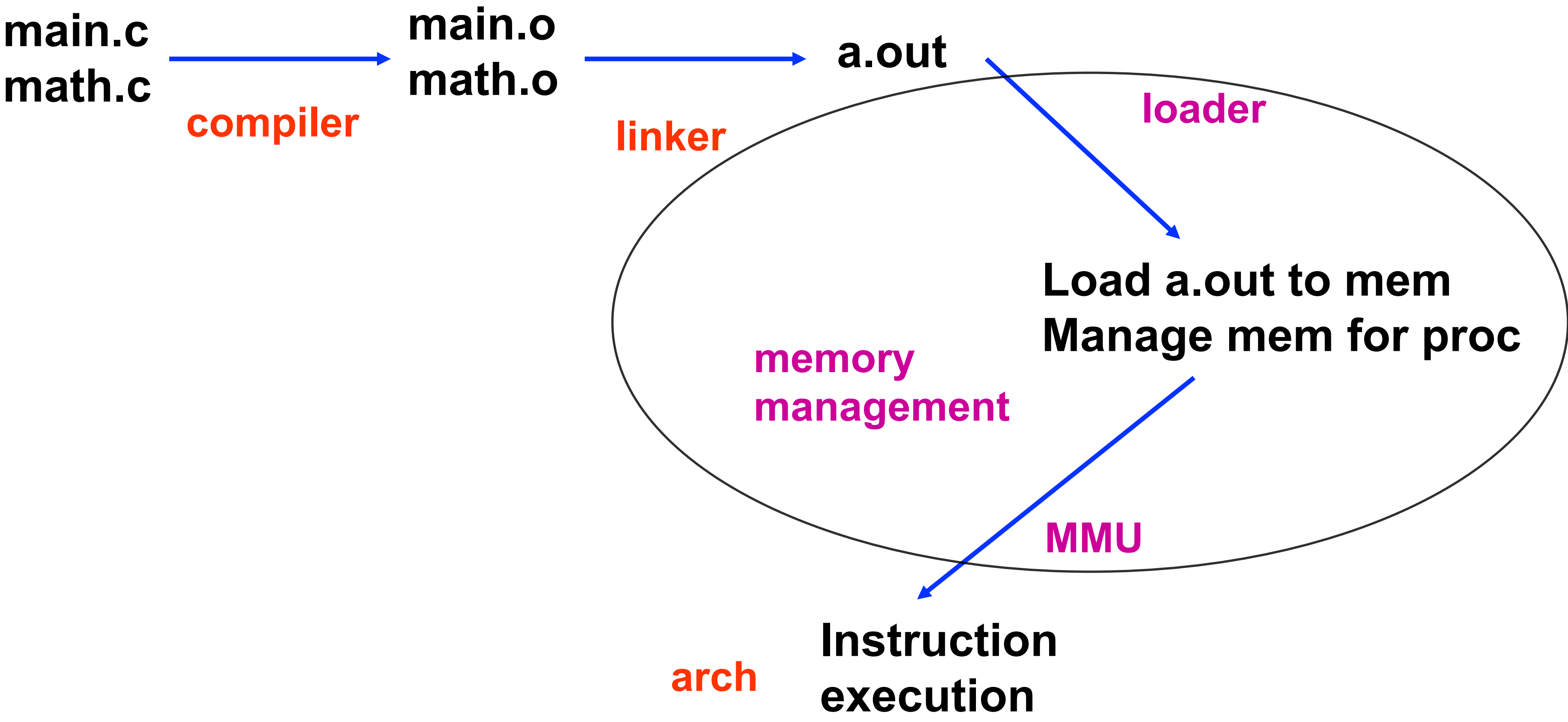
Paging, Inverted Page Table, TLB

ECE469, Feb 28

Yiying Zhang

The big picture



main.c
math.c

compiler

main.o
math.o

linker

a.out

loader

Load a.out to mem
Manage mem for proc

memory
management

MMU

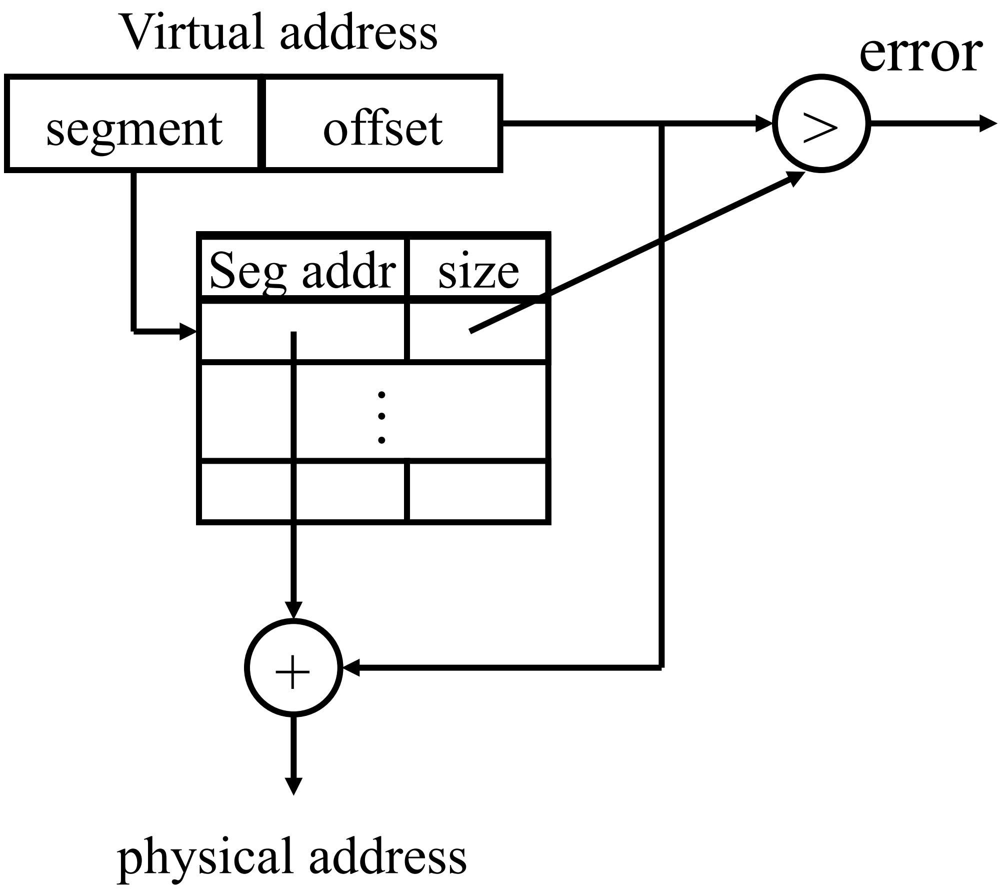
arch

Instruction
execution

Today’s topics

- Basic paging [ctnd]
- Inverted page table
- TLB

[lec13] Segmentation



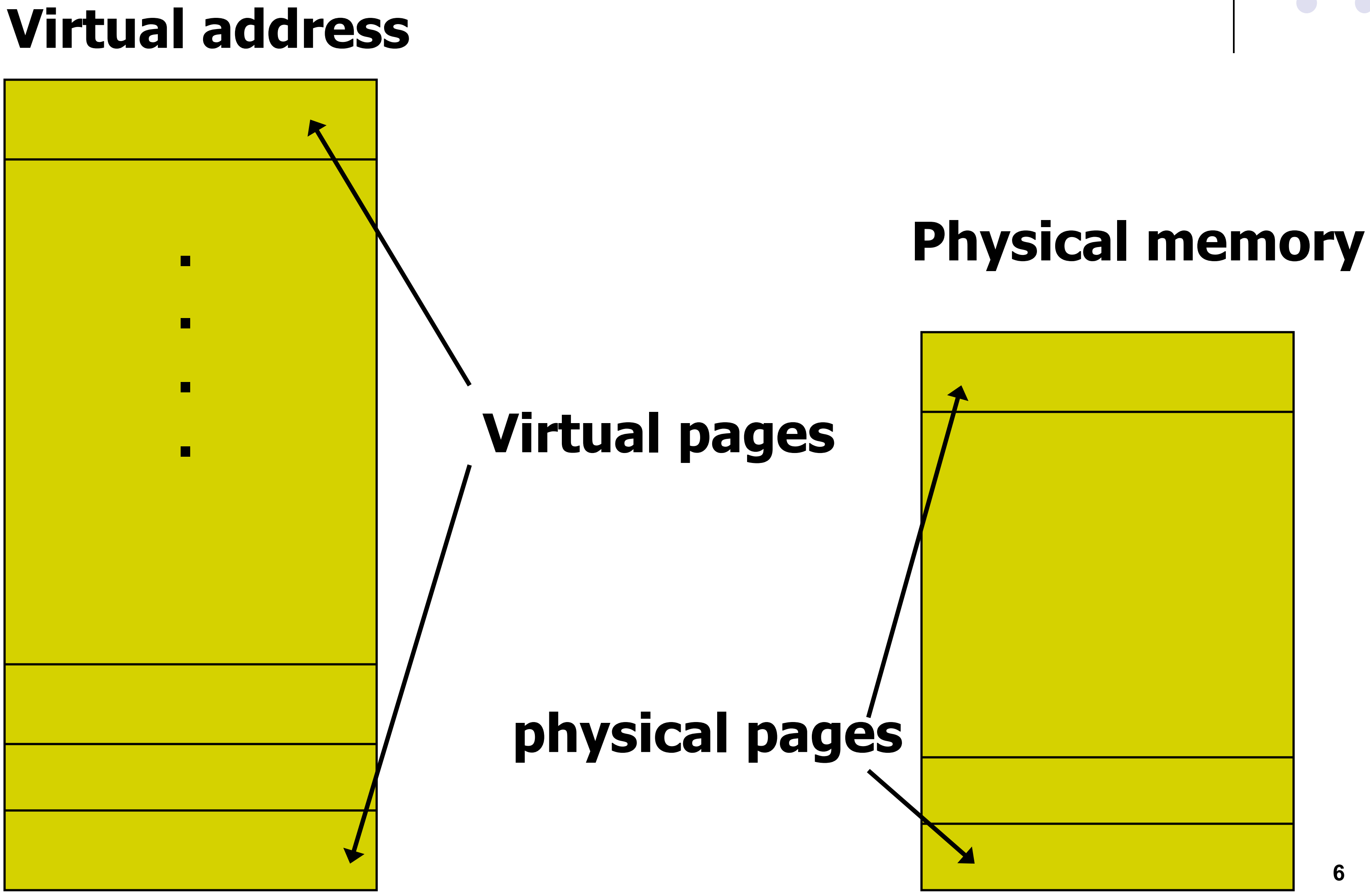
- A table of (seg, size)
- Protection: each entry has
  - (nil, read, write, exec)
- On a context switch: save/restore the table or a pointer to the table to/from PCB

[review] What fundamentally causes external fragmentation?

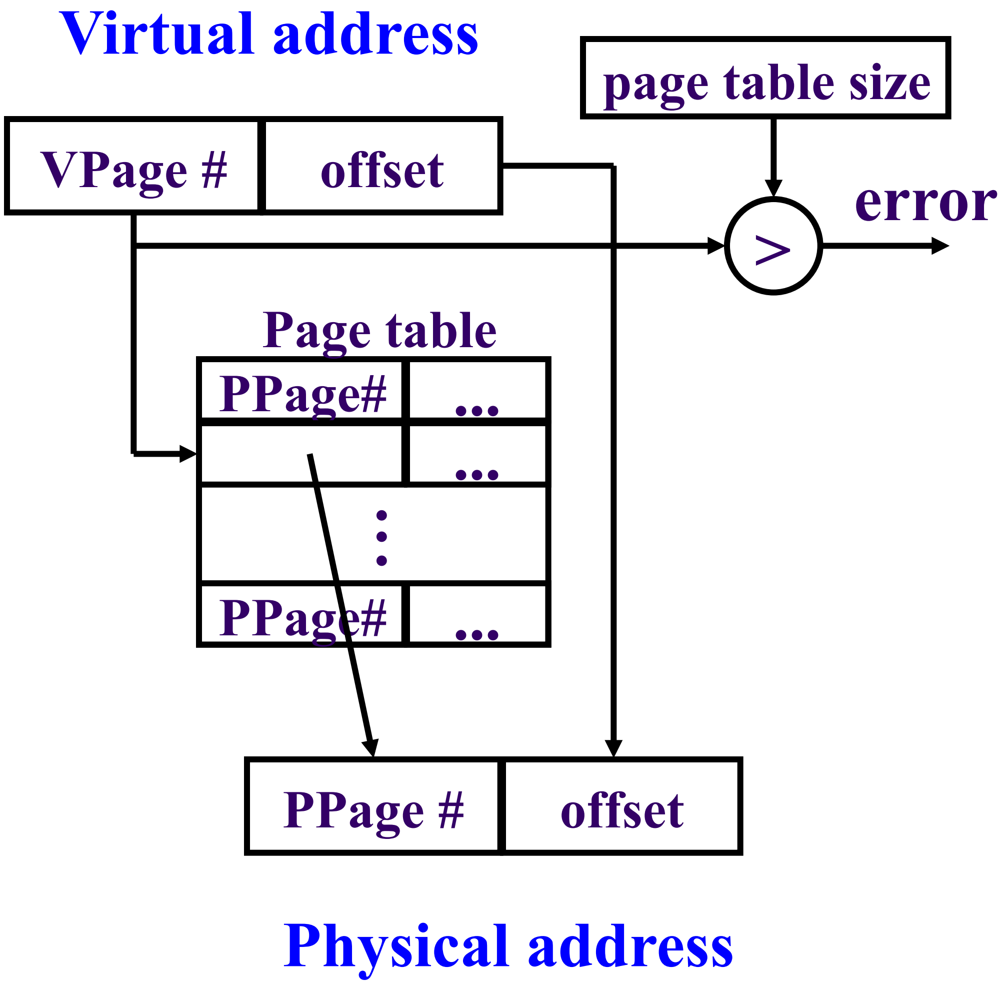
- Segments of many different sizes
- Each has to be allocated contiguously

- “Million-dollar” question:
  Physical memory is precious.
  Can we limit the waste to a single hole of X bytes?

[review] Virtual pages / physical pages



[review] Paging



- Context switch
  - similar to the segmentation scheme
- Pros:
  - easy allocation, keep a free list
  - easy to swap
  - easy to share

[Review] How many PTEs do we need?

(assume page size is 4096 bytes)

- Worst case for 32-bit address machine
  - # of processes × 2^{20}
- What about 64-bit address machine?
  - # of processes × 2^{52}

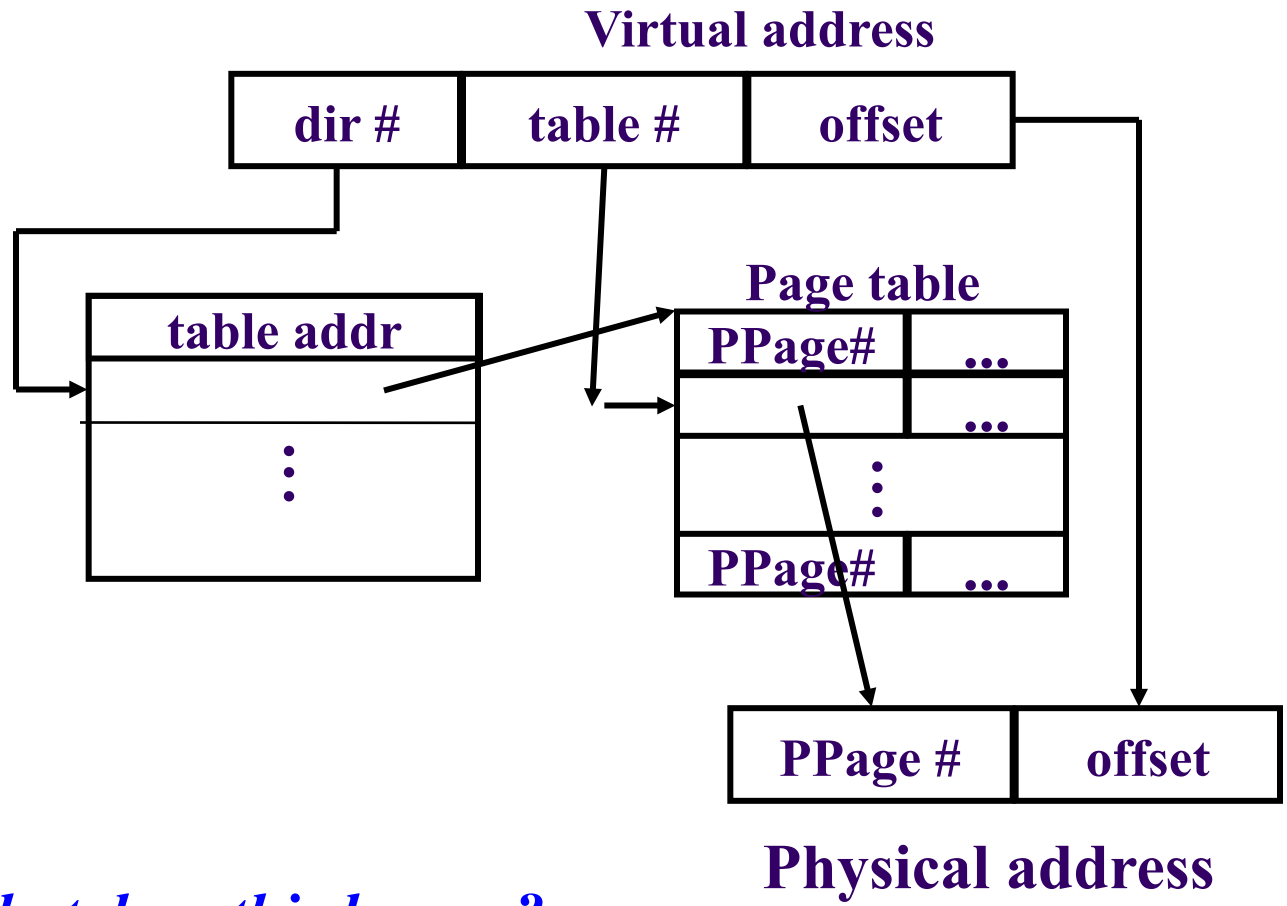
Deep thinking

- In segmentation, why does each segment need to be contiguous in physical memory?

- In segmentation, what to do with heap/stack?
  - What happens when they grow/shrink?

- In paging, do pages belonging to the same "segment" (e.g. heap) need to be contiguous in physical memory?
  - What made this possible?
  - What to do with heap/stack growing/shrinking now?

Two-level page tables



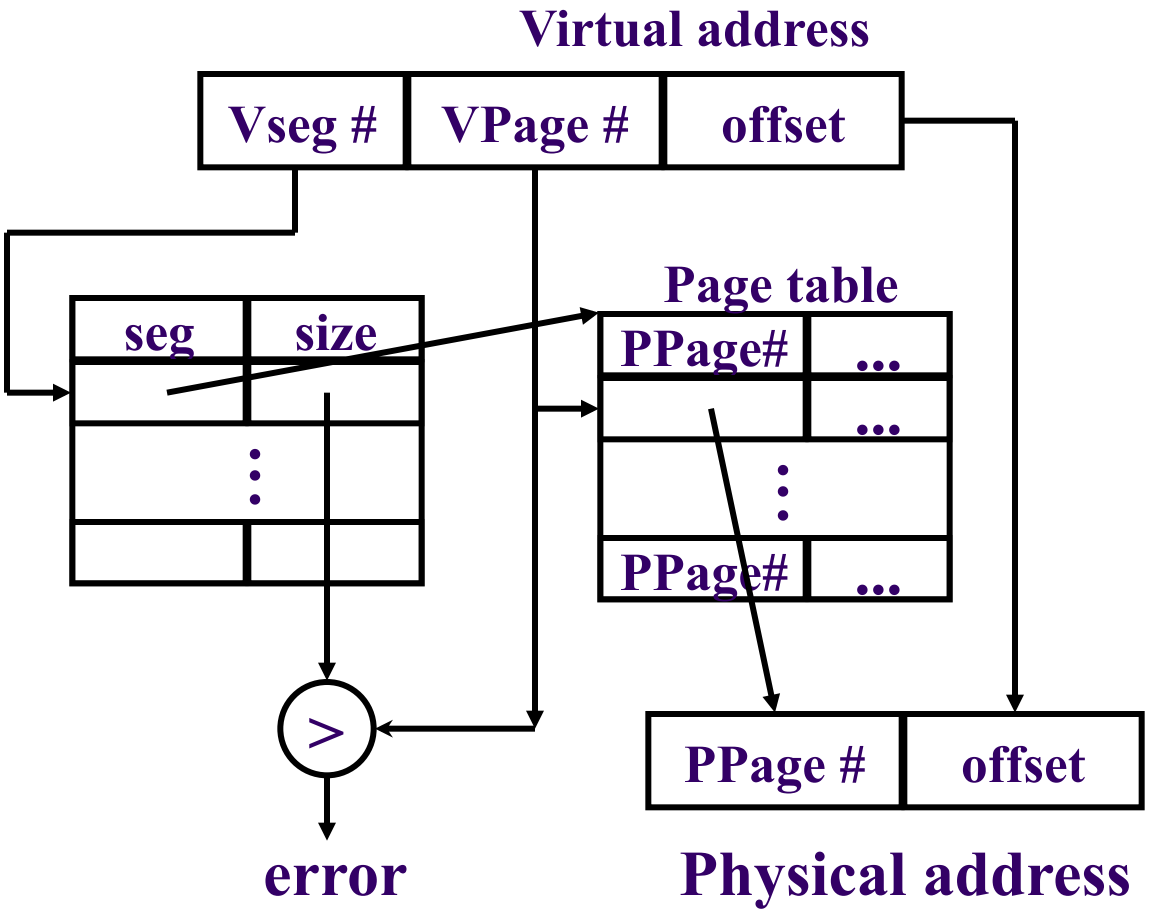
What does this buy us?

Multi-level page tables

- 3 Advantages?
  - L2 page tables do not have to consecutive
  - They do not have to be allocated before use!
  - They can be swapped out to disk!

The power of an extra level of indirection!

Segmentation with paging



Ex: IBM System 370 (24-bit, 4-bit segment #, 8-bit page #)

Segmentation + paging

- Use two levels of mapping to make tables manageable:
  - Each segment contains one or more pages
  - Segments correspond to logical units: code, data, stack
  - Segments vary in size and are often large
  - Pages are for easy of management by OS: fixed size -> easy to allocate/free
  - Going from P to P+S is like going from single segment to multiple segments, except at a high level
    - One page table -> many page tables with bases/bounds

[lec1] Separating Policy from Mechanism

Mechanism – tool to achieve some effect

Policy – decisions on how to use tool

examples:

- All users treated equally
- All program instances treated equally
- Preferred users treated better

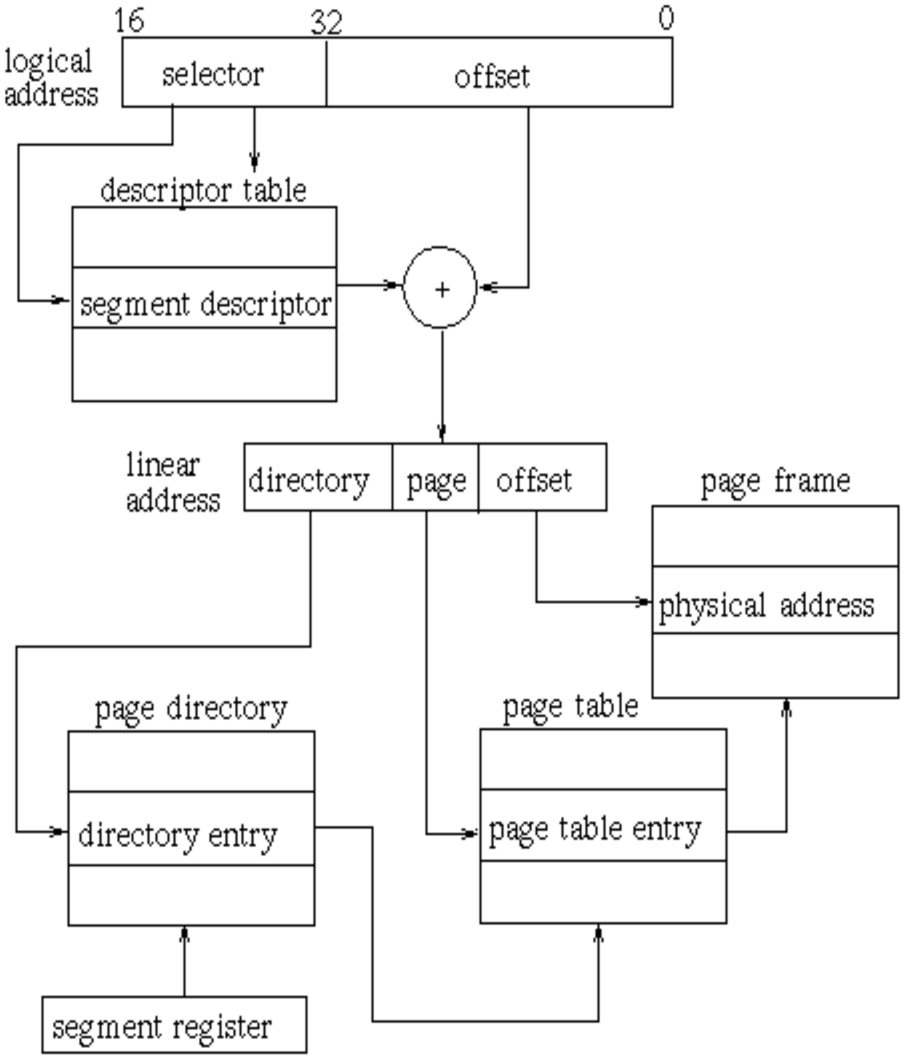
Separation leads to flexibility

Segmentation + paging vs. multi-level paging

- *Mechanisms* are similar
- Difference lies in *policy*
  - Segmentation + paging still maintains notion of segments
  - Multi-level paging deals the whole, uniform address space (like one-level paging)

The Intel Pentium (1993) (pro, II, III, 4) (Ch 8.7, fig 8.22, 8.23)

- Supports both pure segmentation and segmentation with 1-level paging (page size=4M) or 2-level paging (page size=4k)
- CPU generates logical addresses
  - (selector, offset), 16 bits and 32 bits
  - As many as 16K segments
  - Up to 4GB per segment



Linux on Pentium

- Linux uses 3-level paging
  - For both 32-bit and 64-bit architectures
- On Pentium, degenerates to 2-level paging
  - Middle-level directory has zero bits

Today’s topics

- Basic paging
- Inverted page table
- TLB

[review] How many PTEs do we need?

- Worst case for 32-bit address machine
  - # of processes × 2^{20} (if page size is 4096 bytes)
- What about 64-bit address machine?
  - # of processes × 2^{52}

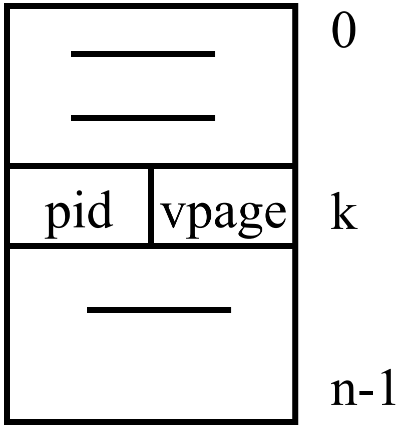
Hmm, but my PC only has 1GB, 256K PTEs should be enough?!

Inverted Page Table

- Motivation
  - Example: 2 processes, page table has 1M entries, 10 phy pages
- Is there a way to save page table space?

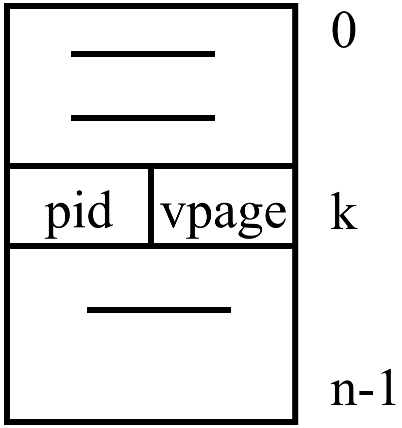
Ideally,

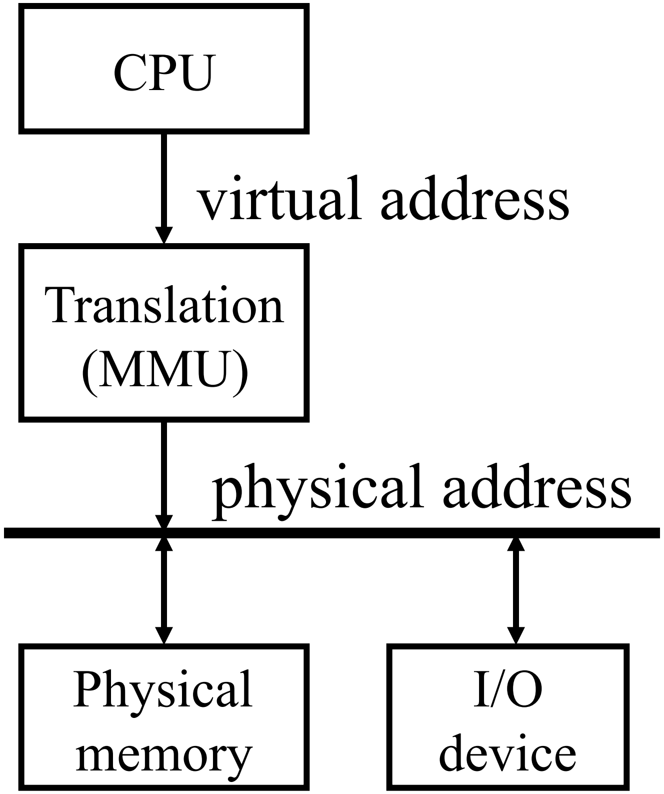
- One PTE for each physical page frame, disregarding how many processes
  - Assuming rest virtual addressed not allocated/used
  - i.e., linear inverted page table (an array of phy pages mapped to virtual addresses)



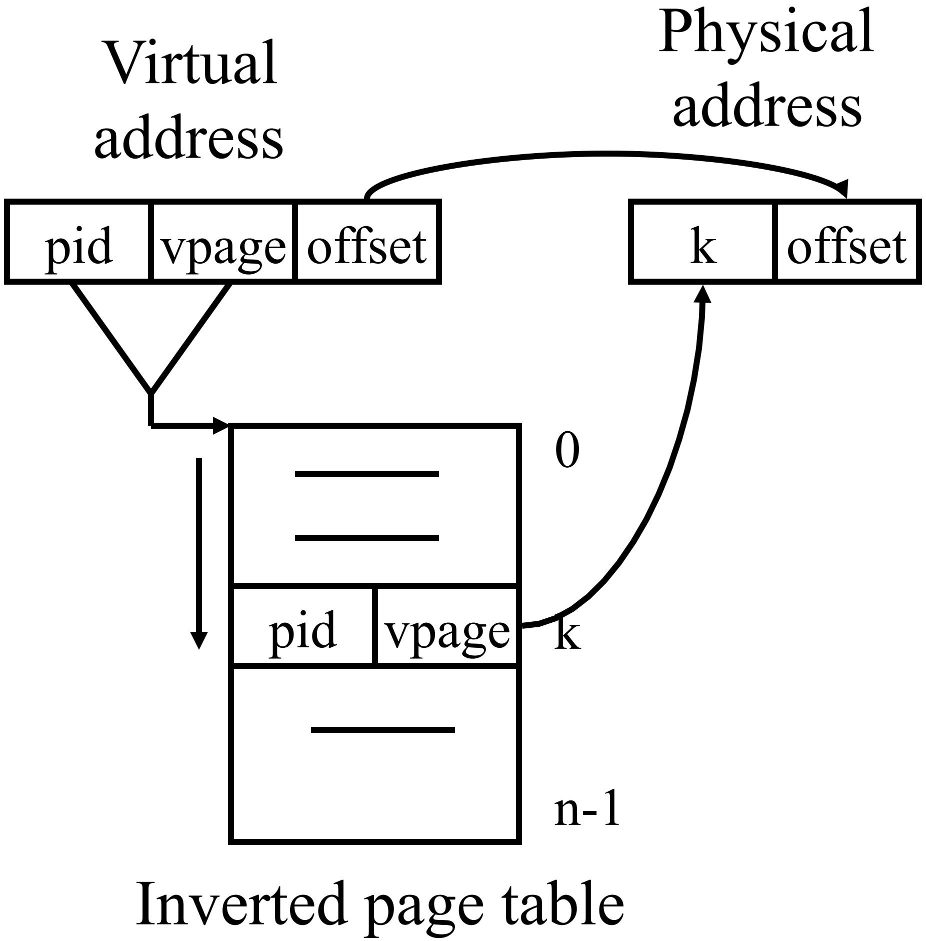
But,

- How do we do lookups with linear inverted page table?
  - Has to go through the entire array and compare!





(Hashed) Inverted page tables



- Main idea
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
  - Overhead of managing hash chains, etc
- Ex: 64-bit UltraSPARC, PowerPC

Deep thinking

- How can two processes share memory under inverted page table?
  - Since the inverted page table provides only one forward mapping, it is very difficult to share memory among processes. For this reason most modern OSs use multi-level page tables.

break

- There is four digit number in aabb form and it is a perfect square. Find out the number.

Today’s topics

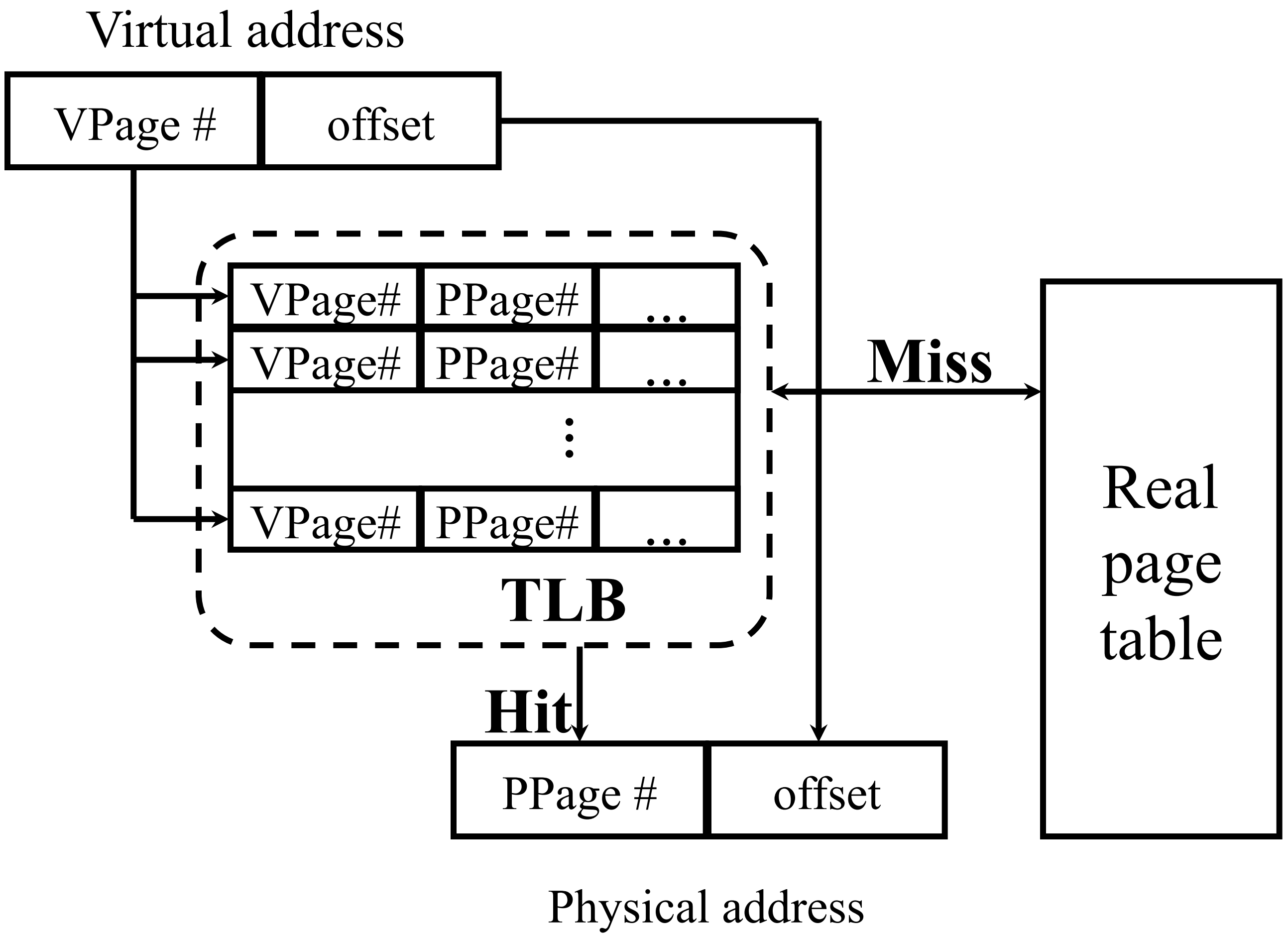
- Basic paging
- Inverted page table
- TLB

Performance problem with paging

- How many extra memory references to access page tables?
  - One-level page table?
  - Two-level page table?

- Solution: *reference locality!*
  - In a short period of time, a process is likely accessing only a few pages
  - Instead of storing only page table starting address in hardware (MMU), store part of the page table that is "hot"

Translation Look-aside Buffer (TLB)



TLB often fully set-associative → least conflict misses
Expensive → typically 64 – 1024 entries

Bits in a TLB Entry

VPage#	PPage#	...
VPage#	PPage#	...
	⋮	
VPage#	PPage#	...

TLB

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - PTE

- Optional (useful) bits
  - ASIDs -- Address-space identifiers (process tags)

Miss handling: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs fault handling
    - Restart the faulting instruction

- On a TLB miss
  - MMU parses page table and loads PTE into TLB
    - Needs to replace if TLB is full
    - PT layout is fixed
  - Same as hit …

Miss handling: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs page fault handling
    - Restart the faulting instruction

- On a TLB miss, HW raises exception, traps to the OS
  - OS parses page table and loads PTE into TLB
    - Needs to replace if TLB is full
    - PT layout is flexible
  - Same as in a hit…

Hardware vs. software controlled

- Hardware approach
  - Efficient – TLB misses handled by hardware
  - OS intervention is required only in case of page fault
  - Page structure prescribed by MMU hardware -- rigid

- Software approach
  - Less efficient -- TLB misses are handled by software
  - MMU hardware very simple, permitting larger, faster TLB
  - OS designer has complete flexibility in choice of MM data structure
    - e.g. 2-level page table, inverted page table

Deep thinking

- Without TLB, how MMU finds PTE is fixed
- With TLB, it can be flexible, e.g. software-controlled is possible
- What enables this?
- TLB is an extra level of indirection!

More TLB Issues

- Which TLB entry should be replaced?
  - Random
  - LRU

- What happens when changing a page table entry (e.g. because of swapping, change read/write permission)?
  - Change the entry in memory
  - flush (eg. invalidate) the TLB entry
    - INGLPG on x86

What happens to TLB in a process context switch?

- During a process context switch, cached translations can not be used by the next process
  - Invalidate all entries during a context switch
    - Lots of TLB misses afterwards
  - Tag each entry with an ASID
    - Add a HW register that contains the process id of the current executing process
    - TLB hits if an entry’s process id matches that reg

Cache vs. TLB

- Similarities:
  - Both cache a part of the physical memory
- Differences:
  - Associatively
    - TLB is usually fully associative
    - Cache can be direct mapped
  - Consistency
    - TLB does not deal with consistency with memory
    - TLB can be controlled by software

More on consistency Issues

- Snoopy cache protocols can maintain consistency with DRAM, even in the presence of DMA

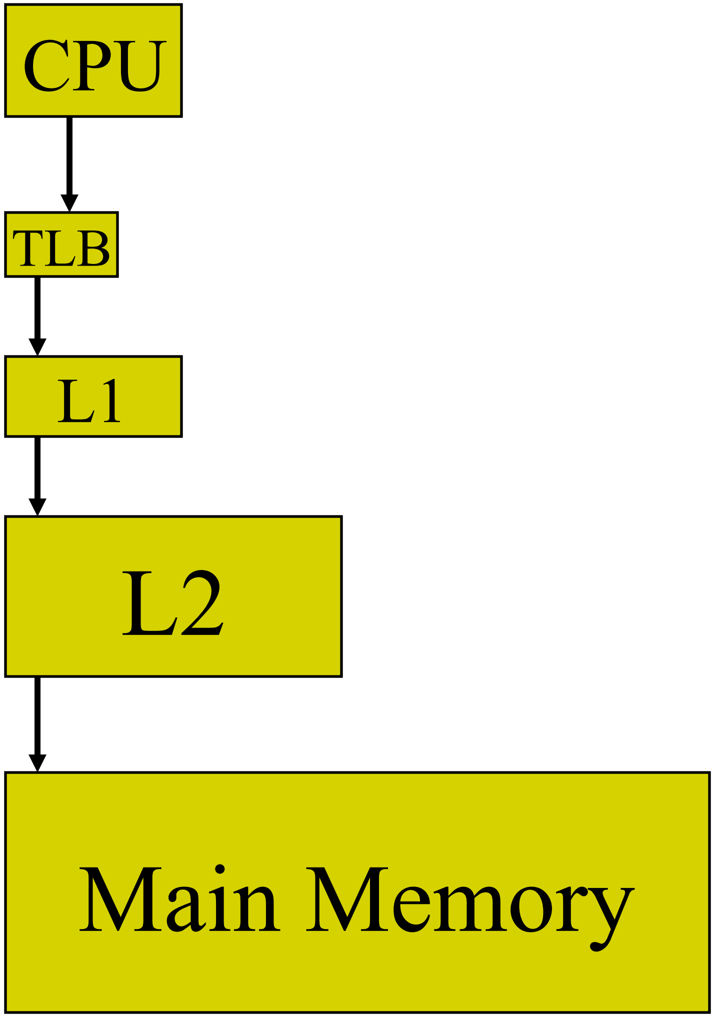
- No hardware maintains consistency between DRAM and TLBs:
  - OS needs to flush related TLBs whenever changing a page table entry in memory

- On multiprocessors, when you modify a page table entry, you need to do "*TLB shoot-down*" to flush all related TLB entries on all processors

Memory Hierarchy Revisited

What does this imply about L1 addresses?

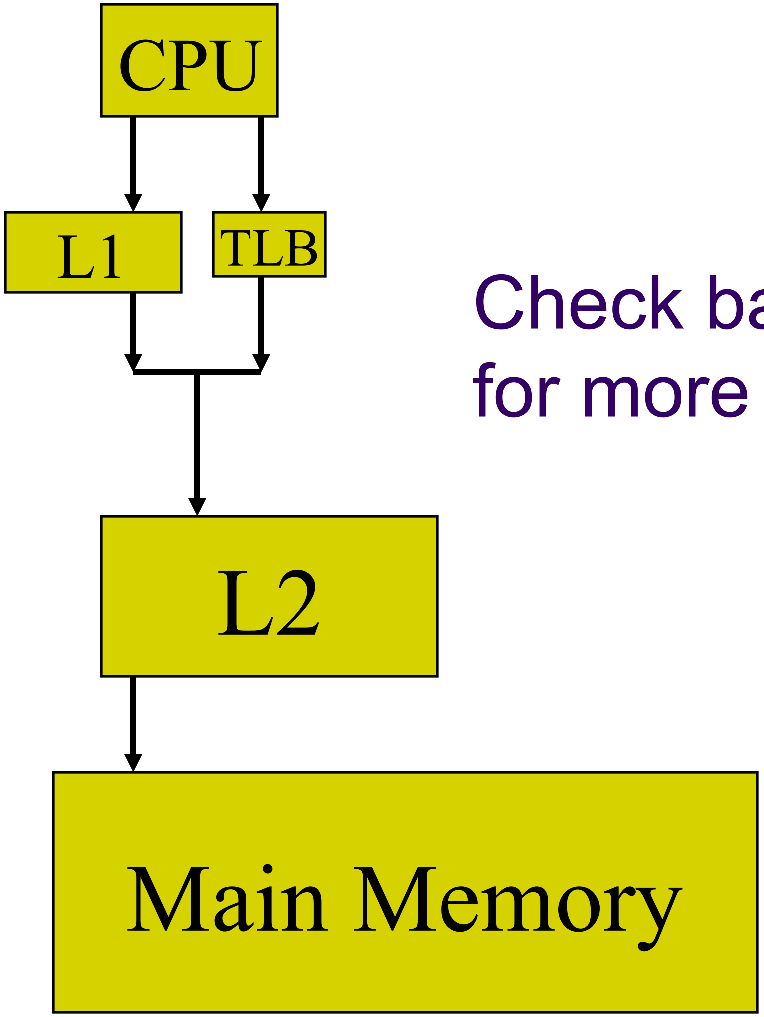
Where do we hope requests get satisfied?



Memory Hierarchy Re-Revisited

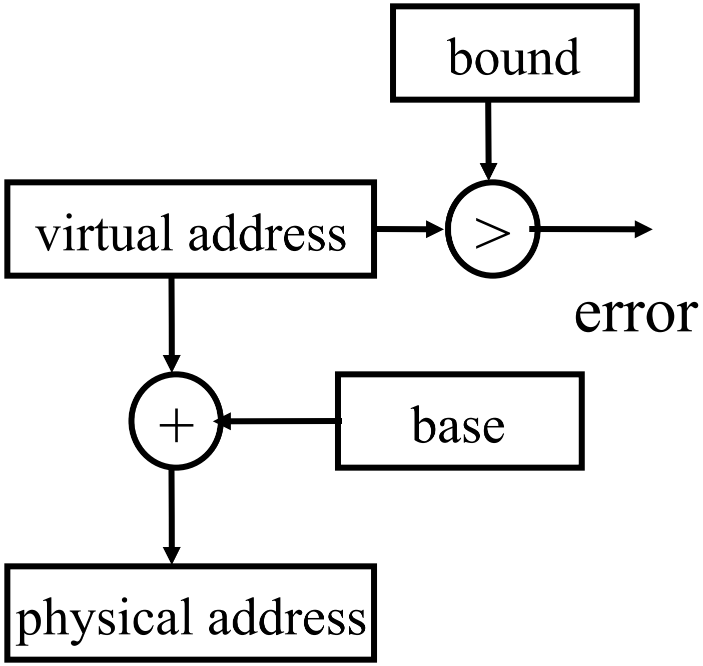
What does this imply about L1 addresses?

Any speed benefits?
Any drawbacks?



Check backup slides for more details!

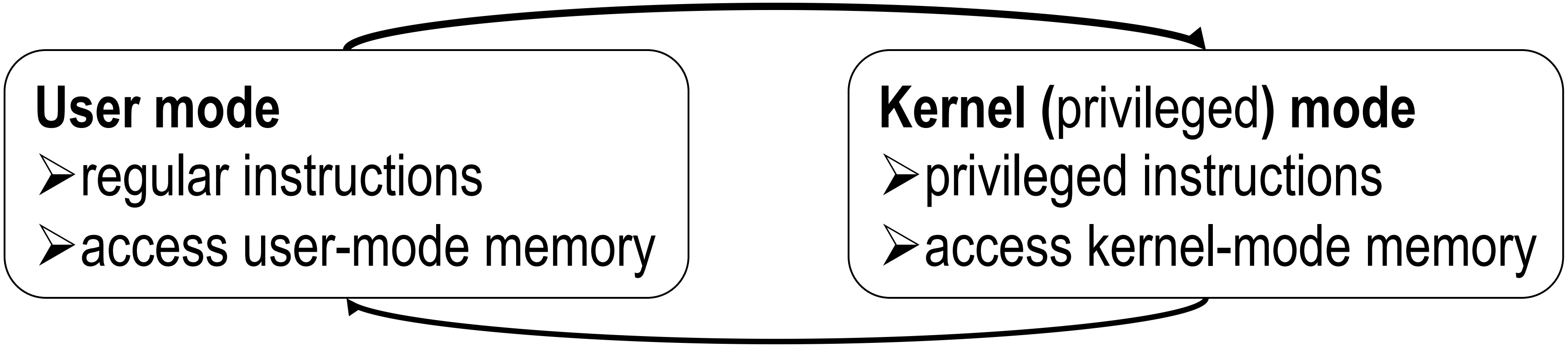
What about the kernel itself?



- What happens when OS is running?
  - OS runs with relocation turned off (a bit in processor status word (PSW) controls relocation)
- How to prevent users from controlling base & bound, relocation?
- Does kernel need multiple address spaces?
- How does OS regain control? Need to atomically
  - Branch into/out of OS
  - Turn relocation on/off

Support in modern processors: User ⇔ Kernel

An interrupt or exception (INT)



User mode

- ➢regular instructions
- ➢access user-mode memory

Kernel (privileged) **mode**

- ➢privileged instructions
- ➢access kernel-mode memory

A special instruction (IRET)

Privileged instructions

- Special Instructions
  - system call (invoked by user program)
  - memory mapping, TLB, etc.
  - device registers
  - I/O operations

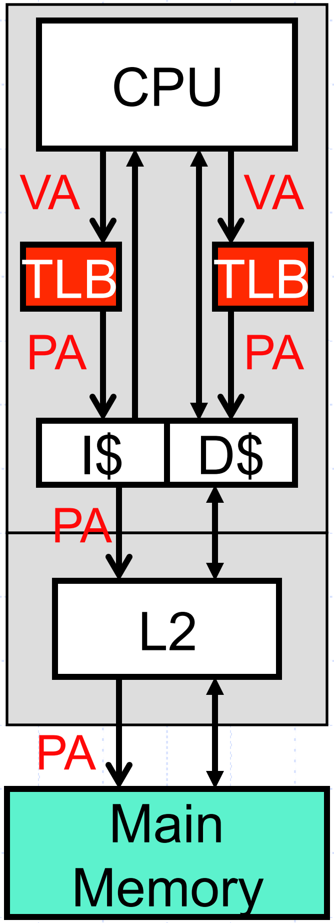
Hardware support for Modes

- *Mode bit* added to hardware to indicate the current mode:
  - kernel (0)
  - user (1)

Backup

- Backup slides will not be in exams, but can help you understand topics that will be in exams

Physical (Address) Caches

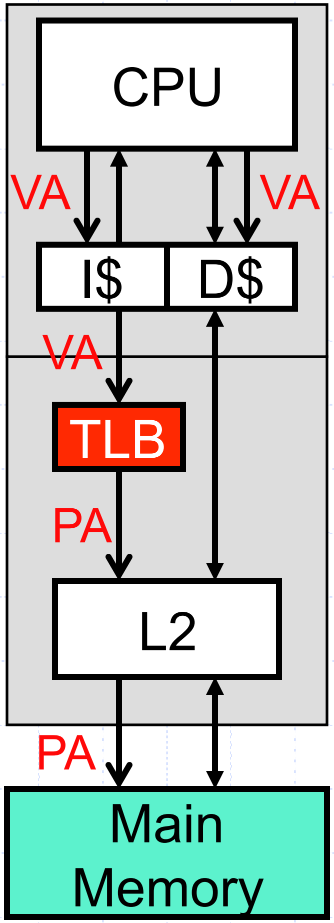


- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by PAs
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA to VA at the outset

  + Cached inter-process communication works
    - Single copy indexed by PA

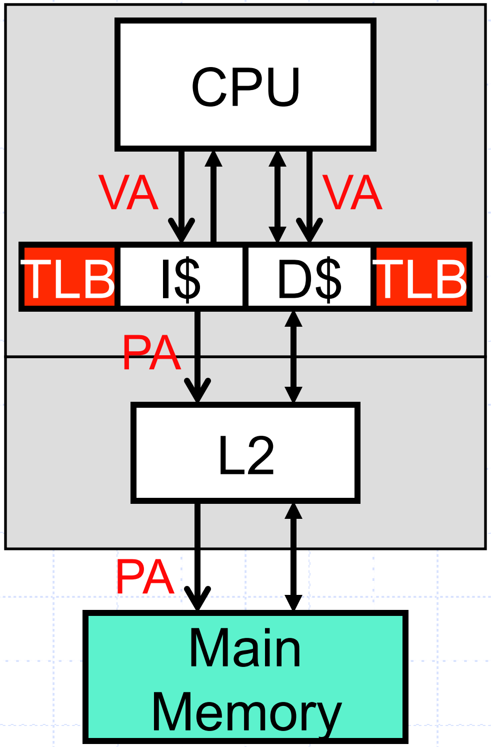
  – Slow: adds at least one cycle to t_{hit}

Virtual Address Caches (VI/VT)



- Alternative: **virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2
  - + Fast: avoids translation latency in common case
  - – Problem: VAs from ***different processes*** are distinct physical locations (with different values) (call homonyms)
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    - Can't allow same PA in the cache twice
    - Also a problem for DMA I/O
  - Can be handled, but very complicated

Parallel TLB/Cache Access (VI/PT)



Compromise: **access TLB in parallel**

- *In small caches, index of VA and PA the same*
  - *VI == PI*
- Use the VA to index the cache
- Tagged by PAs
- Cache access and address translation in parallel

+ No context-switching/aliasing problems

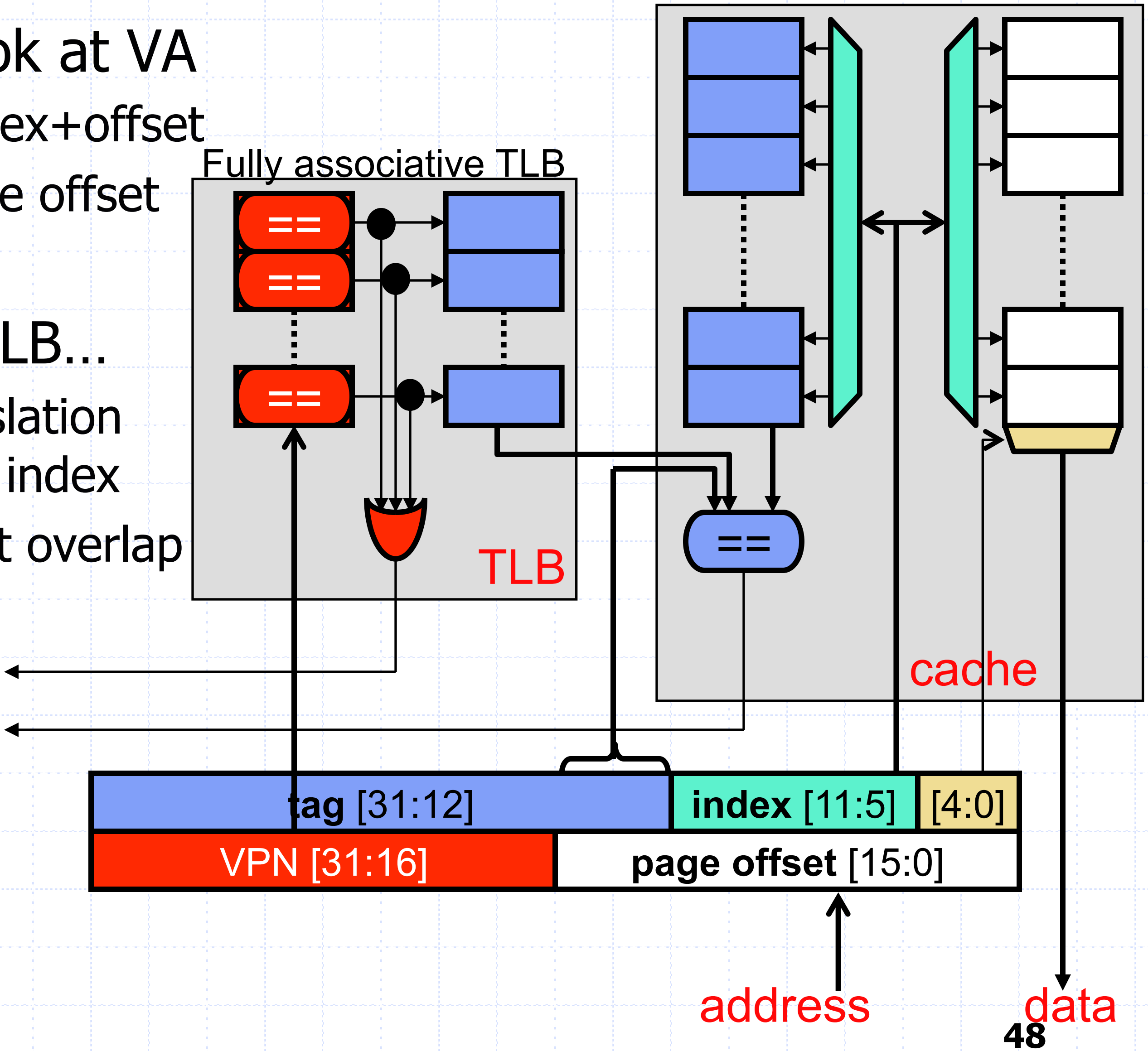
+ Fast: no additional t_{hit} cycles

- Common organization in processors today

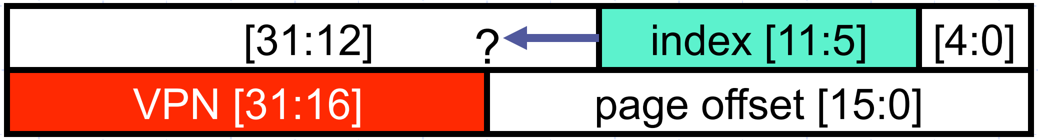
Parallel Cache/TLB Access

- Two ways to look at VA
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset

- Parallel cache/TLB…
  - If address translation doesn't change index
  - VPN/index don't overlap



Cache Size And Page Size



- Relationship between page size and L1 cache size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - **Rule: (cache size) / (associativity) ≤ page size**
  - Result: associativity increases allowable cache sizes
  - Systems are moving towards bigger (64KB) pages
    - To use parallel translation with bigger caches
    - To amortize disk latency
  - Example: Pentium 4, 4KB pages, 8KB, 2-way SA L1 data cache