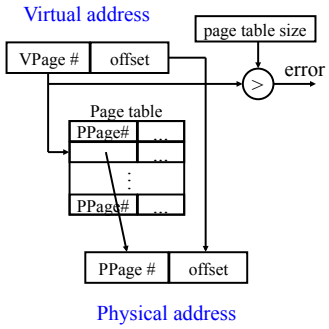
Paging (cont): Inverted Page Table, Bits in PTE

ECE595
Feb 15

Y. Charlie Hu

[lec16] Paging



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

Paging vs. segmentation

- Segmentation:
  - External fragmentation
  - Complicated allocation, swapping
  - \+ Small segmentation table
- Paging
  - Internal fragmentation
  - \+ Easy allocation, swapping
  - Large page table

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?

[lec16] How many PTEs do we need?

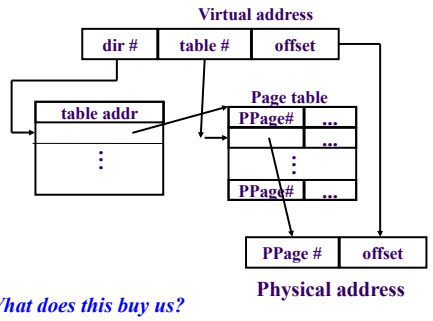
- Worst case for 32-bit address machine
  - # of processes × 2^{20} (if page size is 4096 bytes)
- What about 64-bit address machine?
  - # of processes × 2^{52}

Page table

- The page table has to be contiguous in physical mem
  - Potentially large
  - Consecutive pages in mem hard to find
- How can we be flexible?

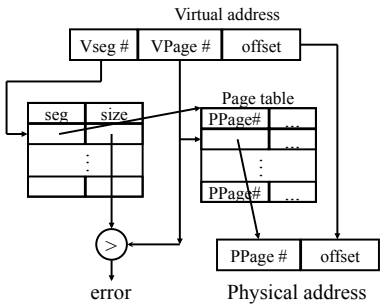
"All computer science problems can be solved with an extra level of indirection."

Two-level page tables



What does this buy us?

Segmentation with paging



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

System design principle: Separating Policy from Mechanism

Mechanism – tool/method 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

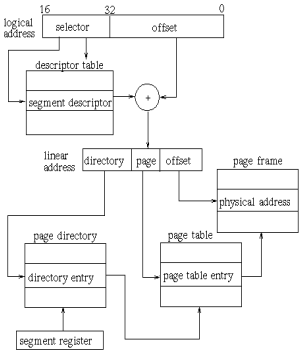
Separation leads to flexibility

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

- Inverted page table
- Bits in a PTE
- TLB

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}

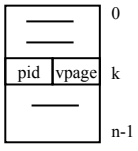
Hmm, but my PC only has 1GB, 256K PTEs should be enough?!

Inverted Page Table

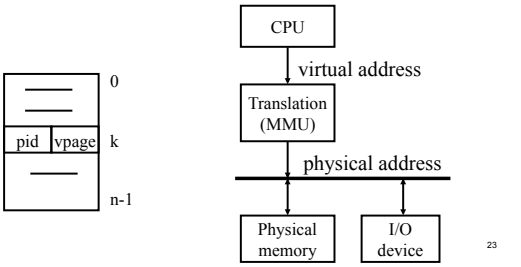
- Motivation
  - Example: 2 processes, page table has 1M entries, 10 phy pages
- Is there a way to save page table space?

Ideally,

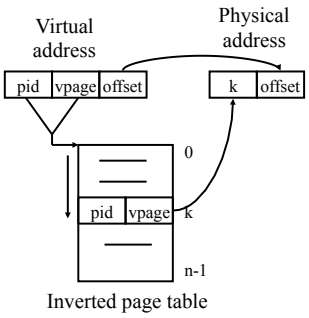
- One PTE for each physical page frame, disregarding how many processes
  - Assuming rest virtual addressed not allocated/used



But,



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 do two processes share memory under 1-level or 2-level page table?
- How can two processes share memory under inverted page table?

Today's topics

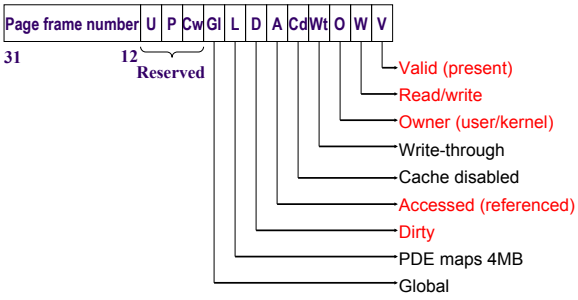
- Inverted page table
- Bits in a PTE
- TLB

What is happening to heap ph mem allocation before and after malloc()?

- Before malloc()?
- After malloc()?
- Upon first access?

- How to capture the first write to a virtual page?
  - e.g. want to trap into page fault handler

x86 Page Table Entry



What is happening before and after malloc()?

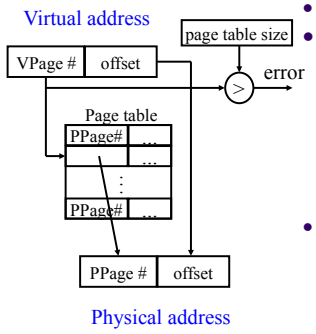
- Before malloc()?
- After malloc()?
- Upon first access?

- How to capture the first write to a virtual page?
  - e.g. want to trap into page fault handler
    - Use valid bit
  - In handler, check if vpage is malloced.
    - If not, segmentation fault
    - Else allocate physical page

Today's topics

- Inverted page table
- Bits in a PTE
- TLB

Paging implementation – how does it really work?

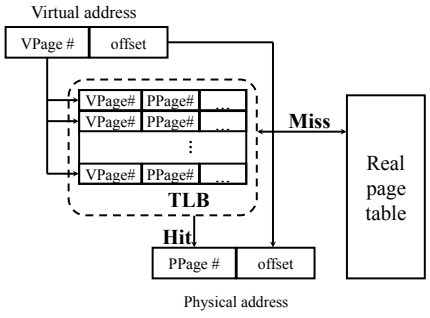


- Where to store page table?
- How to use MMU?
  - Even small page tables too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses
    - What does MMU have to do?
- Page size?
  - Small page -> big table
    - 32-bit with 4k pages
  - Large page ->small table but large internal fragmentation
    - e.g., data seg is 6 Kbytes

Performance problem with paging

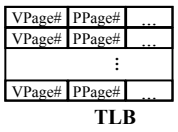
- How many extra memory references to access page tables?
  - One-level page table?
  - Two-level page table (midterm problem)?
- Solution: *reference locality!*
  - In a short period of time, a process is likely accessing only a few pages
  - Instead of storing only page table starting address in MMU,

Translation Look-aside Buffer (TLB)



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

Bits in a TLB Entry



- 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
    - If page not valid, 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 dictated 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?

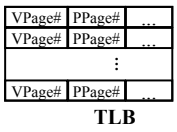
More TLB Issues

- 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 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

Bits in a TLB Entry



TLB

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - PTE
- Optional (useful) bits
  - ASIDs -- Address-space identifiers (process tags)

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

Reading

- Chapter 8