Administrivia

- Homework
  - HW2 will be released later today
  - Due Friday May 19
Recap

- Virtual address space
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses
- Various techniques
  - Fixed partitions, variable partitions, segmentation, paging
- Virtual memory
  - Memory that is not backed by physical memory but swapped to disk
Problems of the naive paging implementation

• Memory required to hold page table can be significant
  • 32 bit address space w/ 4KB pages = $2^{20}$ PTEs
  • 4 bytes/PTE = 4MB/page table
  • Solution – page the page tables

• Memory reference overhead
  • 2 references per address lookup (page table, then memory)
  • Solution – use a hardware cache of lookups
Managing page tables

• Q: if a 32-bit address space w/ 4K pages consumes 4MB for page table, how much memory is needed for 64-bit address space?

• How can we reduce this overhead?
  • Observation: only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

• How do we only map what is being used?
  • Dynamically extend page table
Paging the page table

• Paging - not all virtual addresses are valid/mapped
  • OS allocates/maps physical memory to the address space based on process’ demand
  • The valid bit in the page table entry (PTE) determines whether the corresponding virtual address has a mapping or not

• Page table
  • A contiguous physical memory region to store PTEs (Why?)
  • How allow some part of the table to be invalid
Multi-level page tables

- Virtual addresses (VAs) have several parts:
  - 1st level page number, 2nd level page number, ..., and page offset
  - 1st level page table maps 1st level page number to 2nd level page table
  - 2nd level page table maps 2nd level page number to 3rd level page table
  - ...
  - The last level page tables maps the last level page number to physical page
- Offset indicates where in physical page address is located
Example: 2-level page tables

• How many bits in offset? 4K = 12 bits

• 32-bit physical address space -> 4 bytes/PTE (Why?)

• Want master page table in one page: 4K/4 bytes = 1K entries
  • Hence, 1K secondary page tables

• How many bits?
  • Master page number = 10 bits (because 1K entries)
  • Offset = 12 bits
  • Secondary page number = 32 – 10 – 12 = 10 bits
Recap: 1-level page translation

Virtual Address

Page number	Offset

Page Table

Page frame

Physical Address

Page frame	Offset

Physical Memory

Base Addr	Register storing page table addr
2-level page translation

Virtual Address

Master page number  Secondary  Offset

Page table

Physical Address

Page frame  Offset

Physical Memory

Master Page Table

Secondary Page Table
Intel IA32e translation (from manual)

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging
More flexible paging

• PTE can now points to either the next level table or a physical page
  • But only the last level can points to physical page
• Can this be more flexible?
  • Larger physical pages (2MB, 1GB)
  • Why these sizes?
  • Why larger pages?
Managing page tables

• How does OS manage/access page tables (which address space)?
• Physical memory
  • Easy to address, no translation required
  • But requires mode switch and software address translation
• Virtual memory (kernel virtual address space)
  • How to translate between virtual address and physical address?
Efficient translations

- Recall that our original page table scheme doubled the latency of doing memory lookups
  - One lookup into the page table, another to fetch the data
- 2-level page tables triple the latency!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
Translation Lookaside Buffers (TLB)

- Cache translations in hardware
- Managed by Memory Management Unit (MMU)
- Translate virtual page #s into physical pages (PTEs)
  - Can be done in a single machine cycle
- Fully associative cache (all entries looked up in parallel)
  - Keys are virtual page numbers
  - Values are PTEs (entries from page tables)
TLBs (cont.)

• Why does this help?
  • Exploits locality: Processes use only handful of pages at a time
    • 16-48 entries/pages (64-192K)
    • Only need those pages to be "mapped"
  • Hit rates are therefore very important
TLB hit

CPU Chip

CPU

1 VA

CPU

MMU

2 VPN

3 PTE

Cache/Memory

PA

4

Data

5
Managing TLBs (1)

- Who places translations into the TLB (loads the TLB)?
- Hardware (Memory Management Unit) [x86]
  - Knows where page tables are in main memory
  - OS maintains tables, HW accesses them directly
  - Tables have to be in HW-defined format (inflexible)
Managing TLBs (2)

- Software loaded TLB (OS) [MIPS, Alpha, Sparc, PowerPC]
  - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
  - Must be fast (but still 20-200 cycles)
  - CPU ISA has instructions for manipulating TLB
  - Tables can be in any format convenient for OS (flexible)
Managing TLBs (3)

- OS ensures that TLB and page tables are consistent
  - When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB (special hardware instruction)
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? Who does it?
Managing TLBs (4)

- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple, e.g., Least Recently Used (LRU)
Summary

- Multi-level page tables - reduces page table memory consumption
- TLB - efficient address translation
Next class ...

- More on paging
- Textbook
  - Module 21, 22