## ECE 250 / CS 250 Introduction to Computer Architecture

Virtual Memory Benjamin C. Lee Duke University

Slides from Daniel Sorin (Duke) and are derived from work by Amir Roth (Penn) and Alvy Lebeck (Duke) • Homework #5 – Due Apr 11 @ 11:55pm

#### Physical Main Memory?

- On a 32-bit architecture, there are 2<sup>32</sup> byte addresses
  - Requires 4 GB of memory
  - But not everyone buys machines with 4 GB of memory
  - And what about 64-bit architectures?
- Let's take a step back...

## This Unit: Main Memory



- Memory hierarchy review
- DRAM technology
  - A few more transistors
  - Organization: two level addressing
- Building a memory system
  - Bandwidth matching
  - Error correction
- Organizing a memory system
- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

## Virtual Memory

- Idea of treating memory like a cache
  - Contents are a dynamic subset of program's address space
  - Dynamic content management is transparent to program
- Actually predates "caches" (by a little)
- Original motivation: **compatibility** 
  - IBM System 370: a family of computers with one software suite
  - + Same program could run on machines with different memory sizes
    - Caching mechanism made it appear as if memory was 2<sup>N</sup> bytes
    - Regardless of how much memory there actually was
  - Prior, programmers explicitly accounted for memory size

#### Virtual memory

• Virtual: "in effect, but not in actuality" (i.e., appears to be, but isn't)

# Virtual Memory



- Programs use virtual addresses (VA)
  - 0...2<sup>N</sup>-1
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Itanium is 64-bit
- Memory uses physical addresses (PA)
  - 0...2<sup>M</sup>-1 (M<N, especially if N=64)</li>
  - $2^{M}$  is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

© 2012 Daniel J. Sorin from Roth

ECE 152

## **Other Uses of Virtual Memory**

- Virtual memory is quite useful
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **multiprogramming** 
  - Each process thinks it has 2<sup>N</sup> bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - "System" maps VPs from different processes to different PPs

+ Prevents processes from reading/writing each other's memory



## Still More Uses of Virtual Memory

- Inter-process communication
  - Map VPs in different processes to same PPs
- Direct memory access I/O
  - Think of I/O device as another process
  - Will talk more about I/O in a few lectures

#### Protection

- Piggy-back mechanism to implement page-level protection
- Map VP to PP ... and RWX protection bits
- Attempt to execute data, or attempt to write insn/read-only data?
  - Exception  $\rightarrow$  OS terminates program

## **Address Translation**



- VA→PA mapping called **address translation** 
  - Split VA into virtual page number (VPN) and page offset (POFS)
  - Translate VPN into physical page number (PPN)
  - POFS is not translated why not?
  - $VA \rightarrow PA = [VPN, POFS] \rightarrow [PPN, POFS]$
- Example above
  - 64KB pages  $\rightarrow$  16-bit POFS
  - 32-bit machine  $\rightarrow$  32-bit VA  $\rightarrow$  16-bit VPN (16 = 32 16)
  - Maximum 256MB memory  $\rightarrow$  28-bit PA  $\rightarrow$  12-bit PPN

## **Mechanics of Address Translation**

- How are addresses translated?
  - In software (now) but with hardware acceleration (a little later)
- Each process is allocated a **page table (PT)** 
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
    union { int ppn, disk_block; }
    int is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];
int translate(int vpn) {
    if (pt[vpn].is_valid)
        return pt[vpn].ppn;
}
```



## Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages
- Solution
  - 32-bit machine  $\rightarrow$  32-bit VA  $\rightarrow$  4GB virtual memory
  - 4GB virtual memory / 4KB page size  $\rightarrow$  1M VPs
  - 1M VPs \* 4B PTE  $\rightarrow$  4MB page table
- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get enormous
  - There are ways of making them smaller

## Multi-Level Page Table

- One way: multi-level page tables
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels
- Example: two-level page table for machine on last slide
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs  $\rightarrow$  1K PTEs fit on a single page
    - 1M PTEs / (1K PTEs/page)  $\rightarrow$  1K pages to hold PTEs
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages  $\rightarrow$  1K pointers
    - 1K pointers \* 32-bit VA  $\rightarrow$  4KB  $\rightarrow$  1 upper level page

## Multi-Level Page Table





## Multi-Level Page Table

- Have we saved any space?
  - Isn't total size of L2 PTE pages same as singlelevel table (i.e., 4MB)?
  - Yes, but...
- Large virtual address regions unused
  - Corresponding L2 pages need not exist
  - Corresponding L1 pointers are null
- Ex: 2MB code, 64KB stack, 16MB heap
  - Each L2 page maps 4MB of virtual addresses
  - 1 page for code, 1 for stack, 4 for heap, (+1 L1)
  - 7 total pages for PT = 28KB (<< 4MB)



## **Address Translation Mechanics**

- The six questions
  - What? address translation
  - Why? compatibility, multi-programming, protection
  - How? page table
  - What performs translation?
  - When is translation performed?
  - Where does page table reside?
- Option I: process (program) translates its own addresses
  - Page table resides in process visible virtual address space
  - Bad idea: implies that program (and programmer)...
    - Must know about physical addresses
      - Isn't that what virtual memory is designed to avoid?
    - Can forge physical addresses and mess with other programs
  - Translation on L2 miss or always? How would program know?

## Who? Where? When? Take II

- Option II: operating system (OS) translates for process
  - Page table resides in OS virtual address space
  - + User-level processes cannot view/modify their own tables
  - + User-level processes need not know about physical addresses
  - Translation on L2 miss (requires OS syscall)
- L2 miss: interrupt transfers control to OS handler
  - Handler translates VA by accessing process' s page table
  - Accesses memory using PA
  - Returns to user process when L2 fill completes
  - Still slow: added interrupt handler and PT lookup to memory access
  - What if PT lookup itself requires memory access? Head spinning...

## **Translation Buffer**



- Functionality problem? Add indirection!
- Performance problem? Add cache!

Address translation too slow?

- Cache translations in **translation buffer (TB)** 
  - Small cache: 16–64 entries, often fully assoc
- + Exploits temporal locality in PT accesses

"data

**PPN** 

PPN

PPN

+ OS handler only on TB miss

#### **TB** Misses

- **TB miss:** requested PTE not in TB, but in PT
  - Two ways of handling
- 1) OS routine: reads PT, loads entry into TB
  - Privileged instructions in ISA for accessing TB directly
  - Latency: one or two memory accesses + OS call
- 2) Hardware FSM: does same thing
  - Store PT root pointer in hardware register
  - Store physical addresses PT root and L1 table pointers
    - FSM doesn't have to translate them
  - + Latency: saves cost of OS call

### **Nested TB Misses**

- **Nested TB miss**: when OS handler itself has a TB miss
  - TB miss on handler instructions
  - TB miss on page table VAs
  - Not a problem for hardware FSM: no instructions, PAs in page table
- Software handler implementation is tricky
  - First, save current TB miss info before accessing page table
    - So that nested TB miss info doesn't overwrite it
  - Second, lock nested miss entries into TB
    - Prevent TB conflicts that result in infinite loop
    - Another good reason to have a highly-associative TB

### Page Faults

#### • Page fault: PTE not in TB or in PT

- Page is simply not in memory
- Starts out as a TB miss, detected by OS handler/hardware FSM

#### OS routine

- OS software chooses a physical page to replace
  - "Working set": more refined software version of LRU
    - Tries to see which pages are actively being used
    - Balances needs of all current running applications
  - If dirty, write to disk (like dirty cache block with writeback \$)
- Read missing page from disk (done by OS)
  - Takes so long (10ms), OS schedules another task
- Treat like a normal TB miss from here

## **Virtual Caches**



- Memory hierarchy so far: **virtual caches** 
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - + Fast: avoids translation latency in common case
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
  - Does inter-process communication work?
    - Aliasing: multiple VAs map to same PA
      - How are multiple cache copies kept in sync?
      - Also a problem for I/O (later in course)
    - Disallow caching of shared memory? Slow

## **Physical Caches**



- Alternatively: **physical caches** 
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - + No need to flush caches on process switches
    - Processes do not share PAs
  - + Cached inter-process communication works
    - Single copy indexed by PA
  - Slow: adds 1 cycle to  $t_{hit}$

## **Virtual Physical Caches**



#### Compromise: virtual-physical caches

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel
- + No context-switching/aliasing problems
- + Fast: no additional t<sub>hit</sub> cycles
- A TB that acts in parallel with a cache is a **TLB** 
  - Translation Lookaside Buffer
- Common organization in processors today

## Cache/TLB Access

- Two ways to look at VA
  - Cache: TAG+IDX+OFS
  - TLB: VPN+POFS
- Parallel cache & TLB ...
  - Possible if address translation doesn't change IDX
  - VPN, IDX don't overlap

2 1022 023 TLB TLB hit/miss← cache cache hit/miss ← 31:12] [11:2] 1:0 << VPN [31:16] POFS[15:0] address ECE 152

0

#### Cache Size And Page Size

[31:12]	IDX[11:2]	1:0
VPN [31:16]	[15:0]	

- Relationship between page size and L1 I\$(D\$) size
  - Forced by non-overlap between VPN and IDX portions of VA
  - Which is required for parallel TLB and cache access
  - I\$(D\$) size / **associativity** ≤ page size
  - Big caches must be set associative
    - Big cache  $\rightarrow$  more index bits (fewer tag bits)
    - More set associative  $\rightarrow$  fewer index bits (more tag bits)
  - Systems are moving towards bigger (64KB) pages
    - To amortize disk latency
    - To accommodate bigger caches

## **TLB** Organization

- Like caches: TLBs also have ABCs
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
- **Rule of thumb**: TLB should "cover" L2 contents
  - In other words: #PTEs \* page size  $\geq$  L2 size
  - Why? Think about this ...

### **Flavors of Virtual Memory**

- Virtual memory almost ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even some embedded (in non-computer) processors support it
- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - IA32 uses this
    - Makes life very difficult
  - **Paged segments**: don't ask

## Summary

- DRAM
  - Two-level addressing
  - Refresh, access time, cycle time
- Building a memory system
  - DRAM/bus bandwidth matching
- Memory organization
- Virtual memory
  - Page tables and address translation
  - Page faults and handling
  - Virtual, physical, and virtual-physical caches and TLBs

#### Next part of course: Exceptions, Interrupts, and I/O