Virtual Memory

(plus a bit on error correction)

Slides from
Daniel J. Sorin (Duke) and
Amir Roth (Penn)

Where We Are in This Course Right Now

- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We understand how to design caches
  - We know how to implement main memory in DRAM

- Now:
  - We learn about virtual memory

- Next:
  - We learn about the lowest level of storage (disks) and I/O

One last problem: How to fit into Memory

- Reasonable Memory: 4GB—64GB?
  - 32-bit address space: 4GB/program: run 1—16 programs?
  - 64-bit address space: need 16 Billion GB for 1 program?

- Not going to work
- Instead: virtual memory
  - Give every program the illusion of entire address space
  - Hardware and OS move things around behind the scenes

- How?
  - Functionality problem -> add level of indirection
  - Good rule to know

End of memory hierarchy: Virtual Memory

- 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^n 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)
  - Programs use virtual addresses (VA)
  - Virtual memory is quite useful
- Memory uses physical addresses (PA)
  - Memory uses physical addresses (PA)
  - Protection
- VA → PA at page granularity (VP → PP)
  - VA → PA at page granularity (VP → PP)
  - VA → PA at page granularity (VP → PP)

Other Uses of Virtual Memory

- Virtual memory is quite useful
- Other Uses of Virtual Memory
- Address Translation

Mechanics of Address Translation

- How are addresses translated?
- Mechanics of Address Translation
- Page Table Size

Page Table Size

- How big is a page table on the following machine?
- Page Table Size
- Address Translation

Still More Uses of Virtual Memory

- Inter-process communication
- Still More Uses of Virtual Memory
- Other Uses of Virtual Memory

Address Translation

- VA → PA mapping called address translation
- Address Translation
- Page Table Size
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 → 1K PTEs fit on a single page
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages
    - 1K pointers * 32-bit VA → 4KB → 1 upper level page

Multi-Level Page Table

- Have we saved any space?
  - Isn’t total size of 2nd level PTE pages same as single-level table (i.e., 4MB)?
  - Yes, but...

- Large virtual address regions unused
  - Corresponding 2nd-level pages need not exist
  - Corresponding 1st-level pointers are null

- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level page maps 4MB of virtual addresses
  - 1 page for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages for PT = 28KB (<< 4MB)

Address Translation Mechanics

- The six questions
  - What? address translation
  - Why? compatibility, multi-programming, protection
  - How? page table
  - Who performs it?
  - When?
  - 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 what virtual memory is designed to avoid
  - 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
    - Otherwise, OS SYSCALL before any fetch, load, or store

- L2 miss: interrupt transfers control to OS handler
  - Handler translates VA by accessing process’s page table
  - Returns to user process when L2 fill completes
  - Slow: added interrupt handler and PT lookup to memory access

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
    - 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 (e.g., Alpha)
     - Privileged instructions in ISA for accessing TB directly
     - Latency: one or two memory accesses + OS call

  2) **Hardware FSM**: does same thing (e.g., IA-32)
     - Store PT root pointer in hardware register
     - Make PT root and 1st-level table pointers physical addresses
     - So 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

- Handling is tricky for SW handler, but possible
  - 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
  - Clocked inter-process communication works
    - Single copy indexed by PA
    - Slow: adds 1 cycle to \( t_{an} \)

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 \( t_{an} \) 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
- Can have parallel cache & TLB ...
  - If address translation doesn’t change IDX
    - VPN/IDX don’t overlap

Cache Size And Page Size

- Relationship between page size and L1 I$(D$) size
  - Forced by non-overlap between VPN and IDX portions of VA
  - Which is required for TLB access
    - I$(D$) size / associativity ≤ page size
  - Big caches must be set associative
    - Big cache → more index bits (fewer tag bits)
    - More set associative → 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 ≥ 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

Also good to know about Main Memory

- Error correction

Error Detection and Correction

- One last thing about DRAM technology: errors
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic α-particle strikes
    - Many more bits
    - Modern DRAM systems: built-in error detection/correction
- Key idea: checksum-style redundancy
  - Main DRAM chips store data, additional chips store f(data)
    - |f(data)| < |data|
  - On read: re-compute f(data), compare with stored f(data)
  - Different ? Error...
    - Option I (detect): kill program
    - Option II (correct): enough information to fix error? fix and go on
Error Detection and Correction

- Error detection/correction schemes distinguished by...
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection Example: Parity

- **Parity**: simplest scheme
  - \( f(\text{data}_{b-1}, \ldots, \text{data}_0) = \text{XOR}(\text{data}_{b-1, \ldots, 0}) \)
  - Single-error detect: detects a single bit flip (common case)
    - Will miss two simultaneous bit flips...
    - But what are the odds of that happening?
      - Zero-error correct: no way to tell which bit flipped
  - Many other schemes exist for detecting/correcting errors
    - Take ECE 552 (Fault Tolerant Computing) for more info

Summary

- Virtual memory
  - Page tables and address translation
  - Page faults and handling
  - Virtual, physical, and virtual-physical caches and TLBs
- Error correction

**Next part of course: I/O**