

## ECE 590

Virtual Memory  
(plus a bit on error correction)  
Slides from  
Daniel J. Sorin (Duke) and  
Amir Roth (Penn)

1

## DRAM Packaging

- Just talked about DRAM: here is a picture of a DIMM
  - E.g., 8 DRAM chips, each chip is 4 or 8 bits wide



© 2012 Daniel J. Sorin from Roth

ECE 590

2

## Where We Are in This Course Right Now

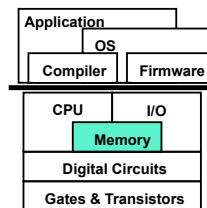
- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We understand how to design caches
  - We know how to implement main memory in DRAM
- Now:
  - We learn about virtual memory
- Next:
  - We learn about the lowest level of storage (disks) and I/O

© 2012 Daniel J. Sorin from Roth

ECE 590

3

## End of memory hierarchy: Virtual Memory



- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

© 2012 Daniel J. Sorin from Roth

ECE 590

4

## 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

© 2012 Daniel J. Sorin from Roth

ECE 590

5

## Virtual Memory

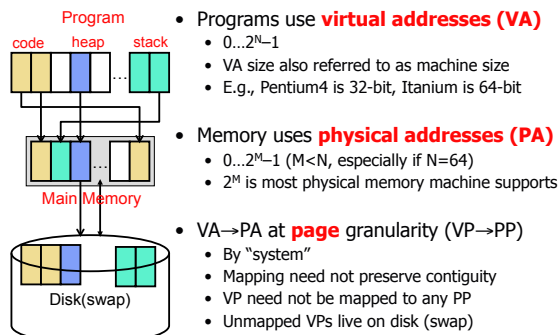
- Idea of treating memory like a cache
  - Contents are a dynamic subset of program's address space
  - Dynamic content management is transparent to program
- Actually predates "caches" (by a little)
- Original motivation: **compatibility**
  - IBM System 370: a family of computers with one software 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)

© 2012 Daniel J. Sorin from Roth

ECE 590

6

## Virtual Memory



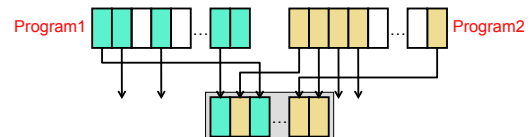
© 2012 Daniel J. Sorin from Roth

ECE 590

7

## 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  $2^N$  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



© 2012 Daniel J. Sorin from Roth

ECE 590

8

## Still More Uses of Virtual Memory

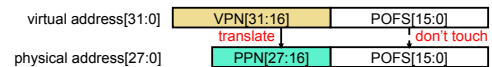
- Inter-process communication
  - Map VPs in different processes to same PPs
- Direct memory access I/O
  - Think of I/O device as another process
  - Will talk more about I/O in a few lectures
- Protection**
  - Piggy-back mechanism to implement page-level protection
  - Map VP to PP ... and RWX protection bits
  - Attempt to execute data, or attempt to write insn/read-only data?
    - Exception  $\rightarrow$  OS terminates program

© 2012 Daniel J. Sorin from Roth

ECE 590

9

## Address Translation



- VA  $\rightarrow$  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

© 2012 Daniel J. Sorin from Roth

ECE 590

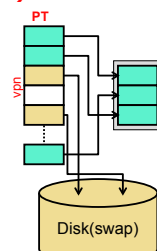
10

## 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;
}
```



© 2012 Daniel J. Sorin from Roth

ECE 590

11

## 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

© 2012 Daniel J. Sorin from Roth

ECE 590

12

## 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
    - 1M PTEs / (1K PTEs/page) → 1K pages to hold PTEs
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers \* 32-bit VA → 4KB → 1 upper level page

## Multi-Level Page Table

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table

```

struct {
    union { int ppn, disk_block; }
    int is_valid, is_dirty;
} PTE;

struct {
    struct PTE ptes[1024];
} L2PT;

struct L2PT *pt[1024];

int translate(int vpn) {
    struct L2PT *l2pt = pt[vpn >> 10];
    if (l2pt && l2pt->ptes[vpn & 1023].is_valid)
        return l2pt->ptes[vpn & 1023].ppn;
}

```

## 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)

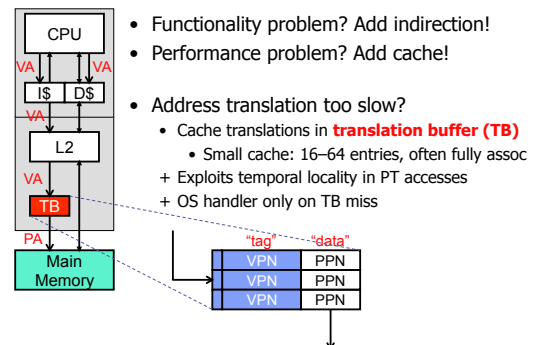
## Address Translation Mechanics

- 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 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
    - 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
  - 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



## 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

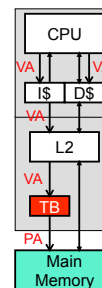
## Nested TB Misses

- **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

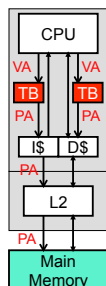
- **Page fault:** PTE not in TB or in PT
  - Page is simply not in memory
  - Starts out as a TB miss, detected by OS handler/hardware FSM
- **OS routine**
  - OS software chooses a physical page to replace
    - **"Working set":** more refined software version of LRU
      - Tries to see which pages 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



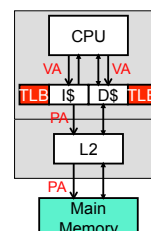
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - + Fast: avoids translation latency in common case
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **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
  - + Cached inter-process communication works
    - Single copy indexed by PA
  - Slow: adds 1 cycle to  $t_{hit}$

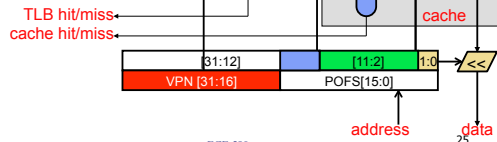
## Virtual Physical Caches



- 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  $t_{hit}$  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



© 2012 Daniel J. Sorin from Roth

ECE 590

## 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

© 2012 Daniel J. Sorin from Roth

ECE 590

26

## 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 ...

© 2012 Daniel J. Sorin from Roth

ECE 590

27

## Flavors of Virtual Memory

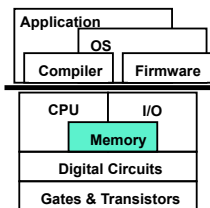
- Virtual memory almost ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even some embedded (in non-computer) processors support it
- Several forms of virtual memory
  - Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - Segmentation:**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

© 2012 Daniel J. Sorin from Roth

ECE 590

28

## Also good to know about Main Memory



- Error correction**

© 2012 Daniel J. Sorin from Roth

ECE 590

29

## Error Detection and Correction

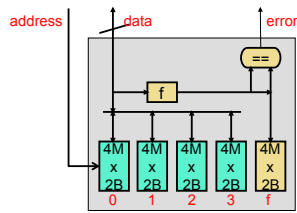
- One last thing about DRAM technology: **errors**
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic  $\alpha$ -particle strikes
    - Many more bits
  - Modern DRAM systems: built-in error detection/correction
- Key idea: checksum-style redundancy**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

© 2012 Daniel J. Sorin from Roth

ECE 590

30

## 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
  - $f(\text{data}_{N-1:0}) = \text{XOR}(\text{data}_{N-1}, \dots, \text{data}_1, \text{data}_0)$
  - + 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**