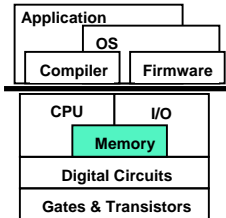


## This Unit: Main Memory

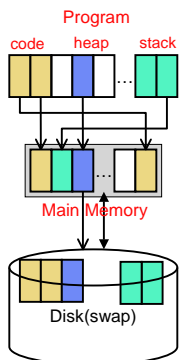


- Memory hierarchy review
- DRAM technology
  - A few more transistors
  - Organization: two level addressing
- Building a memory system
  - Bandwidth matching
  - Error correction
- Organizing a memory system
- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

## 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  $2^N$  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)

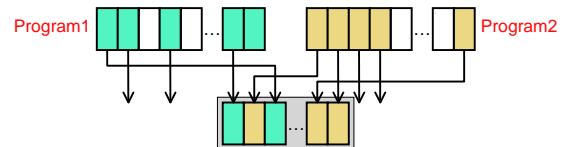
## Virtual Memory



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

## 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





## 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

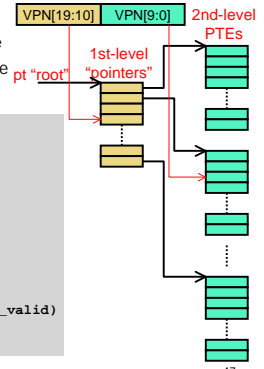
- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table

```

struct {
    union { int ppn, disk_block; }
    int is_valid, is_dirty;
} PTE;
struct {
    struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

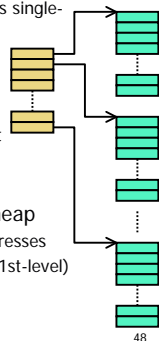
int translate(int vpn) {
    struct L2PT *l2pt = pt[vpn>>10];
    if (l2pt && l2pt->ptes[vpn&1023].is_valid)
        return l2pt->ptes[vpn&1023].ppn;
}

```



## 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)



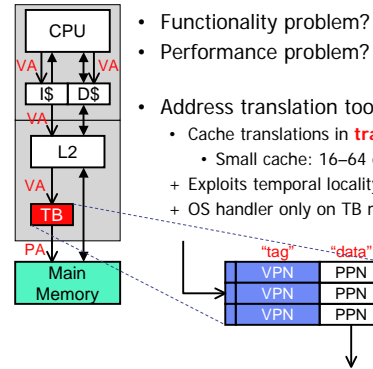
## Address Translation Mechanics

- 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
  - Translate VA by accessing process' 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 FA
  - + 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 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