ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

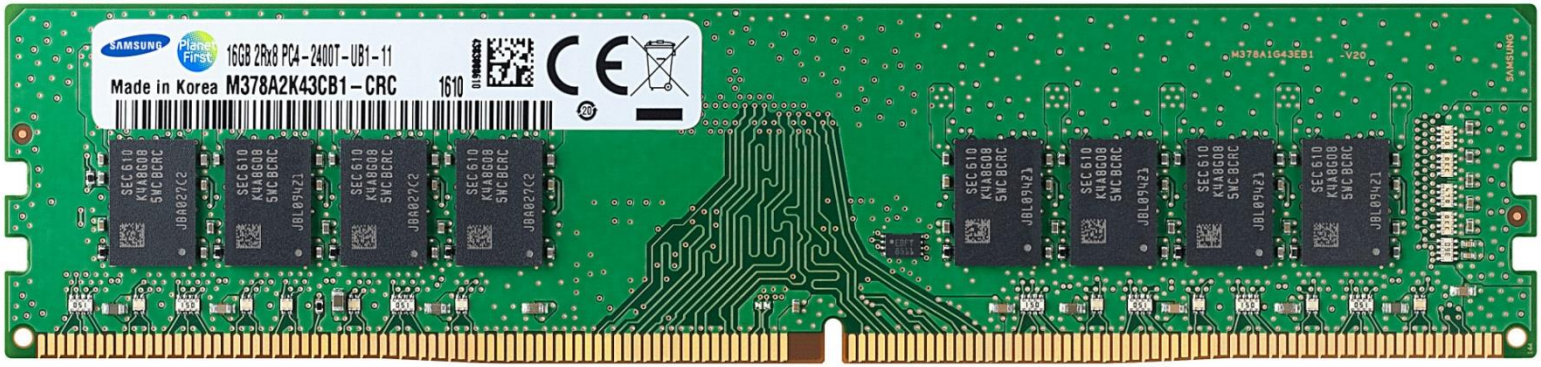
Virtual Memory

Xin Li & Dawei Liu
Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

DRAM Packaging

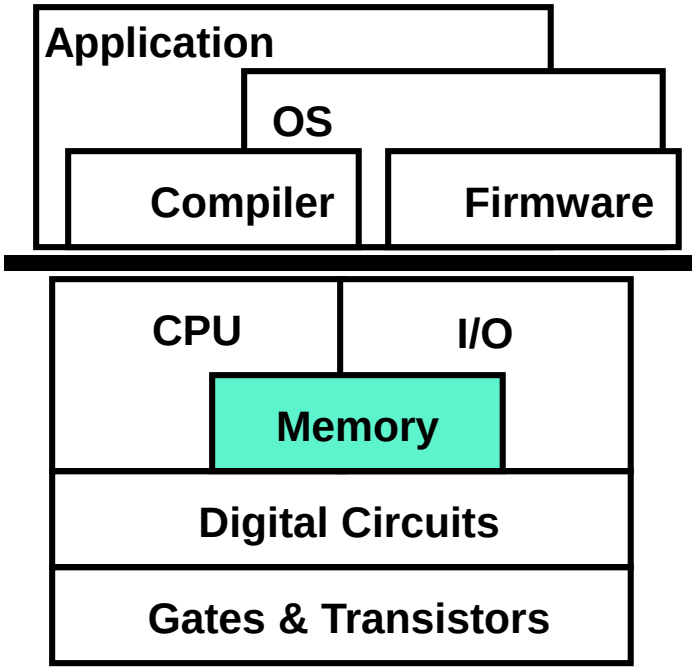
- Just talked about DRAM: here is a picture of a DIMM (dual in-line memory module)
  - E.g., typically 64 bits wide
  - SIMM: 32 bits



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

End of memory hierarchy: Virtual Memory



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

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

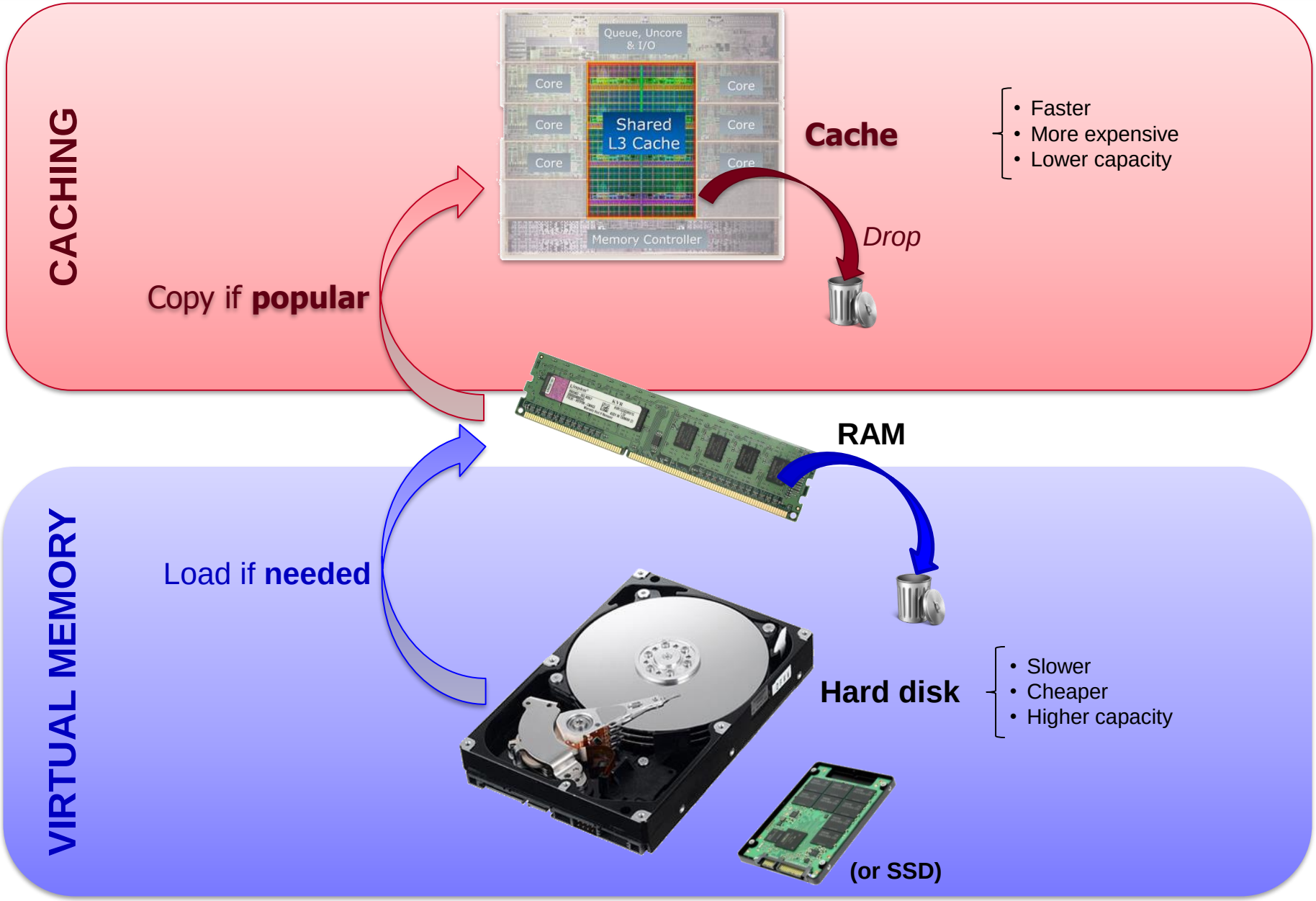
Virtual Memory

- Idea of treating memory like a cache
  - Contents are a dynamic subset of program's address space
  - Dynamic content management is transparent to program

- 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

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

Figure: caching vs. virtual memory

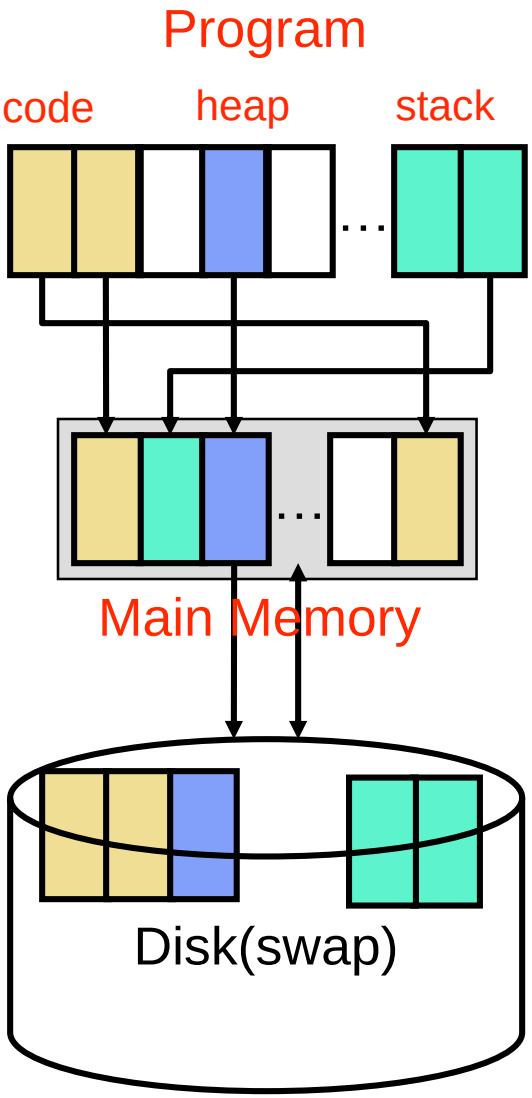


Demand Paging

Page

A chunk of memory with its own record in the memory management hardware. Often 4kB.

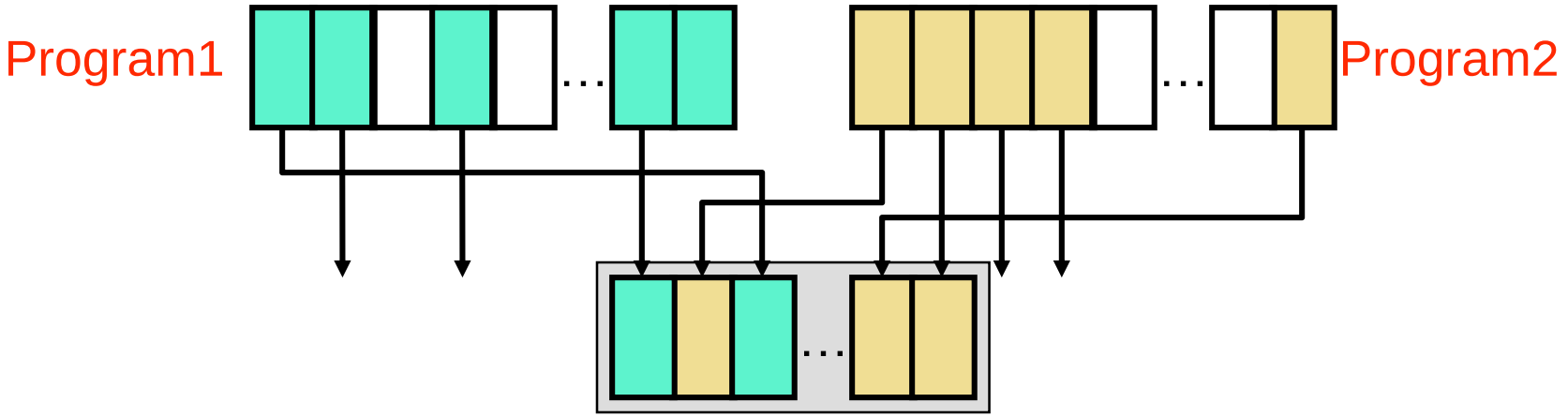
Virtual Memory



- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Itanium is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (M<N, especially if N=64)
  - 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)

Other Uses of Virtual Memory

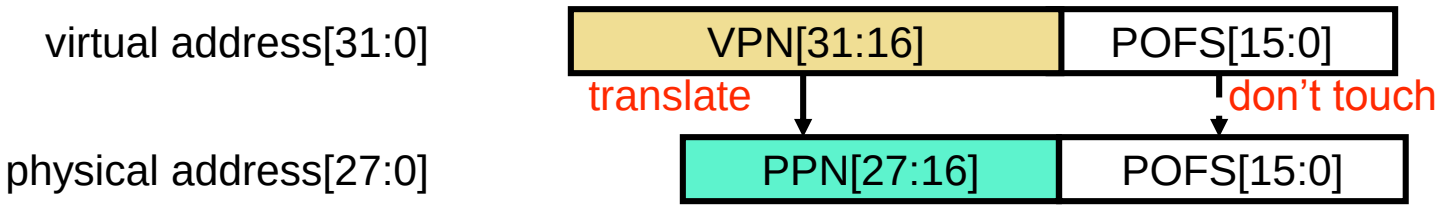
- Virtual memory is quite useful
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF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



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 → OS terminates program

Address Translation



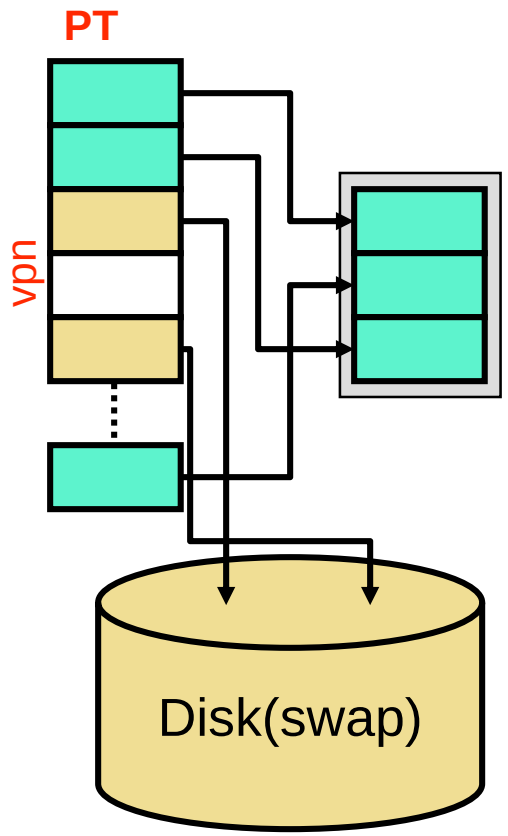
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** and page offset (POFS)
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated – why not?
  - VA→PA = [VPN, POFS] → [PPN, POFS]

- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN (16 = 32 – 16)
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

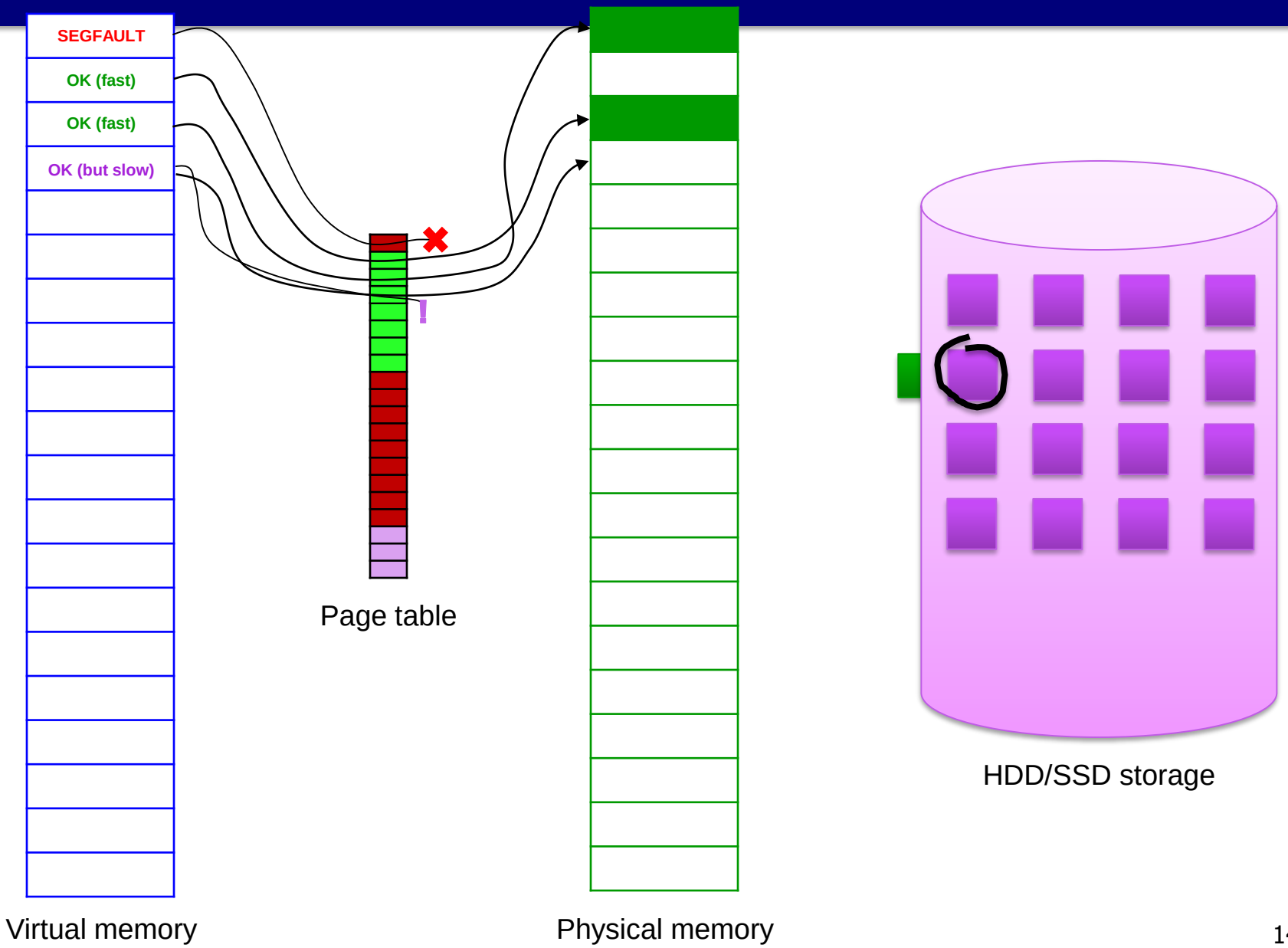
Mechanics of Address Translation

- 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

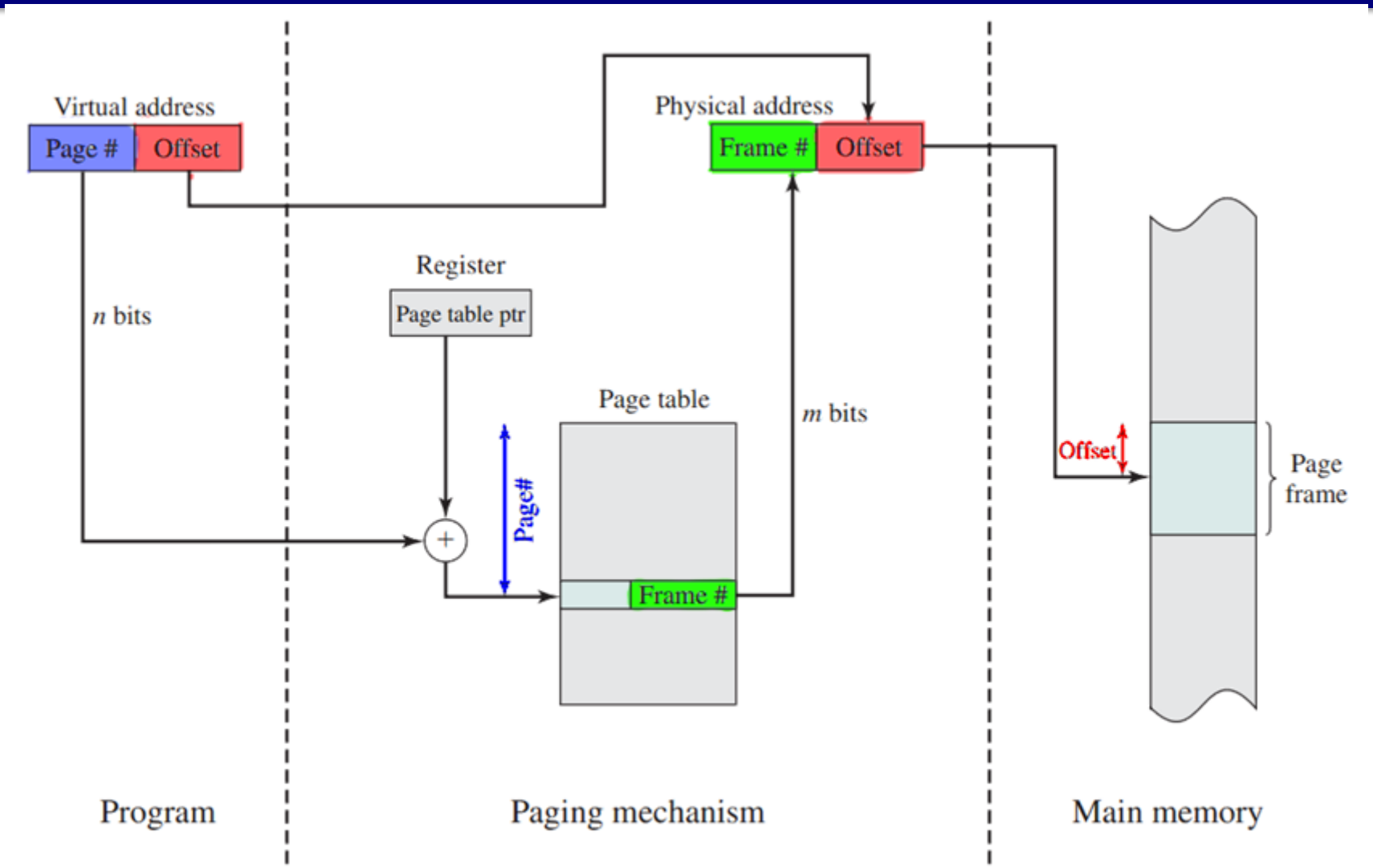
```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (pt[vpn].is_valid)
     return pt[vpn].ppn;
}
```



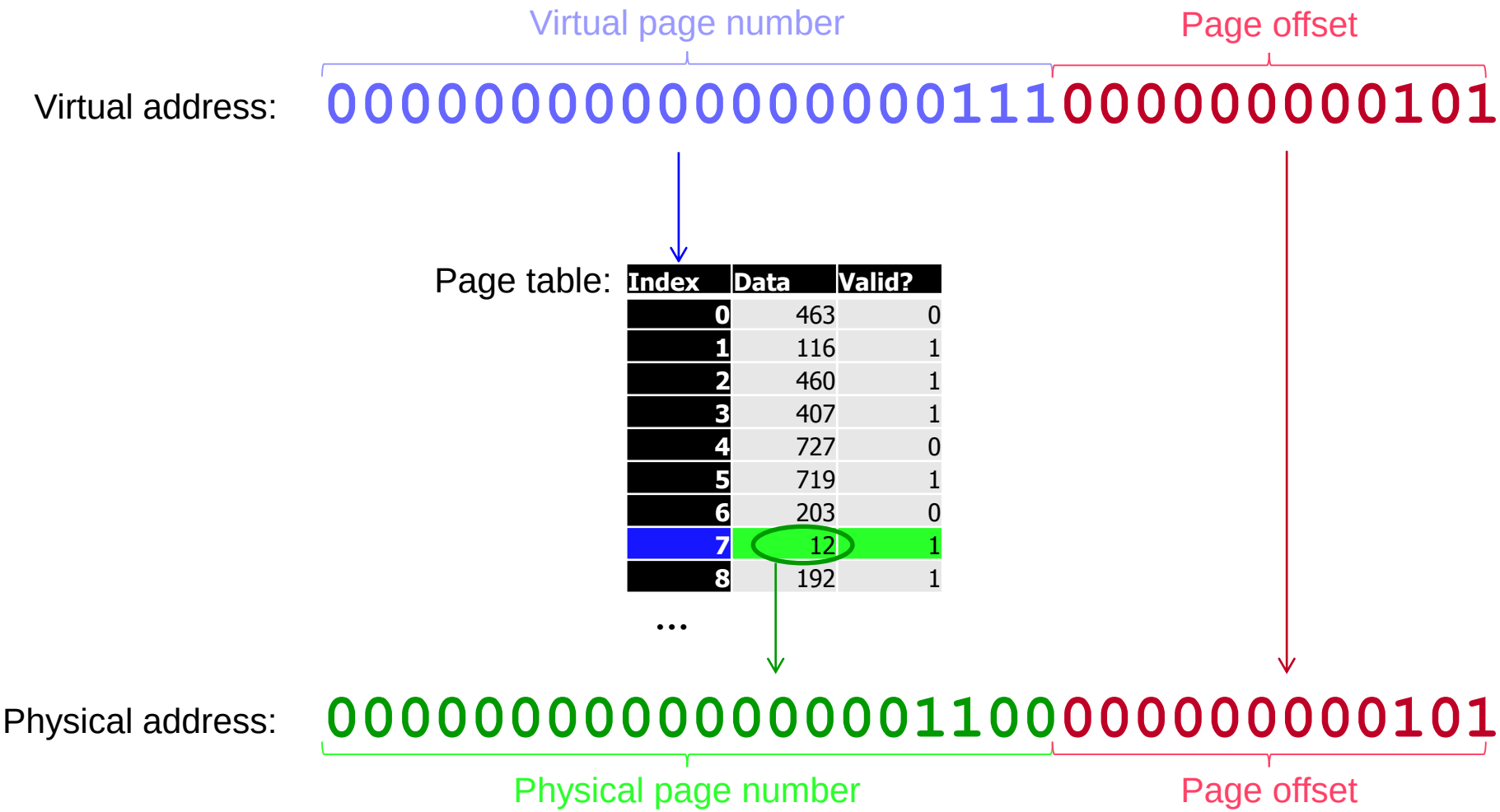
High level operation



Address translation

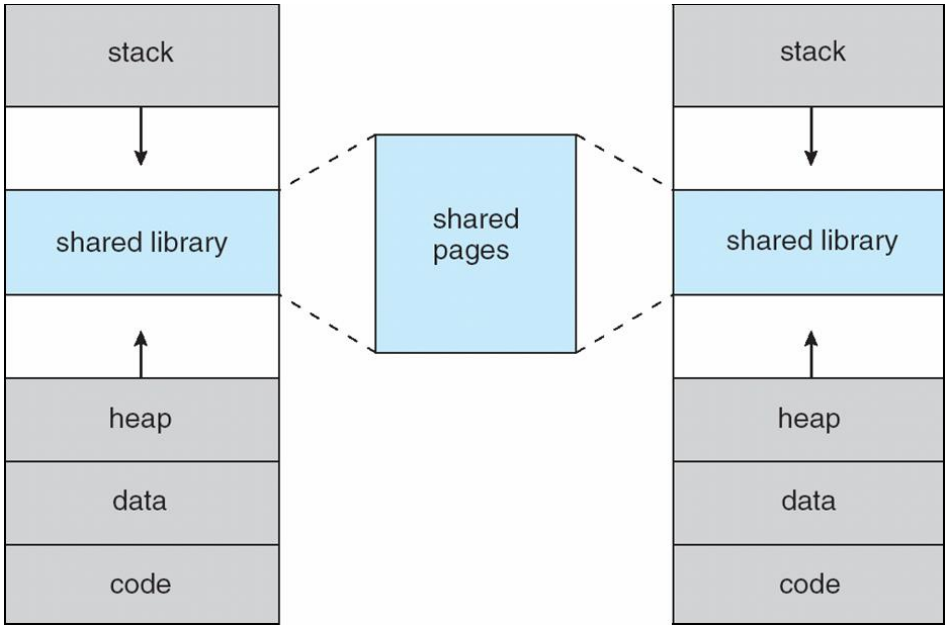


Address translation



Virtual address space

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- **System libraries** shared via mapping into virtual address space
- **Shared memory** by mapping pages read-write into virtual address space
  - Pages can be shared during `fork()`, speeding process creation



Structure of the page table

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 → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB page table
- How big would it be for a 64-bit machine?
- Page tables can get enormous
  - There are ways of making them smaller

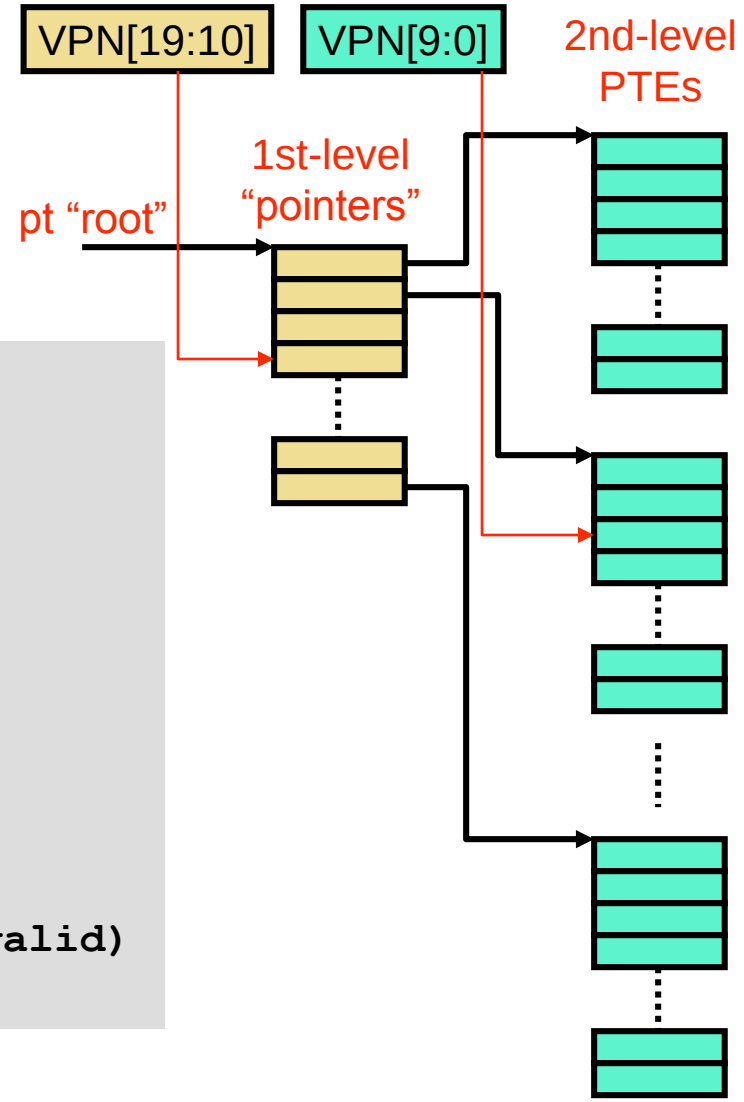
Multi-Level Page Table

- 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 → 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

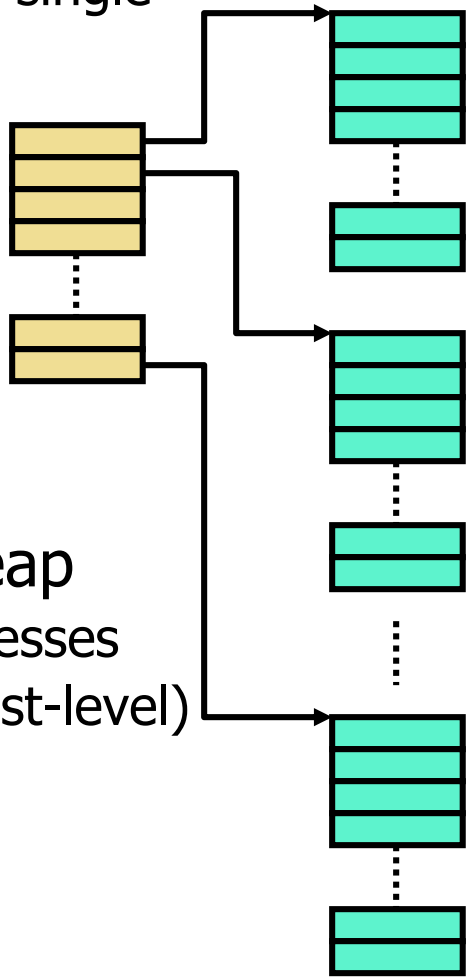
- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table




```
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

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

Translation Buffer (TB)

Still a problem

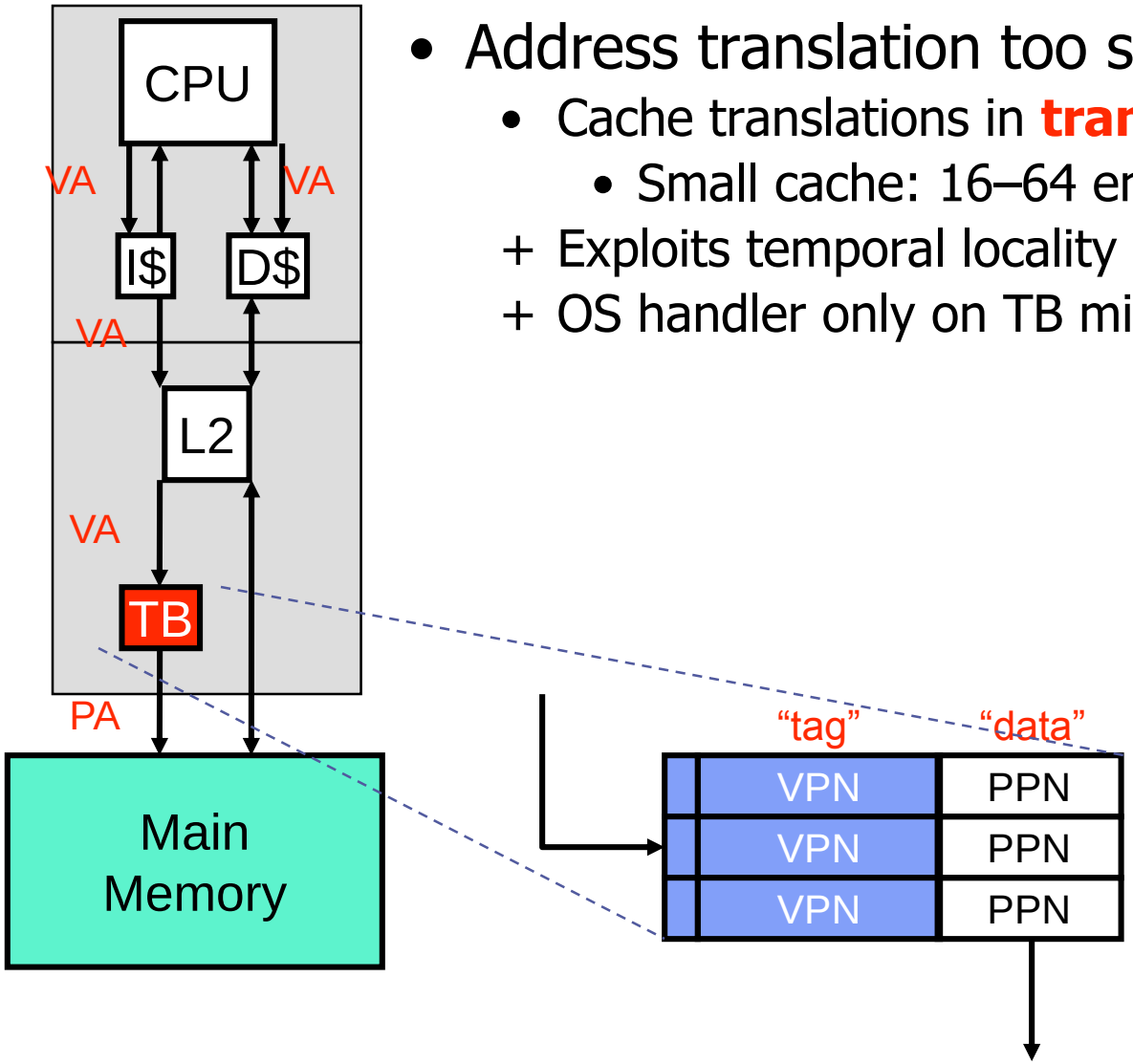
- OS handling of *every* L2 miss is too slow!

- We need to accelerate the **translation** of VA to PA

Answer: **translation buffer**

A special *cache* for the page table

Translation Buffer



- 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

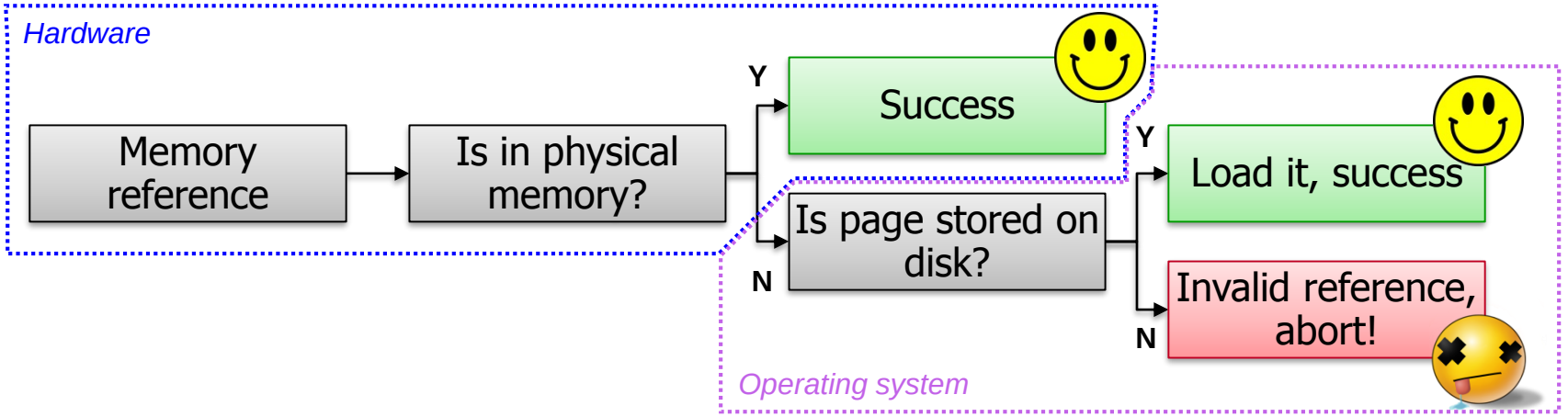
- **2) Hardware FSM**: does same thing (e.g., Intel x86)
  - Store PT root pointer in hardware register
  - Make PT root and 1st-level table pointers physical addresses
    - So FSM doesn't have to translate them
  + Latency: saves cost of OS call

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

Demand Paging



- The swapper process is called the **pager**

Summary

- Virtual memory
  - Page tables and address translation
  - Page faults and handling
  - Translation buffer