Virtual Memory

Xuhao Chen

cxh@mit.edu

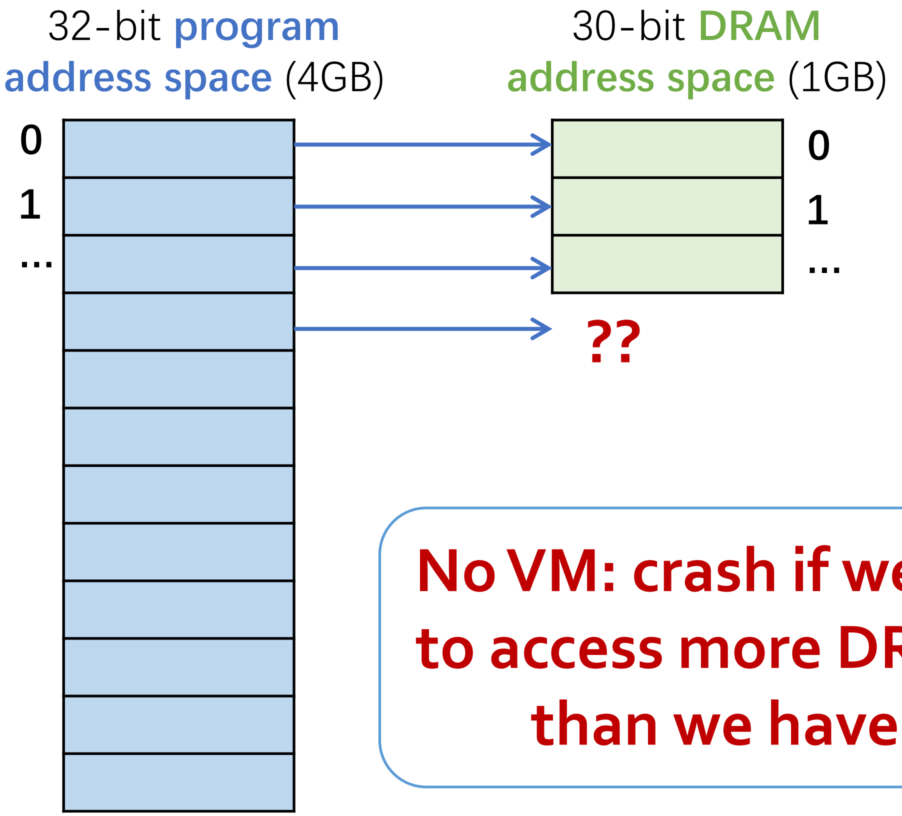
April 15, 2022

Virtual Memory is a layer of **indirection**

All problems in computer science can be solved by another level of indirection —— Butler Lampson

Without Virtual Memory

Program Address = DRAM address


32-bit program address space (4GB)
30-bit DRAM address space (1GB)
0
1
...
0
1
...
??

No VM: crash if we try to access more DRAM than we have

With Virtual Memory

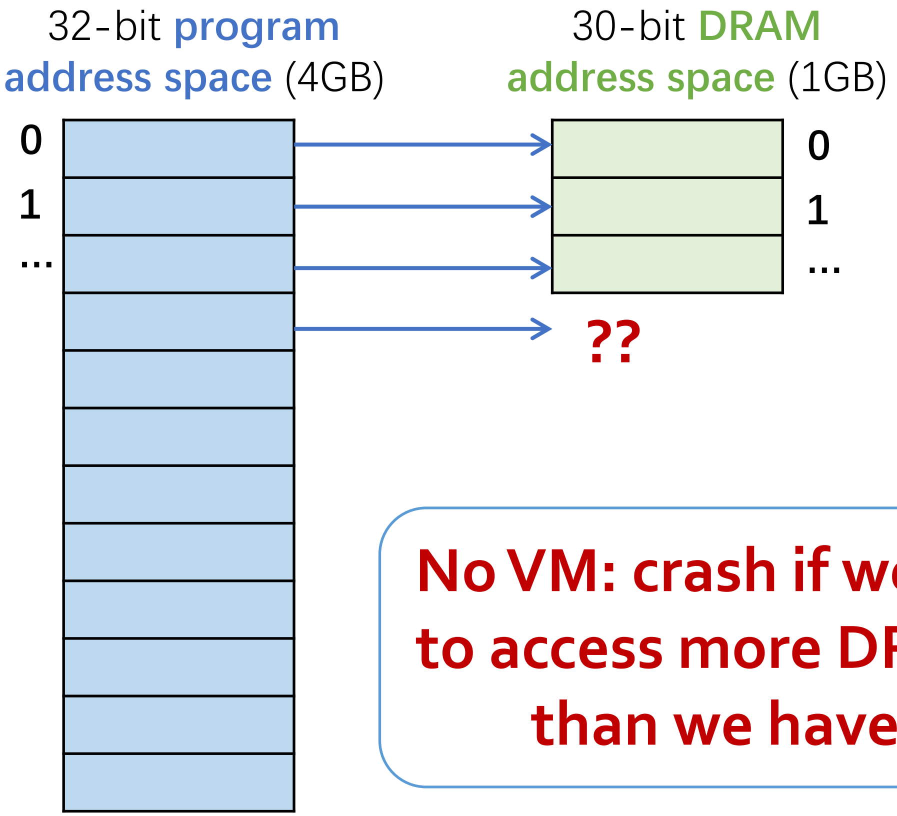
Program Address Maps to DRAM address

Virtual Memory is a layer of indirection

> **All problems in computer science can be solved by another level of indirection** —— Butler Lampson

Without Virtual Memory

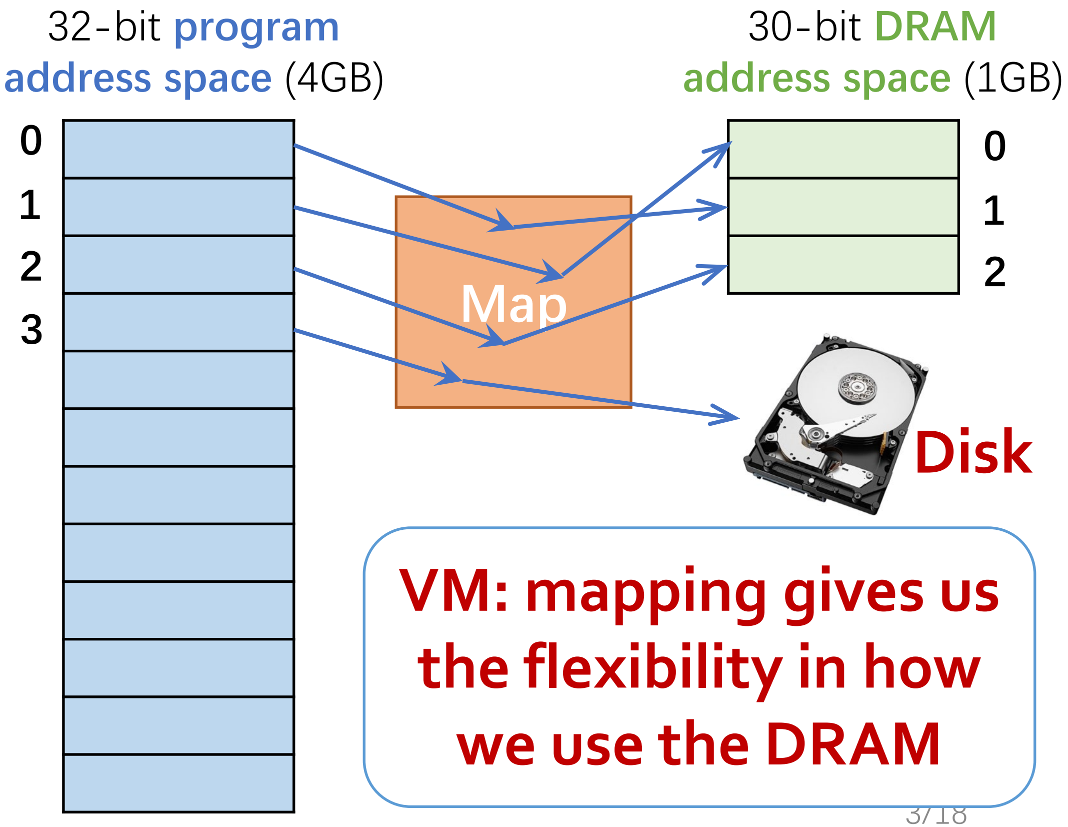
Program Address = DRAM address


32-bit program address space (4GB)
0
1
…
30-bit DRAM address space (1GB)
0
1
…
??

No VM: crash if we try to access more DRAM than we have

With Virtual Memory

Program Address Maps to DRAM address

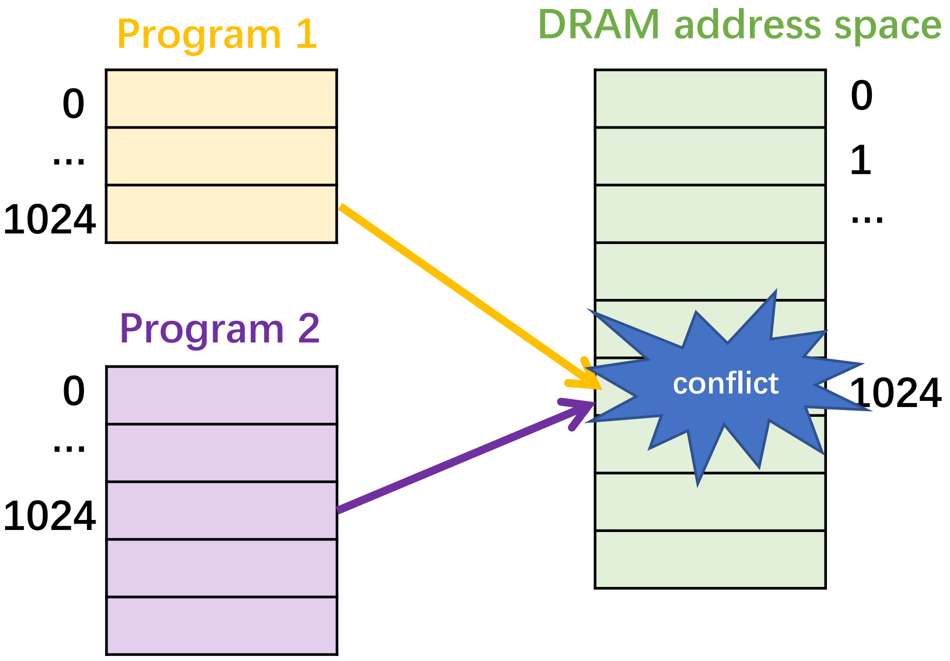

32-bit program address space (4GB)
0
1
2
3
Map
30-bit DRAM address space (1GB)
0
1
2
Disk

VM: mapping gives us the flexibility in how we use the DRAM

Virtual Memory is a layer of indirection

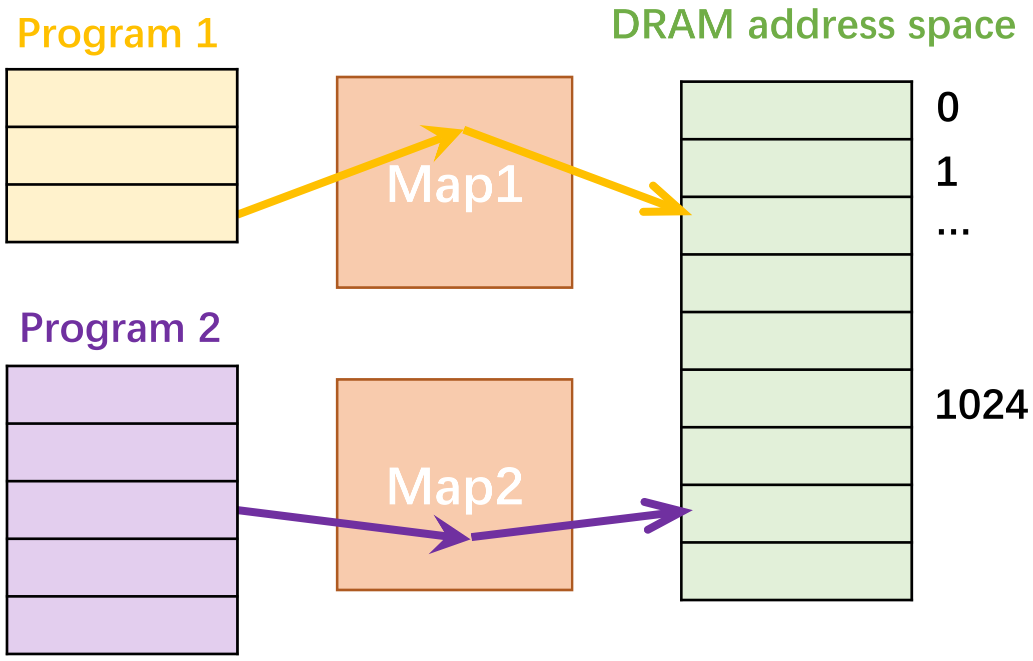
Without Virtual Memory

Program Address = DRAM address



With Virtual Memory

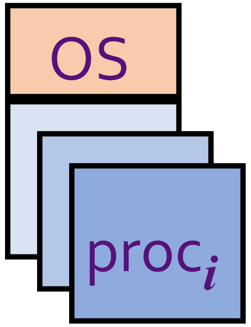
Program Address Maps to DRAM address



Program 1 stores your bank balance at address 1024
Program 2 stores your video game score at address 1024

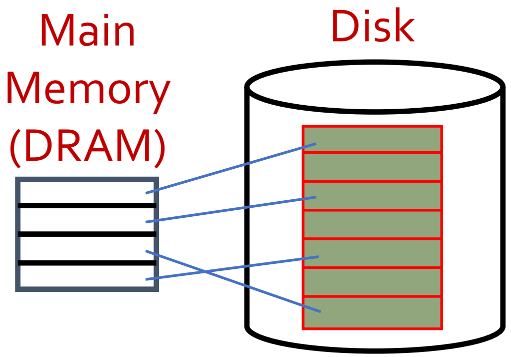
Virtual Memory: abstract storage resources

- **Protection & Privacy**
  - Each process has a **private** address space



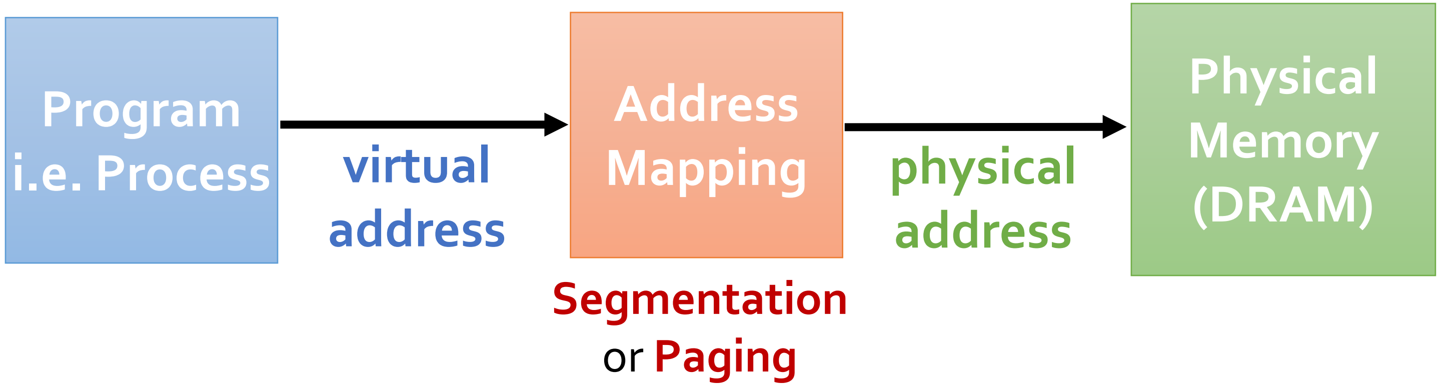
An illusion of a large, private, uniform store

- **Demand Paging**
  - Use DRAM as a **cache**/buffer of disk
  - Enables running programs **larger** than DRAM
  - DRAM holds a **portion** of the process's data



The price of Virtual Memory is
address translation on each memory reference

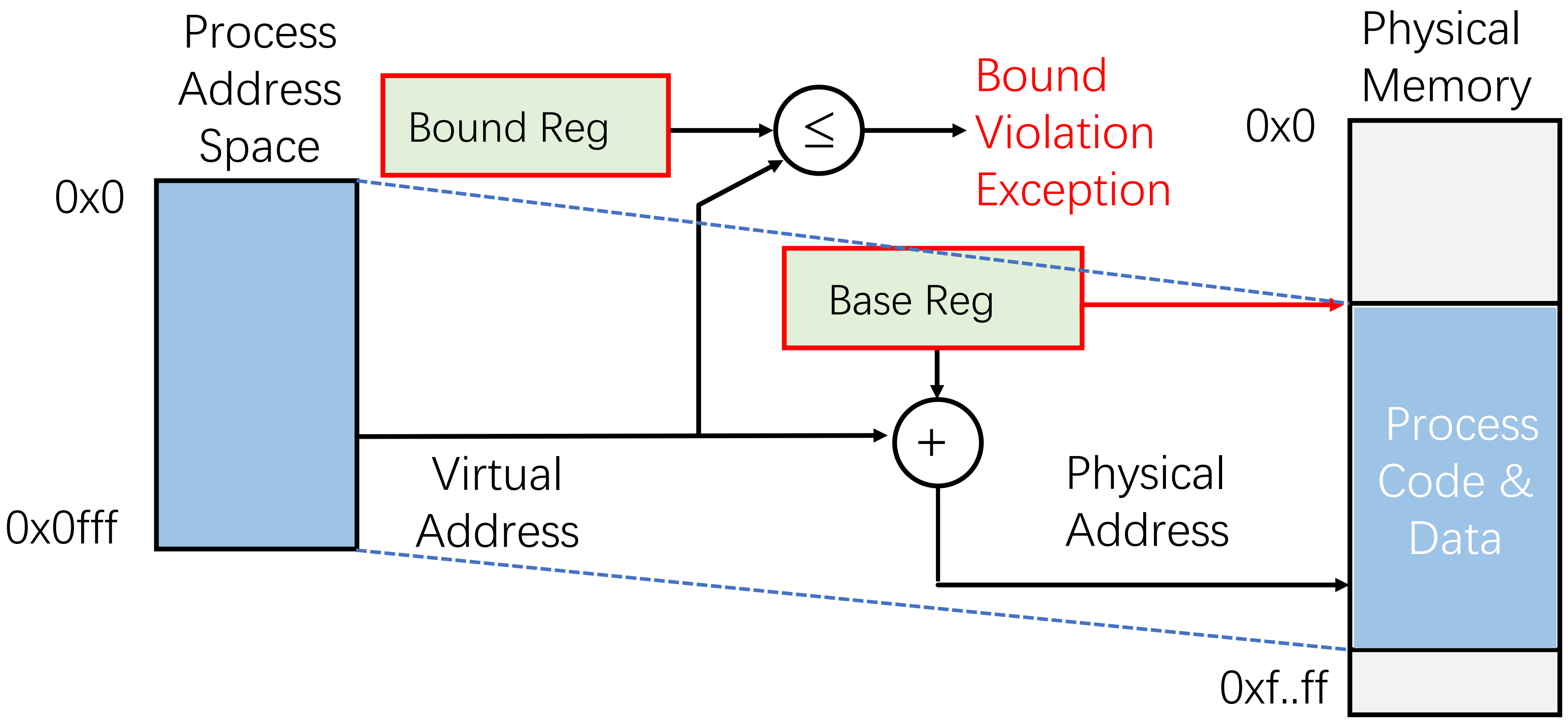
Virtual Address vs. Physical Address



- **Virtual address**
  - Address generated by the program (i.e., process)
  - Specific to the process's **private** address space

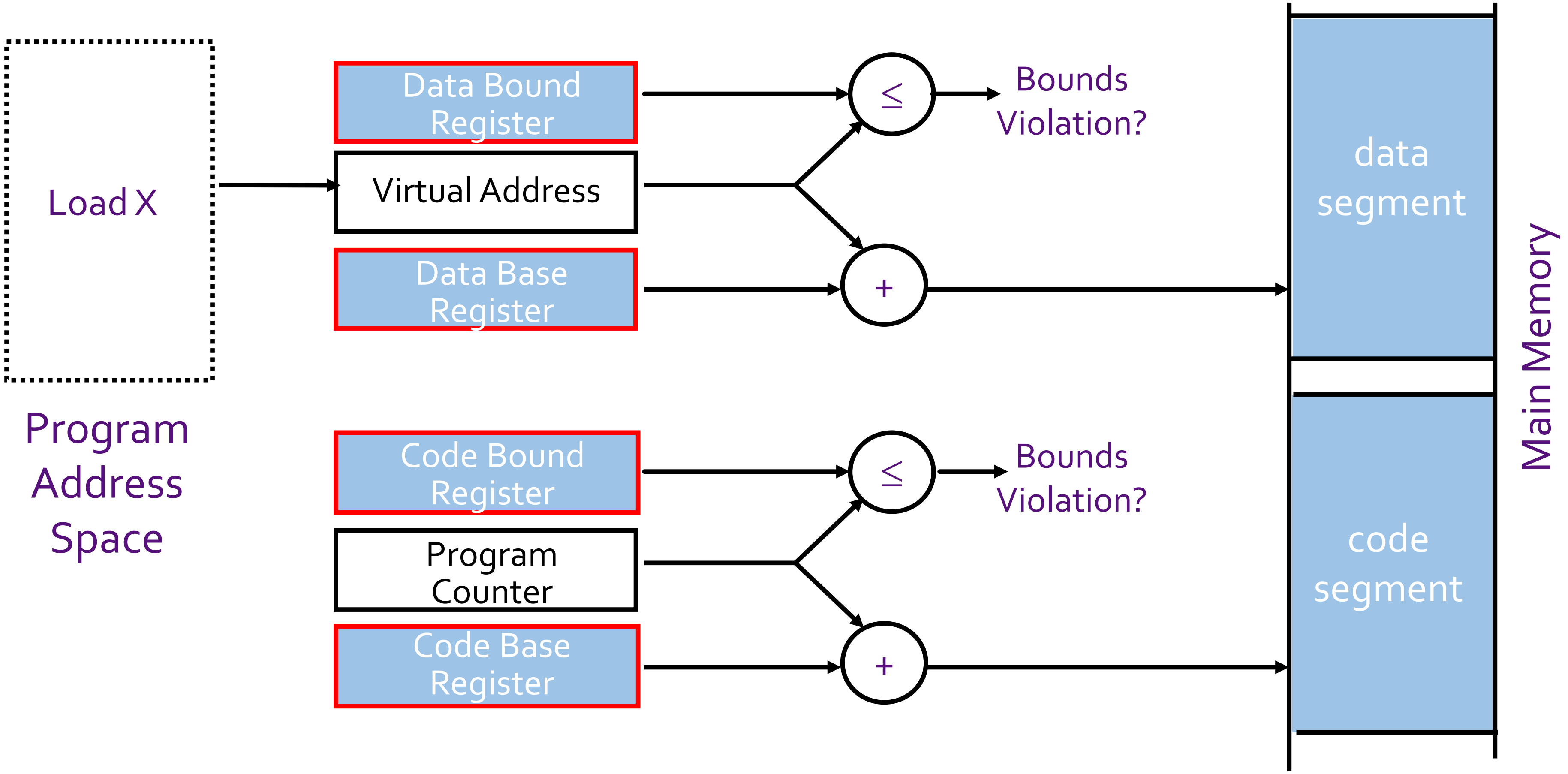
- **Physical address**
  - Address used to access physical (hardware) memory
  - **OS** manages address mapping

Segmentation: *Base-and-Bound* Address Translation



- Each program's data is allocated in a **contiguous** segment of physical memory
- Physical Address = Virtual Address + Segment Base
- Bound register provides safety and isolation
- Base and Bound registers are **NOT accessible** by user programs (only in supervisor mode)

Separate Segments for Code and Data

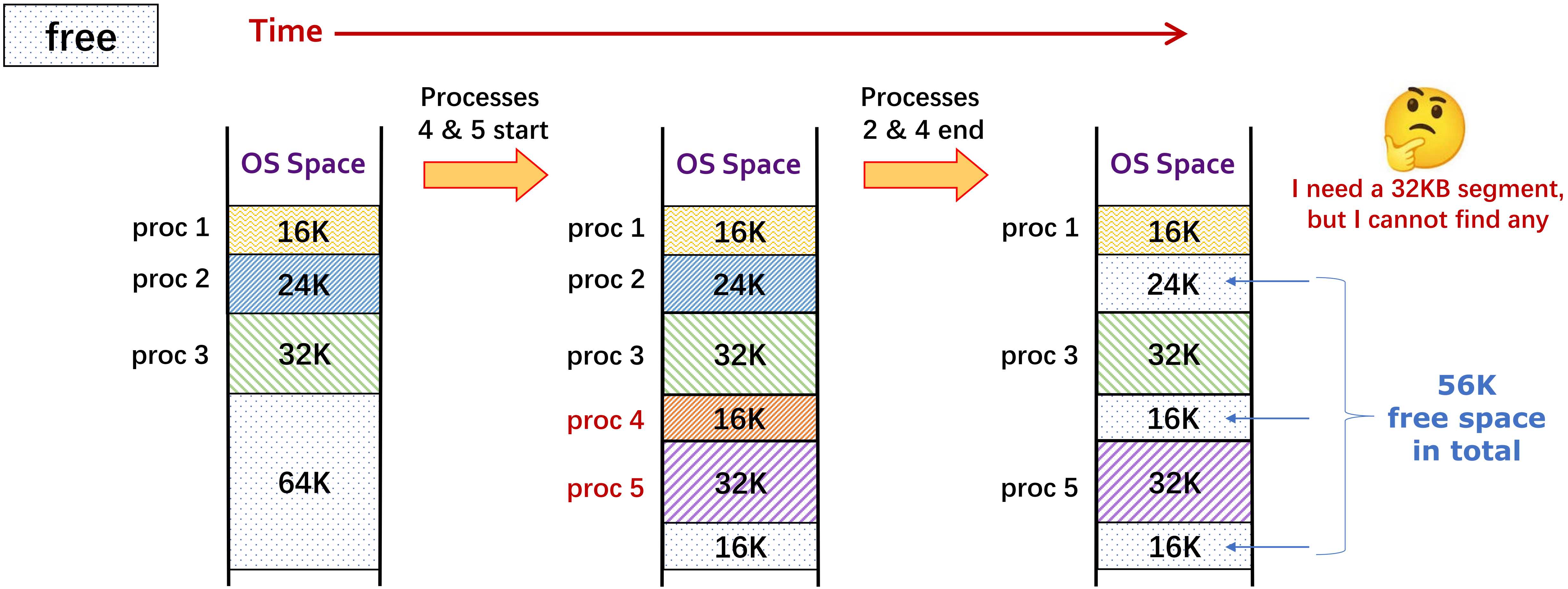


Pros of this separation?

(1) **Prevents buggy program from overwriting code**

(2) **Multiple processes can share code segment**

Memory Fragmentation



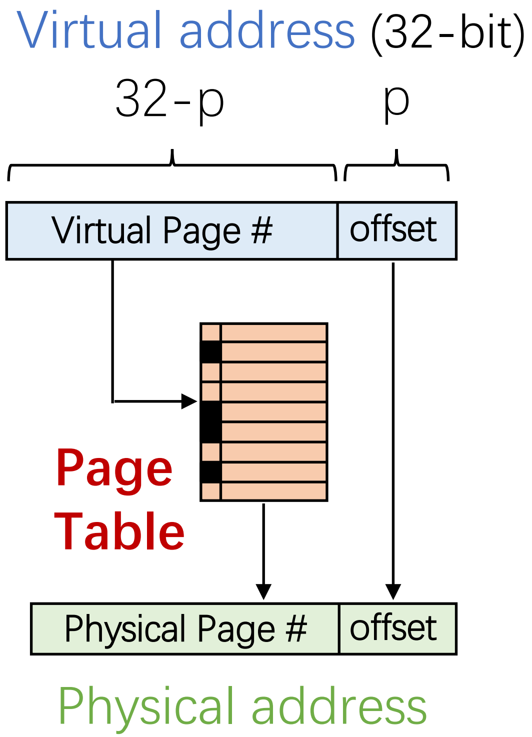
- As processes start and end, storage is “fragmented”
- Segments have to be moved around to compact the free space

Paged Memory Systems

- Divide physical memory in *fixed-size blocks* called pages
  - Typical page size: 4KB (2^{12} bytes)

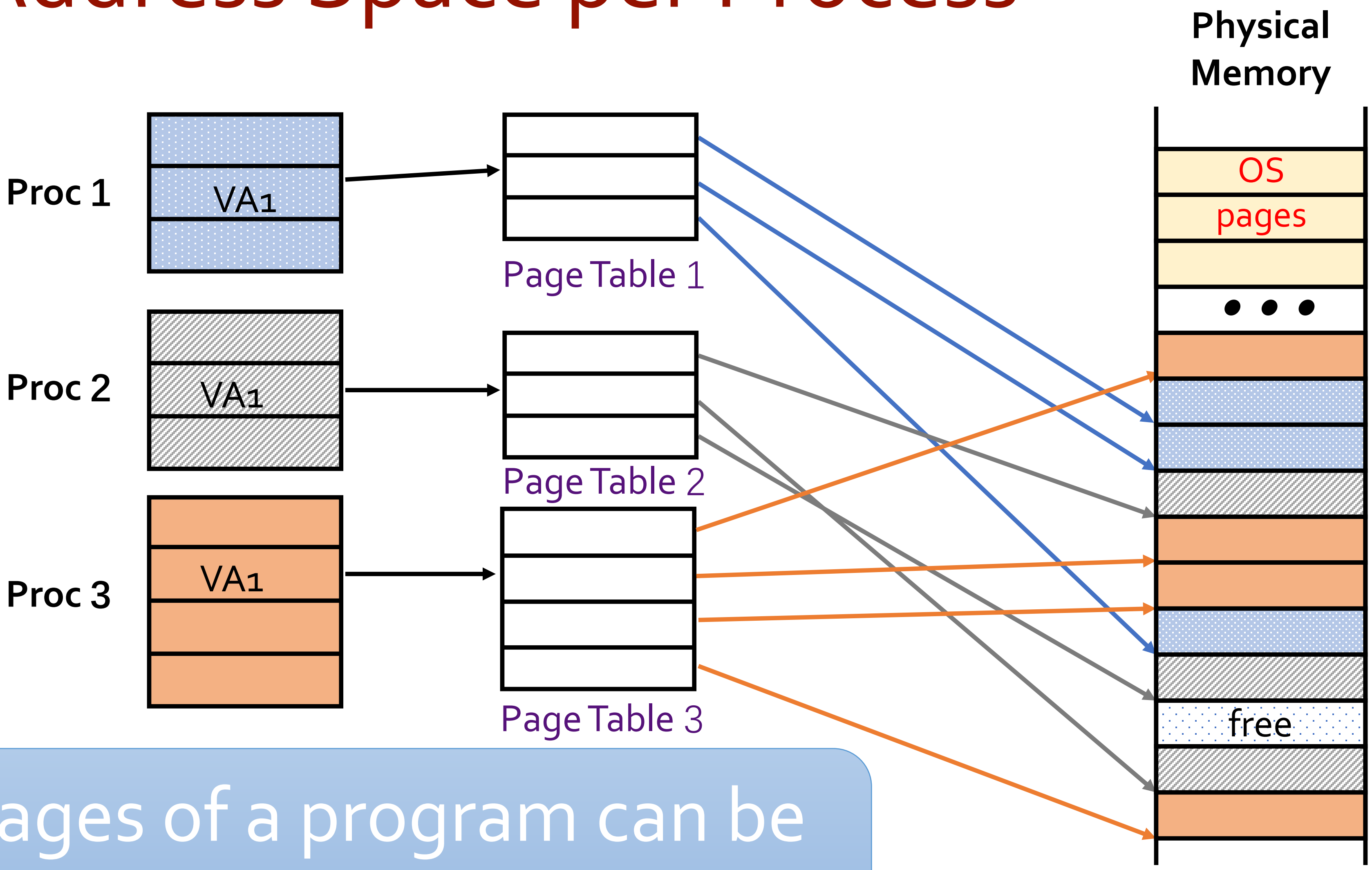
- Each virtual address is a pair:
  `<virtual page number, offset>`
  32-p bits p bits (p=12 for 4K page)

- **Page Table**: translate from virtual page # to physical page #



Pages of a program can be stored **non-contiguously**

Private Address Space per Process



Pages of a program can be stored **non-contiguously**

Paging vs. Segmentation

- Pros of paging
  - Paging avoids fragmentation
  - Paging enables demand paging
    - Allows programs to use more VM than the machine's PM

- What if all the pages can't fit in DRAM?

- Cons of paging
  - Page tables are much larger than base & bound registers

- Where to store the page tables?

Two Remaining Questions

- What if all the pages can’t fit in DRAM?

- Where to store the page tables?

Two Remaining Questions

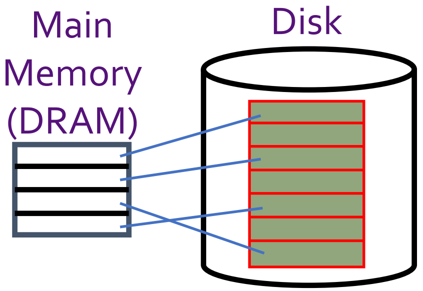
- What if all the pages can't fit in DRAM?

How to implement demand paging?

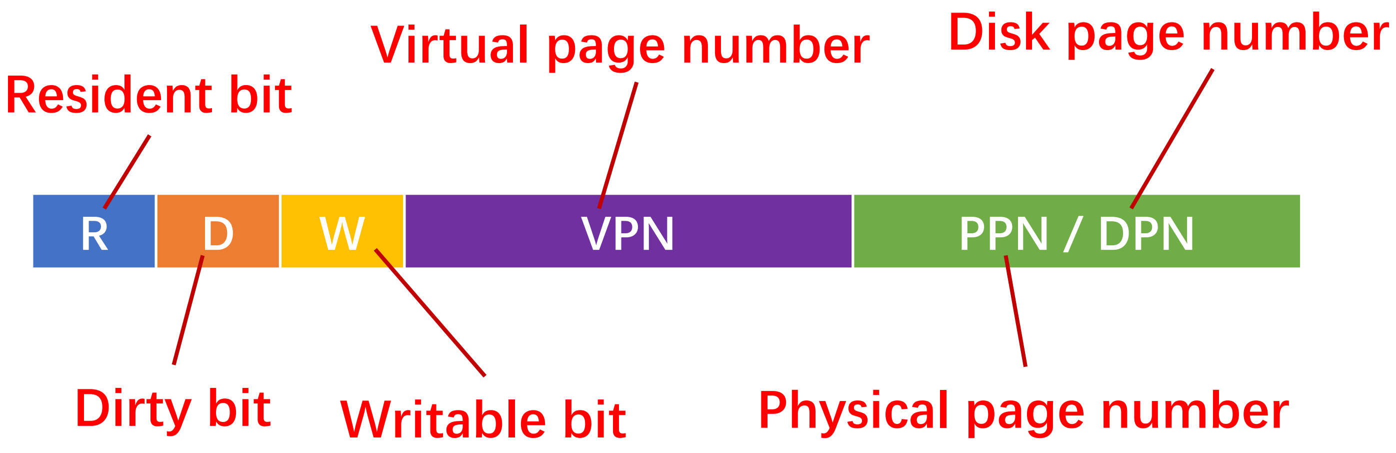
- Where to store the page tables?

Demand Paging: using DRAM as a cache of disk

- DRAM is backed up by ***swap space*** on disk



- Page Table Entry (PTE) contains:



Resident bit

Virtual page number

Disk page number

R | D | W | VPN | PPN / DPN

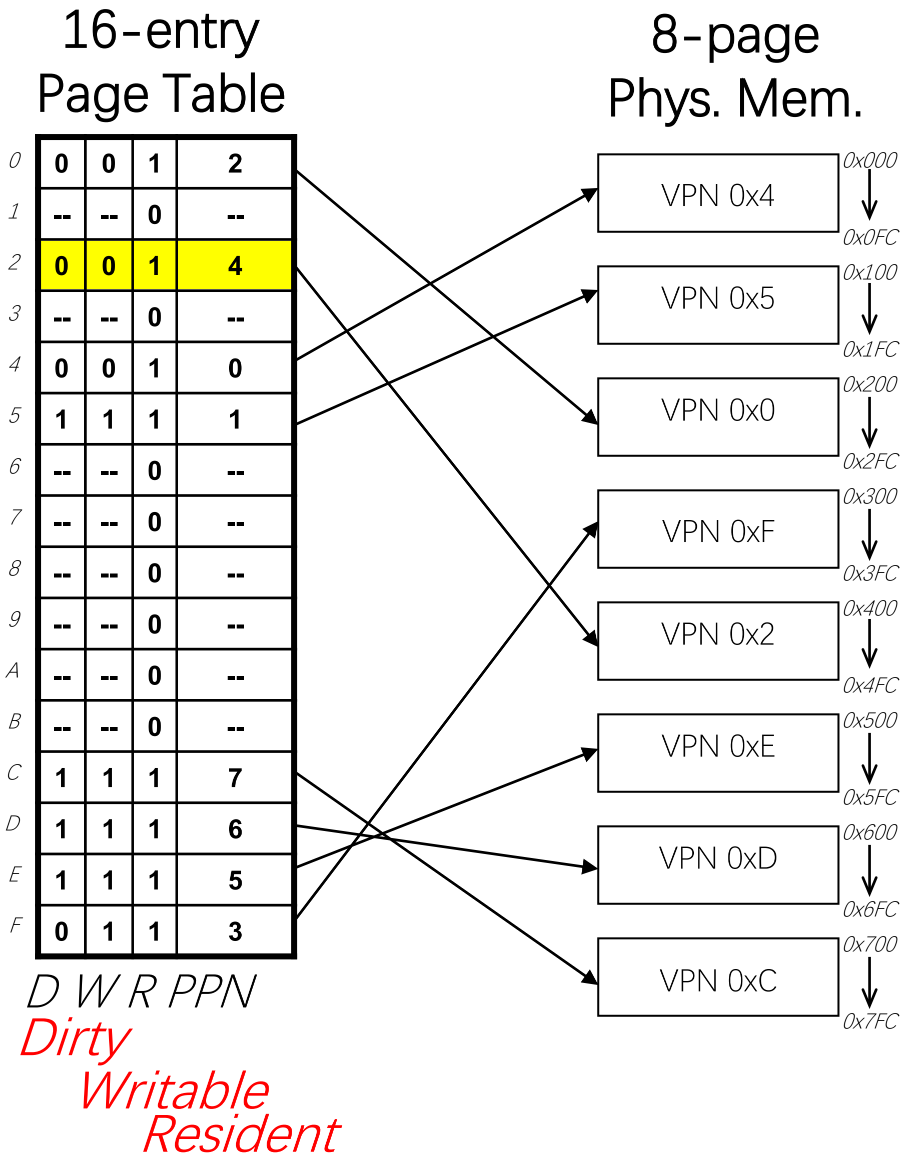
Dirty bit

Writable bit

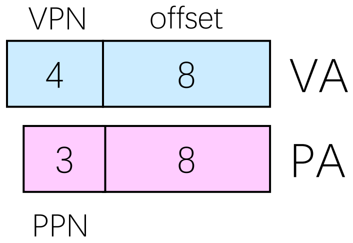
Physical page number

If R=1, PPN (physical page number) for a memory-resident page
If R=0, DPN (disk page number) for a page on the disk

Example: Virtual → Physical Translation



	D	W	R	PPN
0	0	0	1	2
1	--	--	0	--
2	0	0	1	4
3	--	--	0	--
4	0	0	1	0
5	1	1	1	1
6	--	--	0	--
7	--	--	0	--
8	--	--	0	--
9	--	--	0	--
A	--	--	0	--
B	--	--	0	--
C	1	1	1	7
D	1	1	1	6
E	1	1	1	5
F	0	1	1	3



Setup:

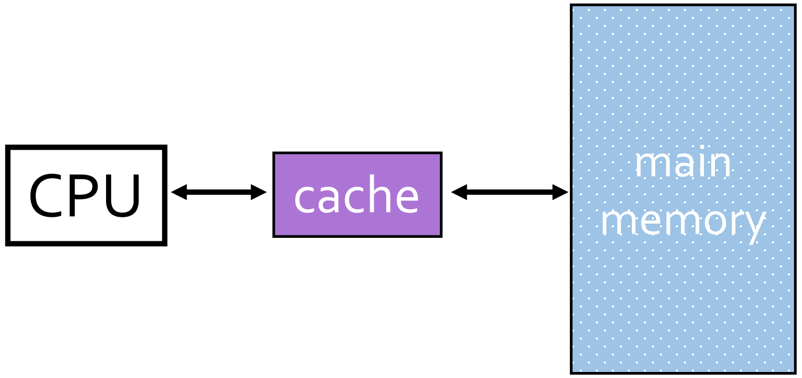
- 256 bytes/page (2^8)
- 16 virtual pages (2^4)
- 8 physical pages (2^3)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)

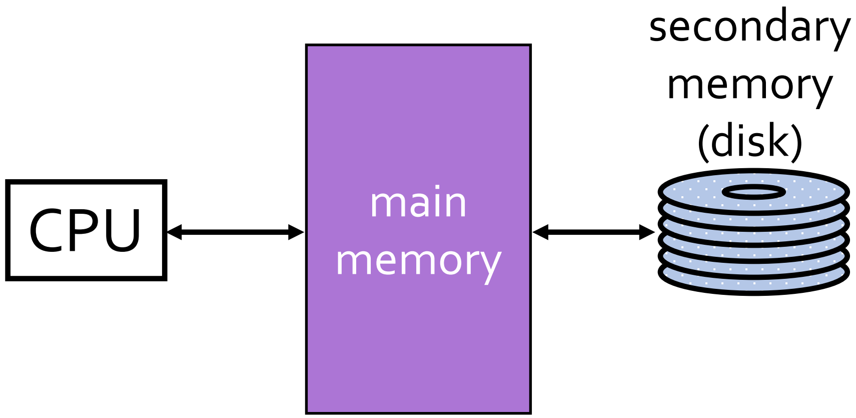
```
lw 0x2C8(x0)
```

VA = 0x2C8, PA = **0x4C8**

VPN = 0x2
→ PPN = 0x4

Caching vs. Demand Paging





Caching

cache entry
cache block (~32 bytes)
cache miss rate (1% to 20%)
cache hit (~1 cycle)
cache miss (~100 cycles)
a miss is handled in *hardware*

Demand paging

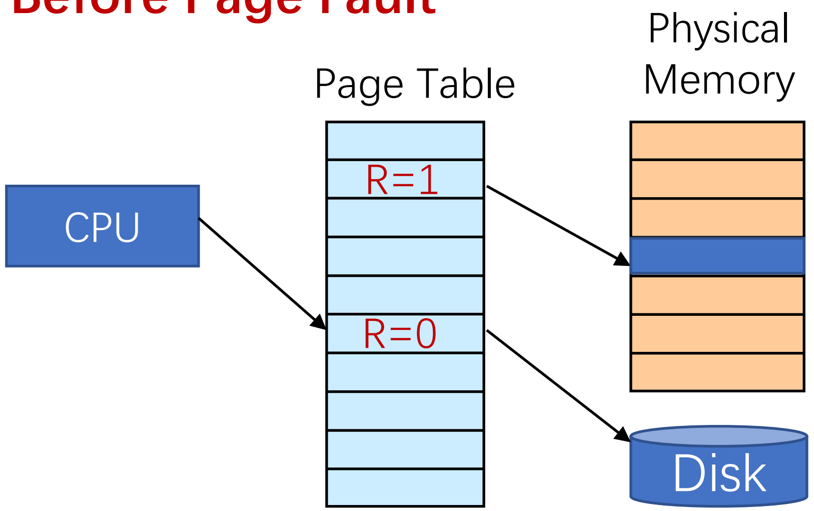
page frame
page (~4K bytes)
page miss rate (<0.001%)
page hit (~100 cycles)
page miss (~5M cycles)
a miss is handled mostly in *software*

Page Faults

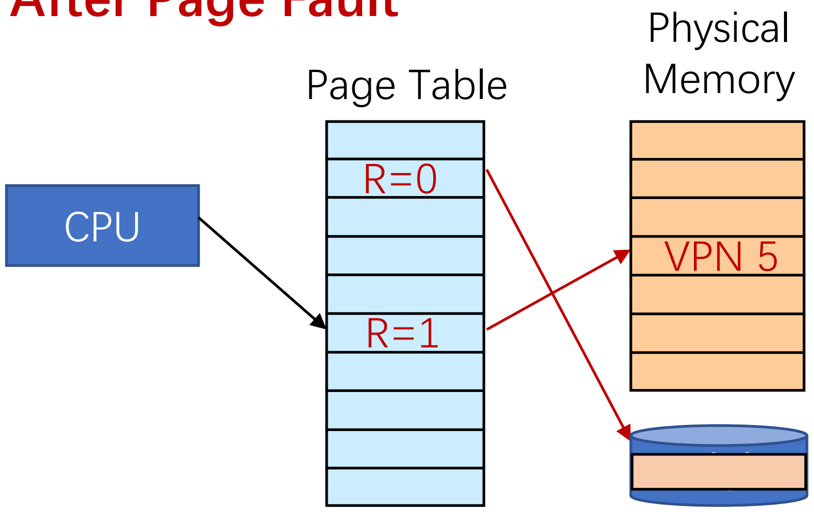
- An access to a page with R=0 causes a page fault exception

- OS page fault handler is invoked:
  - Choose a page to replace, write it back if dirty. Mark page as non-resident
  - Read page from disk into available physical page
  - Update page table to show new page is resident
  - Return control to program, which re-executes memory access

Before Page Fault



After Page Fault

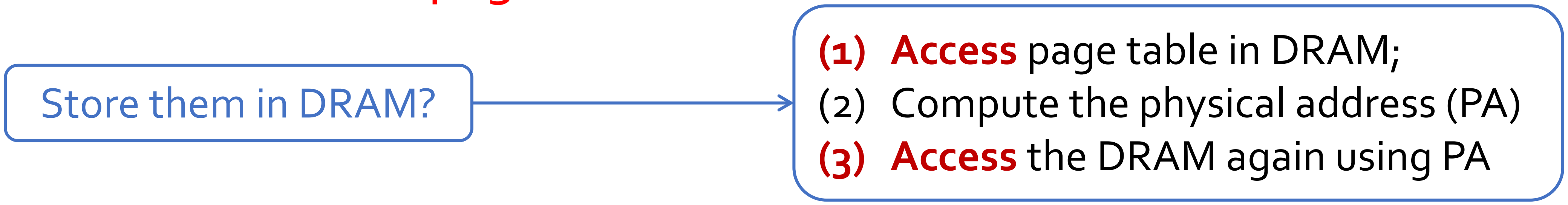


Two Remaining Questions

- What if all the pages can't fit in DRAM?

→ OS handles the page fault

- Where to store the page tables?

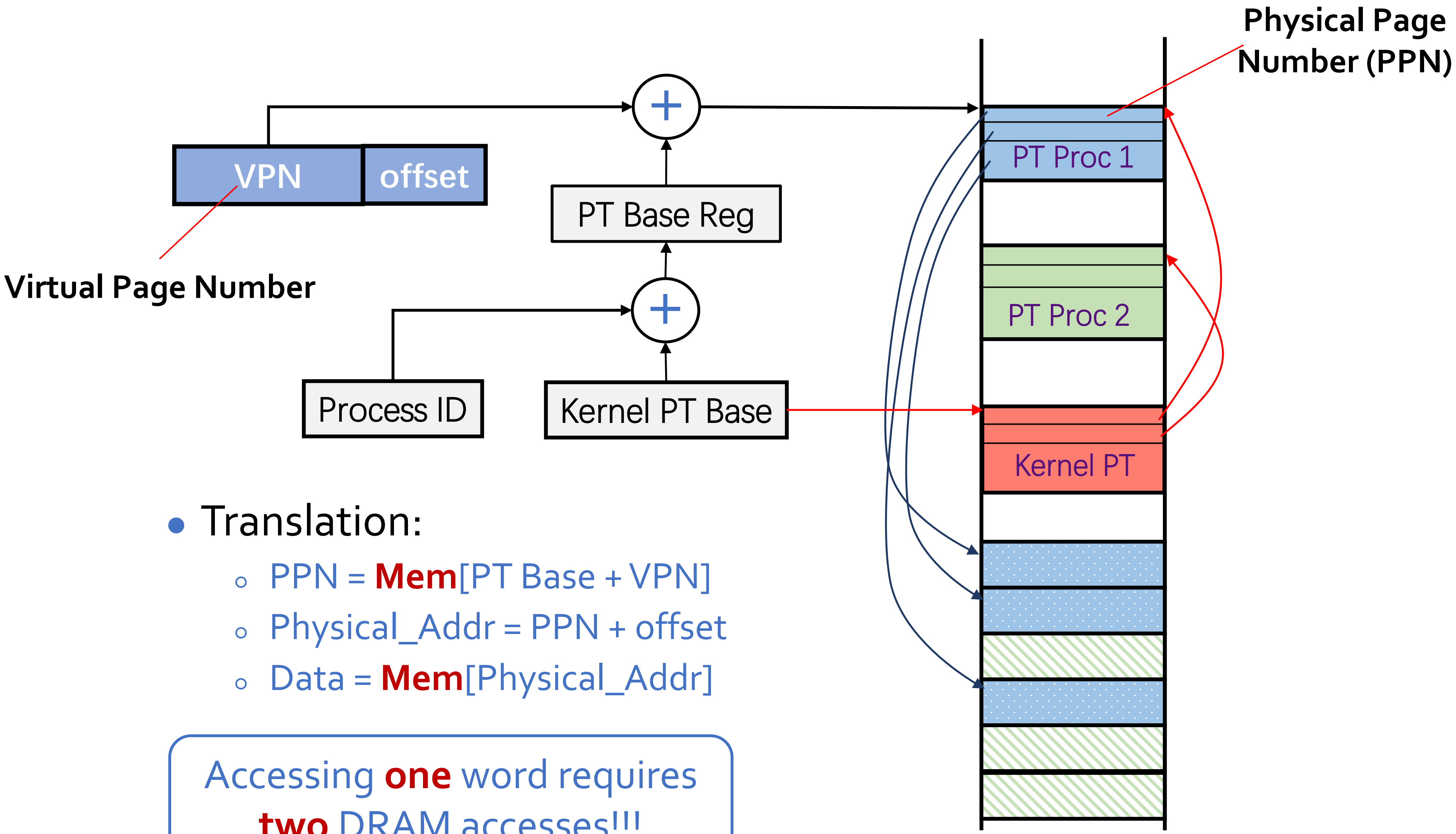


Store them in DRAM?

(1) **Access** page table in DRAM;
(2) Compute the physical address (PA)
(3) **Access** the DRAM again using PA

Accessing **one** word data
requires **two** DRAM accesses!!!

Suppose Page Tables reside in Memory



- Translation:
  - PPN = **Mem**[PT Base + VPN]
  - Physical_Addr = PPN + offset
  - Data = **Mem**[Physical_Addr]

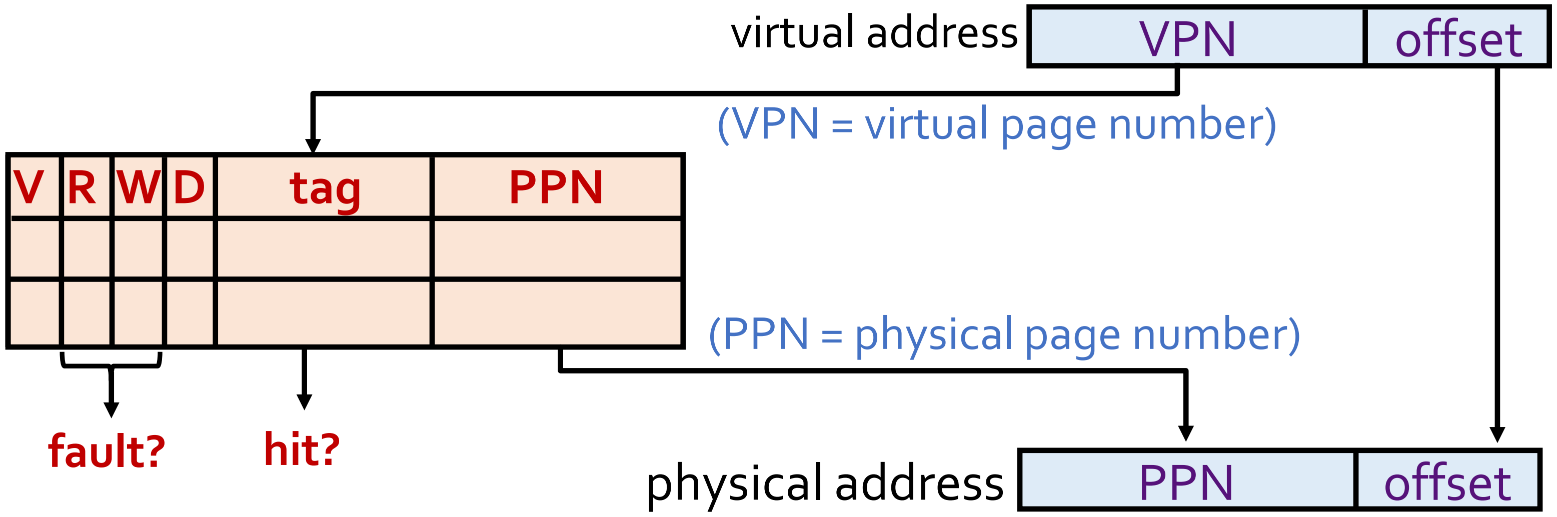
Accessing **one** word requires **two** DRAM accesses!!!

Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive!
Each reference requires accessing page table

Solution: **Cache translations in TLB**

TLB hit → *Single-cycle translation*
TLB miss → *Access page table to refill TLB*



Example: TLB and Page Table

Suppose

- Virtual memory of 2^{32} bytes
- Physical memory of 2^{24} bytes
- Page size is 2^{10} (1 K) bytes
- 4-entry fully associative TLB

TLB

Tag	Data			
VPN	V	R	D	PPN
0	1	0	0	7
6	1	1	1	2
1	1	1	1	9
3	1	0	0	5

Page Table

VPN	R	D	PPN
0	0	0	7
1	1	1	9
2	1	0	0
3	0	0	5
4	1	0	5
5	0	0	3
6	1	1	2
7	1	0	4
8	1	0	1
	…		

1. How many pages can be stored in physical memory at once?
2. How many entries are there in the page table?
3. How many bits per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit)
4. How many pages does page table take?

6. What is the physical address for virtual address 0x1804? What components are involved in the translation?

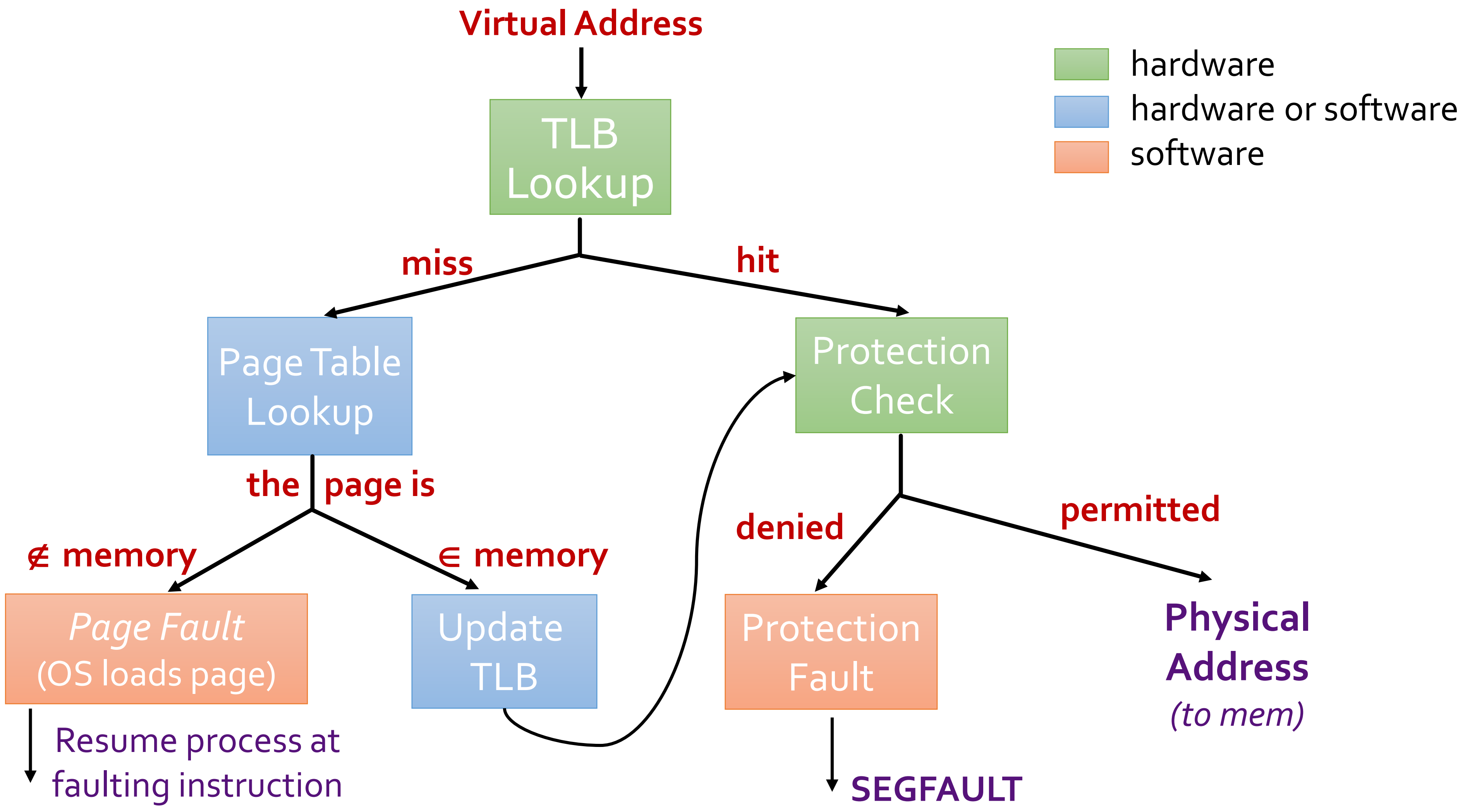
7. Same for 0x1080

8. Same for 0x0FC

TLB Designs

- Typically 32 to 128 entries, 4 to 8-way set-associative
  - Modern processors use a hierarchy of TLBs (e.g., 128-entry L1 TLB + 2K-entry L2 TLB)
- Switching processes is expensive because TLB has to be flushed
  - Alternatively, include process ID in TLB entries to avoid flushing
- Handling a TLB miss: Look up the page table (a.k.a. "walk" the page table)
  - If the page is in memory, load the entry into the TLB. Otherwise, cause a page fault
- Page faults are always handled in software
- Page table walks are usually handled in hardware → memory management unit (MMU)
  - RISC-V, x86 access page table in hardware

Address Translation: *Putting it all together*



Virtual Address

TLB Lookup

miss

hit

Page Table Lookup

Protection Check

the page is

∉ memory

∈ memory

denied

permitted

Page Fault (OS loads page)

Update TLB

Protection Fault

Physical Address *(to mem)*

Resume process at faulting instruction

SEGFAULT

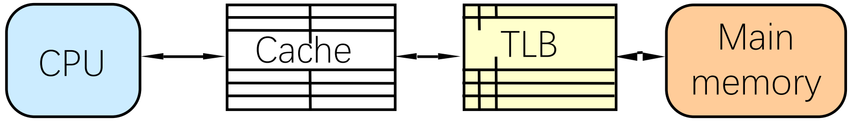
🟩 hardware

🟦 hardware or software

🟧 software

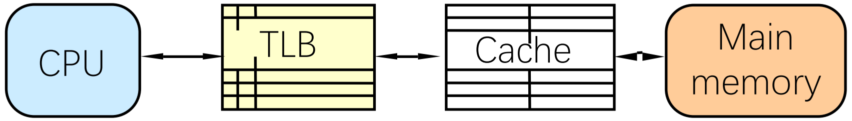
Using Caches with Virtual Memory

Virtually-Addressed
Cache



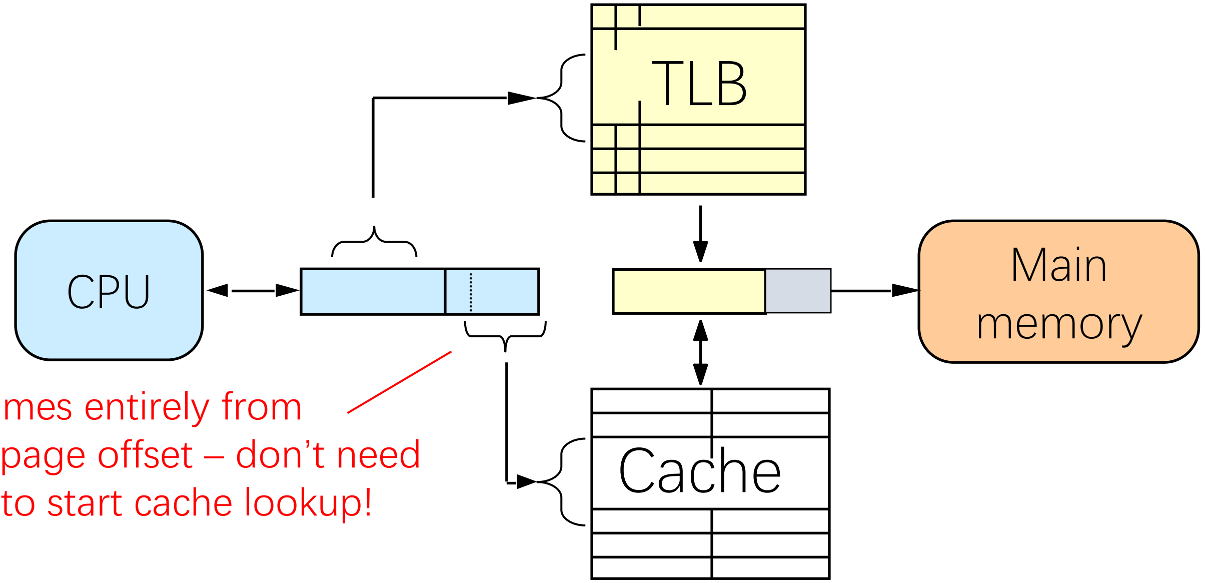
- FAST: No virtual → physical translation on cache hits
- Problem: Must flush cache after context switch

Physically-Addressed
Cache



- Benefit: Avoids stale cache data after context switch
- SLOW: Virtual → physical translation before every cache access

Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)



Cache index comes entirely from address bits in page offset – don't need to wait for TLB to start cache lookup!

OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done in parallel with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!

Summary

Thank You!

- **Virtual memory** is a layer of **indirection**
  - Protection and Privacy: private address space per process
  - Demand Paging: use main memory as a cache of disk

- **Segmentation**: each address space is a **contiguous** block (i.e., a segment)
  - Simple: Base and bound registers
  - Suffers from fragmentation, no demand paging

- **Paging**: divided into fixed-size pages. A **page table** maps virtual to physical pages
  - Avoids fragmentation
  - Enables demand paging: pages can be in main memory or disk
  - Requires an extra page table access on each memory reference

- **TLB** makes paging efficient by caching the page table