

# CS 110

# Computer Architecture

## OS 2

Instructors:  
**Chundong Wang & Siting Liu**

<https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html>

School of Information Science and Technology SIST

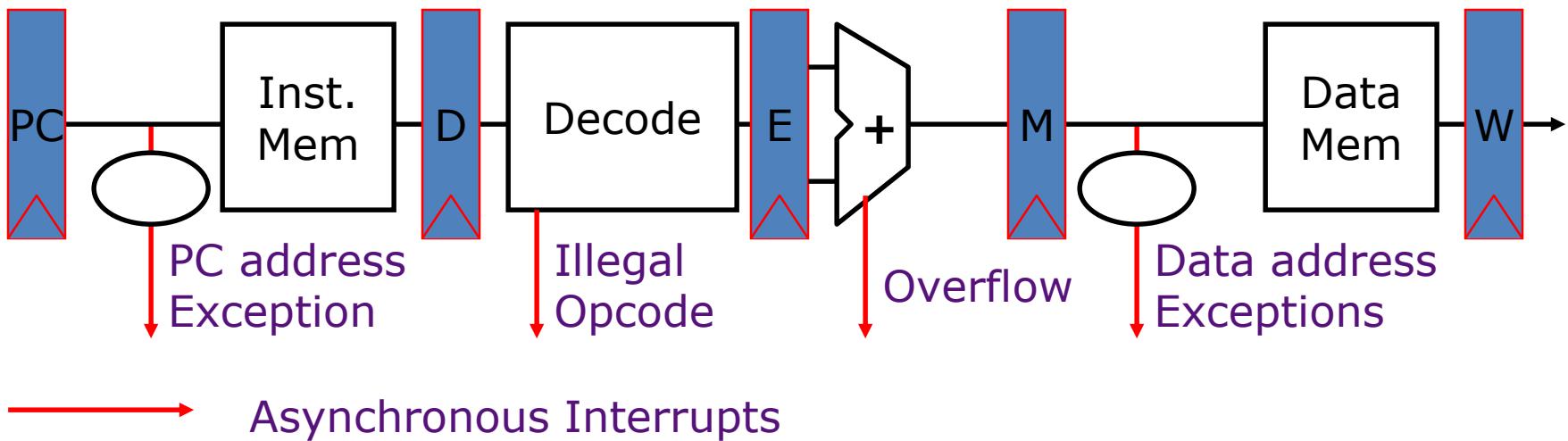
ShanghaiTech University

Slides based on UC Berkeley's CS61C

# Review

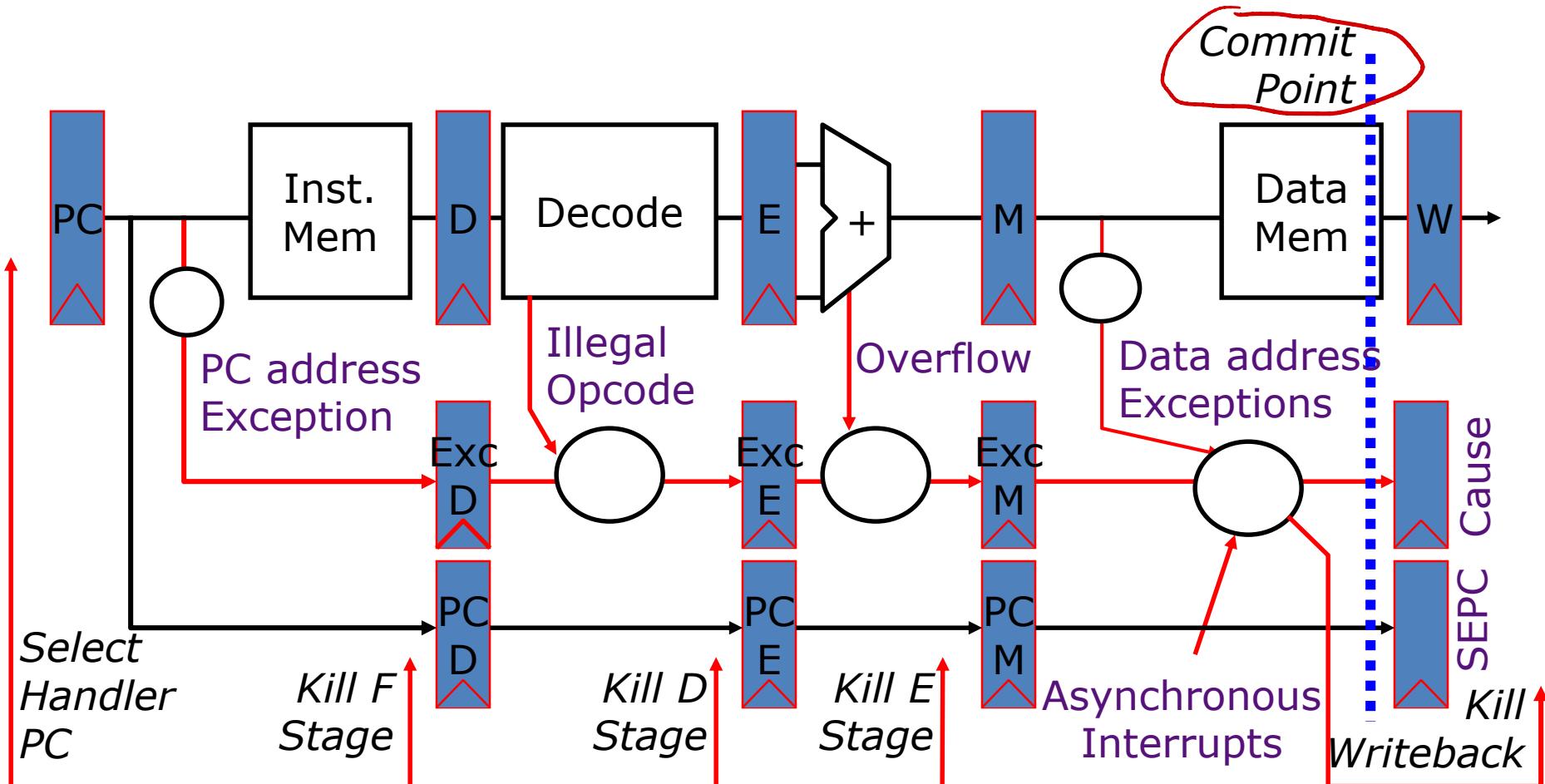
- Booting a Computer
  - BIOS, Bootloader, OS Boot, Init
- Supervisor Mode, Syscalls
- Memory-mapped I/O
- Polling vs. Interrupts
- Interrupt vs. exception, and pipeline

# Trap Handling in 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

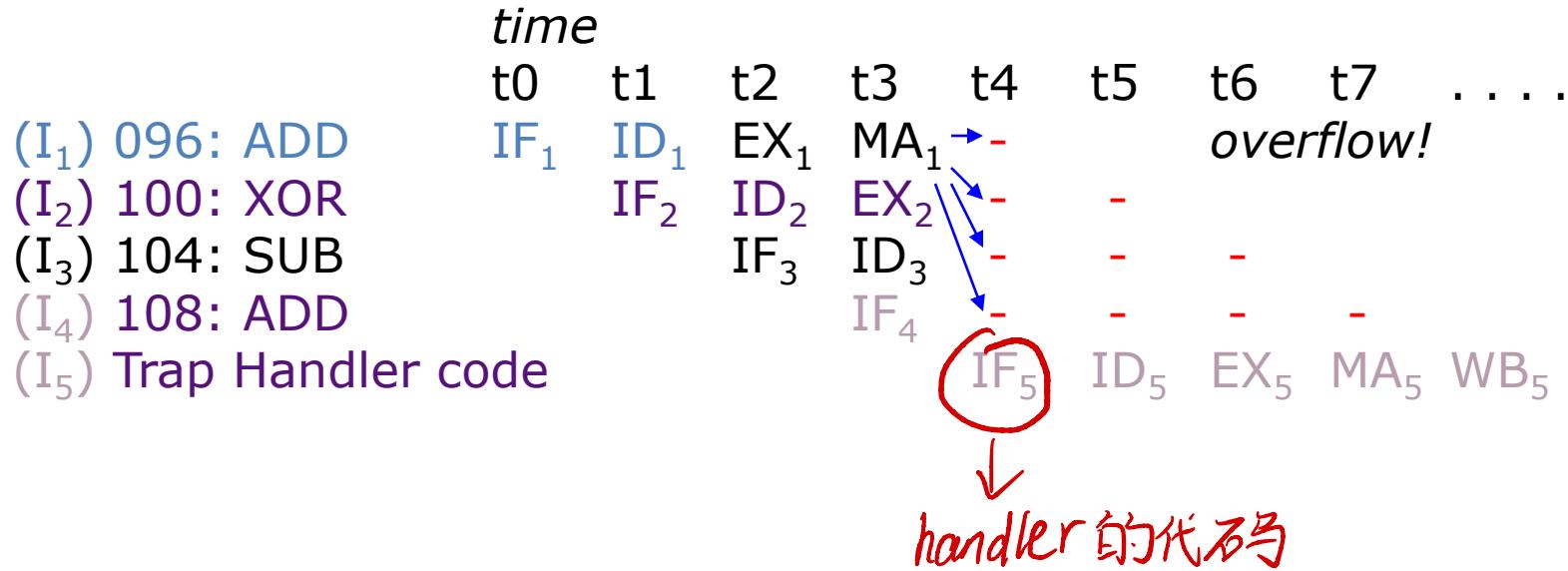
# Save Exceptions Until Commit



# Handling Traps in In-Order Pipeline

- Hold exception flags in pipeline until **commit point** (M stage)
- Exceptions in earlier instructions override exceptions in later instructions
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point
- If exception/interrupt at commit: update Cause and SEPC registers, kill all stages, inject handler PC into fetch stage

# Trap Pipeline Diagram



# Agenda

- OS Boot Sequence and Operation
- Devices and I/O, interrupt and trap
- Application, Multiprogramming/time-sharing

# Launching Applications

- Applications are called “processes” in most OSs.  
进程 – Process: separate memory; (OS层面也有共享 eg. printf)  
线程 – Thread: shared memory
- Created by another process calling into an OS routine (using a “syscall”, more details later). **OS是所有进程的祖先**
  - Depends on OS, but Linux uses fork to create a new process, and execve to load application.
- Loads executable file from disk (using the file system service) and puts instructions & data into memory (.text, .data sections), prepare stack and heap.
- Set argc and argv, jump into the main function.

# Supervisor Mode

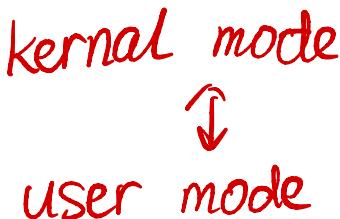
- If something goes wrong in an application, it could crash the entire machine.
  - And what about malware, etc.? 恶意软件
- The OS may need to enforce resource constraints to applications (e.g., access to devices).
- To help protect the OS from the application, CPUs have a **supervisor mode** bit.
  - When not in supervisor mode (user mode), a process can only access a subset of instructions and (physical) memory.
  - Process can enter the supervisor mode by using an **interrupt**, and change out of supervisor mode using a special instruction.

# Syscalls

- What if we want to call into an OS routine?  
(e.g., to read a file, launch a new process,  
send data, etc.)  
*System call*
  - Need to perform a **syscall**: set up function  
arguments in registers, and then raise **software  
interrupt**
  - OS will perform the operation and return to user  
mode
- This way, the OS can mediate access to all  
resources, including devices and the CPU  
itself.

# Multiprogramming

- The OS runs multiple applications at the same time.
- But not really (unless you have a core per process)
  - Time-sharing processor 随时间分配
- When jumping into process, set **timer** interrupt.
  - When it expires, store PC, registers, etc. (process state).
  - Pick a different process to run and load its state.
  - Set timer, change to **user mode**, jump to the new PC.
- Switches between processes very quickly. This is called a “context switch”.
- Deciding what process to run is called **scheduling**. 调度

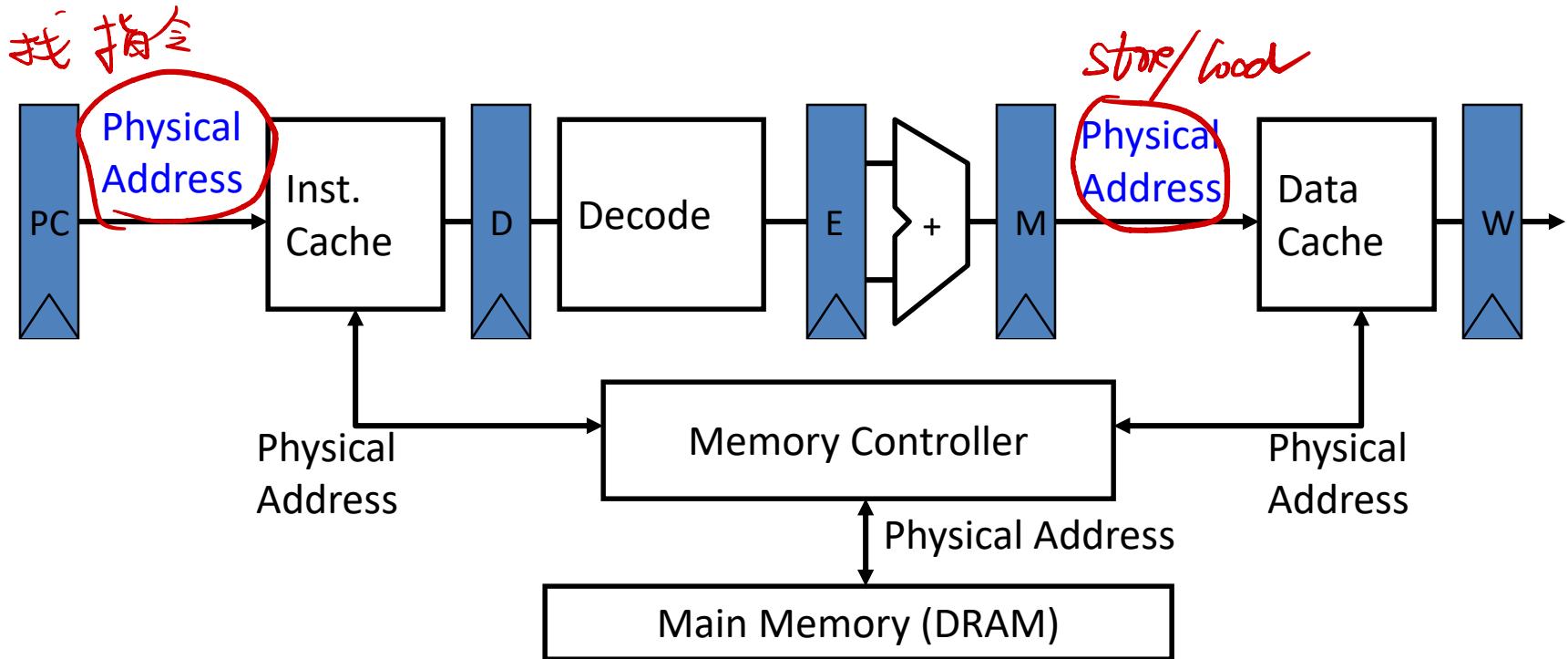


# Protection, Translation, Paging

- Supervisor mode does not fully isolate applications from each other or from the OS.
  - Application could overwrite another application's memory.  
eg. 表现矩阵
  - Also, may want to address more memory than we actually have (e.g., for sparse data structures).
- Solution: Virtual Memory. Gives each process the illusion of a full memory address space that it has completely for itself.

虚拟内存 1. 保护  
                2. 更多 memory

# “Bare” 5-Stage Pipeline



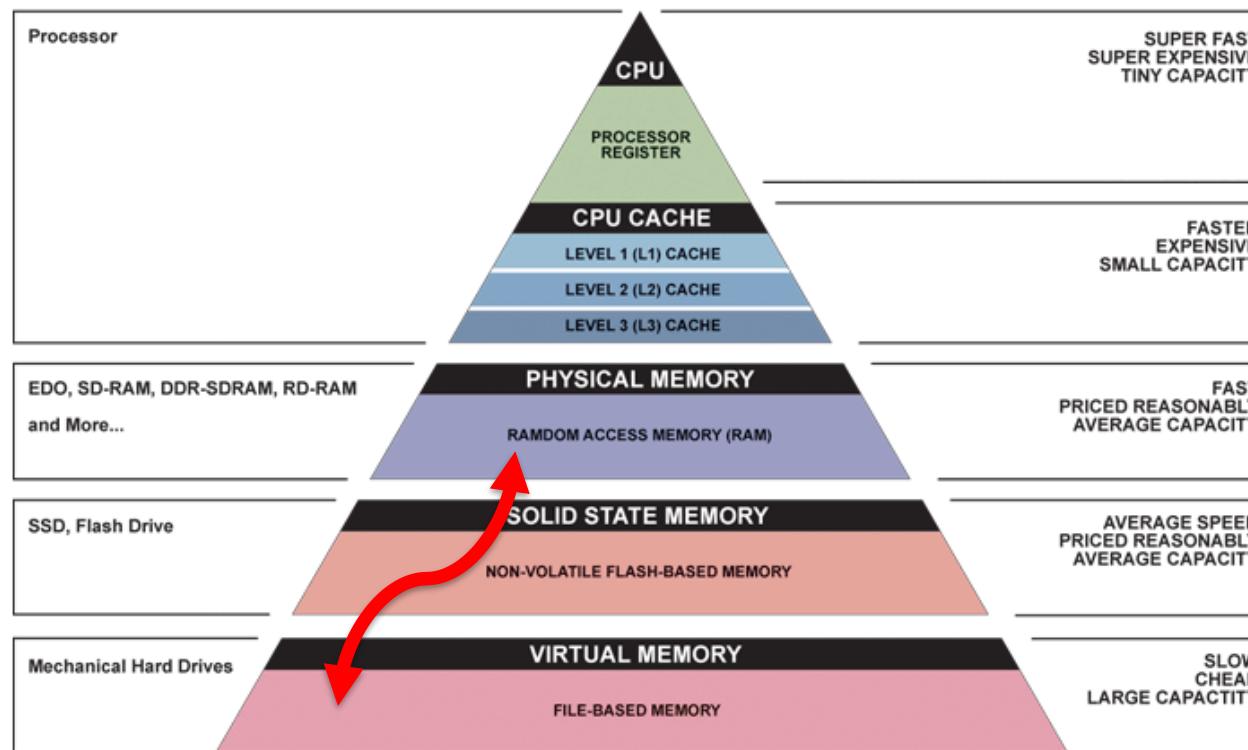
- In a bare machine, the only kind of address is a physical address

# What do we need Virtual Memory for?

## Reason 1: Adding Disks to Hierarchy

大：硬盘 > 内存

- Need to devise a mechanism to “connect” memory and disk in the memory hierarchy

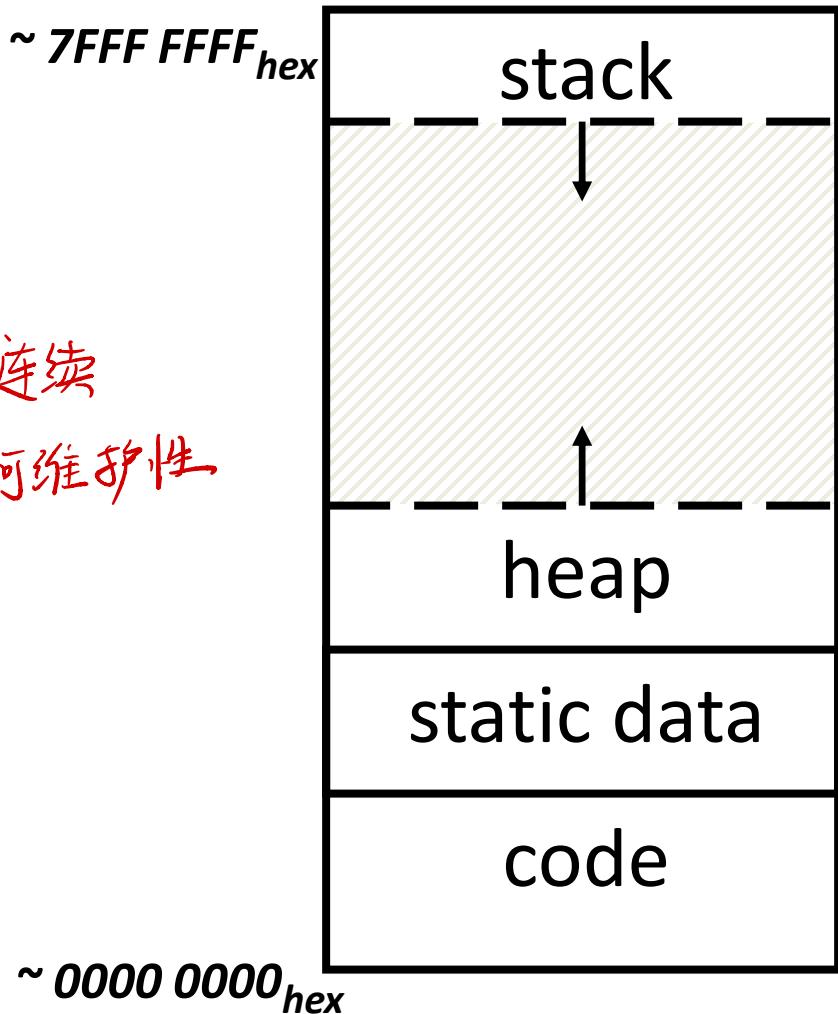


▲ Simplified Computer Memory Hierarchy  
Illustration: Ryan J. Leng

# What do we need Virtual Memory for?

## Reason 2: Simplifying Memory for Apps

- Applications should see the straightforward memory layout we saw earlier ->  
*①连续  
②可维护性*
- User-space applications should think they own all of memory
- So we give them a **virtual** view of memory



# What do we need Virtual Memory for?

## Reason 3: Protection Between Processes

防止写入其它程序的内存

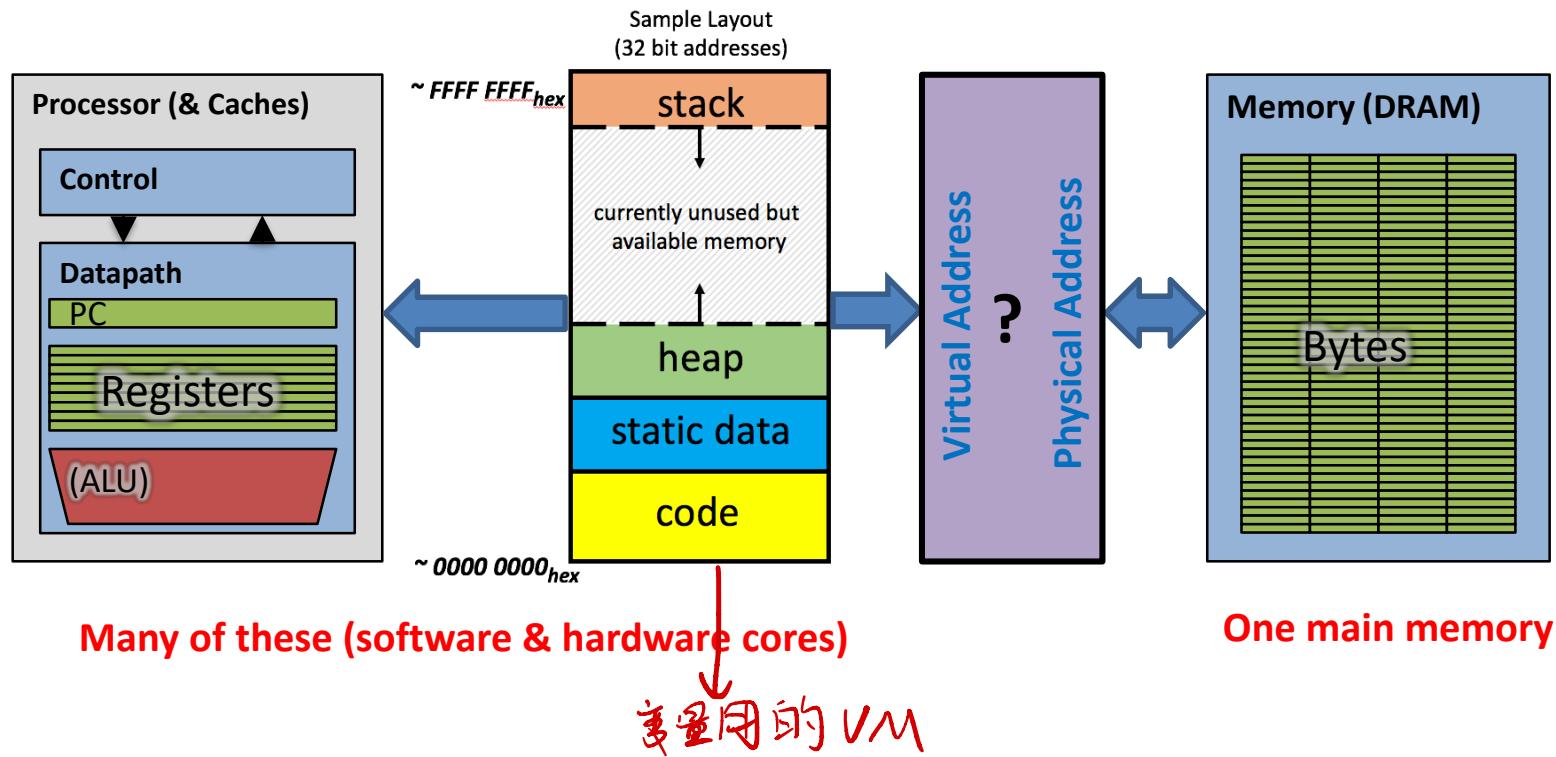
- With a bare system, addresses issued with loads/stores are real physical addresses ~~☆~~
- This means any program can issue any address, therefore can access any part of memory, even areas which it doesn't own
  - Ex: The OS data structures
- We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - a **translation mechanism**

# Address Spaces

整个虚拟地址空间不必同时映射到物理内存。这就是使它"虚拟"的原因。已分配但当前未映射到物理内存的虚拟内存的内容驻留在某种形式的外部存储(通常是磁盘)上。

- The set of addresses labeling all of memory that we can access
- Now, 2 kinds:
  - **Virtual Address Space** - the set of addresses that the user program knows about
  - **Physical Address Space** - the set of addresses that map to actual physical cells in memory
    - Hidden from user applications
- So, we need a way to map between these two address spaces

# Virtual vs. Physical Addresses



- Processes use virtual addresses, e.g., 0 ... 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 ... 0xffff,ffff)
  - *Memory manager maps virtual to physical addresses*

virtual 映射 → physical

# Dynamic Address Translation

## Motivation

Multiprogramming, multitasking: Desire to execute more than one process at a time (more than one process can reside in main memory at the same time).

## Location-independent programs

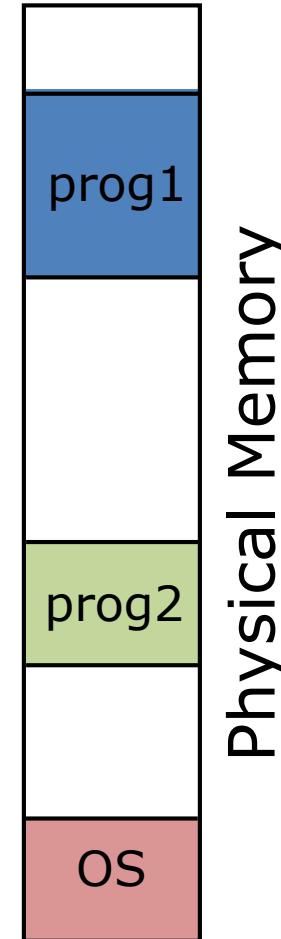
Programming and storage management ease  
=> **base register** ← add offset to each address

## Protection

Independent programs should not affect each other inadvertently

=> **bound register** ← check range of access

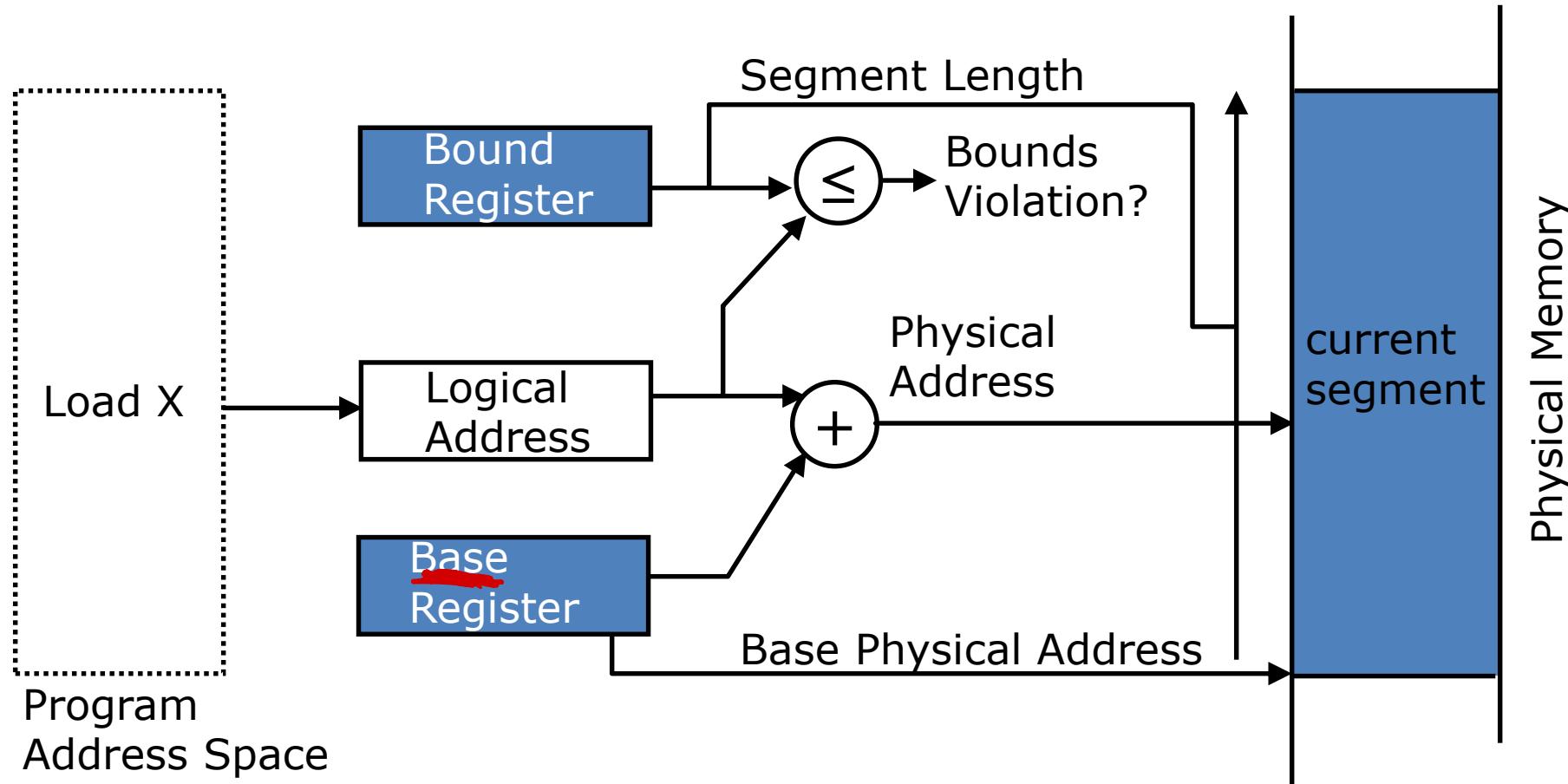
(Note: Multiprogramming drives requirement for resident *supervisor* (OS) software to manage context switches between multiple programs)



e.g.  $a[0], \dots, a[n-1]$

$x$  判斷:  $x \leq n$

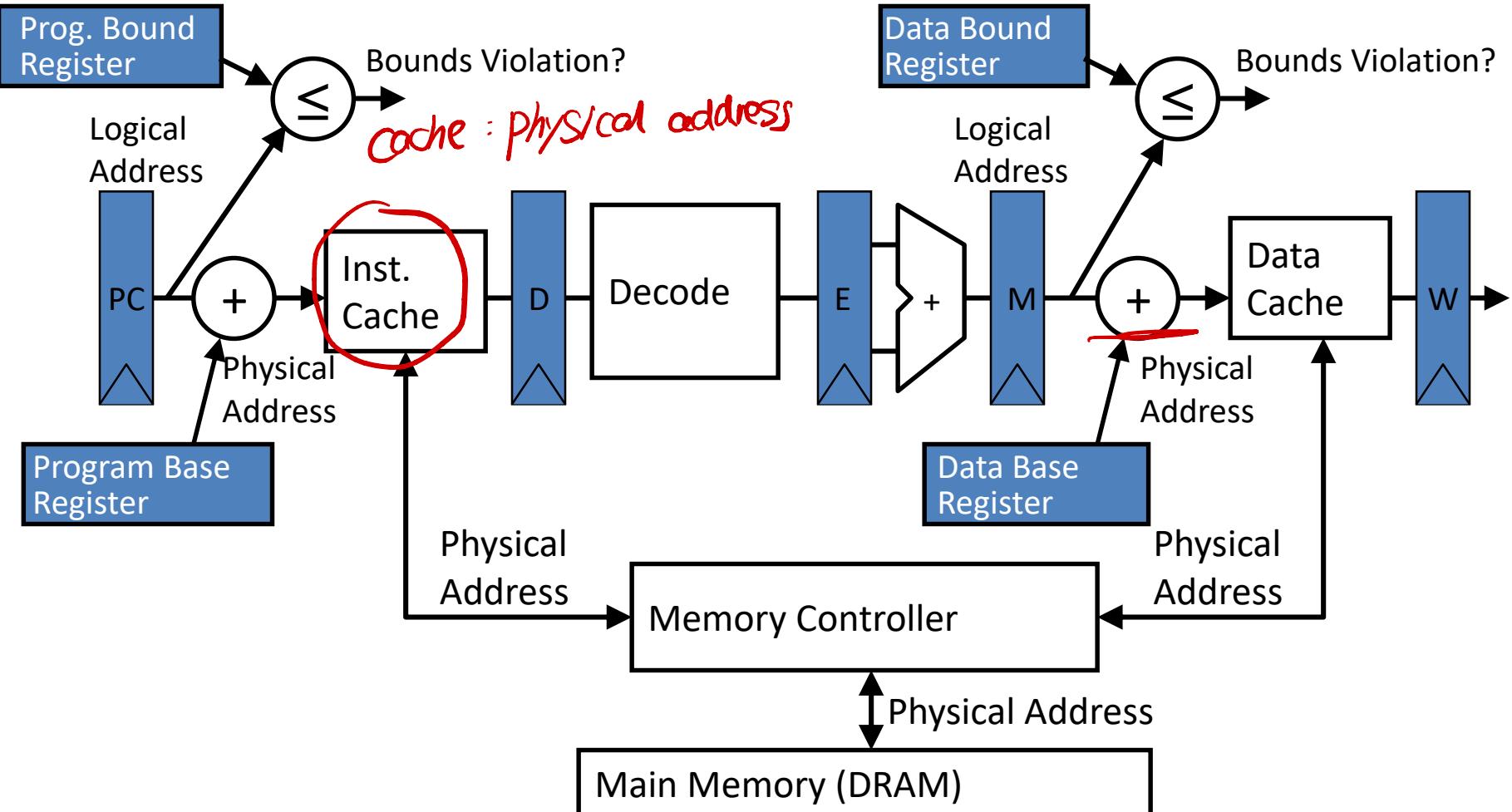
# Simple Base and Bound Translation



Base and bounds registers are visible/accessible only when processor is running in *supervisor mode*

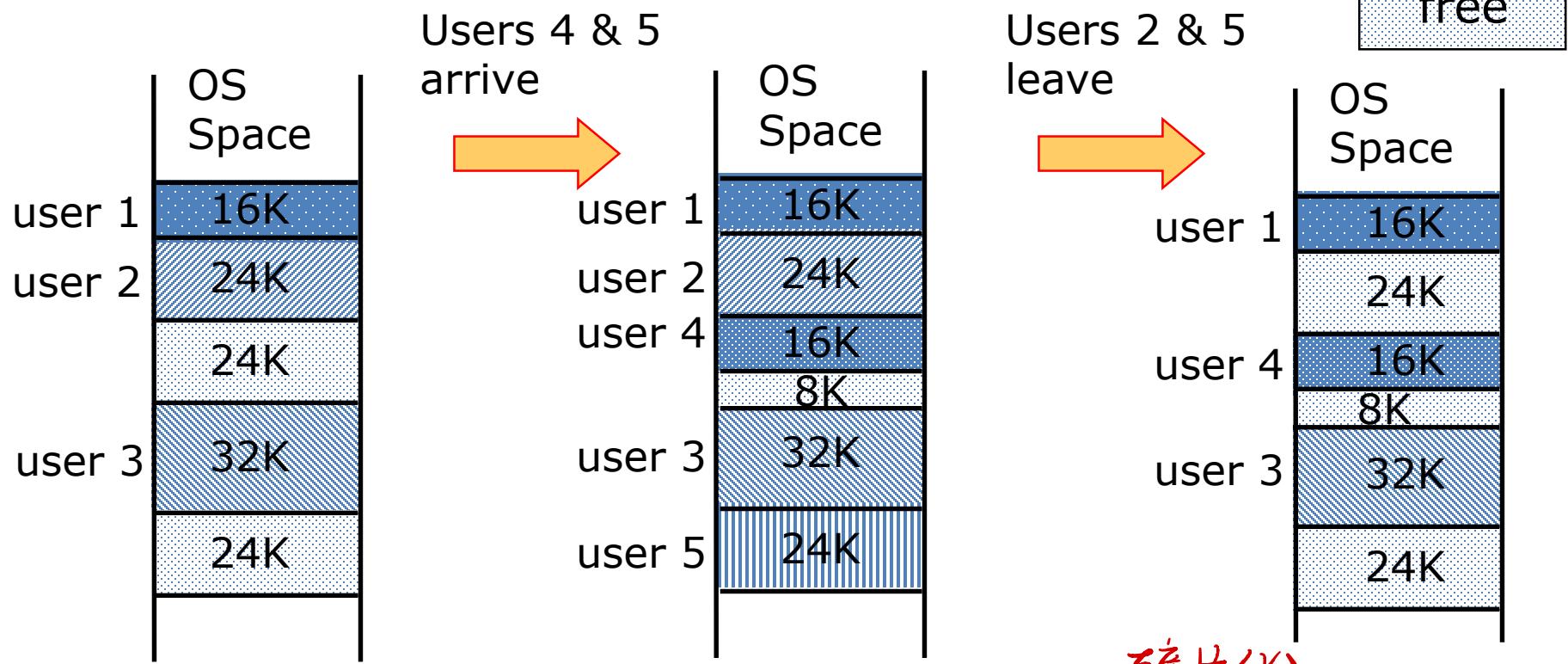
# Base and Bound Machine

取指指令；读写内存  $\Rightarrow$  VM  $\rightarrow$  Physical 转换



[ Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers) ]

# Memory Fragmentation



As users come and go, the storage is “fragmented”.  
Therefore, at some stage programs have to be moved around to compact the storage.

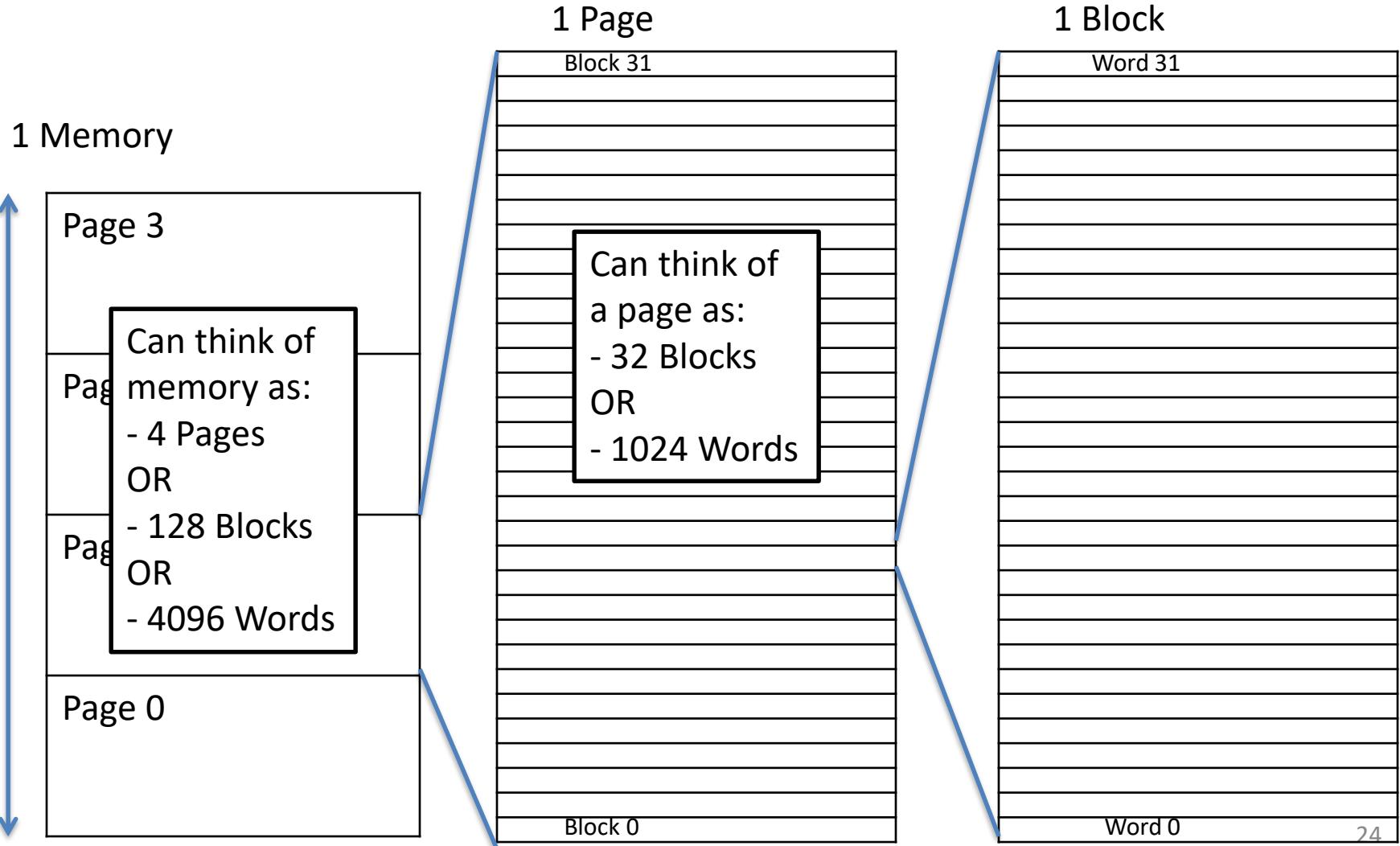
# Blocks vs. Pages

页

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
  - We could “divide” memory into a set of blocks
- In VM, we deal with individual *pages*
  - Usually ~4 KB on modern systems
    - Larger sizes also available: 2MB, very modern 1GB!
  - Now, we’ll “divide” memory into a set of pages
- Common point of confusion: Bytes, Words, Blocks, Pages are all just different ways of looking at memory!

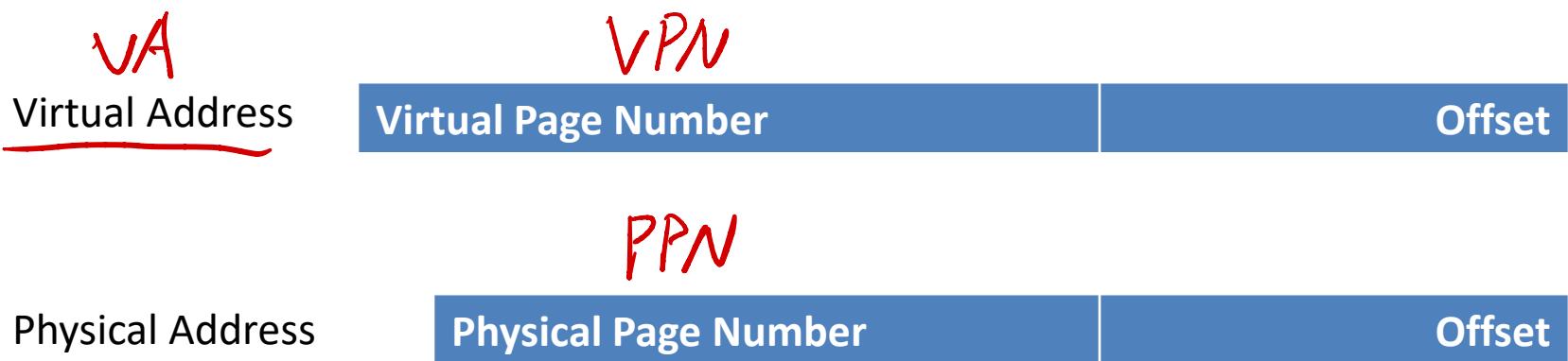
# Bytes, Words, Blocks, Pages

Ex: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)

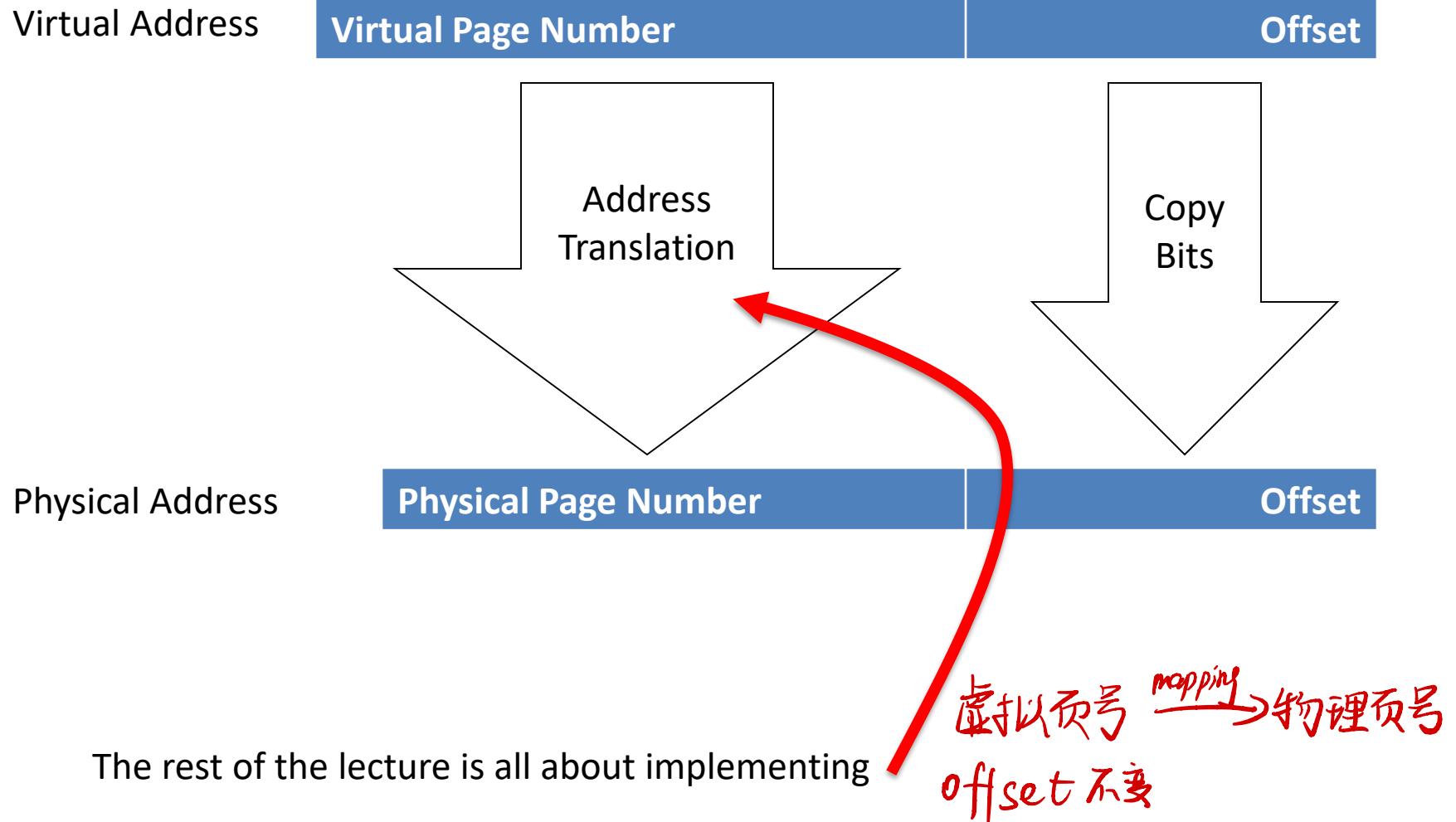


# Address Translation

- So, what do we want to achieve at the hardware level?
  - Take a Virtual Address, that points to a spot in the Virtual Address Space of a particular program, and map it to a Physical Address, which points to a physical spot in DRAM of the whole machine



# Address Translation

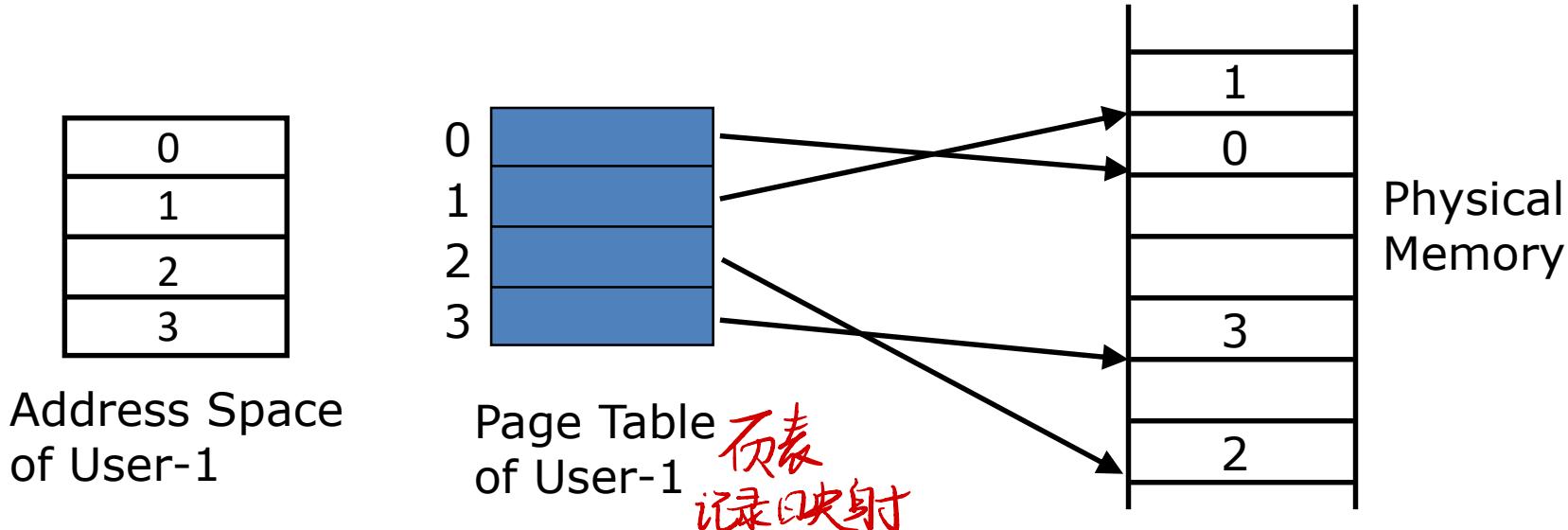


# Paged Memory Systems

- Processor-generated address can be split into:



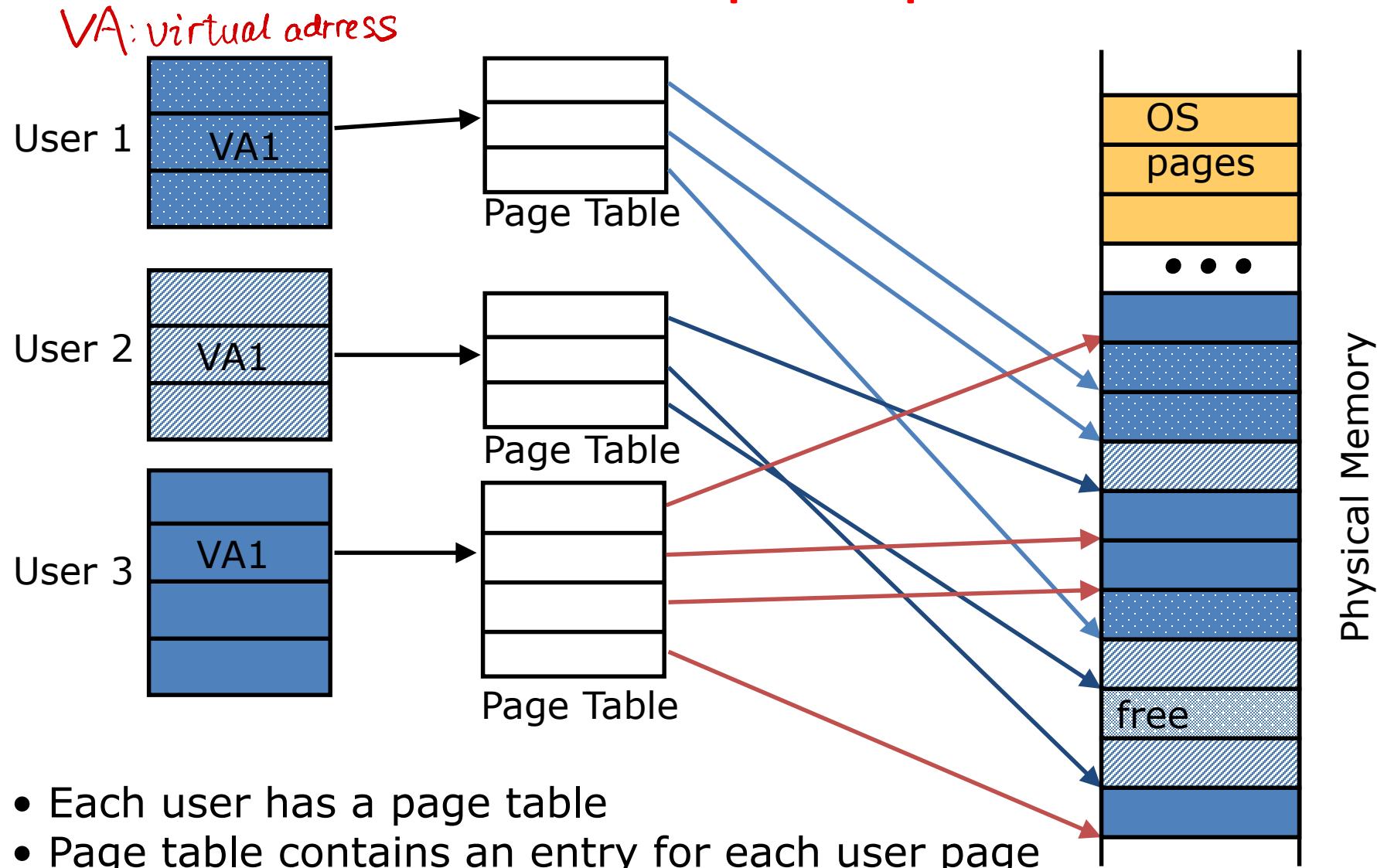
- A **page table** contains the physical address of the base of each page



*Page tables make it possible to store the pages of a program non-contiguously.*

不连续

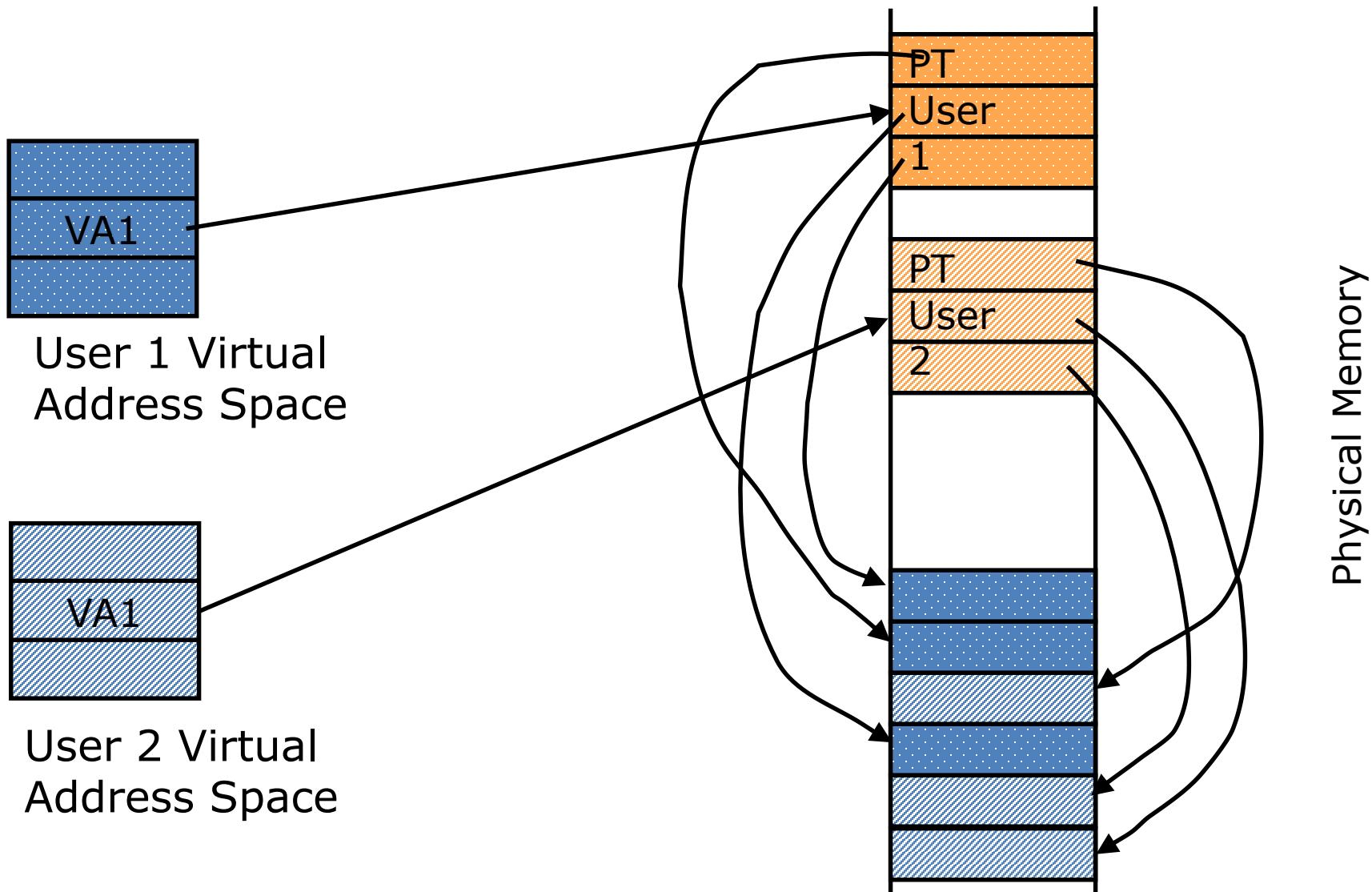
# Private Address Space per User



# Where Should Page Tables Reside?

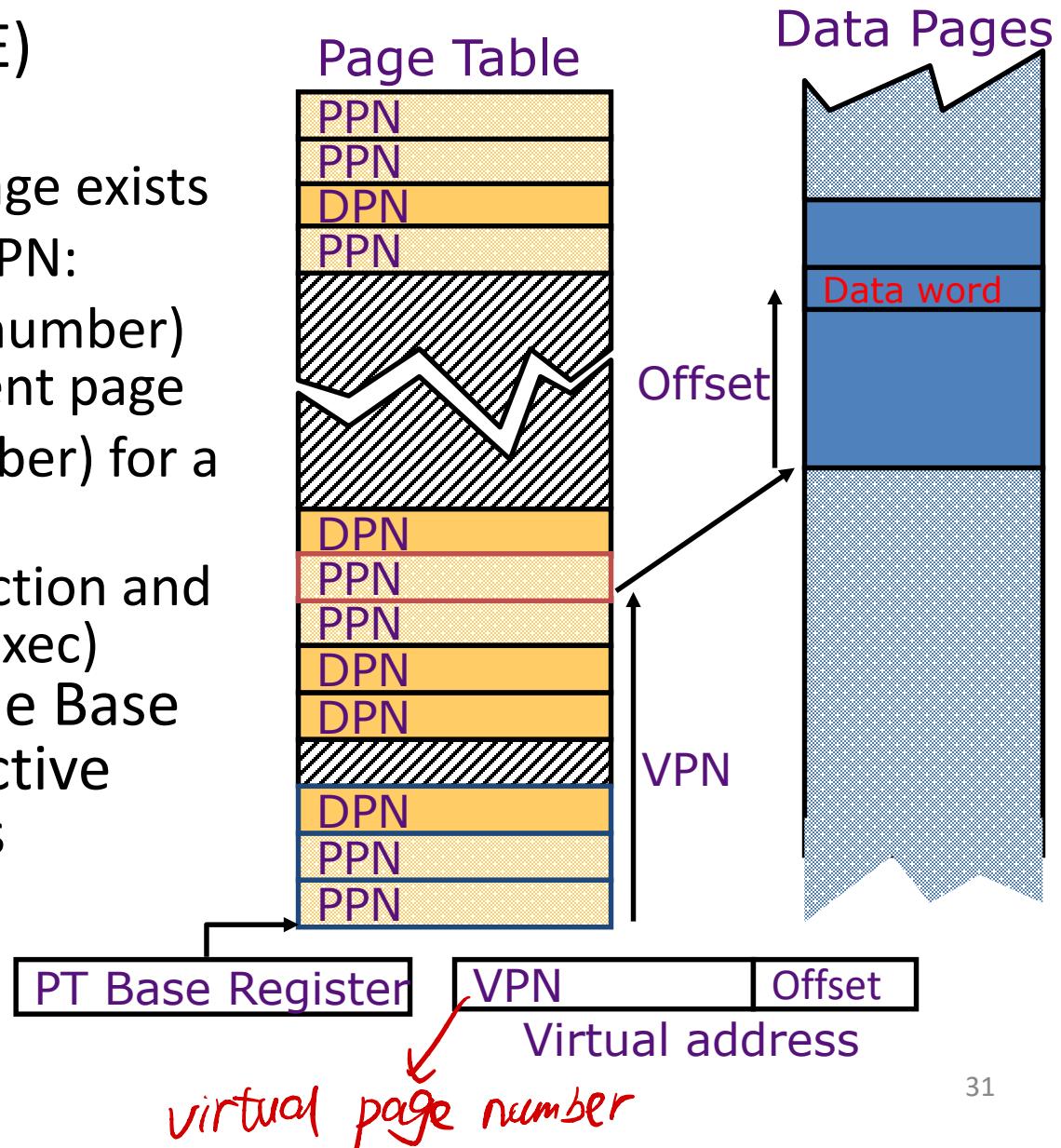
- Space required by the page tables (PT) is proportional to the address space, number of users, ...  
 $\Rightarrow$  *Too large to keep in CPU registers*
- Idea: Keep PTs in the main memory
  - Needs one reference to retrieve the page base address and another to access the data word  
 $\Rightarrow$  *doubles the number of memory references!*

# Page Tables in Physical Memory



# Linear (simple) Page Table

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists
  - And either PPN or DPN:
    - PPN (**physical page number**) for a memory-resident page
    - DPN (**disk page number**) for a page **on the disk**
  - Status bits for protection and usage (read, write, exec)
- OS sets the Page Table Base Register whenever active user process changes



# Suppose an instruction references a memory page that isn't in DRAM?

缺页中断

- We get an exception of type “page fault”
- Page fault handler does the following:
  - If virtual page doesn't yet exist, assign an unused page in DRAM, or if page exists ...  
1° 已经写入 disk
  - Initiate transfer of the page we're requesting from disk to DRAM, assigning to an unused page  
2° replacement (LRU)
  - If no unused page is left, a *page currently in DRAM is selected to be replaced* (based on usage)
  - The replaced page is written (back) to disk, page table entry that maps that VPN->PPN is marked as invalid/DPN
  - Page table entry of the page we're requesting is updated with a (now) valid PPN  
硬盘速度 < memory 速度

# Size of Linear Page Table

With 32-bit memory addresses, 4-KB pages:

=>  $2^{32} / 2^{12} = 2^{20}$  virtual pages per user, assuming 4-Byte PTEs,  
=>  $2^{20}$  PTEs, i.e., 4 MB page table per process!

Larger pages?      内部碎片化

- Internal fragmentation (Not all memory in page gets used)
- Larger page fault penalty (more time to read from disk)

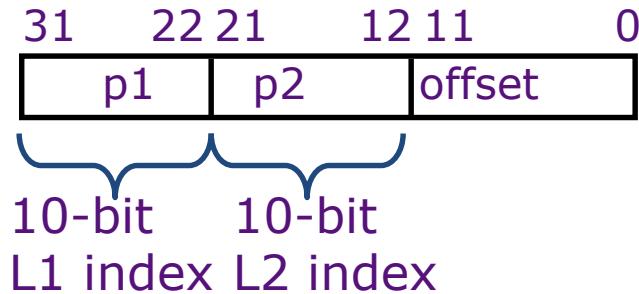
What about 64-bit virtual address space???

- Even 1MB pages would require  $2^{44}$  8-Byte PTEs (**35 TB!**)

*What is the “saving grace”? Most processes only use a set of high address (stack), and a set of low address (instructions, heap)*

# Hierarchical Page Table – exploits sparsity of virtual address space use

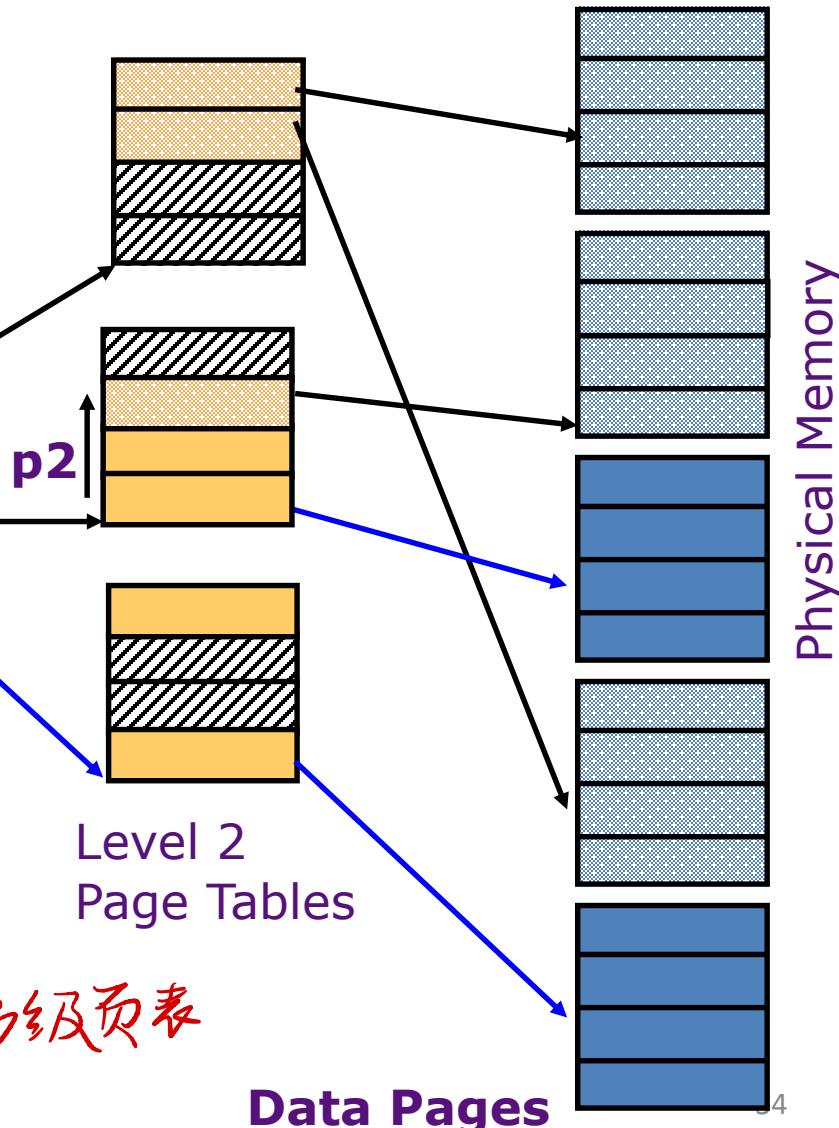
Virtual Address



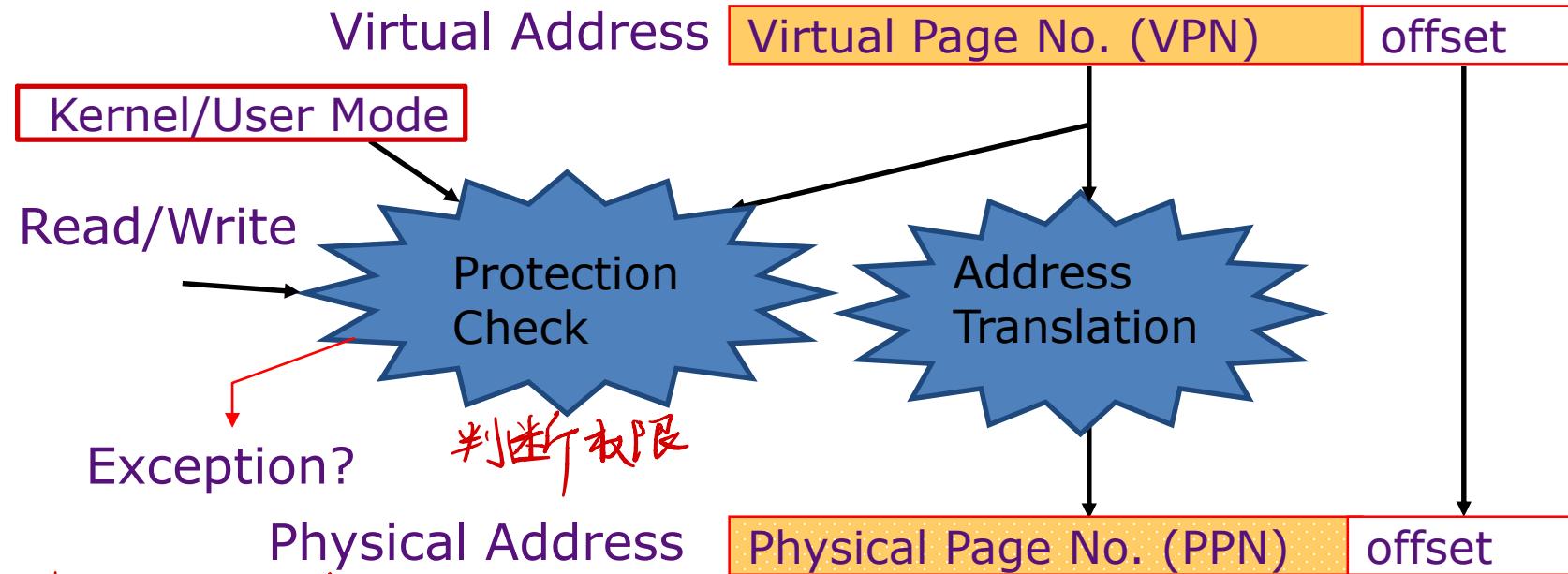
Root of the Current Page Table |  
一级页表的基地址

(Processor Register)

Level 1 Page Table



# Address Translation & Protection



代码页: read only, 不能改  $\Rightarrow$  protection check

- Every instruction and data access needs address translation and protection checks
- A good VM design needs to be fast ( $\sim$  one cycle) and space efficient



# Translation Lookaside Buffers (TLB)

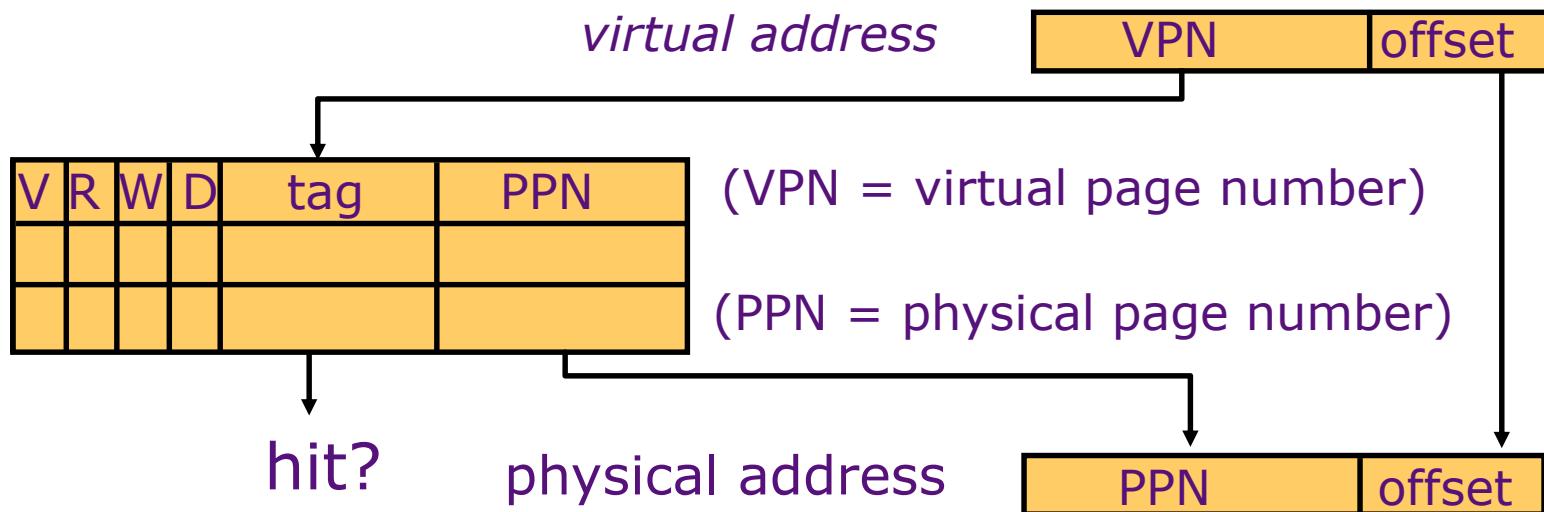
Address translation is very expensive!

快表

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache some translations in TLB*

TLB hit  $\Rightarrow$  Single-Cycle Translation 放最常使用的页的映射关系  
TLB miss  $\Rightarrow$  Page-Table Walk to refill 映射关系,

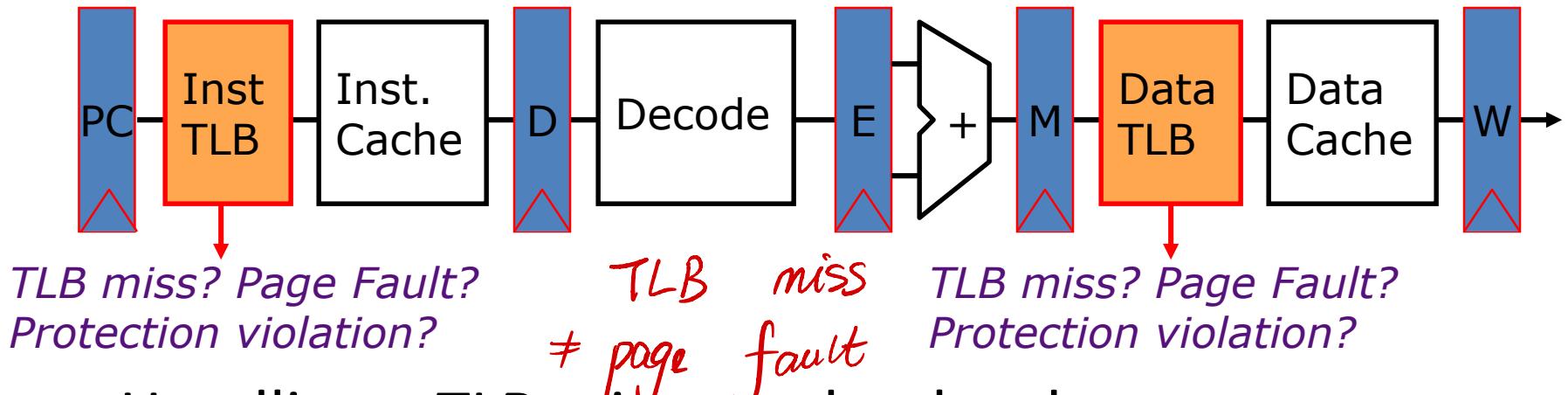


# TLB Designs



- Typically 32-128 entries, usually **fully associative**
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy  $L_1$  i TLB instruction  $L_1$  DTLB data
- Upon context switch? New VM space! Flush TLB
  - ...
- “TLB Reach”: Size of largest virtual address space that can be simultaneously mapped by TLB  $(TLB \text{ shoot down})$

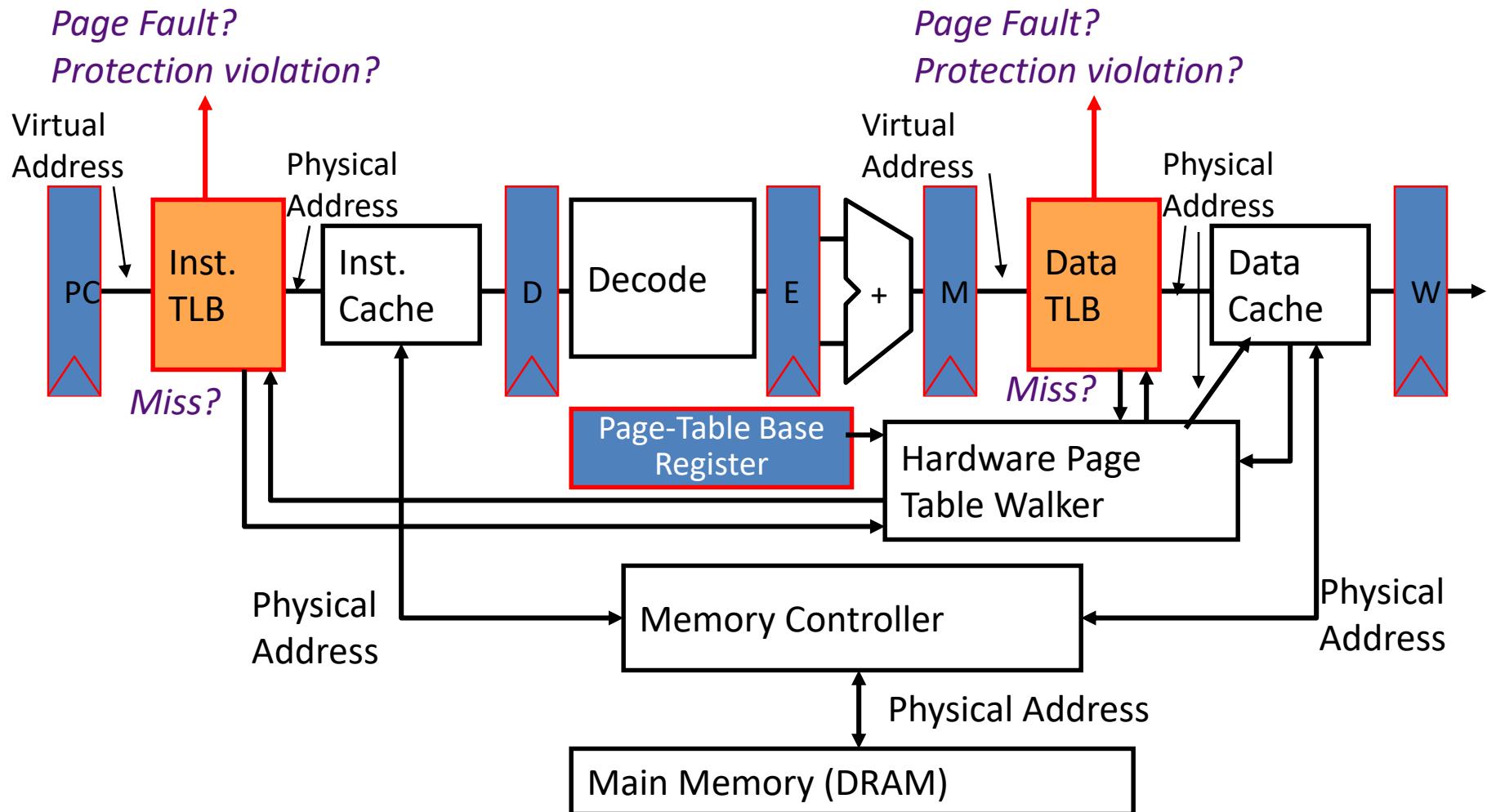
# VM-related events in pipeline



- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware now
- Handling a page fault (e.g., page is on disk) needs a *precise* trap so software handler can easily resume after retrieving page
- Handling protection violation may abort process

# Page-Based Virtual-Memory Machine

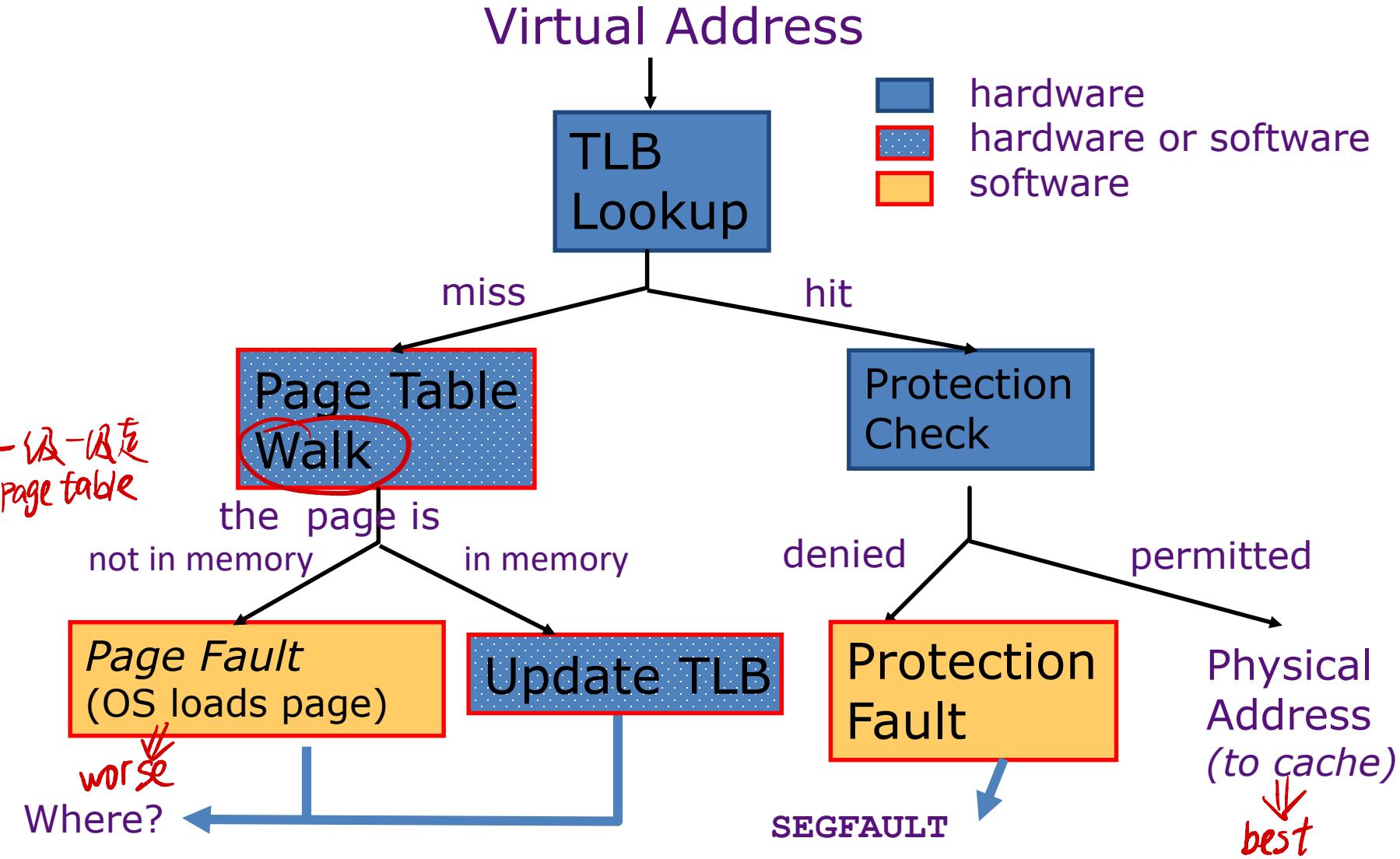
## (Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

# Address Translation:

*putting it all together*



# Modern Virtual Memory Systems

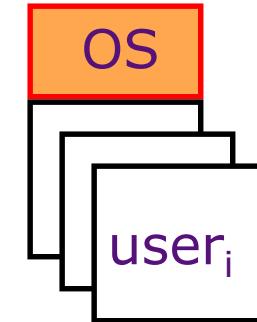
*Illusion of a large, private, uniform store*

## Protection & Privacy

several users, each with their private address space and one or more shared address spaces

page table = name space

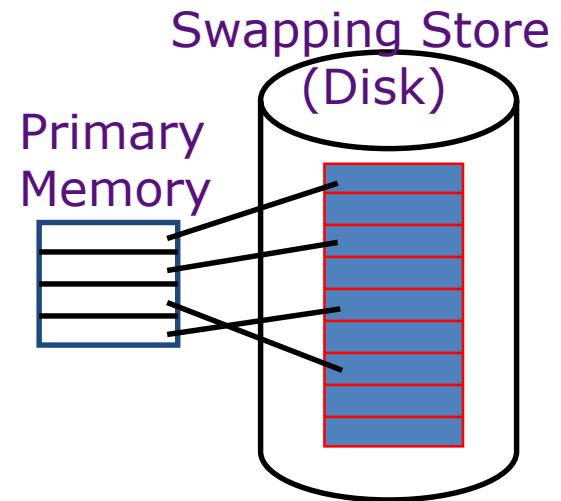
一个进程不同线程  
⇒ 共享 page table



## Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations



*The price is address translation on each memory reference*



# Unlimited?

```
wangc@64G:~$ ulimit -s  
8192  
wangc@64G:~$ ulimit -a  
core file size          (blocks, -c) 0  
data seg size           (kbytes, -d) unlimited  
scheduling priority     (-e) 0  
file size               (blocks, -f) unlimited  
pending signals          (-i) 256820  
max locked memory       (kbytes, -l) 16384  
max memory size         (kbytes, -m) unlimited  
open files               (-n) 1024  
pipe size                (512 bytes, -p) 8  
POSIX message queues    (bytes, -q) 819200  
real-time priority       (-r) 0  
stack size               (kbytes, -s) 8192  
cpu time                 (seconds, -t) unlimited  
max user processes        (-u) 256820  
virtual memory            (kbytes, -v) unlimited  
file locks                (-x) unlimited  
wangc@64G:~$
```

# Remember: Out of Memory

- Insufficient free memory: `malloc()` returns `NULL`

```
1 /*  
2      This is a test for CS 110. All copyrights ...  
3 */  
4  
5 #include <stdio.h>  
6 #include <stdlib.h>  
7  
8 int main(int argc, char **argv) {  
9     const int G = 1024 * 1024 * 1024;  
10    for (int n = 0; ; n++) {  
11        char *p = malloc(G * sizeof(char)); // 1GB every time  
12        if (p == NULL) {  
13            fprintf(stderr,  
14                    "failed to allocate > %g TeraBytes\n",  
15                    n / 1024.0);  
16            return 1;  
17        }  
18        // no free, keep allocating until out of memory  
19    }  
20    return 0;  
21 }
```

```
wangc@64G:~/TT$ gcc test.c -o t -Wall -O3  
wangc@64G:~/TT$ ./t  
failed to allocate > 127.99 TeraBytes  
wangc@64G:~/TT$
```

# Limited VM Space with x86-64

- 64-bit Linux allows up to **128TB** of virtual address space for individual processes, and can address approximately 64 TB of physical memory, subject to processor and system limitations.
- For Windows 64-bit versions, both 32- and 64-bit applications, if not linked with “*large address aware*”, are limited to **2GB** of virtual address space; otherwise, **128TB** for Windows 8.1 and Windows Server 2012 R2 or later.

Source: <https://en.wikipedia.org/wiki/X86-64>

# 48bit for address translation only

- Still provides plenty of space!
- Higher bits “sign extended”: “canonical form”
- Convention: “Higher half” for the Operating System
- Intel has plans (“whitepaper”) for 56 bit translation – no hardware yet
- [https://en.wikipedia.org/wiki/X86-64#Virtual\\_address\\_space\\_details](https://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details)



# Using 128TB of Memory!?

- A lazy allocation of virtual memory
  - Not used → not allocated
  - Try reading and writing from those pointers:  
works!
  - Even writing Gigabytes of memory:  
works!
- Memory Compression!
  - Take not-recently used pages, compress them =>  
free the physical page
- <https://www.lifewire.com/understanding-compressed-memory-os-x-2260327>

Process Name	Memory	Threads	Ports	PID	User	Compressed M...	Real Mem
a.out	60.51 GB	1	10	22329	schwerti	54.30 GB	6.22 GB

# Conclusion: VM features track historical uses

- **Bare machine, only physical addresses**
  - One program owned entire machine
- **Batch-style multiprogramming**
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (not virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived
- **Time sharing**
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory