

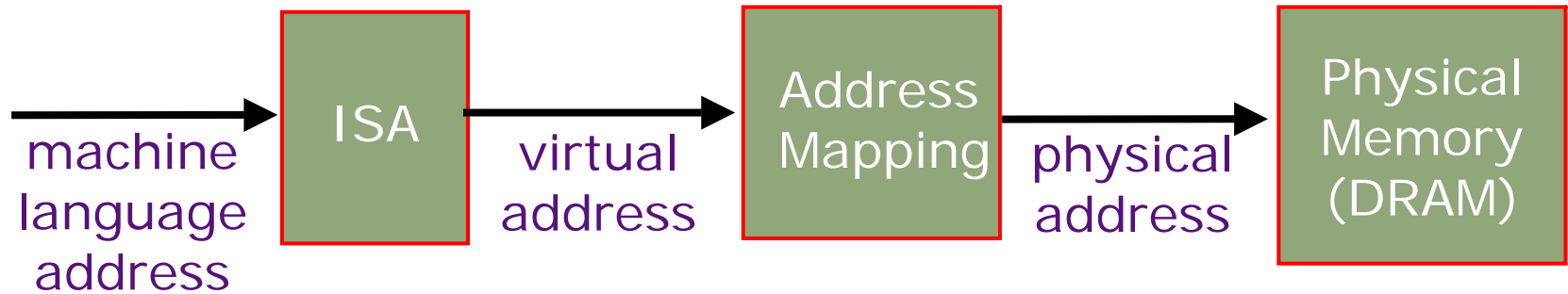
# Memory Management

---

- The Fifties
  - Absolute Addresses
  - Dynamic address translation
- The Sixties
  - Atlas' Demand Paging
  - Paged memory systems and TLBs
- Modern Virtual Memory Systems

# Names for Memory Locations

---



- Machine language address
  - as specified in machine code
- Virtual address
  - ISA specifies translation of machine code address into virtual address of program variable (sometime called *effective* address)
- Physical address
  - ⇒ operating system specifies mapping of virtual address into name for a physical memory location

# Absolute Addresses

---

*EDSAC, early 50's*

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

*How could location independence be achieved?*

*Linker and/or loader modify addresses of subroutines and callers when building a program memory image*

# Dynamic Address Translation

---

## Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*

⇒ *multiprogramming*

## Location-independent programs

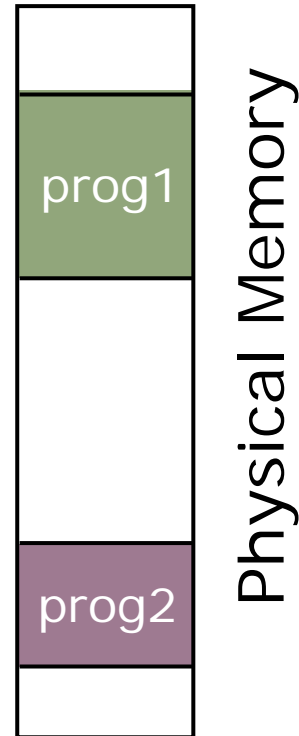
Programming and storage management ease

⇒ need for a *base register*

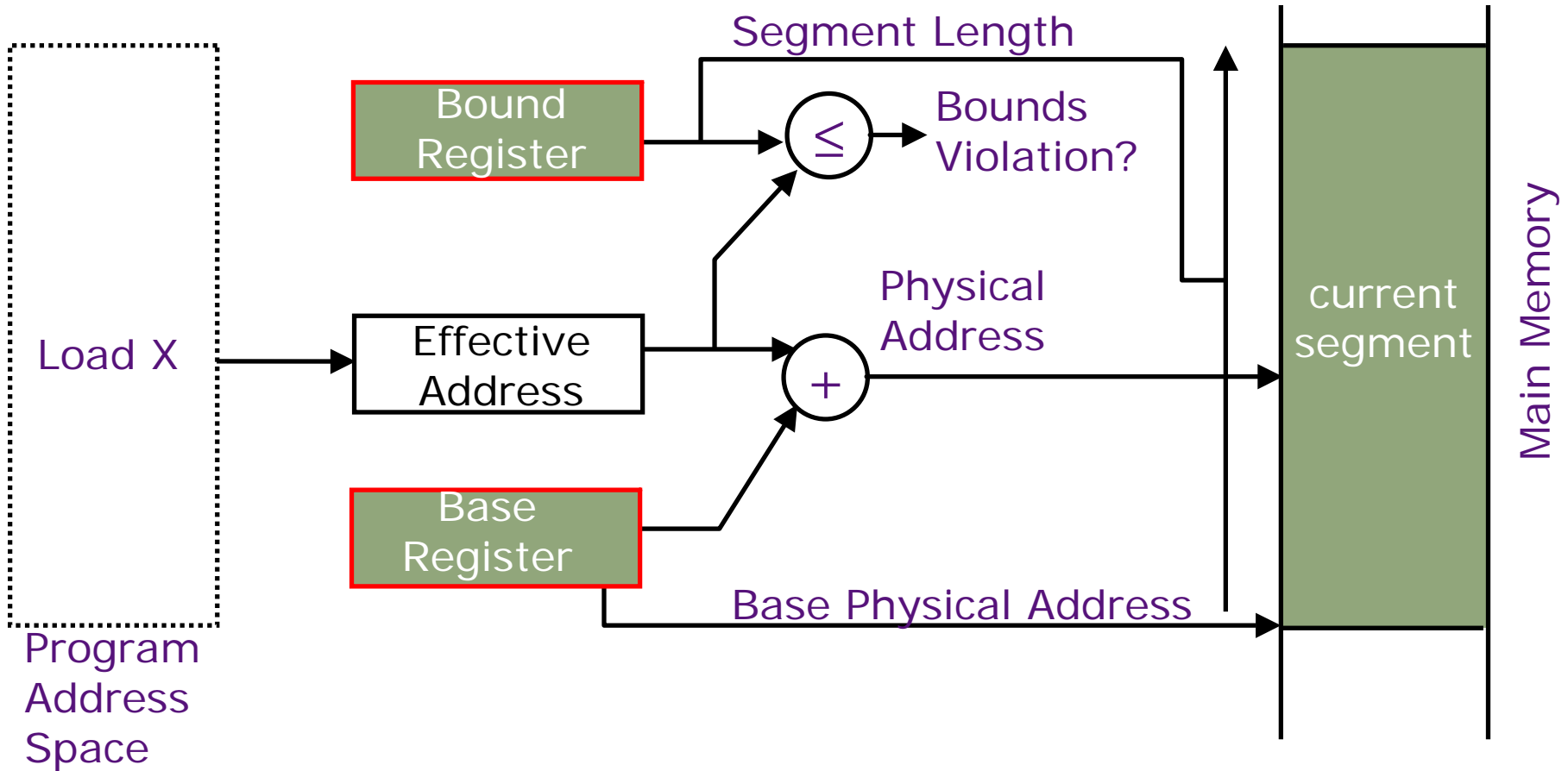
## Protection

Independent programs should not affect each other inadvertently

⇒ need for a *bound register*

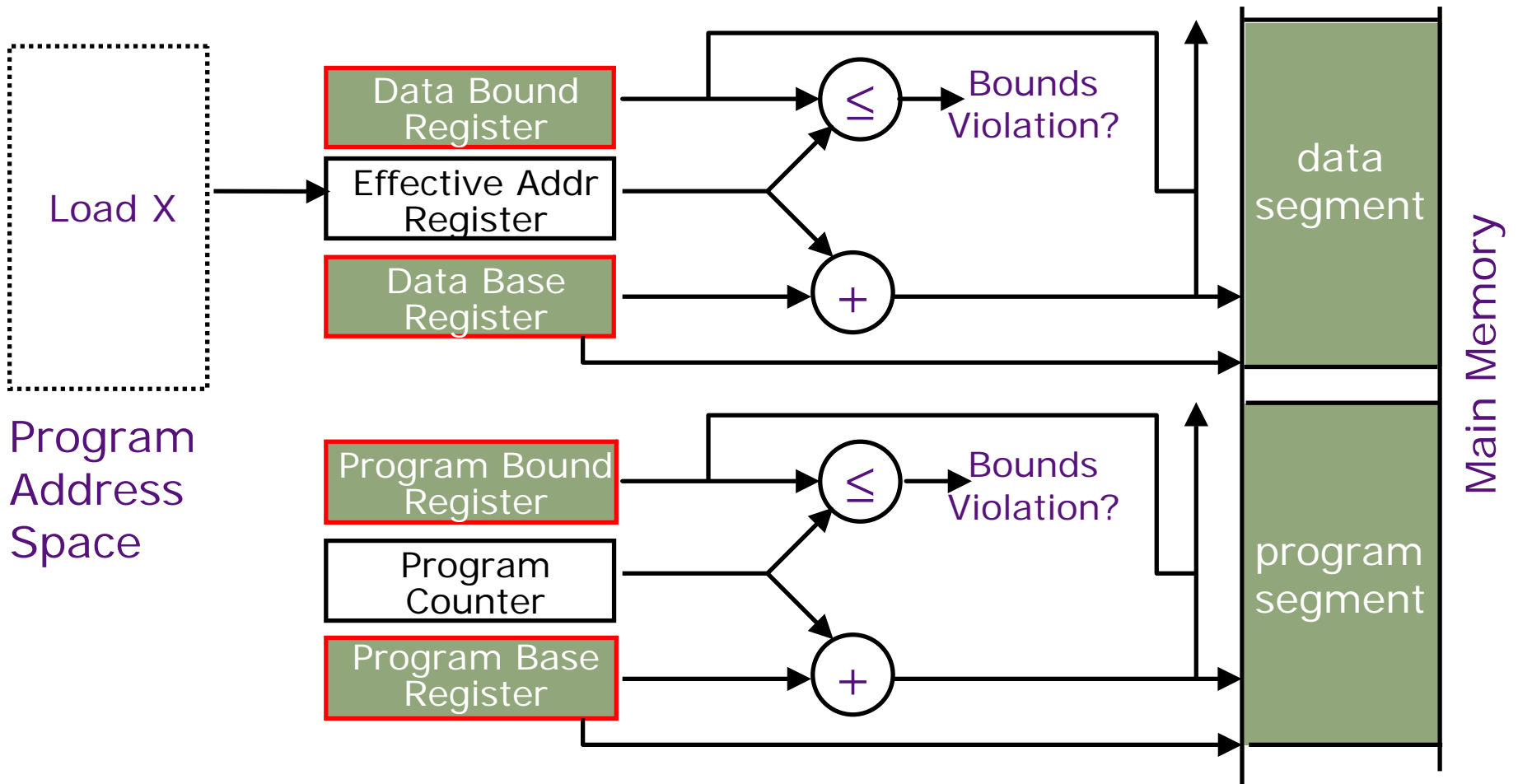


# Simple Base and Bound Translation



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

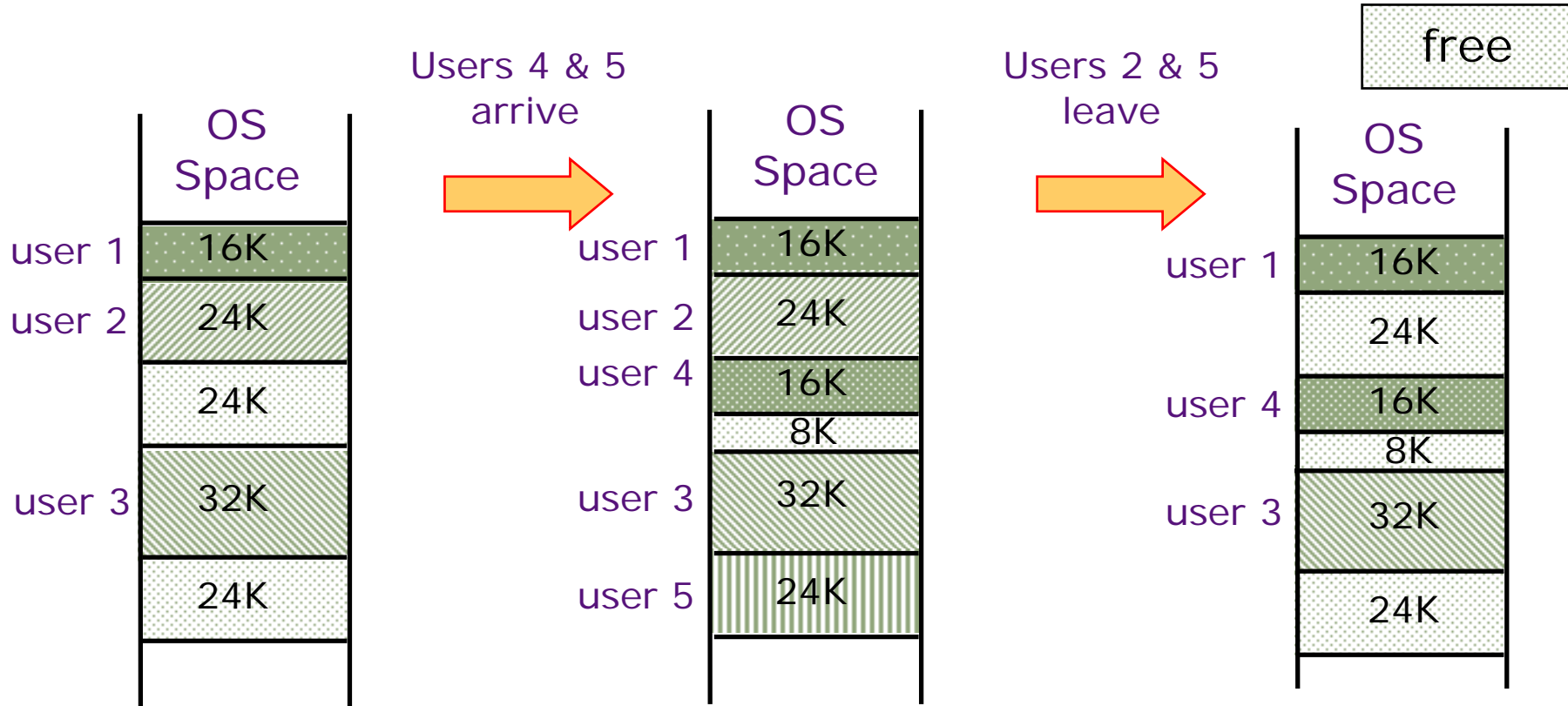
# Separate Areas for Program and Data



*What is an advantage of this separation?*

(Scheme used on all Cray vector supercomputers prior to X1, 2002)

# Memory Fragmentation



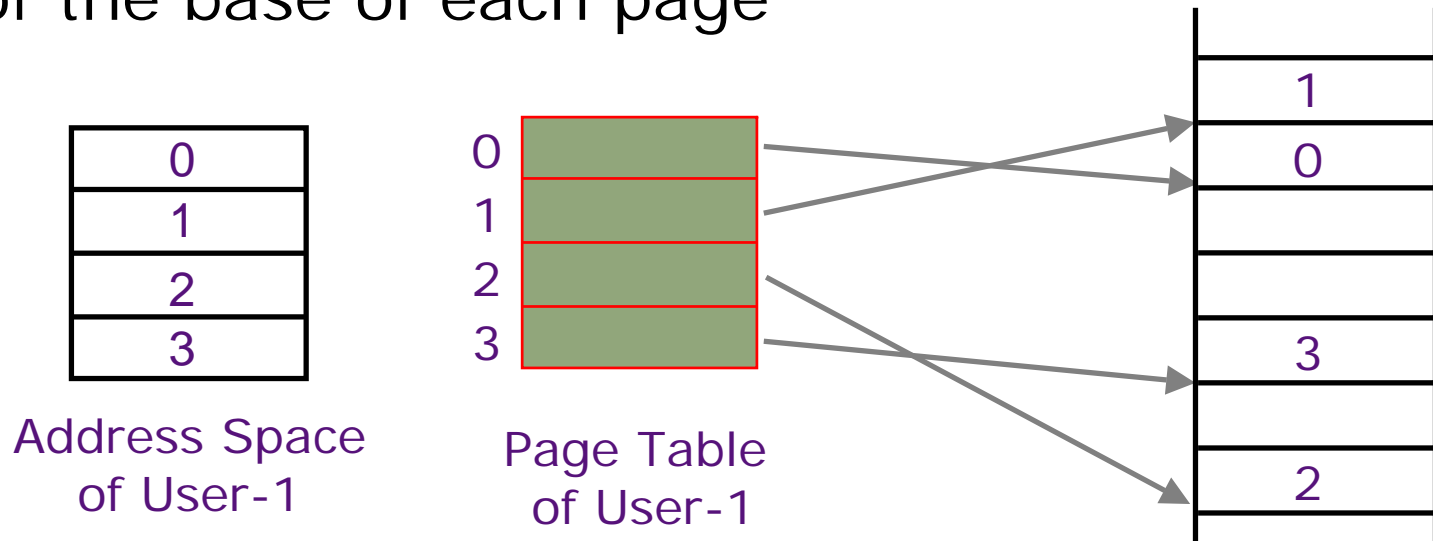
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

# Paged Memory Systems

- Processor generated address can be interpreted as a pair  $\langle \text{page number}, \text{offset} \rangle$

page number	offset
-------------	--------

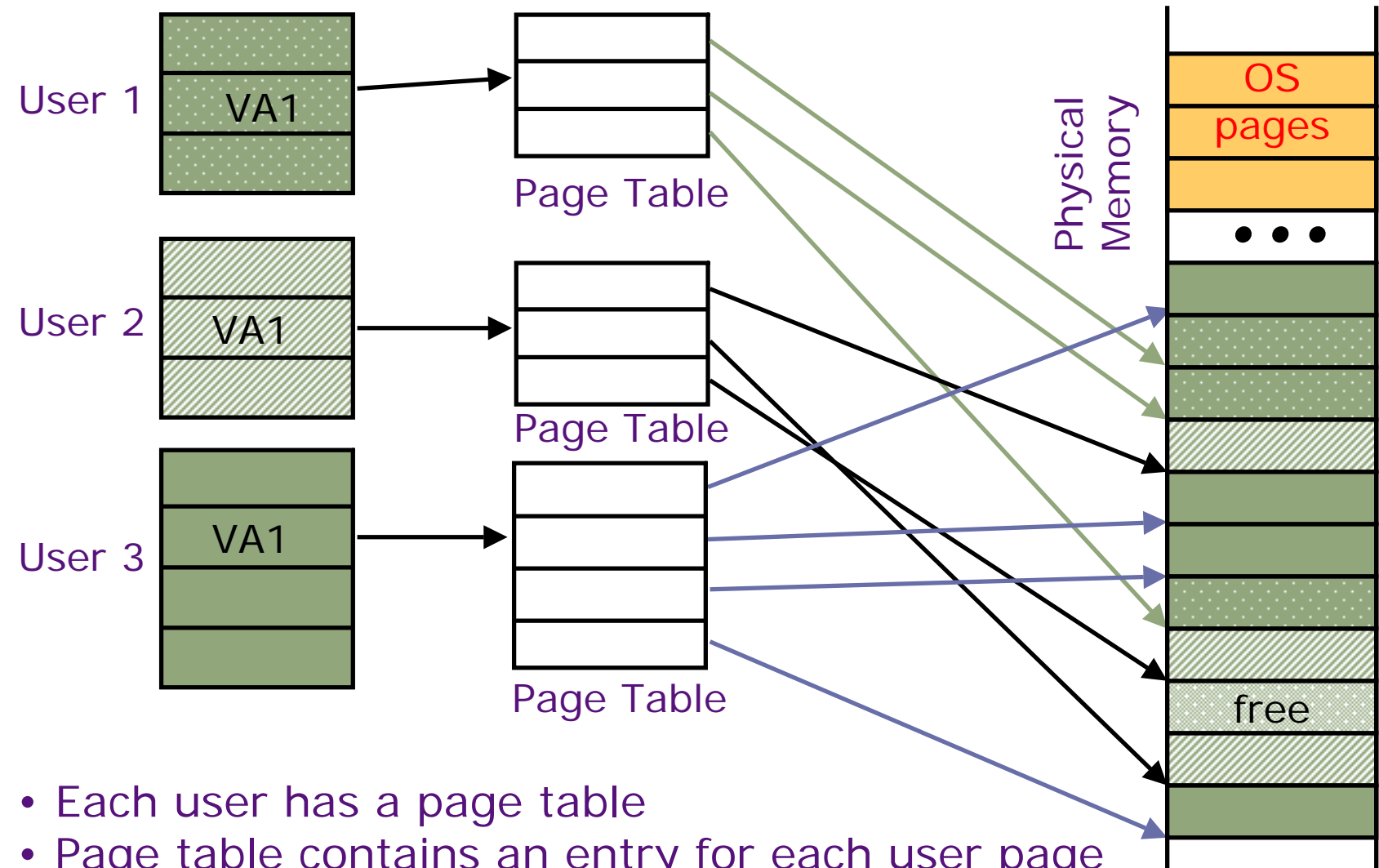
- A page table contains the physical address of the base of each page



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



# Private Address Space per User

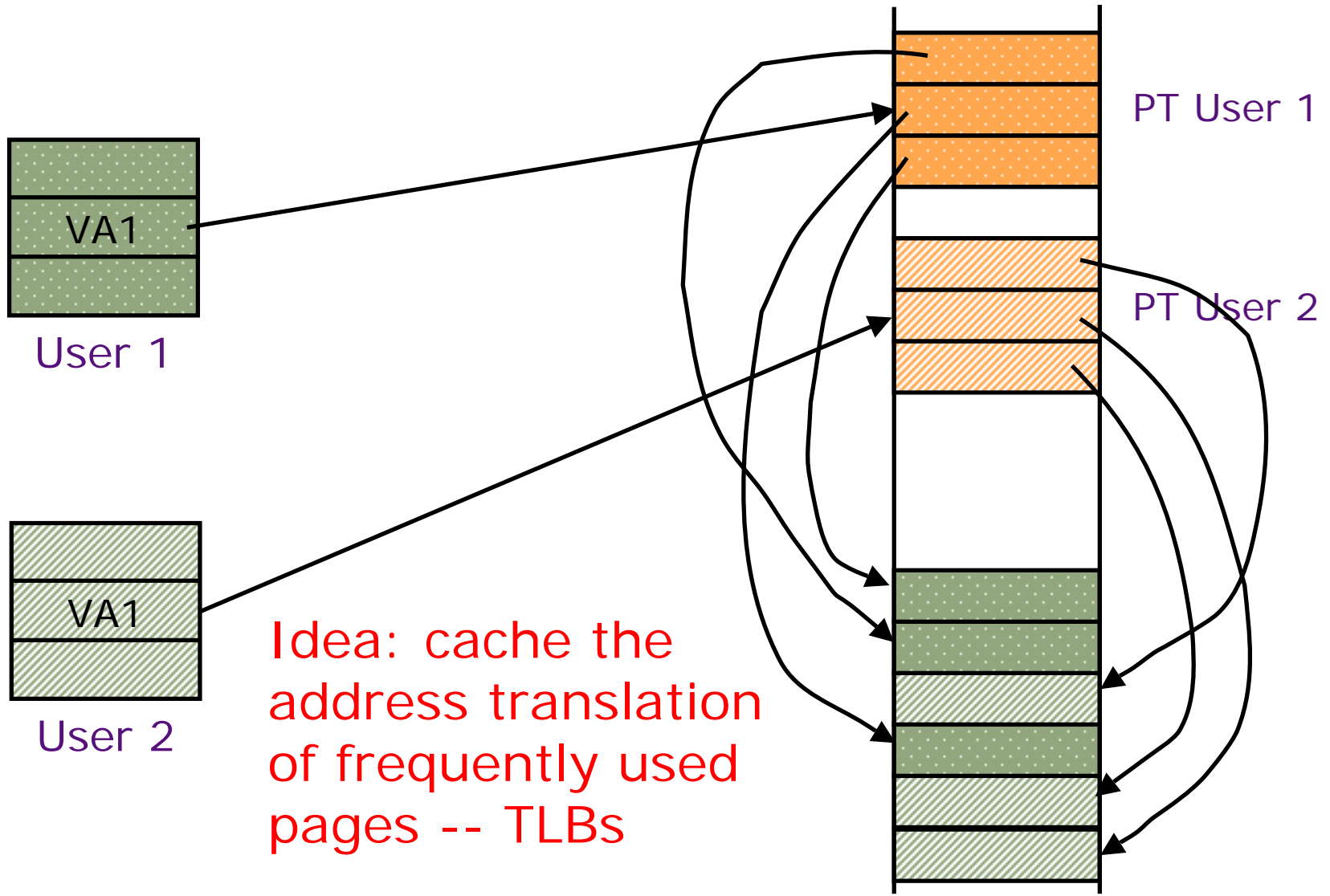


# Where Should Page Tables Reside?

---

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers
- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap
- Idea: Keep PTs in the main memory
  - needs one reference to retrieve the page base address and another to access the data word
    - ⇒ *doubles the number of memory references!*

# Page Tables in Physical Memory



# A Problem in Early Sixties

---

- There were many applications whose data could not fit in the main memory, e.g., payroll
  - *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*
- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

*tricky programming!*

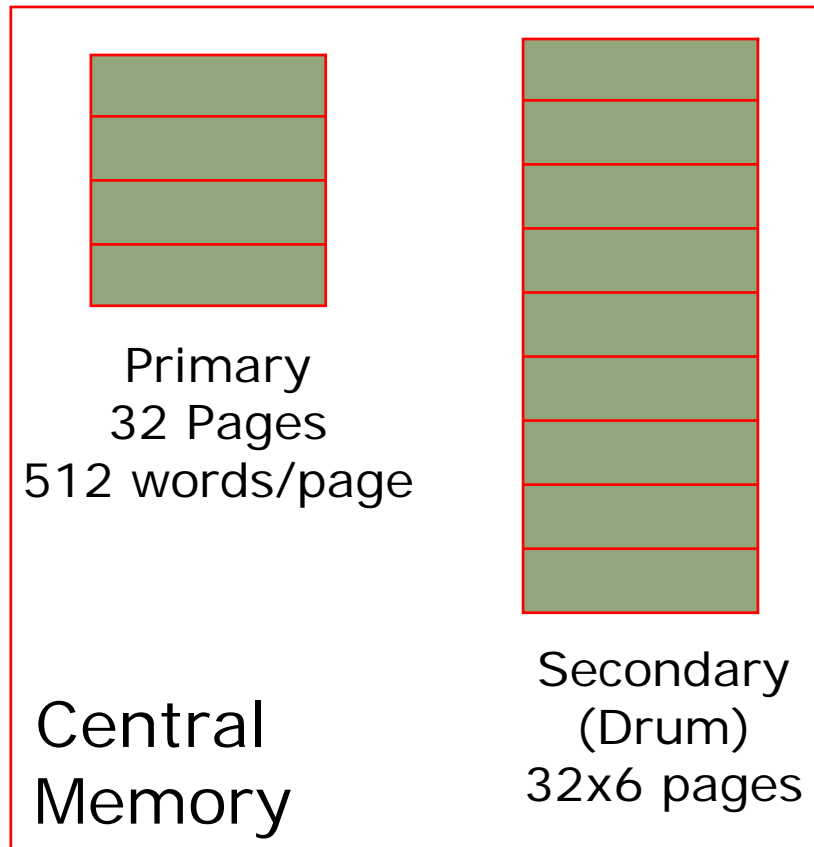
# Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

*Tom Kilburn*

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



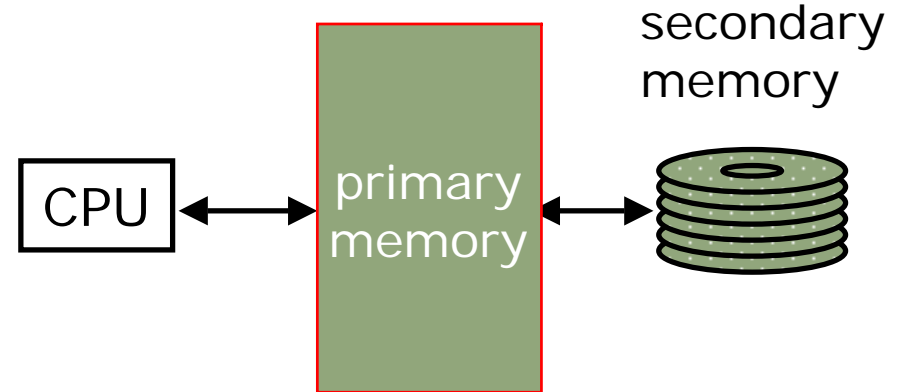
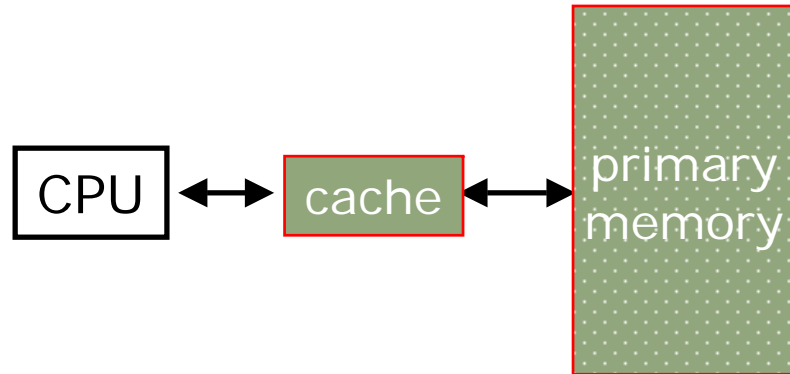
# Atlas Demand Paging Scheme

---

- On a page fault:
  - Input transfer into a free page is initiated
  - The Page Address Register (PAR) is updated
  - If no free page is left, a *page is selected to be replaced* (based on usage)
  - The replaced page is written on the drum
    - to minimize the drum latency effect, the first empty page on the drum was selected
  - The *page table is updated* to point to the new location of the page on the drum

# 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*

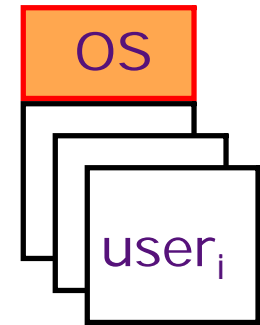
# Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

## Protection & Privacy

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

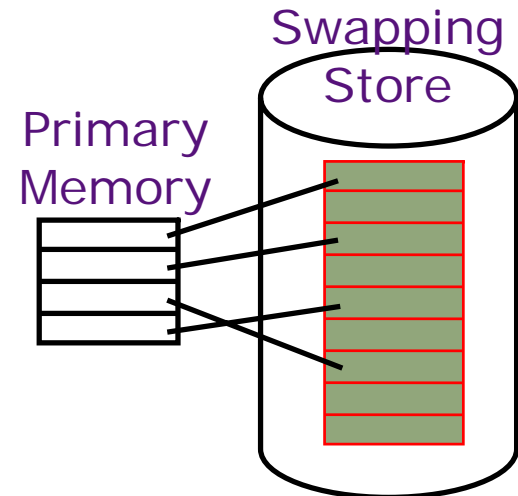
page table  $\equiv$  name space



## Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations



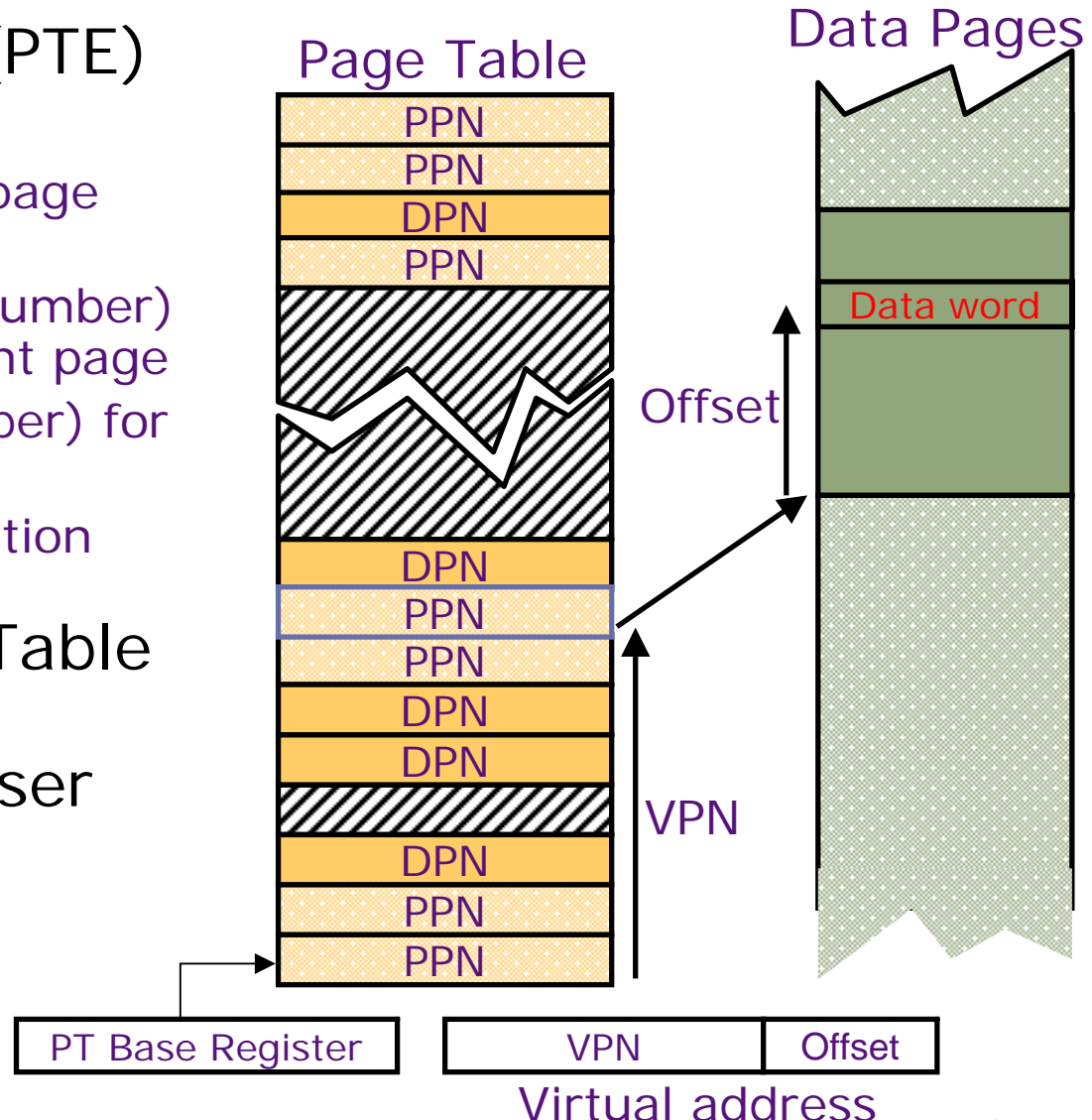
*The price is address translation on each memory reference*





# Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - 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
- OS sets the Page Table Base Register whenever active user process changes



# Size of Linear Page Table

---

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒  $2^{20}$  PTEs, i.e, 4 MB page table per user

⇒ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

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

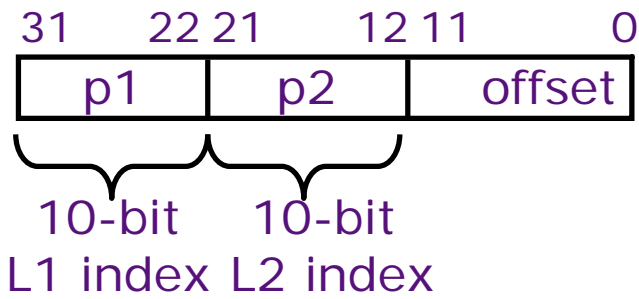
What about 64-bit virtual address space???

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

*What is the "saving grace" ?*

# Hierarchical Page Table

Virtual Address

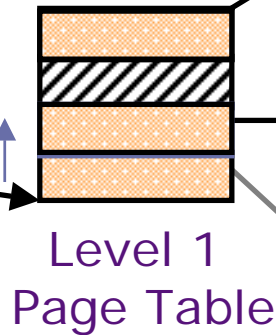


Root of the Current Page Table

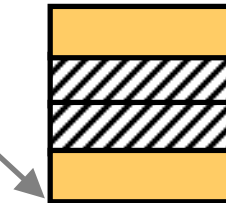
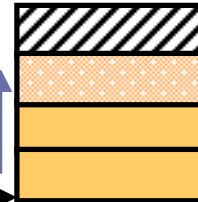
(Processor Register)



p1

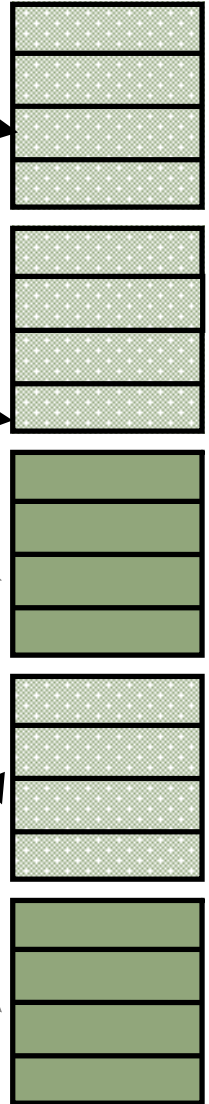


p2



Level 2 Page Tables

Data Pages

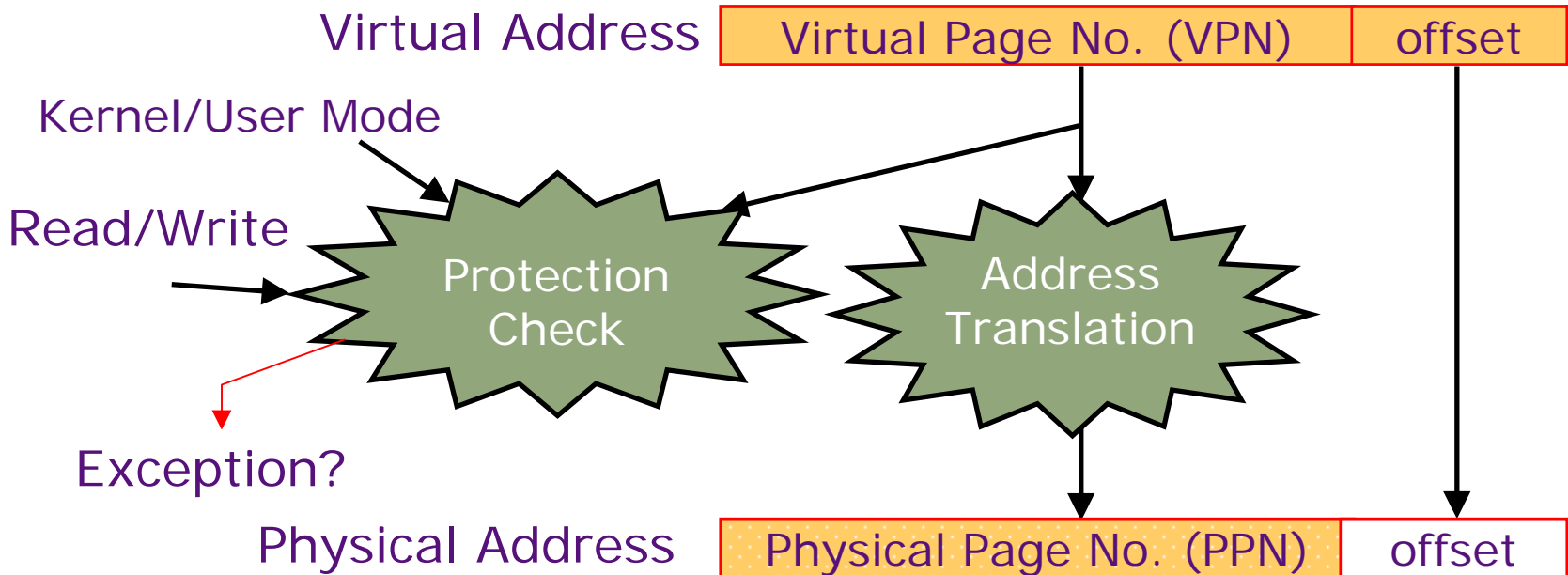


page in primary memory  
page in secondary memory



PTE of a nonexistent page

# Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

*A good VM design needs to be fast (~ one cycle) and space efficient*

# Translation Lookaside Buffers

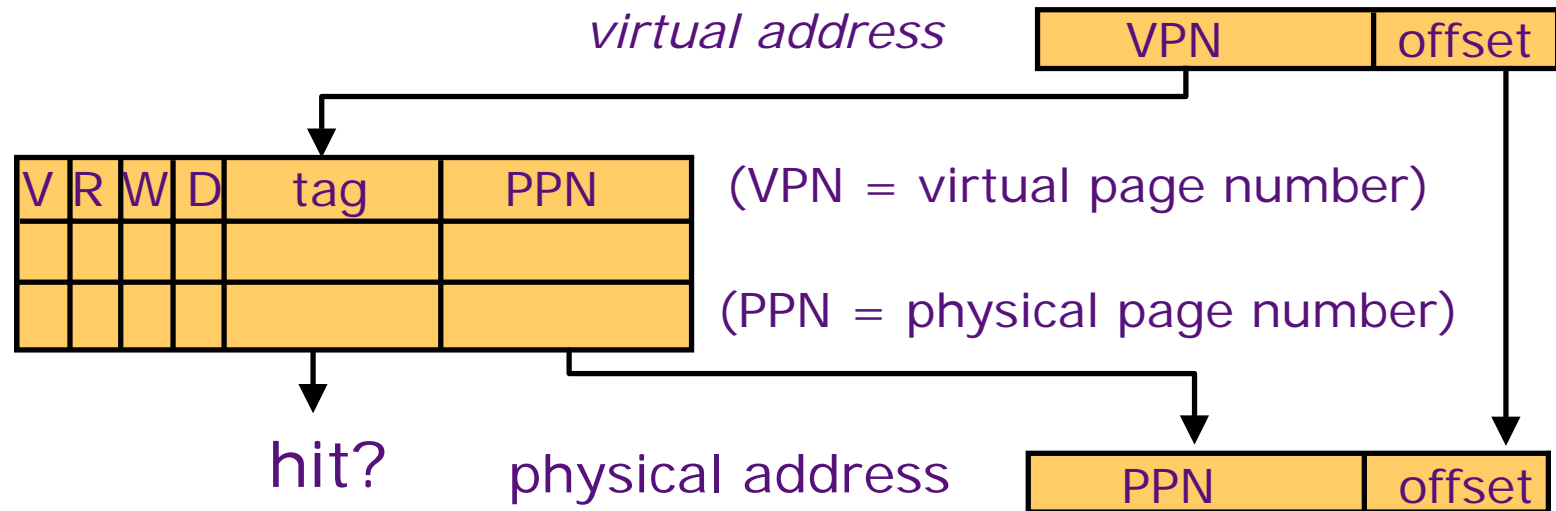
Address translation is very expensive!

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

Solution: *Cache translations in TLB*

TLB hit  $\Rightarrow$  *Single Cycle Translation*

TLB miss  $\Rightarrow$  *Page Table Walk to refill*



# 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
- Random or FIFO replacement policy
- No process information in TLB?
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = 64 entries \* 4 KB = 256 KB (if contiguous) ?

# Handling a TLB Miss

---

## Software (MIPS, Alpha)

TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

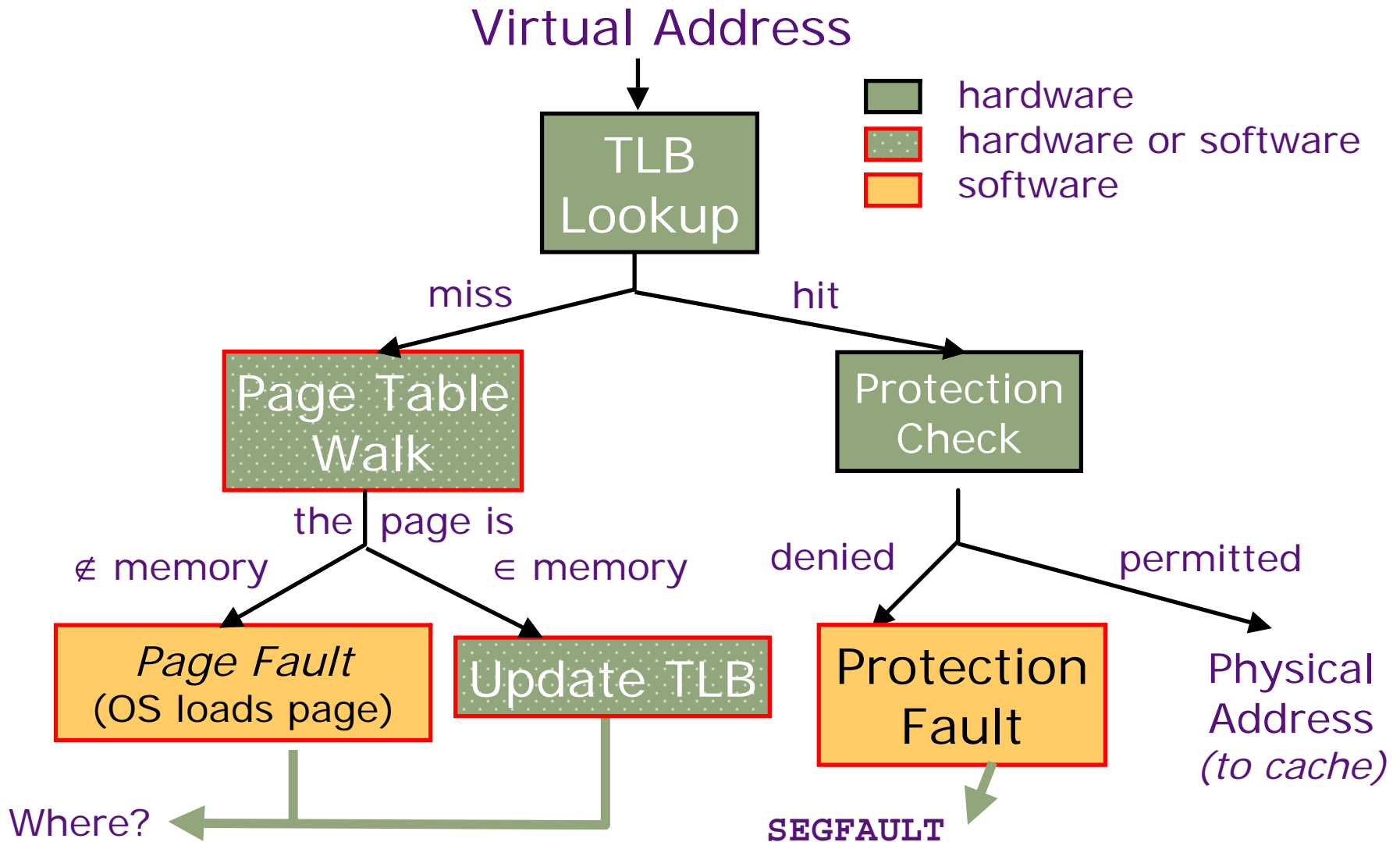
## Hardware (SPARC v8, x86, PowerPC)

A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

# Address Translation: *putting it all together*


---





# Topics

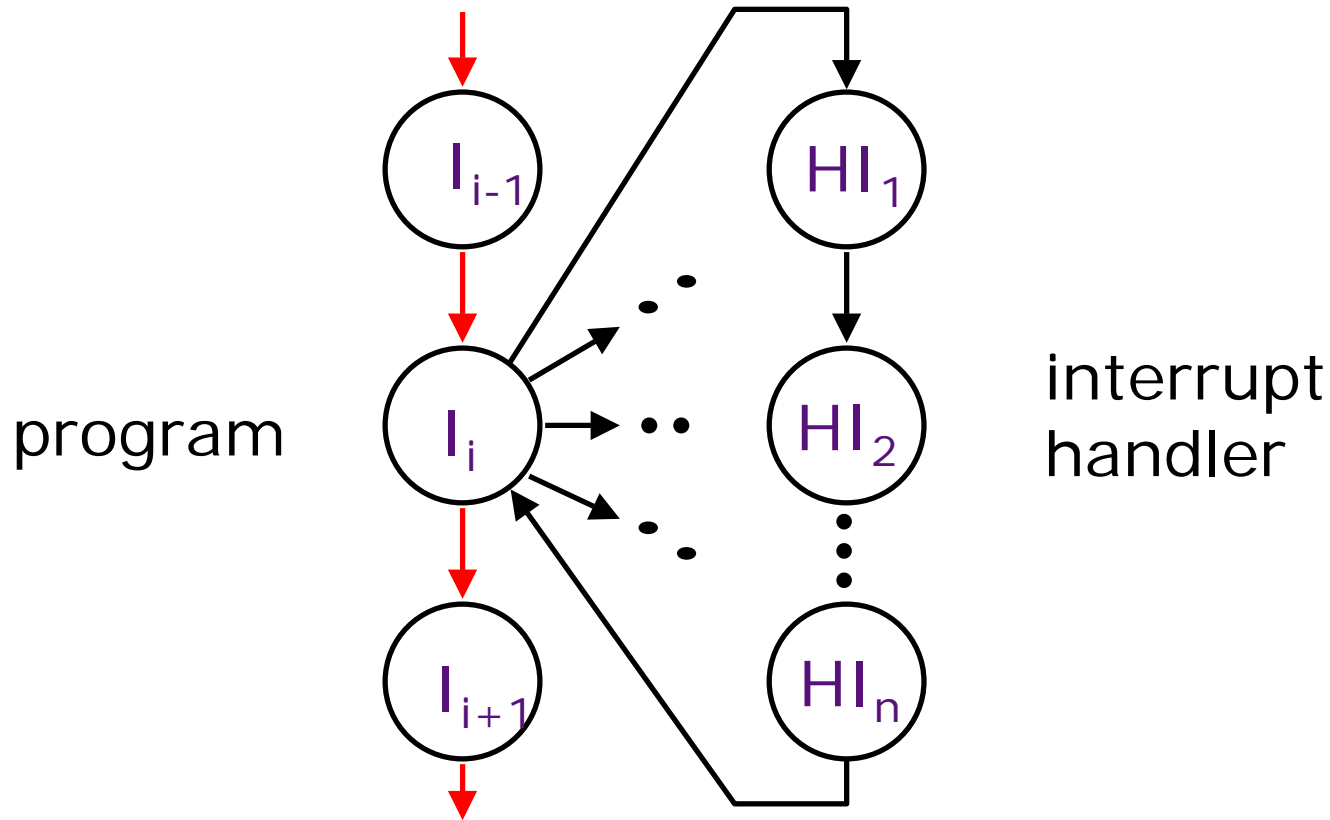
---

- Interrupts 
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

# Interrupts:

altering the normal flow of control

---



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

# Causes of Interrupts

---

Interrupt: an *event* that requests the attention of the processor

- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions*: page faults, TLB misses, protection violations
  - *traps*: system calls, e.g., jumps into kernel

# Asynchronous Interrupts:

## invoking the interrupt handler

---

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process the interrupt
  - It stops the current program at instruction  $I_i$ , completing all the instructions up to  $I_{i-1}$  (*precise interrupt*)
  - It saves the PC of instruction  $I_i$  in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

# Interrupt Handler

---

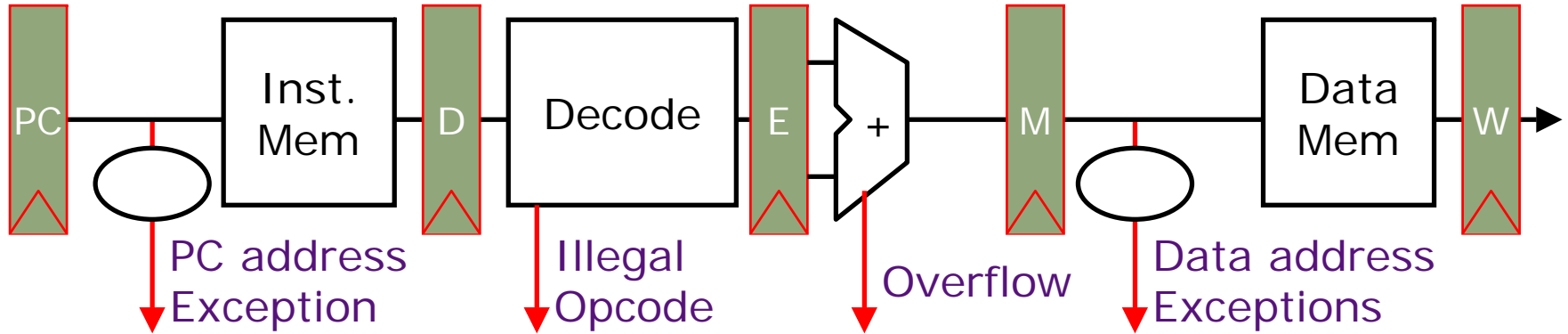
- Saves EPC before enabling interrupts to allow nested interrupts  $\Rightarrow$ 
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

# Synchronous Interrupts

---

- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In case of a trap (system call), the instruction is considered to have been completed
  - a special jump instruction involving a change to privileged kernel mode

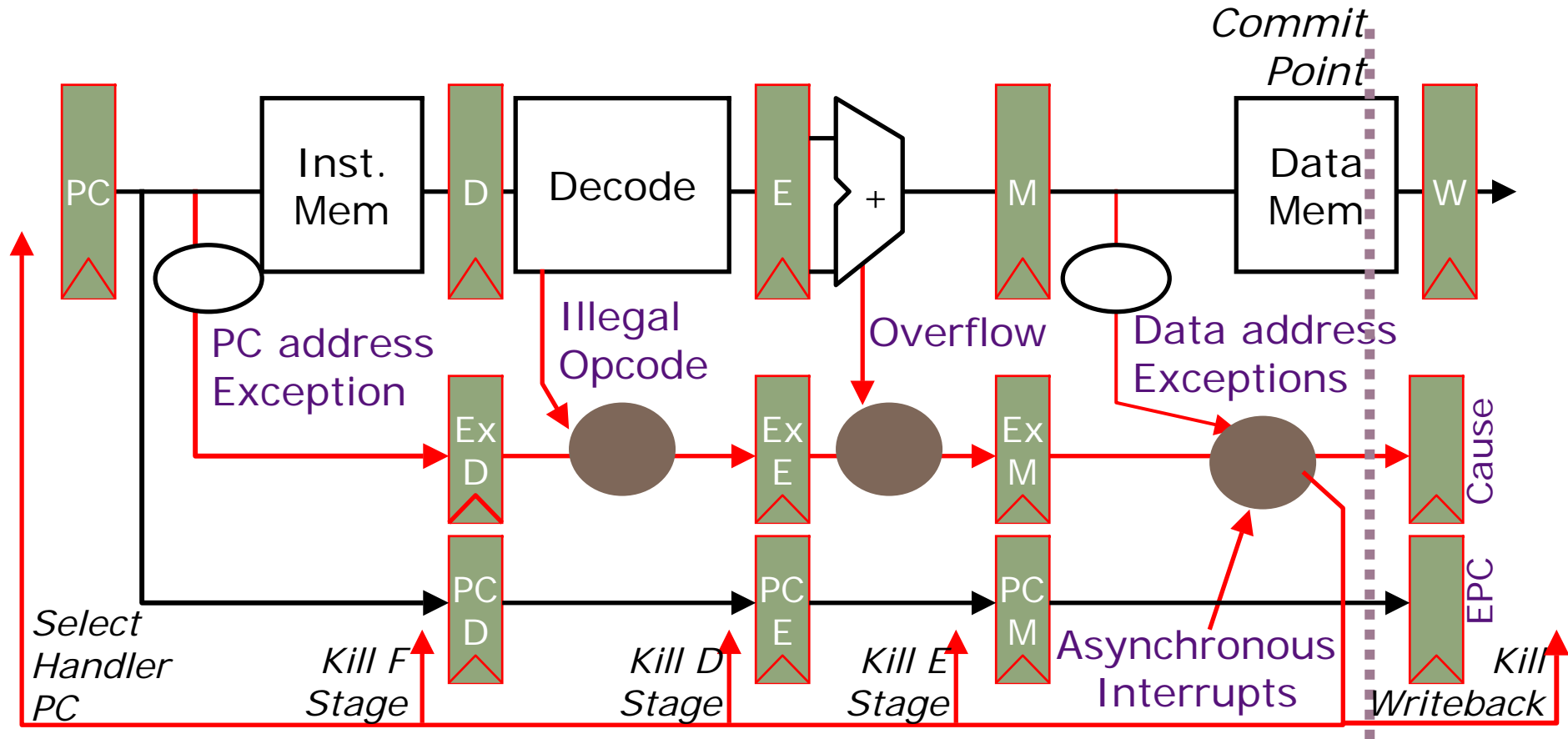
# Exception Handling 5-Stage Pipeline



Asynchronous Interrupts

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

# Exception Handling 5-Stage Pipeline





# Exception Handling 5-Stage Pipeline

---

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

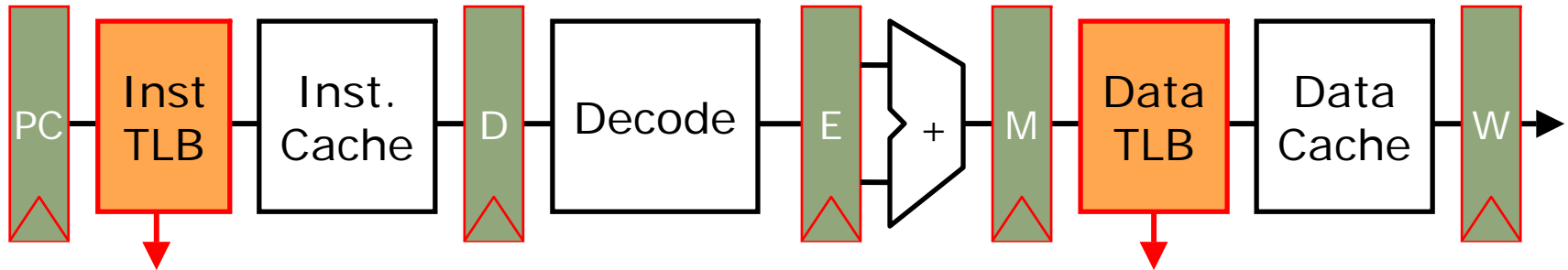
# Topics

---

- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage



# Address Translation in CPU Pipeline

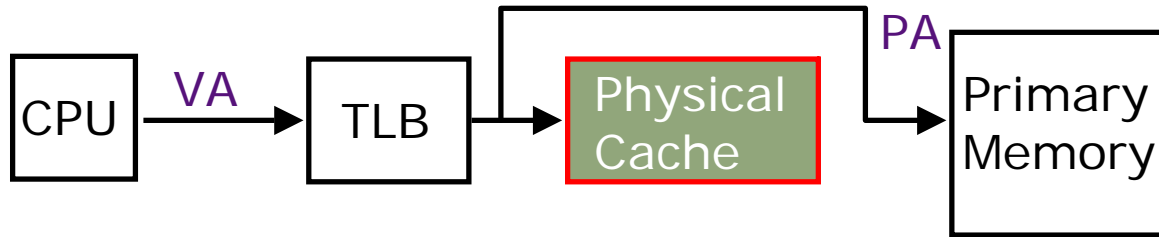


*TLB miss? Page Fault?  
Protection violation?*

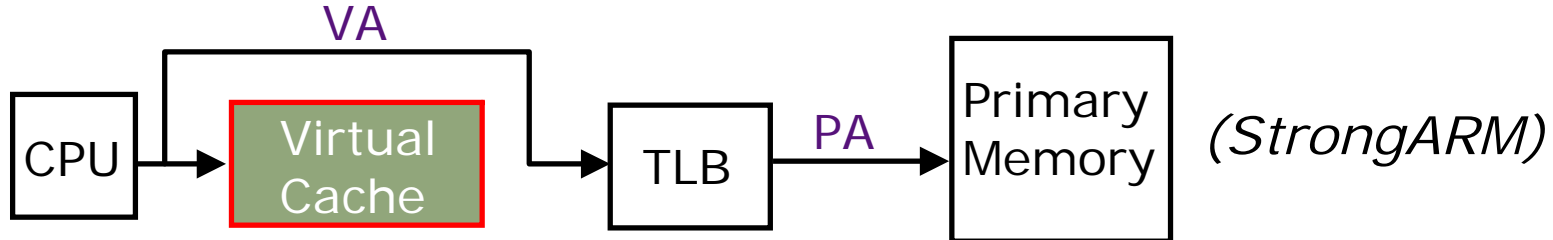
*TLB miss? Page Fault?  
Protection violation?*

- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - virtual address caches
  - parallel TLB/cache access

# Virtual Address Caches

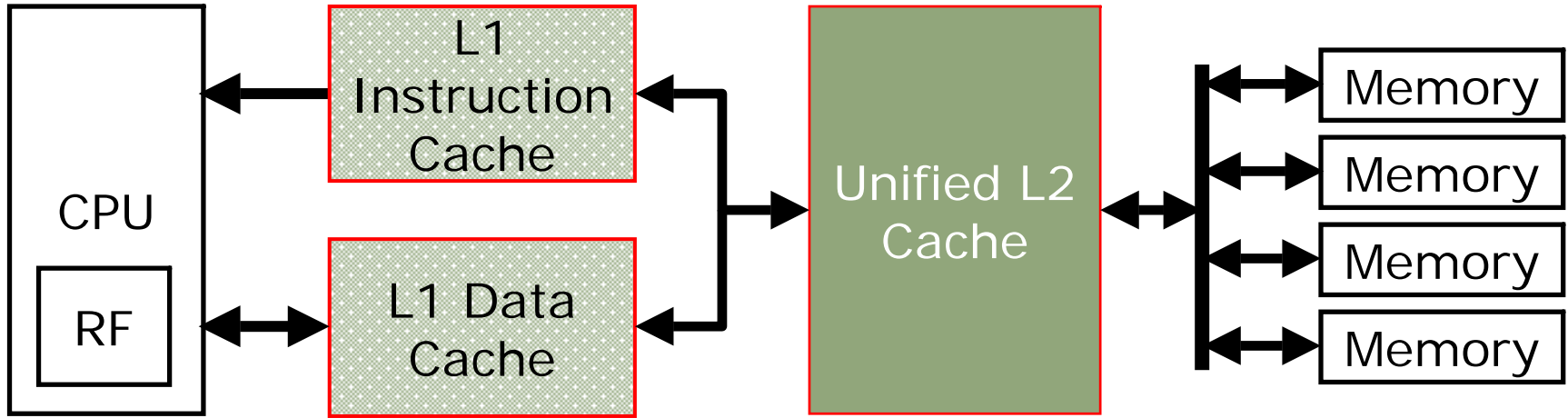


*Alternative: place the cache before the TLB*



- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)

# A solution via Second Level Cache



Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is "inclusive" of both Instruction and Data caches

# Topics

---

- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Speeding up page table walks
- Modern Usage



# Page Fault Handler

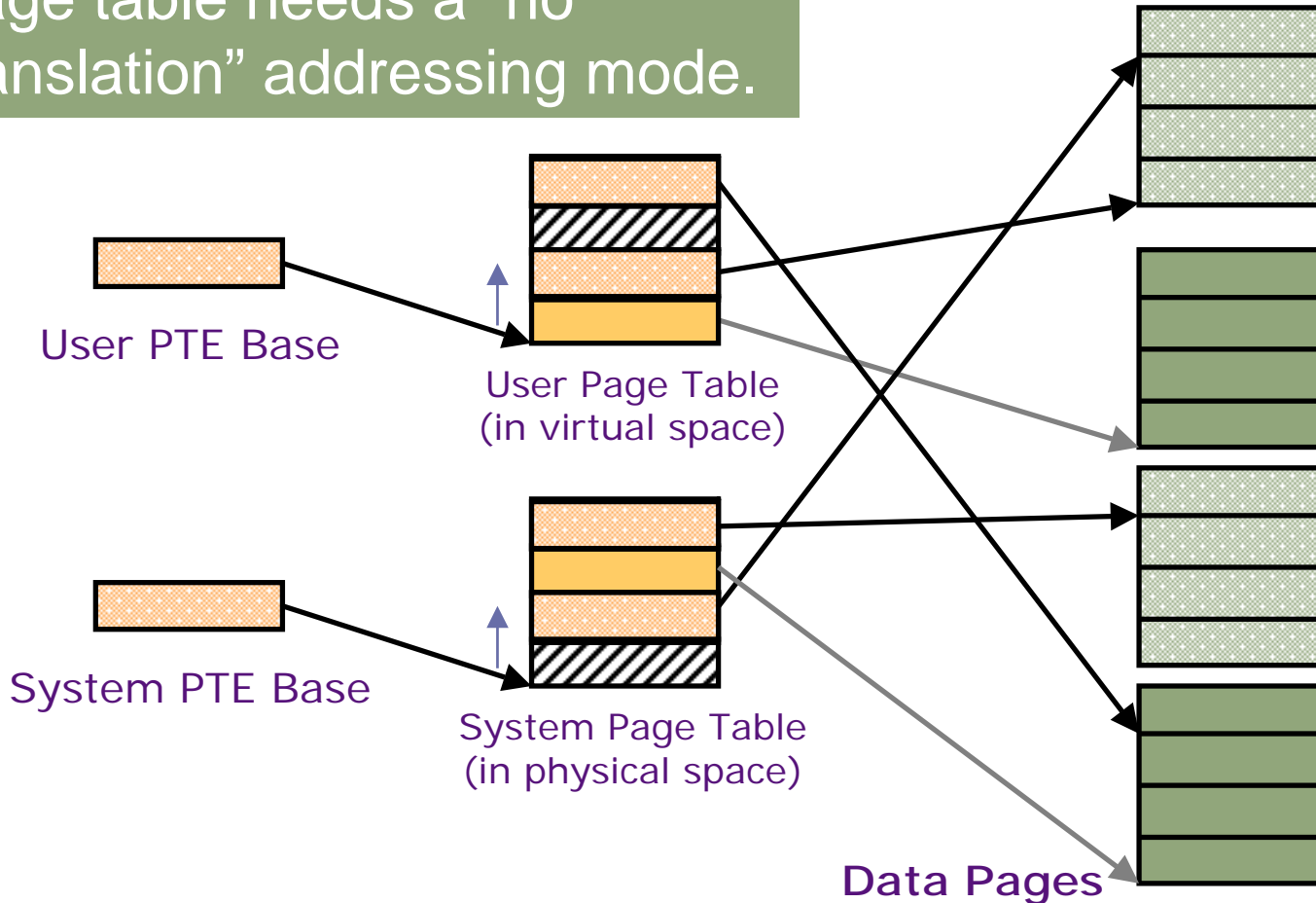
---

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk*
  - If no free pages are left, a page is swapped out
    - Pseudo-LRU replacement policy*
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

# Translation for Page Tables

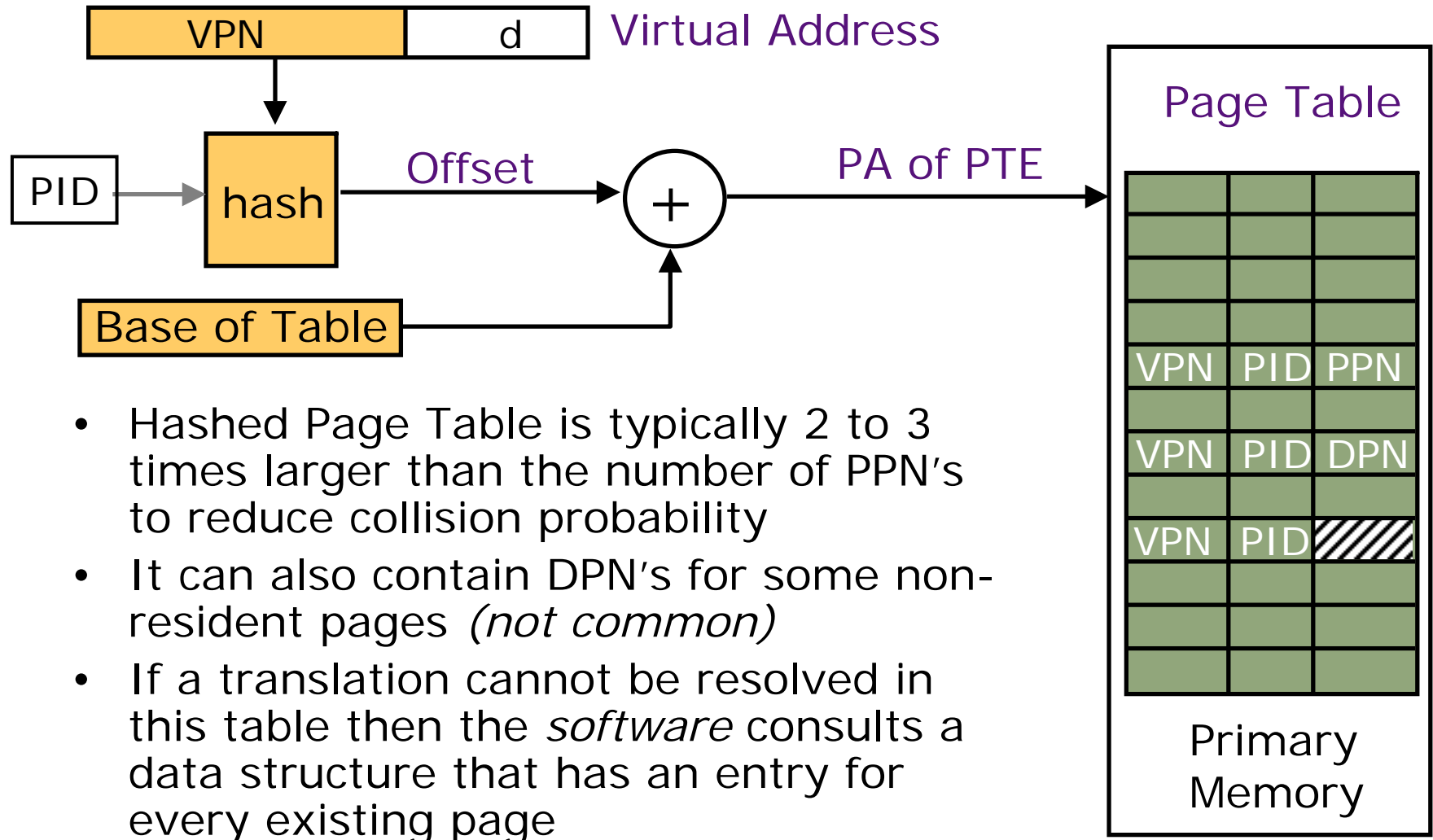
- Can references to page tables cause TLB misses?

A program that traverses the page table needs a “no translation” addressing mode.



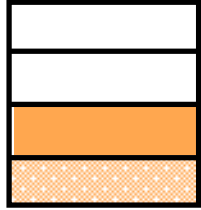


# Hashed Page Table: Approximating Associative Addressing

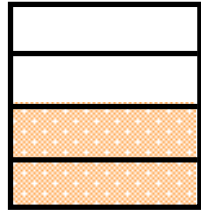


# Swapping a Page of a Page Table

---



A PTE in primary memory contains  
primary or secondary memory addresses



A PTE in secondary memory contains  
*only* secondary memory addresses

⇒ a page of a PT can be swapped out only  
if none its PTE's point to pages in the  
primary memory

# Virtual Memory Use Today - 1

---

- Desktops/servers have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Vector supercomputers have translation and protection but not demand-paging  
(Older Crays: base&bound, Japanese & Cray X1: pages)
  - Don't waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions

# Virtual Memory Use Today - 2

---

- Most embedded processors and DSPs provide physical addressing only
  - Can't afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures

*Given the software demands of modern embedded devices (e.g., cell phones, PDAs) all this may change in the near future!*