Memory Management: *From Absolute Addresses to Demand Paging*

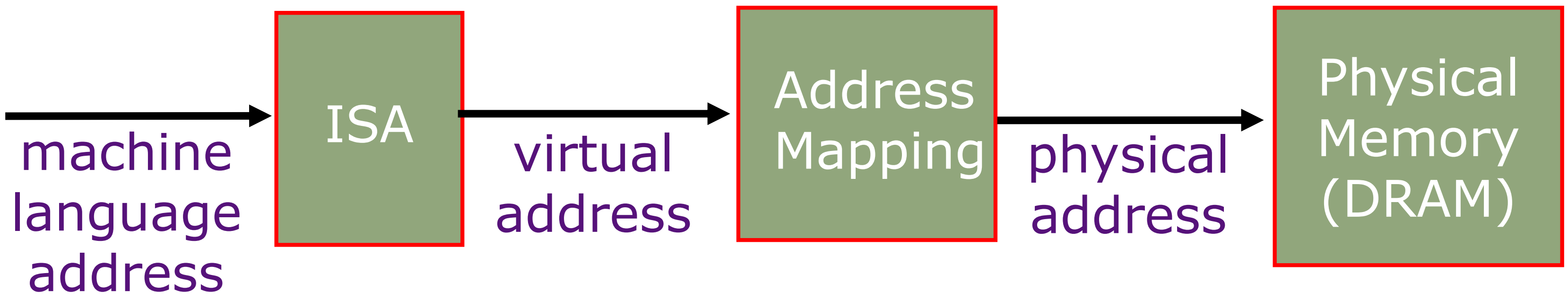
Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823

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

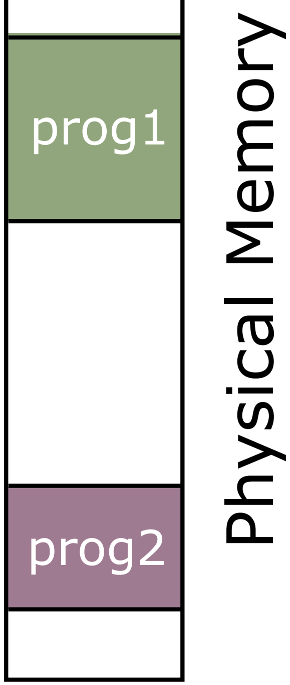
Dynamic Address Translation

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

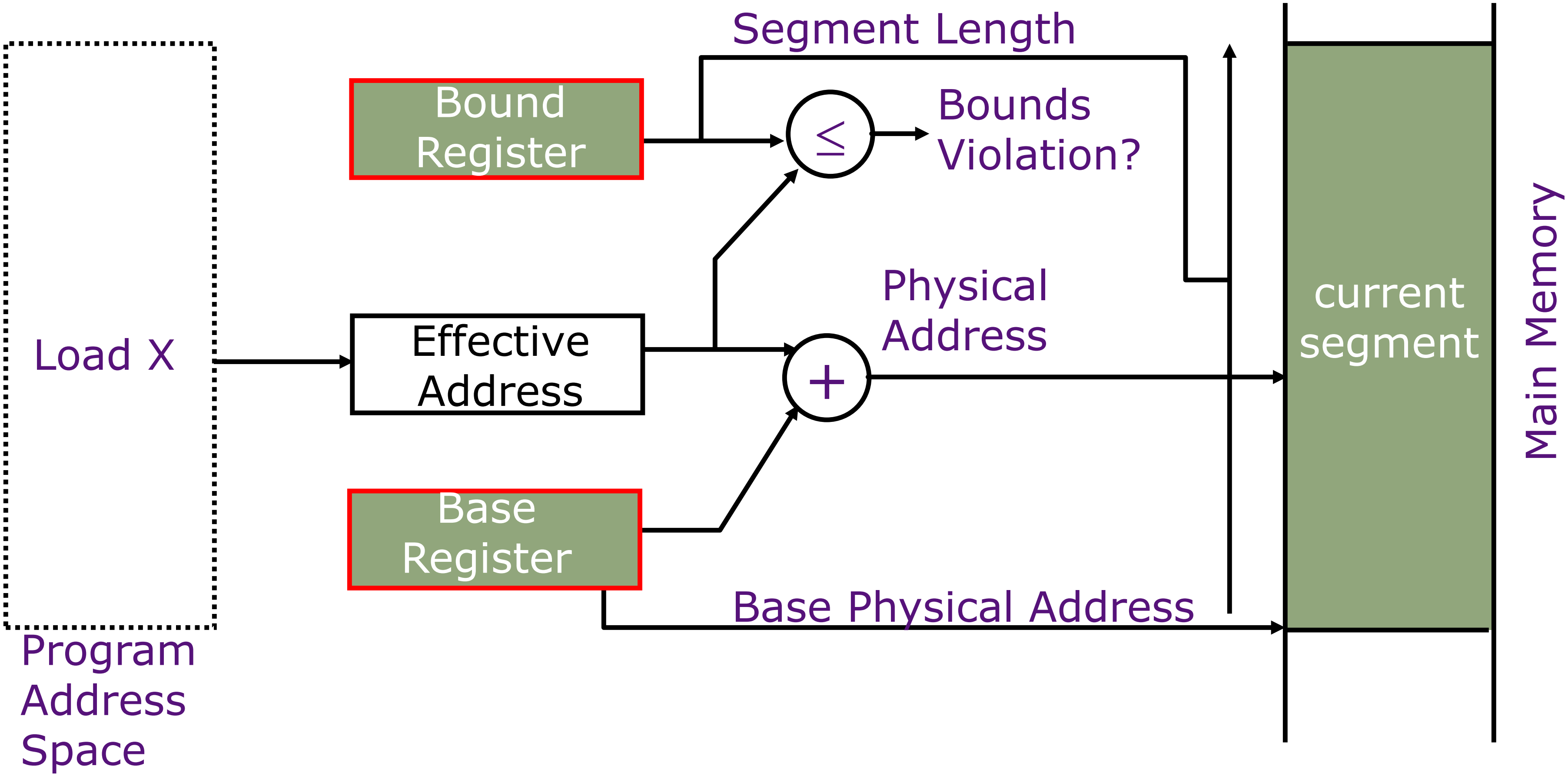
⇒ need for a *base register*

Protection

Independent programs should not affect each other inadvertently

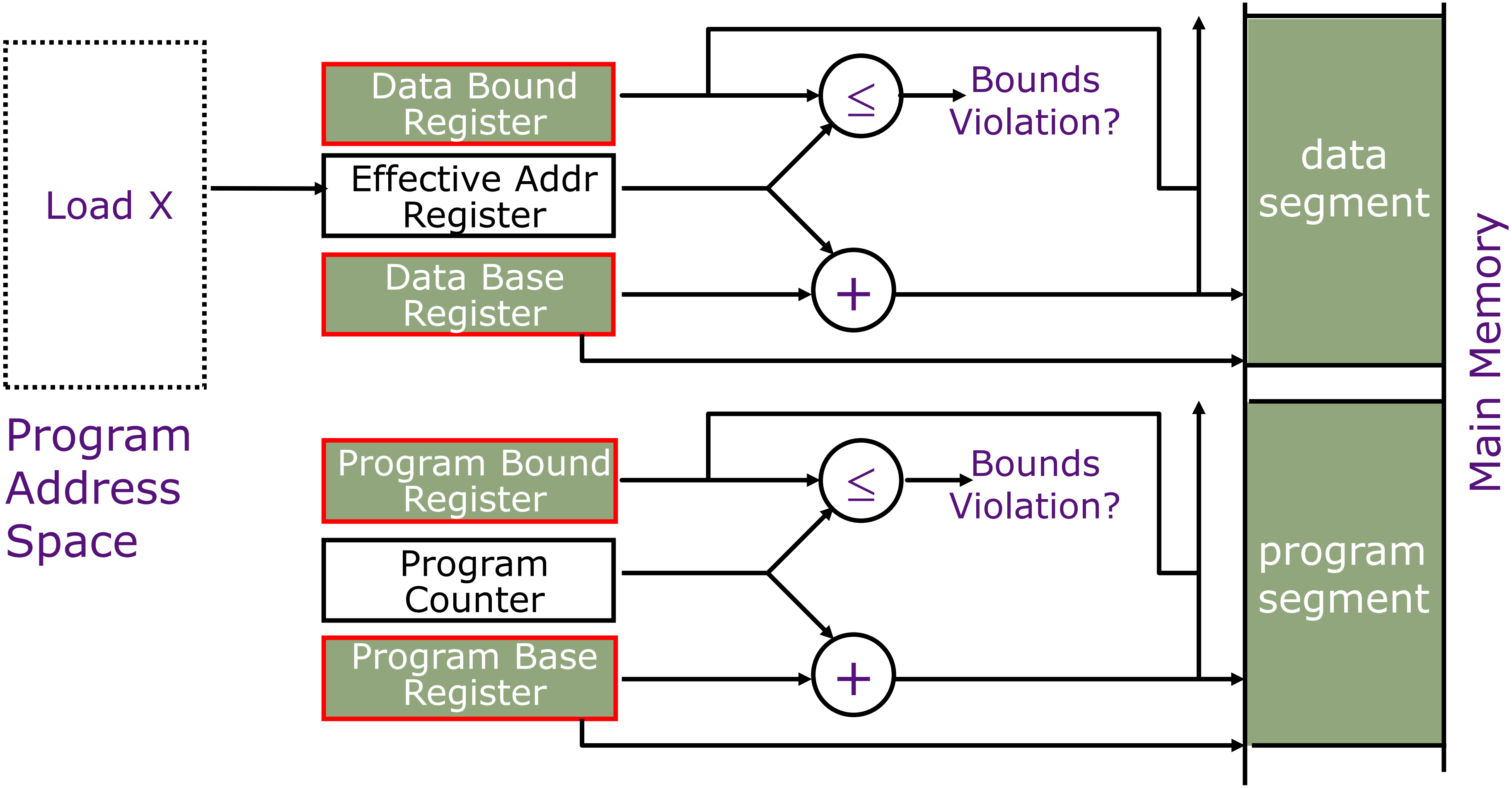
⇒ need for a *bound register*

Simple Base and Bound Translation



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

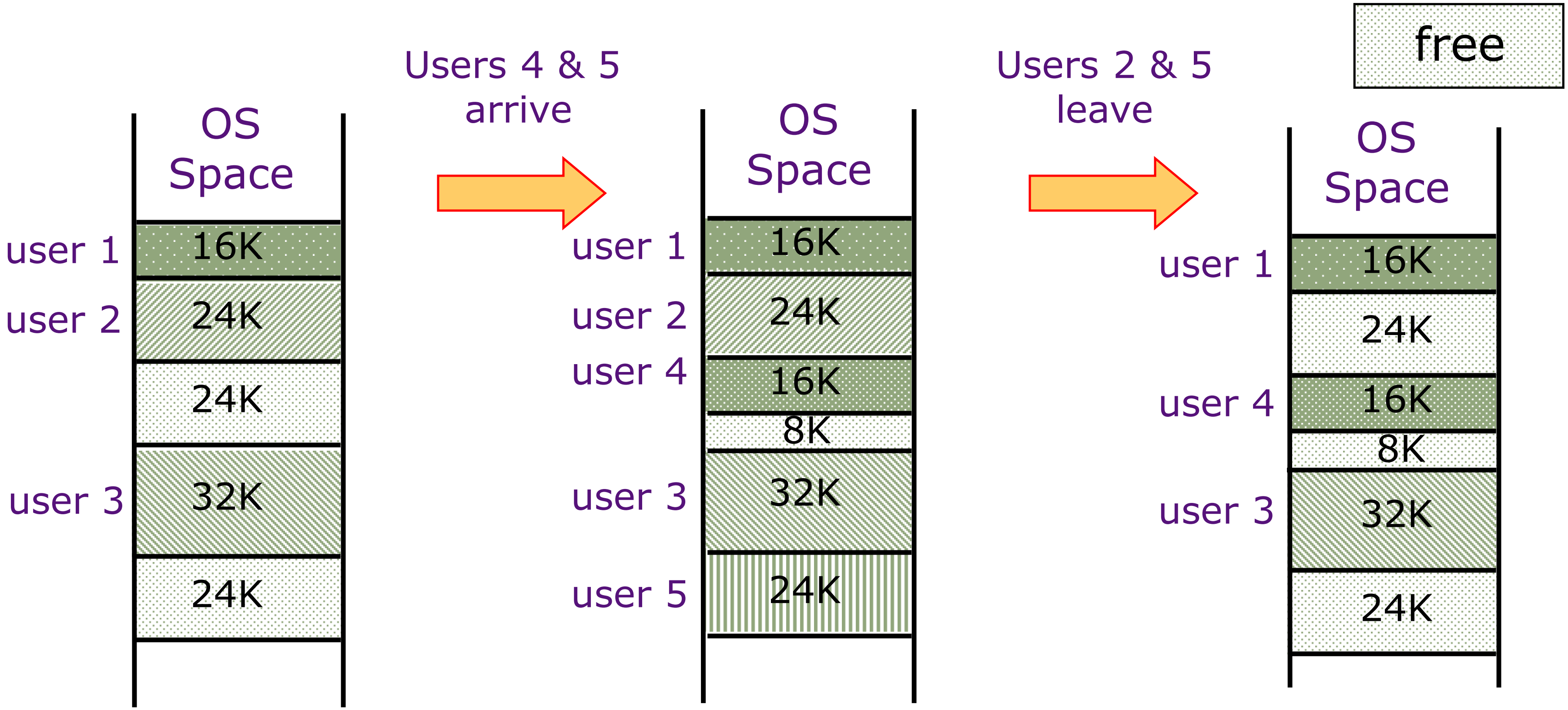
Separate Areas for Program and Data



What is an advantage of this separation?

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

Memory Fragmentation



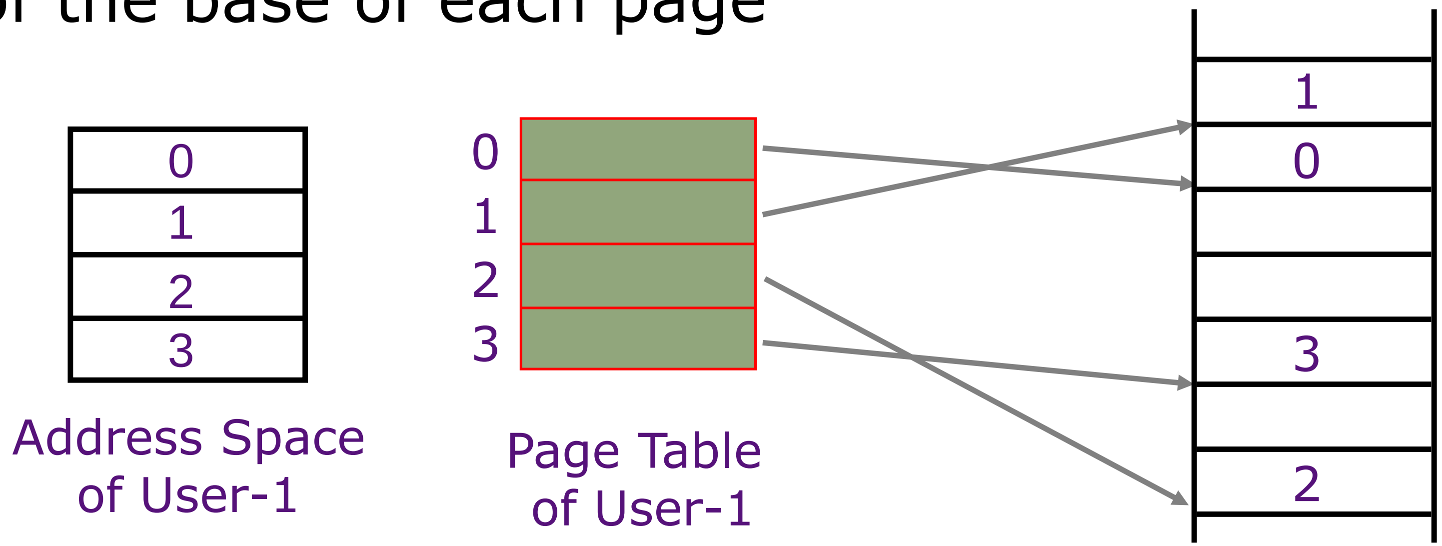
As users come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

- Processor generated address can be interpreted as a pair <page number, offset>

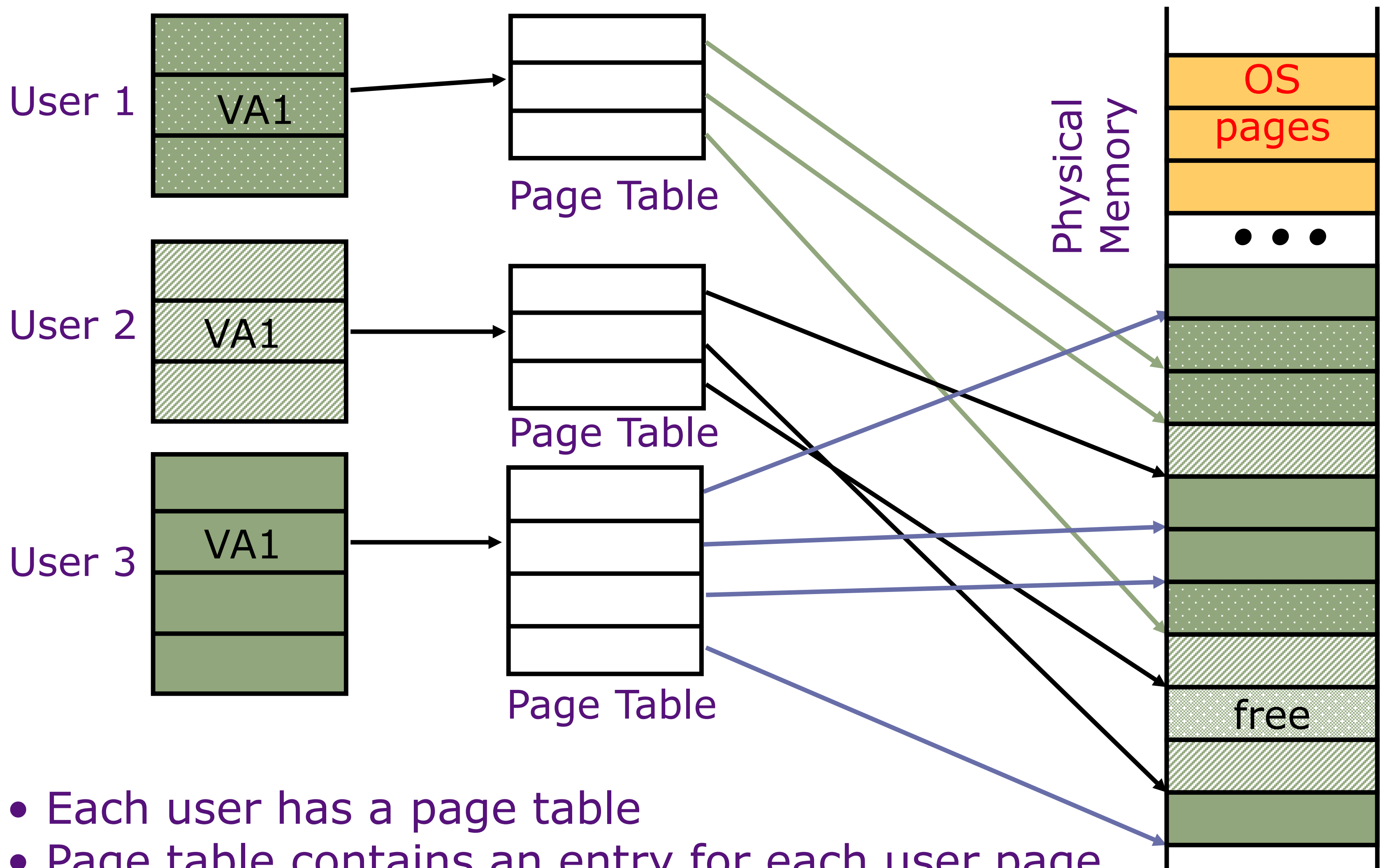
page number	offset

- 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



- Each user has a page table
- Page table contains an entry for each user page

Where Should Page Tables Reside?

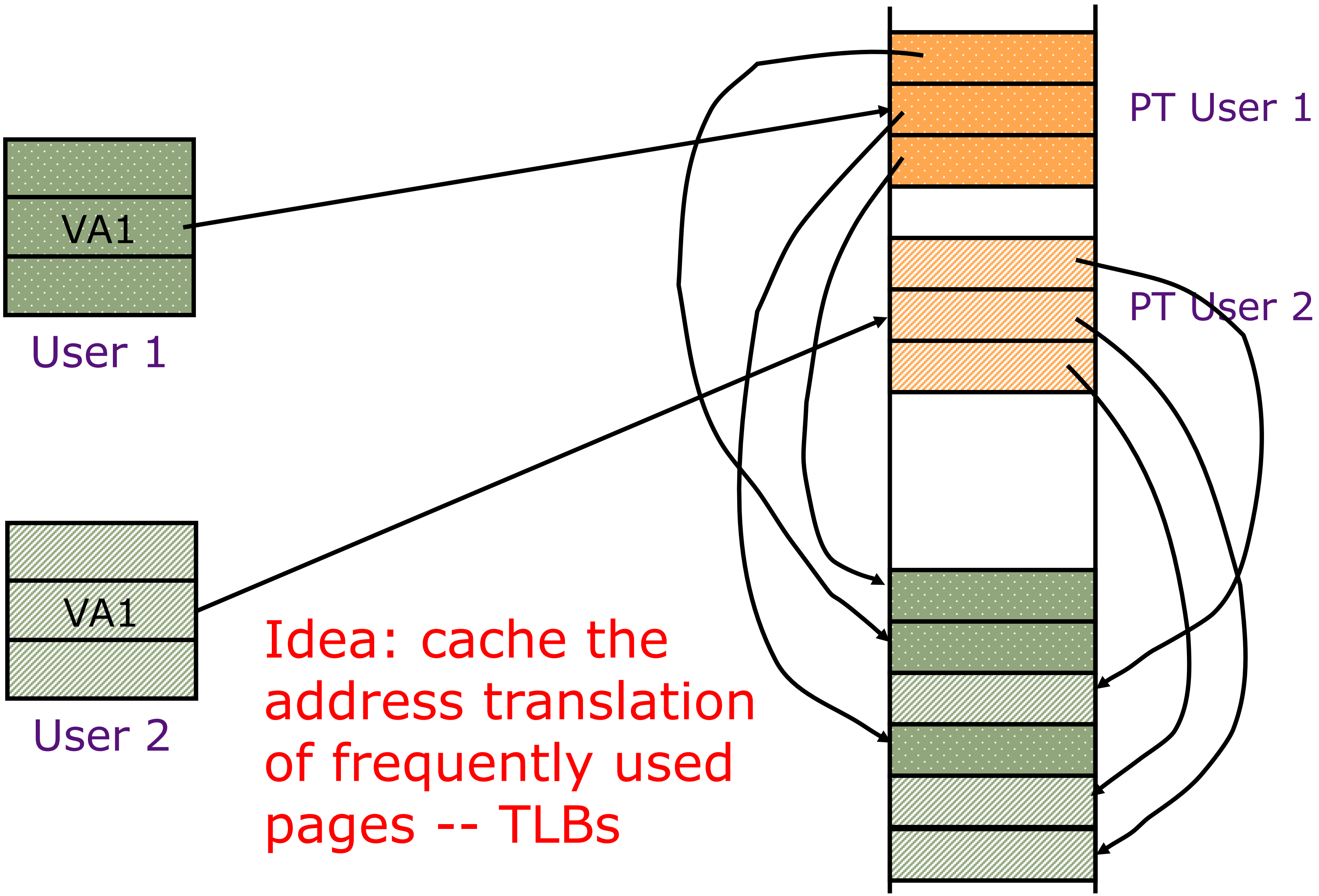
- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - $\Rightarrow$ Space requirement is large
  - $\Rightarrow$ Too expensive to keep in registers

- 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

    $\Rightarrow$ *doubles the number of memory references!*

Page Tables in Physical Memory



Idea: cache the address translation of frequently used pages -- TLBs

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*

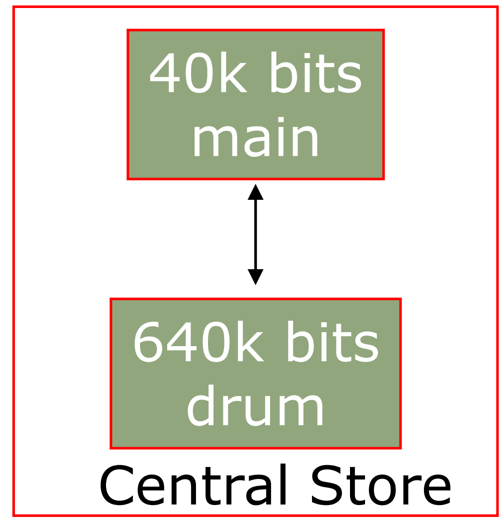
- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

  Brooker's interpretive coding, 1960



Ferranti Mercury 1956

Problems?

Method1: Difficult, error prone
Method2: Inefficient

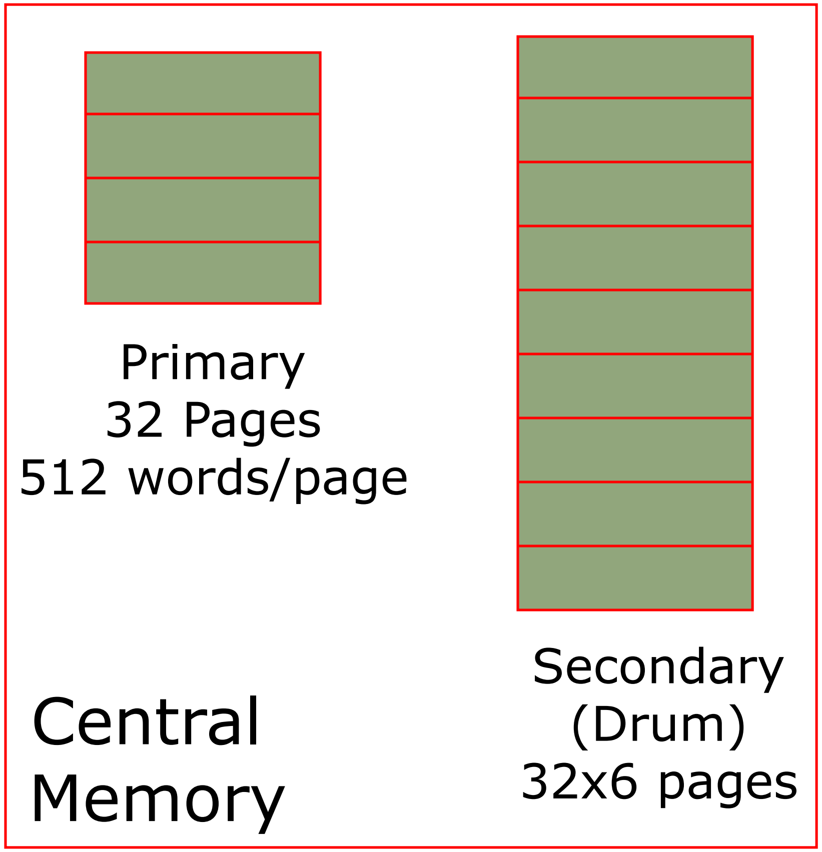
Demand Paging in Atlas (1962)

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

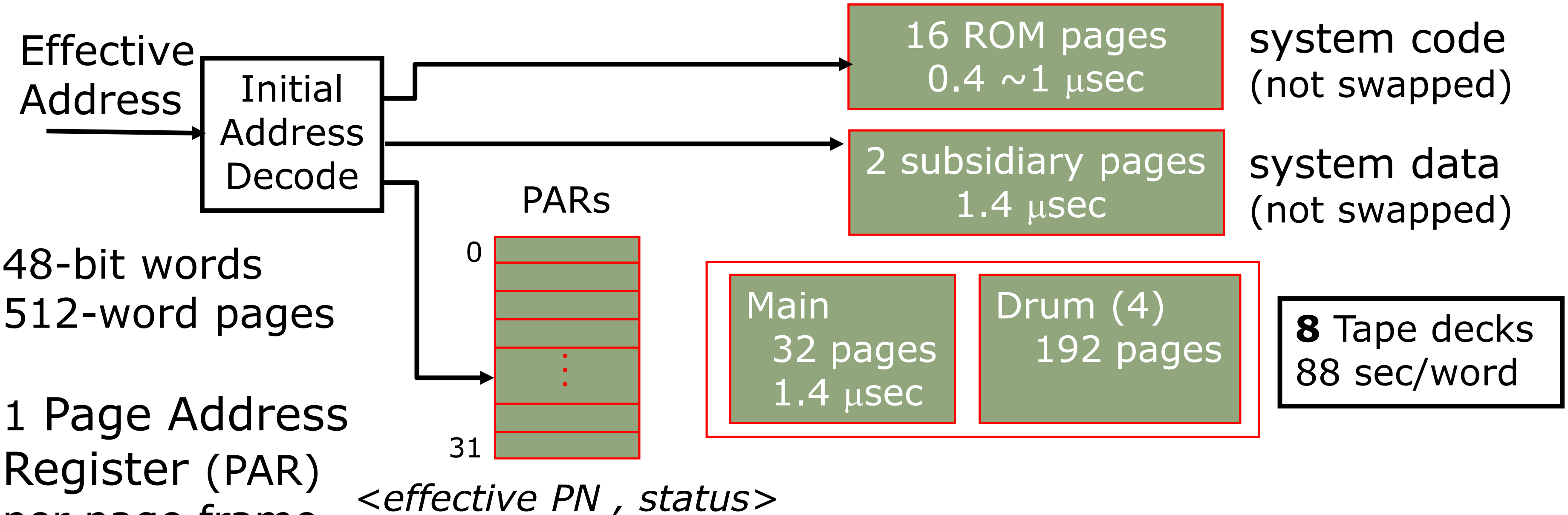
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



Hardware Organization of Atlas



48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

Compare the effective page address against all 32 PARs

match $\Rightarrow$ normal access

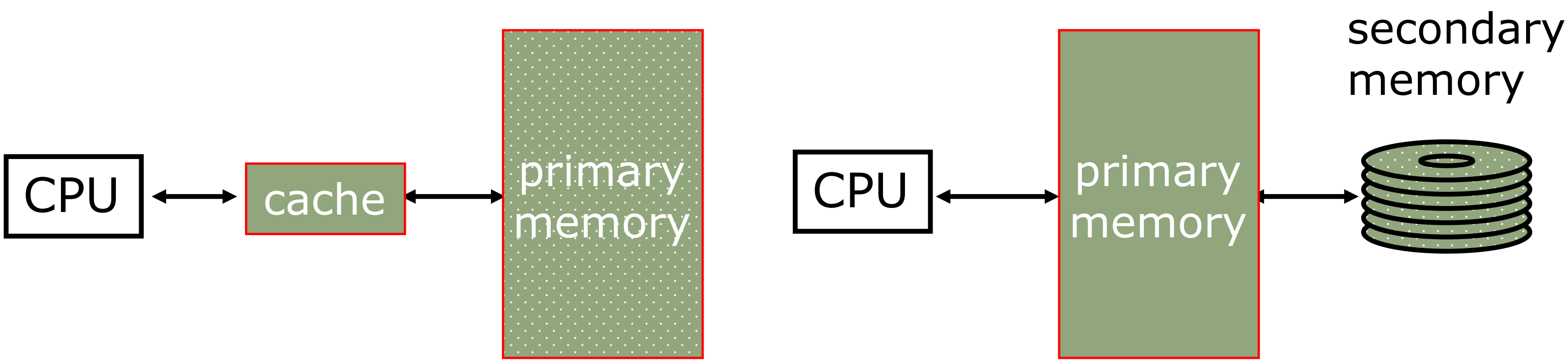
no match $\Rightarrow$ *page fault*

save the state of the partially executed instruction

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	*Demand paging*
cache entry	page frame
cache block (~32 bytes)	page (~4K bytes)
cache miss rate (1% to 20%)	page miss rate (<0.001%)
cache hit (~1 cycle)	page hit (~100 cycles)
cache miss (~100 cycles)	page miss (~5M cycles)
a miss is handled in *hardware*	a miss is handled mostly in *software*

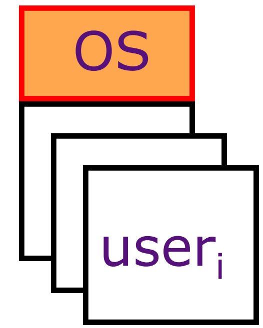
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

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

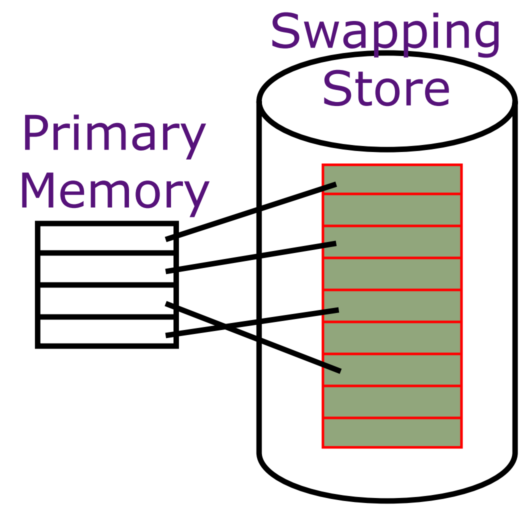
page table ≡ name space



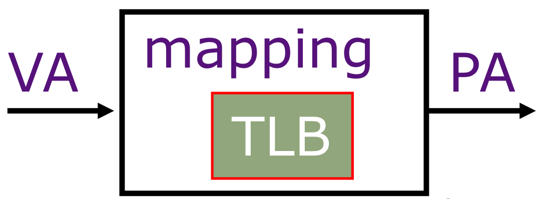
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

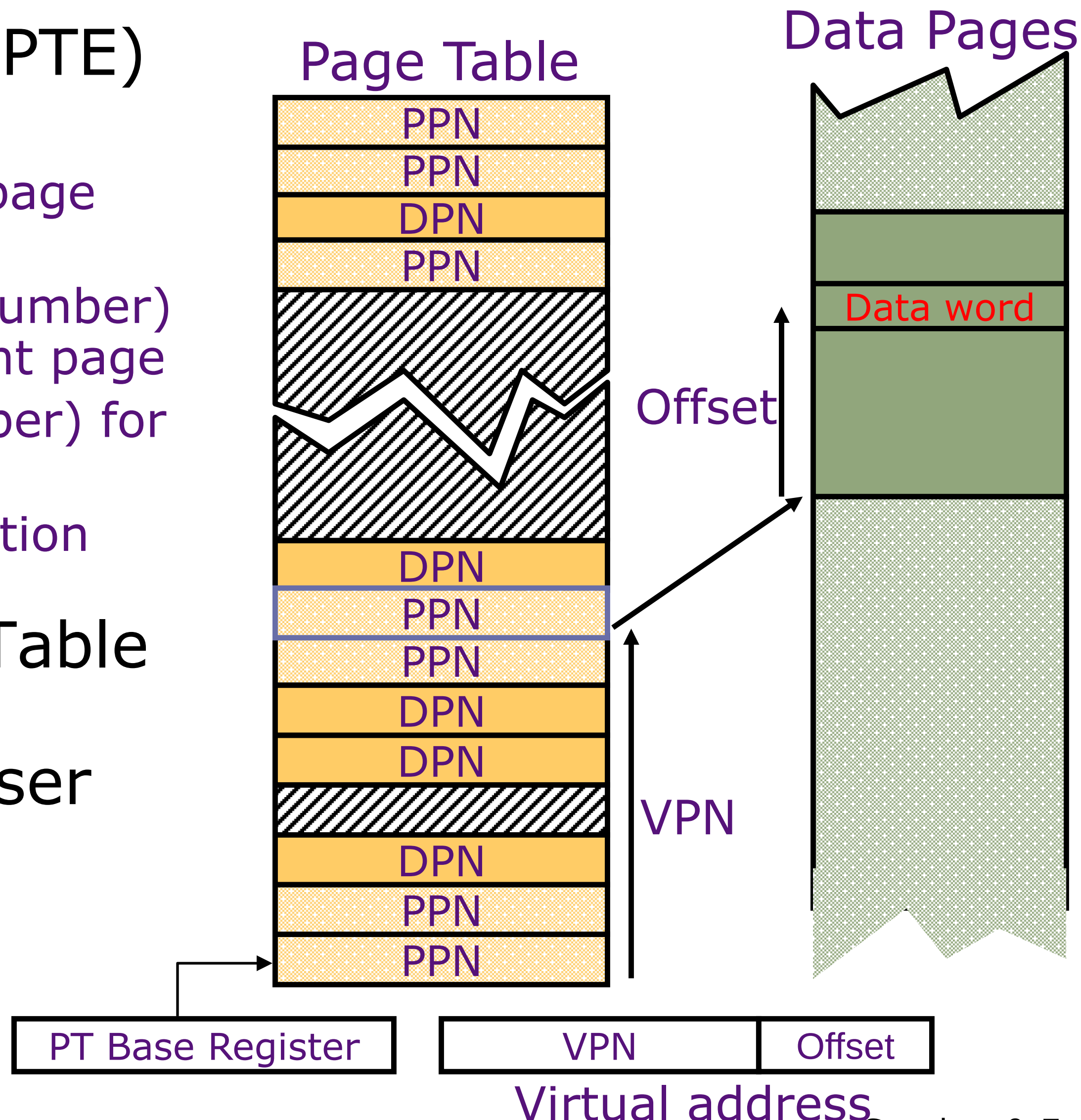


The price is address translation on each memory reference



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:

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

$\Rightarrow$ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

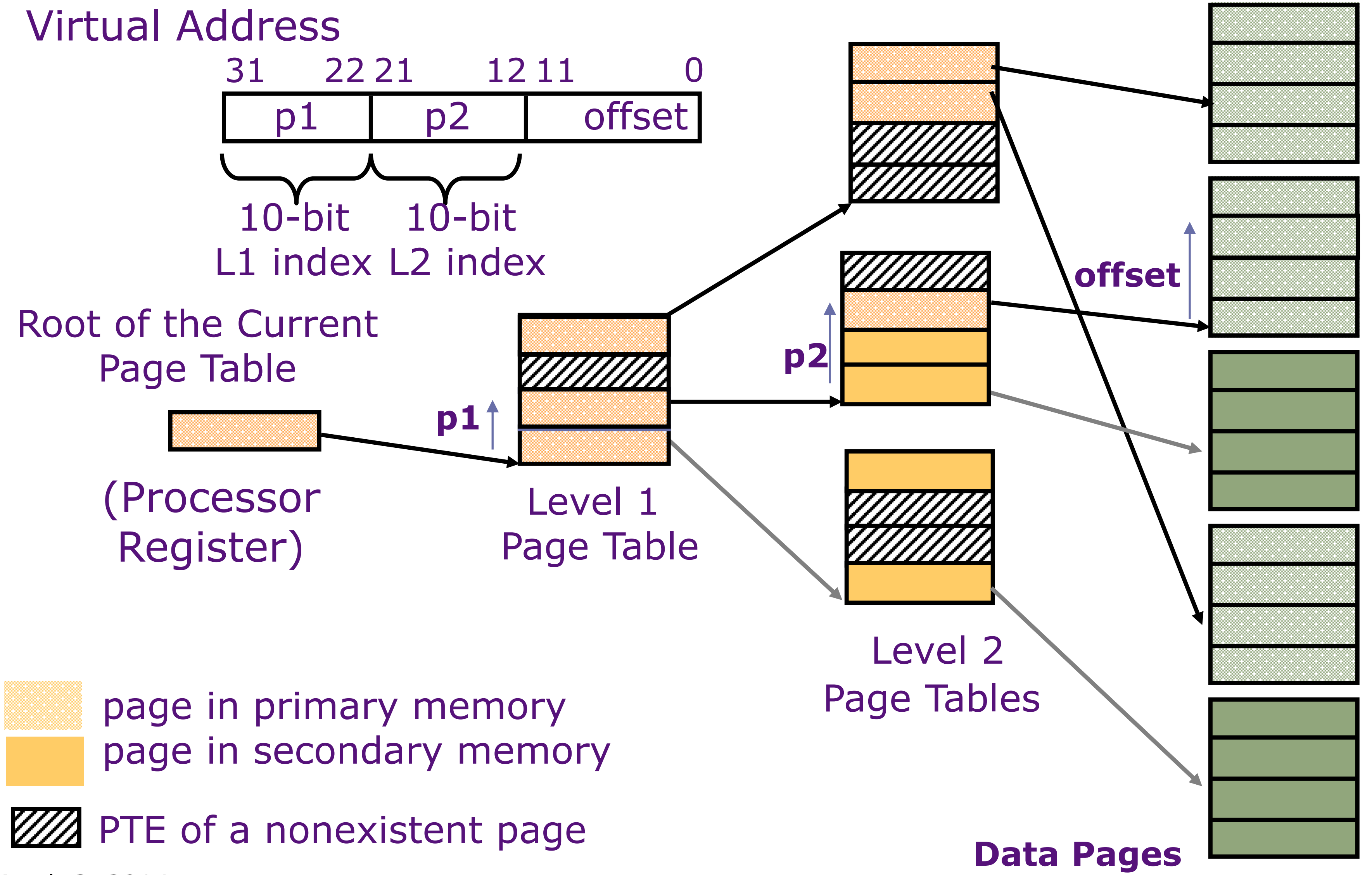
- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

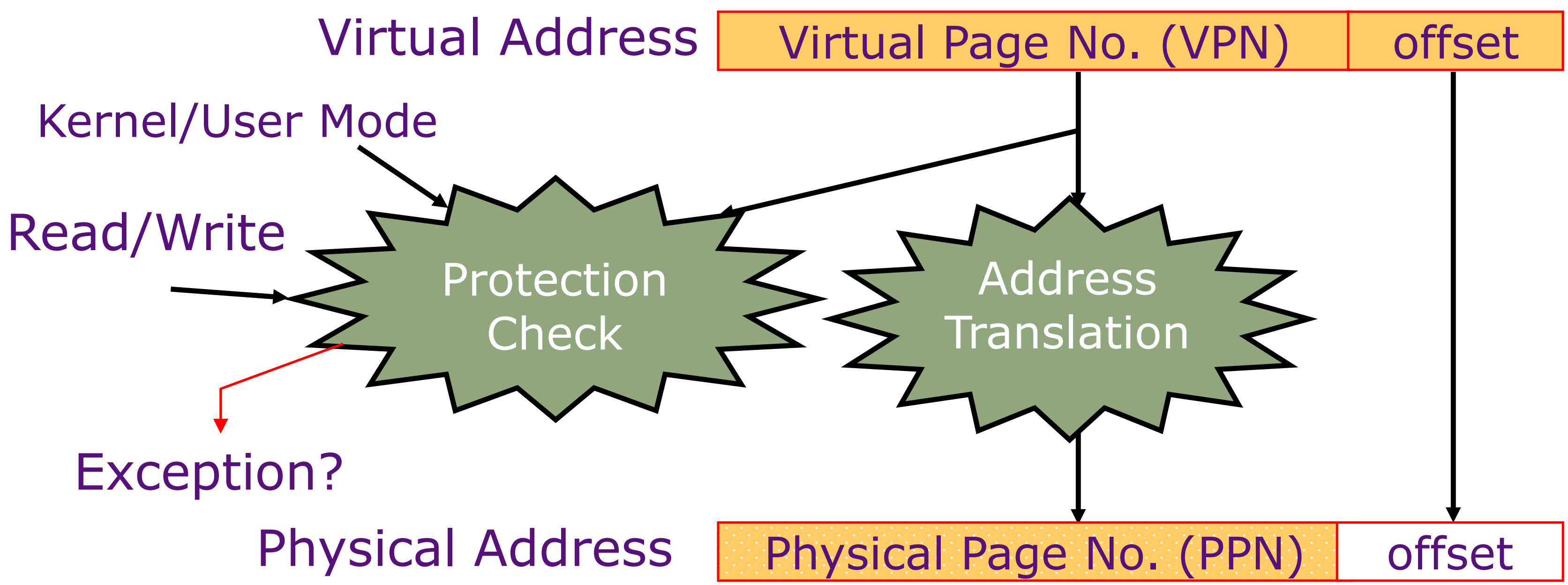
- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

What is the "saving grace" ?

Hierarchical Page Table



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

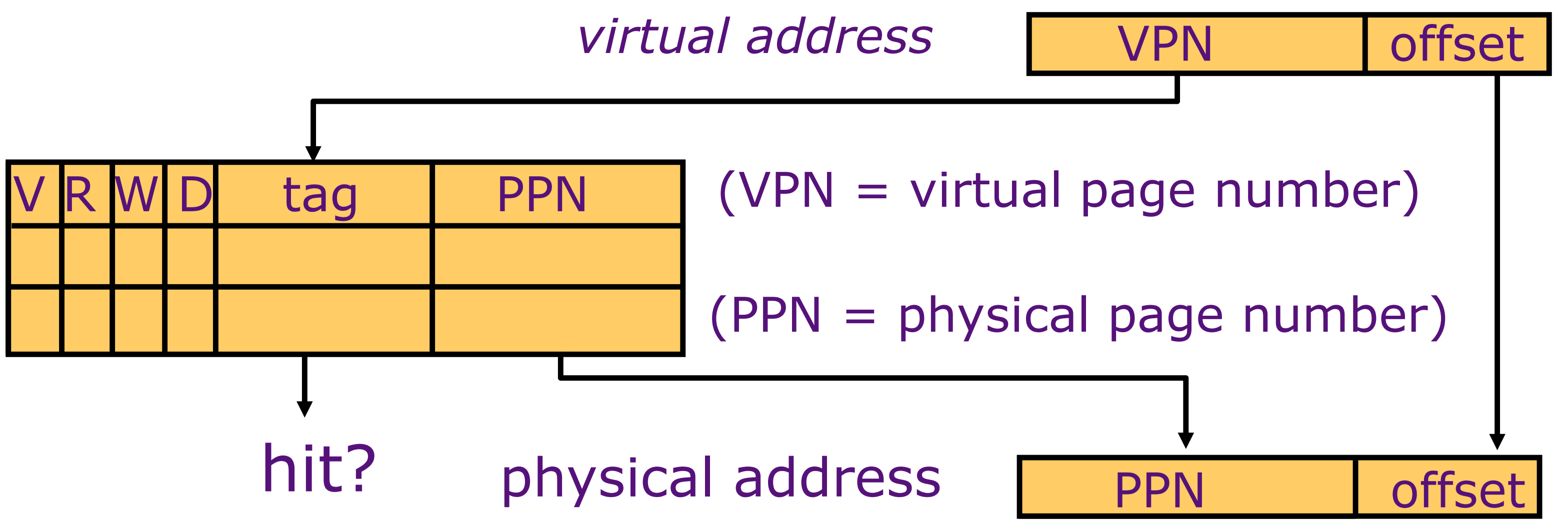
Address translation is very expensive!

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*



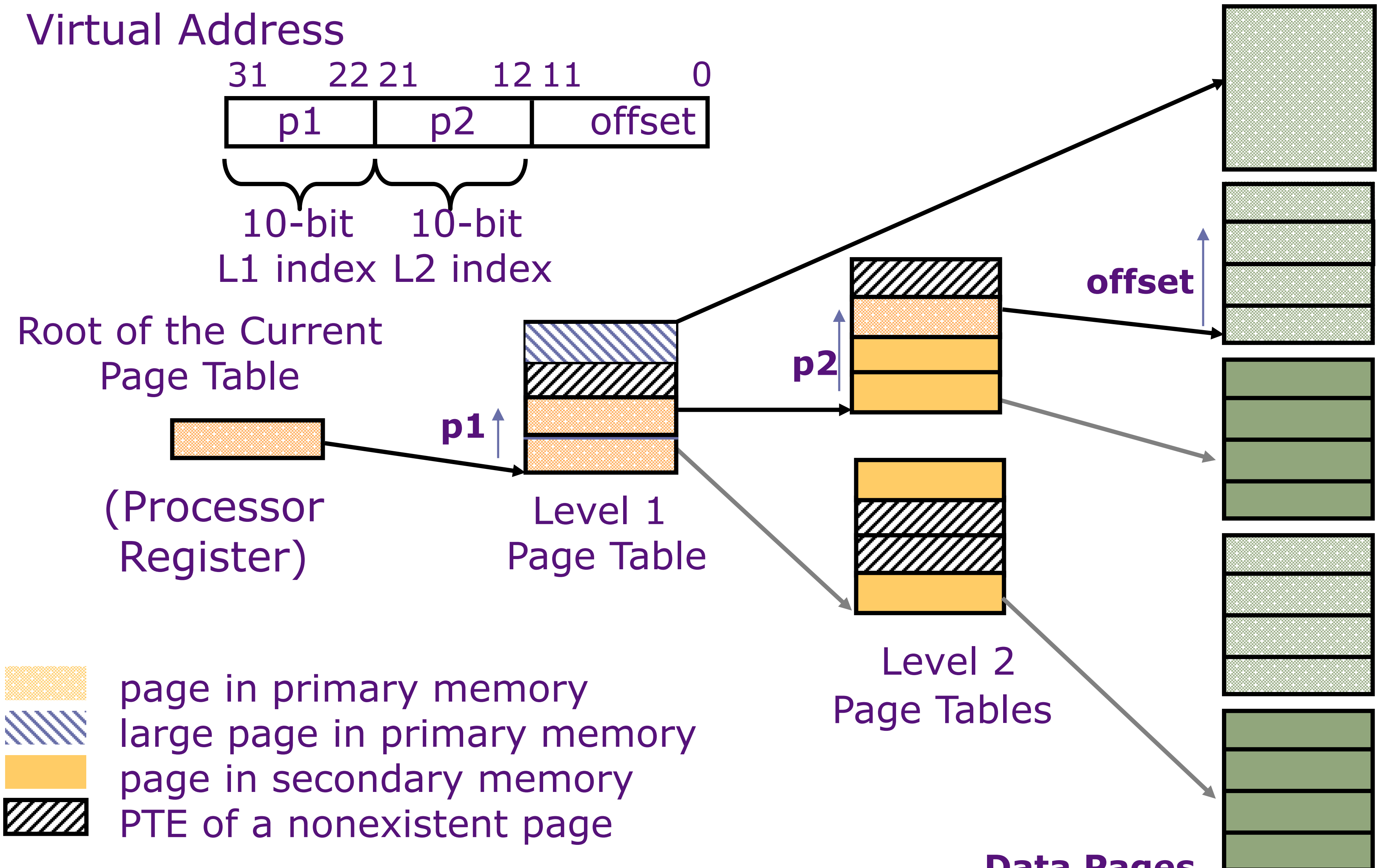
TLB Designs

- 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

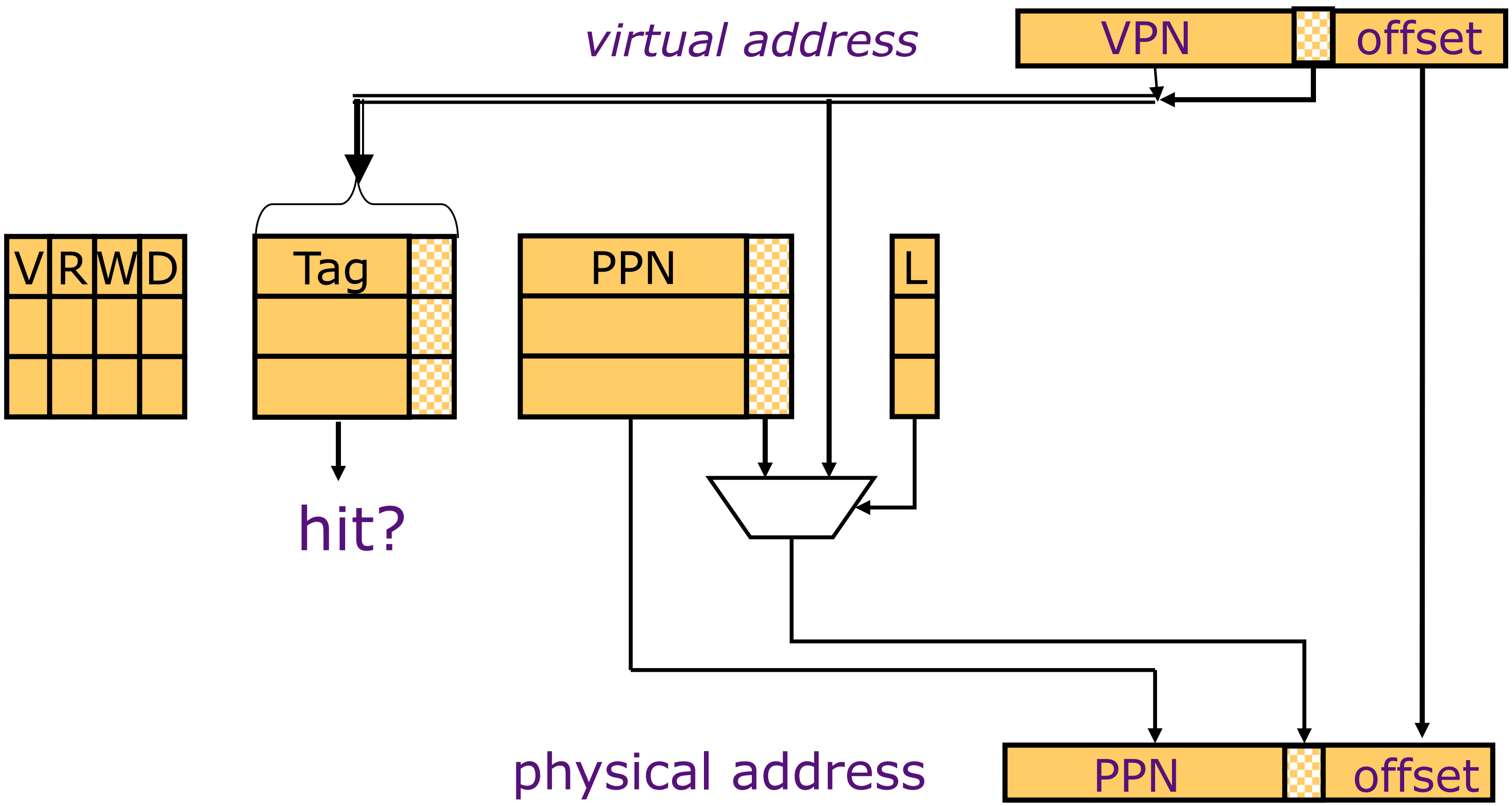
TLB Reach = *64 entries * 4 KB = 256 KB (if contiguous)* ?

Variable-Sized Page Support



Variable-Size Page TLB

Some systems support multiple page sizes.



Handling a TLB Miss

Software (MIPS, Alpha)

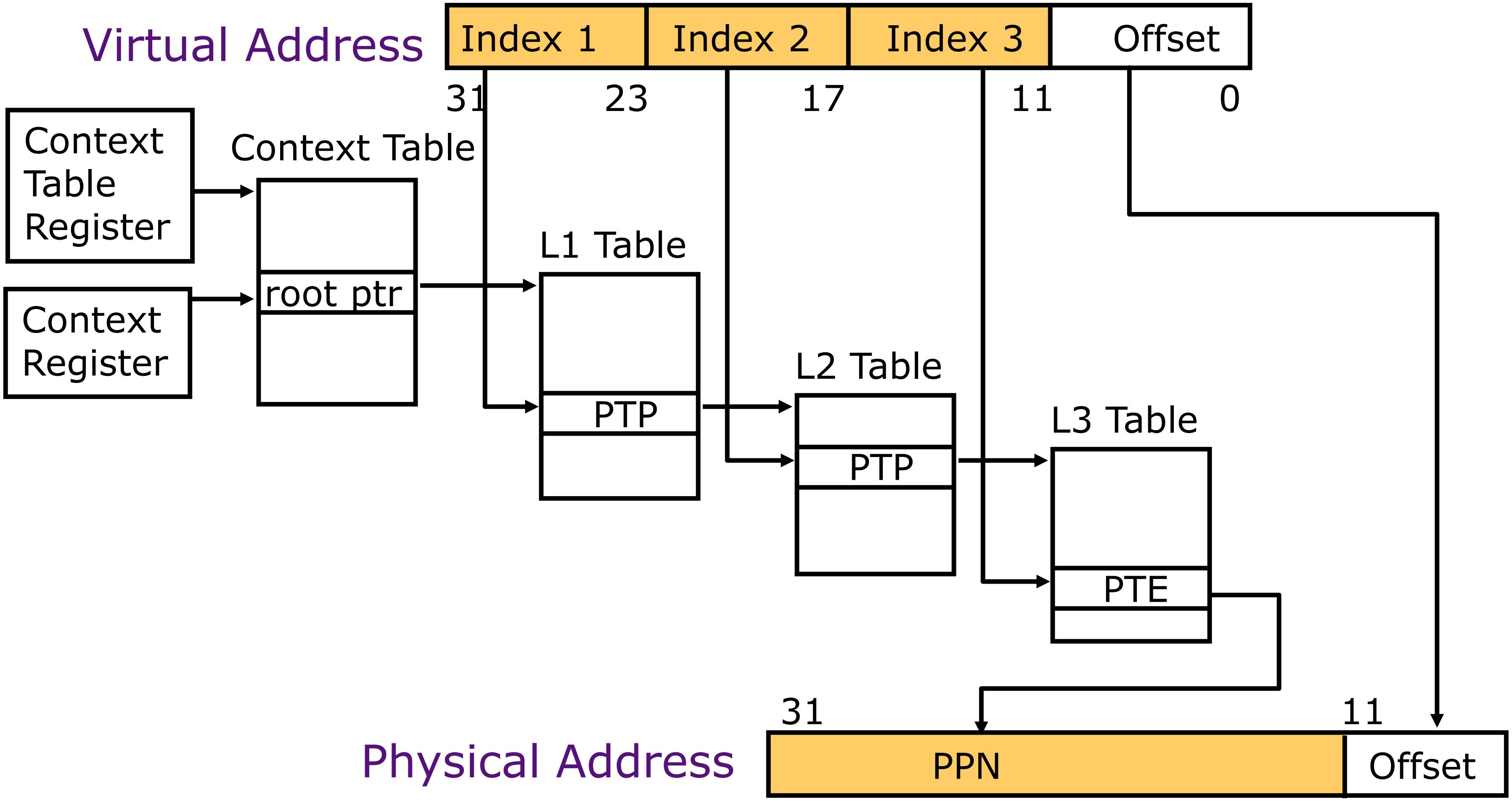
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

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

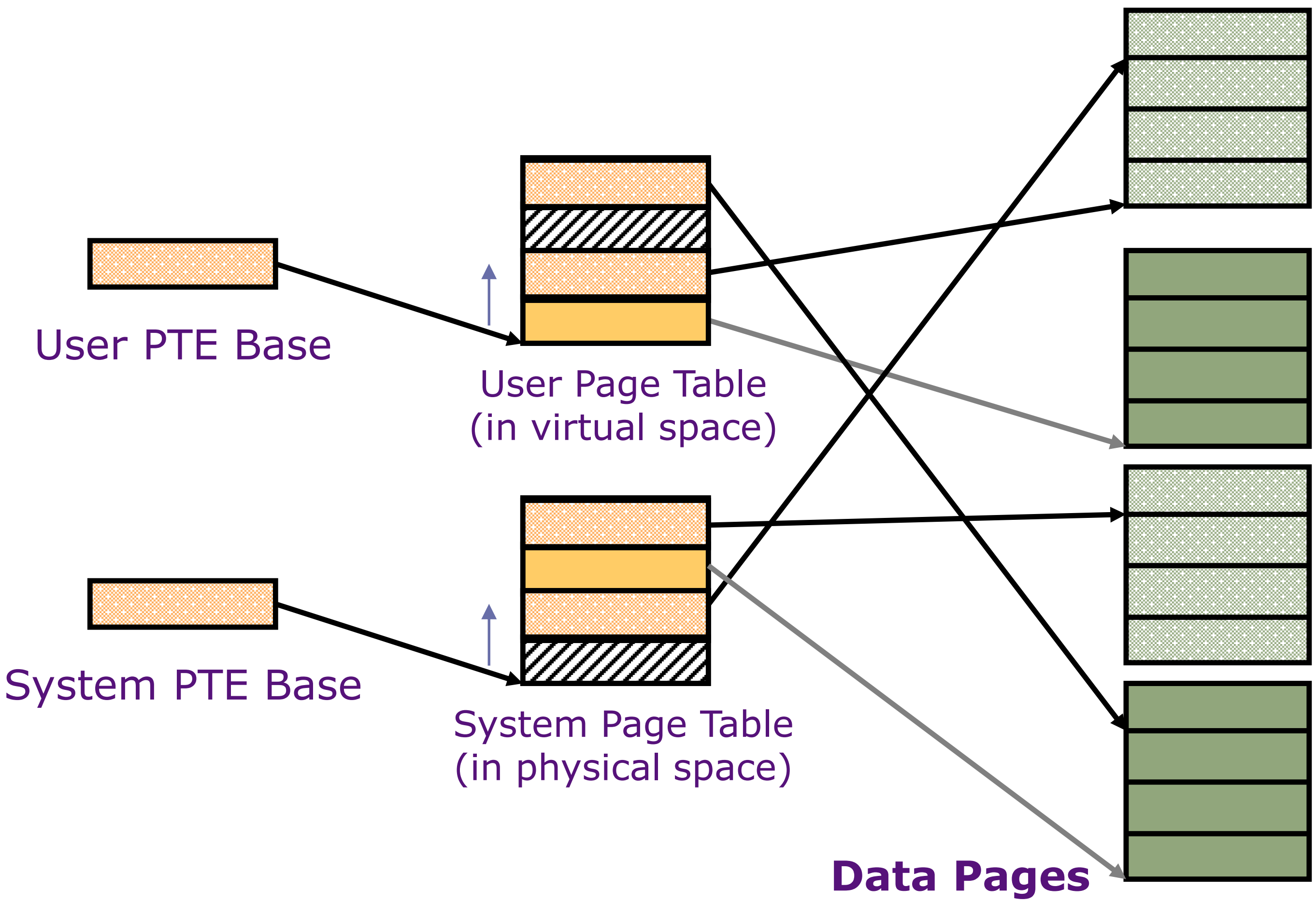
Hierarchical Page Table Walk: SPARC v8



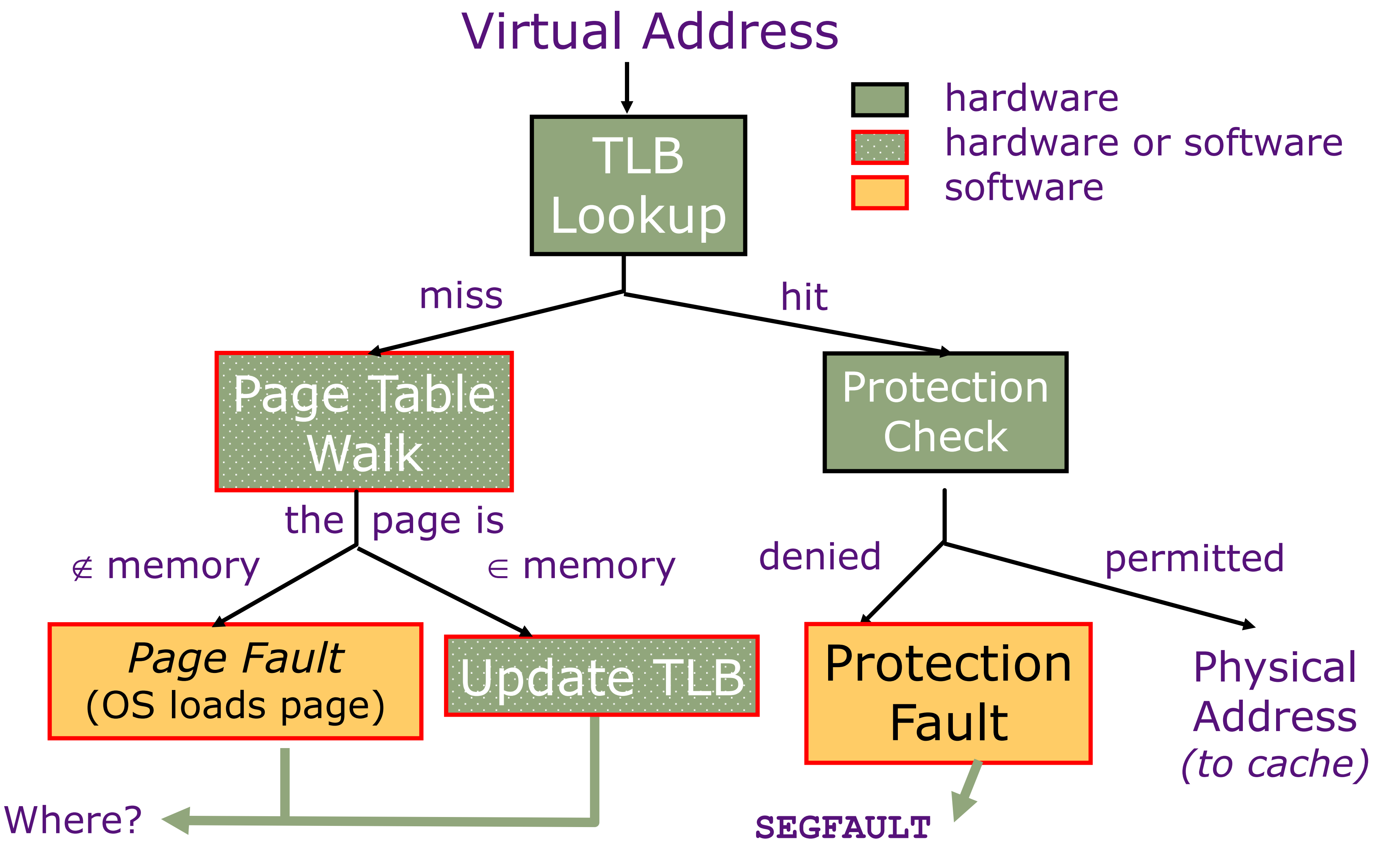
MMU does this table walk in hardware on a TLB miss

Translation for Page Tables

- Can references to page tables cause TLB misses?
- Can this go on forever?



Address Translation: *putting it all together*



Next lecture:

Modern Virtual Memory Systems