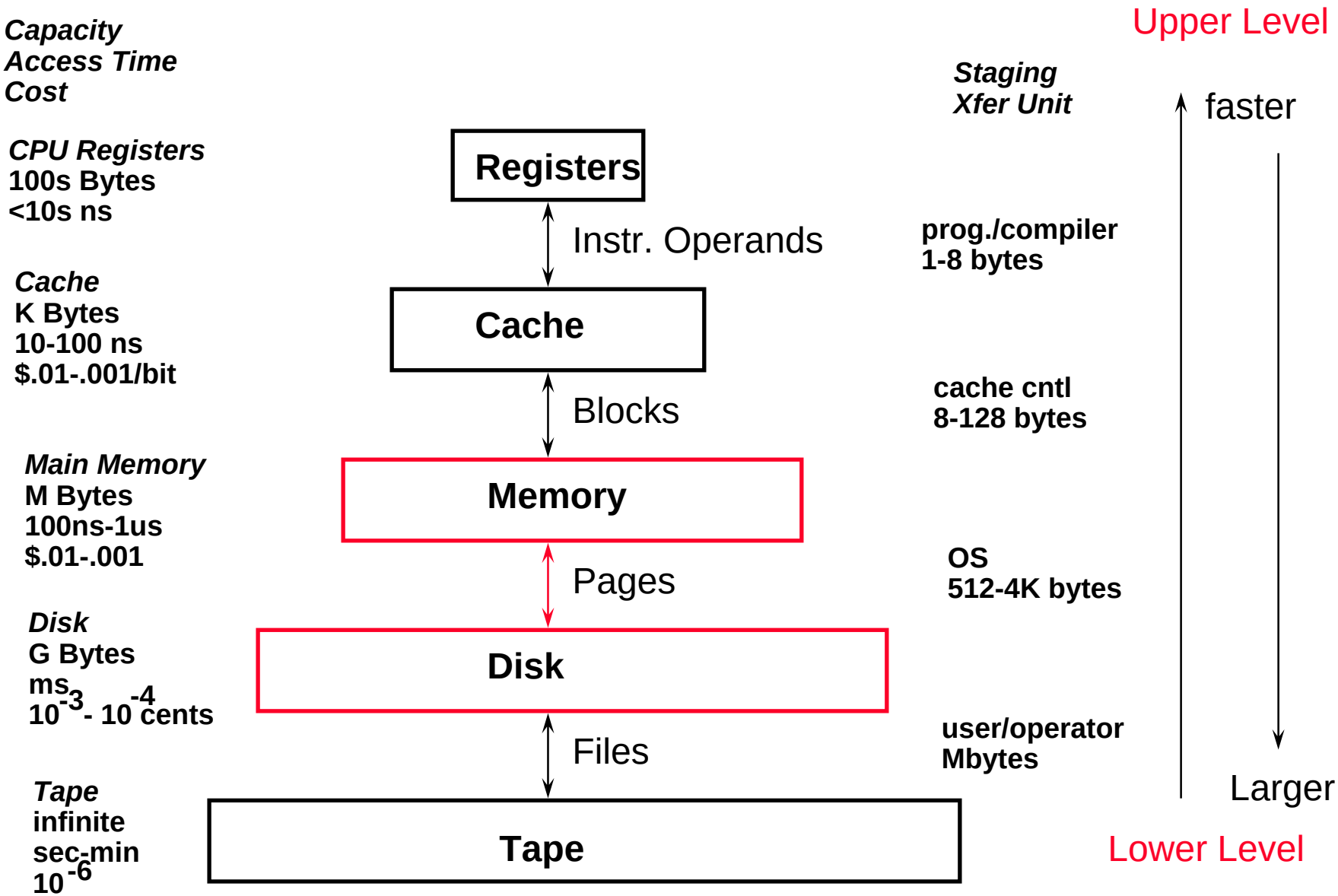
EECS 361
Computer Architecture
Lecture 16: Virtual Memory

Review: The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
  - Example: 90% of time in 10% of the code

Review: Levels of the Memory Hierarchy



Outline of Today's Lecture

- Recap of Memory Hierarchy
- Virtual Memory
- Page Tables and TLB
- Protection

Virtual Memory?

Provides *illusion* of very large memory
- sum of the memory of many jobs greater than physical memory
- address space of each job larger than physical memory

Allows available (fast and expensive) physical memory to be very well utilized

Simplifies memory management

Exploits memory hierarchy to keep average access time low.

Involves at least two storage levels: *main* and *secondary*

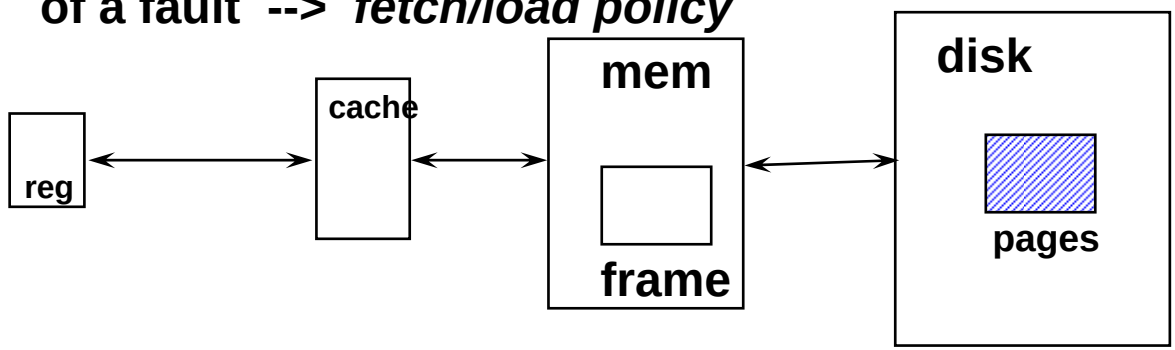
***Virtual Address* -- address used by the programmer**

***Virtual Address Space* -- collection of such addresses**

***Memory Address* -- address of word in physical memory**
also known as "physical address" or "real address"

Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage
- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy*
- which region of M is to hold the new block --> *placement policy*
- missing item fetched from secondary memory only on the occurrence of a fault --> *fetch/load policy*



Paging Organization

virtual and physical address space partitioned into blocks of equal size

page frames

pages

Address Map

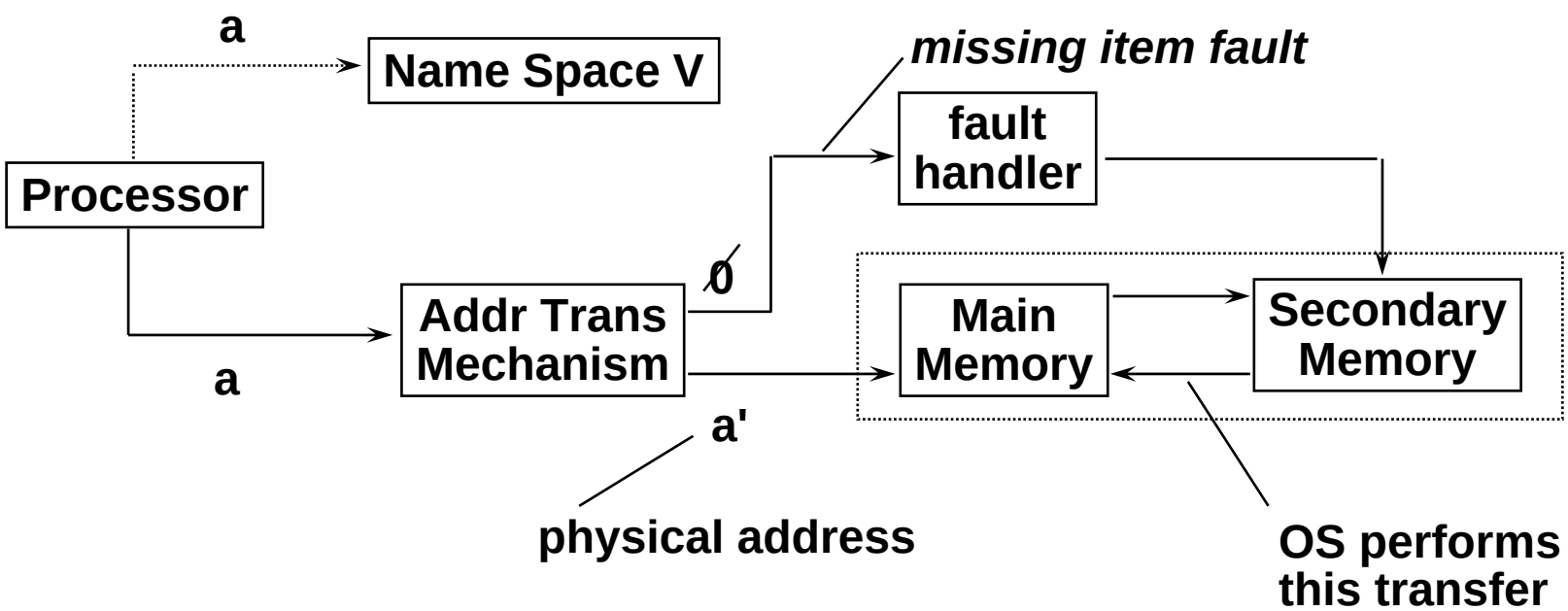
V = {0, 1, . . . , n - 1} virtual address space
M = {0, 1, . . . , m - 1} physical address space

n > m

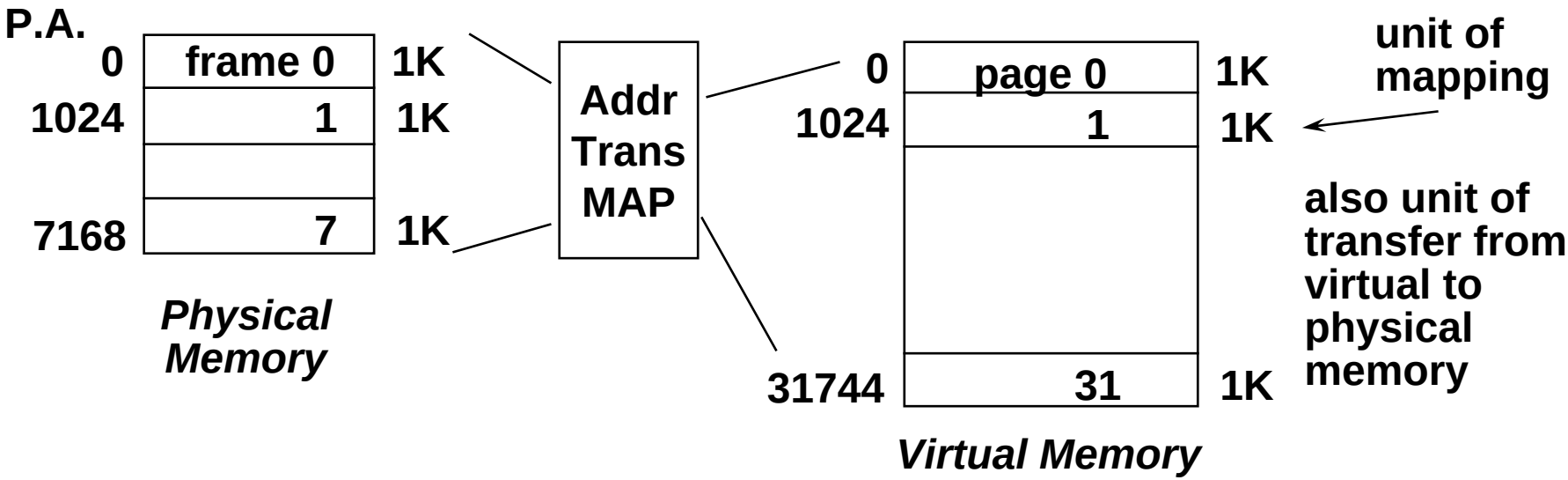
MAP: V --> M U {ø} address mapping function

MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

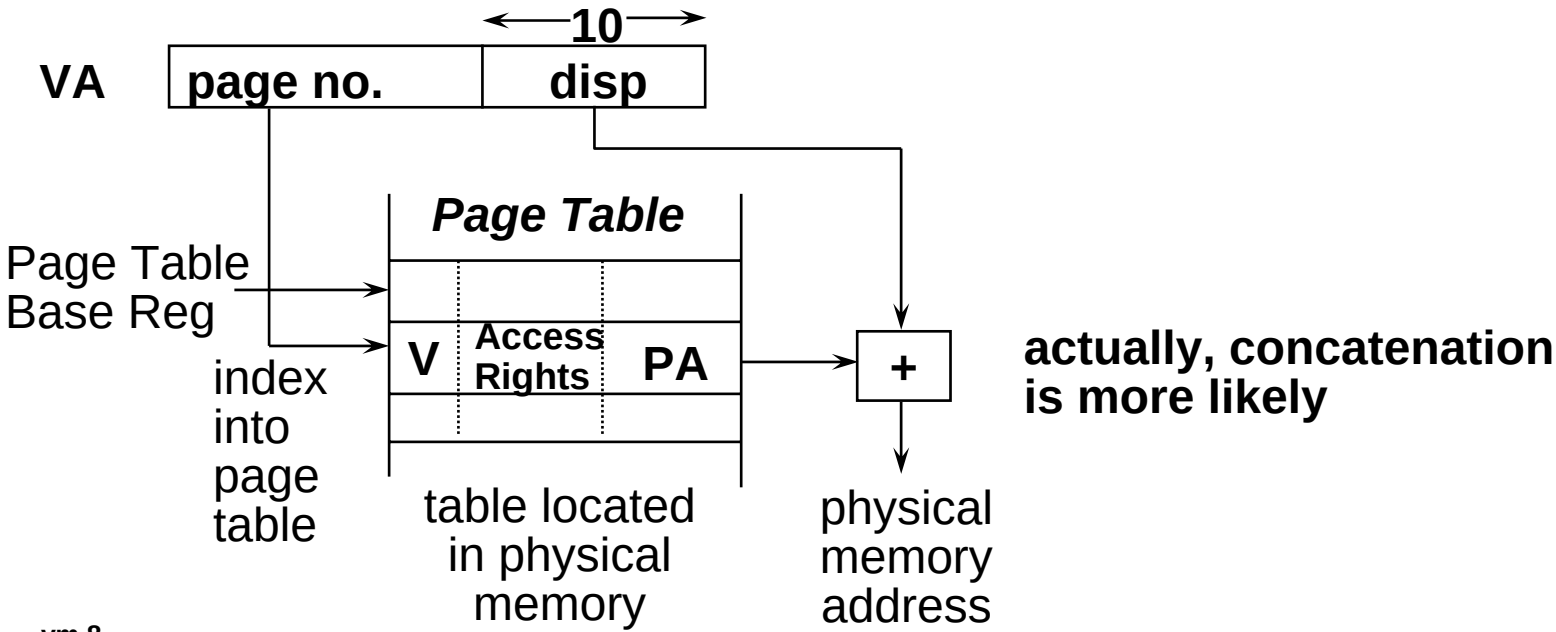
= ø if data at virtual address a is not present in M



Paging Organization



Address Mapping



Address Mapping Algorithm

If V = 1
then page is in main memory at frame address stored in table
else address located page in secondary memory

Access Rights
R = Read-only, R/W = read/write, X = execute only

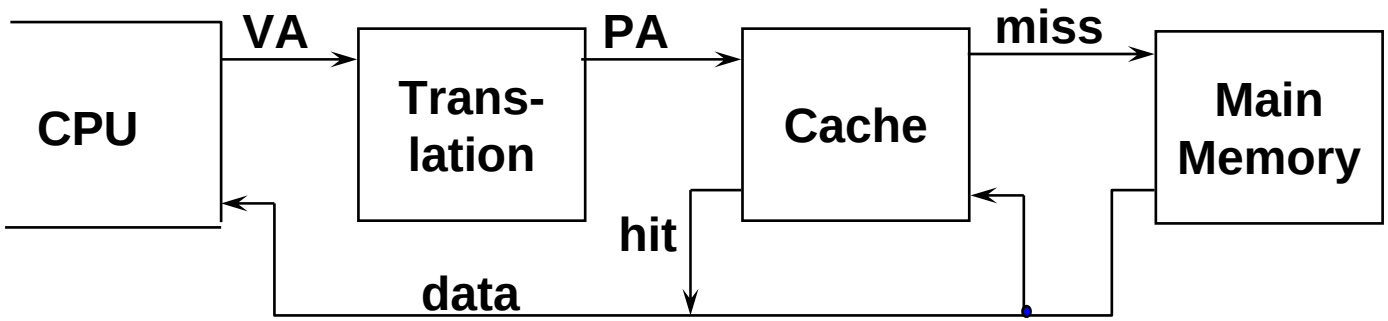
If kind of access not compatible with specified access rights,
then *protection_violation_fault*

If valid bit not set then *page fault*

Protection Fault: **access rights violation; causes trap to hardware, microcode, or software fault handler**

Page Fault: **page not resident in physical memory, also causes a trap; usually accompanied by a *context switch:* current process suspended while page is fetched from secondary storage**

Virtual Address and a Cache

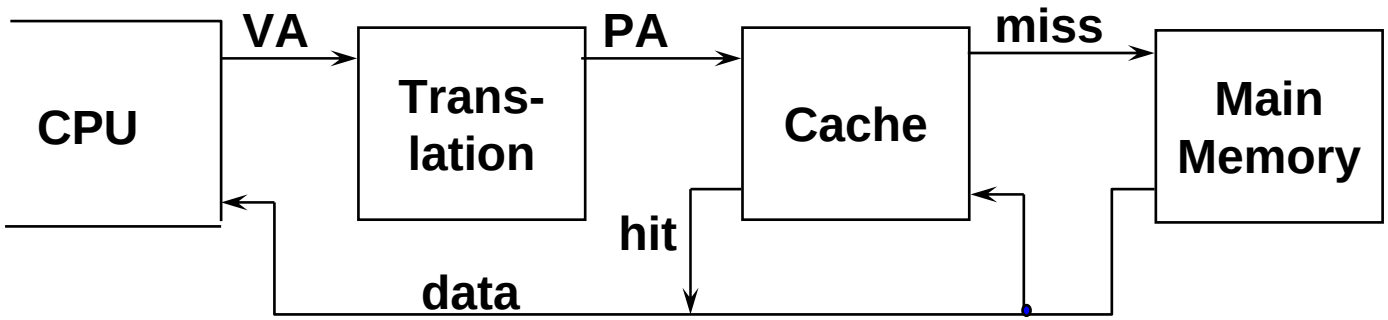


It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

ASIDE: Why access cache with PA at all? VA caches have a problem!

Virtual Address and a Cache



It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

ASIDE: Why access cache with PA at all? VA caches have a problem!

synonym problem:
two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

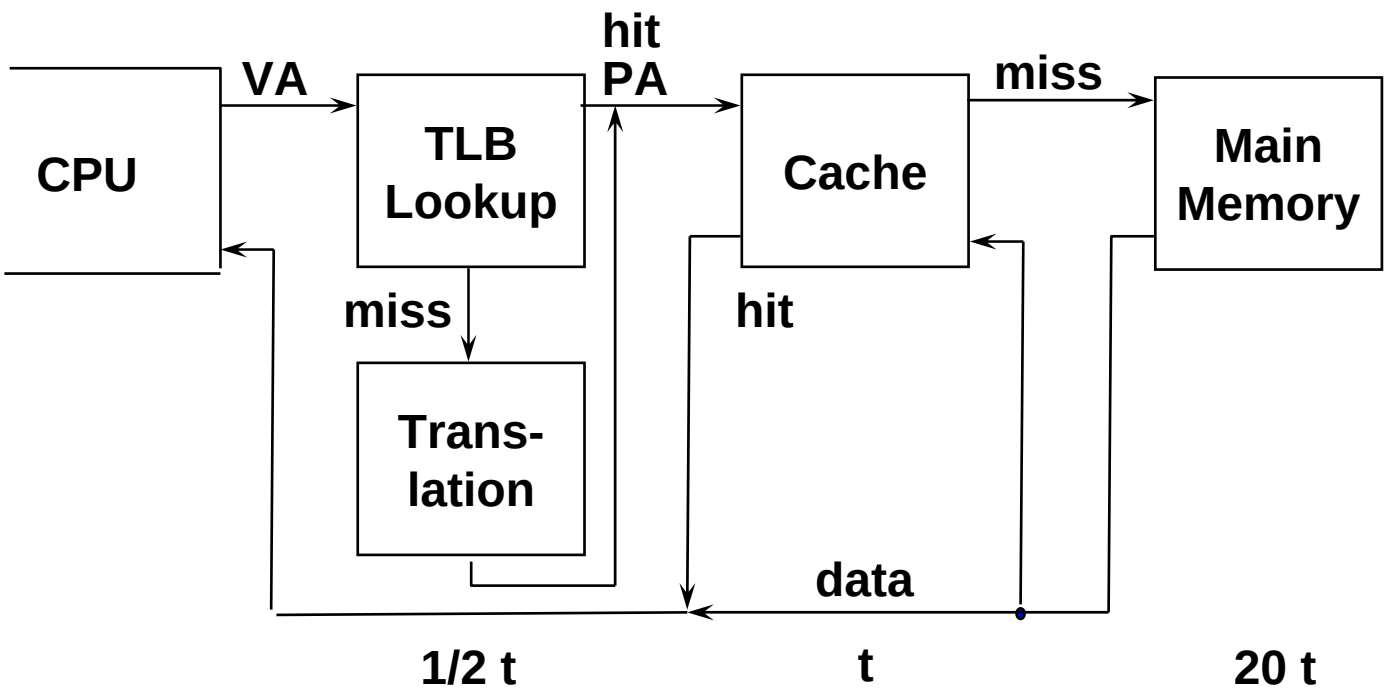
TLB access time comparable to, though shorter than, cache access time (still much less than main memory access time)

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB



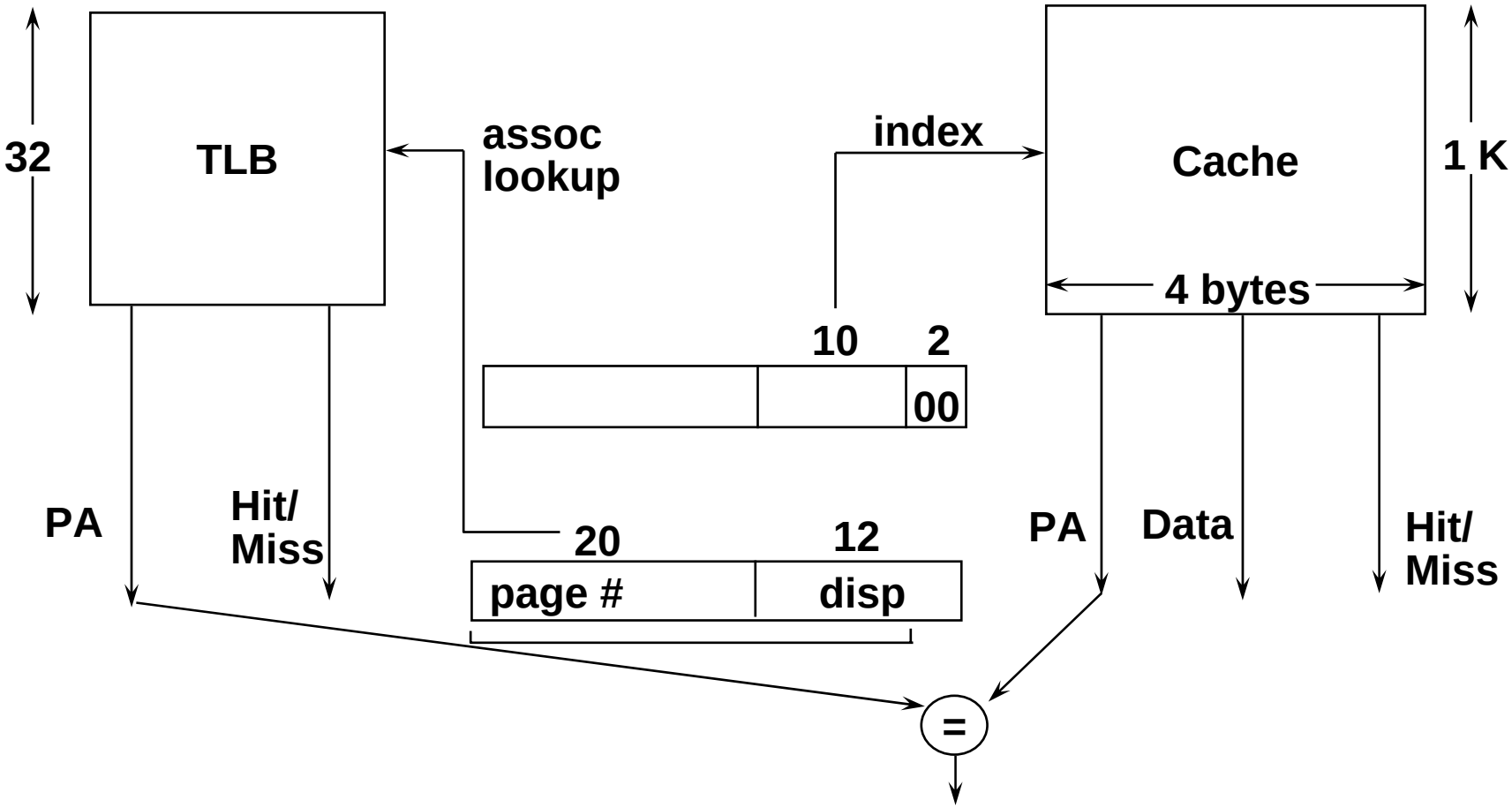
Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

Overlapped Cache & TLB Access



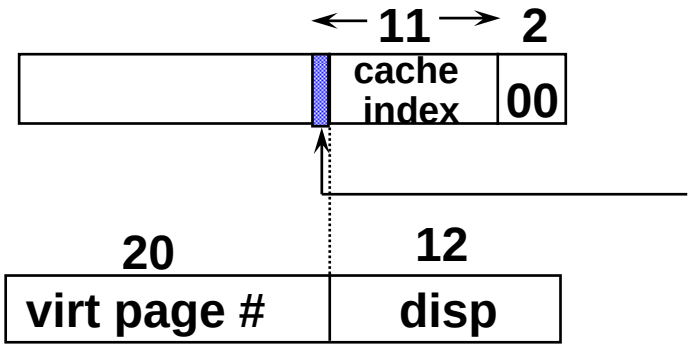
IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

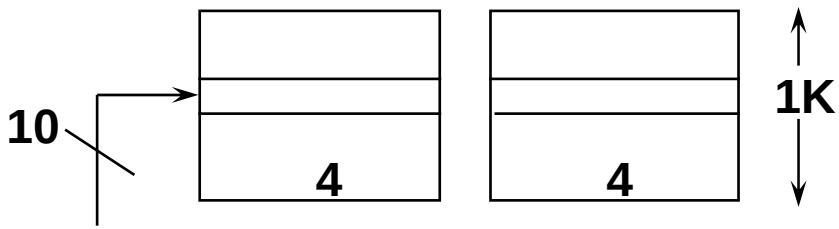


This bit is changed by VA translation, but is needed for cache lookup

Solutions:

go to 8K byte page sizes

go to 2 way set associative cache (would allow you to continue to use a 10 bit index)



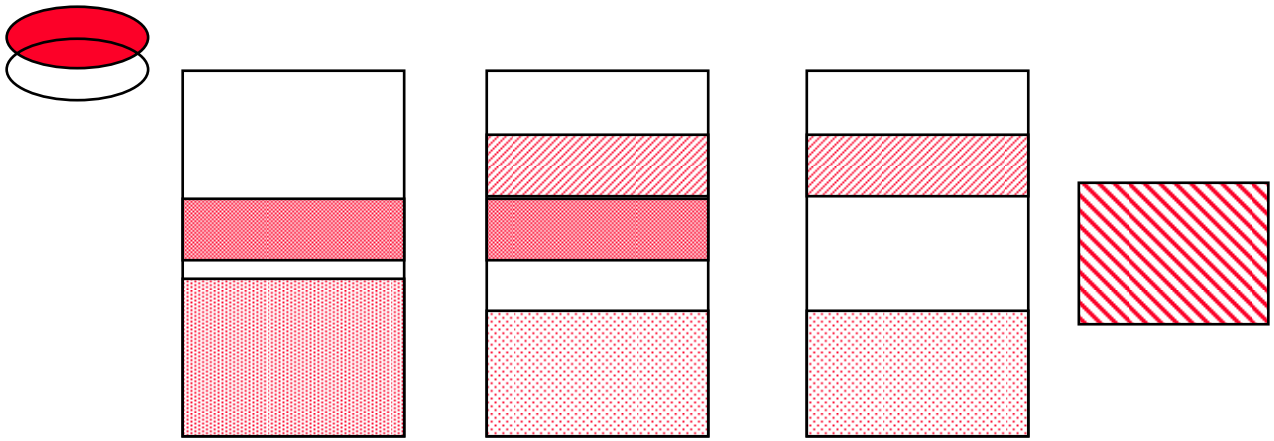
2 way set assoc cache

Fragmentation & Relocation

Fragmentation is when areas of memory space become unavailable for some reason

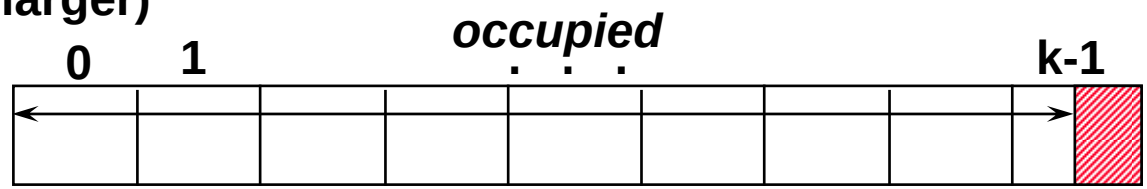
Relocation**:** move program or data to a new region of the address space (possibly fixing all the pointers)

External Fragmentation**: Space left between blocks.**



Internal Fragmentation:

program is not an integral # of pages, part of the last page frame is "wasted" (obviously less of an issue as physical memories get larger)



Optimal Page Size

Choose page that minimizes fragmentation

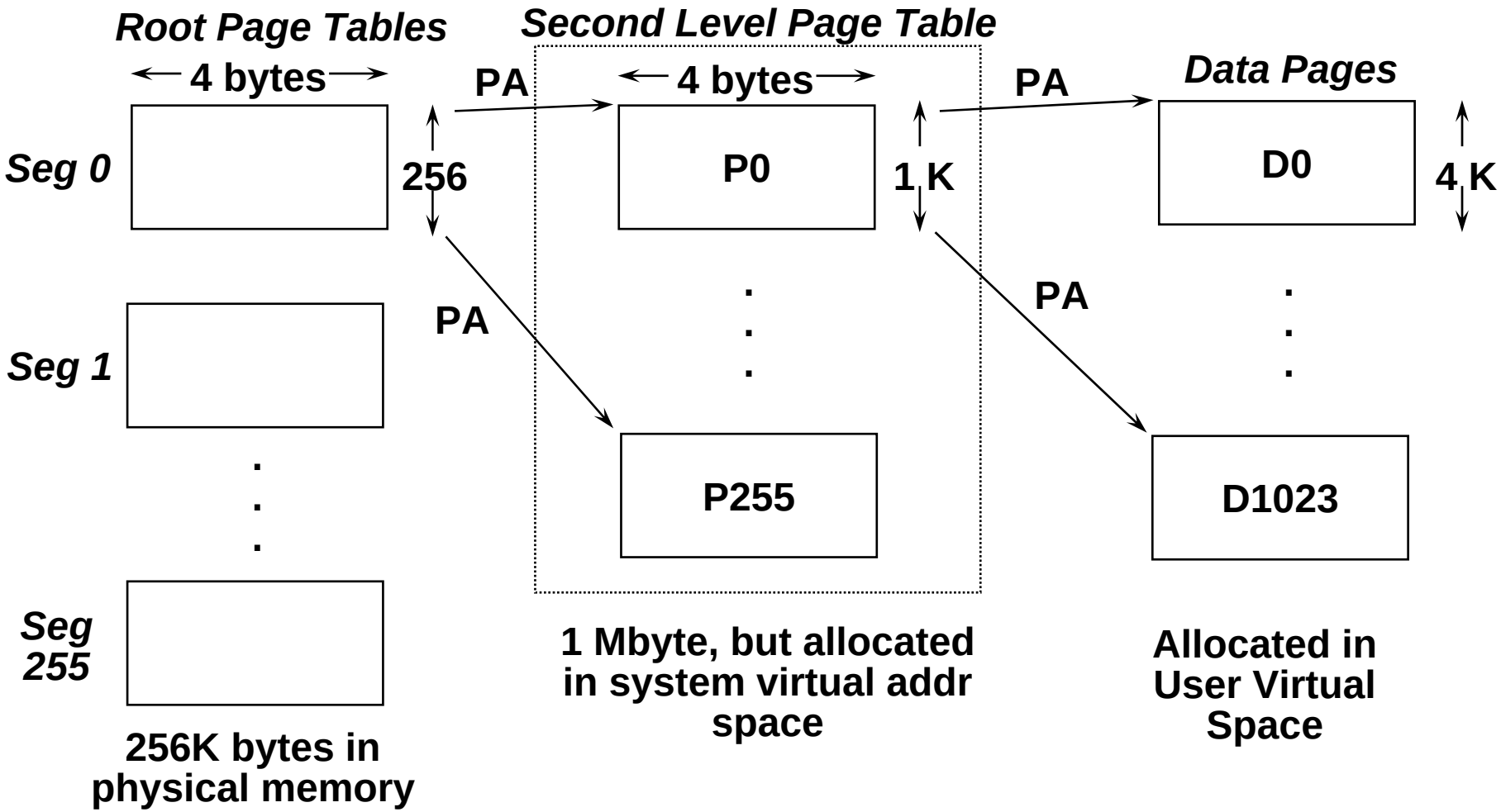
large page size => internal fragmentation more severe
BUT increase in the # of pages / name space => larger page tables

In general, the trend is towards larger page sizes because

- **-- memories get larger as the price of RAM drops**
- **-- the gap between processor speed and disk speed grow wider**
- **-- programmers desire larger virtual address spaces**

Most machines at 4K-64K byte pages today, with page sizes likely to increase

2-level page table



$2^8 \times 2^8 \times 2^{10} \times 2^{12} = 2^{38}$

Page Replacement Algorithms

Just like cache block replacement!

Least Recently Used:

-- selects the least recently used page for replacement

-- requires knowledge about past references, more difficult to implement (thread thru page table entries from most recently referenced to least recently referenced; when a page is referenced it is placed at the head of the list; the end of the list is the page to replace)

-- good performance, recognizes principle of locality

Page Replacement (Continued)

Not Recently Used:

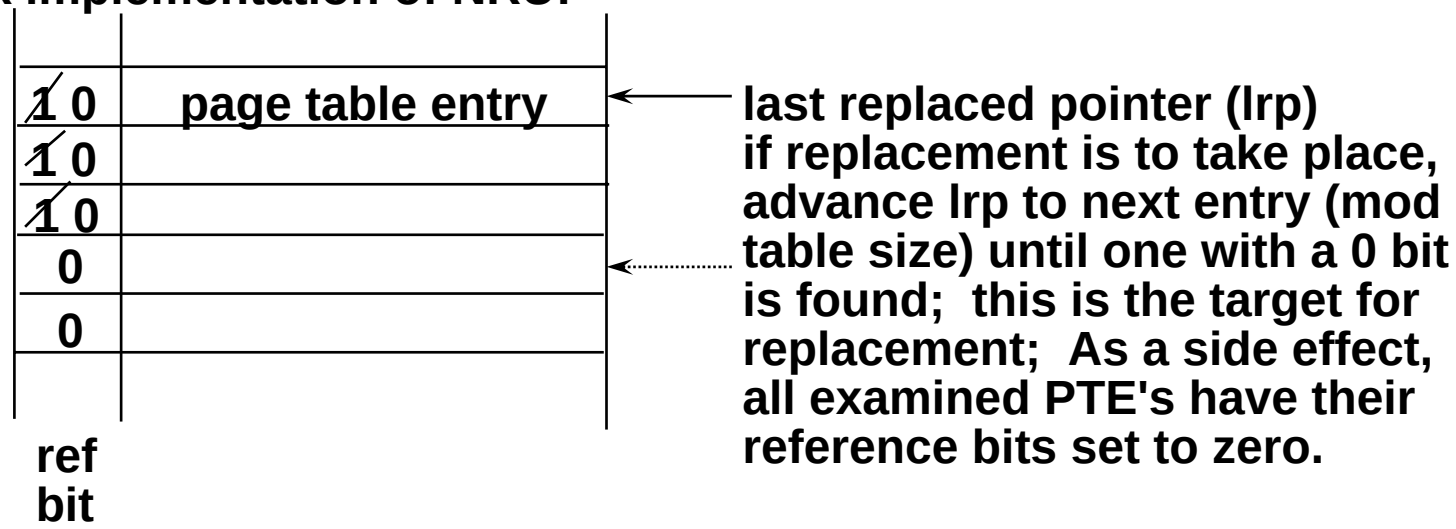
Associated with each page is a reference flag such that

ref flag = 1 if the page has been referenced in recent past

= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

-- clock implementation of NRU:



An optimization is to search for the a page that is both not recently referenced AND not dirty.

Demand Paging and Prefetching Pages

Fetch Policy

when is the page brought into memory?

if pages are loaded solely in response to page faults, then the policy is *demand paging*

An alternative is *prefetching:*

anticipate future references and load such pages before their actual use

+ reduces page transfer overhead

- removes pages already in page frames, which could adversely affect the page fault rate

- predicting future references usually difficult

Most systems implement demand paging without prepaging

(One way to obtain effect of prefetching behavior is increasing the page size

Summary

- Virtual memory a mechanism to provide much larger memory than physically available memory in the system
- Placement, replacement and other policies can have significant impact on performance
- Interaction of Virtual memory with physical memory hierarchy is complex and addresses translation mechanisms must be designed carefully for good performance.