EEL-4713
Computer Architecture
Virtual Memory

Outline

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

Memory addressing - physical

- So far we considered addresses of loads/stores go directly to caches/memory
  - As in your project

- This makes life complicated if a computer is multi-processed/multi-user
  - How do you assign addresses within a program so that you *know* other users/programs will not conflict with them?

Program A:	Program B:
store 0x100,1	store 0x100,5
	load R1,0x100

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 efficiently utilized

Simplifies memory management and programming

Exploits memory hierarchy to keep average access time low.

Involves at least two storage levels: *main* and *secondary*
Main (DRAM): nanoseconds, M/GBytes
Secondary (HD): miliseconds, G/TBytes

***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"

Memory addressing - virtual

Program A:
store 0x100,1

Program B:
store 0x100,5
load R1,0x100

Translation A:
0x100 -> 0x40000100

Translation B:
0x100 -> 0x50000100

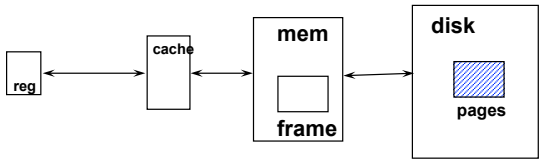
Use software and hardware to guarantee no conflicts
Operating system: keep software translation tables
Hardware: cache recent translations

Basic Issues in VM System Design

size of information blocks (pages) that are transferred from secondary (disk) to main storage (Mem)

Page brought into Mem, if Mem is full some page of Mem must be released to make room for the new page --> *replacement policy*

missing page fetched from secondary memory only on the occurrence of a page fault --> *fetch/load policy*



Paging Organization

virtual and physical address space partitioned into blocks of equal size

page frames

pages

Address Map

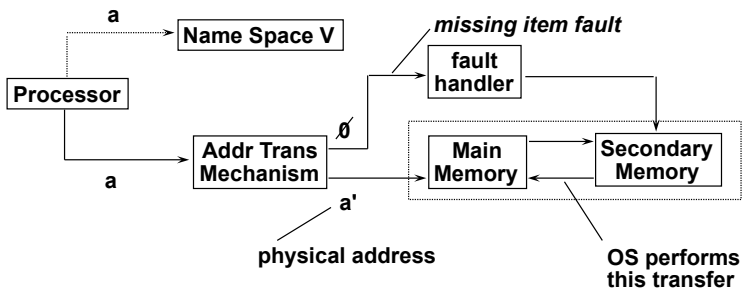
V = {0, 1, . . . , n - 1} virtual address space
M = {0, 1, . . . , m - 1} physical address space

n can be > m

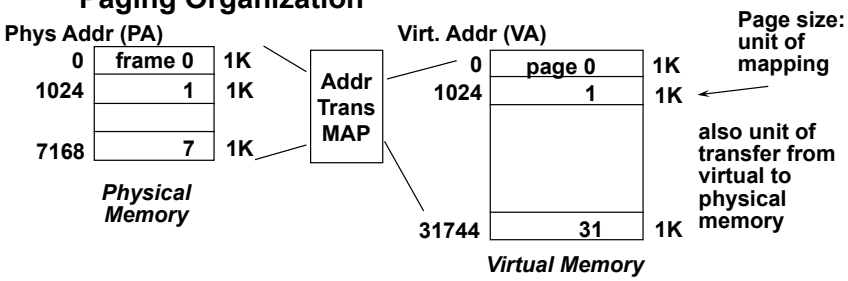
MAP: V --> M U {ø} address mapping function

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

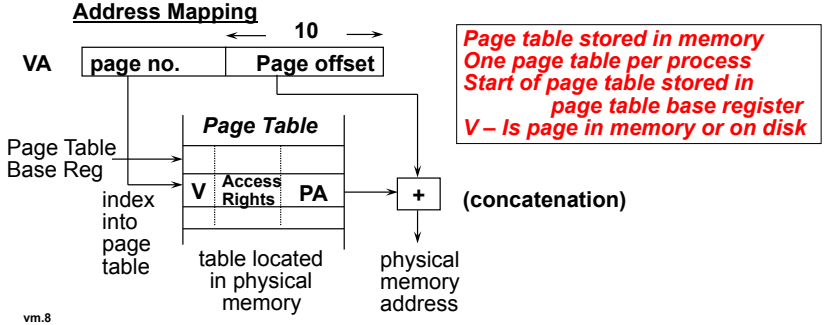
= ø if data at virtual address a is not present in M need to allocate address in M



Paging Organization



Address Mapping



Page table stored in memory
One page table per process
Start of page table stored in page table base register
V – Is page in memory or on disk

Address Mapping Algorithm

If V = 1 (where is page currently stored?)
then page is in main memory at frame address stored in table
else page is located in secondary memory (location determined at process creation)

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*

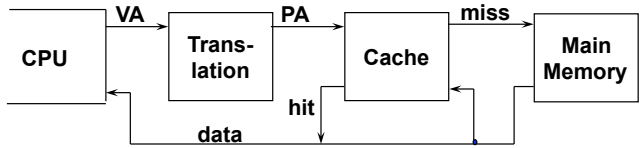
Terms:
Protection Fault: access rights violation; hardware raises exception, microcode, or software fault handler

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; page faults usually handled in software by OS because page fault to secondary memory takes million+ cycles

*Hardware/software interface

° What checks does the processor perform during a load/store memory access?
- Effective address computed in pipeline is virtual
- Before accessing memory, must perform virtual-physical mapping
  - At hardware speed, critical to performance
- If there is a valid mapping, load/store proceeds as usual; address sent to cache, DRAM is the mapped address (physical addressed)
- If there is no valid mapping, or if there is a protection violation, processor does not know how to handle it
  - Throw an exception
    - Save the PC of the instruction that caused the exception so that it can be retried later
    - Jump into an operating system exception handling routine
  - O/S handles exception using its specific policies (Linux, Windows will behave differently)
  - Once it finishes handling, issue "return from interrupt" instruction to recover PC and try instruction again

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:
1. two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address! (data sharing between different processes)
2. two same virtual addresses (from different processes) map to different physical addresses

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*

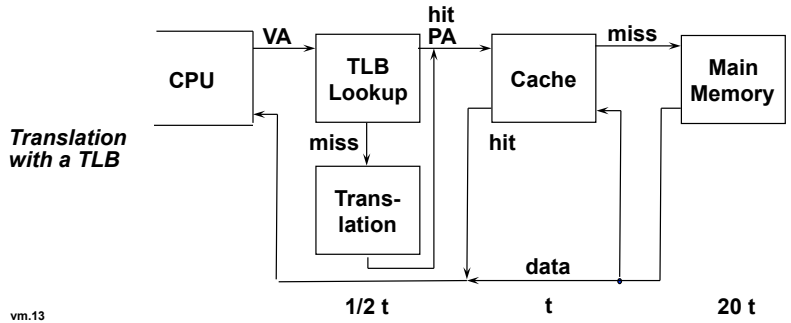
Virtual Address	Physical Address	Dirty	Ref	Valid	Access

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 hundreds of entries. This permits large/full associativity.



Translation with a TLB

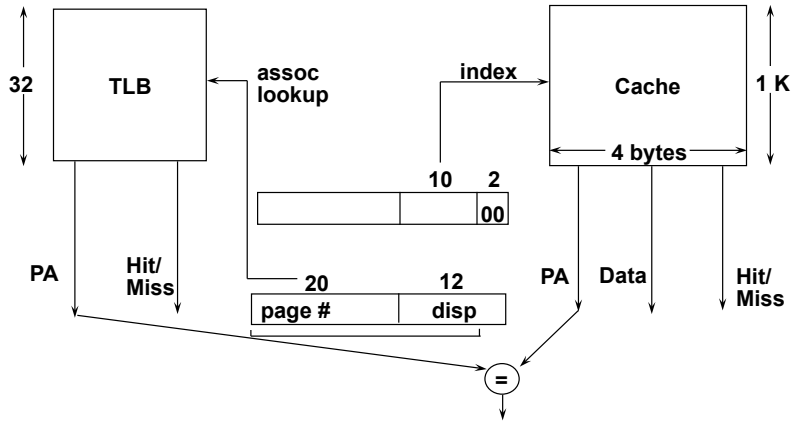
Reducing Translation Time

Machines with TLBs can 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 up in the TLB while low order bits are used as index into cache
* Virtually indexed, physically tagged.

Overlapped Cache & TLB Access



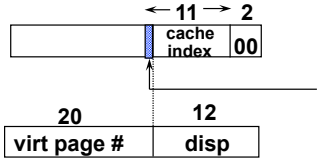
IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag != PA)] and TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

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

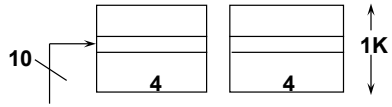
This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache (e.g., cache size and page size need to match)

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

Hardware versus software TLB management

- The TLB misses can be handled either by software or hardware
  - Software: processor has instructions to modify TLB in the architecture; O/S handles replacement
  - E.g. MIPS
  - Hardware: processor handles replacement without need for instructions to store TLB entries
  - E.g. x86
- Instructions that cause TLB "flushes" are needed in hardware case too

Optimal Page Size

Choose page that minimizes fragmentation

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

In general, the trend is towards larger page sizes because

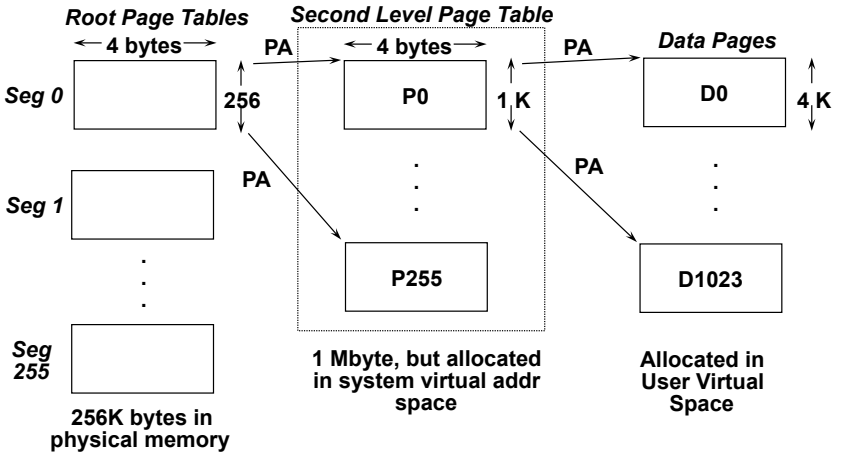
-- **memories get larger as the price of RAM drops**

-- **the gap between processor speed and disk speed grows wider**
Larger pages can exploit more spatial locality in transfers between disk and memory

-- **programmers desire larger virtual address spaces**

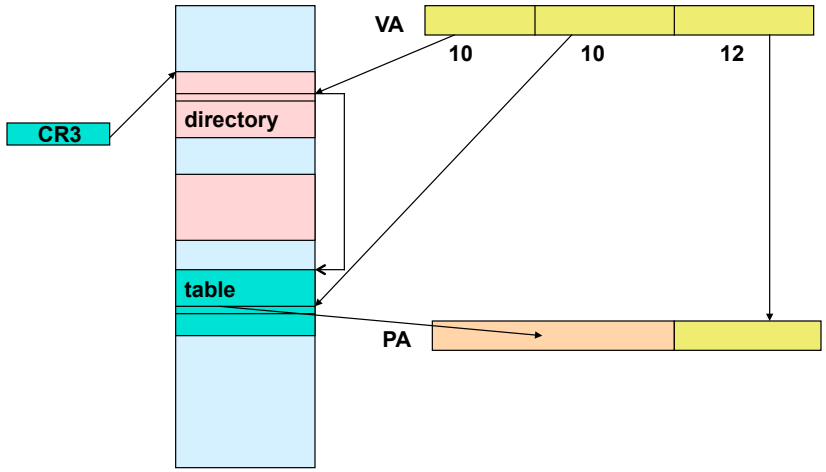
Most machines at 4K-64K byte pages today, with page sizes likely to increase

2-level page table



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

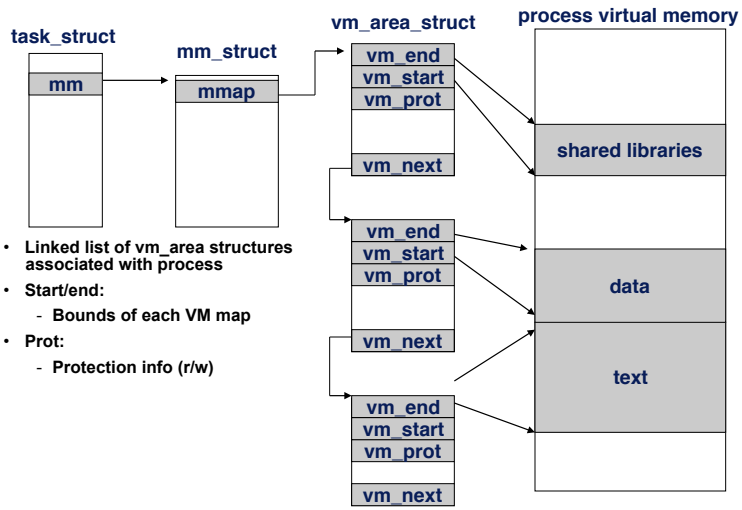
Example: two-level address translation in x86



Example: Linux VM

- Demand-paging
  - Pages are brought into physical memory when referenced
- Kernel keeps track of each process' virtual address space using a mm_struct data structure
  - Which contains pointers to list of "area" structures (vm_area_struct)

Linux VM areas

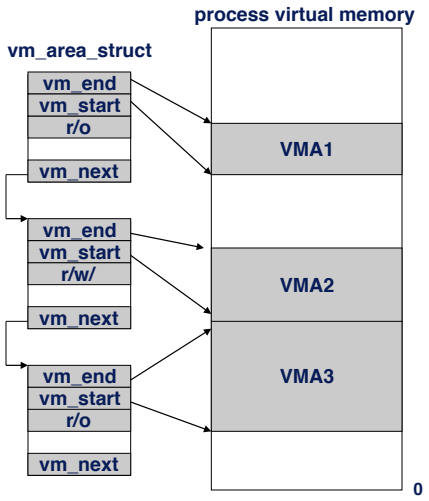


- Linked list of vm_area structures associated with process
- Start/end:
  - Bounds of each VM map
- Prot:
  - Protection info (r/w)

VM "areas"

- Linked vm_area_struct structures
- A VM area: a part of the process virtual memory space that has a special rule for the page-fault handlers (i.e. a shared library, the executable area etc).
- These are specified in a vm_area_struct
  - Start and end VM address of area
  - Protection information, flags

Linux fault handling



- Exception triggers O/S handling if address out of bounds or protection violated
- Traverse vm_area list, check for bounds
  - If not mapped, it is a segmentation violation – signal to process
- If mapped, check protection of vm_area_struct
  - E.g. r/o, r/w
  - Signal protection violation to process if access not allowed
  - Otherwise, handle fault and bring page to memory

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

-- good performance, recognizes principle of locality

-- hard to keep track – update a structure on each memory reference?

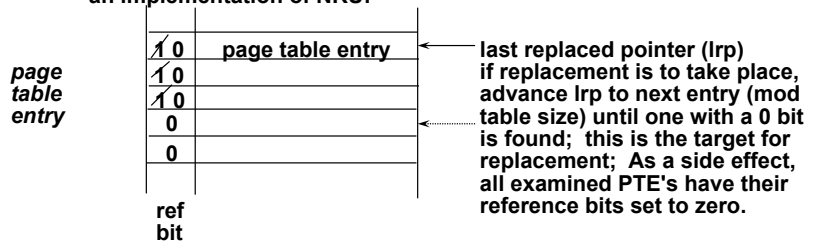
Page Replacement (Continued)

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

-- an 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

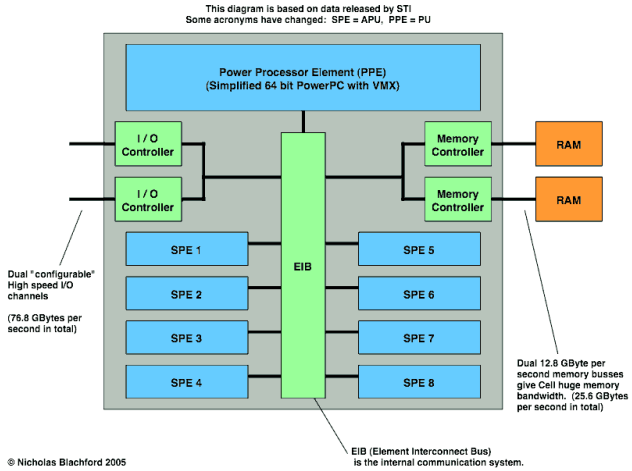
Most systems implement demand paging without prefetching

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

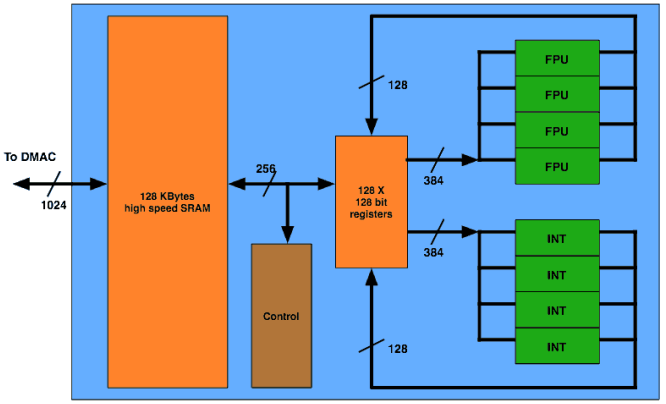
Discussion – caches or no caches?

- **Caches help performance when there is locality but can be overhead if locality is not high**
  - **Each hit/miss decision at each cache level requires a lookup**
  - **Some applications can run better with fewer cache levels**
- **Example: "Cell" processor**
  - **No cache on attached processing units**
  - **There is a small memory array next to each unit, but it is handled by software (not a cache controller)**

Cell Processor Architecture



Cell APU Architecture



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.