Virtual Memory

Today

- Handling bigger address spaces
- Speeding translation

Considerations with page tables

Two key issues with page tables

- Mapping must be fast
  - Done on every memory reference, at least 1 per instruction
- With large address spaces, page tables will be large
  - 32 bit addresses & 4KB page → 12 bit offset, 20 bit page # ~ 1million PTE (2^{20})
  - 64 bit addresses & 4KB page → … 2^{52} pages ~ 4.5×10^{15}!!!

- Simplest solutions
  - Page table in registers
    - Fast during execution, $$$ & slow to context switch
  - Page table in memory & Page Table Base Register (PTBR)
    - Fast to context switch & cheap, but slow during execution

Page table and page sizes

- Bigger pages
  - Smaller page tables
  - But more internal fragmentation
- Smaller pages
  - Less internal fragmentation
  - Less unused program in memory,
  - But … more I/O time, getting page from disk … most of the time goes into seek and rotational delay!
  - Larger page tables

Page table and page sizes

- Average process size *s* bytes
- Page size *p* bytes
- Page entry size *e* bytes
- Approximate number of pages needed per process *s/p* occupying *s/p*e* bytes of page table space
- *overhead* = *se / p* + *p/2*

Internal fragmentation

- Taking first derivative respect to p and equating it to zero

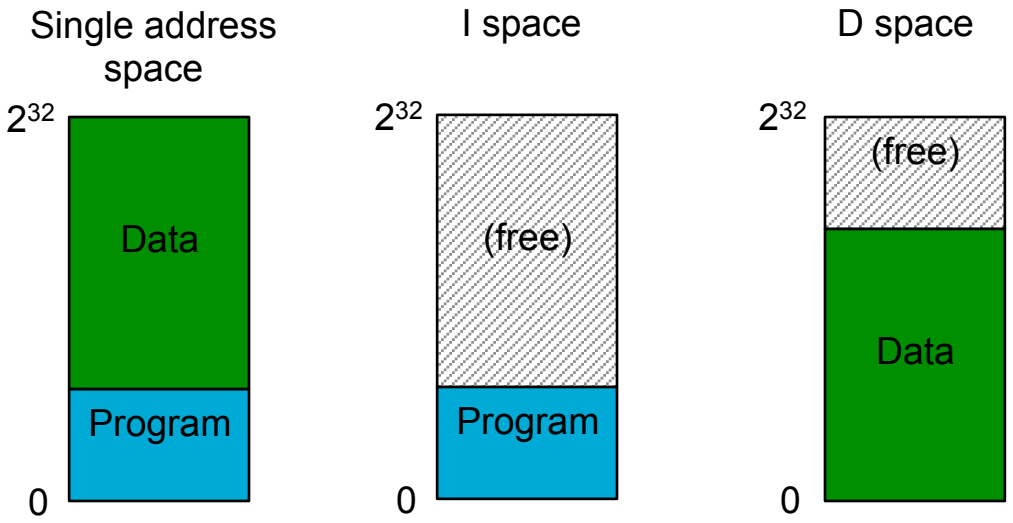
$$-s\,e / p^2 + 1/2 = 0$$

$$p = \sqrt{2se}$$

- *s* = 1MB *e* = 8 bytes ➔ Optimal p = 4KB

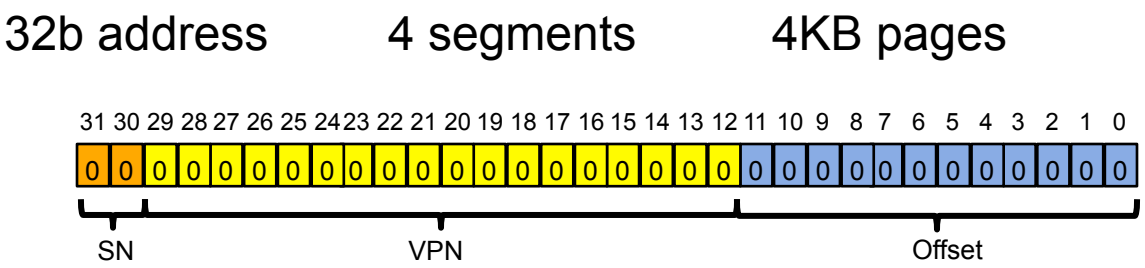
Separate instruction & data spaces

- One address space – size limit
- Pioneered by PDP-11: 2 address spaces, Instruction and Data spaces
  - Double the space
  - Each with its own page table & paging algorithm



A hybrid approach – pages and segments

- As MULTICS
  - Instead of a single page table, one per segment
  - The base register of the segment points to the base of the page table

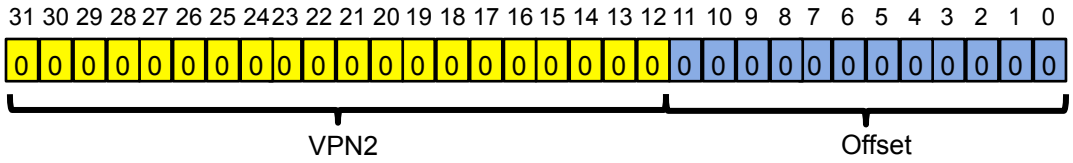


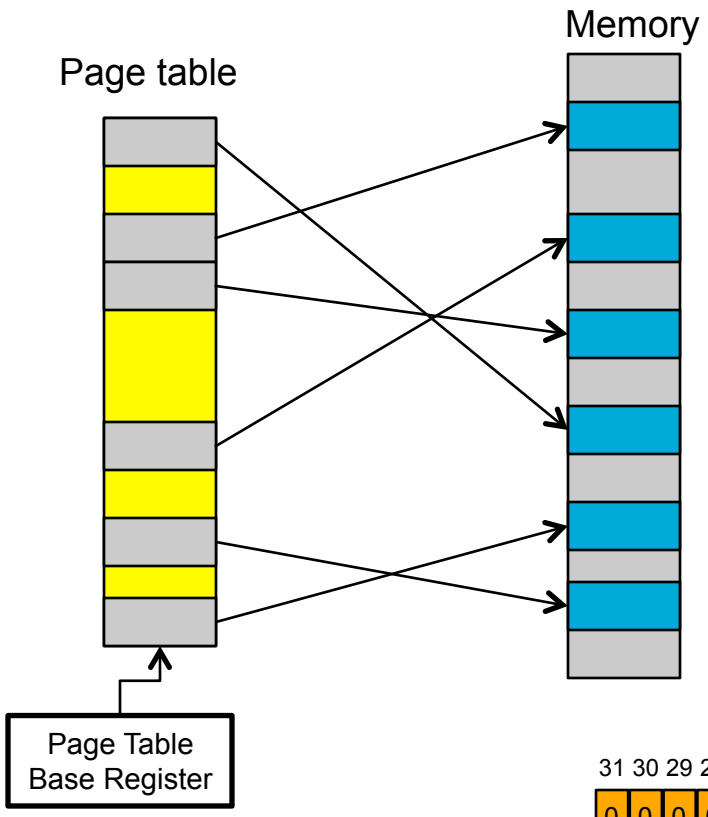
- But
  - Segmentation is not as flexible as we may want
  - Fragmentation is still an issue
  - Page tables can be of any size, so here we go again

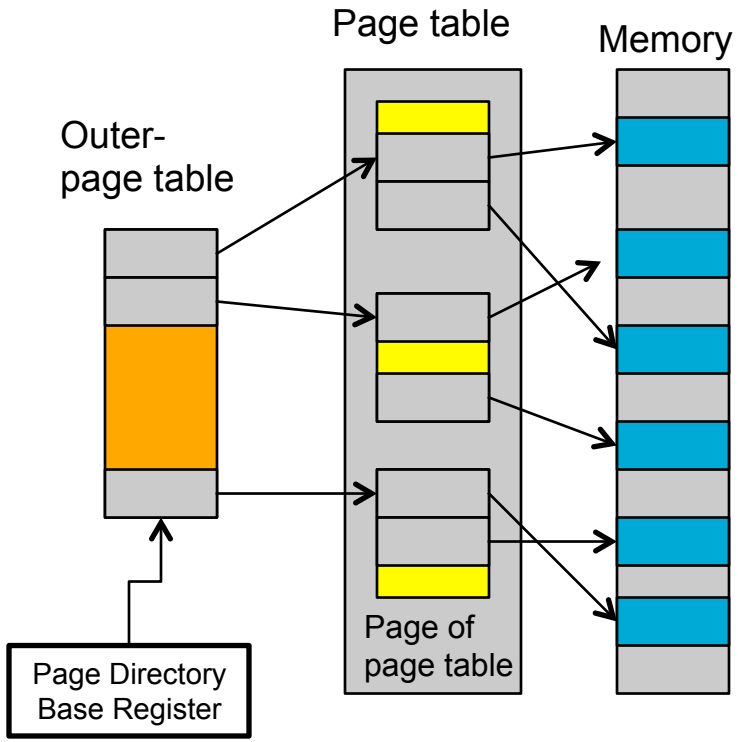
Hierarchical or multilevel page table

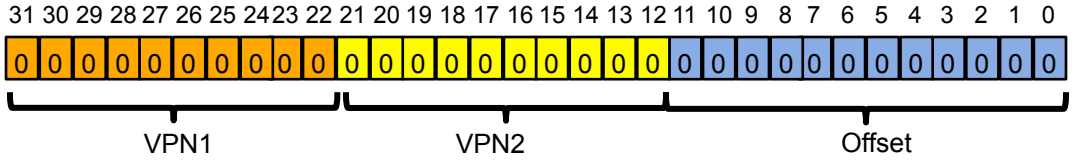
- Another approach – page the page table!
  - Same argument – you don't need the full page table
- Example
  - Virtual address (32-bit machine, 4KB page):
    Page # (20 bits) + Offset (12 bits)
  - Since page table is paged, page number is divided:
    Page number (10 bits) + Page offset in 2nd level (10 bits)
- Pros and cons
  - Allocate page table space as needed
  - If carefully done, each portion of the page table fits neatly within a page
  - More effort for translation
  - And a bit more complex

Hierarchical page table



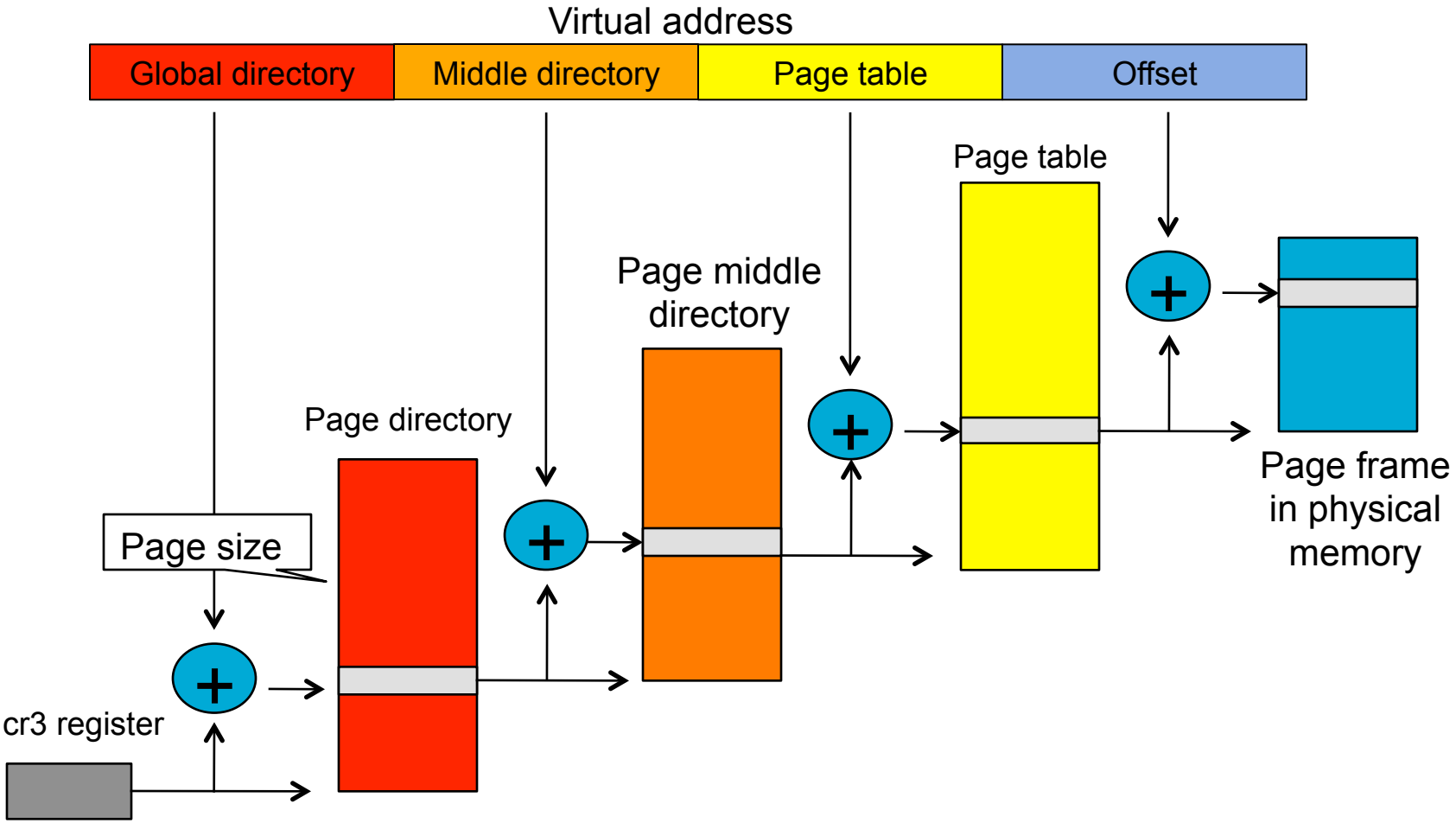






Three-level page table in Linux

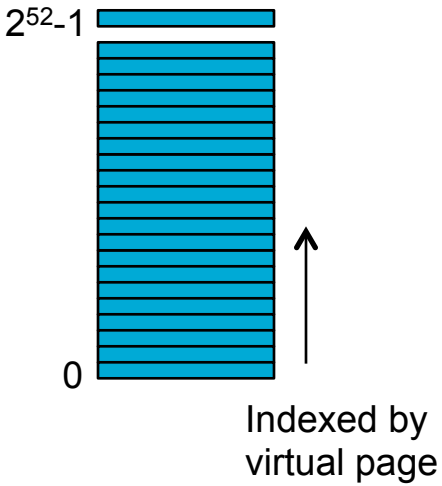
- Designed to accommodate the 64-bit Alpha
  - To adjust for a 32-bit proc. – middle directory of size 1



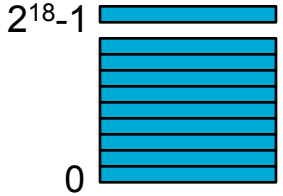
Inverted page tables

- Another way to save space – inverted page tables
  - Page tables are index by virtual page #, hence their size
  - Inverted page tables – one entry per page frame
    - But to get the page you are still given a VPN
      - Straightforward with a page table, but
    - Linear with inverted page tables – too slow mapping!

Traditional page table with an entry per each 2^{52} pages

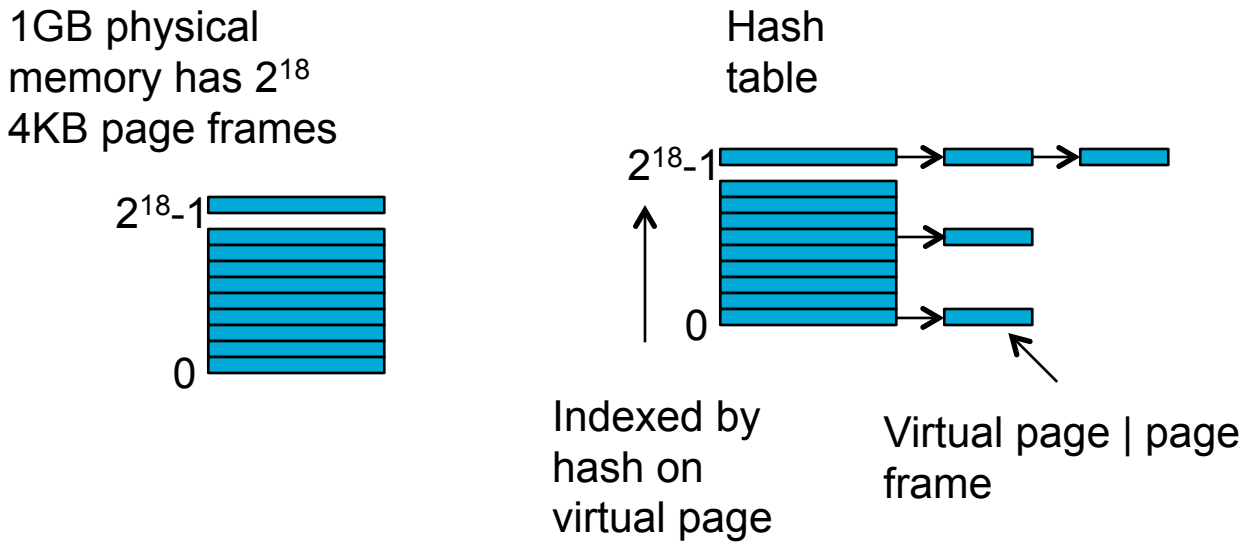


1GB physical memory has 2^{18} 4KB page frames



Inverted and hashed page tables

- Slow inverted page tables … hash tables may help
  - Different virtual page number might map to identical hash values – use a collision-chain mechanism
  - Problem – poor cache locality



  - Also Translation Lookaside Buffers (TLB) …

And now a short break …

Speeding things up a bit

- Simple page table 2x cost of memory lookups
  - First into page table, a second to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into page table and then fetch the data
- And two is not enough …

- *How can we make this more efficient?*

Speeding things up a bit

- Ideally, make fetching from a virtual address almost as efficient as from a physical address
- Observation – locality of references (lot of references to few pages)
- Solution – hardware cache inside the CPU
  - Translation Lookaside Buffer
  - Cache the virtual-to-physical translations in the hardware
    - A better name would be address-translation cache
  - Traditionally managed by the MMU

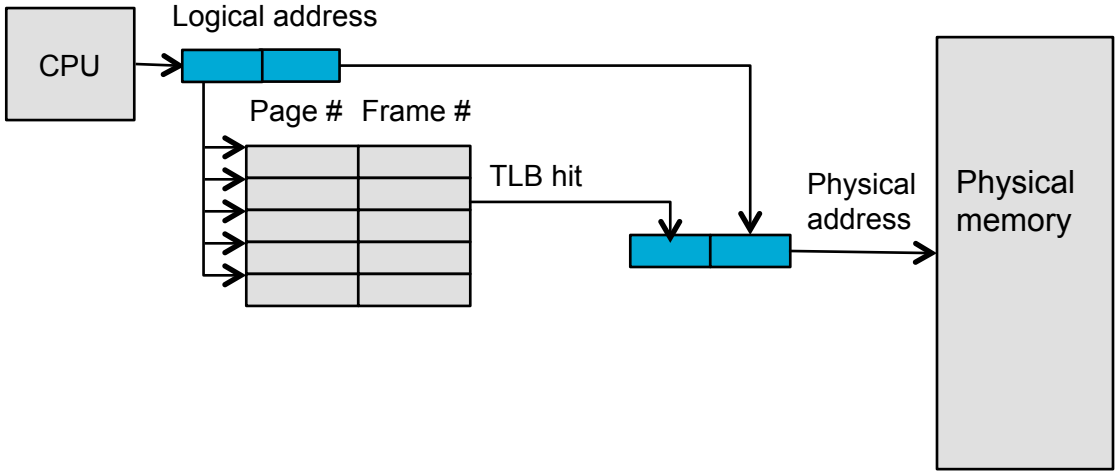
TLBs

- Translates virtual page #s into page frame #s
  - Can be done in single machine cycle
- Implemented in hardware
  - A fully associative cache (parallel search)
  - Cache tags are virtual page numbers
  - Cache values are page frame numbers
    - With this + offset, MMU can calculate physical address
  - A typical TLB entry might look like this

    VPN | PFN | Other bits

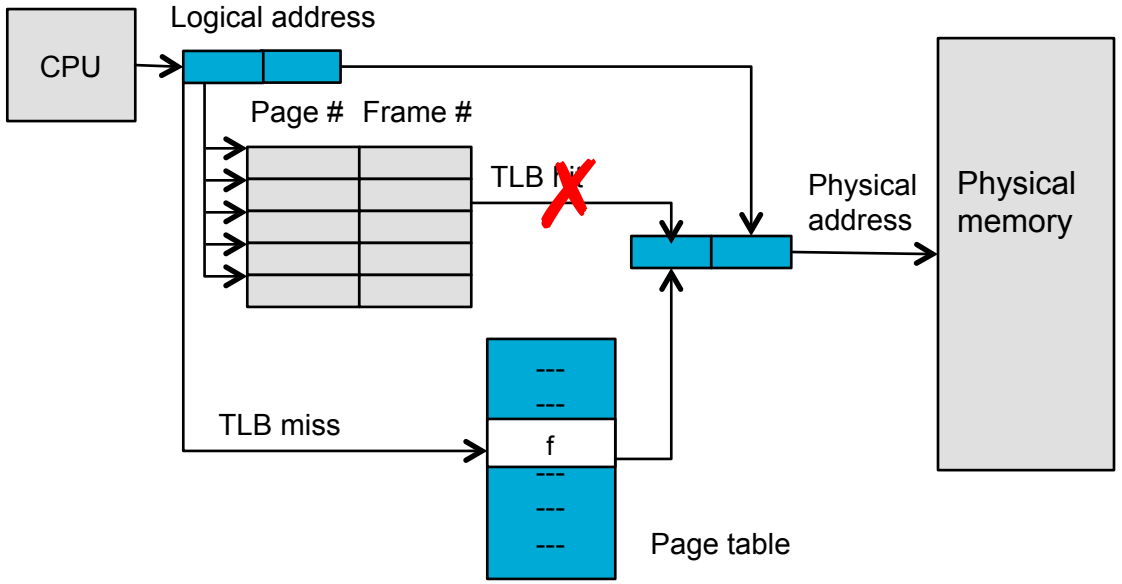
TLBs hit

```
VPN = (VirtAddr * VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
If (Success == True) // TLB Hit
   if (CanAccess(TlbEntry.ProtectBits) == True)
       Offset = VirtAddr & OFFSET_MASK
       PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
       Register = AccessMemory(PhysAddr)
   else
     RaiseException(PROTECTION_FAULT)
```



TLBs miss

```
else                                // TLB Miss
   PTEAddr = PTBR + (VPN * sizeof(PTE))
   PTE = AccessMemory(PTEAddr)
   if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
   else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```



Managing TLBs

- Address translations mostly handled by TLB
  - >99% of translations, but there are TLB misses
  - If a miss, translation is placed into the TLB
- Who manages the TLB miss?
  - Hardware (memory management unit – MMU)
    - Knows where page tables are in memory
      - OS maintains them, HW access them directly
  - E.g., Intel x86
- Software loaded TLB (OS)
  - TLB miss faults to OS, OS finds page table entry & loads TLB
  - Must be fast
    - CPU ISA has instructions for TLB manipulation
    - OS gets to pick the page table format
  - E.g. MIPS R10k, Sun's SPARC v9

Managing TLBs

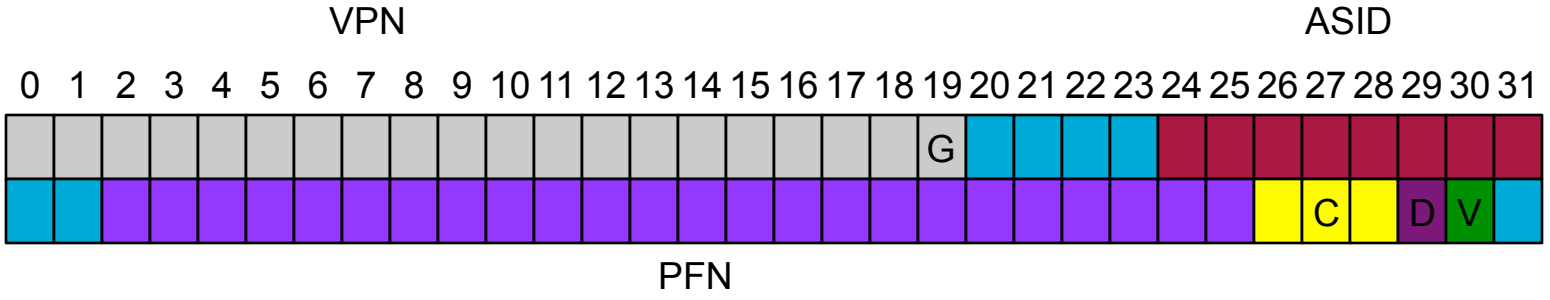
- OS must ensure TLB and page tables are consistent
  - When OS changes protection bits in an entry, it needs to invalidate the line if it is in the TLB
- When the TLB misses, and a new process table entry is loaded, a cached entry must be evicted
  - How to choose a victim is called “TLB replacement policy”
  - Implemented in hardware, usually simple (e.g., LRU)

Managing TLBs

- What happens on a process context switch?
  - Remember, each process typically has its own page tables
  - Need to invalidate all the entries in TLB! (flush TLB)
    - A big part of why process context switches are costly
  - *Can you think of a hardware fix to this?*
  - Add an Address Space Identifier field to the TLB

An example TLB

- From MIPS R4000 – software-managed TLB
  - 32b address space with 4KB pages
    - 20b VPN and 12b offset
  - But TLB has only 19b for VPN!
    - *User addresses will only come from half the address space – the rest is for the kernel*
    - So 19b is enough



An example TLB

- …
  - VPN translates to up to a 24b physical frame number and can thus support systems with up to 64GB of physical memory
  - *How is that?!*
  - *G* is for pages globally shared (so ASID is ignored)
  - *C* is for coherence; *D* is for dirty and *V* is for valid
  - Since it is software managed, OS needs instructions to manipulate it
    - TLBP – probes
    - TLBR – reads
    - TLBWI and TLBWR to replaces a specific or a random entry

Effective access time

- Associative Lookup = ε time units
- Hit ratio - α - percentage of times that a page number is found in the associative registers (ratio related to TLB size)

Effective Memory Access Time (EAT)

TLB hit

EAT = α * (ε + memory-access) +

TLB miss

(1 - α) (ε + 2* memory-access)

Why 2?

α = 80% ε = 20 nsec memory-access = 100 nsec

EAT = 0.8 * (20 + 100) + 0.2 * (20 + 2 * 100) = 140 nsec

Next time

- Virtual memory policies
- Some other design issues