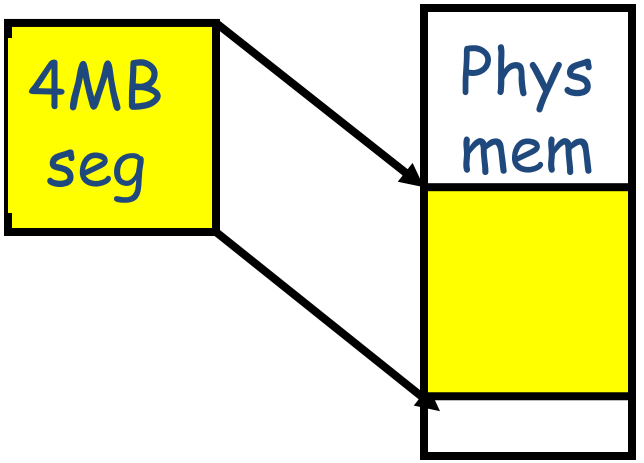
CSL373: Operating Systems more VM fun

The past: translating VAs to PAs

- Load-time reloc: translate when load from disk
  Pro: don't need hardware support
  Con: hard to redo after running + no protection
- Base & Bounds = add base and check bounds
  Pro: simple relocation + protection, fast
  Con: inflexible
- Segmentation = multiple base & bounds (segments)
  Pro: Gives more flexible sharing and space usage
  Con: segments need contiquous memory; fragmentation
- Paging: quantize memory into fixed pages, map these
  Pro: Eliminates external fragmentation; flexible mappings
  Con: internal frag; mapping contiguous ranges more costly
- Today: paging + segmentation & speed issues

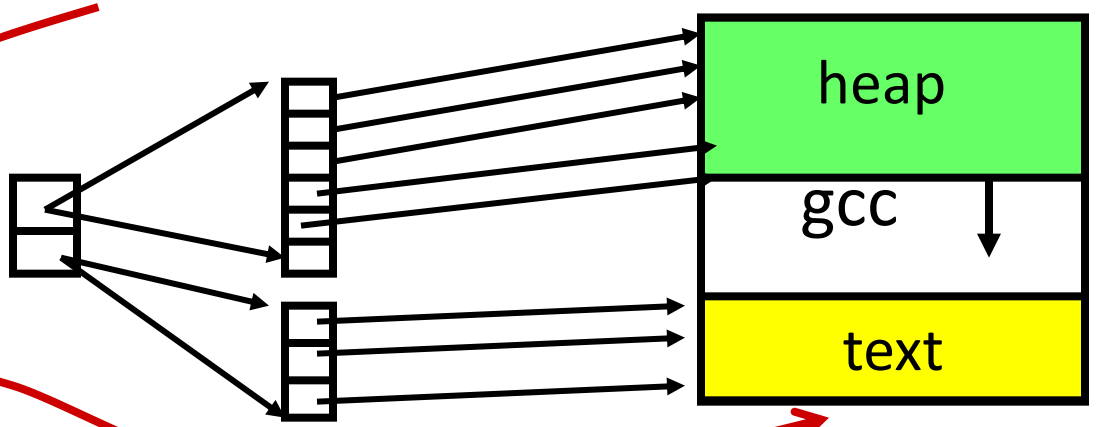
Paging + segmentation: best of both?

- Dual problems:
  Paged VM: simple page table
  = lots of storage
  Segmentation: All bytes in segment must map to contiguous set of storage locations. Makes paging problematic: all of seg in mem or none.



- Idea: use paging + segmentation!
  Map program mem with page table
  Map page table mem with seg table

Call virtual address chunk
a "segment"
Each segment has
>= 1 pages



Paging + segmentation tradeoffs

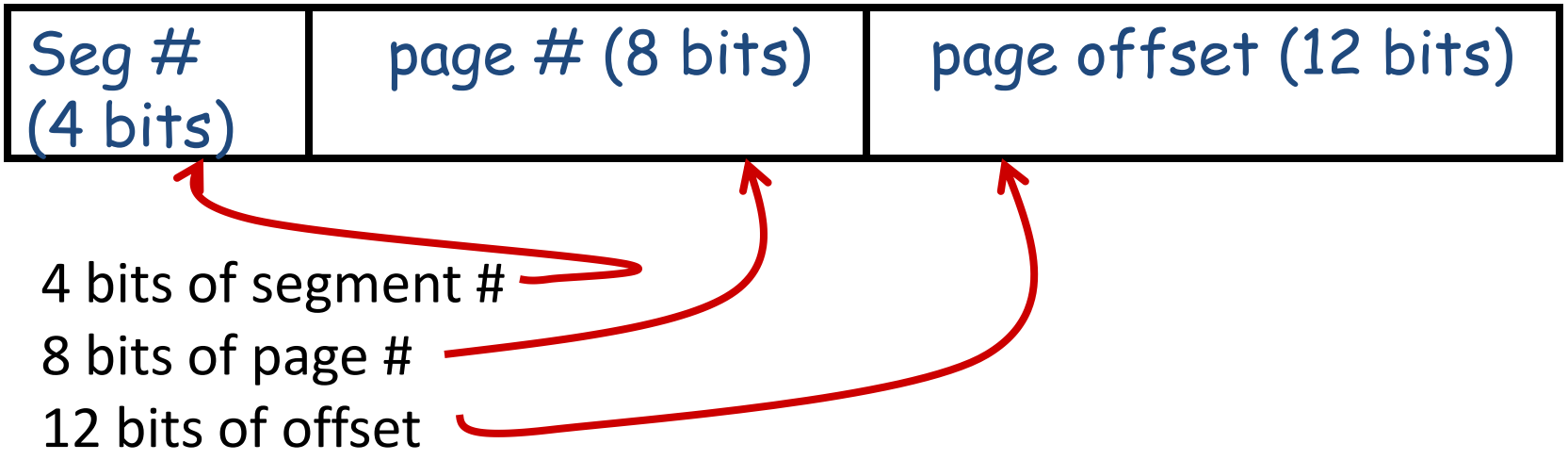
- Page-based virtual memory lets segments come from non-contiguous memory
  - marks allocation flexible
  - portion of segment can be on disk!

- Segmentation = way to keep page table space manageable
  - Page tables correspond to logical units (code, stack)
  - Often relatively large
  - (But what happens if page tables really large?)

- Going from paging to P+S is like going from single segment to multiple segments, except at a higher level.
  - Instead of having a single page table, have many page tables with base and bound for each.

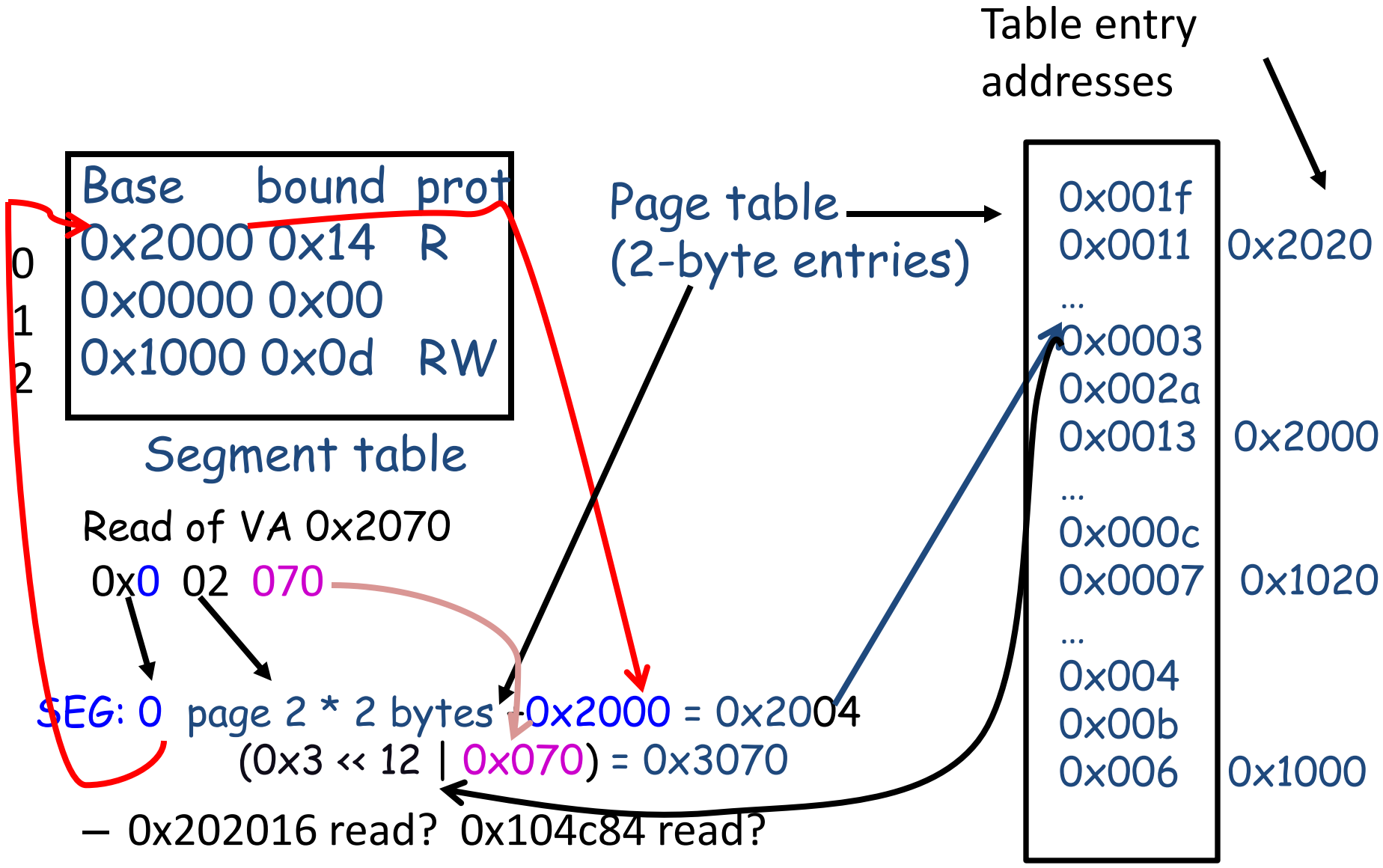
Example: system 370

- System 370: 24 bit virtual address space (old machine):



- The mappings:
  - Segment table: maps segment number to physical address & size of that segment's page table.
  - Page table maps virtual page to 12 bit physical page number, which is concatenated to the 12 bit offset to get a 24 bit address

Example system 370 translation



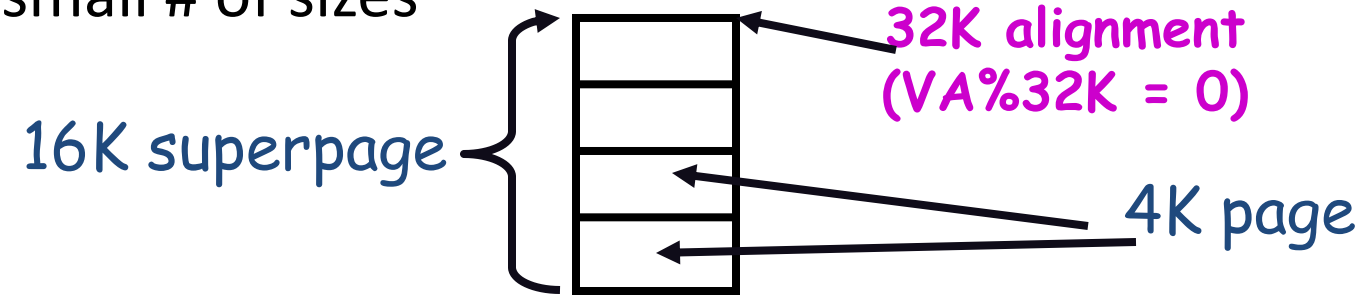
- 0x202016 read? 0x104c84 read?
- 0x011424 read? 0x210014 write?

P+S discussion

- If segment not used then no need to have page table for it
- Can share at two levels:
  - single page or single segment (whole page table)
- Pages eliminate external fragmentation and make it possible for segments to grow, page without shuffling
- If page size small compared to most fragments then internal fragmentation not too bad (.5 of page per seg)
- User not given write access to page tables
  - Read access can be quite useful (e.g., to let garbage collectors see dirty bits), but only experimental OSes provide this…
- If translation tables kept in main memory, overhead high
  - 1 or 2 overhead reference for every real reference (i.e., mem op)
- Other example systems: VAX, Multics, x86, …

Who won?

- Simplicity = good (and fast): Most new architectures have gone with pages
  - Examples: MIPS, SPARC, Alpha.
  - And many old architectures have added paging even if they started with segmentation!
- But: to efficiently map large regions, many new machines backsliding into "super pages"
  - Large, multi-page pages: have strict alignment restrictions; (typically) small # of sizes



  - inflexibility = speed, but handles 80% of the cases we want (e.g., that 8MB framebuffer)

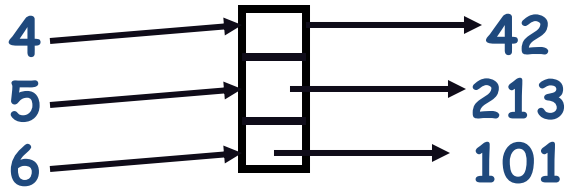
Virtual memory is complicated, but not deep

- Again: Despite the obscure terminology, all virtual memory is trying to do is map ints to ints:

VA:int ⟶ PA:int

- Most complexity from fact that CS has no good way to construct a general integer function.

  Mapping between two arbitrary sets of integers, fundamentally requires a table of some sort



  Page table = a lookup table to manually build function

  The usual variations: an array ("direct" page table), hash tables, weird trees of arrays.

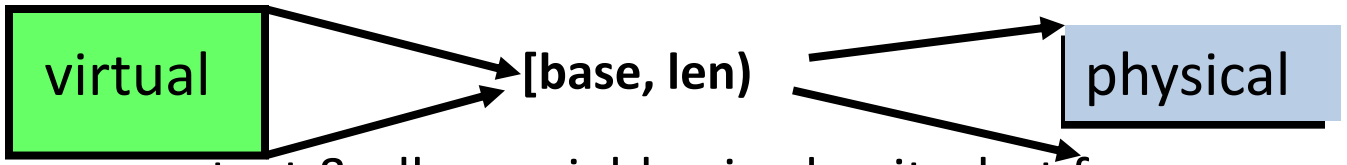
  The usual complications: speed, space

Paging and segmentation not so different

- Main difference is the way they map ints.

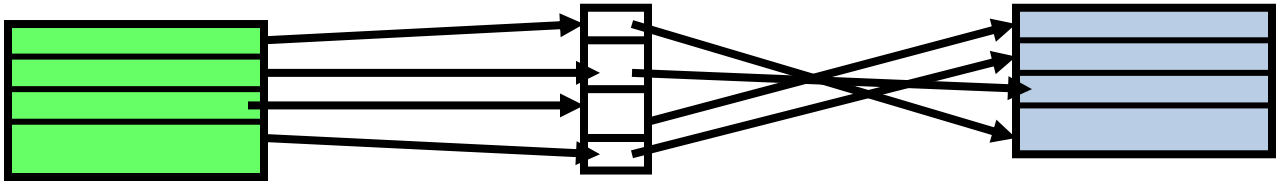
  Segmentation: represents VA's as byte ranges ([va, nbytes])

  Restricts outputs so it can map them using a single table entry



  result: can protect & alloc variable-sized units, but force mapped range to be contiguous, hard code index for speed.

  Paging: represents VA's as pages, maps them using a tuple ([vpn, ppn]) for each page sized unit (forces base to be page aligned)



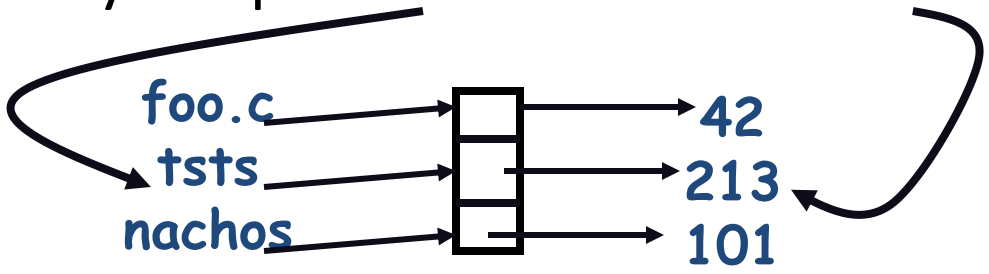
  Result: can only protect and allocate fixed-size units, but can map any page to any other.

  Hardware caches and lookups differ, but only because of different table layouts and logic, nothing fundamental…

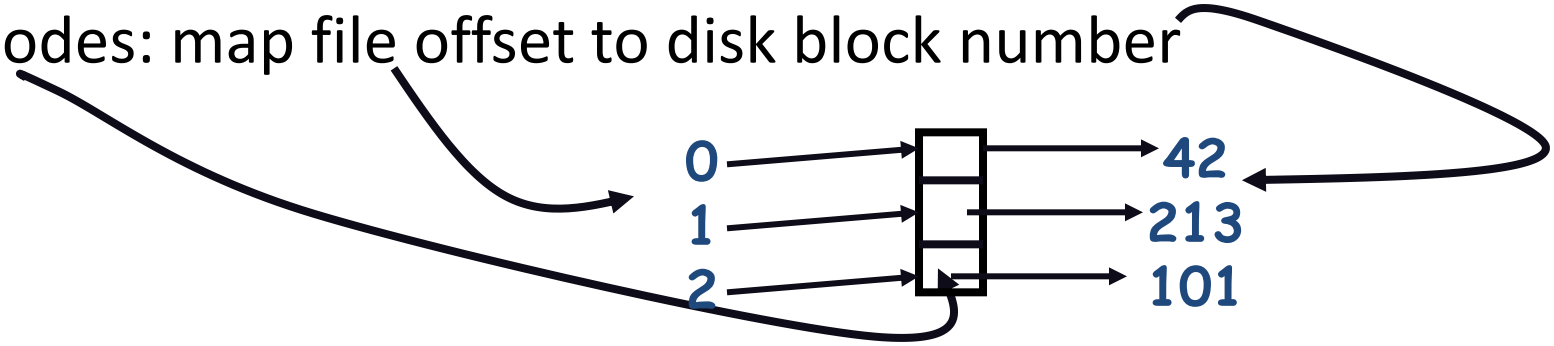
Mapping functions: a perennial OS/CS theme

- OSes are constantly in the business of constructing a mapping function and then trying to make it fast
- Example: File systems:

  Directory: map file name to inode



  inodes: map file offset to disk block number



- Others: DNS names to IP addrs, IP addrs to eth, to routes, variables to registers, symbols to VAs, …

Variable versus fixed size

- Can view as space/flexibility vs time

  fixed size: simple, fast but inflexible + internal frag

  nice: allows simple, fast int-to-int mappings

  e.g., computing address of array[i] = base+ i*element size

  variable size: more flexible but complex, slower, external frag

- Constant computer science theme:

  variable sized instructions (CISC) vs fixed size (RISC)

  variable sized packets (Ethernet) vs fixed size (ATM)

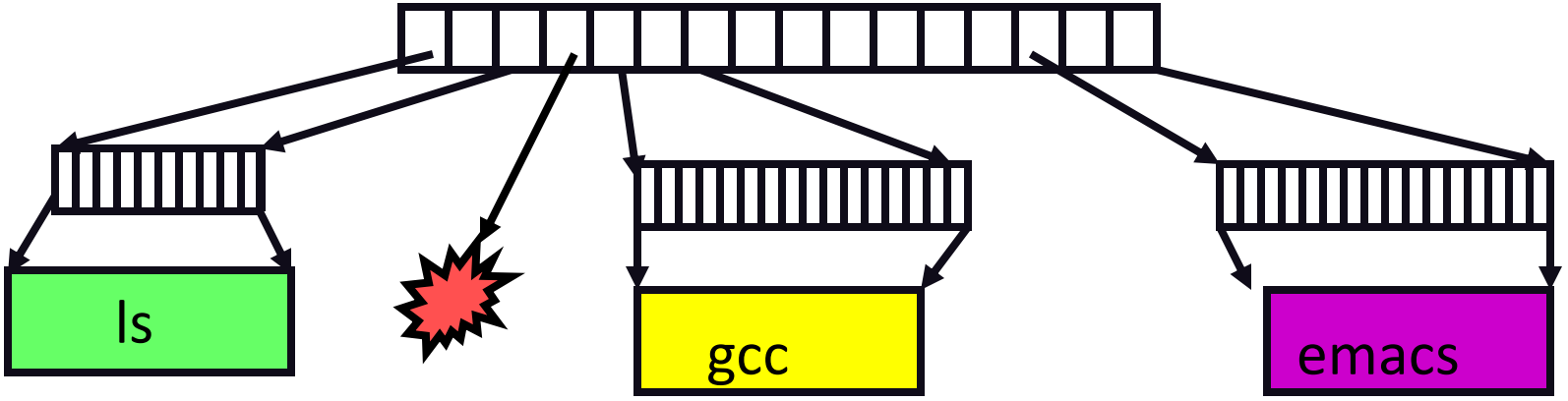
  fixed sized examples: cache entires, disk blks, IP addrs

  variable sized e.g.'s: variables names&sizes, files, DNS names

  our primal mud: digital (discrete) vs analog (continuous signal)

Problem: large range = large page tables

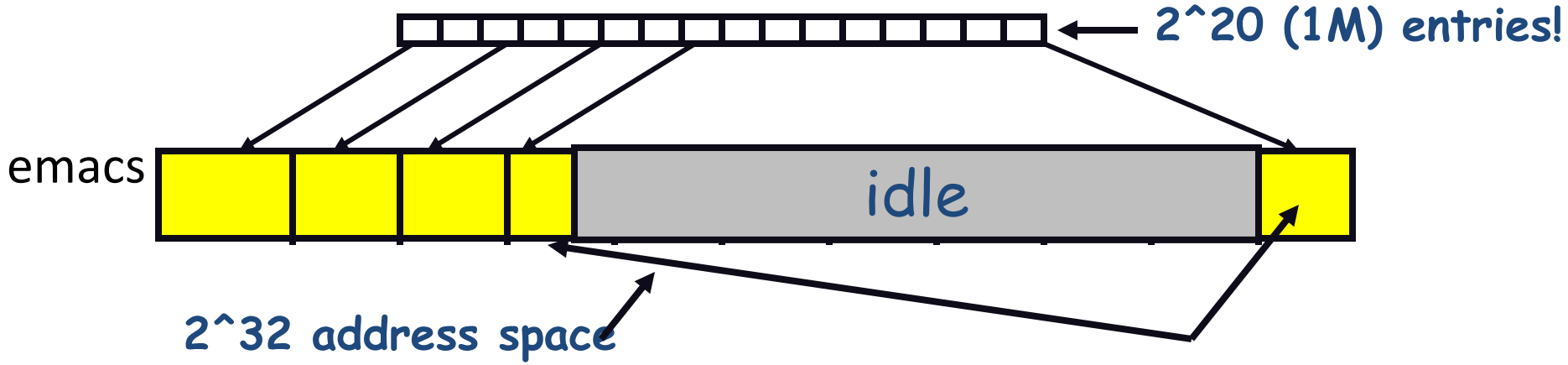
- Same problem as memory: Don't want to have to allocate page tables contiguously
  - So use same solution: map page tables using another page table
  - To stop recursion: the page-table page table ("system page table") resides in contiguous memory, usually at a known fixed address



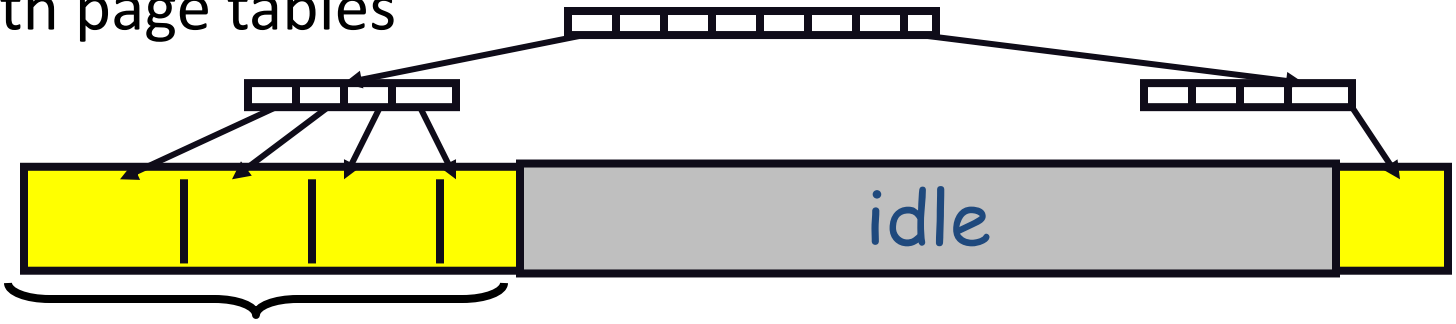
  - Win: page tables can be pieced together from scatterred pages
  - Win: invalid mappings can be represented with invalid addresses rather than requiring space in a table
  - Lose: Worst case lookup?

Extending idea: hierarchical page tables

- Large address range = lots of unused space = unwieldy PTs



- Conceptually: map small regions with direct page table, then these with page tables



Space savings with 2^22 (4M) regions?

When are hierarchical (much) worse than non-hierarchical?

Inverted Page Table

- One entry for each physical page frame specifying PID and VPN for that page

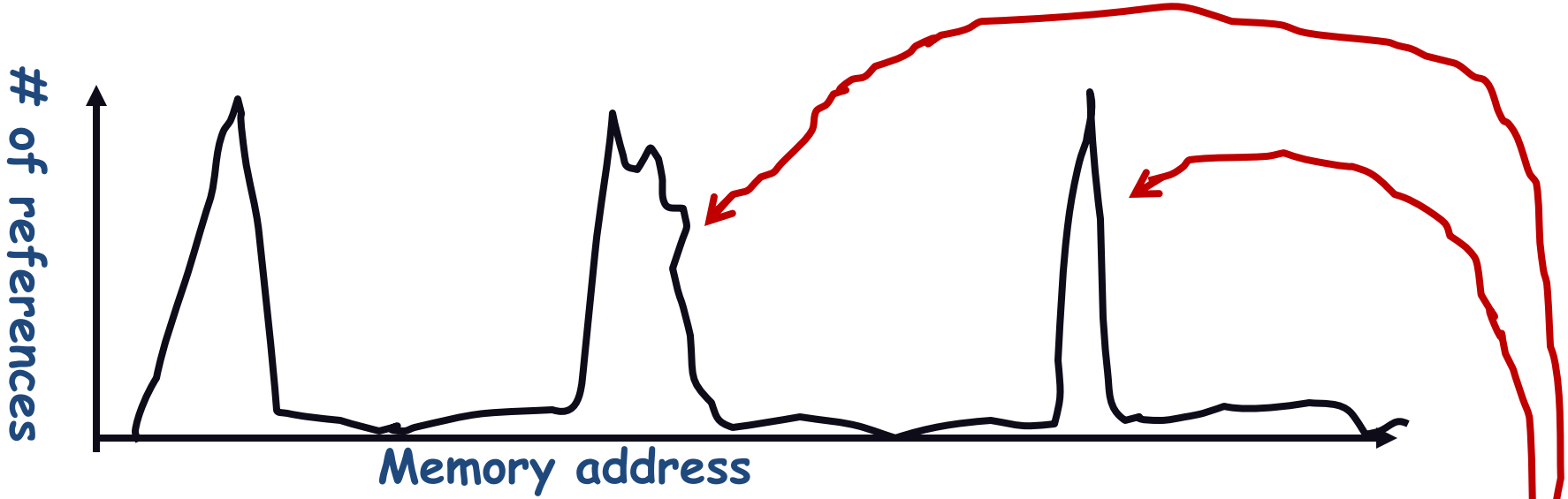
- On a TLB miss, search inverted page table data structure to find physical page frame for the virtual address of process. Speed up using:
  - Hashing
  - Associative table of recently accessed entries (h/w)

  e.g., IBM (RS/6000, RT, System38), HP Spectrum

Problem: mapping = slow

- If each memory reference requires 1 or more page table translations, speed will be bad
- The obvious idea: caching

  What to cache? VA-to-PA translations

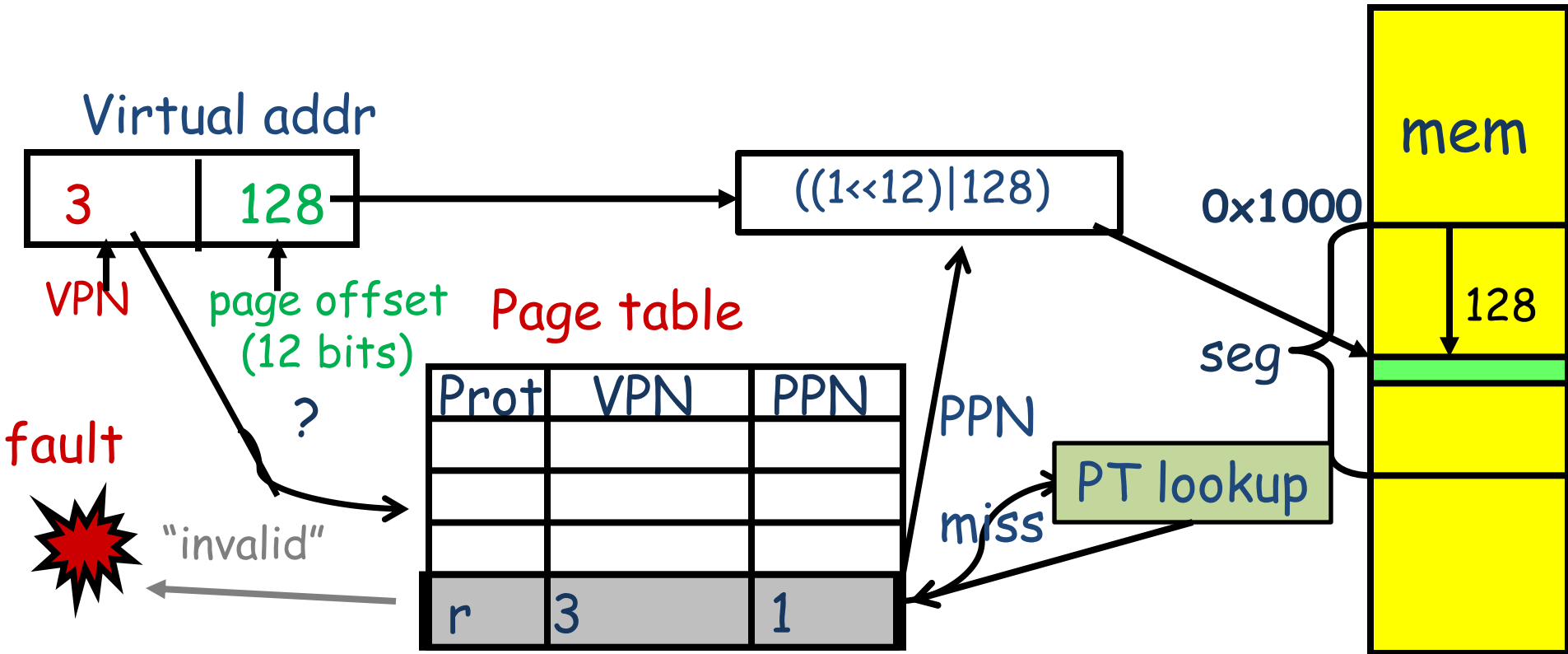


Why? Program doesn't wildly access entire address space usually references (relatively) small, slowly-changing subsets

So, just cache the translations for these regions!

Translation look-aside buffer (TLB)

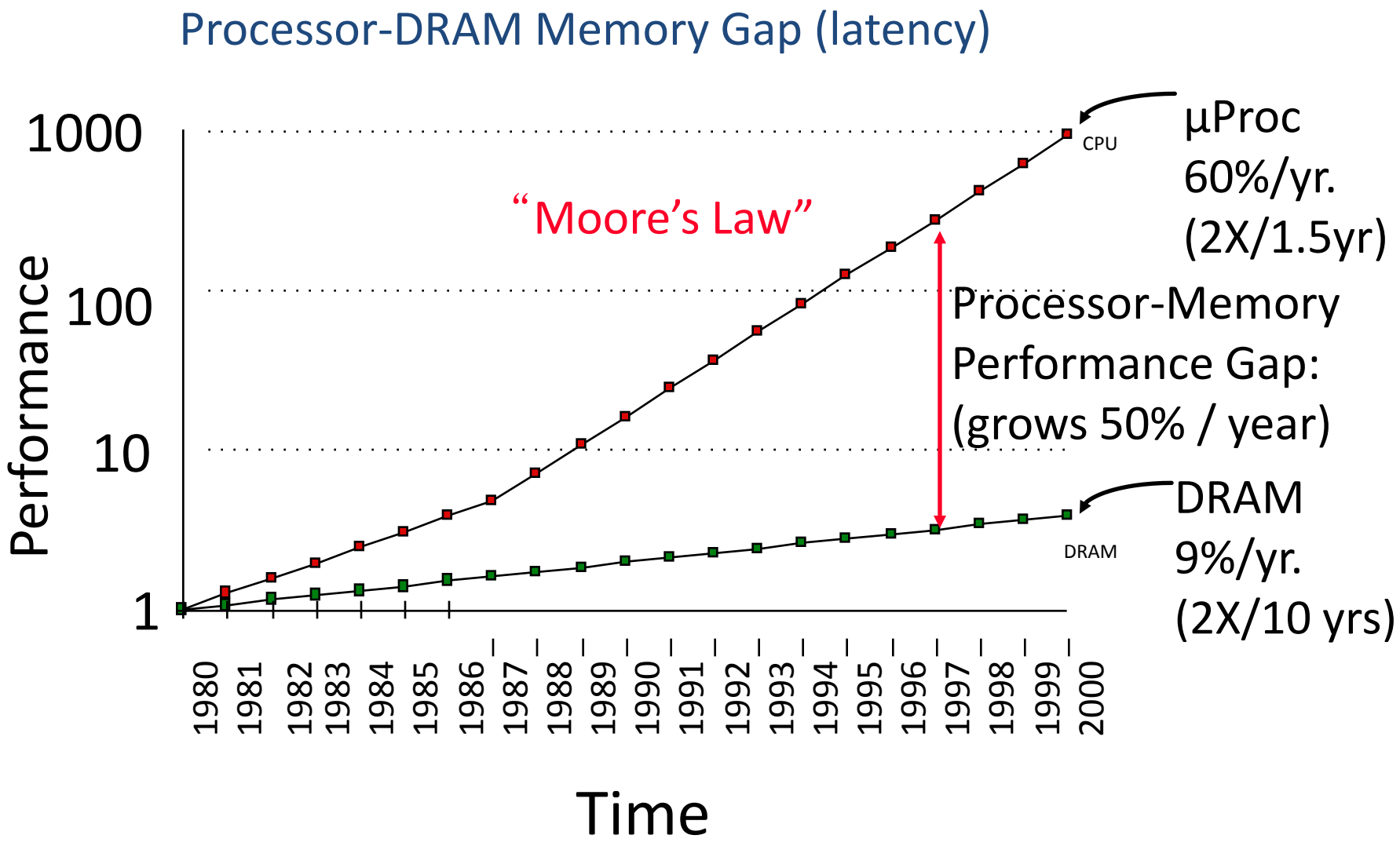
- TLB is just a very fast memory with some comparators
  - Each TLB entry maps a VPN to PPN + protection information
  - On each memory reference: check TLB, if there, fast. If not insert in TLB for next time. (Must remove some entry)
  - Typical: 64-2K entries, 95% hit rate



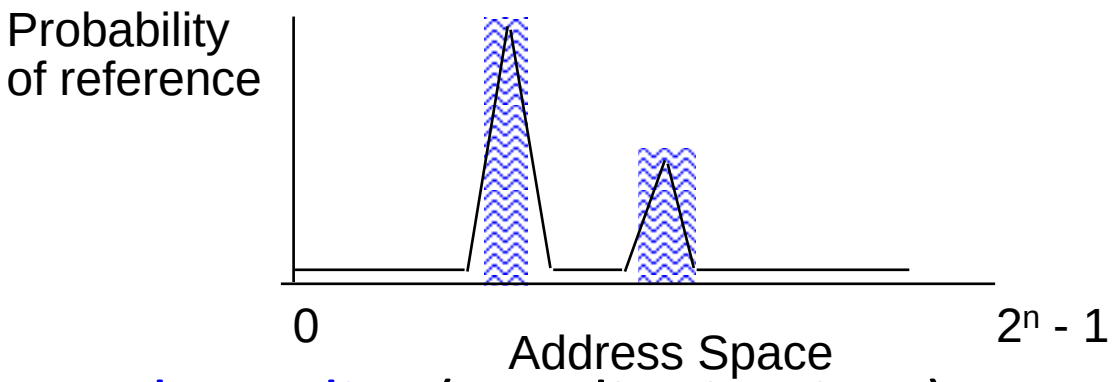
Caching in real world

- Refrigerator -- kitchen
- Book issue -- library
- Water bottle – water cooler
- Lecture slides – book
- More?

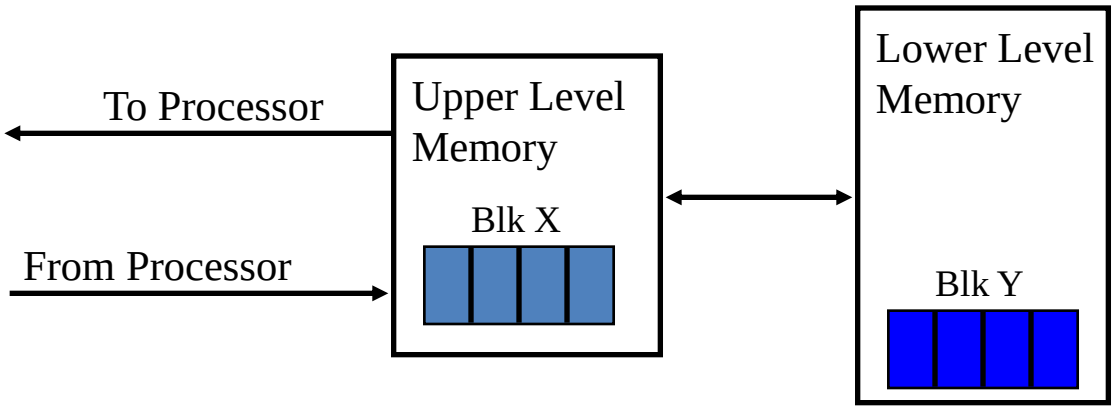
Why Bother with Caching?



The precondition to caching: locality

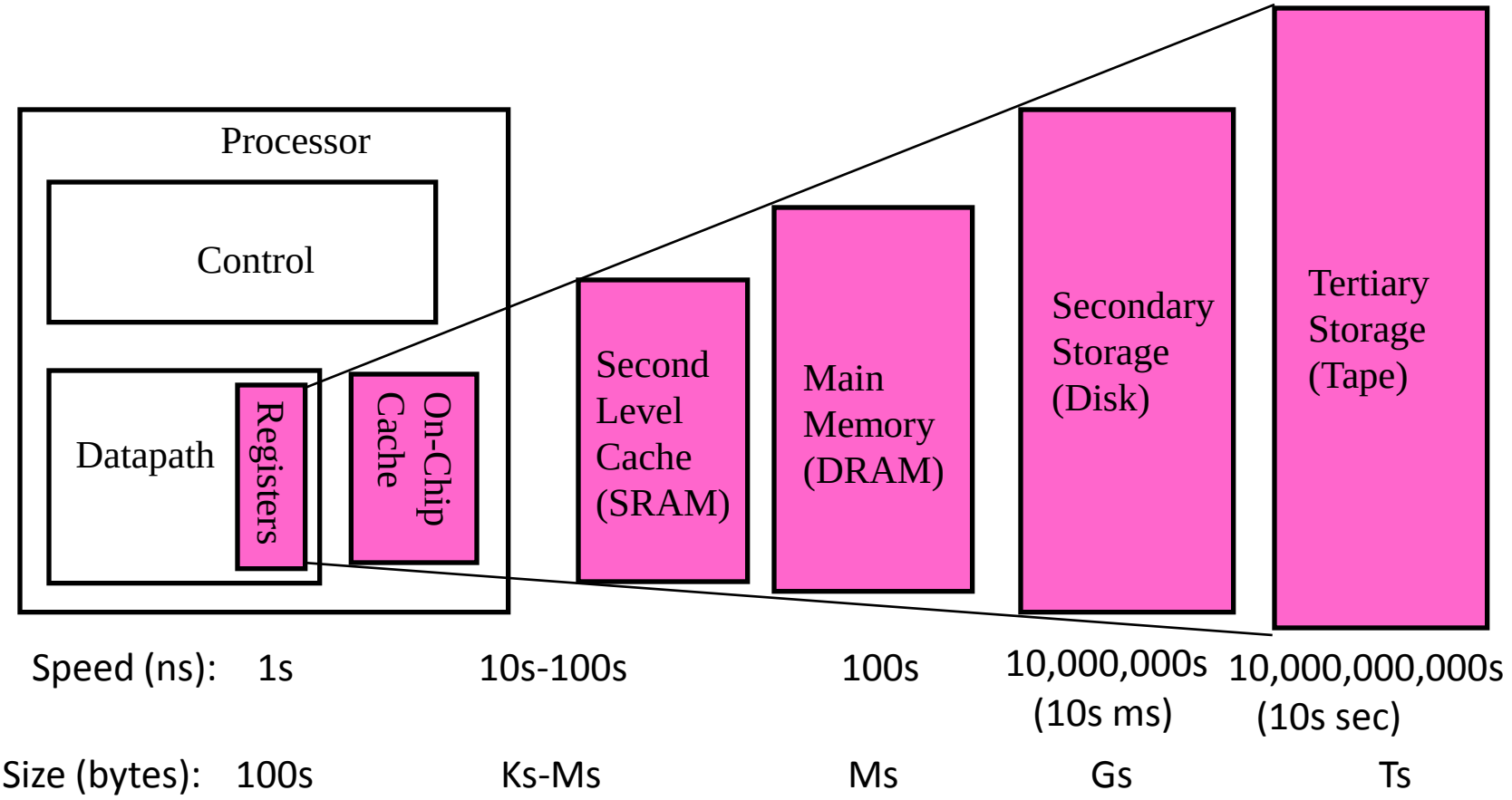


- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



Memory Hierarchy of a Modern Computer System

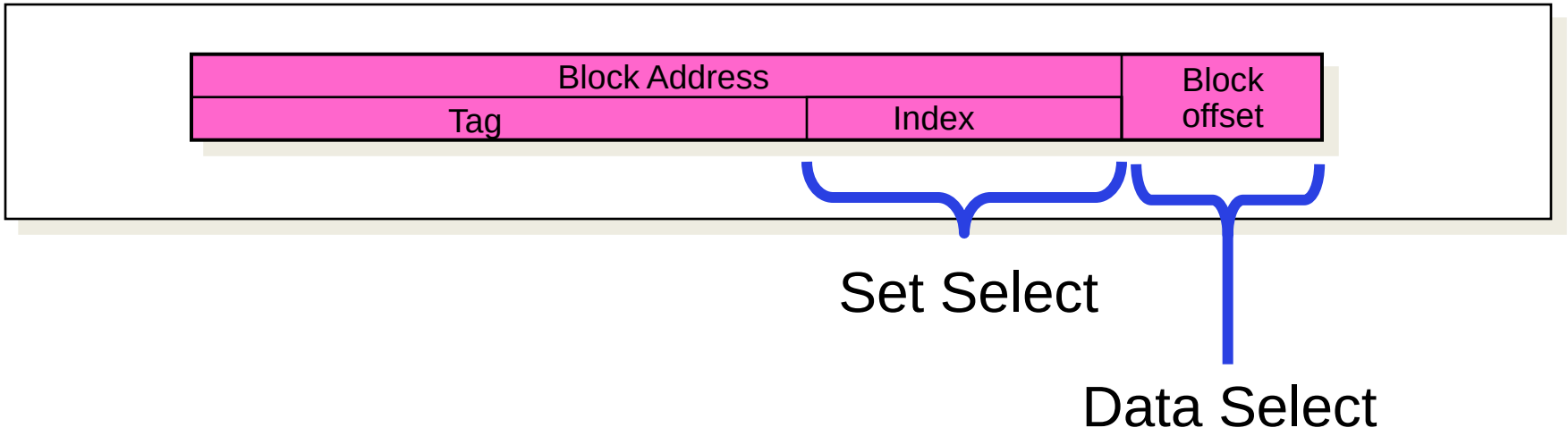
- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



A Summary on Sources of Cache Misses

- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

How is a Block found in a Cache?



- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field

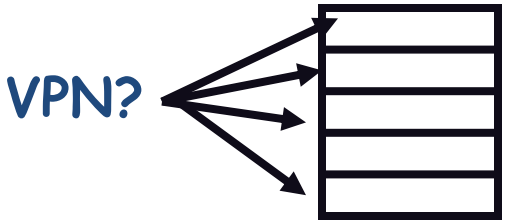
The (usual) possible cache organizations

- Fully associative:

  Entries can go anywhere

  replacement?

  minimizes conflict misses, but lookup is expensive: requires searching entire table on every memory ref

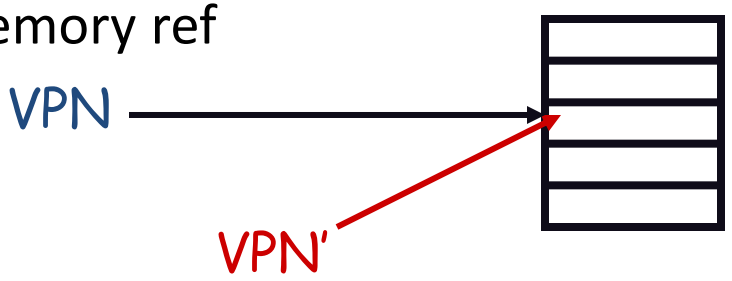


- Direct mapped:

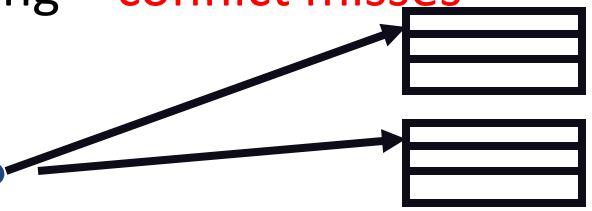
  entries can go exactly one place

  example: position = VPN % TLB-size

  fast location but inflexible mapping = conflict misses



- Hybrid: Set associative



  Entries can go in any of the sets (set assoc) but exactly in one place in each. (Search sets in parallel = speed)

TLB details

- TLB like hash table, but simpler
  fixed size, no pointer chains

- How to build hash function?
  In general: more information = better decisions.
  In this case exploit fact about programs: spatial locality.
  Better to use upper to lower address bits to map to entry?
  If we have temporal locality, what are good to replace?
  (Also, allow system to reserve space for some translations.)

- Complication: what to do when switch address space?
  Flush TLB on context switch (e.g., x86)
  Tag each entry with associated process's ID (e.g., MIPS)
  In general, OS must manually keep TLB in valid state

MIPS R3000

- L1 cache: 64K bytes, direct mapped. Write allocate, Write through. Physically addressed
- Write buffer: 4 entries
- L2 cache: 256K bytes, direct mapped. Write back.
- Physical Memory: 64M bytes, 4K page frames.
- Backing Store Disk: 256M bytes of swap space

- TLB = 64 entry fully associative

MIPS R3000 Segments

- **kuseg**: top bit=0
- **kseg0:** top bits=100. Cached and unmapped. Kernel instructions and data
- **kseg1:** top bits=101. Uncached and unmapped. Disk buffers, I/O registers, ROM code.
- **kseg2:** top bits=11. Cached and Mapped. Map different for each process. Per-process kernel data structures (e.g., user page tables)

Translation of user addresses

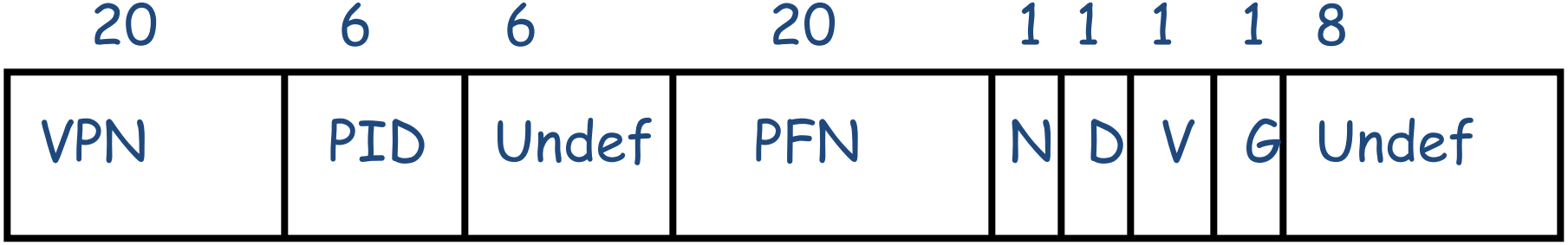
- 31 bits of virtual address + 6 bits of process ID → 32 bit physical address
- Page size = 4k. So VPN=how many bits?
- Mapping VPN to PPN using a linear page table? Where is this page table stored?
- Page the page table using top 9 bits. Where is the page table for the page table stored?

Example: MIPS R2000/R3000 TLB

- Used in DecStations and SGI machines

  64 entries, fully associative

  TLB entry format (64 bits per entry)



G – Global, valid for any PID (why?)

V – entry is valid (why have this?)

D – dirty bit, page has been modified

N – don't cache memory addresses (why?)

PFN – physical address of the page

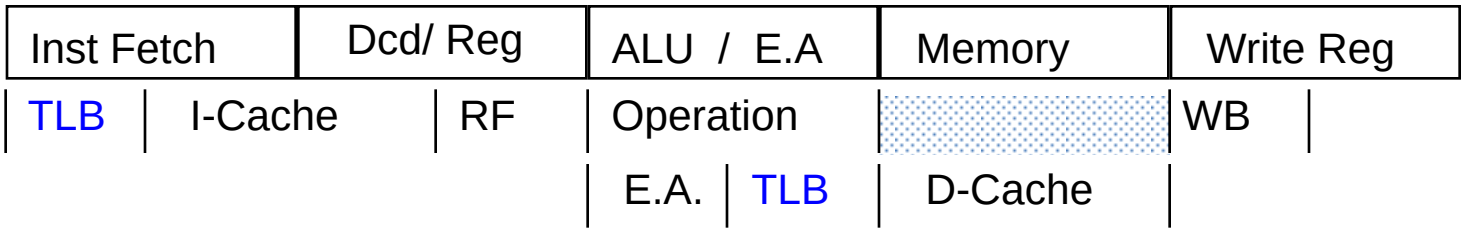
PID – process id for the page (how many? How to reuse?)

VPN – virtual page number

Undef - undefined bits, set by OS (why?)

Example: R3000 pipeline includes TLB “stages”

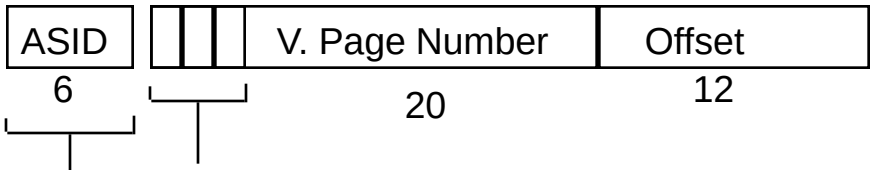
MIPS R3000 Pipeline



TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

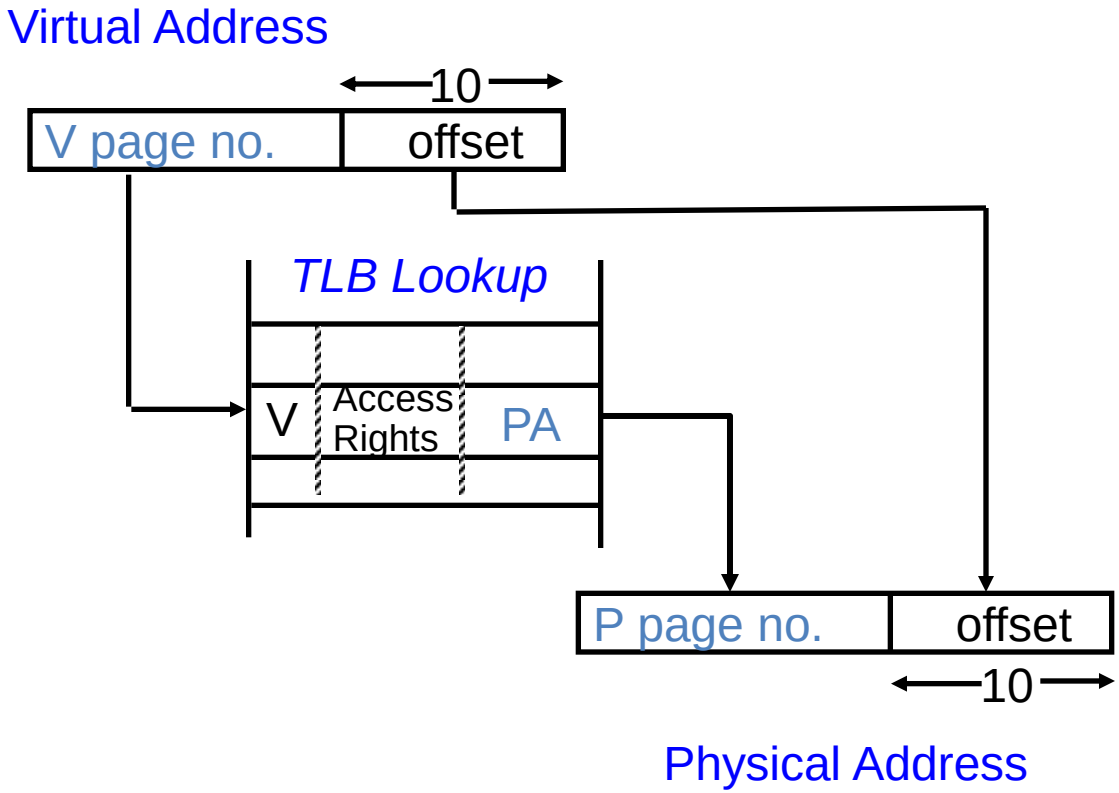


0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Reducing translation time further

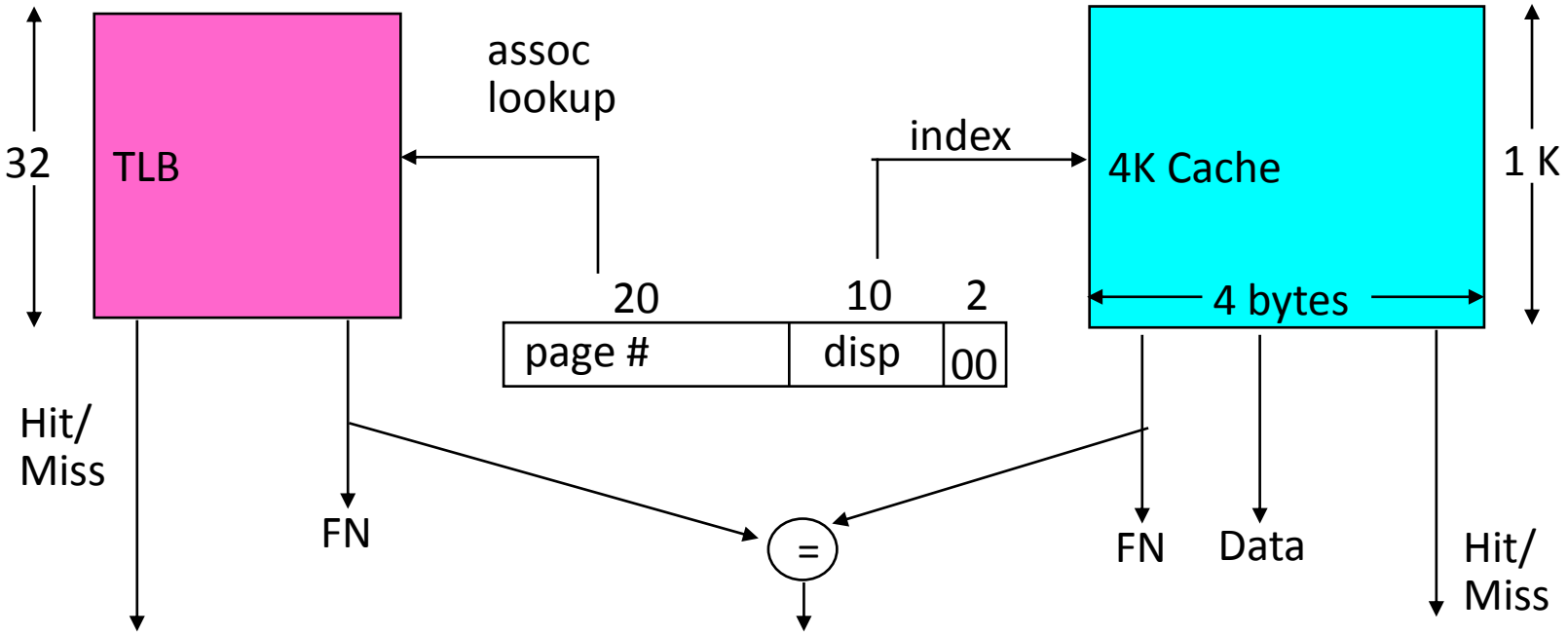
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:



MIPS R3000 Lookup

- Map on upper half of TLB entry, use lower half of TLB entry. Can generate 3 different types of TLB misses:
  - UTLB miss (access to kuseg, no match in TLB)
  - TLB miss (access to kseg0, kseg1, kseg2. no match)
  - TLB mod (mapping is loaded, but access is a write and the D bit is not set)

Each type of miss has it’s own exception handler

Alternatives

- OS may run completely unmapped. Problem: can’t page OS data structures
- OS may have separate address space from user. Problem: can’t as easily access user space
- Cache may be virtually addressed. Problem: flush cache on context switch, can’t alias multiple virtual addresses to same physical address
- TLB reload may be done entirely in hardware. Problem: Locks OS into using a specific data structure for page table (x86)

Problem: accessing user data

- OS needs to access user memory
  - I/O routines read/write user buffers. BUT:
  - User pointers cannot be trusted
    - obvious: user passes in null pointer or invalid address “0xfffee000”
    - Less obvious: user passes in valid *kernel* address

- User memory might be paged out.
  - OS will get a page fault in middle of system call
  - If it switches to new process then:
    - Prob 1: if kernel held single OS lock, system will deadlock
    - Prob 2: if system call halfway through, and depends on current state, when restarted will have invalid view of world
    - Prob 3: if system call partially altered OS data structures…

- User pointers only valid in user address space
  - What happens if OS stores them away?

Where does OS live?

- Different address space?

  "Microkernels"

  Nice: "catch" wild pointer writes

  Bad: Accessing user stuff clumsy.

- Common: user addresses co-exist with OS's

  OS aliased into every application address space at the same place (allows user addresses to co-exist with OS's)

  e.g., Windows32 (2GB OS + 2GB user)

  Linux32 (1GB OS + 3GB user)

  As a refinement: OS runs "unmapped"

  special address range in "privileged" mode that mapped to physical memory by subtracting constant

  e.g., Linux: addresses >= 0xc0000000 have this subtracted

  What happens when an application tries to read/write OS data?

Summary: Virtual memory mapping

- What?

  Give programmer logical (virtual) set of addresses rather than actual physical set

- How?

  translate every memory operation using table (page table, segment table)

  Speed: cache frequently used translations

- Result?

  each user has a private address space

  programs run independently of actual physical memory addresses used, and actual memory size

  protection: check that they only access their own memory