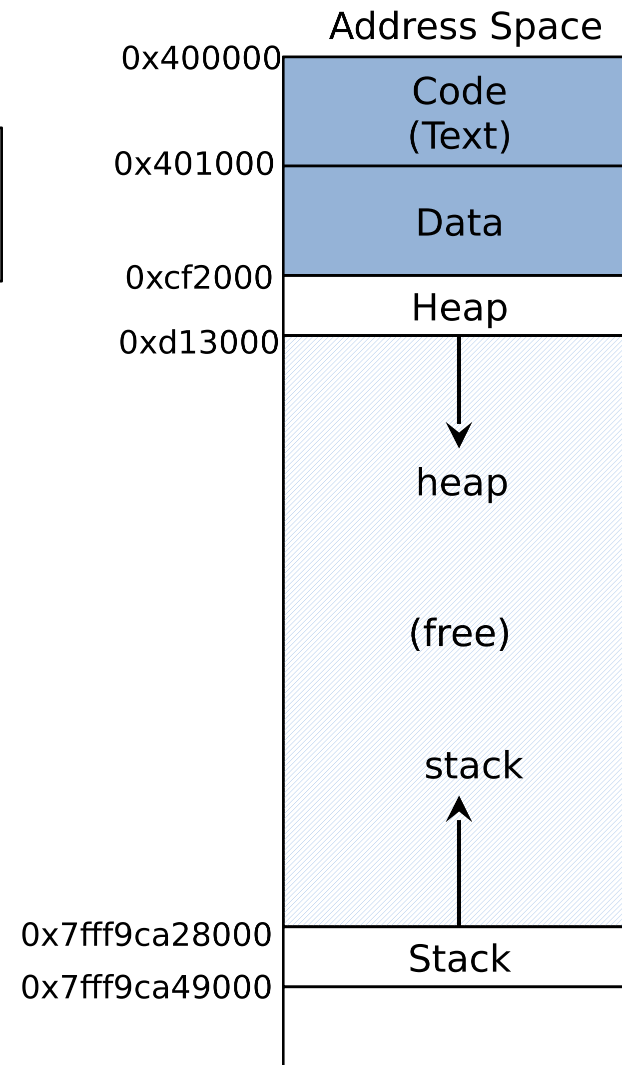
Virtual Memory

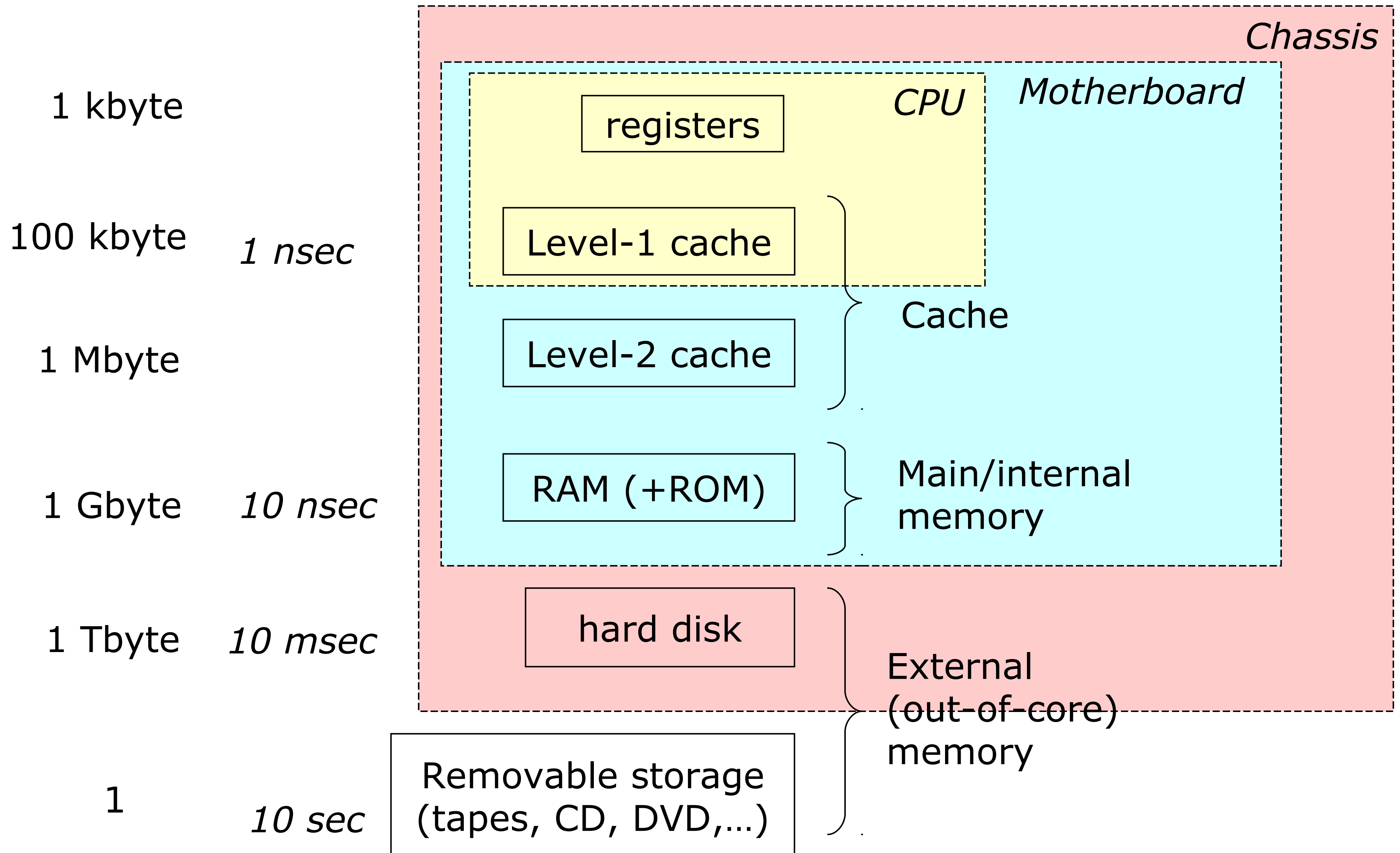
Yannis Smaragdakis, U. Athens

Example Modern Address Space
(64-bit Linux)

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



(Sample) Memory Hierarchy



Questions

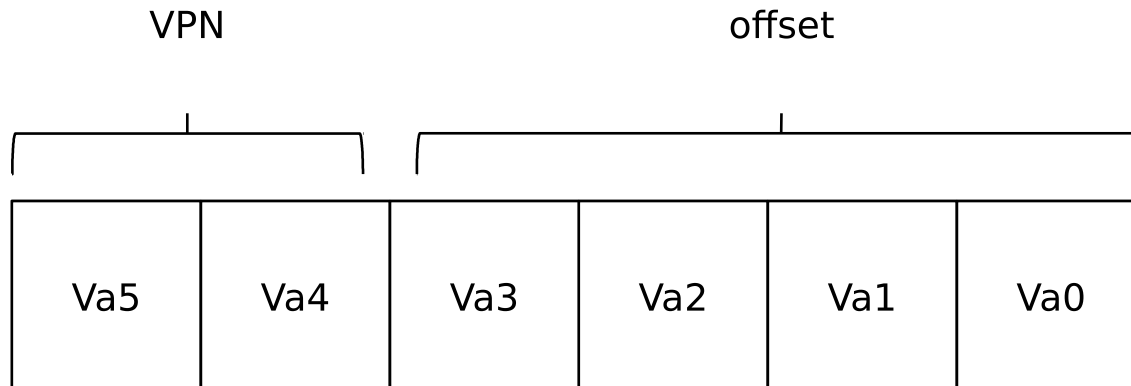
- Which addresses are physical, which virtual?
  - what happens with multiple processes?
  - who translates addresses?
- Why do caches work well? (locality!)
- How do you get more stack memory? More heap memory? How do you return it?
- What do you think the OS does? What system calls?
  - “man mmap”, “man sbrk”

How Do Addresses Get Translated?

- Several older techniques:
  - base and bounds registers
  - segmentation
    - with hw support, on x86
    - “segmentation fault” message?
- Modern technique: paging
  - also on xv6

Paging

- Translate addresses at page granularity
  - addr = virtual page num (VPN) + offset
- Keep directory of page mappings



Page Tables

- A directory of page addresses
  - page table entry: may include other info
    - valid, protected, present, dirty, recently-accessed bit
- What data structures can you imagine?
- What's the **space** overhead of each?

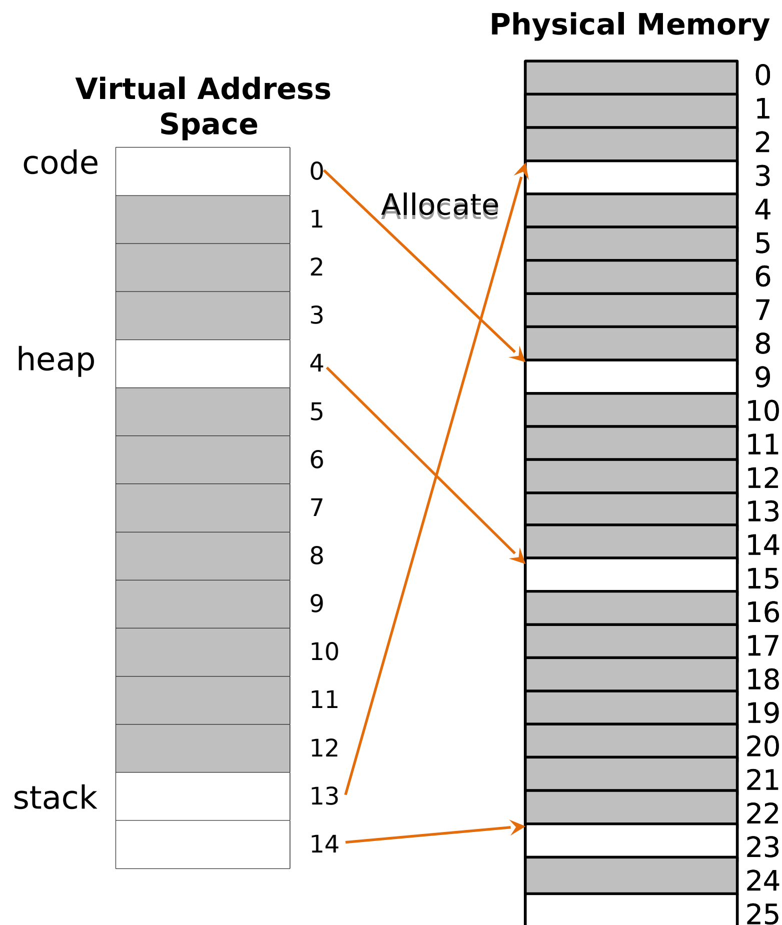
Translation Logic (pseudocode)

```
1    // Extract the VPN from the virtual address
2    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4    // Form the address of the page-table entry (PTE)
5    PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7    // Fetch the PTE
8    PTE = AccessMemory(PTEAddr)
9
10   // Check if process can access the page
11   if (PTE.Valid == False)
12       RaiseException(SEGMENTATION_FAULT)
13   else if (CanAccess(PTE.ProtectBits) == False)
14       RaiseException(PROTECTION_FAULT)
15   else
16       // Access is OK: form physical address and fetch it
17       offset = VirtualAddress & OFFSET_MASK
18       PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19       Register = AccessMemory(PhysAddr)
```

What is the **time** overhead?

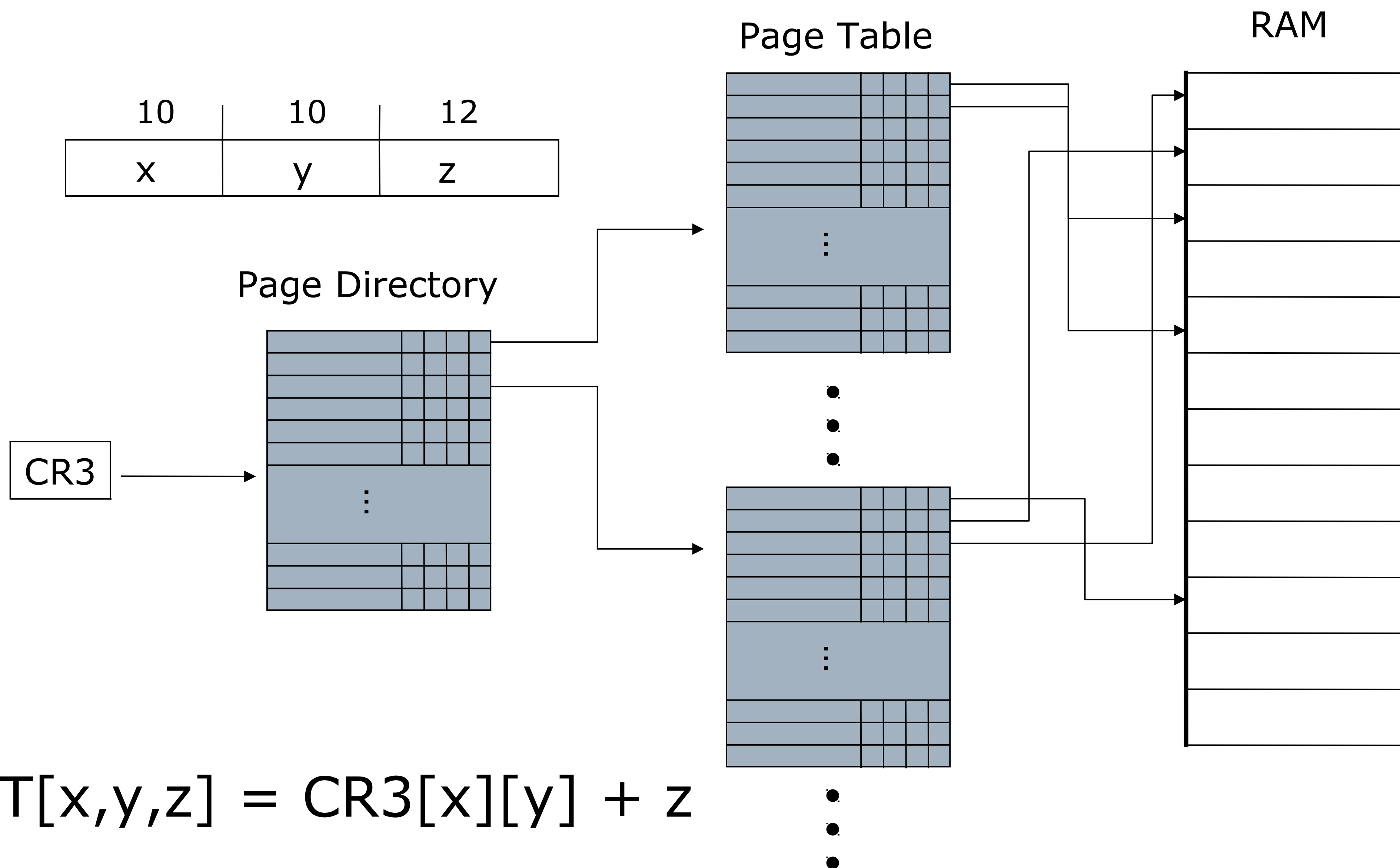
Minimizing Space Overhead

- Most of a flat table would be unused



PFN	valid	prot	present	dirty
9	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

Solution: 2-level Page Tables



Let's Get Some Numbers

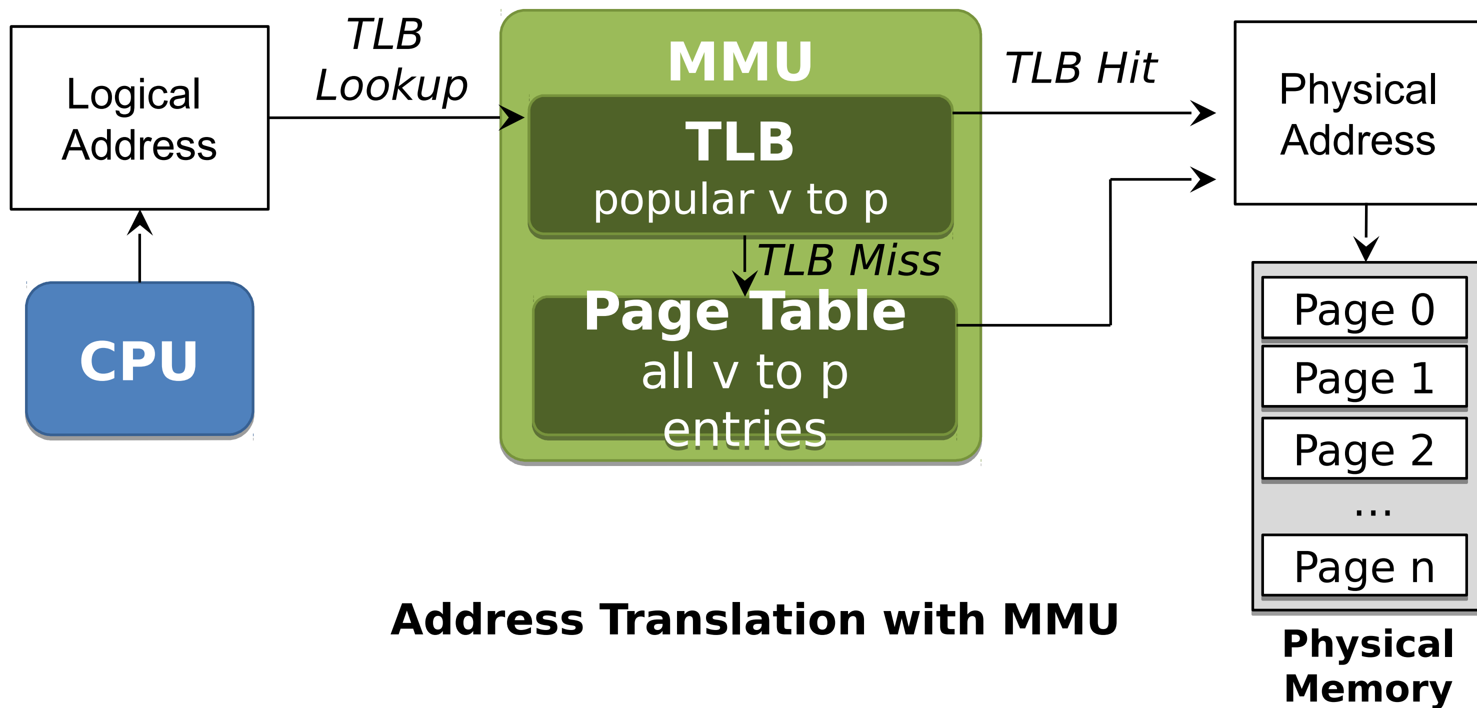
- 10 + 10 + 12 bits = 32
  - 2^10 (= 1024) entries of 4 bytes = a 4096-byte page!
  - 2-level tables work well for 32-bit machines
- 64-bit machines use a 3-level table!
- We save space, but we could do the same with other associative structures, right?
- Need to jointly solve time overhead!

Minimizing Time Overhead

- Hardware to the rescue
  - Translation Lookaside Buffer (TLB)
    - a cache of recent page address lookups
      - exploits spatial, temporal locality, like all caches
    - a few tens of entries, fully associative
  - (optional) Hardware-traversal of page tables
    - when TLB lookup fails
    - “hardware-managed TLB” common on CISC machines
    - “software-managed TLB” on RISC machines
      - interrupt, handled by kernel

TLB Schematically

- A cache on the chip's memory-management unit (MMU)



Address Translation with MMU

What Happens to TLB When Switching Processes?

- Typically entries have “address space id” (ASID)
- LRU replacement otherwise
- Other options? Shared parts of address space?

Time for a New Problem: What If Physical Memory Gets Full?

- Idea: *replace* a page to make room
  - much like any other cache
  - disk used as larger memory
    - recall memory hierarchy picture
  - also called *eviction*
- *Policy* vs *mechanism* distinction
  - with *algorithm* in the middle

Mechanisms

- *Swap* space (file, partition)
- "present" bit in PTE
- *Page fault* when accessed page not present
- *Page replacement* algorithm
  - implements/approximates a *replacement policy*
- Terminology: *page in*/*page out*
  - in: just mmap swap file page!?
  - out: only if page has *dirty/modified* bit on
- Terminology: *paging*, *thrashing*

Replacement Policies: OPT

- Ideal replacement policy: replace page that will be accessed the furthest in the future
  - policy called OPT or MIN
  - optimal (provably)
  - but requires knowing the future

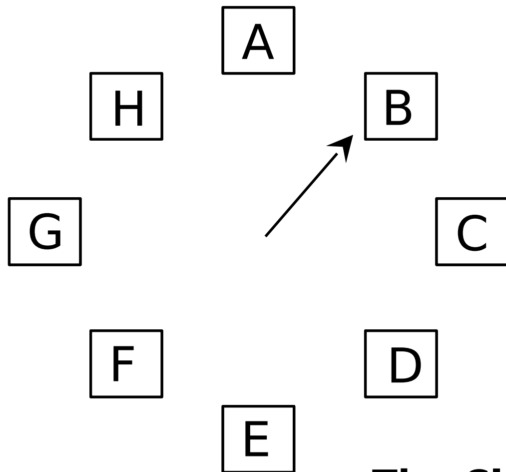
Other Replacement Policies

- FIFO
  - memories too large for it to work well, misses all behavior
- RANDOM
- LRU (Least Recently Used)
  - the golden standard of caching, uses history of accesses
- LFU (Least Frequently Used)
  - bad for programs, good for data streams

Approximating LRU

- The policy requires work per memory access
- Approximation algorithms minimize the overhead
  - Clock: uses recently-referenced bit, resets it
  - FIFO/LRU hybrid queue: FIFO for some pages, then move-to-front for those evicted from FIFO

Clock



Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the Use bit

Other Interesting Tidbits

- Virtual memory is only a part of memory!
  - explicit I/O and mmapped file contents treated the same
    - replaced in unified way, typically
    - remember LFU policy?
- Replacement only a part of the performance story
  - prefetching
  - clustering writes
    - more in file systems