Lecture 12: Virtual Memory Optimizations and Implementation

CS343 – Operating Systems

Branden Ghena – Fall 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Northwestern

Administrivia

- Driver Lab is due next week Thursday!
  - There’s quite a lot of work for this one
  - You need to write your own tests for the GPU
    - There are lots of edge cases where students commonly lose points
  - Get started ASAP

- Reminder: get help in office hours when you need it
  - 19 hours across Monday-Friday
  - Come ask questions about the class, labs, debugging, etc.
  - Chronically underutilized this quarter

Today’s Goals

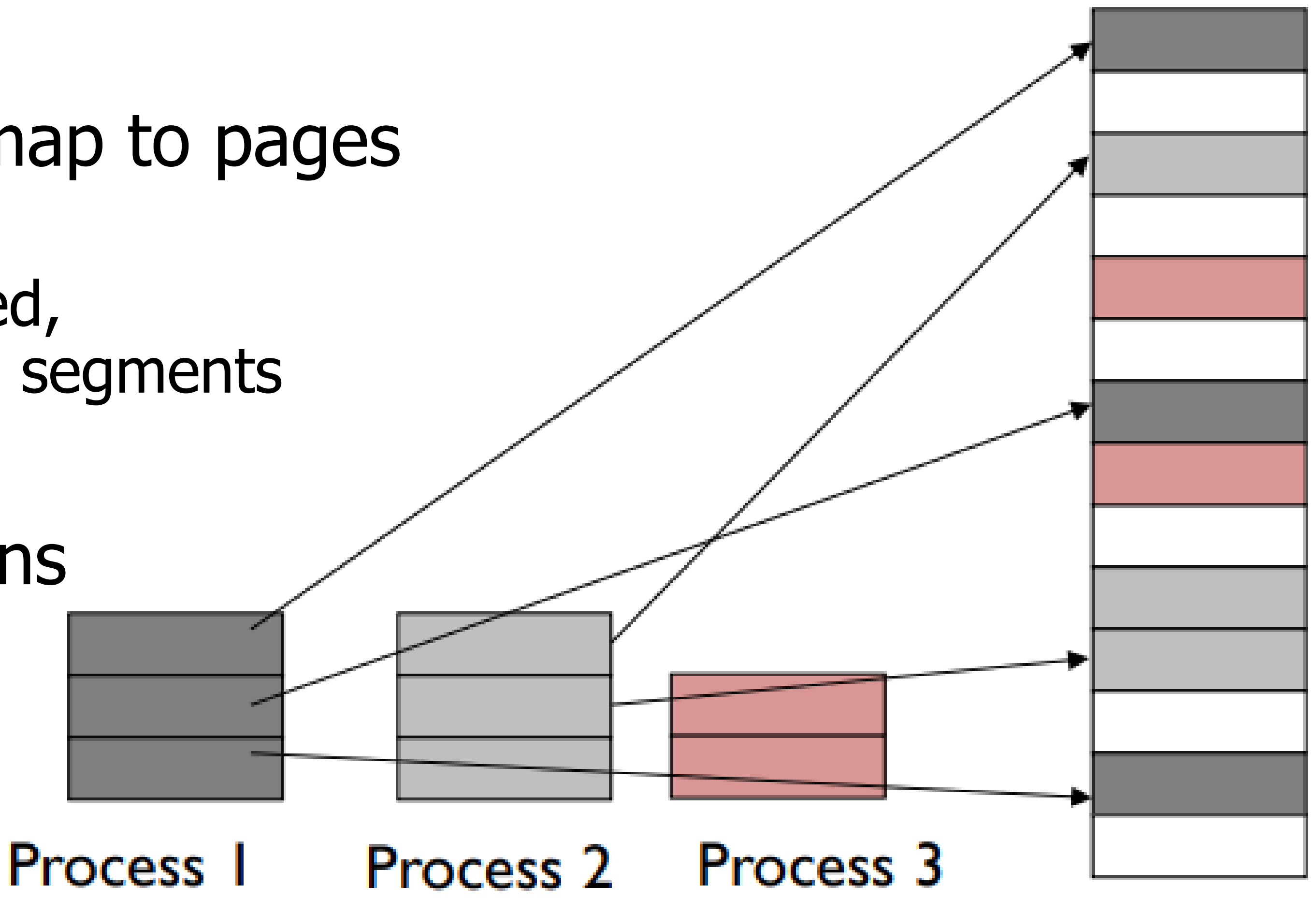
- Explore optimizations to memory paging.

- Insight into how virtual memory is used and what it looks like in today’s systems.

- Review of the memory hierarchy and how the OS interacts with each level.

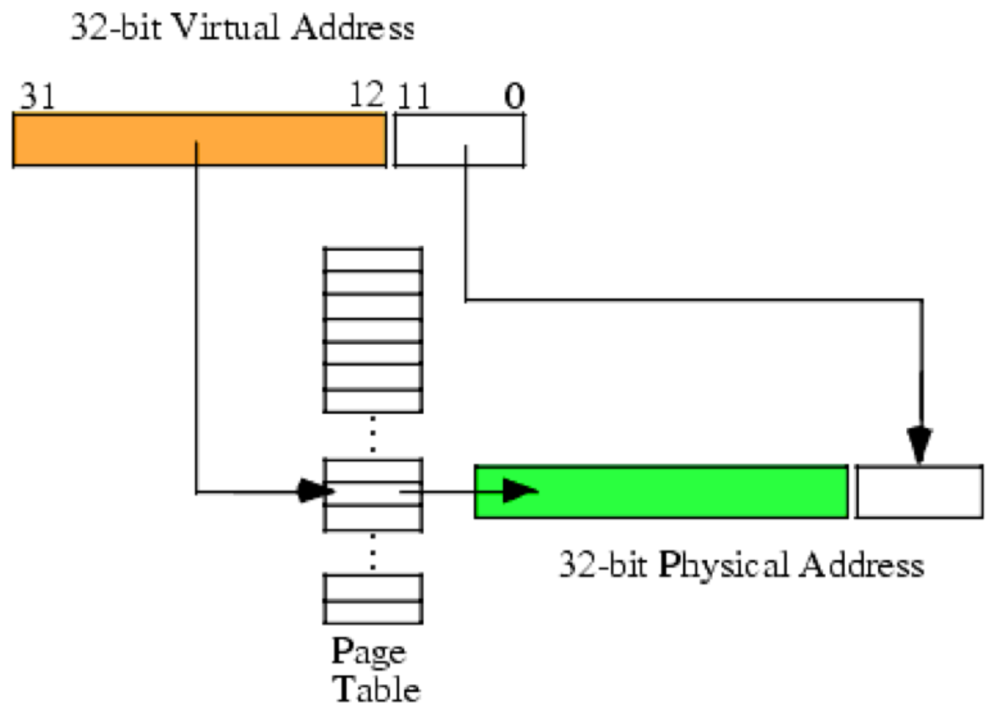
Memory paging

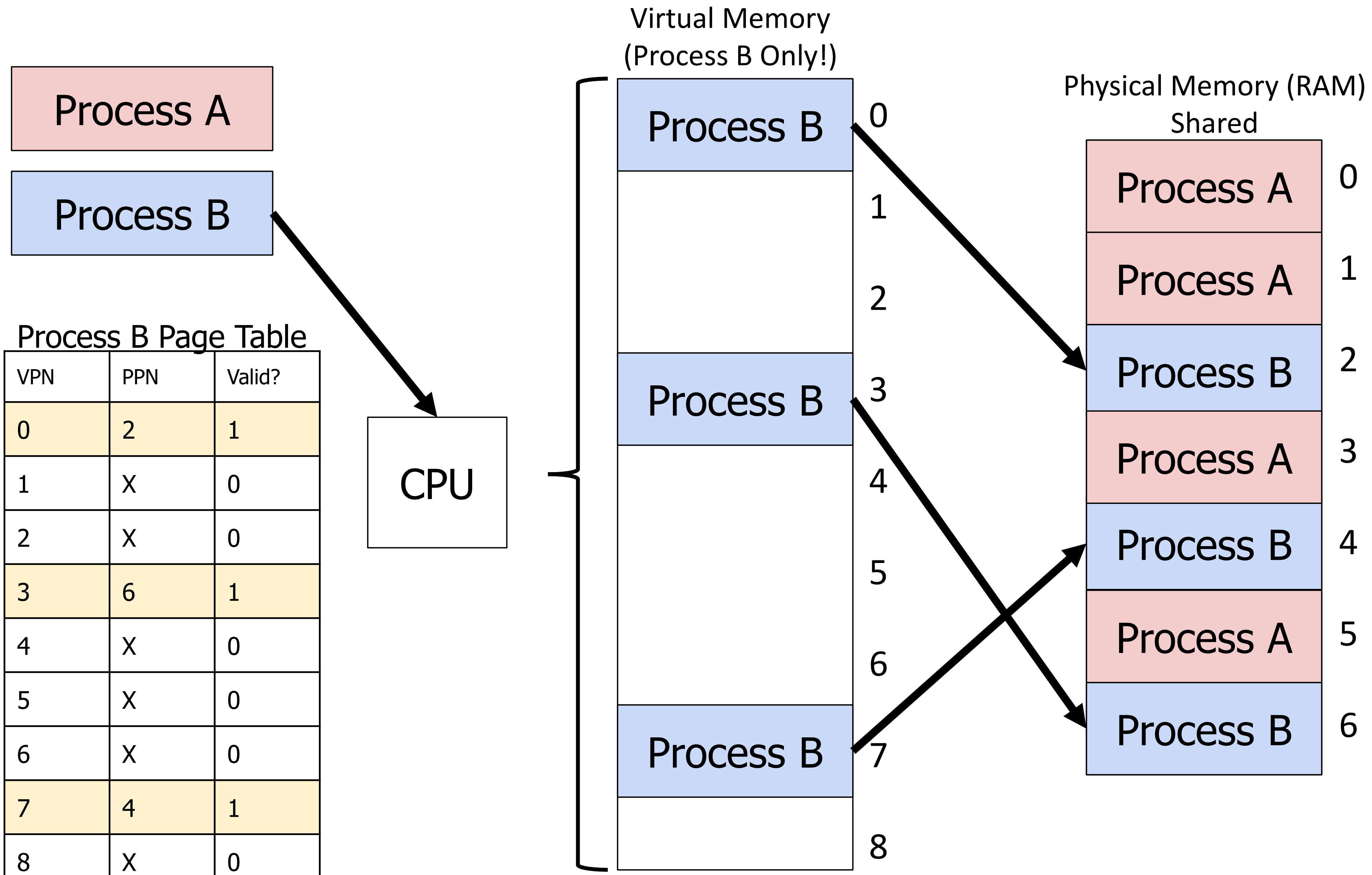
- Divide memory into small, **fixed-sized** pages
- Pages of virtual memory map to pages of physical memory
  - Like segments were mapped, but *many* more pages than segments
- Processes and their sections can be mapped to any place in memory



Page table translates virtual addresses to physical addresses

- Use topmost bits of virtual address to select page table entry
  - One page table entry per each virtual page
- Add address at page table entry to bottommost bits
  - Actually just concatenate the two
- Just like segment tables, there will be a different page table for each process





Process B Page Table

VPN	PPN	Valid?
0	2	1
1	X	0
2	X	0
3	6	1
4	X	0
5	X	0
6	X	0
7	4	1
8	X	0

Paging challenges

- Page tables are slow to access
  - Page tables need to be stored in memory due to size
  - MMU only holds the base address of the page table and reads from it
  - Two memory loads per load!!!
  - Going to have to fix this…

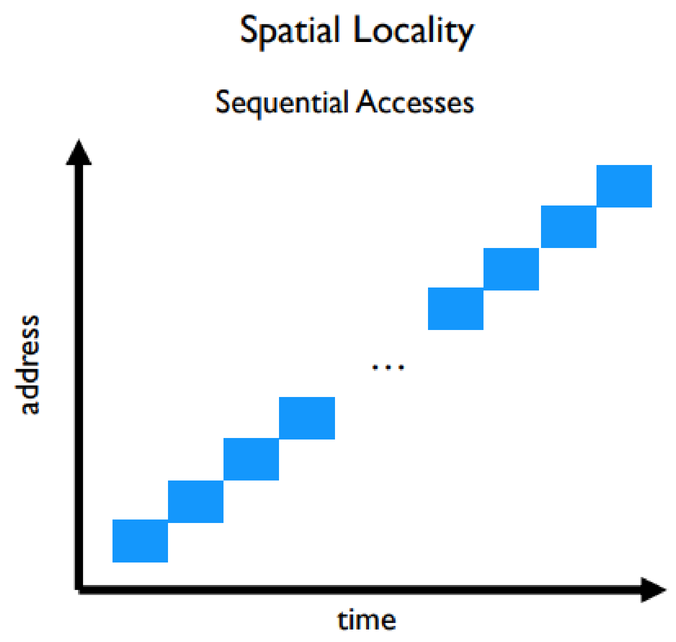
- Page tables require a lot of storage space
  - Mapping must exist for each virtual page, even if unused
  - Becomes a serious issue on 64-bit systems

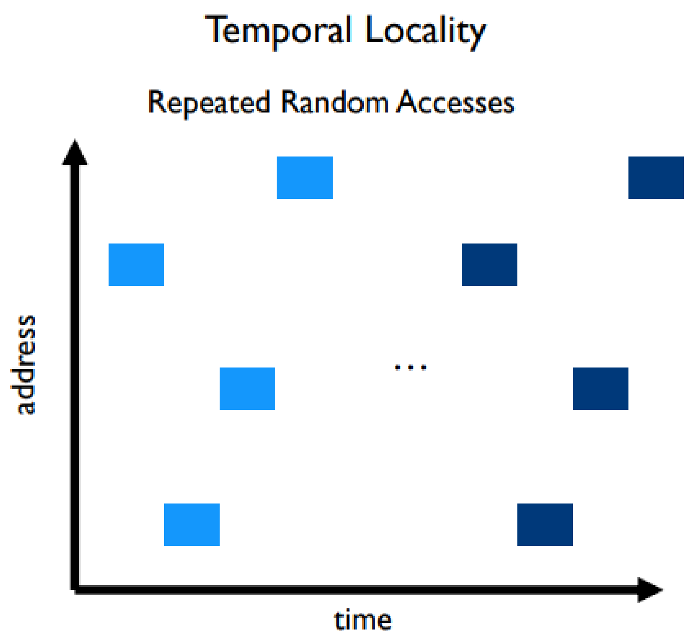
Outline

- **Paging improvements**
  - **Improving translation speed**
  - Improving table storage size
- OS Paging Implementation
- Memory Hierarchy

Caching can speed up page table access

- How do we make page table access faster?
  - How do we make memory access faster?
  - Cache it!
- Code and Stack have very high spatial locality

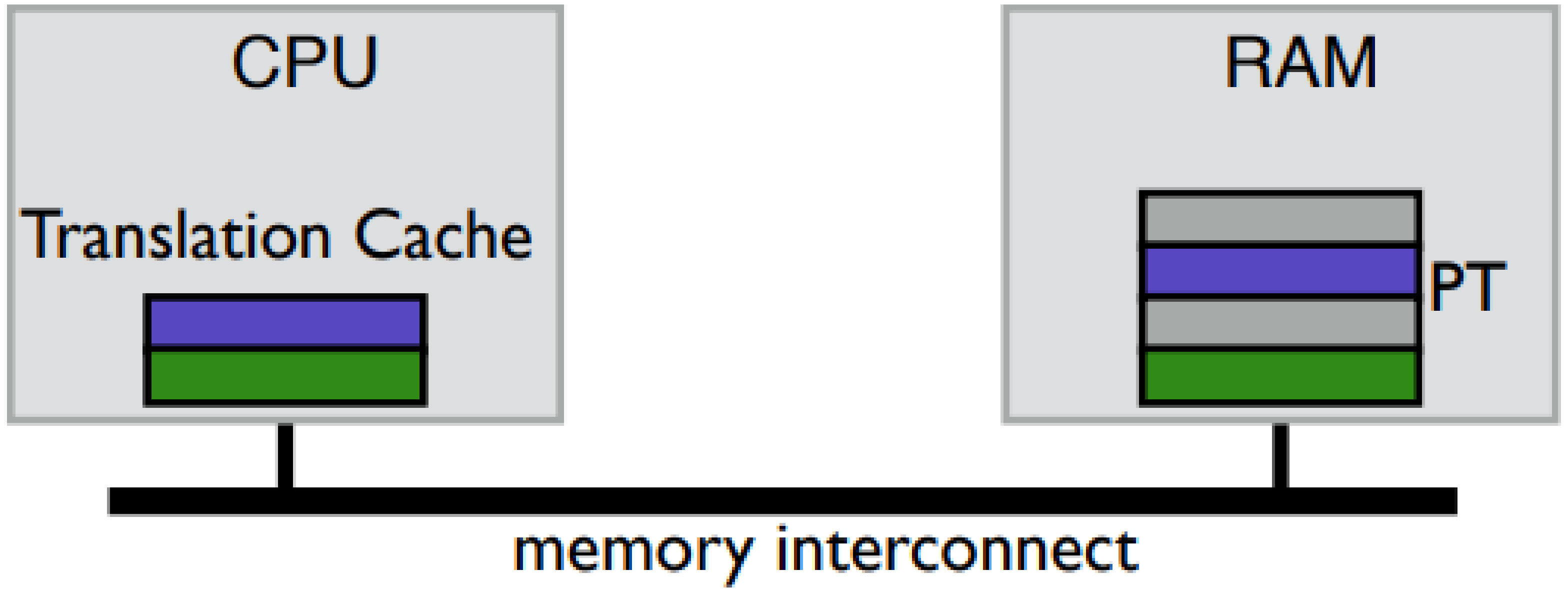




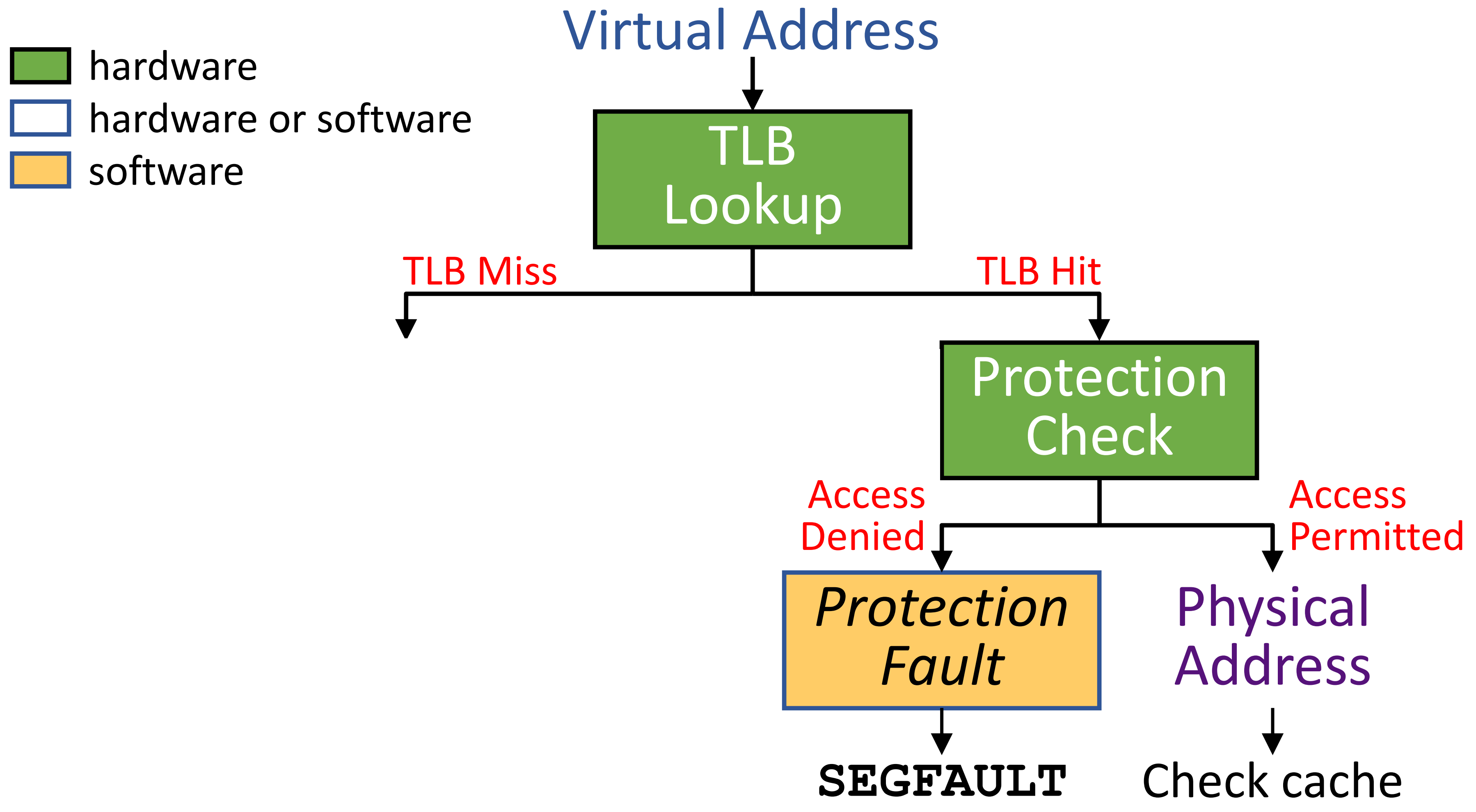
TLB caches page table entries

- Translation Lookaside Buffer
  - Fully-associative cache (only compulsory misses)
  - Holds a subset of the page table (VPN->PPN mapping and permissions)

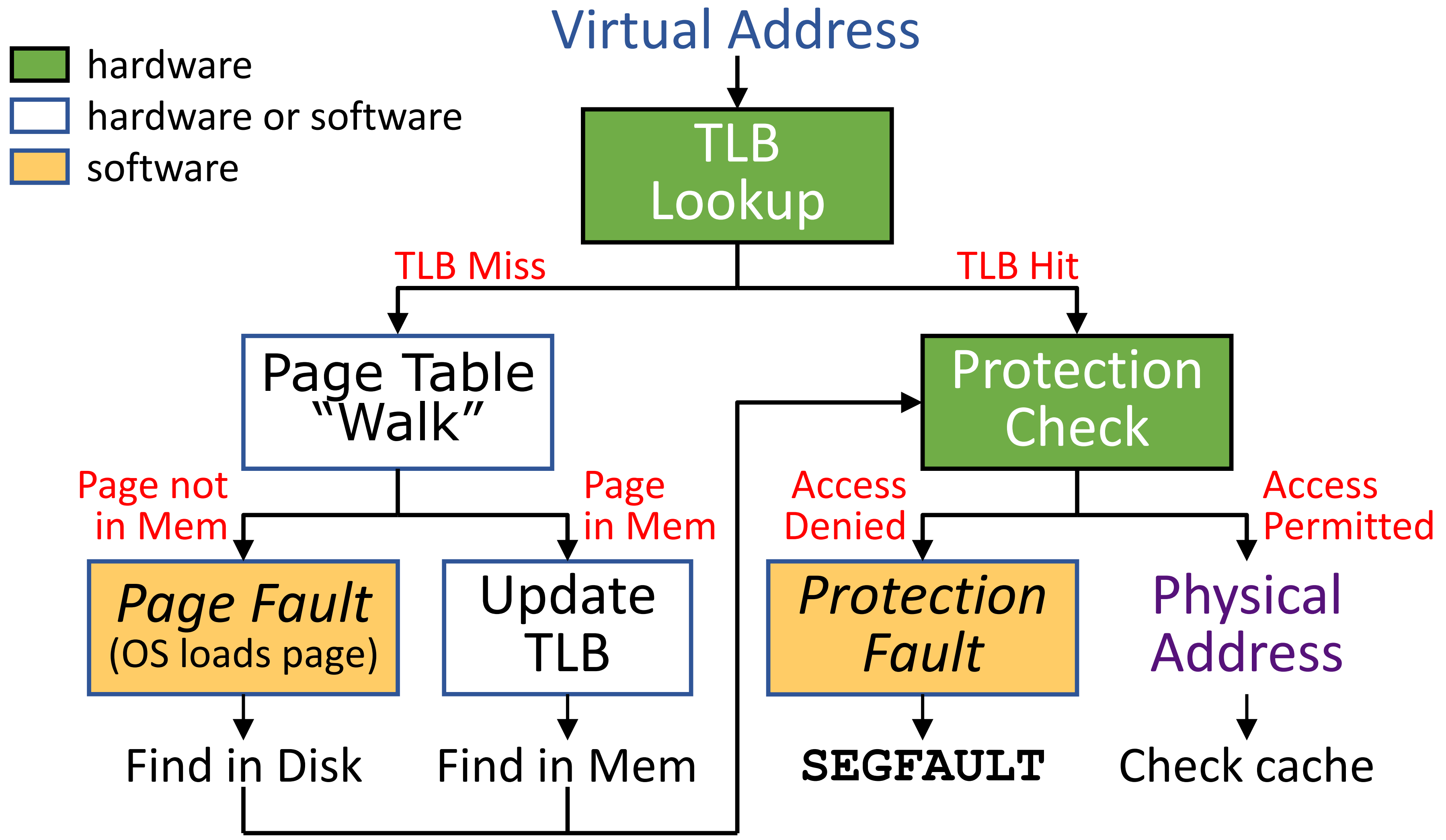
- On a TLB miss, go check the real page table (done in hardware)



Address translation with TLB



Address translation with TLB



Context switches with a TLB

- A process must only access its own page table entries in the TLB!
    - Otherwise, the mapping is wrong, and it accesses another process…
    - OS needs to manage the TLB

- Option 1: Flush TLB on each context switch
    - Costly to lose recently cached translations

- Option 2: Track with process each entry corresponds to
    - x86-64 Process Context Identifiers (12-bit -> 4096 different processes)
        - Extra state for the OS to manage if it has more processes than that

Software controlled TLBs

- Some RISC CPUs have a software-managed TLB
  - TLB still used for translation, but a miss causes a fault for OS to handle
    - OS looks in page table for proper entry
    - OS evicts an existing entry from TLB
    - OS inserts correct entry into TLB

  - Special instruction allows OS to write to TLB

  - Hardware is simpler and OS has control over the TLB functionality
    - Can prefetch page table entries it thinks might be important
    - Can flush entries relevant to other processes

  - TLB misses take longer to complete, however

Break + Question

- How many TLB entries do you need?
  (i.e., how many page table entries do you need to cache)

Break + Question

- How many TLB entries do you need?
  (i.e., how many page table entries do you need to cache)

  - Remember: each page is 4096 bytes of memory, which is a quite a bit
  - Want at least
    - One page in the code section (maybe a couple here for libraries)
    - One page in the stack section
    - One page in the data section

  - Could get away with a pretty small TLB: 4-16 entries
  - BUT, we *really* want to avoid misses, so more would be great

Outline

- **Paging improvements**
  - Improving translation speed
  - **Improving table storage size**
- OS Paging Implementation
- Memory Hierarchy

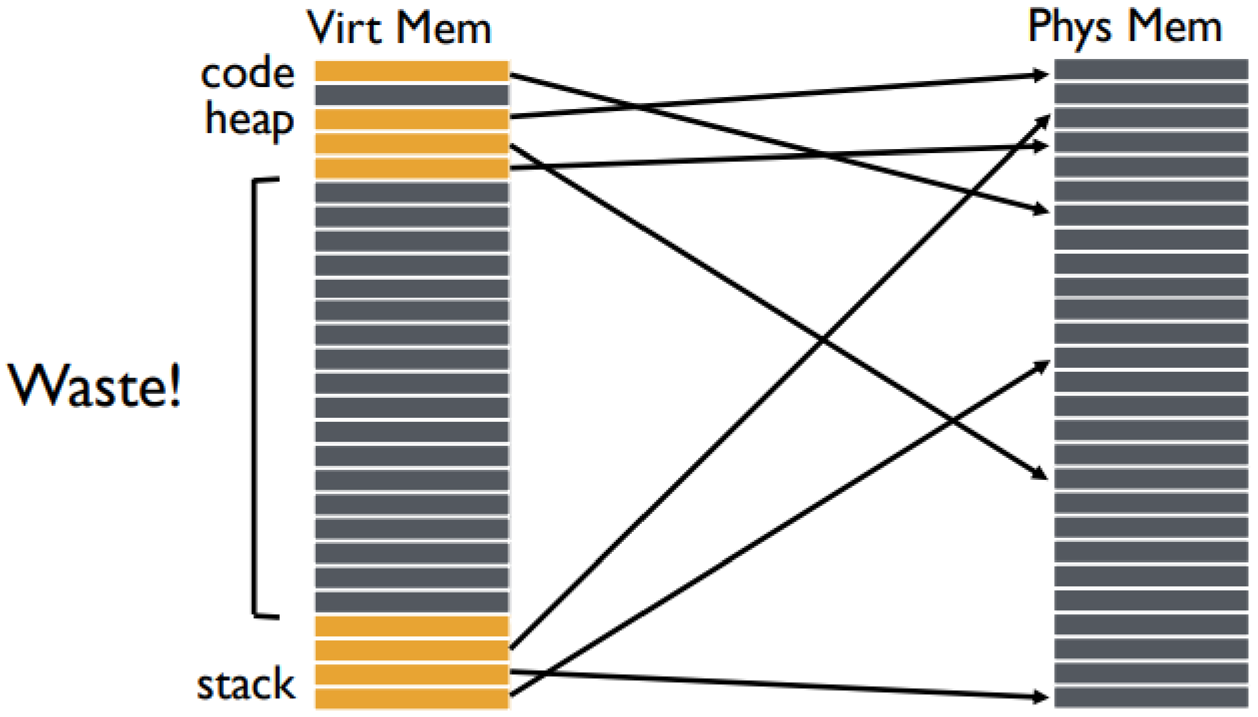
Paging disadvantages

1. Page tables are slow to access
    - Memory access for page table before any other memory access
    - TLB can speed this up considerably for common execution

2. Page tables require a lot of storage space
    - Mapping must exist for each virtual page, even if unused
    - Becomes a serious issue on 64-bit systems

Why do page tables take so much storage space?

- For every virtual page, there must exist an entry in the page table
    - Even though most virtual addresses aren’t used!



- 64-bit address space with 4 kB pages
    - 2^64/2^12 = 4,503,599,627,370,496 pages
    - At 8 bytes per page table entry: 36 Petabytes of storage space

Create multiple page tables, each with useful mappings only

- How do we eliminate extraneous entries from the page tables?

Virtual Page Number	**Valid?**	**Physical Page Number**
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
  (call them "page table entry pages"?)

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
- Only keep groups that have valid mappings in them

Virtual Page Number	**Valid?**	**Physical Page Number**
0	1	2
1	1	3
4	0	
5	1	7

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3

Virtual Page Number	Valid?	Physical Page Number
4	0	
5	1	7

Create multiple page tables, each with useful mappings only

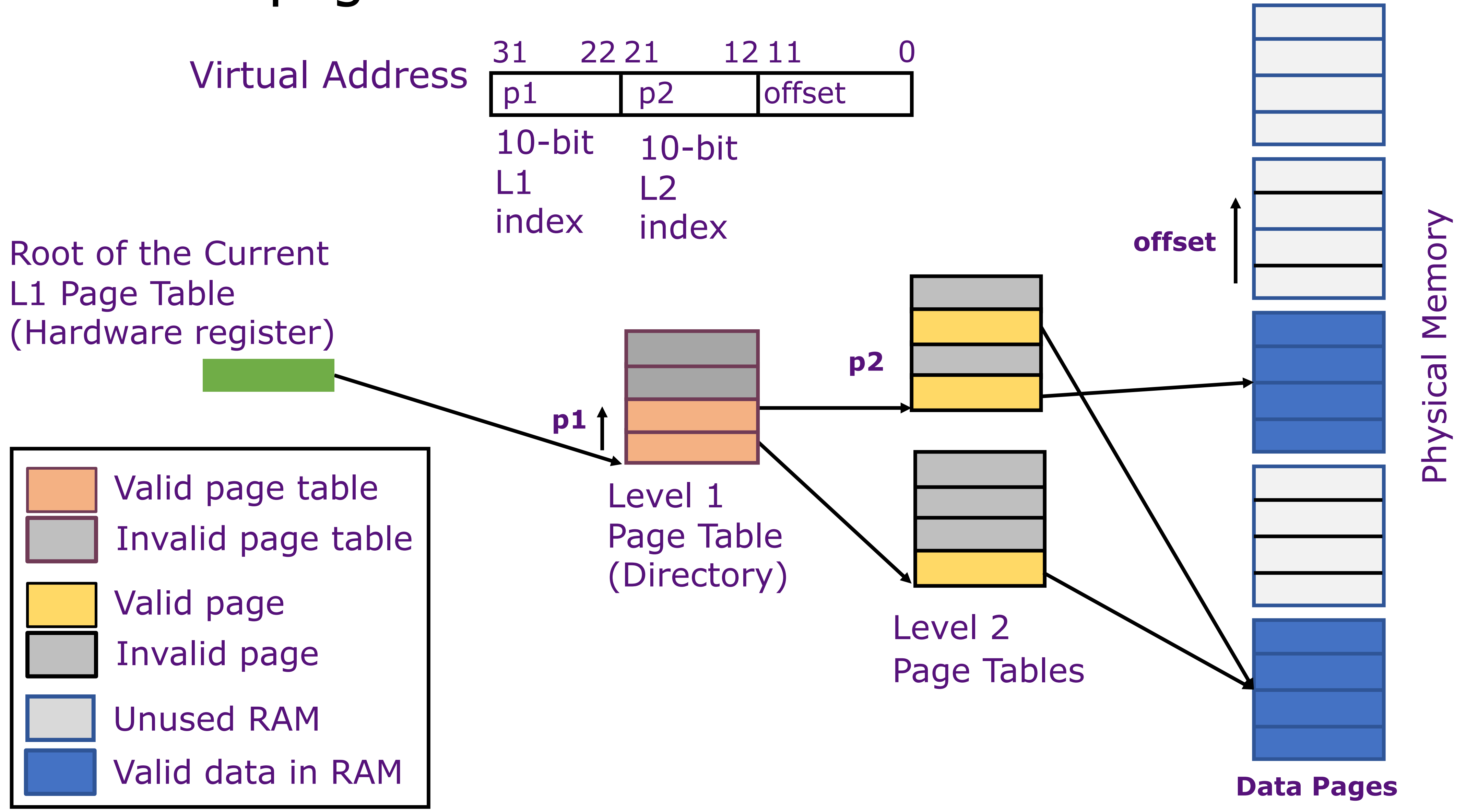
- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables
- Create a directory of page tables to collect existing page tables

Virtual Page Number	**Valid?**	**Physical Page Number**
0	1	2
1	1	3

Virtual Page Number	**Valid?**	**Physical Page Number**
4	0	
5	1	7

Virtual Page Number Range	**Valid?**	**Page Table Address**
0-1	1	
2-3	0	
4-5	1	
6-7	0	

Multilevel page tables



Multilevel page table logistics

- Virtual address is broken down into three or more parts
  - Highest bits index into highest-level page table
- A missing entry at any level triggers a page fault

- Size of tables in memory proportional to number of pages of virtual memory used
  - Small processes can have proportionally small page tables

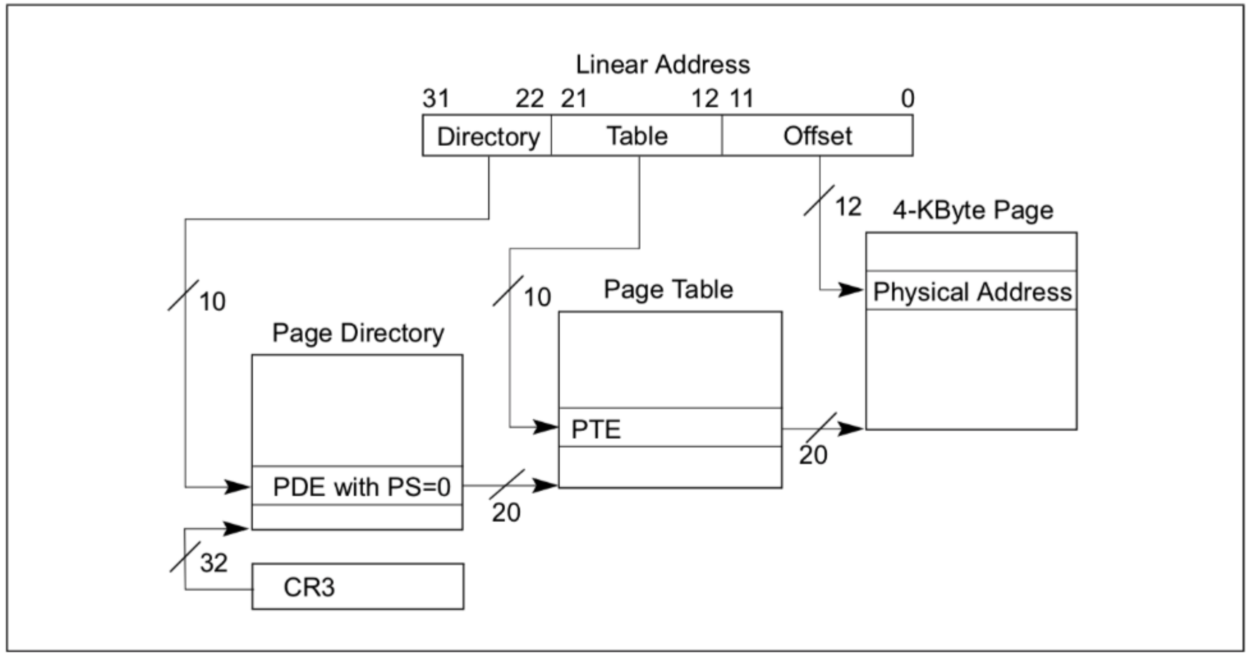
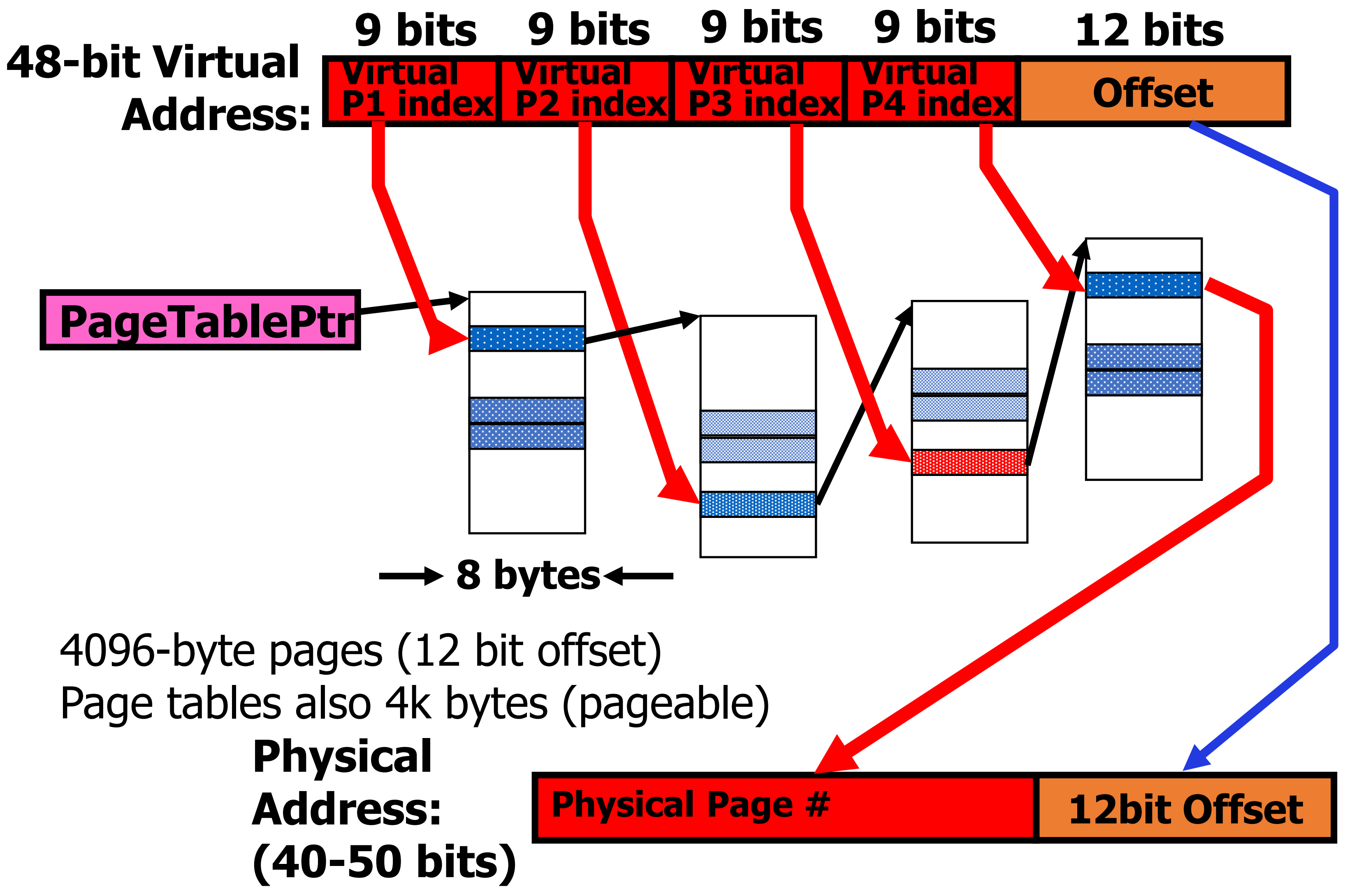


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Multilevel page tables can keep nesting

- Even page table directory is often sparse, so break it up too
- x86-64
  - Four levels of page table
  - 48-bit addresses (256 TB RAM ought to be enough for everyone right?)



Intel Ice Lake (2019): 5 layers!!

- Allows for 57-bit virtual memory space
  - 128 PB of virtual address space

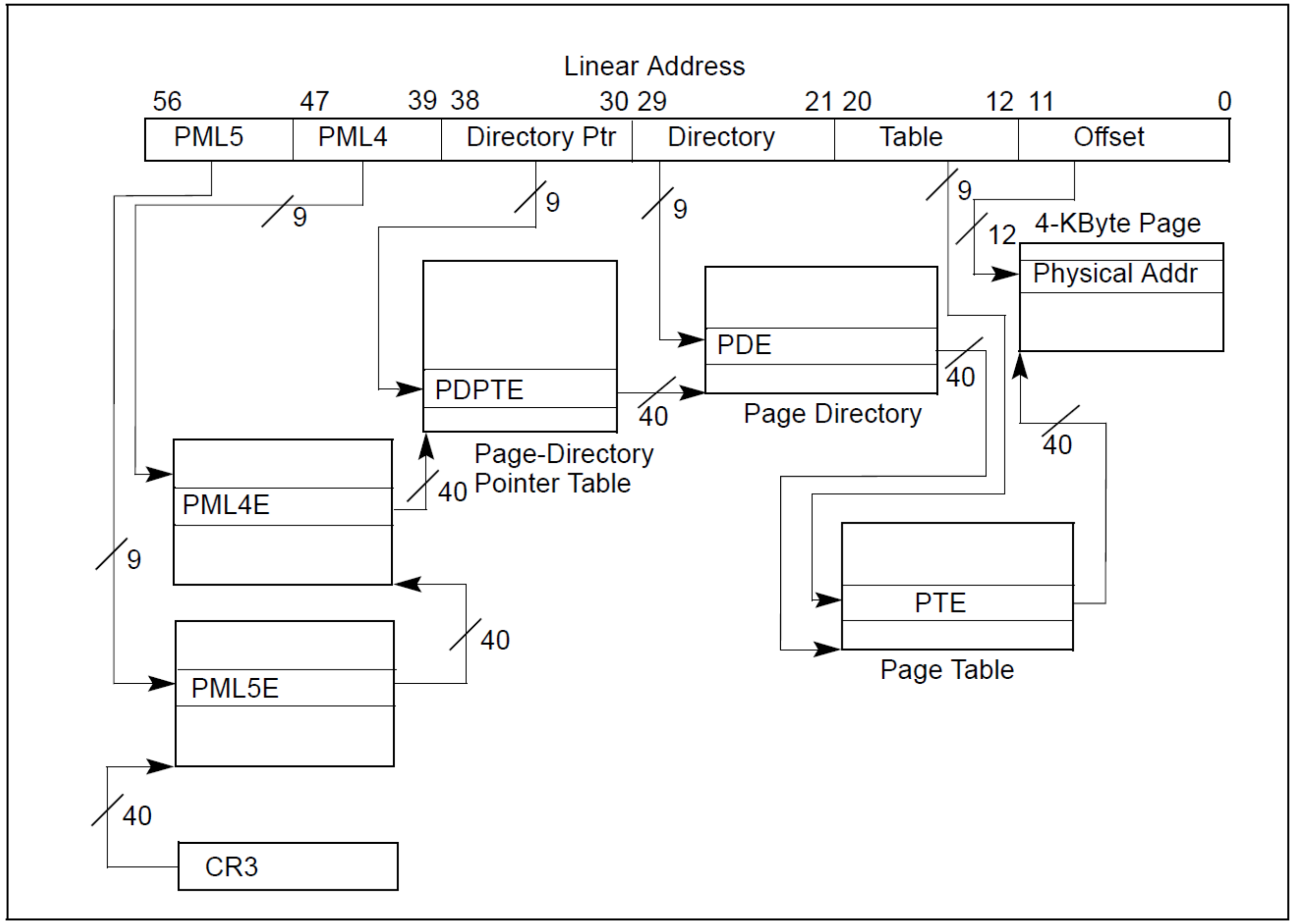


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

- How many memory loads per read are there now?

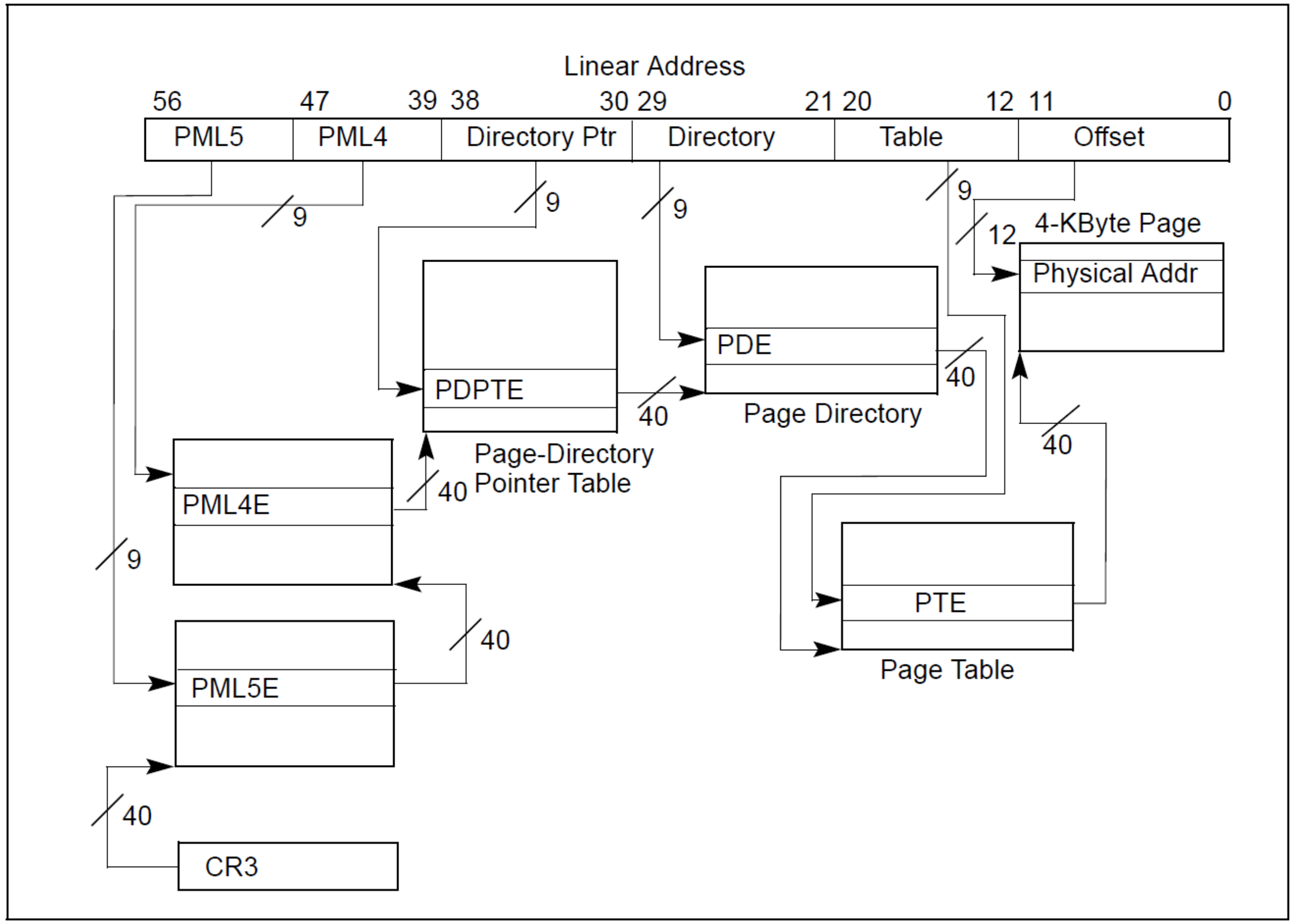


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

- How many memory loads per read are there now?
  - 6
  - As in each memory access takes six times as long

- TLB is ***extremely*** important

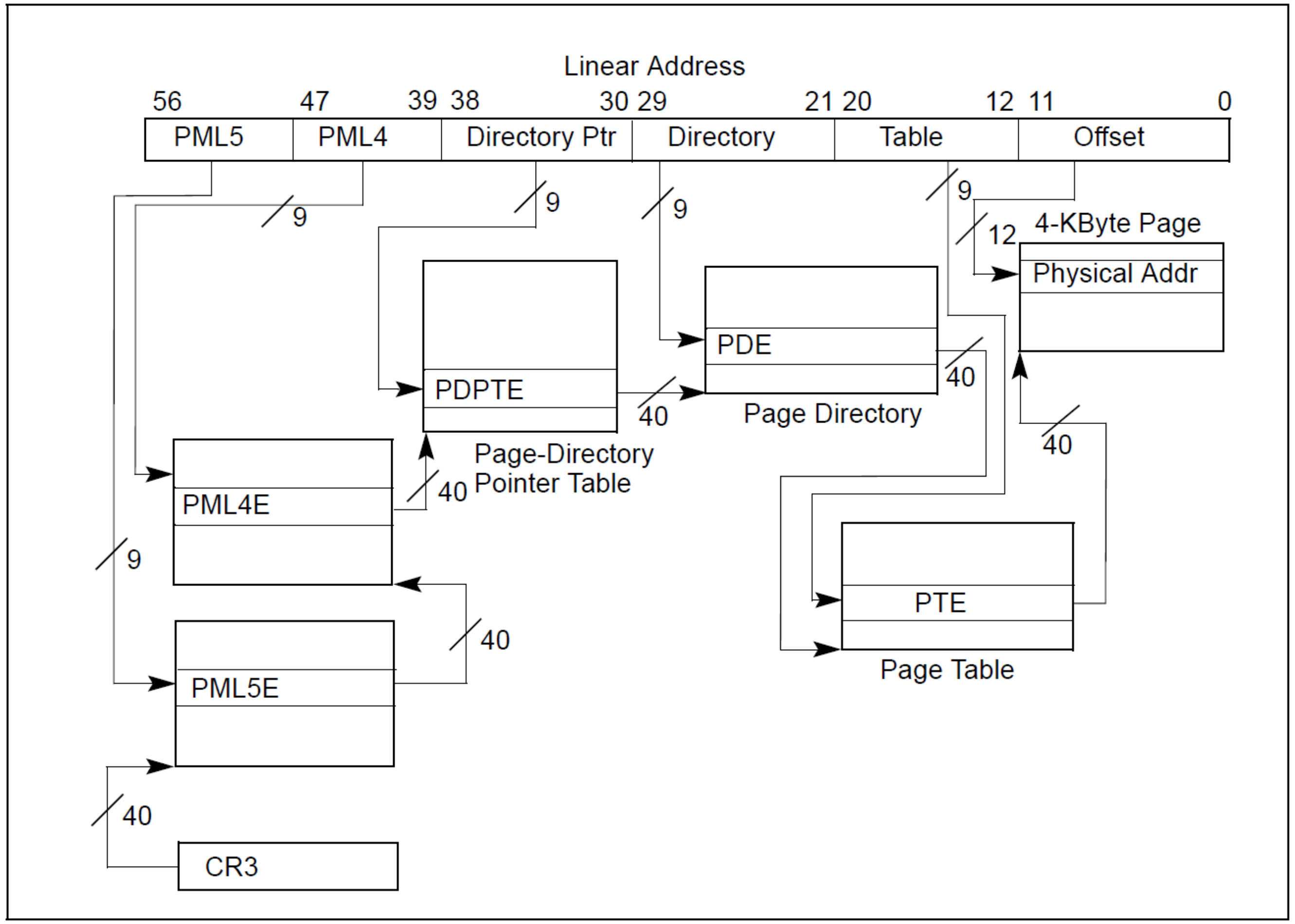


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Additional optimization: large pages

- Always using large pages results in wasted memory
    - Example: 1 MB page where only 1 KB is used

- Always using small pages results in unnecessary page table entries
    - Example: 250 entries in a row to represent 1 MB of memory

- Can we mix in larger pages opportunistically?
    - Small pages normally
    - Large pages occasionally
    - Huge pages rarely

x86-64 allows multiple-sized pages: 4 KB

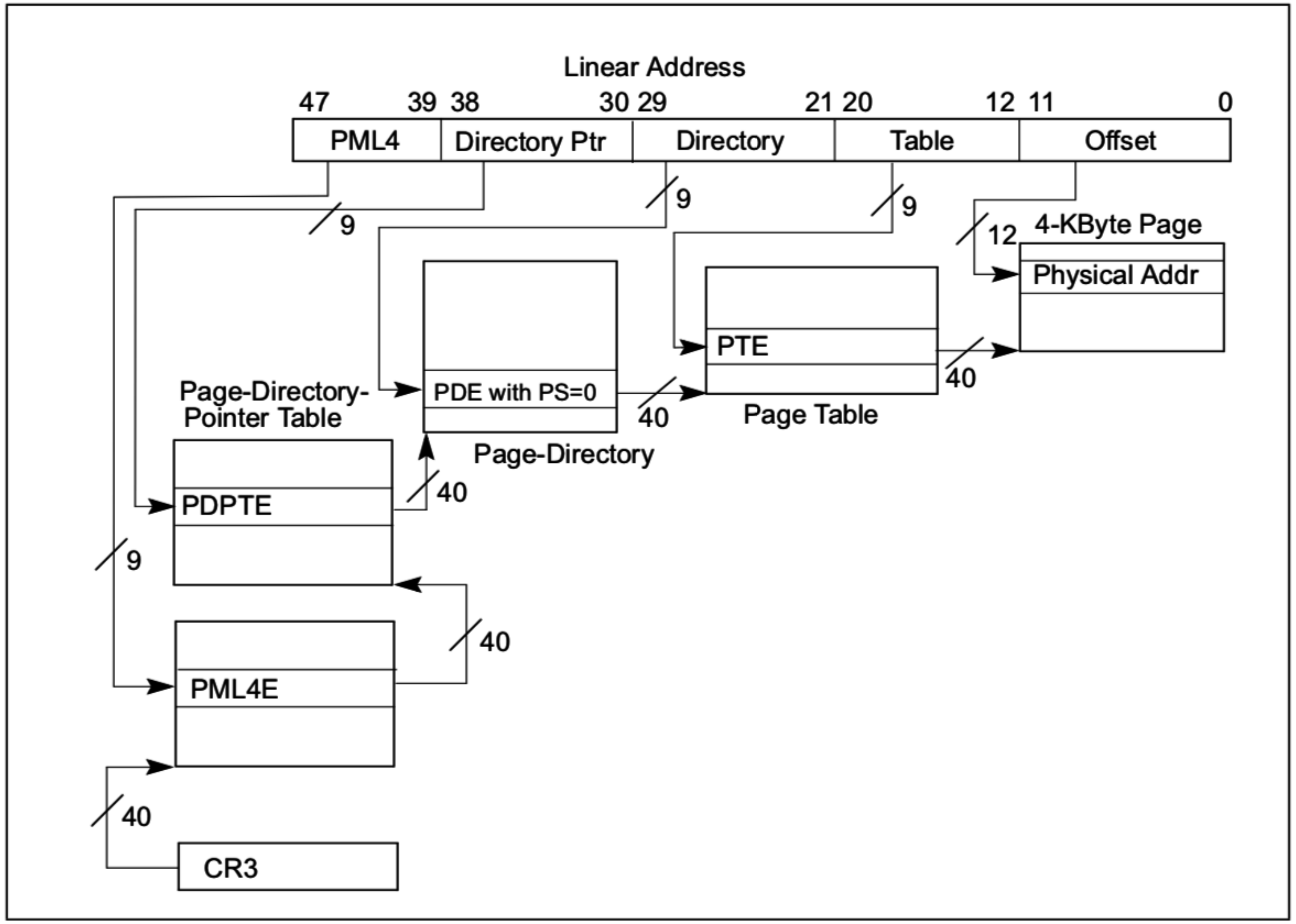


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- Normal x86-64 paging

x86-64 allows multiple-sized pages: 2 MB

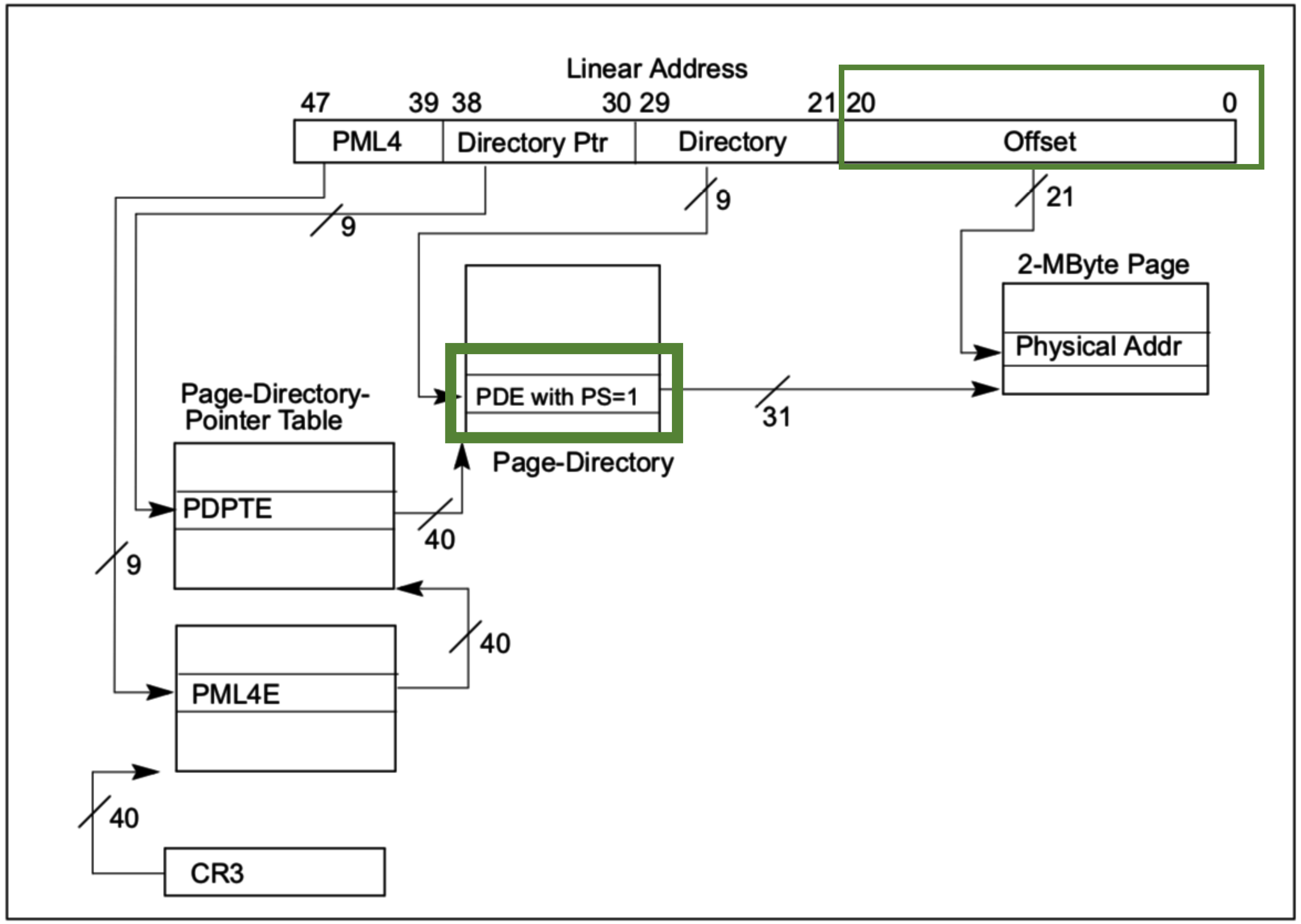


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

- Page Size bit skips next table and goes straight to 2 MB page in memory
- Remaining address bits are used as offset into larger page

x86-64 allows multiple-sized pages: 1 GB

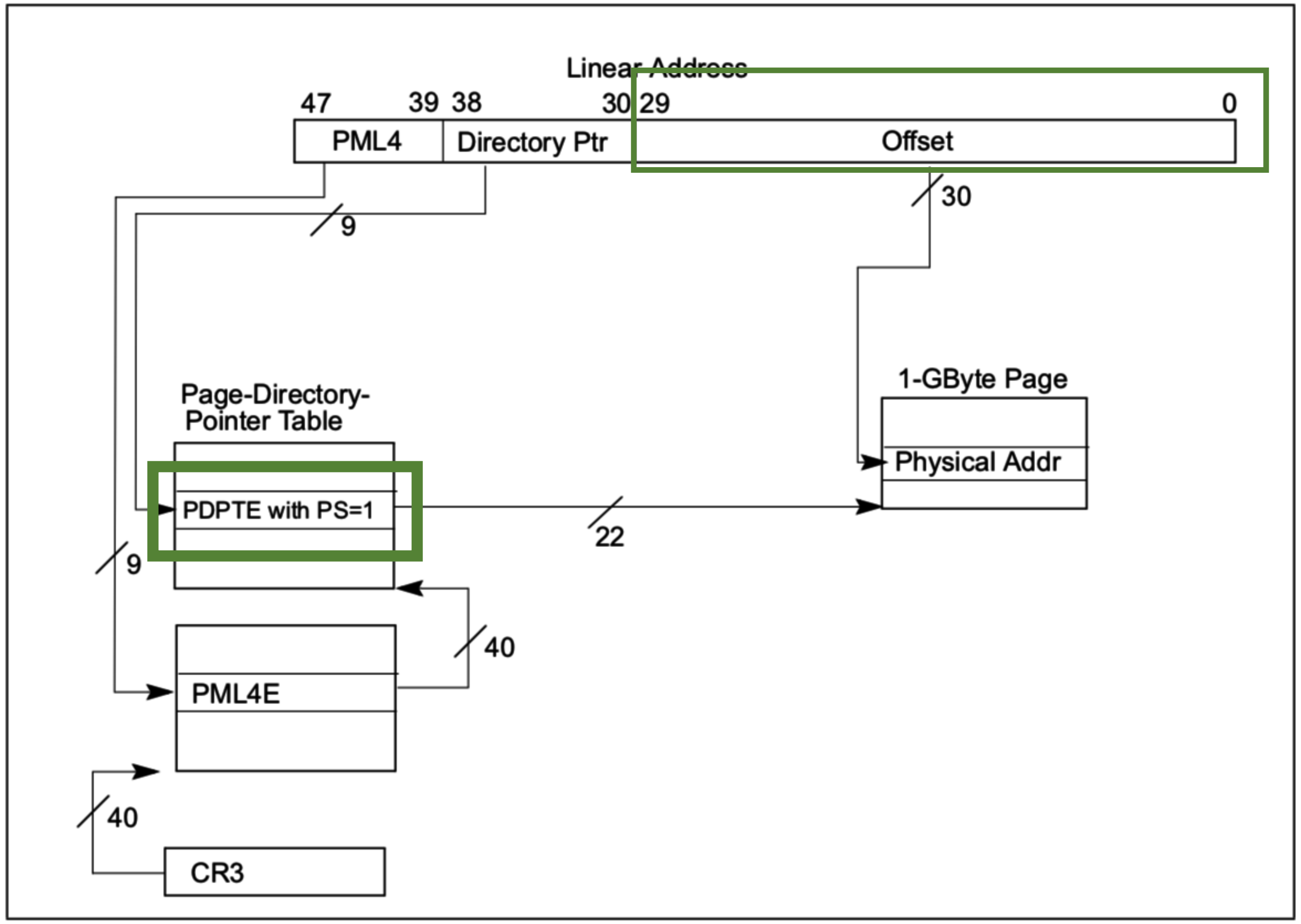


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Can also skip straight to 1 GB pages
- With a bit of extra hardware, TLB can hold large page entries
  - Occupies a single TLB entry for 1 GB of data (250000 normal entries)

Page Table data structure

- If hardware handles TLB misses
  - Need a regular structure it can “walk” to find page table entry
  - x86-64 requires the use of multilevel page tables with a specific layout
- If software handles TLB misses
  - OS can use whatever data structure it pleases
  - Example: inverted page tables
    - Only store entries for virtual pages with valid physical mappings
    - Use hash of VPN+PCID to find the entry you need

Real-world example TLB size

- My office processor: i5-8260U (8th generation: Coffee Lake)
  - Separate Data and Instruction TLBs. Two levels of TLB

- Data LTB
  - 4 – 1 GB pages
  - 32 – 2 MB pages
  - 64 – 4 kB pages

- Instruction TLB
  - 8 – 2 MB pages
  - 64 – 4 kB pages

- L2 unified TLB
  - 1536 – 4 kB / 2 MB pages

Break + Question

- If every page of virtual memory was used, would a multi-level page table take more or less space than a “flat” page table?

- How often is every page of virtual memory used?

Break + Question

- If every page of virtual memory was used, would a multi-level page table take more or less space than a “flat” page table?
  - More! Still need an entry for every “used” page
  - Now would have to add tree structure as well

- How often is every page of virtual memory used?
  - Never! That would be 18 exabytes of storage in one process

  - For reference: ~150,000 Exabytes is all of human digital storage (2024)

https://www.statista.com/statistics/871513/worldwide-data-created/

Multi-level Virtual Memory Example

Virtual Address breakdown (48 bits)

9 bits	9 bits	9 bits	9 bits	12 bits
L4 Index	L3 Index	L2 Index	L1 Index	Offset

- Given an address: 0xBC00CD00DE00 (48-bit)
  - Determine the index for each level of the cache

Multi-level Virtual Memory Example

Virtual Address breakdown (48 bits)

9 bits	9 bits	9 bits	9 bits	12 bits
L4 Index	L3 Index	L2 Index	L1 Index	Offset

- Given an address: 0xBC00CD00DE00 (48-bit)
  - Determine the index for each level of the cache

- Step 1: convert to binary, so we can pull out each part
  - 0b101111000000000011001101000000001101111000000000
  - 0b101111000000000011001101000000001101111000000000

Multi-level Virtual Memory Example

Virtual Address breakdown (48 bits)

9 bits	9 bits	9 bits	9 bits	12 bits
L4 Index	L3 Index	L2 Index	L1 Index	Offset

- Given an address: 0xBC00CD00DE00 (48-bit)
- Step 1: convert to binary, so we can pull out each part
  - 0b101111000000000011001101000000001101111000000000
- Step 2: convert sections back to hex or decimal (we'll do hex)
  - L4: 0x178 L3: 0x003 L2: 0x068 L1: 0x00D Offset: 0xE00

Outline

- Paging improvements
  - Improving translation speed
  - Improving table storage size
- **OS Paging Implementation**
- Memory Hierarchy

OS tracks regions rather than pages

- A **Region** is a collection of one or more pages for a process
  - An Address Space is a collection of regions for a process

- The OS will keep a data structure of regions for each process
  - Includes starting page/address and size
  - Protection fields
  - Additional bookkeeping information
    - Is it a kernel region or an application?
    - Is it a “pinned” region, i.e. a region we should never remove?
    - Is the region in RAM or on disk?
    - Has the region been modified?

Operations on regions

- Add
  - Create a new region for an address space
  - Accesses to virtual addresses in that range should now succeed
- Remove
  - Remove the region entirely from an address space
  - Accesses to virtual addresses in that range should now fault
- Move
  - Change physical addresses associated with the region
- Protect
  - Change protection status of region
  - Could change from read-only to read-and-write

Considerations when adding new regions

- The new region goes in the OS data structure immediately
- However, we don’t necessarily need to allocate space in RAM immediately or update the Page Table
  - Those actions are a lot of work
  - But maybe the process is never going to actually use most pages

Page faults enable lazy allocation and lazy loading

- Paging is not just translation and error checking
  - Paging provides an opportunity to be lazy about loading requested data

- Trick: don’t load data upfront, do it later when it’s first needed!
  - This is an important performance optimization,
    reducing program start time

Lazy loading in practice

- If a process requests a huge chunk of memory, maybe it will not use all that memory immediately (or ever!).
  - Programmers and compilers are sometimes ***greedy*** in their requests
  - We can *virtually* allocate memory, but mark most of the pages “not present”
  - Let the CPU raise an exception when the memory is really used
  - Then really allocate the demanded page

- Lazy allocation minimizes latency of fulfilling the request and it prevents OS from allocating memory that will not be used.

Lazy allocation makes Fork faster

- Recall that ***fork + exec*** is the only way to create a child process in unix

- Fork clones the entire process, including all virtual memory
  - This can be very slow and inefficient, especially if the memory will just be overwritten by a call to **exec**.

Lazy allocation via copy-on-write with Fork

- ***Copy on write*** is a performance optimization:
  - Don’t copy the parent’s pages, ***share*** them
    - Make the child process’ page table point to the parent’s physical pages
    - Mark all the pages as “read only” in the PTEs (temporarily)
  - If parent or child writes to a shared page, a page fault exception will occur
  - OS handles the page fault by:
    - Copying parent’s page to the child & marking both copies as writeable
    - When the faulting process is resumed, it retries the memory write.

Back to adding regions

- Adding a Lazy Region
  - Just add the region to the process data structure
  - Later, when an exception occurs you can load data update the Page Table as necessary

- Adding an Eager Region (a.k.a. a non-lazy region)
  - Do everything right away
    - Add to data structure, load into RAM, update Page Table

  - Example: a process's code might be eagerly loaded along with the first couple pages of the stack

Region “mappings”

- Another choice when creating a new region
  - Where does the memory come from??

- Three possibilities
  1. It’s a duplicate of somewhere else in RAM
     - For sharing memory, or for kernel memory at boot
  2. It’s fresh, zeroed-out memory (Anonymous Mapping)
     - Stack and heap memory
     - Should start as all zeroes for security
  3. It’s a copy of a file from disk (File-backed Mapping)
     - Program code and initial variable values
     - Likely read-only, and can be reloaded from disk whenever necessary

Lazy loading can apply to both of these

- Lazy loading works for large, file-backed code binaries
  - Delay loading a page of instructions from disk until it’s needed
  - Maybe won’t need some library files at all

- New anonymous pages must be all zeroes
  - Program does not necessarily expect the new memory to contain zeros,
  - But we clear the memory for security, so that other process’ data is not leaked.
  - OS can keep one read-only physical page filled with zeros and just give a reference to this at first.
    - After the first page fault (due to writing a read-only page), allocate a real page.

Removing, moving, and protecting pages

- Modify the region in the data structure

- Also update the Page Table immediately
  - Can’t do this lazily, as future accesses to pages MUST change

- But what if page table data is already in the TLB?!! Two options:
  1. Flush the entire TLB (remove all entries in it)
  2. Invalidate particular pages (removes individual entry from TLB if it exists)

  - For performance, which to do depends on how many pages you’re updating. Answer depends on the processor hardware, threshold is: 2-1000

OS management of processes with paging

- When loading a process
  - Add regions to data structure
    - For eager regions, also allocate RAM pages and update Page Table
    - For lazy regions (most), don't do anything now
  - Some regions might connect to shared libraries already in RAM

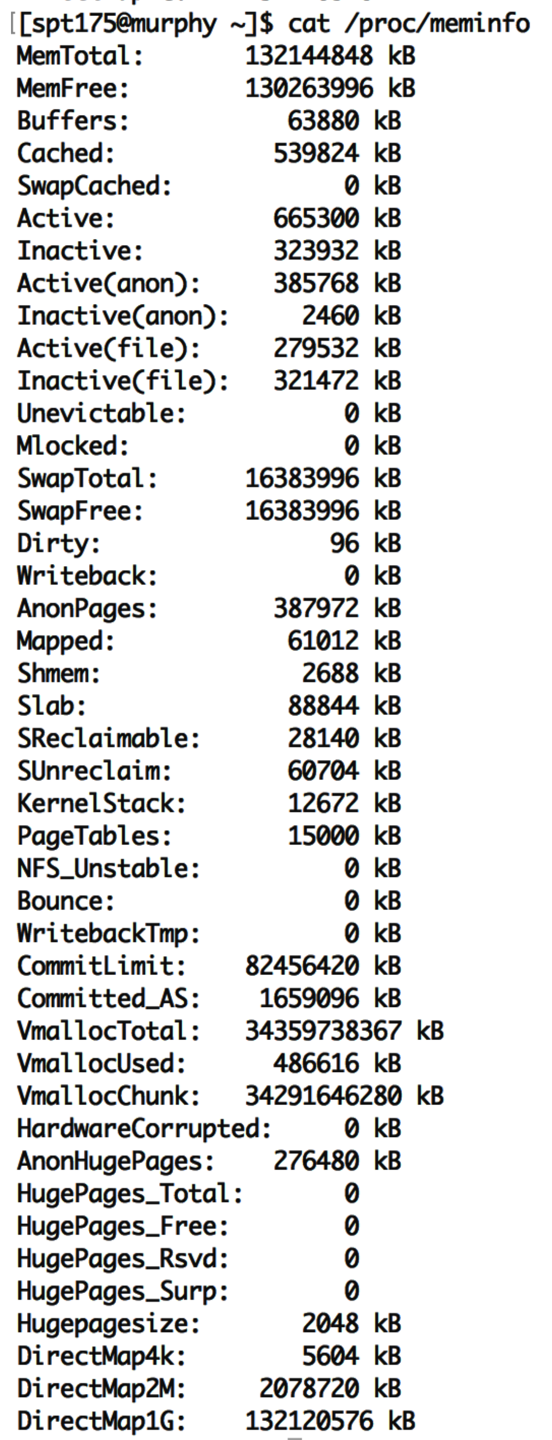
- When a context switch occurs
  - OS changes which page table is in use (%CR3 register in x86)

- When a fault occurs
  - OS handles it by checking the region data structure and the page table
    - Might be an invalid access (based on address or permissions)
    - Might be a page that's on disk or was lazily allocated

To see virtual memory info on Linux

- cat /proc/meminfo
- vmstat
- pmap
- top

- Try these commands yourself sometime!



```
[spt175@murphy ~]$ cat /proc/meminfo
MemTotal:       132144848 kB
MemFree:        130263996 kB
Buffers:           63880 kB
Cached:           539824 kB
SwapCached:            0 kB
Active:           665300 kB
Inactive:         323932 kB
Active(anon):     385768 kB
Inactive(anon):     2460 kB
Active(file):     279532 kB
Inactive(file):   321472 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:      16383996 kB
SwapFree:       16383996 kB
Dirty:                96 kB
Writeback:             0 kB
AnonPages:        387972 kB
Mapped:            61012 kB
Shmem:              2688 kB
Slab:              88844 kB
SReclaimable:      28140 kB
SUnreclaim:        60704 kB
KernelStack:       12672 kB
PageTables:        15000 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    82456420 kB
Committed_AS:    1659096 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      486616 kB
VmallocChunk:   34291646280 kB
HardwareCorrupted:     0 kB
AnonHugePages:    276480 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        5604 kB
DirectMap2M:     2078720 kB
DirectMap1G:    132120576 kB
```

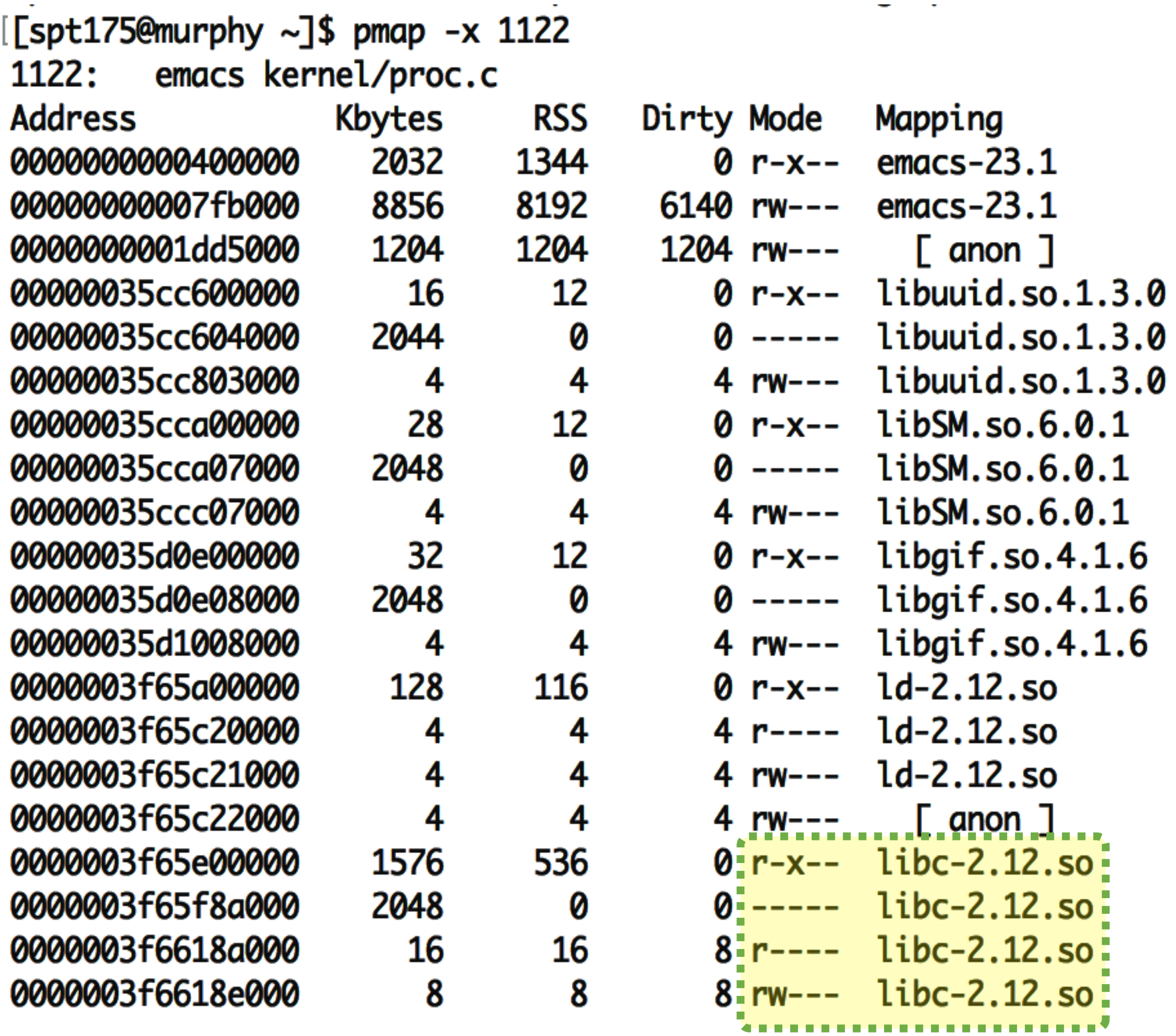
Virtual memory in practice

- On Linux, the `pmap` command shows a process' VM mapping.
- We see:
  - OS tracks which file code is loaded from, so it can be lazily loaded
  - The main process binary and libraries are ***lazy loaded***, not fully in memory
  - Libraries have read-only sections that can be shared with other processes
- `cat /proc/<pid>/smaps` shows even more detail

References:

- https://unix.stackexchange.com/a/116332
- https://www.akkadia.org/drepper/dsohowto.pdf

pmap on emacs



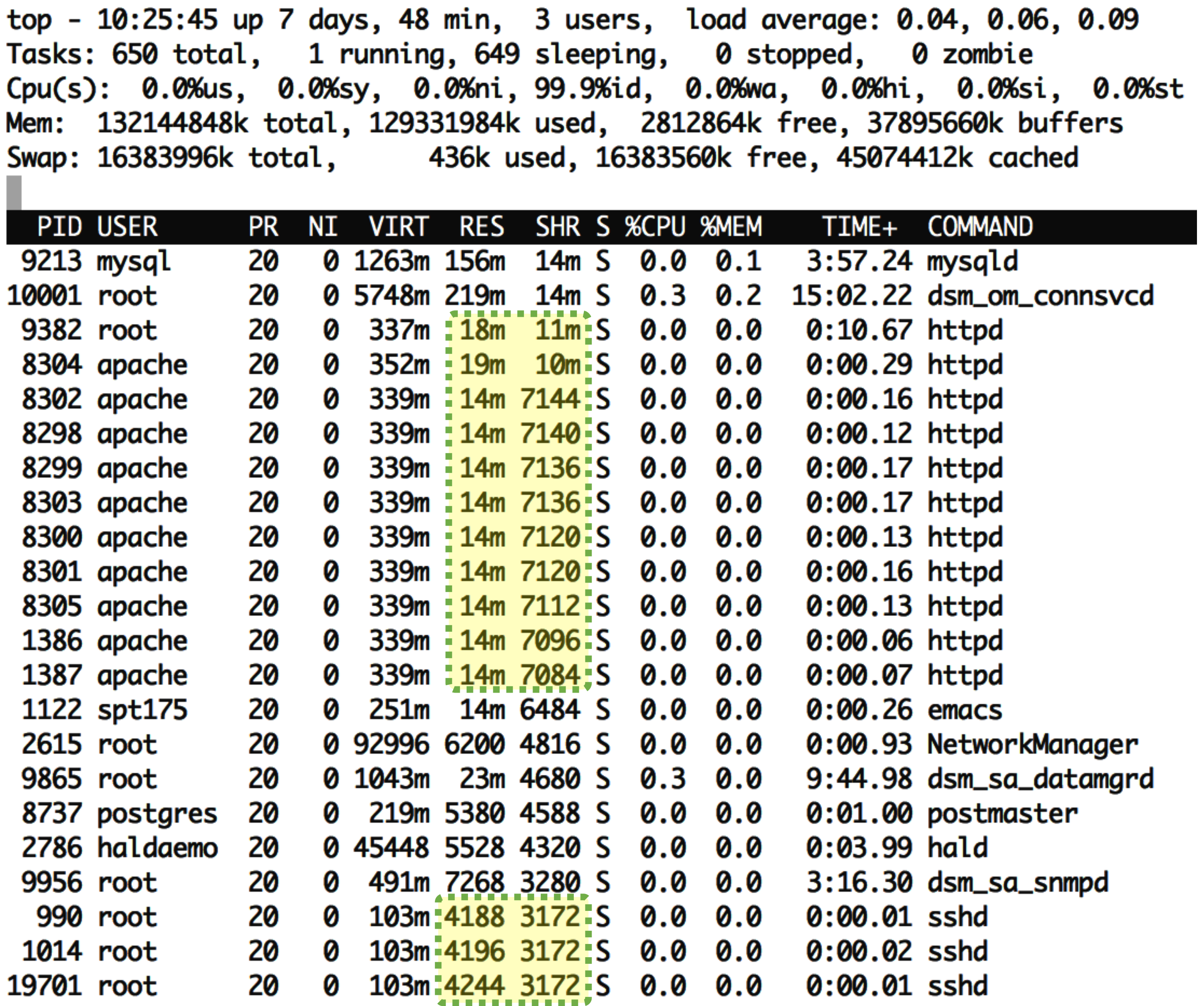
```
[spt175@murphy ~]$ pmap -x 1122
1122:   emacs kernel/proc.c
Address           Kbytes     RSS   Dirty Mode   Mapping
0000000000400000    2032    1344       0 r-x--  emacs-23.1
00000000007fb000    8856    8192    6140 rw---  emacs-23.1
0000000001dd5000    1204    1204    1204 rw---    [ anon ]
00000035cc600000      16      12       0 r-x--  libuuid.so.1.3.0
00000035cc604000    2044       0       0 -----  libuuid.so.1.3.0
00000035cc803000       4       4       4 rw---  libuuid.so.1.3.0
00000035cca00000      28      12       0 r-x--  libSM.so.6.0.1
00000035cca07000    2048       0       0 -----  libSM.so.6.0.1
00000035ccc07000       4       4       4 rw---  libSM.so.6.0.1
00000035d0e00000      32      12       0 r-x--  libgif.so.4.1.6
00000035d0e08000    2048       0       0 -----  libgif.so.4.1.6
00000035d1008000       4       4       4 rw---  libgif.so.4.1.6
0000003f65a00000     128     116       0 r-x--  ld-2.12.so
0000003f65c20000       4       4       4 r----  ld-2.12.so
0000003f65c21000       4       4       4 rw---  ld-2.12.so
0000003f65c22000       4       4       4 rw---    [ anon ]
0000003f65e00000    1576     536       0 r-x--  libc-2.12.so
0000003f65f8a000    2048       0       0 -----  libc-2.12.so
0000003f6618a000      16      16       8 r----  libc-2.12.so
0000003f6618e000       8       8       8 rw---  libc-2.12.so
```

- "Mapping" shows source of the section
  - "**anon**" are regular program data, requested by *sbrk* or *mmap*. (usually heap or stack)
- Each library has several sections:
  - "r-x--" for code
  - "r----" for constants

    $\left.\begin{array}{l}\\ \\ \end{array}\right\}$ *can be shared*
  - "rw---" for global data
  - "-----" for guard pages:

    (not mapped to anything, just reserved to generate page faults)
- **RSS** means resident in physical mem.
- **Dirty** pages have been written and therefore cannot be shared with others

top has a column showing shared memory



```
top - 10:25:45 up 7 days, 48 min,  3 users,  load average: 0.04, 0.06, 0.09
Tasks: 650 total,   1 running, 649 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132144848k total, 129331984k used,  2812864k free, 37895660k buffers
Swap: 16383996k total,      436k used, 16383560k free, 45074412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9213 mysql     20   0 1263m 156m  14m S  0.0  0.1   3:57.24 mysqld
10001 root      20   0 5748m 219m  14m S  0.3  0.2  15:02.22 dsm_om_connsvcd
 9382 root      20   0  337m  18m  11m S  0.0  0.0   0:10.67 httpd
 8304 apache    20   0  352m  19m  10m S  0.0  0.0   0:00.29 httpd
 8302 apache    20   0  339m  14m 7144 S  0.0  0.0   0:00.16 httpd
 8298 apache    20   0  339m  14m 7140 S  0.0  0.0   0:00.12 httpd
 8299 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8303 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8300 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.13 httpd
 8301 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.16 httpd
 8305 apache    20   0  339m  14m 7112 S  0.0  0.0   0:00.13 httpd
 1386 apache    20   0  339m  14m 7096 S  0.0  0.0   0:00.06 httpd
 1387 apache    20   0  339m  14m 7084 S  0.0  0.0   0:00.07 httpd
 1122 spt175    20   0  251m  14m 6484 S  0.0  0.0   0:00.26 emacs
 2615 root      20   0 92996 6200 4816 S  0.0  0.0   0:00.93 NetworkManager
 9865 root      20   0 1043m  23m 4680 S  0.3  0.0   9:44.98 dsm_sa_datamgrd
 8737 postgres  20   0  219m 5380 4588 S  0.0  0.0   0:01.00 postmaster
 2786 haldaemo  20   0 45448 5528 4320 S  0.0  0.0   0:03.99 hald
 9956 root      20   0  491m 7268 3280 S  0.0  0.0   3:16.30 dsm_sa_snmpd
  990 root      20   0  103m 4188 3172 S  0.0  0.0   0:00.01 sshd
 1014 root      20   0  103m 4196 3172 S  0.0  0.0   0:00.02 sshd
19701 root      20   0  103m 4244 3172 S  0.0  0.0   0:00.01 sshd
```

- The duplicate processes are using a lot of shared memory:
  - ~50% of resident memory for httpd is shared
    ~75% of resident memory for sshd is shared
- Even if there is just one instance of emacs running, it may share many libraries with other running programs.
- Total virtual memory is ~10x larger than resident memory
  - Processes only use a small fraction of their VM!
  - Due to sharing and lazy loading.

Process side: requesting memory from the OS – brk()

- System call to change data segment size (the program “break”)
  - Either set a new virtual address pointer for top of data segment
  - Or increment the size of the data segment by N bytes

- These are the old system calls to dynamically change program memory
  - How malloc creates space

- “sbrk() and brk() are considered legacy even by 1997 standards”
  - Removed from POSIX in 2001
  - Still exists in some form in lots of OSes (including Nautilus and Tock)

Process side: modern requesting memory from the OS – mmap()

- Map (or unmap) files or devices into memory

- Given a file, places the file in the process's virtual address space
    - Process can request an address to place it at, which OS *might* follow

- Given flag MAP_ANONYMOUS, creates empty memory
    - Initialized to zero and accessible from process
    - Malloc implementation uses this

- Many other options
    - Create huge page, create memory for a stack, shared memory

Break + Consideration

- Why use mmap() to put a file in your address space, when you could just read()/write() it instead?

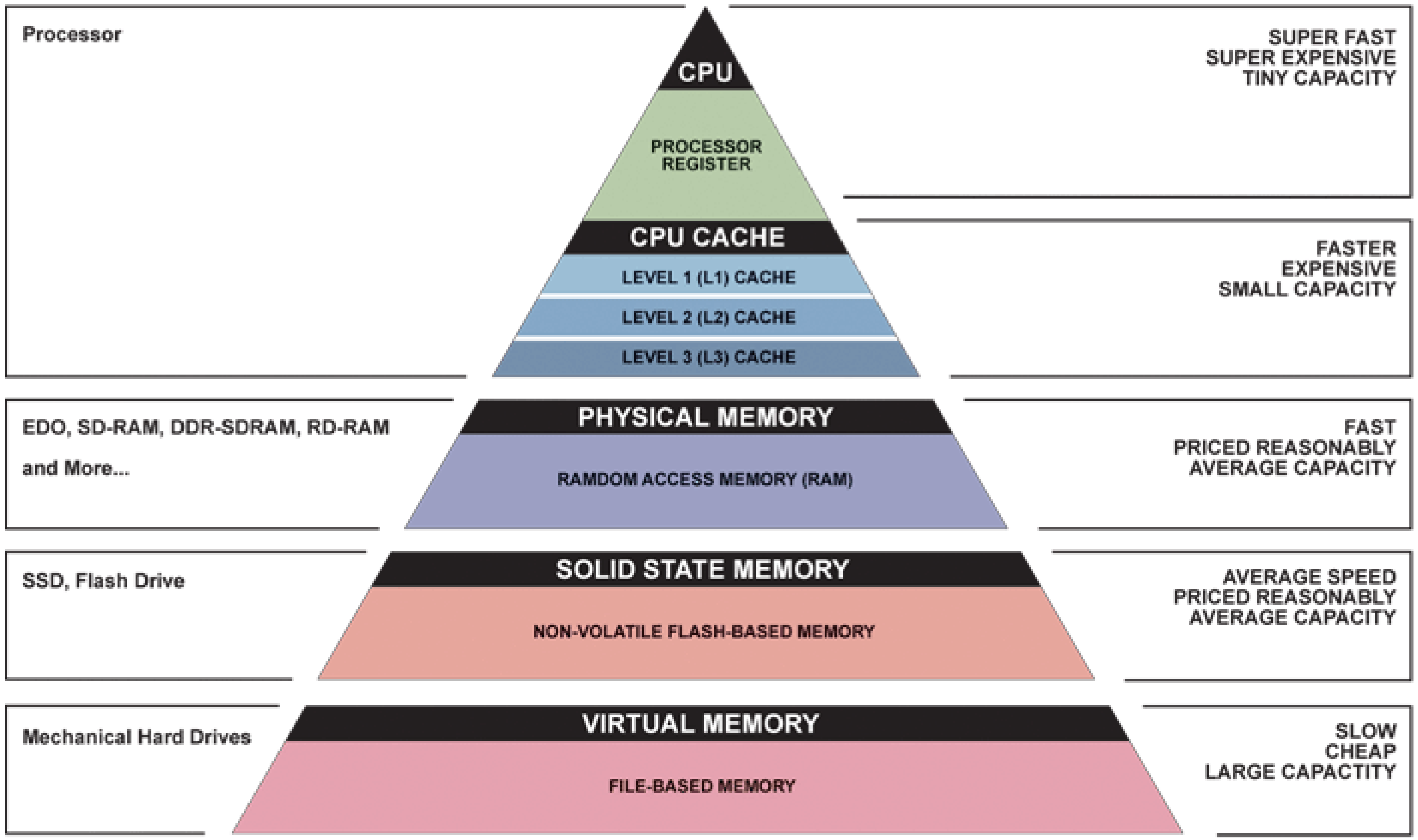
Break + Consideration

- Why use mmap() to put a file in your address space, when you could just read()/write() it instead?
  - Speed! No longer need to make system calls for each file access
  - A downside: now you need to handle file interactions yourself
    - Track offset for reading and writing
    - Make sure you don’t go past the end of the file

Outline

- Paging improvements
  - Improving translation speed
  - Improving table storage size
- OS Paging Implementation
- **Memory Hierarchy**

Memory Hierarchy



The OS view on registers

- Illusion: separate set for each process
- Reality: separate set for each core (or each thread in a core)
- OS needs to save and update registers whenever the currently running process changes
- Process and hardware handle moving memory into registers

The OS view on caches

- Mostly ignore them, handled by the hardware automatically
- Occasionally might need to clear them for security purposes
    - Or maybe there’s a way to tag cache entries with a process ID

- Addresses in the caches are either entirely physical addresses
- Or are virtually indexed, physically tagged
    - Cache lookup and TLB lookup happen in parallel
    - TLB result is used as Tag for cache to determine if there was a hit

The OS view on memory

- Managed through virtual memory translation
    - Paging (or Segmentation) that we talked about last time

- OS chooses which portions of processes go in RAM
    - Other portions of memory get “swapped” to disk
    - Writeable memory regions (stack, heap, global data) must be preserved
    - Read-only memory regions (code) can be reloaded from original location

The OS view on disk

- Non-volatile memory store
  - Everything else on the system disappears when power is removed (and cannot be trusted across reboots)

- Backing store for lots of information
  - Boot information: via “Master Boot Record” on disk
  - Filesystem, which the OS manages access to through system calls
  - Swap space, which the OS moves extra pages in and out of
    - Disk is significantly bigger than RAM, so this will work

- Disk is a **device** that the OS manages and reads in “blocks”
  - Compare to memory, which is directly addressed by processes

Outline

- Paging improvements
  - Improving translation speed
  - Improving table storage size
- OS Paging Implementation
- Memory Hierarchy