

Lecture 24: Virtual Memory III

Philipp Koehn

March 30, 2026

601.229 Computer Systems Fundamentals



More refinements

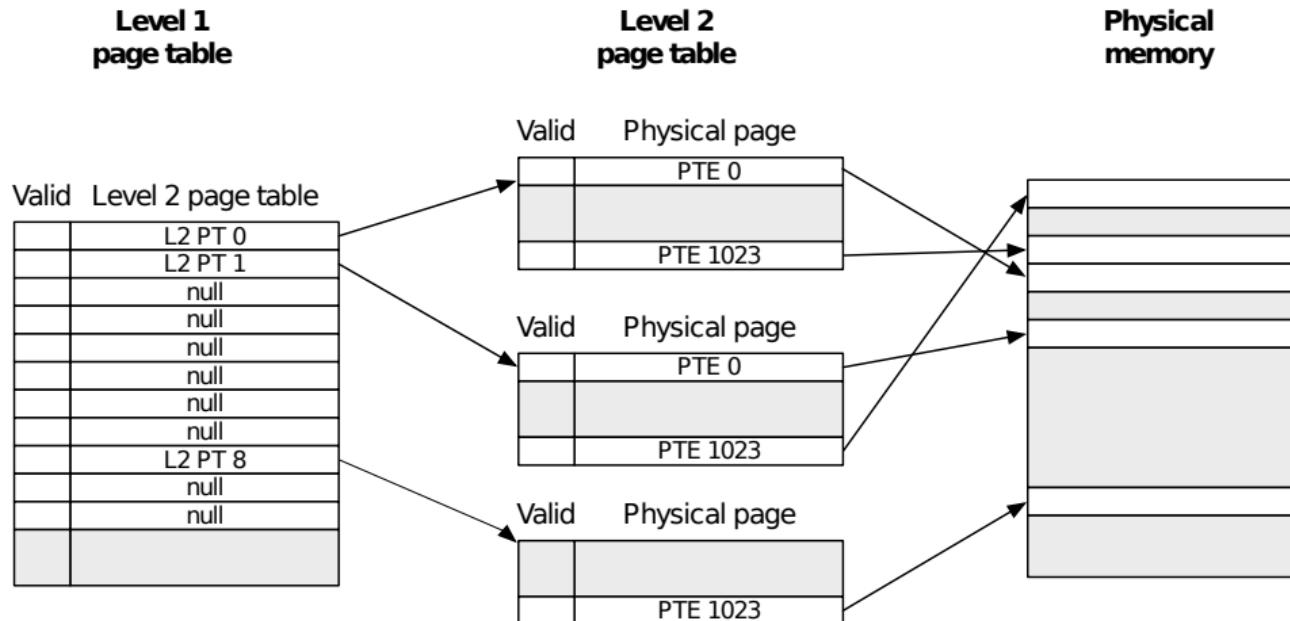
Refinements

- ▶ On-CPU cache
 - integrate cache and virtual memory
- ▶ Slow look-up time
 - use translation lookahead buffer (TLB)
- ▶ **Huge address space**
 - **multi-level page table**
- ▶ Putting it all together

Page Table Size

- ▶ Example
 - ▶ 32 bit address space: 4GB
 - ▶ Page size: 4KB
 - ▶ Size of page table entry: 4 bytes
 - Number of pages: 1M
 - Size of page table: 4MB
- ▶ Recall: one page table per process
- ▶ Very wasteful: most of the address space is not used

2-Level Page Table



Multi-Level Page Table

- ▶ Our example: 1M entries
- ▶ 2-level page table
 - each level 1K entry ($1K^2=1M$)
- ▶ 4-level page table
 - each level 32 entry ($32^4=1M$)

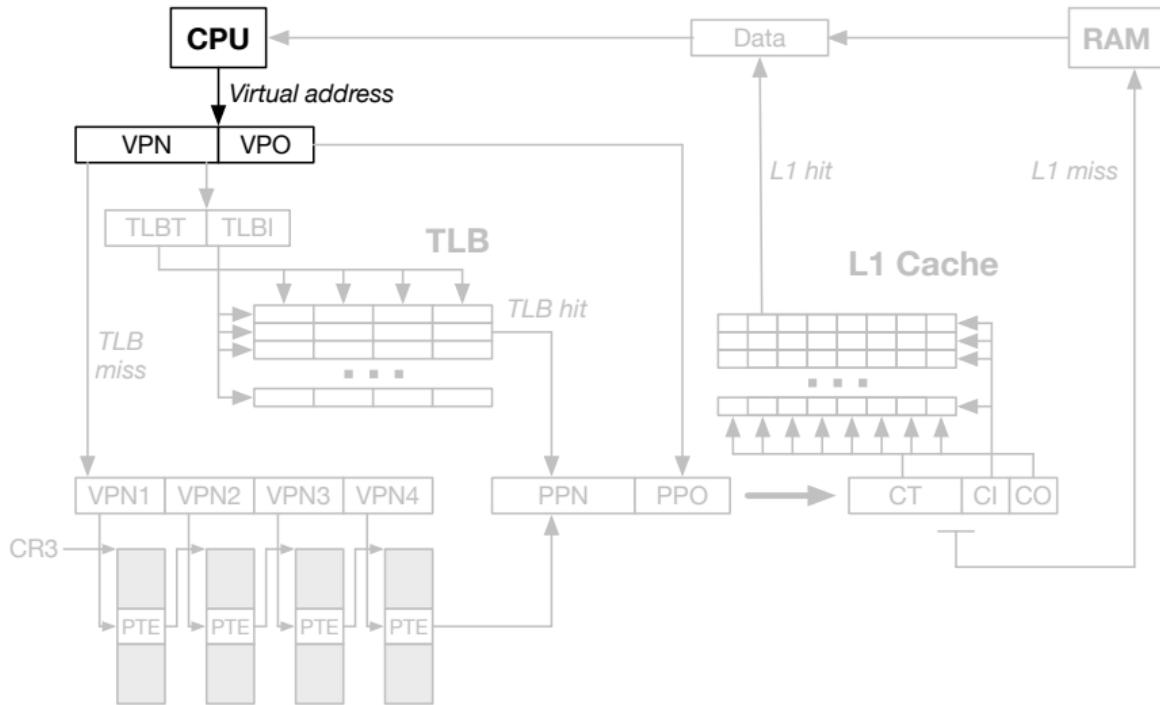
Clicker quiz!

Clicker quiz omitted from public slides

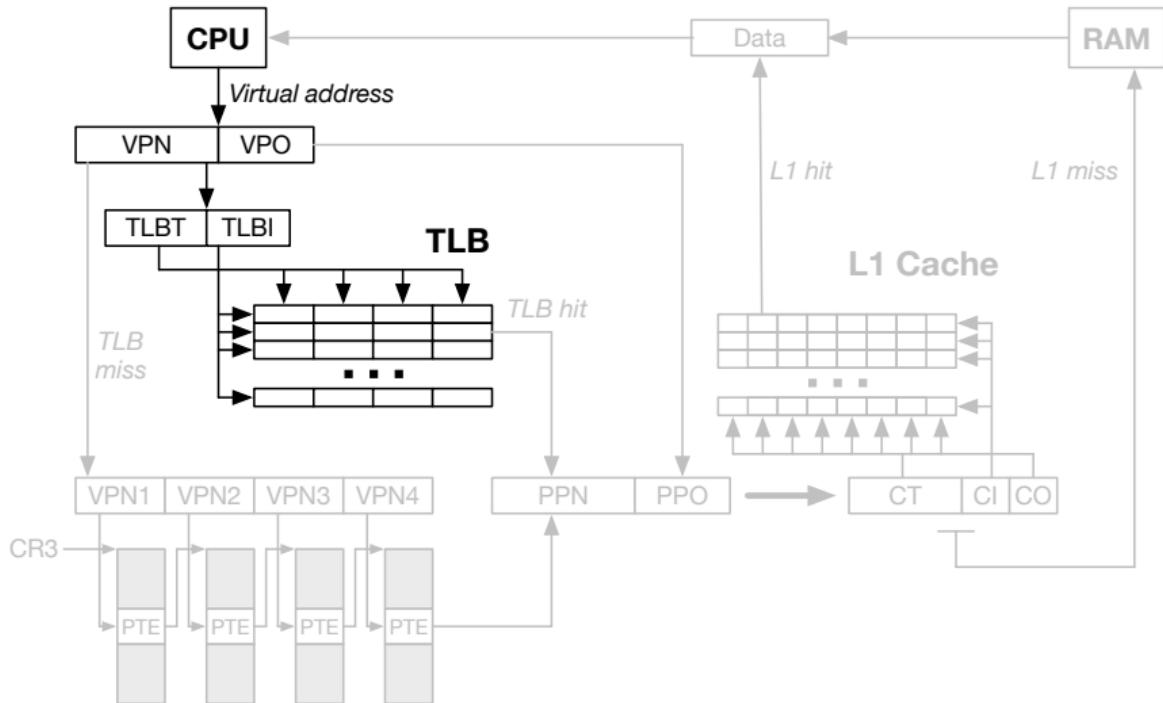
Refinements

- ▶ On-CPU cache
 - integrate cache and virtual memory
- ▶ Slow look-up time
 - use translation lookahead buffer (TLB)
- ▶ Huge address space
 - multi-level page table
- ▶ **Putting it all together**

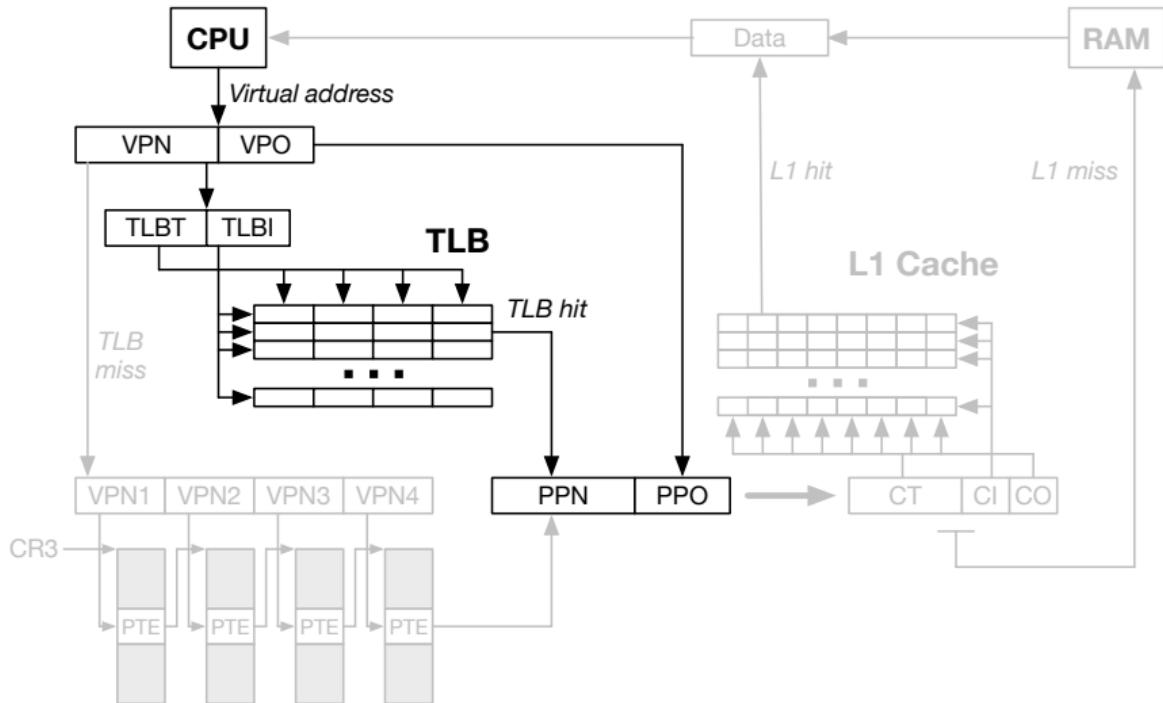
Virtual Address



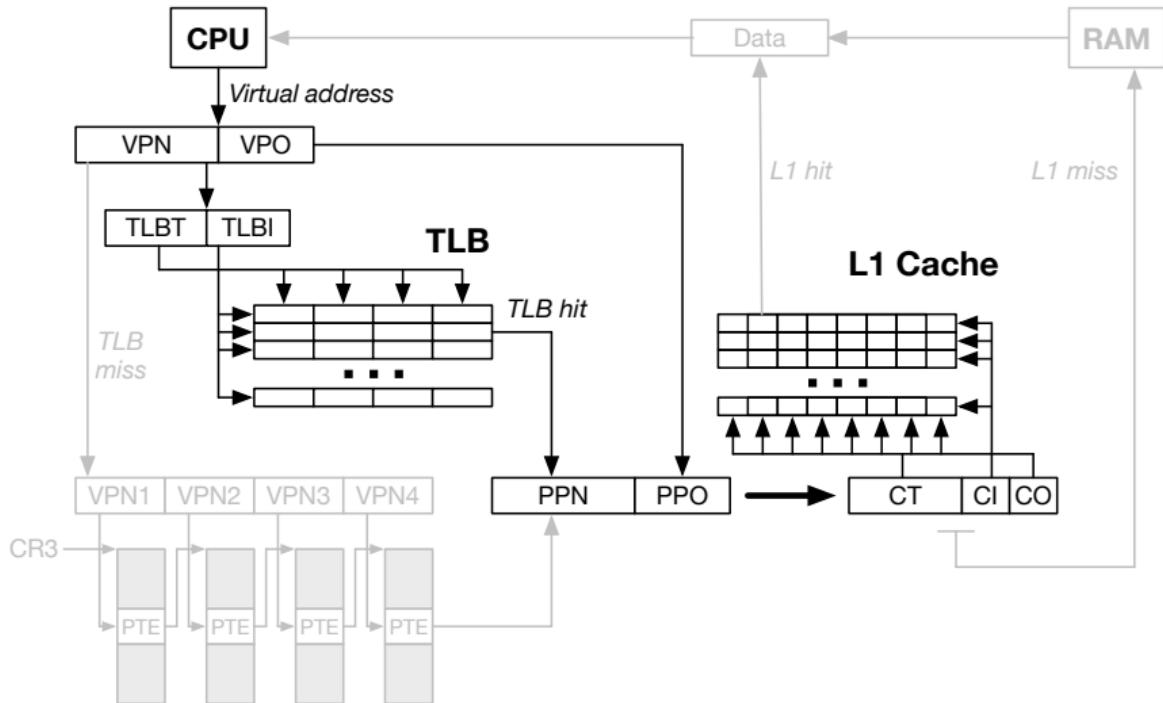
Translation Lookup Buffer



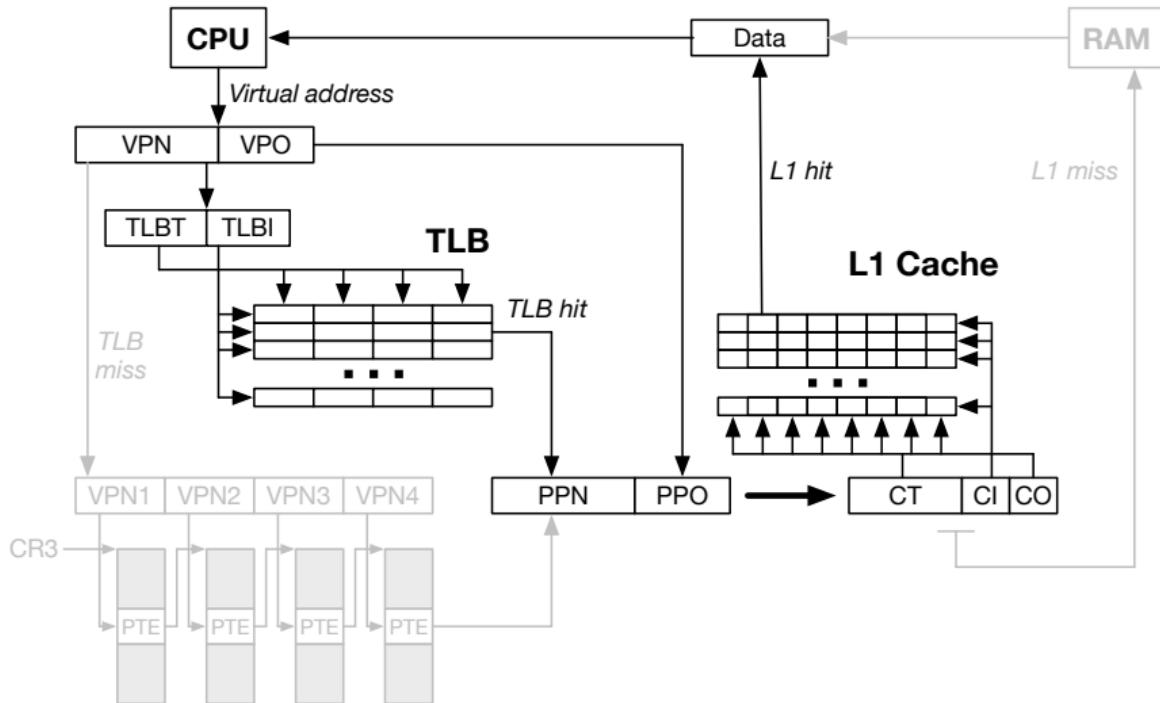
Compose Address



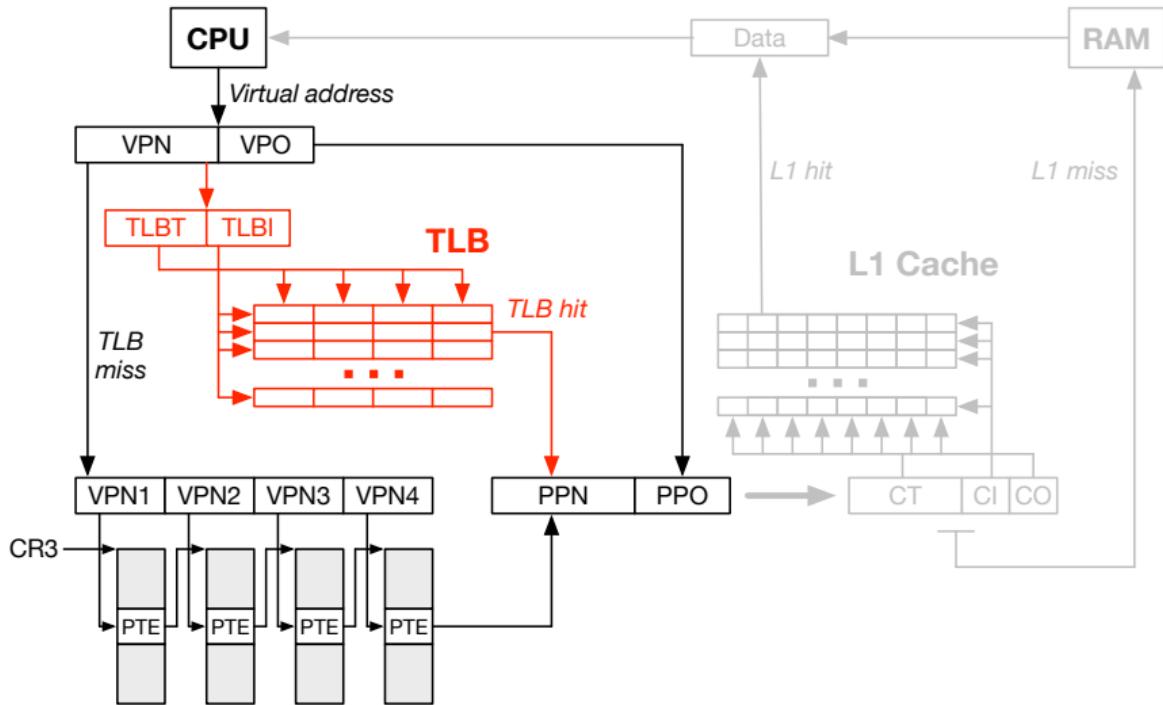
L1 Cache Lookup



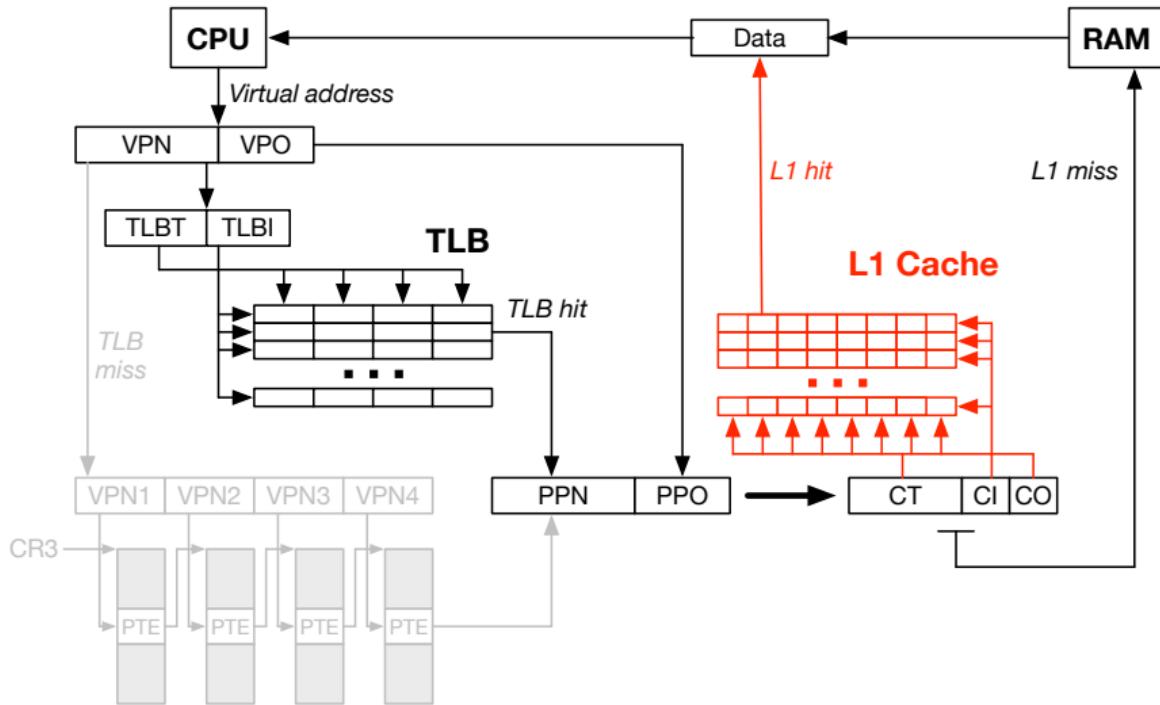
Return Data From L1 Cache



Translation Lookup Buffer Miss

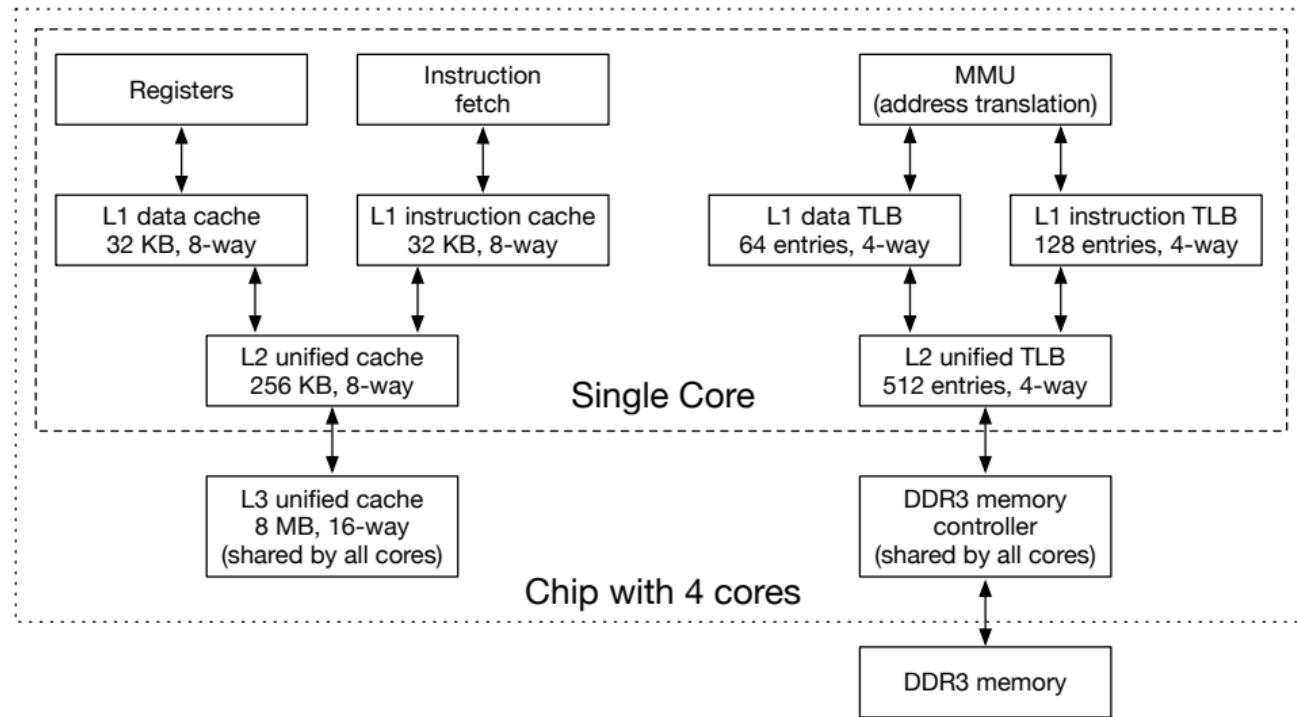


L1 Cache Miss



Core i7

Chip Layout



Sizes

- ▶ Virtual memory: 48 bit ($\rightarrow 2^{48} = 256\text{TB}$ address space)
- ▶ Physical memory: 52 bit ($\rightarrow 2^{52} = 4\text{PB}$ address space)
- ▶ Page size: 12 bit ($\rightarrow 2^{12} = 4\text{KB}$)
 $\Rightarrow 2^{36} = 64\text{G}$ entries, split in 4 levels (512 entries each)
- ▶ Translation lookup buffer (TLB): 4-way associative, 16 entries
- ▶ L1 cache: 8-way associative, 64 sets, 64 byte blocks (32 KB)
- ▶ L2 cache: 8-way associative, 512 sets, 64 byte blocks (256 KB)
- ▶ L3 cache: 16-way associative, 8K sets, 64 byte blocks (8 MB)

Linux

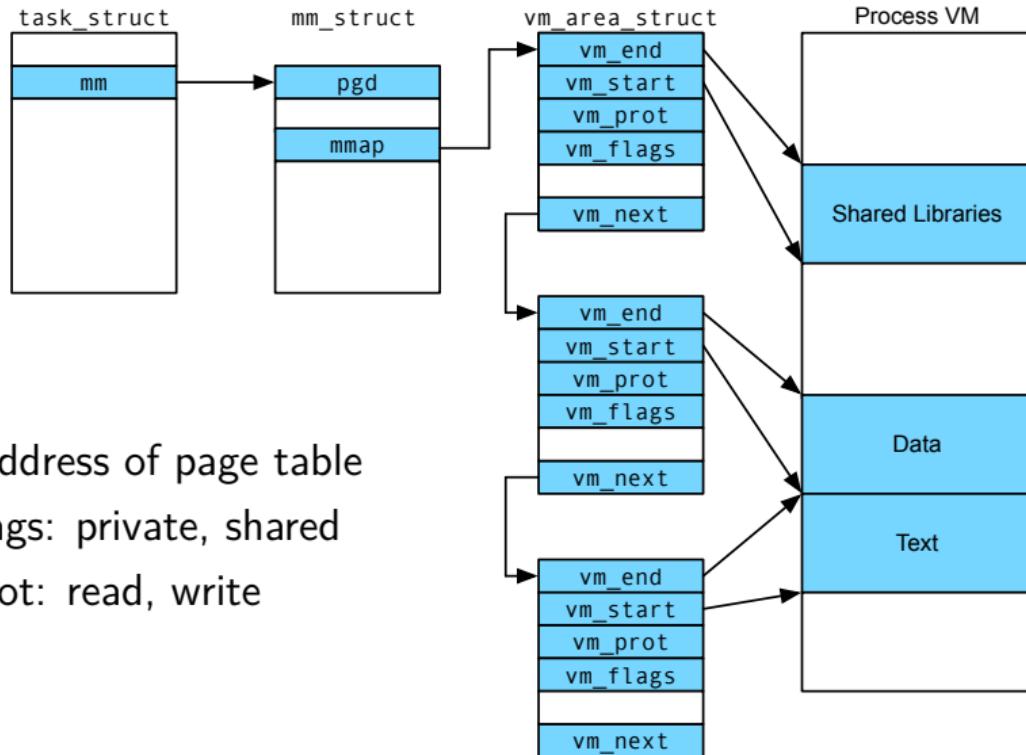
Big Picture

- ▶ Close co-operation between hardware and software
- ▶ Each process has its own virtual address space, page table
- ▶ Translation look-up buffer
 - when switching processes → flush
- ▶ Page table
 - when switching processes → update pointer to top-level page table
- ▶ Page tables are always in physical memory
 - pointers to page table do not require translation

Handling Page Faults

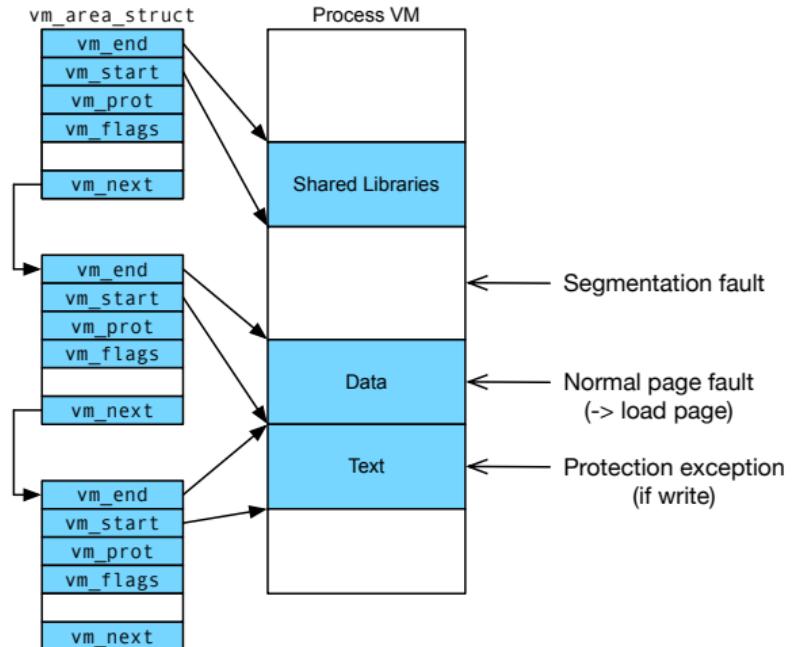
- ▶ Page faults trigger an exception (hardware)
- ▶ Exception is handled by software (Linux kernel)
- ▶ Kernel must determine what to do

Linux Virtual Memory Areas



- ▶ `pgd`: address of page table
- ▶ `vm_flags`: private, shared
- ▶ `vm_prot`: read, write

Handling Page Faults



Kernel walks through `vm_area_struct` list to resolve page fault

Memory mapping

Objects on Disk

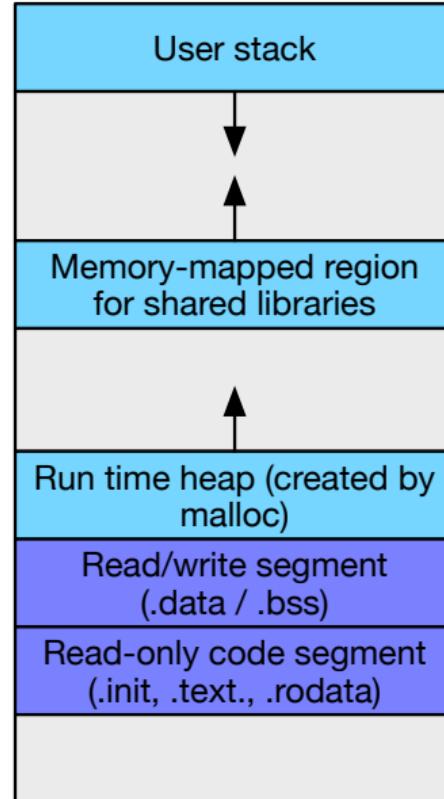
- ▶ Area of virtual memory = file on disk
- ▶ Regular file in file system
 - ▶ file divided up into pages
 - ▶ demand loading: just mapped to addresses, not actually loaded
 - ▶ could be code, shared library, data file
- ▶ Anonymous file
 - ▶ typically allocated memory
 - ▶ when used for the first time: set all values to zero
 - ▶ never really on disk, except when swapped out

Shared Object

- ▶ A shared object is a file on disk
- ▶ Private object
 - ▶ only its process can read/write
 - ▶ changes not visible to other processes
- ▶ Shared object
 - ▶ multiple processes can read/write
 - ▶ changes visible to other processes

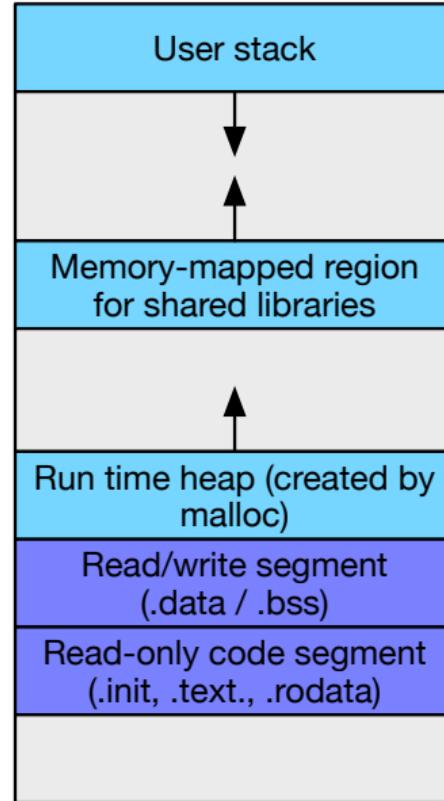
fork()

- ▶ Creates a new child process
- ▶ Copies all
 - ▶ virtual memory area structures
 - ▶ memory mapping structures
 - ▶ page tables
- ▶ New process has identical access to existing memory



execve()

- ▶ Creates a new process
- ▶ Deletes all user areas
- ▶ Map private areas (.data, .code, .bss)
- ▶ Map shared libraries
- ▶ Set program counter



User-Level Memory Mapping

- ▶ Process can create virtual memory areas with `mmap` (may be loaded from file)
- ▶ Protection options (handled by kernel / hardware)
 - ▶ executable code
 - ▶ read
 - ▶ write
 - ▶ inaccessible
- ▶ Mapping options
 - ▶ anonymous: data object initially zeroed out
 - ▶ private
 - ▶ shared

Dynamic memory allocation

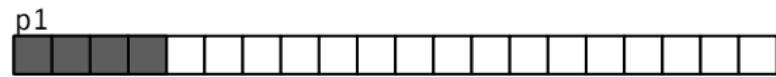
Memory Allocation in C

- ▶ `malloc()`
 - ▶ allocate specified amount of data
 - ▶ return pointer to (virtual) address
 - ▶ memory is allocated on heap
- ▶ `free()`
 - ▶ frees memory allocated at pointer location
 - ▶ may be between other allocated memory
- ▶ Need to track of list of allocated memory

Assumptions

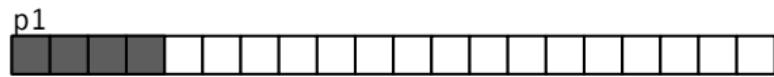
- ▶ Each square is a 4-byte word
- ▶ Heap consists of 20 words
- ▶ Allocations must be aligned on a multiple of 8
- ▶ Shading indicates use:
 - ▶ No shading: unallocated memory
 - ▶ Dark: allocated memory
 - ▶ Light: padding to ensure alignment

Example



```
p1 = malloc(4*sizeof(int))
```

Example

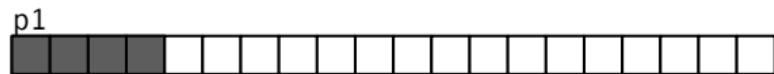


```
p1 = malloc(4*sizeof(int))
```



```
p2 = malloc(5*sizeof(int))
```

Example



```
p1 = malloc(4*sizeof(int))
```

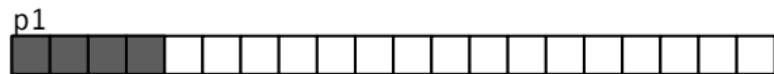


```
p2 = malloc(5*sizeof(int))
```

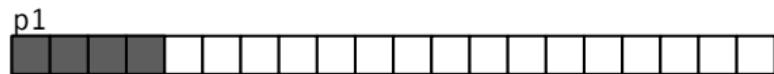


```
p3 = malloc(6*sizeof(int))
```

Example



Example



```
p1 = malloc(4*sizeof(int))
```



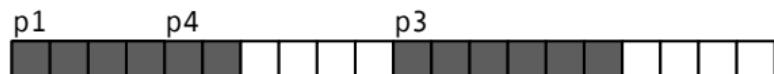
```
p2 = malloc(5*sizeof(int))
```



```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```



```
p4 = malloc(2*sizeof(int))
```

Fragmentation

- ▶ Internal: unused space due to padding for
 - ▶ alignment
 - ▶ minimum block size
- ▶ External: as memory is allocated and freed:
 - ▶ allocated blocks are scattered over the heap area
 - ▶ there are gaps of various sizes between allocated blocks
 - ▶ it might not be possible to find a large enough gap to satisfy an allocation request, even though enough aggregate memory is available

Free list

- ▶ Free list
 - ▶ need to maintain a list of free memory areas
 - ▶ implicit: space between allocated memory
 - ▶ explicit: maintain a separate list