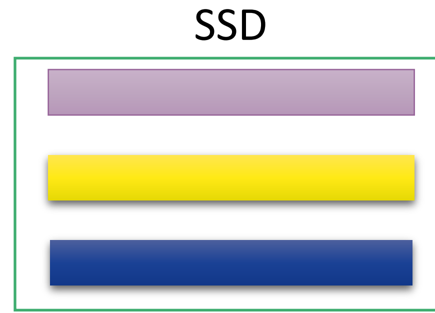
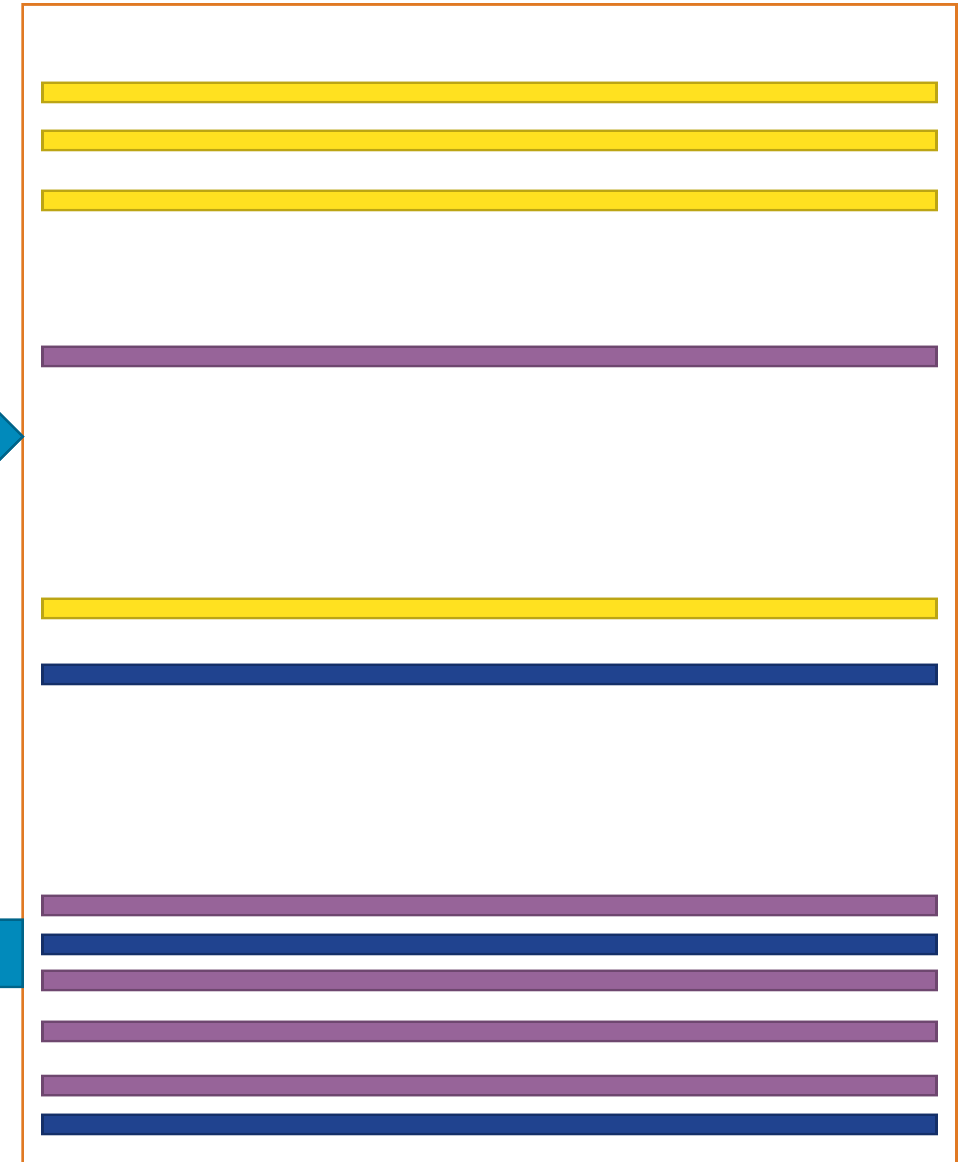
# Paging Virtual Memory

# Abstractions

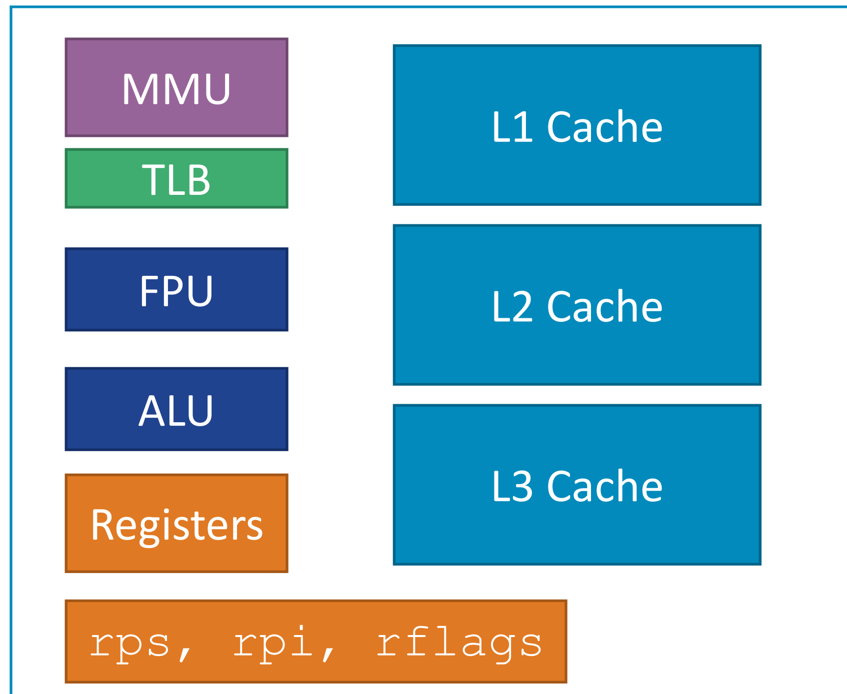
SSD



Physical Memory

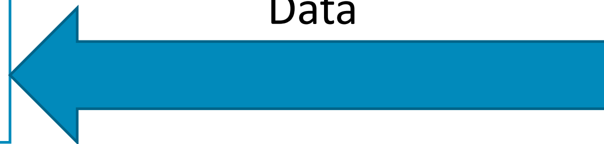


Core




Address

Data

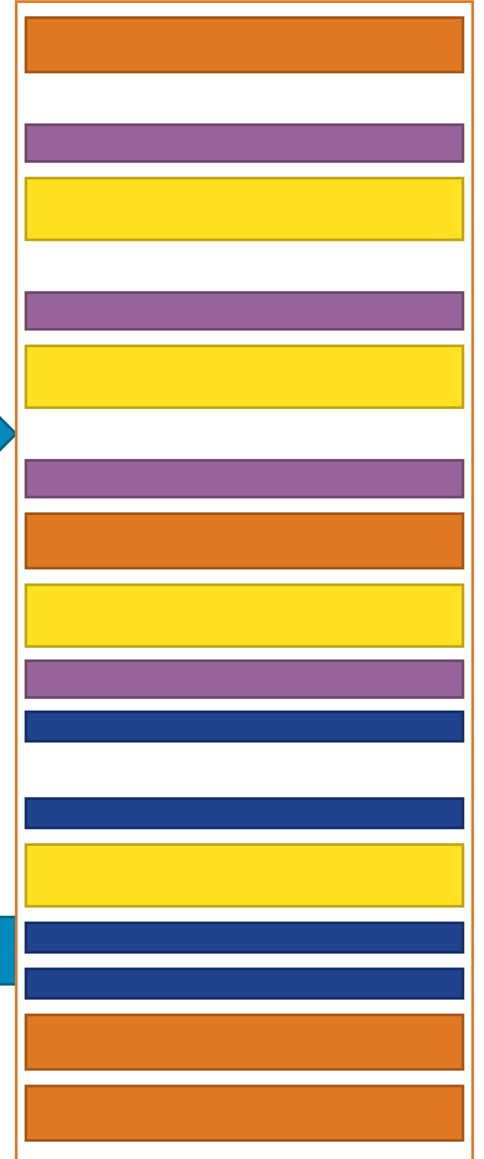


What if we introduce another process and we've run out of physical memory?

# Paging

SSD

Physical Memory



Exception

Invalid page or invalid access

Core

MMU Page Table

| P Frame | Permissions |
| --- | --- |
| 47 | R-X |
| NULL | RW- |
| 13 | RW- |
| 42 | RW- |
| 105 | R-X |

Virtual Address (VA)

Physical Address (PA)

Cache

Physical Address

Data

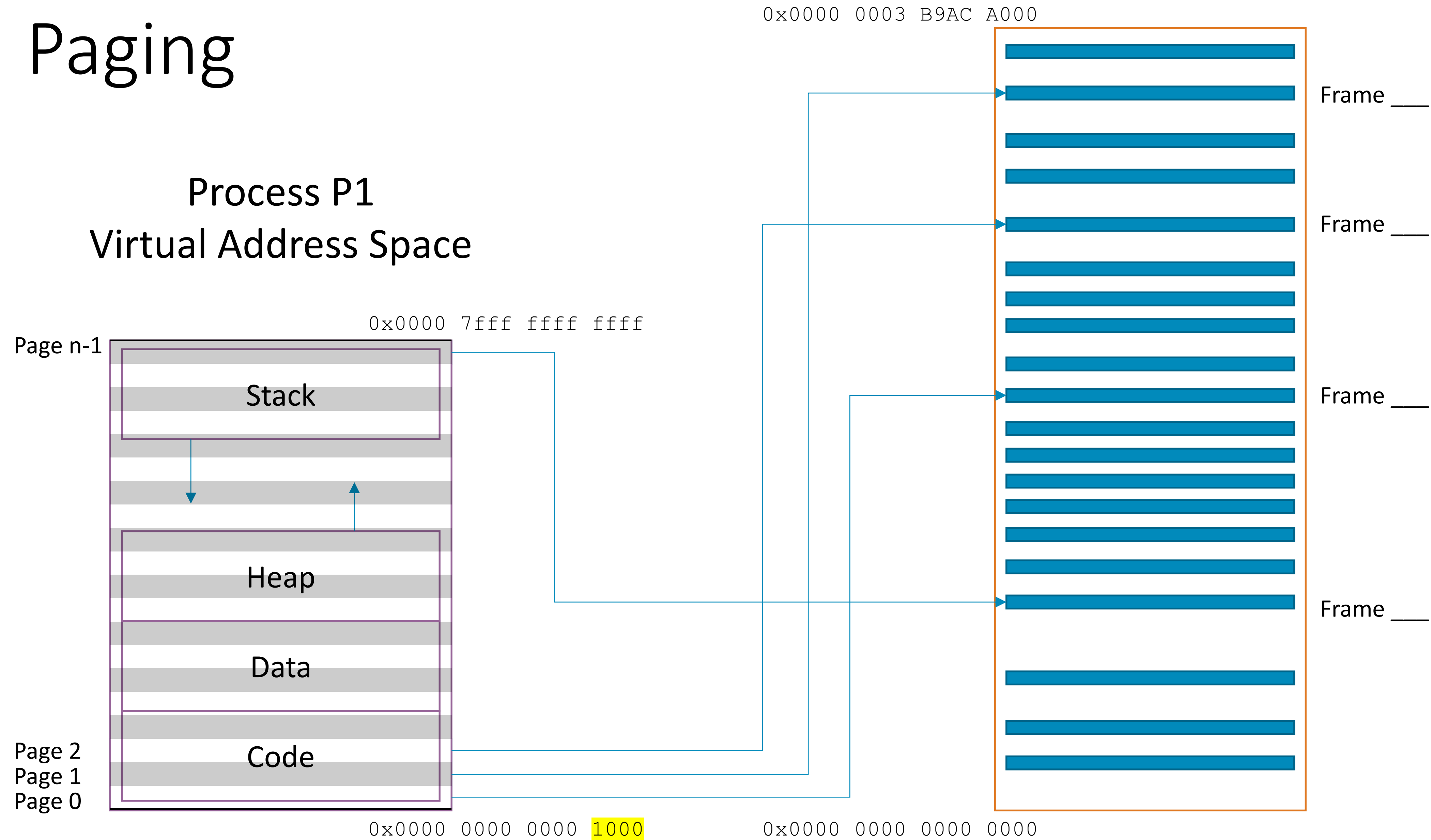
PA = Frame[Page] +++ Offset

Concatenation

VA must be decoded into Page and Offset (like caching).
Page # is index into page table (gives P Frame)
Offset gives byte address inside P Frame

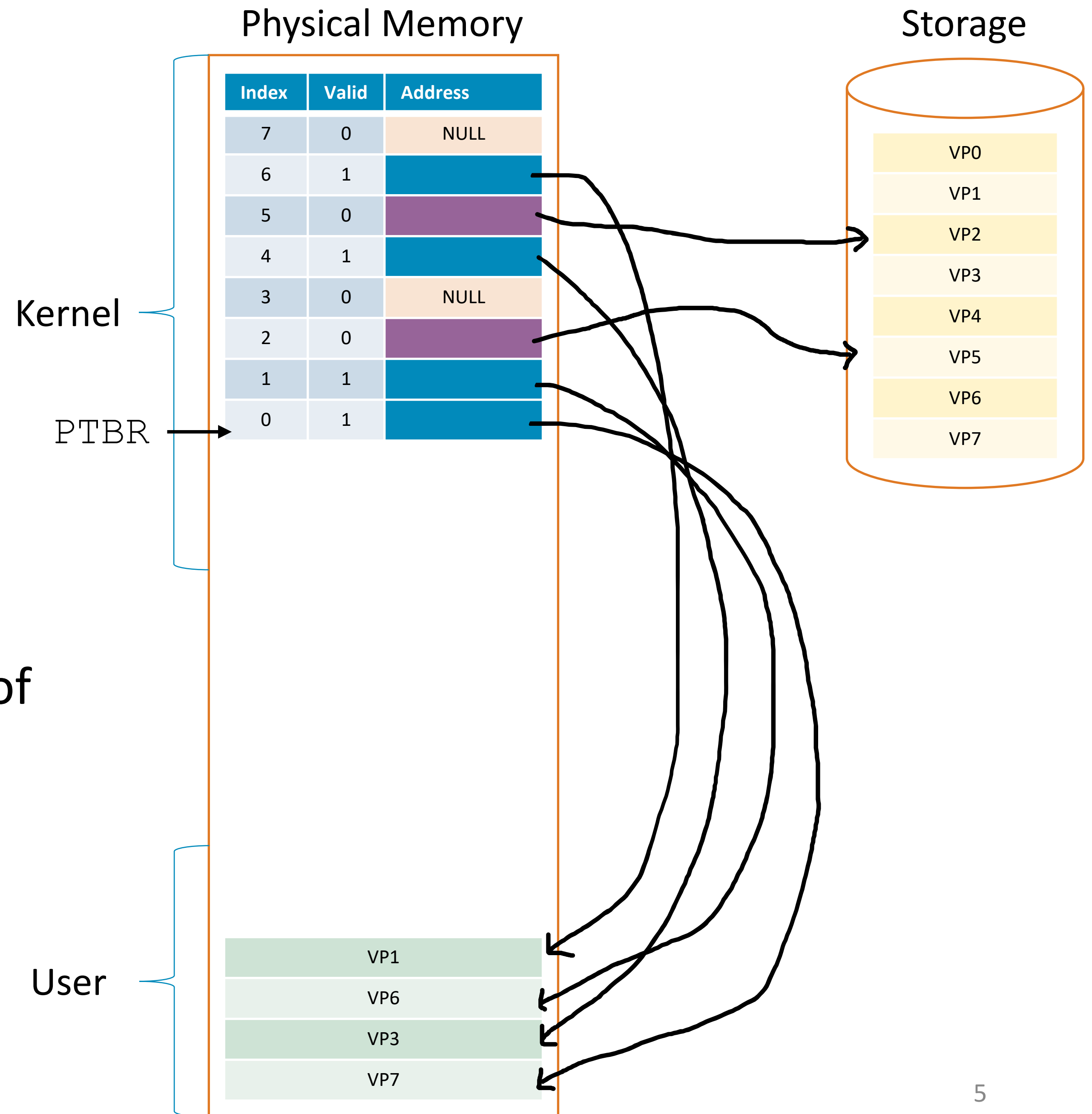
VA:

# Paging

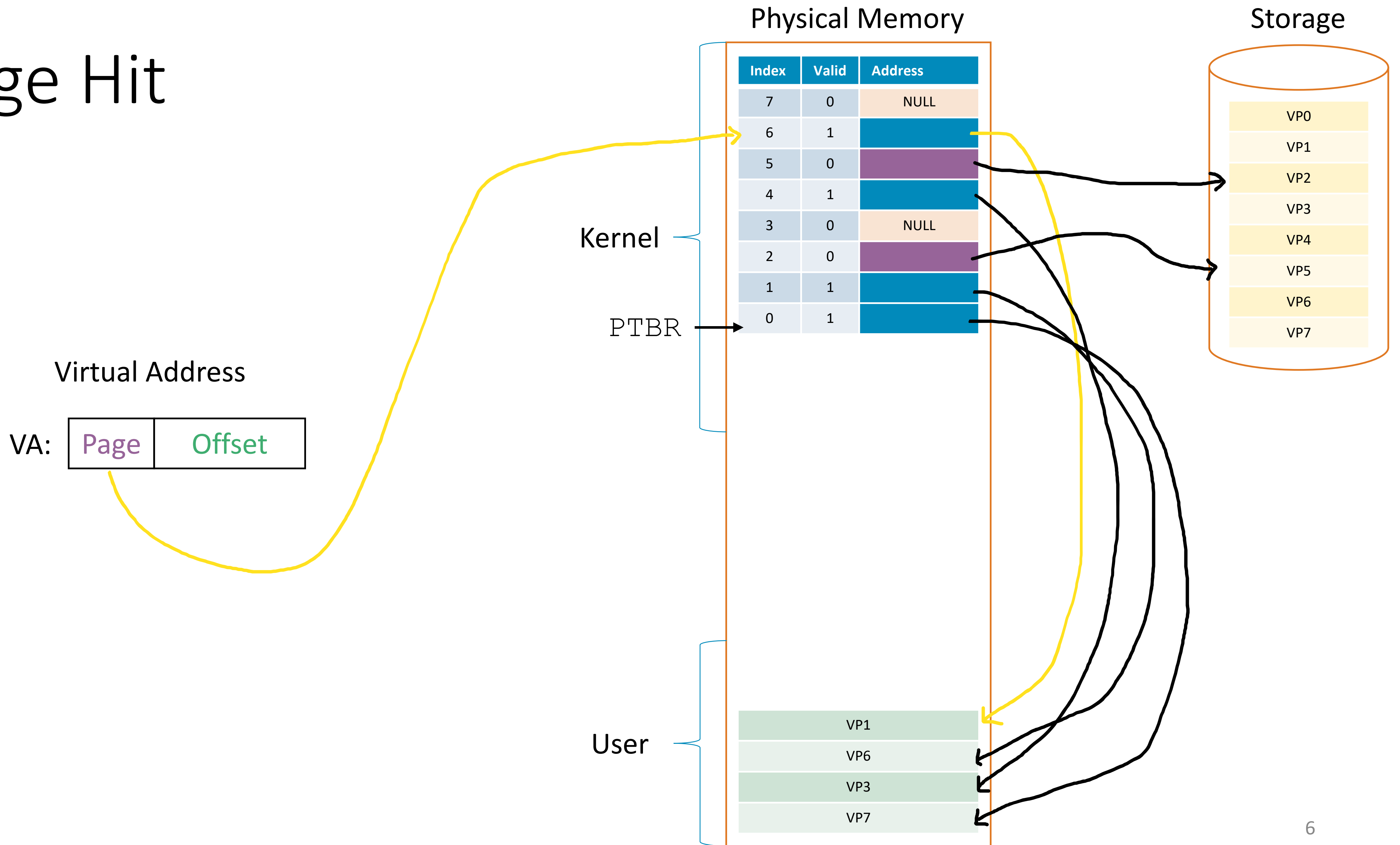


# Simple (Bad) Paging

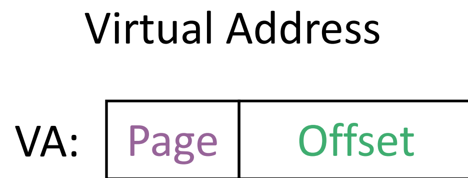
- The page table is stored in physical memory
- It is implemented as array of page table entries
- The page table base register (PTBR) stores physical address of beginning of page table
- Page table entries are accessed by using the page number as the index into the page table



# Page Hit



# Page Fault




What should we do?

Replace existing page and update page table.

Physical Memory

Kernel

PTBR

| Index | Valid | Address |
|---|---|---|
| 7 | 0 | NULL |
| 6 | 1 | |
| 5 | 0 | |
| 4 | 1 | |
| 3 | 0 | NULL |
| 2 | 0 | |
| 1 | 1 | |
| 0 | 1 | |

User

VP1
VP6
VP3
VP7

Storage

VP0
VP1
VP2
VP3
VP4
VP5
VP6
VP7

# Process Memory

- Nothing loaded into physical memory until use (<u>not</u> loaded on `fork`)
- Working/resident set of pages are those currently in physical memory
  - Loaded on demand
- Typically, one page table per process
- Can share pages among multiple processes (shared libraries)
- Enables memory protection (mark as readable, writable, executable)

# Problems with paging as presented

- **Memory Consumption:** page table is really big
  - 32-Bit address space → 4GB of memory that we can address
  - Page size: 4 KB
  - Size of page table entry: 4 bytes
  - → 1M pages
  - → 4 MB for the page table (for each process!)
  - → 400 MB for 100 processes (10% of memory just for the page tables)

Multi-Level Page Tables

- **Performance:** every data/instruction access requires *two* memory accesses:
  - One for the page table
  - One for the data/instruction

Translation Lookaside Buffer

# Multi-Level Page Tables (2-Level Example)

Page Table Directory
(Level 1 Page Table)

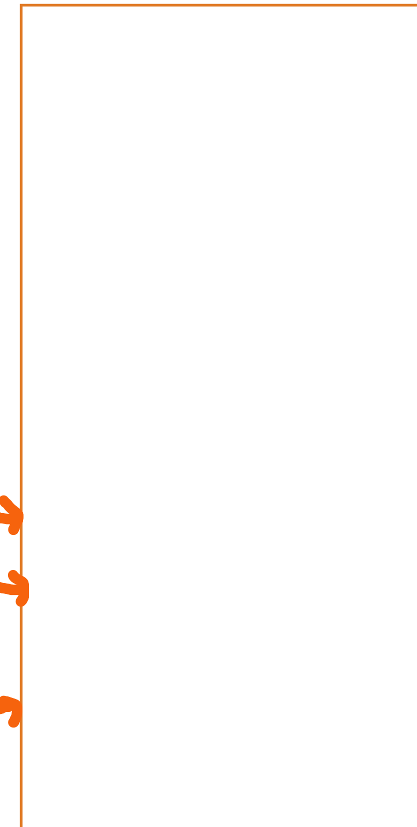
| Valid | Page Table Address |
|---|---|
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |

Page Tables
(Level 2 Page Table)

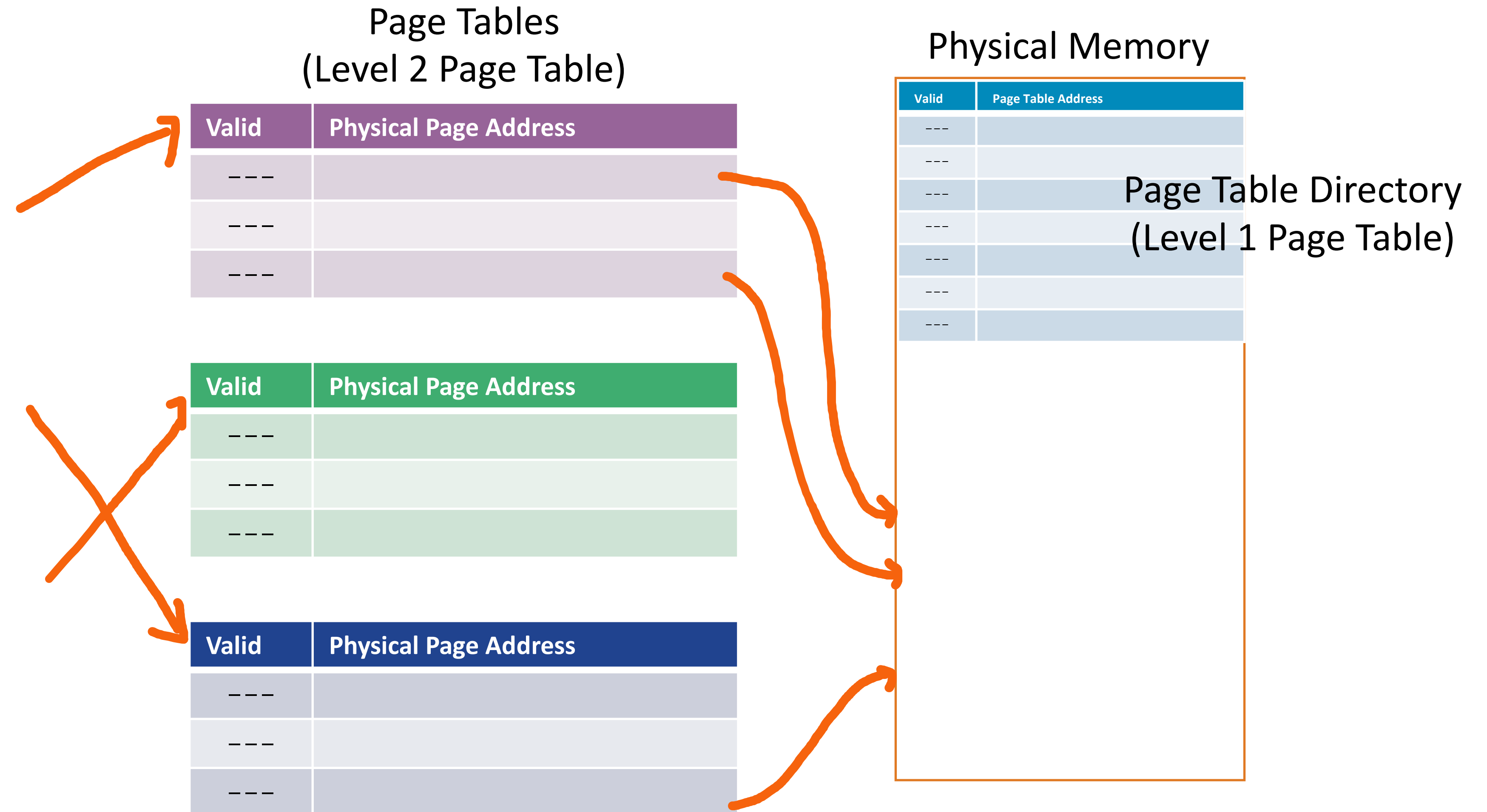
| Valid | Physical Page Address |
|---|---|
| --- | |
| --- | |
| --- | |

| Valid | Physical Page Address |
|---|---|
| --- | |
| --- | |
| --- | |

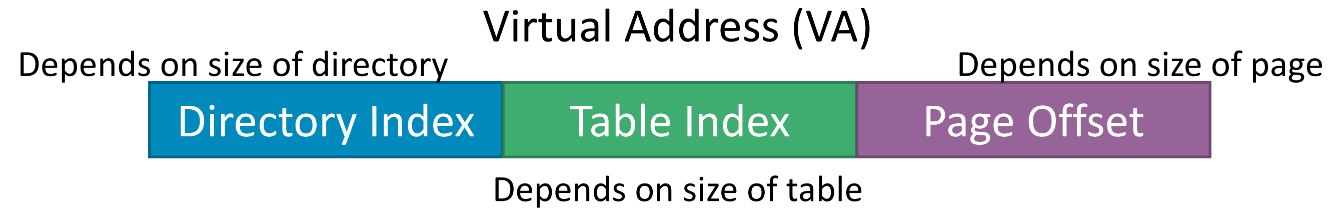
| Valid | Physical Page Address |
|---|---|
| --- | |
| --- | |
| --- | |



# Multi-Level Page Tables (2-Level Example)



# 2-Level Page Table Lookup

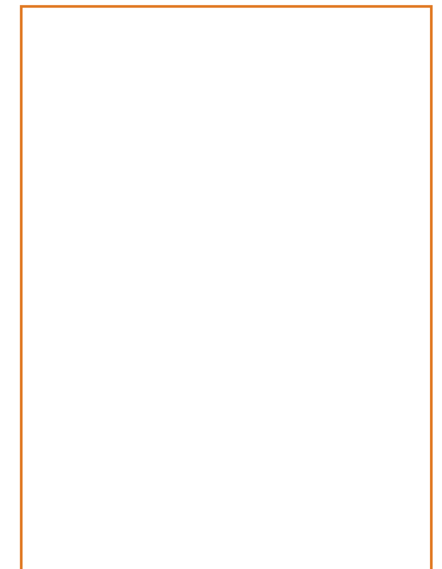



| Valid | Page Table Address |
|---|---|
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |
| --- | |

Page Table Directory
(Level 1 Page Table)

| Valid | Physical Page Address |
|---|---|
| --- | |
| --- | |
| --- | |

Page Tables
(Level 2 Page Table)



Physical Memory

# Multi-Level Page Tables

- Example from before
  - 32-Bit address space → 4GB of memory that we can address
  - Page size: 4 KB
  - Size of page table entry: 4 bytes
  - → 1M pages
  - → 4 MB for the page table (for each process!)
  - → 400 MB for 100 processes (10% of memory just for the page tables)

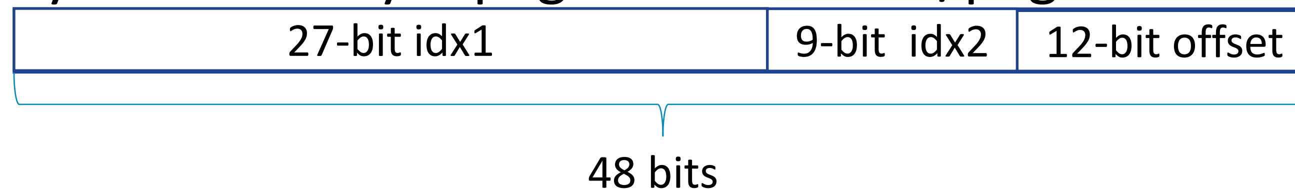
- 2-level page table with 1,000 directory entries
  - → 1 KB for the page directory ($1000 \times 1000 = 1M$)

- 4-level page table with 32 directory entries per level
  - → 32 bytes for the page directory ($32^4 = 1M$)

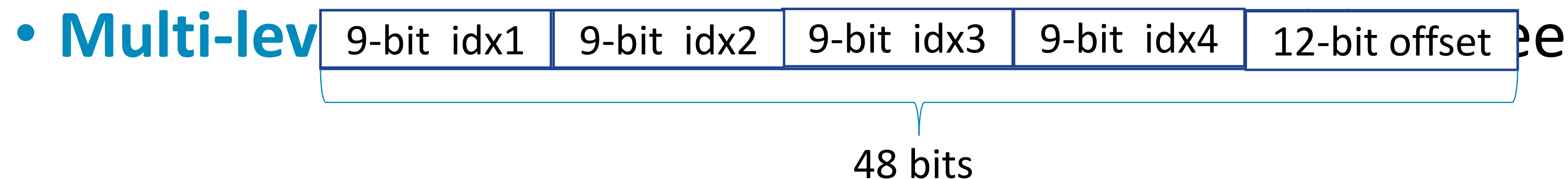
# Multi-level Page Tables

**128 MB**

- Problem: How big does the page directory get?
  - Assume you have a 48-bit address space
  - Assume you have 4KiB pages
  - Assume you have 8-byte page table entries/page directory entries




- Goal: Page Table Directory should fit in one frame
- **Multi-lev** ... **ee**



# Exercise 1: Two-level Page Tables

- Assume you are working on an architecture with a 32-bit virtual address space in which idx1 is 4 bits, idx2 is 12 bits, and offset is 16 bits.

- How big is a page in this architecture? $2^{16}$ **bytes = 64 KB**

- How big is a page table entry in this architecture? **16 bytes**

# Exercise 2: Two-level Page Tables

Assume you are still working on that

| 4 bit idx1 | 12 bit idx2 | 16 bit offset |
|---|---|---|

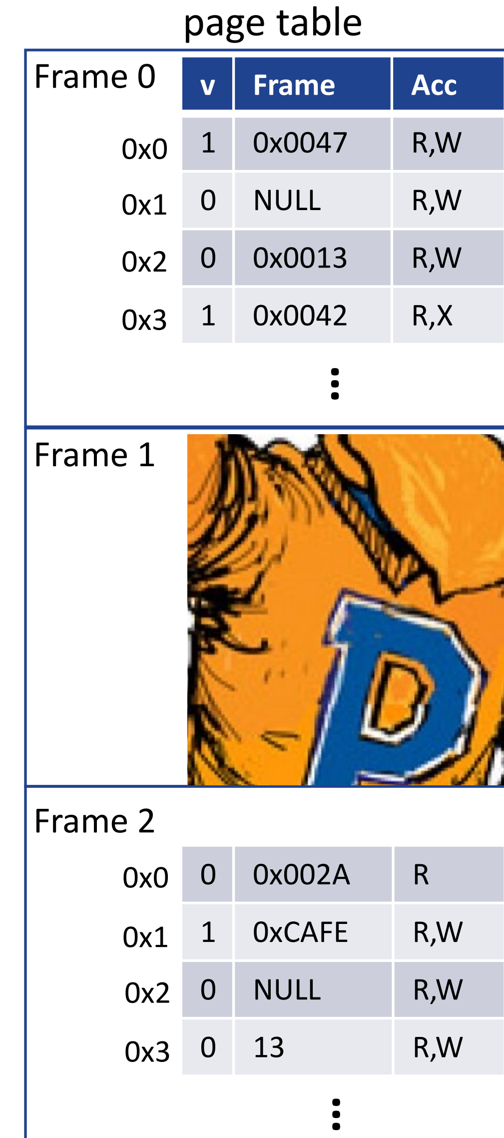
Compute the physical address corresponding to each of the virtual address (or answer "invalid"):

a) 0x00000000 **0x00470000**
b) 0x20022002 **invalid**
c) 0x10015555 **0xCAFE5555**

page directory

| | v | PTFrame |
|---|---|---|
| 0x0 | 1 | 0x0 |
| 0x1 | 1 | 0x2 |
| 0x2 | 0 | NULL |
| 0x3 | 0 | NULL |
| | ⋮ | |
| 0xF | 0 | NULL |

page table



Frame 0

| | v | Frame | Acc |
|---|---|---|---|
| 0x0 | 1 | 0x0047 | R,W |
| 0x1 | 0 | NULL | R,W |
| 0x2 | 0 | 0x0013 | R,W |
| 0x3 | 1 | 0x0042 | R,X |
| | | ⋮ | |

Frame 1

Frame 2

| | v | Frame | Acc |
|---|---|---|---|
| 0x0 | 0 | 0x002A | R |
| 0x1 | 1 | 0xCAFE | R,W |
| 0x2 | 0 | NULL | R,W |
| 0x3 | 0 | 13 | R,W |
| | | ⋮ | |

# Problems with paging as presented

- **Memory Consumption:** page table is really big
  - 32-Bit address space → 4GB of memory that we can address
  - Page size: 4 KB
  - Size of page table entry: 4 bytes
  - → 1M pages
  - → 4 MB for the page table (for each process!)
  - → 400 MB for 100 processes (10% of memory just for the page tables)

Multi-Level Page Tables

- **Performance:** every data/instruction access requires *two* memory accesses:
  - One for the page table
  - One for the data/instruction

Translation Lookaside Buffer

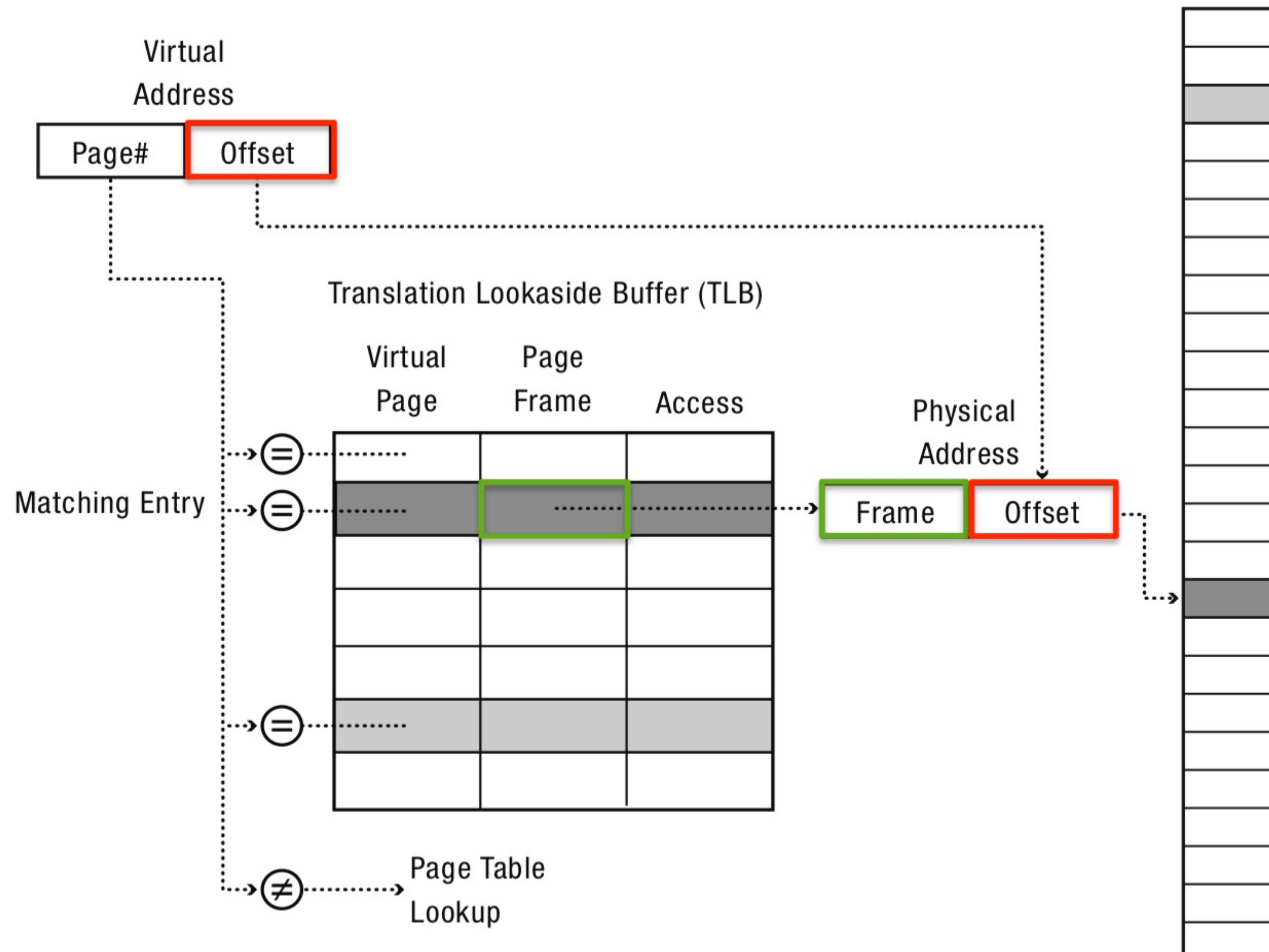
# Translation-Lookaside Buffer (TLB)

- Every memory-related instruction passes through the MMU
- Very frequent → must be very fast
- How about some additional locality and caching on the CPU?

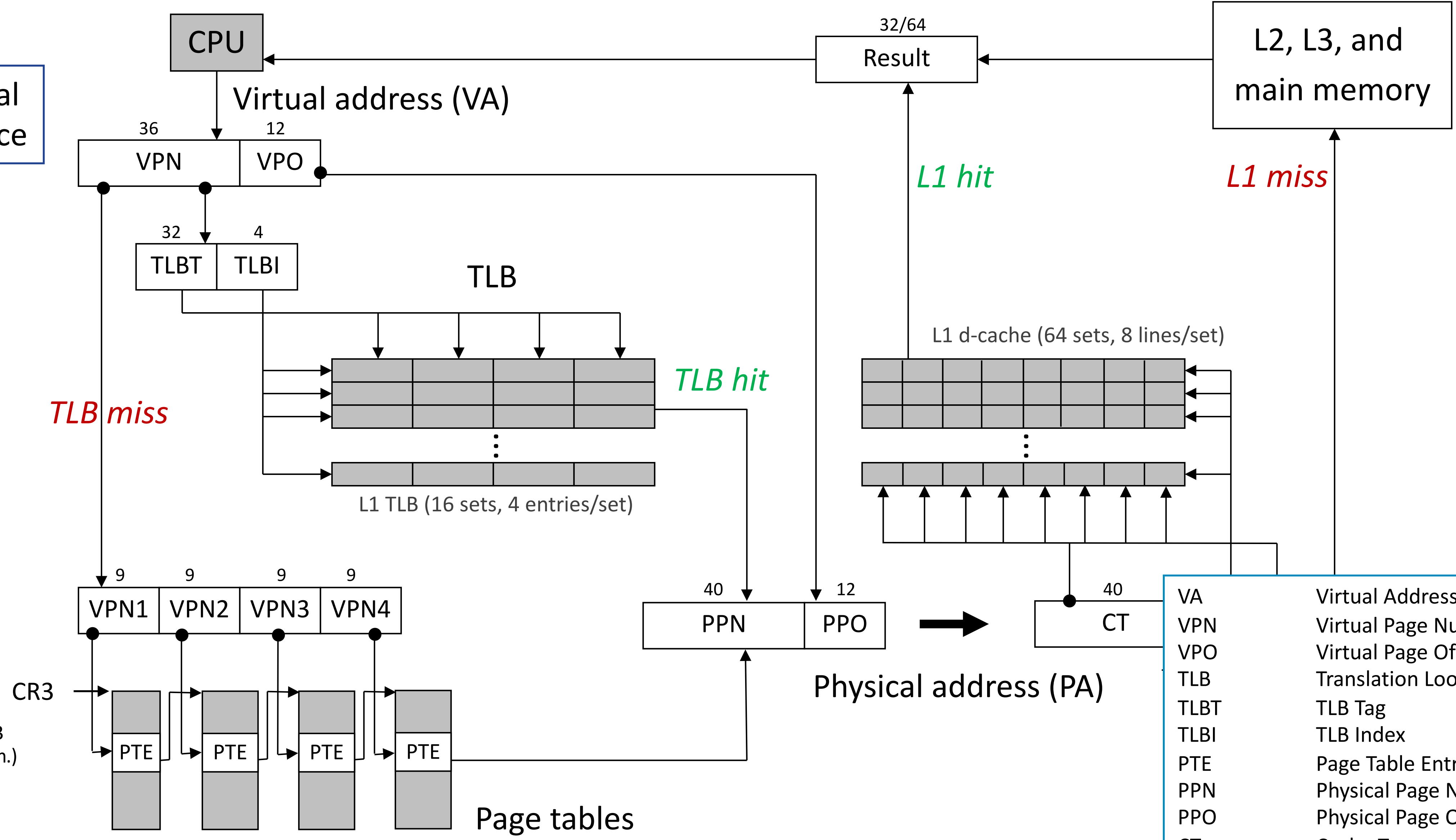
- **Translation-lookaside buffer** is an address translation cache that is built into the MMU

- Using the TLB: First look up page table entry in the TLB (instead of page table)

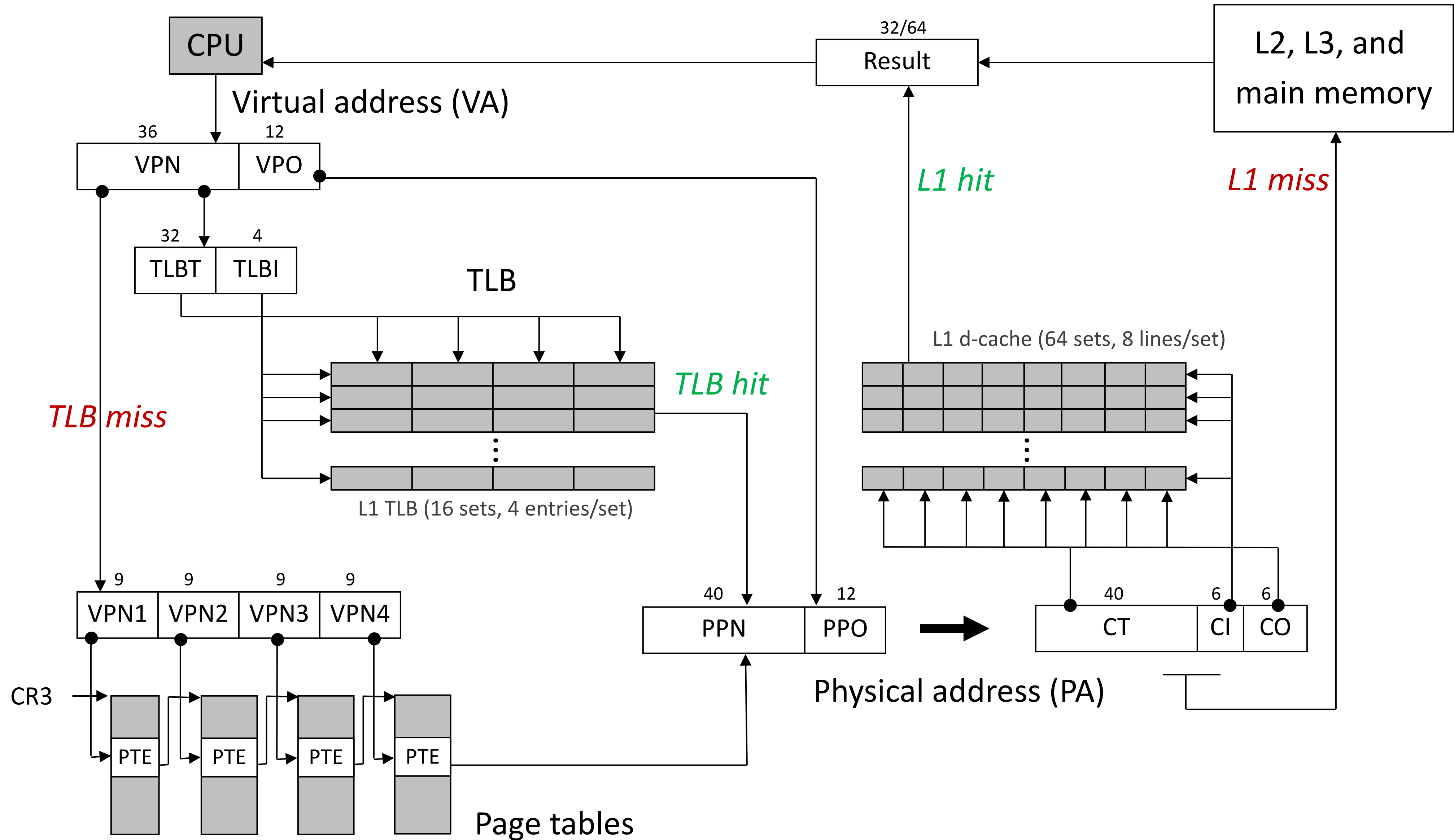
# Translation-Lookaside Buffer (TLB)

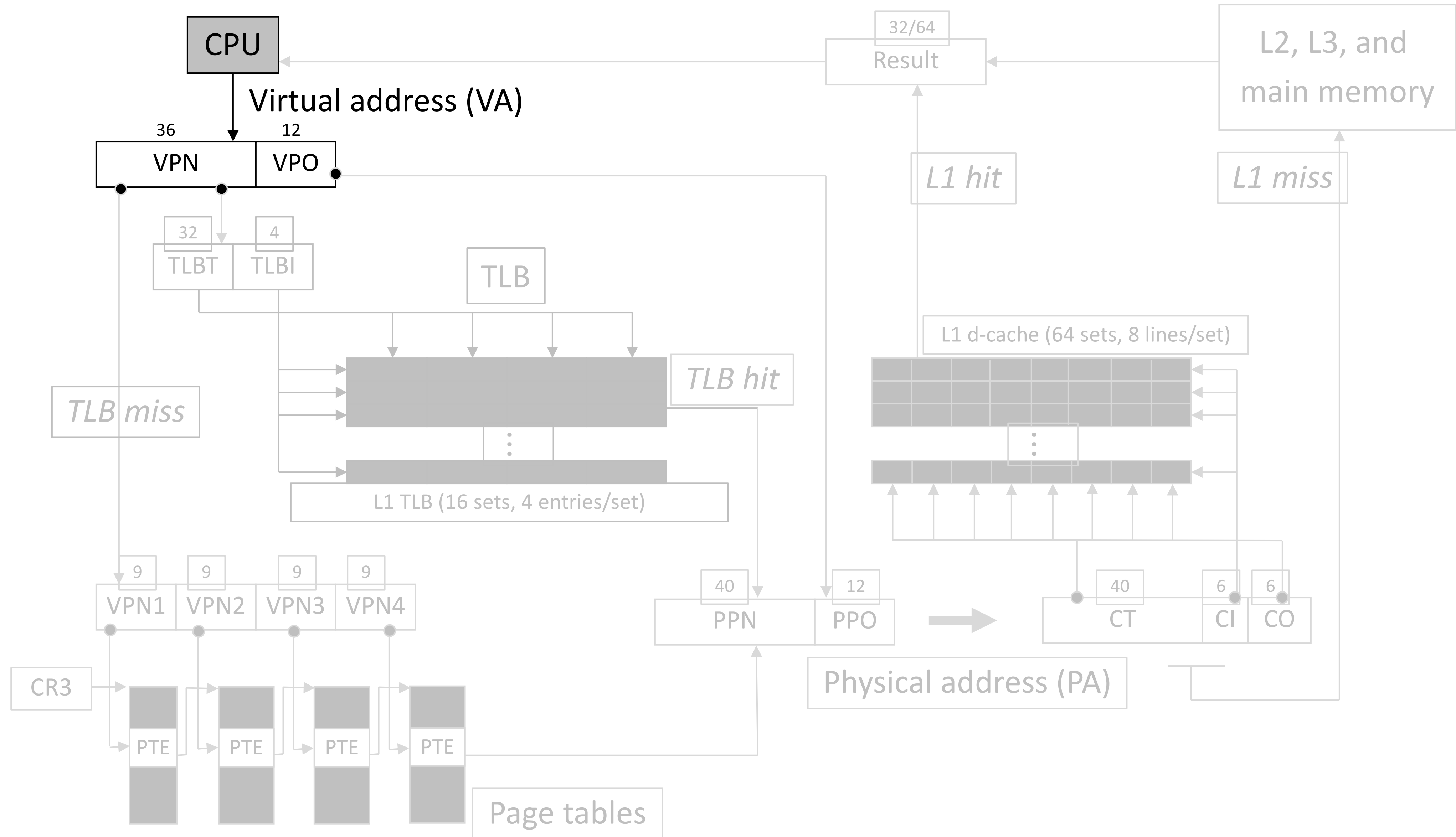


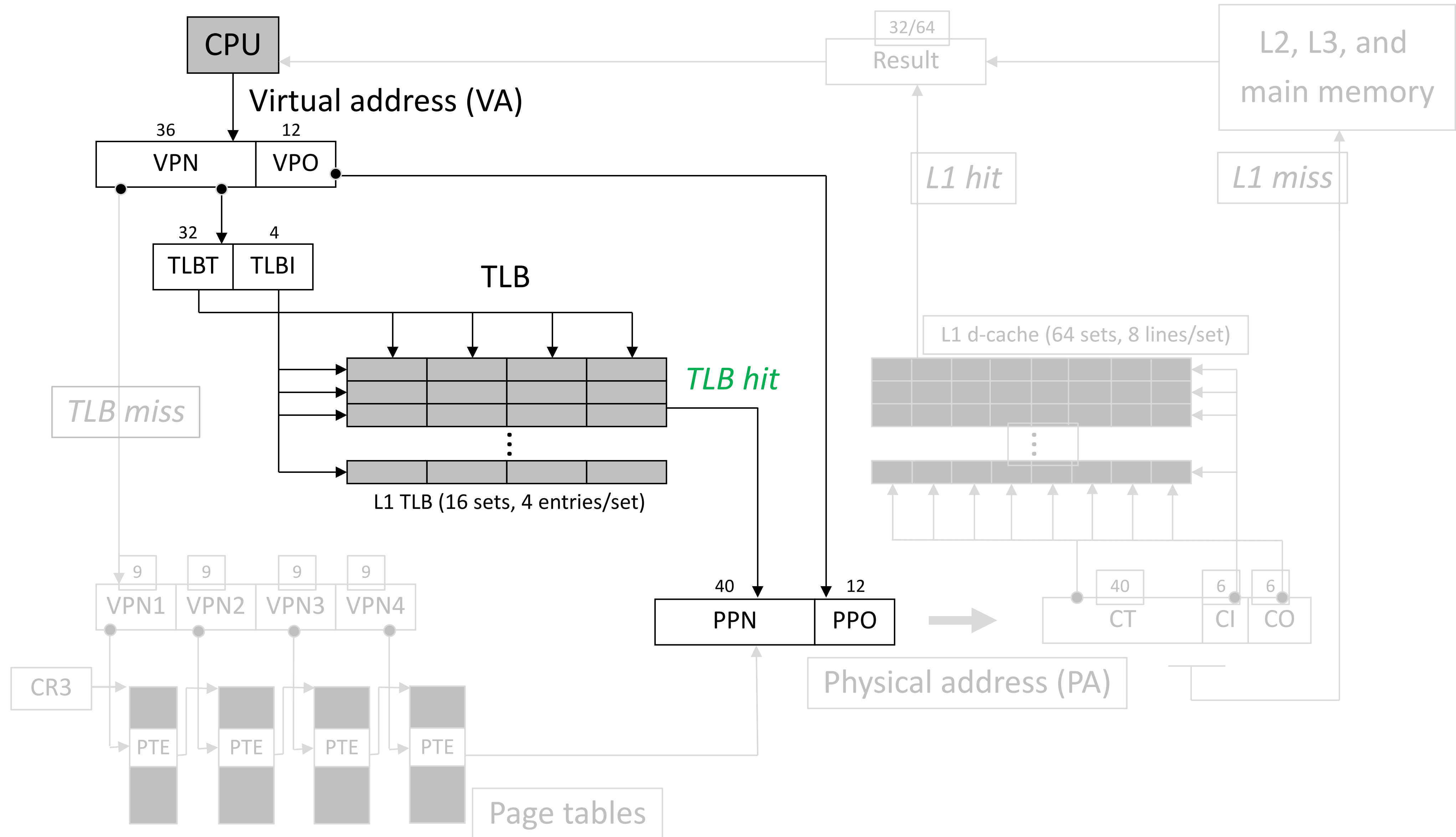
# Intel Core i7 Address Translation (Simplified)



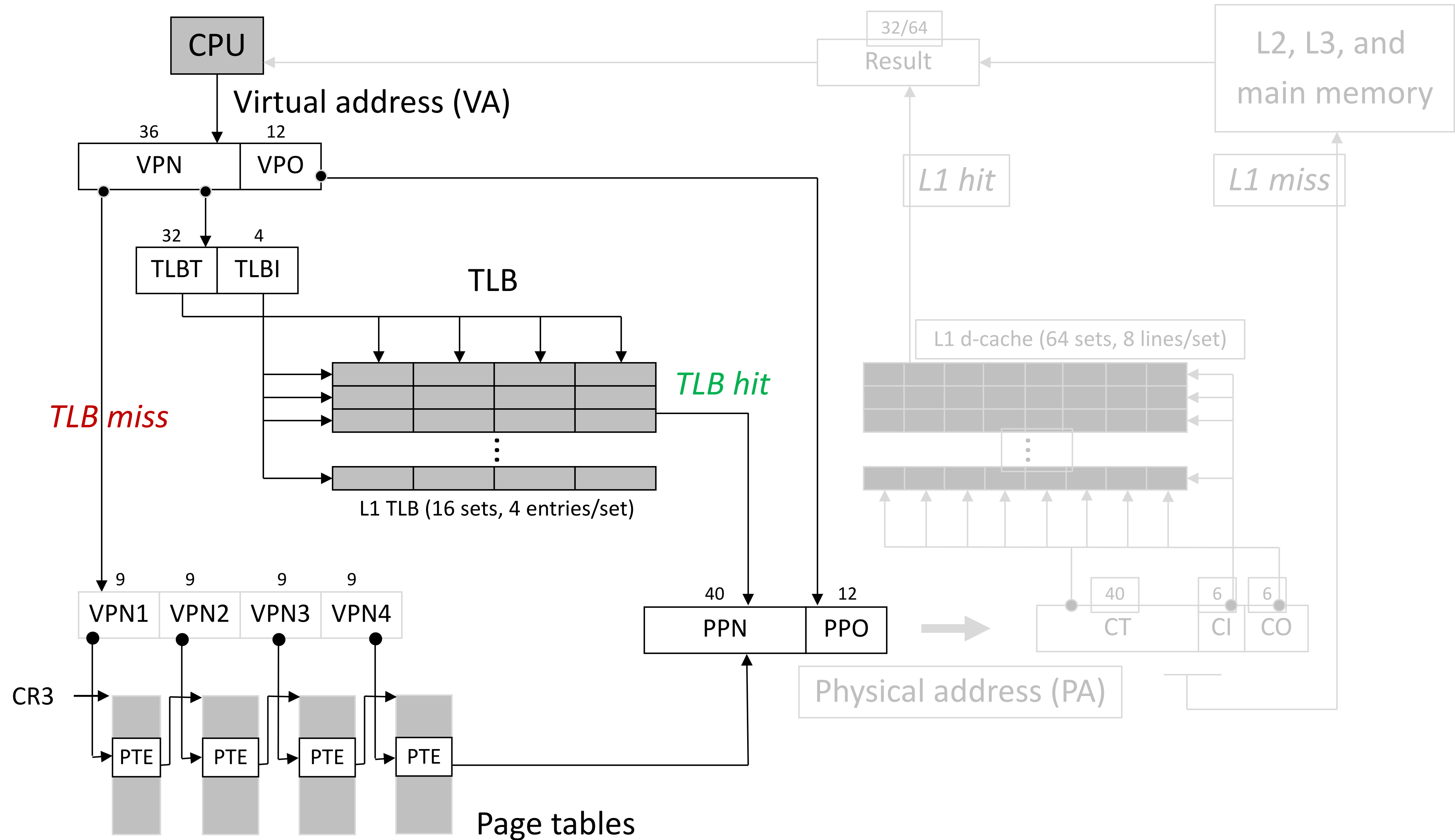

| | |
|---|---|
| VA | Virtual Address |
| VPN | Virtual Page Number |
| VPO | Virtual Page Offset |
| TLB | Translation Lookaside Buffer |
| TLBT | TLB Tag |
| TLBI | TLB Index |
| PTE | Page Table Entry |
| PPN | Physical Page Number |
| PPO | Physical Page Offset |
| CT | Cache Tag |
| CI | Cache Index |
| CO | Cache Offset |

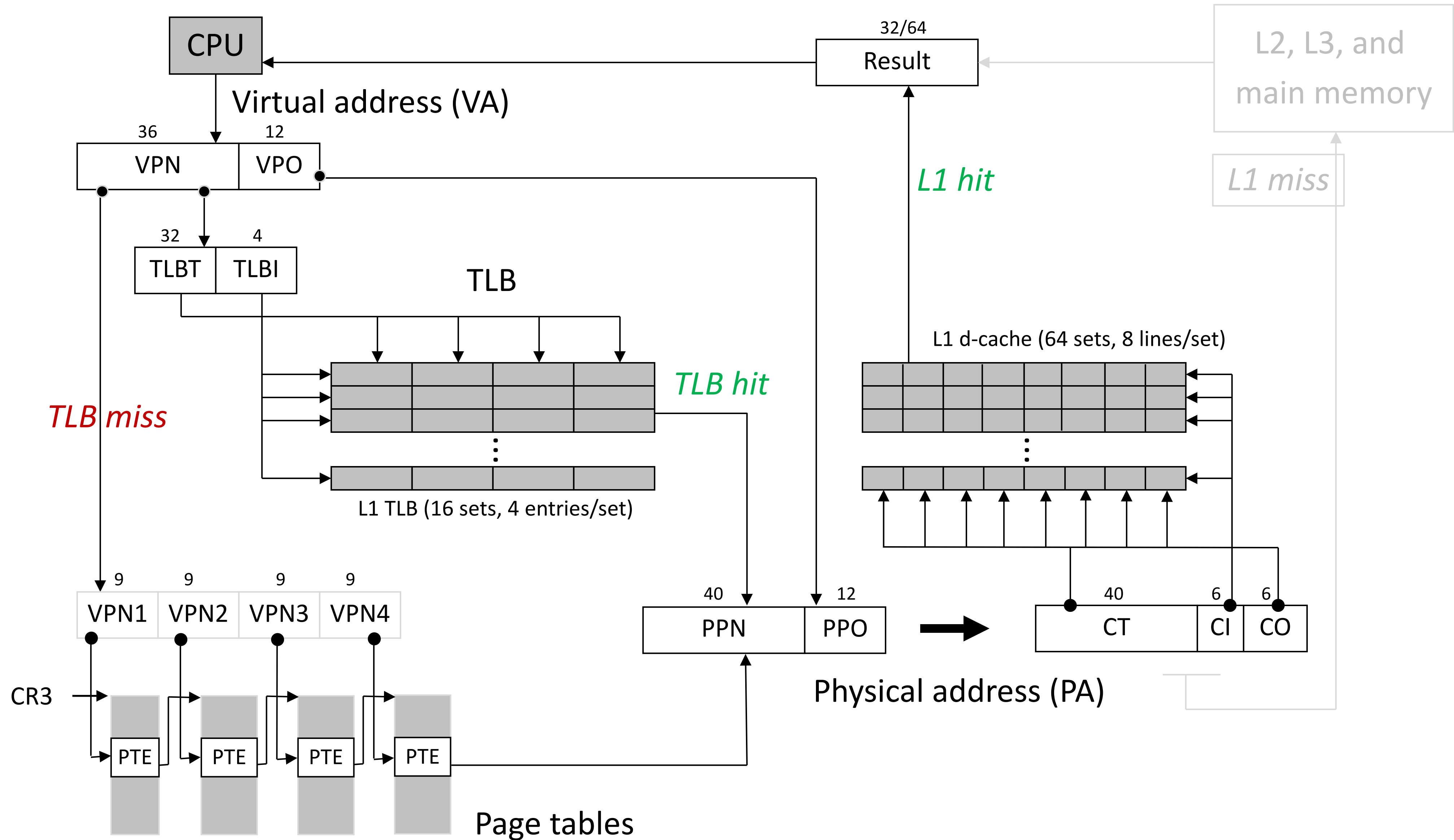

CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB hit
TLB miss
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
Physical address (PA)
40
6
6
CT
CI
CO
L1 d-cache (64 sets, 8 lines/set)
L1 hit
L1 miss
32/64
Result
L2, L3, and
main memory

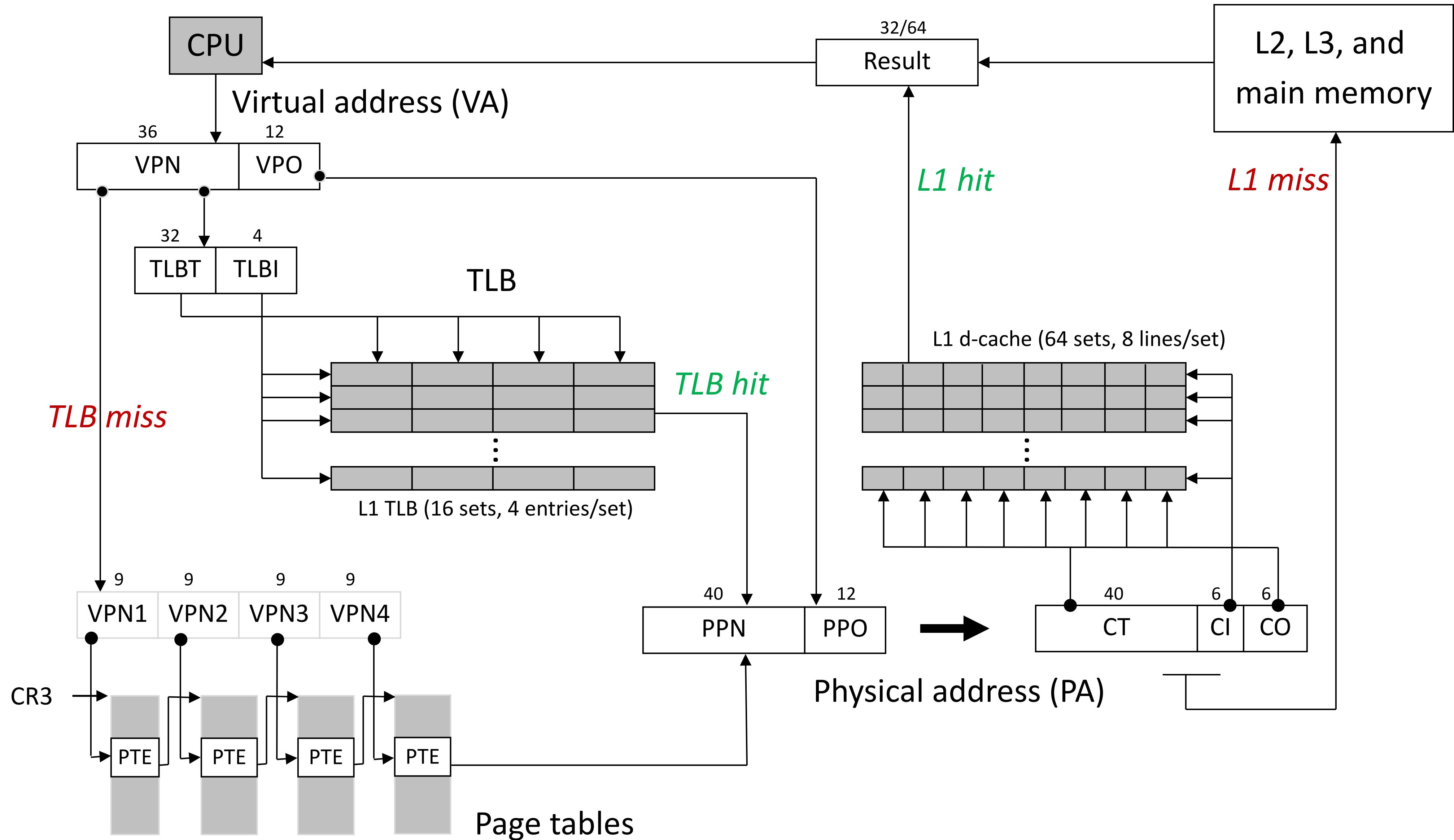

CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB hit
TLB miss
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
Physical address (PA)
40
6
6
CT
CI
CO
L1 d-cache (64 sets, 8 lines/set)
L1 hit
L1 miss
32/64
Result
L2, L3, and
main memory



CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB hit
TLB miss
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
Physical address (PA)
40
6
6
CT
CI
CO
L1 d-cache (64 sets, 8 lines/set)
32/64
Result
L1 hit
L1 miss
L2, L3, and
main memory


CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB hit
TLB miss
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
32/64
Result
L2, L3, and main memory
L1 hit
L1 miss
L1 d-cache (64 sets, 8 lines/set)
40
6
6
CT
CI
CO
Physical address (PA)


CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB miss
TLB hit
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
Physical address (PA)
40
6
6
CT
CI
CO
L1 d-cache (64 sets, 8 lines/set)
L1 hit
32/64
Result
L2, L3, and main memory
L1 miss



CPU
Virtual address (VA)
36
12
VPN
VPO
32
4
TLBT
TLBI
TLB
TLB hit
TLB miss
L1 TLB (16 sets, 4 entries/set)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
CR3
PTE
PTE
PTE
PTE
Page tables
40
12
PPN
PPO
Physical address (PA)
40
6
6
CT
CI
CO
L1 d-cache (64 sets, 8 lines/set)
L1 hit
L1 miss
32/64
Result
L2, L3, and
main memory

# Virtual Memory (Specifically Paging)

- Provides process isolation (good for security and simplicity)
- Not fast, but we have some hardware support
- Provides a mechanism for sharing memory (space efficiency)
- Good use of resources (at most we waste a bit of each page)
- Implements virtualization (we can support more processes than we have room for in physical memory.