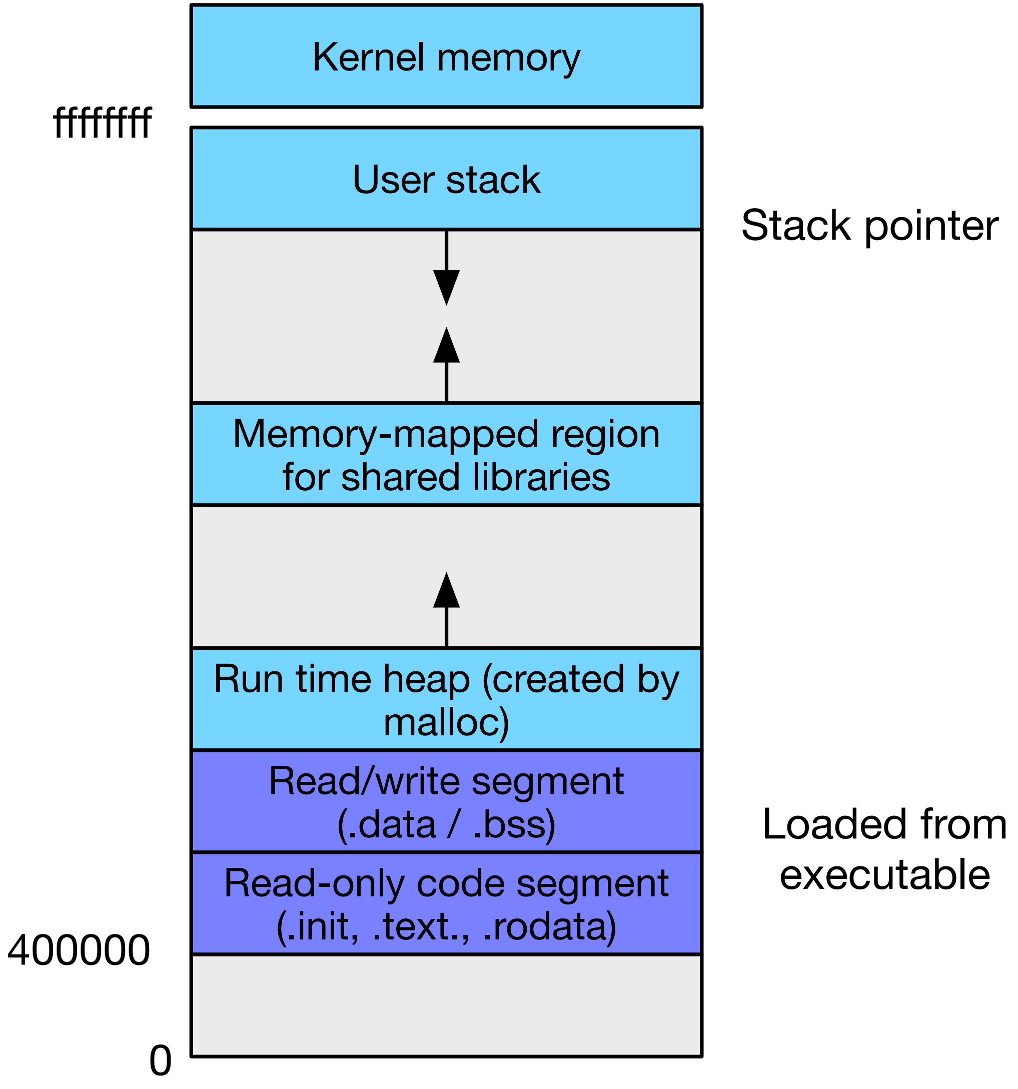
Virtual Memory

Philipp Koehn

25 April 2018

Recall: Process Address Space


Kernel memory
ffffffff
User stack
Stack pointer
Memory-mapped region for shared libraries
Run time heap (created by malloc)
Read/write segment (.data / .bss)
Loaded from executable
Read-only code segment (.init, .text., .rodata)
400000
0

Virtual Memory

- Abstraction of physical memory
- Purpose
  - appearance of more available memory than physically exists (DRAM)
  - handles disk caching / loading
  - insulates memory of each process

Virtual Memory

- Abstraction of physical memory

- Purpose
  - appearance of more available memory than physically exists (DRAM)
  - handles disk caching / loading
  - insulates memory of each process

- Page table: maps from virtual address to physical addresses

Virtual Memory

- Abstraction of physical memory

- Purpose
  - appearance of more available memory than physically exists (DRAM)
  - handles disk caching / loading
  - insulates memory of each process

- Page table: maps from virtual address to physical addresses

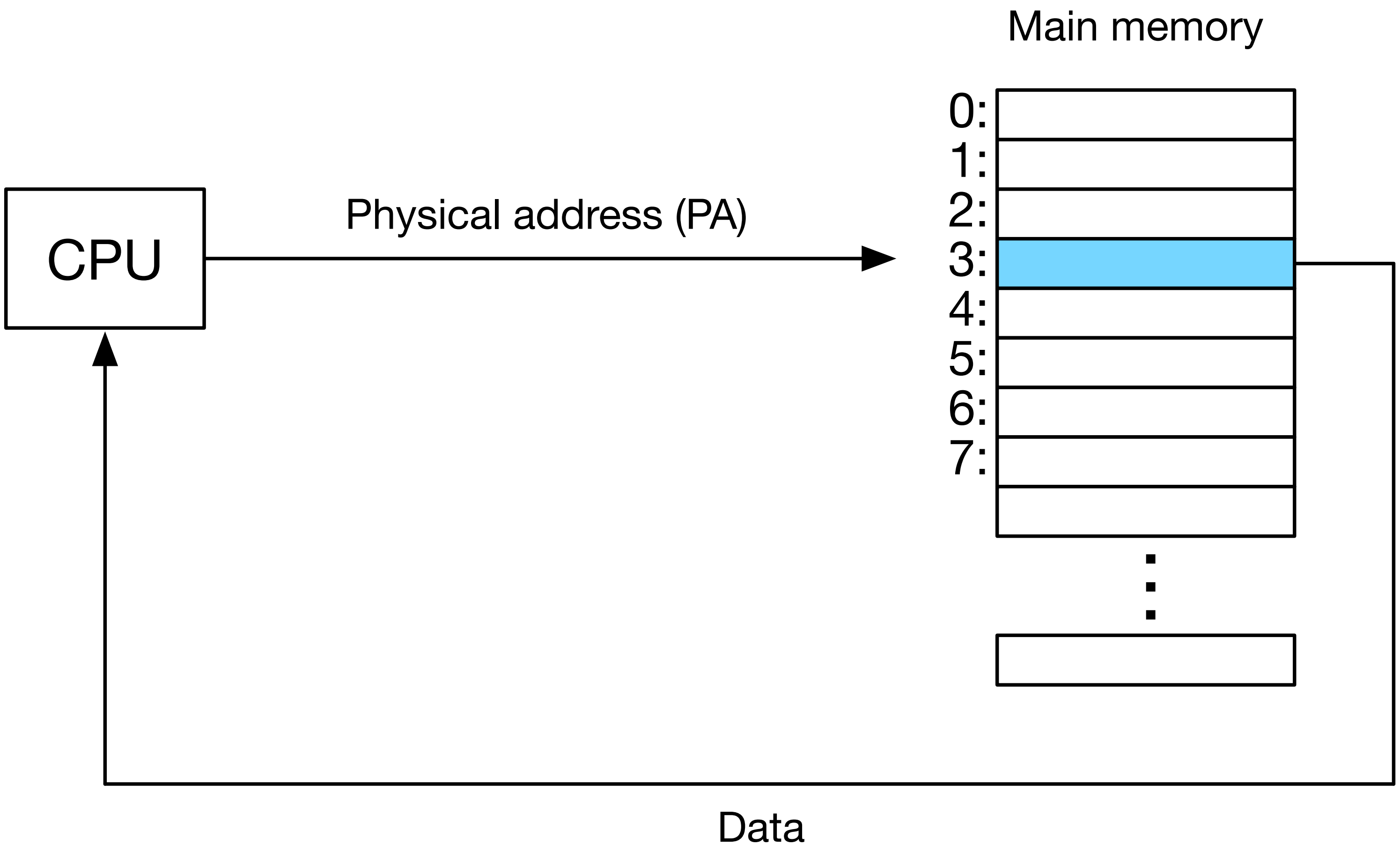
- Memory management unit (MMU):
  hardware implementation of address translation

Warning

- This is going to get very complex
- Closely tied with multi-tasking (multiple processes)
- Partly managed by hardware,
  partly managed by software

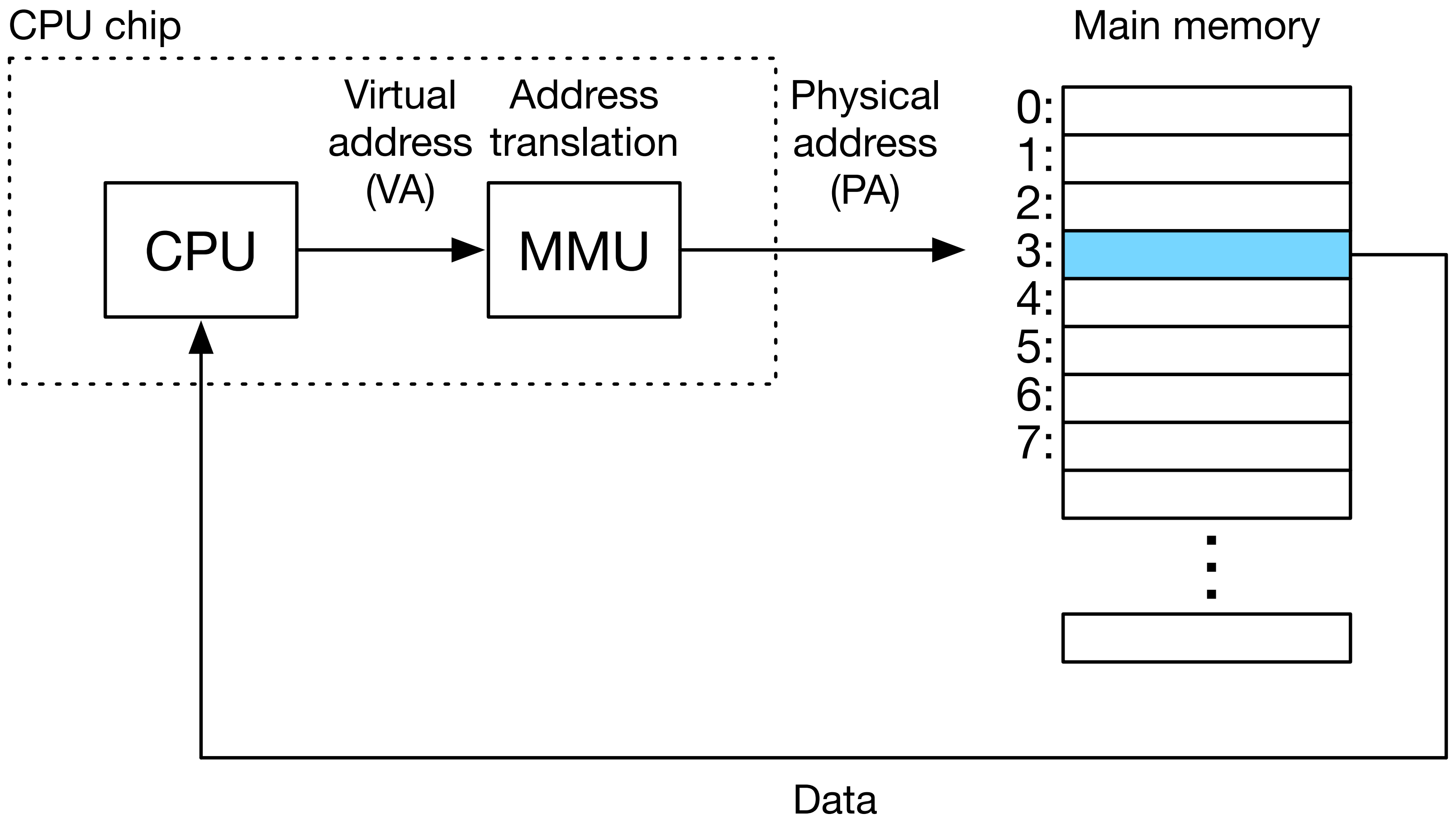
virtual addressing

Physical Addressing



- So far, assumed CPU addresses physical memory

Virtual Addressing



- Memory management unit (MMU): maps virtual to physical addresses

Address Space

- Virtual memory size: $N = 2^n$ bytes, e.g., 256TB

- Physical memory size: $M = 2^m$ bytes, e.g., 16GB

- Page (block of memory): $P = 2^p$ bytes, e.g., 4KB

- A virtual address can be encoded in n bits

caching

Caching... Again?

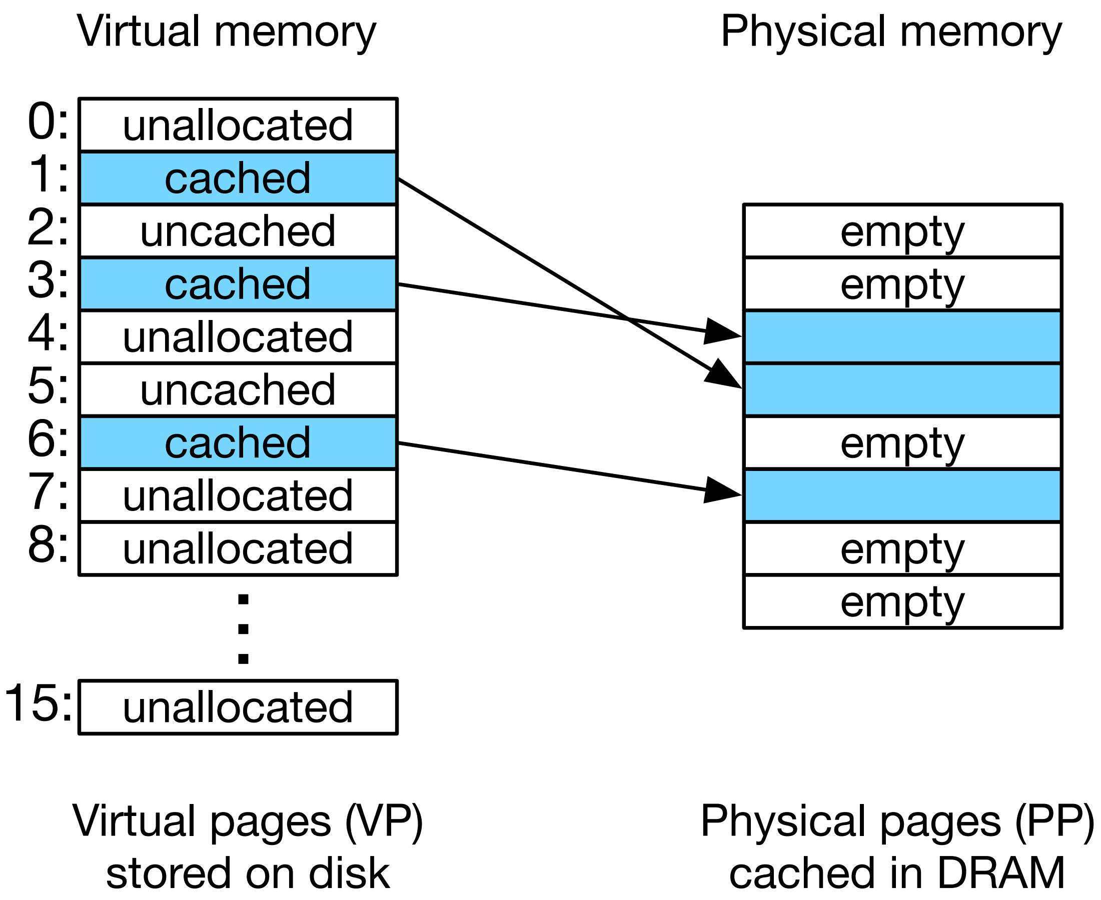
- Yes, we already discussed caching

  but: for on-chip cache of DRAM memory

- Now
  - caching between RAM and disk
  - driven by a large virtual memory address space
  - to avoid unnecessary and duplicate loading

- Jargon
  - previously "block", now "page"
  - now: "swapping" or "paging"

Mapping


Virtual memory
Physical memory
0: unallocated
1: cached
2: uncached
3: cached
4: unallocated
5: uncached
6: cached
7: unallocated
8: unallocated
15: unallocated
empty
empty
empty
empty
empty
Virtual pages (VP)
stored on disk
Physical pages (PP)
cached in DRAM

State of Virtual Memory Page

- Cached
  - allocated page
  - stored in physical memory

State of Virtual Memory Page

- Cached
  - allocated page
  - stored in physical memory

- Uncached
  - allocated page
  - not in physical memory

State of Virtual Memory Page

- Cached
  - allocated page
  - stored in physical memory

- Uncached
  - allocated page
  - not in physical memory

- Unallocated
  - not used by virtual memory system so far

Page Table

- Array of page table entries (PTE)

Page Table

- Array of page table entries (PTE)

- Valid bit
  - set if PTE currently maps to physical address (cached)
  - not set otherwise (uncached or unallocated)

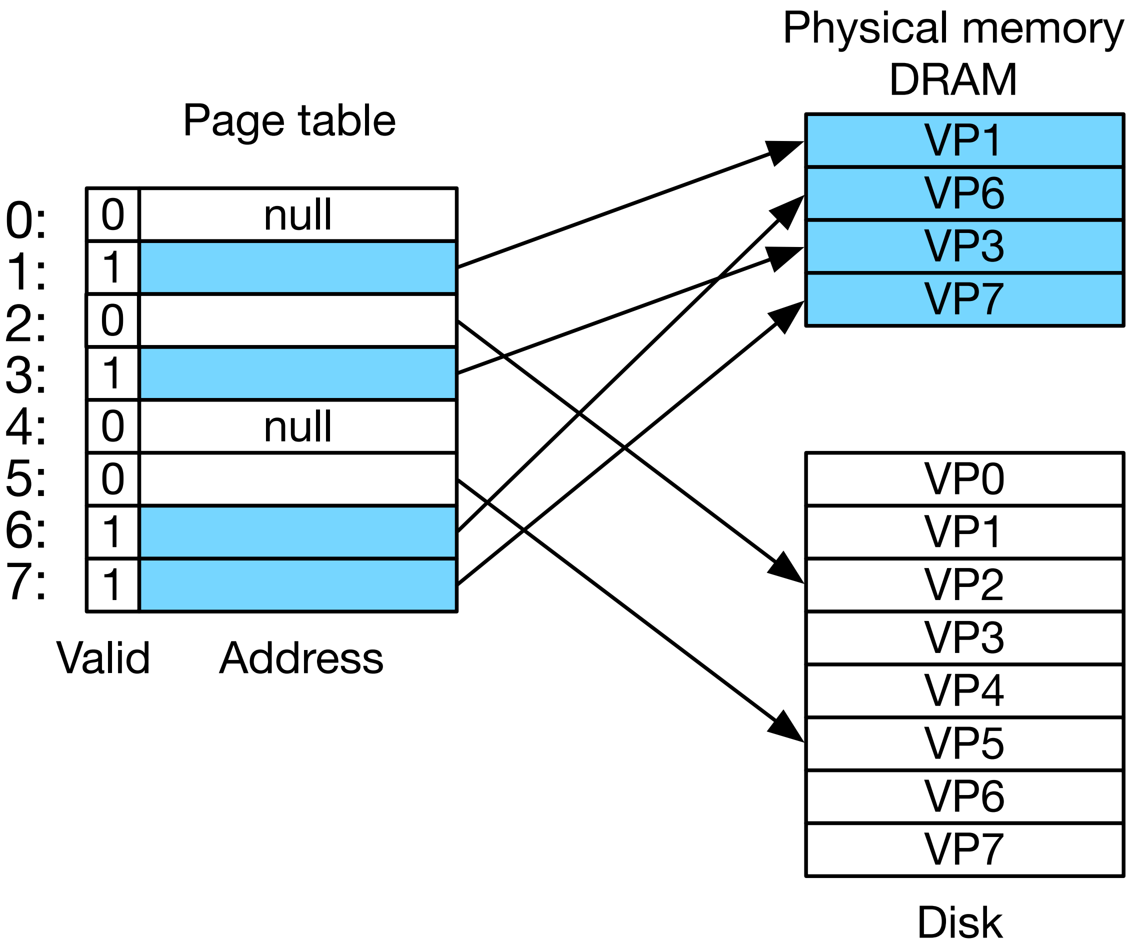
Page Table

- Array of page table entries (PTE)

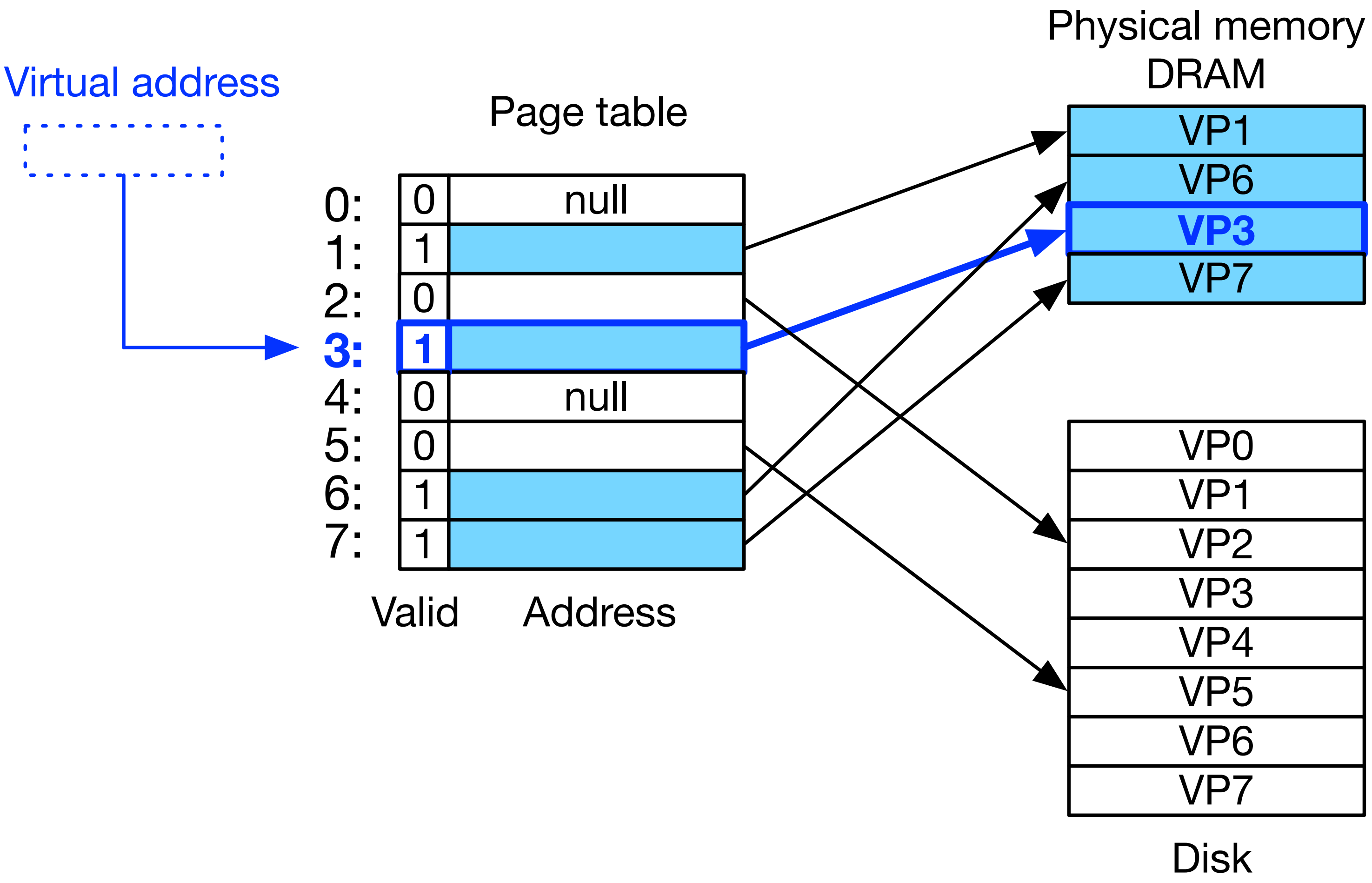
- Valid bit
  - set if PTE currently maps to physical address (cached)
  - not set otherwise (uncached or unallocated)

- Mapped address
  - if cached: physical address in DRAM
  - if not cached: physical address on disk

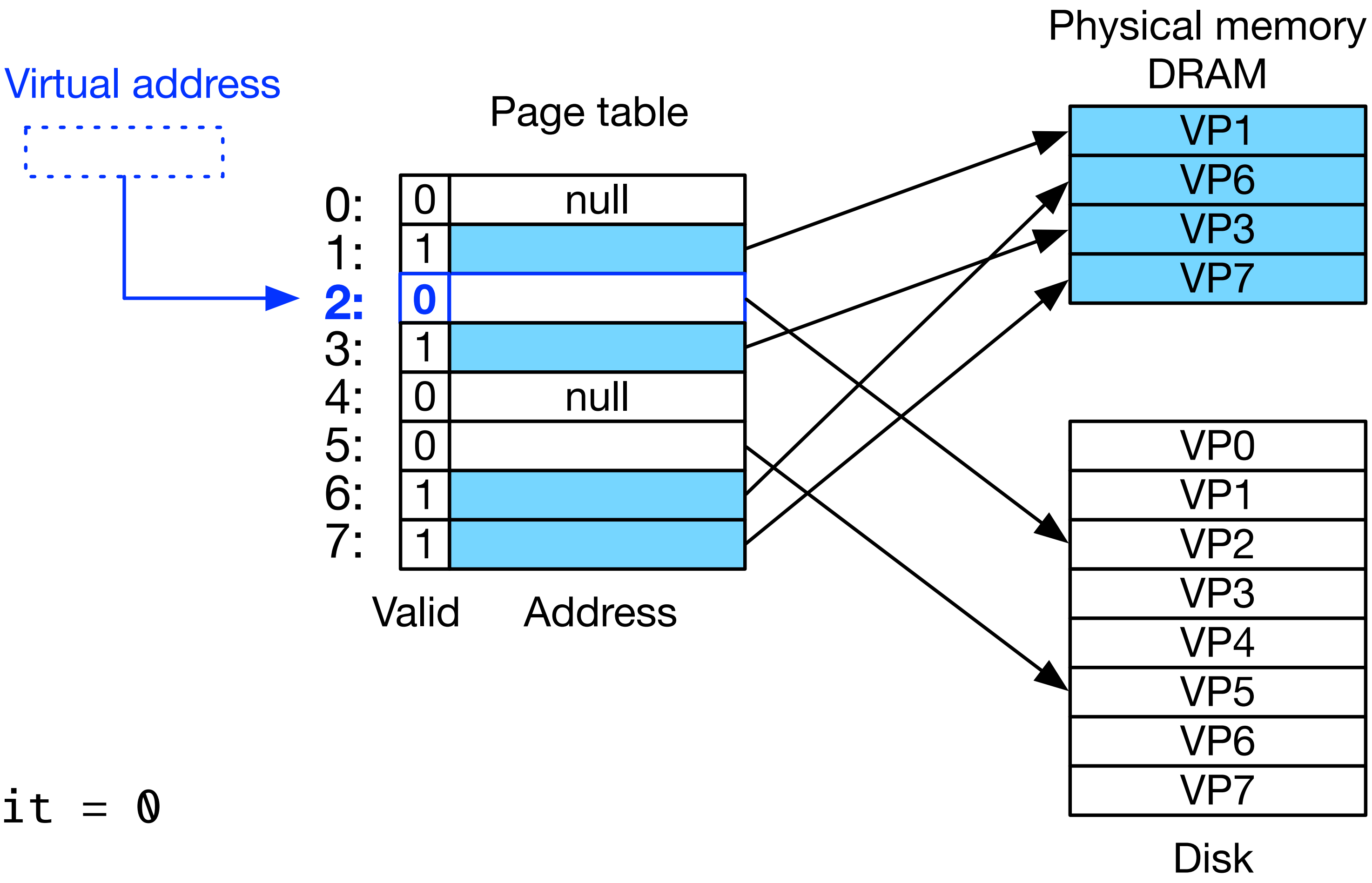
Page Table


Physical memory
DRAM
VP1
VP6
VP3
VP7
Page table
0:
1:
2:
3:
4:
5:
6:
7:
0
1
0
1
0
0
1
1
null
null
Valid
Address
VP0
VP1
VP2
VP3
VP4
VP5
VP6
VP7
Disk

Page Hit

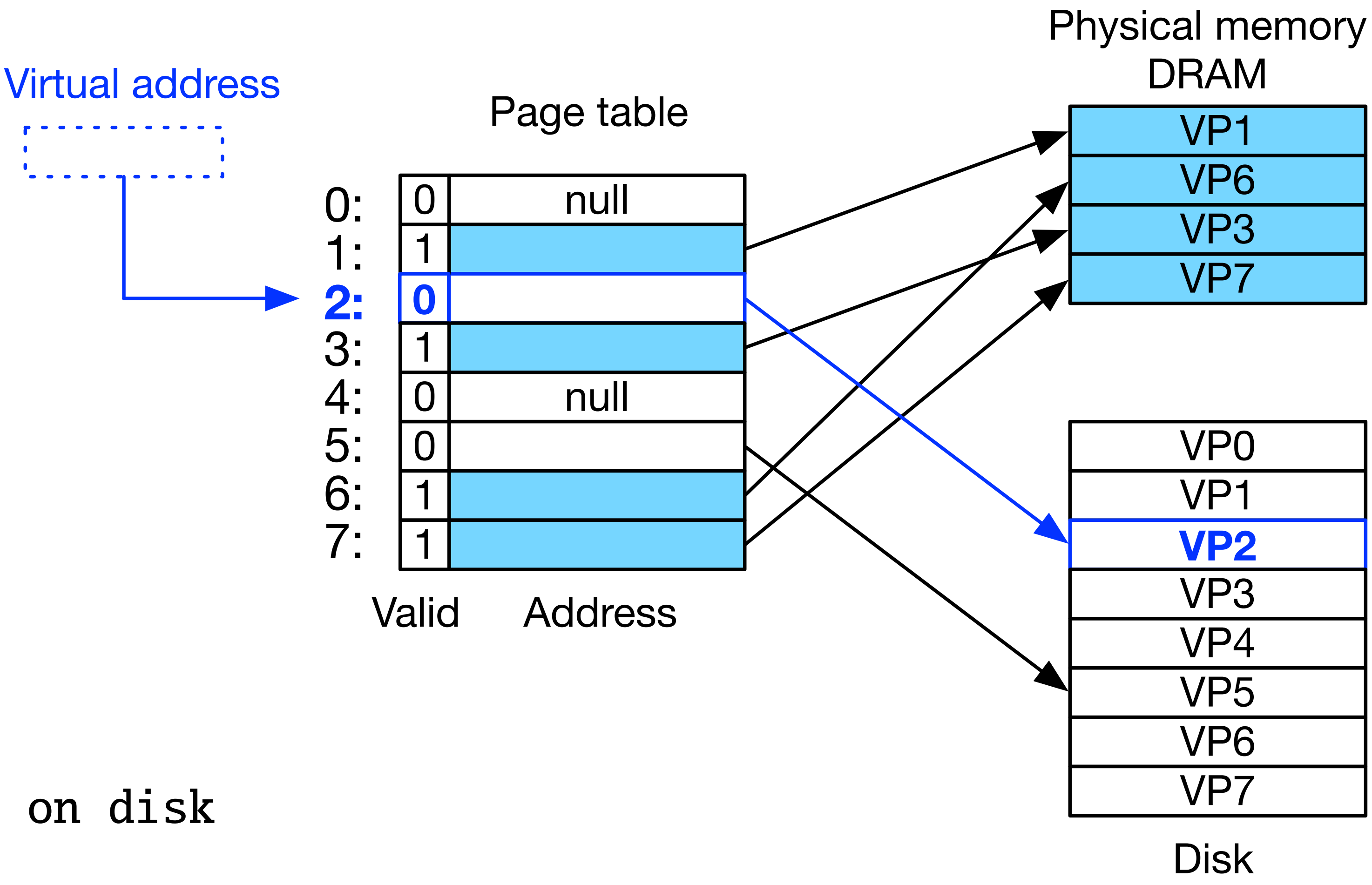

Virtual address
Page table
Physical memory
DRAM
VP1
VP6
VP3
VP7
0:
1:
2:
3:
4:
5:
6:
7:
0
null
1
0
1
0
null
0
1
1
Valid
Address
VP0
VP1
VP2
VP3
VP4
VP5
VP6
VP7
Disk

Page Fault



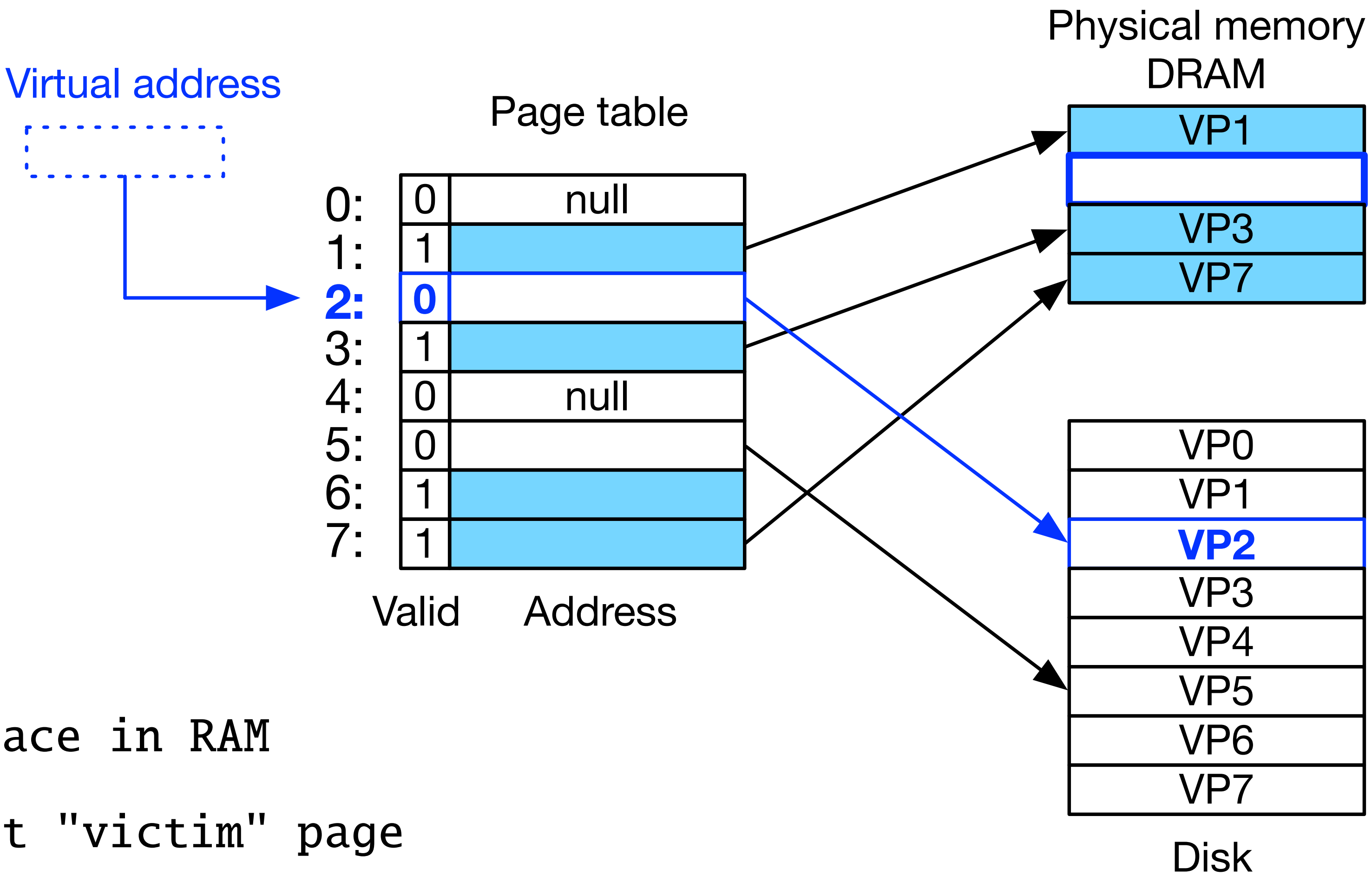
- Valid bit = 0
- Page not in RAM

Page Fault



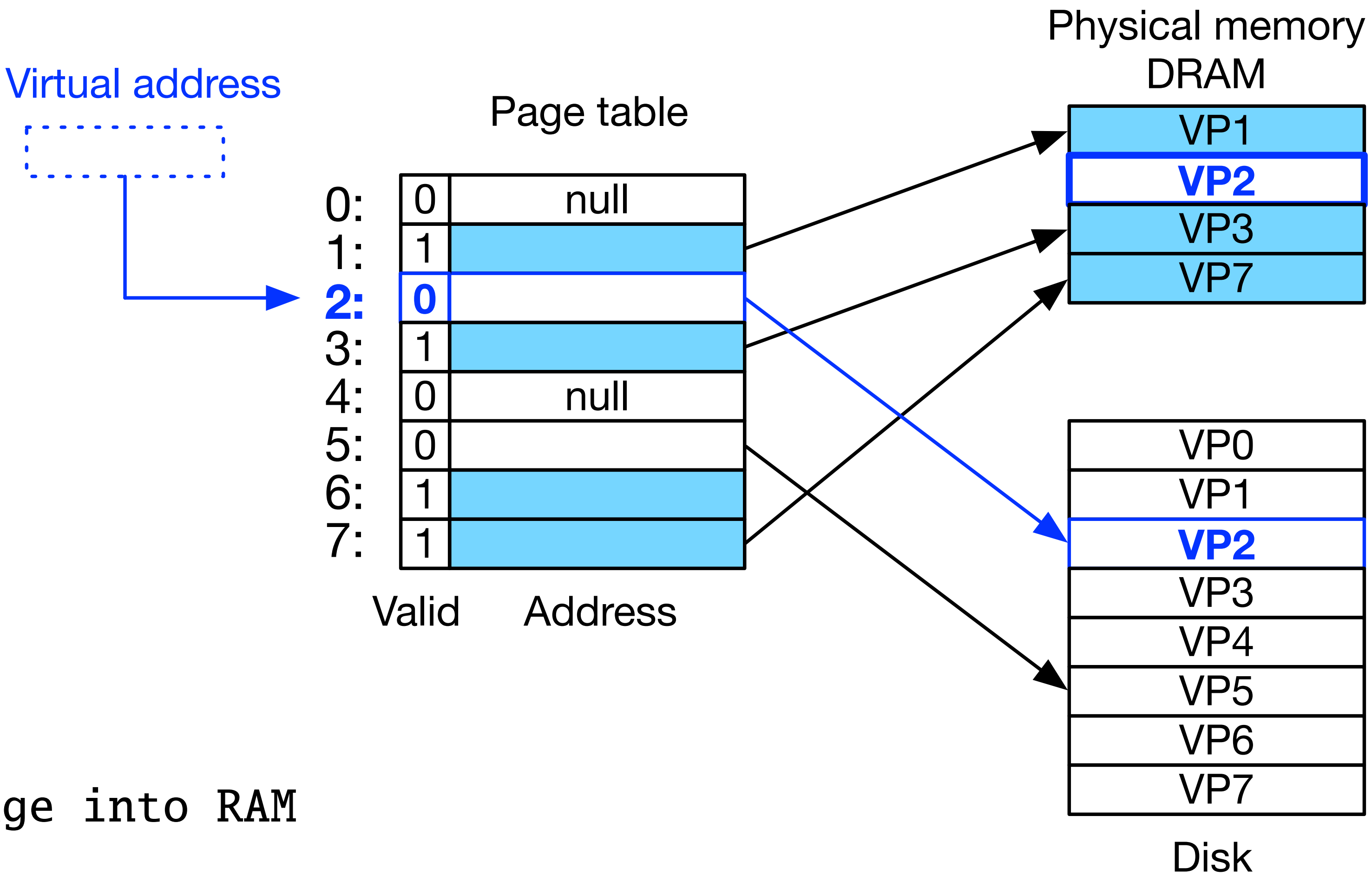
- Page is on disk

Page Fault



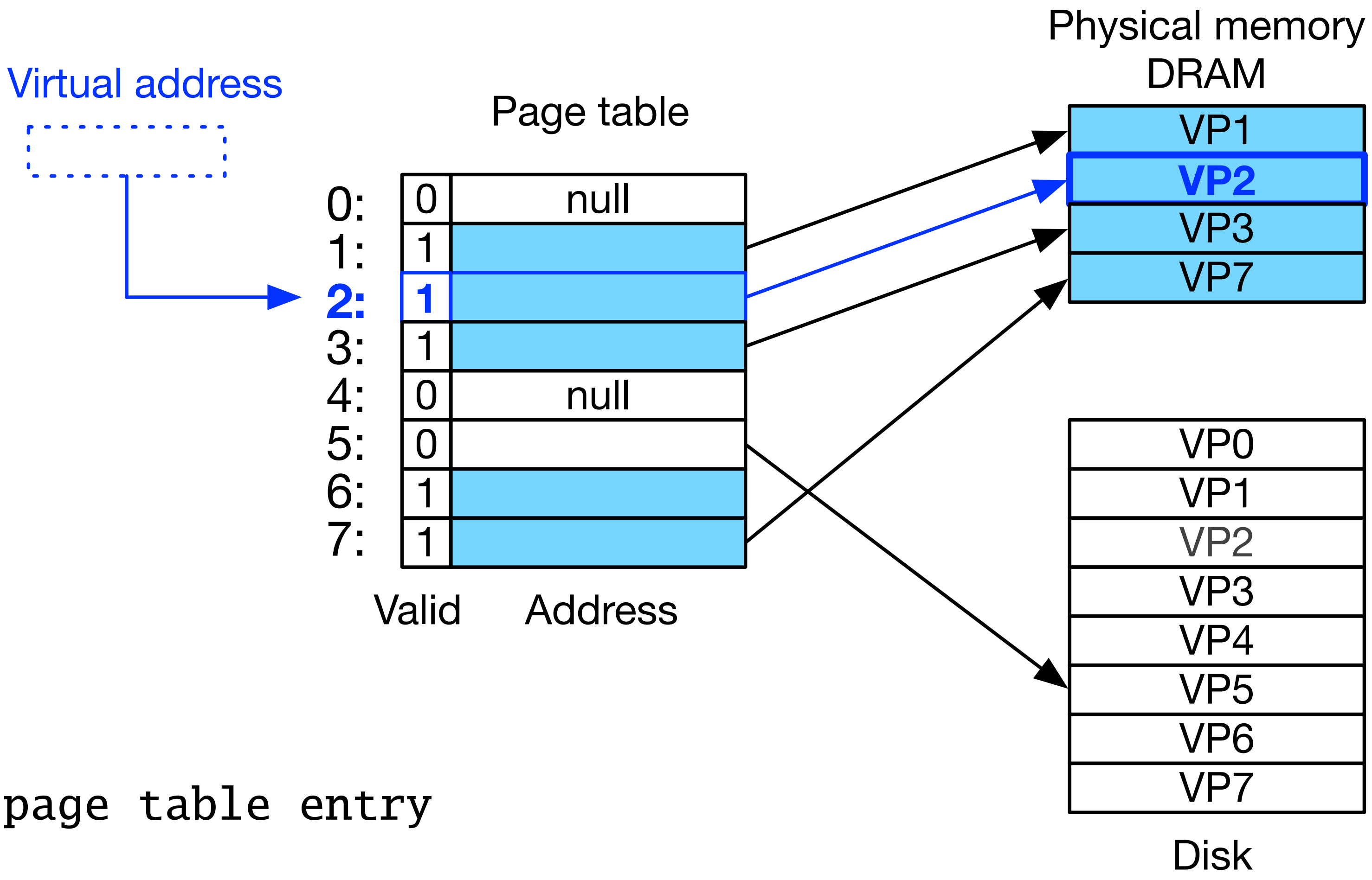
- Make space in RAM
- Pre-empt "victim" page
- Typically out-dated cached page

Page Fault



- Load page into RAM

Page Fault



- Update page table entry

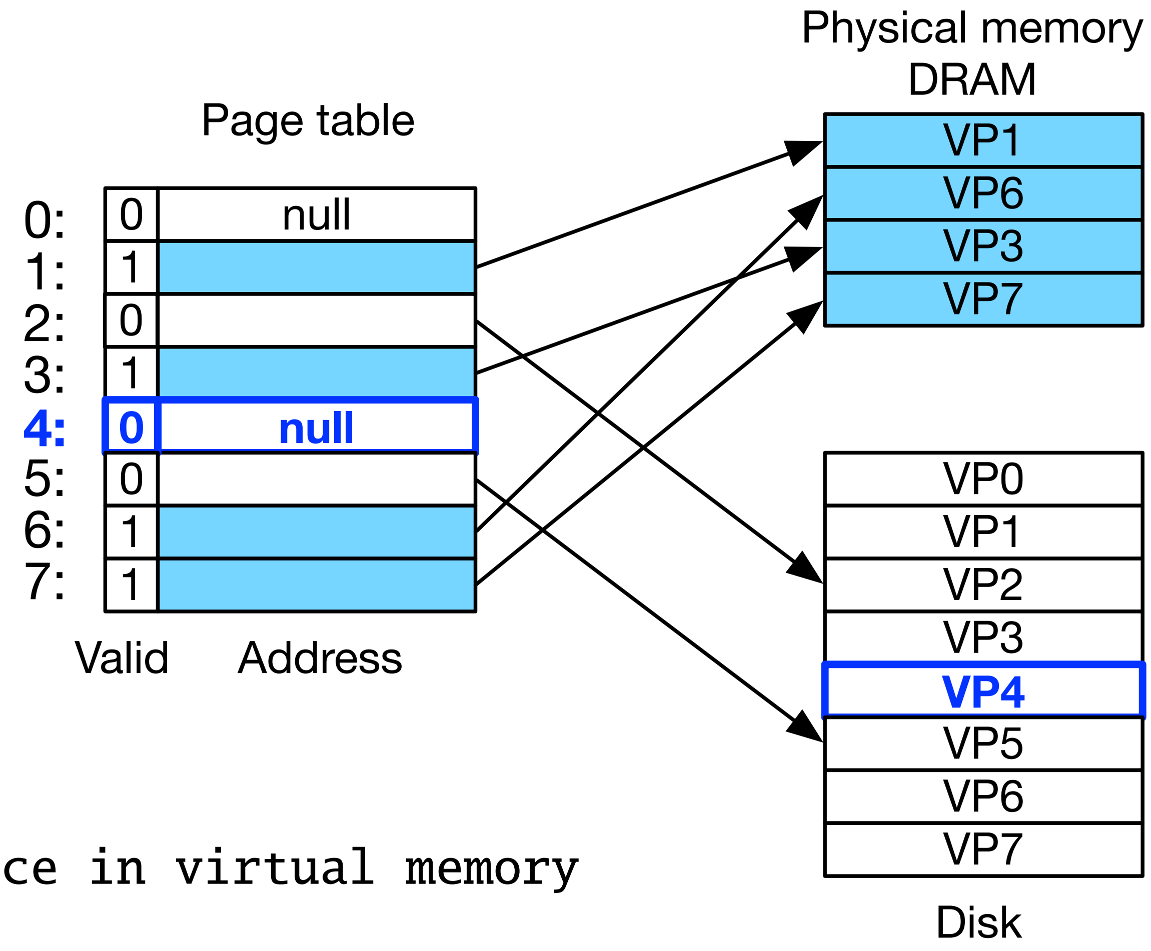
Allocating Pages

- What happens when we load a program?

- We need to load its executable into memory

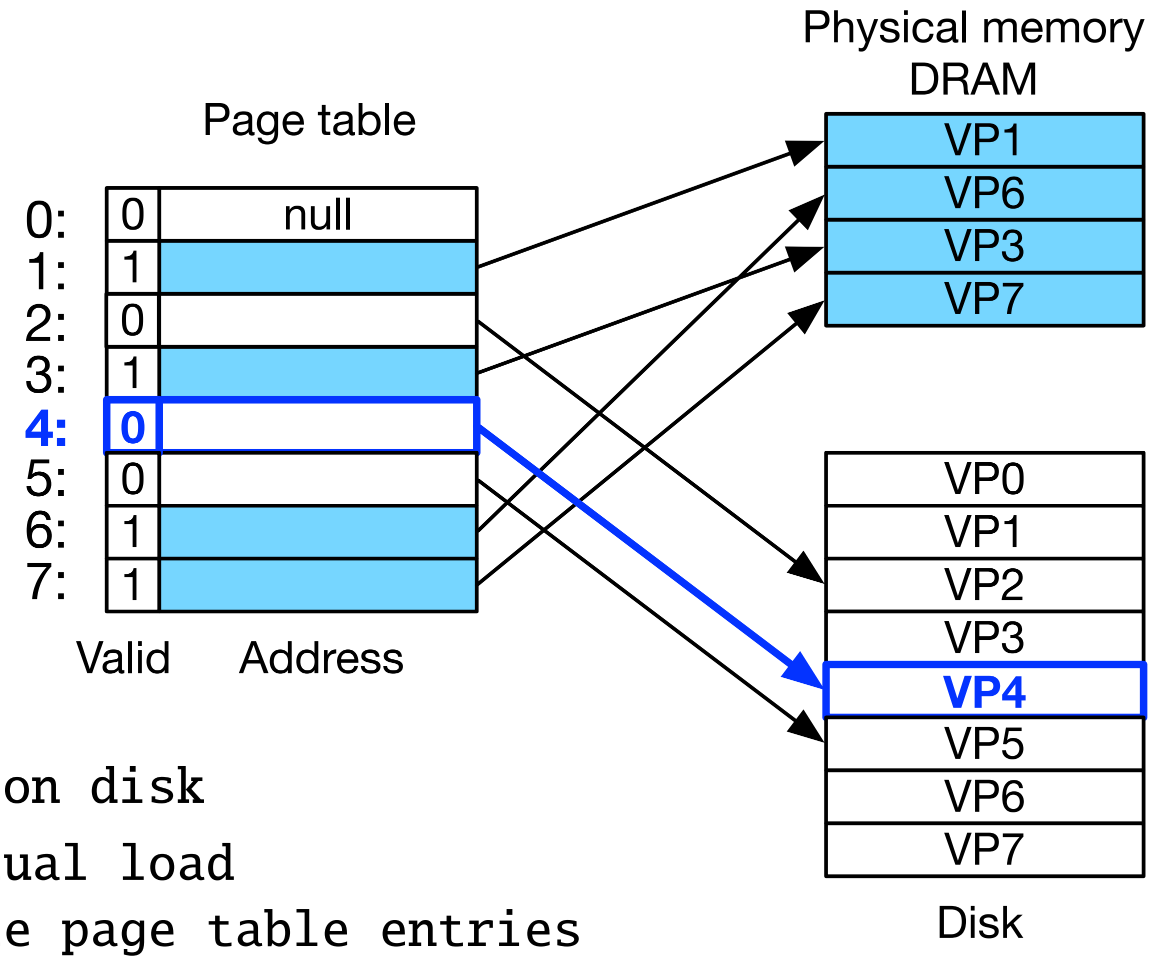
- Similar: create data objects when program is running ("allocating" memory)

Allocating Page



- Identify space in virtual memory

Allocating Page



- Map to data on disk
  - do not actual load
  - just create page table entries
  - let virtual memory system handle loading

⇒ On-demand loading

Process Memory

- Nothing loaded at startup

Process Memory

- Nothing loaded at startup

- Working set (or resident set)
  - pages of a process that are currently in DRAM
  - loaded by virtual memory system on demand

Process Memory

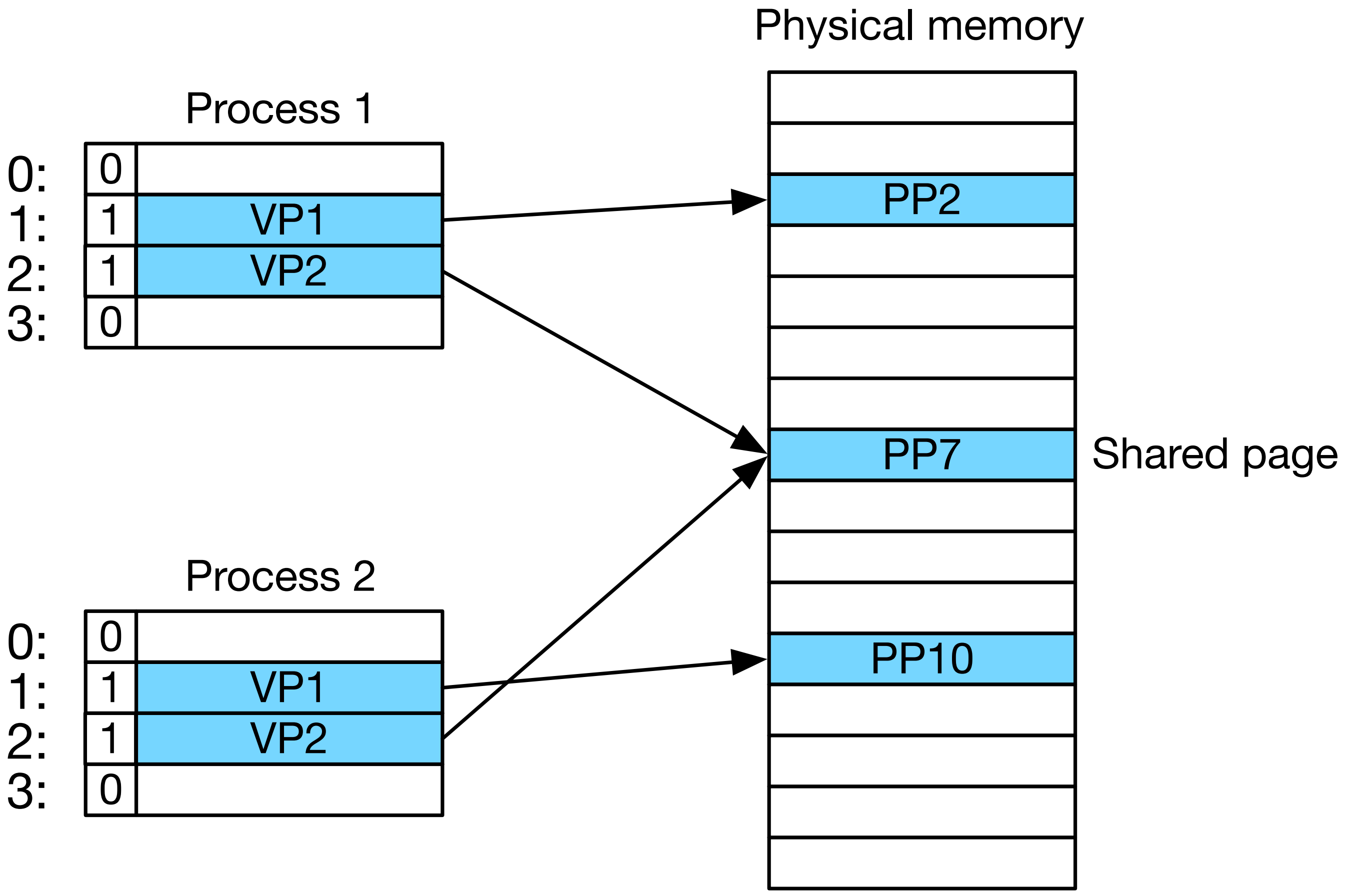
- Nothing loaded at startup

- Working set (or resident set)
  - pages of a process that are currently in DRAM
  - loaded by virtual memory system on demand

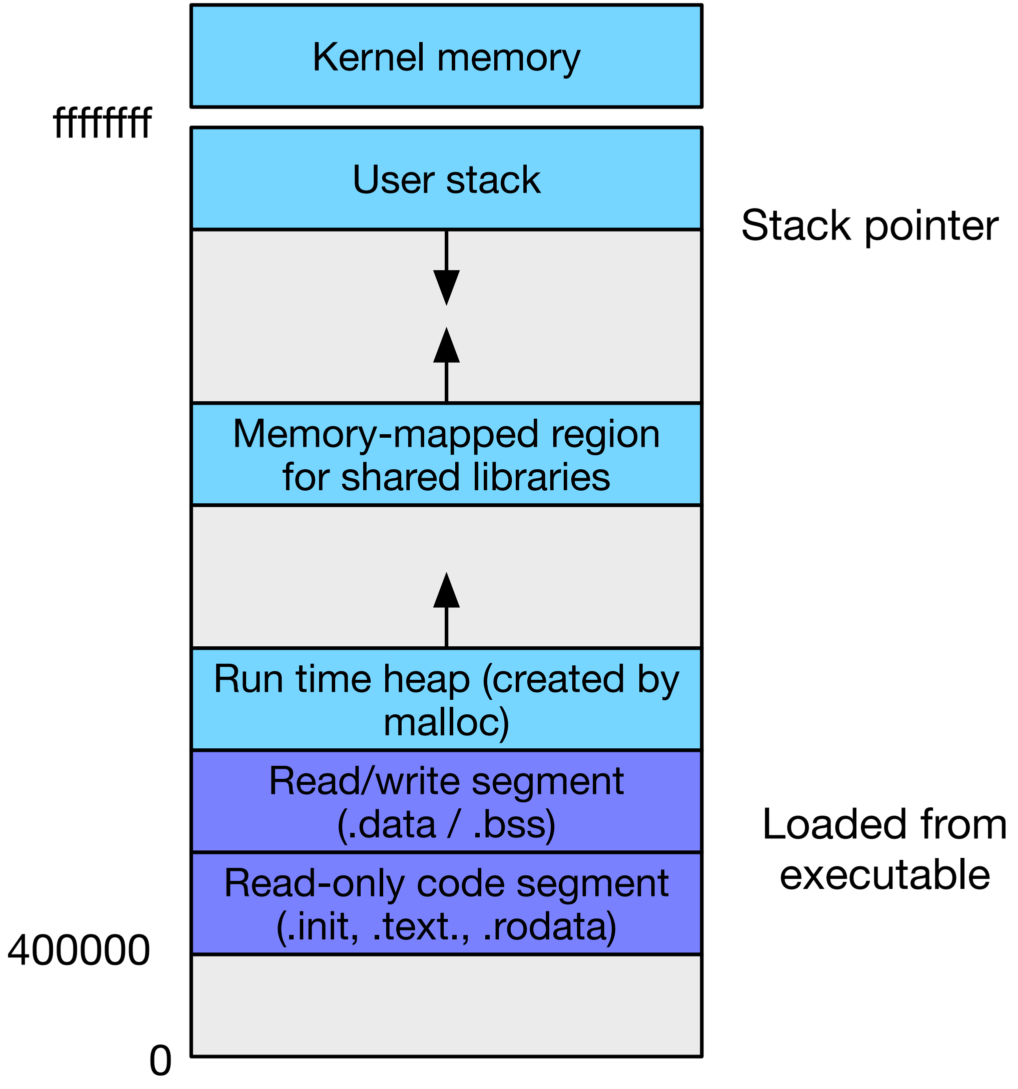
- Thrashing
  - memory actively required by all processes larger than physically available
  - frequent swapping of memory to/from disk
  - very bad: slows down machine dramatically

memory management

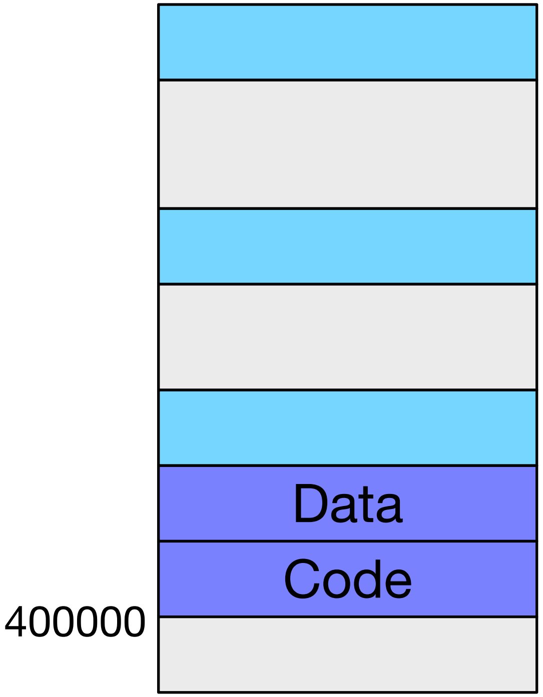
One Page Table per Process



Process Address Space


Kernel memory
ffffffff
User stack
Stack pointer
Memory-mapped region for shared libraries
Run time heap (created by malloc)
Read/write segment (.data / .bss)
Loaded from executable
Read-only code segment (.init, .text., .rodata)
400000
0

Simplified Linking



- Each process has its code in address 0x400000
- Easy linking: Linker can establish fixed addresses

Simplified Loading

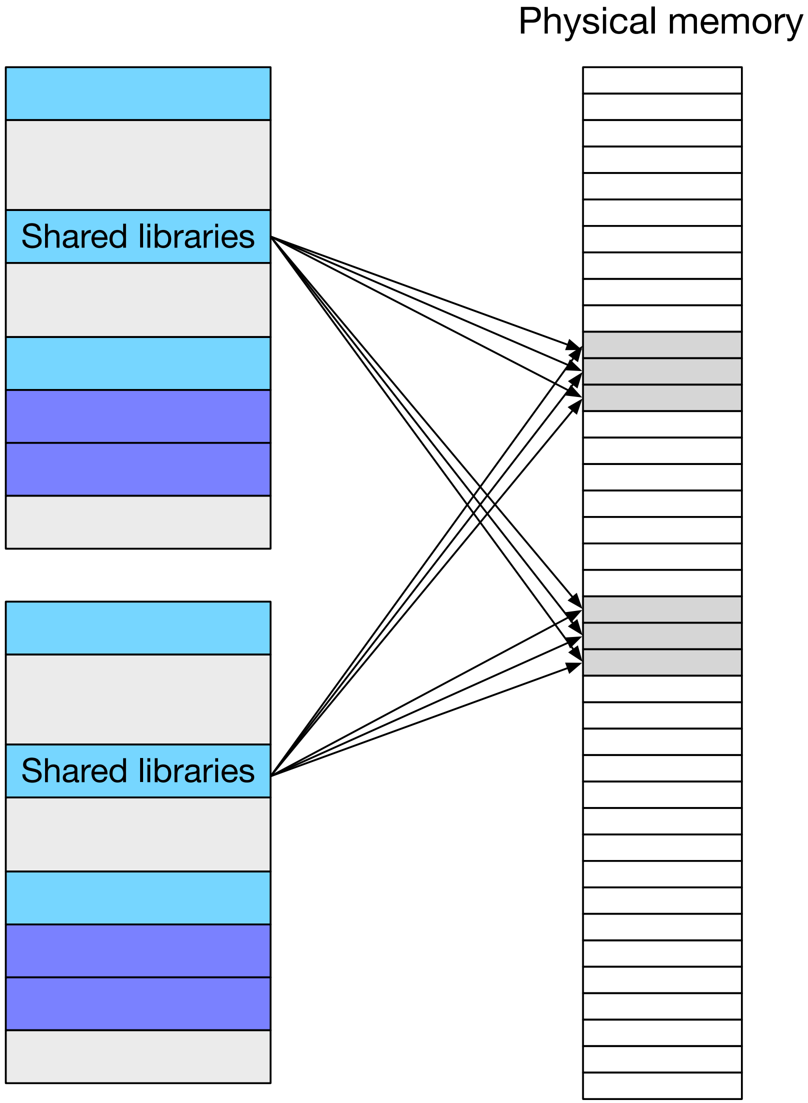
- When loading process into memory...
- Enter .data and .text section into page table
- Mark them as invalid (= not actually in RAM)
- Called memory mapping (more on that later)

Simplified Sharing

- Shared libraries
  used by several processes

  e.g., stdio providing printf,
  scanf, open, close, ...

- Not copied multiple times
  into RAM



Simplified Memory Allocation

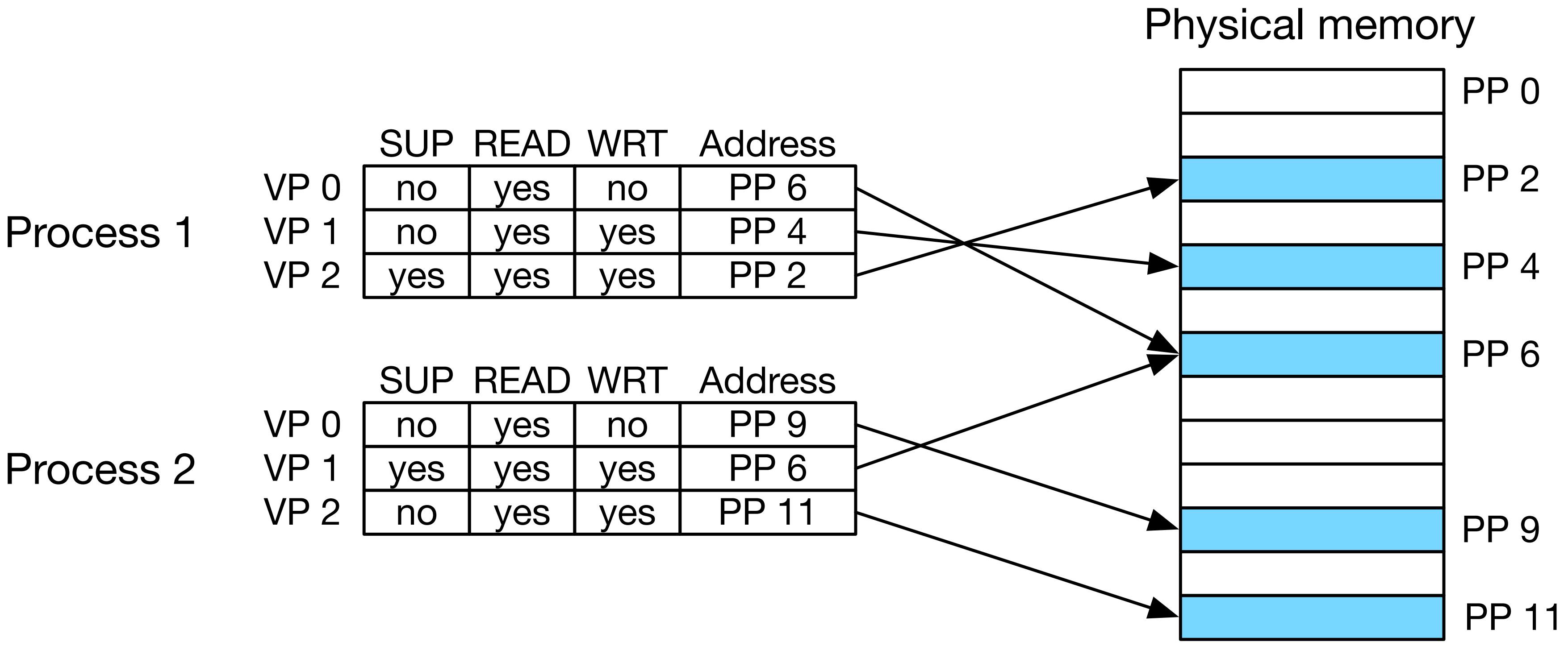
- Process may need more memory (e.g., malloc call)

⇒ New entry in page table

- Mapped to arbitrary pages in physical memory

- Do not have to be contiguous

Memory Protection



Process 1

	SUP	READ	WRT	Address
VP 0	no	yes	no	PP 6
VP 1	no	yes	yes	PP 4
VP 2	yes	yes	yes	PP 2

Process 2

	SUP	READ	WRT	Address
VP 0	no	yes	no	PP 9
VP 1	yes	yes	yes	PP 6
VP 2	no	yes	yes	PP 11

- Page may be kernel only: SUP=yes
- Page may be read-only (e.g., code)

address translation

Address Space

- Virtual memory size: N = 2^n bytes
- Physical memory size: M = 2^m bytes
- Page (block of memory): P = 2^p bytes
- A virtual address can be encoded in n bits

Address Translation

- Task: mapping virtual address to physical address
  - virtual address (VA): used by machine code instructions
  - physical address (PA): location in RAM

Address Translation

- Task: mapping virtual address to physical address
  - virtual address (VA): used by machine code instructions
  - physical address (PA): location in RAM
- Formally

$$\text{MAP: VA} \rightarrow \text{PA} \cup \emptyset$$

where:

$$\begin{aligned}\text{MAP(A)} &= \text{PA if in RAM}\\ &= \emptyset \text{ otherwise}\end{aligned}$$

Address Translation

- Task: mapping virtual address to physical address
  - virtual address (VA): used by machine code instructions
  - physical address (PA): location in RAM
- Formally

$$\text{MAP: VA} \rightarrow \text{PA} \cup \emptyset$$

  where:

$$\begin{aligned}\text{MAP(A)} &= \text{PA if in RAM}\\ &= \emptyset \text{ otherwise}\end{aligned}$$

- Note: this happens very frequently in machine code

Address Translation

- Task: mapping virtual address to physical address
  - virtual address (VA): used by machine code instructions
  - physical address (PA): location in RAM

- Formally

$$\text{MAP: VA} \rightarrow \text{PA} \cup 0$$

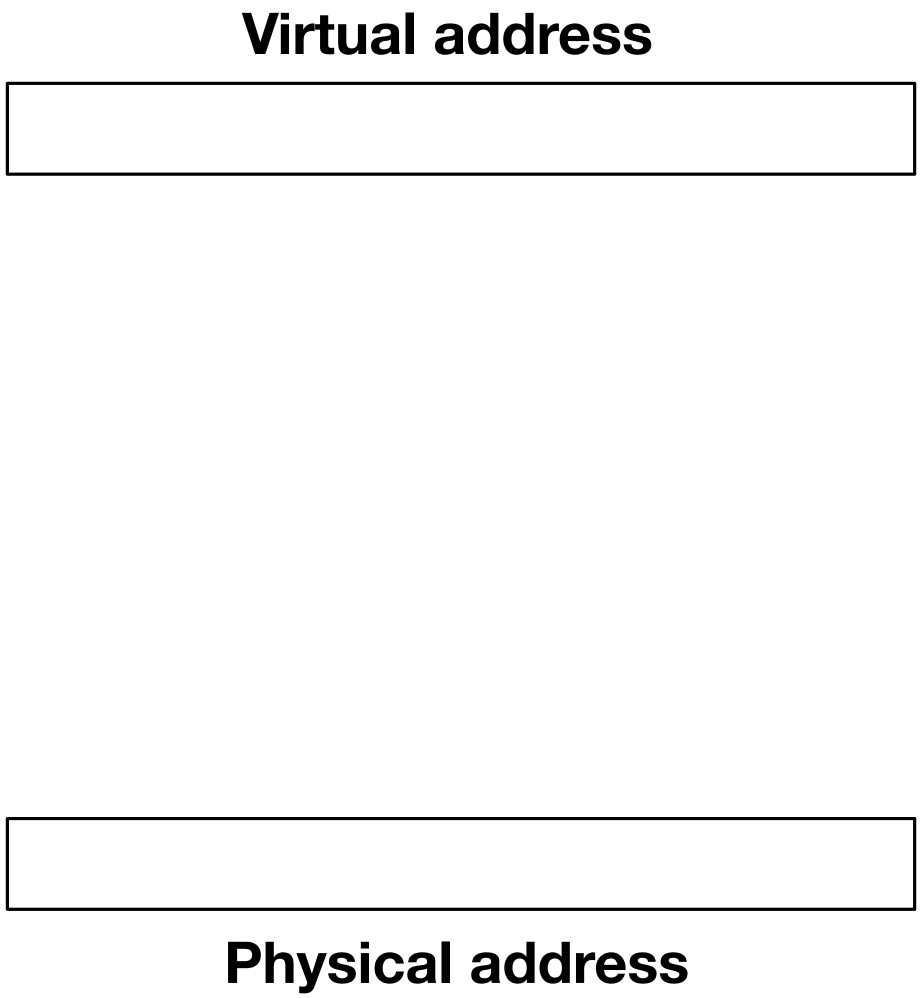
where:

$$\begin{aligned}\text{MAP(A)} &= \text{PA if in RAM}\\ &= 0 \text{ otherwise}\end{aligned}$$

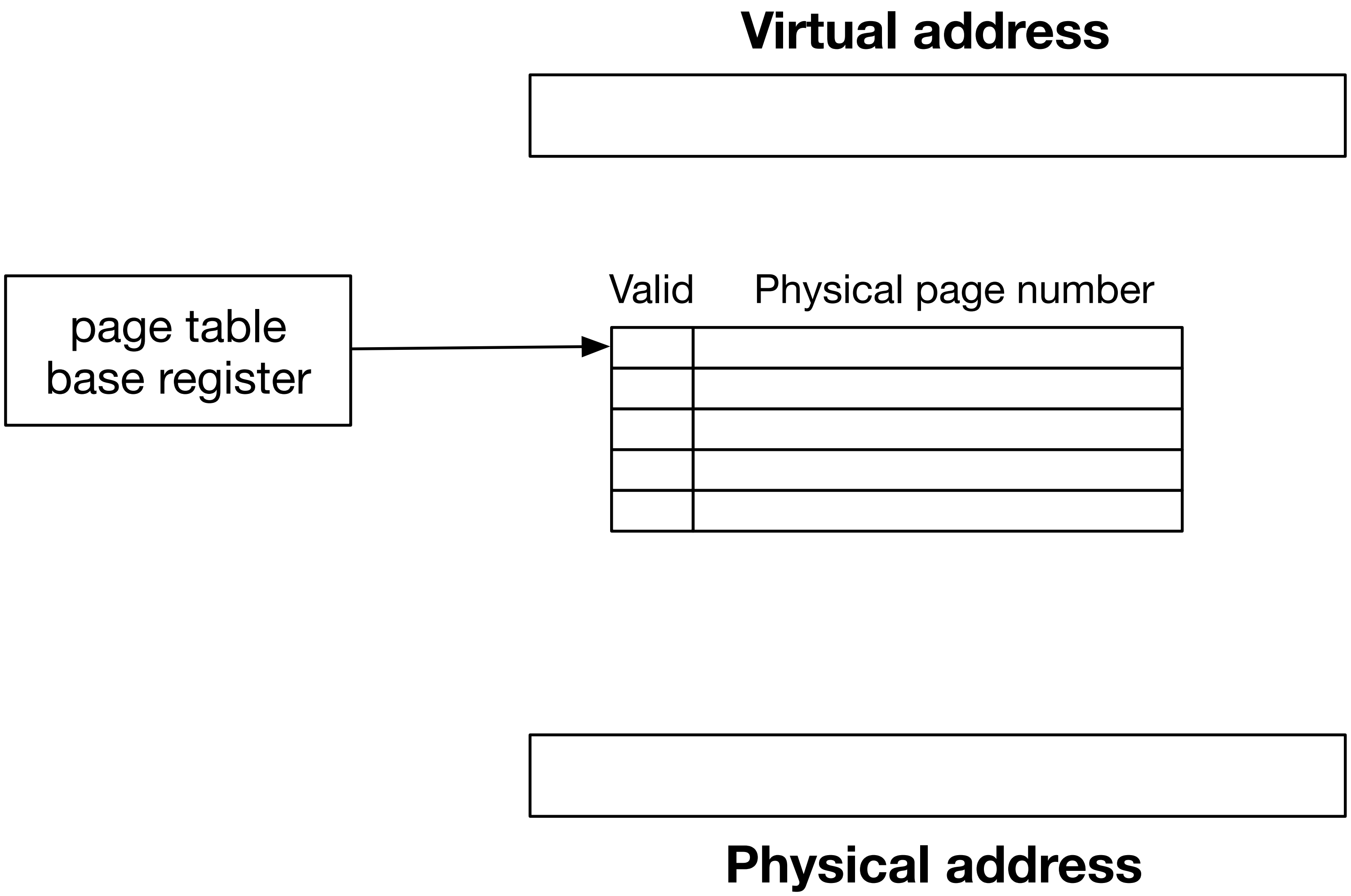
- Note: this happens very frequently in machine code

- We will do this in hardware: Memory Management Unit (MMU)

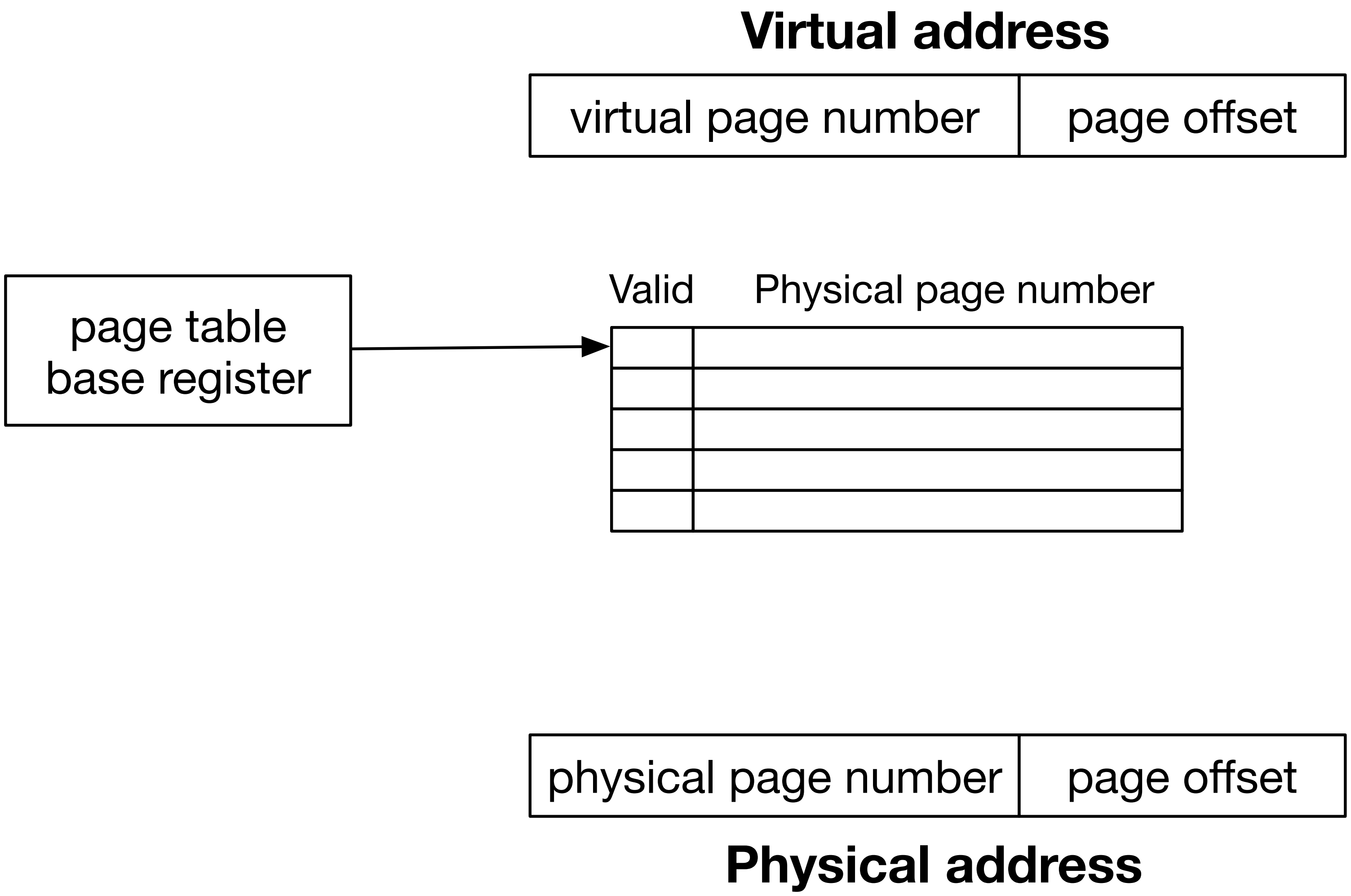
Basic Architecture



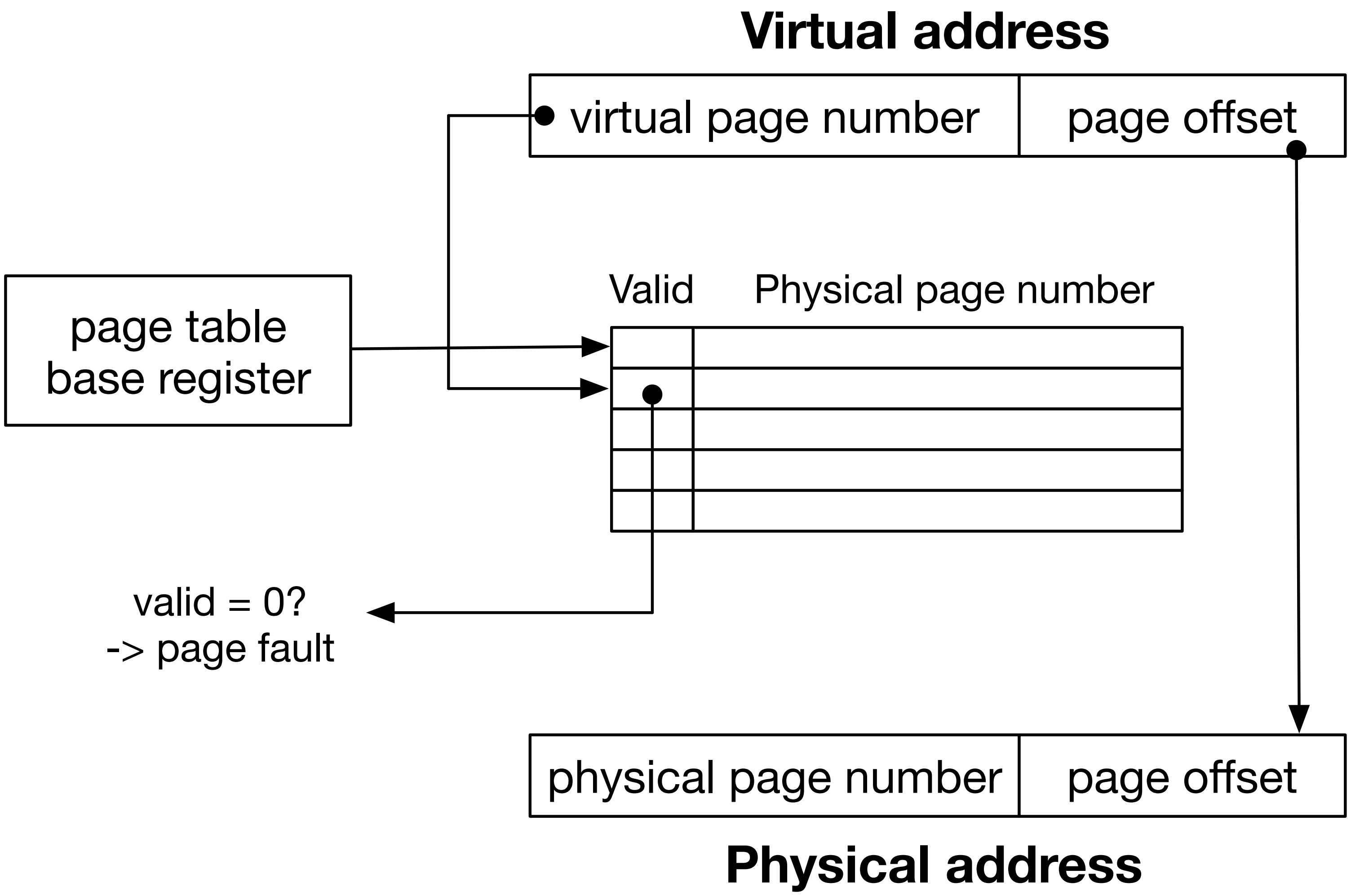
Basic Architecture



Basic Architecture



Basic Architecture


Virtual address
virtual page number
page offset
Valid
Physical page number
page table
base register
valid = 0?
-> page fault
physical page number
page offset
Physical address