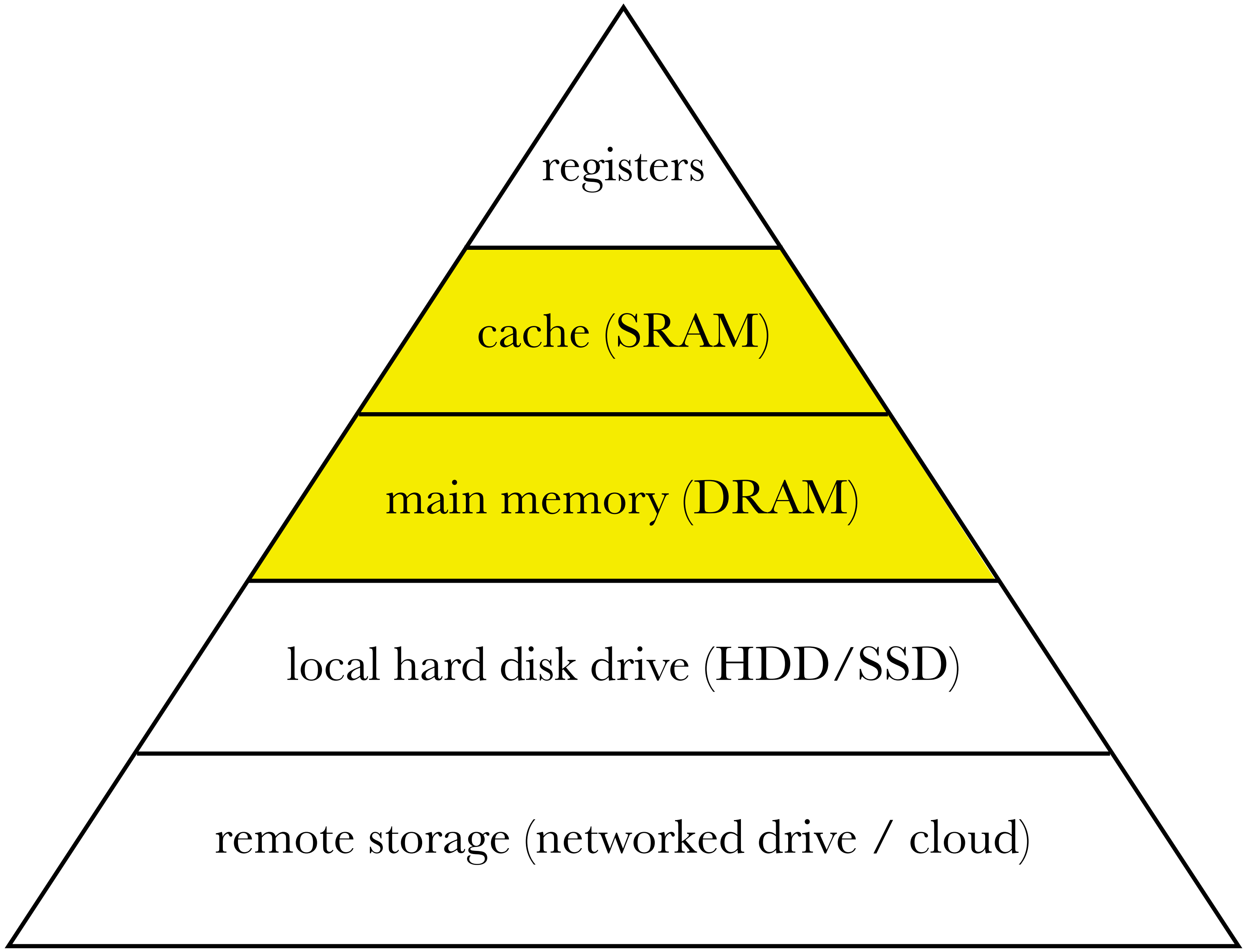
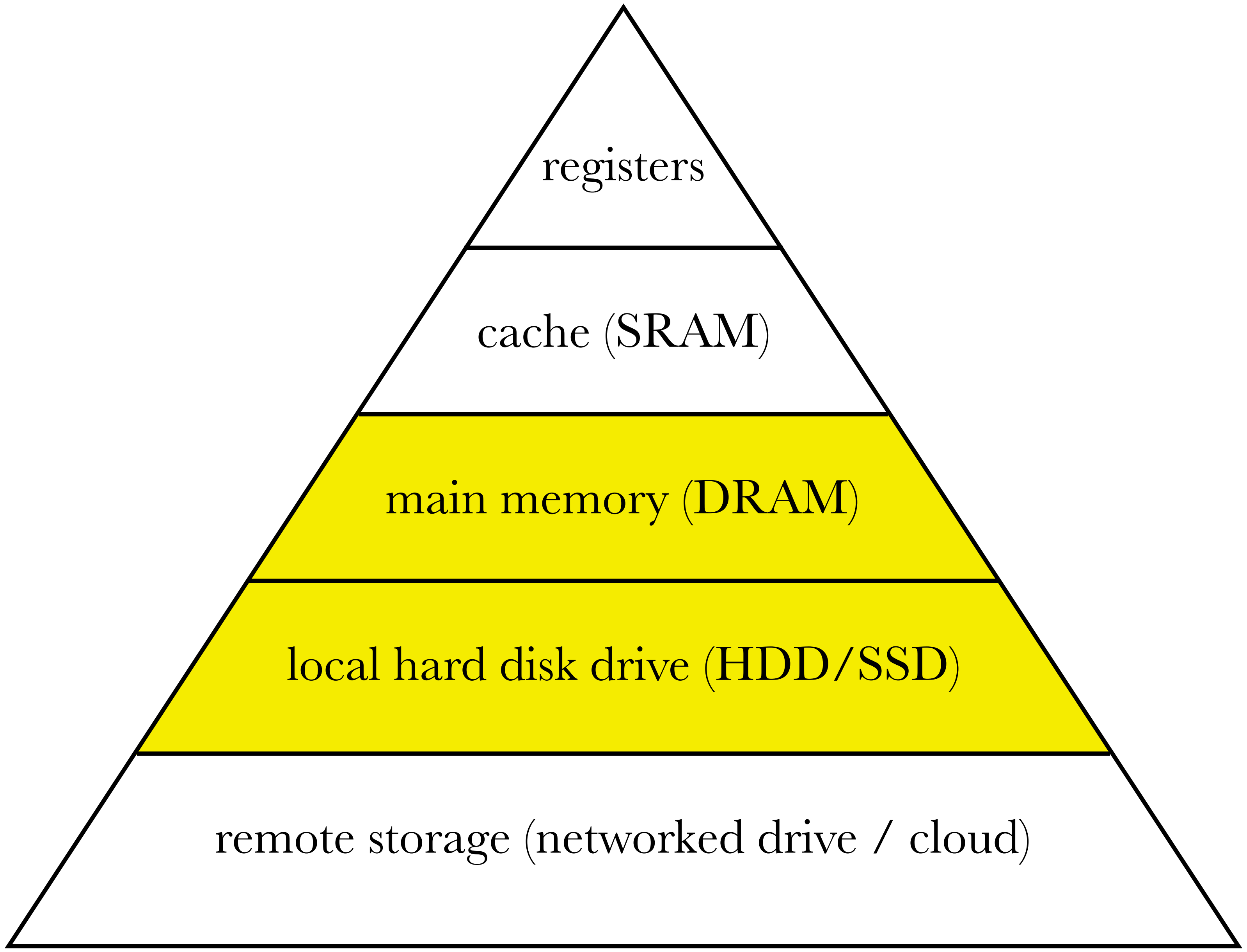
Virtual Memory

CS 351: Systems Programming

Michael Lee <lee@iit.edu>


registers
cache (SRAM)
main memory (DRAM)
local hard disk drive (HDD/SSD)
remote storage (networked drive / cloud)

previously: SRAM ⇔ DRAM


registers
cache (SRAM)
main memory (DRAM)
local hard disk drive (HDD/SSD)
remote storage (networked drive / cloud)

next: DRAM ⇔ HDD, SSD, etc.

i.e., memory as a “cache” for disk

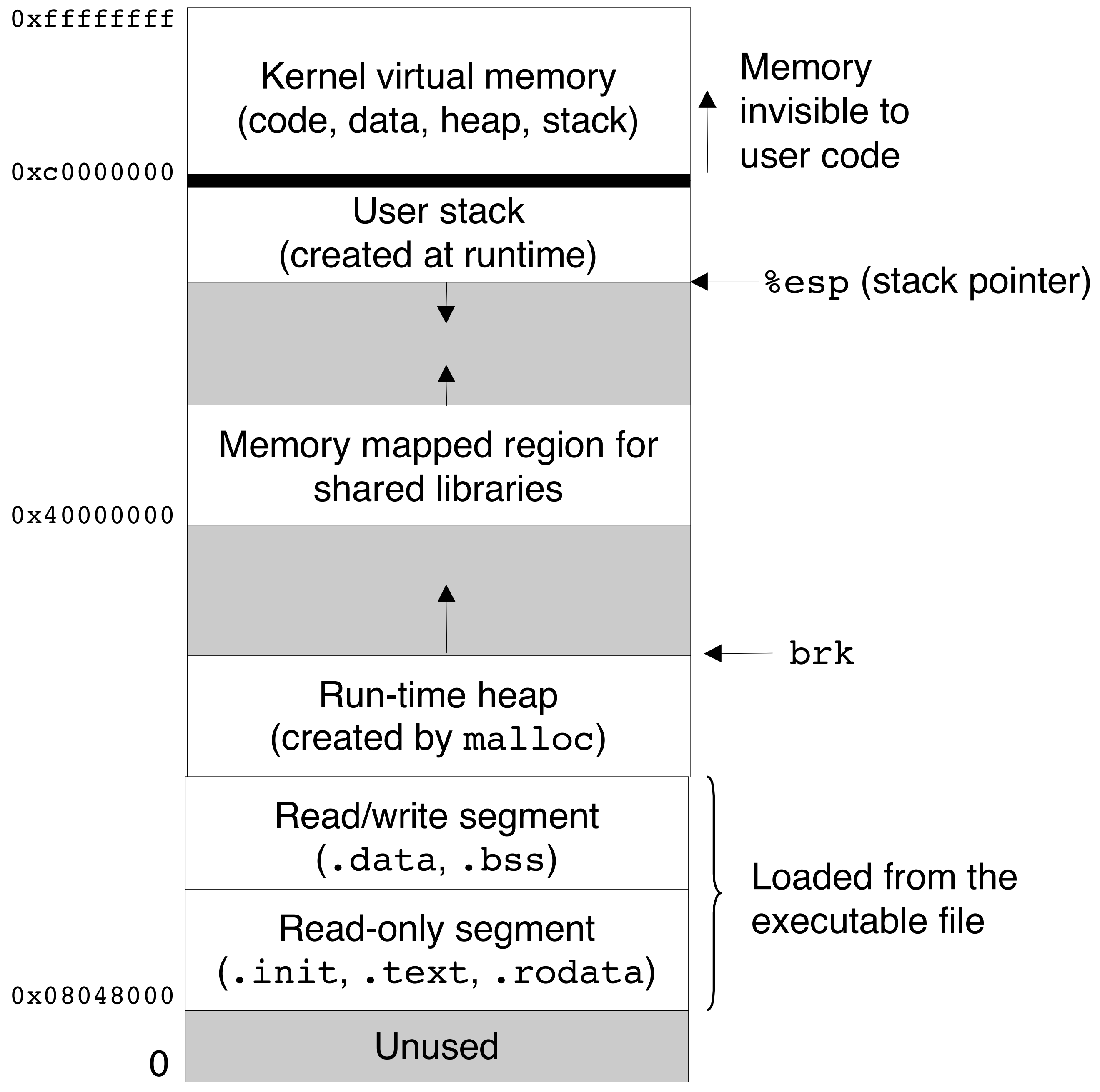
main goals:

1. maximize memory *throughput*
2. maximize memory *utilization*
3. provide *address space consistency* & *memory protection* to processes

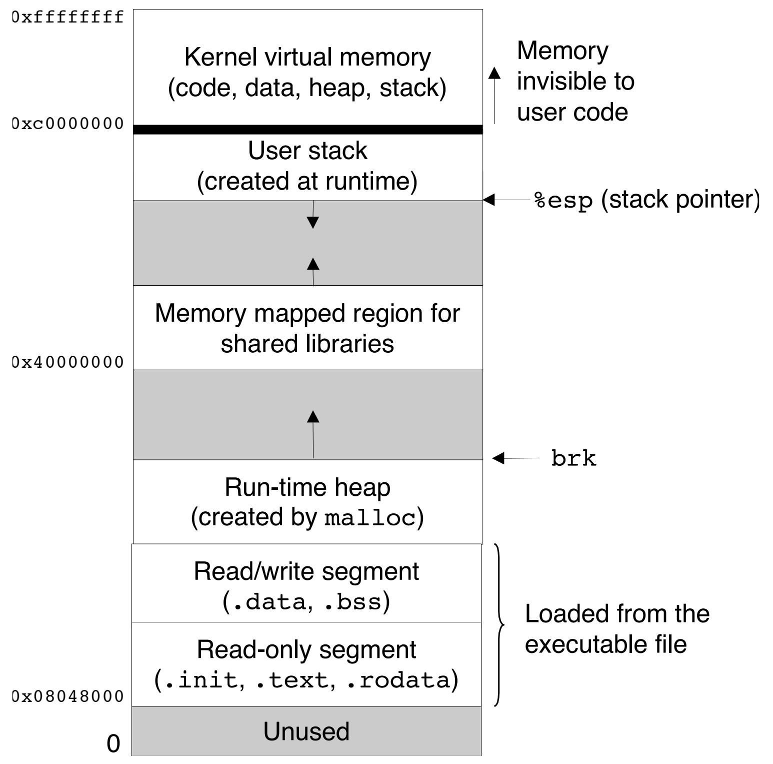
throughput = # bytes per second

- depends on access latencies (DRAM, HDD, etc.) and "hit rate" *utilization* = fraction of allocated memory that contains "user" data (aka *payload*)

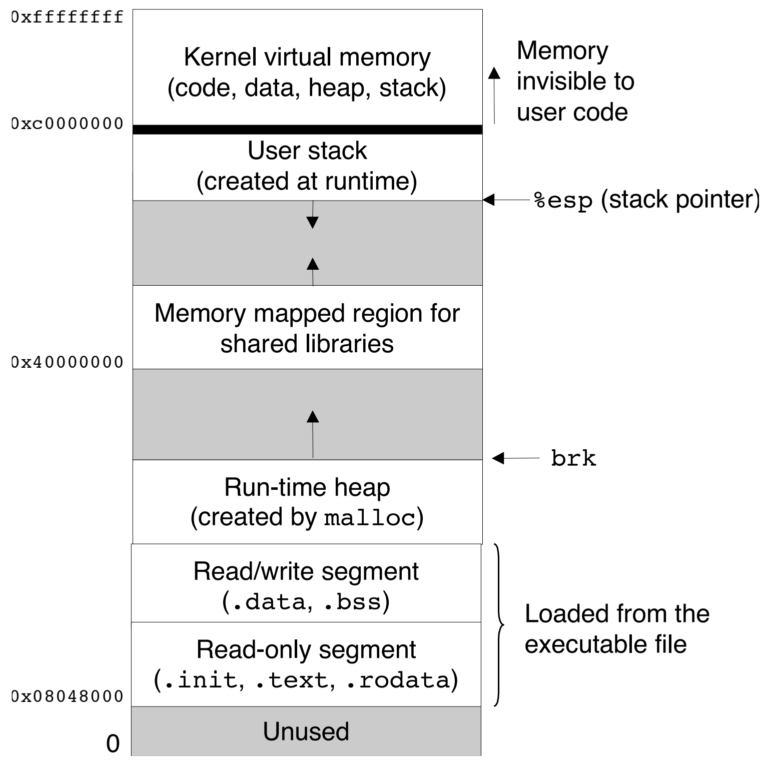
- vs. metadata and other overhead required for memory allocation *address space consistency* → provide a uniform "view" of memory to each process


0xffffffff
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
0xc0000000
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
0x40000000
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Loaded from the executable file
Read-only segment (.init, .text, .rodata)
0x08048000
Unused
0

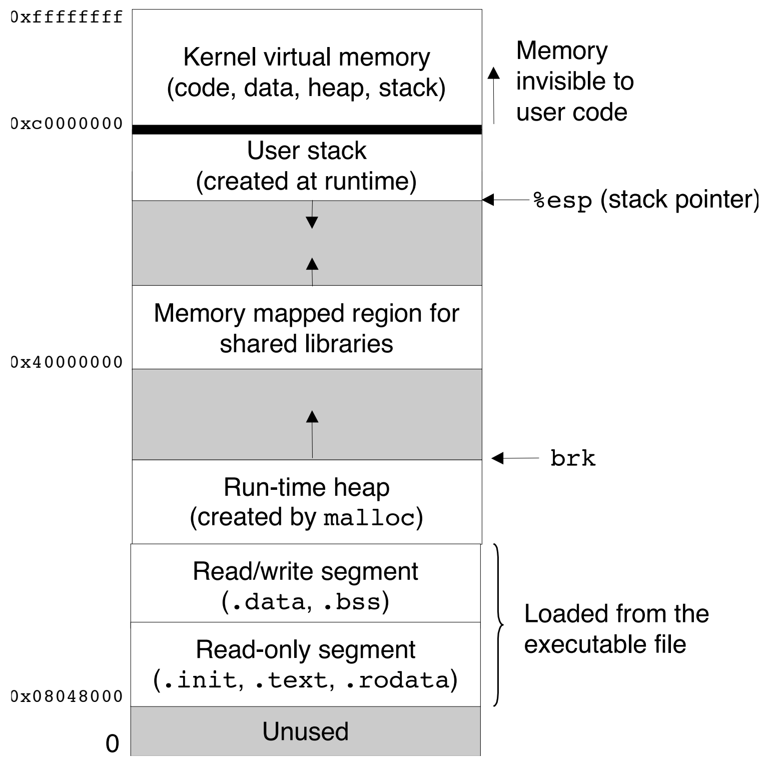
address space consistency → provide a uniform "view" of memory to each process *memory protection* → prevent processes from directly accessing each other's address space


0xffffffff
0xc0000000
0x40000000
0x08048000
0
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
Unused

P_0


0xffffffff
0xc0000000
0x40000000
0x08048000
0
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
Unused

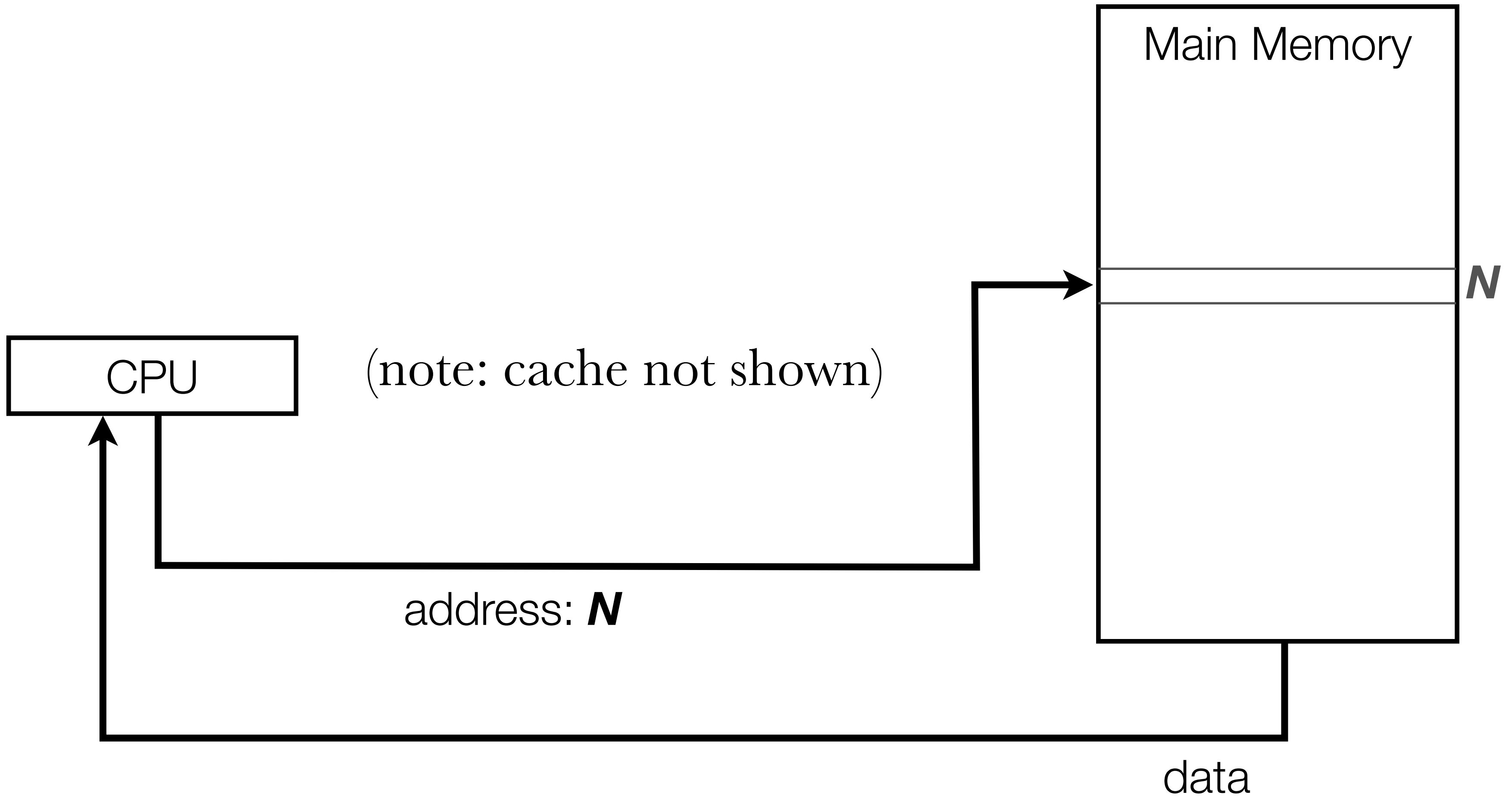
P_1


0xffffffff
0xc0000000
0x40000000
0x08048000
0
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
Unused

P_2

memory protection → prevent processes from directly accessing each other's address space i.e., every process should be provided with a managed, ***virtualized*** address space

“memory addresses”: what are they, really?



“physical” address: (byte) index into DRAM

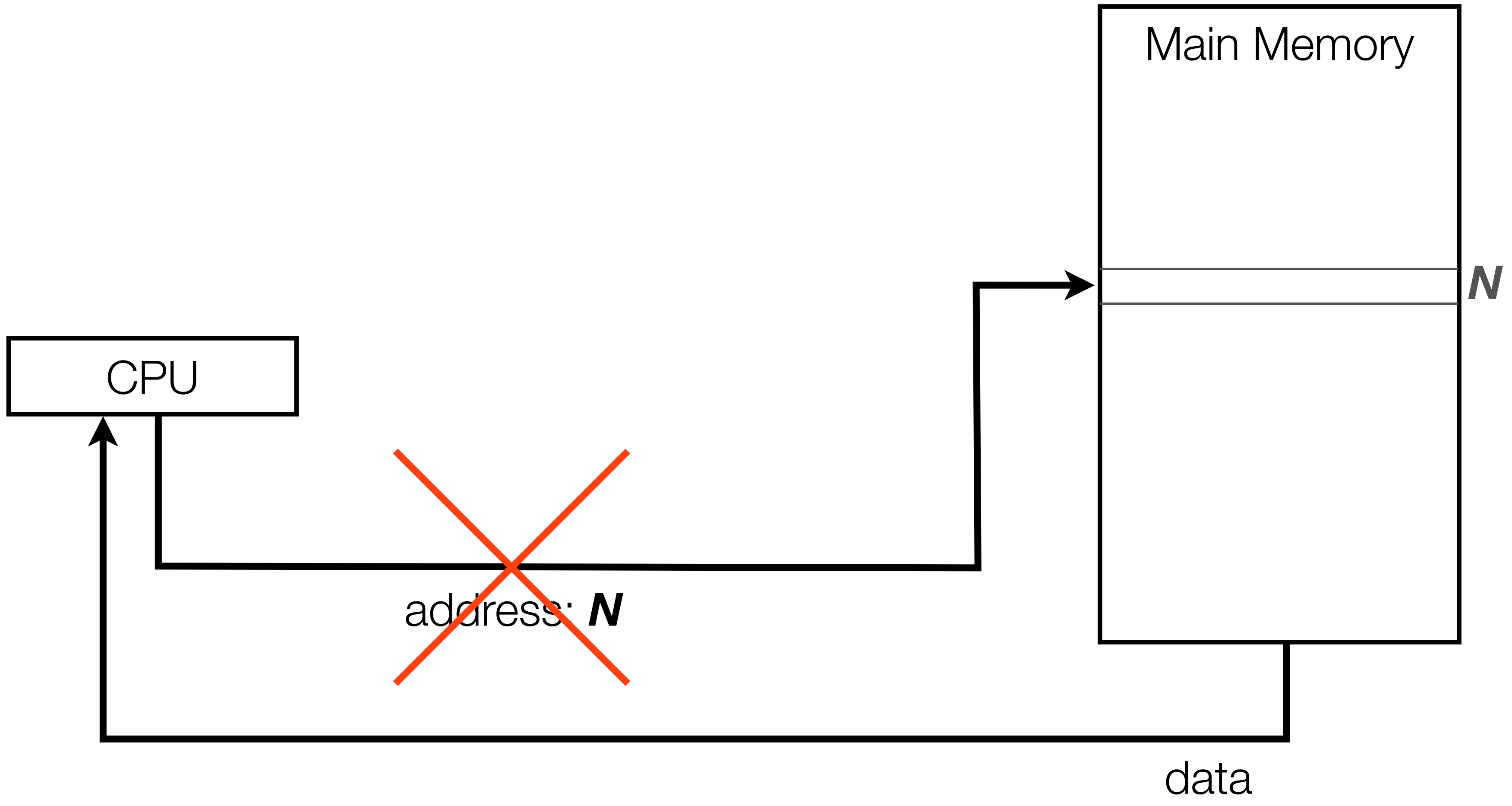
```
int glob = 0xDEADBEEE;

main() {
    fork();
    glob += 1;
}
```

```
(gdb) set detach-on-fork off
(gdb) break main
Breakpoint 1 at 0x400508: file memtest.c, line 7.
(gdb) run
Breakpoint 1, main () at memtest.c:7
7           fork();
(gdb) next
[New process 7450]
8           glob += 1;
(gdb) print &glob
$1 = (int *) 0x6008d4
(gdb) next
9       }
(gdb) print /x glob
$2 = 0xdeadbeef
(gdb) inferior 2
[Switching to inferior 2 [process 7450]
#0  0x000000310acac49d in __libc_fork ()
131         pid = ARCH_FORK ();
(gdb) finish
Run till exit from #0  in __libc_fork ()
8           glob += 1;
(gdb) print /x glob
$4 = 0xdeadbeee
(gdb) print &glob
$5 = (int *) 0x6008d4
```

parent

child

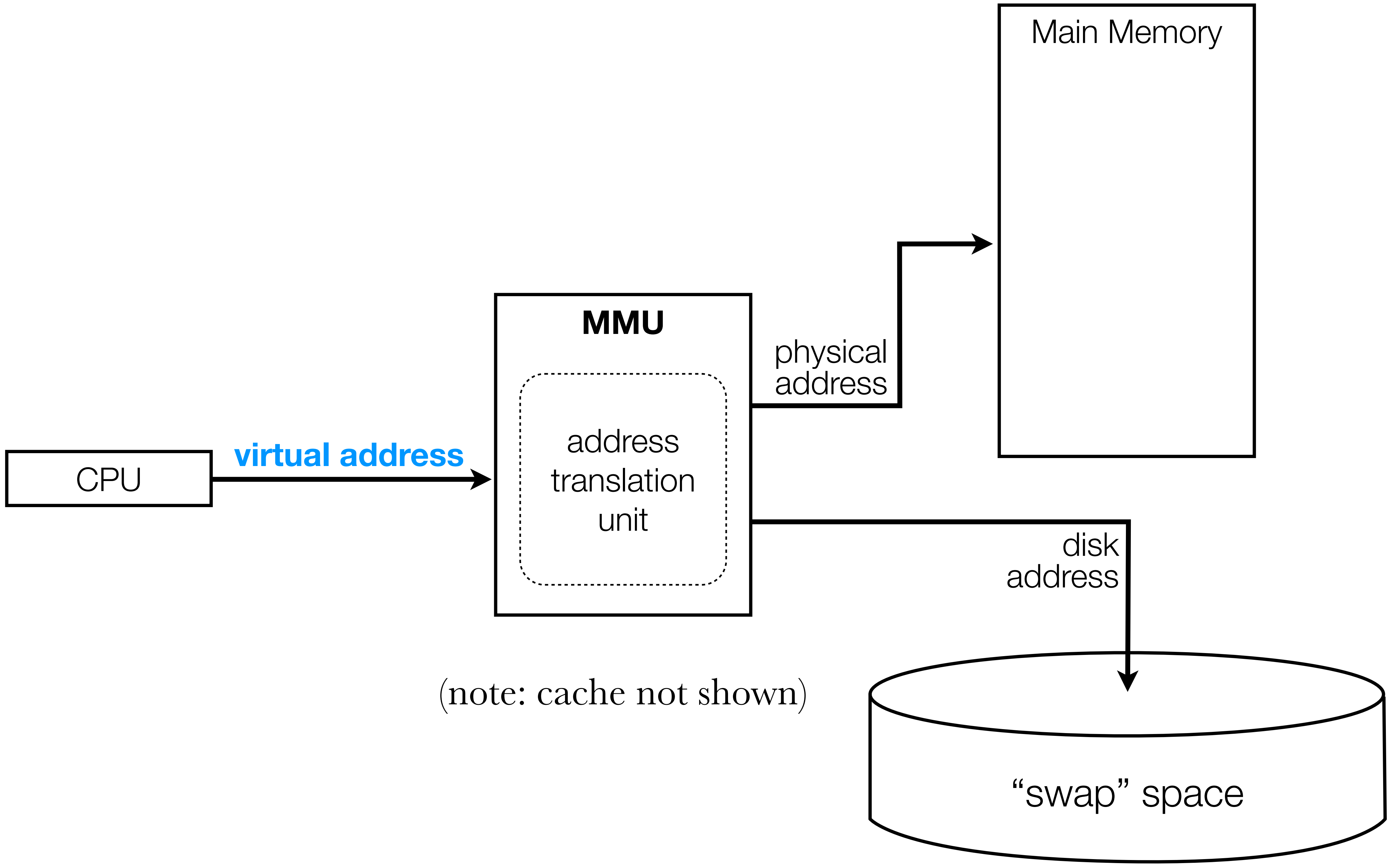


instructions executed by the CPU *do not* refer directly to *physical* addresses!

processes reference *virtual* addresses,

the CPU relays virtual address requests to the *memory management unit* (MMU),

which are *translated* to physical addresses



(note: cache not shown)

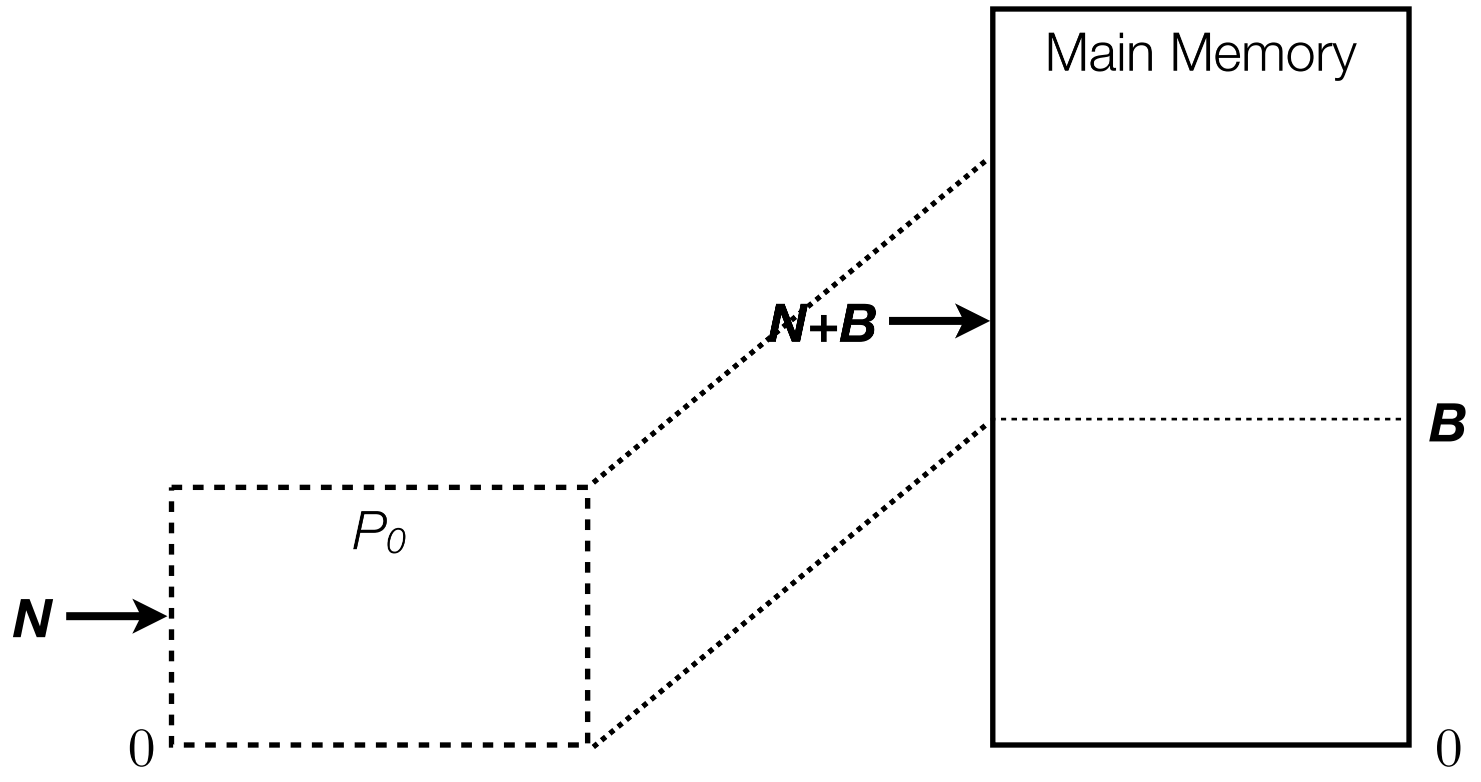
essential problem: translate request for a virtual address → physical address

… this must be **FAST**, as *every* memory access from the CPU must be translated both hardware/software are involved:

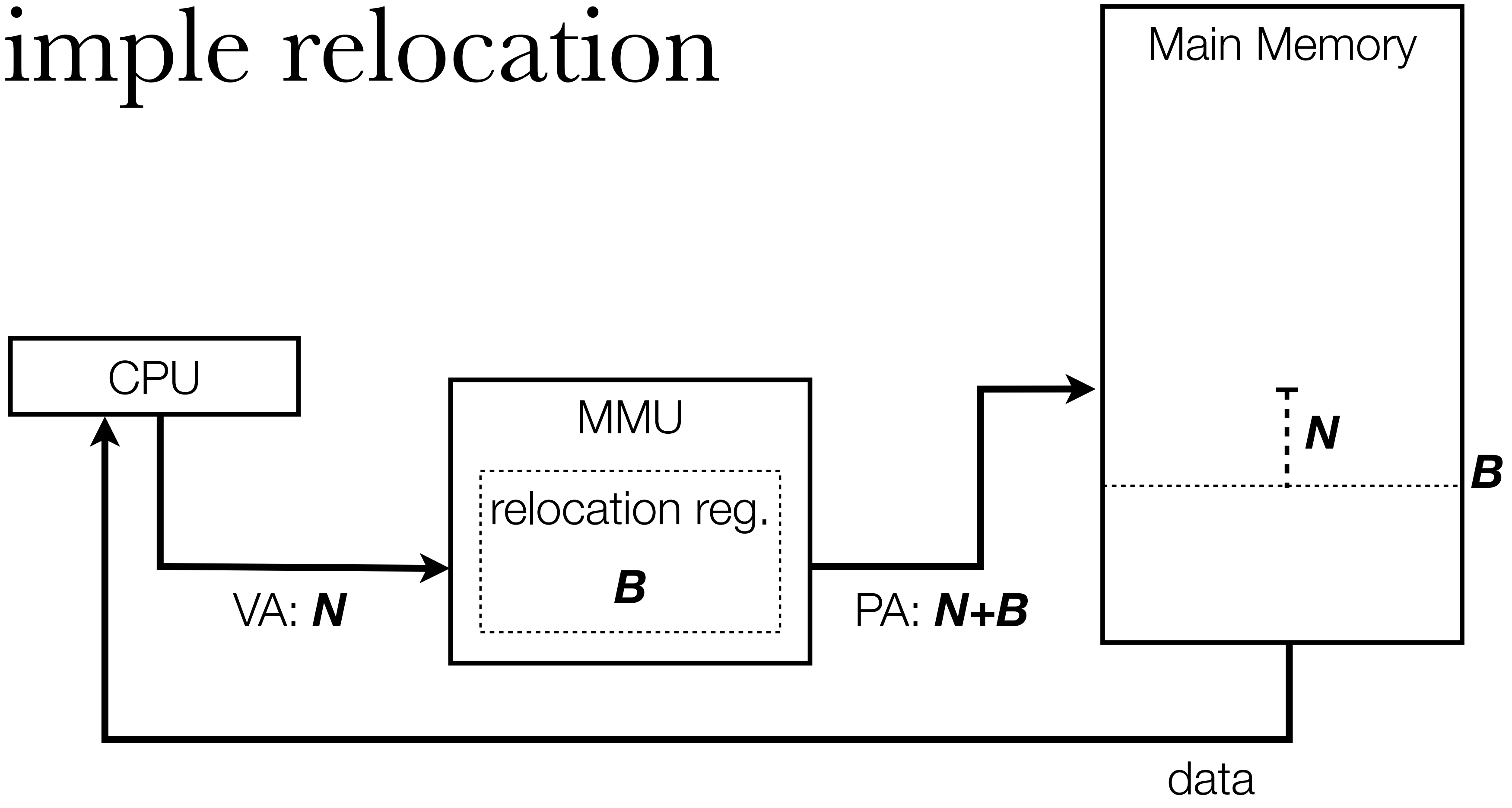
- MMU (hw) handles simple and fast operations (e.g., table lookups)
- Kernel (sw) handles complex tasks (e.g., eviction policy)

§Virtual Memory Implementations

1. simple relocation

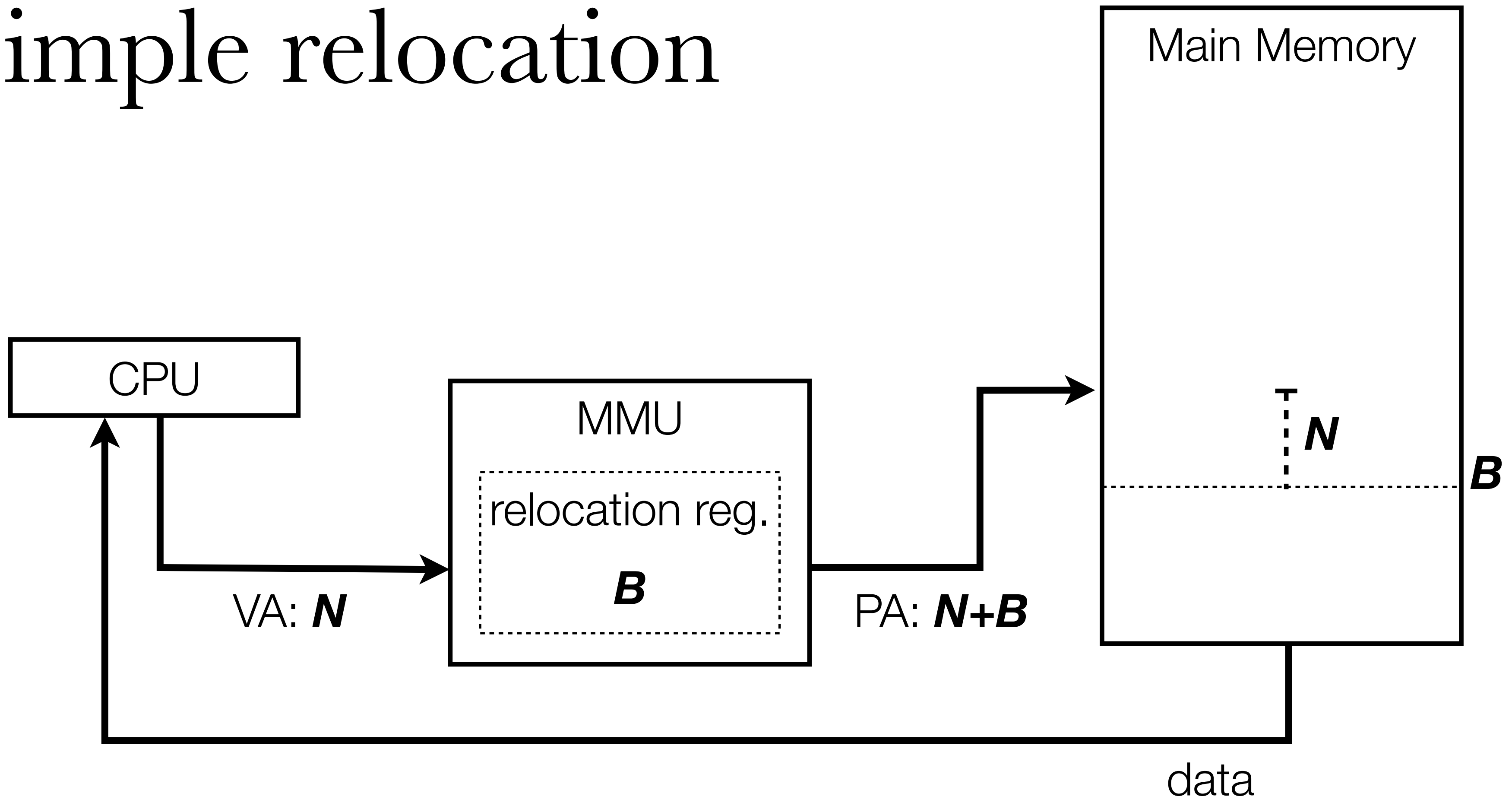


1. simple relocation



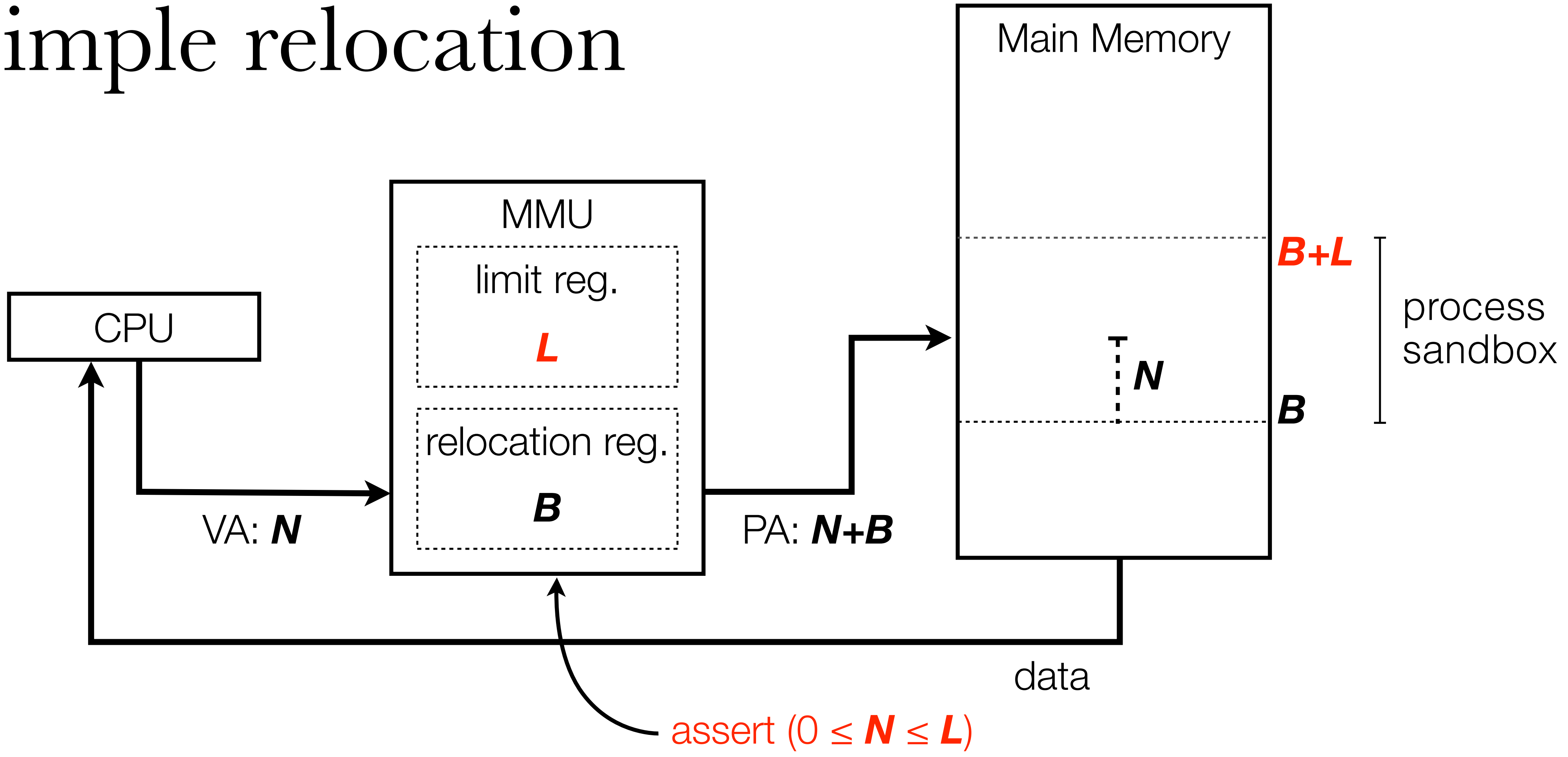
- per-process relocation address is loaded by kernel on every context switch

1. simple relocation



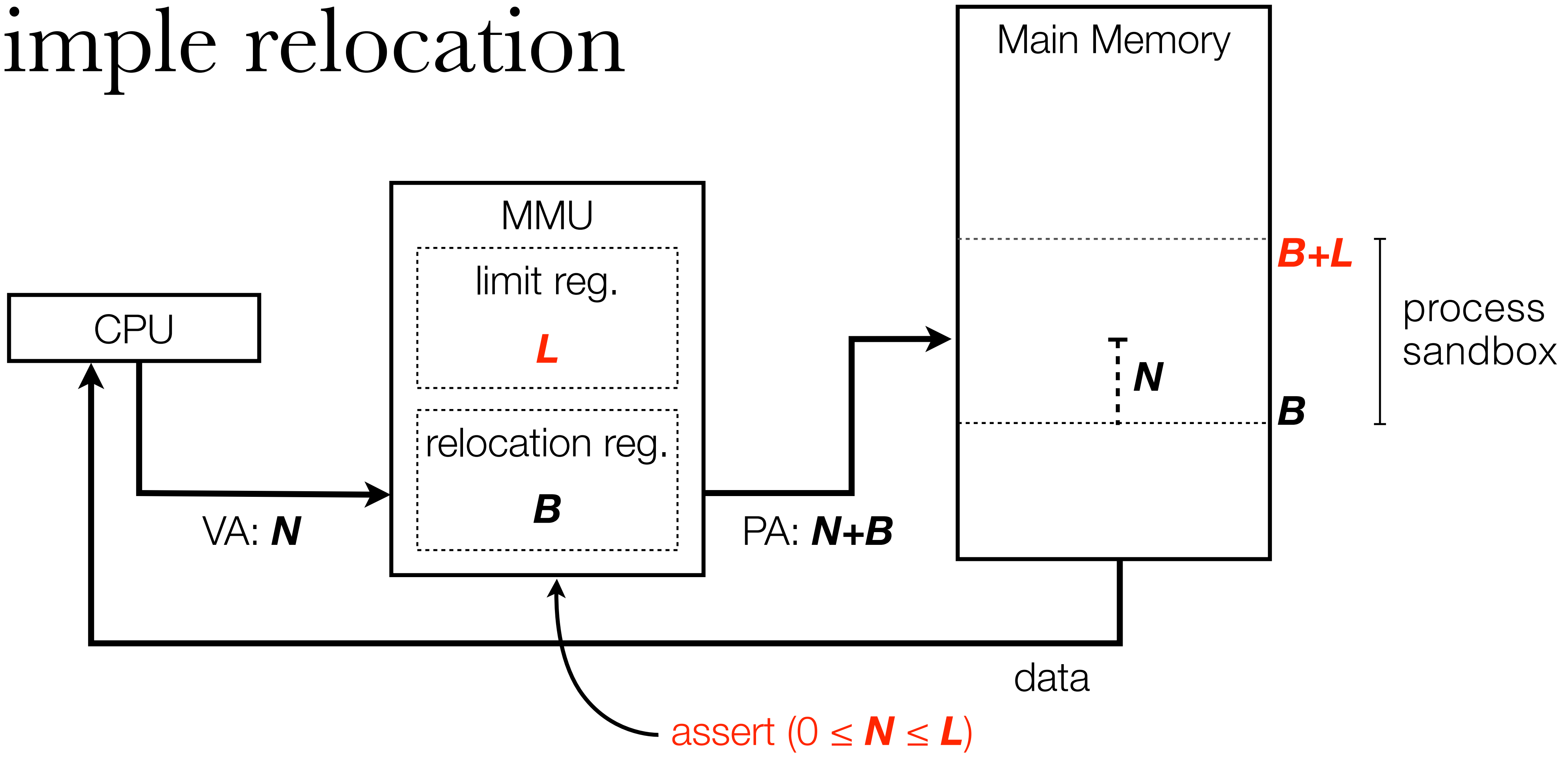
- problem: processes may easily overextend their bounds and trample on each other

1. simple relocation



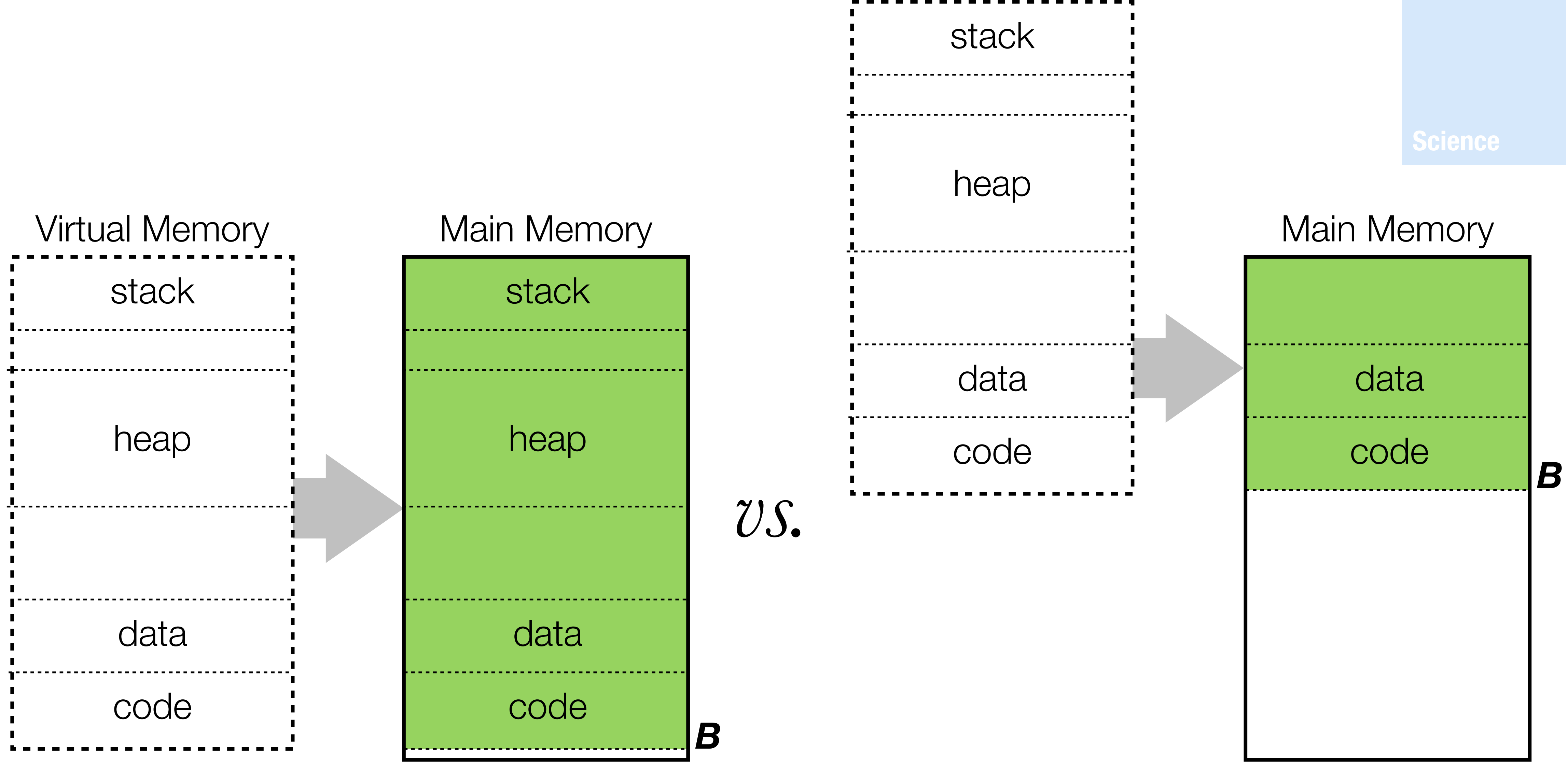
- incorporate a *limit* register to provide memory protection

1. simple relocation



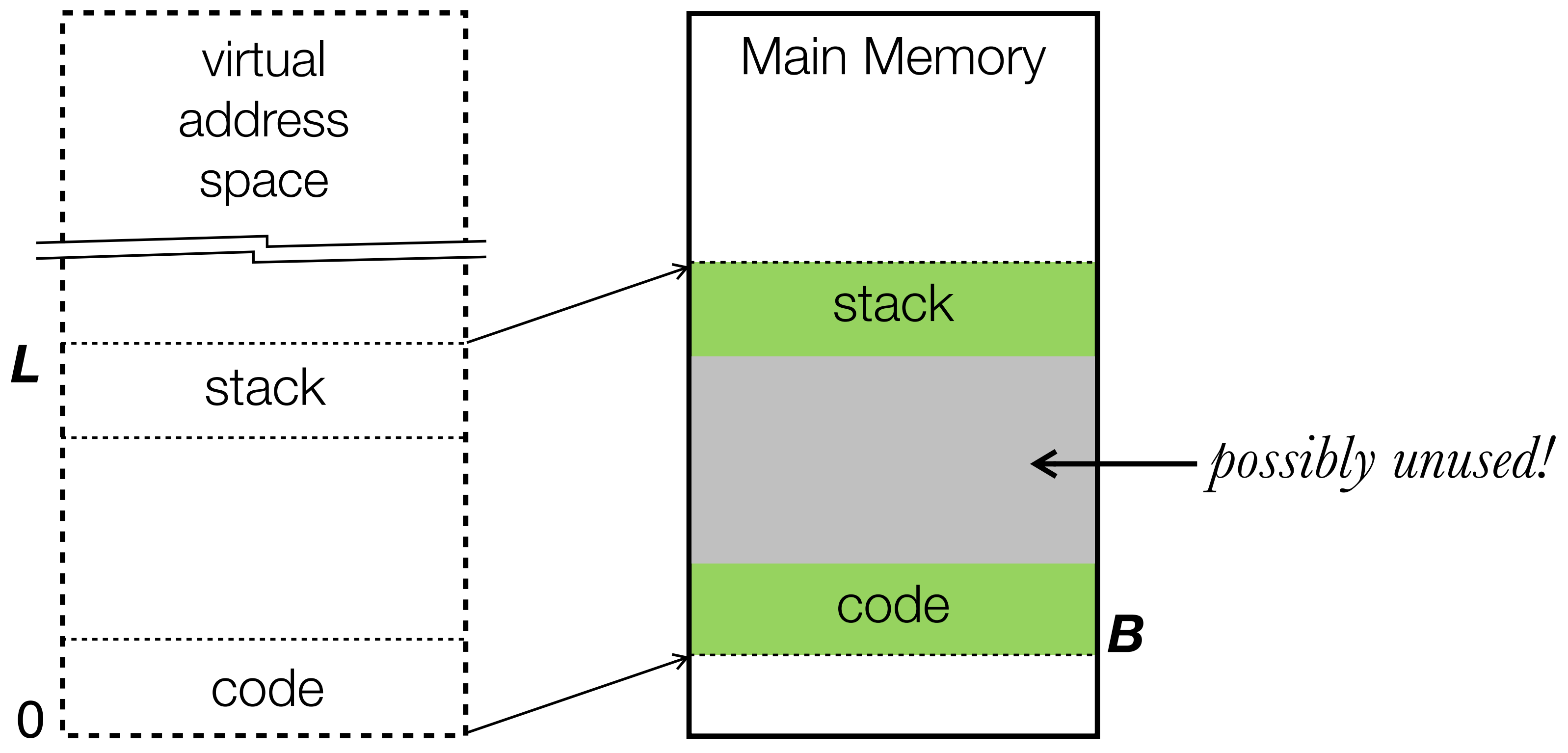
- assertion failure triggers a fault, which summons kernel (which signals process) pros:

- simple & fast!
- provides protection



but: available memory for mapping depends on value of base address

i.e., address spaces are *not consistent*!

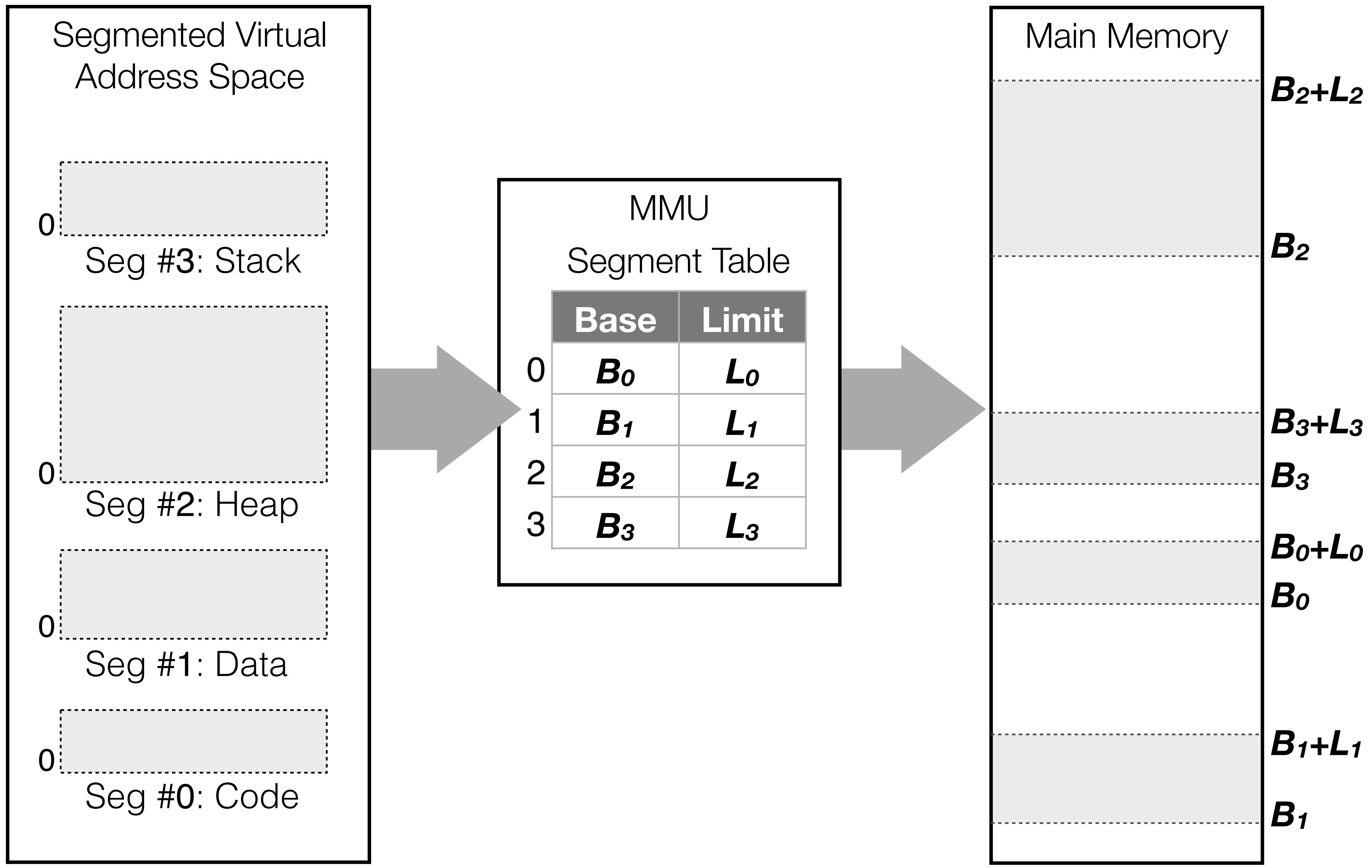


also: all of a process *below the address limit* must be loaded in memory

i.e., memory may be *vastly under-utilized*

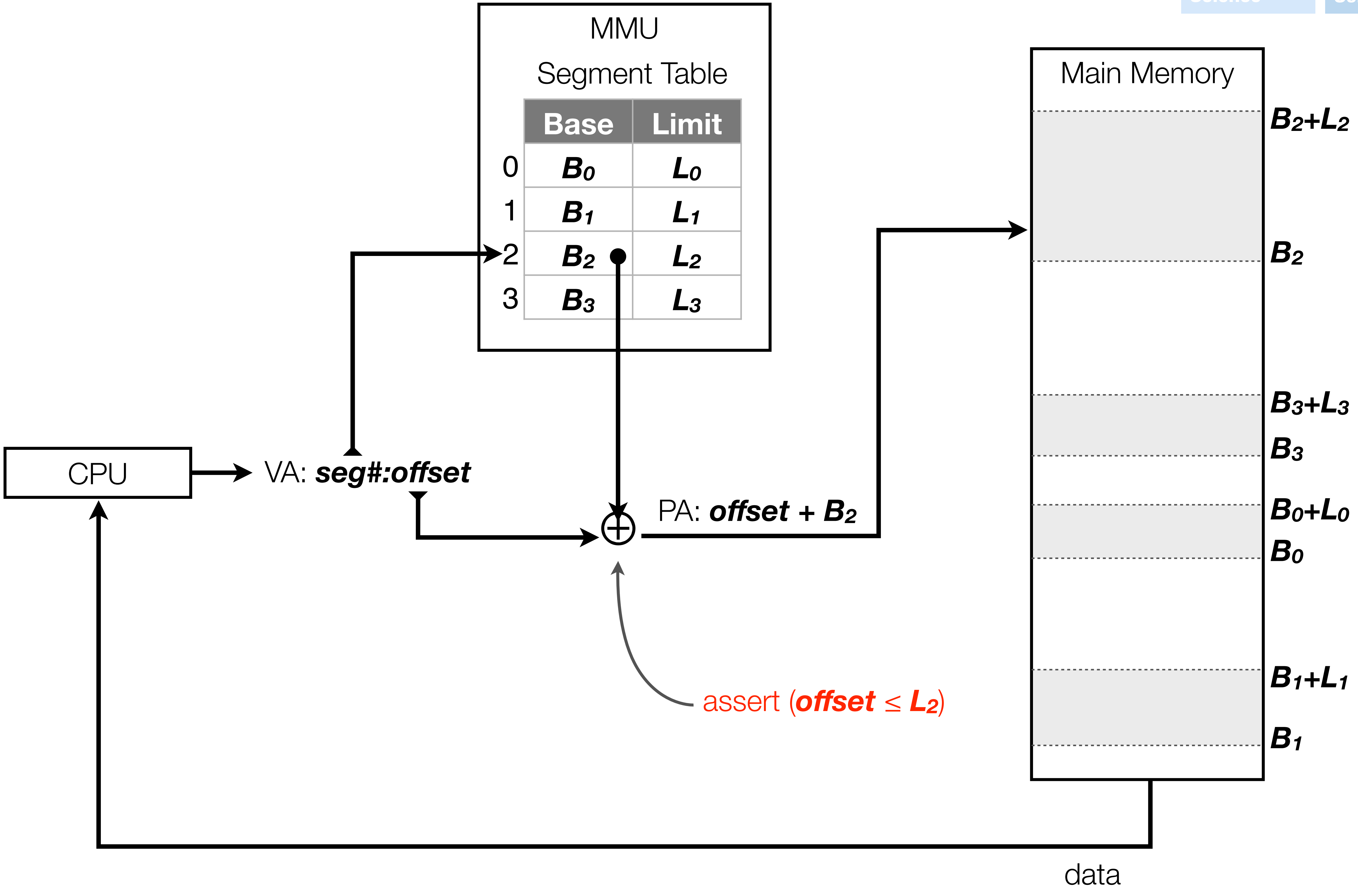
2. segmentation

- partition virtual address space into *multiple logical segments*
- individually map them onto physical memory with relocation registers



	Base	Limit
0	B_0	L_0
1	B_1	L_1
2	B_2	L_2
3	B_3	L_3

virtual address has form ***seg#:offset***


MMU
Segment Table
Base
Limit
0
1
2
3
B_0
L_0
B_1
L_1
B_2
L_2
B_3
L_3
CPU
VA: ***seg#:offset***
PA: ***offset + B_2***
assert (***offset*** ≤ ***L_2***)
Main Memory
B_2+L_2
B_2
B_3+L_3
B_3
B_0+L_0
B_0
B_1+L_1
B_1
data

Segment Table

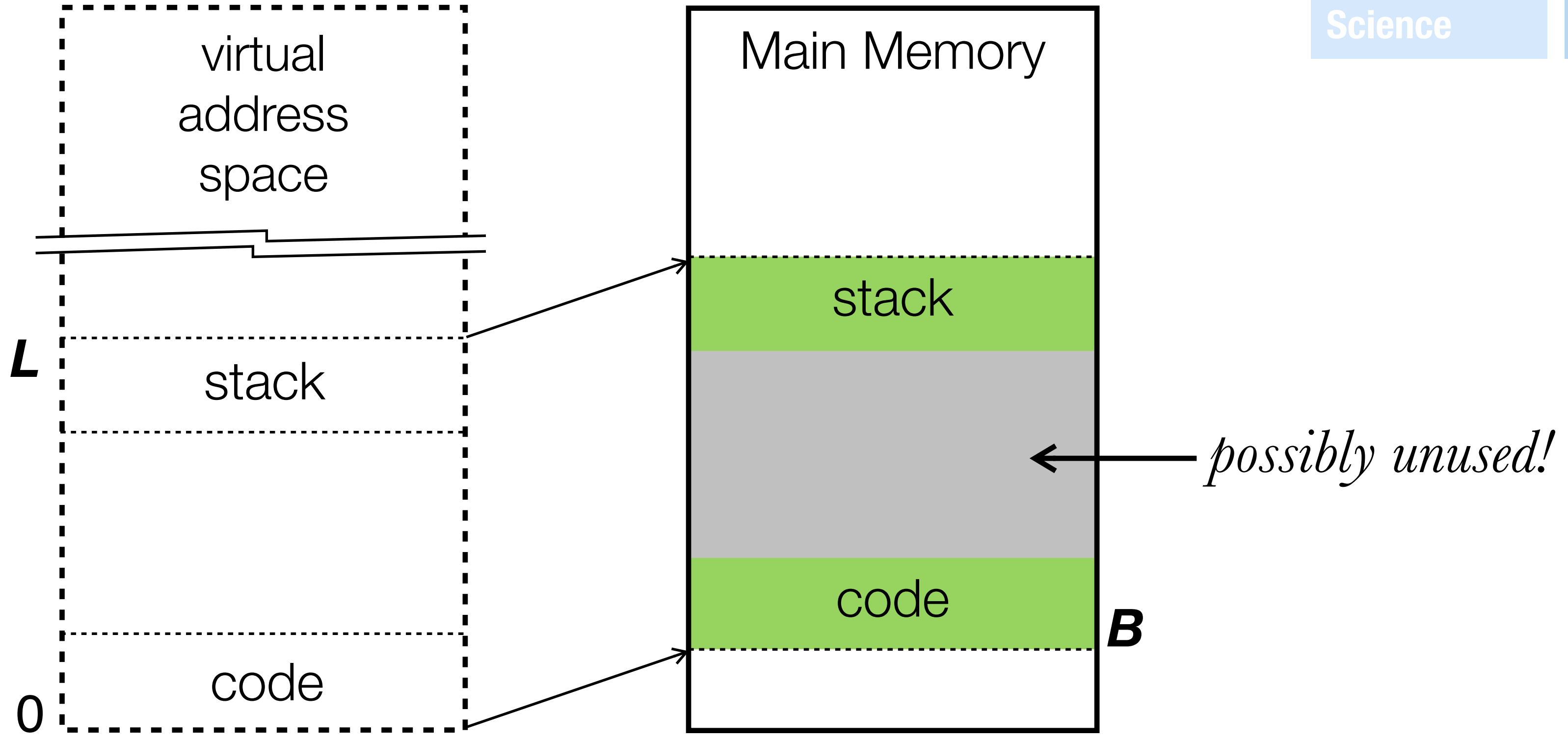
	Base	Limit
0	B_0	L_0
1	B_1	L_1
2	B_2	L_2
3	B_3	L_3

- implemented as MMU registers
- part of kernel-maintained, per-process metadata (aka "process control block")
- re-populated on each context switch

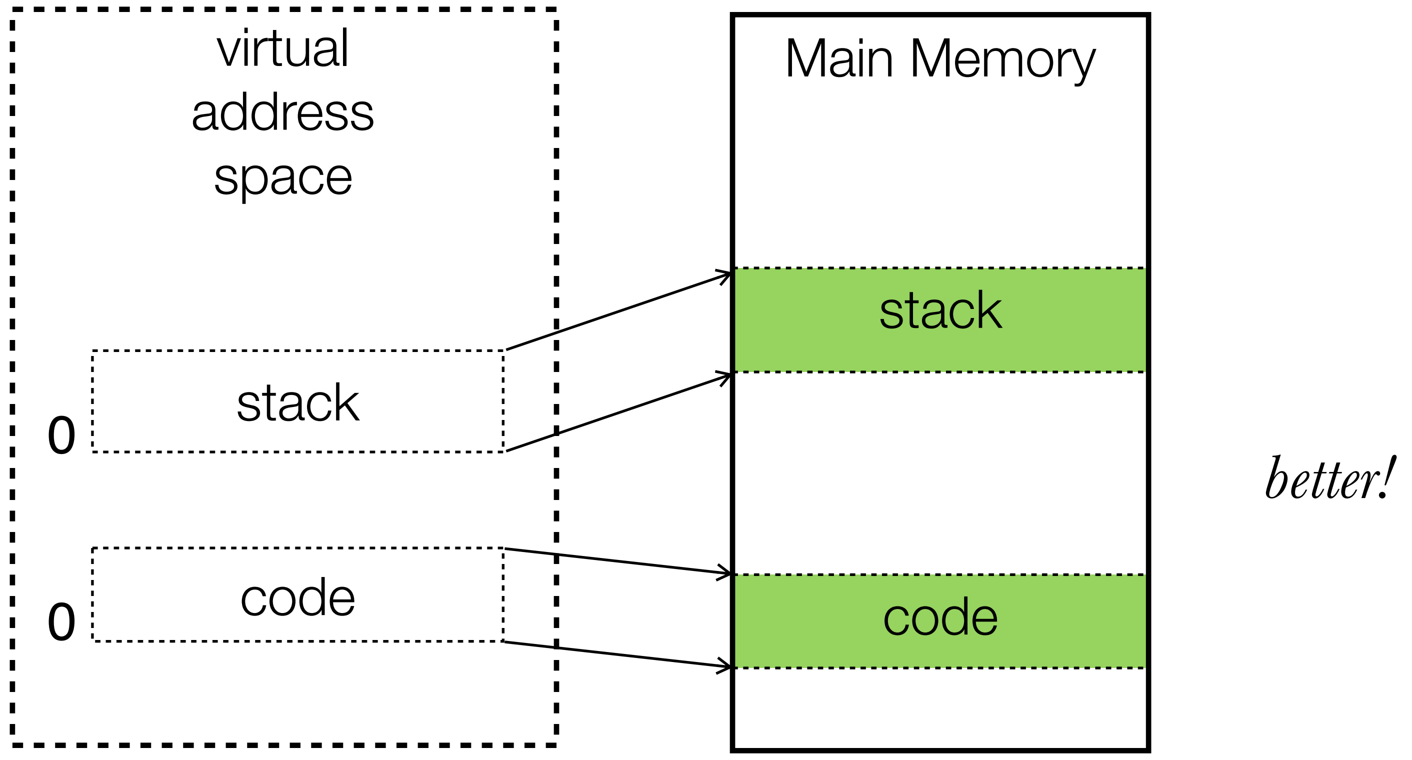
pros:

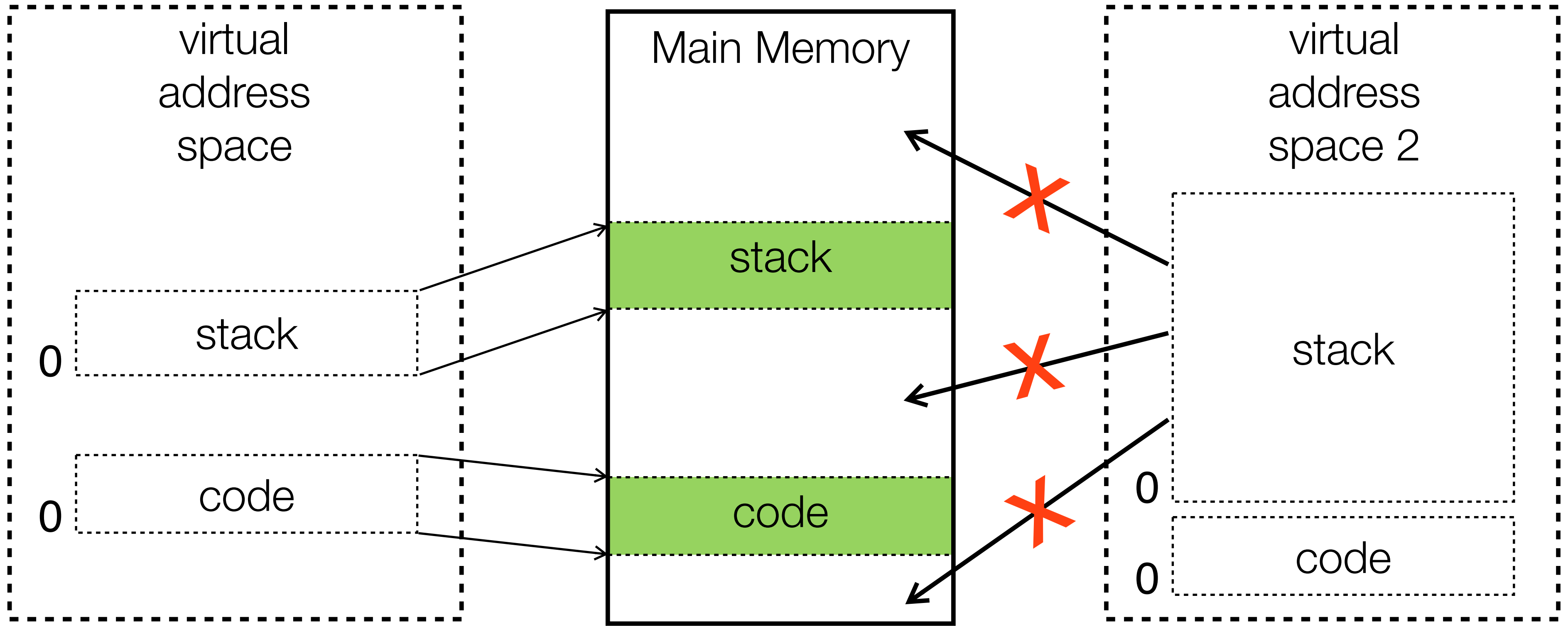
- still very *fast*
  - translation = register access & addition
- memory *protection* via limits
- segmented addresses improve *consistency*

simple relocation:



segmentation:



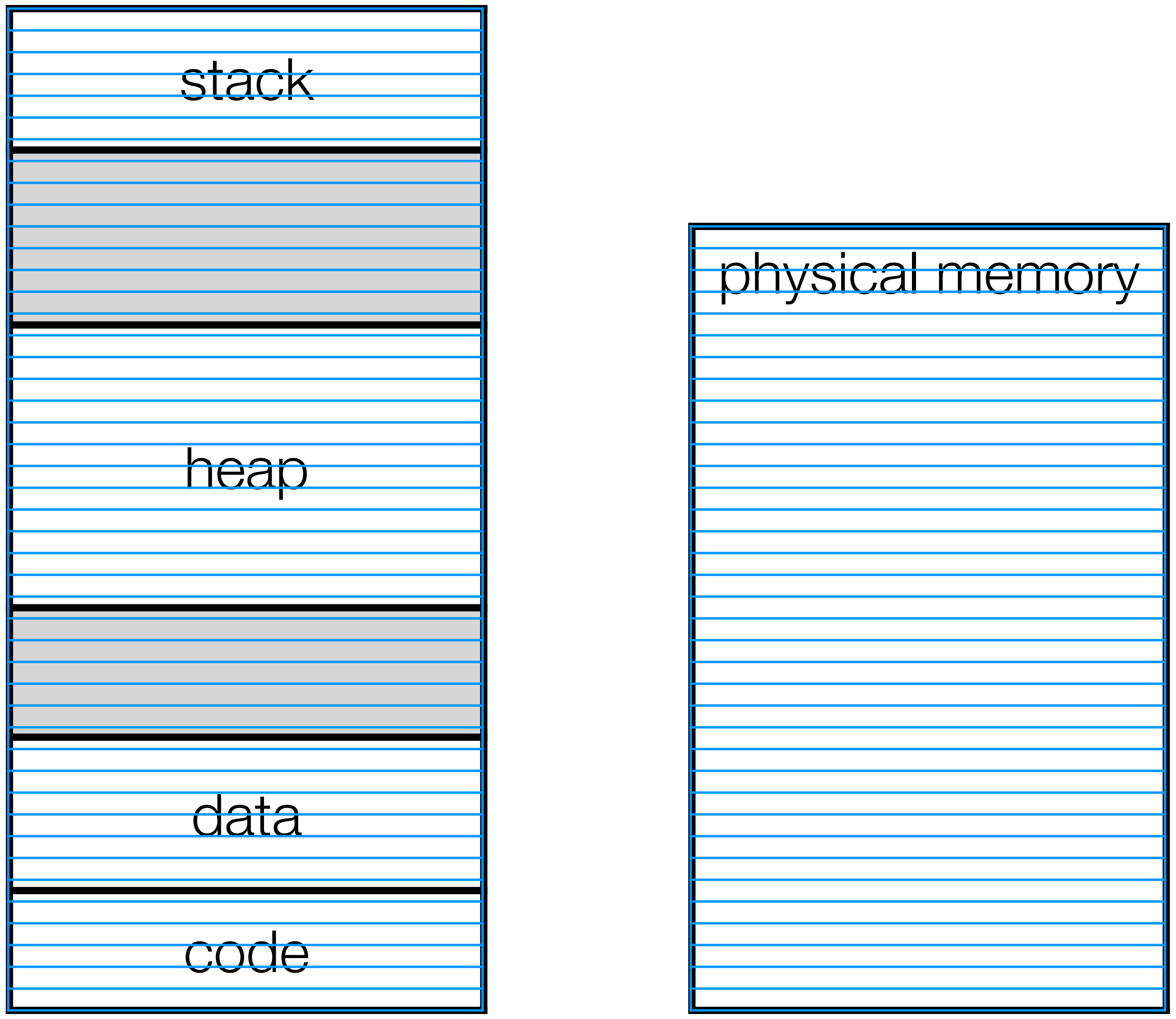


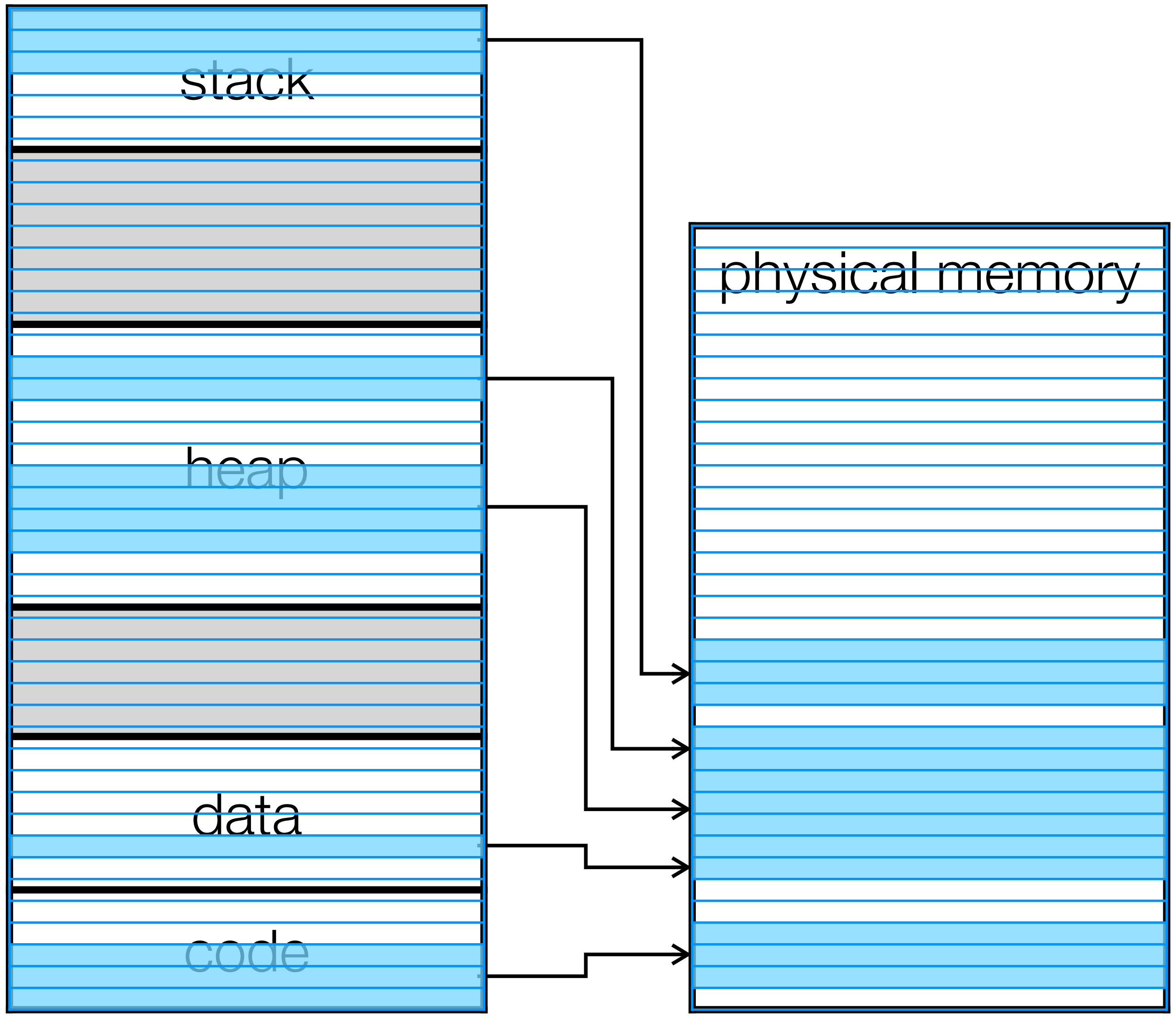
- variable segment sizes → *memory fragmentation*
- fragmentation potentially *lowers utilization*
- can fix through compaction, but expensive!

3. paging

- partition virtual and physical address spaces into *uniformly sized* ***pages***

- virtual pages map onto physical pages


stack
heap
data
code
physical memory

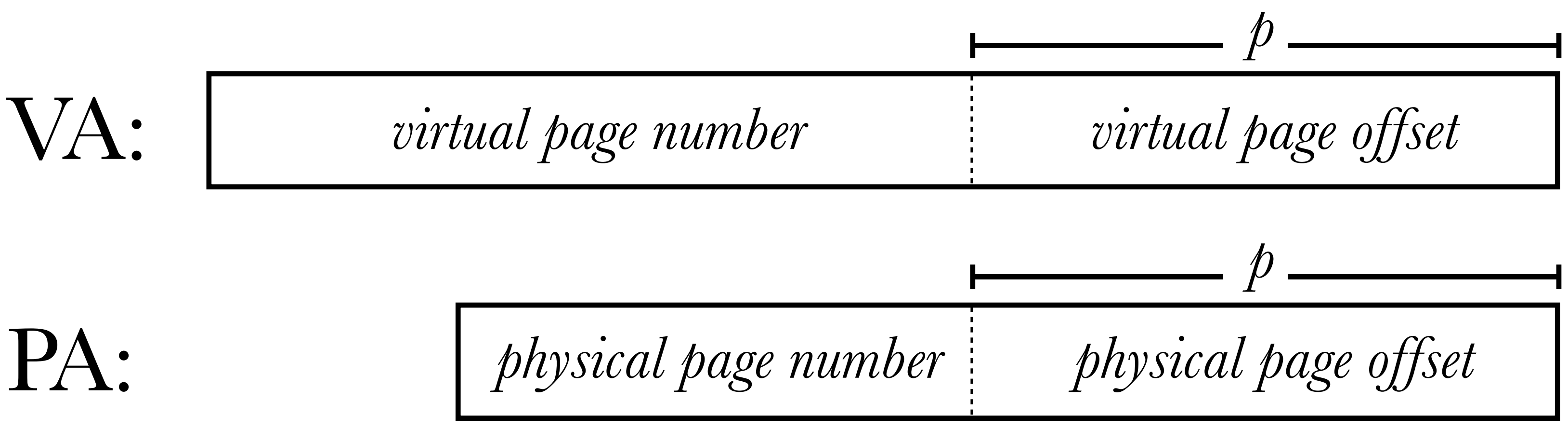


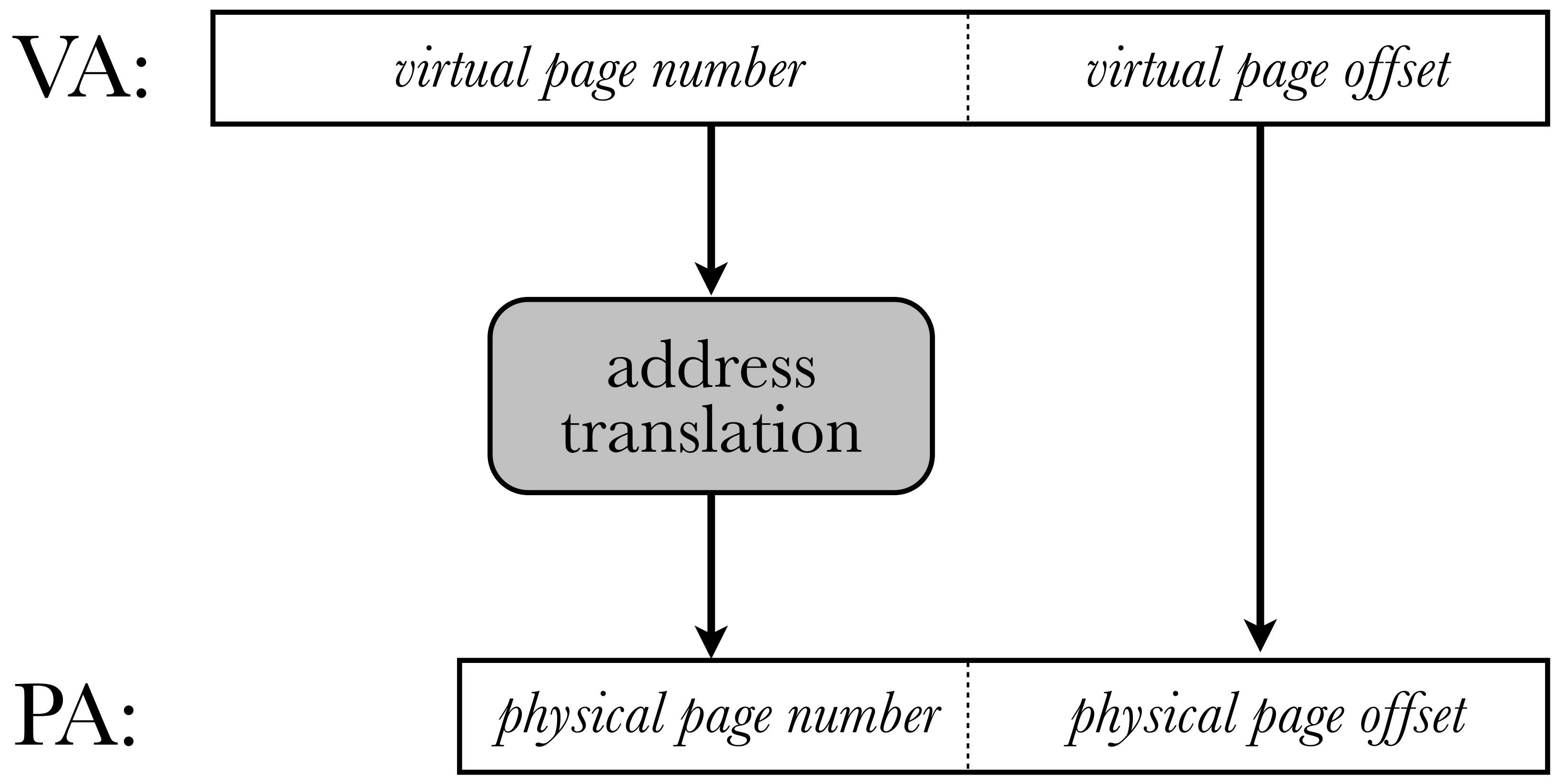
- minimum mapping granularity = page
- not all of a given segment need be mapped

modified mapping problem:

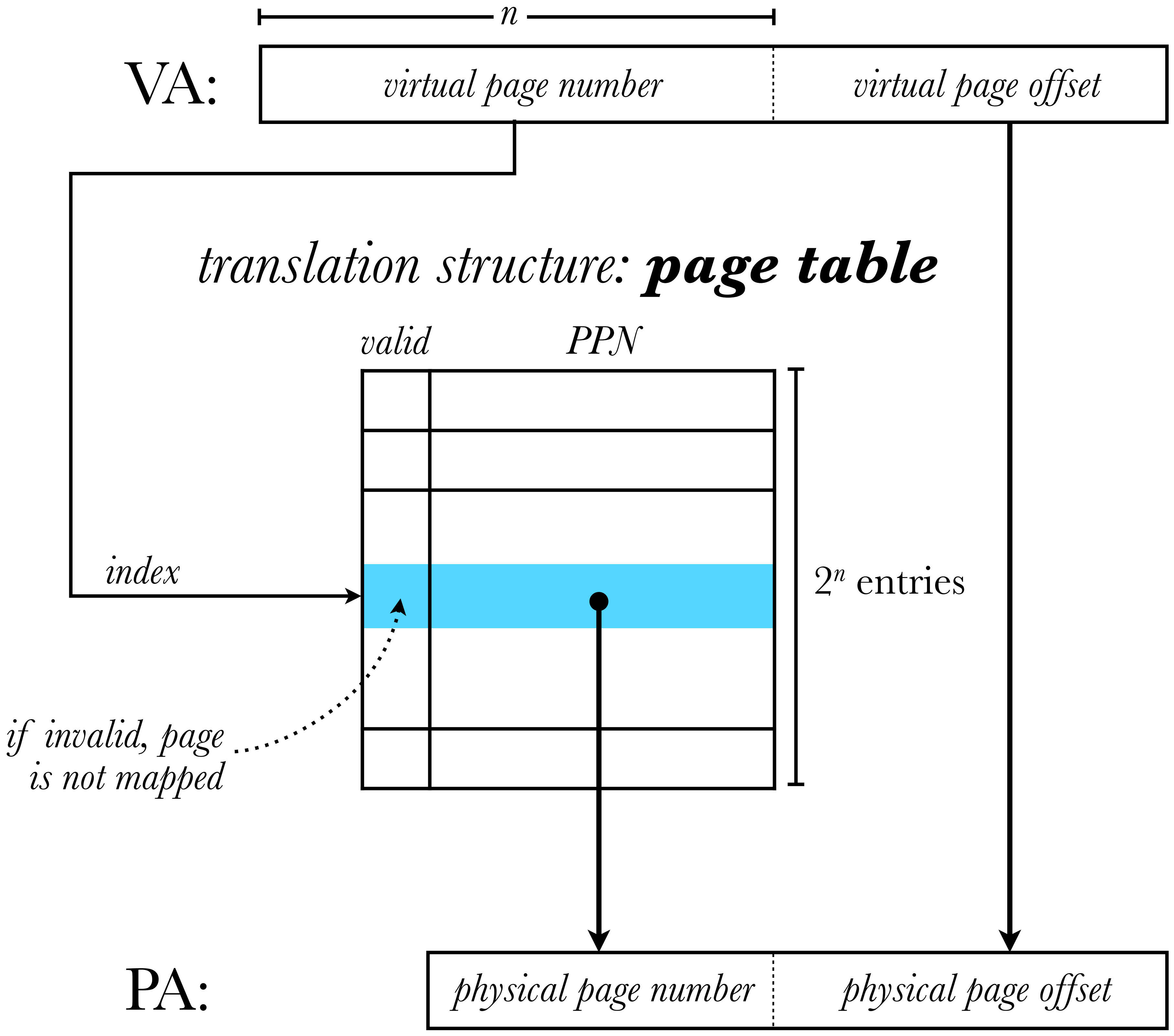
- a virtual address is broken down into *virtual page number* & *page offset*
- determine which physical page (if any) a given virtual page is loaded into
- if physical page is found, use page offset to access data

Given page size = 2^p bytes




VA:
virtual page number
virtual page offset
address translation
PA:
physical page number
physical page offset

translation structure: ***page table***



page table entries (PTEs) typically contain additional metadata, e.g.:

- dirty (modified) bit
- access bits (shared or kernel-owned pages may be read-only or inaccessible)

e.g., 32-bit virtual address,
4KB (2^{12}) page size,
4-byte PTE size;

- size of page table?

e.g., 32-bit virtual address,
4KB (2^{12}) pages,
4-byte PTEs;

- # pages = $2^{32} \div 2^{12} = 2^{20} = 1M$

- page table size = $1M \times 4$ bytes = **4MB**

4MB is much too large to fit in the MMU — insufficient registers and SRAM!

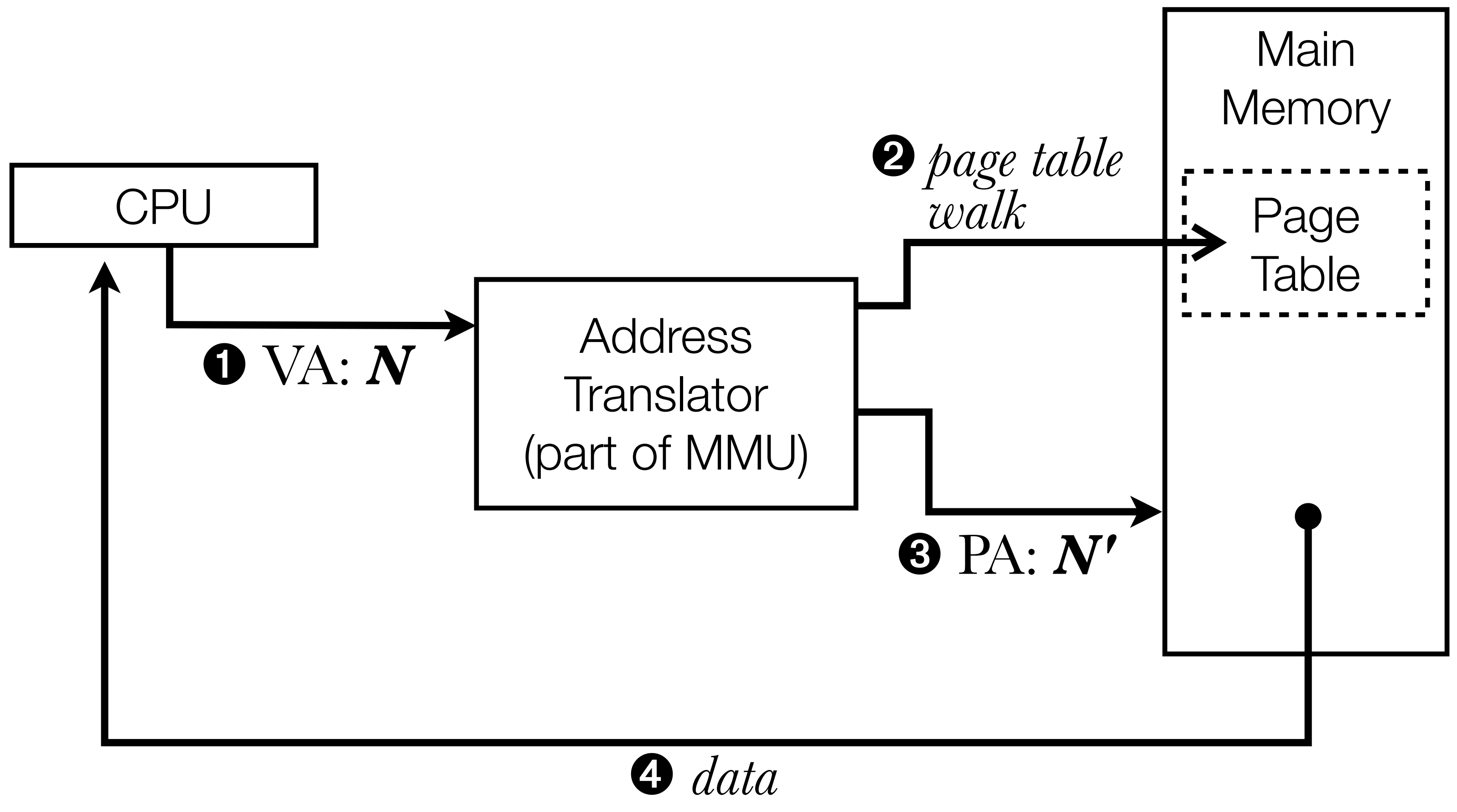
Page table resides in **main memory**

The translation process (aka *page table walk*) is performed by hardware (MMU).

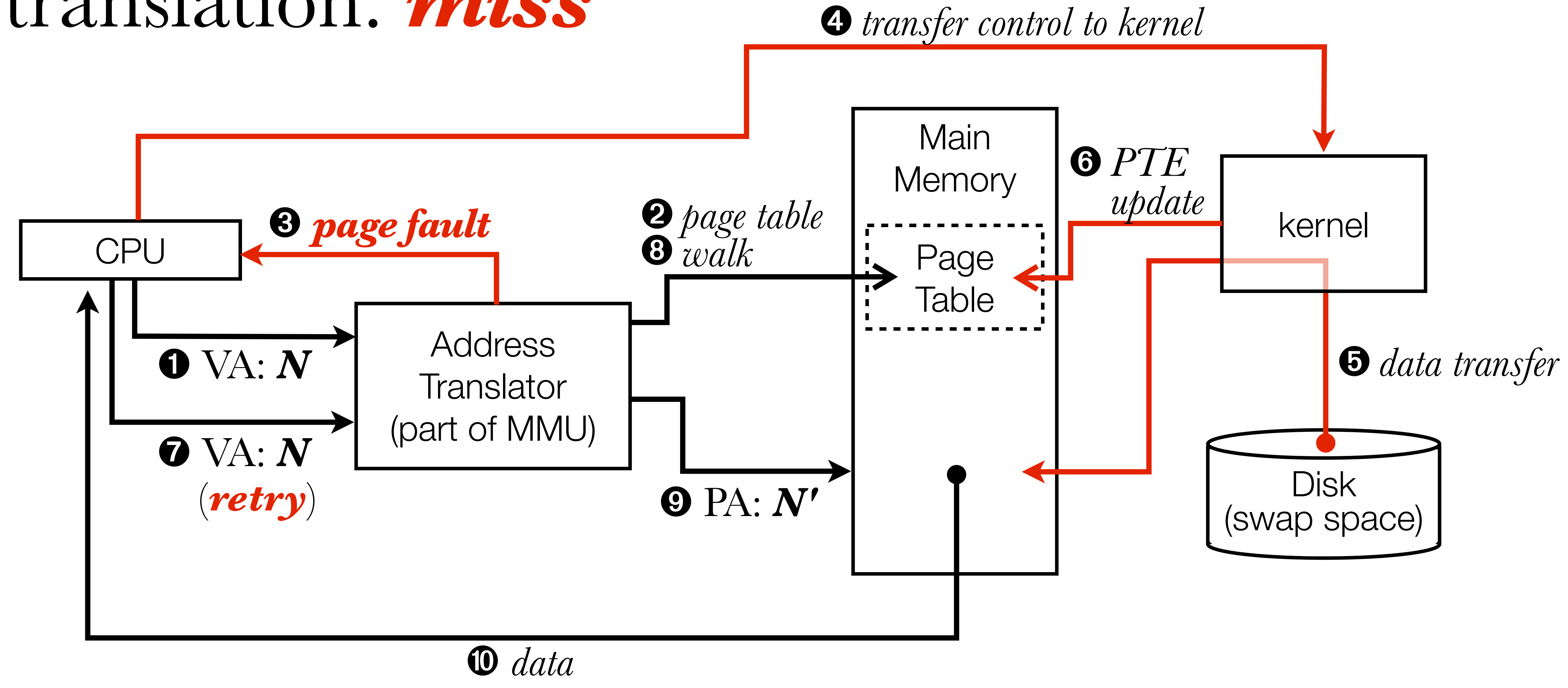
The kernel must initially populate, then continue to manage a process's page table

The kernel also populates a *page table base register* on context switches

translation: *hit*



translation: ***miss***



kernel decides where to place page, and what to evict (if memory is full)

- e.g., using LRU replacement policy

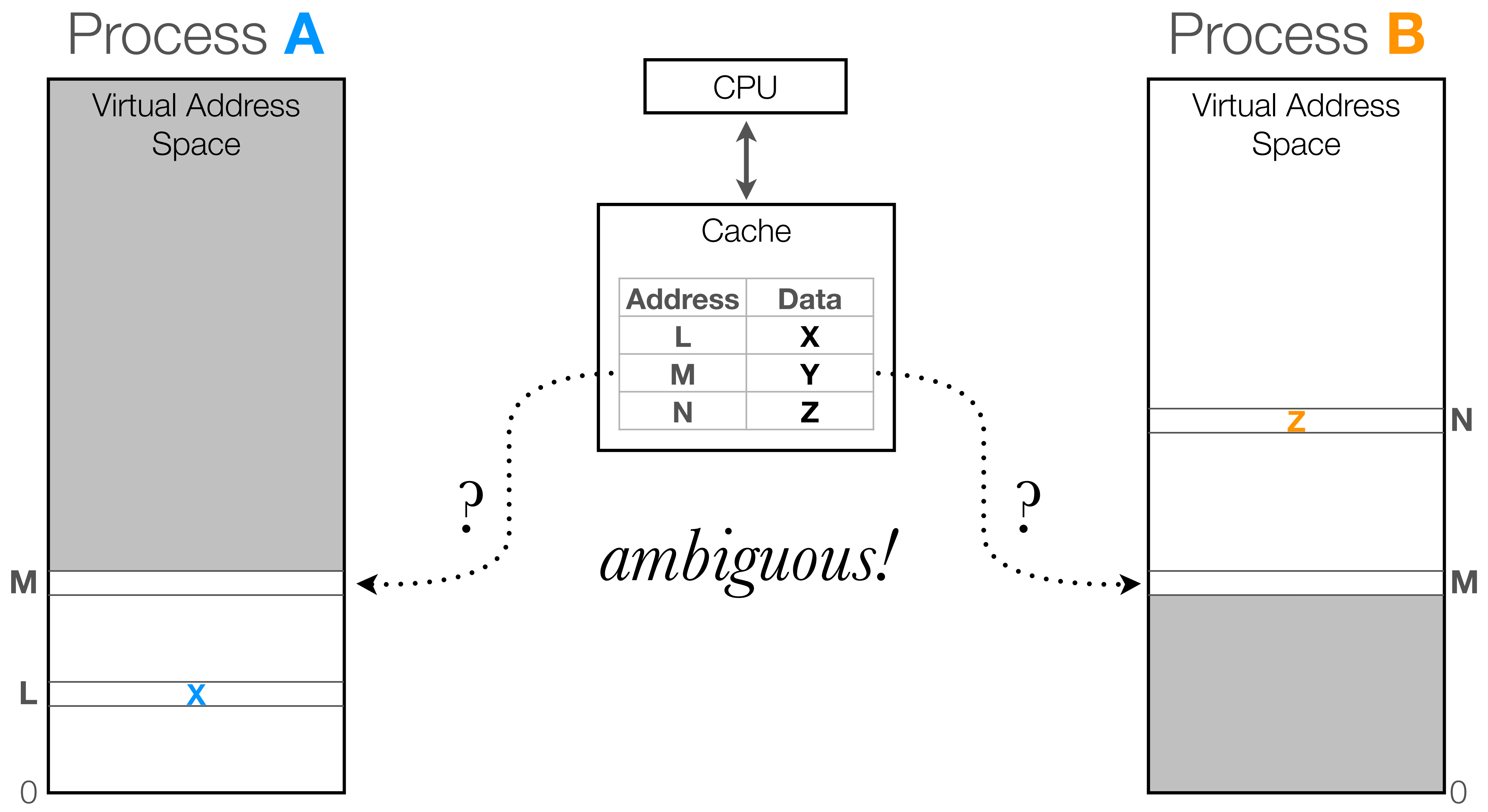
this system enables **on-demand paging**

i.e., an active process need only be partly in memory (load rest from disk dynamically) but if working set (of active processes) exceeds available memory, we may have **swap thrashing**

integration with caches?

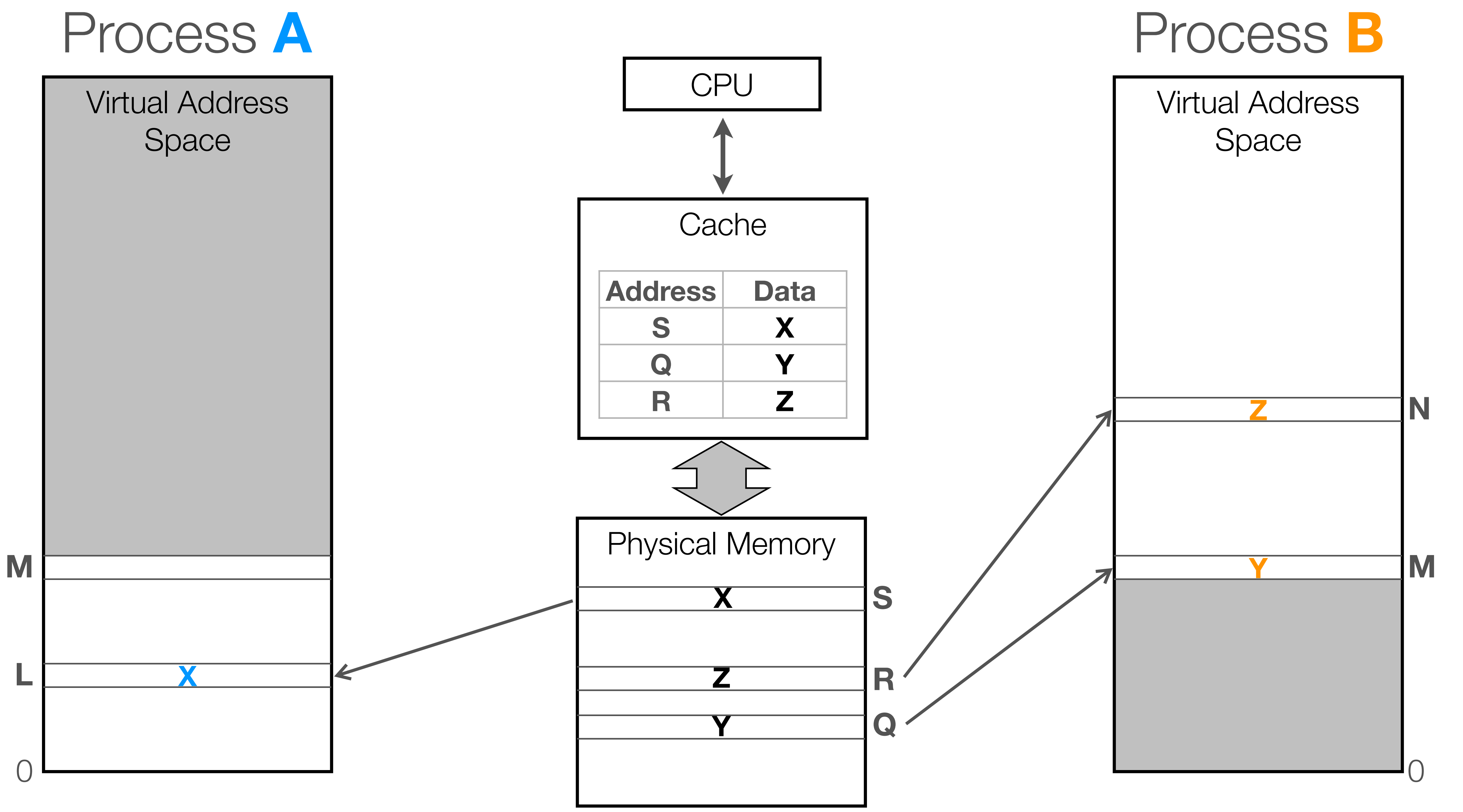
Q: do caches use physical or virtual addresses for lookups?

Virtual address based Cache



Address	Data
L	X
M	Y
N	Z

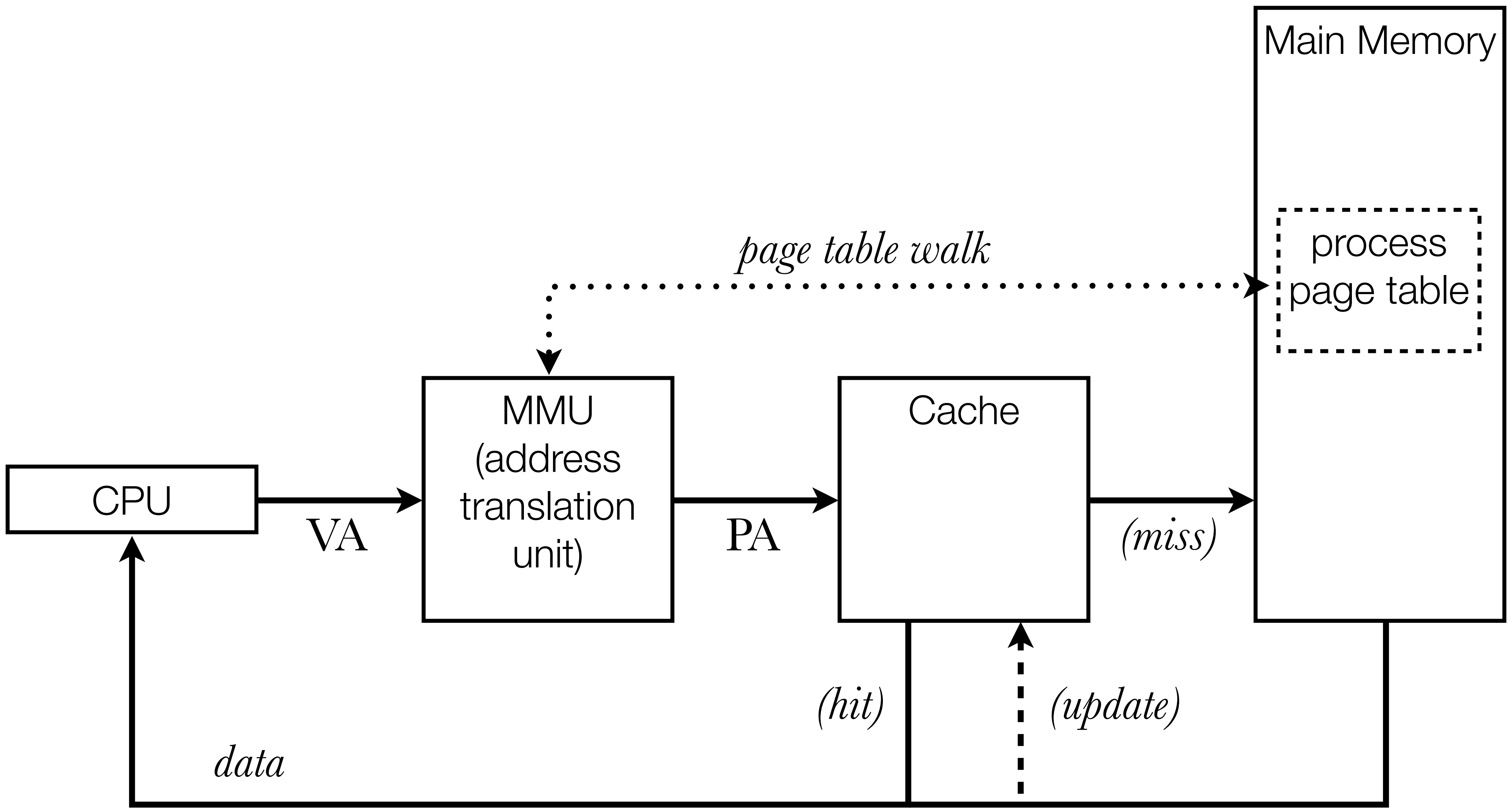
Physical address based Cache



Address	Data
S	X
Q	Y
R	Z

Q: do caches use physical or virtual addresses for lookups?

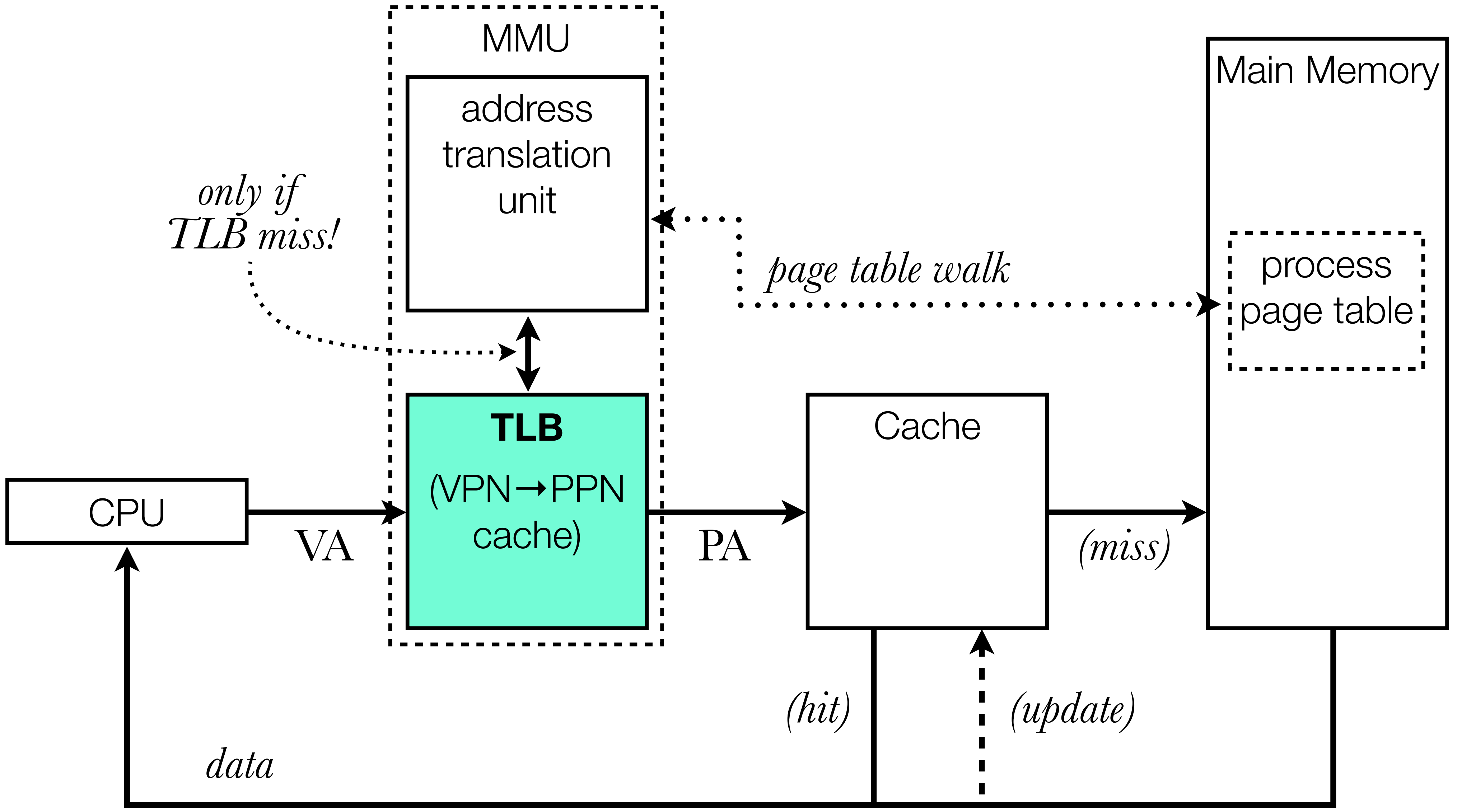
A: caches typically use *physical* addresses


Main Memory
process page table
page table walk
MMU (address translation unit)
Cache
CPU
VA
PA
(miss)
(hit)
(update)
data

%*@$&#!!!

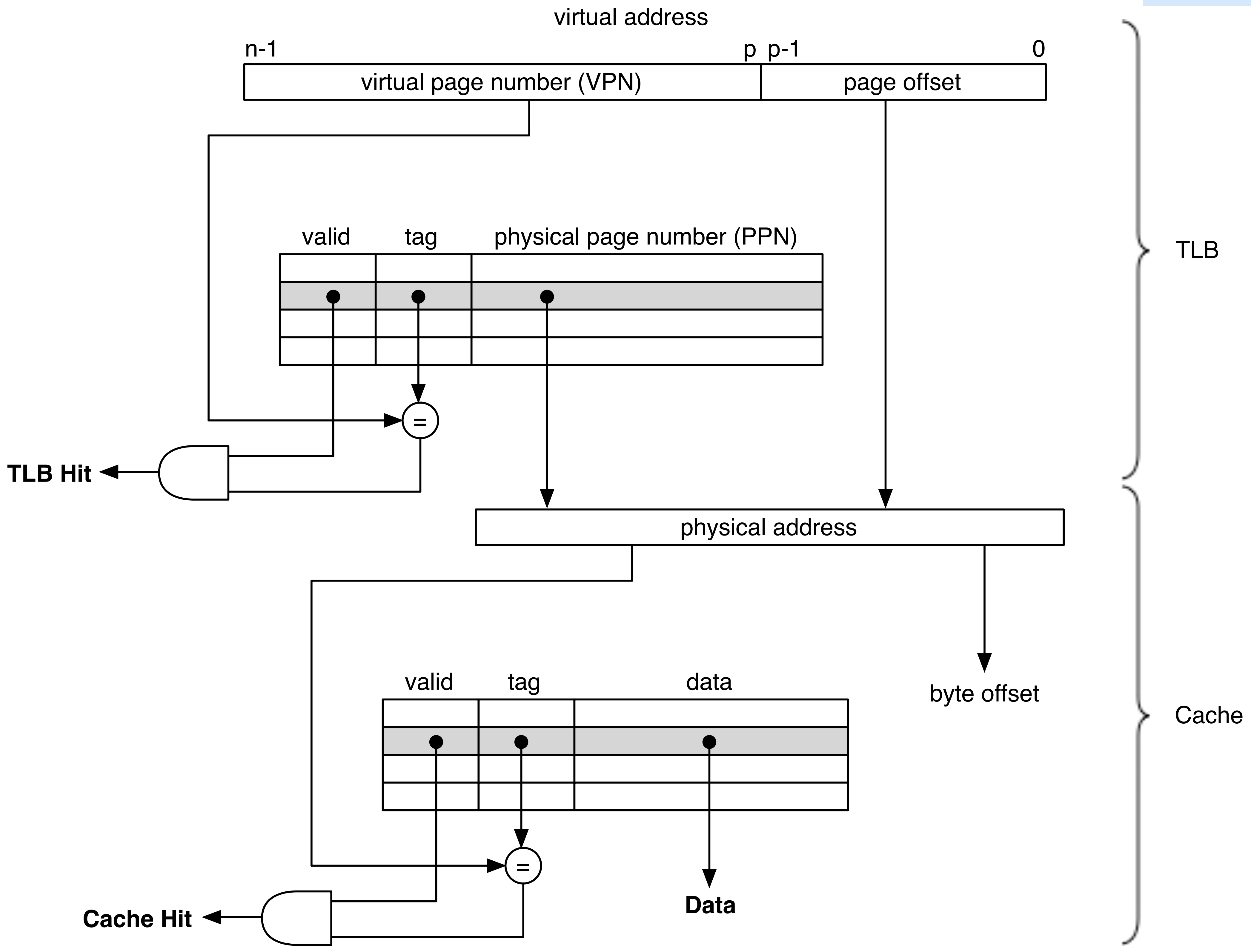
saved by hardware:

the *Translation Lookaside Buffer* (TLB) — a cache used solely for VPN→PPN lookups



TLB + Page table

(exercise for reader: revise earlier translation diagrams!)


virtual address
n-1
p
p-1
0
virtual page number (VPN)
page offset
valid
tag
physical page number (PPN)
=
TLB Hit
physical address
valid
tag
data
byte offset
=
Cache Hit
Data
TLB
Cache

TLB mappings are *process specific* — requires flush & reload on context switch

- some architectures store PID (aka "virtual space" ID) in TLB

Familiar caching problem:

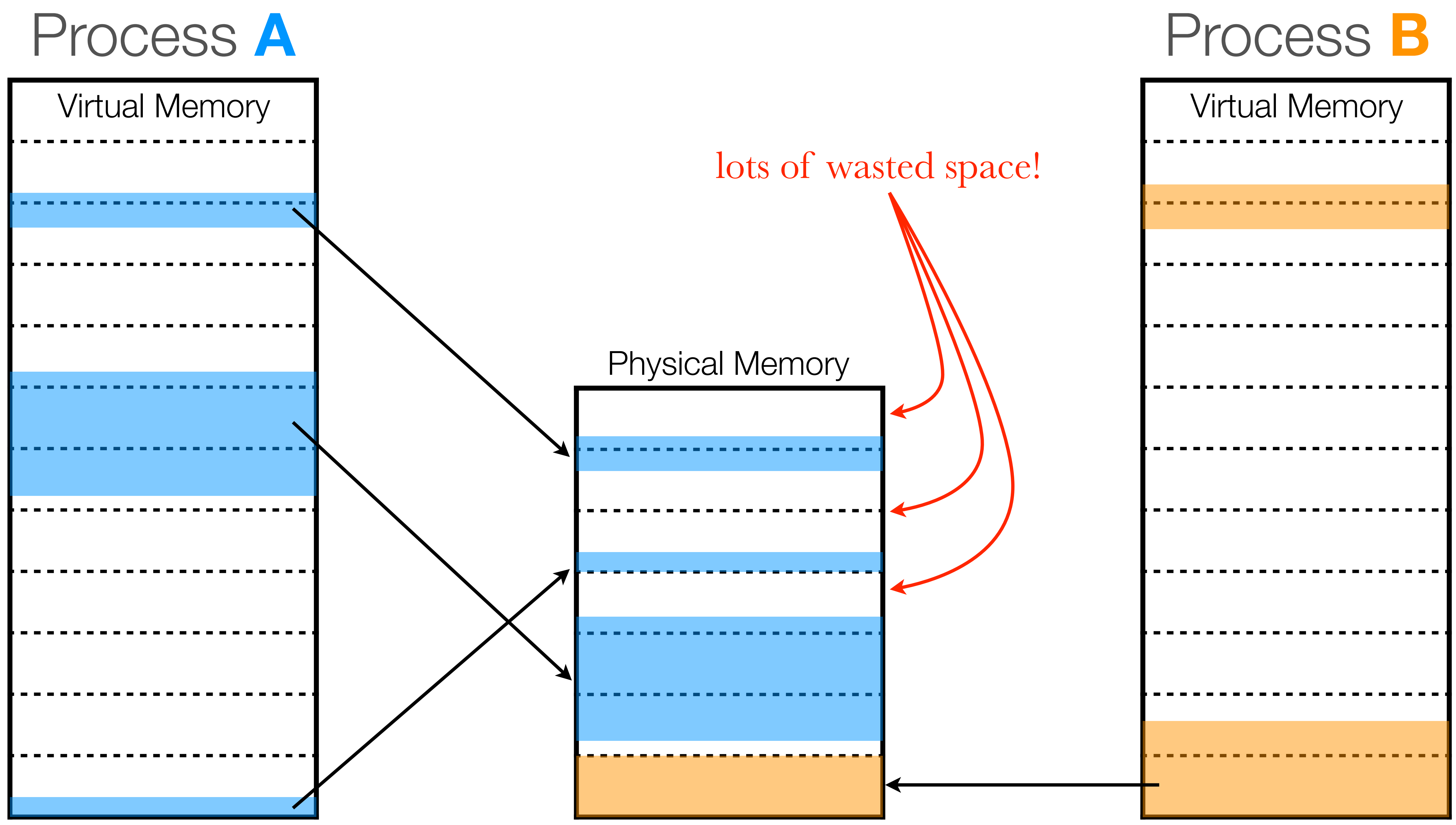
- TLB caches a few thousand mappings
- vs. *millions* of virtual pages per process!

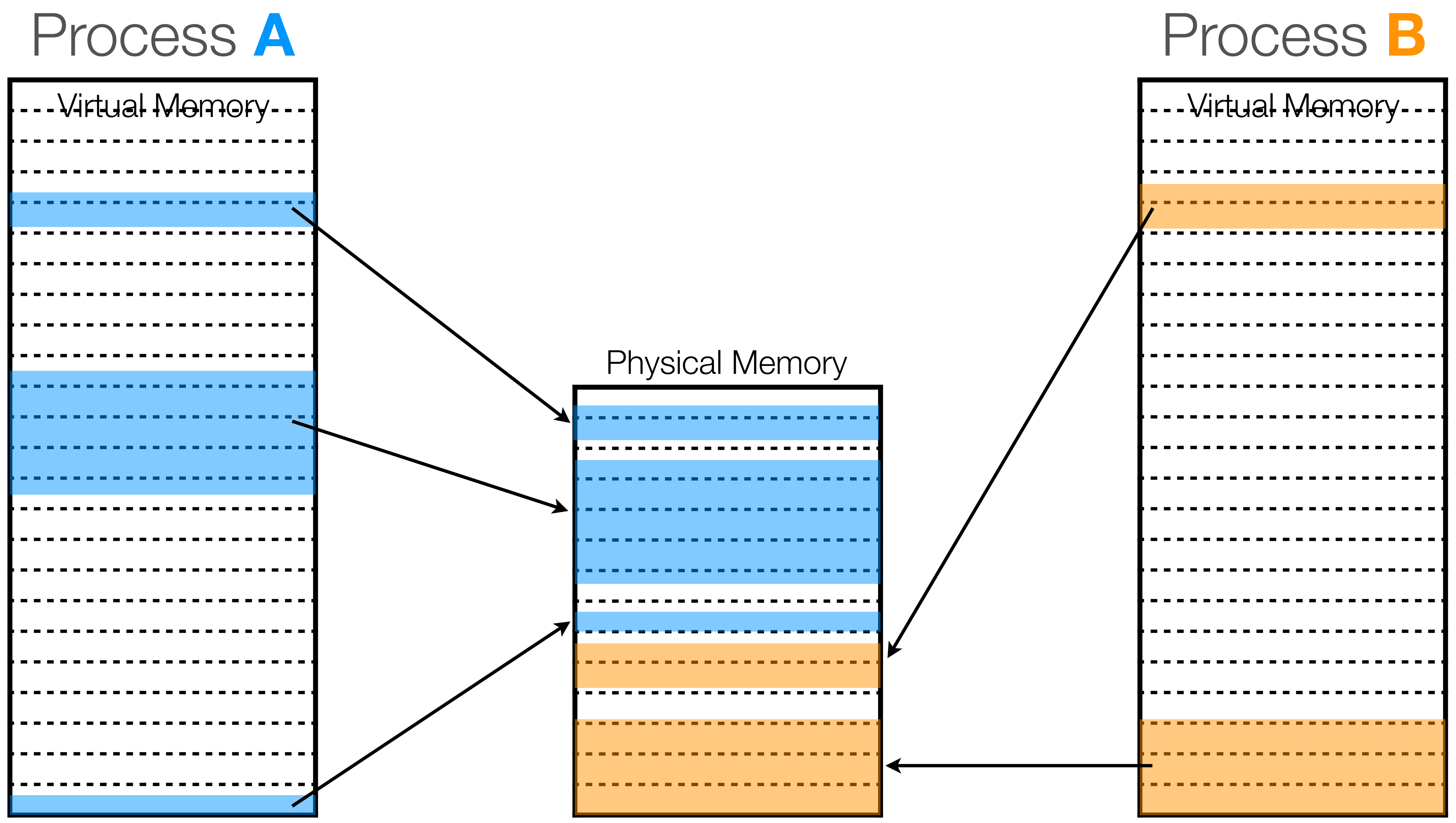
we can improve TLB hit rate by reducing the number of pages …

by increasing the size of each page

compute # pages for 32-bit memory for:

- 1KB, 512KB, 4MB pages
  - $2^{32} \div 2^{10} = 2^{22}$ = 4M pages
  - $2^{32} \div 2^{19} = 2^{13}$ = 8K pages
  - $2^{32} \div 2^{22} = 2^{10}$ = 1K pages (not bad!)


Process A
Virtual Memory
Process B
Virtual Memory
Physical Memory
lots of wasted space!


Process A
Virtual Memory
Physical Memory
Process B
Virtual Memory

increasing page size results in increased *internal fragmentation* and *lower utilization* i.e., TLB effectiveness needs to be balanced against memory utilization so what about 64-bit systems?

2^{64} = 16 Exabyte address space

$\approx$ 4 billion x 4GB

most modern implementations support a max of 2^{48} (256TB) addressable space page table size (assuming 4K page size)?

- # pages $= 2^{48} \div 2^{12} = 2^{36}$
- PTE size $= 8$ bytes (64 bits)
- PT size $= 2^{36} \times 8 = 2^{39}$ bytes

  = **512GB**

512GB

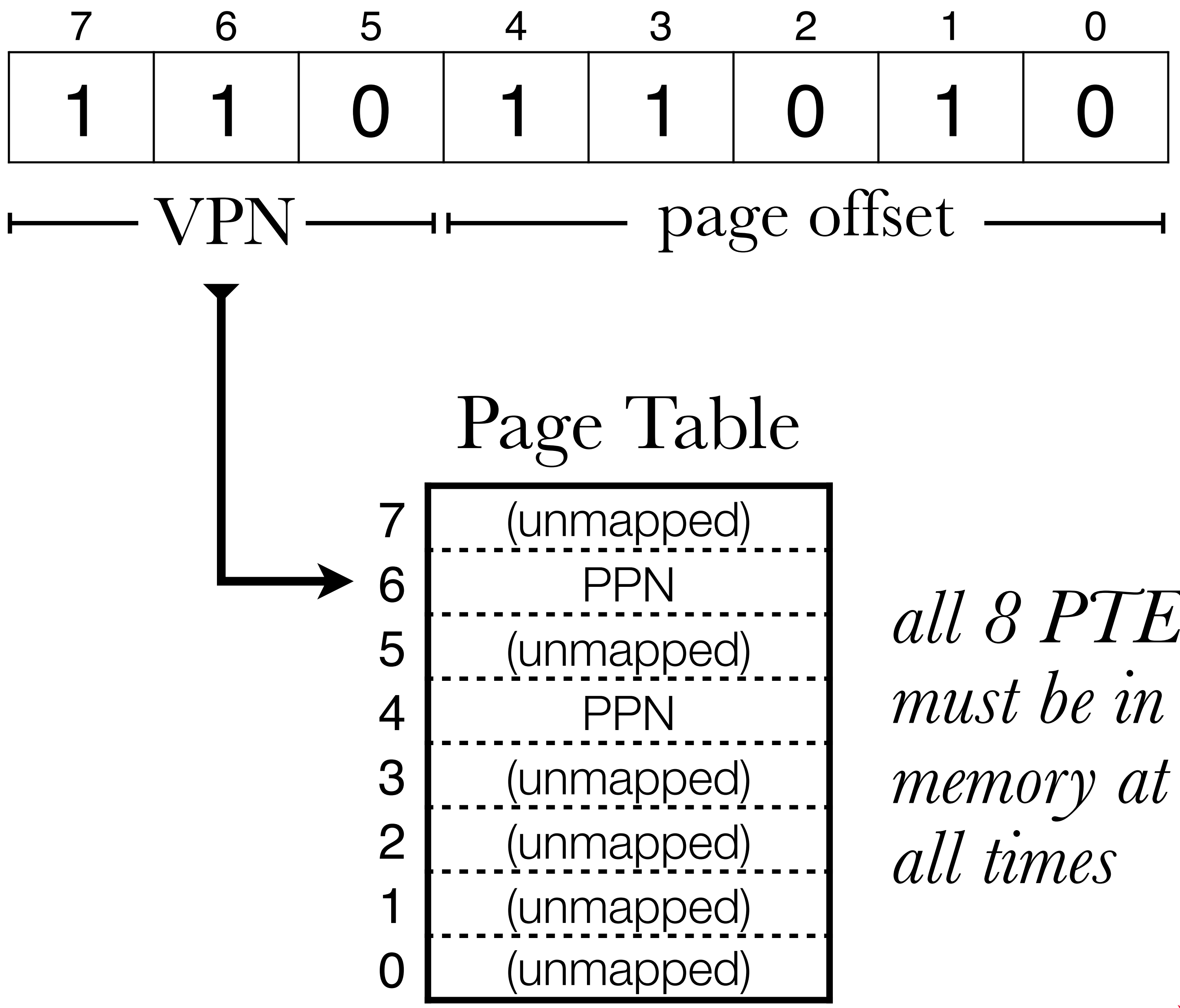
(just for the virtual memory *mapping* structure)

(and we need *one per process*)

(these things aren’t going to fit in memory) instead, use *multi-level* page tables:

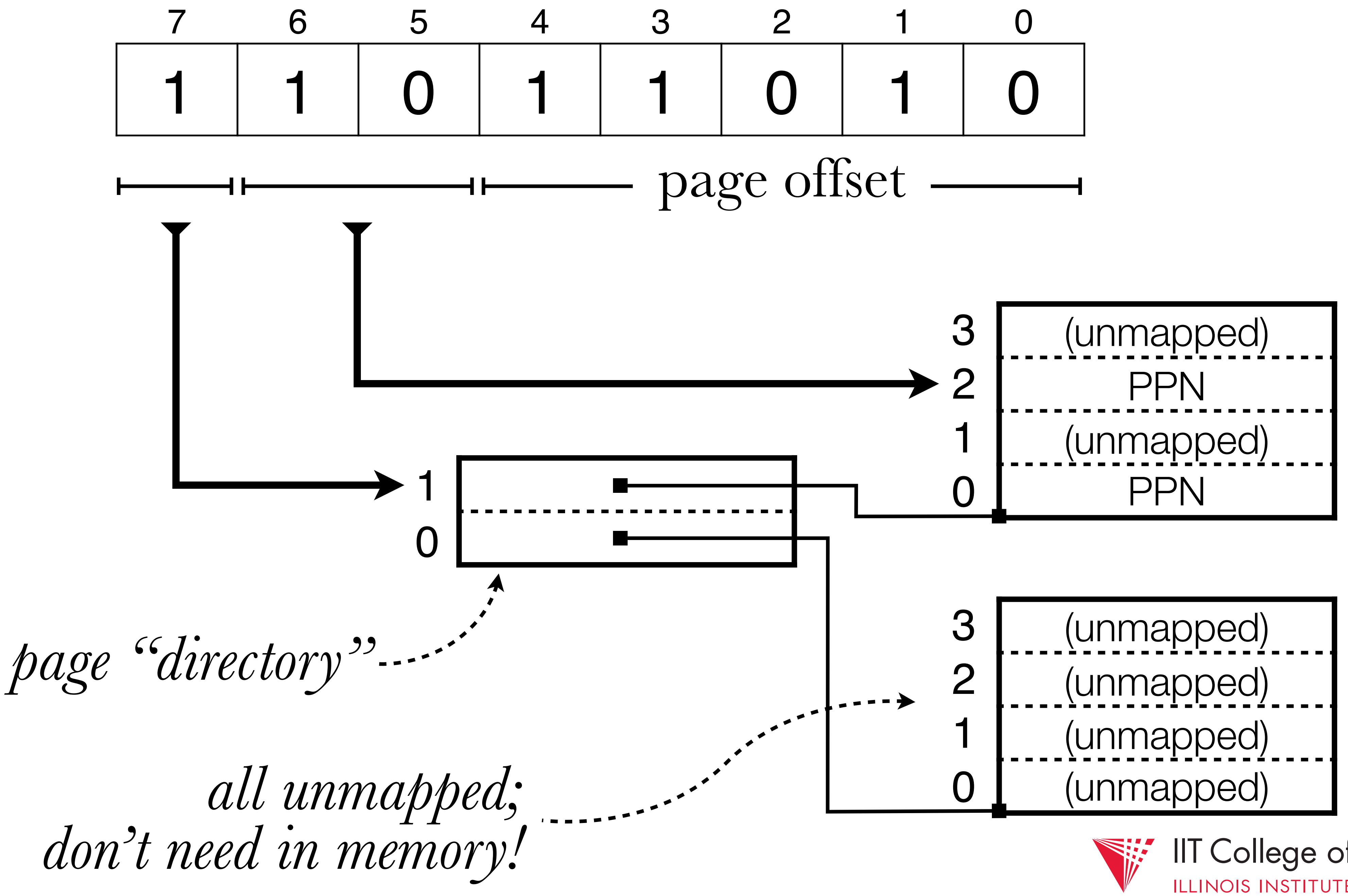
- split an address translation into two (or more) separate table lookups
- unused parts of the table don't need to be in memory!

"toy" memory system
- 8 bit addresses
- 32-byte pages

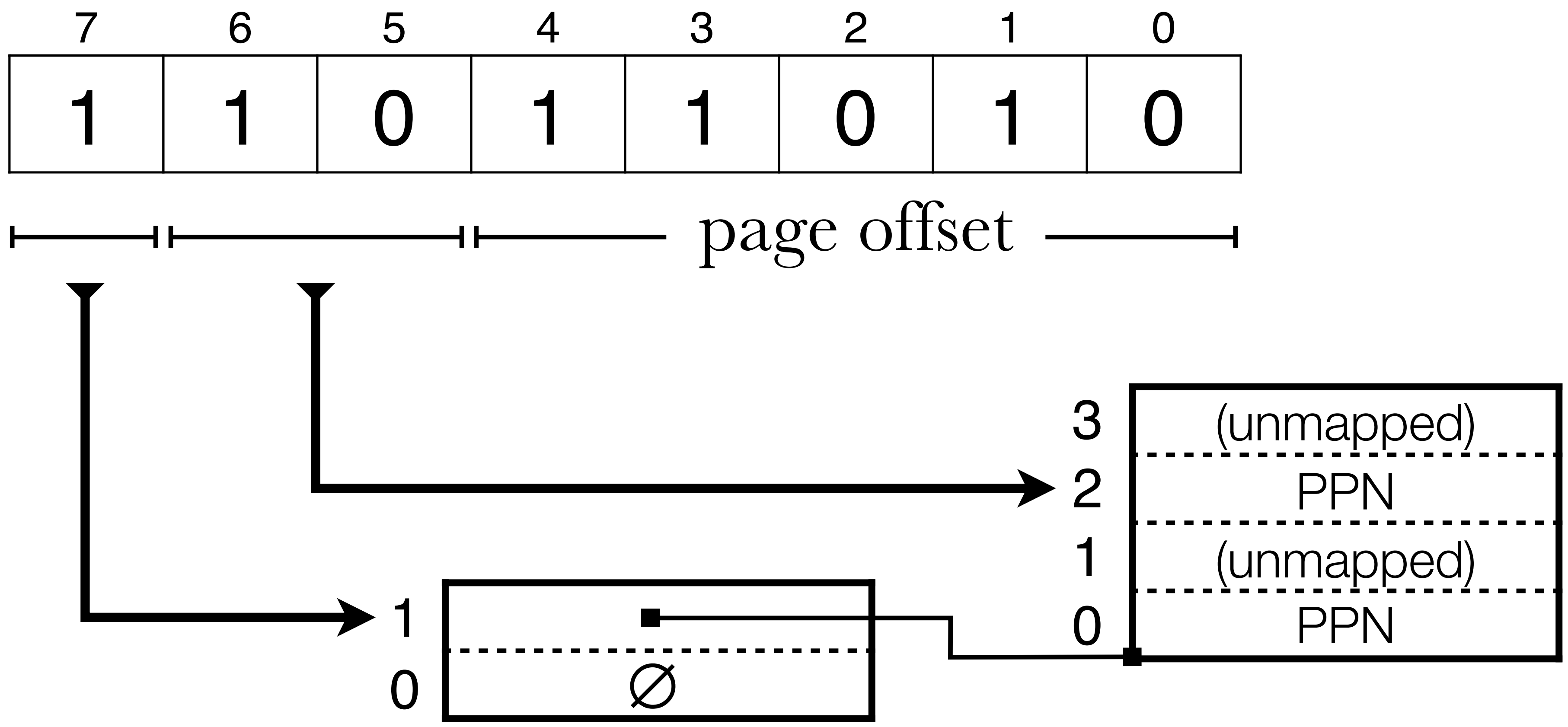


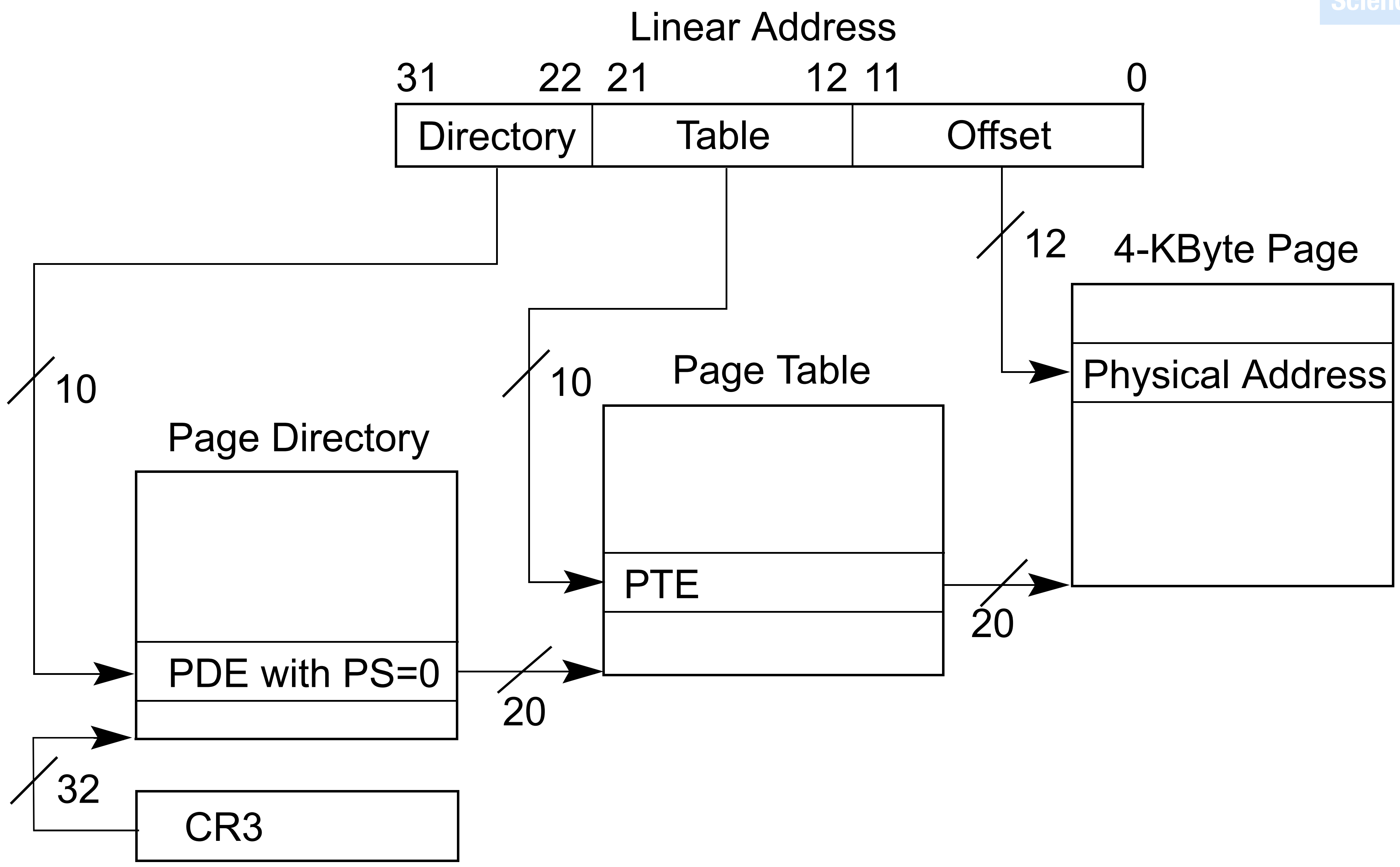
all 8 PTEs must be in memory at all times

"toy" memory system
- 8 bit addresses
- 32-byte pages

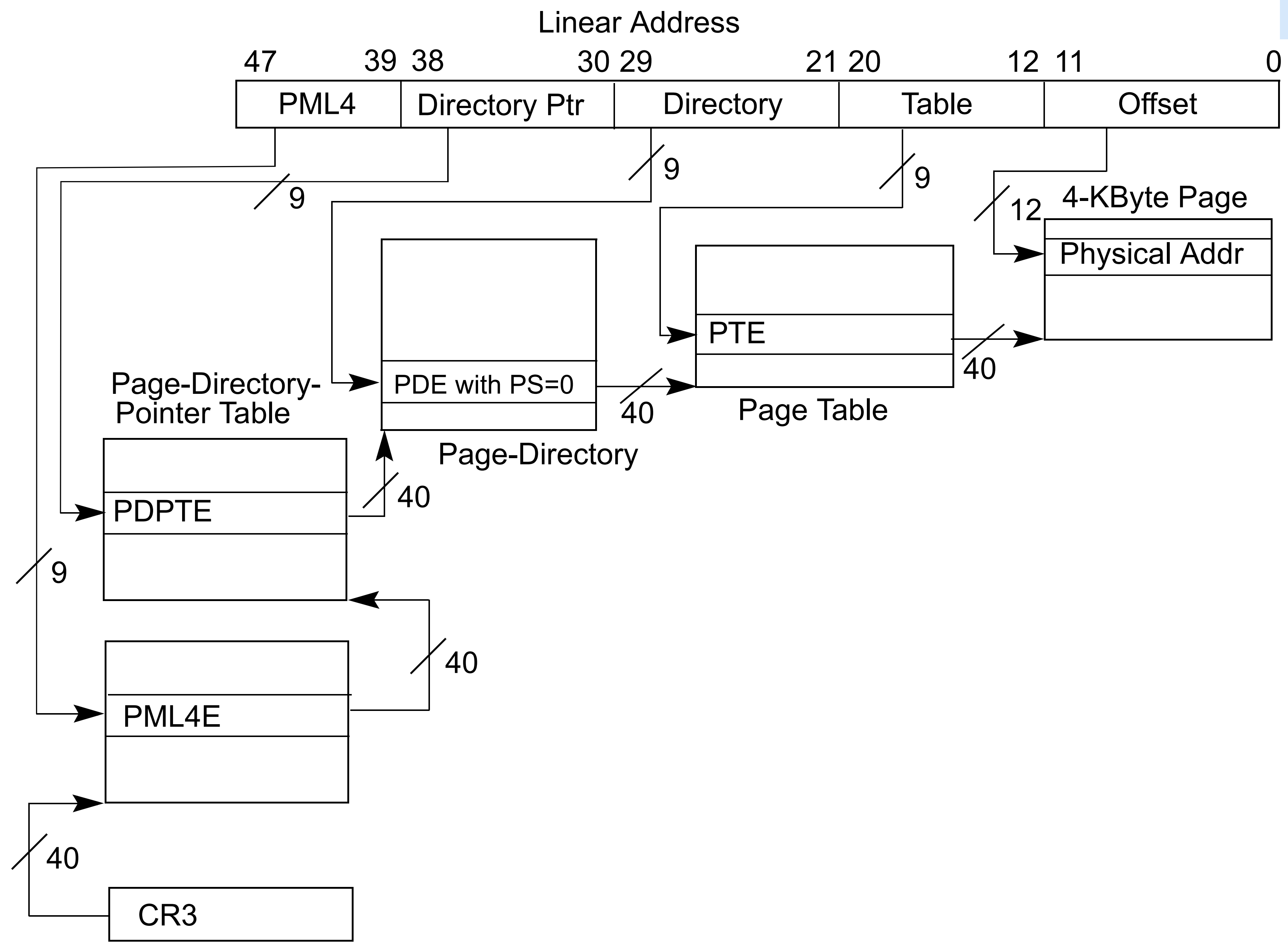


"toy" memory system
- 8 bit addresses
- 32-byte pages





IA-32 paging (4KB pages)



x86-64 paging (4KB pages)