CSE 451: Operating Systems

Lab Section: Week 6

Today

- Project 3
- Virtual Address Spaces
    - Part II: fun virtual memory tricks
- Paging

(I have no idea what's on tomorrow's quiz ☹)

Project 3

- Due Wednesday, Feb 16 at 11:59pm
  - next week!
- Questions?

A rant on code optimization

"premature optimization is the root of all evil"

- Donald Knuth

- Write the simple version first
  - the profile (this tells you where time is spent)
  - then optimize
- You don't need to super optimize every line of code!

“Numbers every engineer should know”

(from Jeff Dean, Google)

not so important

Simple instruction	<1 ns
L1 cache reference	<1 ns
Main memory reference	100 ns
Mutex lock/unlock	100 ns
Compress 1Kb of data	10,000 ns

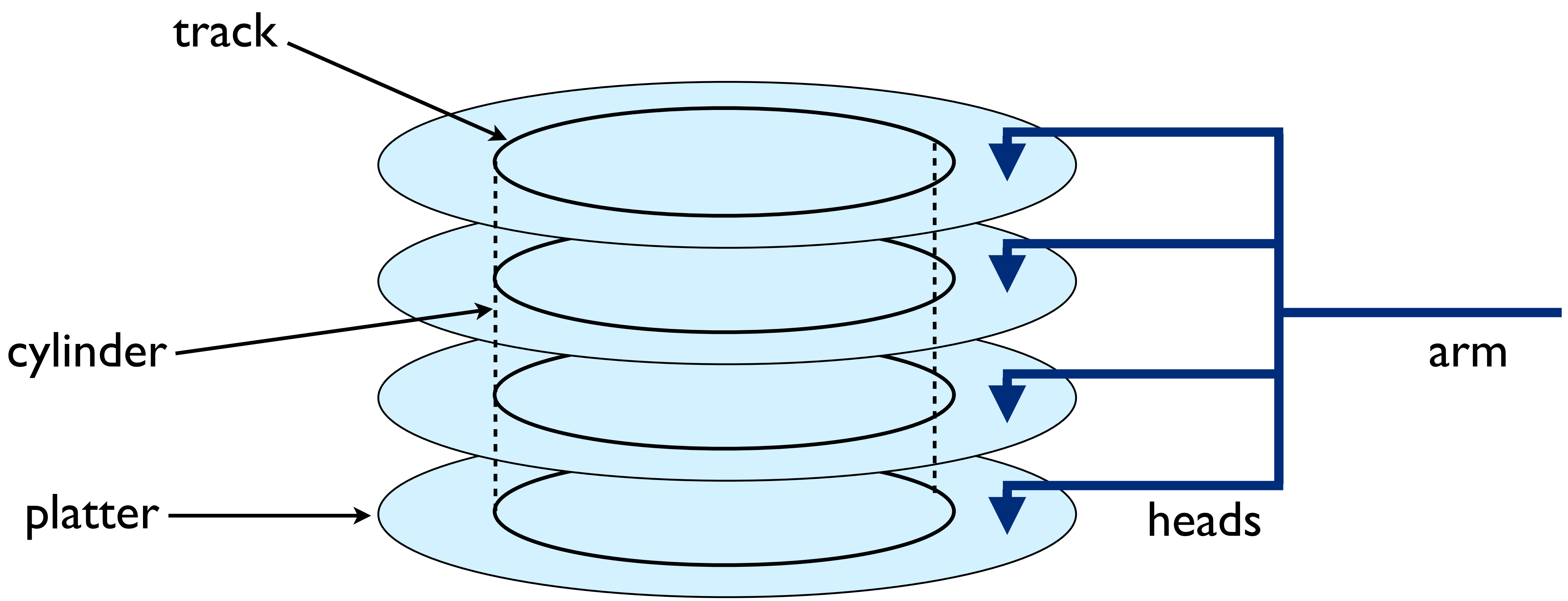
important for Project 3

Send 2Kb over local network	20,000 ns
Read 1Mb sequentially from flash drive	5,000,000 ns
Read 1Mb sequentially from network	10,000,000 ns
Disk seek (random access)	10,000,000 ns
Read 1 Mb sequentially from disk	30,000,000 ns
Send packet CA→Netherlands→CA	250,000,000 ns

flash numbers added by me

Hard disk geometry

(what is a seek?)



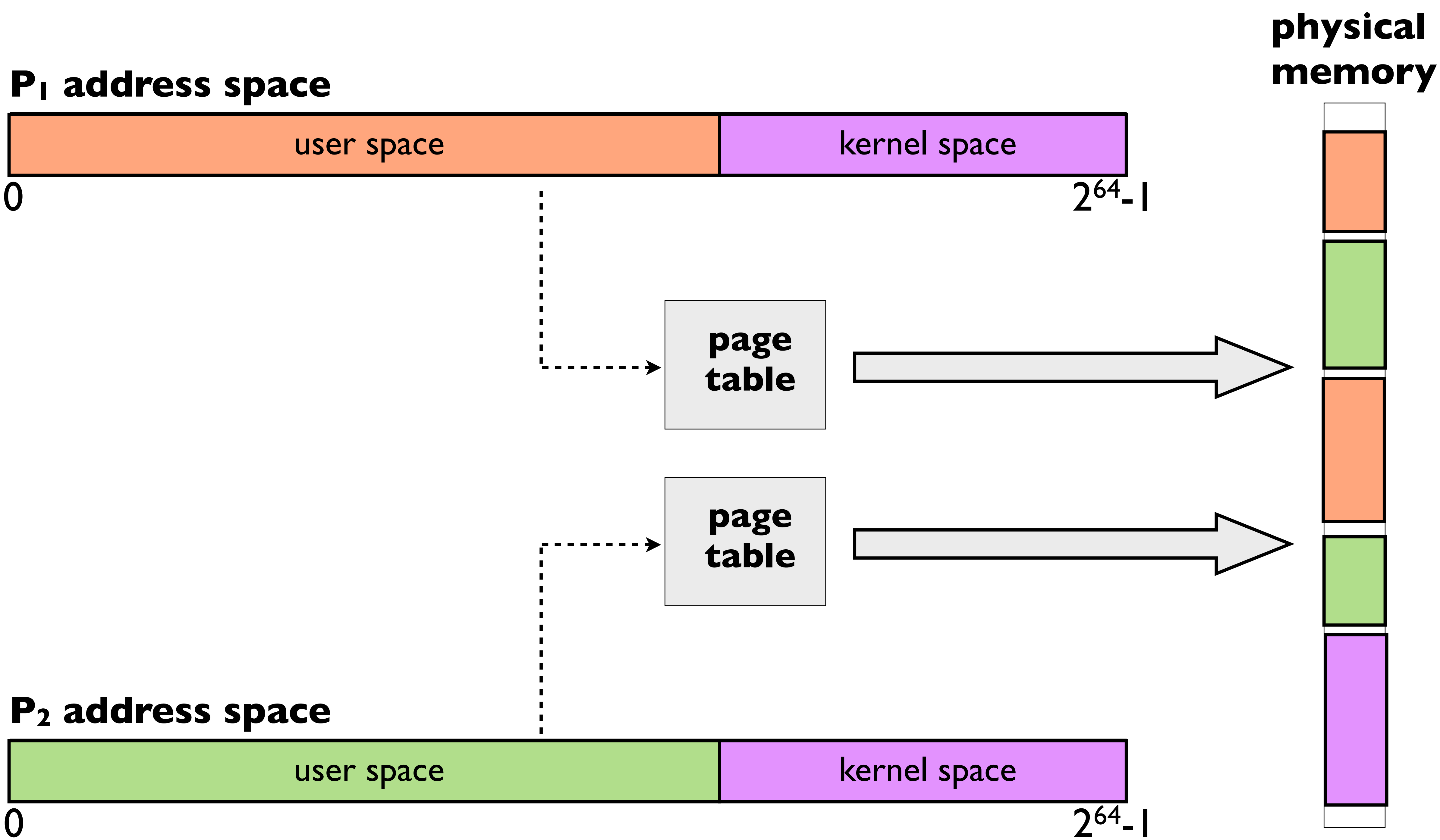
seek: moving the arm

Today

- ~~Project 3~~
- Virtual Address Spaces
    - Part II: fun virtual memory tricks
- Paging

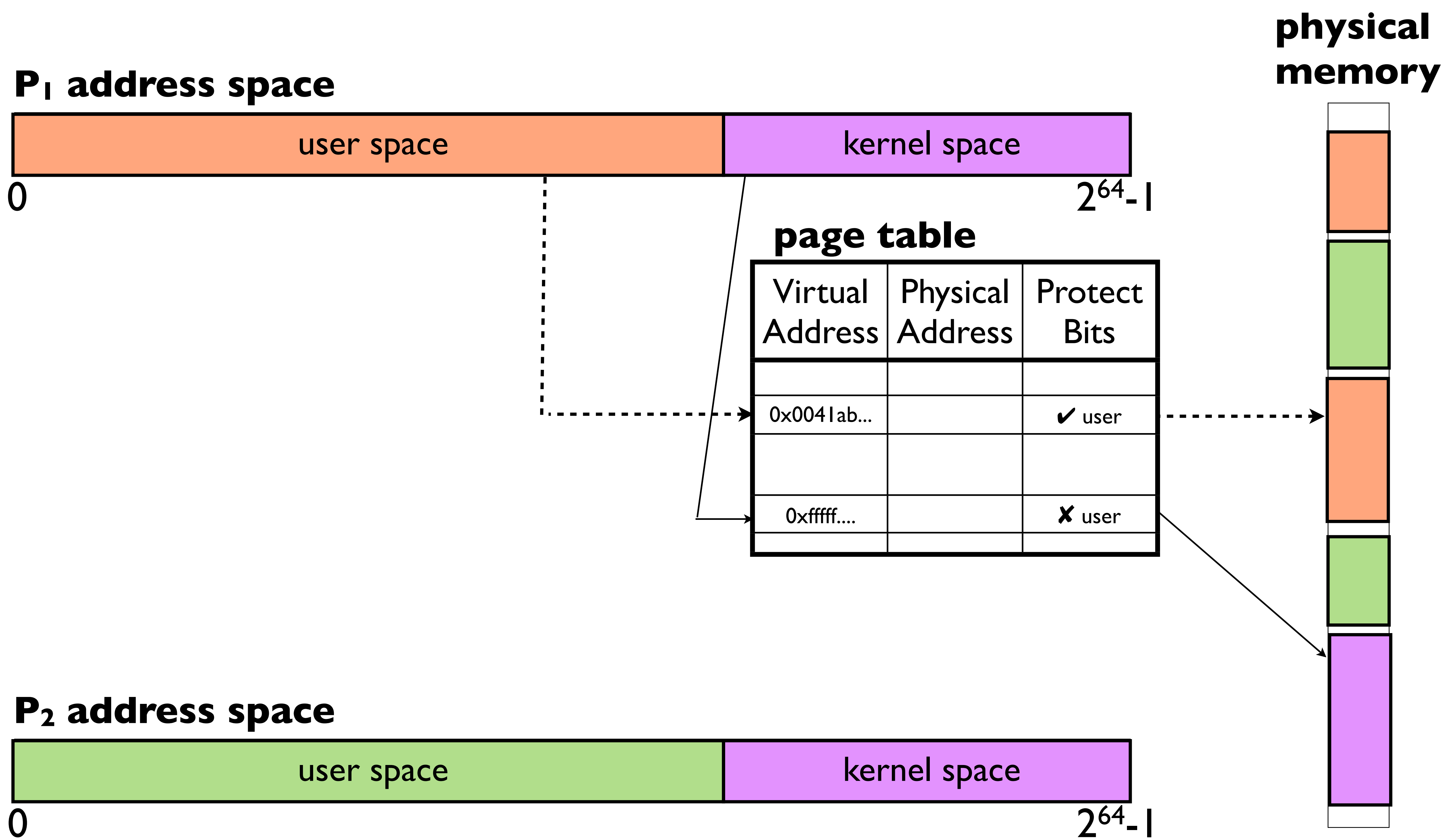
Virtual Address Spaces

(review)



Virtual Address Spaces

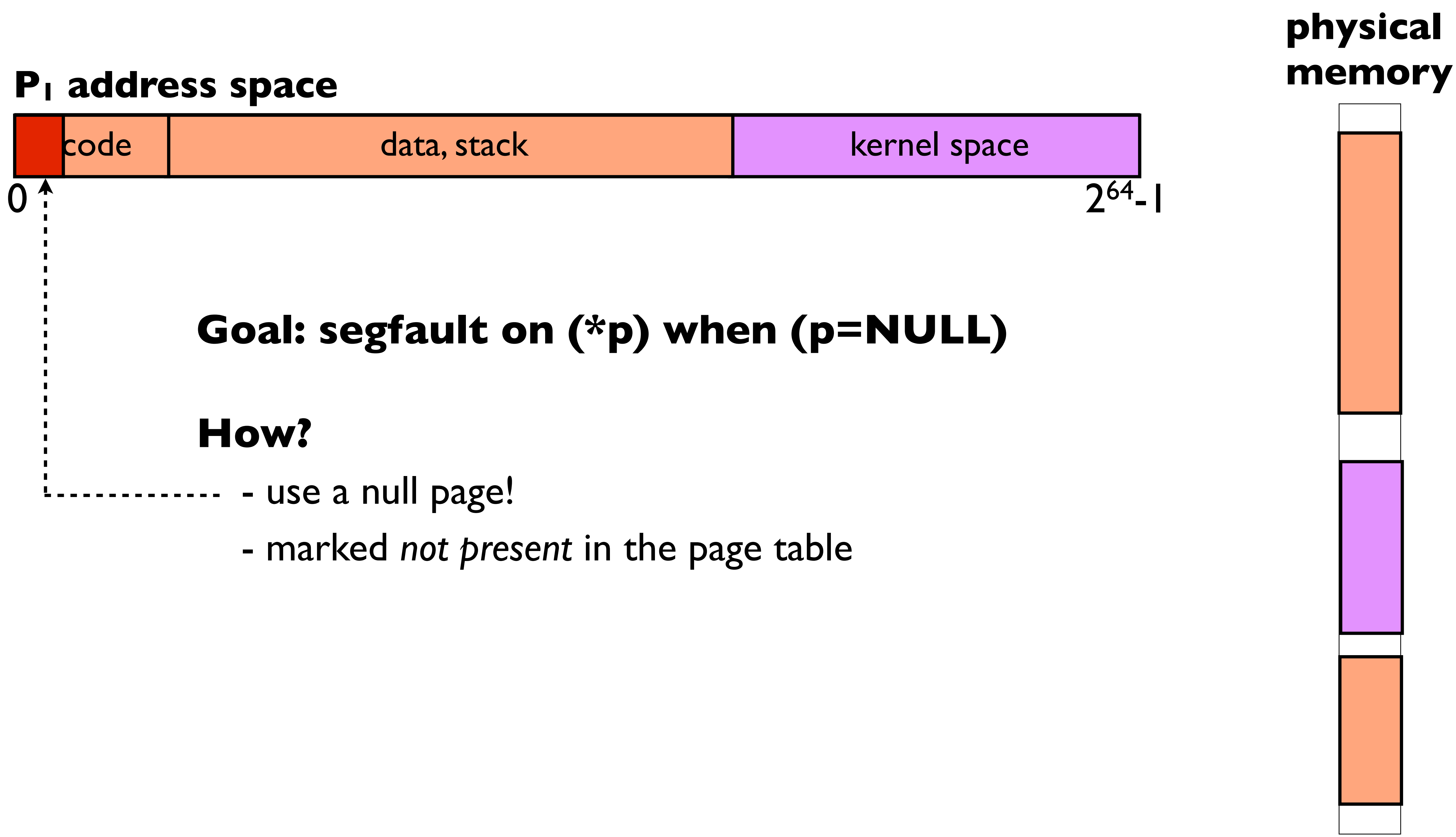
(review)



Page table protection bits

- **user** bit
  - we just saw this
  - used to hide kernel pages from user programs
- **present** bit
  - is there a physical page allocated for this virtual address?
- **writable** bit
  - is the page writable?
  - when unset, the page is *read-only* (we'll see this in a bit)
- What if a protection bit is violated?
  - hardware triggers a *page fault*
  - OS decides what to do

VM trick (1): NULL pointers

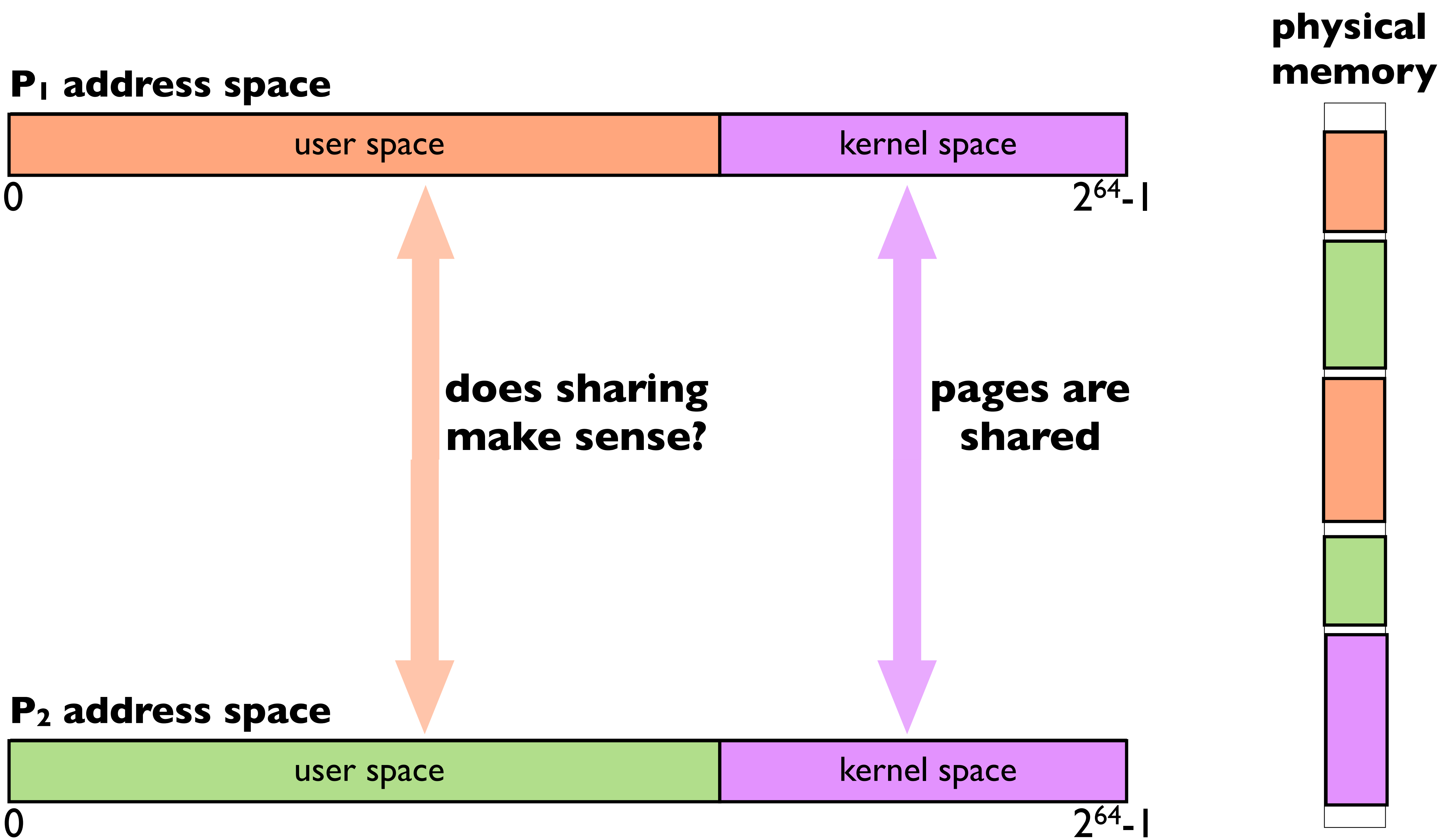


Goal: segfault on (*p) when (p=NULL)

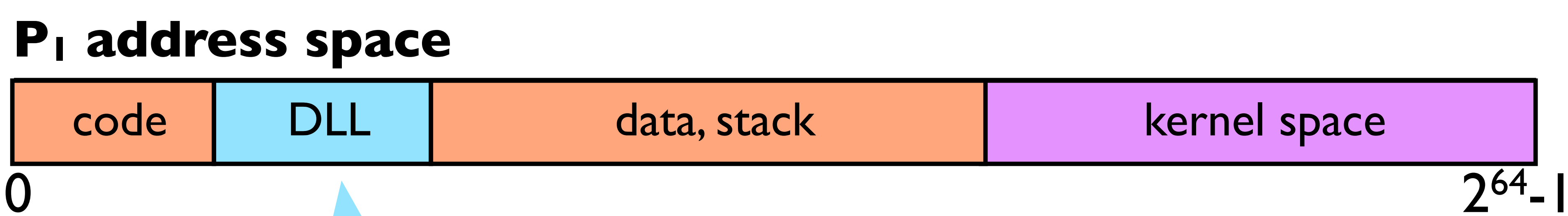
How?

- use a null page!
- marked *not present* in the page table

VM trick (2): sharing



VM trick (2): sharing

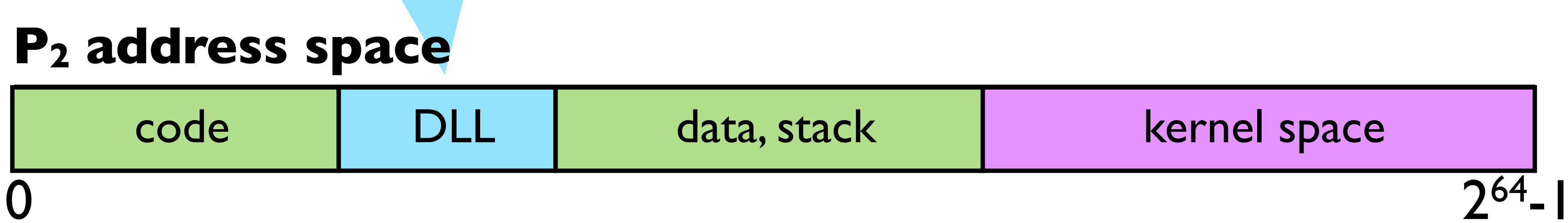


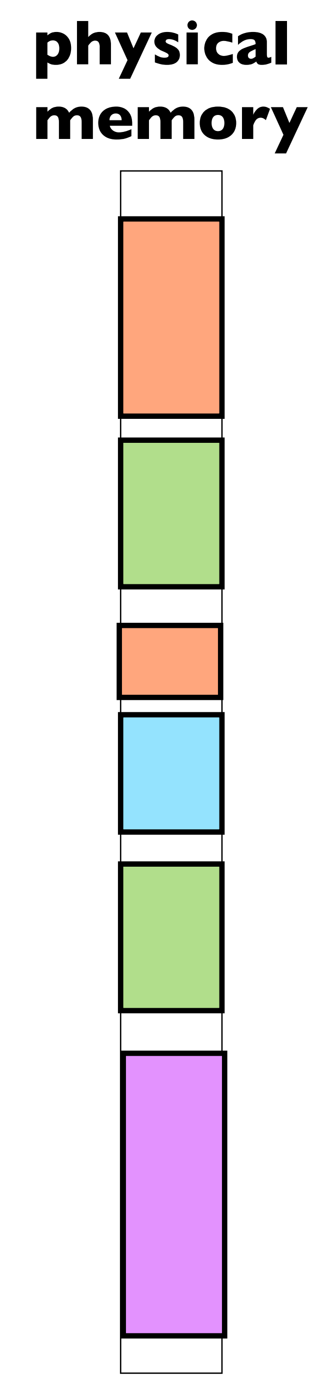
A shared library:

- share code pages in multiple address spaces (saves space!)

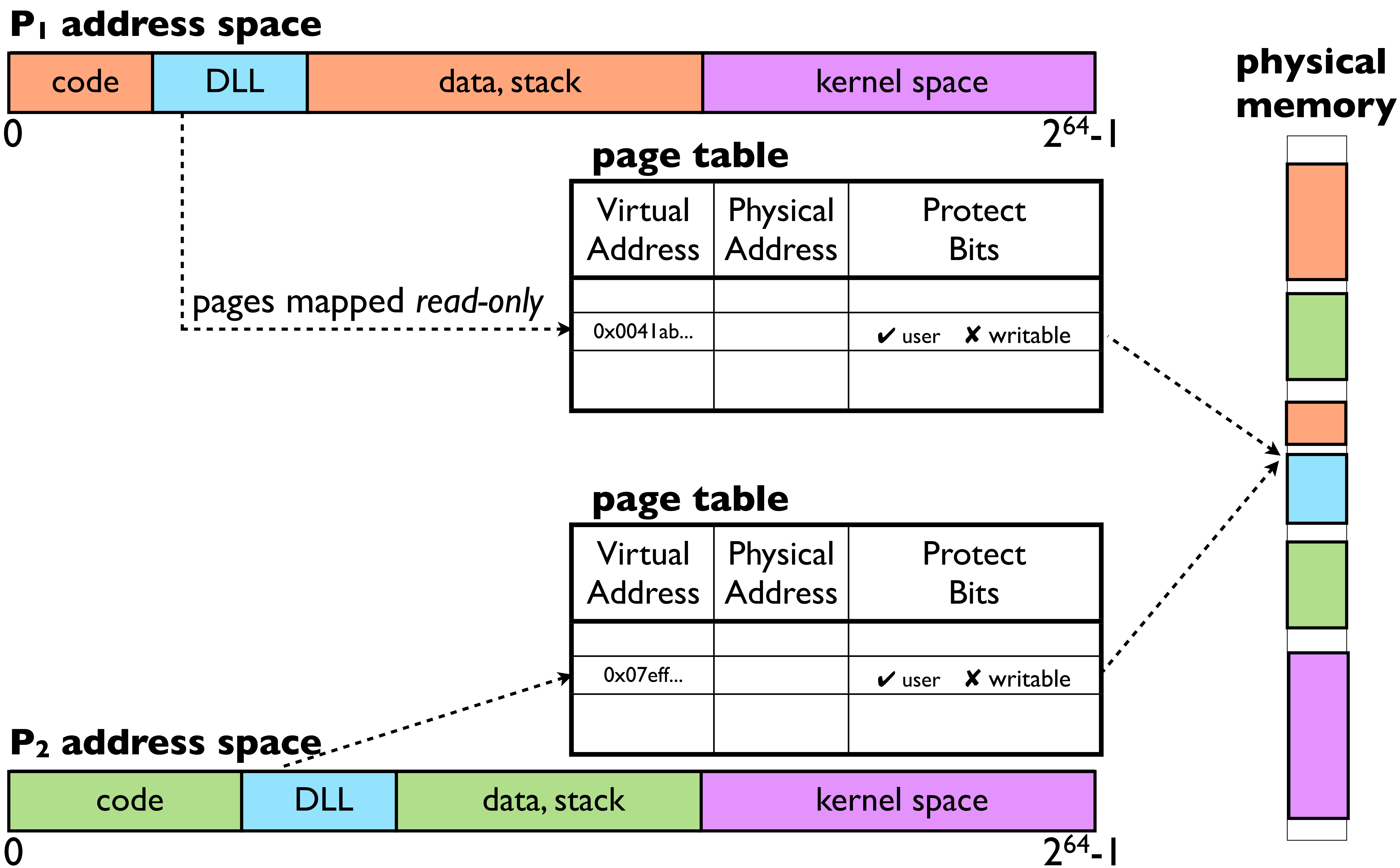
Problem: can't let P_2 write to P_1's DLL!

- solution: map pages *read-only*

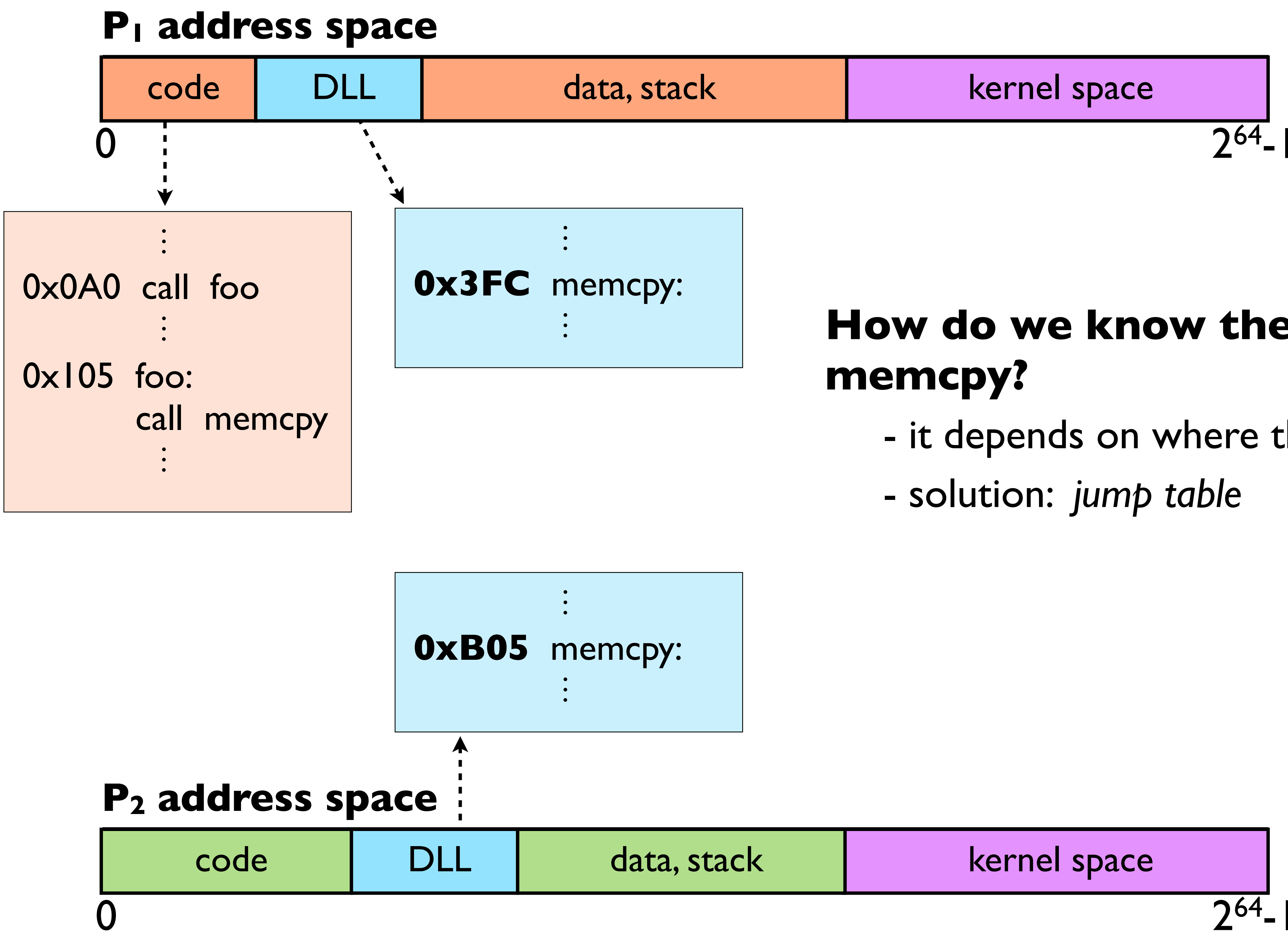




VM trick (2): sharing



VM trick (2): sharing

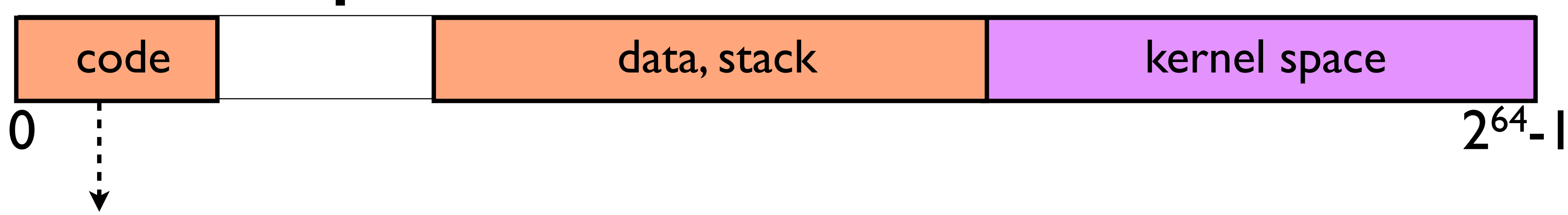


How do we know the address of memcpy?

- it depends on where the DLL was loaded!
- solution: *jump table*

VM trick (2): sharing

P_1 address space



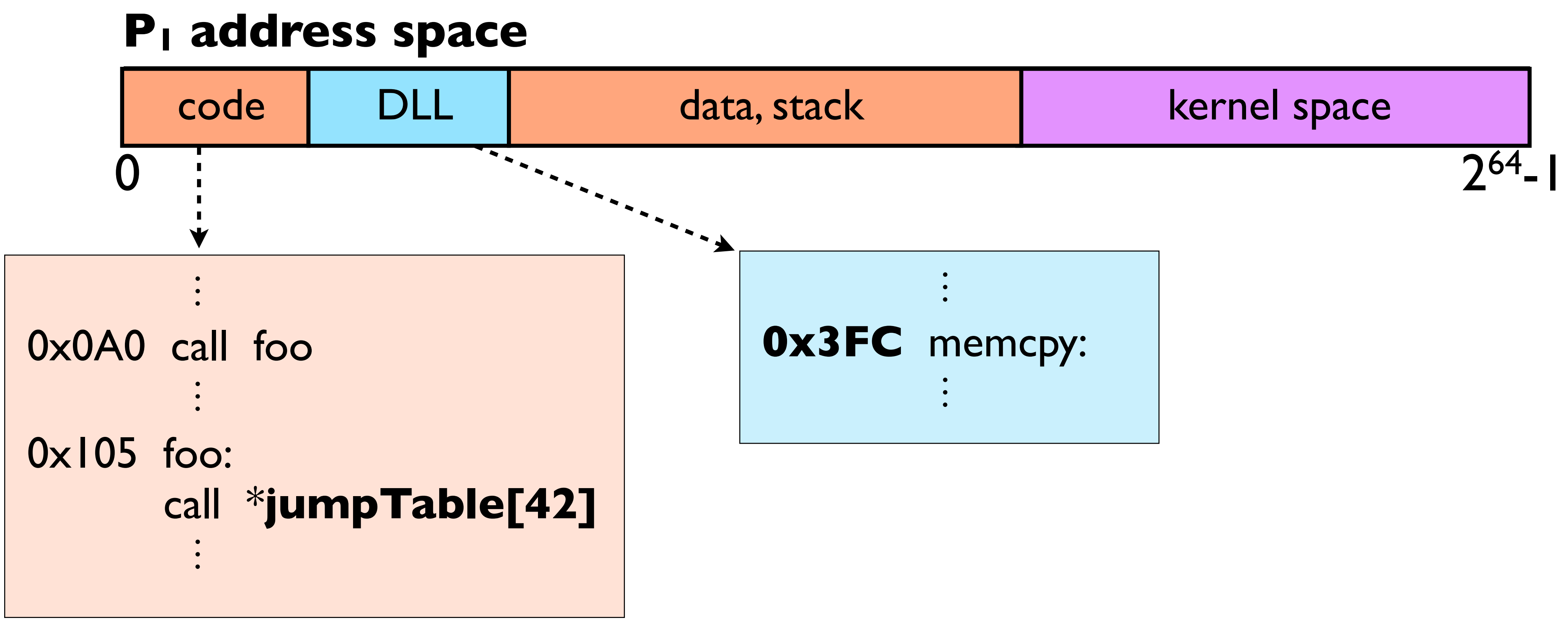

```
        ⋮
0x0A0  call  foo
        ⋮
0x105  foo:
       call  *jumpTable[42]
        ⋮
```

Library call indirects through *jump table*

```
jumpTable = {
   [0] = ?
   [1] = ?
     ⋮
  [42] = ?
     ⋮
}
```

Jump table initially empty

VM trick (2): sharing

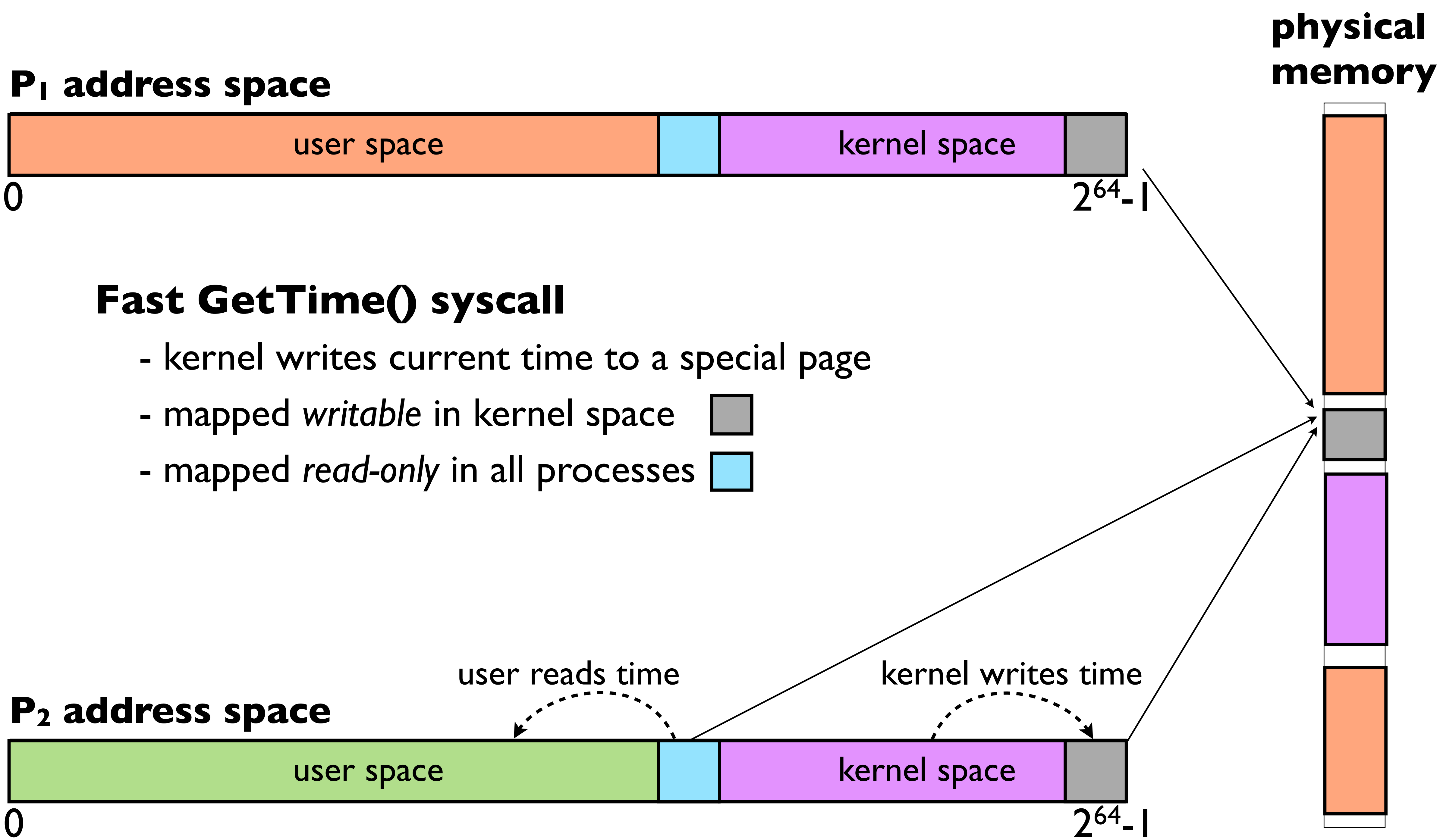



```
jumpTable = {
   [0] = ?
   [1] = ?
    ⋮
  [42] = &memcpy,
    ⋮      0x3FC
}
```

Jump table fixed when DLL is loaded

- by a program called a *loader*

VM trick (3): fast system calls



Fast GetTime() syscall

- kernel writes current time to a special page
- mapped *writable* in kernel space
- mapped *read-only* in all processes

VM trick (4): fork

P_1 address space

user space | kernel space

0 … 2^{64}-1

The UNIX fork() syscall

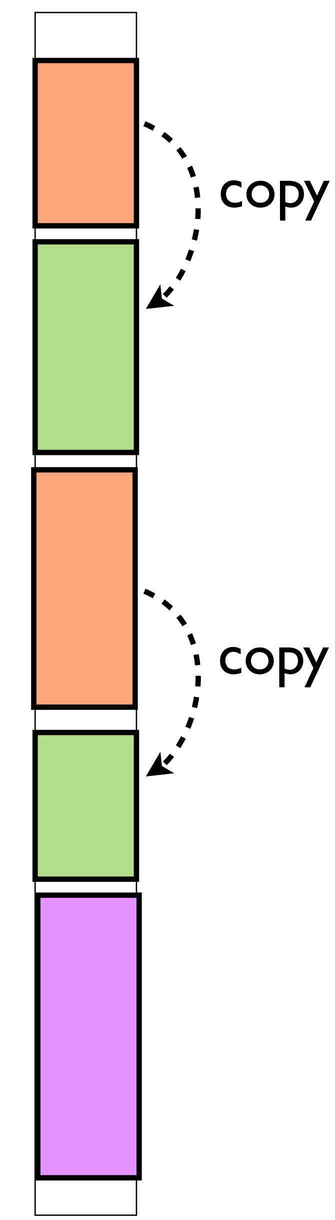
```
r = fork()  // spawns a new process
            //  as a copy of this one
if (r > 0)
    // in the parent (P1)
else if (r == 0)
    // in the child (P2)
```

P_2 address space

user space **(copy of P_1)** | kernel space

0 … 2^{64}-1

physical memory

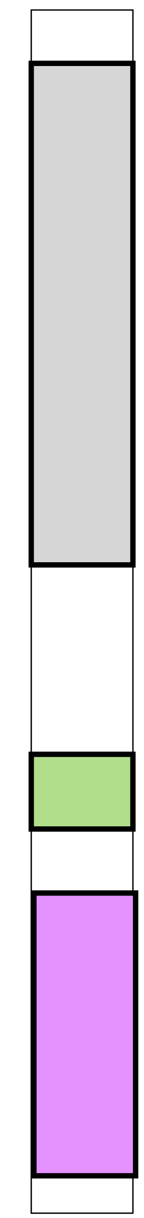


VM trick (4): fork

copy-on-write

P_1 address space

physical memory



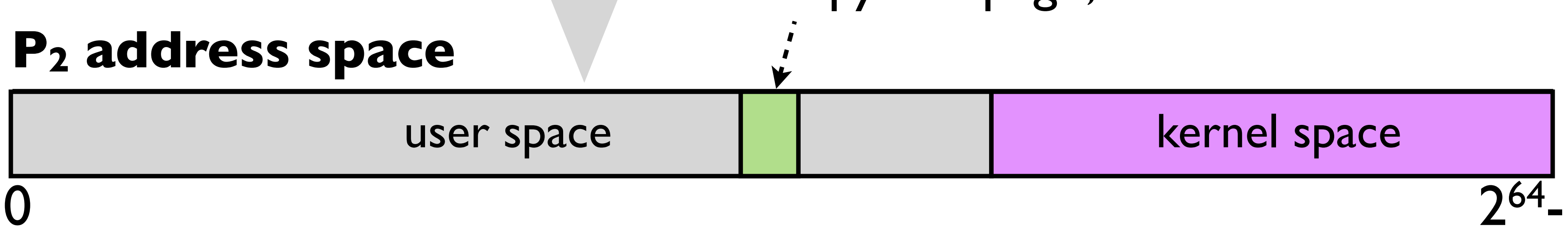
Efficient fork() via copy-on-write

- copy all page table mappings
- mark *read-only* in both processes
- lazy copy

On P_2 write:

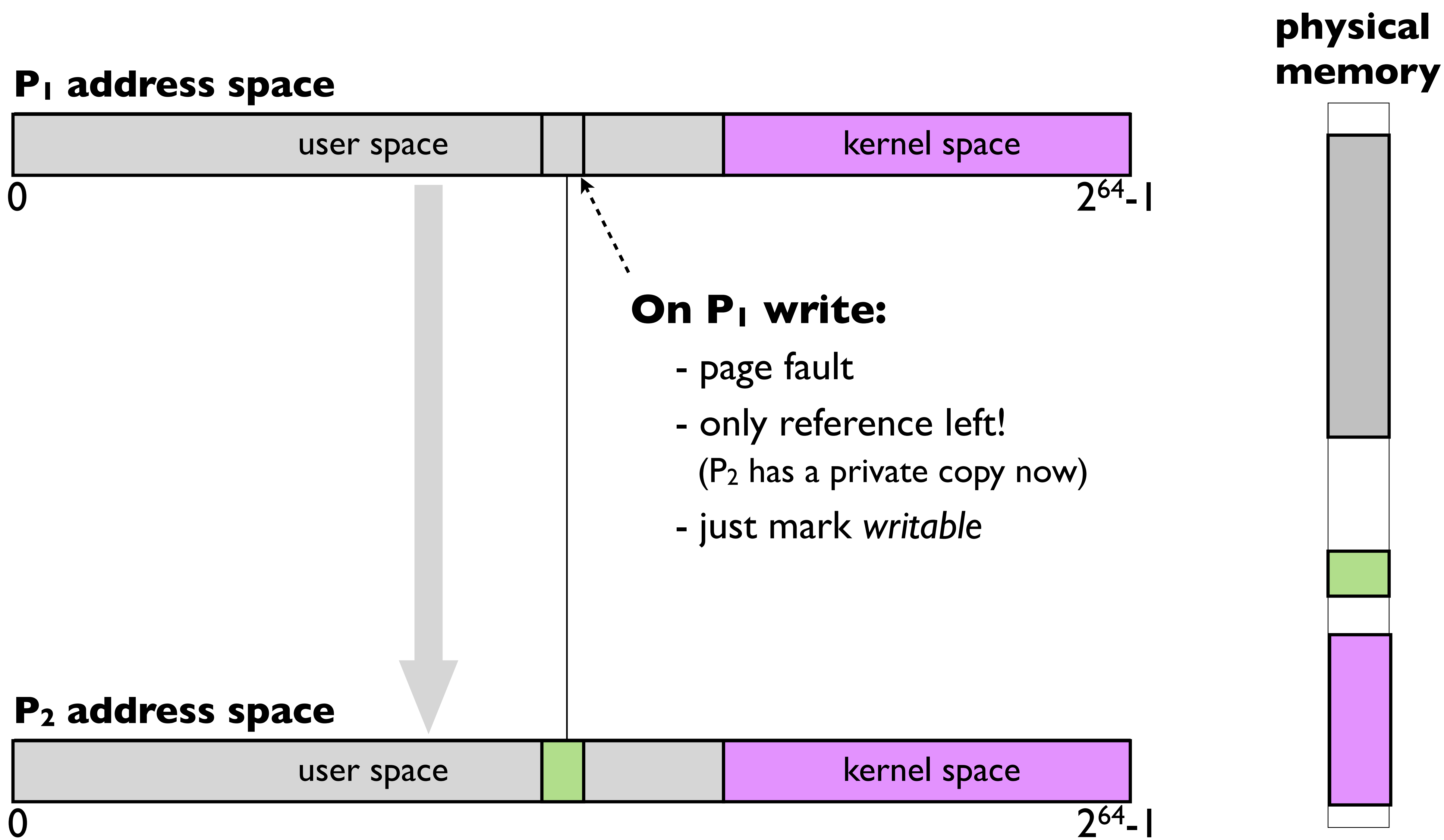
- page fault
- copy the page, mark *writable*

P_2 address space



VM trick (4): fork

copy-on-write



More VM tricks please!

See this excellent paper by Andrew Appel and Kai Li

“Virtual Memory Primitives for User Programs” (ASPLOS 1991)

- garbage collection, distributed shared memory, more ...

Check out Emery Berger’s work

Professor at UMASS

has made a career out of inventing VM tricks (among other things)

Today

- ~~Project 3~~
- ~~Virtual Address Spaces~~
  - ~~Part II: fun virtual memory tricks~~
- Paging

Paging

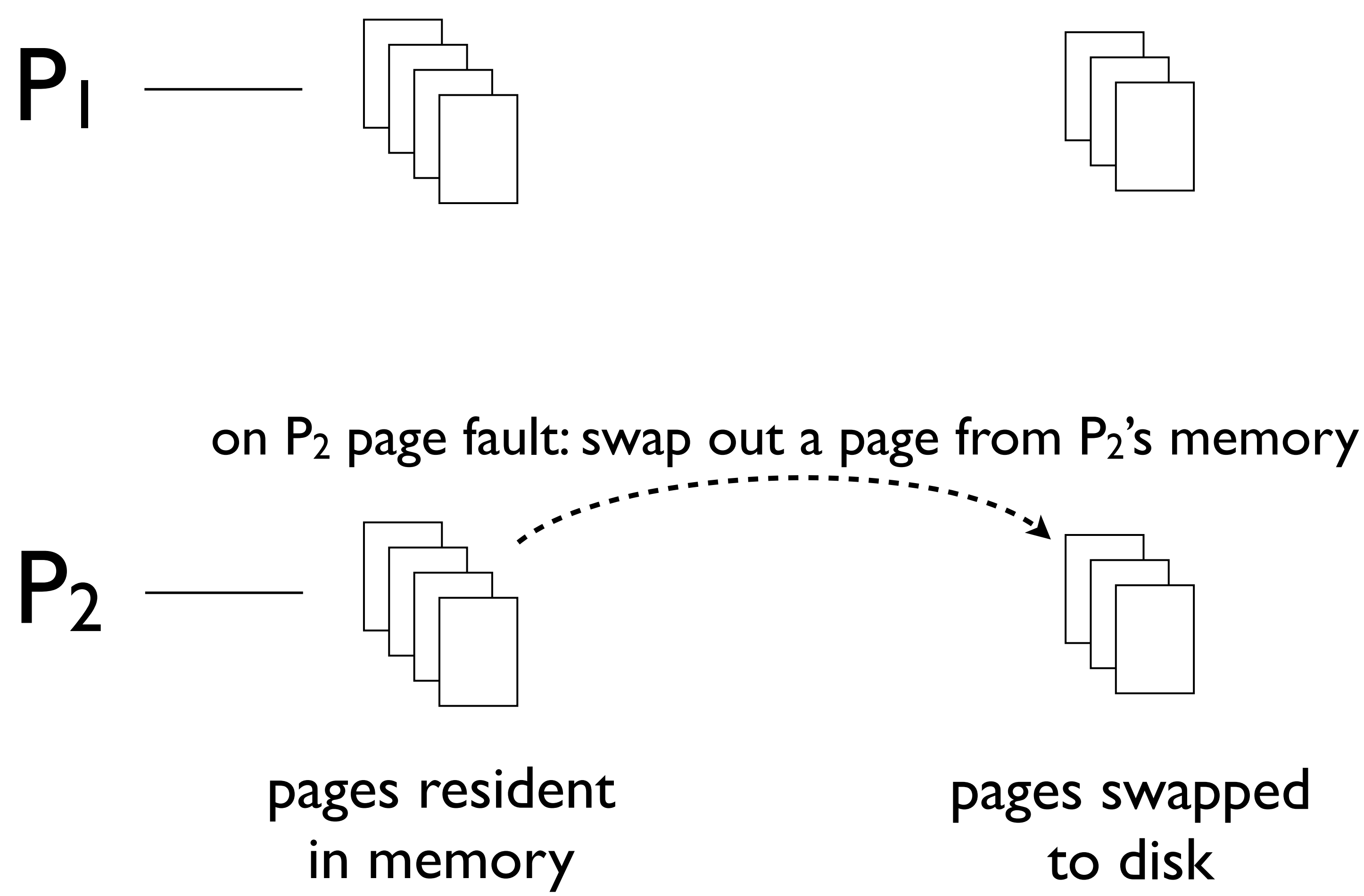
- What if we need more pages than available in physical memory?
  - page to disk
- Isn't this slow?
  - yes!
  - but processes have *locality*

Working Set

- W(t, w)
  - set of pages used in time [t-w, t]
- This is usually a small-ish subset of memory
  - demonstrated empirically
- Ideally: keep the working set in memory
  - page out everything else
  - see lecture slides for algorithms

Paging

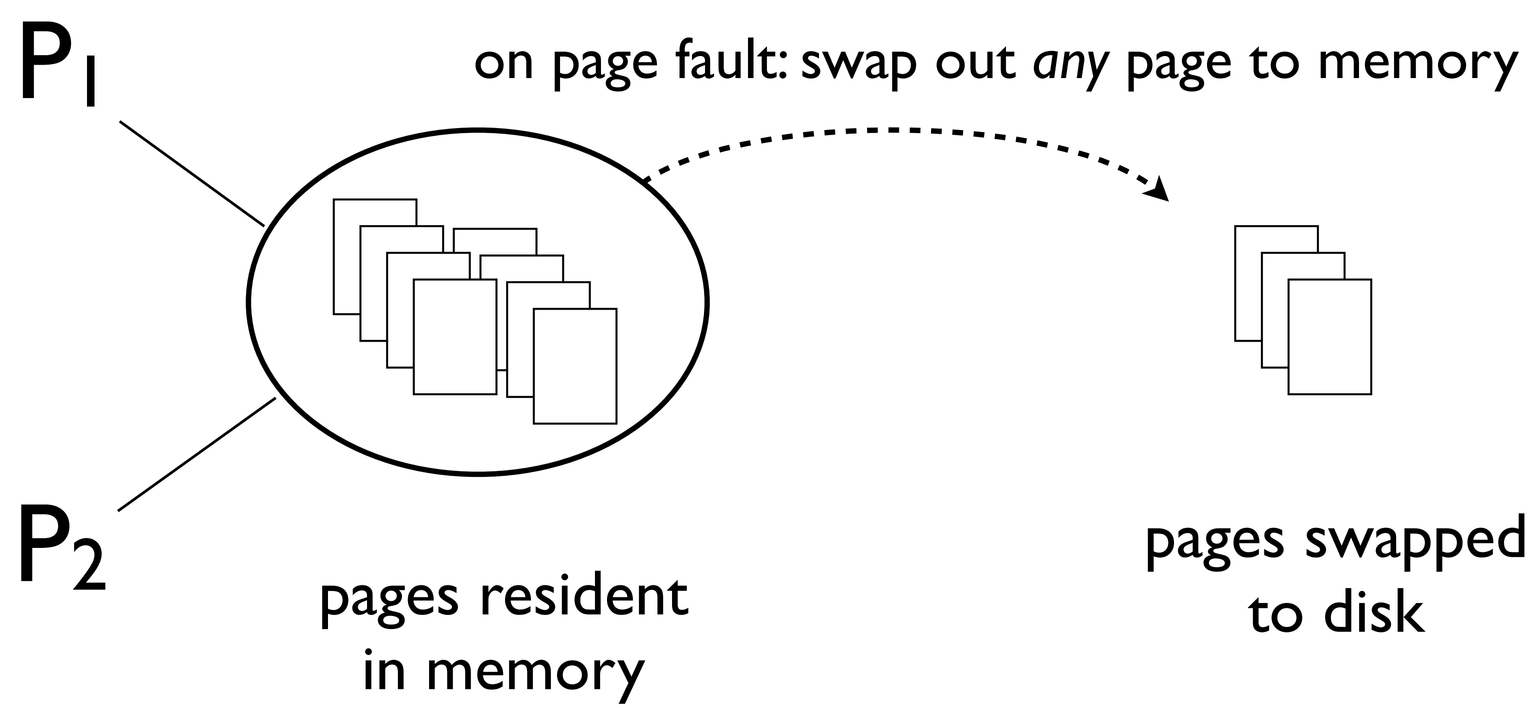
- *local* page replacement



(fixed quota per process)

Paging

- *global* page replacement



Paging

- *local* page replacement
  - fixed quota per process
  - why bad?
    - not globally optimal
    - e.g.: foreground tasks should get more pages
- *global* page replacement
  - no quotas
  - why bad?
    - more variability, possibility for unfairness

Working set

- When is the working set the entire program?
  - garbage collection! (mark-and-sweep...)
  - Java performance tanks when paging to disk