ANNOUNCEMENTS

- Recitation for this week will cover required material (Barrier Synchronization) assigned in the reading (C. 21 of the Harmony book.
- Recitation recording will be available!

Memory Management

(3EP, Ch. 12-24)

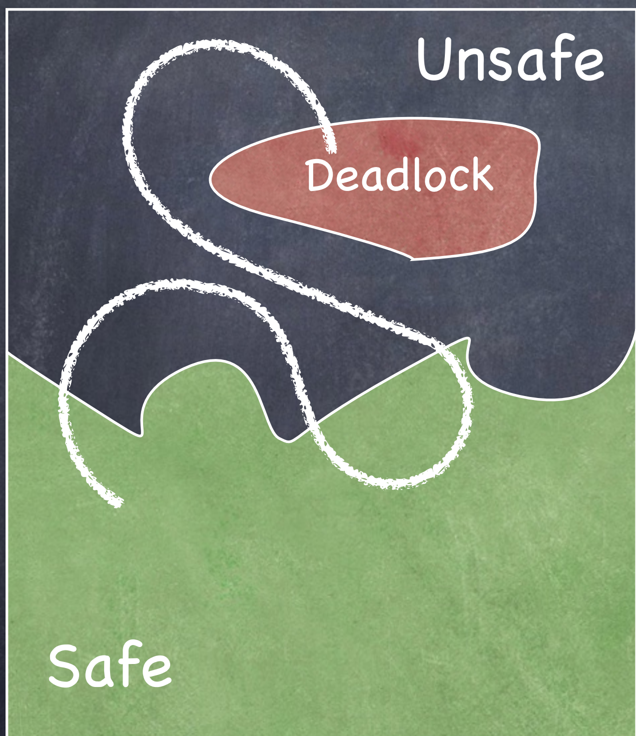
Previously, on CS4410...

Avoiding Deadlock: The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Sum of max resources needs can exceed total available resources
- Acquiring all resources at once can be inefficient!
- Allow to parcel out resources incrementally as long as
  - there exists a schedule of loan fulfillments such that
    - all clients receive their maximal loan
    - build their house
    - pay back all the loan

Living dangerously: Safe, Unsafe, Deadlocked



A system's trajectory through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce that safe schedule!
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, independent of the schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P	2	0	0
P	3	0	3
P	2	1	1
P	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P	2	0	2
P	0	0	0
P	1	0	2
P	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

but it is not a safe state!

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P	2	0	0
P	3	0	3
P	2	1	1
P	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P	2	0	2
P	0	0	1
P	1	0	2
P	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P	2	0	0
P	3	0	3
P	2	1	1
P	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P	2	0	2
P	0	0	1
P	1	0	2
P	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- Without Max, can we avoid deadlock by delaying granting requests?
  - NO! Deadlock triggered when request formulated, not granted!

Abstraction is our Business

- What I have
  - A single (or a finite number) of CPUs
  - Many programs I would like to run
- What I want: a Thread
  - Each program has full control of one or more virtual CPUs

Abstraction is our Business

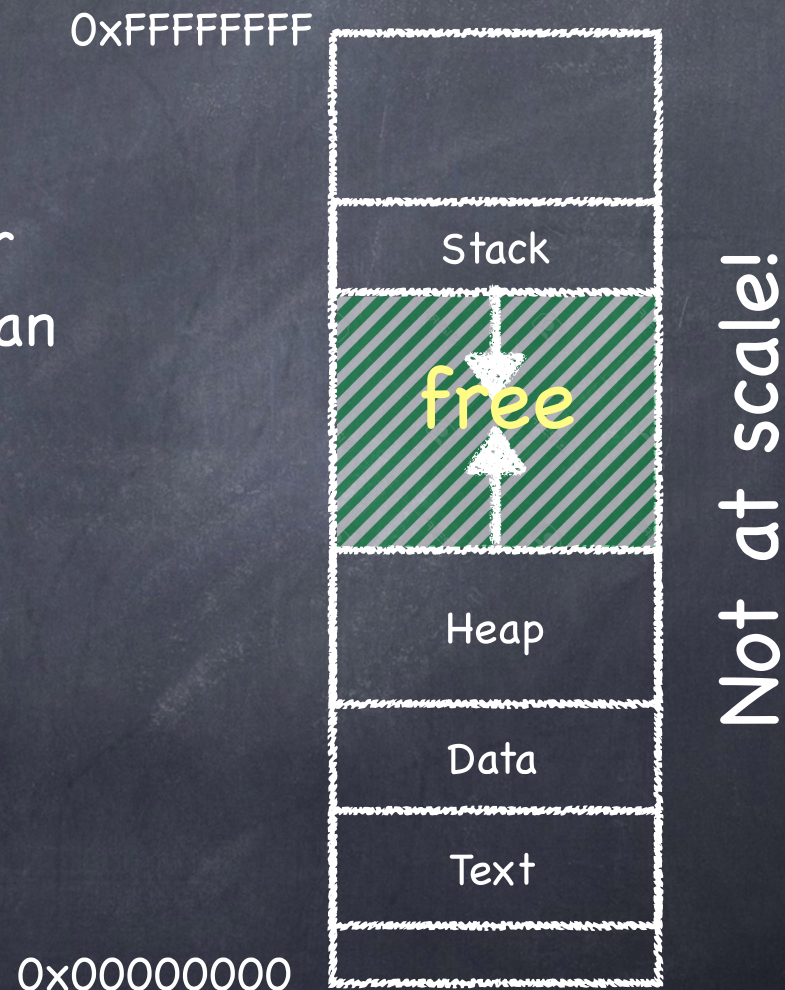
- What I have
  - A certain amount of physical memory
  - Multiple programs I would like to run
    - together, they may need more than the available physical memory
- What I want: an Address Space
  - Each program has as much memory as the machine's architecture will allow to name
  - All for itself

Address Space

- Set of all names used to identify and manipulate unique instances of a given resource
  - memory locations (determined by the size of the machine's word)
    - for 32-bit-register machine, the address space goes from 0x00000000 to 0xFFFFFFFF
  - memory locations (determined by the number of memory banks mounted on the machine)
  - phone numbers (XXX) (YYY-YYYY)
  - colors: R (8 bits) + G (8 bits) + B (8 bits)

Virtual Address Space: An Abstraction for Memory

- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine



Physical Address Space: How memory actually looks

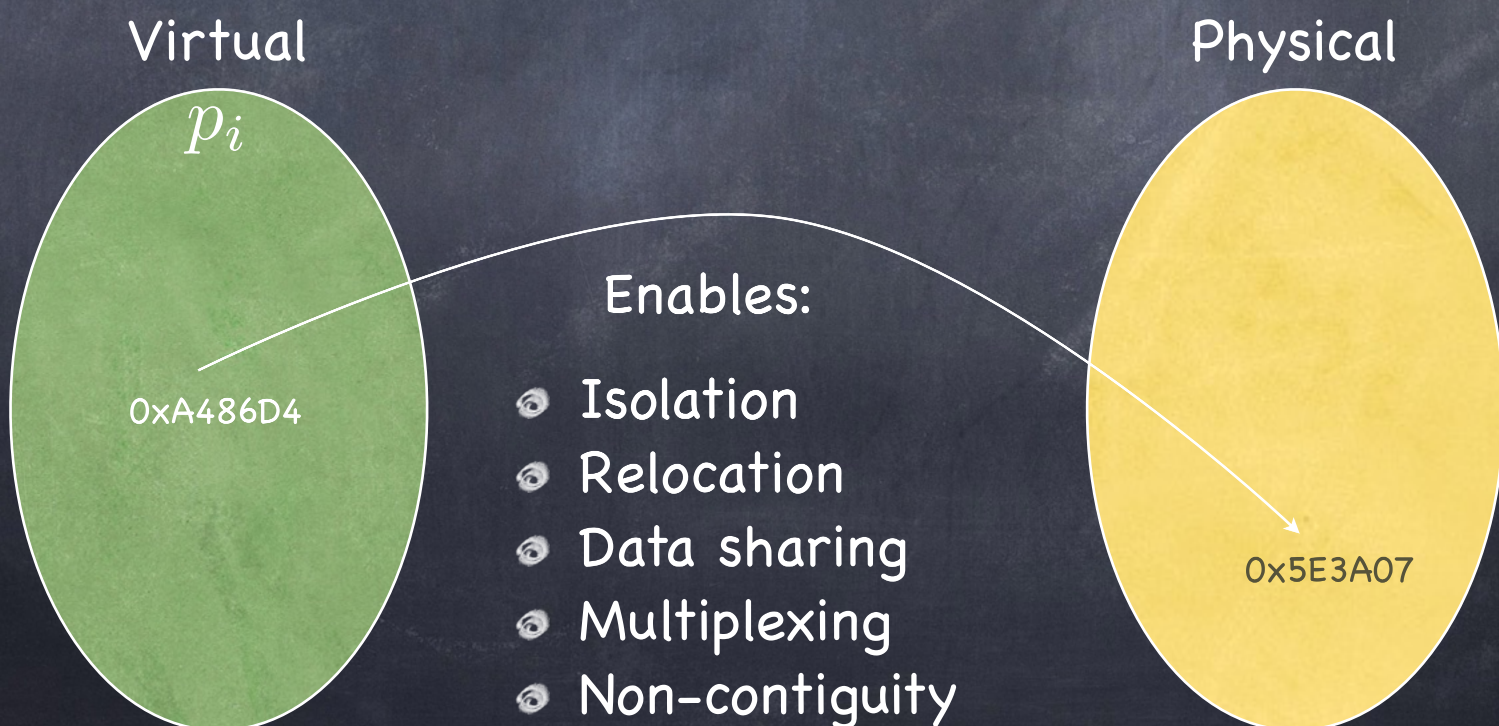
- Processes loaded in memory at some memory location
  - virtual address 0 is not loaded at physical address 0
- Multiple processes may be loaded in memory at the same time, and yet...
- ...physical memory may be too small to hold even a single virtual address space in its entirety
  - 64-bit, anyone?

II. Memory Isolation

Step 2: Address Translation

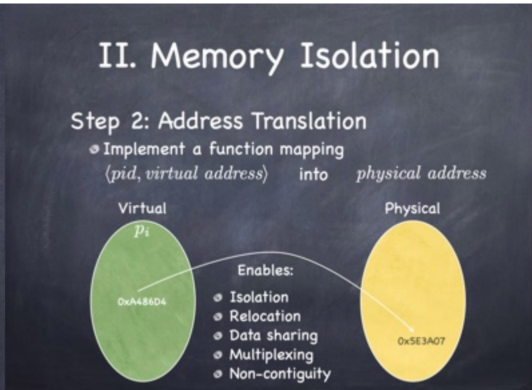
- Implement a function mapping

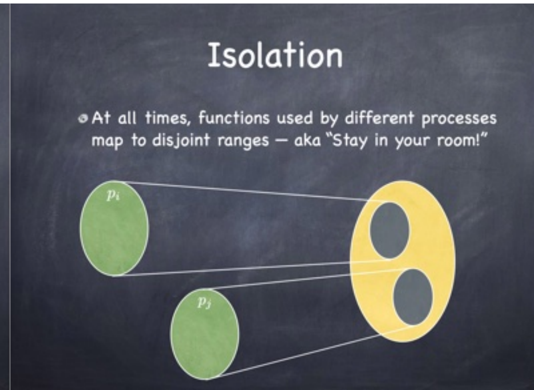
$\langle pid, virtual\ address \rangle$ into $physical\ address$

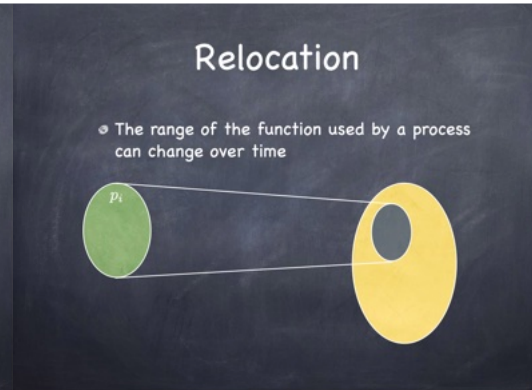


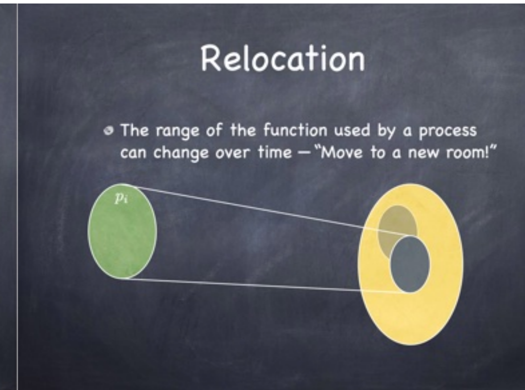
Enables:

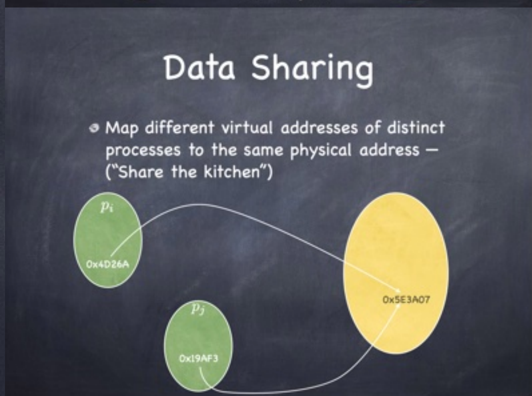
- Isolation
- Relocation
- Data sharing
- Multiplexing
- Non-contiguity

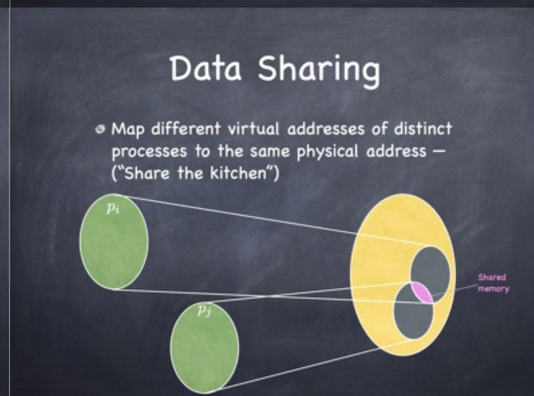


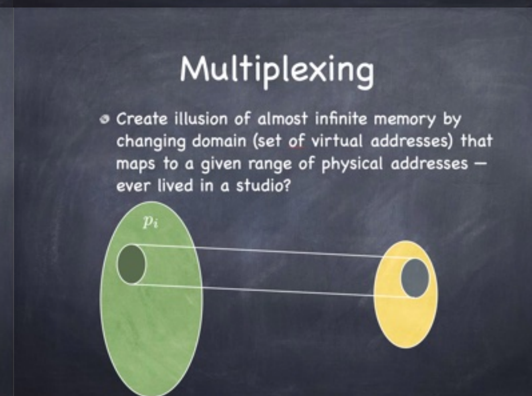


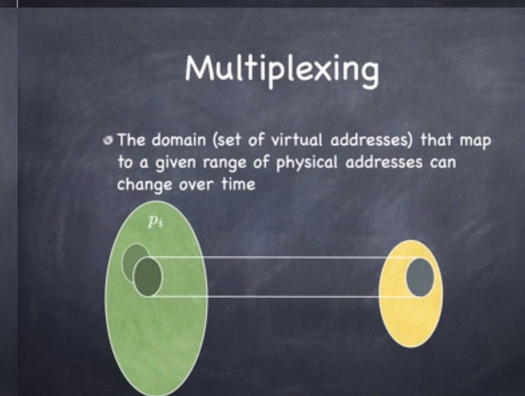


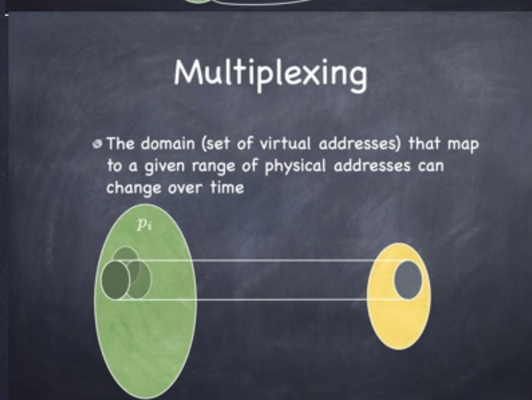


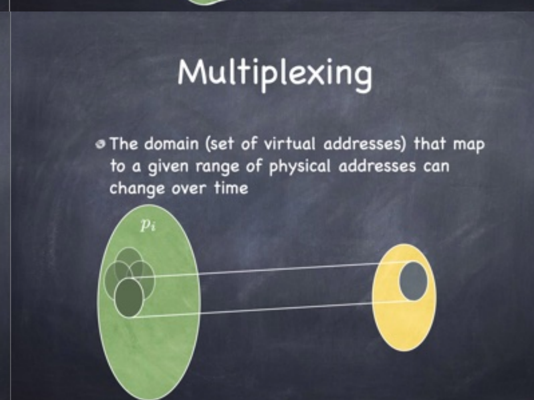


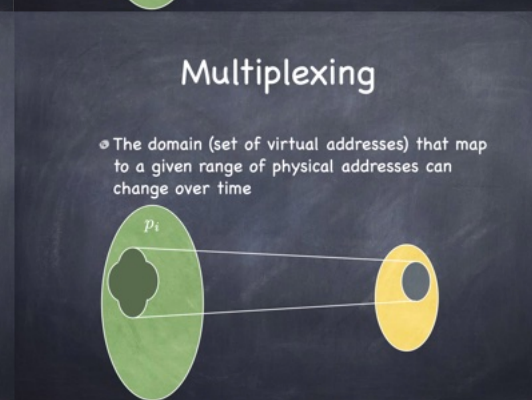


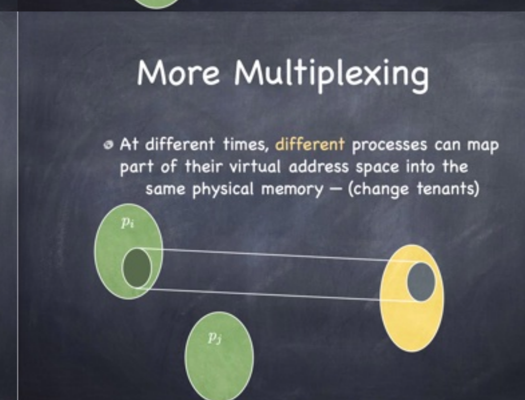


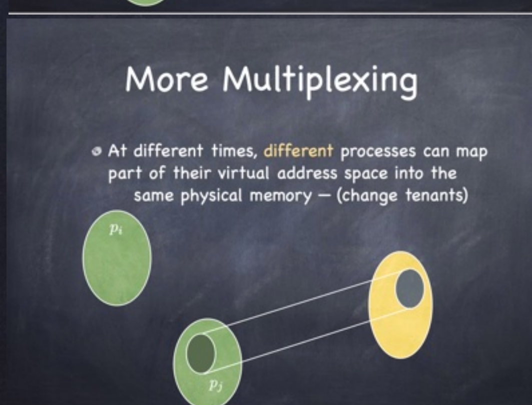


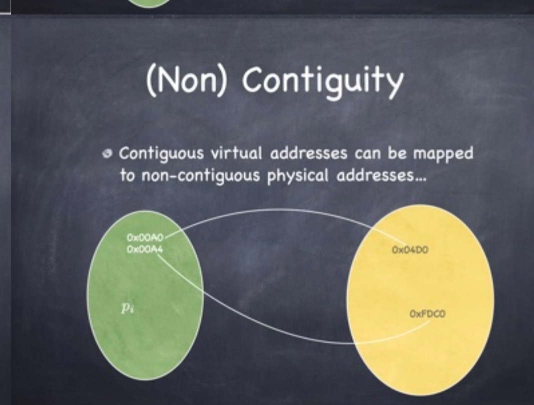


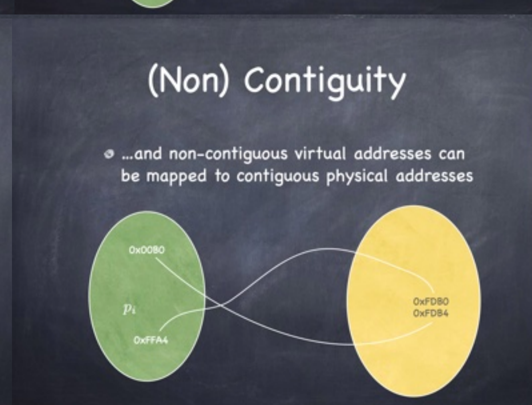






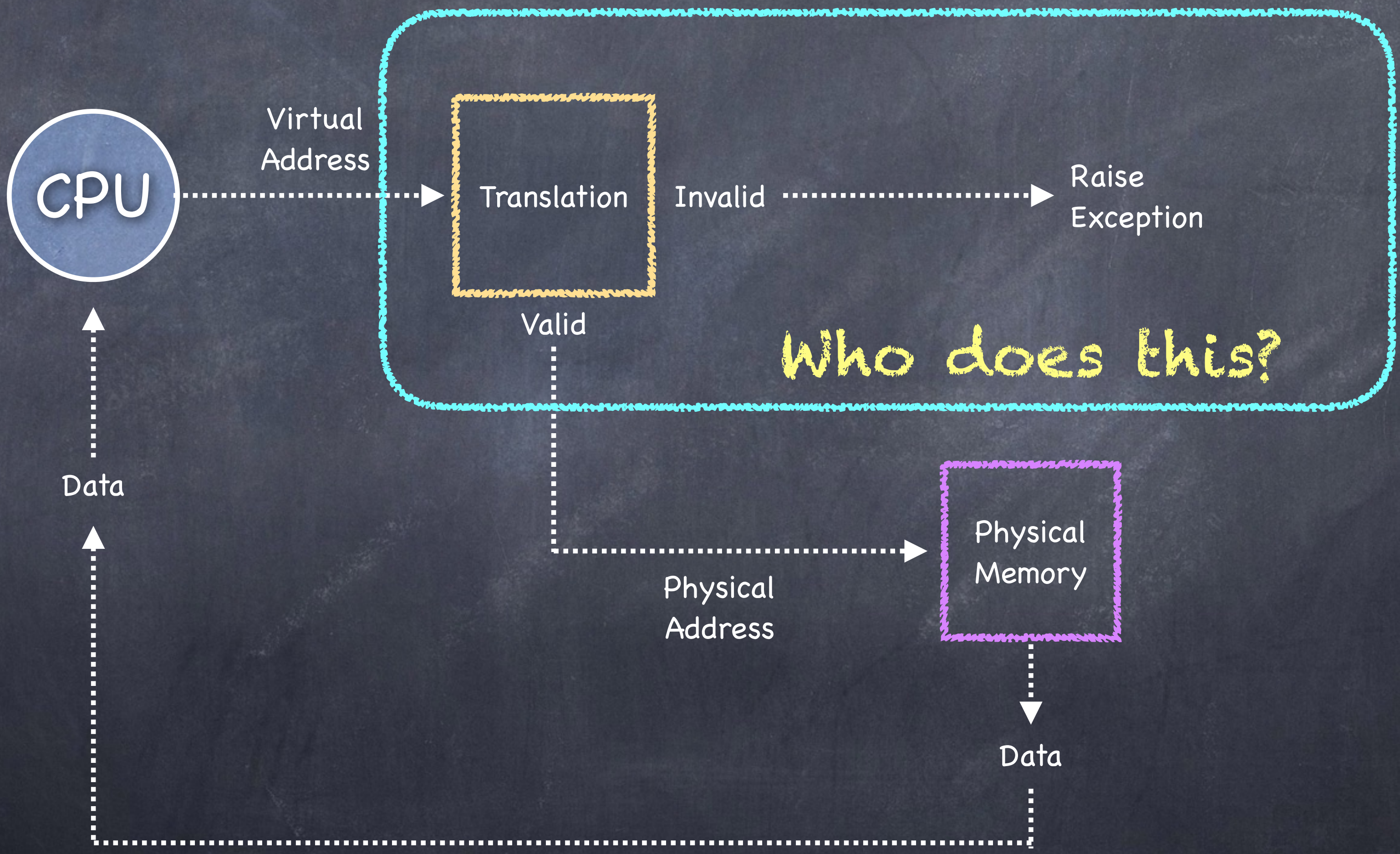






The Power of Mapping

Address Translation, Conceptually



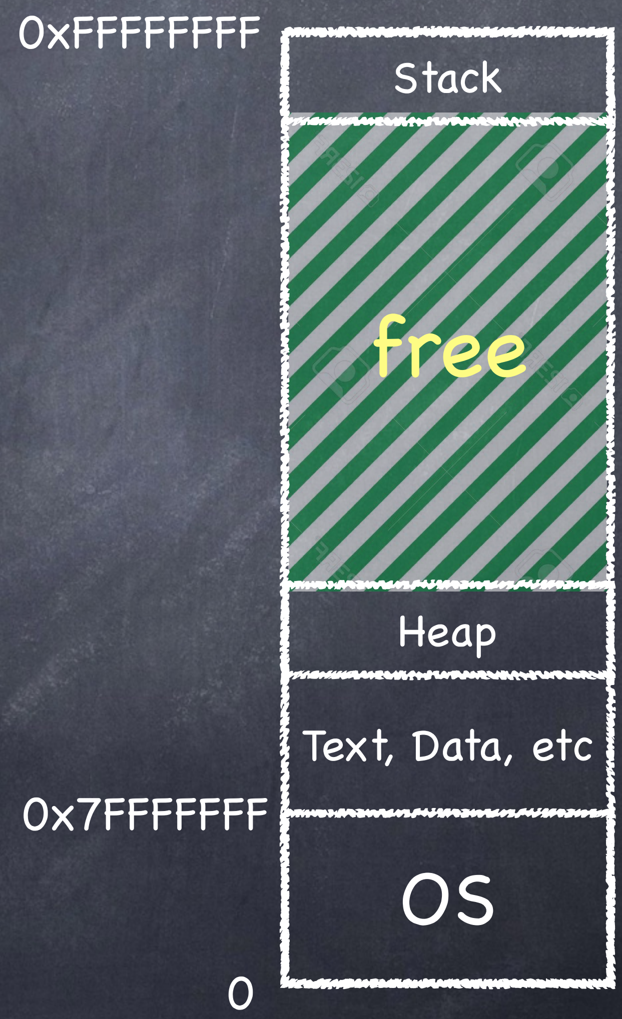
Memory Management Unit (MMU)

- Hardware device
  - Maps virtual addresses to physical addresses

- User process
  - deals with virtual addresses
  - never sees the physical address
- Physical memory
  - deals with physical addresses
  - never sees the virtual address

The Identity Mapping

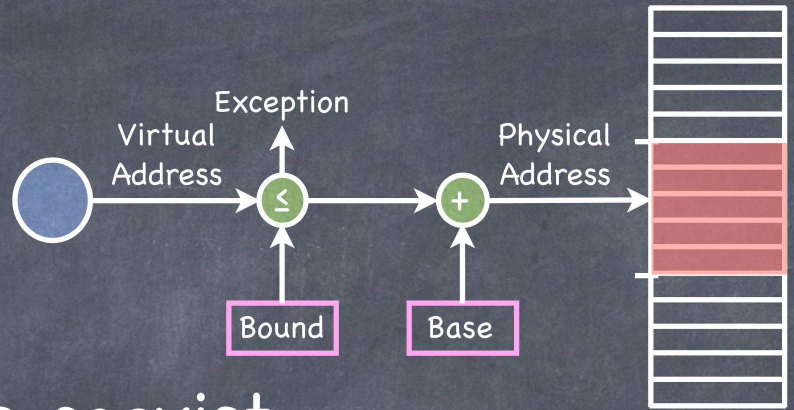
- Map each virtual address onto the identical physical address
  - Virtual and physical address spaces have the same size
  - Run a single program at a time
    - OS can be a simple library
    - very early computers
- Friendly amendment: leave some of the physical address space for the OS
  - Use loader to relocate process
    - early PCs



More sophisticated address translation

- How to perform the mapping efficiently?
  - So that it can be represented concisely?
  - So that it can be computed quickly?
  - So that it makes efficient use of the limited physical memory?
  - So that multiple processes coexist in physical memory while guaranteeing isolation?
  - So that it decouples the size of the virtual and physical addresses?
- Ask hardware for help!

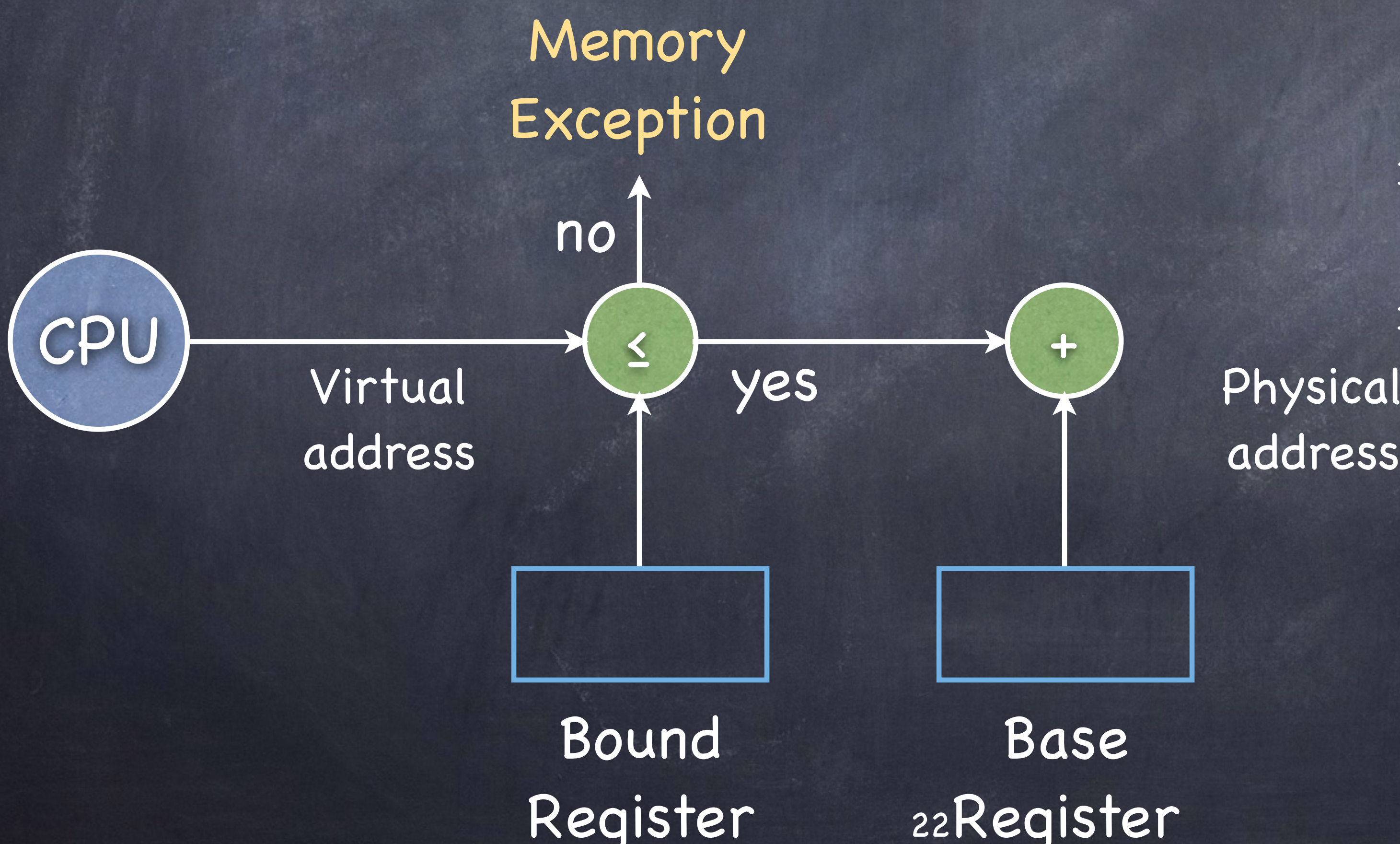
Base & Bound



- Goal: let multiple processes coexist in memory while guaranteeing isolation
- Needed hardware
  - two registers: Base and Bound (a.k.a. Limit)
  - Stored in the PCB
- Mapping
  - pa = va + Base
    - as long as 0 ≤ va ≤ Bound
  - On context switch, change B&B (privileged instruction)

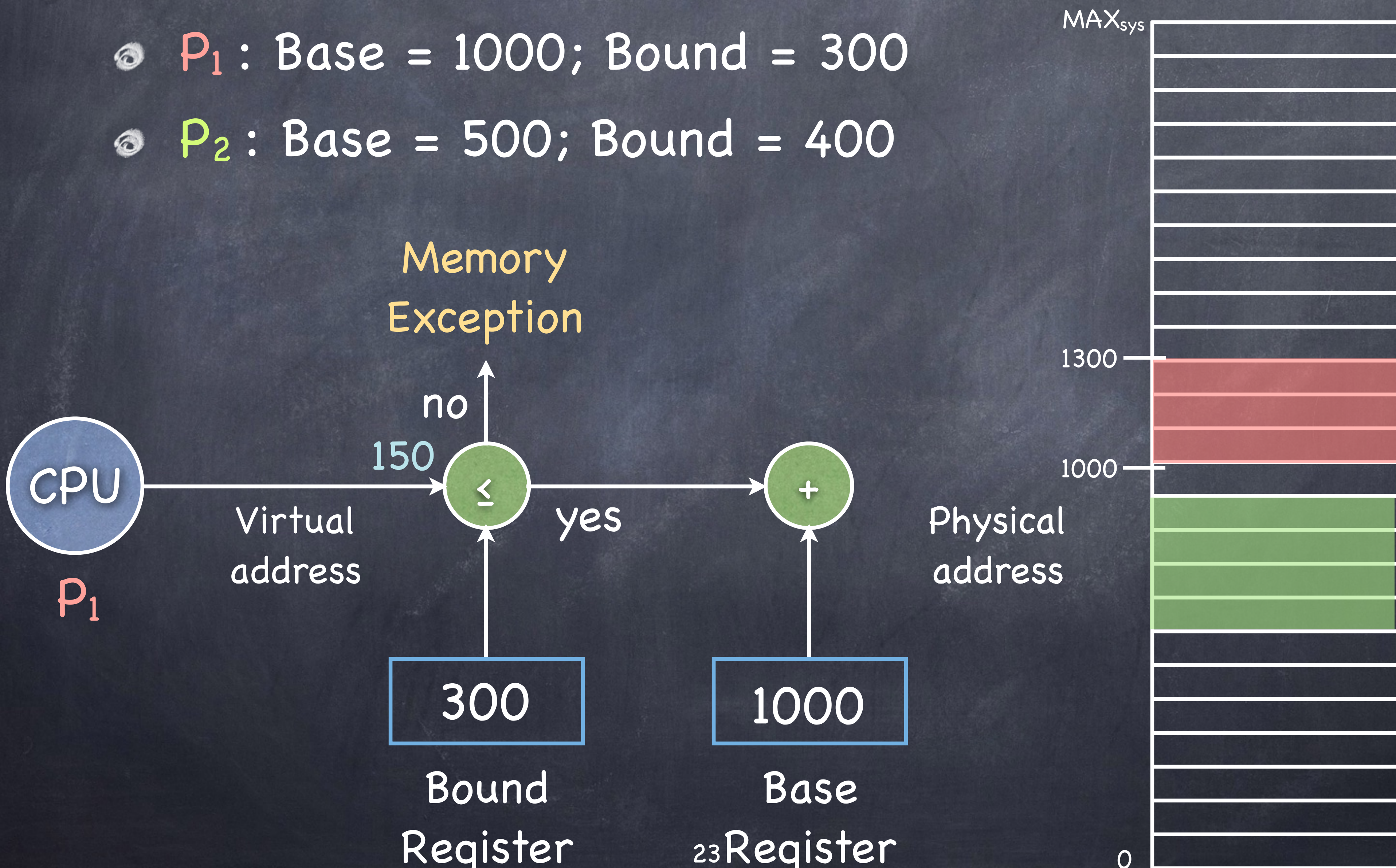
Base & Bound

- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



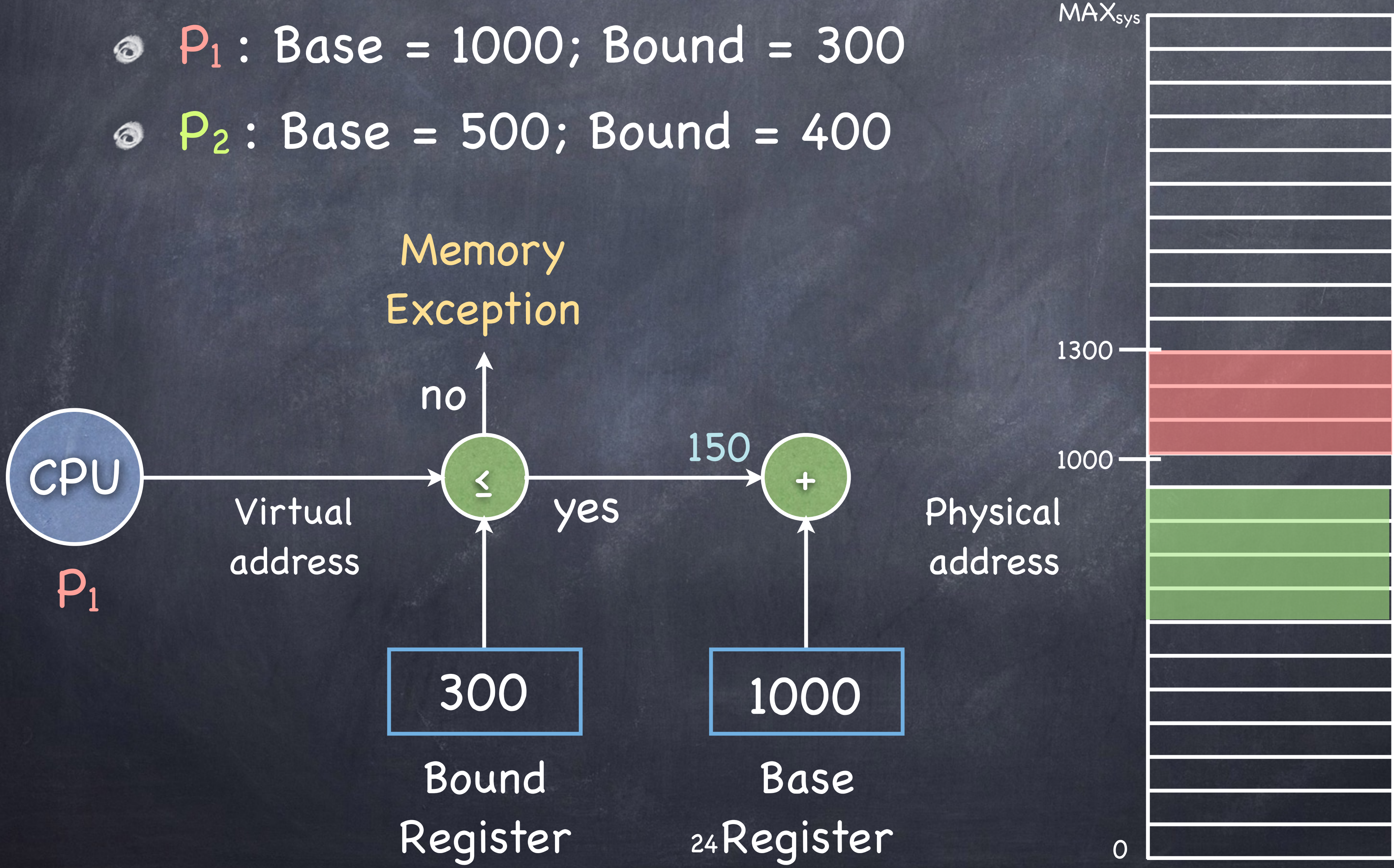
Base & Bound

- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



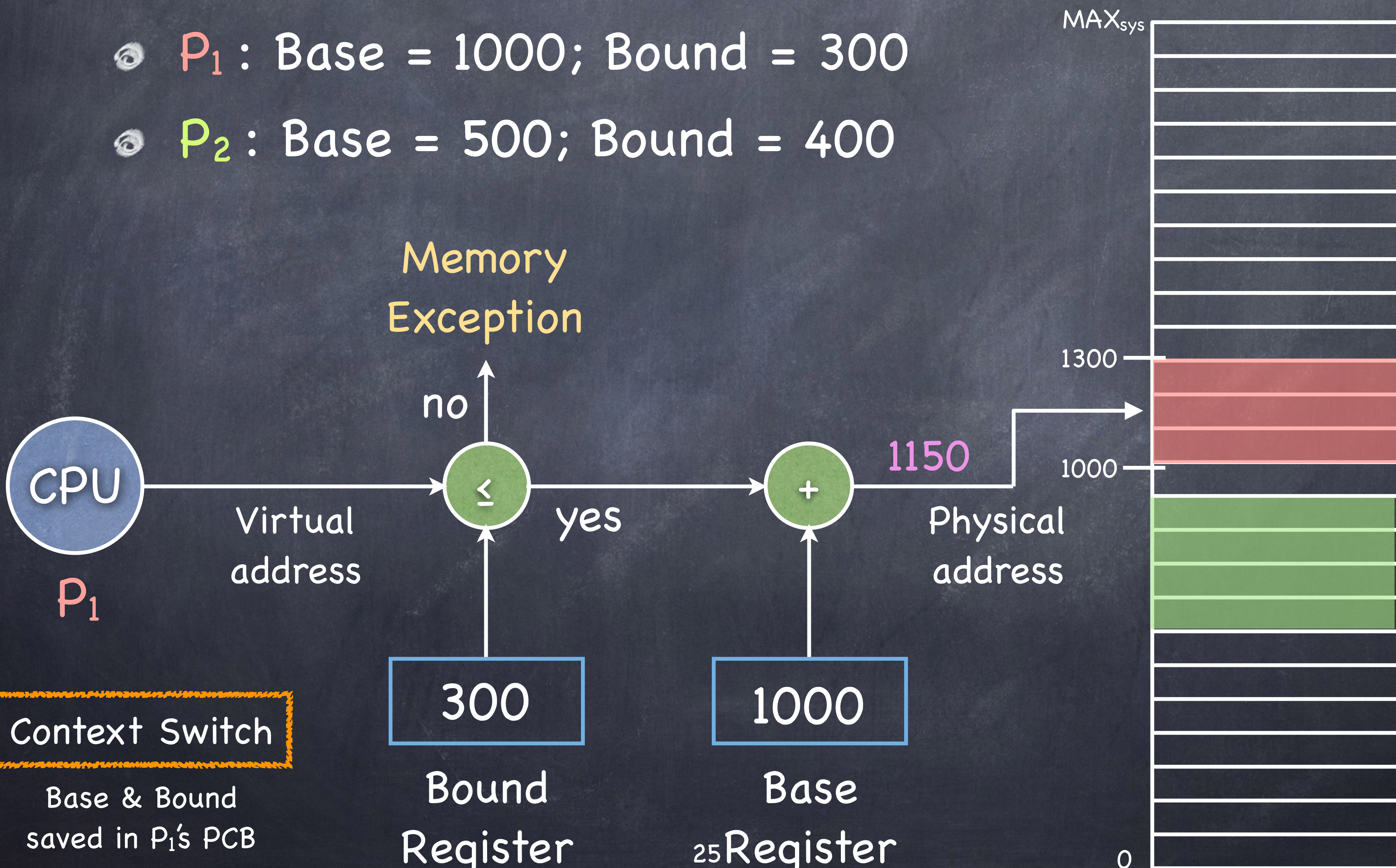
Base & Bound

- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



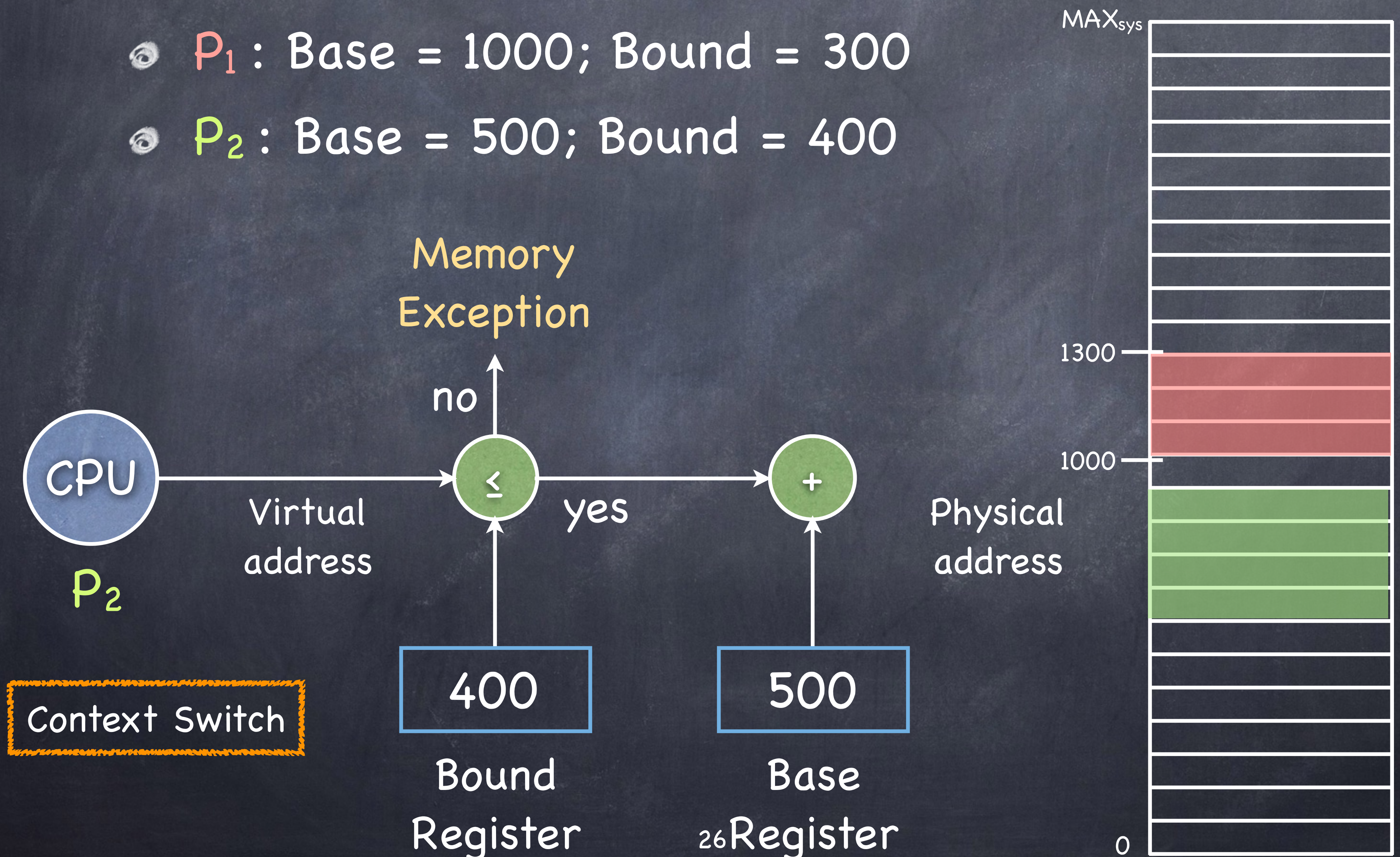
Base & Bound

- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



Base & Bound

- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



On Base & Bound

- Contiguous Allocation
  - contiguous virtual addresses are mapped to contiguous physical addresses
- But mapping entire address space to physical memory
  - is wasteful
    - lots of free space between heap and stack...
    - makes sharing hard
  - does not work if the address space is larger than physical memory
    - think 64-bit registers...

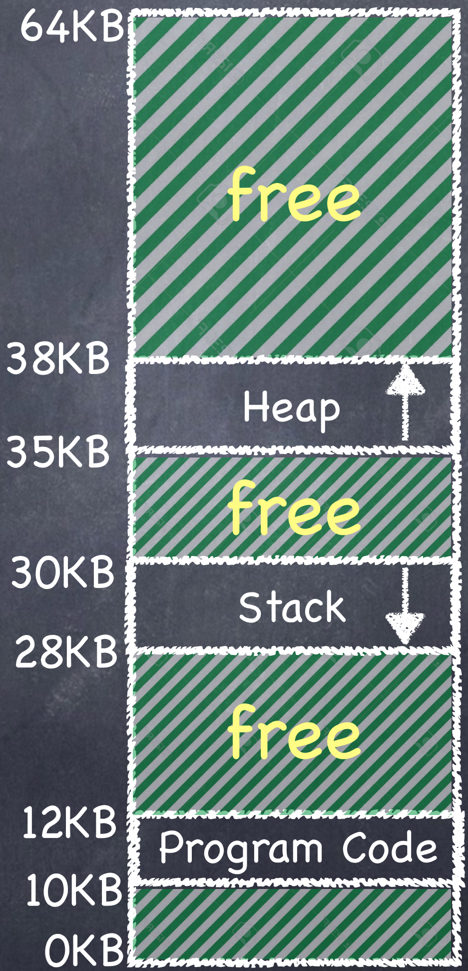
E Pluribus Unum

- An address space comprises multiple segments
  - contiguous sets of virtual addresses, logically connected
    - heap, code, stack, (and also globals, libraries...)
  - each segment can be of a different size

Segmentation: Generalizing Base & Bound

- Base & Bound registers to each segment
  - each segment independently mapped to a set of contiguous addresses in physical memory
    - no need to map unused virtual addresses

Segment	Base	Bound
Code	10K	2K
Stack	28	2K
Heap	35K	3K



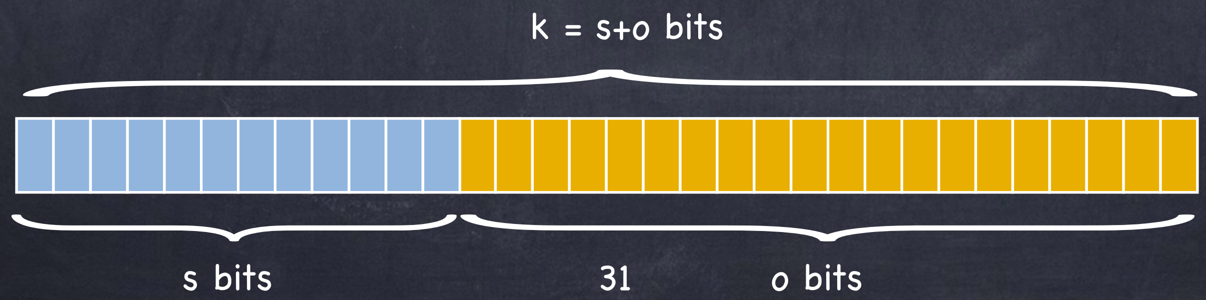
(not to scale)

Segmentation

- Goal: Supporting large address spaces (while allowing multiple processes to coexist in memory)
- Needed hardware
  - two registers (Base and Bound) per segment
    - values stored in the PCB
  - if many segments, a segment table, stored in memory, at an address pointed to by a Segment Table Register (STBR)
    - process' STBR value stored in the PCB

Segmentation: Mapping

- How do we map a virtual address to the appropriate segment?
  - Read VA as having two components
    - s most significant bits identify the segment
      - at most 2^s segments
    - o remaining bits identify offset within segment
      - each segment's size can be at most 2^o bytes



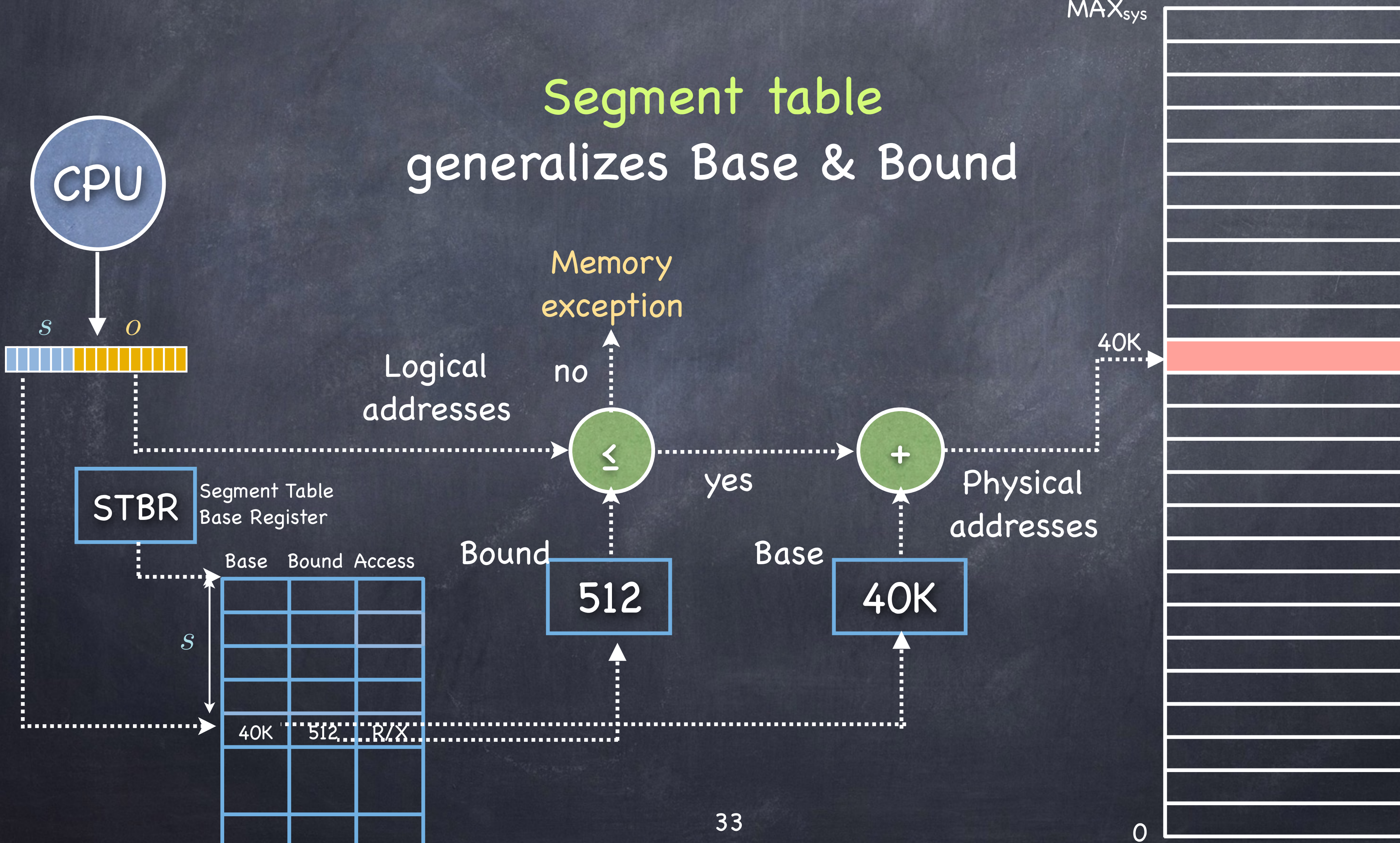
Segment Table

- Use s bits to index to the appropriate row of the segment table

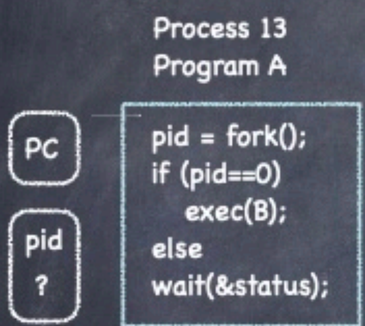
	Base	Bound (Max 4k)	Access
$Code_{00}$	32K	2K	Read/Execute
$Heap_{01}$	34K	3K	Read/Write
$Stack_{10}$	28K	3K	Read/Write

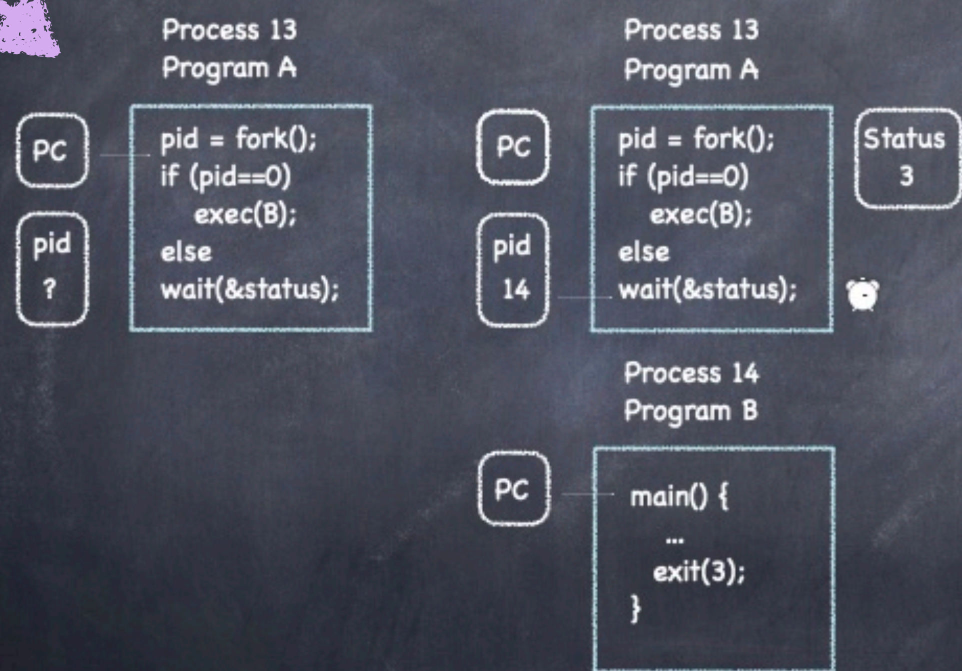
- Segments can be shared by different processes
  - use protection bits to determine if shared Read only (maintaining isolation) or Read/Write (if shared, no isolation)
    - processes can share code segment while keeping data private

Implementing Segmentation



Revisiting fork()





Revisiting fork()

- Copying an entire address space can be costly...
  - especially if you proceed to obliterate it right away with exec()!

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Child

- but change all writeable segments to Read only

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Child

- but change all writeable segments to Read only

- Segments in VA spaces of parent and child point to same locations in physical memory

Copy on Write (COW)

- When trying to modify an address in a COW segment:
  - exception!
    - exception handler copies just the affected segment, and changes both the old and new segment back to writeable
- If exec() is immediately called, only stack segment is copied!
  - it stores the return value of the fork() call, which is different for parent and child

Managing Free space

- Many segments, different processes, different sizes
- OS tracks free memory blocks ("holes")
  - Initially, one big hole
- Many strategies to fit segment into free memory (think "assigning classrooms to courses")

  - First Fit: first big-enough hole
  - Next Fit: Like First Fit, but starting from where you left off
  - Best Fit: smallest big-enough hole
  - Worst Fit: largest big-enough hole

External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes

External Fragmentation

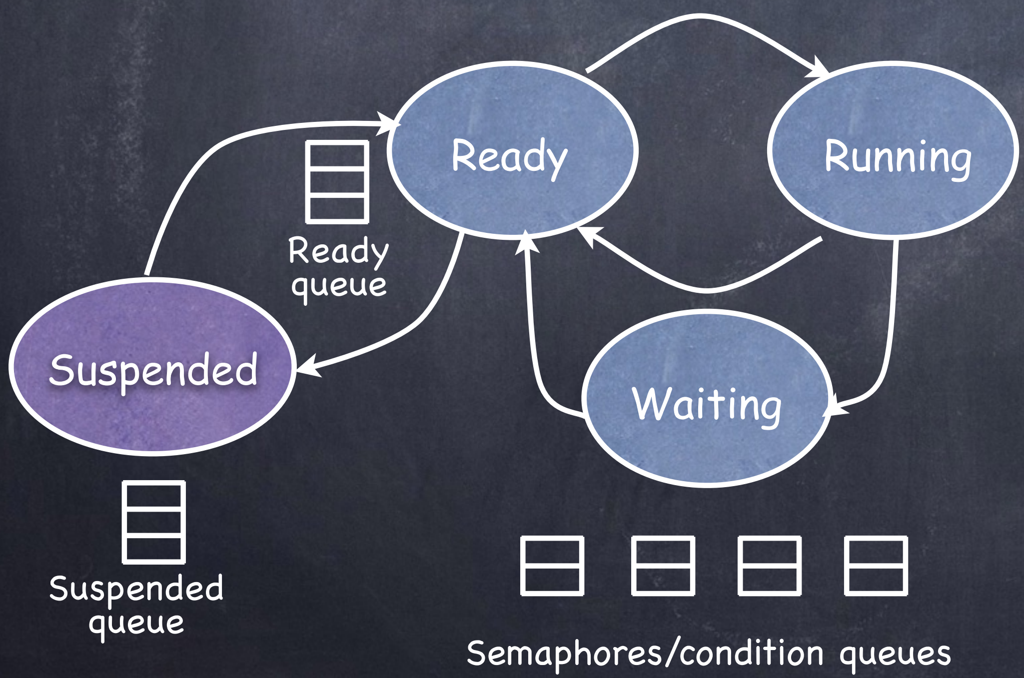
- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes

External Fragmentation

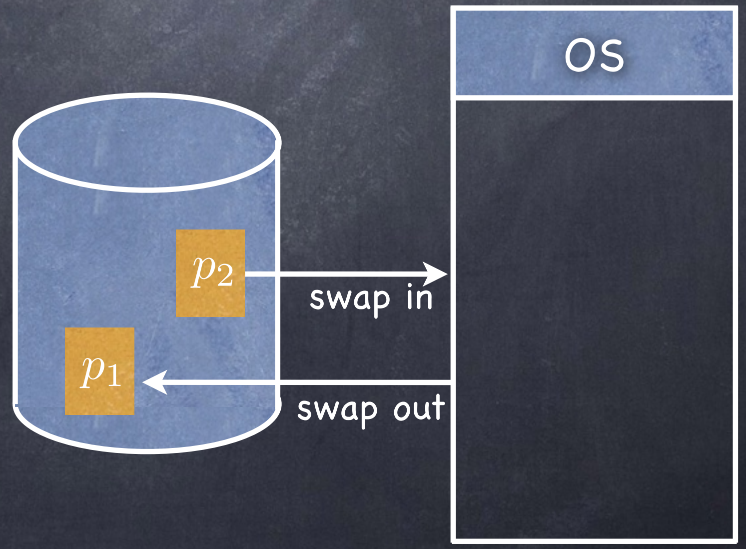
- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes
    - Copying eats up a lot of CPU time!
      - if 4 bytes in 10ns, 8 GB in 20s!
- But what if a segment wants to grow?

Eliminating External Fragmentation: Swapping

- Preempt processes and reclaim their memory

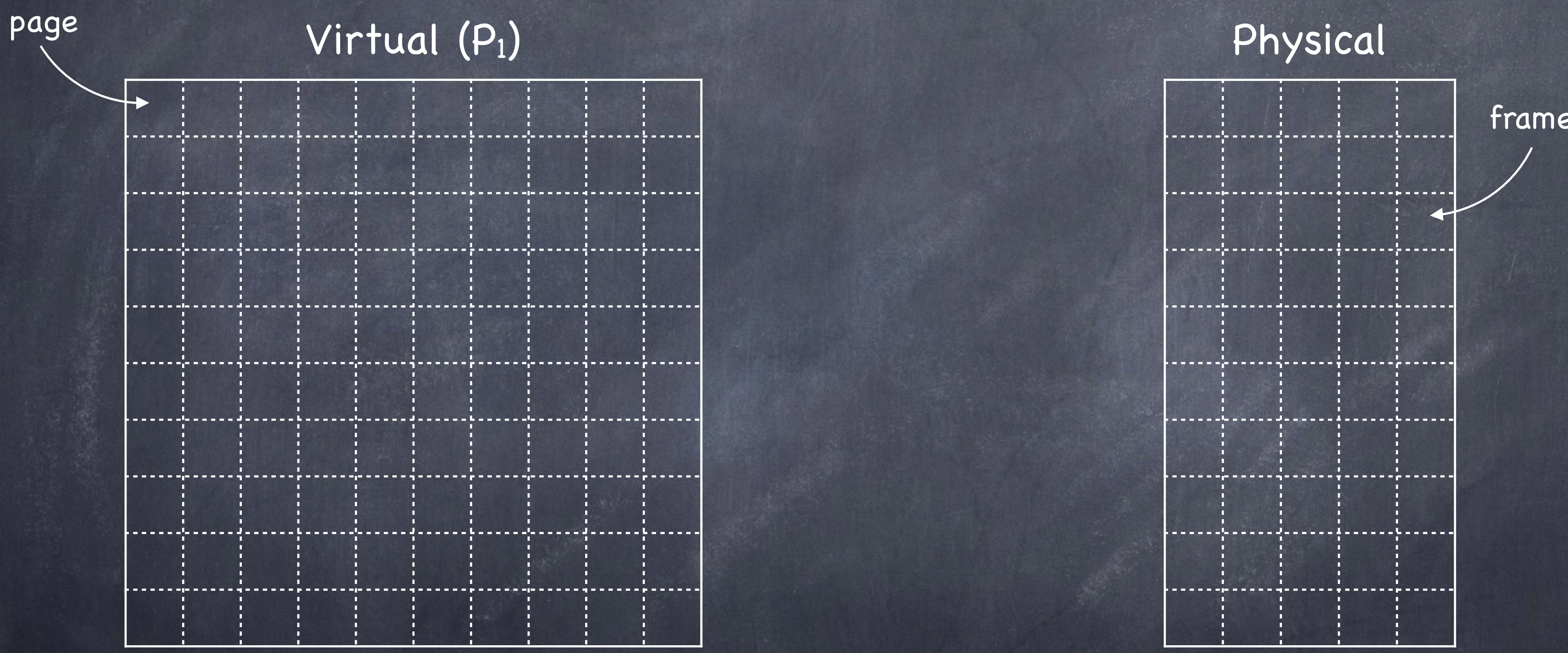


- Move images of suspended processes to backing store

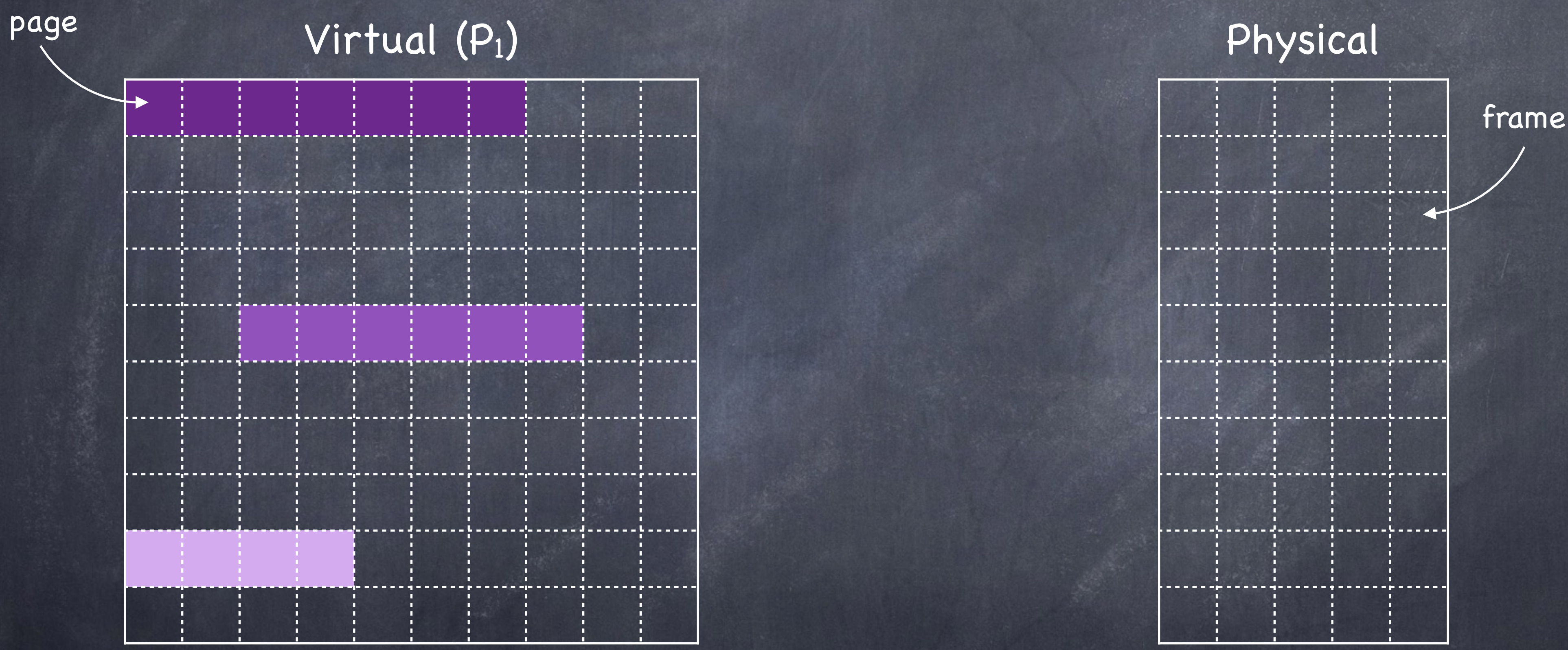


Eliminating External Fragmentation: Tiling Memory

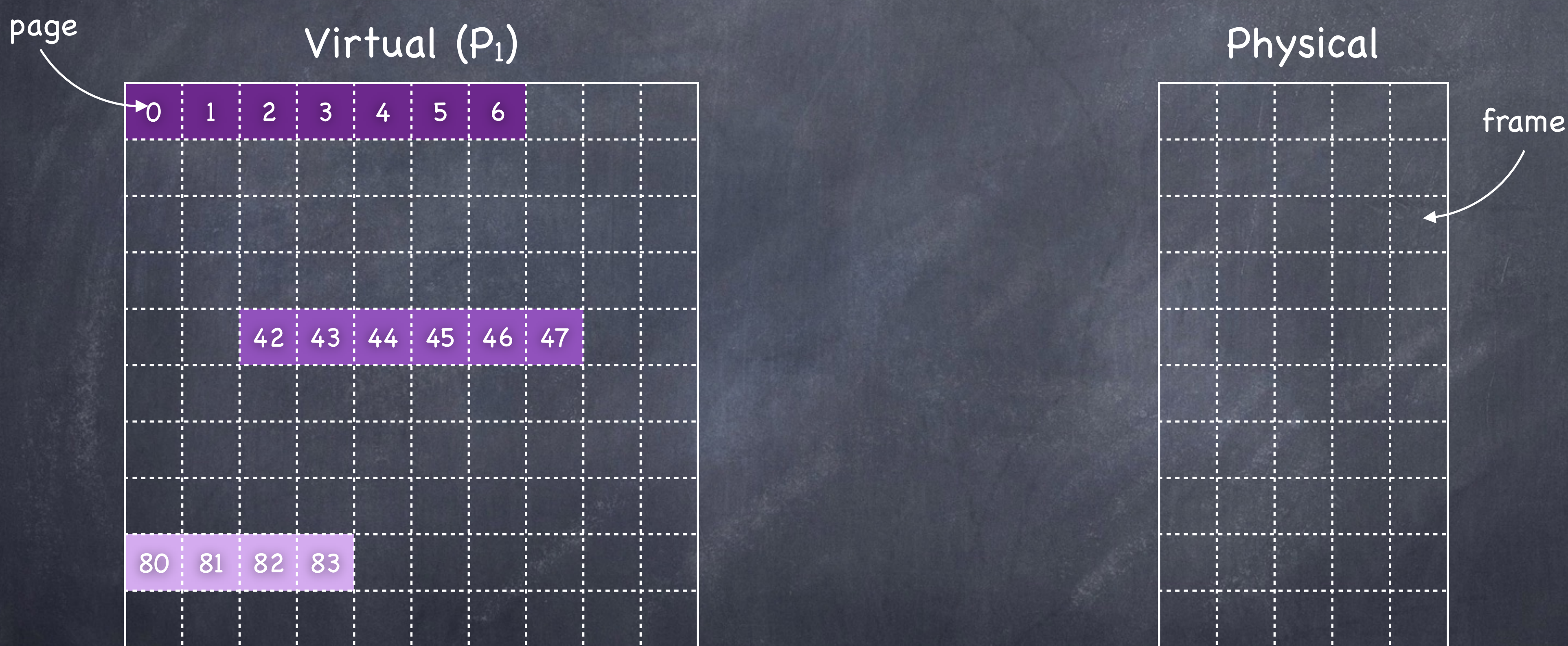
Tiling Memory



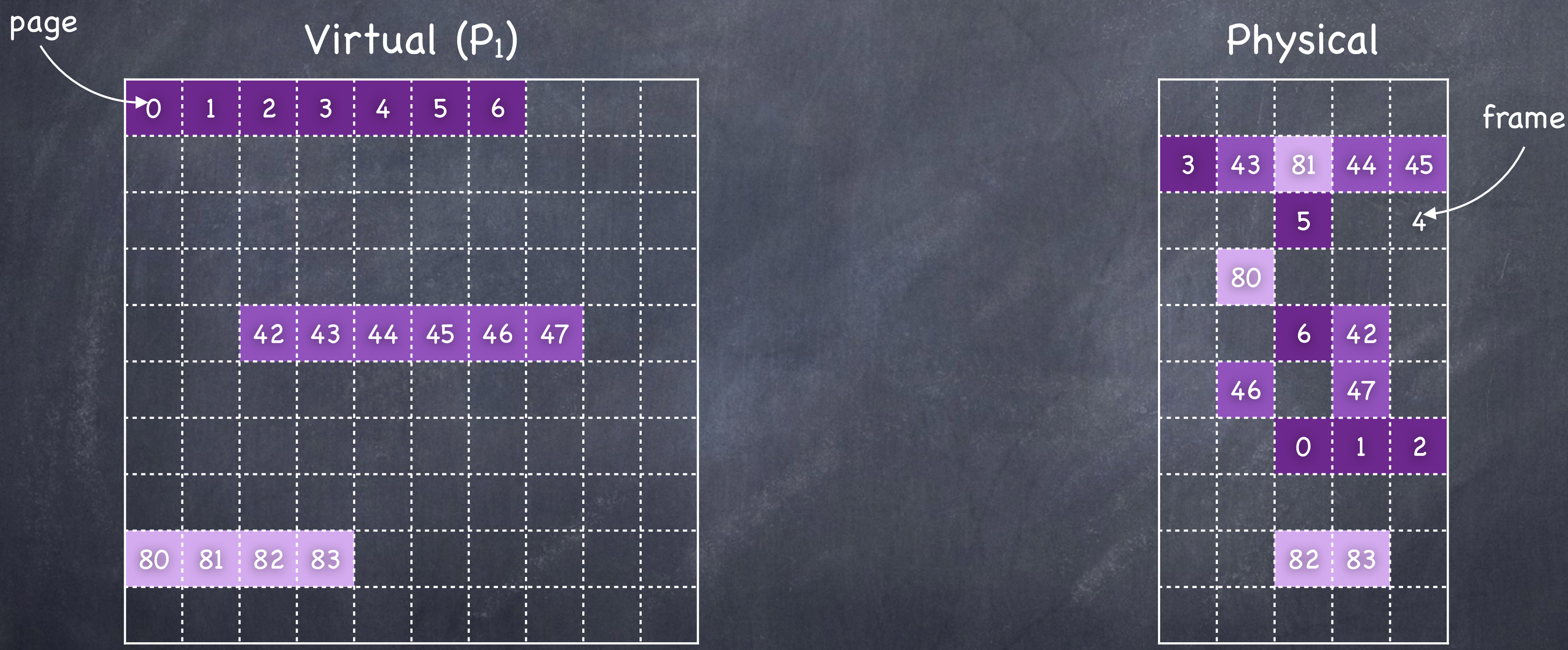
Tiling Memory



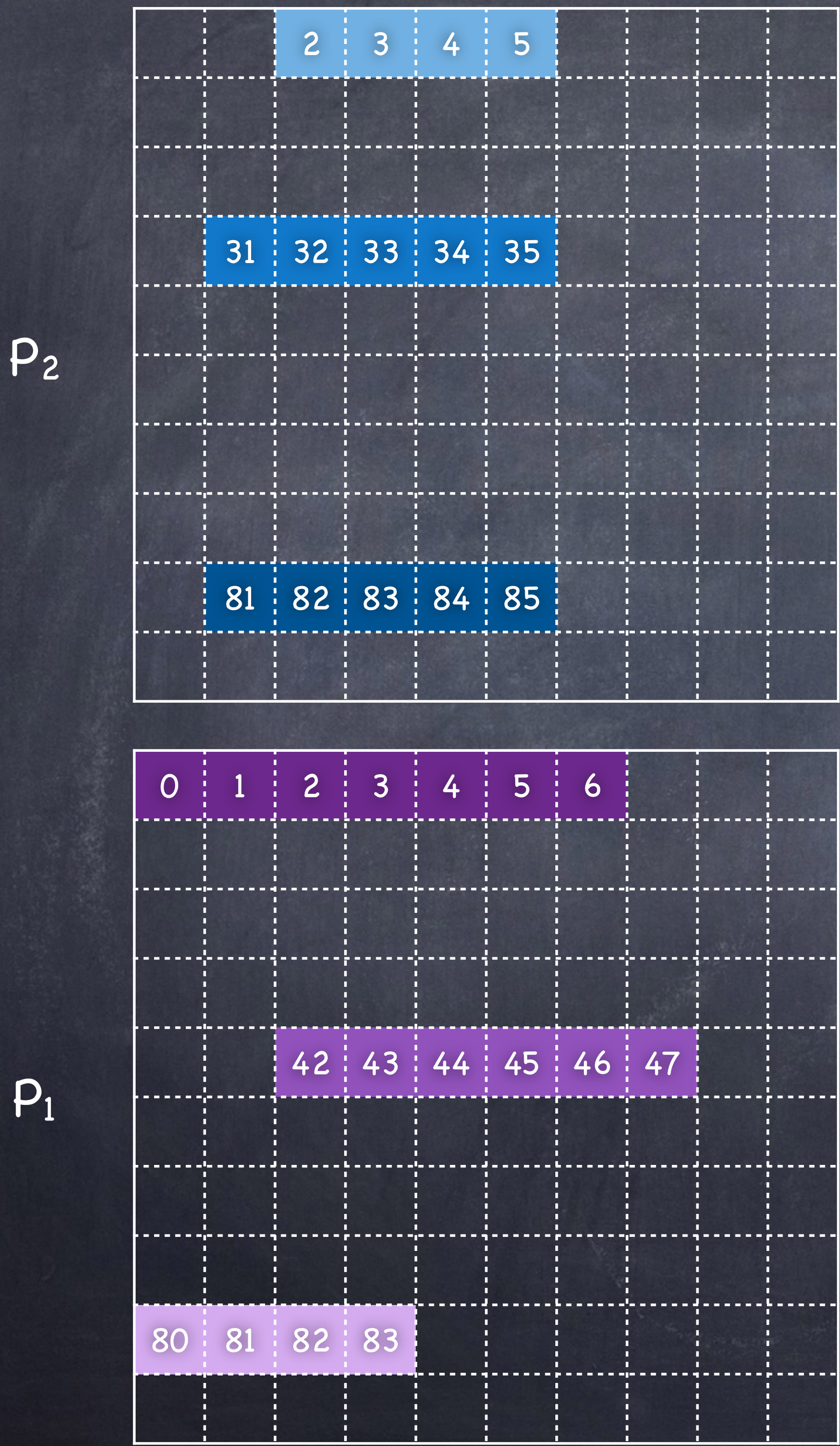
Tiling Memory



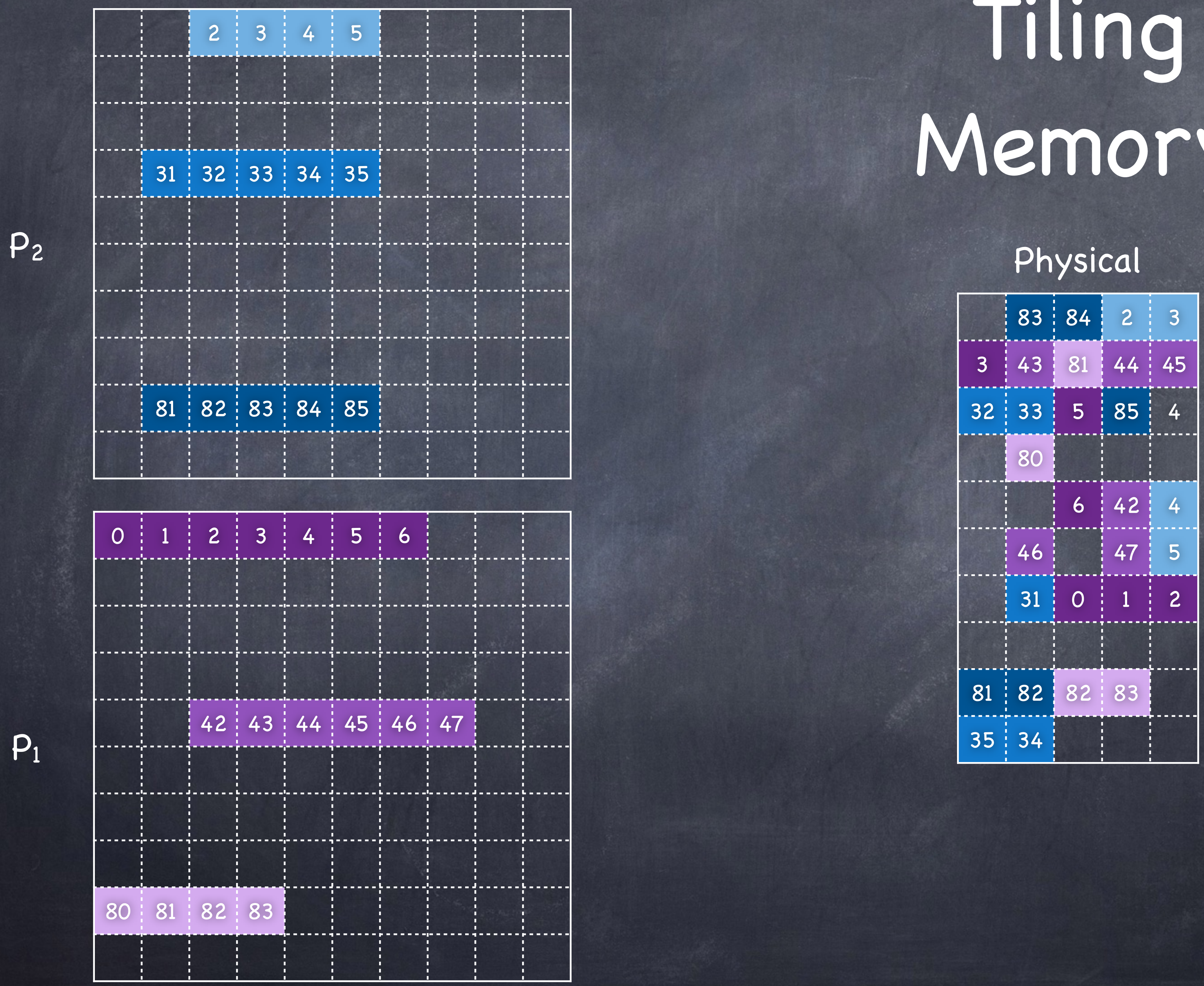
Tiling Memory



Tiling Memory



Tiling Memory

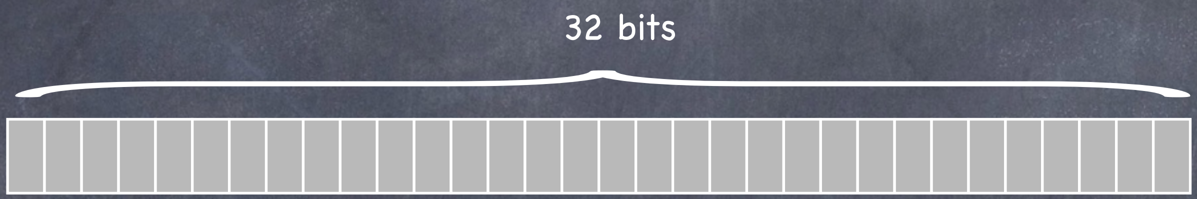


Eliminating External Fragmentation: Paging

- Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)
- Adjacent pages in VA (say, within the stack) need not map to contiguous frames in PA!
  - Free frames can be tracked using a simple bitmap
    - 0011111001111011110000 one bit/frame
  - No more external fragmentation!
  - But now internal fragmentation (you just can't win…)
    - when memory needs are not a multiple of a page
    - typical size of page/frame: 4KB to 16KB

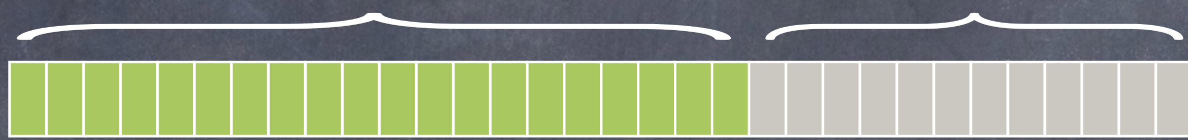
How can I reference a byte in VA space?

Virtual address



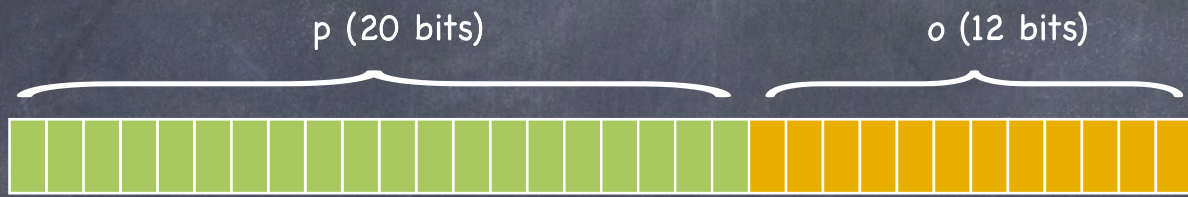
- Interpret VA as comprised of two components
  - page: which page?
  - offset: which byte within that page?

Virtual address



- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?

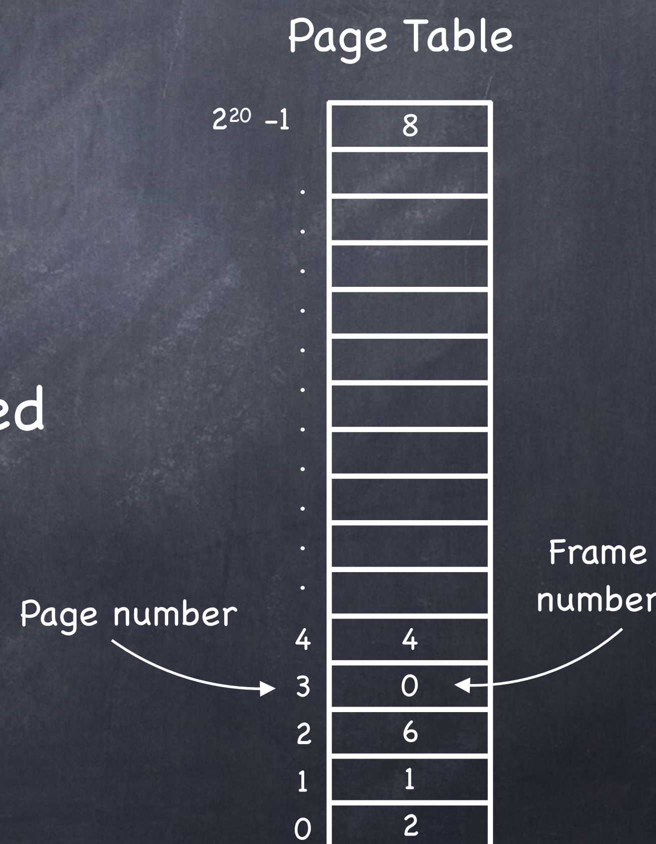
Virtual address



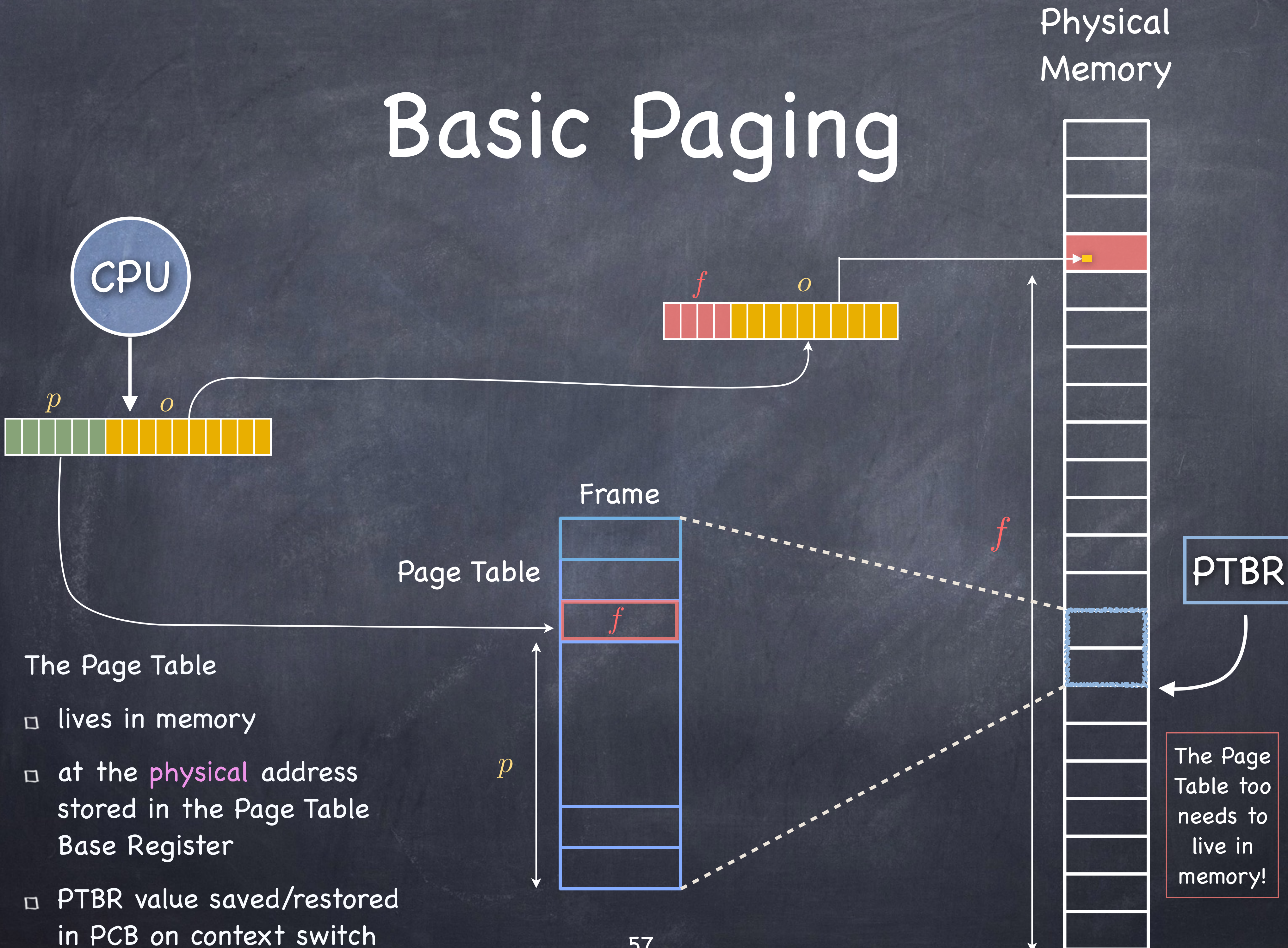
- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?
    - no. of bits specifies size of page/frame

Virtual address

- To access a byte
  - extract page number
  - map that page number into a frame number using a page table
    - Note: not all pages may be mapped to frames
  - extract offset
  - access byte at offset in frame



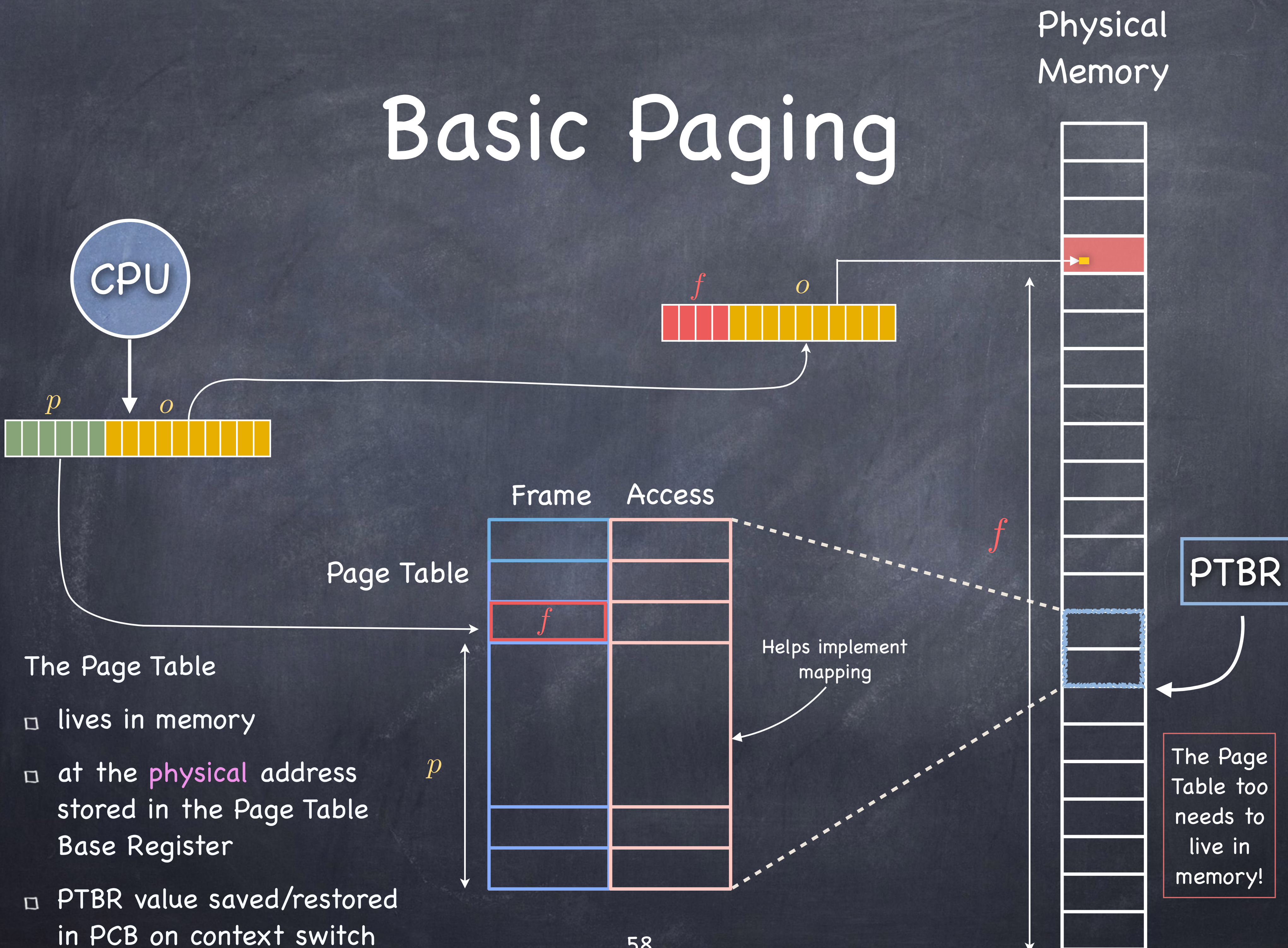
Basic Paging



The Page Table

- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Basic Paging



The Page Table

- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Page Table Entries

- Frame number
- Present (Valid/Invalid) bit
  - Set if entry stores a valid mapping. If not, and accessed, page fault
- Referenced bit
  - Set if page has been referenced
- Modified (dirty) bit
  - Set if page has been modified
- Protection bits (R/W/X)