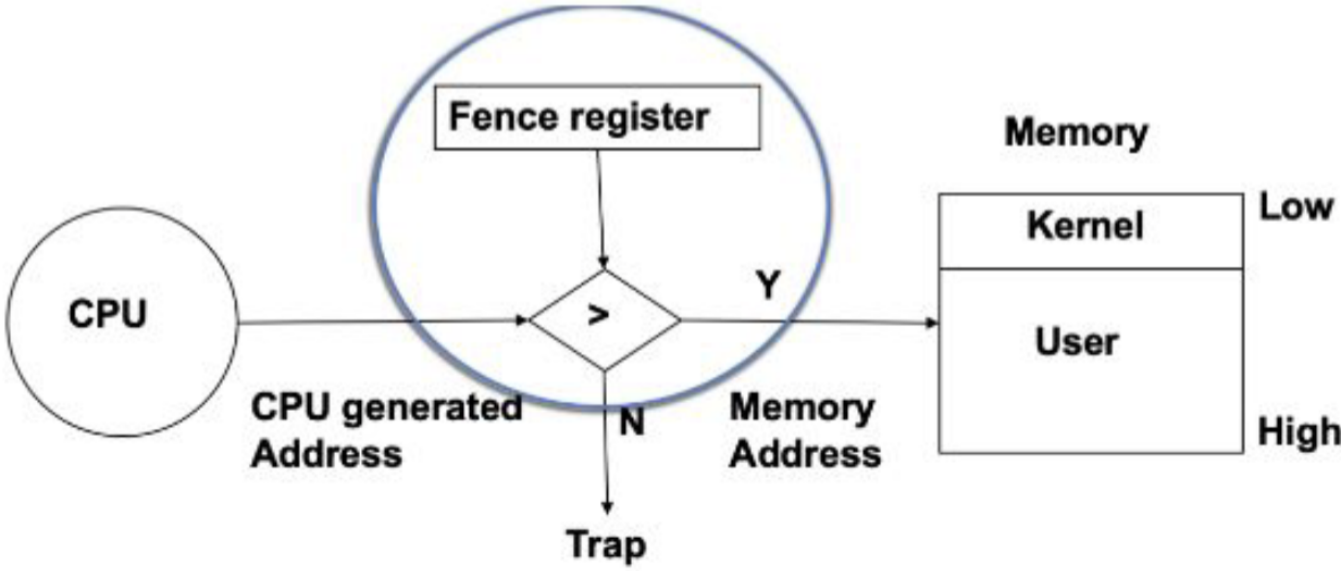
Memory Management Review

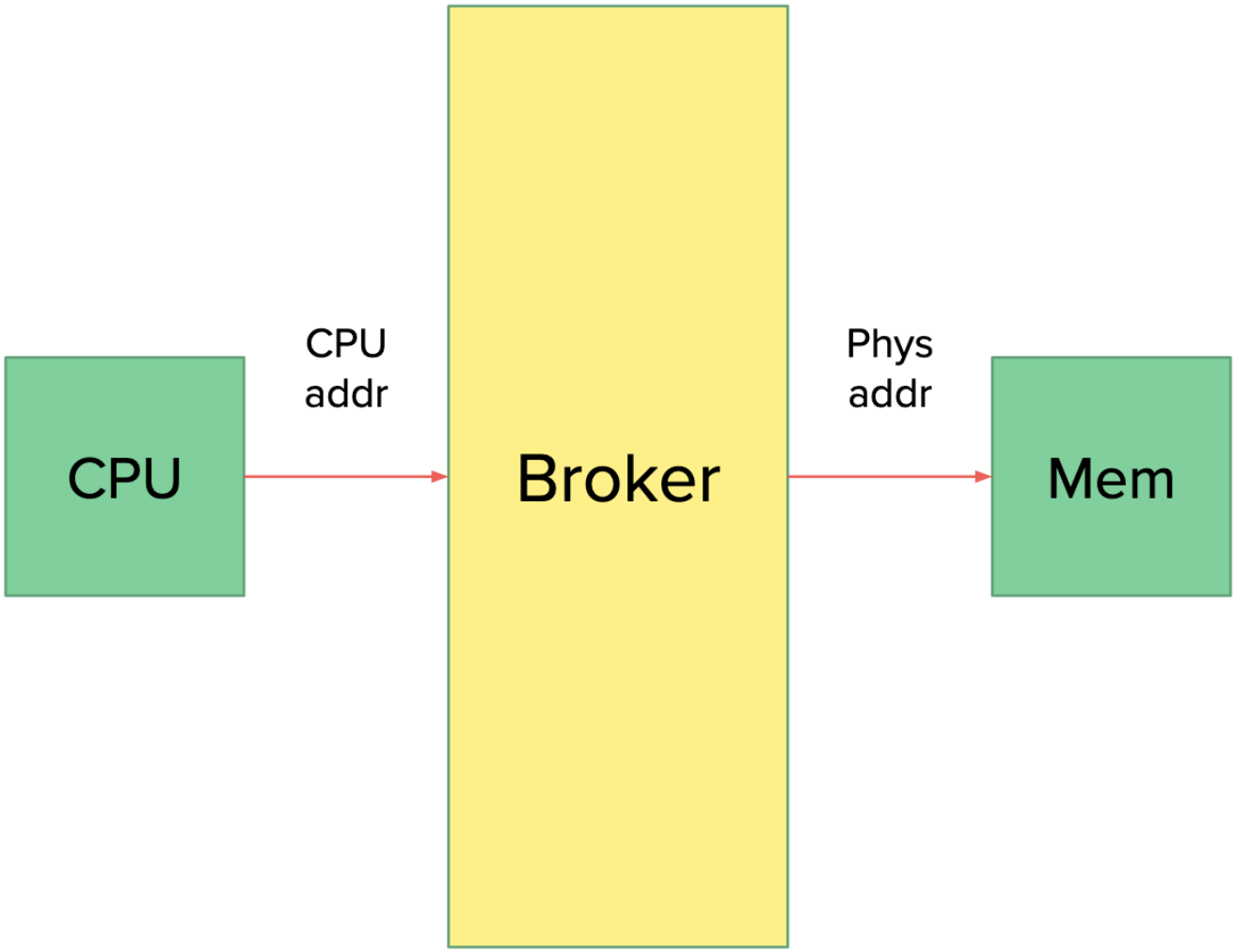
Fence Register

- Check memory address to separate kernel and user space
- Create a "fence" between the user-kernel boundary in memory
  - Kernel space is located in the lower portion of memory
  - Broker sends a trap if the user tries to access kernel space



Memory Broker

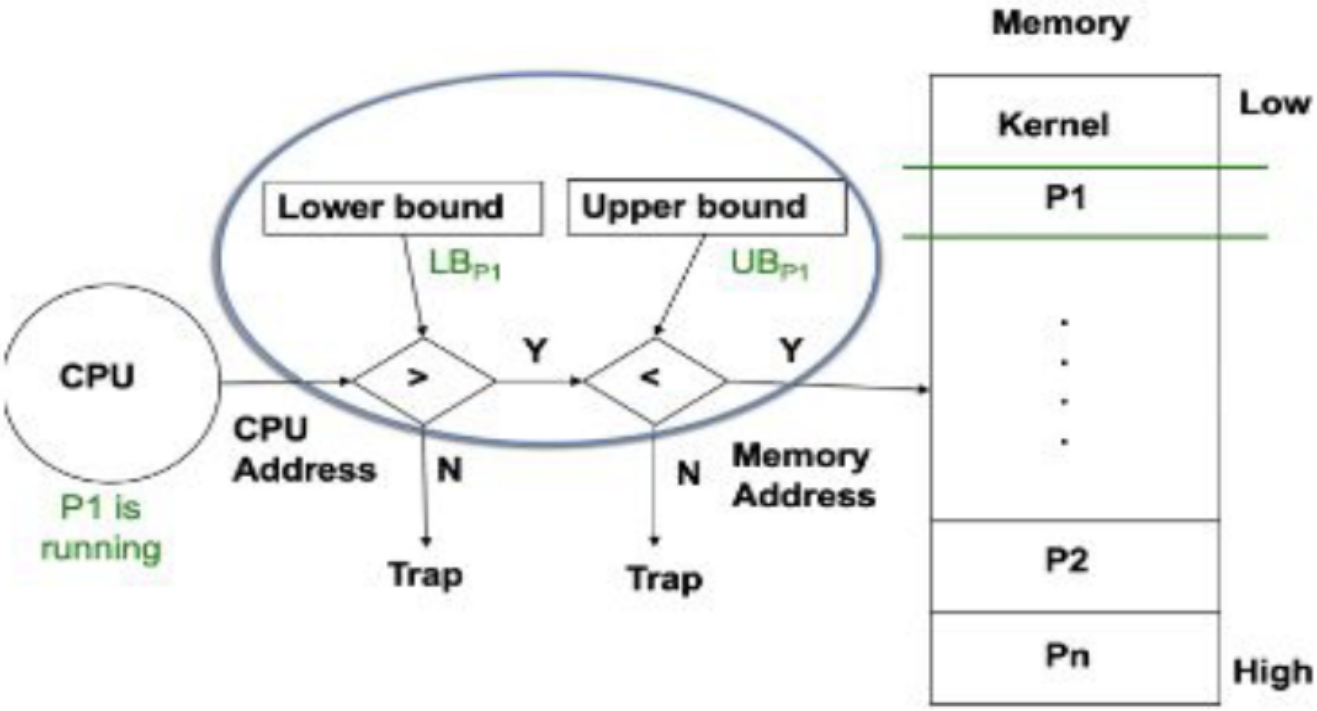
- Hardware (ewwwww)
  - Doesn't require too much external hardware
- Facilitates the goals of memory management
- Allows us to implement management policies like fence register



Static Relocation vs Dynamic Relocation

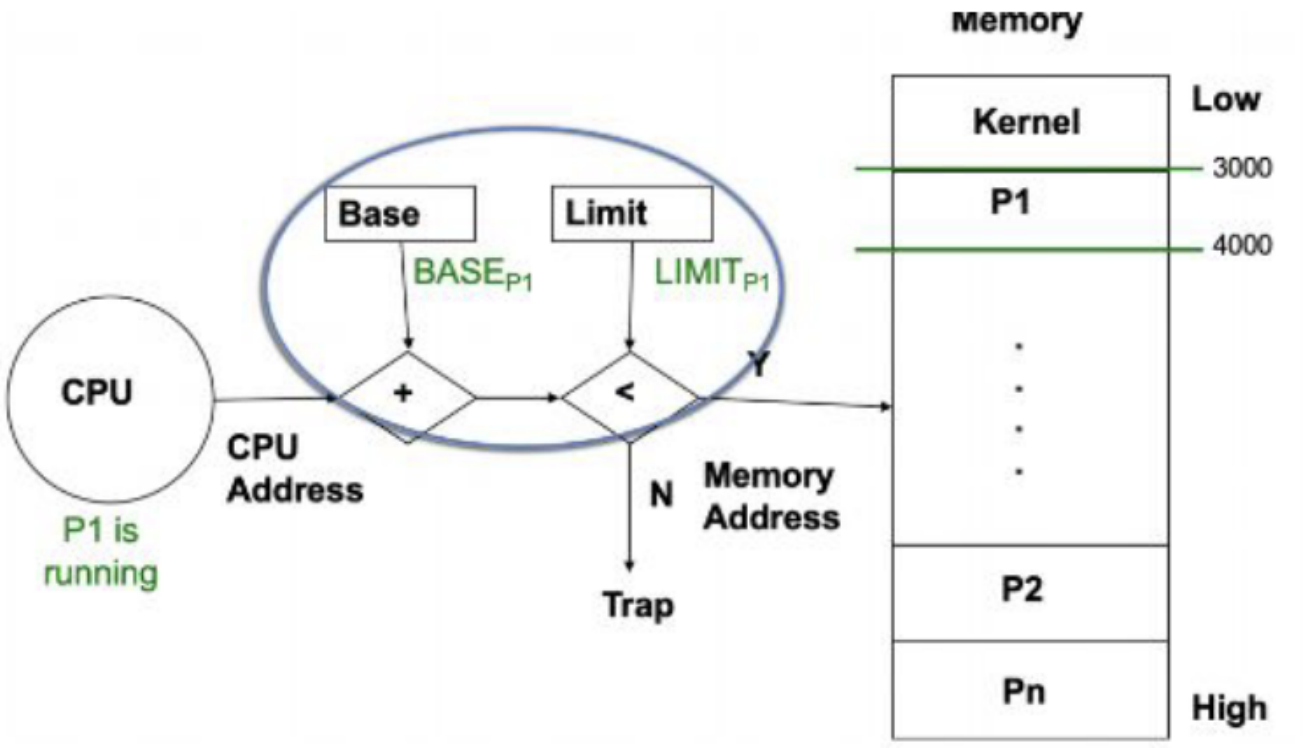
Static Relocation

- Create upper and lower bounds
- If process A and B have the same overlapping assignment, we have to swap out one to fit the other – poor memory utilization



Dynamic Relocation

- Base + limiter
- Treat addresses from the CPU as offsets from a base register
- Opportunities for better memory utilization depending on partitions



Note that the virtual paging addresses are not actual physical addresses!

Partitions

Fixed

- Allocate fixed chunk of memory for every process
- internal fragmentation = allocated space is unused → poor utilization
  - internal fragmentation = size of fixed partition - actual memory request
  - in other words, you give space to a process and the process doesn't use the whole chunk

Variable

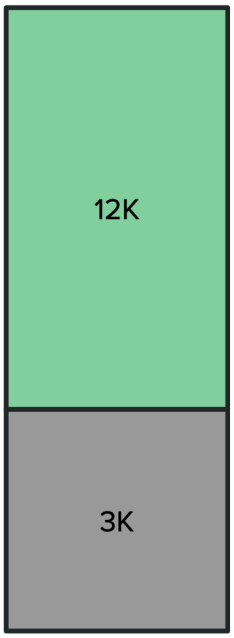
- Allocate only the size needed upon process request
  - external fragmentation = might not be able to allocate large contiguous memory chunks later
- external fragmentation = sum(non-contiguous memory partitions)
  - can be solved by compacting processes in memory as much as possible

Example Time

Adding Processes

Kinda boring, let's add some processes to our free space :)

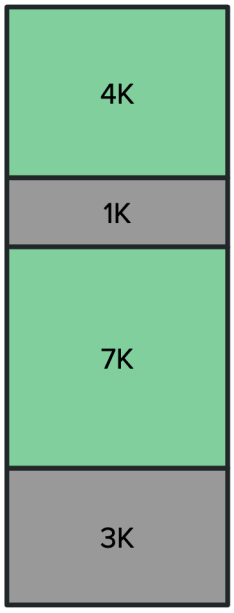
start address	size	process
0	4K	P1
4K	1K	P2
5K	7K	P3
12K	3K	*free*



P2 is done!

Process 2 is completed! Do we still need it in memory?

start address	size	process
0	4K	P1
4K	1K	P2
5K	7K	P3
12K	3K	*free*

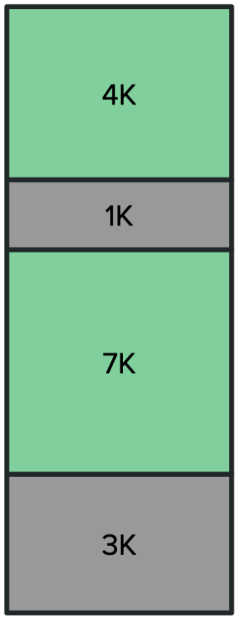


sorry the blocks are not to scale :sob:

Hello, Process 4

Let's try to add a P4 to our allocation table and memory:

start address	size	process
0	4K	P4

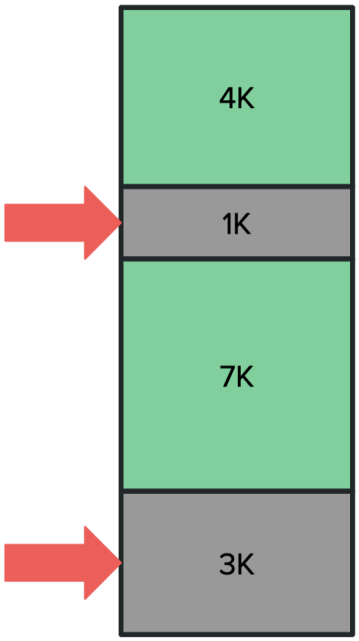


Is this possible?

What kind of fragmentation does this pose?

Mathin' the Math

What is the maximum external fragmentation represented by the memory block shown through this example?



External fragmentation = sum of non-contiguous free spaces

We have:

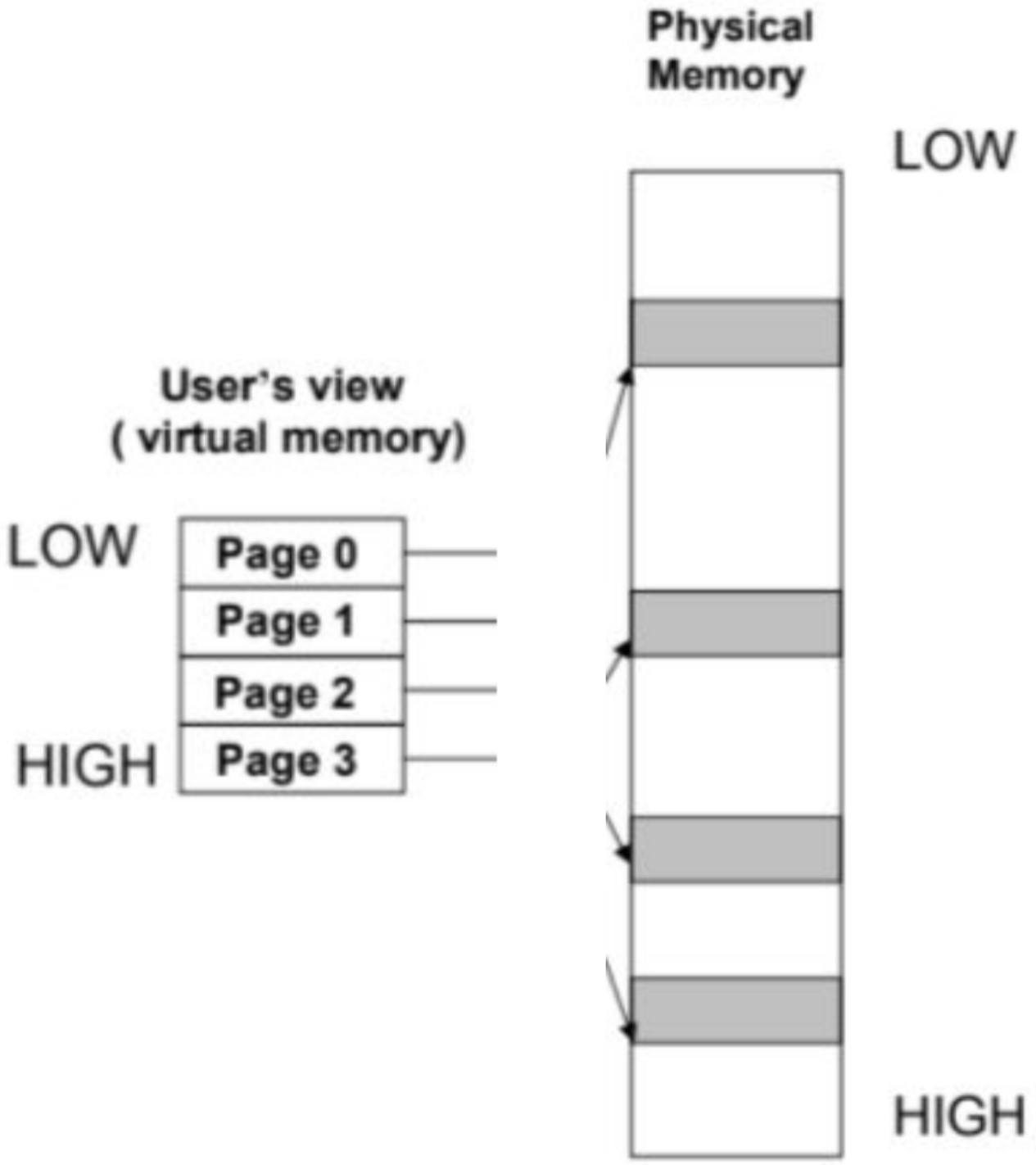
- 1K non-contiguous free space
- 3K non-contiguous free space

Hence,
External fragmentation = 1K + 3K = 4K bytes

Paging and Virtual Memory

Fragmentation Woes

- How can we minimize external fragmentation and allocate memory?
- Divide memory (physical and virtual) into fixed-sized portions called pages
  - Virtual memory = virtual pages
  - Physical memory = physical frames
- Overview of how it works
  - Memory management hardware maps virtual pages → physical frames
  - User only sees contiguous virtual pages, but in reality they are all over the place



Virtual Addresses

We use Virtual addresses to store things in discontiguous locations in memory

- Page offset: n is the number of least significant bits
- Virtual page number: virtual address length - n most signficant bits (offset size)

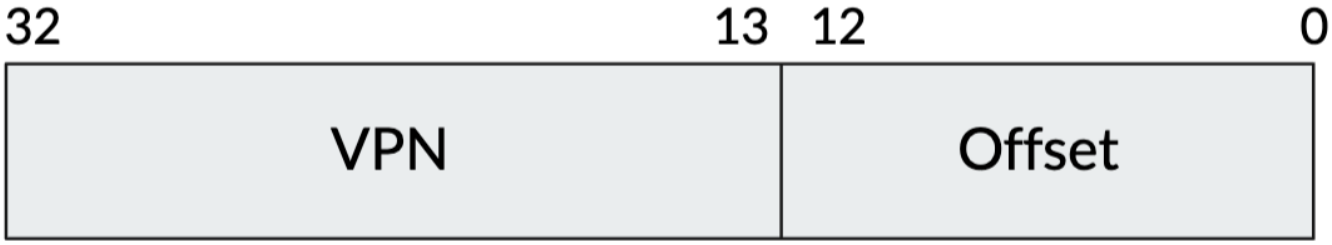
The page size will always equal 2^n. This n is used to calculate the page offset and VPN

- we need n bits to access any given portion in a page of size 2^n

Virtual Address Anatomy and Example

Let's say we have a page of 8KB and 32 bit virtual address space

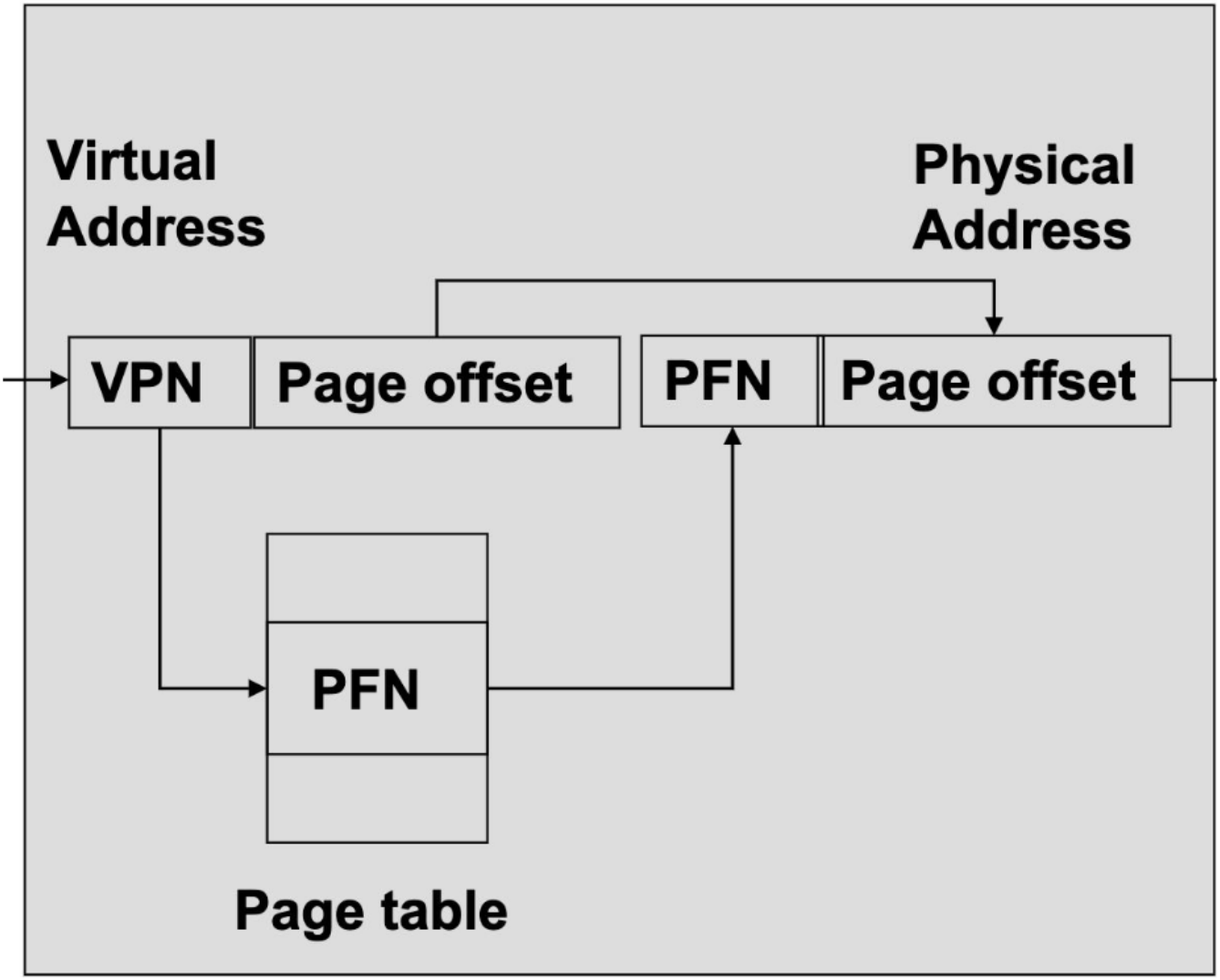
- 8KB = 2^{13}, so 13 bits for the offset
- $32 - 13 = 19$ bits for the VPN



Page Table and Address Translation

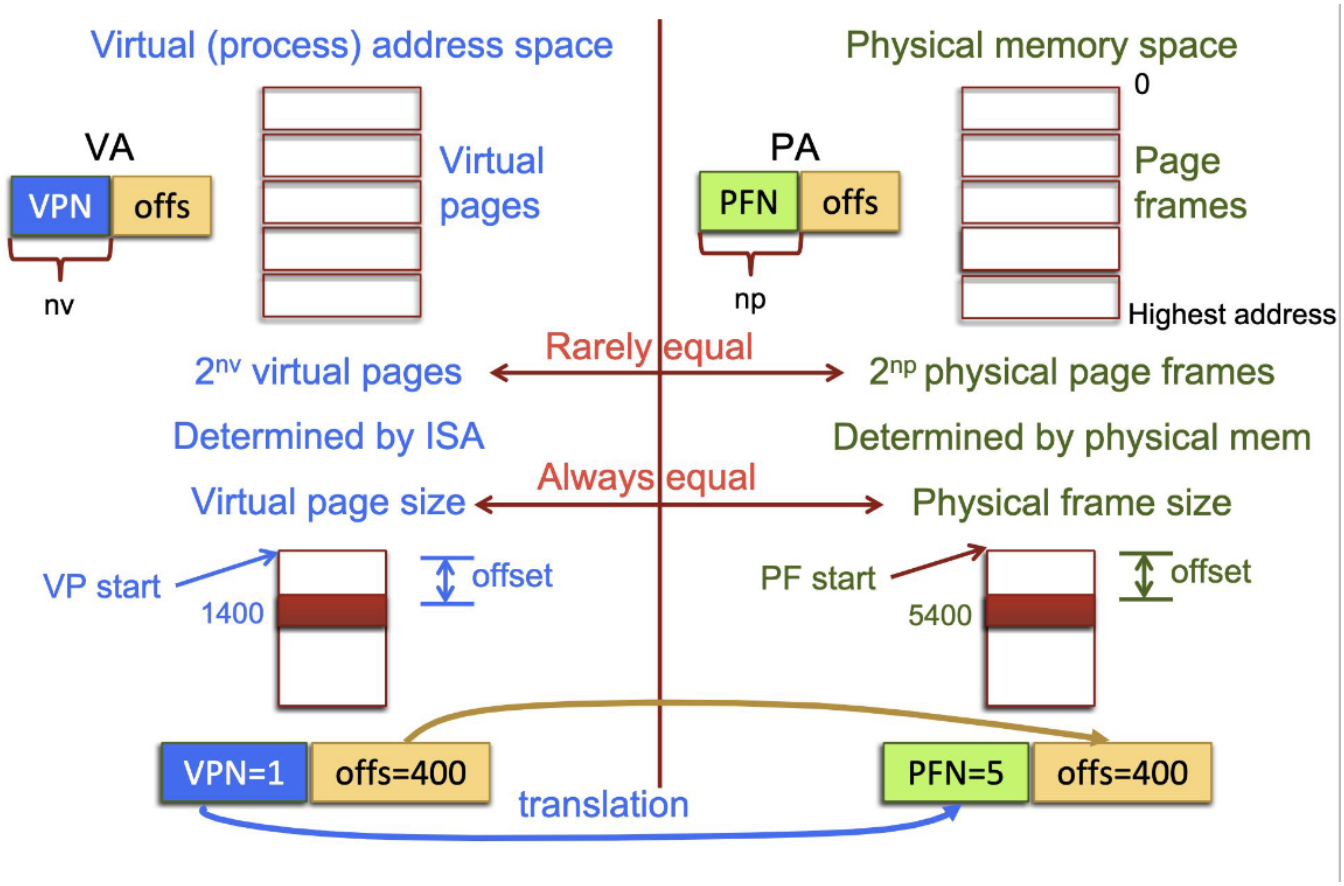
We use the VPN to index the Page Table

- Page table holds **Page Frame Numbers (PFNs)**
- PFNs tells us which page frame to look in
- Offset is carried over from the virtual address



Notable Facts

- The size of virtual pages and physical frames are always the same
  - This is because they have the same offset size
- **of page table entries = number of bits in the VPN**
- **of virtual pages and physical frames is rarely the same**



More Page Tables!!

How many page tables do we have?

- One for each active process

Where do we store the page table

- In memory!
- No need for any additional hardware

We use a **Frame Table** for reverse translations (PFN → PID, VPN)

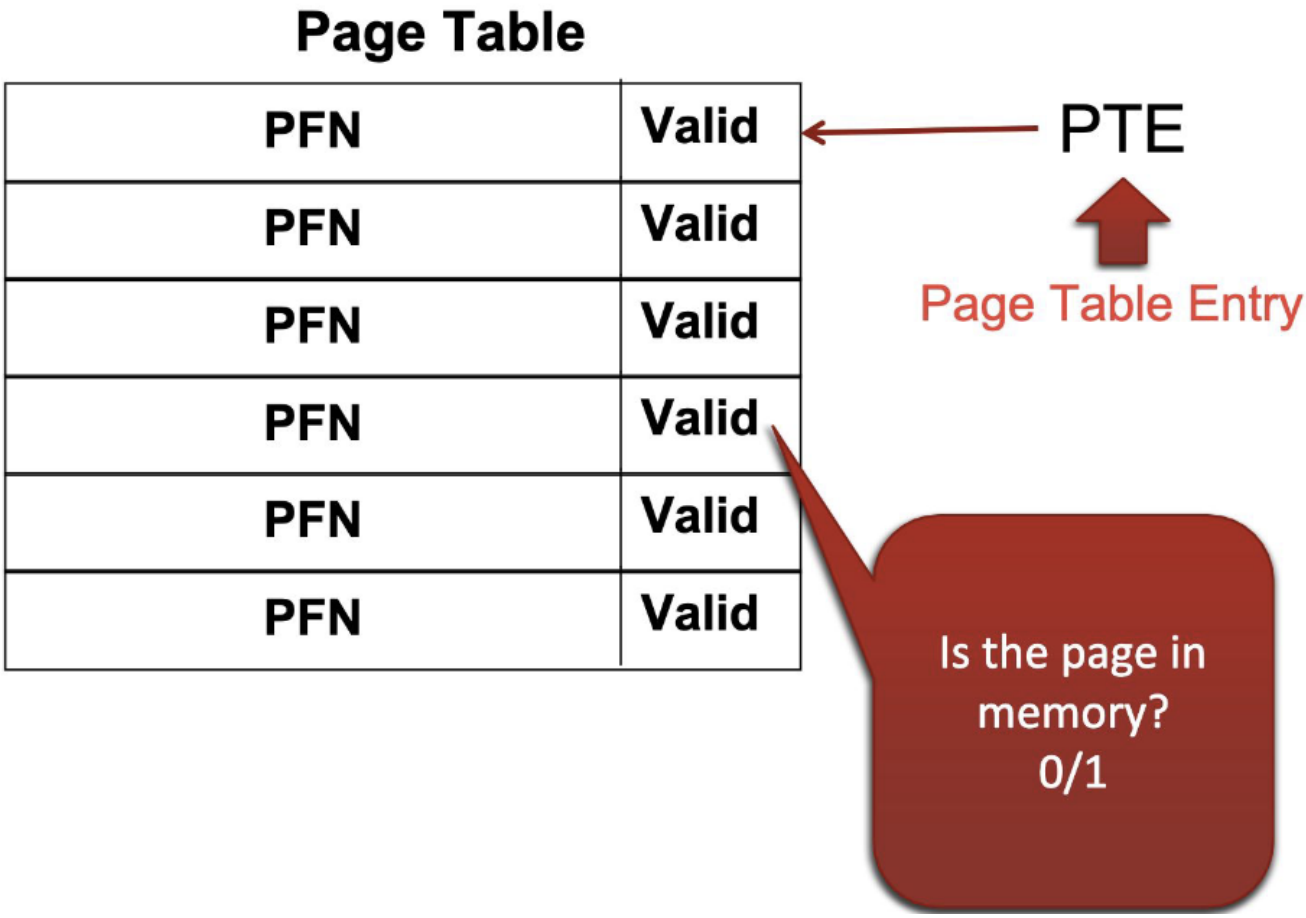
We will add the **Page Table Base Register** (**PTBR**) to our PCB to keep track of the base physical address of the current page table.

- We need to add a hardware register for this
- We change PTBR after each context switch (different process)

Page Allocation

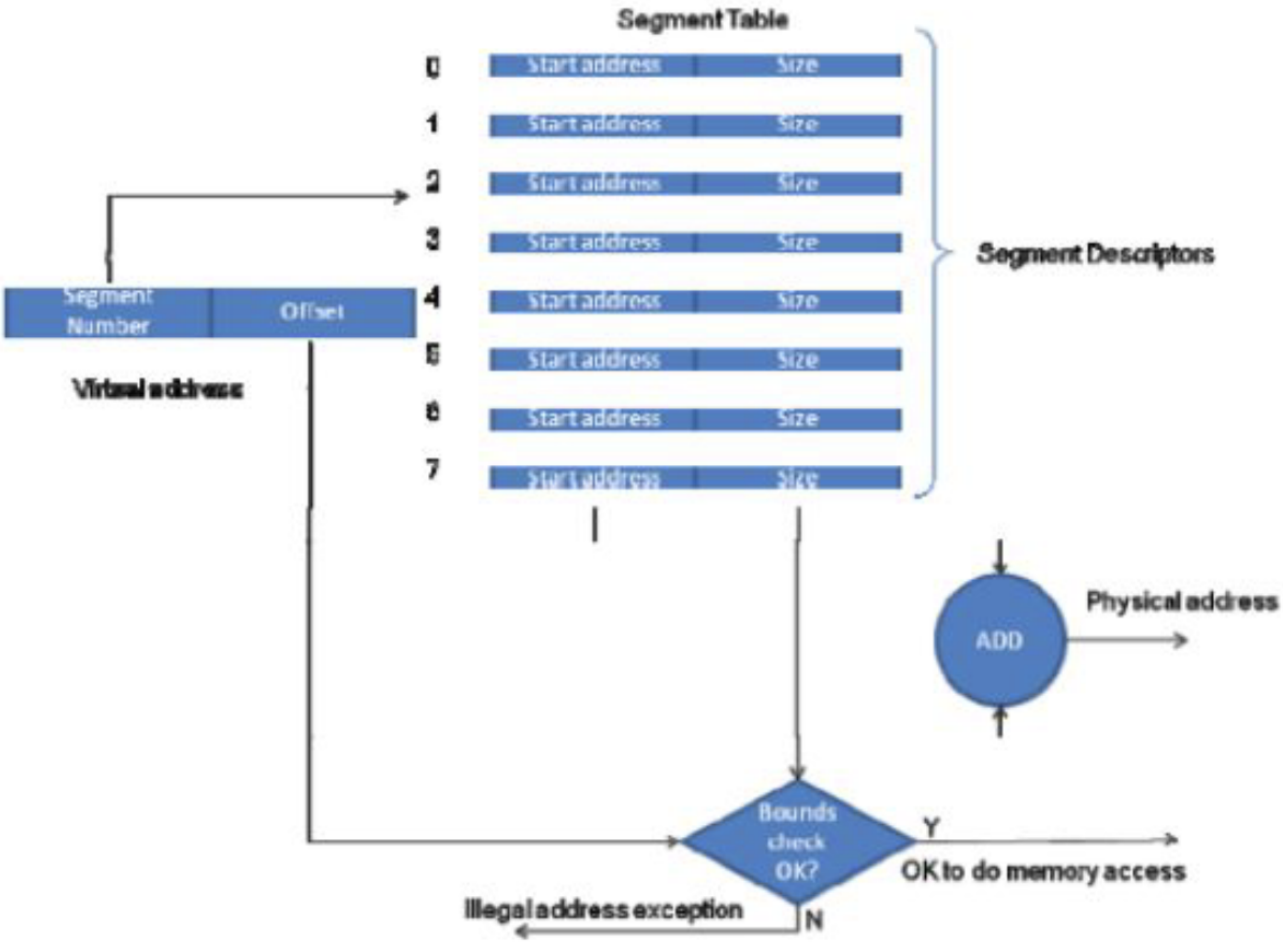
- We want to allocate pages on demand as it is more efficient than allocating pages all at once

- We need a new data structure to keep track of valid physical frames
- **PTE** (**page table entry**): We use each page table entry to store the PFN and if that given page is valid or not



An Alternative: Segmented Virtual Memory

- Divides a process's address space into logical sections called segments
  - Code, global data, heap, stack
- Segments have a unique segment number and size
  - Stored in segment table
- Addresses have a segment number and offset
  - Address translation: get starting address of segment from segment table, add offset



Paging vs Segmentation

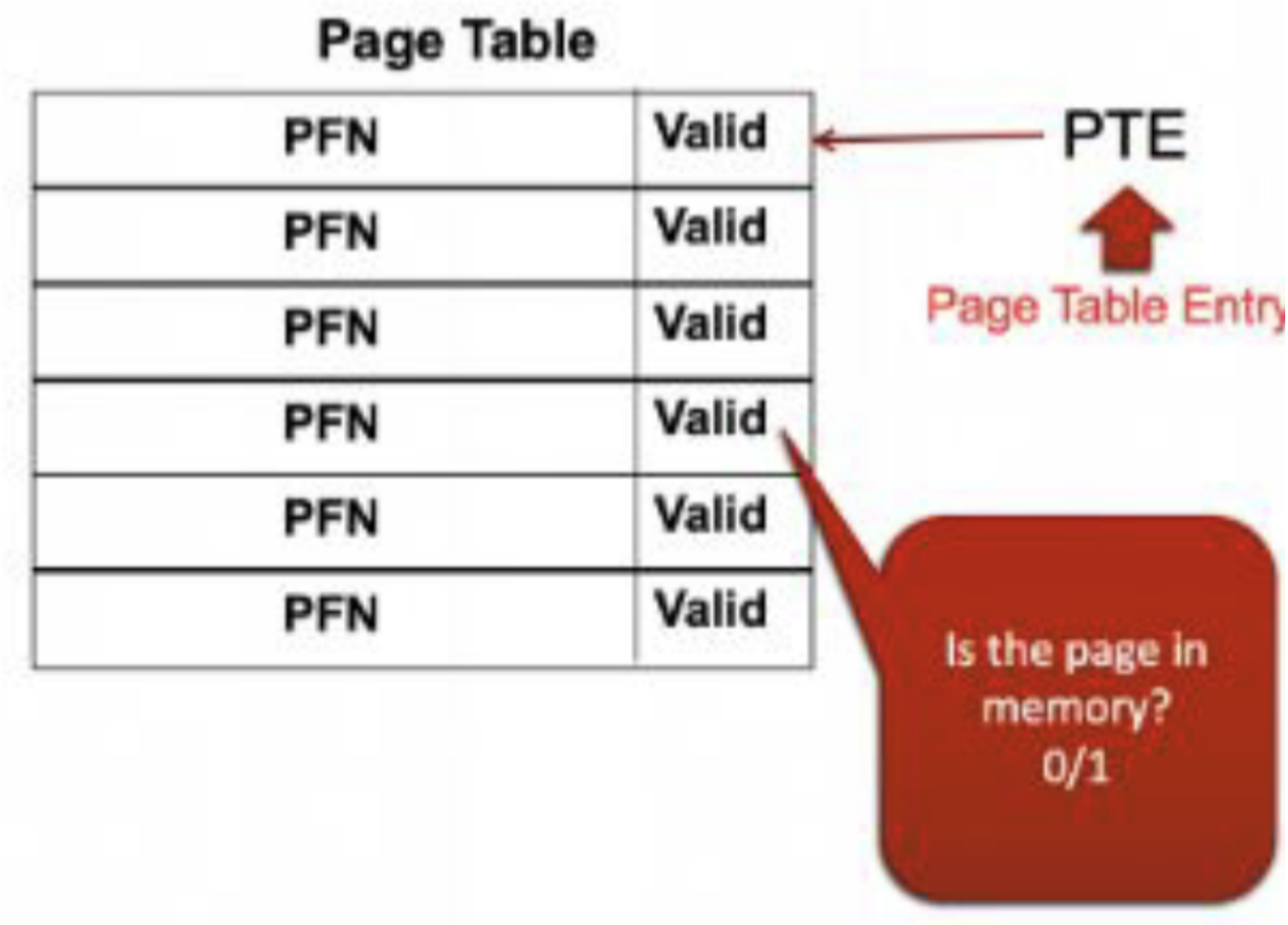
Key takeaways:

- Paging presents internal fragmentation, segmentation presents external fragmentation
- User is more aware and has more control in segmented systems because everything is split up into individual segments
- User can't pick page size but can pick segment size

Page Faults

Demand Paging

- Don't have to allocate all the needed memory when the process starts
- Just need to allocate when we access!
- Need new data structure to keep track of frames and page fault handler



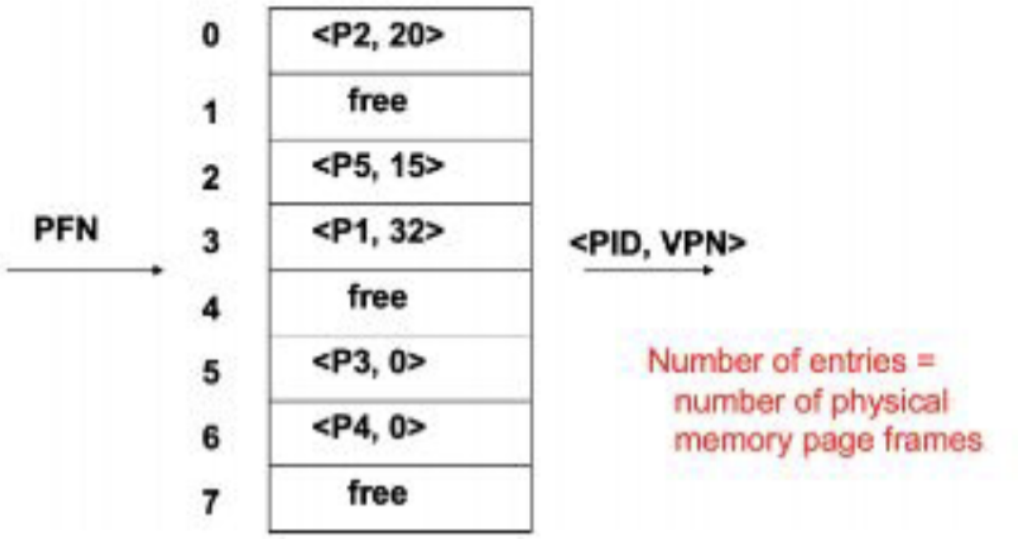
Page Fault Steps

1. Find a free page frame
2. Load the faulting virtual page from the disk into the page frame
3. Give up the CPU while waiting for the paging I/O to complete
4. Update the page table entry for the faulting page
5. Place the PCB of the process back in the *ready queue* of the scheduler
6. Call the scheduler

Data Structures Involved

- Free-list
  - Contains a list of unused frames
  - Memory manager can use any of the frames from this list during a page fault
- Frame table
  - Reverse mapping from a frame number to process and VPN
  - Can contain other metadata

- Disk Map
  - Maps the processes' virtual pages to locations
  - Why do we need this?
- Modifications to Page Table
  - Add a valid bit
  - Add a dirty bit



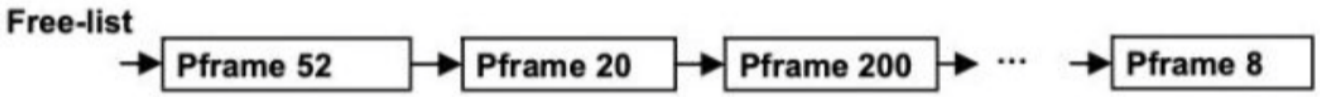


Figure 8.3: Free-list of page frames

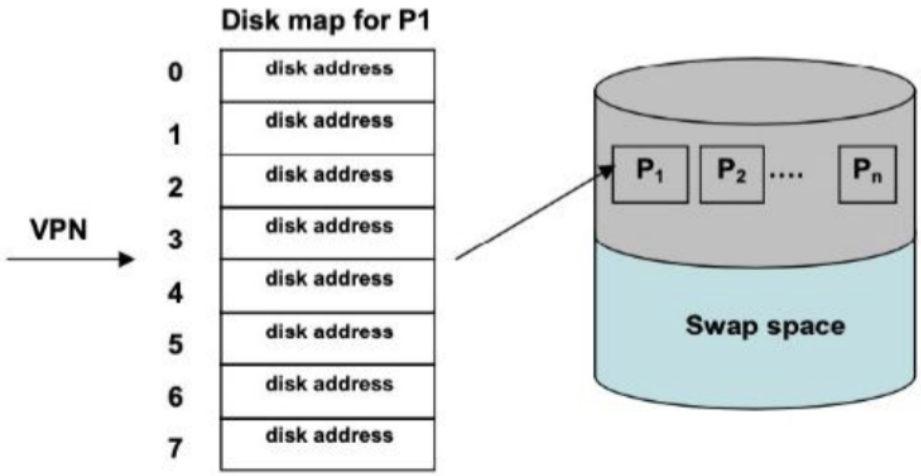


Figure 8.5: Disk map for Process P1

Victim Page Selection

Overview

- How do we decide which page to evict?
- How do we evict the page?

If we select a victim frame, we still need a way to recover the frame because the other process might still need it

- Frame table comes in handy
- Swap the page onto secondary storages if needed
  - Clean page: contents of the frame has not changed
  - Dirty page: the contents of the frame has changed. Memory manager needs to flush (write page back to disk)

The Lucky Frame

Goal: minimize the number of page faults

Local Victim Selection

- Select victim frame from the faulting process, don't disturb other processes residing in memory
- we won't need the frame table for local victim selection
- poor utilization of memory, because we'll have wasted frames

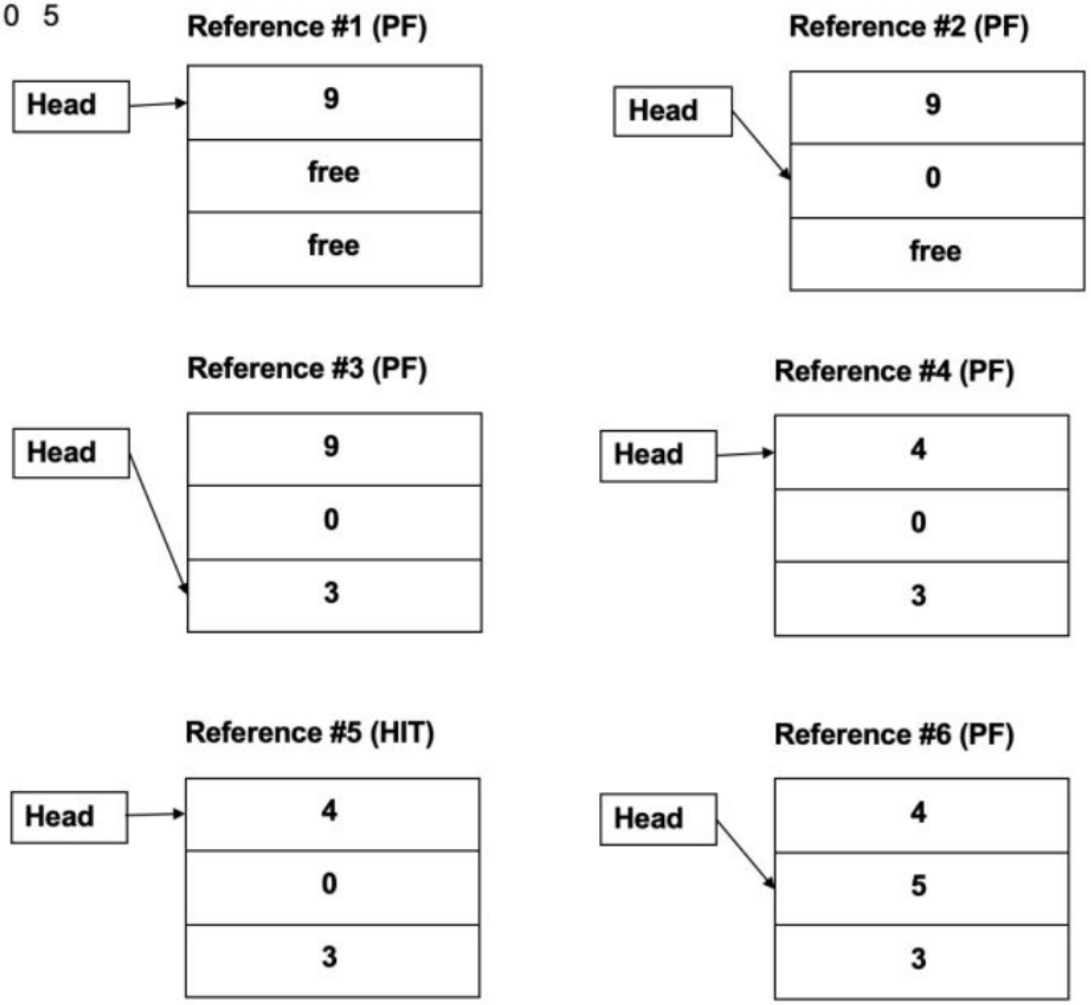
Global Victim Selection

- select the victim frame from any processes
- good utilization due to the potential to use the entire memory

First In First Out (FIFO)

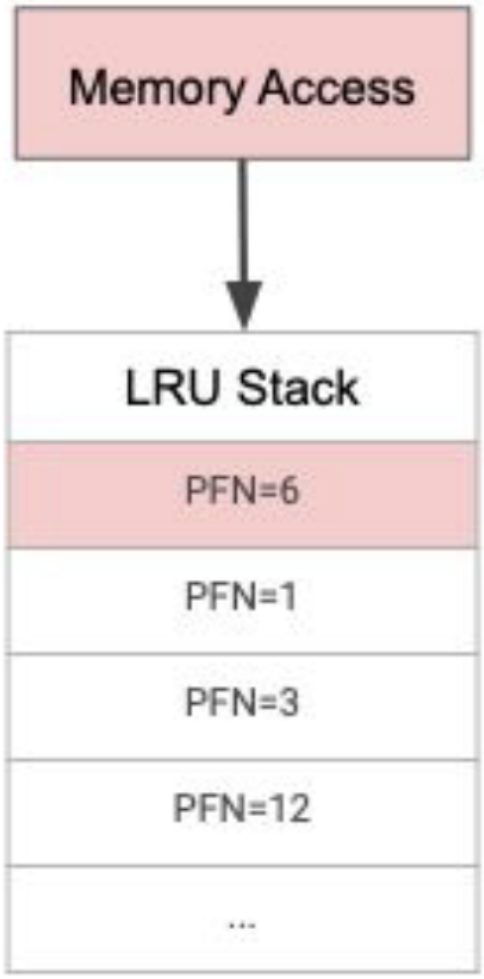
- Evicts the first used page frame
- Can use Frame Table as a queue
- Points the head pointer to the first frame table

Ref: 1 2 3 4 5 6
VPN: 9 0 3 4 0 5



Least Recently Used (LRU)

- Free the page that is least recently used
- Use a stack to keep track of recently referenced pages
- Ideal but not practical
  - Data structure is too large to implement

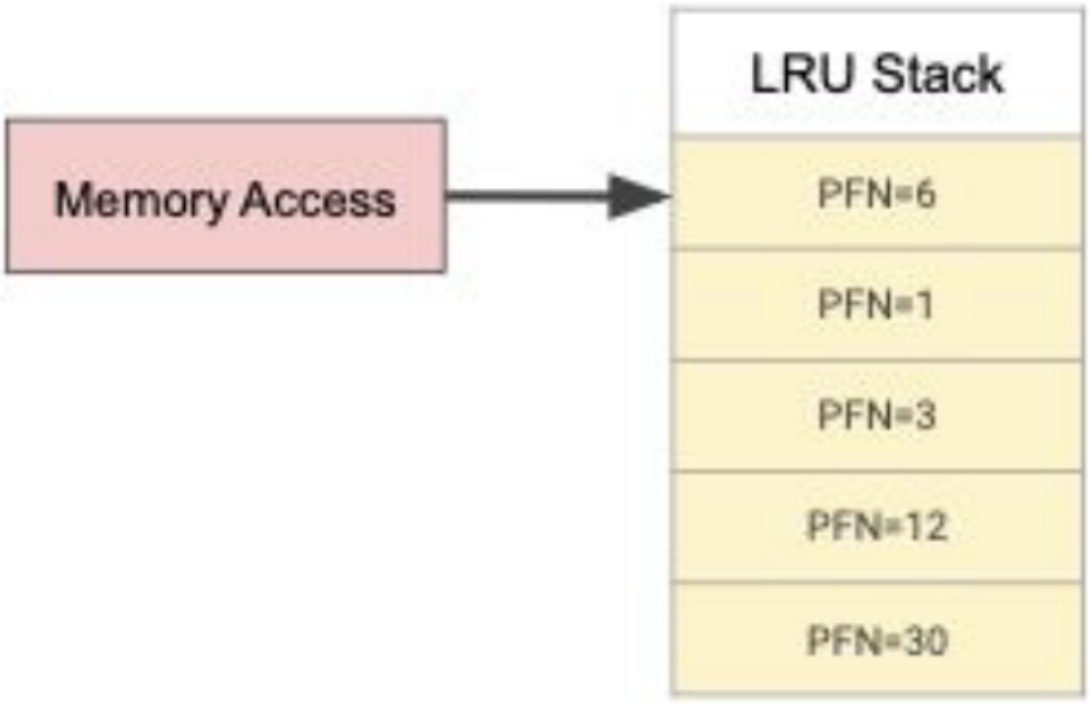


We can use a **hardware stack** to keep track of the recently referenced frames...but is it really practical?

Approximating Least Recently Used (LRU)

Method 1: Use smaller stack to keep track of the N least recently used pages
Method 2: Use a reference bit

- Referenced bit set to 1 when accessed
- Periodically dumps reference bit to the counter
  - `counter = (ref << 31 | counter >> 1)`
  - The frame with the smallest counter value is the least recently referenced



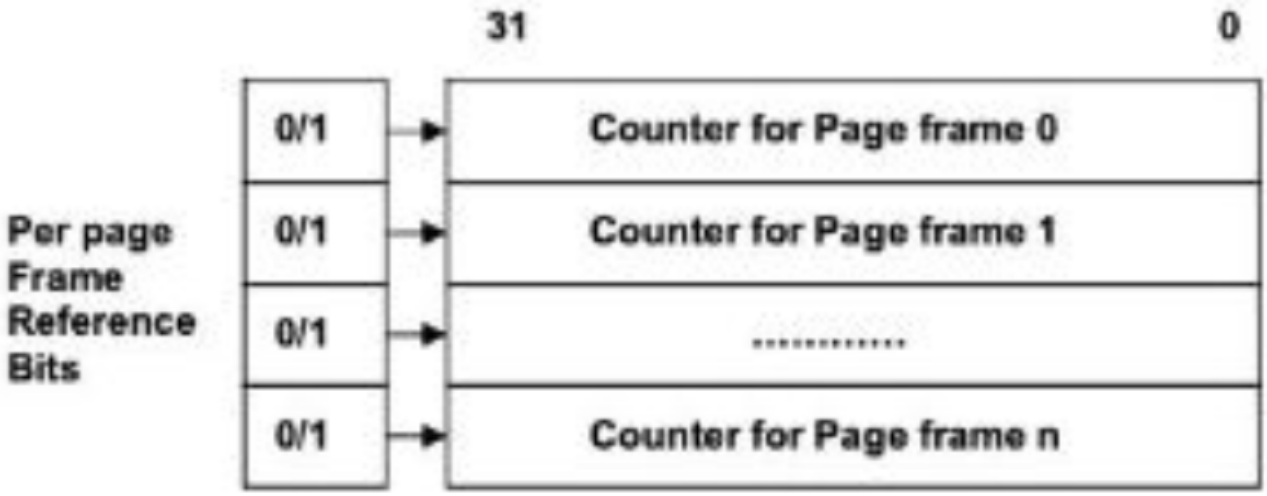


Figure 8.10: Page frame reference counters; The reference bit for each page may be 0 or 1

Second Chance Replacement

Optimization that we can apply to FIFO is to utilize the reference bit and avoid evicting the recently

- Hardware sets the reference bit
- When a page needs to a replaced, OS selects a page in FIFO manner, but skips any frame with the reference bit set

- If all of the frames have been referenced, the victim is the first candidate in FIFO order

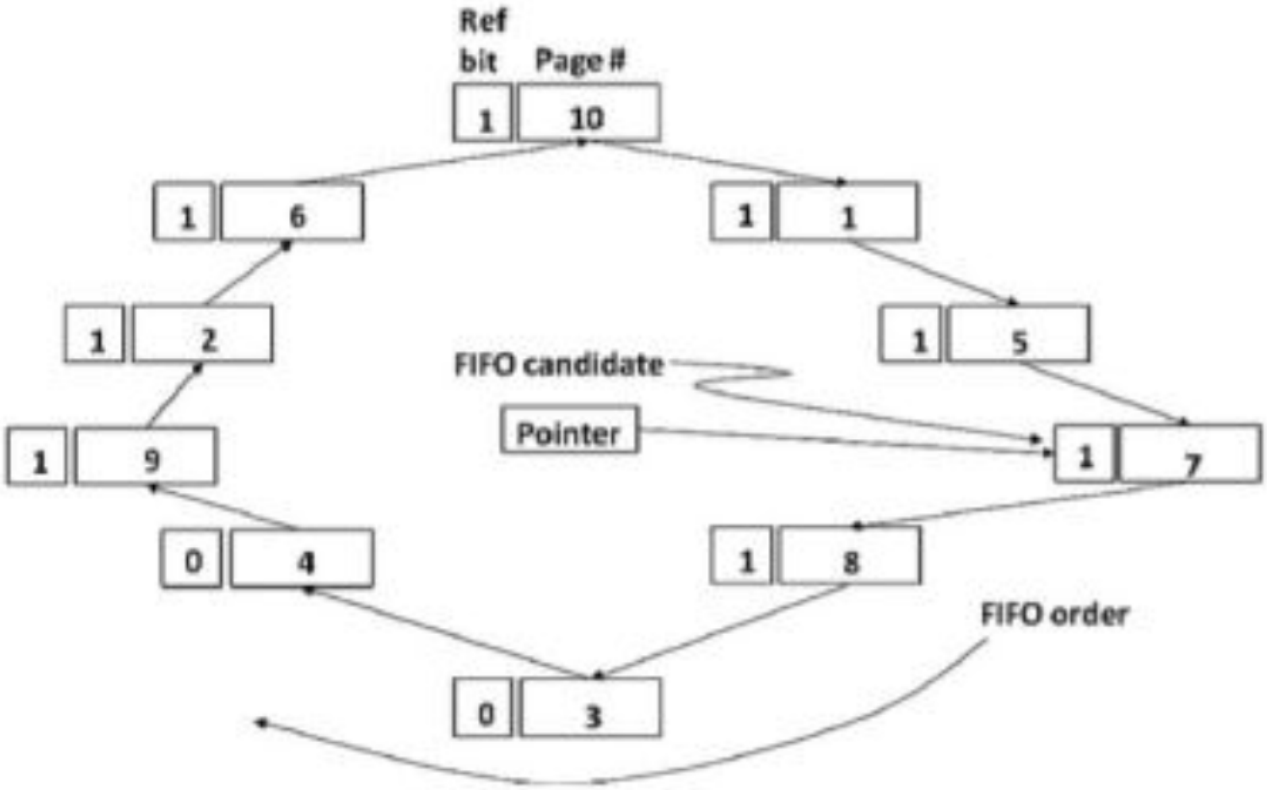


Figure 8.11a: Second chance replacement – the memory manager keeps a software pointer that points to the FIFO candidate at the start of the algorithm. Note the frames that have reference bits set and the ones that have their reference bits cleared. The ones that have their reference bits set to 1 are those that have been accessed by the program since the last sweep by the memory manager.

Example

Second Chance Replacement

Let's take a look at the following reference string:

0 4 1 4 2 4 3 4 2 4 0 4

(these numbers represent pages that host processes we need to run)

Frame 1	Frame 2	Frame 3
??	??	??

Initialization

We'll keep the reference string here for easy access:

0 4 1 4 2 4 3 4 2 4 0 4

Let's set up the frame table to have the first three pages, and the second chance array to fill with 0s.

Frame 1	Frame 2	Frame 3
0	4	1

0	0	0

P4

We'll keep the reference string here for easy access:

0 4 1 4 2 4 3 4 2 4 0 4

Checking for and adding 4 to the frame table will just change the second chance

Frame 1	Frame 2	Frame 3
0	4	1

0	1	0

P5

We'll keep the reference string here for easy access:

0 4 1 4 2 4 3 4 2 4 0 4

Checking for and adding 2 to the frames

Frame 1	Frame 2	Frame 3
2	4	1

0	1	0

P6

We'll keep the reference string here for easy access:

0 4 1 4 2 4 3 4 2 4 0 4

Checking for and adding 2 to the frames

Frame 1	Frame 2	Frame 3
2	4	1

0	1	0

P7

We'll keep the reference string here for easy access:

0 4 1 4 2 4 3 4 2 4 0 4

Checking for and adding 3 to the frames and updating the second chance bit

Frame 1	Frame 2	Frame 3
2	4	3

0	0	0

This is essentially the clock-speed algorith for project 3

Project 3

The system has a 24-bit virtual address space and memory is divided into 16KB pages

- 2^{14} = 16KB or 14 bits for the offset
- $24 - 14 = 10$ bits for the VPN

Translating has to do with the disk map (that is where your physical addresses are)

- Any physical translations can be associating with the disk map

`(PFN * PAGE_SIZE) + Offset`

If you're indexing into frames, use the frame table and not memory!

- Frame table does opposite of page table
- Remember when to use which
- Frame = take physical and give you virtual page number