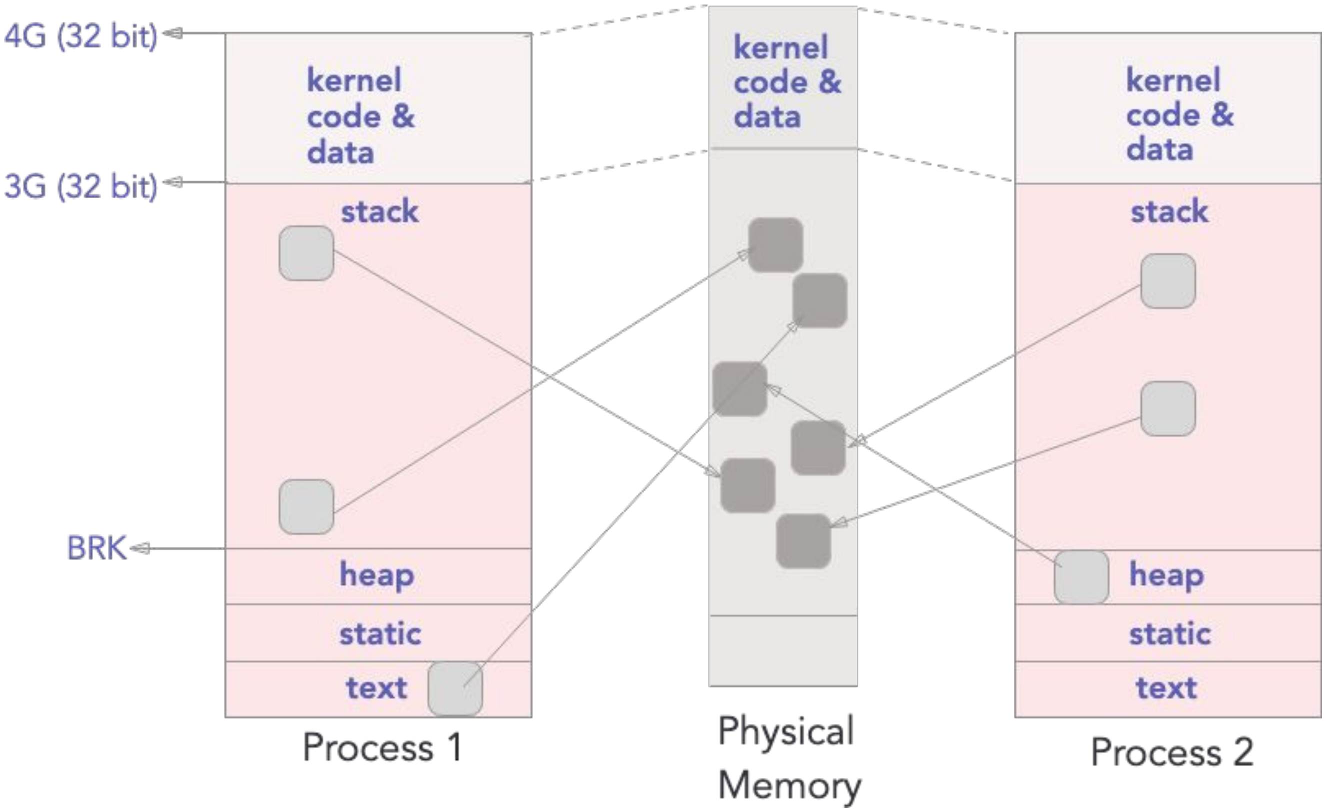
Paging

W4118 Operating Systems I

https://cs4118.github.io/www/2024-1/

Credits to Jae

Reminder: Virtual Address Space

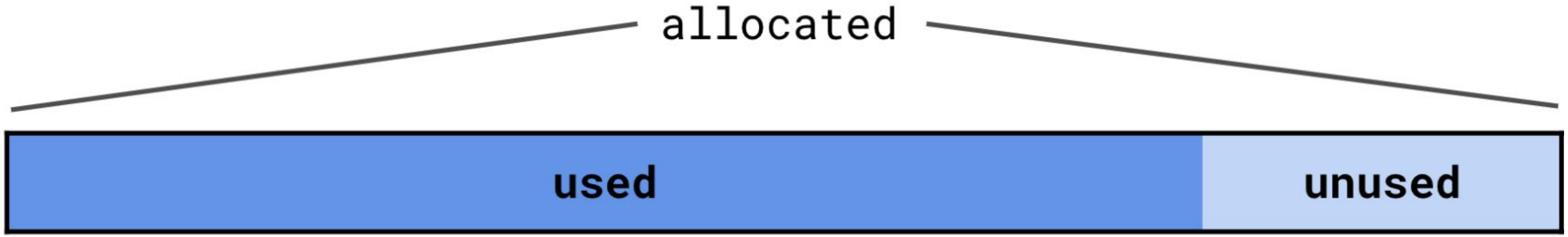


Memory Management Goals

- **Sharing:** multiple processes should coexist in physical memory
- **Transparency:** a given process shouldn’t be aware about sharing physical memory
- **Protection:** processes shouldn’t be able to access memory belonging to other processes or kernel
- **Efficiency:** physical memory should not be wasted
- **Performance:** shouldn’t trap into the kernel for every pointer dereference

Efficiency: Avoid internal fragmentation

Space in an allocated chunk of memory goes unused



- **Solution: Allocate memory in smaller chunks**
- **Pitfall: Too many allocations and high bookkeeping cost**
- **Goal: Balance chunk size with allocation/bookkeeping overhead**

Efficiency: Avoid external fragmentation

While there may be X bytes of free space, those X bytes may not be contiguous, meaning that the allocator can't create a chunk of X bytes

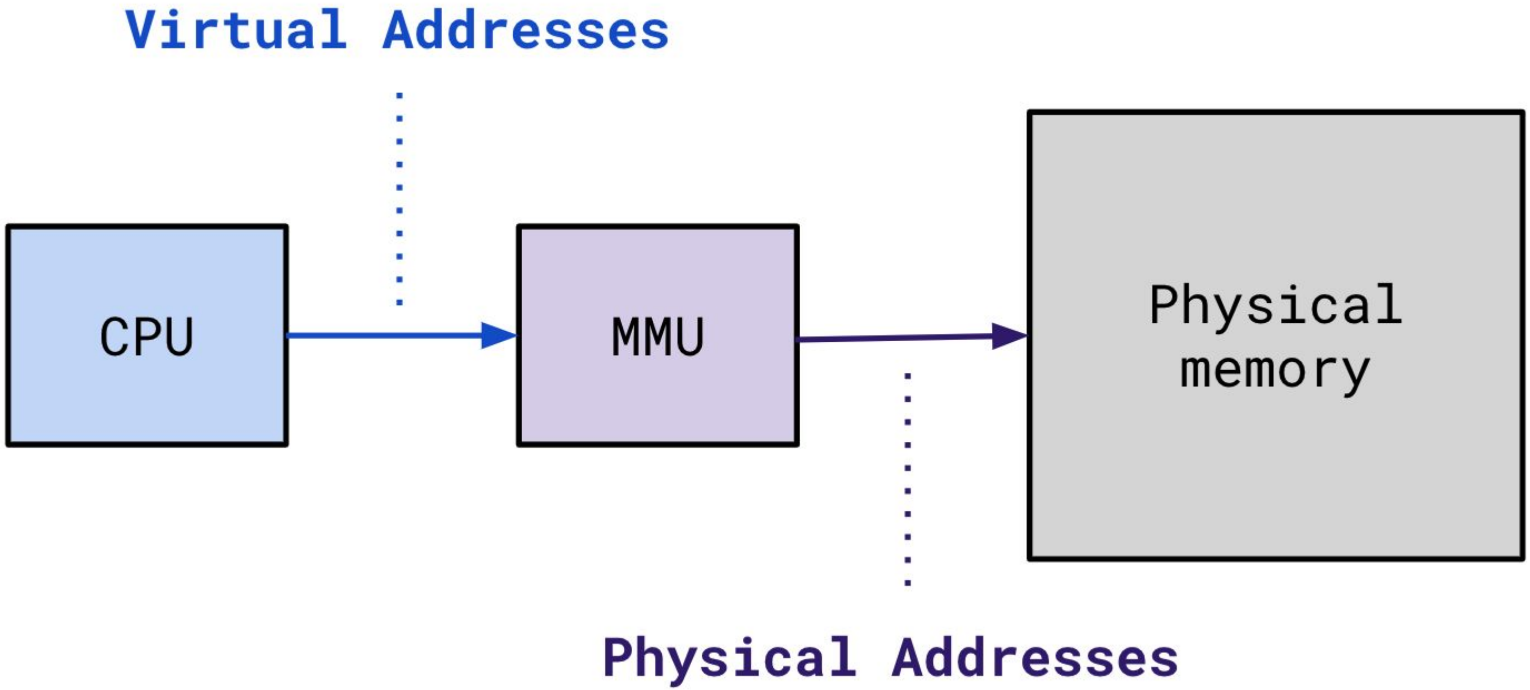
- **Solution:** **Defragmentation (make free chunks contiguous)**
- **Pitfall:** **Requires extensive data movement**
- **Need to avoid doing it as much as possible**

Selecting where to allocate memory

- **Best Fit:** Try to reduce space wastage and fit as closely as possible
- **Worst Fit:** Find largest chunk with the goal of having big chunks left
- **First Fit:** Allocate in the first chunk that fits, very fast
- **Next Fit:** Continue searching for the first chunk that fits after previous allocation, fast and spreads allocations across the address space

How to keep track of the available chunks?

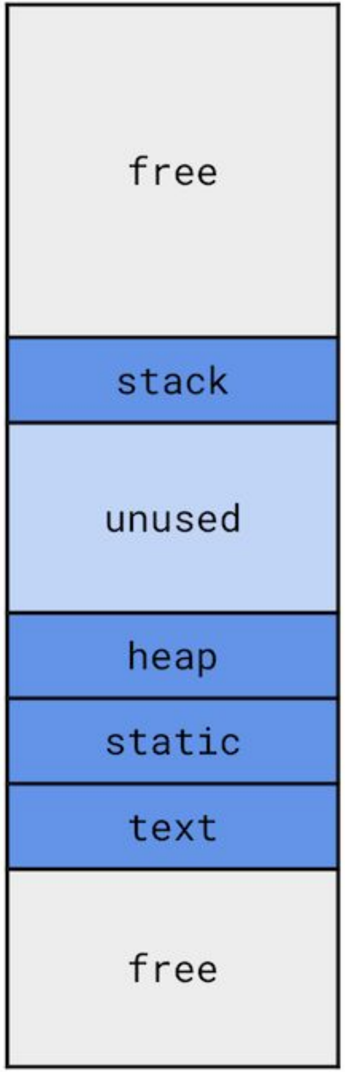
Memory Management Unit



Attempt 1: Contiguous Mapping

Problem: Internal Fragmentation

Huge unused region between heap and stack

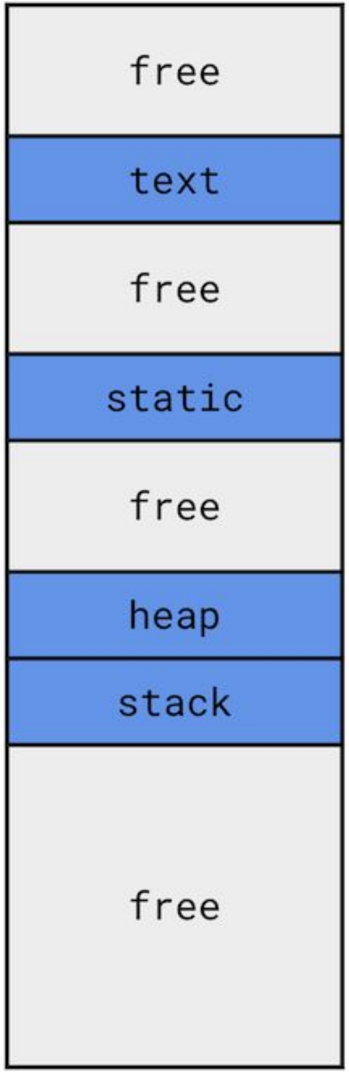


Attempt 2: Segmentation

Map each region (“segment”) to memory independently

Each segment has an associated base address and size.

Invalid access: **Segmentation Fault**



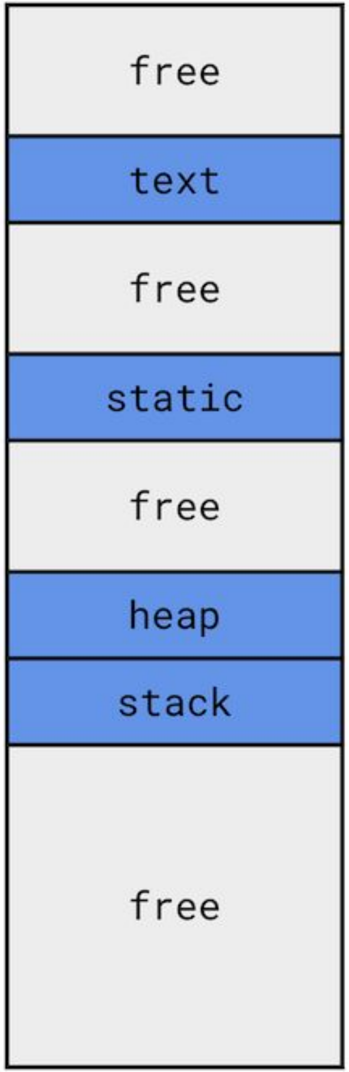
Segmentation Example

```
Sample 14-bit virtual address: 11000010010010
Assuming max segment size is 4KB (need 12 bits for offset).

        1    1        000010010010
|--segment selector--|---offset---|

        +----------+-------+-------+
        | segment  | base  | size  |
        +----------+-------+-------+
        |    00    |  6KB  |  2KB  |
        +----------+-------+-------+
        |    01    |  8KB  |  2KB  |
        +----------+-------+-------+
        |    10    |  12KB |  2KB  |
        +----------+-------+-------+
        |    11    |  16KB |  2KB  |
        +----------+-------+-------+

physical address: segment base + offset
```



Attempt 2: Segmentation

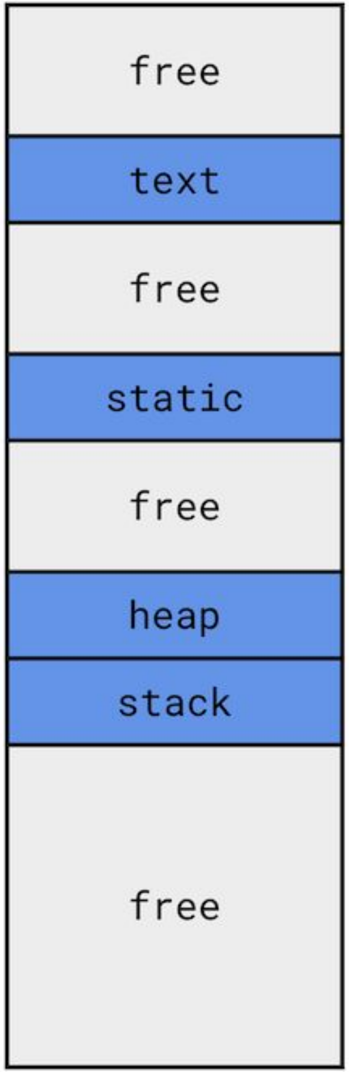
Map each region ("segment") to memory independently

Each segment has an associated base address and size.

Invalid access: **Segmentation Fault**

Problems:

- External fragmentation
- Impossible to do fine-grain sharing
- What if two segments collide in the physical address space?



Refined Goals

- Minimize internal fragmentation
- Minimize external fragmentation
- Enable fine-grain sharing

Attempt 3: Paging

Divide virtual and physical memory into fixed-sized pages

Still have selector bits and interpret virtual address as two parts:

- Virtual Page Number (VPN)
- Page Offset

Translate VPN into Physical Frame(Page) Number PFN using page table:

```
phys_addr = page_table[virt_addr / page_size] + virt_addr % page_size
```

Attempt 3: Paging

Divide virtual and physical memory into fixed-sized pages

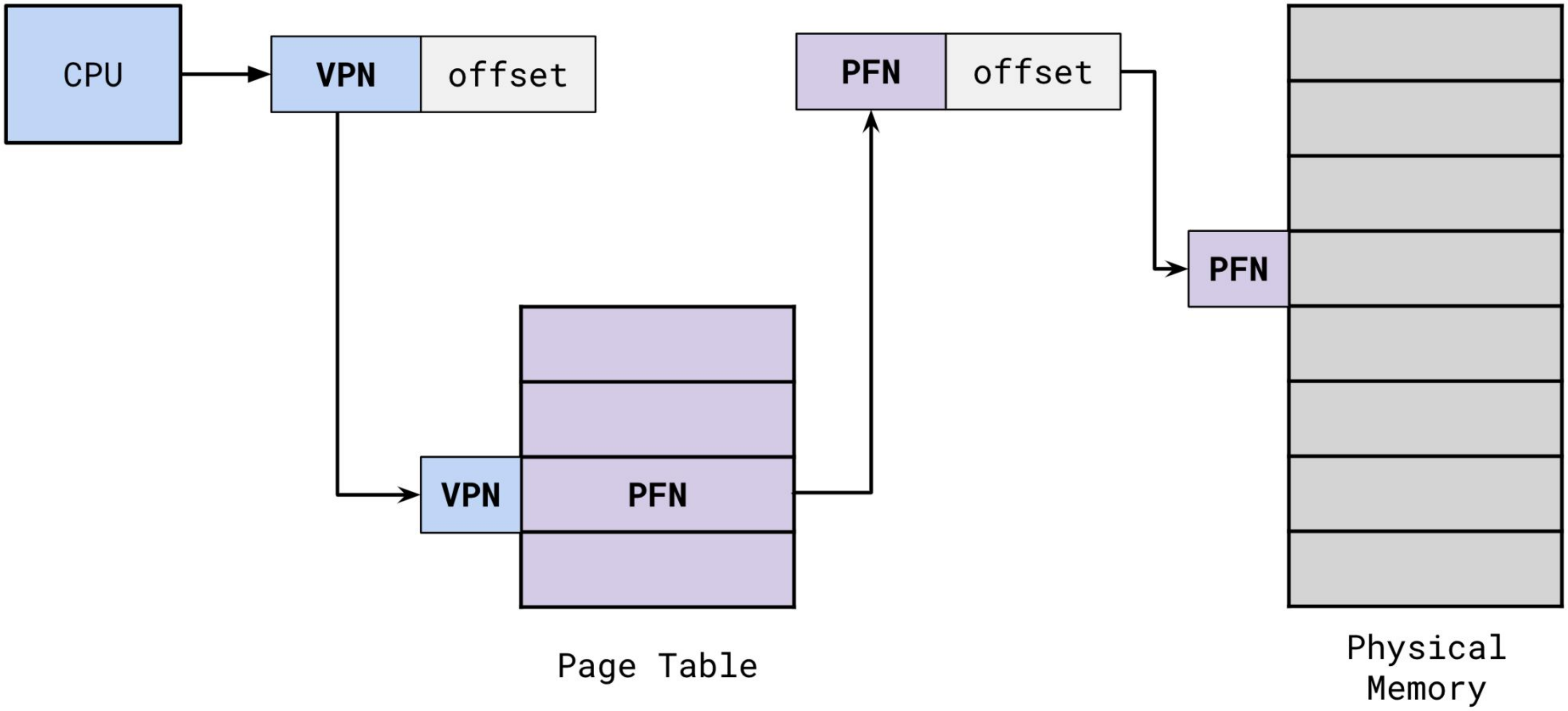
Still have selector bits and interpret virtual address as two parts:

- Virtual Page Number (VPN)
- Page Offset

Translate VPN into Physical Frame(Page) Number PFN using page table:

```
phys_addr = page_table[virt_addr / page_size] + virt_addr % page_size
```

Paging Bird's Eye View



Paging Bird's Eye View Example

Page 0
Page 1
Page 2
Page 3

Virtual
Memory

0	Frame 1
1	Frame 4
2	Frame 3
3	Frame 7

Page Table

0	
1	Page 0
2	
3	Page 2
4	Page 1
5	
6	
7	Page 3

Physical Memory

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- How many virtual pages per process?

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- How many virtual pages per process?
  - Can address 2^8 = 256 of virtual bytes
  - 256B / 64B = 4 virtual pages

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- How many virtual pages per process?
  - Can address 2^8 = 256 of virtual bytes
  - 256B / 64B = 4 virtual pages
- How many physical frames in RAM?

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- How many virtual pages per process?
  - Can address 2^8 = 256 of virtual bytes
  - 256B / 64B = 4 virtual pages
- How many physical frames in RAM?
  - Can address 2^10 = 1024 of physical bytes
  - 1024B / 64B = 16 physical frames

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

VPN	PFN
0	2
1	5
2	1
3	8

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

1. Divide virtual address by page size to get VPN: 241 / 64 == 3

```
VPN   PFN
     +---+
0    | 2 |
     +---+
1    | 5 |
     +---+
2    | 1 |
     +---+
3    | 8 |
     +---+
```

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

1. Divide virtual address by page size to get VPN: 241 / 64 == 3
2. VPN 3 translates to PFN 8.

```
VPN   PFN
     +----+
0    | 2 |
     +----+
1    | 5 |
     +----+
2    | 1 |
     +----+
3    | 8 |
     +----+
```

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

1. Divide virtual address by page size to get VPN: 241 / 64 == 3
2. VPN 3 translates to PFN 8.
3. Modulo virtual address by page size to get offset: 241 % 64 == 49

```
VPN  PFN
    +---+
0   | 2 |
    +---+
1   | 5 |
    +---+
2   | 1 |
    +---+
3   | 8 |
    +---+
```

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

1. Divide virtual address by page size to get VPN: 241 / 64 == 3
2. VPN 3 translates to PFN 8.
3. Modulo virtual address by page size to get offset: 241 % 64 == 49

PFN 8 == 0b1000

Offset: 49 == 0b110001

Physical address: (8 * 64) + 49 == 561 == 0b1000110001

```
VPN  PFN
     +---+
0    | 2 |
     +---+
1    | 5 |
     +---+
2    | 1 |
     +---+
3    | 8 |
     +---+
```

Paging Example

8-bit virtual address space, 10-bit physical address space, 64-byte pages

- Translate the virtual address 241 to a physical address:

1. Divide virtual address by page size to get VPN: 241 / 64 == 3
2. VPN 3 translates to PFN 8.
3. Modulo virtual address by page size to get offset: 241 % 64 == 49

PFN 8 == 0b1000

Offset: 49 == 0b110001

Physical address: (8 * 64) + 49 == 561 == 0b1000110001

What if 241 was given in binary 0b11110001?

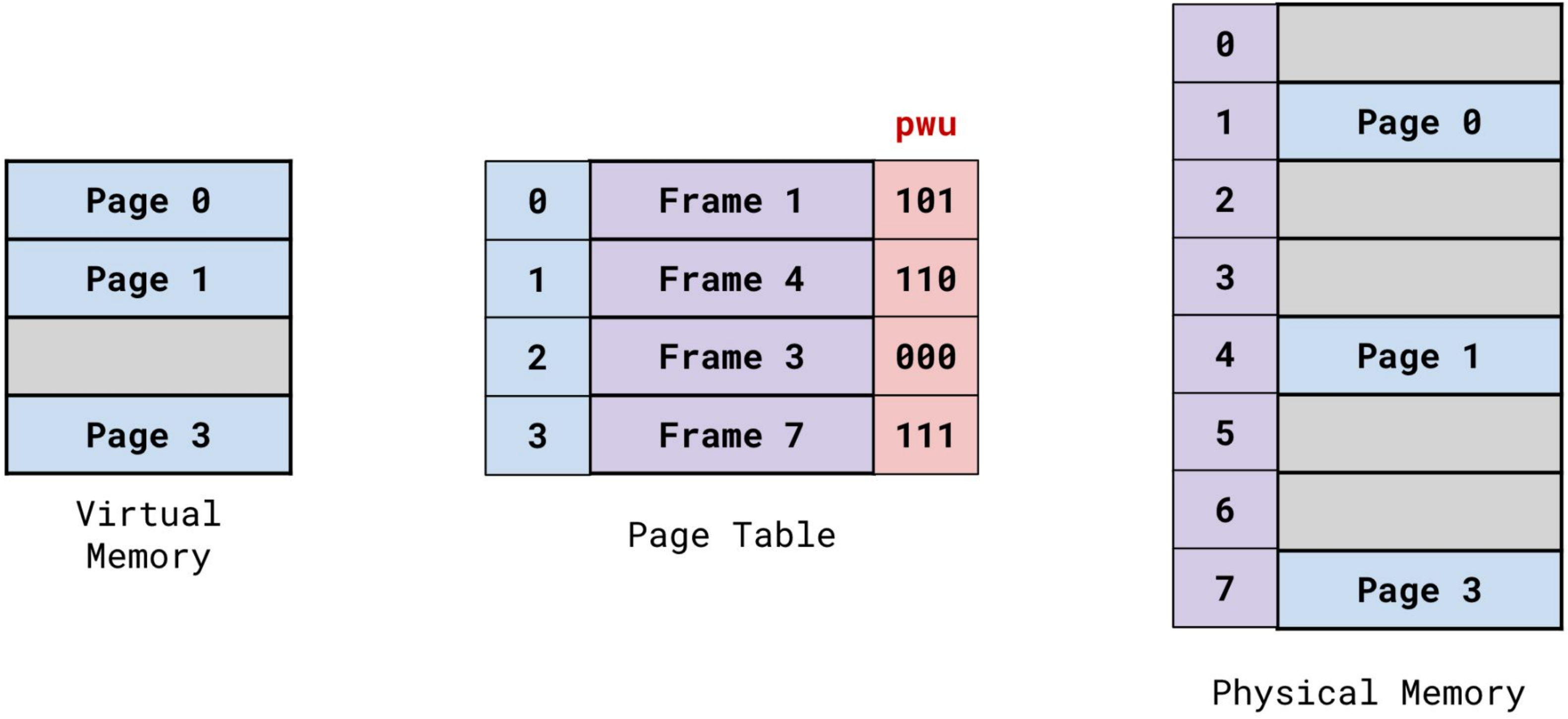
VPN	PFN
0	2
1	5
2	1
3	8

Page Protection

Each page table entry also carries some metadata bits, e.g.:

- **present (p):** whether or not this mapping is active. This virtual page is not mapped to physical memory
- **writable (w):** whether or not this page can be written to. Some architectures have readable/executable bits too
- **user (u):** can this page be accessed by userspace, i.e. to protect kernel pages from user programs

Page Protection Example



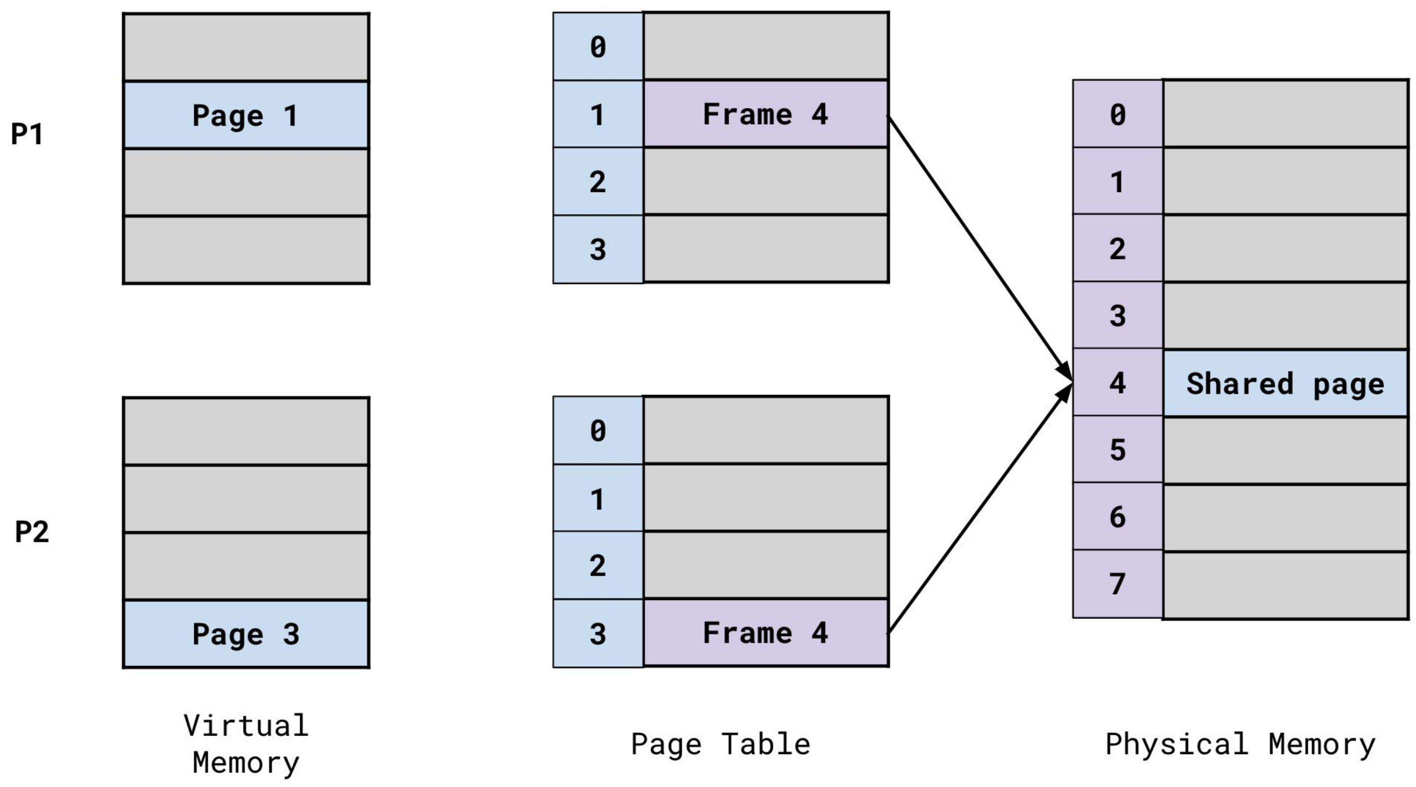
High-level Hardware Implementation

- Hardware has a dedicated Page Table Base Register (PTBR) that points to the base of the page table
  - e.g. cr3 register in x86

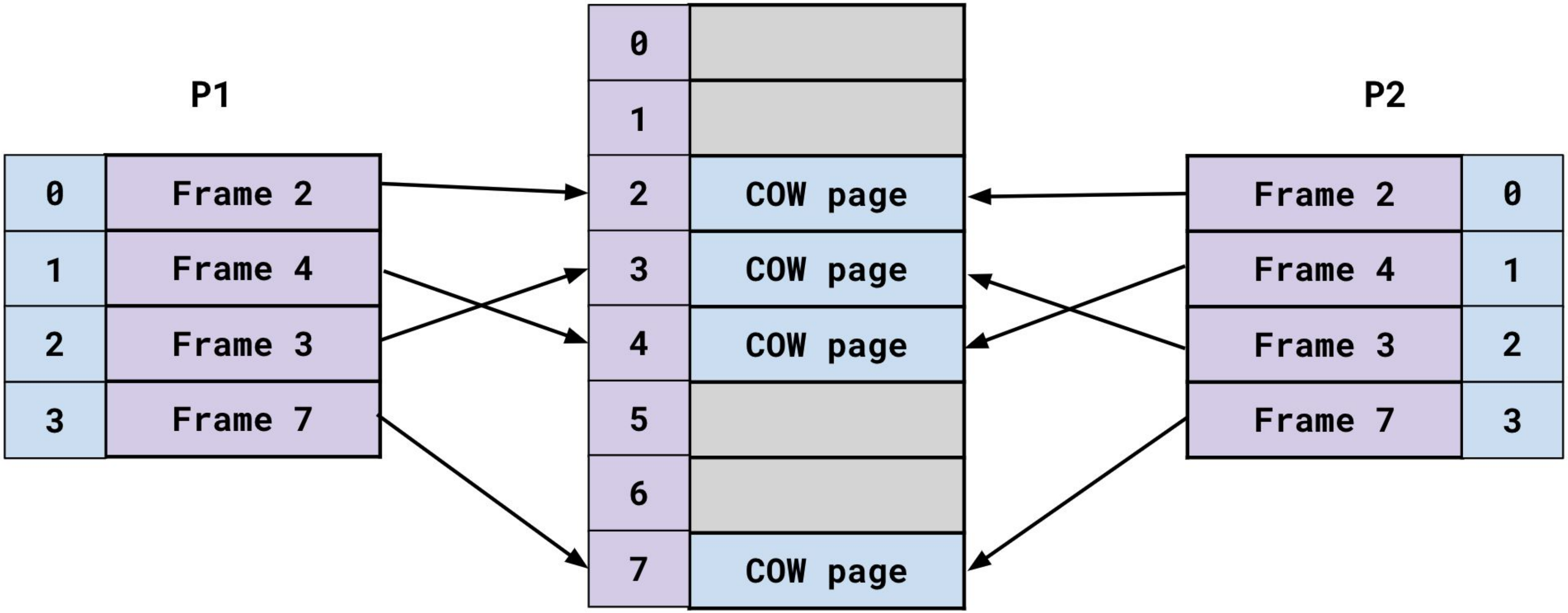
- OS also needs to manage the page table – stores the base address in the process control block (PCB)
  - e.g. task_struct in Linux

- PTBR is updated with new page table base address on context switch

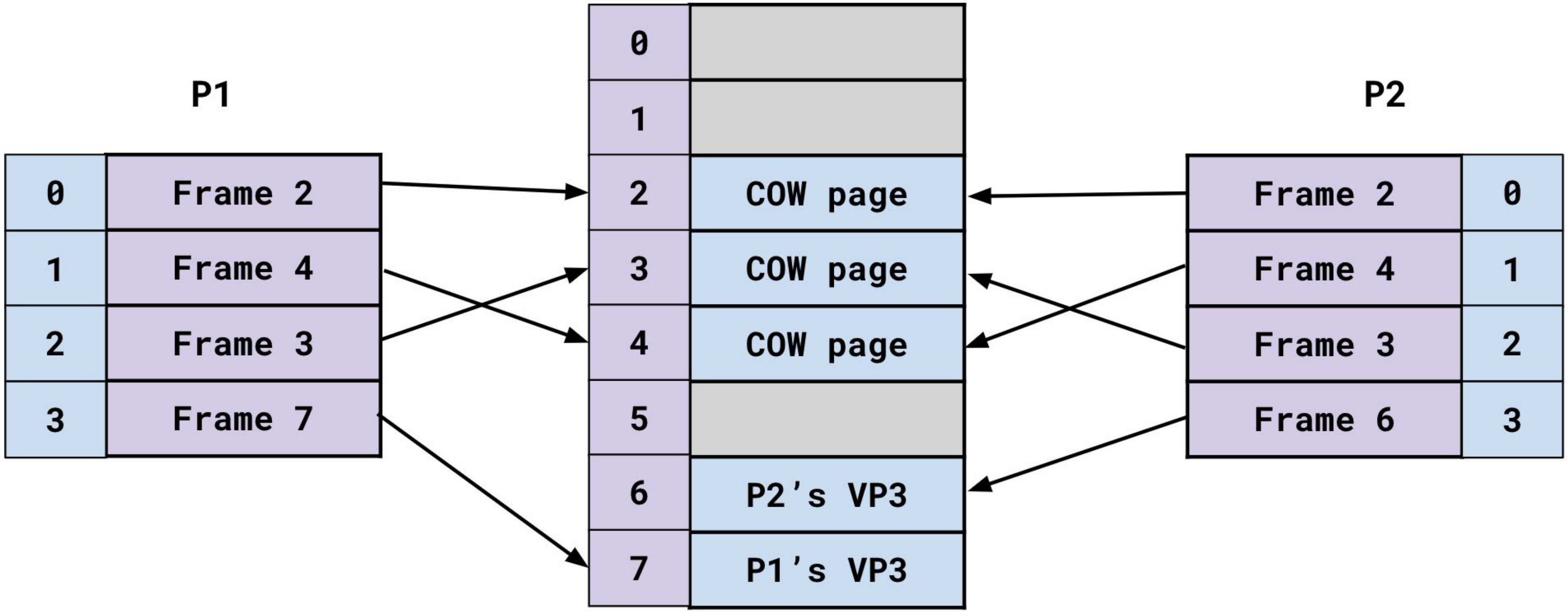
Page Sharing



Copy-on-Write (COW)



Copy-on-Write (COW)



Issues with simple single-level page table

Efficiency: Data access now seems to require two memory accesses, i.e., one extra access for page table

Memory Usage: Page table consumes unreasonable amount of space!

Consider 32-bit virtual address space (4GB), 4KB page size, page table entry size of 4B.

- num virtual pages: 2 ^ 32 / 2 ^ 12 == 2 ^ 20 == 1M
- Need page table entry per virtual page: 1M pages * 4B entry == 4MB per process?!