Memory Management and Paging

W4118 Operating Systems I

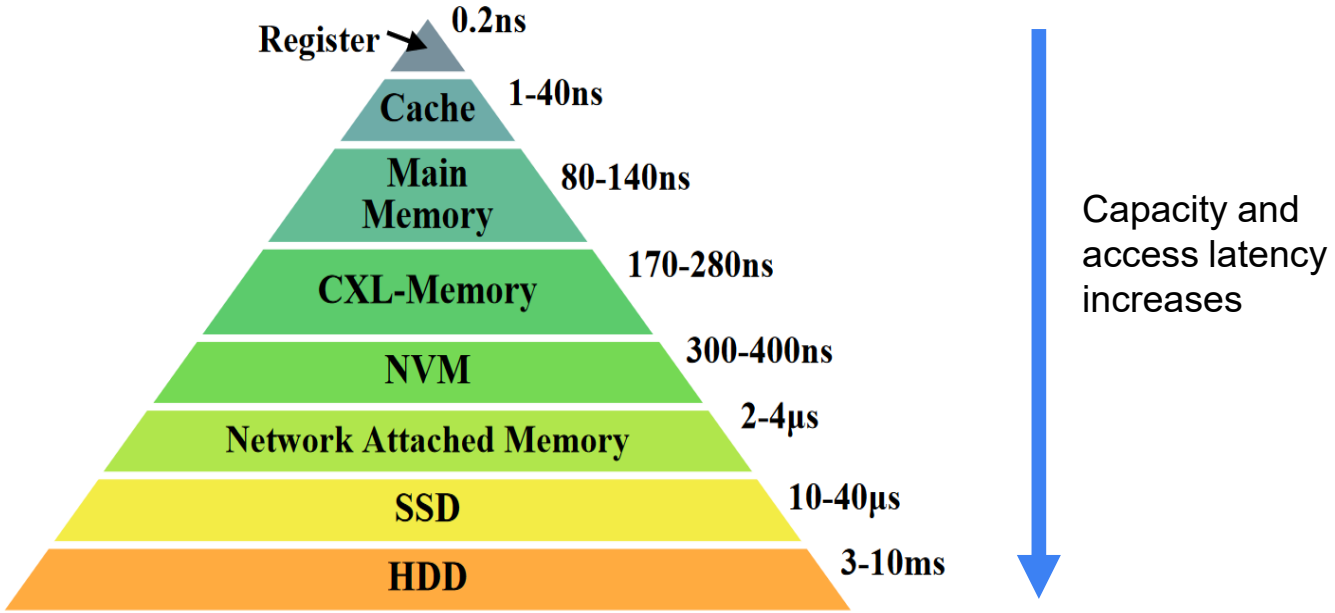
columbia-os.github.io

Credits to Jae

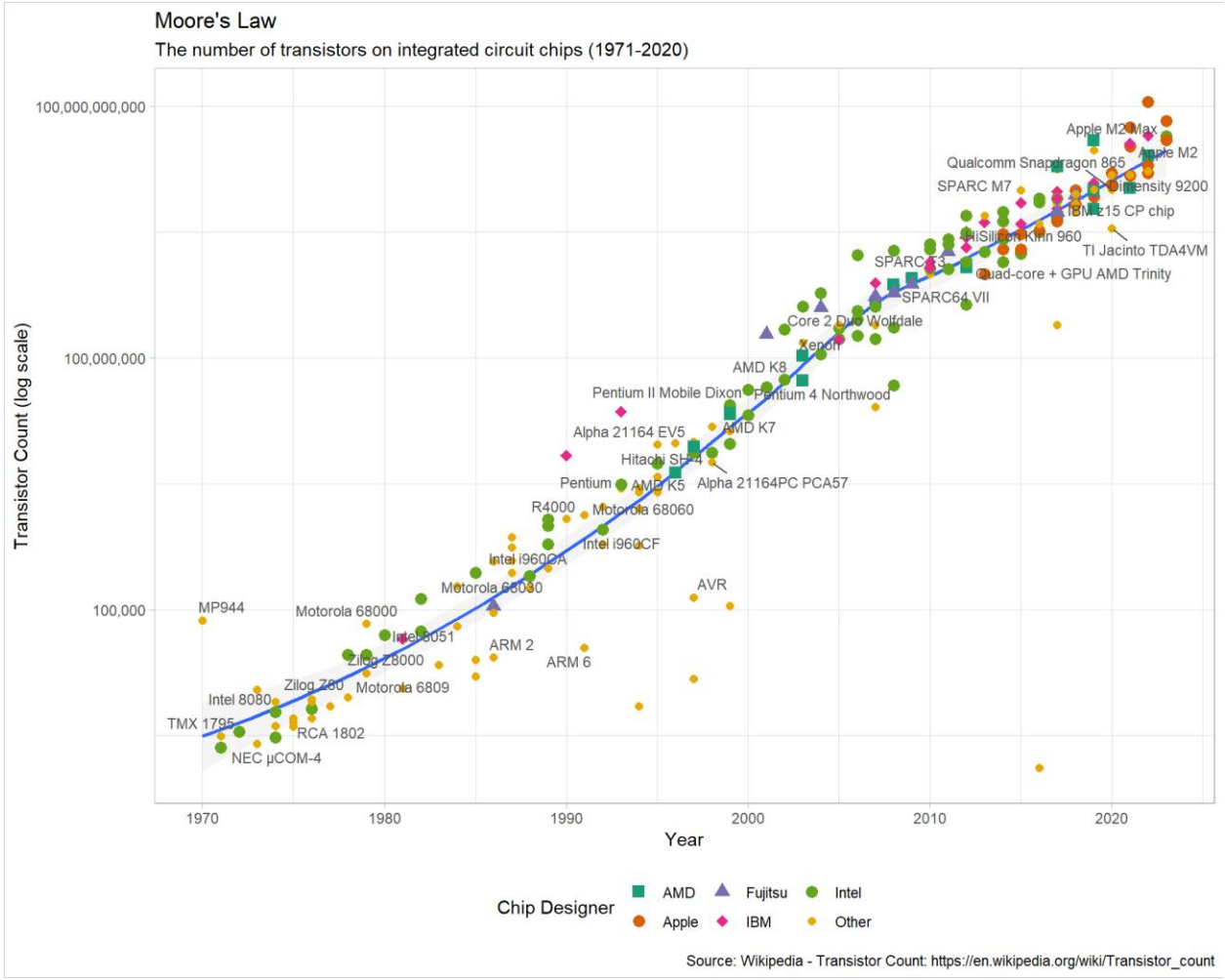
Programmer’s dream

- Private
- Infinitely large
- Infinitely fast
- Non-volatile
- Inexpensive

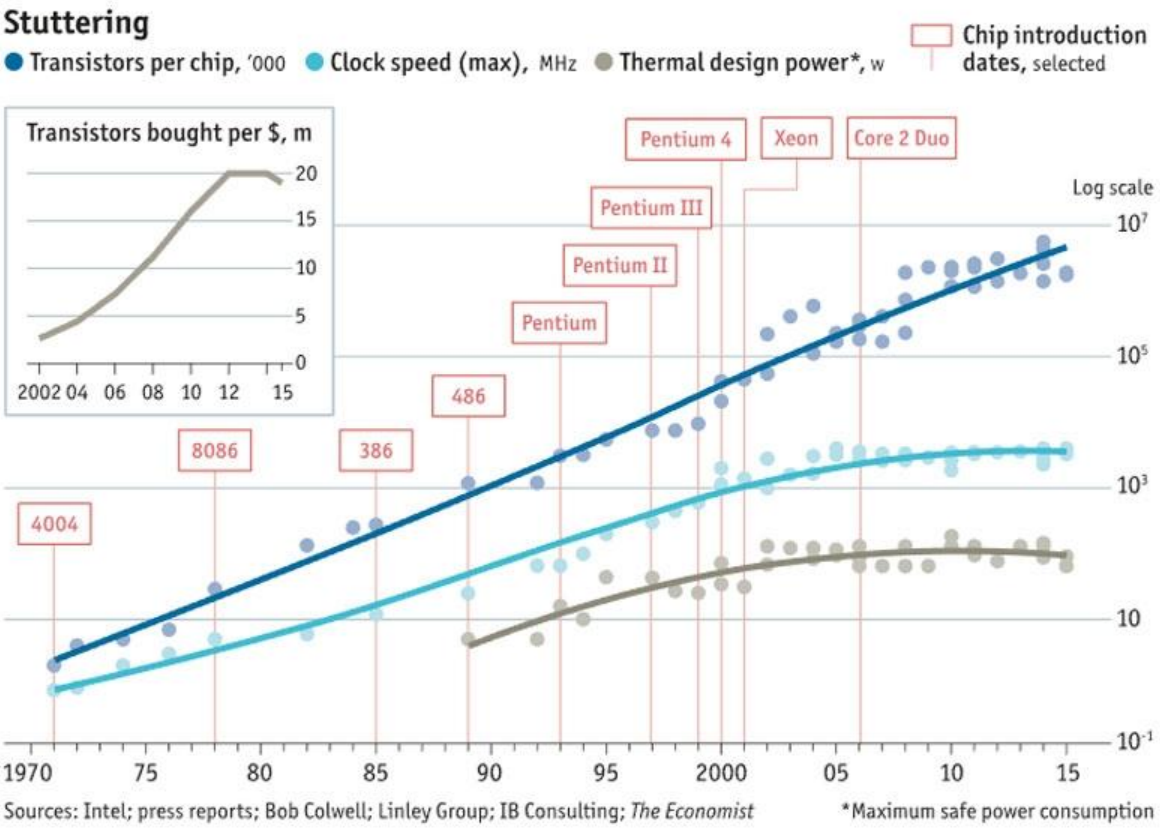
Memory Hierarchy



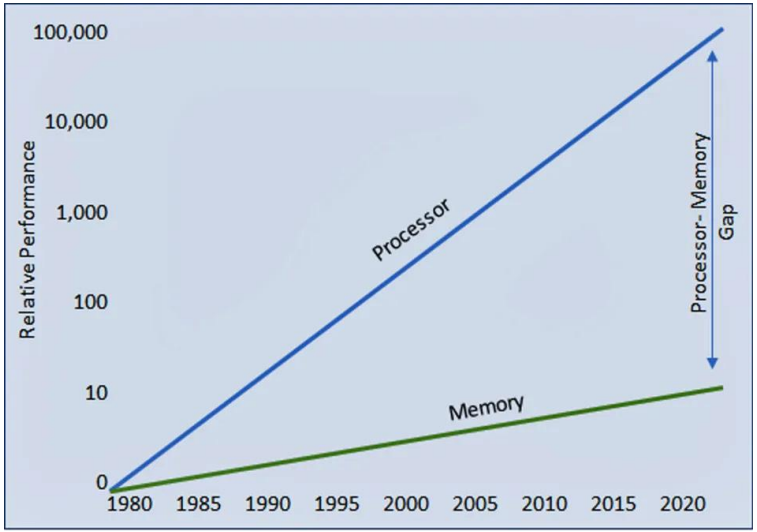
Moore’s Law


Moore's Law
The number of transistors on integrated circuit chips (1971-2020)
Transistor Count (log scale)
100,000,000,000
100,000,000
100,000
1970
1980
1990
2000
2010
2020
Year
Apple M2 Max
Apple M2
Qualcomm Snapdragon 865
SPARC M7
Dimensity 9200
IBM z15 CP chip
HiSilicon Kirin 960
TI Jacinto TDA4VM
SPARC T3
Quad-core + GPU AMD Trinity
SPARC64 VII
Core 2 Duo Wolfdale
Xenon
AMD K8
Pentium II Mobile Dixon
Pentium 4 Northwood
Alpha 21164 EV5
AMD K7
Hitachi SH-4
Pentium
AMD K5
Alpha 21164PC PCA57
R4000
Motorola 68060
Intel i960CF
Intel i960CA
Motorola 68030
AVR
MP944
Motorola 68000
Intel 8051
ARM 2
ARM 6
Zilog Z8000
Zilog Z80
Motorola 6809
Intel 8080
TMX 1795
RCA 1802
NEC μCOM-4
Chip Designer
AMD
Fujitsu
Intel
Apple
IBM
Other
Source: Wikipedia - Transistor Count: https://en.wikipedia.org/wiki/Transistor_count

Chip Evolution



Why care about Memory Hierarchy? (1)



µProc
60%/yr

CPU-DRAM
Gap widening

DRAM
7%/yr.

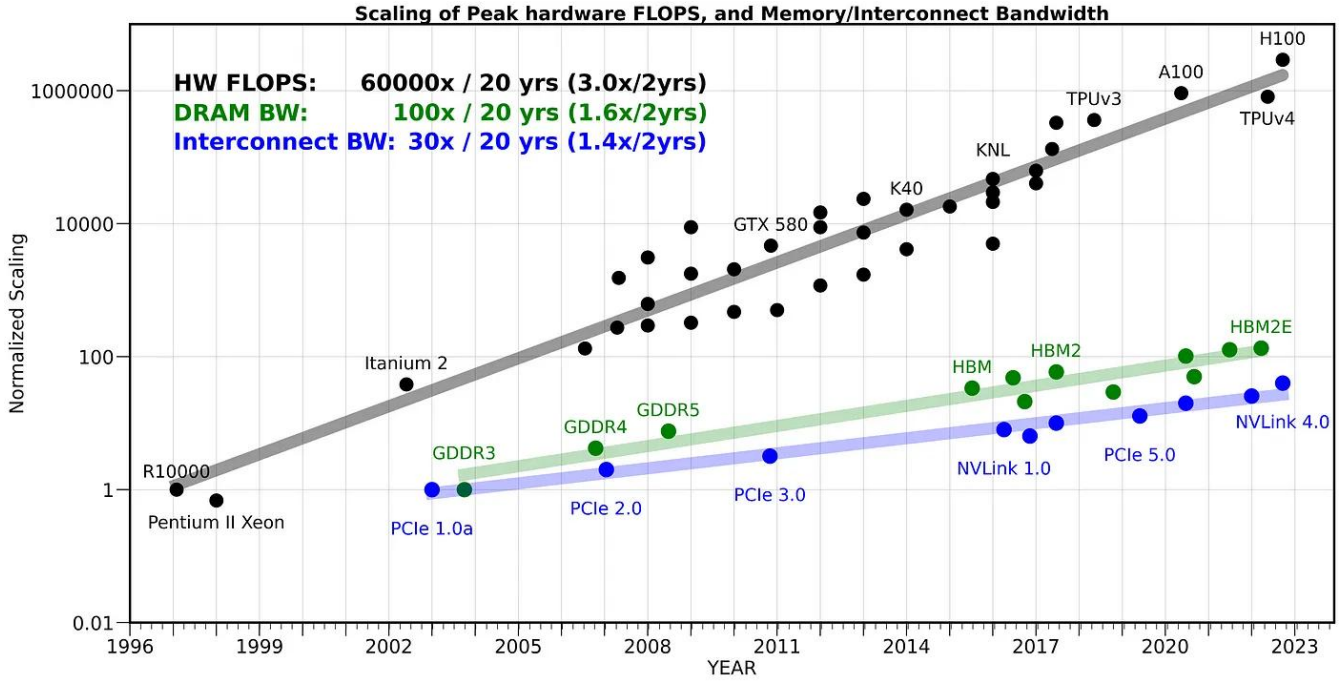
Implications:

- Longer per cycle memory access times → Memory seems “further away”
- New Technologies: SDR -> DDR-1/2/3/4/5 -> QDR (technologies)

  Still problem widens

Why care about Memory Hierarchy? (2)

- Despite new technologies in memory the discrepancy continues



Basic Expectations

- Concept of address spaces and virtual and physical addresses was discussed before
- Multiple applications must be concurrently in memory for rapid switching
- Originally required to allow overlap of computation and I/O
- Now also because we have O(10) cores in a system with high degree of parallelism

1960s+ Address Space Mgmt: Base and Limit (1)

- Map each process address space onto a different part of physical memory
- Two registers: Base and Limit
  - **Base**: start address of a program in physical memory
  - **Limit**: length of the program
- For every memory access
  - Base is added to the address
  - Result compared to Limit
- Only OS can modify Base and Limit

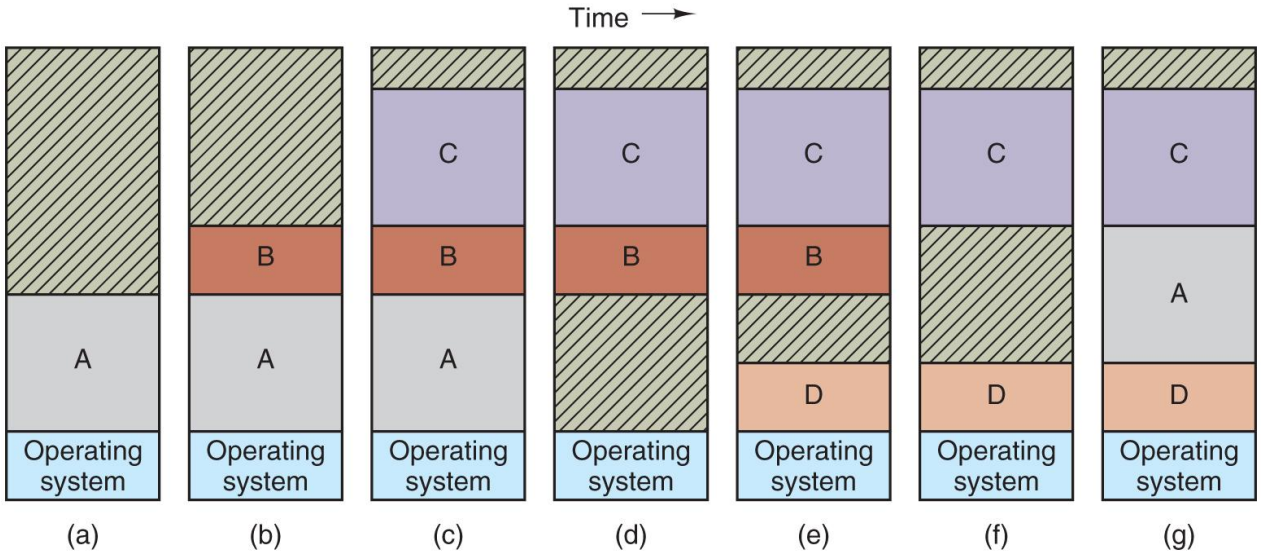
1960s+ Address Space Mgmt: Base and Limit (2)

- Need to add and compare for each memory address:
  - can be done in HW
  - doesn’t add significantly add to latency

- What if memory space is not enough for all programs?
  - We may need to swap some programs out of the memory.
  - remember swapping means moving entire program to disk ➔ expensive

Address Space Swapping (1)

- Sequence of program placements
- When out of memory programs need to be swapped



Address Space Swapping (2)

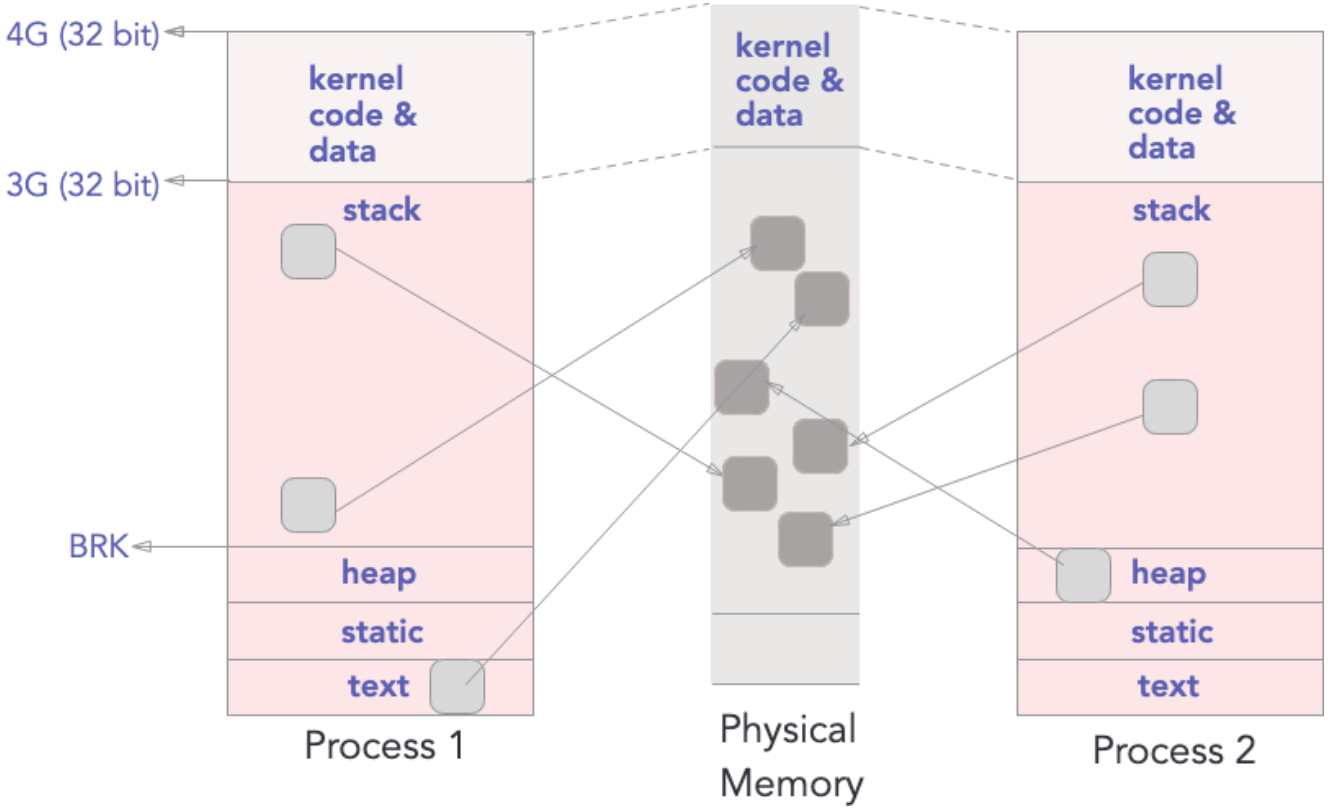
- Programs move in and out of memory
- Holes are created
- Holes can be combined -> memory compaction
- What if a process needs more memory?
  - If a hole is adjacent to the process, it is allocated to it
  - Process has to be moved to a bigger hole
  - Process suspended till enough memory is there

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

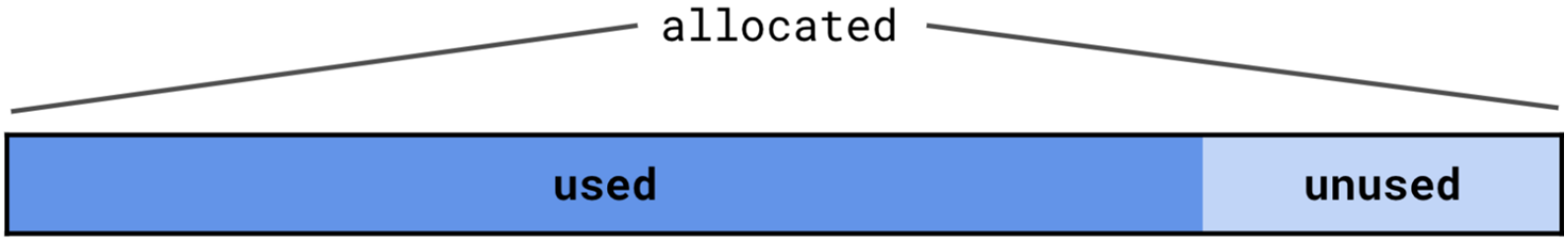
Reminder: Virtual Address Space

Since the 80s memory mgmt has different approaches and different demands



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

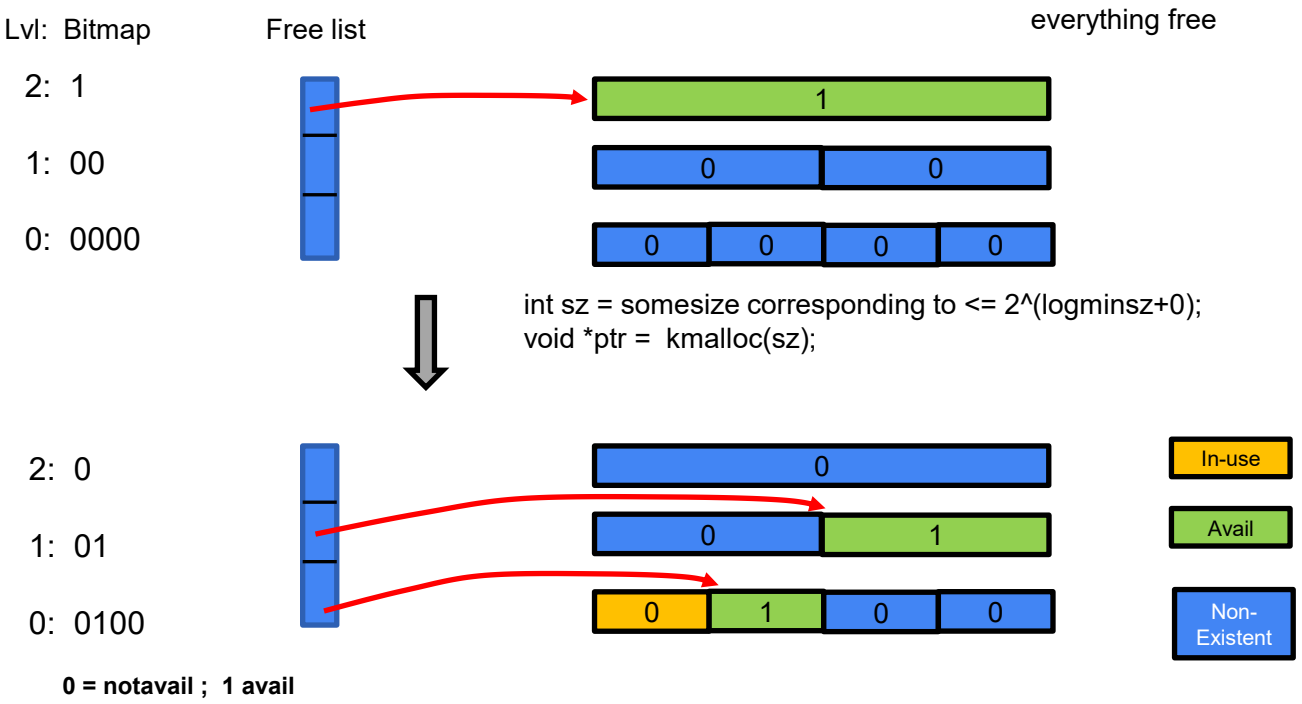
How to keep track of the available chunks?

Buddy Allocator (1)

- Considers blocks of memory only as 2^N
- You can have many small blocks or few large blocks or combination of those
- Potential for fragmentation (drawback)
- If no block of a size is available, it splits higher blocks into smaller blocks
- Easy to implement and fast O(log2(MaxBlockSz/MinBlockSize))
  e.g. 4K .. 128B = 2^(12-7) = 5 steps
- This works for managing physical memory or memory inside and address-space

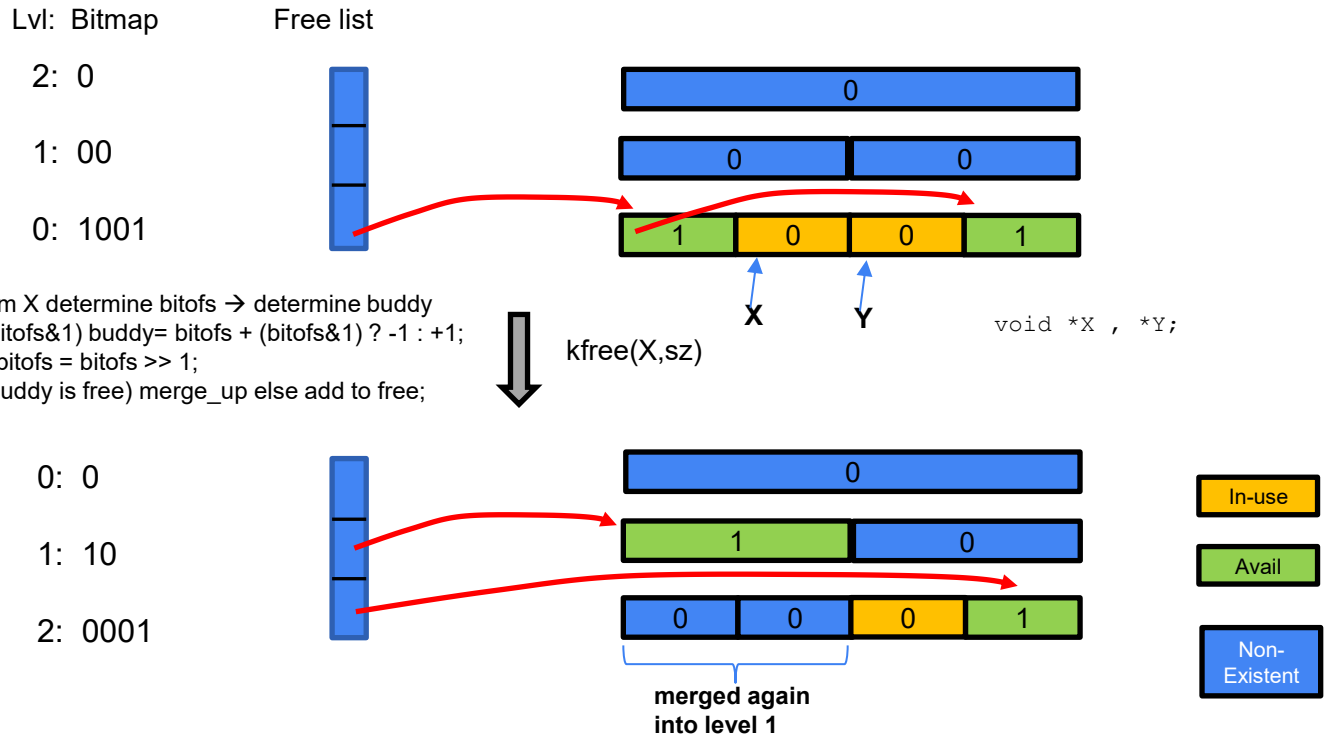
Buddy Allocator (2)

- Allocation at level 0

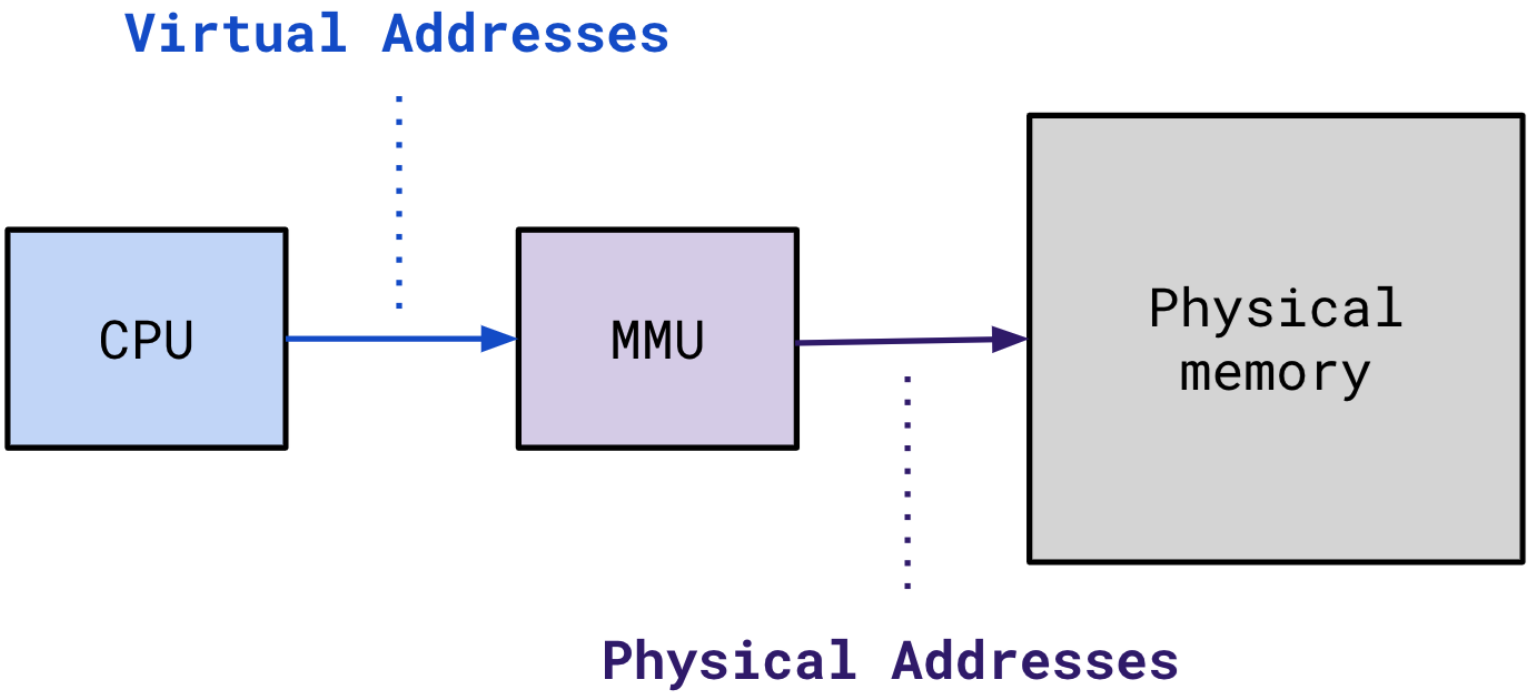


Buddy Allocator (3)

- Freeing “X” at level 0 leading to coalescing



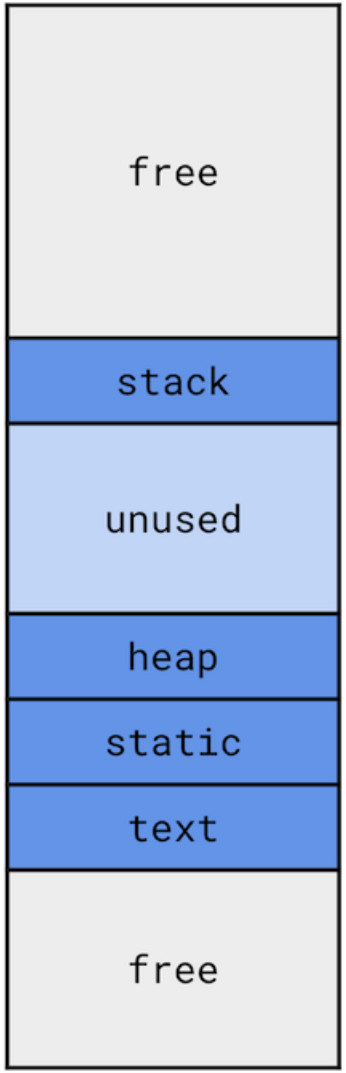
Memory Management Unit



Attempt 1: Contiguous Mapping

Problem: **Internal Fragmentation**

Huge unused region between heap and stack

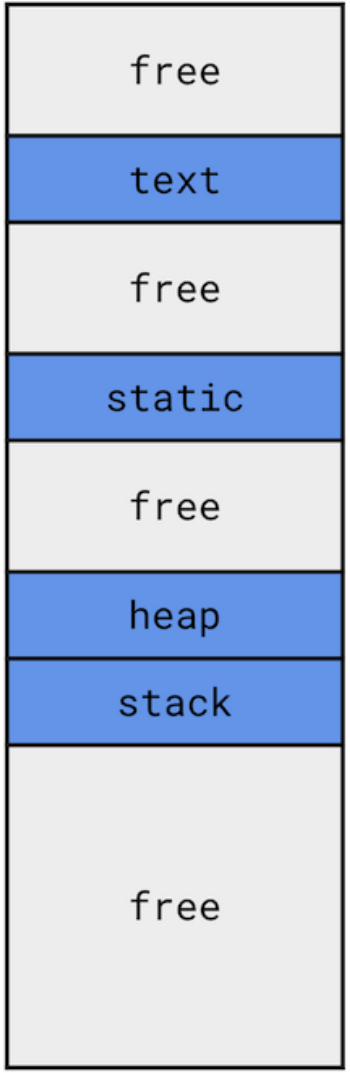


Attempt 2: Segmentation

Map each region (“segment”) to memory independently

Each segment has an associated base address and size.

Invalid access: **Segmentation Fault**



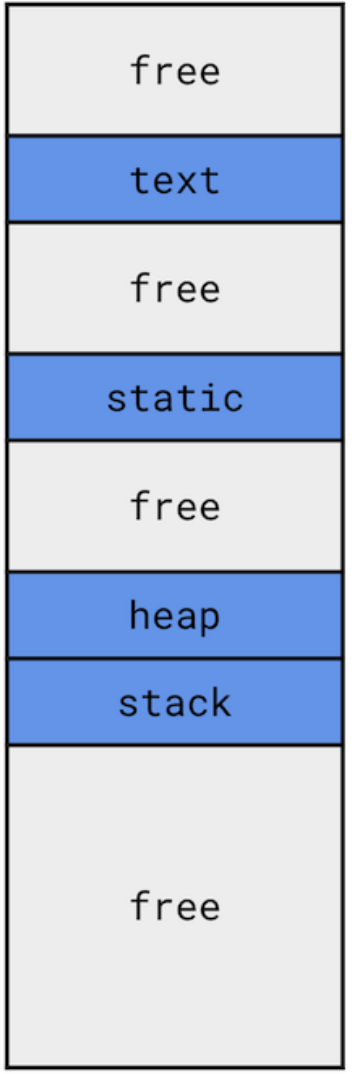
Segmentation Example

```
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

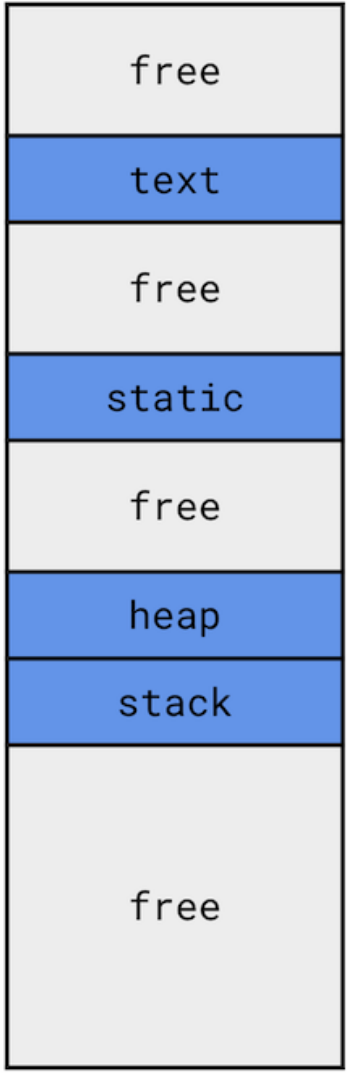
Map each region (“segment”) to memory independently

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

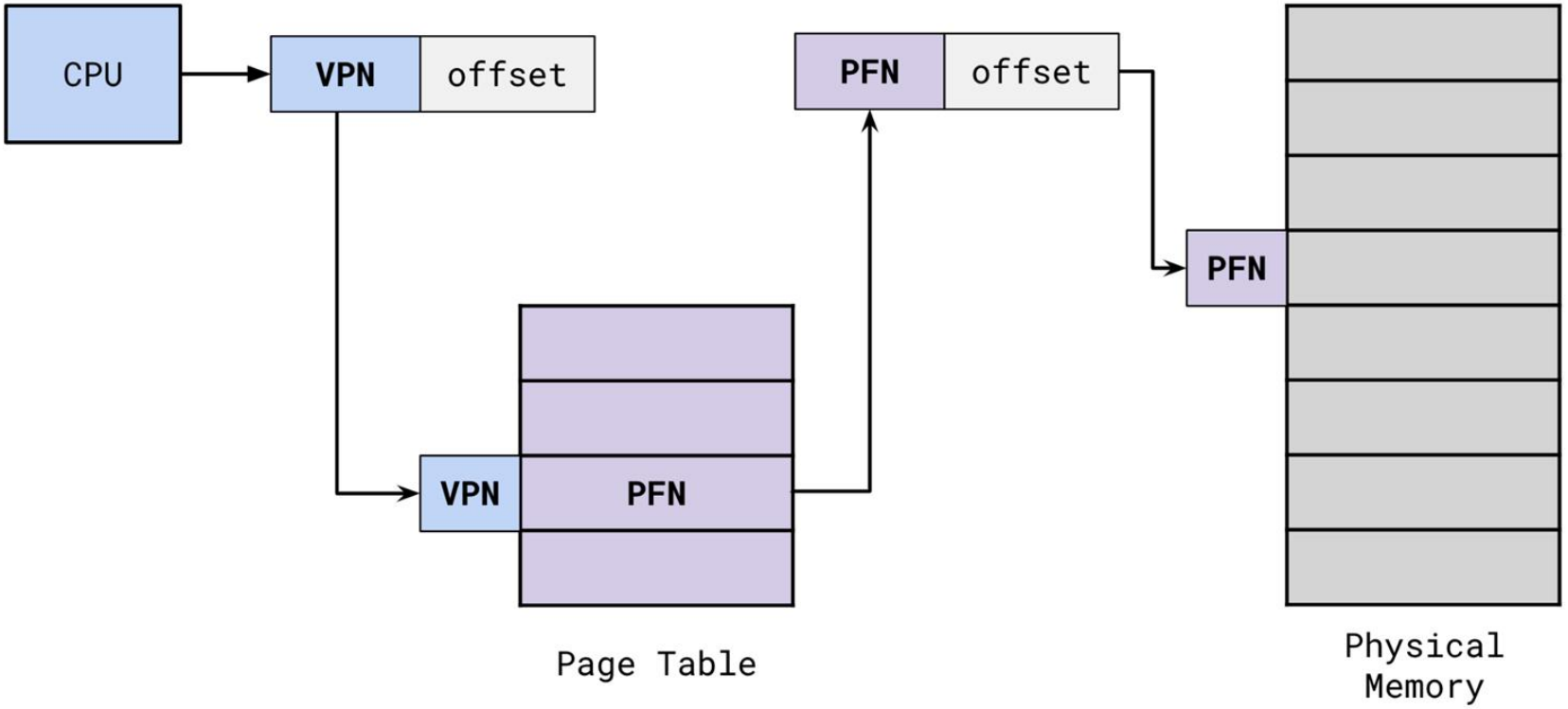
Still have selector bits and interpret virtual address as two parts:

- Virtual Page Number (VPN)
- Page Offset

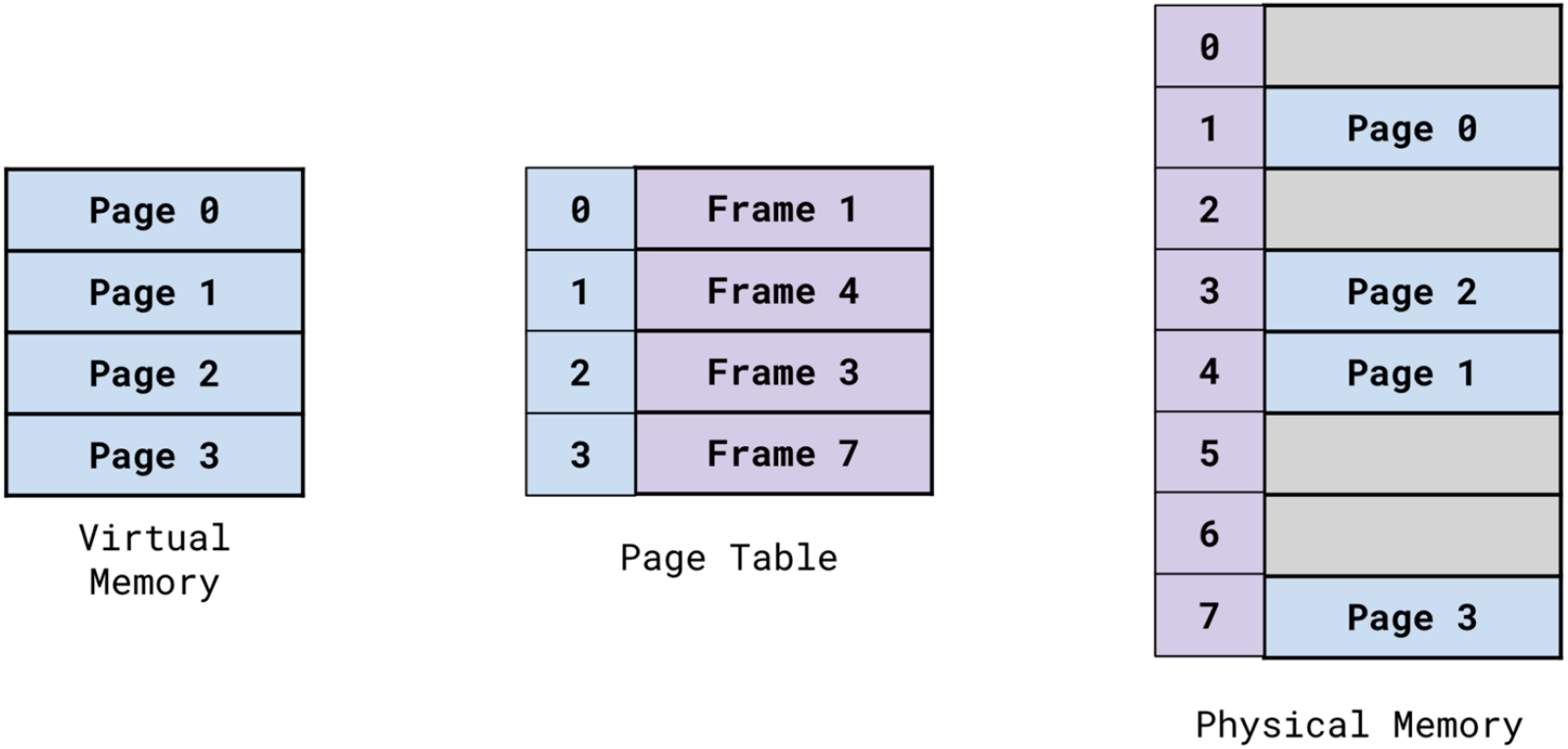
Translate VPN into Physical Frame(Page) Number PFN using page table:

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

Paging Bird's Eye View



Paging Bird’s Eye View Example



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

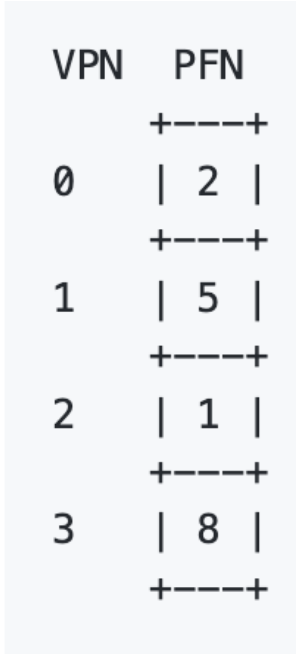
VPN	PFN
0	2
1	5
2	1
3	8

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.

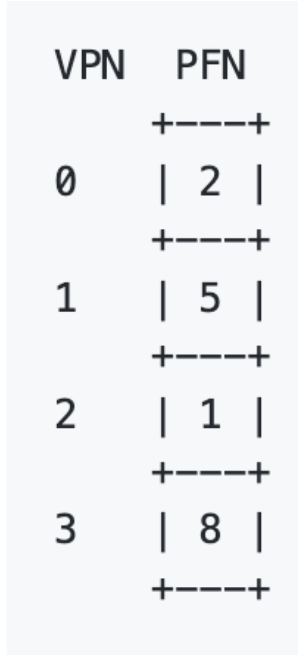


Paging Example

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



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?

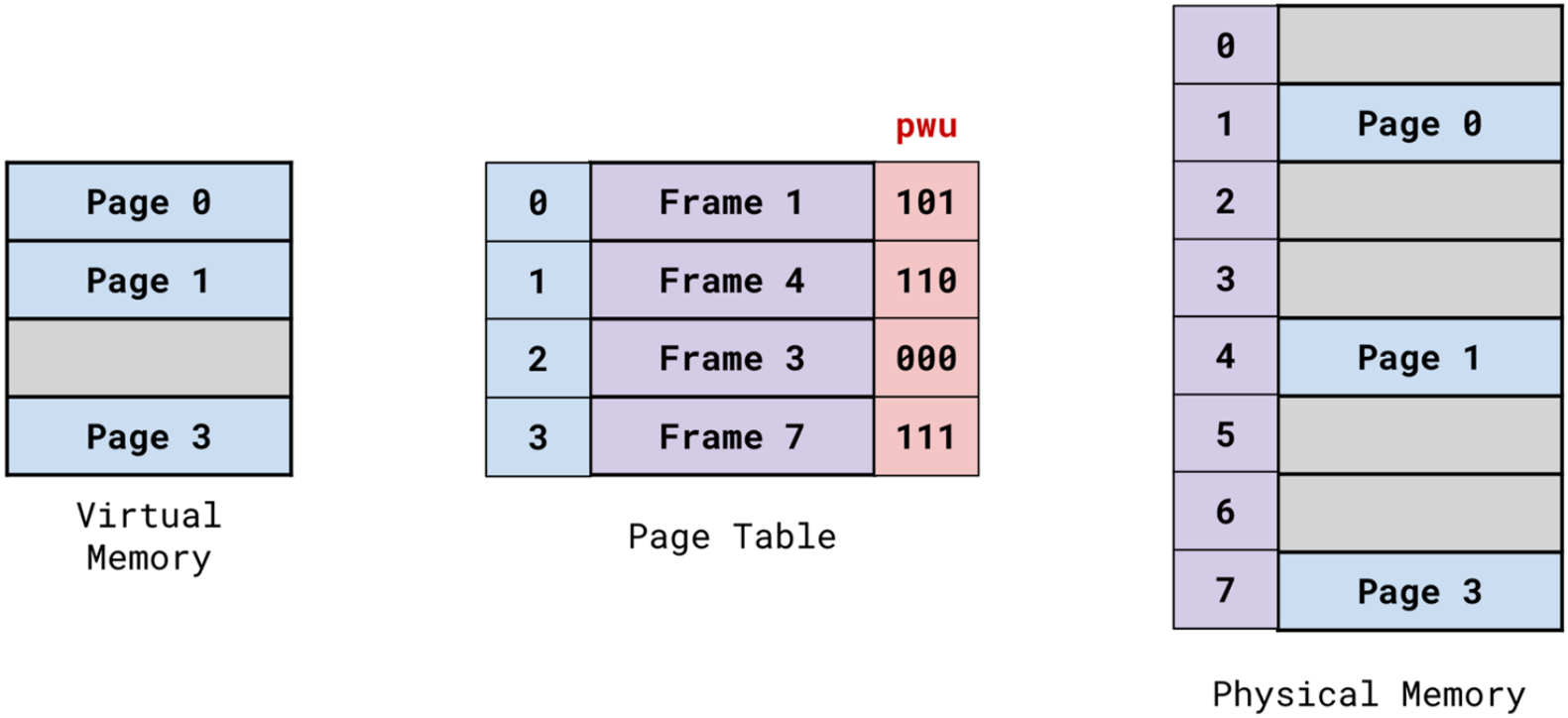
VPN	PFN
0	2
1	5
2	1
3	8

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



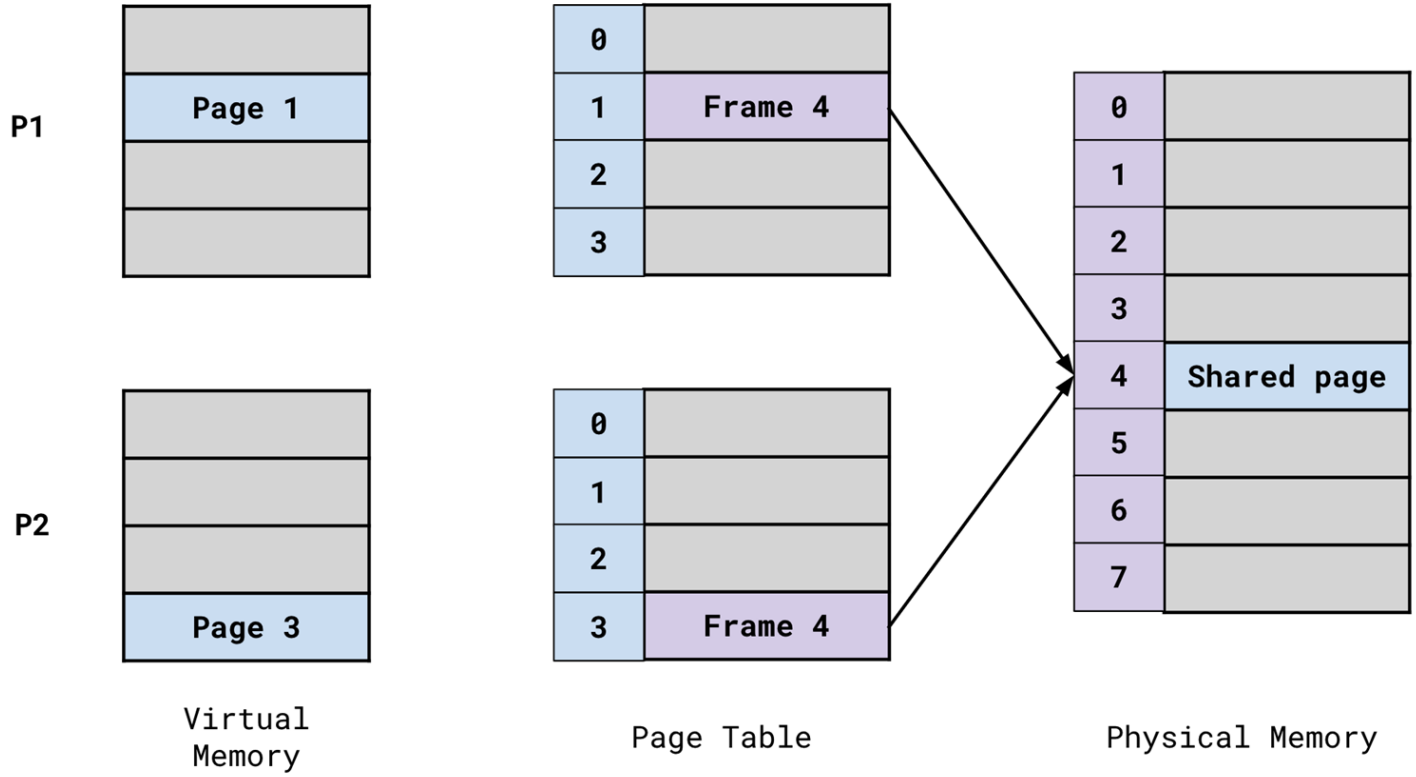
High-level Hardware Implementation

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

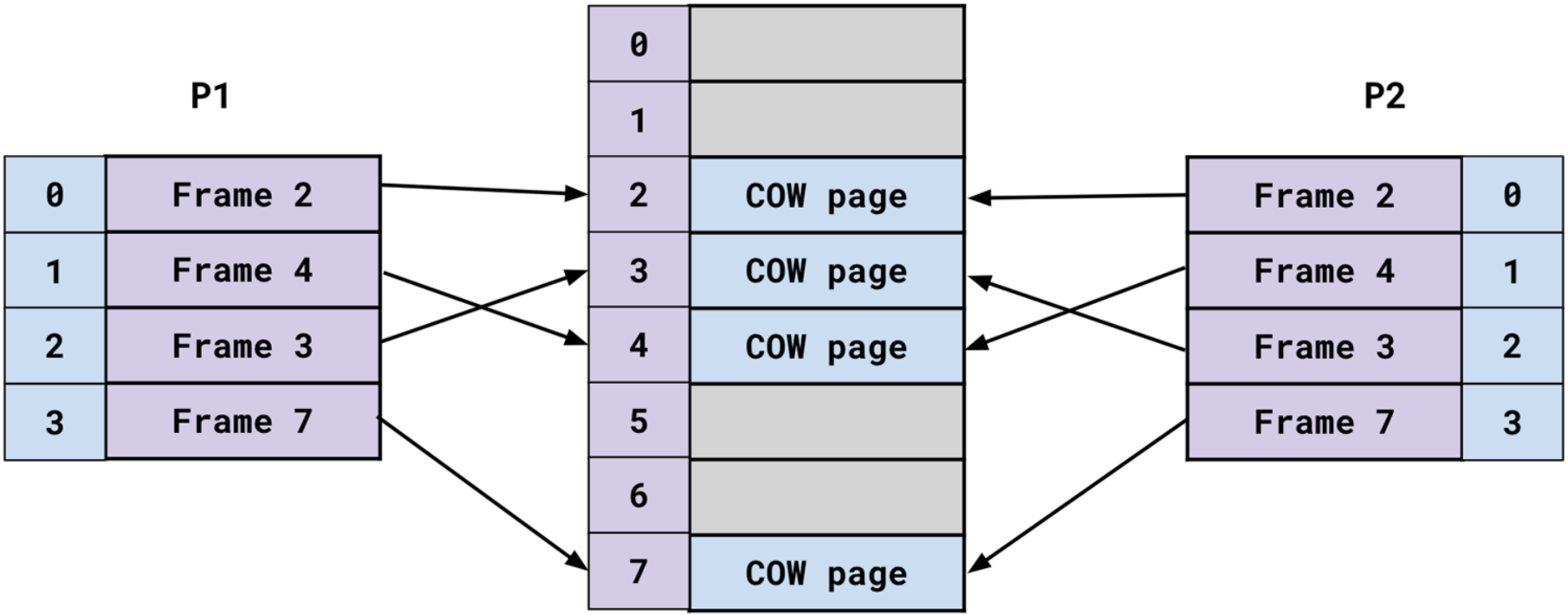
- OS also needs to manage the page table – stores the base address in the process control block (PCB)
  - e.g. task_struct in Linux

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

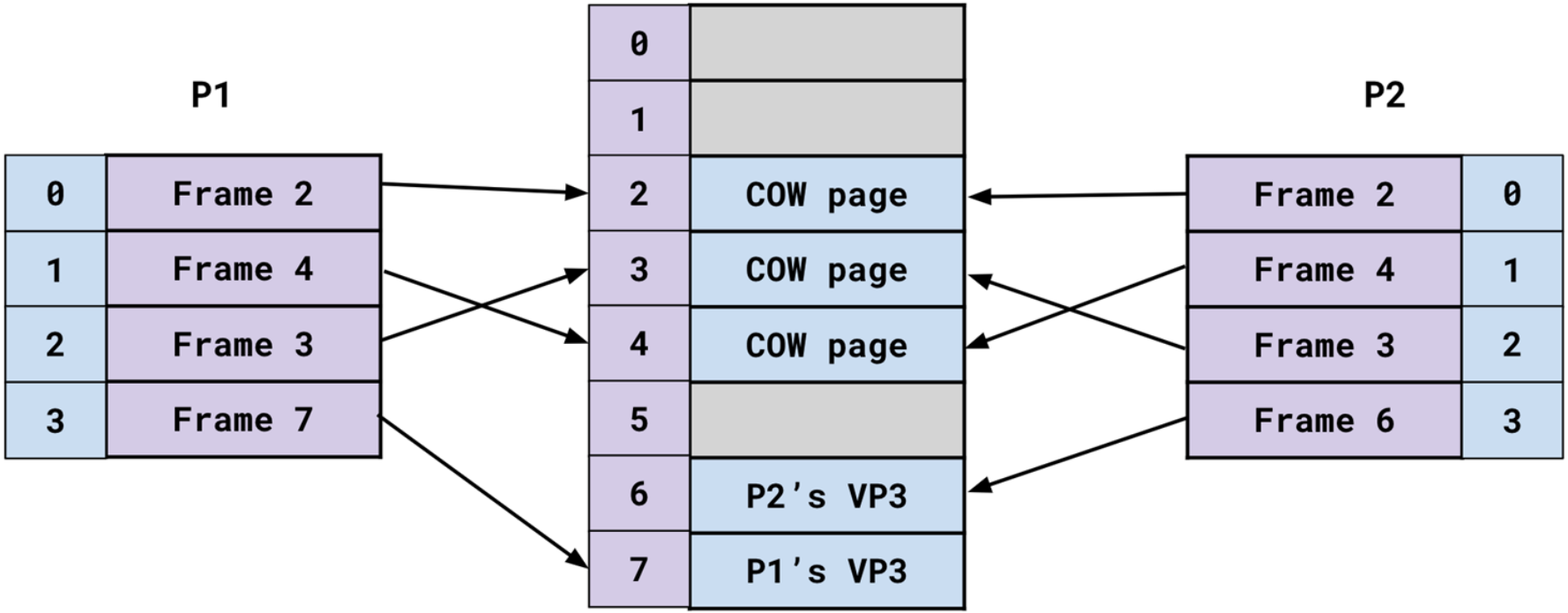
Page Sharing



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?!

x86 Memory Management

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

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!

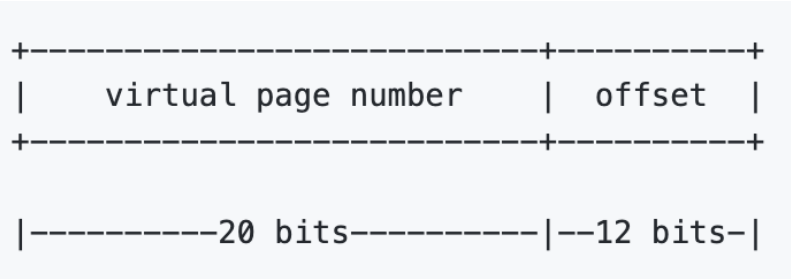
Consider 32-bit virtual address space (4GB), 4KB page size, page table entry size of 4B.

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

Page Table so Far

32-bit virtual addresses → 4GB virtual memory

4KB pages → 4GB / 4KB = 2^20 = 1M pages

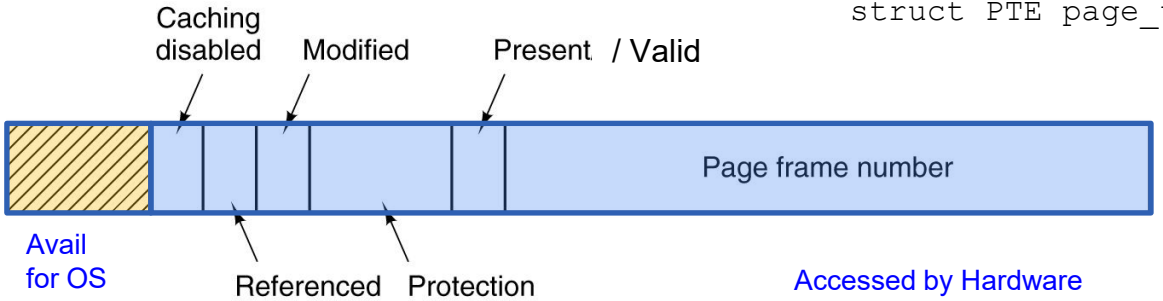


Page table entry = 4B = 20 bits for PFN (assuming 4GB physical memory)

+ 12 bits for metadata

Structure of Page Table and a Page Table Entries

```
struct PTE page_table[];
```

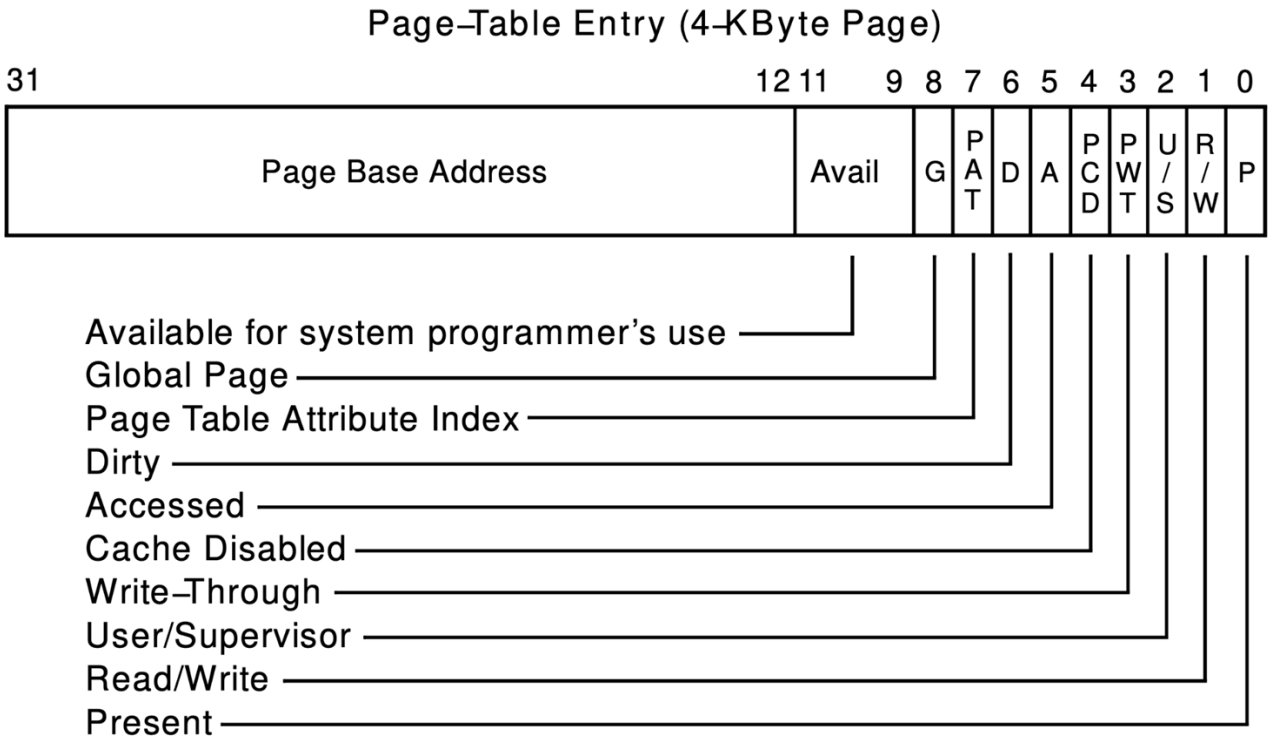


- **Present/Valid bit**: '1' if the values in this entry is valid, otherwise translation is invalid and a pagefault exception will be raised
- **Frame Number:** this is the physical frame that is accessed based on the translation.

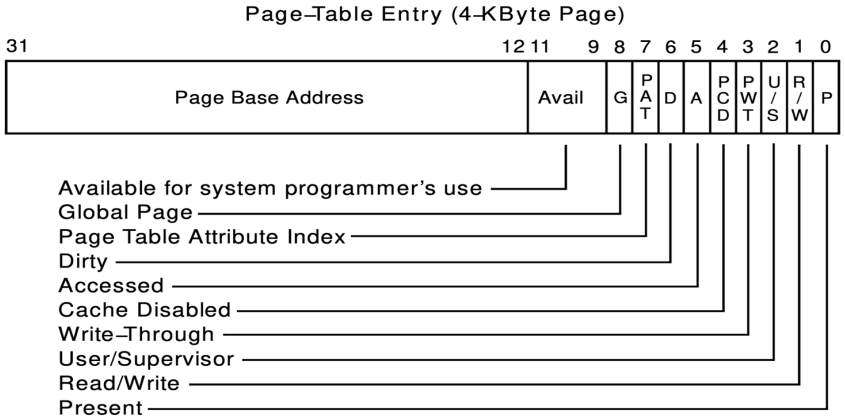
- **Protection bits**: 'kernel' + 'w' specifies who and what can be done on this page
  if *kernel bit* is set then only the kernel can translate this page. If user accesses the page a 'privilege exception' will be raised.
  If *writeprotect bit* is set the page can only be read (load instruction). If attempted write (store instruction), a write protection exception is raised.
- **Reference bit:** every time the page is accessed (load or store), the reference bit is set.
- **Modified bit:** every time the page is written to (store), the modified bit is set.
- **Caching Disabled:** required to access I/O devices, otherwise their content is in cpu cache

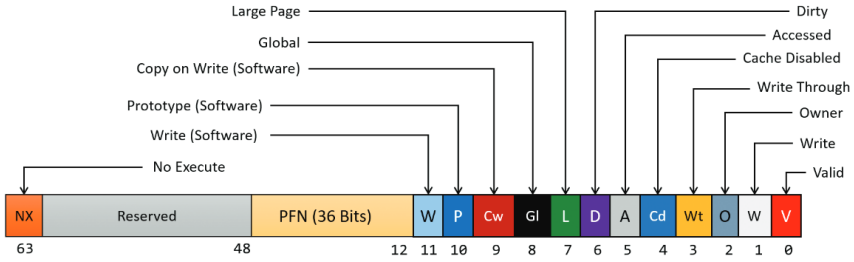
- Other unused bits typically available for the operating system to "remember" information in

Page Table Entry Structure



X86/x64 examples for PTE





64-bit (hardware specification)

Mapping hardware specification into C bit-structure (Linux)

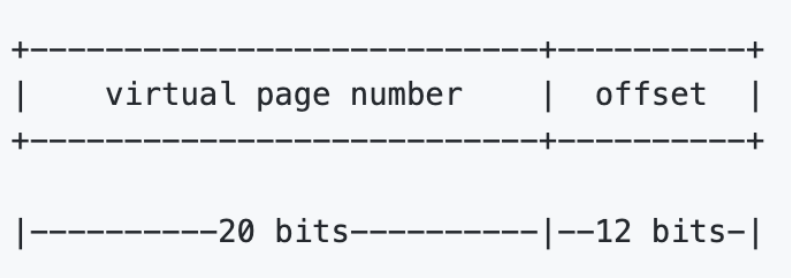
```
typedef struct
{
  uint64 present                :1;
  uint64 writeable              :1;
  uint64 user_access            :1;
  uint64 write_through          :1;
  uint64 cache_disabled         :1;
  uint64 accessed               :1;
  uint64 ignored_3              :1;
  uint64 size                   :1; // must be 0
  uint64 ignored_2              :4;
  uint64 page_ppn               :28;
  uint64 reserved_1             :12; // must be 0
  uint64 ignored_1              :11;
  uint64 execution_disabled     :1;
} __attribute__((__packed__)) PageMapLevel4Entry;
```

64-bit (C Data Structure)

Page Table so Far

32-bit virtual addresses → 4GB virtual memory

4KB pages → 4GB / 4KB = 2^20 = 1M pages

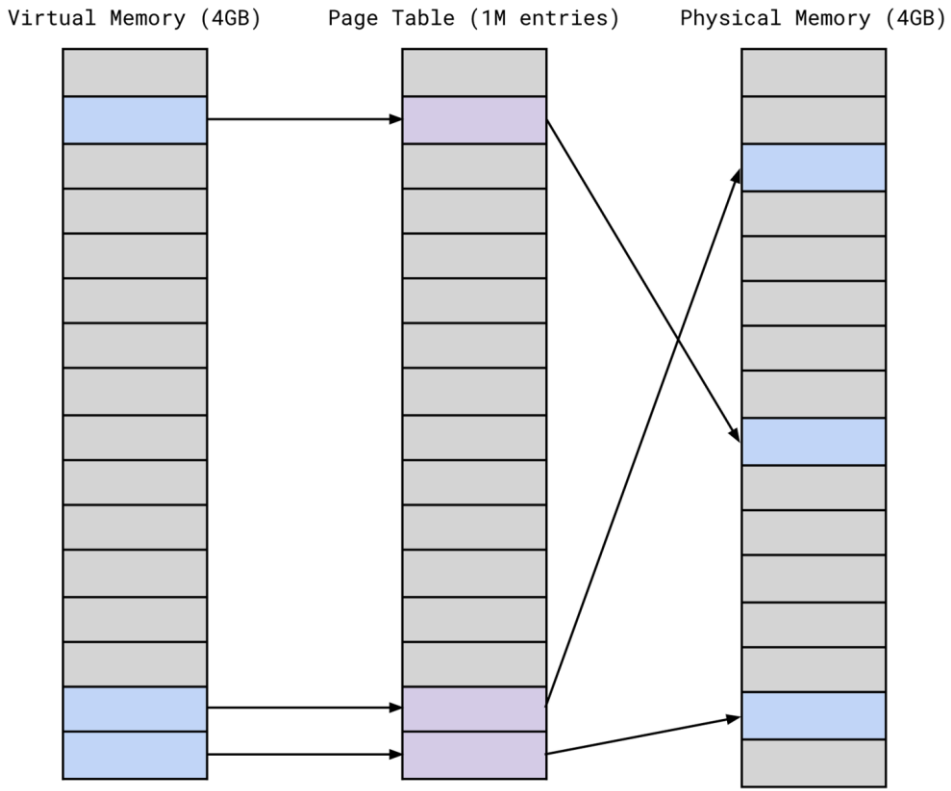


Page table entry = 4B = 20 bits for PFN (assuming 4GB physical memory)

+ 12 bits for metadata

=> 4B x 1M entries = **4MB per page table**

Problem: Sparsity



2-Level Page Table

Page table entry = 4B and page = 4KB

⇒ We can fit 4KB / 4B = 1024 entries into one page

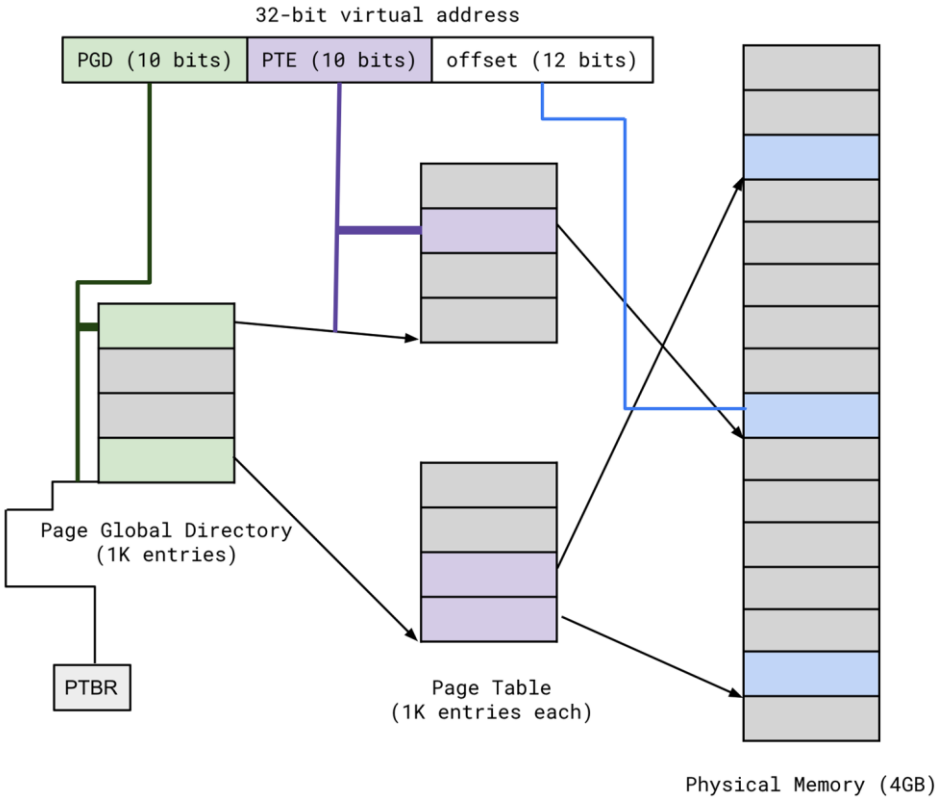
⇒ 1M entries and 1K entries per page ⇒ 1K pages needed

Idea: Let's allocate only the page table pages we are going to use

How: Page table global directory (PGD), which has 1024 entries, 1 entry per page table chunk

!!!The PGD fits exactly into 1 page!!!

2-Level Page Table



⇒ PGD Entry = PGD base (from PTBR)

+ PGD index

⇒ Frame Base = [PTE Base (from PGD entry)

+ PTE Index]

⇒ Physical Addr = Frame Base

+ offset

Bigger Memory?

Let's say we have 64GB of physical memory (but still 4GB virtual address space)

⇒ 36-bits physical addresses [12-bit offset and 24-bit PFN]

⇒ 32-bit page table entry too small → 64-bit = 8B

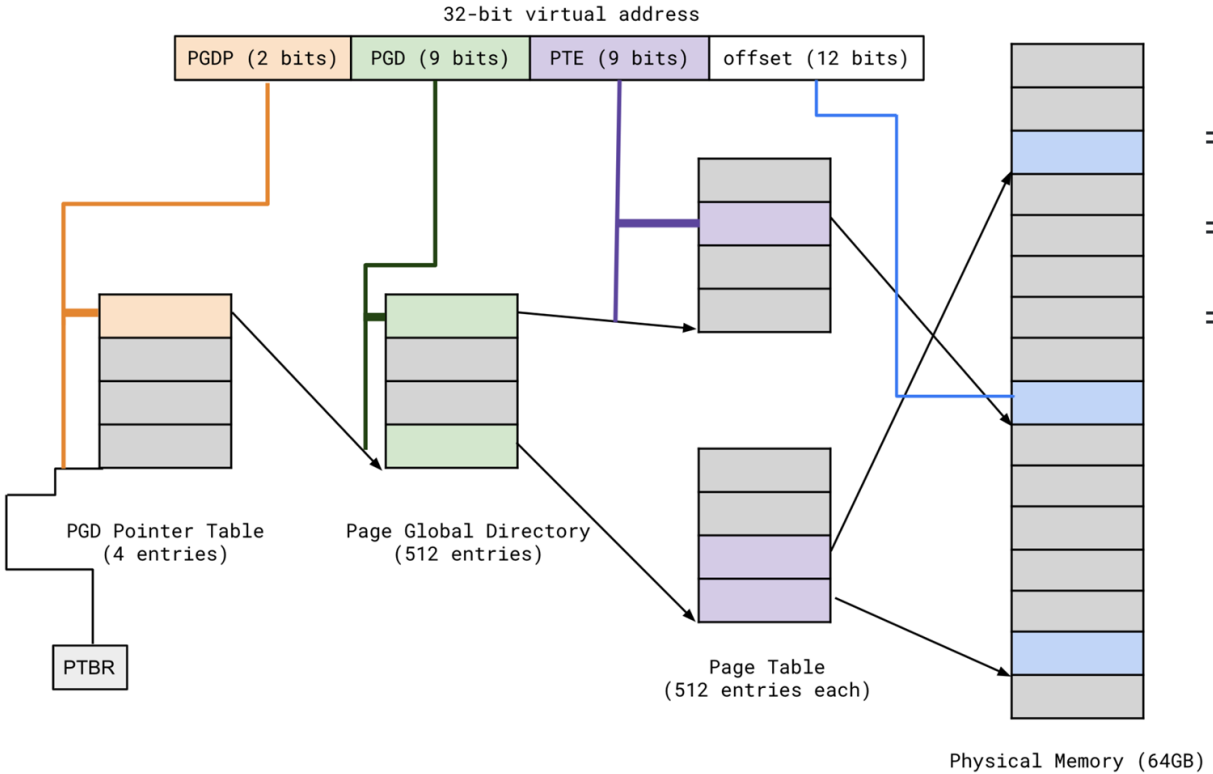
⇒ 512 entries per page → 2K pages needed

PGD used to have 1024 entries but now:

- PGD can only fit 512 entries
- There are 2K page table chunks

Problem: We now have 4 PGDs. How do we choose what PGD to use?

Bigger Memory? More levels



⇒ 9 bits for PTE (512 pages)

⇒ 9 bits for PGD (512 chunks)

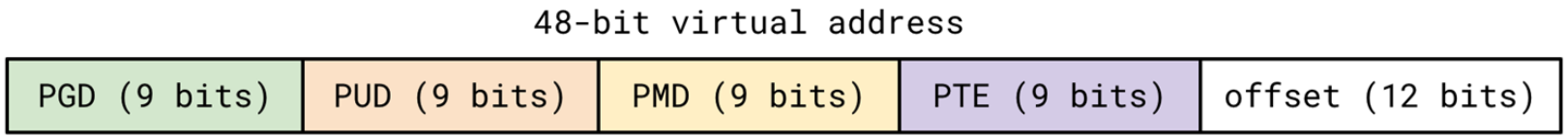
⇒ 2 bits for PGDP (4 PGDs)

Even more (4!) levels

64-bit CPU → 16EB

48-bit virtual addresses → 256 TB of addressable virtual memory

⇒ 64-bit page table entry and 4KB pages



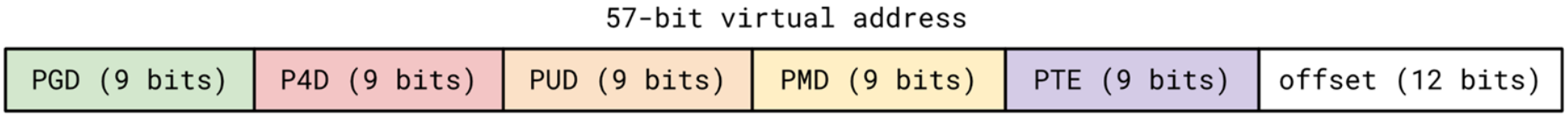
1. Page Global Directory
2. Page Upper Directory
3. Page Medium Directory
4. Page Table Entry

Why not 5 levels?

64-bit CPU → 16EB

57-bit virtual addresses → 128 PB of addressable virtual memory

⇒ 64-bit page table entry and 4KB pages



Another Idea: Inverted Page Tables

The number of physical pages/frames is limited.

Why not have one page table entry for every physical page?

E.g., 512GB physical memory → 128 GB / 4KB = 32M entries in total

The entry contains:

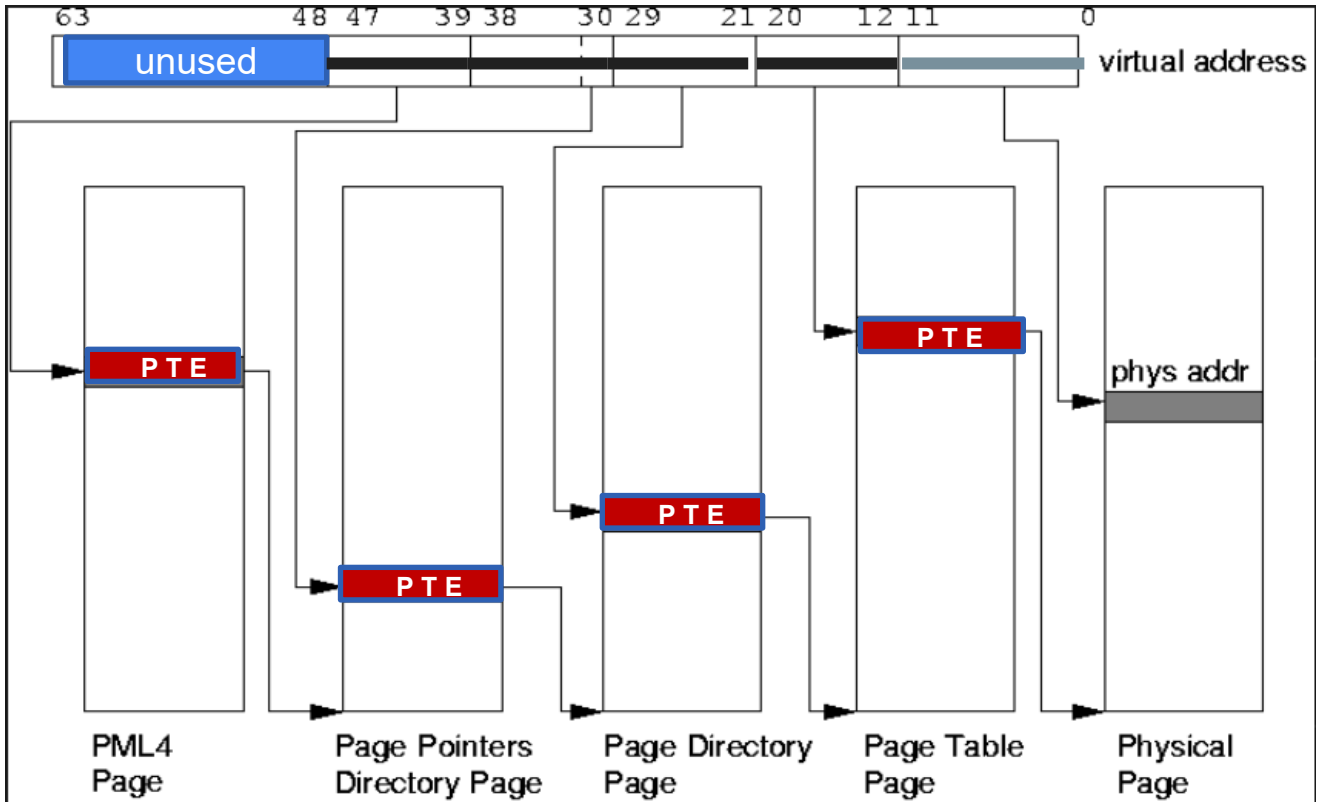
- Physical page number
- Virtual page number
- Metadata
- **PID**

Challenge: How to make a fast lookup?

Used in IBM Power

Problem: Memory Access Amplification

In a 4-level page table, we can incur **four** additional memory accesses per pointer dereference!



Problem: Memory Access Amplification

In a 5-level page table, we can incur **five** additional memory accesses per pointer dereference!

But memory accesses exhibit locality:

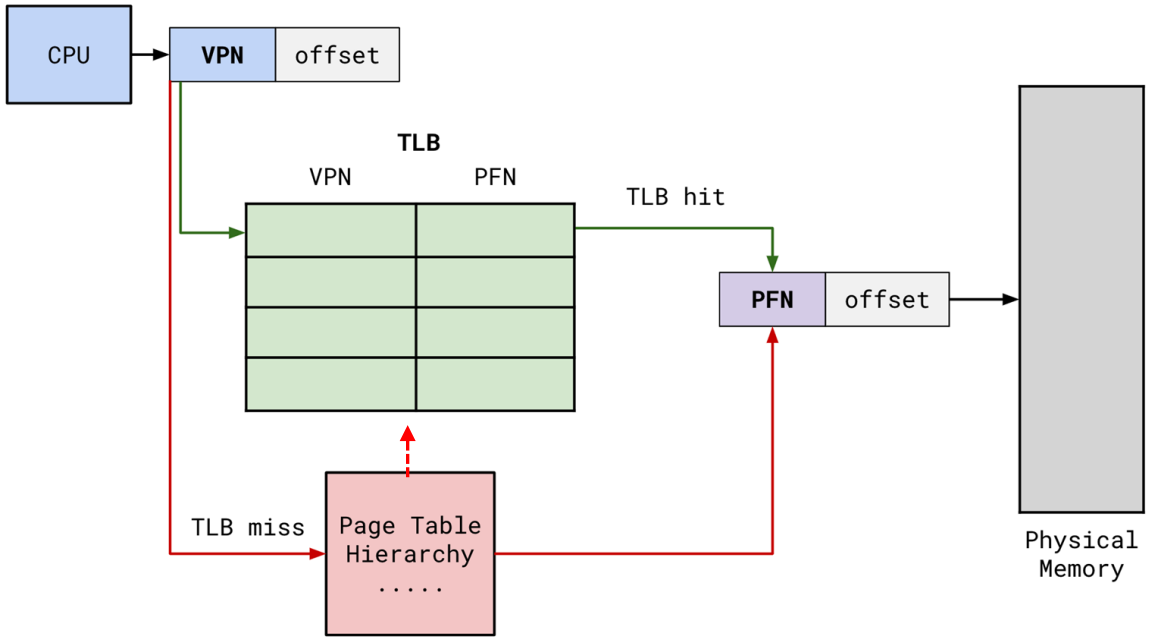
- Temporal: programs typically work within recently accessed memory
- Spatial: programs tend to access adjacent memory locations (e.g. array)

⇒ **Observation**

At any given time, program only needs a small number of VPN->PFN mappings!

Translation Lookaside Buffer (TLB)

MMU employs a fast-lookup hardware cache called “associative memory” which is ***small in size*** and ***supports fast parallel search***



Why TLB helps?

Assumptions:

- memory cycle consumes 1 unit of time
- TLB lookup time: ε
- TLB hit ratio: α, percentage of times than a VPN->PFN mapping is found in the TLB
  - Expect hit ratio to be high, like .95-.99.
    4KB pages are pretty big and locality says we will probably stay within the region.
- 1-level page table

Why TLB helps?

EAT = (1 + e) a + (2 + e)(1 − a)
- If TLB hit, then just incur TLB lookup and memory cycle

EAT = a + ea + 2 + e − ea − 2a
EAT = 2 + e − a
- Assuming a high TLB-hit ratio and a low TLB lookup time, EAT approaches the cost of 1 memory cycle (worth it!)

TLB when context switching

Option 1: flush the entire TLB

- x86 has `load cr3` instruction: load page table base and flush TLB
- TLB entries have metadata bits, e.g. “valid” bit, set all to 0 to “flush” TLB
- this makes context switch pretty expensive, we lose all our cached lookups

Option 2: attach ID to TLB entries

- associate each task with an address space identifier (ASID)
- don’t have to flush TLB on context switch, just check ASID associated with caller

x86 also has `INVLPG addr` instruction, invalidates 1 TLB entry

- e.g., after `munmap()`, region is no longer mapped

TLBs in x86

- Typical: 64-128 entries, 4-way to fully associative, 95% hit rate
- Often separate instruction and data L1 TLBs
- Modern CPUs add second-level TLB with ~1,024 entries (typically unified)

Need to take caching into account!

Do caches use virtual or physical addresses?

Quick Primer: Cache Lookup

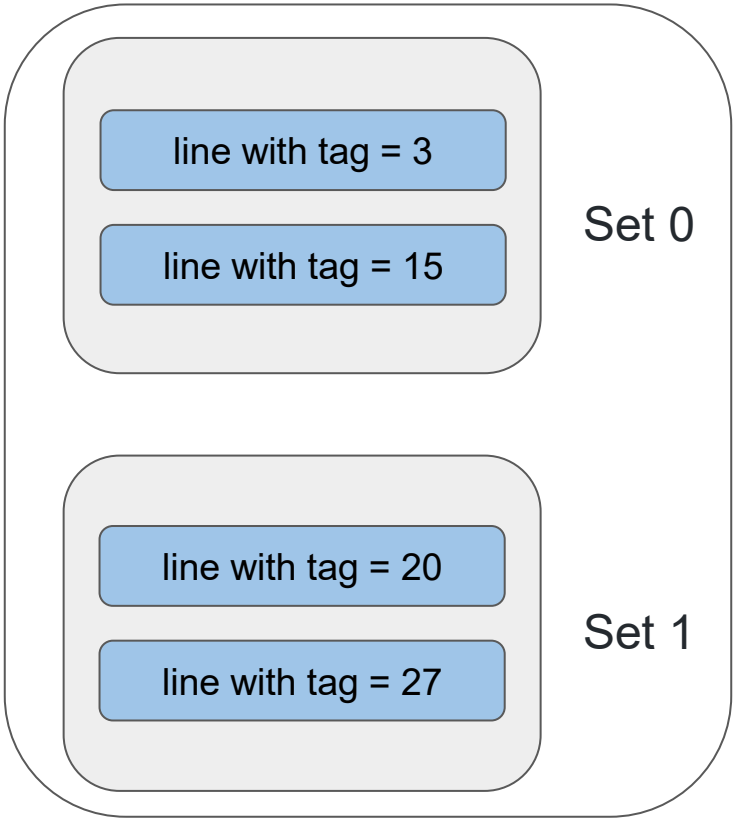
Cache is organized in **sets**

Each **set** has a number of **lines** (64 bytes)

```
Address = tag | index | offset
```

1. Use **index** to find the correct set
2. Use **tag** to check if the slot contains the correct **line**

Q: Why not just have no index and check the tag?



TLBs in x86

- Typical: 64-2K entries, 4-way to fully associative, 95% hit rate
- Modern CPUs add second-level TLB with ~1,024 entries
- Often separate instruction and data TLBs

Need to take caching into account!

Do caches use virtual or physical addresses?

- The L1 is virtually indexed/physically tagged
- L2/L3 etc. use physical addresses

A different MMU: MIPS

- **Hardware checks TLB on application load/store**
  - References to addresses not in TLB trap to kernel
- **Each TLB entry has the following fields: Virtual page, Pid, Page frame, metadata**
- **Kernel itself is unpaged**
  - All of physical memory is contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses
- **User TLB fault handler is very efficient**
  - Two hardware registers reserved for it
  - OS is free to choose page table format!