

# x86 Memory Management

W4118 Operating Systems I

[columbia-os.github.io](https://github.com/columbia-os)

# 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!

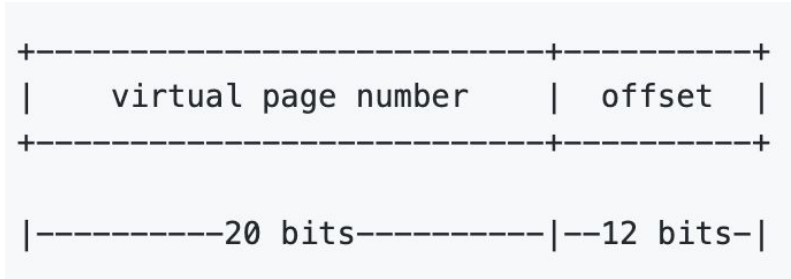
Consider 32-bit virtual address space (4GB), 4KB page size, page table entry size of 4B.

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

# Page Table so Far

32-bit virtual addresses → 4GB virtual memory

4KB pages →  $4\text{GB} / 4\text{KB} = 2^{20} = 1\text{M}$  pages

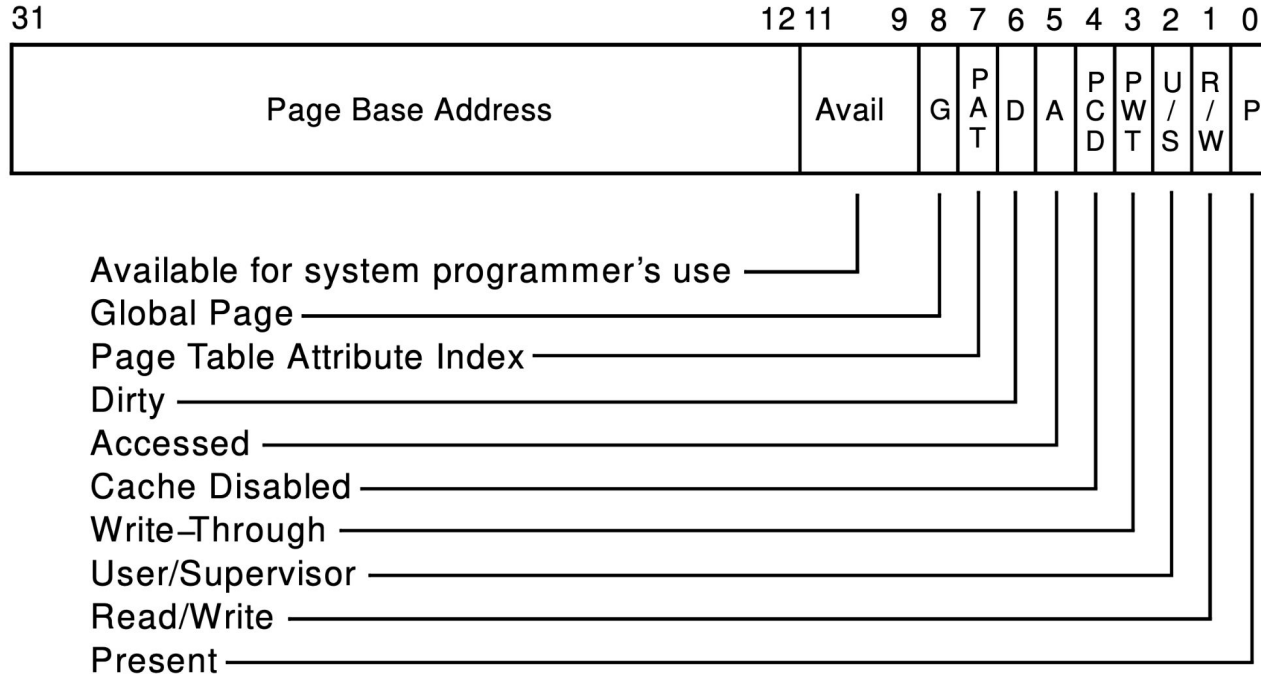


Page table entry = 4B = 20 bits for PFN (assuming 4GB physical memory)

+ 12 bits for metadata

# Page Table Entry Structure

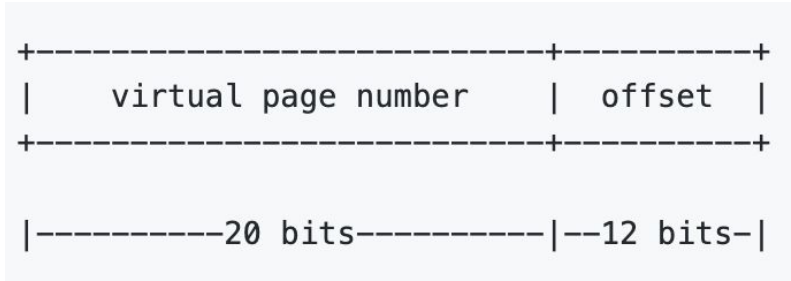
Page-Table Entry (4-KByte Page)



# Page Table so Far

32-bit virtual addresses → 4GB virtual memory

4KB pages →  $4\text{GB} / 4\text{KB} = 2^{20} = 1\text{M}$  pages

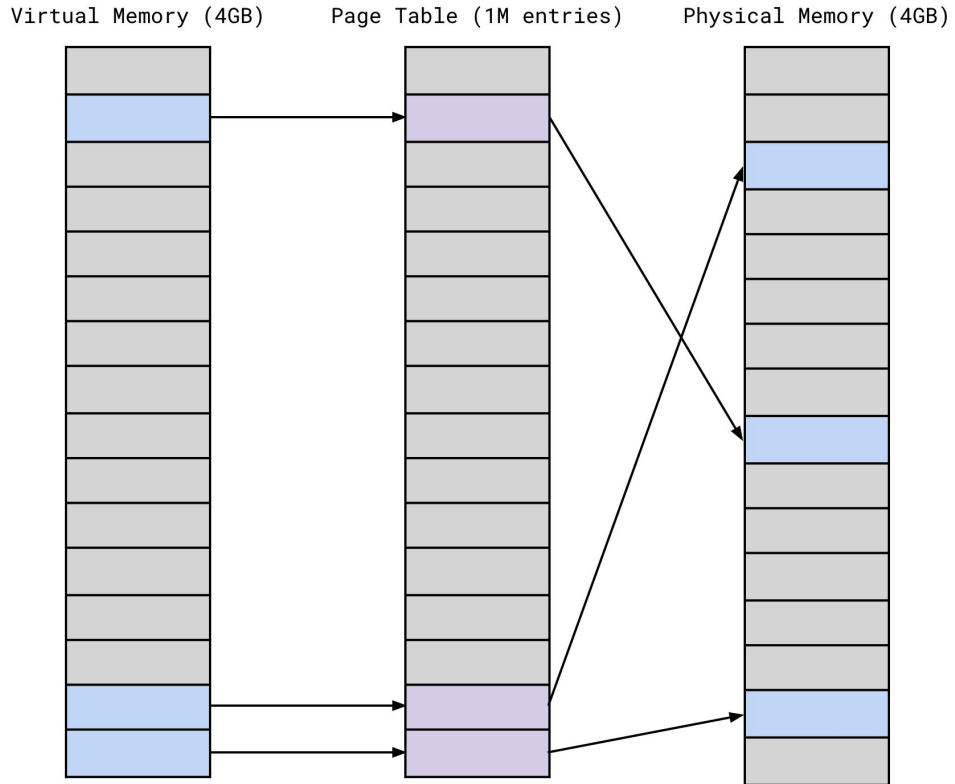


Page table entry = 4B = 20 bits for PFN (assuming 4GB physical memory)

+ 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  $4\text{KB} / 4\text{B} = 1024$  entries into one page

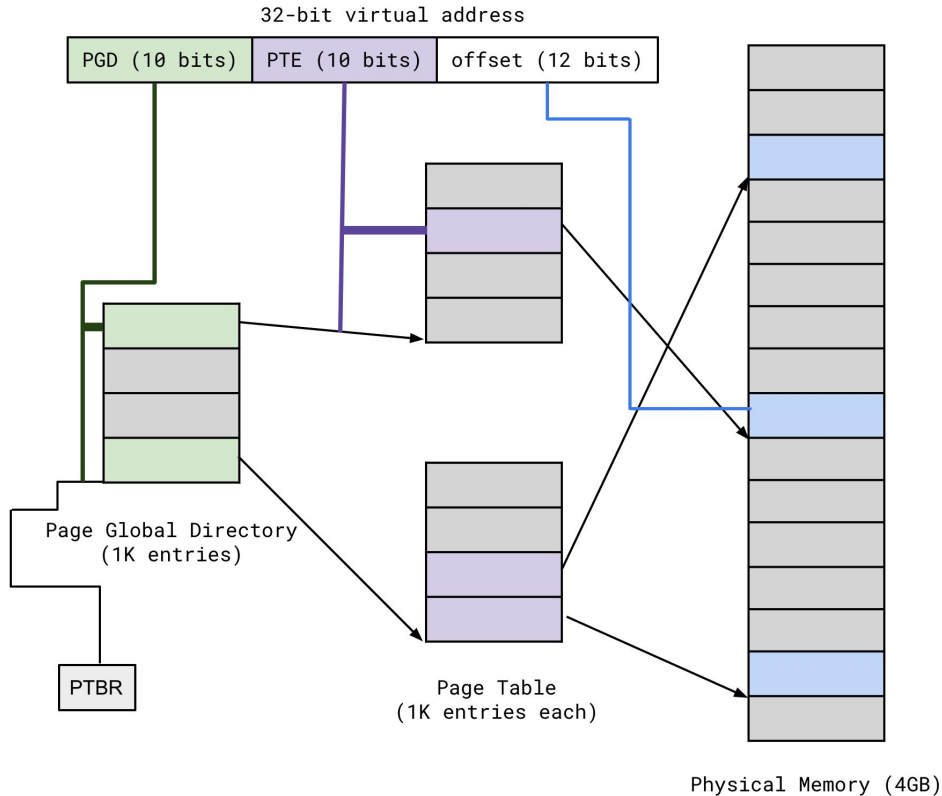
⇒ 1M entries and 1K entries per page ⇒ 1K pages needed

**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

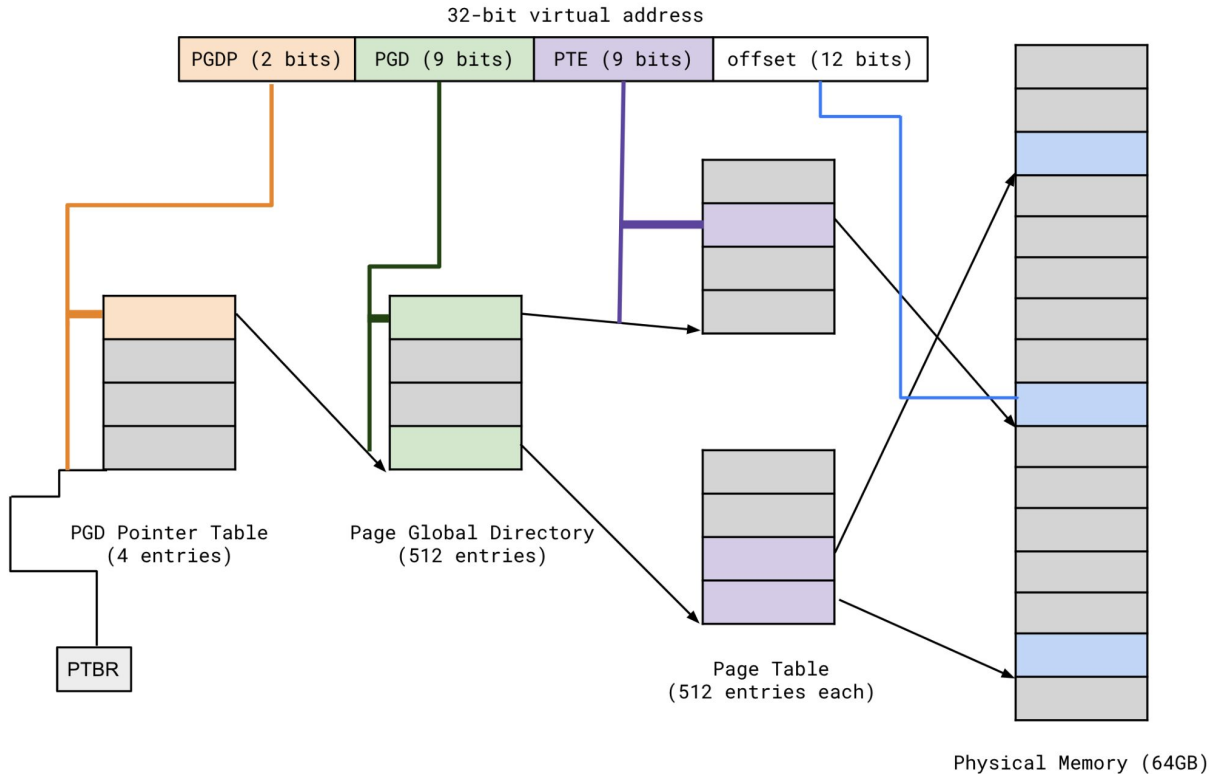
⇒ 512 entries per page → 2K pages needed

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

48-bit virtual address



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

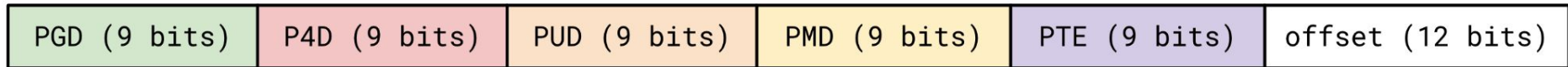
# Why not 5 levels?

64-bit CPU → 16EB

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

⇒ 64-bit page table entry and 4KB pages

57-bit virtual address



# 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  $\rightarrow$  128 GB / 4KB = 32M entries in total

The entry contains:

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

Used in IBM Power

**Challenge:** How to make a fast lookup?

# 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:

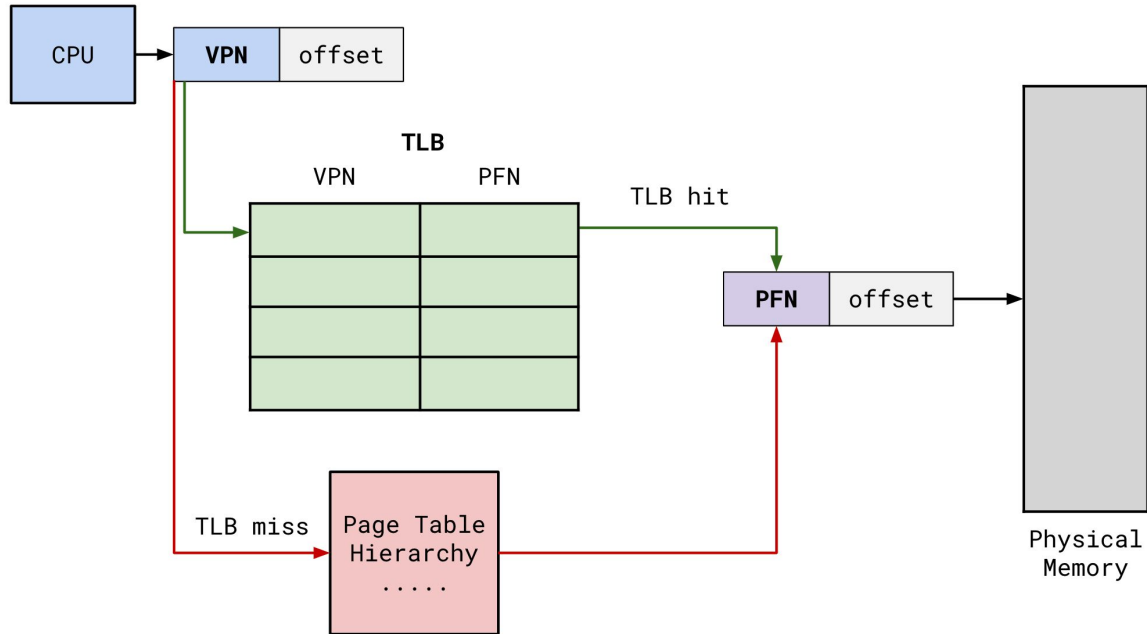
- Temporal: programs typically work within recently accessed memory
- Spatial: programs tend to access adjacent memory locations (e.g. array)

⇒ **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:  $\epsilon$
- TLB hit ratio:  $\alpha$ , percentage of times that 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?

$$\text{EAT} = (1 + e) a + (2 + e)(1 - a)$$

- If TLB hit, then just incur TLB lookup and memory cycle

$$\text{EAT} = a + ea + 2 + e - ea - 2a$$

$$\text{EAT} = 2 + e - a$$

- 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
- Modern CPUs add second-level TLB with ~1,024 entries
- Often separate instruction and data L1 TLBs

**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 unpagged**
  - 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!