6.828: Virtual Memory

Adam Belay

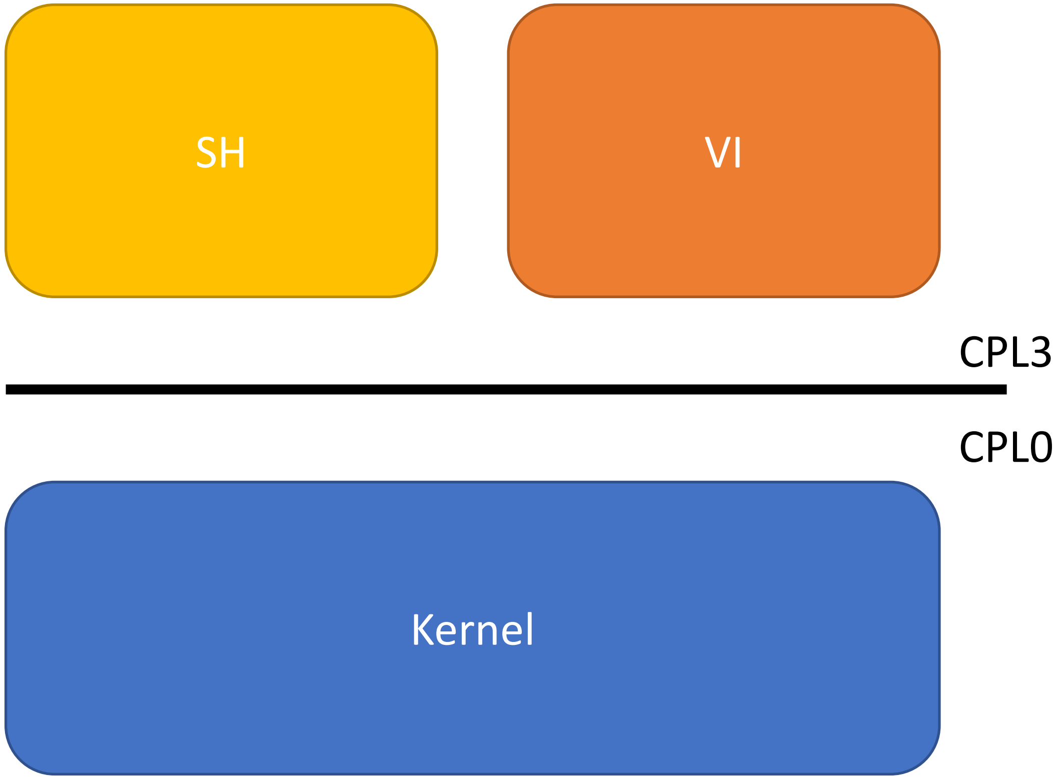
abelay@mit.edu

Outline

- Address spaces
- x86 Paging hardware
- xv6 VM code
- System call homework solutions

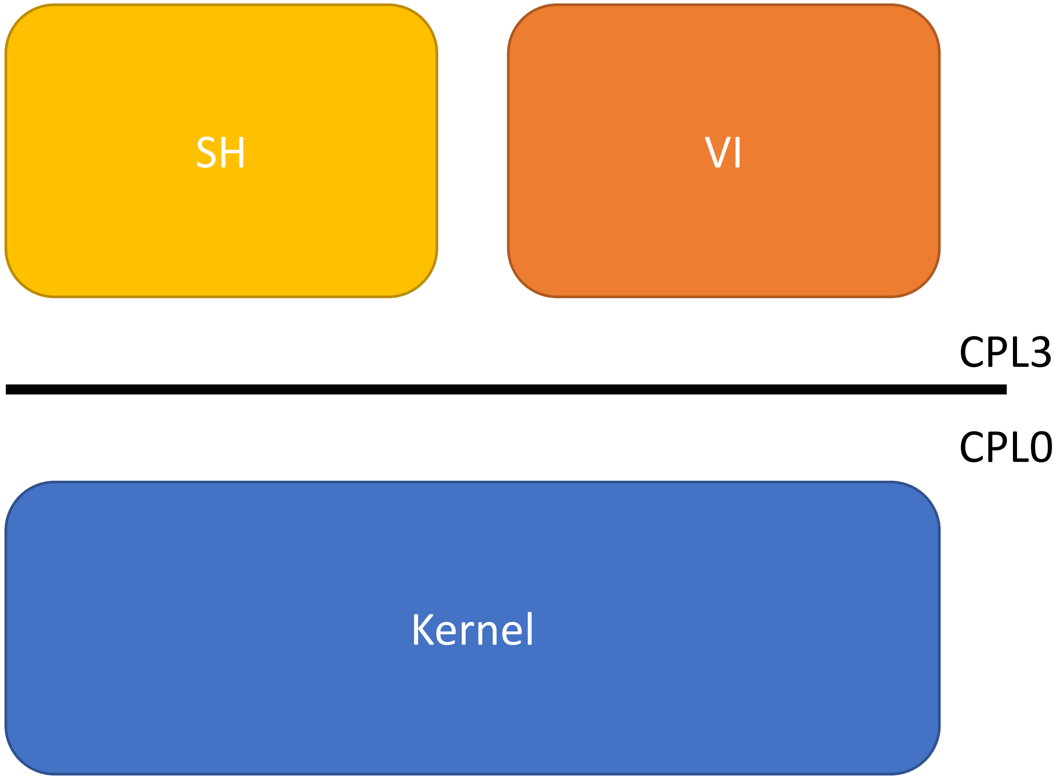
Today’s problem

Protection View:

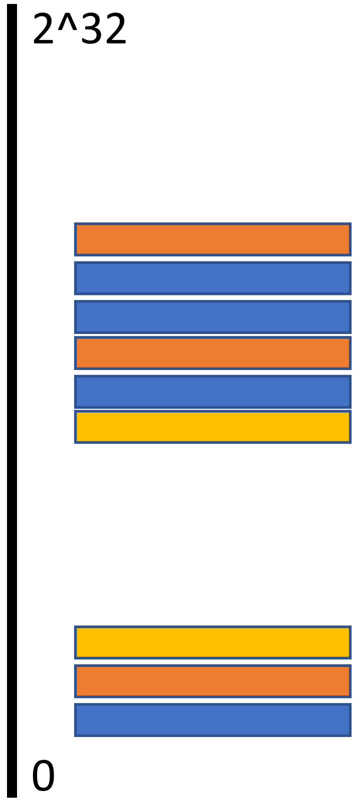


Today's problem

Protection View:



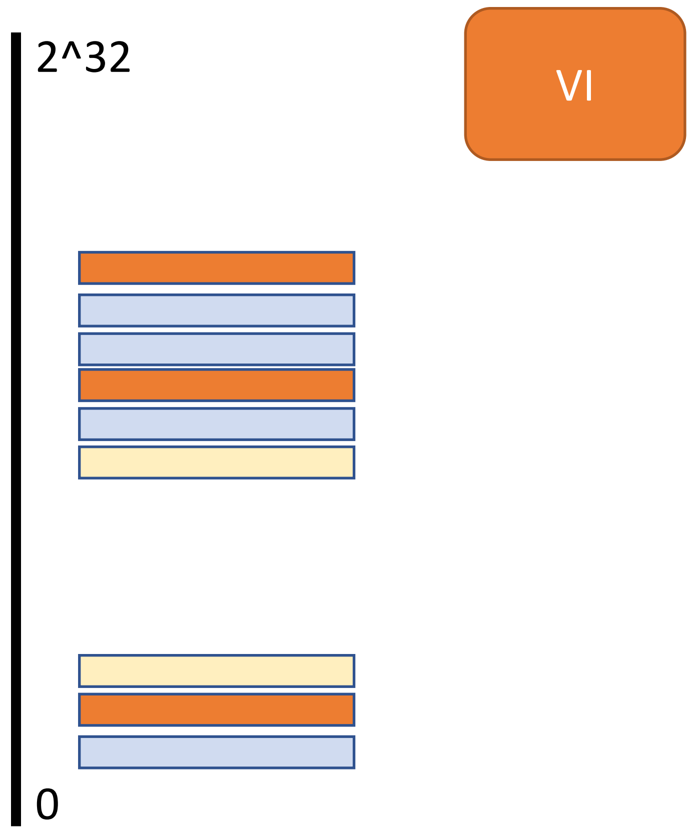
Physical Memory View:



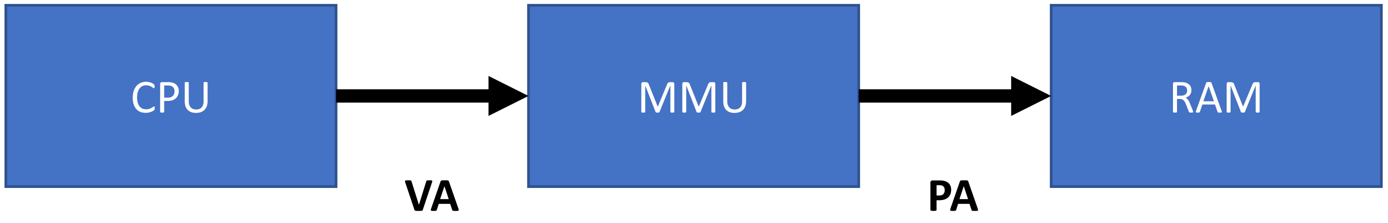
Goal: Isolation

- Each process has its own memory
- Can read and write its own memory
- But cannot read or write the kernel's memory or another process' memory

Physical Memory View:



Solution: Introduce a level of indirection

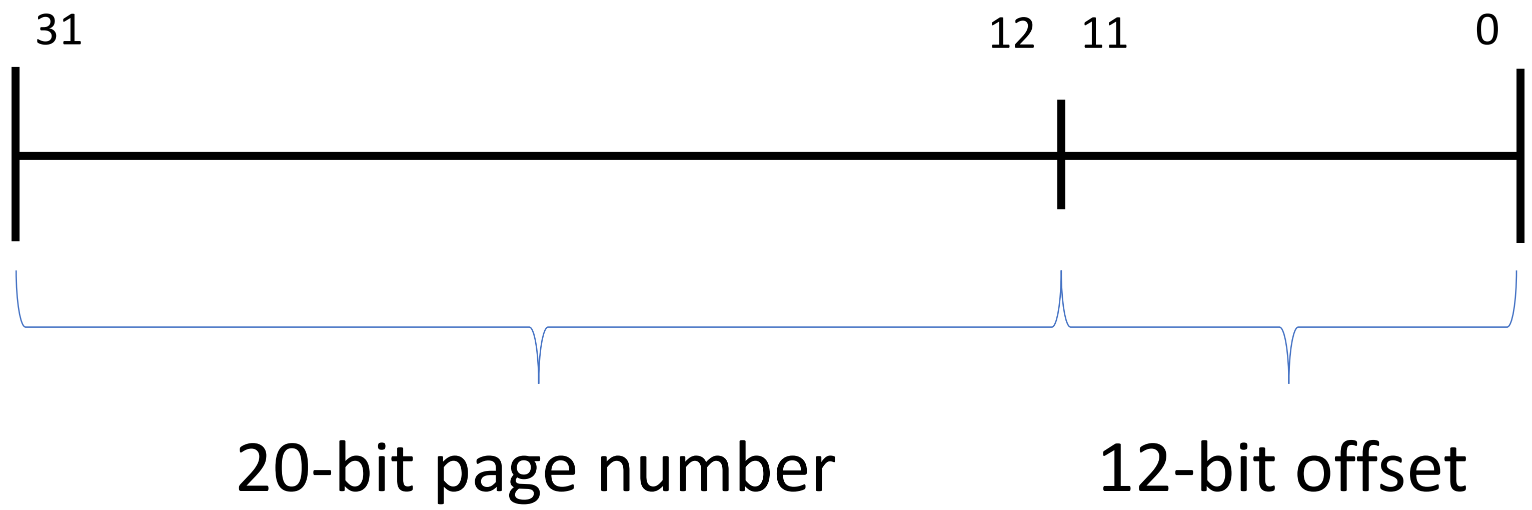


- Plan: Software can only read and write to virtual memory
- Only kernel can program MMU
- MMU has a **page table** that maps virtual addresses to physical
- Some virtual addresses restricted to kernel-only

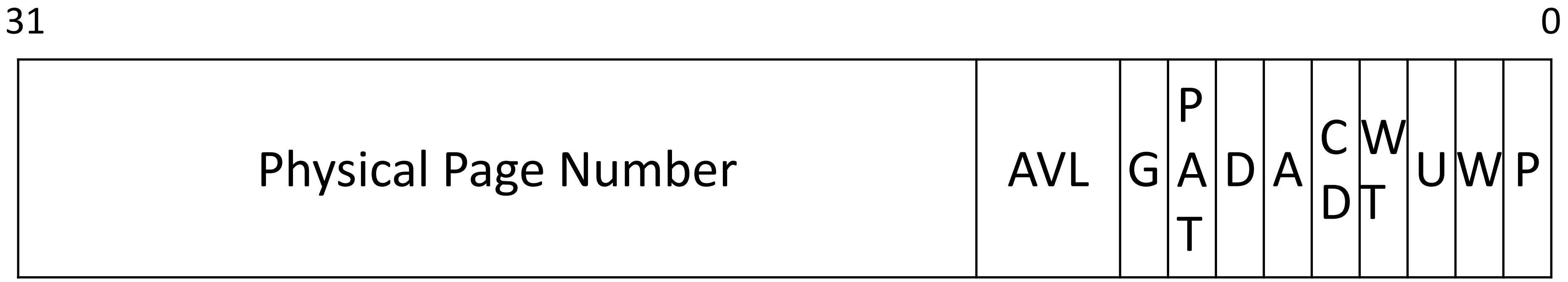
Virtual memory in x86

Virtual addresses are divided into 4-KB "**pages**"

Virtual Address:



Page table entries (PTE)



Some important bits:

- **Physical page number**: Identifies 20-bit physical page location; MMU replaces virtual bits with these physical bits
- **U**: If set, userspace (CPL3) can access this virtual address
- **W**: If set, the CPU can write to this virtual address
- **P**: If set, an entry for this virtual address exists
- **AVL**: Ignored by MMU

Strawman: Store PTEs in an array

GET_PTE(va) = &ptes[va >> 12]

How large is the array?

PPN										
...										
...										
...										
...										
...										
...										
...										
...										

...

Strawman: Store PTEs in an array

GET_PTE(va) = &ptes[va >> 12]

PPN										
…										
…										
…										
…										
…										
…										
…										
…										

…

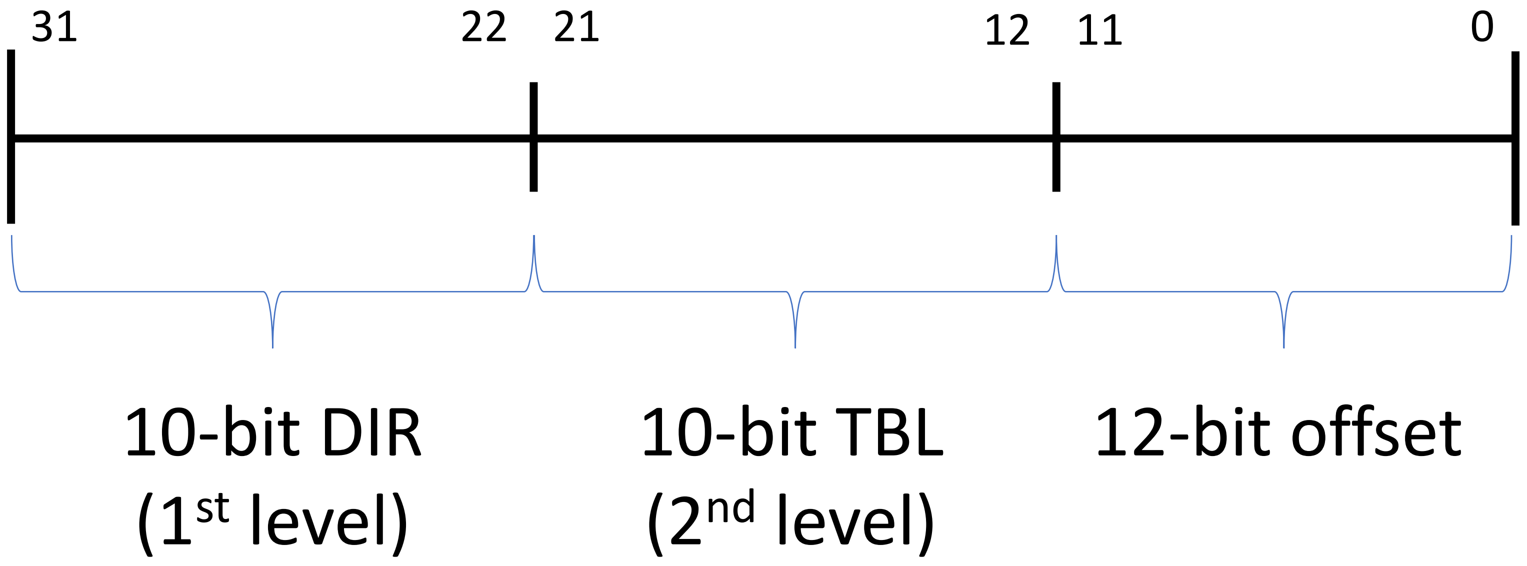
How large is the array?

2^20 * 32 bits

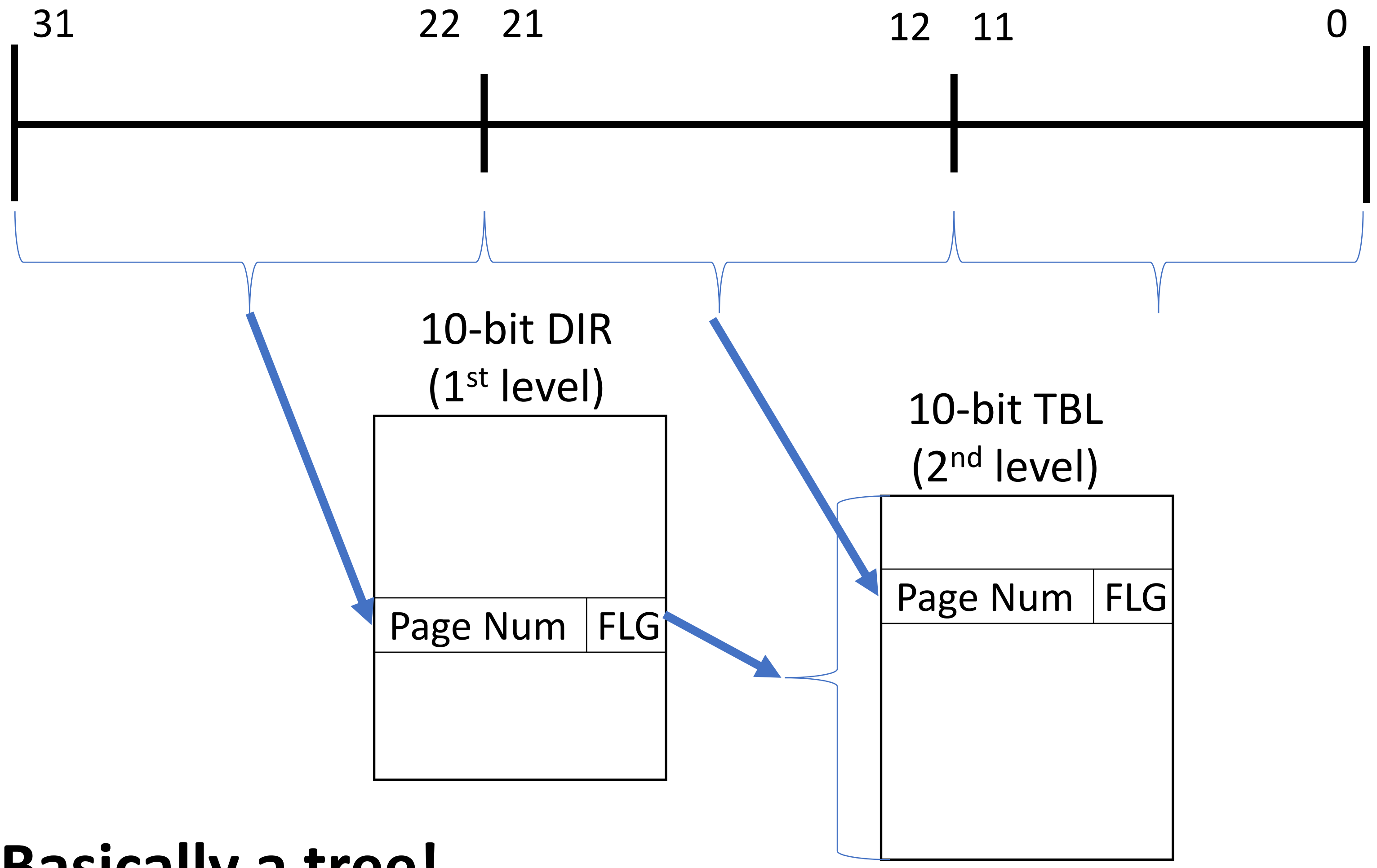
2^20 * 4 bytes

4 Megabytes!

x86 solution: Use two levels to save space

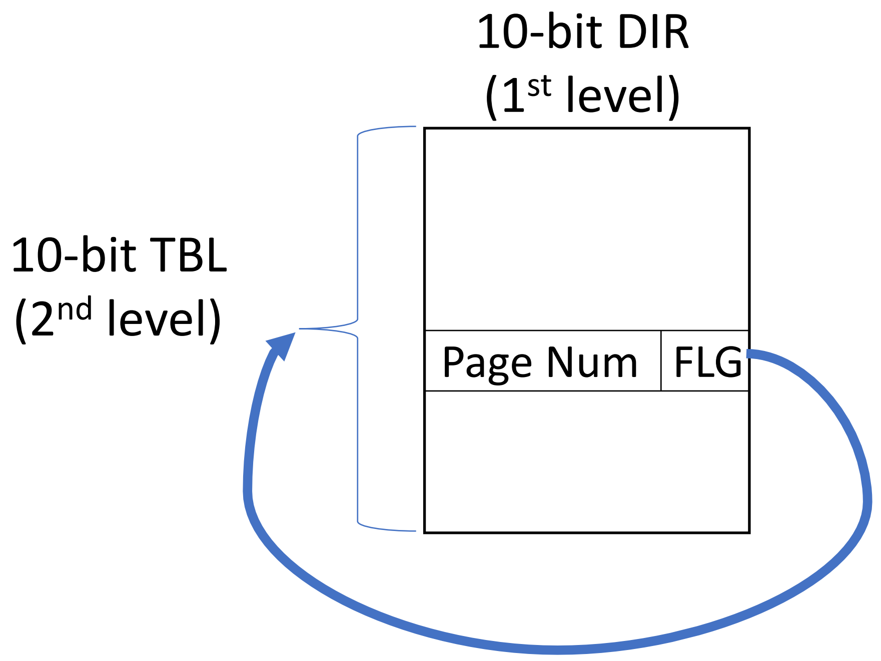


x86 solution: Use two levels to save space

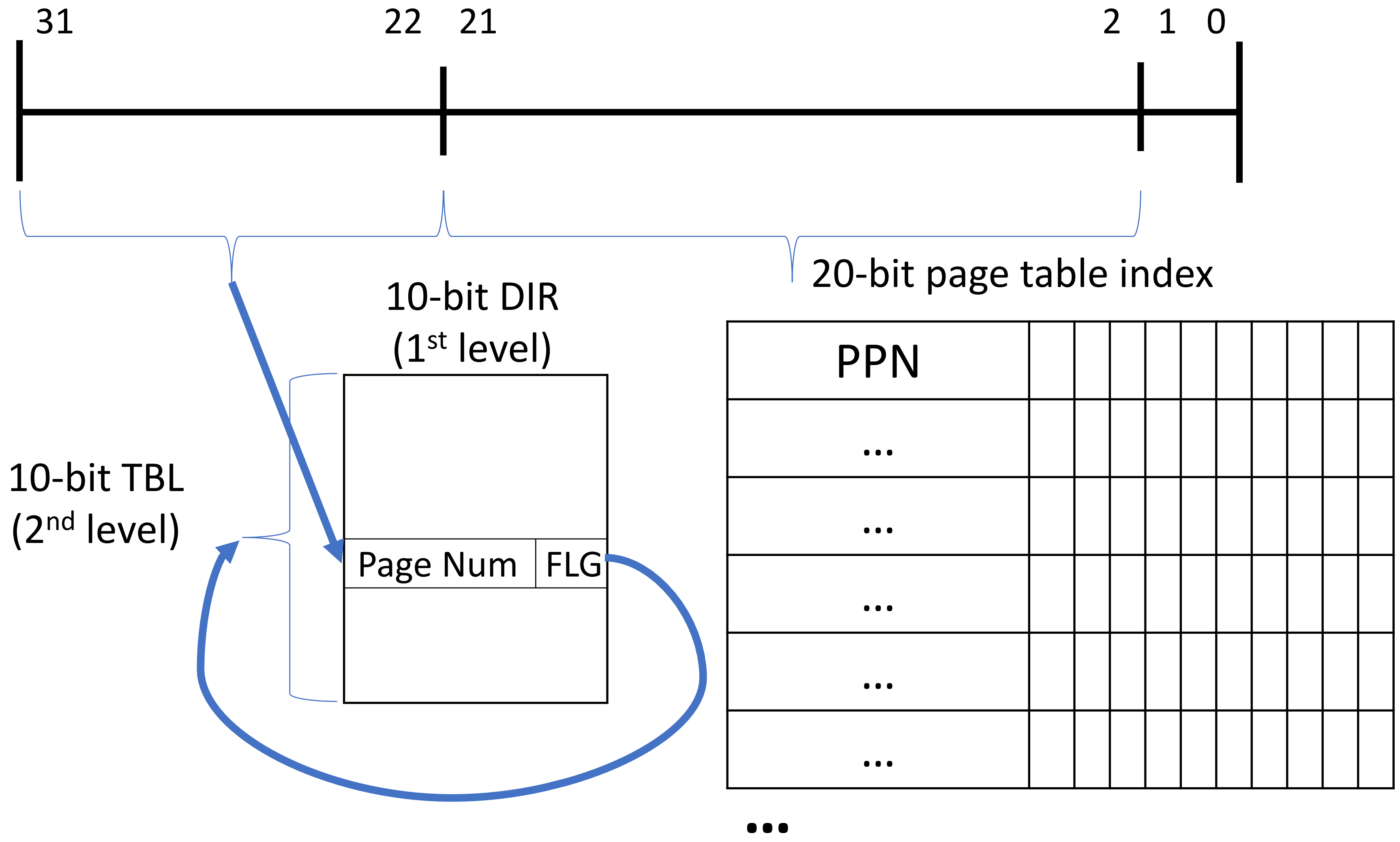


Basically a tree!

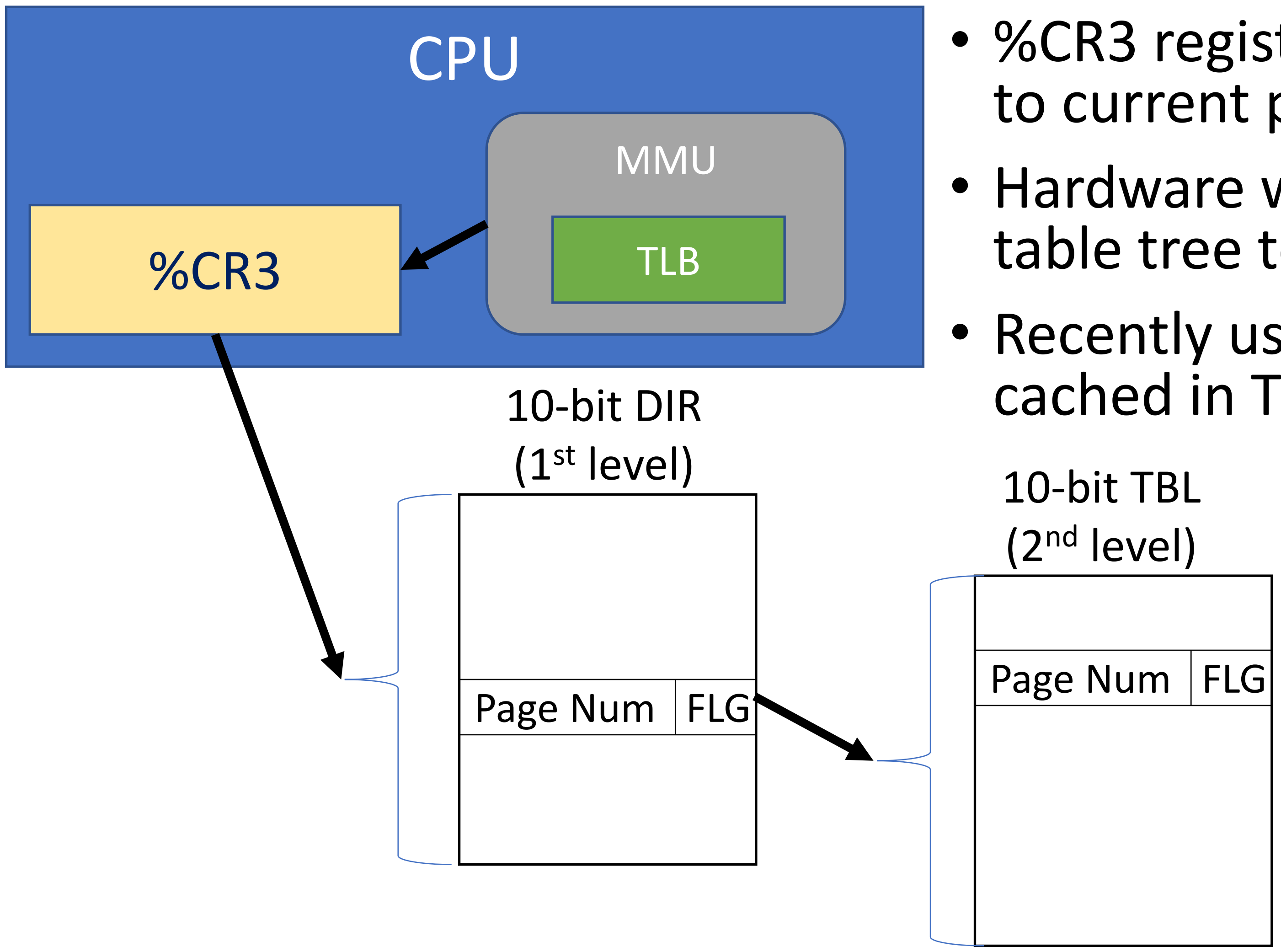
What about a recursive mapping?



What about a recursive mapping?



How do we program the MMU?



- %CR3 register is a pointer to current page table
- Hardware walks page table tree to find PTEs
- Recently used PTEs cached in TLB

Let’s talk more about flags

	Read Not Allowed	Read Allowed
Write Not Allowed	No Flags	PTE_P
Write Allowed	Not Possible	PTE_P | PTE_W

- If **PTE_U** is cleared, only the kernel can access
  - Why is this needed?
- What happens if flag permission is violated?
  - We get a page fault!
  - Then what happens?