XV6 Virtual Memory

Paging provides a level of indirection for addressing

- CPU -> MMU -> RAM
- VA -> PA

How a VA gets translated to PA?

- use index bits of VA to find a page table entry (PTE) construct physical address using PPN from PTE + offset of VA

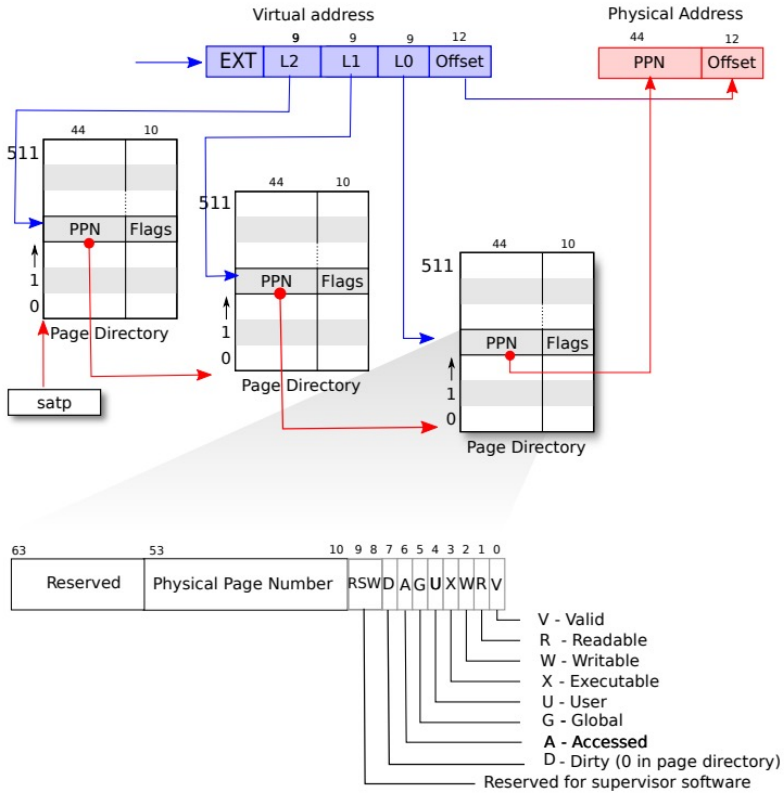

Figure 3.2: RISC-V page table hardware.

RISC-V VA index

- RISC-V maps 4-KB “pages” and aligned — start on 4 KB boundaries 4 KB = 12 bits the RISC-V used in xv6 has 64-bit for addresses thus page table index is top 64-12 = 52 bits of VA except that the top 25 of the top 52 are unused no RISC-V has that much memory now can grow in future so, **index is 27 bits**

What is in PTE?

- each PTE is 64 bits, but only 54 are used top 44 bits of PTE are top bits of physical address “physical page number” low 10 bits of PTE flags Present, Writeable, &c

Where does page table stored?

- in RAM

Would it be reasonable for page table to just be an array of PTEs?

- No. Waste lots of memory for small program. 2^27 entry * 64 bits = 1 GB per page table. If one address space per app, too much wasting!

How does RISC-V solve this?

- RISC-V 64 uses a “three-level page table” to save space Each page directory page (PD) has 512 PTEs. PTEs point to another PD or is a leaf so 512*512*512 PTEs in total. PD entries can be invalid, and PTE pages do not exist. So a page table for a small app is small.

How can MMU figure out where the page table is?

- Register satp holds the physical address of the root page table address. OS saves and rewrite satp when switching to another address space/application.

What if PTE V bit is not set, or R/W bit not set?

- Page fault. Transfer to kernel. Kernel could output error, kill process, or install a PTE and resume. See: how to kill process, how to lazy allocate, how to copy on write.

VM in xv6

- Each process has its own address space, and its own page table. Kernel switches page tables when switching processes.

Jump between user program and kernel

- trampoline and trapframe aren’t writable by user process. both kernel and user map trampoline and trapframe page.

- Two good reasons:
  - eases transition user -> kernel and back
  - kernel doesn’t map user applications
- Not easy for kernel to r/w user memory. Need translate user virtual address to kernel virtual address But good for isolation (see spectre attacks)

How to translate address

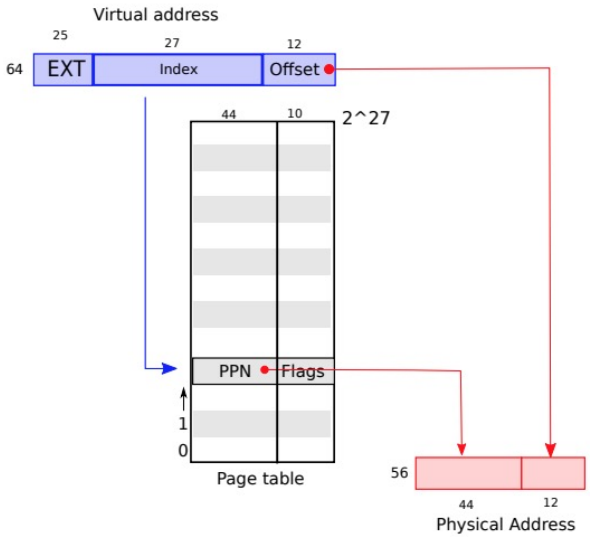

Figure 3.1: RISC-V virtual and physical addresses.

- The total supported number of pages are: 2^27
- Each size is 2^12.
- GB size is 2^30.
- MAXVA is actually one bit less than the max allowed by Sv39, to avoid having to sign-extend virtual addresses that have the high bit set.
- #define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
- So the total virtual memory can support up to 2^(27+12-30 - 1) => 2^8.
- The available virtual memory is 256 GB.

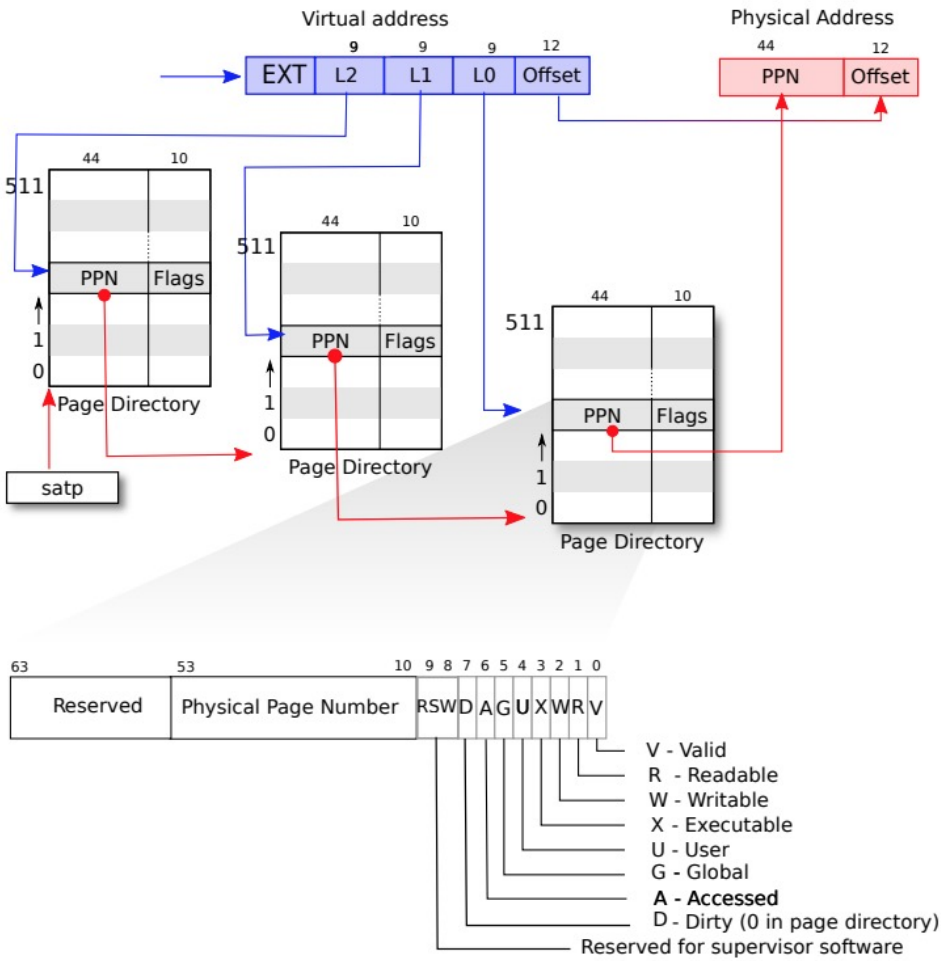

Figure 3.2: RISC-V page table hardware.

Page Fault and Swap

- Page can store in disk, if memory is running out of space. Page Table will have bit indicate page is valid but not present. When reference a not in memory page, the handler will generate a page fault. Memory and disk can swap pages whenever needed.

Setup kernel address space

- shift a physical address to the right place for a PTE. #define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)
- Create PTEs for virtual addresses starting at va that refer to physical addresses starting at pa. va and size might not be page-aligned.

Code: Map physical address to virtual address

```
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

Code: Walk the Pagetable to find PTE

```
*// extract the three 9-bit page table indices from a virtual address.*
#define PXMASK          0x1FF // 9 bits. 0x111111111
#define PXSHIFT(level)  (PGSHIFT+(9*(level)))
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)
```

```
*// Return the address of the PTE in page table pagetable*
*// that corresponds to virtual address va.  If alloc!=0,*
*// create any required page-table pages.*
*//*
*// The risc-v Sv39 scheme has three levels of page-table*
*// pages. A page-table page contains 512 64-bit PTEs.*
*// A 64-bit virtual address is split into five fields:*
*//   39..63 — must be zero.*
*//   30..38 — 9 bits of level-2 index.*
*//   21..39 — 9 bits of level-1 index.*
*//   12..20 — 9 bits of level-0 index.*
*//    0..12 — 12 bits of byte offset within the page.*
static pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

Key Features

- The purpose of virtual memory is **isolation**.
- Each process has its own address space.
- Lazy/on-demand page allocation.
- Guard page to protect against stack overflow
- one zero-filled page
- Share kernel page tables in XV6
- **Copy-on-write fork**
- **Demand paging**: on page fault, read the page from file and update page table entry.
- **Memory-mapped files**: Can read and write part of file. Can pay-in pages on demand, and page-out if memory is full. (Code it up!)
- exec nows loads entire file to memory, files reading is slow, and some parts are never used. The solution is to use demand paging (Code it up!)