

Lecture7 Paging

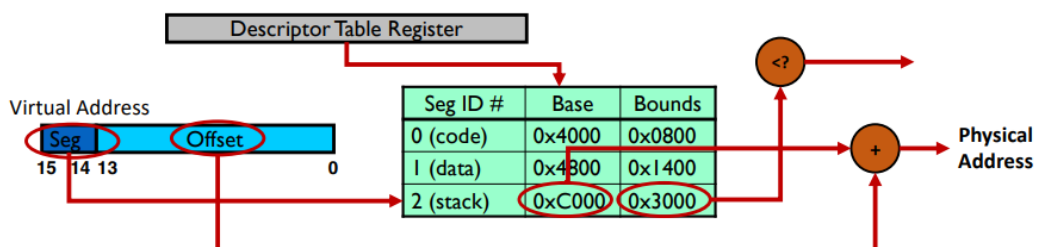
1. Introduction to paging

Problems with segmentation

- OS context switch must also save and restore all pairs of segment registers
- A segment may grow, which may or may not be possible
- Management of free spaces of physical memory with variable- sized segments
- **External fragmentation:** gaps between allocated segments
- **Internal fragmentation:** Segmentation may also have internal fragmentation if more space allocated than needed

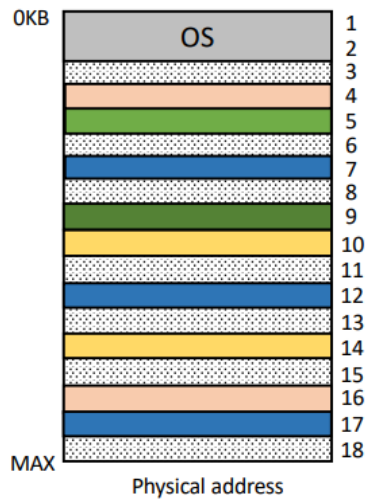
Solutions for Segmentation

- OS context switch must also **save and restore all pairs of segment registers**
 - *Table of base/bounds** stored in memory rather than registers
 - **Pointer to the table** stored in a register
 - Cons: one **more** memory read per **address translation**



- A segment may **grow**, which may or may not be possible
 - Virtually continuous memory can be **physically discontinuous**

Fixed-sized segments

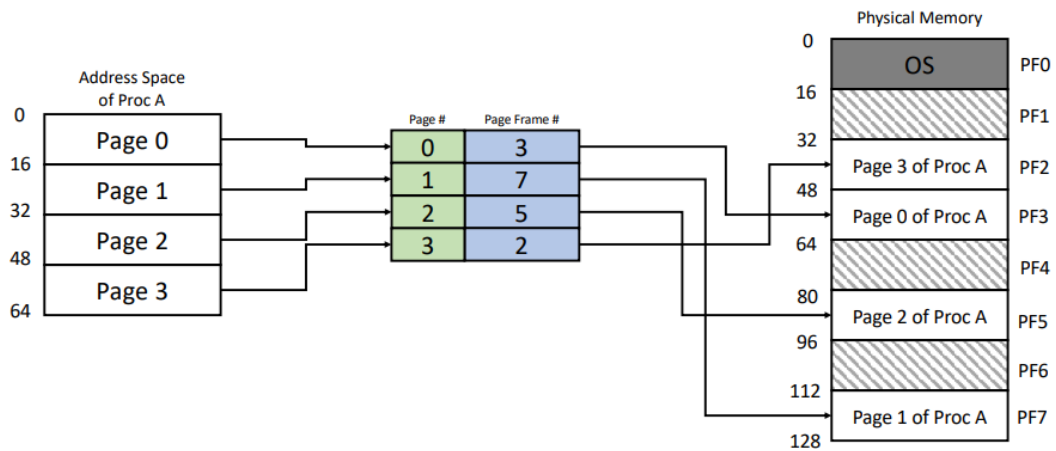
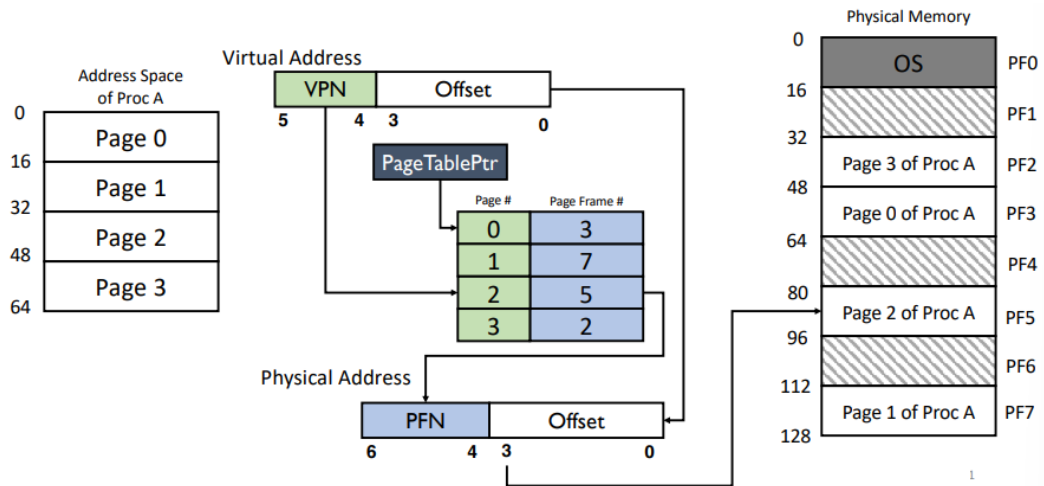
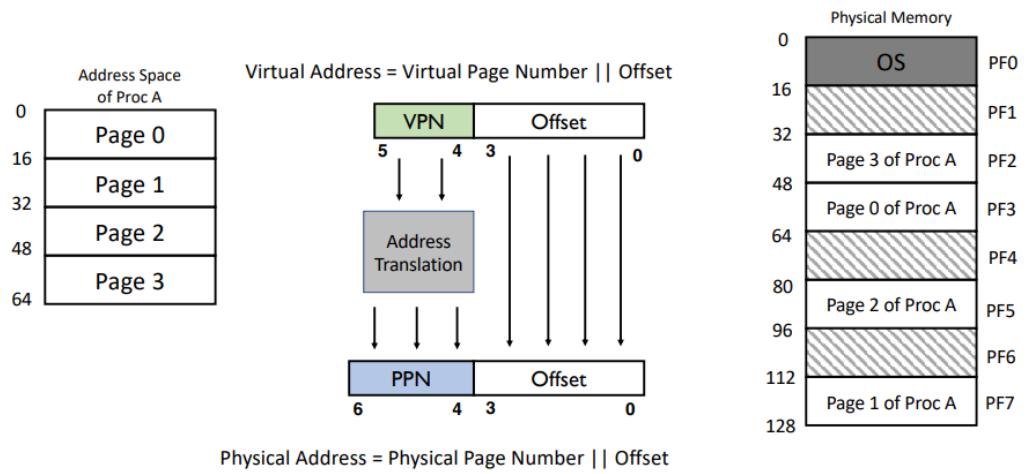


- Management of free spaces of physical memory with fixed-sized segments
 - Fixed-sized segments
 - Physical memory curved into **fixed sized chunks**, **index** by an integer
 - Can use simple vector of bits to handle allocation (1-allocated, 0-free) 1101101011010110
- External fragmentation: solved
- Internal fragmentation: exist

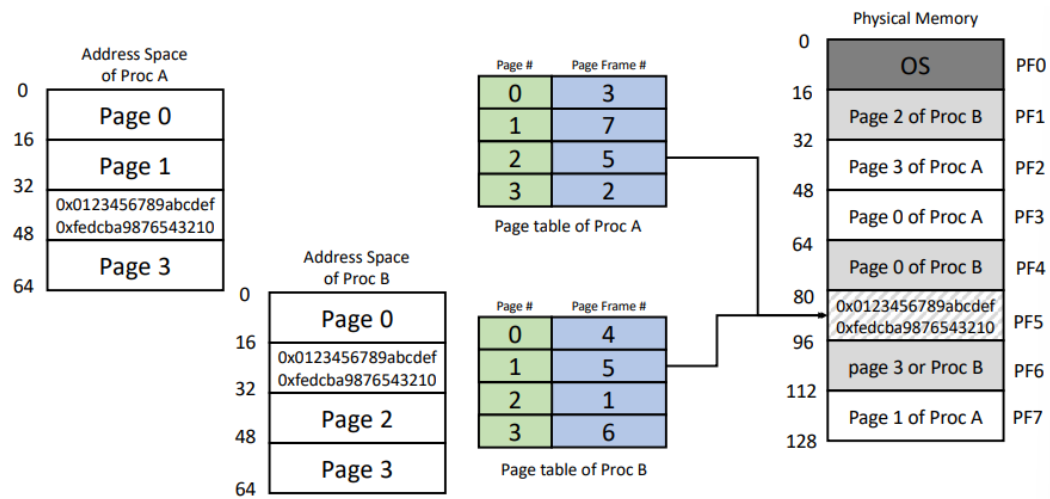
Paging

- **Physical memory** conceptually divided into **fixed size**
 - Each is called a **page frame**
 - Typical size 1KB to 16KB, most ISA uses **4KB**
- **Virtual address** space conceptually divided into the **same size**
 - Each is called a **page**
- Page **mapped** to page frame
 - One to one mapping
 - Many to one mapping -> memory sharing
- **One page table per process**
 - Resides in physical memory
 - One entry for one virtual->physical translation

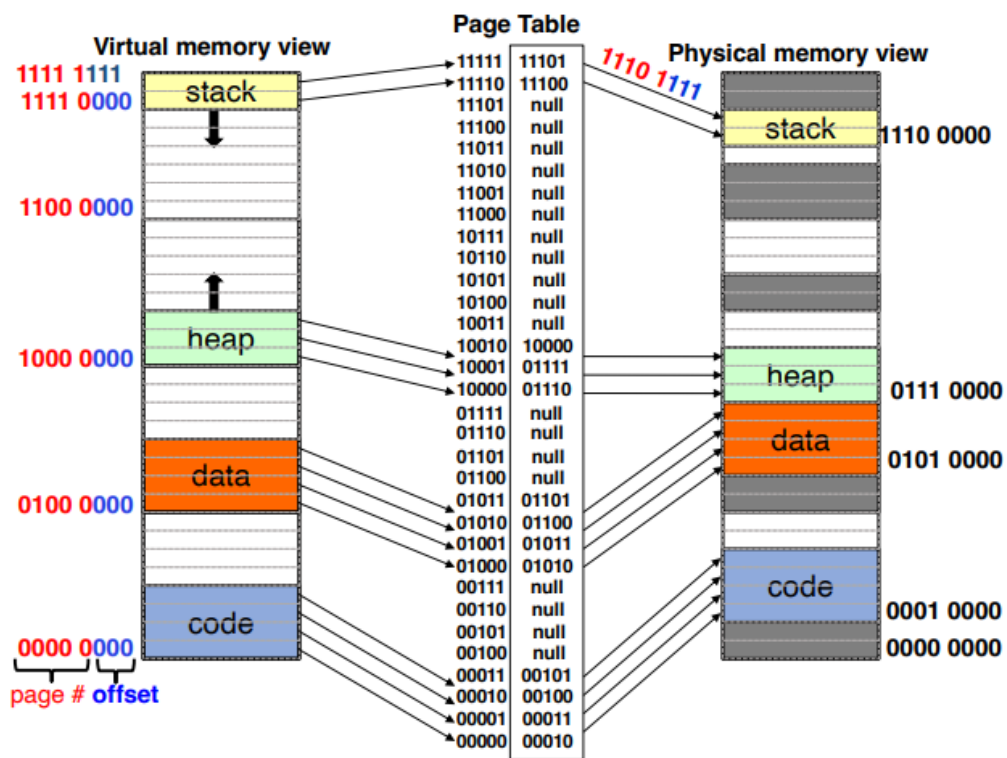
A mapping from a **virtual page** to a **physical page frame**



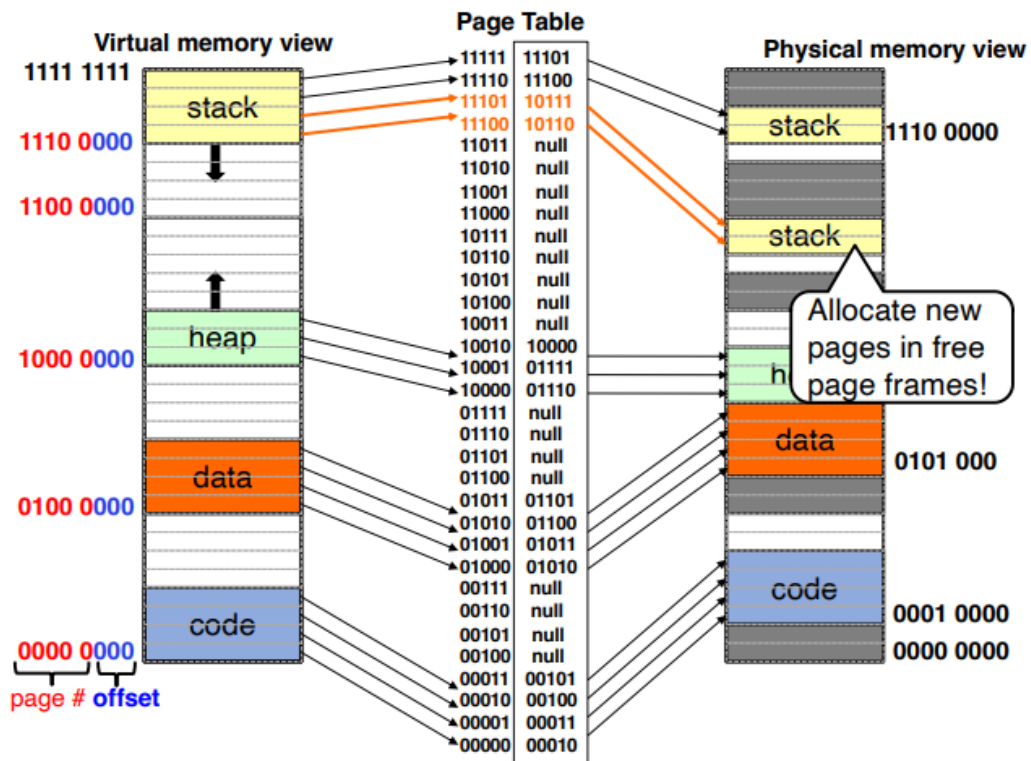
Memory sharing with paging



Growing Problem



- What happens if stack grows to 1110 0000 ?

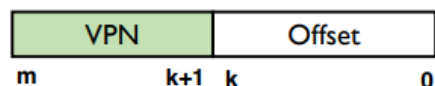


- Allocate new pages in free page frames

Virtual Address

- Length of **virtual address** determines **size of address space**
- Length of **offset** determines size of a **page/page frame**
- A m -bit virtual address and k -bit offset
 - Size of address space: 2^m
 - Size of a page: 2^k
 - Size of the page table: 2^{m-k}
- A 32-bit virtual address, 4KB page
 - virtual address: 32-bit
 - offset: 12-bit

Virtual Address



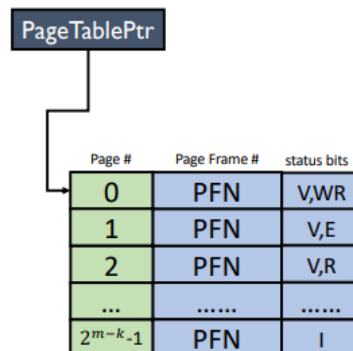
Page #	Page Frame #
0	PFN
1	PFN
2	PFN
...
$2^{m-k}-1$	PFN

Page Table Entry (PTE)

- An entry in the page table is called a page table entry (PTE)
- PTE also contains a **valid bit**
 - Virtual pages with no valid mapping: valid bit = 0
 - Important for sparse address space
- PTE also contains **protection bits**
 - Permission to read from or write, or execute code on this page
- PTE also contains **an access bit, a dirty bit, a present bit**
 - **Present bit**: whether this page is in physical memory or on disk
 - **Dirty bit**: whether the page has been modified since it was brought into memory
 - **Access bit**: whether a page has been accessed

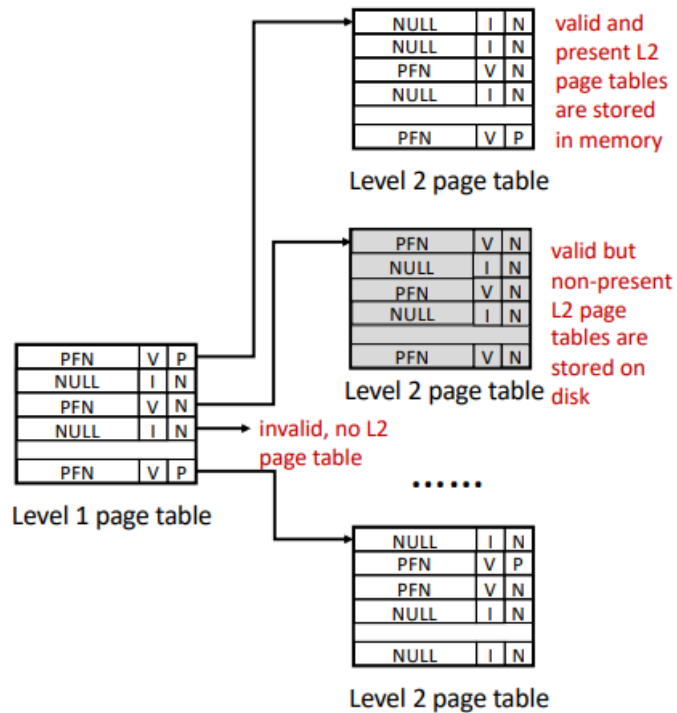
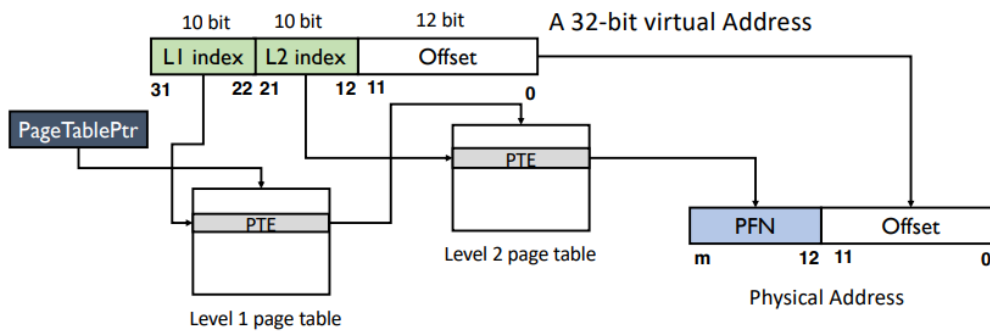
2. Multi-level page tables

How big are page tables



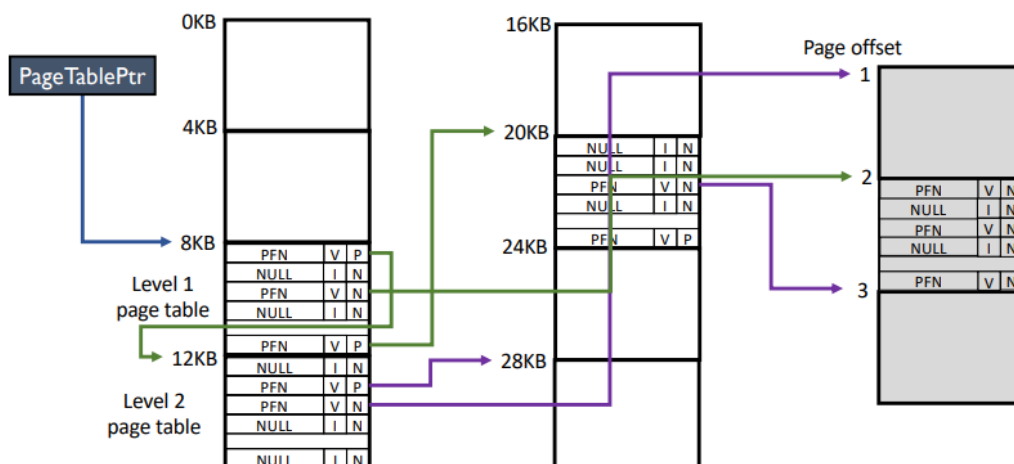
- Assume 32-bit machine and 4KB pages, and each PTE takes 4 bytes (PFN plus status bits)
 - Number of PTEs: $2^{32-12} = 2^{20}$
 - Size of page table: 4MB
- Page tables are stored in **memory** and context switch only changes the pointer to page table

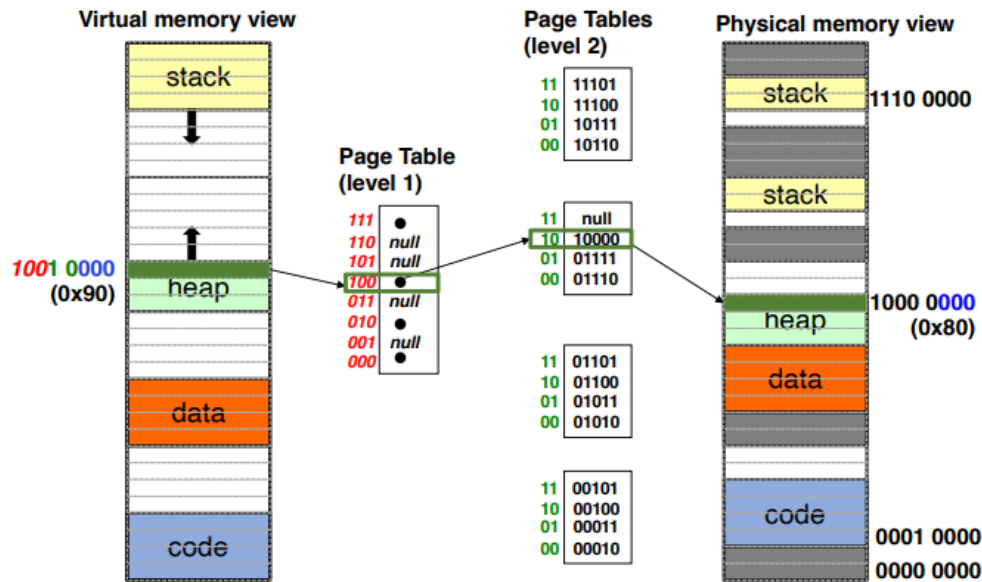
two-level page tables



- 4KB page and 4-byte/PTE: 1024 entries per table
- Two-level page table forms a **tree** of page tables
 - In theory 1024 level-2 page tables, but only a subset are valid
 - A **subset** of valid page tables are stored in memory (others on disk)

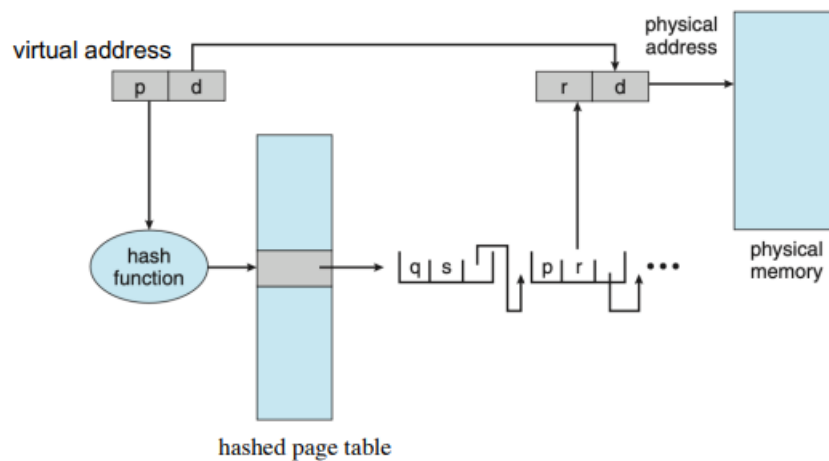
A tree of page table





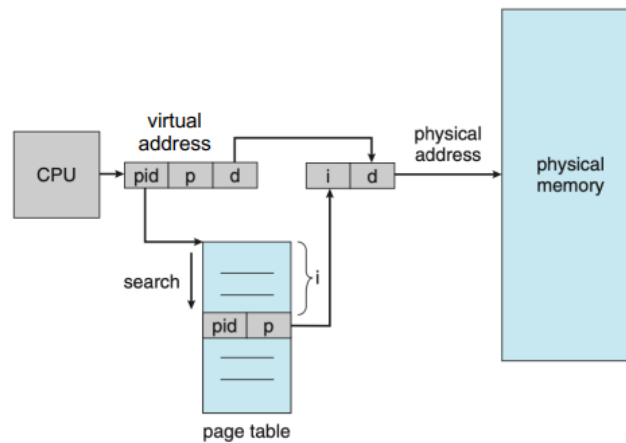
3. Other page table structures

Hashed page tables



- Hash function
 - input: **VPN**
 - output: **index** in the hashed page table
- Collision handling
 - a linked list
 - each element consists of three fields
 - VPN
 - PFN
 - pointer to the next element

Inverted page tables



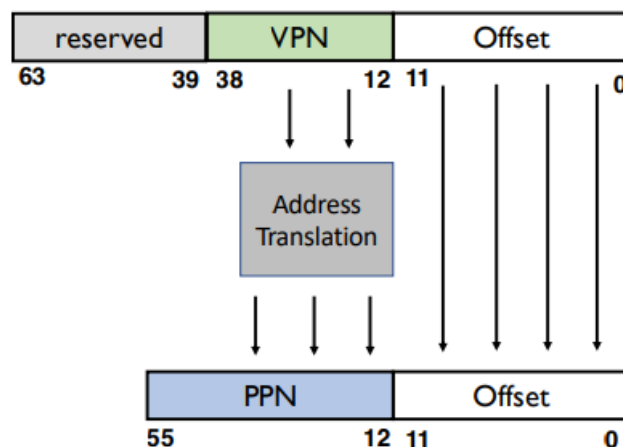
- One page table for the whole system
 - used in 64-bit UltraSPARC and PowerPC
- Each entry corresponds to one physical page frame
 - Process ID and VPN
- Page table lookup requires **linear search** of the entire table
- Memory sharing is hard

4. Real-world paging schemes

Virtual Memory on RISC-V

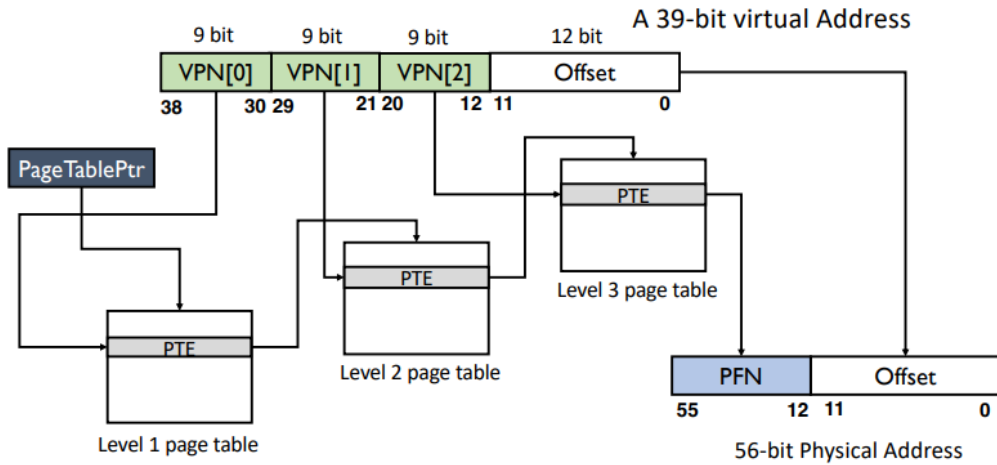
- RISC-V supports multiple MMU
 - For RV32: SV32
 - For RV64: SV39 and SV48

SV39: Three Levels of Page Tables



- Page size: 4KB
- Virtual address: 39-bits
 - remaining bits of total 64 bits reserved

- Physical address: 56-bits

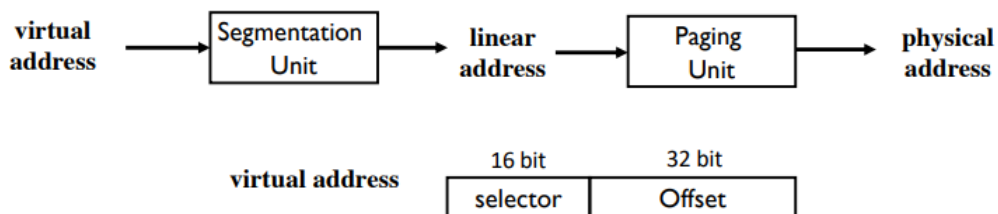


- PTE: 8 bytes (64 bits)
 - Bit 8-0 status bits
 - Bit 53-10 is PFN (or physical page number, PPN)
 - Bit 63-54 reserved
- Status bits: 9 bits

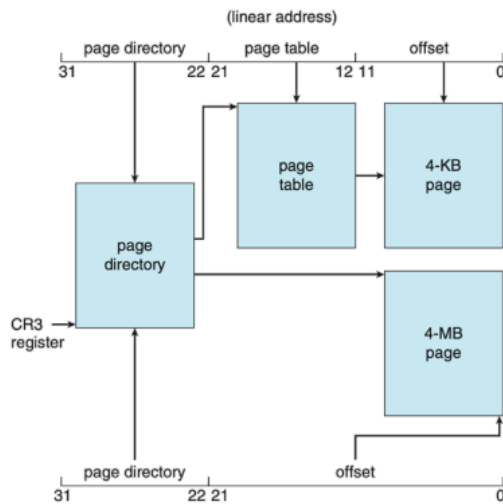
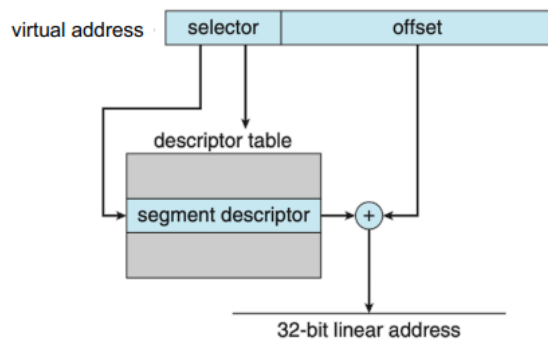
63	54	53	28	27	19	18	10	9	8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V					
10	26	9	9	2	1	1	1	1	1	1	1	1					

- D (dirty), A (accessed), V (valid)
- G (Global): G=1 the page is mapped in all address spaces
- U (User): U-mode code may access this page
- RSW: reserved for s-mode

IA-32



- Intel's 32-bit CPU (IA-32) uses two stage address translation: segmentation + paging
- A virtual address contains a 16-bit segment selector and 32-bit offset



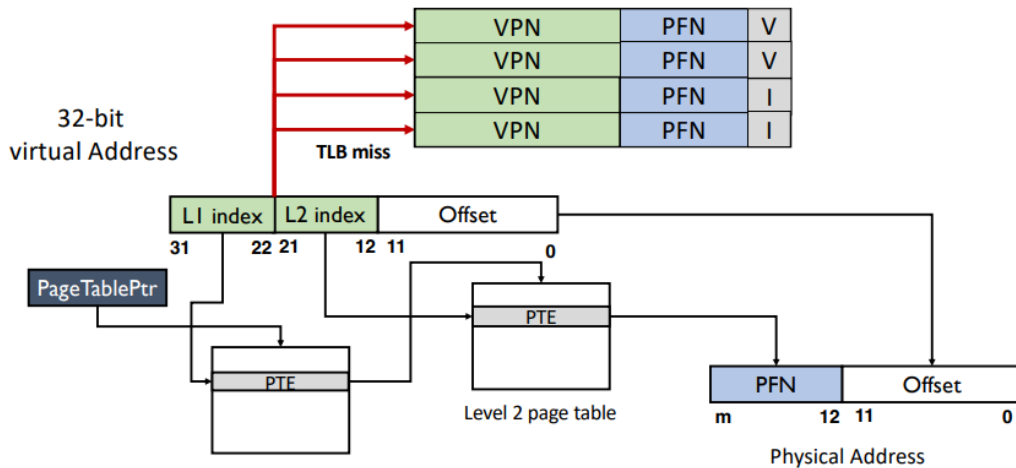
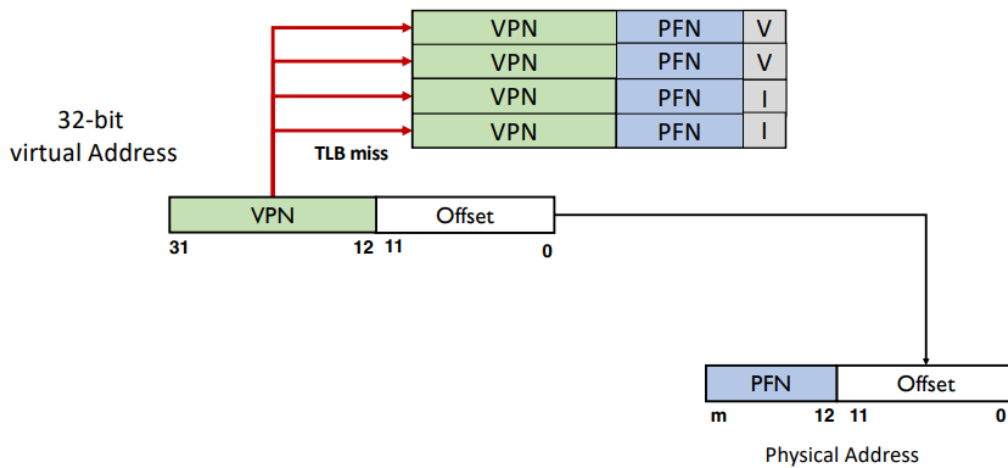
- Two descriptor tables: GDT & LDT
 - Six segment register to cache segment base addresses

5. Translation lookaside buffer (TLB)

Issue of paging

- Time complexity
 - Extra memory references during address translation
 - Three-level page tables requires 3 additional memory reads
 - If every memory reference needs 4 memory reads

TLB



- A translation lookaside buffer (TLB) is a **hardware cache** that is **part of the MMU**
 - A cache for the PTEs: holding a translation likely to be re-used
 - Replacement policy: LRU, FIFO, random
 - Each entry holds **mapping of a virtual address to a physical address**
- Before a virtual-to-physical address translation is to be performed, TLB is looked up using **VPN**
 - **TLB hit:** VPN is found, and the PFN of the same entry used
 - **TLB miss:** VPN not found, page table walk

Why Does TLB Work?

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- Ideally one page table walk for the entire page
 - TLB is smaller than page table, but
- **Spatial locality**
 - Sequentially executed instructions, local variables (on stack), arrays (on heap) likely on the same page
- **Temporal locality**
 - Accesses to the same page tend to be close in time

Issues with Context Switch

VPN	PFN	valid
-	-	I
100	110	V
-	-	I
100	170	V

- Two process may use the same virtual address
 - P1: 100 -> 110
 - P2: 100 -> 170

VPN	PFN	valid	ASID
-	-	I	-
100	110	V	1
-	-	I	-
100	170	V	2

- Solutions
 - **Flush TLB** upon context switch
 - Invalidate all entries: V -> I
 - **Extending TLB with address space ID**
 - No need to flush TLB

