Lecture 9: Paging: TLB and Multi-Level Page Tables

Operating Systems

Content taken from: https://www.cse.iitb.ac.in/~mythili/os/

https://pages.cs.wisc.edu/~remzi/OSTEP/

Last few classes

- MMU does translation from virtual address to physical address
- Dynamic relocation with base + bound registers
- Segmentation
- Paging
  - Given a virtual address, how to convert to physical address?
    - Extract VPN and offset from virtual address
    - Use VPN to index into the page table (using PTBR) and get PFN
    - Concatenate PFN and offset to get the physical address

Faster Translation Using TLBs

- Add some hardware support to MMU
  - Translation-lookaside buffer (TLB)
- TLB is simply a hardware cache of popular virtual-to-physical address translations
- Upon each virtual memory reference, the hardware first checks the TLB
- If the desired translation is held in TLB, the translation is performed (quickly) without having to consult the page table (which has all translations)

TLB Contents

- A typical TLB may have 32, 64 or 128 entries
- **Fully-associative:** Hardware searches the entire TLB in parallel to find the desired translation

VPN | PFN | other bits

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

Example: Accessing An Array

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

- Which of the memory accesses are going to be TLB miss and TLB hit?
  - a[0], a[3], a[7] will result in a TLB miss.
  - The rest of the array elements will result in a TLB hit.

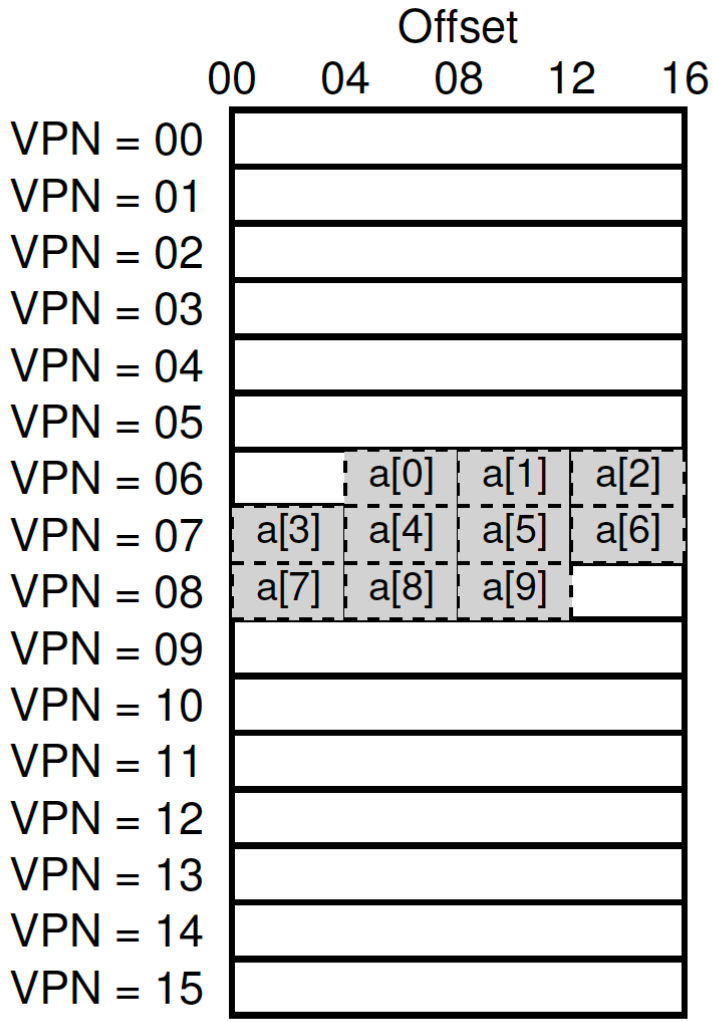


Figure 19.2: **Example: An Array In A Tiny Address Space**

Locality

- Spatial Locality
  - If a program accesses memory at address x, it will likely soon access memory near x.
- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.

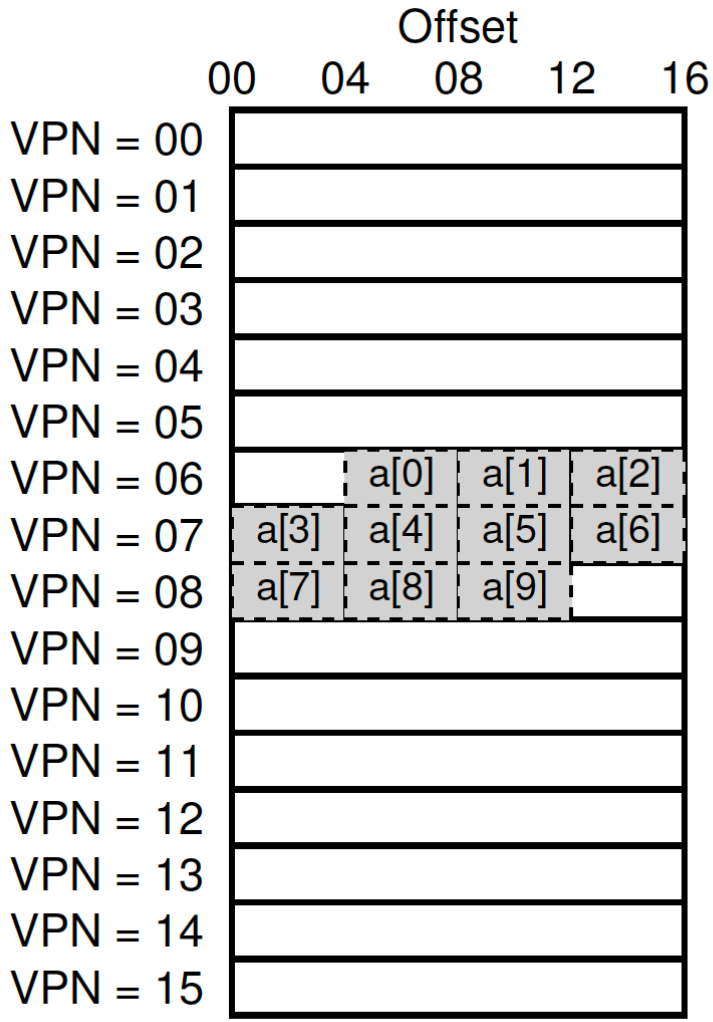


Figure 19.2: **Example: An Array In A Tiny Address Space**

Who handles the TLB Miss?

- **Hardware-managed TLBs**
  - Hardware handles the TLB miss entirely
  - Hardware should know where the page table is stored in memory (PTBR)
  - On a miss, the hardware finds the correct PTE, extracts the desired translation, updates the TLB with the translation and retry the instruction
- **Software-managed TLBs**
  - Hardware simply raises an exception / trap
  - Pause the current instruction stream and raise the privilege level to kernel mode
  - OS Jumps to trap handler code for handling TLB misses
  - Code will find the desired translation from page table, use it to update the TLB and return from trap
  - On TLB miss, after returning from trap, the PC should point to the instruction that caused the trap and not the one after it (which is the normal execution flow).
  - There is a chance that while executing the TLB-miss handling code, an infinite chain of TLB misses can occur.
    - TLB miss handlers could be kept in physical memory (no mapping is required).
    - Reserve entries in the TLB for permanent translation slots for the handler.

TLB Issue: Context Switches

- TLB contains virtual-to-physical translations that are only valid for the currently running process
- **Obvious solution:** simply flush the TLB on context switches, thus emptying it before running the next process
- What is the problem here?

Sharing TLB across context switches

- Add an address space identifier (ASID) field in TLB
- ASID is analogous to process identifier (PID)

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

TLB Entry Replacement Policy

- **Option 1:** Evict the **Least Recently Used (LRU)** entry
- **Option 2:** Evict any entry **randomly**

How big can page tables be?

- Let us take an example to understand
  - 32-bit virtual address space
  - 4KB page size (2^12)
  - 4-byte page-table entry
  - Each address location stores 1 byte of data
- For the above example,
  - How many unique addresses are possible? (2^32)
  - What is the total size of virtual memory? (2^32 * 1)
  - How many number of pages are required? (2^32/2^12 = 2^20)
  - What is the size of page-table? (2^20 * 4 = 2^22 = 4MB)
  - How many bits are required for identifying a VPN? (20 bits = #pages)
  - How many bits are required for the offset within a page? (32-20 = 12)

Can we reduce the page table size?

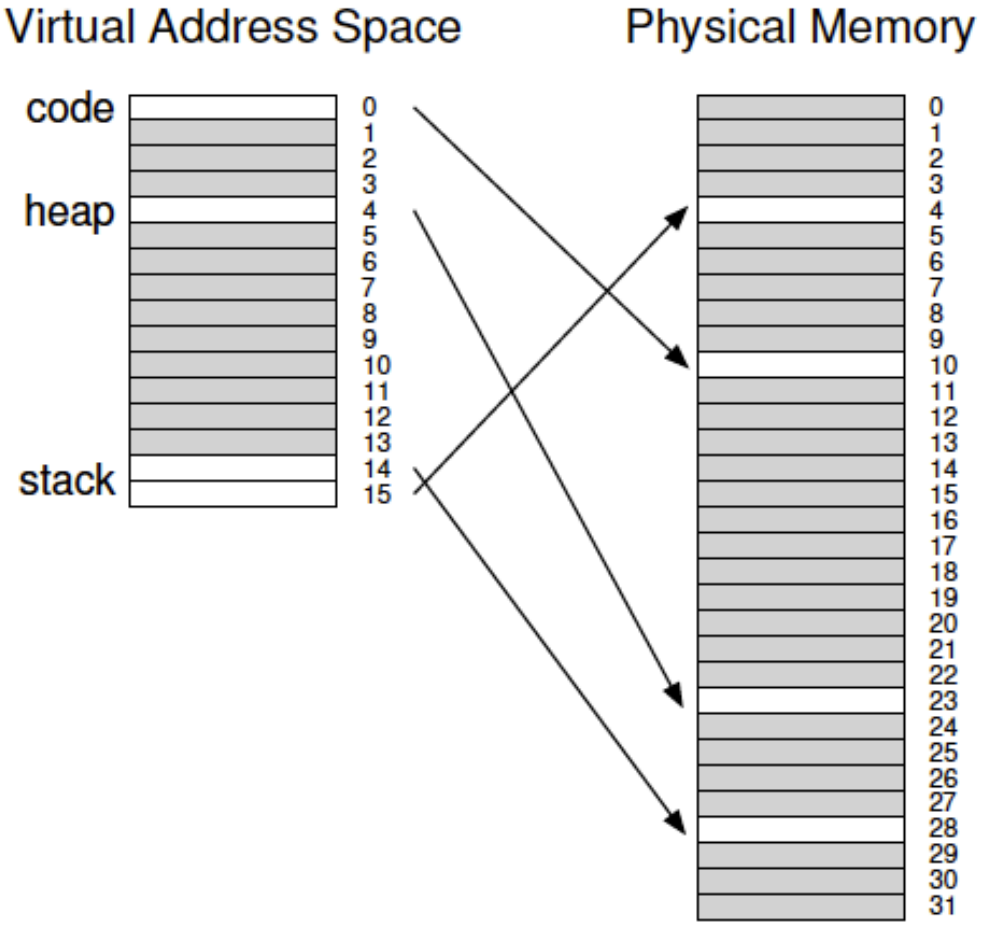


Figure 20.1: A 16KB Address Space With 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figure 20.2: A Page Table For 16KB Address Space

Multi-Level Page Tables

- Chop up the page table into page-sized units
- Don’t allocate any page of page-table entries if it is entirely invalid
- Use **page directory** to track which page of page-table is valid/invalid

Multi-Level Page Tables

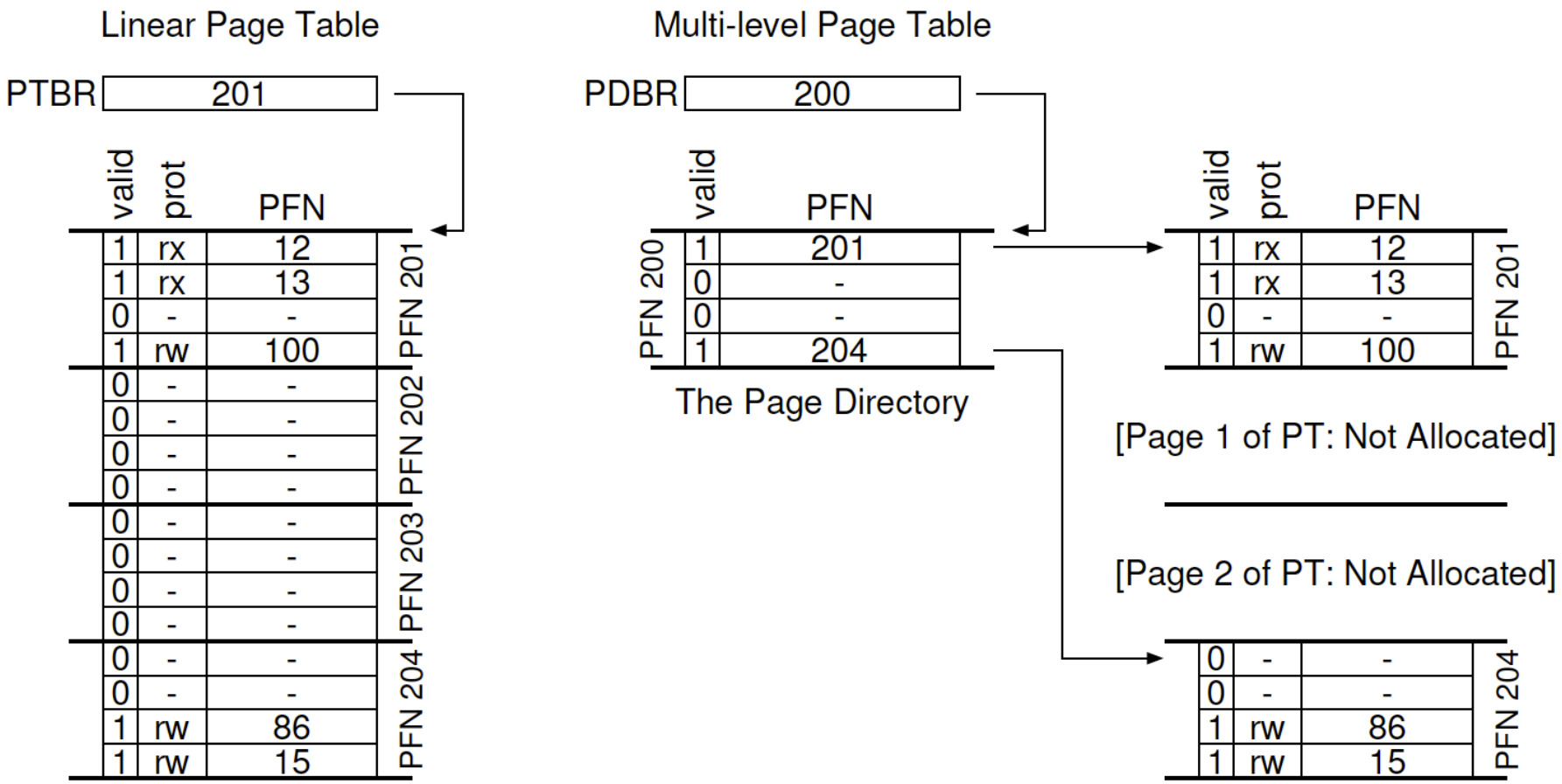


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

Multi-Level Page Tables

- **Advantages**
  - Allocates page-table space in proportion to the amount of address space used
  - Does not require the entire page table to reside contiguously in physical memory
- **Disadvantages**
  - On a TLB miss, two loads from memory will be required to get the PFN information for data
    - One for the page directory (PDE) and another for the PTE
  - More complex page-table lookup
- **Time-Space Tradeoff**
- Compare the above advantages and disadvantages with respect to linear page tables

Multi-Level Page Table: Example

- 16KB virtual address space
- 64-byte pages
- PTE is 4 bytes in size
- How many number of pages will be there in total? (16KB/64 = 2^14/2^6 = 2^8)

How many bits are required for fully addressing this address space? (14 bits)

- How many bits for VPN? (8 bits)
- How many bits for offset? (6 bits)

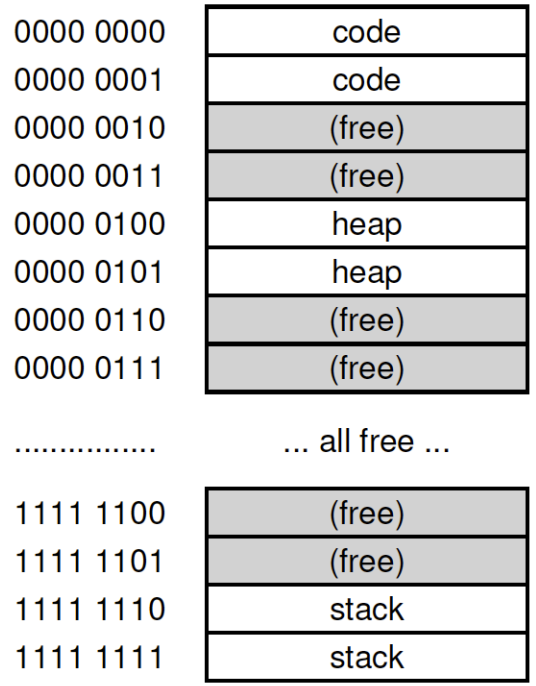


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

Multi-Level Page Table: Example

- What will be the number of entries in a linear page table? (2^8 = #pages)
- What is the size of linear page table? (2^8*4 = 2^10 bytes)
- How many pages will be there in the linear page table? (2^10/64 = 2^4)
- How many PTEs can be stored in one page? (64/4 = 16)
- How many entries will be there in the page directory? (2^4 = #pages in the page table)

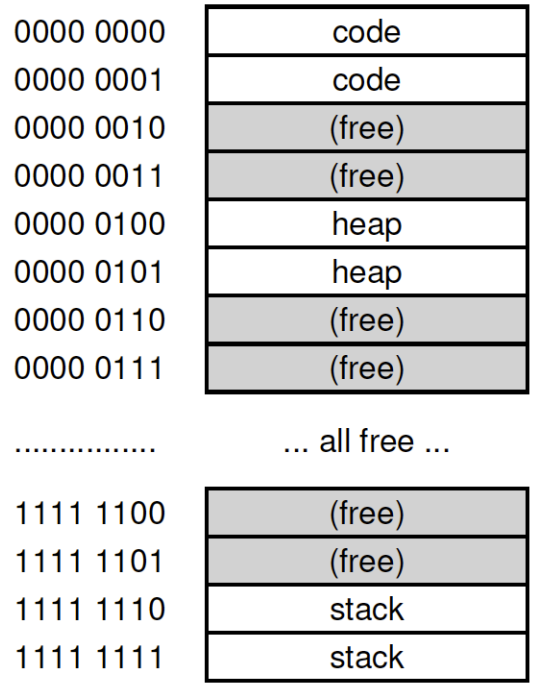
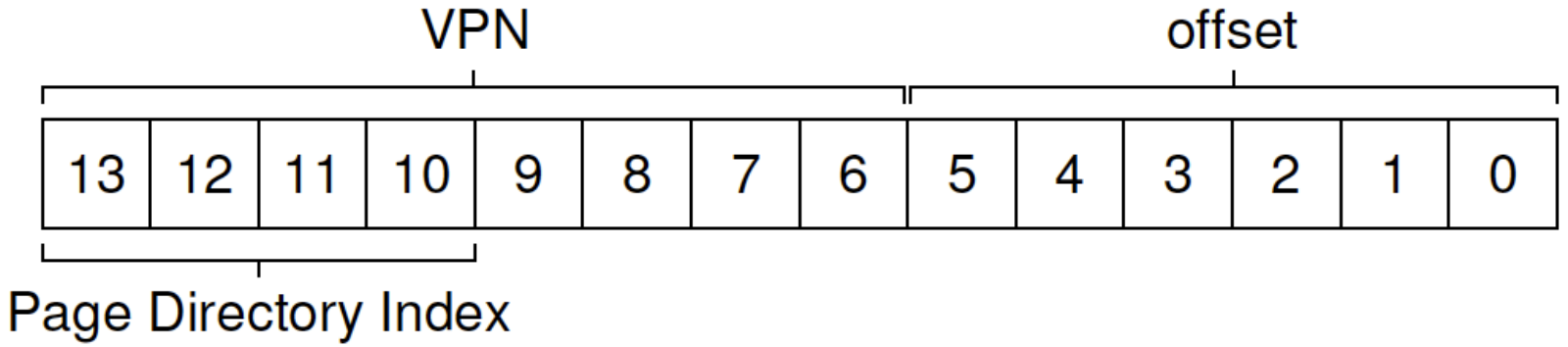


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

Multi-Level Page Table: Example

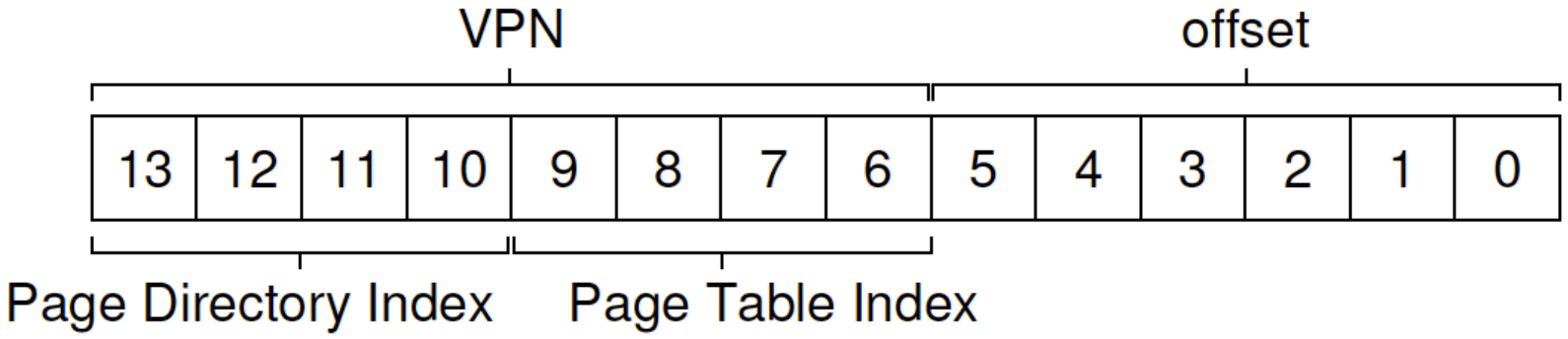
- What is the number of bits required to uniquely identify an entry in the page directory? Or what is the number of bits required for indexing into the page directory?



- PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))

Multi-Level Page Table: Example

- Once we have the PDEAddr, we can get the address of the required page of the page table.
- What is the number of bits required to uniquely identify an entry in a page of the page table? Or what is the number of bits required for indexing into the page of the page table?



- PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))

Multi-Level Page Table: Example

- Translate the virtual address **11 1111 1000 0000** to physical address.
  - Top 4 bits (1111) for PDIndex. Lookup entry 15. This is 101.
  - Next 4 bits (1110) is for PTIndex. Lookup entry 14. This is 55.

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

Figure 20.6: **Multi-level Page Table Control Flow**

Inverted Page Tables

- A single page table that has an entry for each physical page of the system
- Each entry tells
  - Which process is using this physical page?
  - Which virtual page of that process maps to this physical page?
- Given a VPN, how do we find the PFN?
  - Linear scan
  - Hash tables

Swapping the Page Tables to Disk

- Swap some of the page tables to disk when low on physical memory