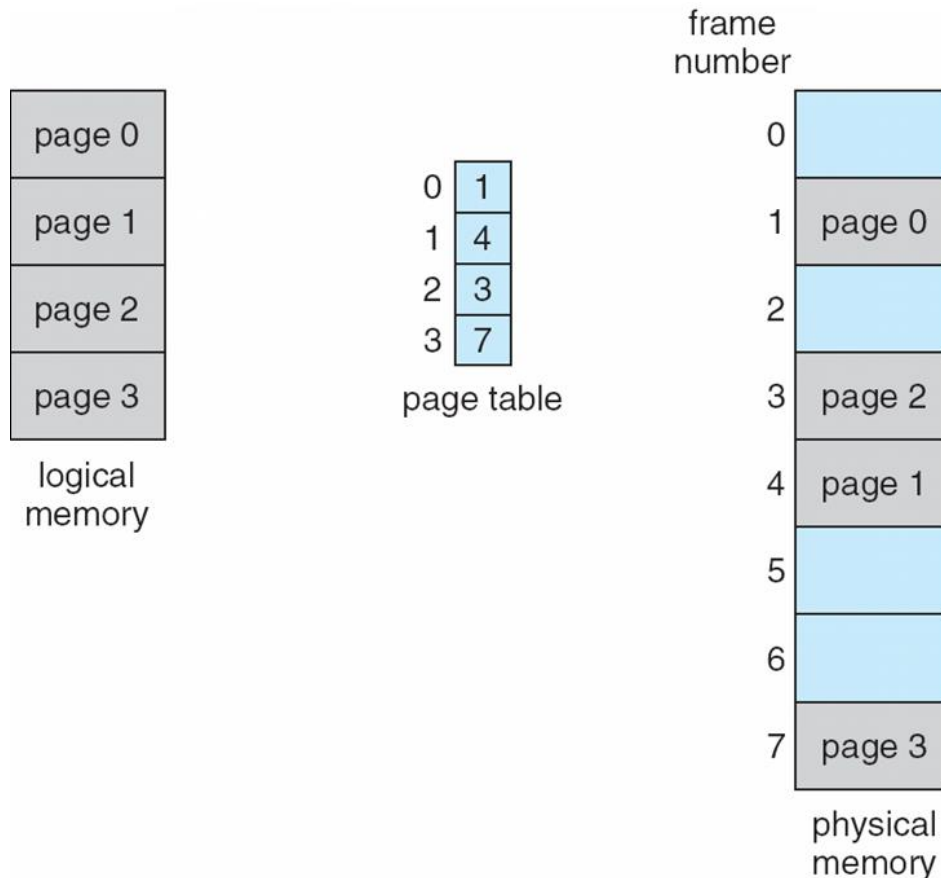

Paging

Chap 18, 19, 20

Paging

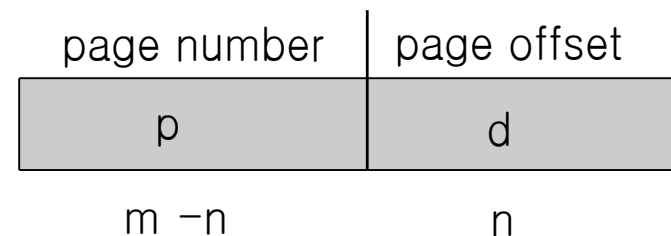
- Memory management scheme that permits the physical address space of a process to be noncontiguous
- Implemented by closely integrating the hardware and operating system



Paging

- **Basic method**

- Frames (page frames)
 - Breaking physical memory into fixed-size blocks
- Pages
 - Breaking logical memory into blocks of the same size
 - Page size
 - Typically power of 2
 - 512B ~ 16MB (typically 4KB ~ 8KB)
- Logical address
 - $v = (p, d)$
 - p: page number
 - d: page offset (displacement)



- logical address space 2^m
- page size 2^n

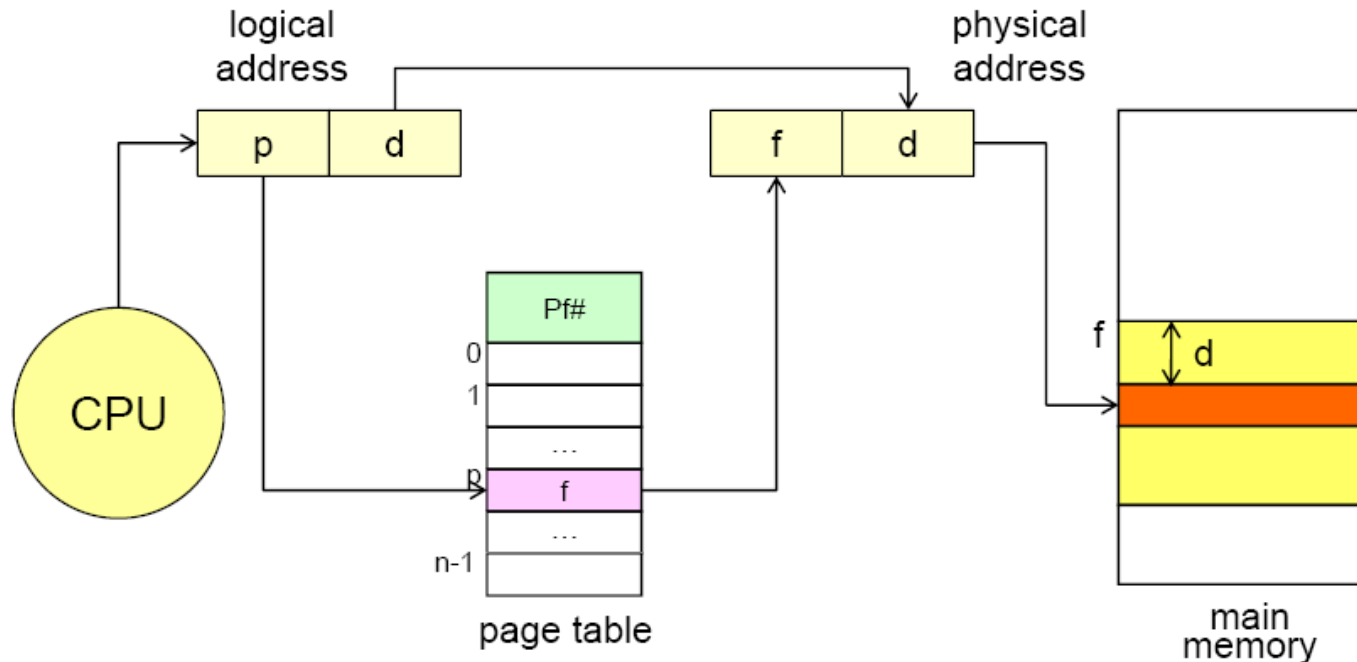
Paging

- **Basic method**
 - Address mapping
 - Logical address → physical address
 - Hidden from the user and controlled by the operating system
 - Uses page map table (PMT)
 - A page table for each process
 - PMT resides in kernel space of main memory

Paging

- **Basic method**

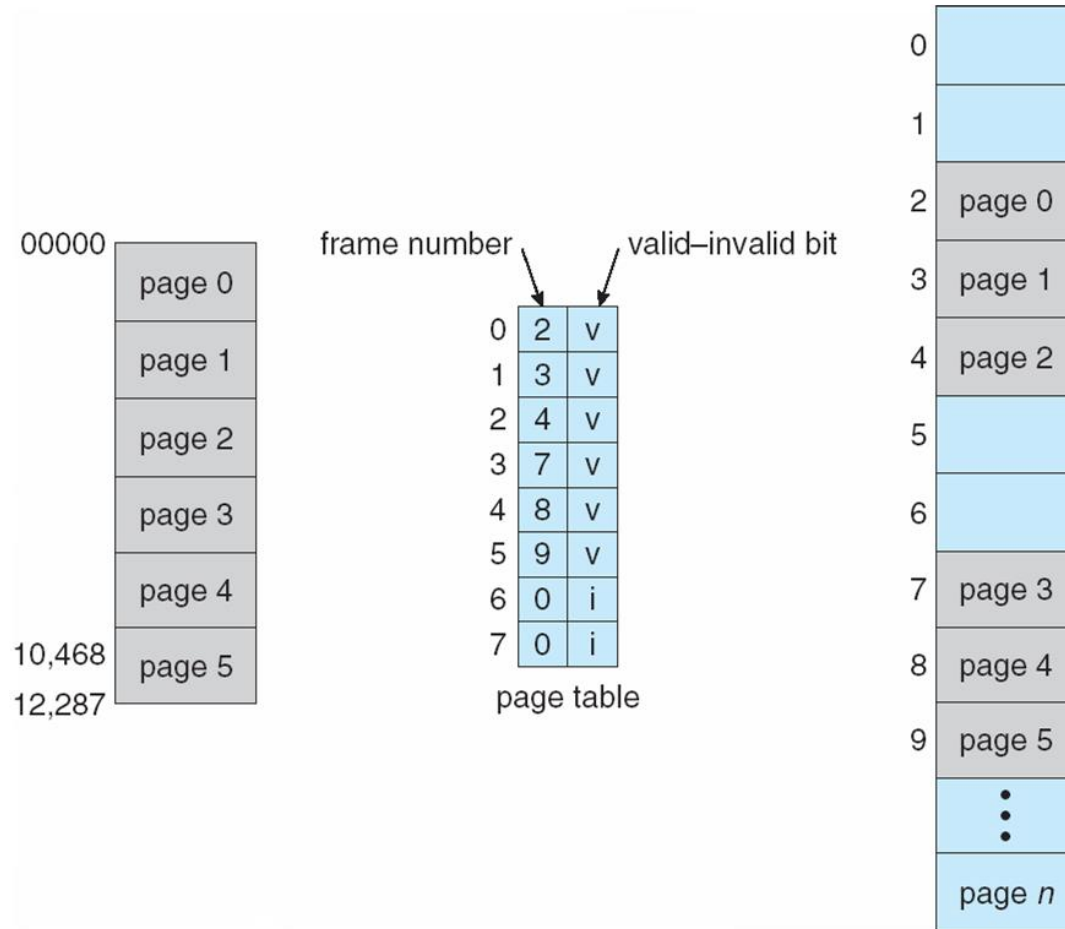
- Paging hardware and address mapping
- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table



Memory Protection

- **Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed**
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid bit attached to each entry in the page table:**
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
 - Or use **page-table length register (PTLR)**
- **Any violations result in a trap to the kernel**

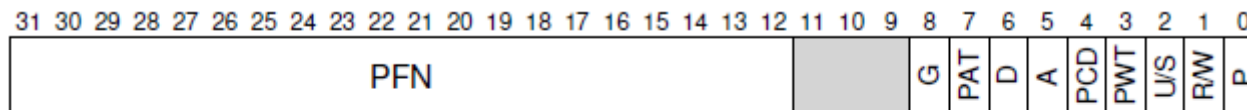
Valid (v) or Invalid (i) Bit In A Page Table



Page-table entry (PTE)

- **Valid bit**
 - indicate whether the particular translation is valid
 - All the unused space will be marked invalid, and if the process tries to access such memory, it will generate a trap
 - crucial for supporting a sparse address space
 - remove the need to allocate physical frames for unallocated pages and thus save a great deal of memory.
- **Protection bit (R/W, U/S)**
- **Present bit (P)**
 - Indicates whether this page is in physical memory or on disk
- **Dirty bit (D)**
 - Indicates whether the page has been modified
- **Reference bit (accessed bit) (A)**
 - used to track whether a page has been accessed

No valid bit?



PWT, PCD, PAT, and G determine how hardware caching works

Figure 18.5: An x86 Page Table Entry (PTE)

Page Table Location

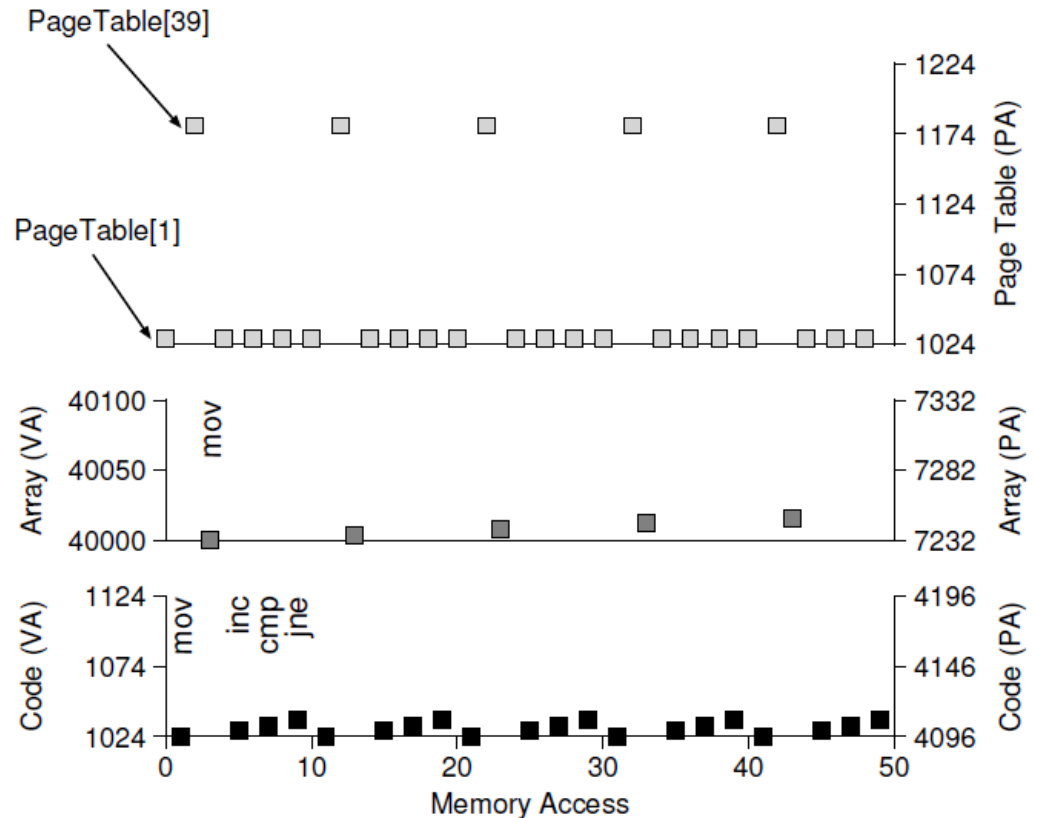
- **Direct mapping**
 - Doubles the number of memory accesses during execution of the program - One for the page table and one for the data/instruction
 - Performance degradation
- **TLB (Translation Look-aside Buffer)**
 - PMT: maintained in kernel space of the memory
 - Subset of PMT entries: maintained in small TLB
 - Entries of the recently used pages
 - dedicated registers or cache memories

Memory Trace of Direct mapping

```
int array[1000];  
...  
for (i = 0; i < 1000; i++)  
array[i] = 0;
```

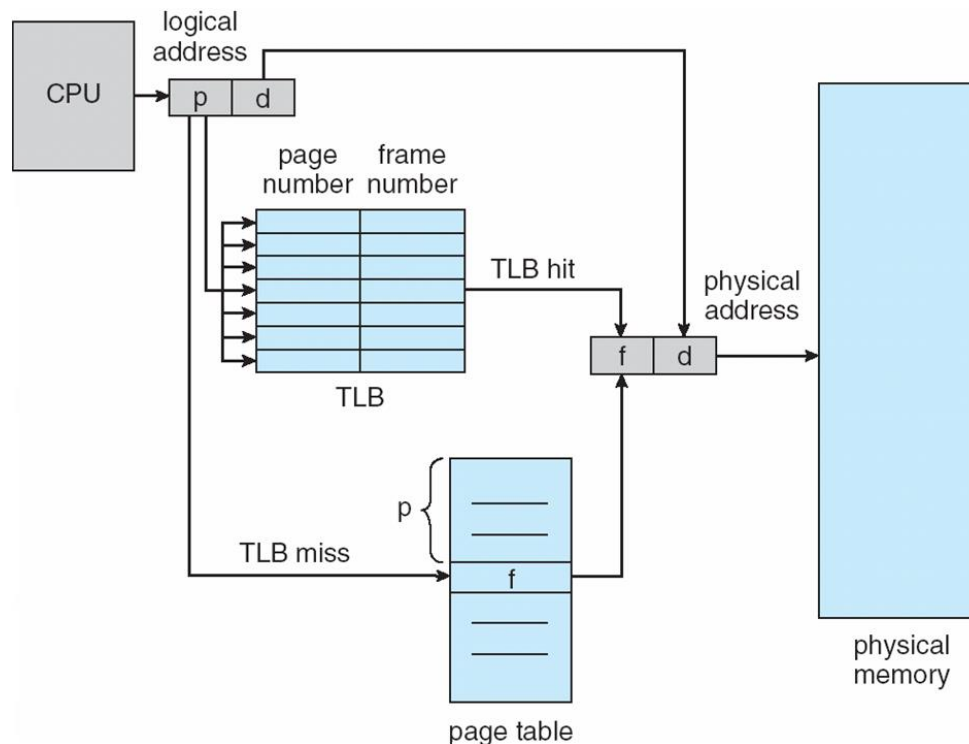
↓ compile

```
1024 movl $0x0,(%edi,%eax,4)  
1028 incl %eax  
1032 cmpl $0x03e8,%eax  
1036 jne 0x1024
```



TLB (Translation Look-aside Buffer)

- **Associative memory in MMU – parallel search**
- **Address translation (p, d)**
 - If p is in associative register, get frame # out
 - Otherwise get frame # from page table in memory
 - TLBs typically small (64 to 1,024 entries)



Effective Access Time

- TLB hit ratio $\alpha = 80\%$
- TLB search: 20 ns
- memory access: 100 ns
- $EAT = 0.80 \times (20+100) + 0.20 \times (20+100+100) = 140\text{ns}$
 - 40% slowdown in memory access time

Who Handles The TLB Miss?

```
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()
```

software-managed TLB (RISC)

On a TLB miss, hardware simply raises an exception

`RaiseException(TLB_MISS)`

Need to replace an old one (LRU, Random)

hardware-managed TLB (CISC)

Hardware has to know exactly where the page tables are located in memory (via PTBR), as well as their exact format; on a miss. Hardware “walks” the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction

Software-managed TLB

- When returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that **caused** the trap
 - This retry thus lets the instruction run again, this time resulting in a TLB hit.
- When running the TLB miss-handling code, the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur.
 - Keep TLB miss handlers in physical memory (where they are unmapped and not subject to address translation),
 - Reserve some entries in the TLB for permanently-valid translations
- Flexible
 - OS can use any data structure it wants to implement the page table, without necessitating hardware change.
- Simple
 - hardware doesn't have to do much on a miss; it raises an exception, and the OS TLB miss handler does the rest.

TLB Valid Bit \neq Page Table Valid Bit

- **TLB entry also has several status bits**
 - valid, protection, dirty, ...
- **Invalid in PTE**
 - The page has not been allocated by the process
 - The access to an invalid page causes a trap to the OS
- **Invalid in TLB**
 - a TLB entry has not a valid translation within it
 - When a system boots, a common initial state for each TLB entry is to be set to invalid, because no address translations are yet cached there.
 - Context switch will set all TLB entries to invalid

TLB Issue: Context Switches

- TLB contains virtual-to-physical translations that are only valid for the currently running process
- When switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.
- Solutions
 - Flush the TLB on context switches, sets all valid bits to 0
 - incur TLB misses when a new process runs → high cost
 - **Address space identifier (ASID) \approx (PID)**
 - enable sharing of the TLB across context switches

	VPN	PFN	valid	prot
P1 →	10	100	1	rwX
	—	—	0	—
P2 →	10	170	1	rwX
	—	—	0	—

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

Shared Pages

- **Shared code**

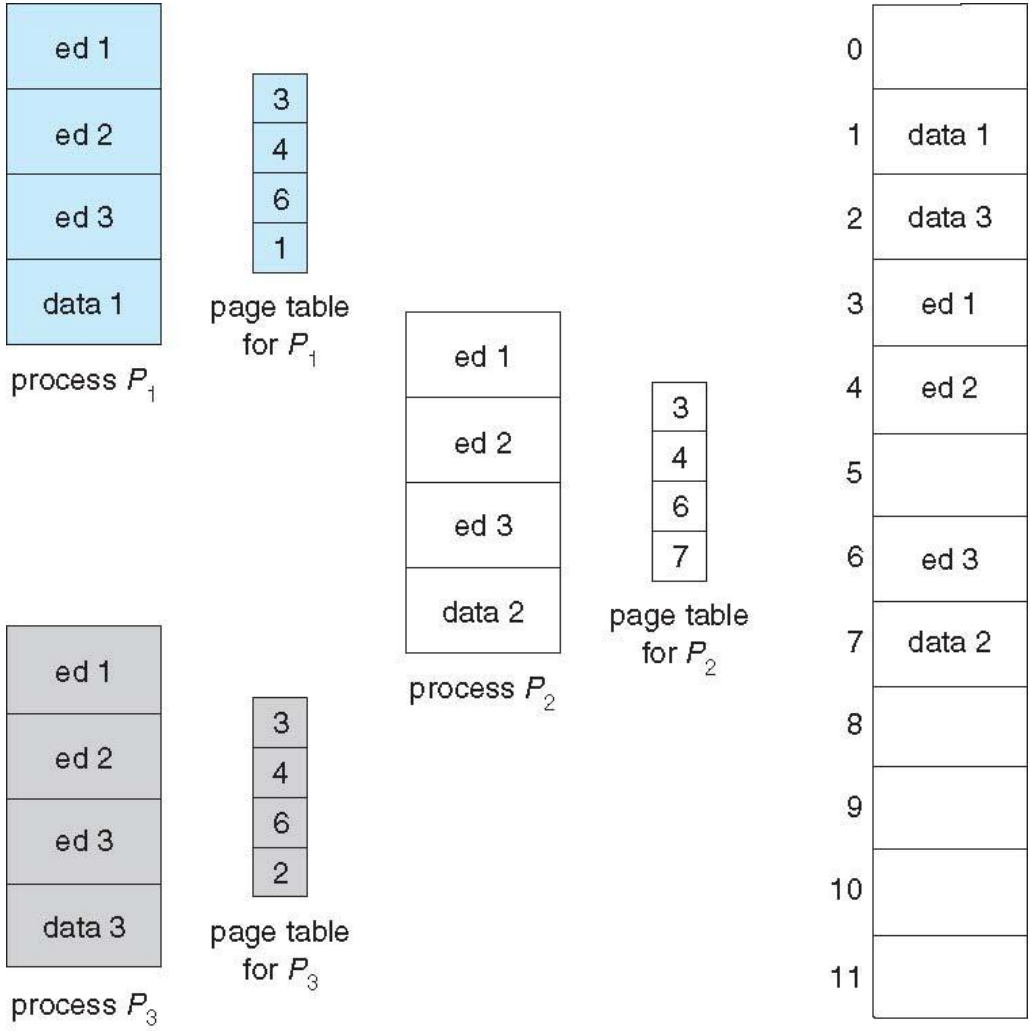
- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

- **Private code and data**

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

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

Shared Pages Example



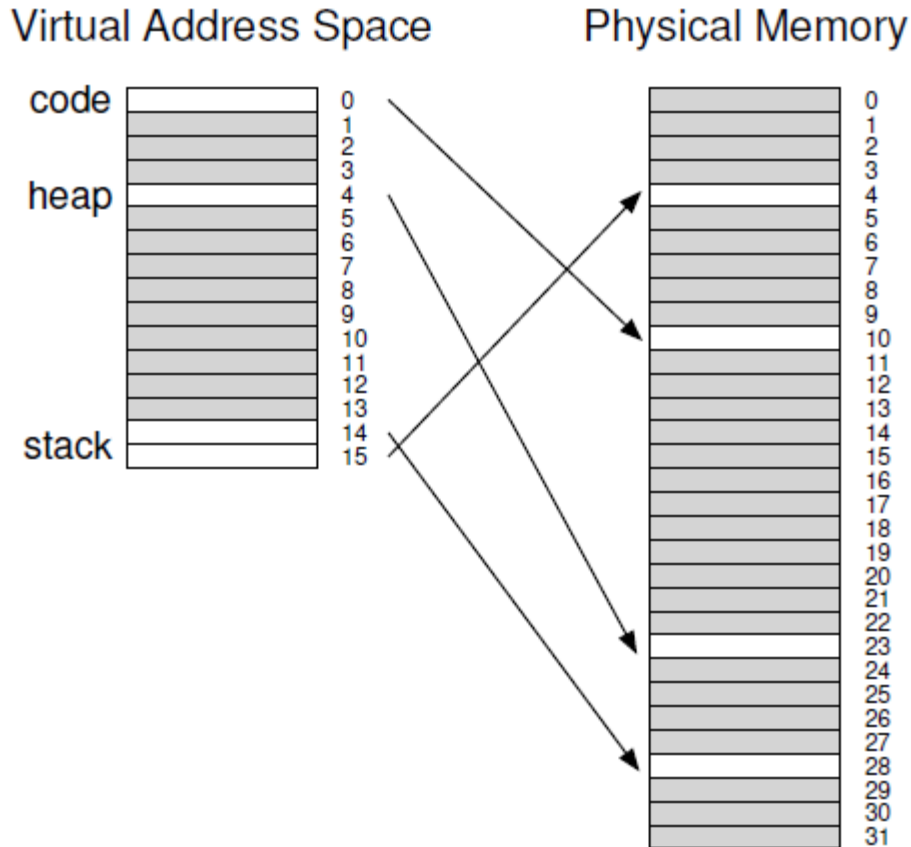
Paging: Smaller Tables

- **Memory structures for paging can get huge using straightforward methods**
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes -> **4 MB** of physical address space / memory for page table alone
 - That amount of memory used to cost a lot
 - Don't want to allocate that contiguously in main memory
- **Bigger Page**
- **Hybrid Approach: Paging and Segment**
- **Multi-level Page Table (Hierarchical Paging)**
- **Hashed Page Table**
- **Inverted Page Table**

Simple Solution: Bigger Pages

- **16KB pages (2^{14})**
 - $(2^{32} / 2^{14}) * 4B = 2^{20}B = 1MB$
 - Large TLB coverage
- **Internal fragmentation**
- Most systems use relatively small page sizes
 - 4KB (x86) or 8KB (SPARCV9)
- Many architectures now support multiple page sizes
 - MIPS, SPARC, x86-64

Sparse Page Table



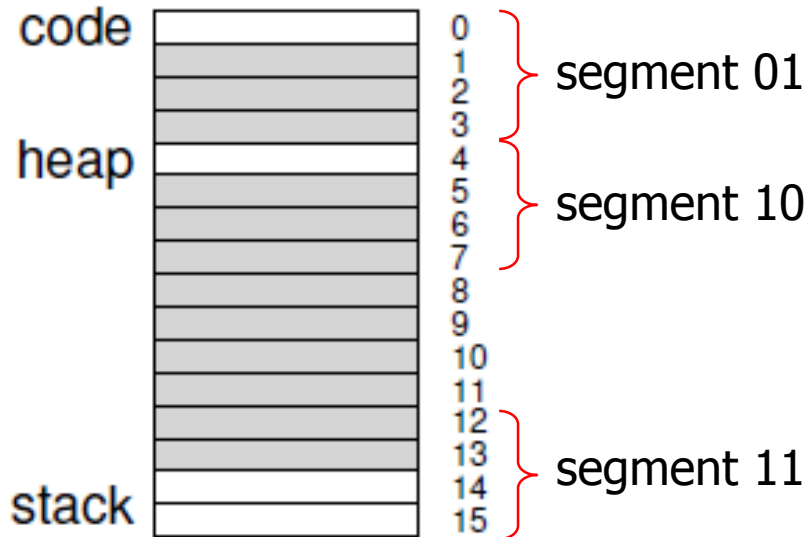
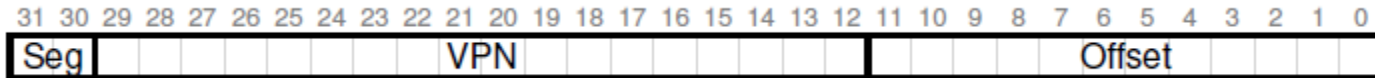
A 16KB Address Space With 1KB Pages

Most of the page table is unused,
full of invalid entries

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	---	-	-
-	0	---	-	-
-	0	---	-	-
-	0	---	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Page Table

Hybrid Approach: Paging and Segments

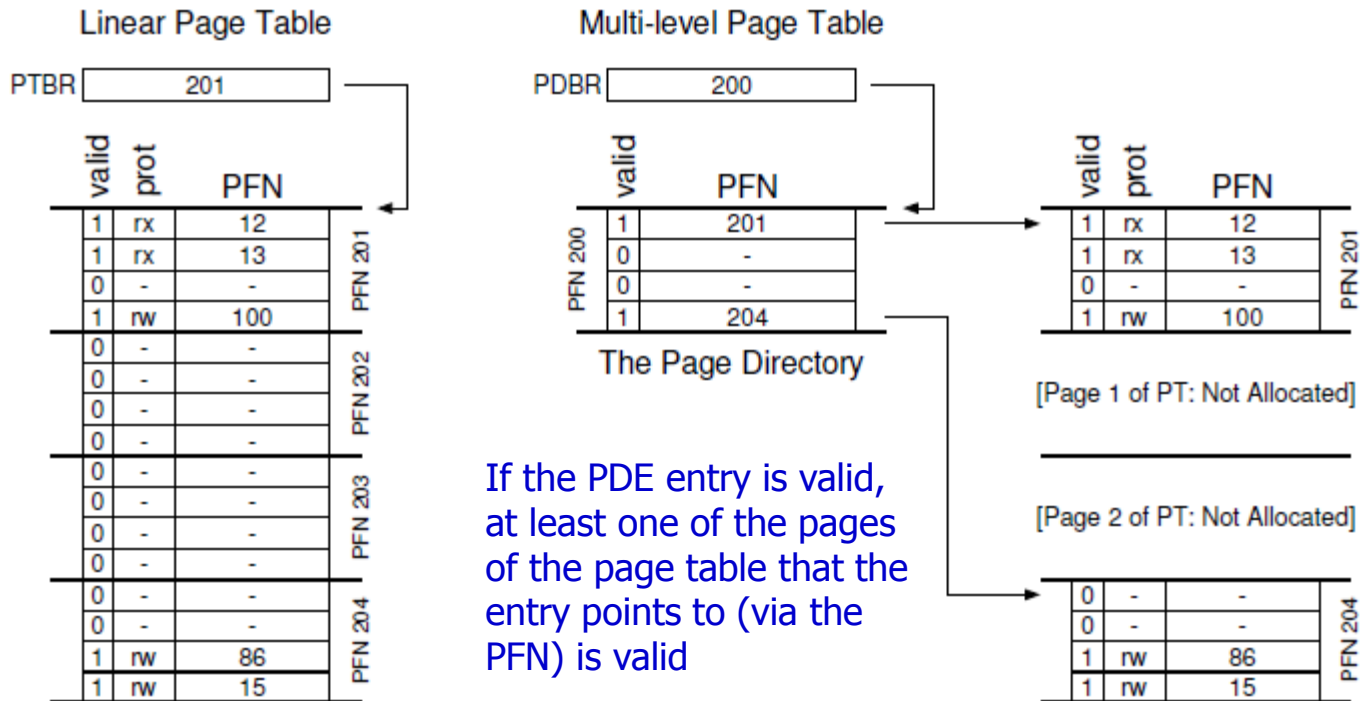


- The **base** register for each of these segments contains the physical address of a linear page table for that segment.
- Each process in the system now has **three** page tables associated with it.
- Page tables now can be of arbitrary size
- Unallocated pages between the stack and the heap no longer take up space in a page table

- Not flexible: if we have a large but sparsely-used heap, we can still end up with a lot of page table waste
- External fragmentation

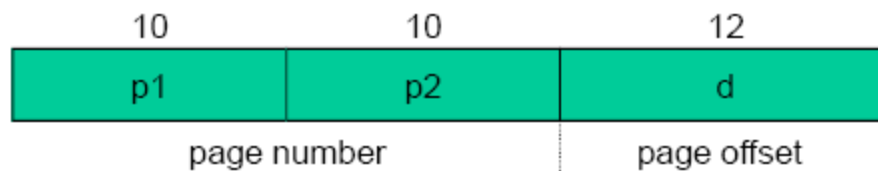
Multi-level (Hierarchical) Page Tables

- Break up the logical address space into multiple page tables
- Multi-level paging, in which the page table is also paged
- A simple technique is a two-level page table



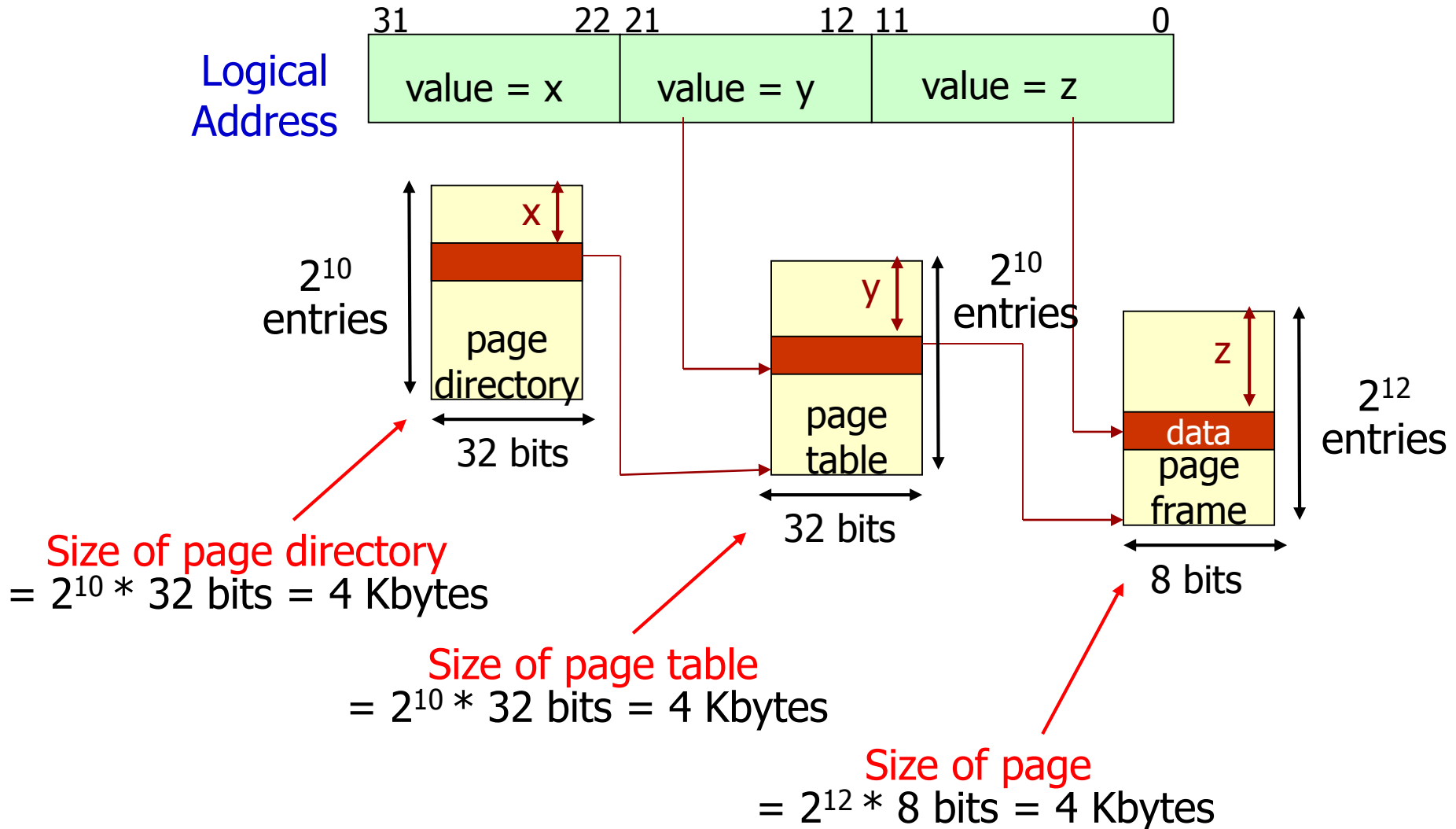
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits (4KB page)
- Since the page table is paged, the page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the page directory, and p_2 is the displacement within the page of the page table

Address-Translation Scheme



Two-Level Page Table

- **Small (typical) system**

- One page table might be enough
 - Page directory size + Page table size = 8 Kbytes of memory would suffice for virtual memory management
- How much *physical* memory could *this one page table* handle?
 - Number of page tables * Number of page table entries * Page size =
 $1 * 2^{10} * 2^{12}$ bytes = 4 Mbytes

- **Large system**

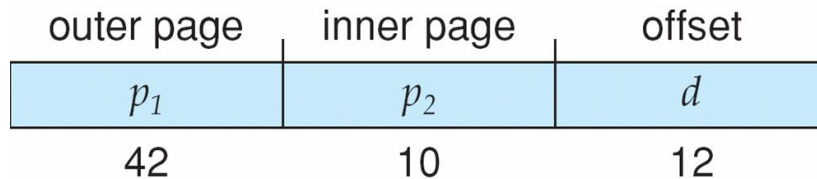
- You might need the maximum number of page tables
 - Max number of page tables * Page table size =
 2^{10} directory entries * 2^{12} bytes = 2^{22} bytes = 4 Mbytes of memory would be needed for virtual memory management
- How much *physical* memory could *these 2^{10} page tables* handle?
 - Number of page tables * Number of page table entries * Page size =
 $2^{10} * 2^{10} * 2^{12}$ bytes = 4 Gbytes

Pros/Cons of multi-level page tables

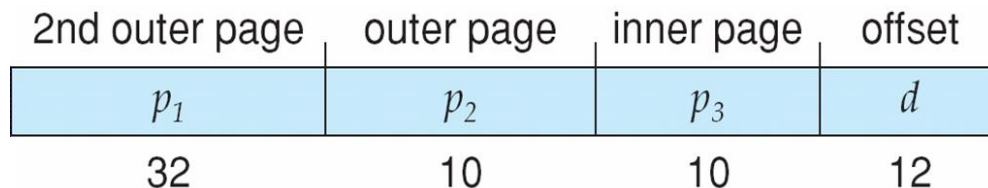
- Only allocates page-table space in proportion to the amount of address space you are using
 - compact and supports sparse address spaces
- if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory
 - For a large page table (say 4MB), finding such a large chunk of unused contiguous free physical memory can be quite a challenge.
- On a TLB miss, two loads from memory will be required to get the right translation information from the page table
 - one for the page directory, and one for the PTE itself
 - time-space trade-off
- Complex

64-bit Logical Address Space

- **Even two-level paging scheme not sufficient**
- **If page size is 4 KB (2^{12})**
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries, and outer page table has 2^{42} entries or 2^{44} bytes

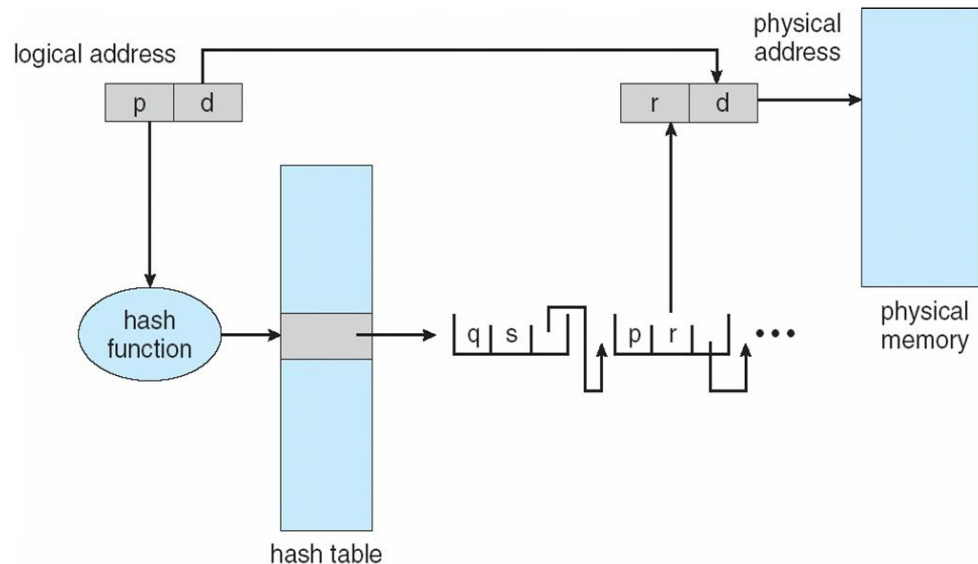


- One solution is to add a 2nd outer page table
- But 2nd outer page table is still 2^{34} bytes in size
 - And possibly 4 memory access to get to one physical memory location



Hashed Page Tables

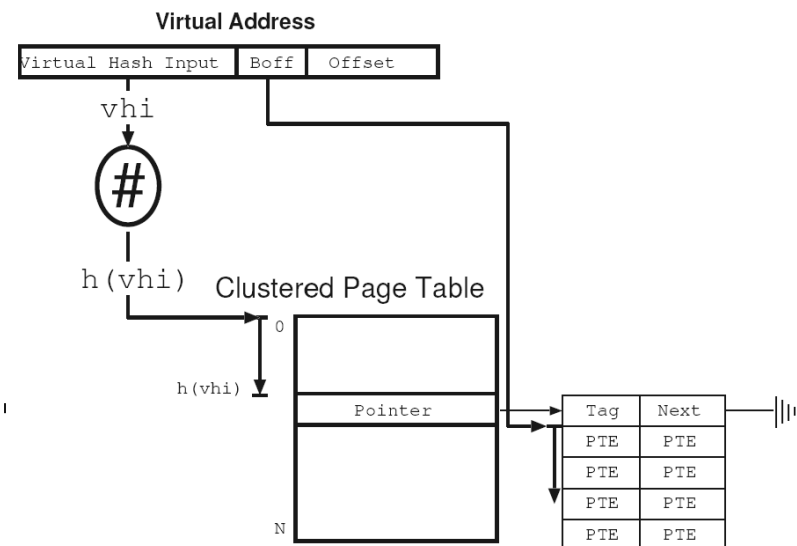
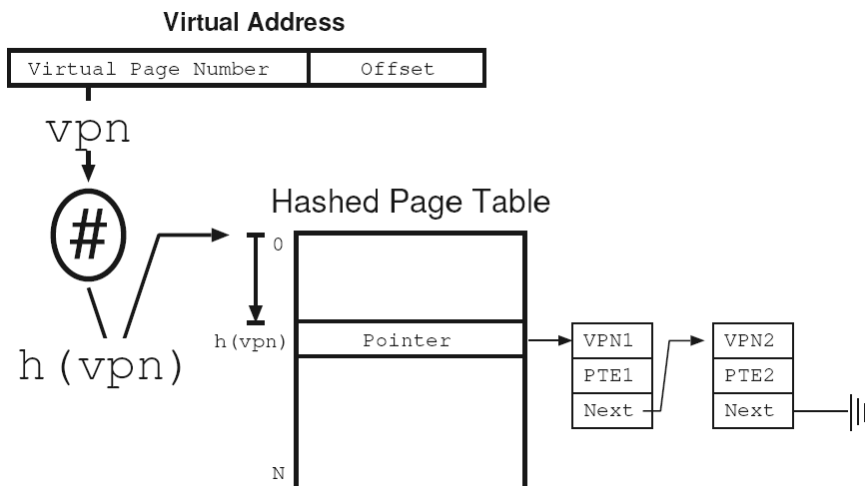
- Common approach for handling address spaces larger than 32 bits
- Logical page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains
 - (1) the logical page number
 - (2) the value of the mapped page frame
 - (3) a pointer to the next element
- Logical page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted



Hashed Page Table

- **Clustered page table**

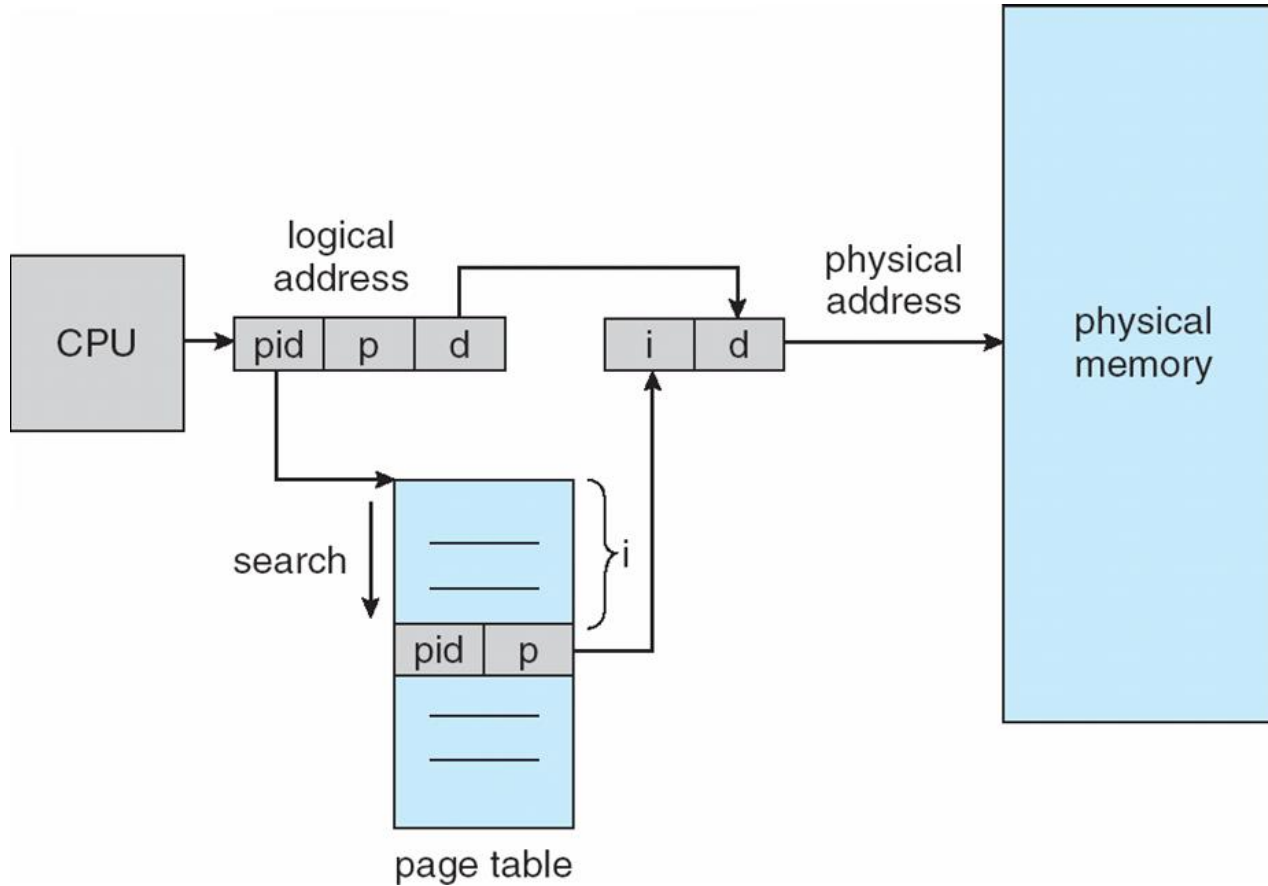
- Variation for 64-bit addresses
- Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Single page table entry can store the mappings for several physical page frames
- Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



Inverted Page Table

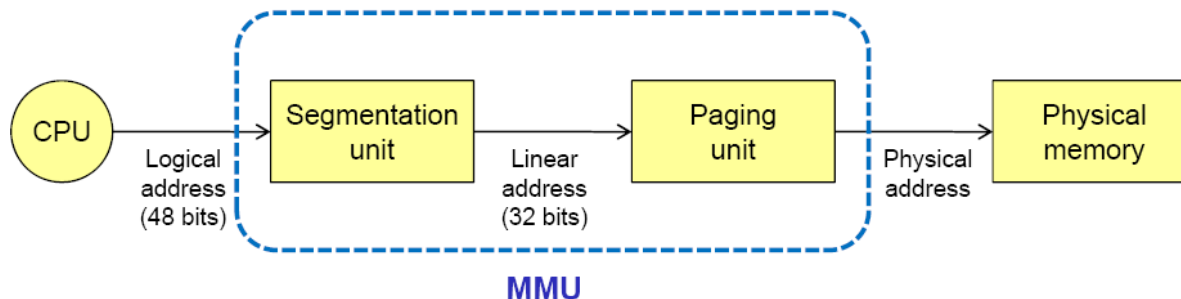
- PowerPC
- One entry for each real page of memory
 - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
 - Decreases memory needed to store each page table,
 - Increases time needed to search the table when a page reference occurs
 - Use hash table to limit the search to one or at most a few page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Intel IA-32 Architecture

- **Supports both segmentation and segmentation with paging**
 - Each segment can be 4 GB
 - Up to 16K (2^{14}) segments per process
 - Divided into two partitions
 - 1st partition: 8K (2^{13}) private segments
 - kept in **local descriptor table (LDT)**
 - 2nd partition: 8K (2^{13}) shared segments
 - kept in **global descriptor table (GDT)**
 - Each entry of LDT and GDT
 - 8-byte segment descriptor
 - Detailed information on a particular segment
 - Base location, limit of the segment, etc



Intel IA-32 Architecture

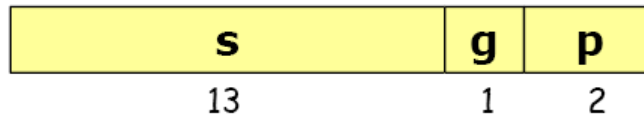
- **CPU generates 48 bit logical address (selector, offset)**

- Selector (s, g, p): 16 bits

- s: segment number (13 bits)

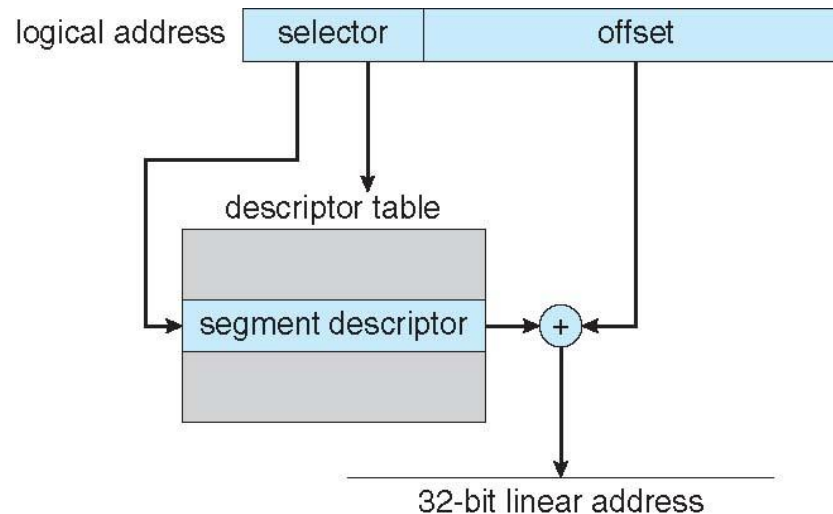
- g: whether the segment is in the LDT or GDT (1 bit)

- p: deals with protection (2 bits)



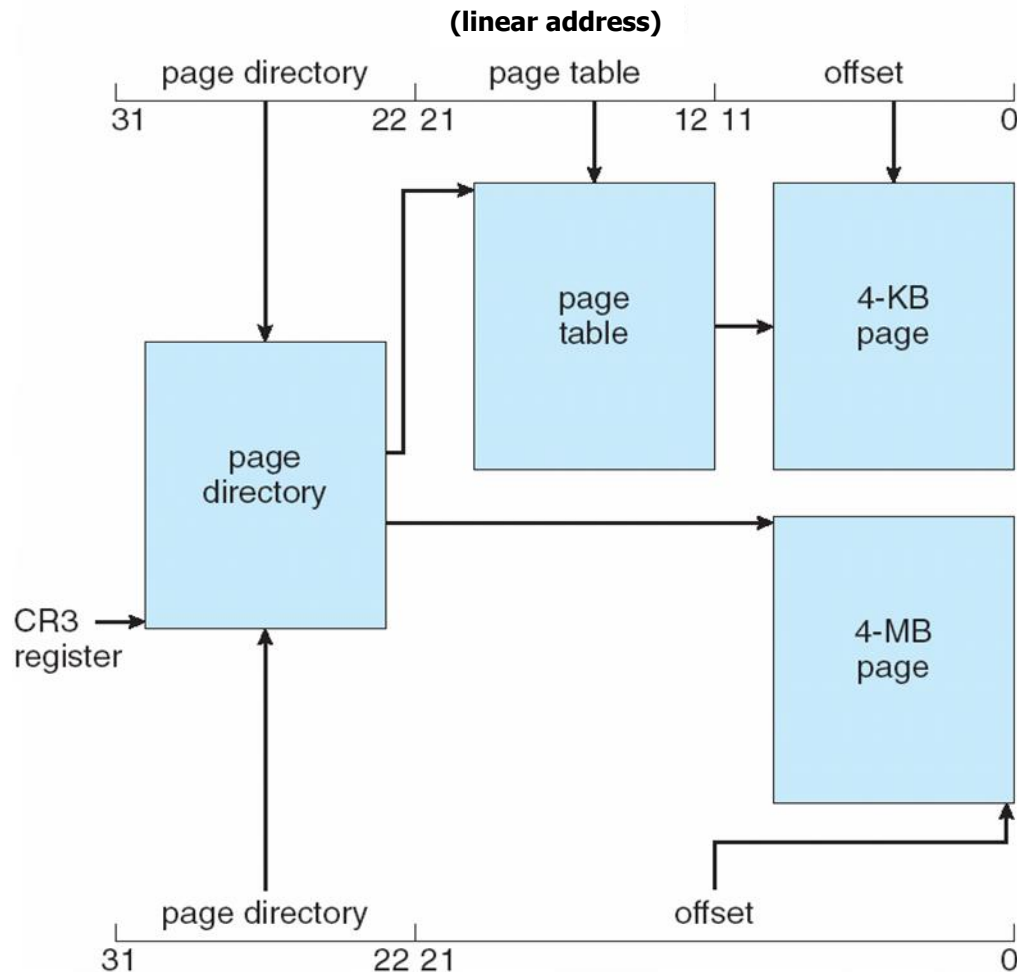
- Offset

- 32-bit number (location within the segment)



Intel IA-32 Paging Architecture

- Page size: 4KB or 4MB
- CR3 points to the page directory for the current process



Intel x86-64

- **Current generation Intel x86 architecture**
- **64 bits is ginormous (> 16 exabytes)**
- **In practice only implement 48 bit addressing**
 - Page sizes of 4 KB, 2 MB, 1 GB
 - Four levels of paging hierarchy
- **Can also use PAE (Page Address Extension) so virtual addresses are 48 bits and physical addresses are 52 bits**

