

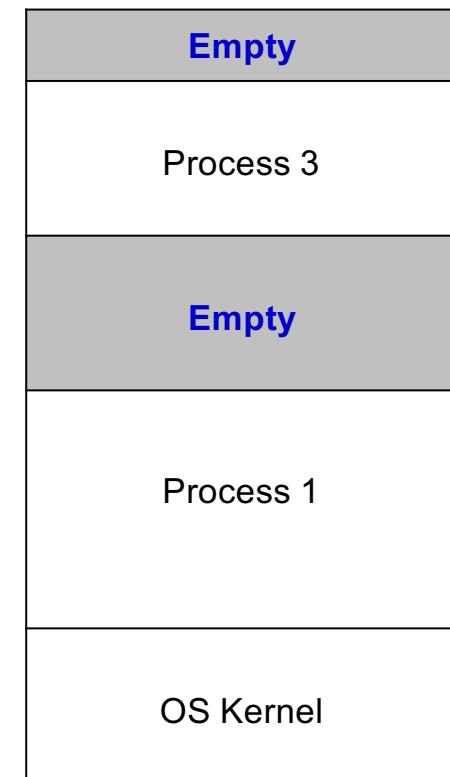
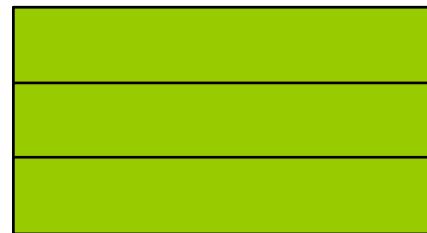
Paging



Paging: basic idea

Contiguous → Noncontiguous

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
 - Fixed and variable partitions are physical **contiguous** allocation.
 - Avoids **external fragmentation**
 - Avoids problem of varying sized memory chunks



Paging

Basic methods

- Divide **physical** address into fixed-sized blocks called **frames**
 - Size is power of 2, usually 4KB
- Divide **logical** address into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size **N** pages, need to find **N** free frames and load program
- Set up a **page table** to translate logical to physical addresses

Paging: Internal Fragmentation

- Paging has **no external fragmentation**, but **internal fragmentation**
 - e.g., page size: 2,048, program size: 72,766 (35 pages + 1,086 bytes)
 - internal fragmentation: $2,048 - 1,086 = 962$
 - **worst case internal fragmentation: 1 frame – 1 byte**
 - **average internal fragmentation: $1 / 2$ frame size**
 - In practice, average is much smaller
 - Program contains several pages, only last page has fragmentation
- Small frame sizes more desirable than large frame size?
 - Small size means **less** internal fragmentation, but **more** page table entries
 - Large size means **more** ..., but **less** ...
 - page size is usually 4KB, can be 64KB, block can be 2MB or 1GB (64-bit)

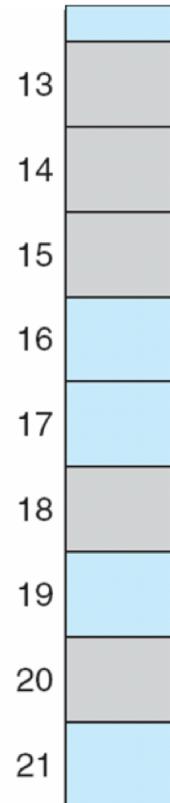
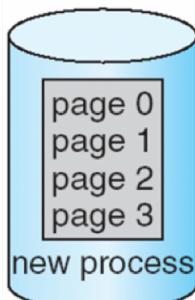
Page Table

- Page table
 - Stores the logical page to physical frame mapping
- Frame table
 - OS is managing physical memory, it should be aware of the allocation details of physical memory (frame)
 - Which frame is free, and how many frames have been allocated ...
 - One entry for each physical frame
 - the allocated frame belongs to which process

Paging example

free-frame list

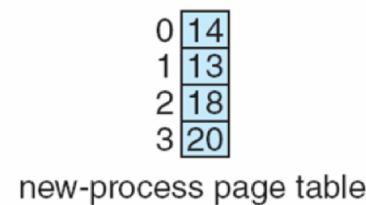
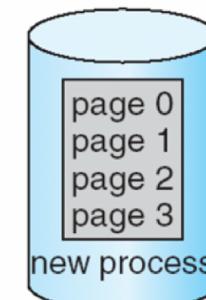
14
13
18
20
15



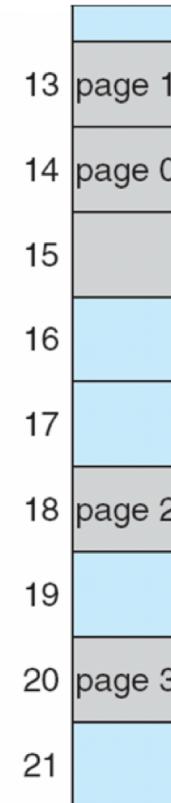
(a)

free-frame list

15



new-process page table



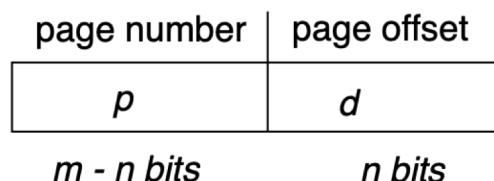
(b)

Before allocation

After allocation

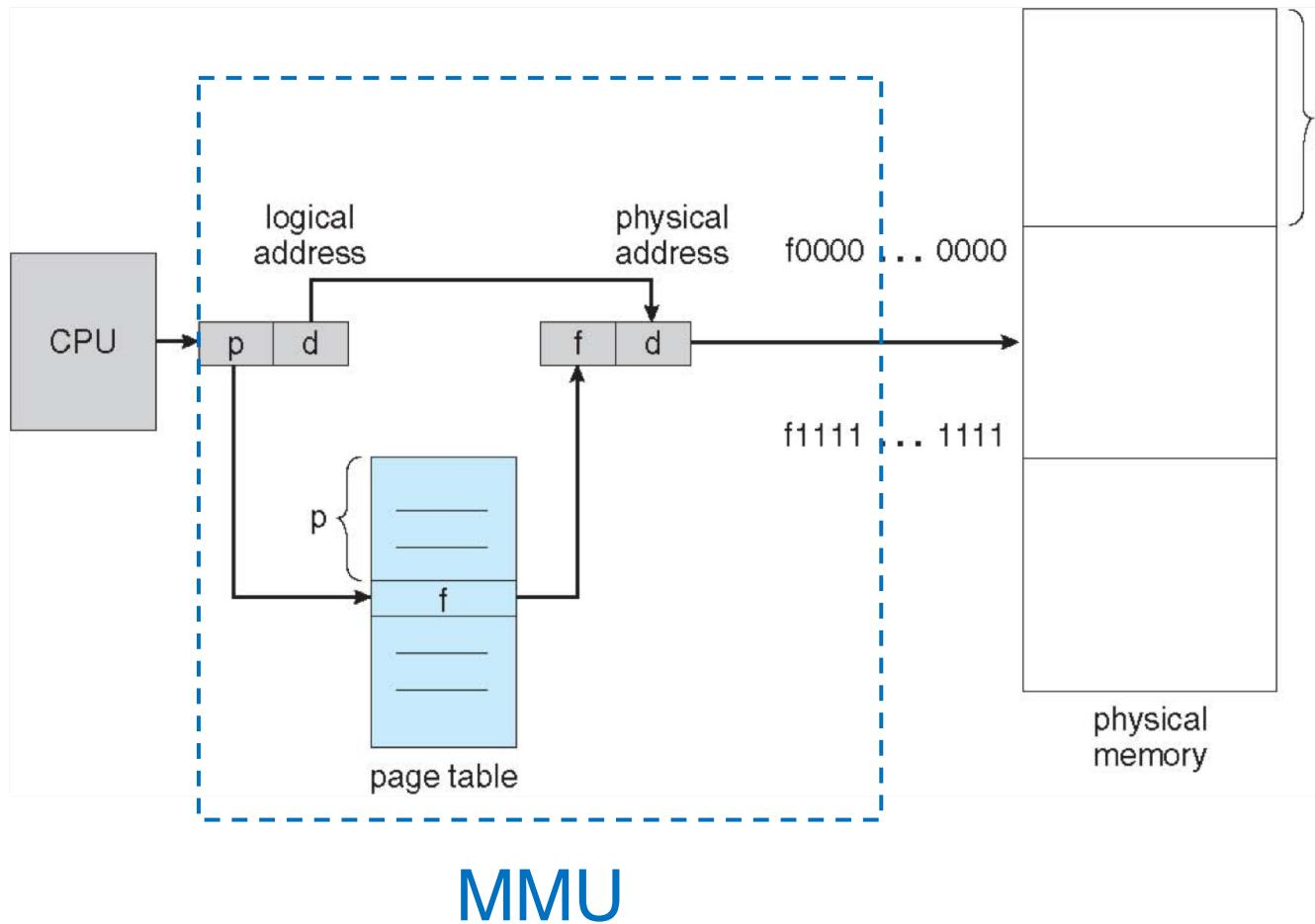
Paging: Address Translation

- A logical address is divided into:
 - **page number (p)**
 - used as an index into a page table
 - page table entry contains the corresponding **physical frame number**
 - **page offset (d)**
 - offset within the page/frame
 - combined with frame number to get the physical address

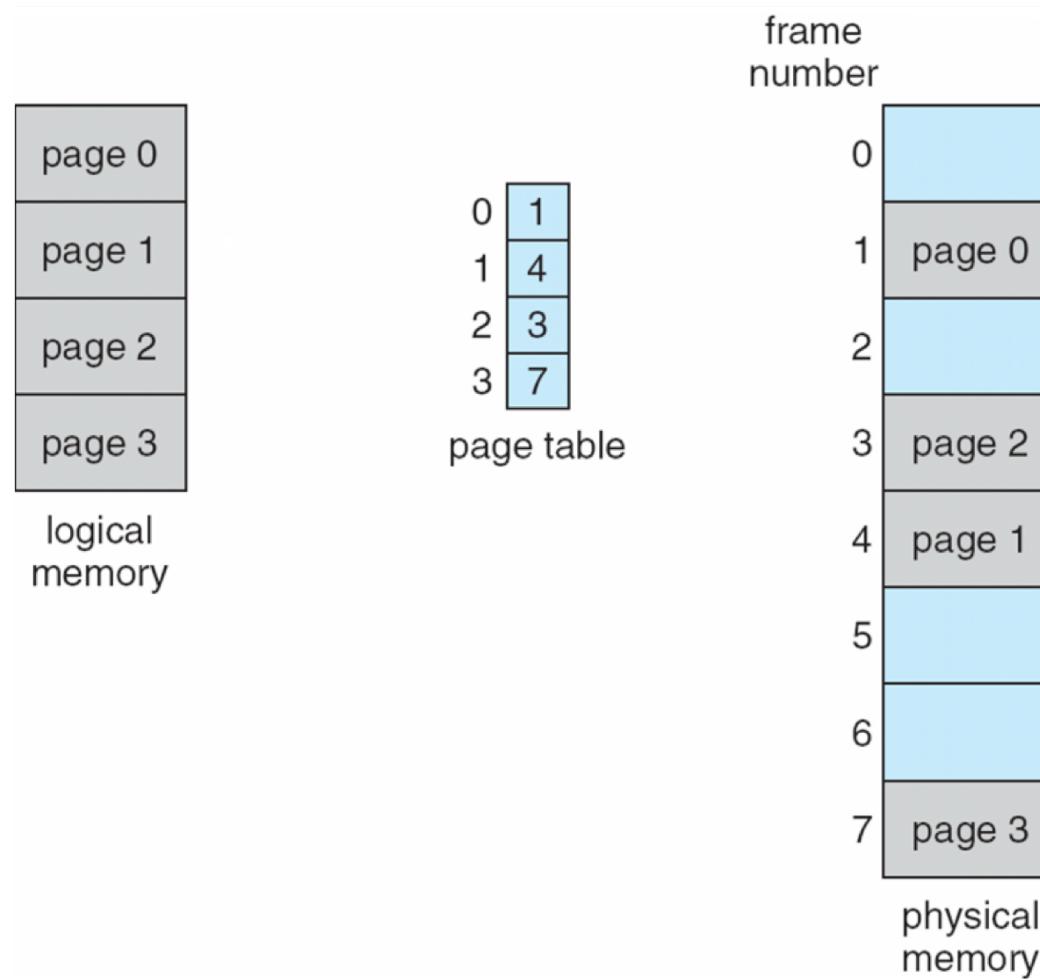


m bit logical address space, n bit page size

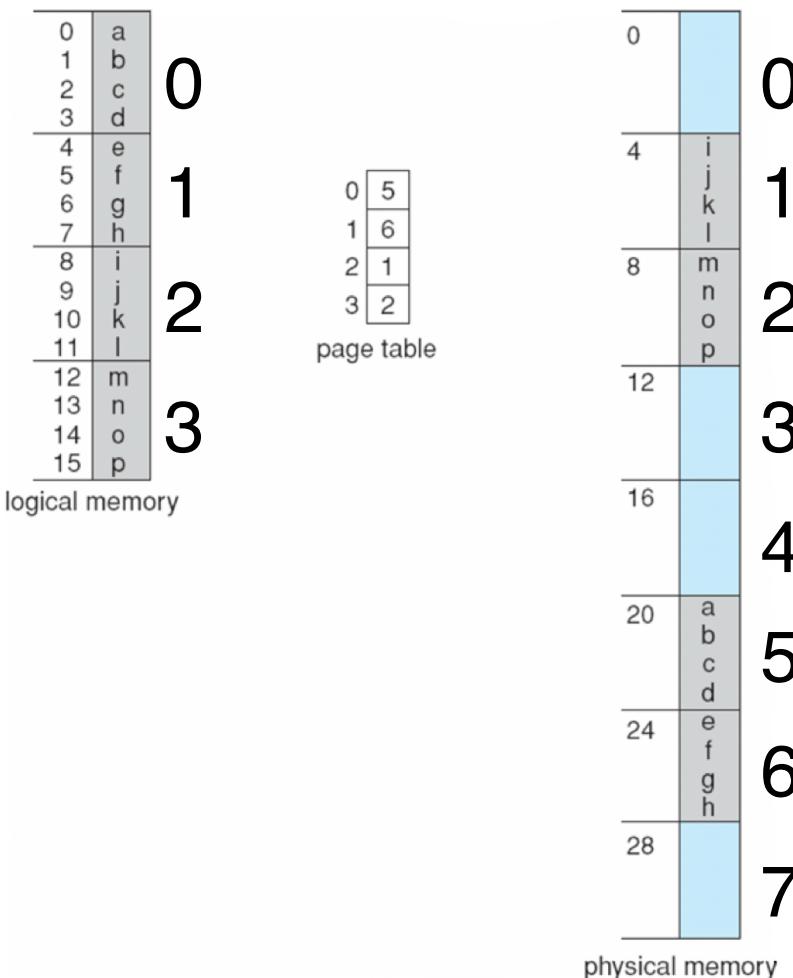
Paging Hardware



Paging Example



Paging Example II



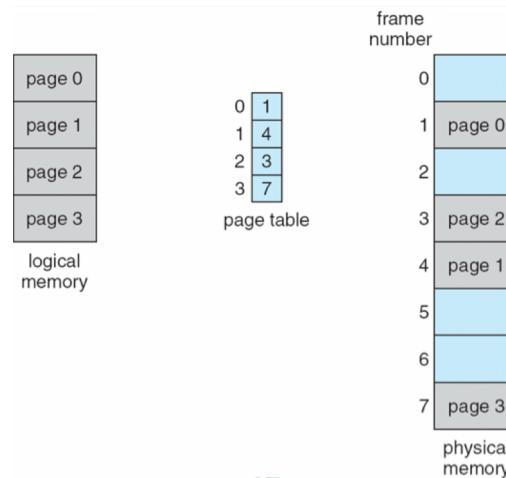
$m = 4$ and $n = 2$ 32-byte memory and 4-byte pages

Page Table Hardware Support: Simplest Case

- Page table is in a set of dedicated registers
 - Advantages: very efficient - access to register is fast
 - Disadvantages:
 - register number is limited → the table size is very small
 - the context switch need to save and restore these registers

Page Table Hardware Support: Alternative Way

- One big page table maps logical address to physical address
 - the page table should be **kept in main memory**
 - **page-table base register (PTBR)** points to the page table
 - does PTBR contain physical or logical address?
 - **page-table length register (PTLR)** indicates the size of the page table
- Every data/instruction access requires **two memory accesses**
 - one for the page table and one for the data / instruction
 - CPU can cache the translation to avoid one memory access (**TLB**)



TLB

- TLB (**translation look-aside buffer**) caches the address translation
 - if page number is in the TLB, no need to access the page table
 - if page number is not in the TLB, need to replace one TLB entry
 - TLB usually use a fast-lookup hardware cache called **associative memory**
 - TLB is usually small, 64 to 1024 entries
- Use with page table
 - TLB contains a few page table entries
 - Check whether page number is in TLB
 - If → frame number is available and used
 - If not → **TLB miss**. access page table and then fetch into TLB

TLB

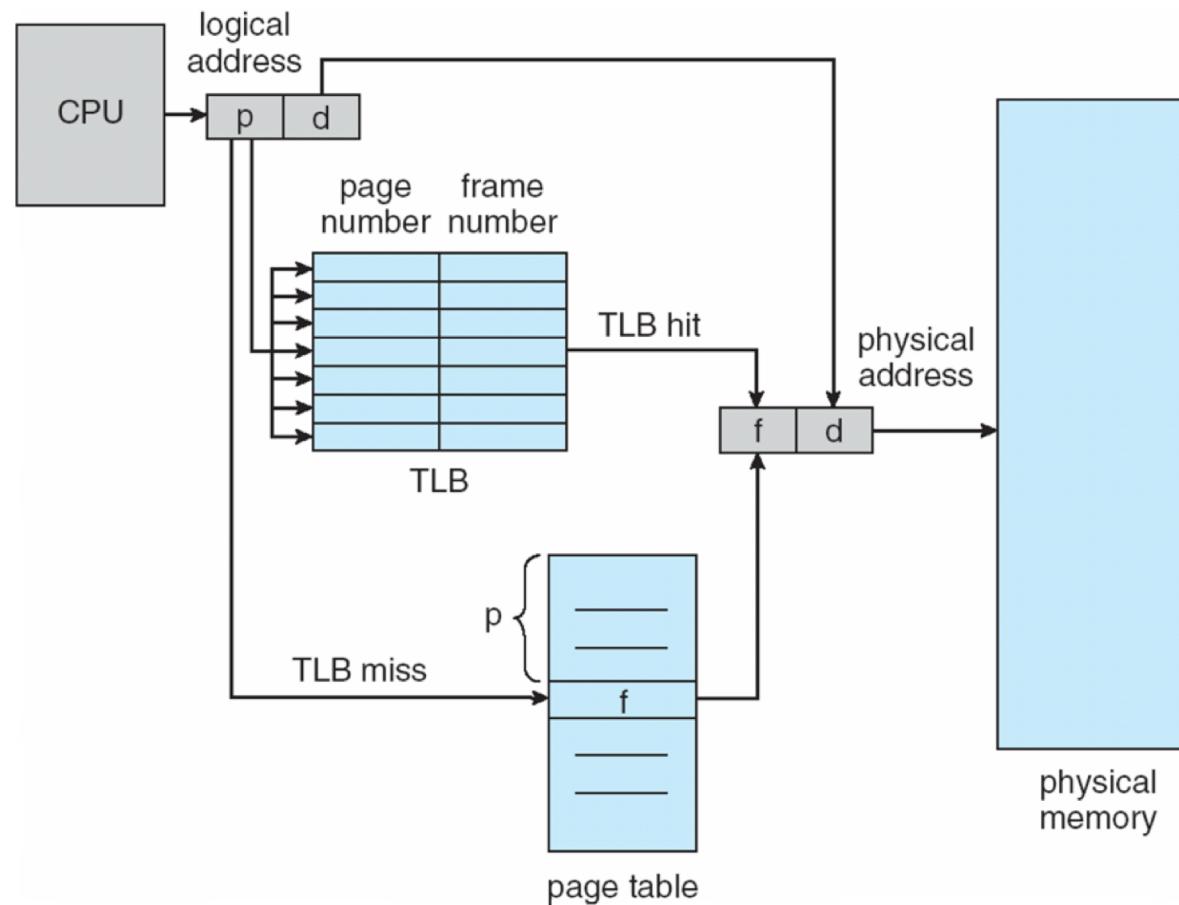
- TLB and context switch
 - Each process has its own page table
 - switching process needs to switch page table
 - **TLB must be consistent with page table**
 - Option I: Flush TLB at every context switch, **or**,
 - Option II: Tag TLB entries with **address-space identifier (ASID)** that uniquely identifies a process
 - some TLB entries can be **shared** by processes, and fixed in the TLB
 - e.g., TLB entries for the kernel
- TLB and operating system
 - MIPS: OS should deal with TLB miss exception
 - X86: TLB miss is handled by hardware

Associative Memory

- Associative memory: memory that supports **parallel search**
- Associative memory is not addressed by “addresses”, but **contents**
 - if page# is in associative memory’s key, return frame# (value) directly

Page #	Frame #
1	7
2	12
3	15
4	31

Paging Hardware With TLB



Effective Access Time

- Hit ratio - percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise we need **two memory access** so it is 20 ns: page table + memory access
- Effective Access Time (EAT)
 - $EAT = 0.80 \times 10 + 0.20 \times 20 = 12$ nanoseconds
 - implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,
 - $EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$
 - implying only 1% slowdown in access time.

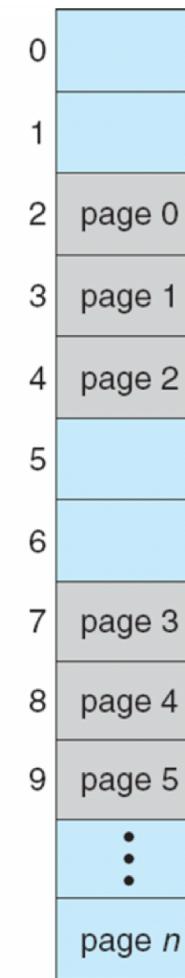
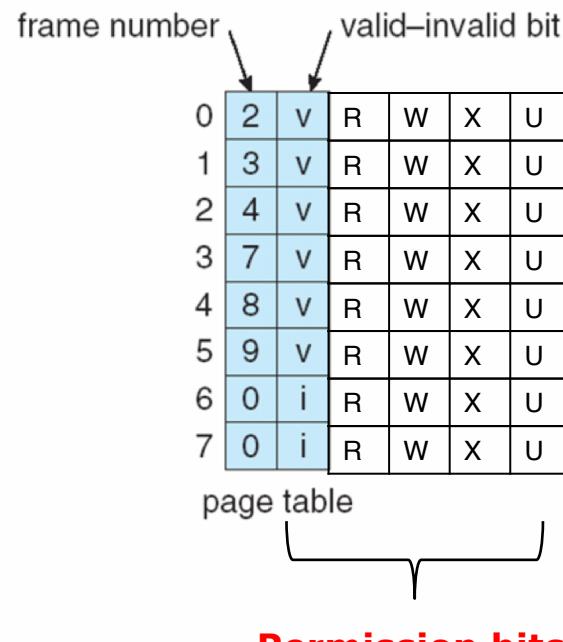
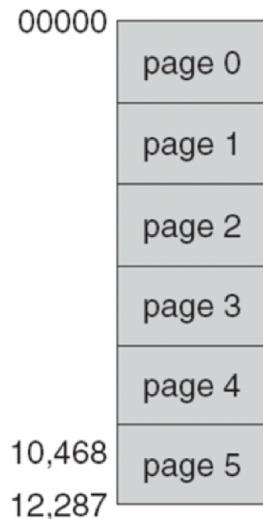
Memory Protection

- Accomplished by protection bits with each frame
- Each page table entry has a **present** (aka. valid) bit
 - present: the page has a valid physical frame, thus can be accessed
- Each page table entry contains some protection bits
 - **kernel/user, read/write, execution?, kernel-execution?**
 - why do we need them?
- Any violations of memory protection result in a trap to the kernel

Memory Protection (more)

- XN:protecting code
 - segregate areas of memory for use by either storage of processor instructions (code) or for storage of data
 - Intel: XD(execute disable), AMD: EVP (enhanced virus protection), ARM: XN (execute never)
- PXN: Privileged Execute Never (intel: SMEP)
 - A Permission fault is generated if the processor is executing at **EL1(kernel)** and attempts to execute an instruction fetched from the corresponding memory region when this PXN bit is 1 (usually user space memory)

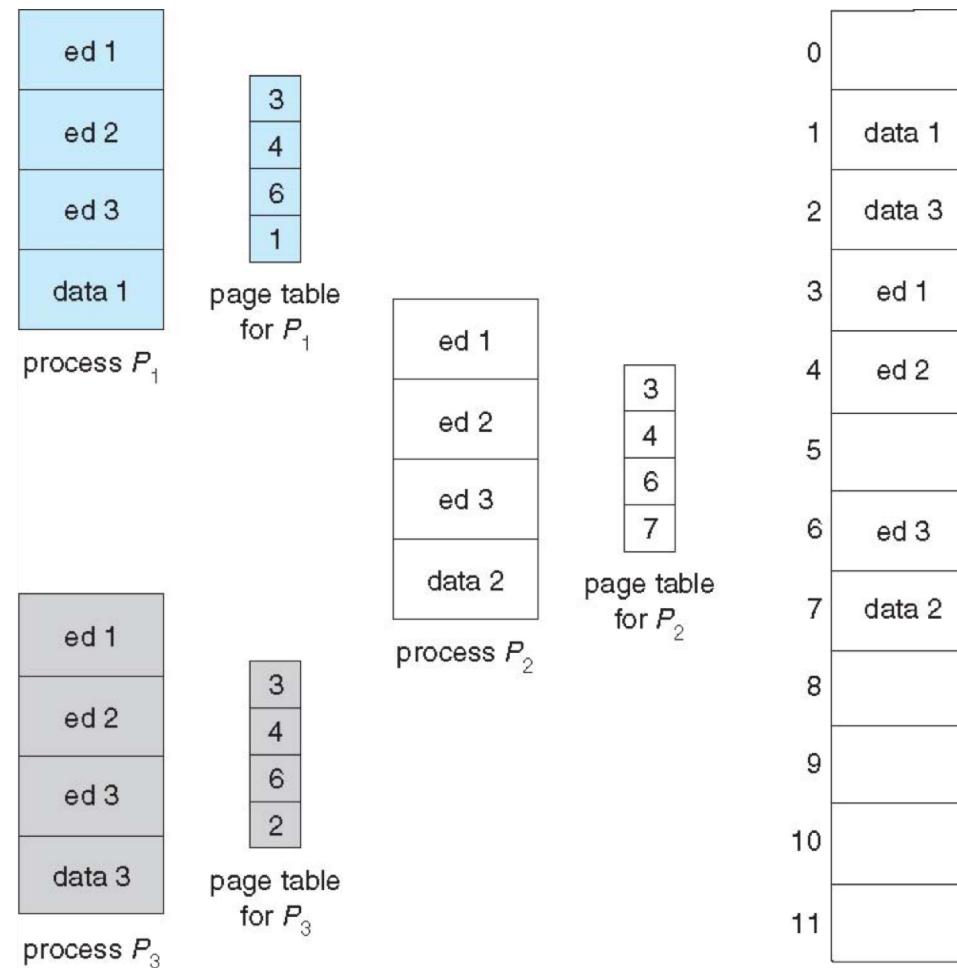
Memory Protection



Page Sharing

- Paging allows to share memory between processes
 - e.g., one copy of **code** shared by **all processes of the same program**
 - text editors, compilers, browser..
 - shared memory can be used for **inter-process communication**
 - shared libraries
 - Reentrant code: non-self-modifying code: never changes between execution
 - Each process can, of course, have its private code and data

Page Sharing



Structure of Page Table

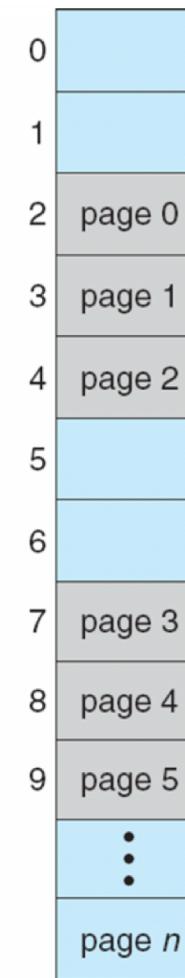
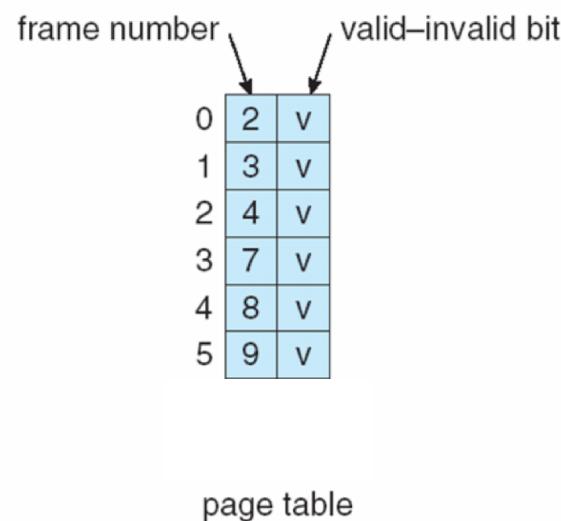
- One-level page table can consume lots of memory for page table
 - e.g., 32-bit logical address space and 4KB page size
 - page table would have 1 million entries ($2^{32} / 2^{12}$)
 - if each entry is 4 bytes → 4 MB of memory for page table alone
 - each process requires its own page table
 - page table must be **physically contiguous**
- To reduce memory consumption of page tables:
 - **hierarchical page table**
 - **hashed page table**
 - **inverted page table**

Hierarchical Page Tables

- Break up the logical address space into **multiple-level** of page tables
 - e.g., two-level page table
 - first-level page table contains the **frame#** for second-level page tables
 - “page” the page table
 - Why hierarchical page table can save memory for page table?

One-Level Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	



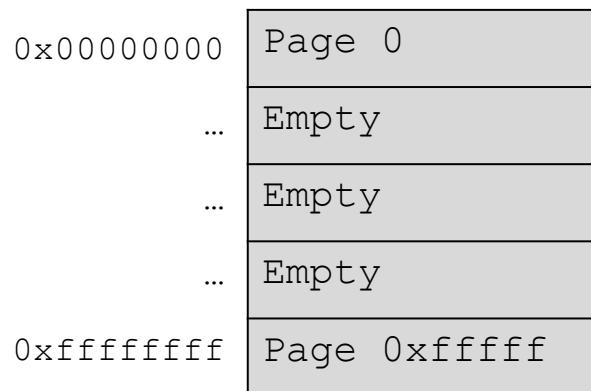
ELF binary basics

- Logical addresses have holes

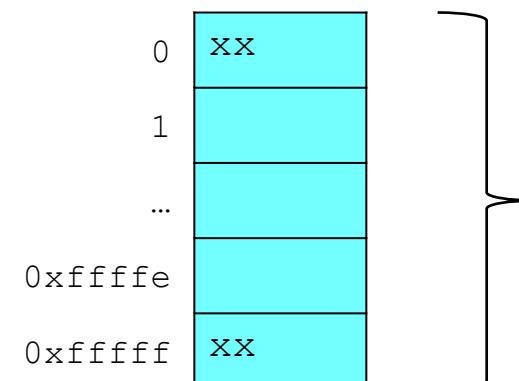
```
wenbo@wenbo-ThinkPad:~$ cat /proc/self/maps
559a61b8d000-559a61b95000 r-xp 00000000 08:05 3145753          /bin/cat
559a61d94000-559a61d95000 r--p 00007000 08:05 3145753          /bin/cat
559a61d95000-559a61d96000 rw-p 00008000 08:05 3145753          /bin/cat
559a63860000-559a63881000 rw-p 00000000 00:00 0                [heap]
7f8fc2690000-7f8fc305f000 r--p 00000000 08:05 4725339          /usr/lib/locale/locale-archive
7f8fc305f000-7f8fc3246000 r-xp 00000000 08:05 2626368          /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc3246000-7f8fc3446000 ---p 001e7000 08:05 2626368          /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc3446000-7f8fc344a000 r--p 001e7000 08:05 2626368          /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc344a000-7f8fc344c000 rw-p 001eb000 08:05 2626368          /lib/x86_64-linux-gnu/libc-2.27.so
7f8fc344c000-7f8fc3450000 rw-p 00000000 00:00 0
7f8fc3450000-7f8fc3477000 r-xp 00000000 08:05 2626340          /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3635000-7f8fc3659000 rw-p 00000000 00:00 0
7f8fc3677000-7f8fc3678000 r--p 00027000 08:05 2626340          /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3678000-7f8fc3679000 rw-p 00028000 08:05 2626340          /lib/x86_64-linux-gnu/ld-2.27.so
7f8fc3679000-7f8fc367a000 rw-p 00000000 00:00 0
7ffd416e0000-7ffd41701000 rw-p 00000000 00:00 0                [stack]
7ffd4178b000-7ffd4178e000 r--p 00000000 00:00 0                [vvar]
7ffd4178e000-7ffd41790000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                [vsyscall]
```

One-Level Page Table

- Worst case for 32-bit
 - access first page and last page
 - Page table needs to have 0x100000 entries
 - Size is 4MB, physically continuous



Process
(logical address range)



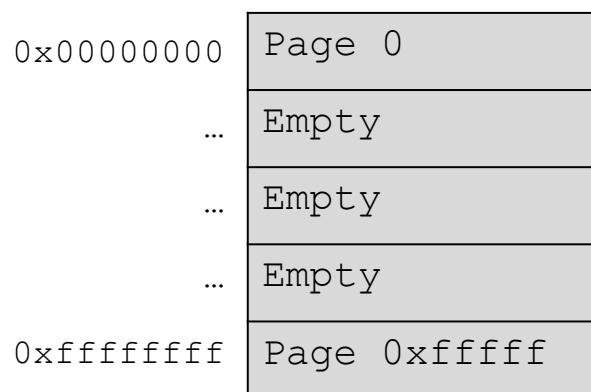
Page table

- $1K=1024=0x400$
- $4K=4096=0x1000$
- $1M=1024 \times 1024=0x100000$

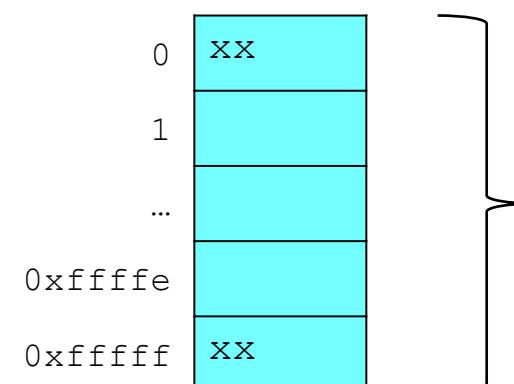
1M entries
Size is 4MB

One-Level Page Table

- Can we break down page table into pages?
 - Page size is 4KB, how many entries per page?



Process
(logical address range)

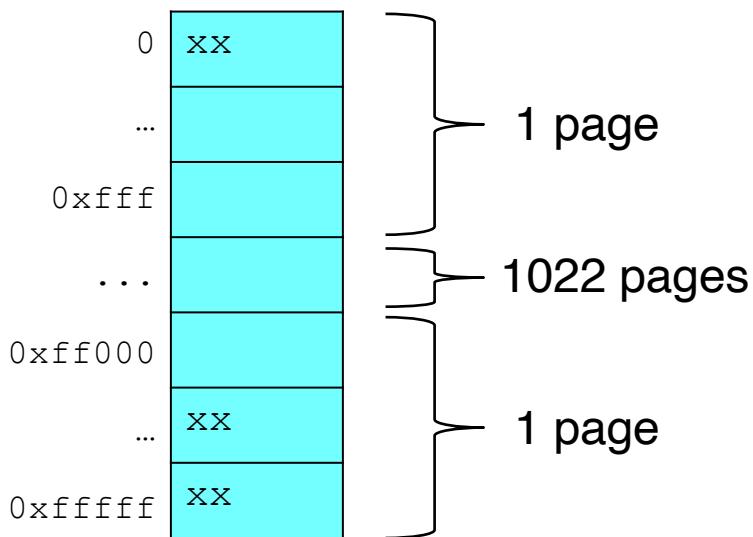


Page table

1M entries
Size is 4MB

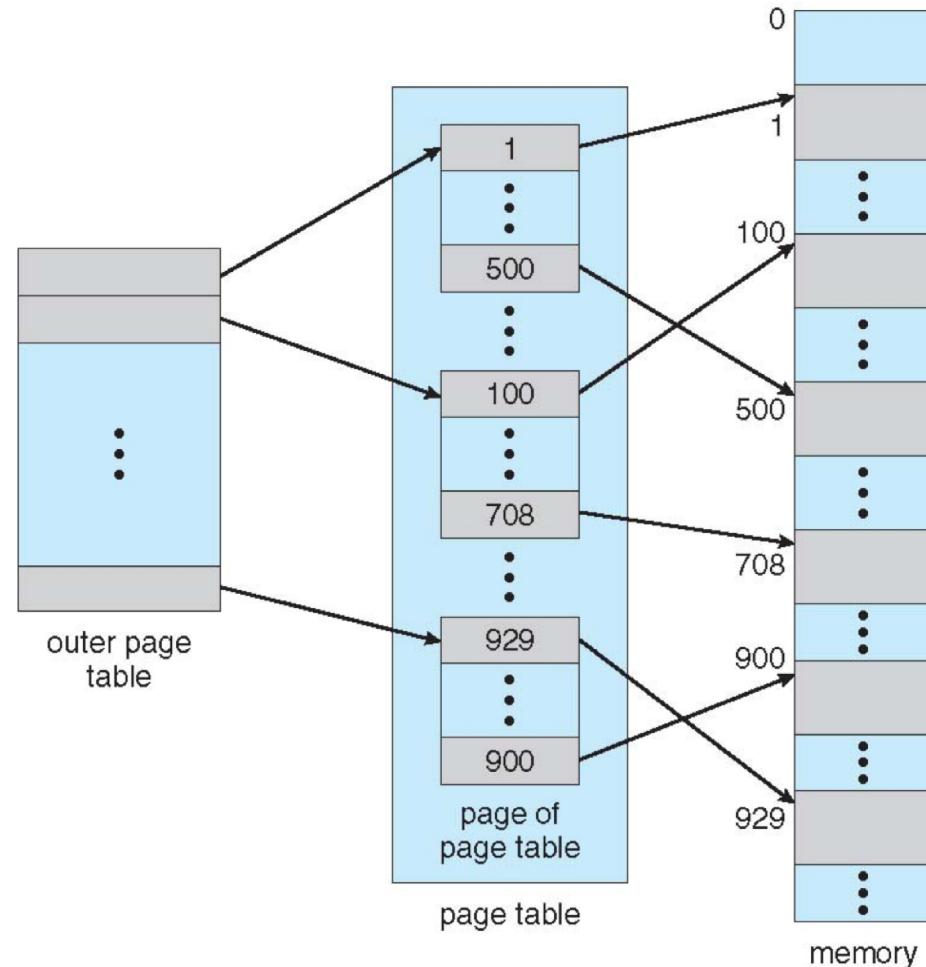
One-Level Page Table

- Can we break down page table into pages?
 - Page size is 4KB, how many entries per page?
 - 1K entries / page
 - Can break down to 1K pages -> Can be indexed by 1K entries -> Put into another page



Page table

Two-Level Page Table

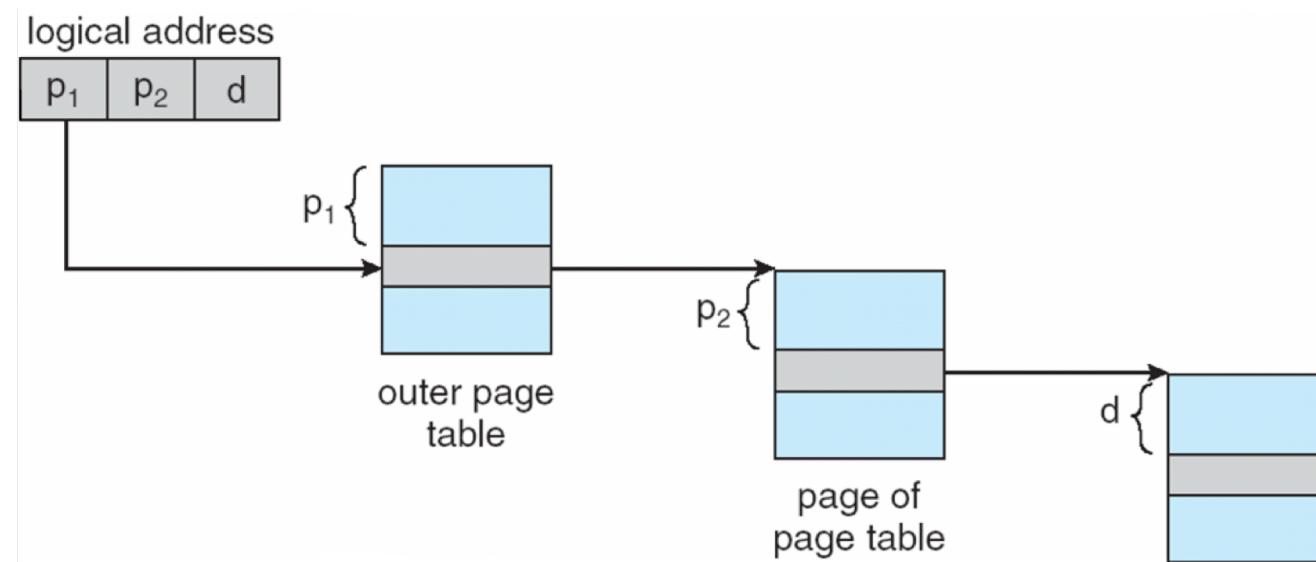


Two-Level Paging

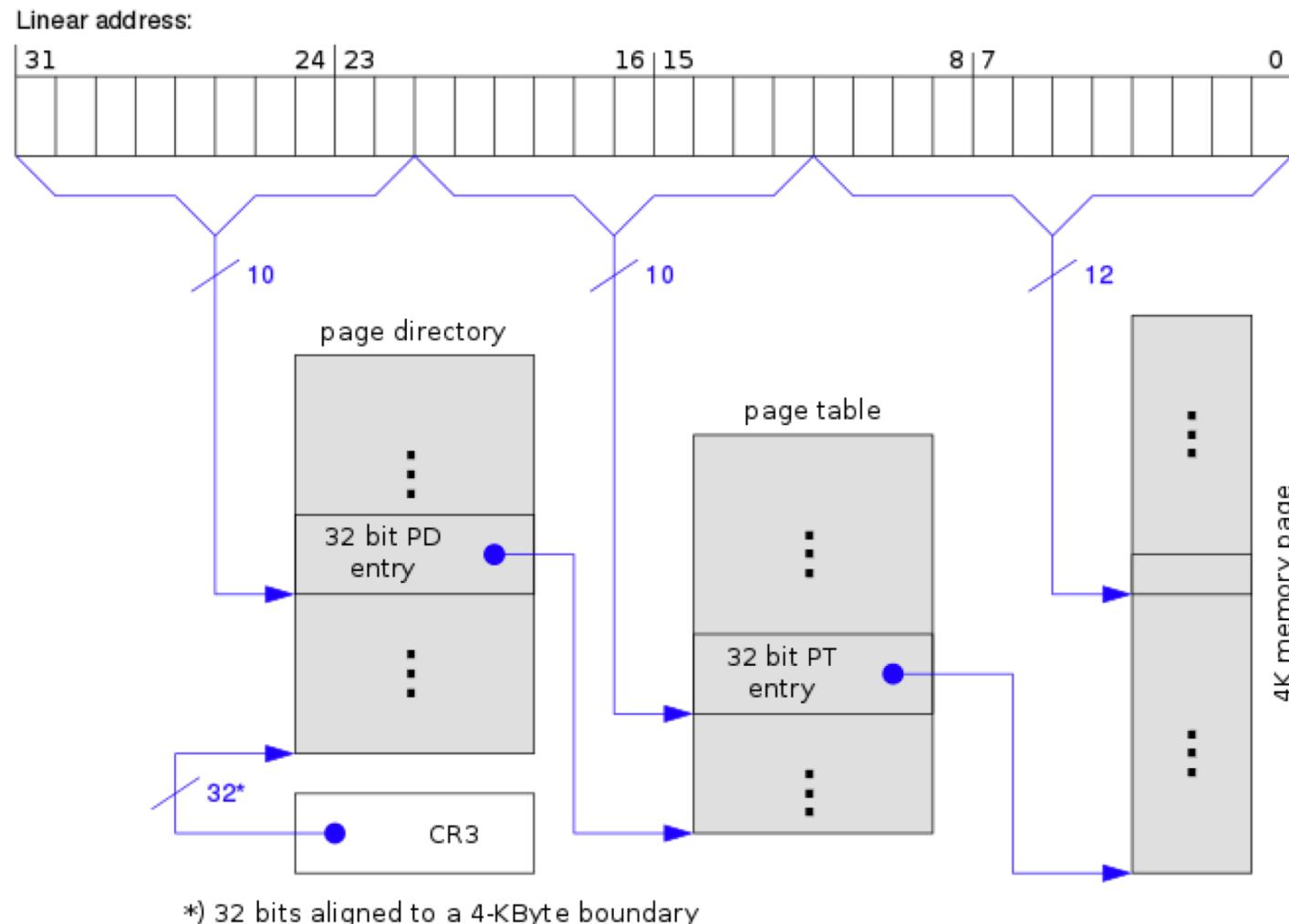
- A logical address is divided into:
 - a **page directory number** (first level page table)
 - a **page table number** (2nd level page table)
 - a **page offset**
- Example: 2-level paging in 32-bit Intel CPUs
 - 32-bit address space, 4KB page size
 - 10-bit page directory number, 10-bit page table number
 - each page table entry is 4 bytes, one frame contains 1024 entries (2^{10})

p_1	p_2	d
10	10	12

Address-Translation Scheme



Page Table in Linux

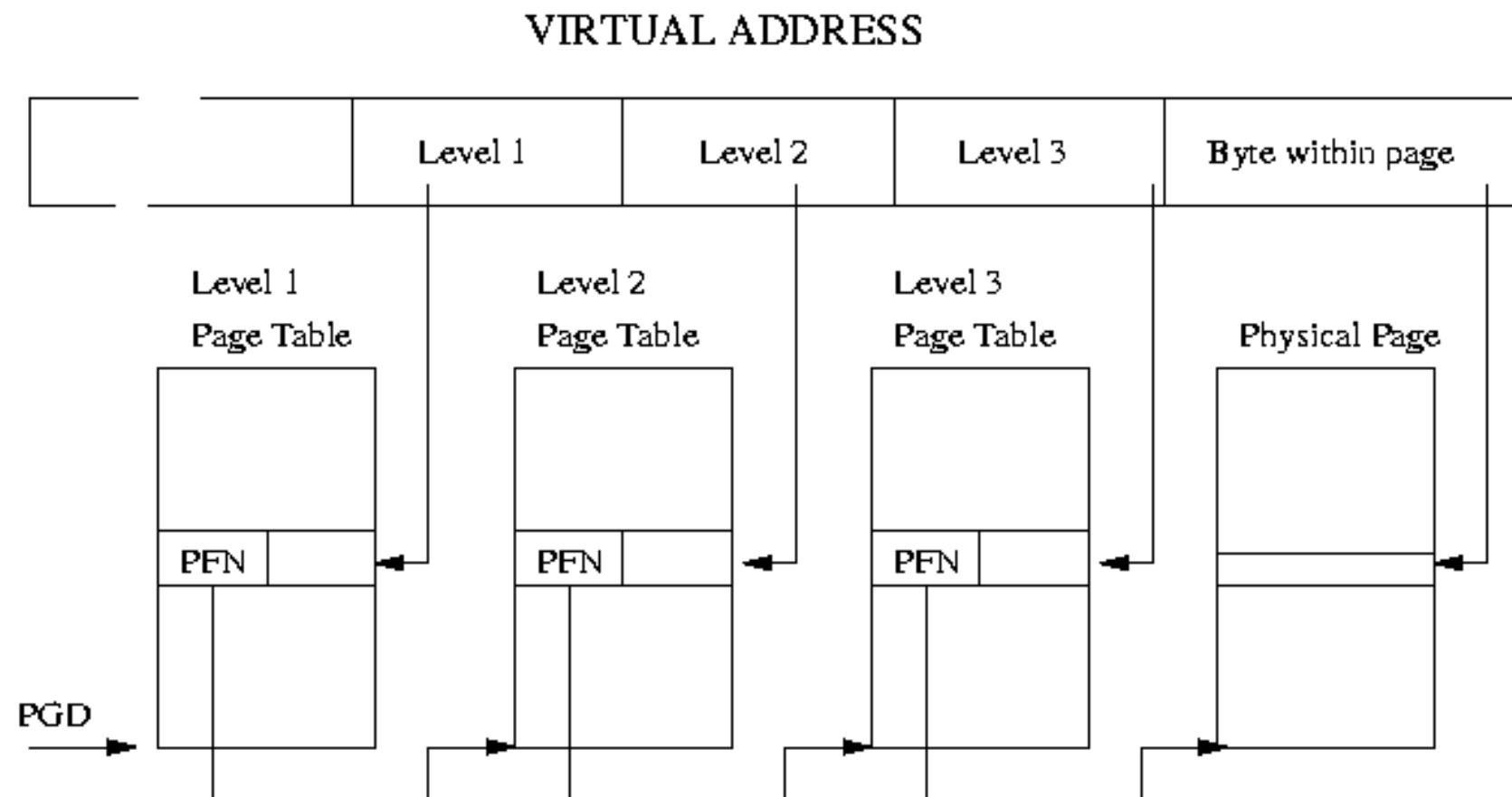


64-bit Logical Address Space

- 64-bit logical address space requires more levels of paging
 - two-level paging is not sufficient for 64-bit logical address space
 - if page size is 4 KB (2^{12}), outer page table has 2^{42} entries, inner page tables have 2^{10} 4-byte entries
 - one solution is to add more levels of page tables
 - e.g., three levels of paging: 1st level page table is 2^{32} bytes in size
 - and possibly 4 memory accesses to get to one physical memory location
 - usually **not support full 64-bit virtual address space**
 - AMD-64 supports 48-bit
 - ARM64 supports 39-bit, 48-bit

64-bit Logical Address Space

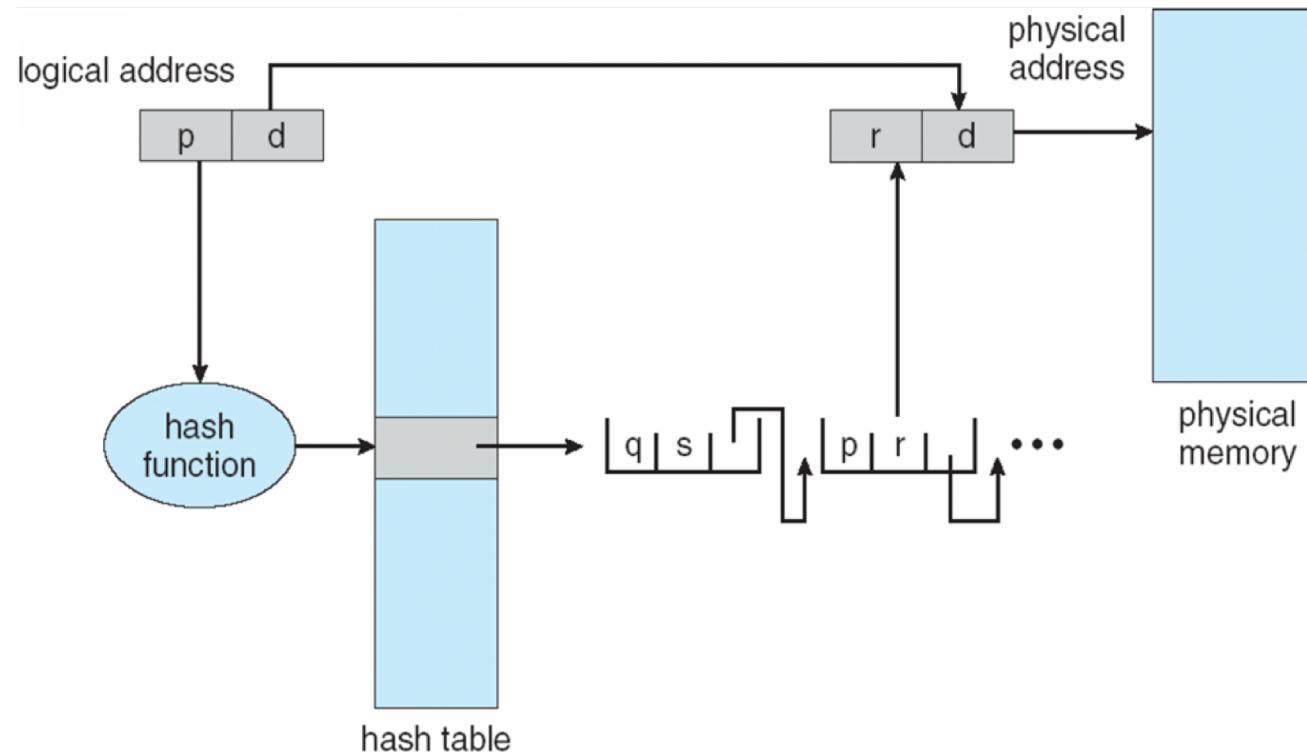
- ARM64: 39 bits = 9+9+9+12



Hashed Page Tables

- In hashed page table, virtual page# is hashed into a frame#
 - the page table contains a chain of elements hashing to the same location
 - each element contains: page#, frame#, and a **pointer to the next element (resolving conflict)**
 - virtual page numbers are compared in this chain searching for a match
 - if a match is found, the corresponding frame# is returned
- Hashed page table can be used in address spaces > 32 bits
- **Clustered page tables**
 - Each entry refers to several pages

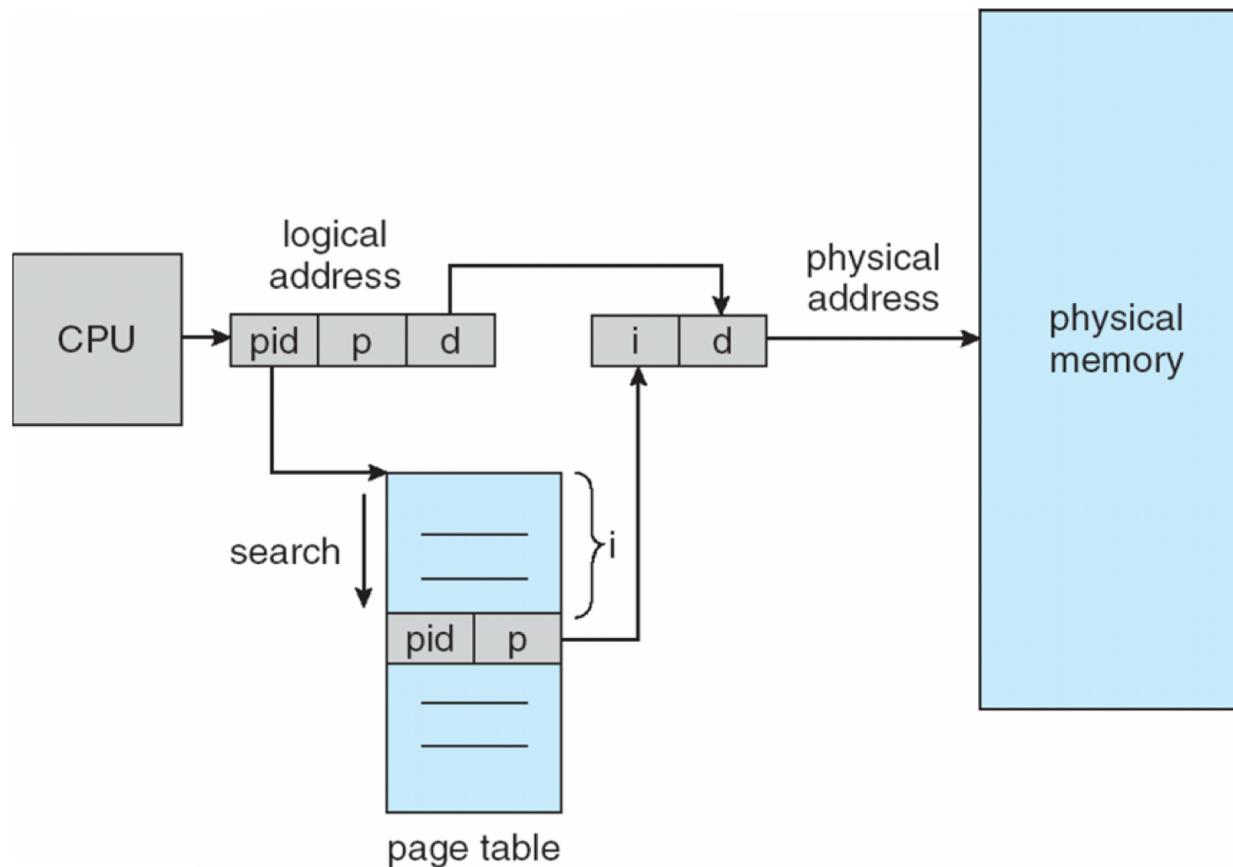
Hashed Page Table



Inverted Page Table

- Inverted page table tracks allocation of physical frame to a process
 - one entry for each physical frame → fixed amount of memory for page table
 - each entry has the **process id** and the **page#** (virtual address)
- Sounds like a brilliant idea?
 - to translate a virtual address, it is necessary to search the (**whole**) page table
 - can use TLB to accelerate access, TLB miss could be very expensive
 - how to implement shared memory?
 - a physical frame can only be mapped into one process!
 - Because one physical memory page cannot have multiple virtual page entry!

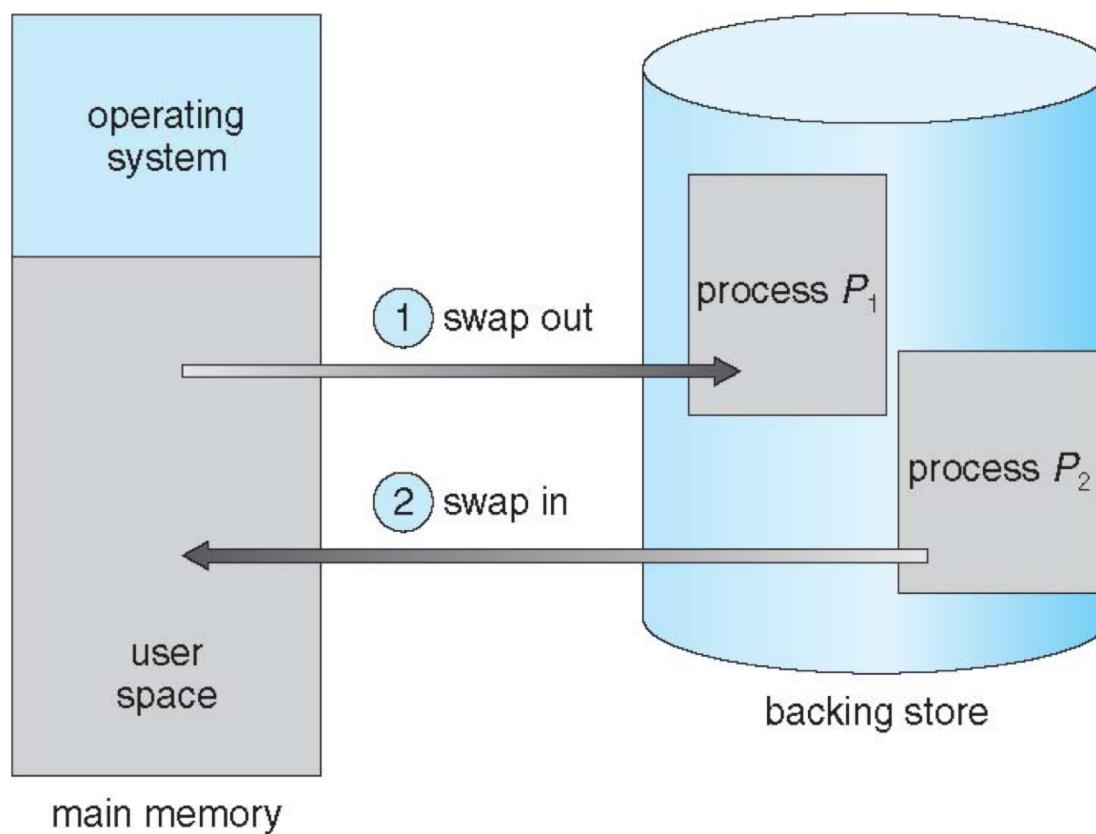
Inverted Page Table



Swapping

- Swapping extends physical memory with backing disks
 - a process can be swapped temporarily out of memory to a backing store
 - backing store is usually a (fast) disk
 - the process will be brought back into memory for continued execution
 - does the process need to be swapped back in to same physical address?
- Swapping is usually only initiated under memory pressure
- Context switch time can become very high due to swapping
 - if the next process to be run is not in memory, need to swap it in
 - disk I/O has high latency

Swapping

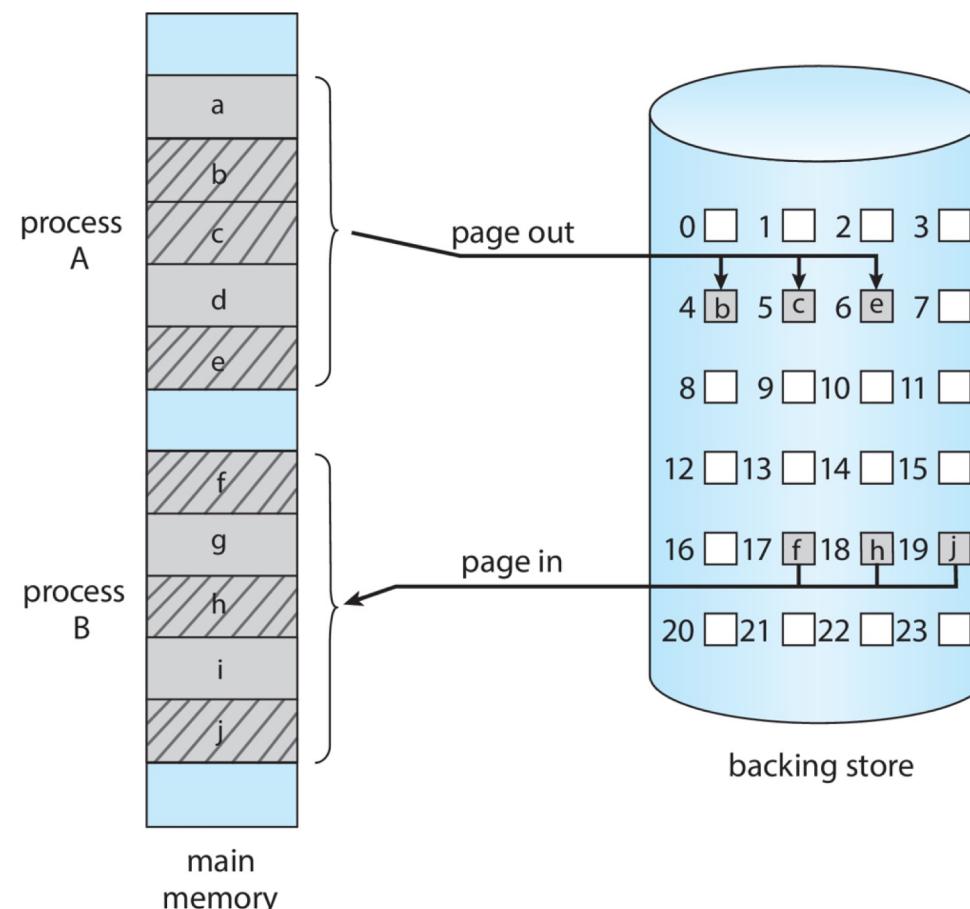


Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be **very high**
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap out time of 2,000 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped - by knowing how much memory really being used

Swapping with Paging

- Swap pages instead of entire process



Swapping on Mobile Systems

- Not typically supported
 - Flash memory based
 - Small amount of space
 - Limited number of write cycles
 - Poor throughput between flash memory and CPU on mobile platform
 - Instead use other methods to free memory if low
 - iOS asks apps to voluntarily relinquish allocated memory
 - Read-only data thrown out and reloaded from flash if needed
 - Failure to free can result in termination
 - Android terminates apps if low free memory, but first writes application state to flash for fast restart

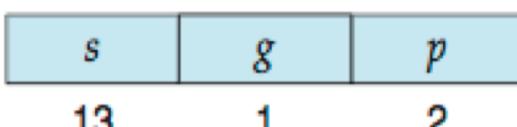
Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

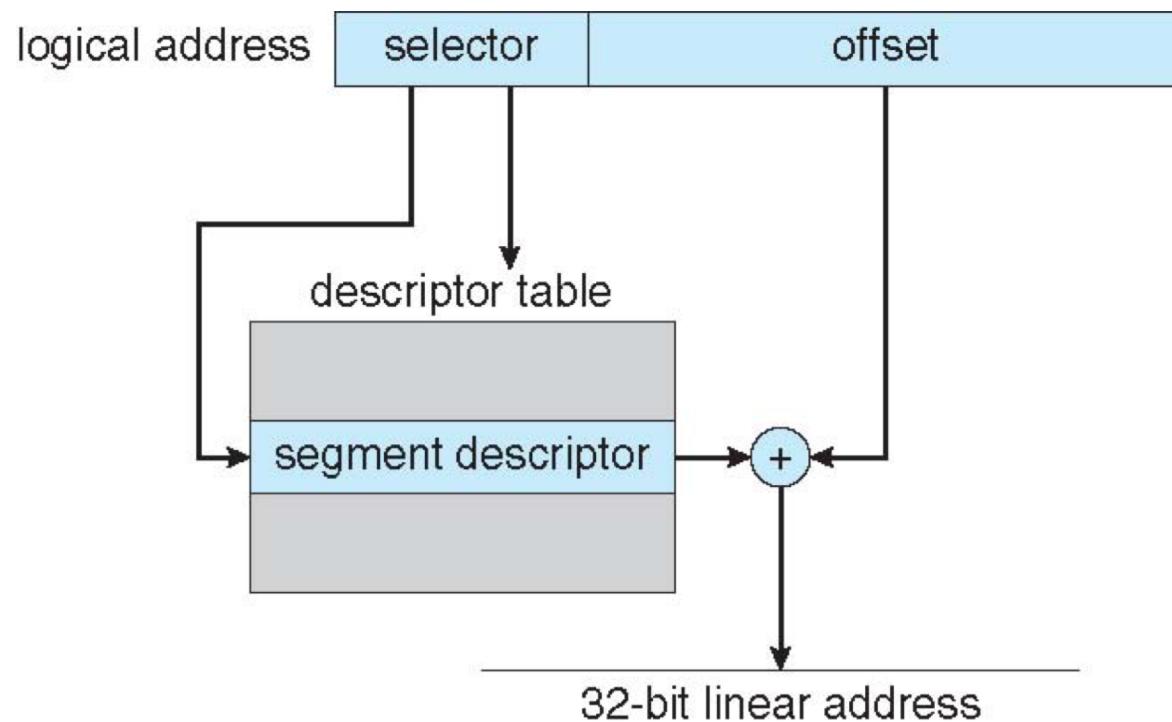
- Supports both segmentation and segmentation with paging
 - Each segment can be 4 GB
 - Up to 16 K segments per process
 - Divided into two partitions
 - First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)

Example: The Intel IA-32 Architecture

- CPU generates logical address
 - Selector given to segmentation unit
 - Which produces linear addresses
 - 

s	g	p
13	1	2
 - s-> segment number, g-> local/global, p->protection
 - Segment address is stored in the segment registers
 - CS:segment address of the code segment of the currently executing instruction
 - DS: many data segments, can be one data segment address
 - GDTR, LDTR -> base address of the descriptor table
 - descriptor: base, limit and other bits

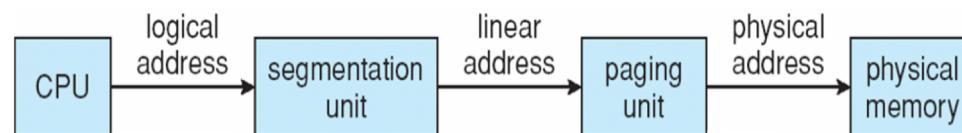
Intel IA-32 Segmentation



Example: The Intel IA-32 Architecture

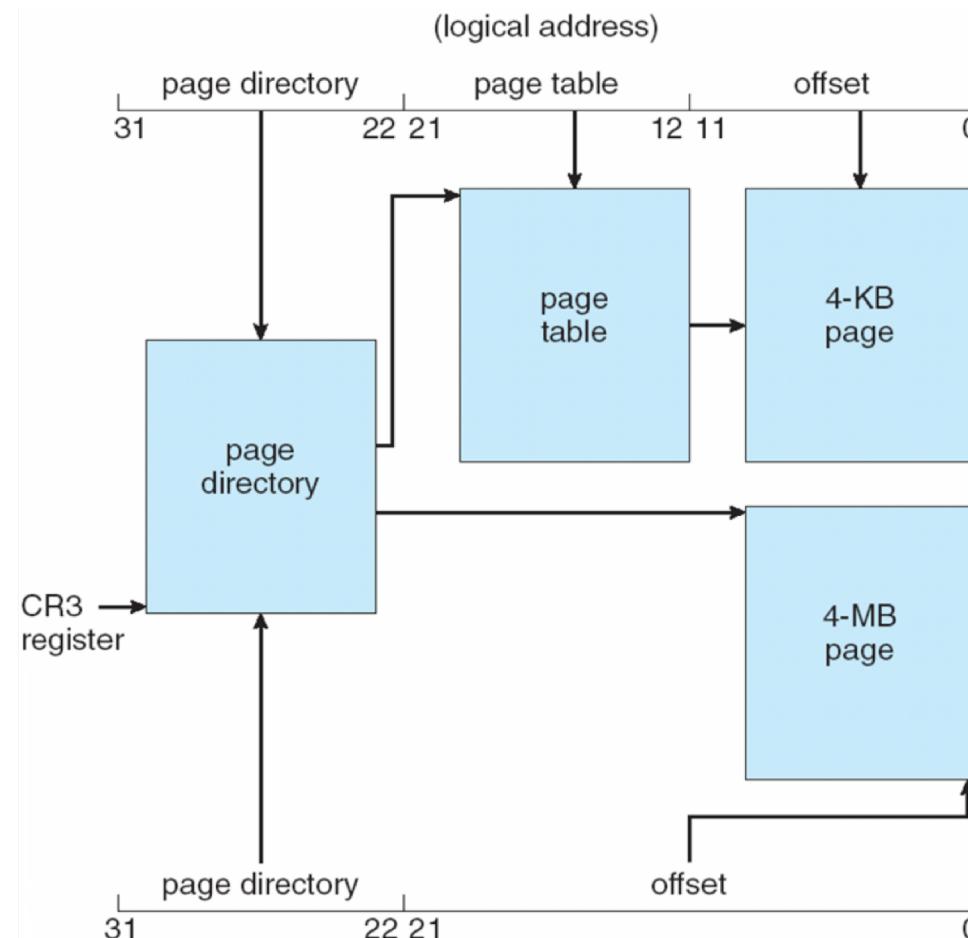
- Linear address given to paging unit
 - Which generates physical address in main memory
 - Paging units form equivalent of MMU
 - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32



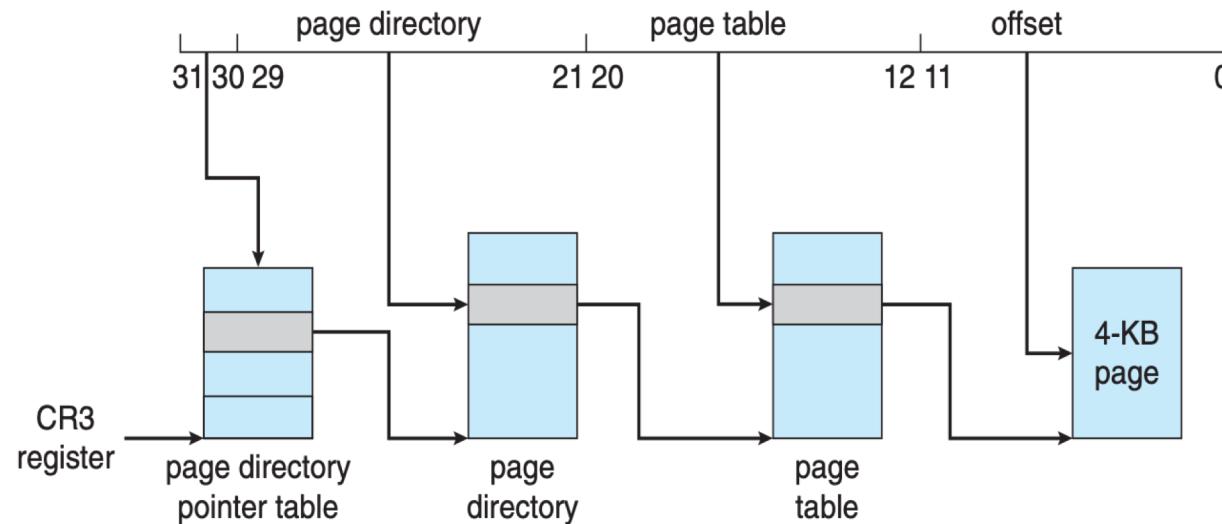
page number		page offset
p_1	p_2	d
10	10	12

Intel IA-32 Paging Architecture



Intel IA-32 Page Address Extensions

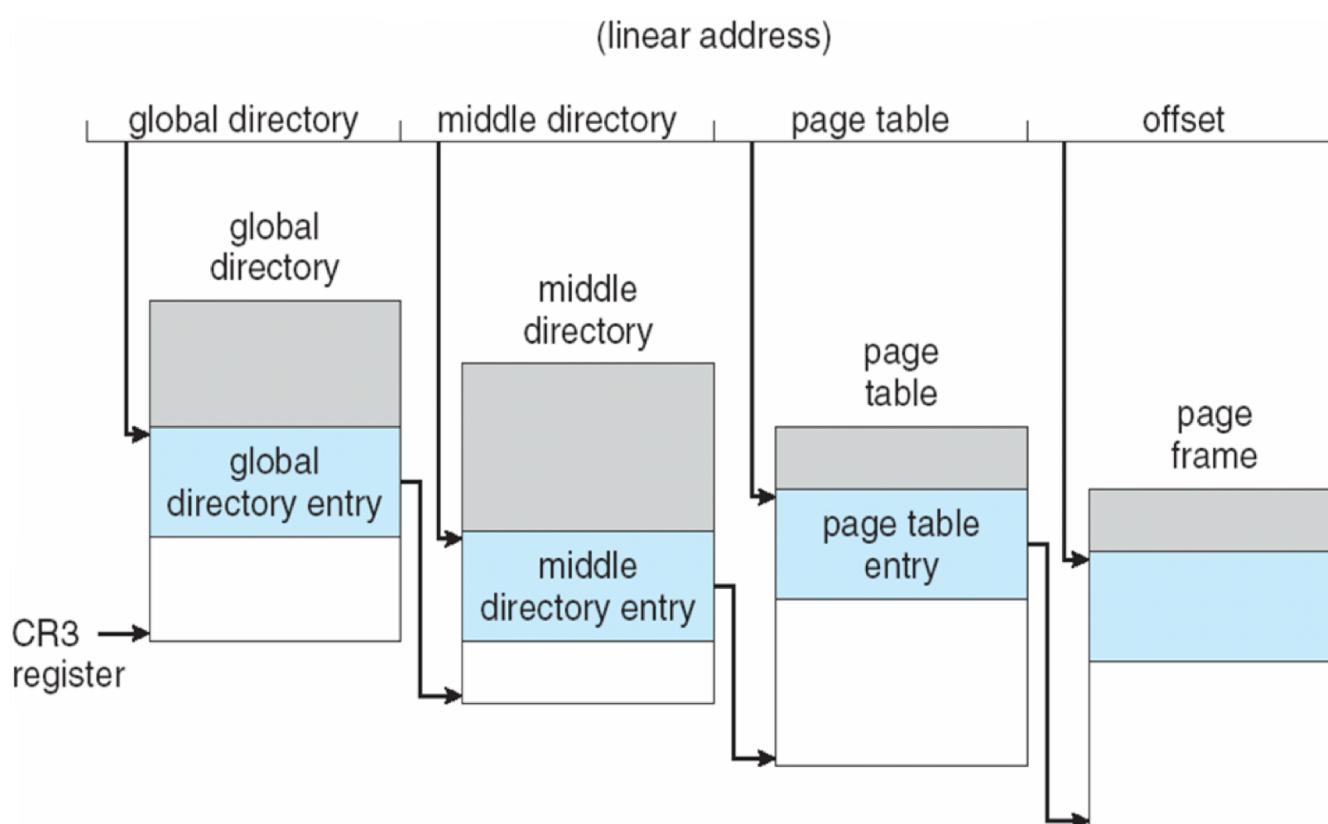
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a **page directory pointer table**
 - Page-directory and page-table entries moved to 64-bits in size
 - Net effect is increasing address space to 36 bits - 64GB of **physical** memory
 - Virtual address space is still 4GB
 - Page table entries is 64 bits, can index 36 bits physical address (64GB)



Linux Support for Intel Pentium

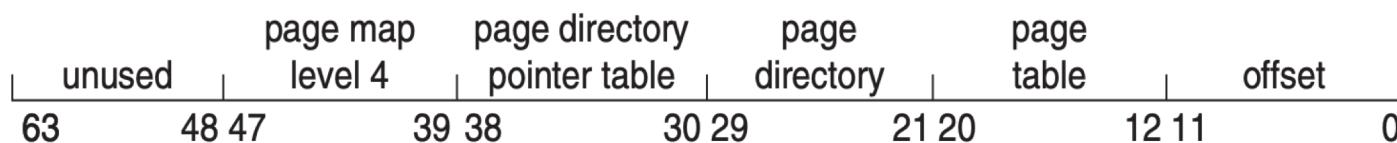
- Linux uses only 6 segments
 - kernel code, kernel data, user code, user data
 - task-state segment (TSS), default LDT segment
- Linux only uses two of four possible modes
 - kernel: ring 0, user space: ring 3
- Uses a generic four-level paging for 32-bit and 64-bit systems
 - Two-level paging, middle and upper directories are omitted
 - Three-level page, upper directories are omitted

Three-level Paging in Linux



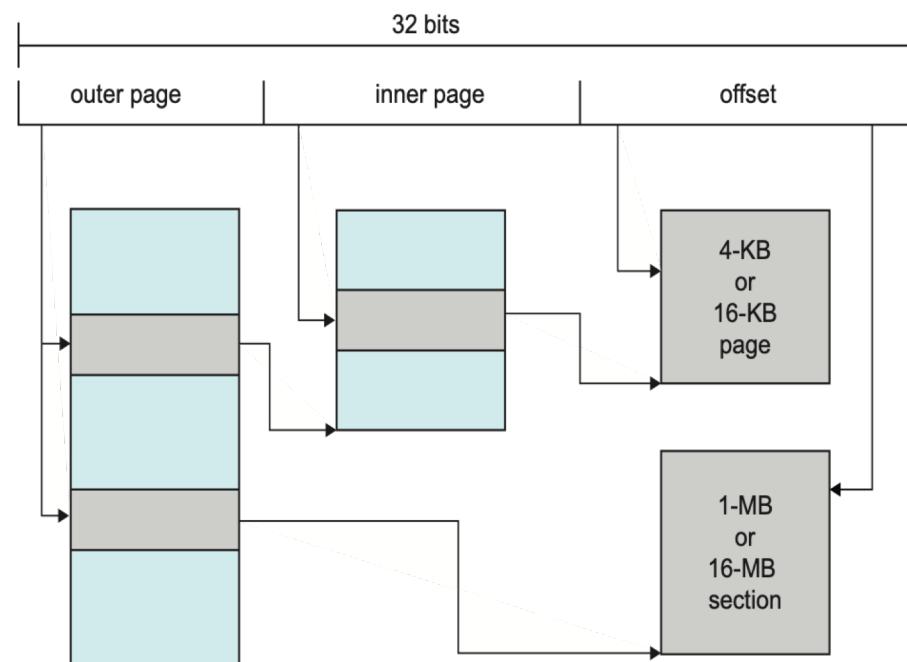
Intel x86-64

- 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 so virtual addresses are 48 bits and physical addresses are 52 bits
 - Usually no needs



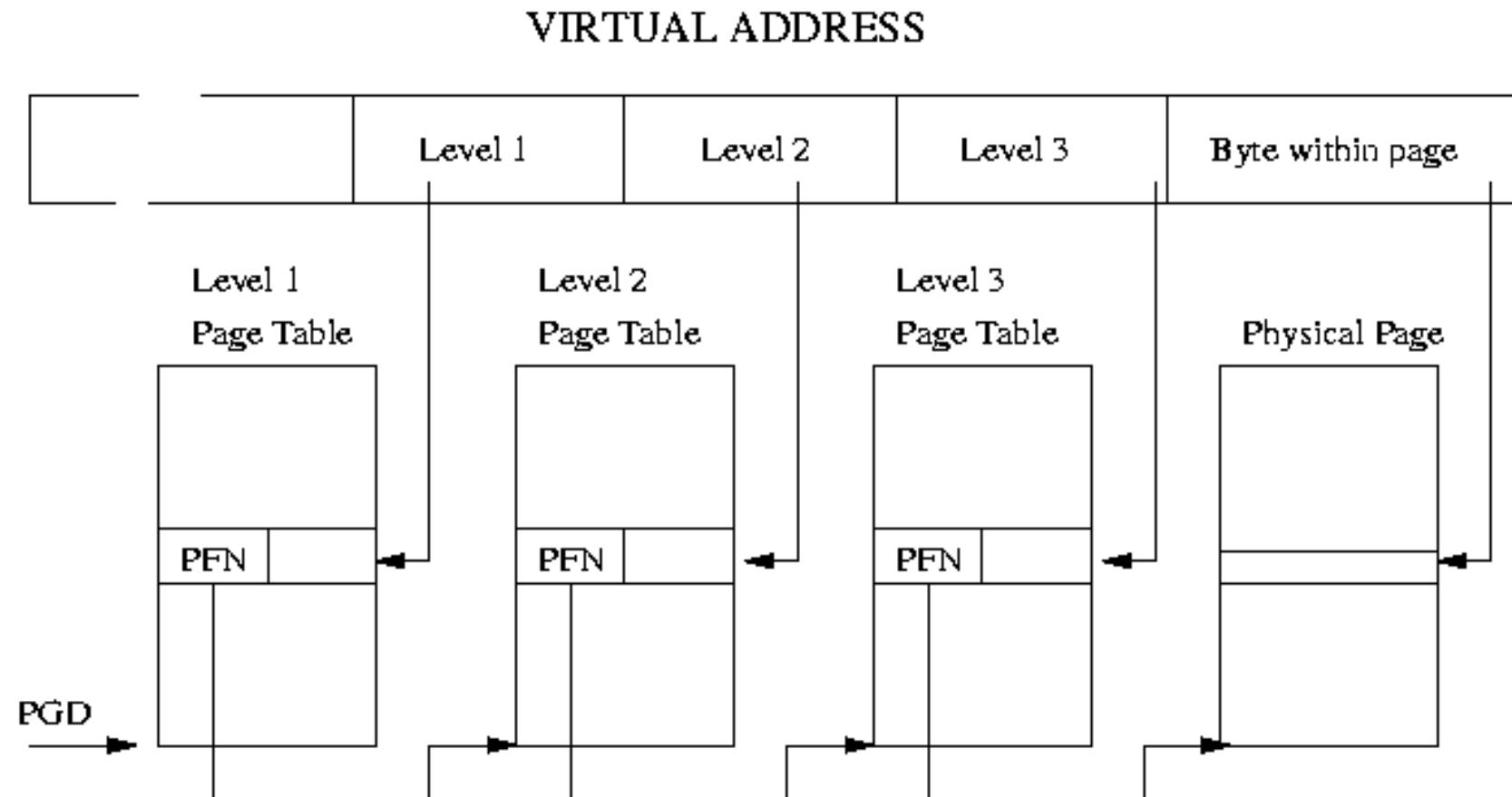
Example: ARM32 Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, **32-bit CPU**
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
 - Outer level has **two micro TLBs** (one data, one instruction)
 - Inner is **single main TLB**
 - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Example: ARM64 Architecture

- Support
 - 39 bits addressing = 9+9+9+12, three-level page table
 - 48 bits addressing = 9+9+9+9+12, four-level page table



Takeaway

- Contiguous allocation
 - Fixed, variable
 - Segmentation
- Fragmentation
 - Internal, external
- MMU
- Paging
 - Page table
 - Hierarchical, hashed page table, inverted
 - Two-level, three-level, four-level
 - For 32 bits and 64 bits architectures