Chapter 9: Virtual memory-hardware

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Examples

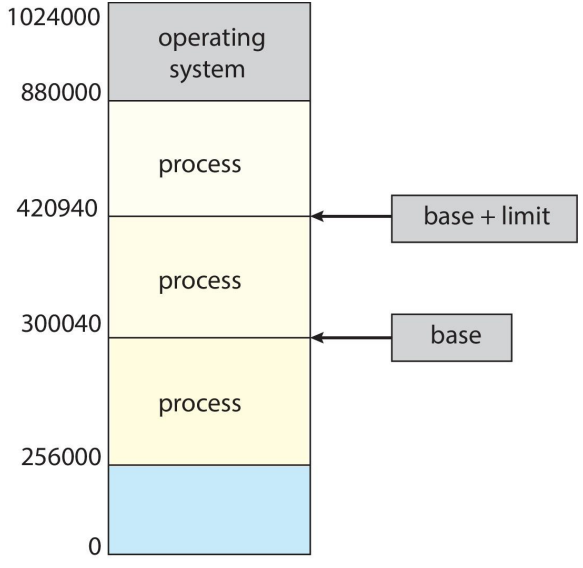
prepared based on https://www.scs.stanford.edu/24wi-cs212/notes/vm_hardware.pdf
and the Chapter 9 of the book

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Consider multiprogramming on physical memory

- Need to ensure that a process can access only access those addresses in its address space.
  - use a pair of **base** and **limit registers** define the logical address space of a process



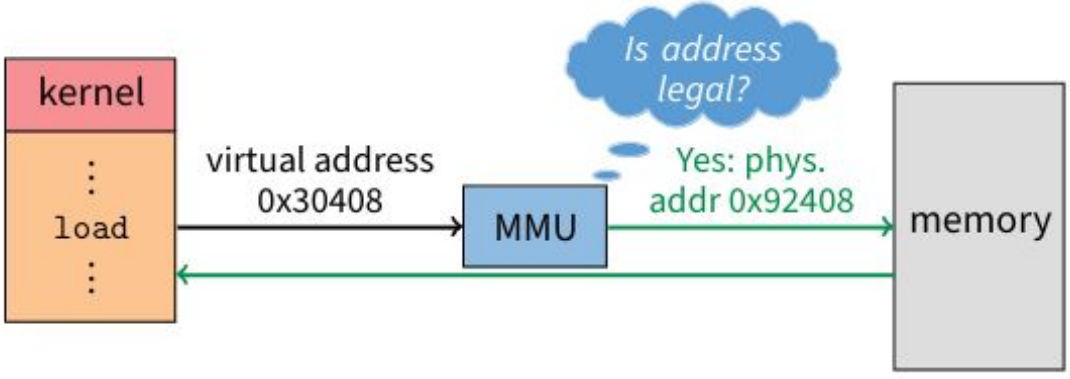
- What happens if a process
  - needs to expand?
  - needs more memory than is on the machine?
  - has an error and writes to a different address?
  - isn't using its memory?
- When does gcc have to know it will run at 0x4000?

Issues in sharing physical memory

- Protection
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process A from trashing B's memory
  - Also prevent A from even observing B's memory (ssh-agent)
- Transparency
  - A process shouldn't require particular physical memory bits
  - Yet processes often require large amounts of contiguous memory(for stack, large data structures, etc.)
- Resource exhaustion
  - Programmers typically assume machine has "enough" memory
  - Sum of sizes of all processes often greater than physical memory

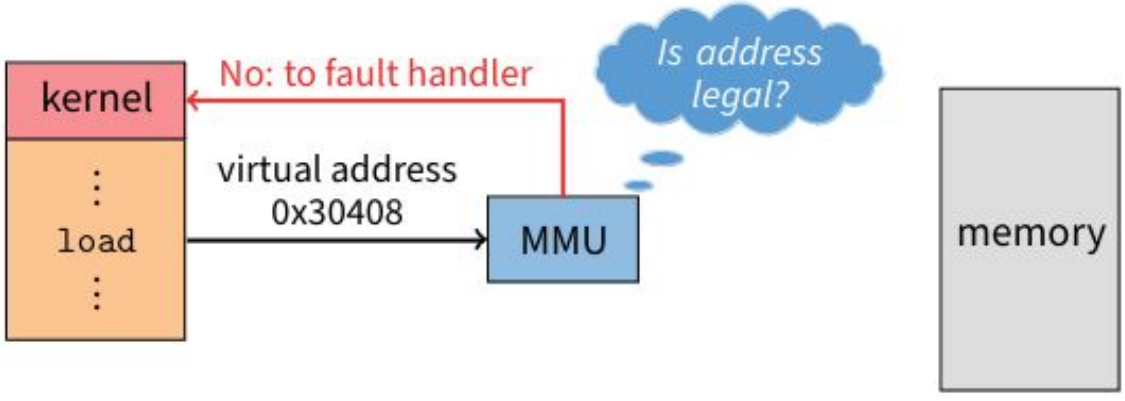
https://www.scs.stanford.edu/24wi-cs212/notes/vm_hardware.pdf

Virtualization of memory: Virtual memory goals



- Give each program its own virtual address space
  - At runtime, Memory-Management Unit relocates each load/store
  - Application doesn't see physical memory addresses
- Also enforce protection
  - Prevent one app from messing with another's memory
- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk

Virtualization of memory: Virtual memory goals



- Give each program its own virtual address space
  - At runtime, Memory-Management Unit relocates each load/store
  - Application doesn't see physical memory addresses
- Also enforce protection
  - Prevent one app from messing with another's memory
- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk

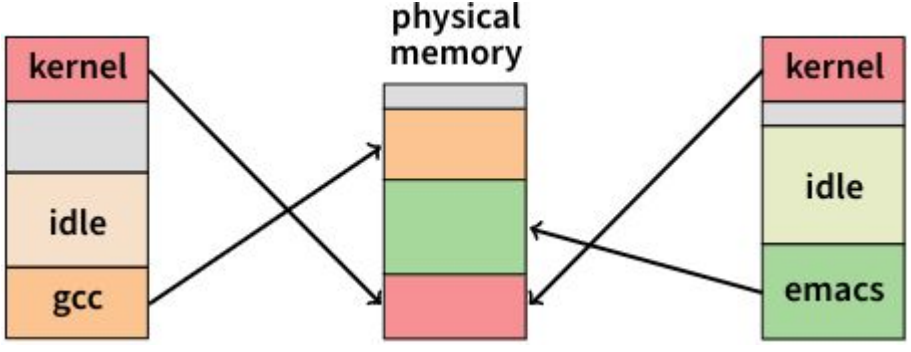
Advantages of virtual memory

Can re-locate program while running

- Run partially in memory, partially on disk

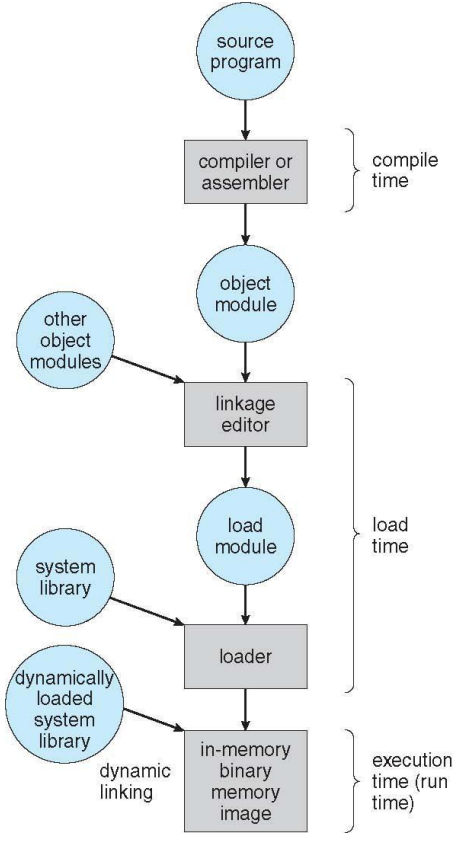
Most of a process's memory may be idle (80/20 rule).

- Write idle parts to disk until needed
- Let other processes use memory of idle part
- Like CPU virtualization: when process not using CPU, switch
  - Not using a memory region?
  - switch it to another process



Challenge: VM = extra layer, could be slow

Multistep Processing of a User Program



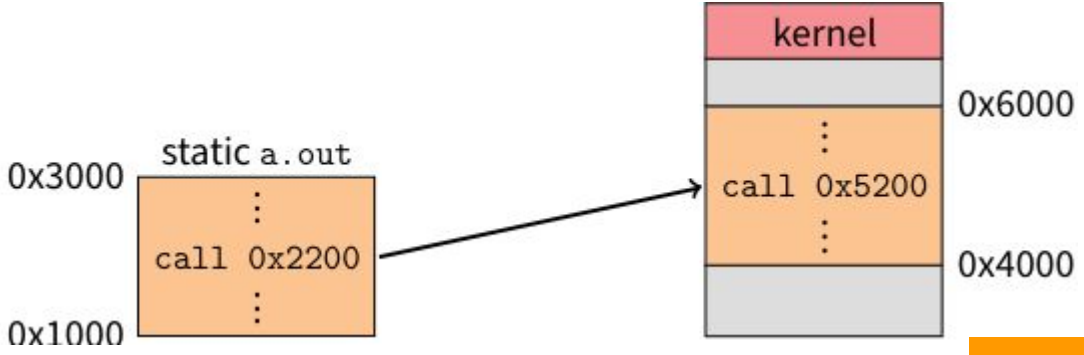
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Idea1: no hardware, load time linking



Linker patches addresses of symbols like **printf**

Idea: link when process executed, **not at compile time**

- Already have PIE (position-independent executable) for security
- Determine where process will reside in memory at launch
- Adjust all references within program (using addition)

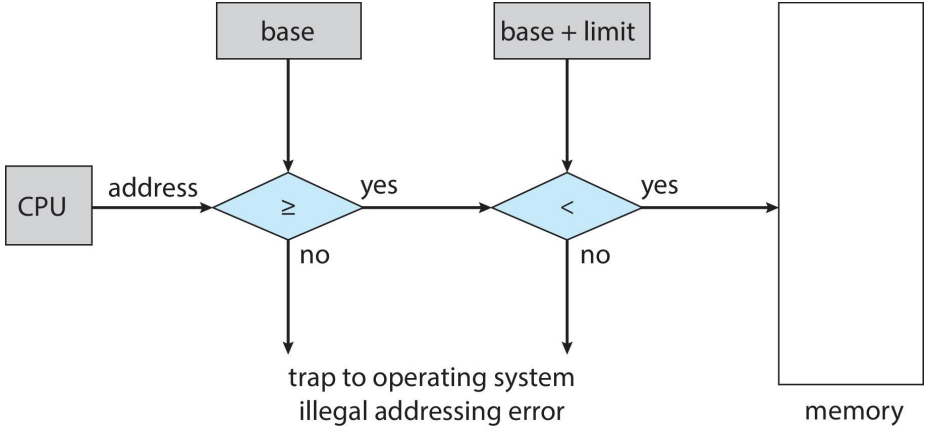
Problems?
- How to enforce protection?
- How to move once already in memory? (consider data pointers)
- What if no contiguous free region fits program?

Idea2: with hardware, base + bound register

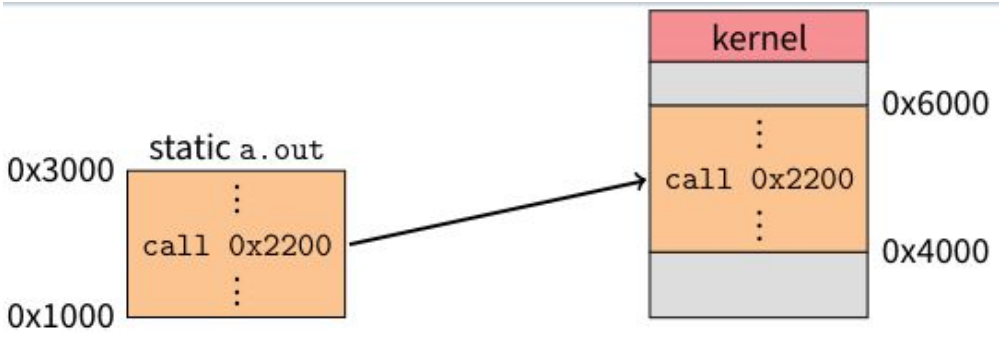
Two special privileged registers: base and bound

On each load/store/jump:

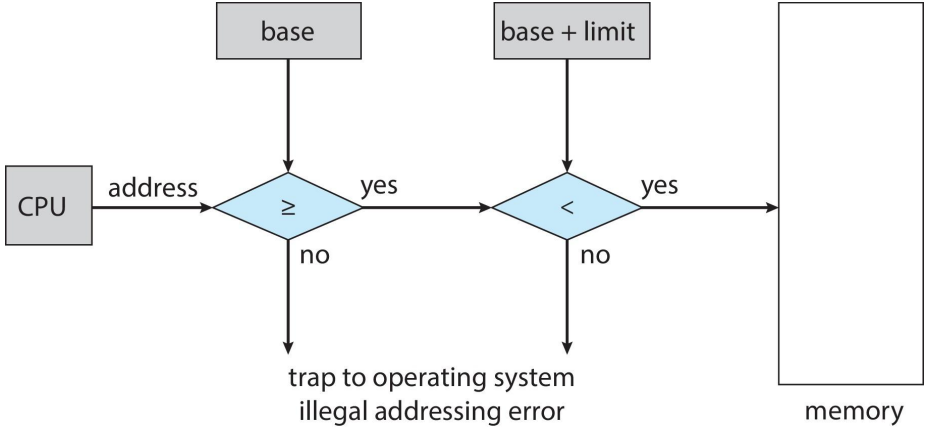
- **Physical address = virtual address + base**
- **Check 0 ≤ virtual address < bound,**
  - **else trap to kernel**



Idea2: hardware support: base + bound register

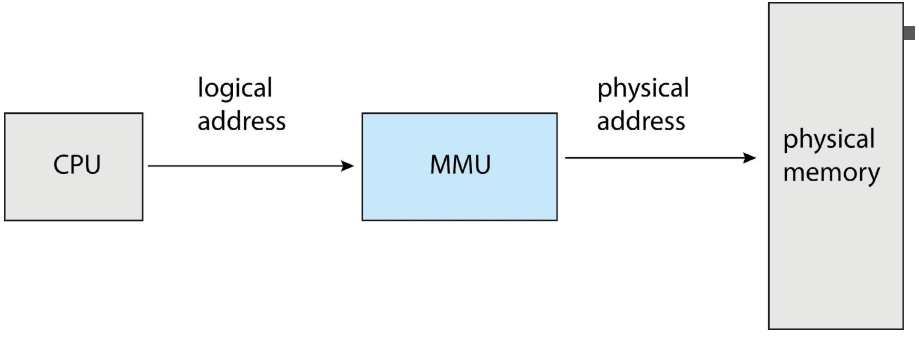


- **How to move process in memory?**
  - **Change base register**
- **What happens on context switch?**
  - **Kernel must re-load base and bound registers**
  - **the instructions to loading the base and limit registers are privileged**



Logical vs. Physical Address Space

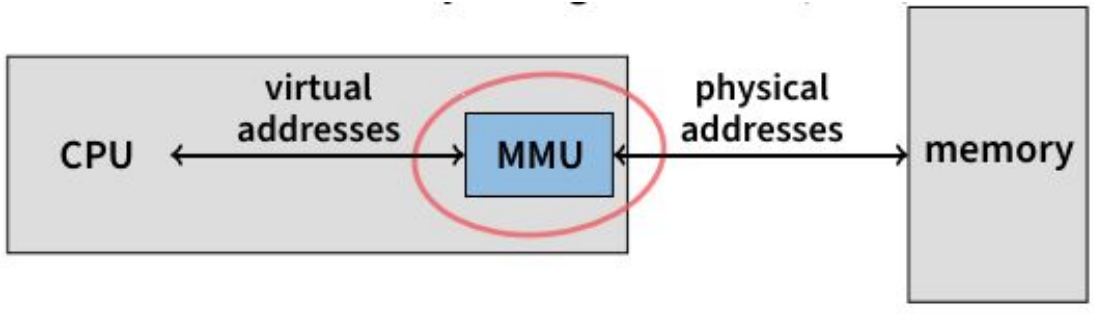
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit



- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

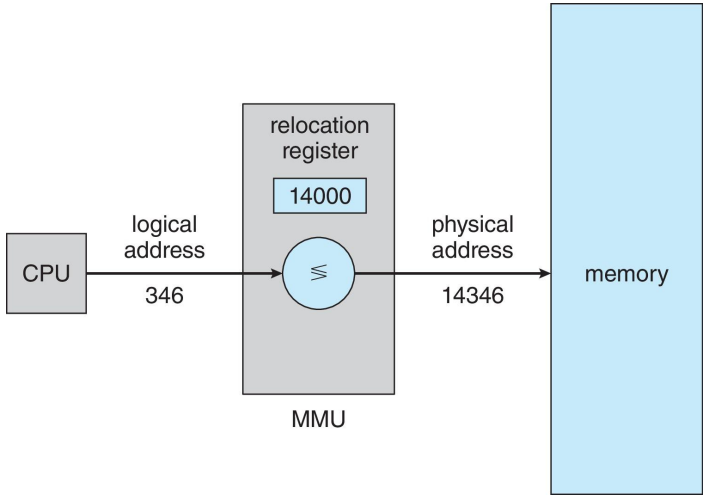
Memory-Management Unit (MMU)

- Programs load/store to **virtual addresses**
- Actual memory uses **physical addresses**
- VM Hardware is Memory Management Unit (MMU)



- Usually part of CPU core (one address space per hyperthread)
- Configured through privileged instructions (e.g., load bound reg)
- Translates from virtual to physical addresses
- Gives per-process view of memory called **address space**

Base+bound trade-offs



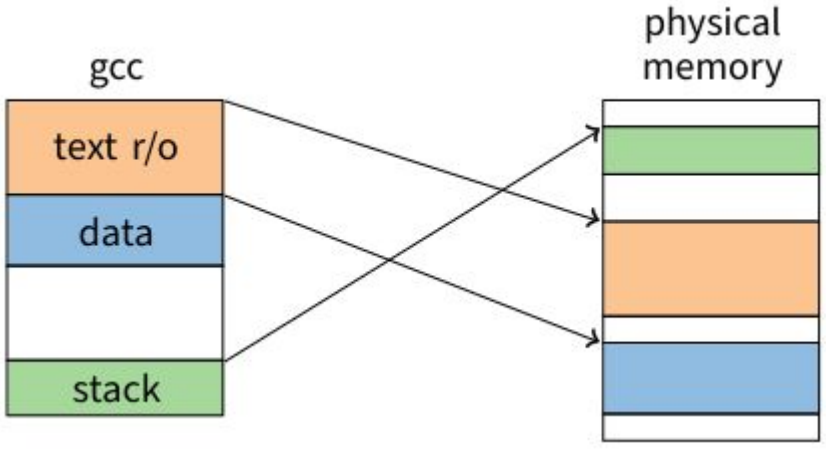
Advantages

- Cheap in terms of hardware: only two registers
- Cheap in terms of cycles: do add and compare in parallel
- Examples: Cray-1 used this scheme

Disadvantages

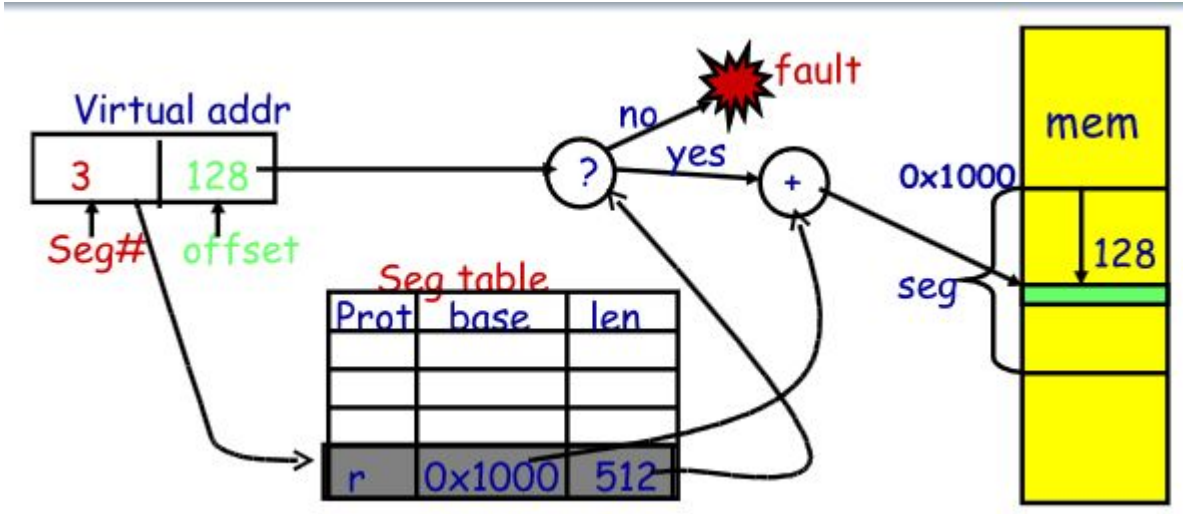
- Growing a process is expensive or impossible
- No way to share code or data (E.g., two copies of bochs, both running pintos)
- **One solution:** Multiple segments
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments

Segmentation



- Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Must specify segment as part of virtual address

Segmentation mechanics

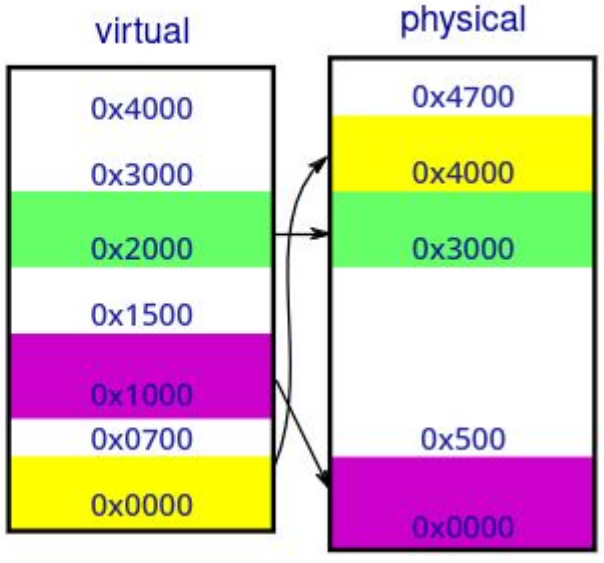


- Each process has a segment table
- Each VA indicates a segment and offset:
  - Top bits of addr select segment, low bits select offset (PDP-10)
  - Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)

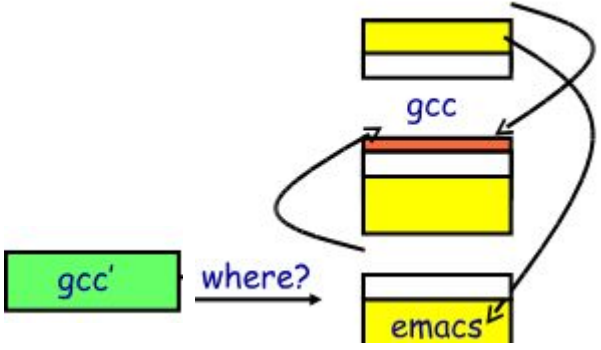
Segmentation example

- 2-bit segment number (1st digit), 12 bit offset (last 3)
- Where is 0x0240?
- 0x1108?
- 0x265c?
- 0x3002?
- 0x1600?

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



Segmentation trade-offs



Advantages

- Multiple segments per process
- Allows sharing! (how?)
- Don’t need entire process in memory

Disadvantages

- Requires translation hardware, which could limit performance
- Segments not completely transparent to program
  - e.g., default segment faster or uses shorter instruction
- n byte segment needs n contiguous bytes of physical memory
- Makes **fragmentation** a real problem.

Fragmentation

Dynamic allocation problem:

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** segment that is big enough
- **Best-fit**: Allocate the ***smallest*** segment that is big enough;
  - must search entire list, unless ordered by size
  - Produces the smallest leftover segment
- **Worst-fit**: Allocate the ***largest*** segment; must also search entire list
  - Produces the largest leftover segment
  - must search entire list, unless ordered by size

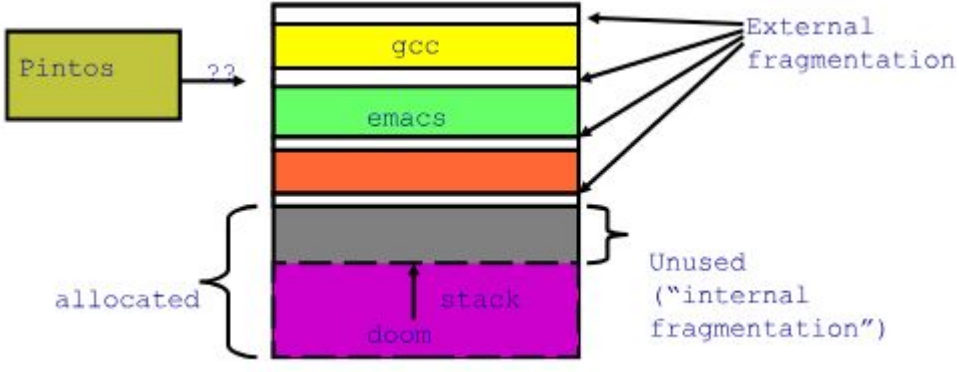
Fragmentation ==⇒ Inability to use free memory

Over time:

- Variable-sized pieces = many small holes **(external fragmentation)**
- Fixed-sized pieces = no external holes, but force internal waste **(internal fragmentation)**

First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation

1/3 may be unusable -> 50-percent rule



Alternatives to hardware MMU

Language-level protection (JavaScript)

- Single address space for different modules

- Language enforces isolation

- Singularity OS does this with C# Singularity (operating system) - Wikipedia

Software fault isolation

- Instrument compiler output

- Checks before every store operation prevents modules from trashing each other

- Google's now deprecated Native Client does this for x86 [Native Client: A Sandbox for Portable, Untrusted x86 Native Code]

- Easier to do for virtual architecture, e.g., WebAssembly

- Works really well on ARM64 [Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing]

Paging

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Map **virtual pages** to **physical pages**
  - Each process has separate mapping
  - Set up a **page table** to translate logical to physical addresses
- Keep track of all free frames
  - To run a program of size N pages, need to find N free frames and load program

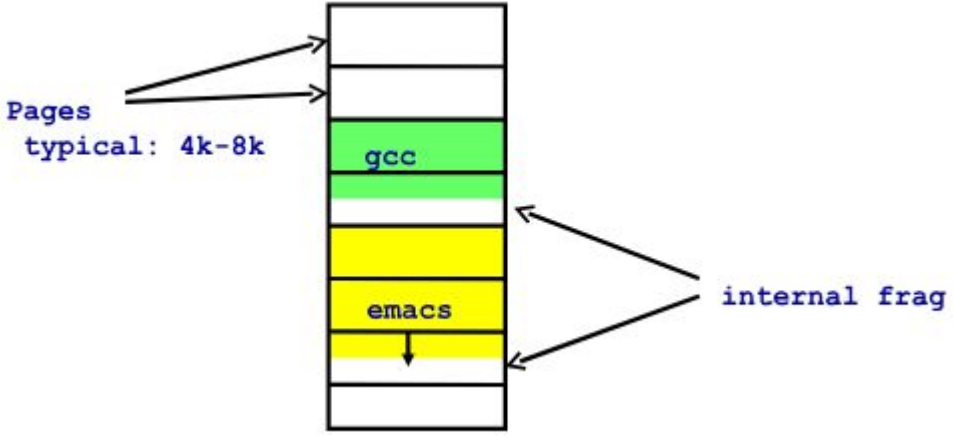
Allow OS to gain control on certain operations

- Read-only pages trap to OS on write
- Invalid pages trap to OS on read or write
- OS can change mapping and resume application

Other features sometimes found:

- Hardware can set “accessed” and “dirty” bits
- Control page execute permission separately from read/write
- Control caching or memory consistency of page

Paging trade-offs

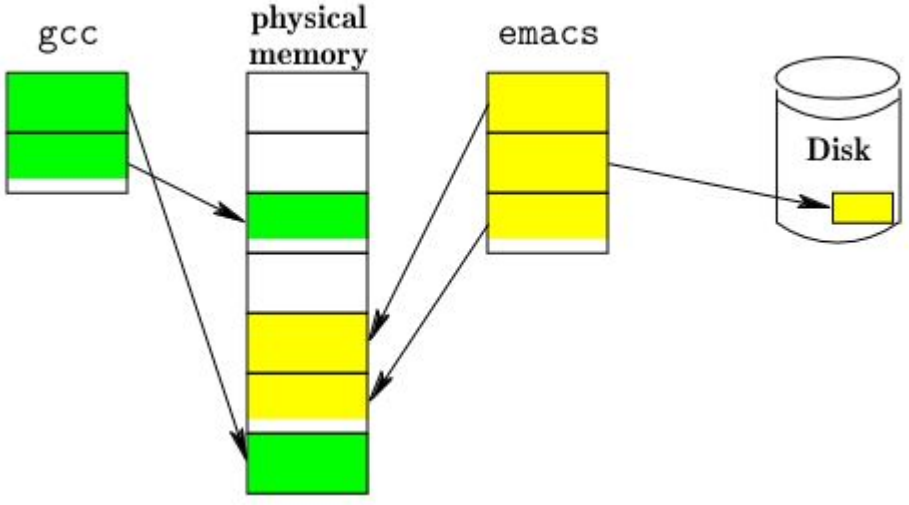


- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per “segment”

Paging simplifies allocation

Allocate any physical page to any process

Can store idle virtual pages on disk



Paging data structures

Pages are fixed size, e.g., 4 KiB

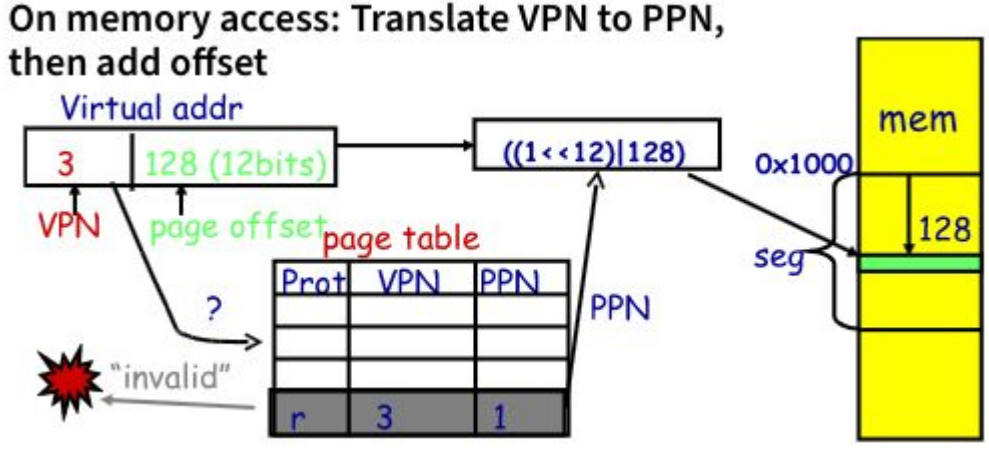
- Least significant 12 (log2 4 Ki) bits of address are page offset

- Most significant bits are page number

Each process has a page table

- Maps virtual page numbers (VPNs) to physical page numbers (PPNs)

- Also includes bits for protection, validity, etc



Examples: Paging on PDP-11

64 KiB virtual memory, 8 KiB pages

- Separate address space for instructions & data
- I.e., can’t read your own instructions with a load

Entire page table stored in registers

- 8 Instruction page translation registers
- 8 Data page translations

Swap 16 machine registers on each context switch

x86 Paging

Paging enabled by bits in a control register (%cr0)

- Only privileged OS code can manipulate control registers

Normally 4 KiB pages

%cr3: points to physical address of 4 KiB page directory

Page directory: 1024 PDEs (**page directory entries**)

- Each contains physical address of a page table

Page table: 1024 PTEs (**page table entries**)

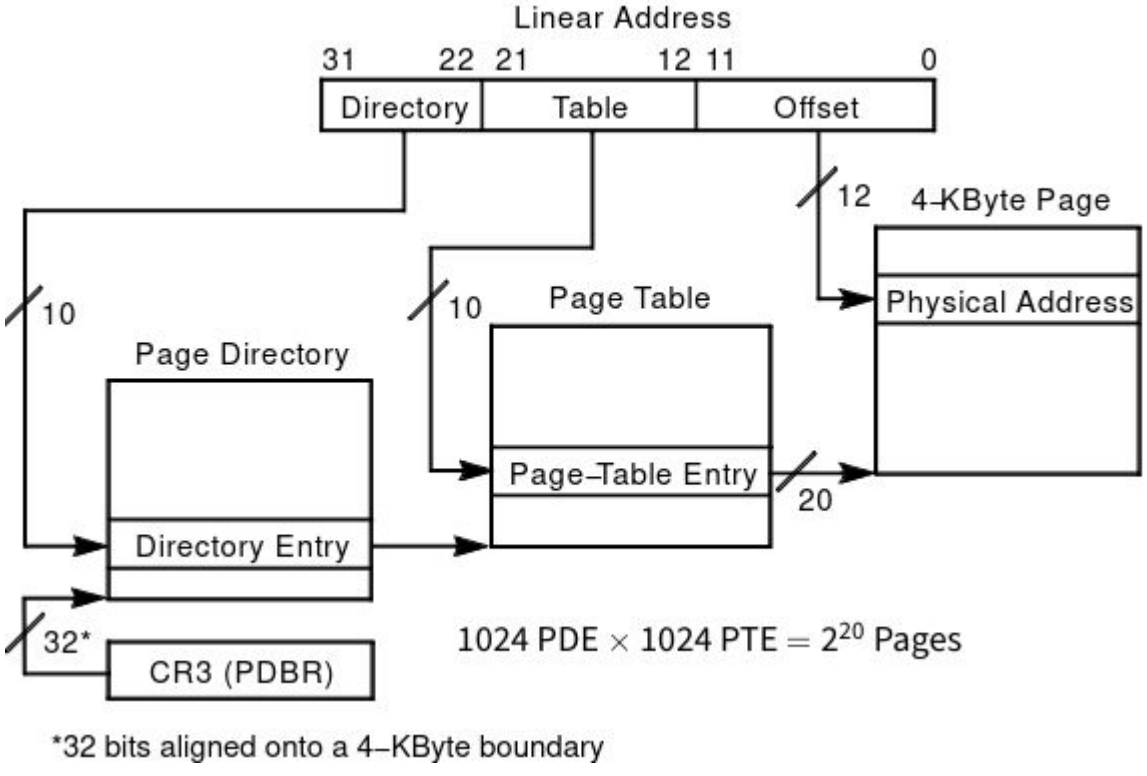
- Each contains physical address of virtual 4K page
- Page table covers 4 MiB of Virtual mem

See old intel manual for simplest explanation

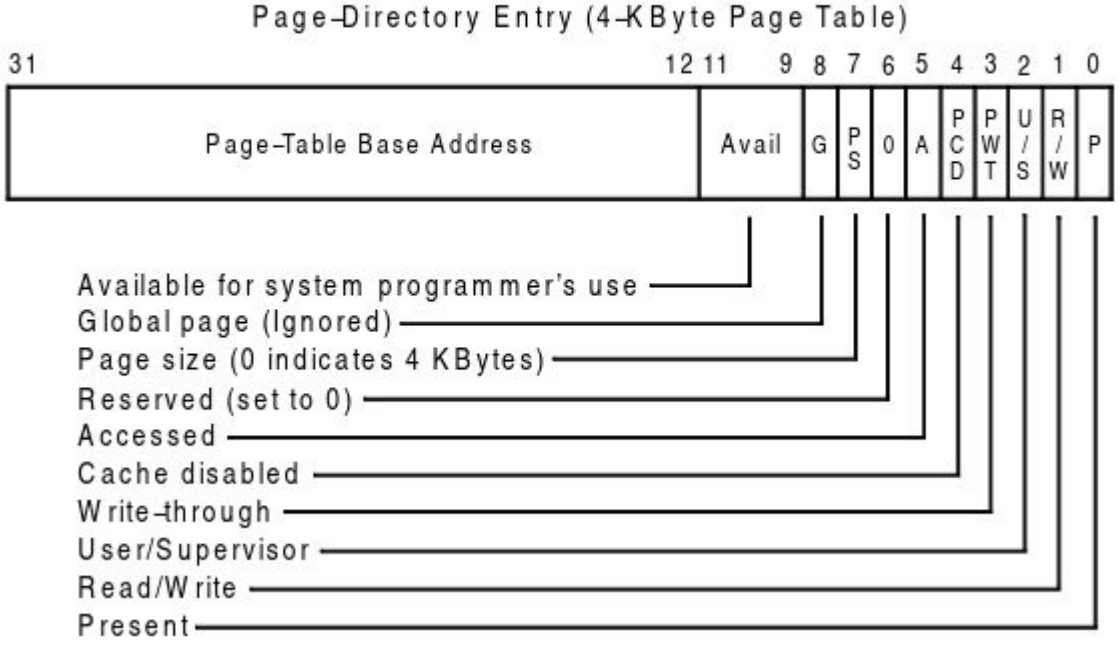
- Also volume 2 of AMD64 Architecture docs
- Also volume 3A of latest intel 64 architecture manual
- also How does x86 paging work? - Stack Overflow

x86 page translation

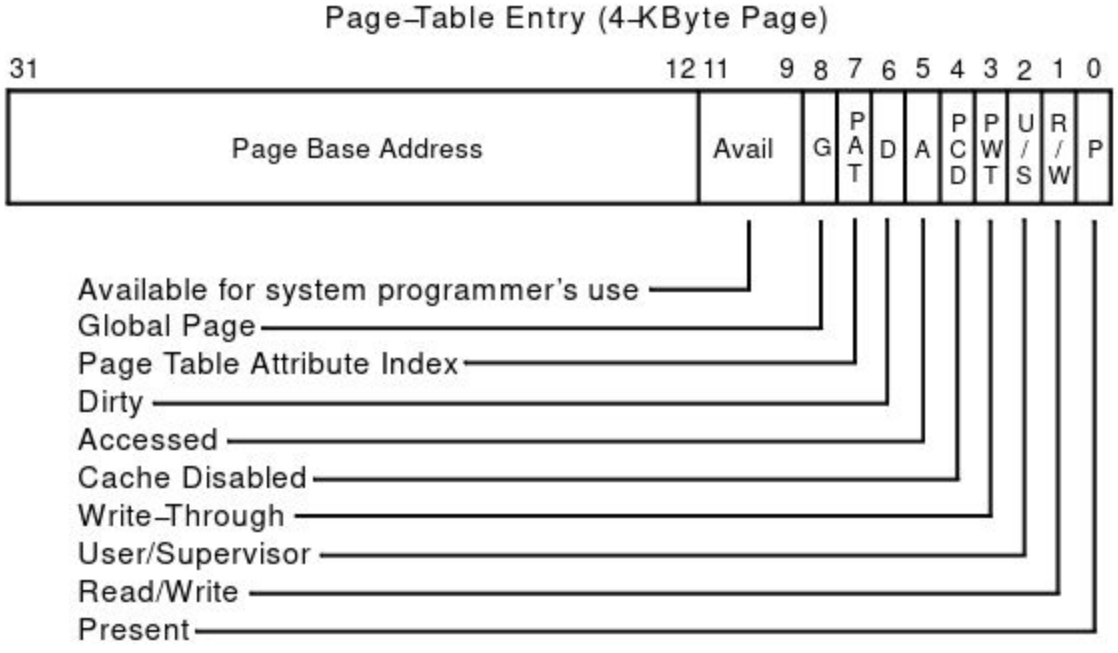
two level(Hierarchical Paging)
page table structure



x86 page directory entry



x86 page table entry



x86 hardware segmentation

x86 architecture also supports segmentation

- Segment register base + pointer val = linear address
- Page translation happens on linear addresses

Two levels of protection and translation check

- Segmentation model has four privilege levels (CPL 0–3)
- Paging only two, so 0–2 = kernel, 3 = user

Why do you want both paging and segmentation?

- Short answer: You don't – just adds overhead
  - Most OSes use "flat mode" – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support
- Long answer: Has some fringe/incidental uses
  - Keep pointer to thread-local storage w/o wasting normal register
  - 32-bit VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for W⅄X when no PTE NX bit

Making paging faster

x86 PTs require 3 memory references per load/store

- Look up page table address in page directory
- Look up physical page number (PPN) in page table
- Actually access physical page corresponding to virtual address

For speed, CPU caches recently used translations

- Called a translation lookaside buffer or **TLB**

- Typical: 64-2k entries, 4-way to fully associative, 95% hit rate

- Modern CPUs add second-level TLB with ~1,024+ entries; often separate instruction and data TLBs

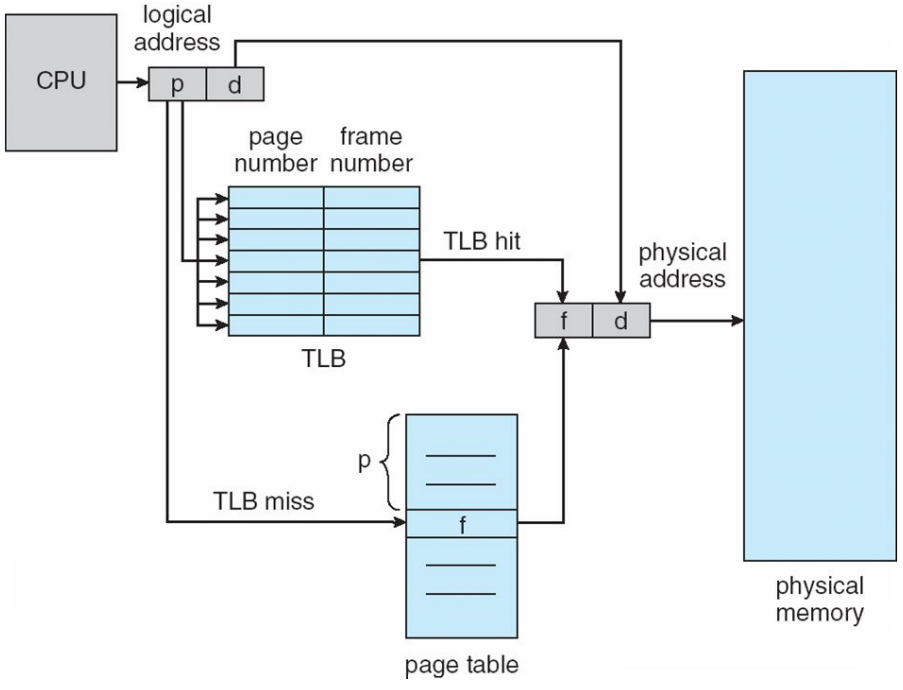
- Each TLB entry maps a VPN → PPN + protection information

On each memory reference

- Check TLB, if entry present get physical address fast

- If not, walk page tables, insert in TLB for next time (Must evict some entry)

Paging Hardware With TLB



TLB operates at CPU pipeline speed ==⇒ small, fast

Complication: what to do when switching address space?

- Flush TLB on context switch (e.g., old x86)
- Tag each entry with associated process's ID (e.g., MIPS)

In general, OS must manually keep TLB valid

Changing page table in memory won't affect cached TLB entry

- E.g., on x86 must use invlpg instruction
  - Invalidates a page translation in TLB
  - Note: very expensive instruction (100–200 cycles)
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change

More Complex on a multiprocessor (TLB shootdown)

  - Requires sending an interprocessor interrupt (IPI)
  - Remote processor must execute invlpg instruction

Paging extensions

PSE: Page size extensions

- Setting bit 7 in PDE makes a 4 MiB translation (no PT)

PAE Page address extensions

- Newer 64-bit PTE format allows 36+ bits of physical address

- Page tables, directories have only 512 entries

- Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits

- PDE bit 7 allows 2 MiB translation

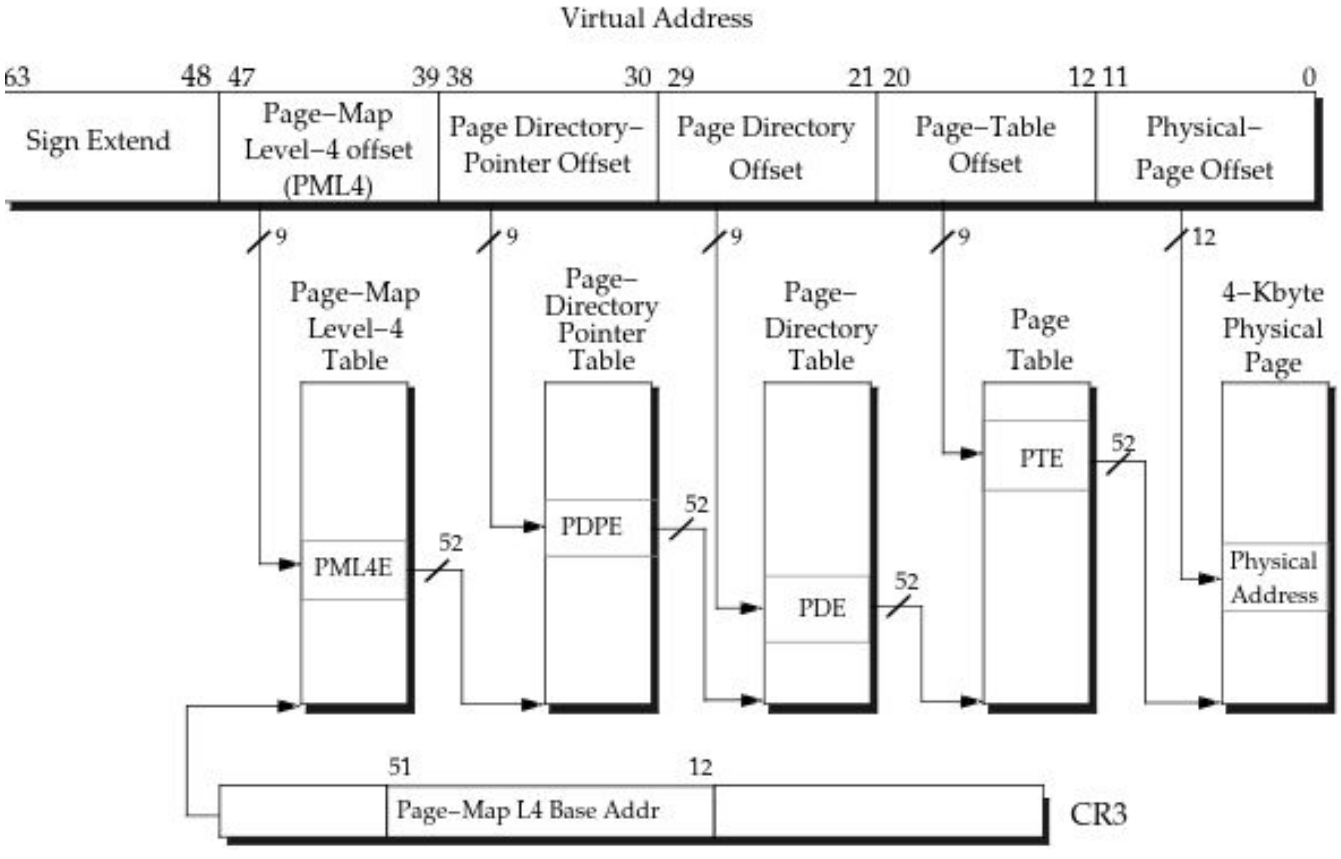
Long mode PAE (x86-64)

- In Long mode, pointers are 64-bits

- Extends PAE to map 48 bits of virtual address (next slide)

- Why are aren’t all 64 bits of VA usable?

x86 long mode paging



In Long mode, pointers are 64-bits

Extends PAE to map 48 bits of virtual address (next slide)

Why are aren't all 64 bits of VA usable?

Where does the OS live?

In its own address space?

- Can’t do this on most hardware (e.g., syscall instruction won’t switch address spaces)
- Also would make it harder to parse syscall arguments passed as pointers

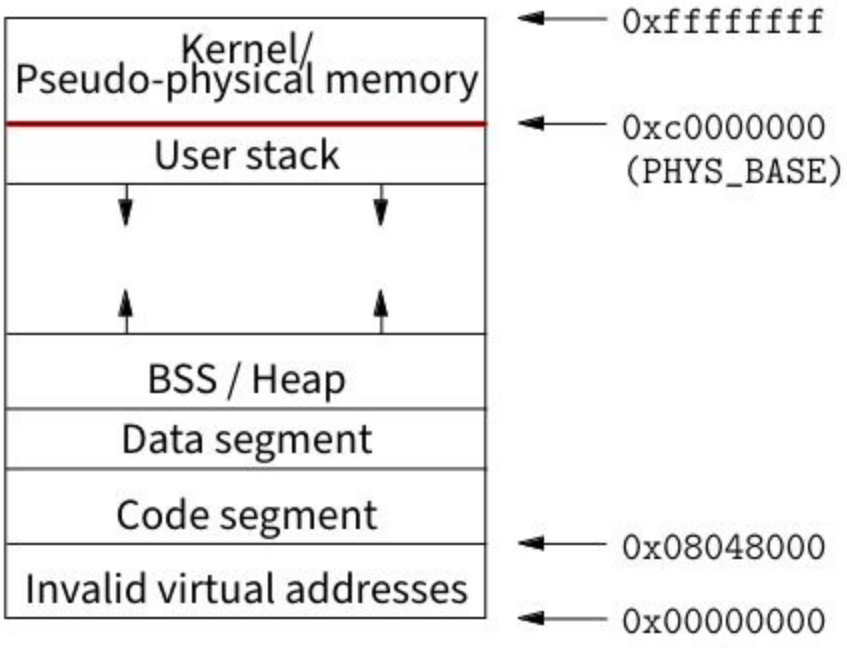
So in the same address space as process

- Use protection bits to prohibit user code from writing kernel

Typically all kernel text, most data at same VA in every address space

- On x86, must manually set up page tables for this
- Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
- Some hardware puts physical memory (kernel-only) somewhere in virtual address space
- Typically kernel goes in high memory; with signed numbers, can mean small negative addresses (small linker relocations)

Pintos memory layout



Paging in day to day use

- Demand paging
- Growing the stack
- BSS page allocation
- Shared text
- Shared libraries
- Shared memory
- Copy-on-write (fork, mmap, etc.)

Q: Which pages should have global bit set on x86?

- it controls the scope of the physical page mapped by this entry.
- Think it as an optimization for system calls
  - kernel maps its pages global so no TLB flushes
- Shared memory for two process: when it does not invalidate TLB.
  - kernel can use G==1 for both processes

More details of some of the terms

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
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

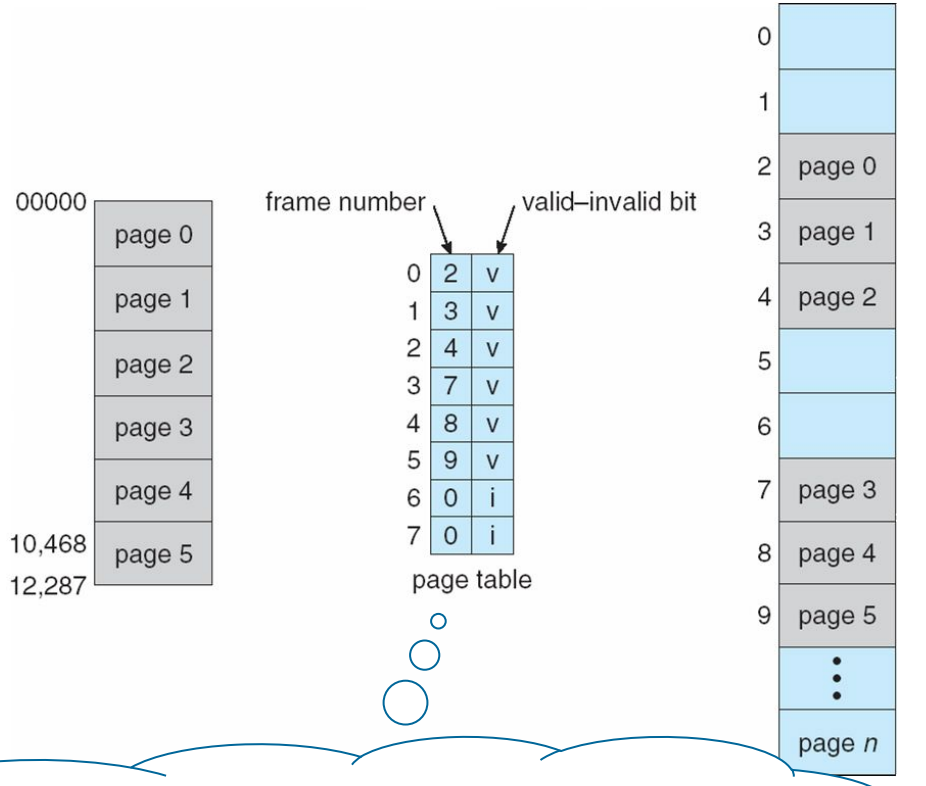
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

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

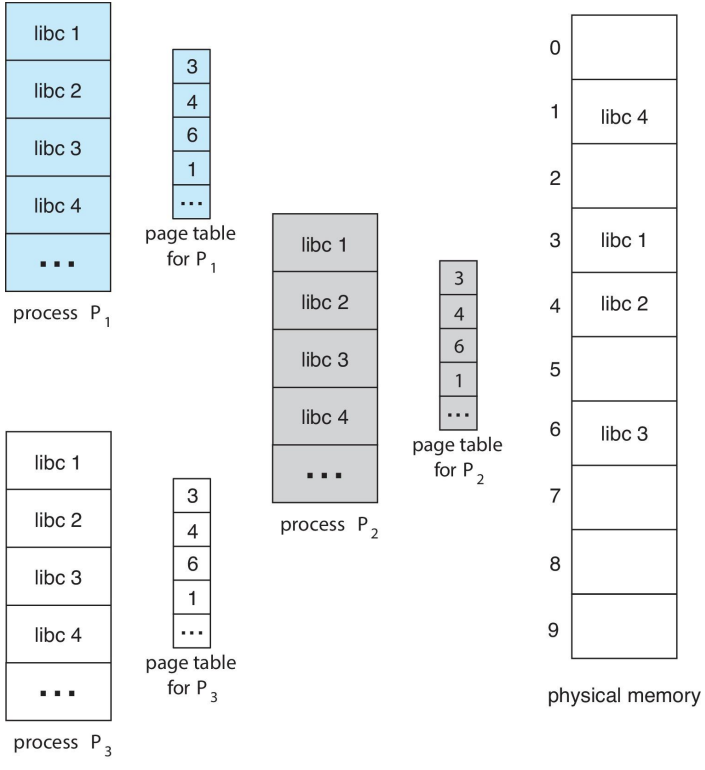


- in demand paging, valid/invalid also indicates the pages not in **main memory**!

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

Shared Pages Example

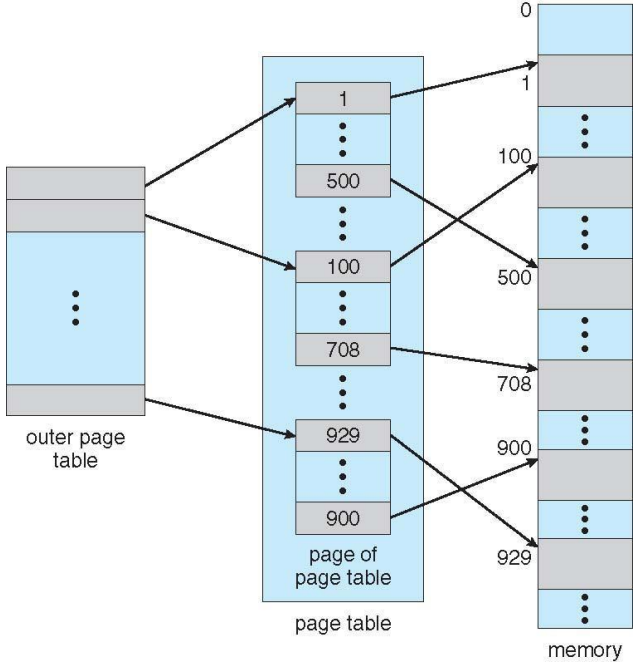


Structure of the Page Table

- 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, each process 4 MB of physical address space for the page table alone
    - Don’t want to allocate that contiguously in main memory
- One simple solution is to divide the page table into smaller units
  - Hierarchical Paging**(we have seen this)**
  - Hashed Page Tables
  - Inverted Page Tables

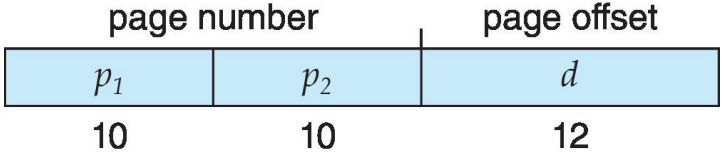
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



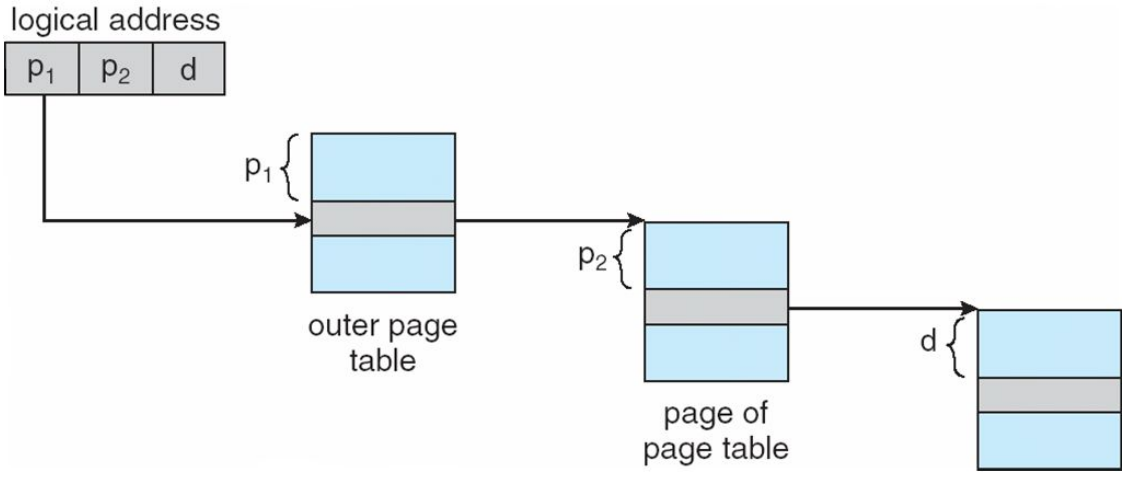
Two-Level Paging Example

- 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
- 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:



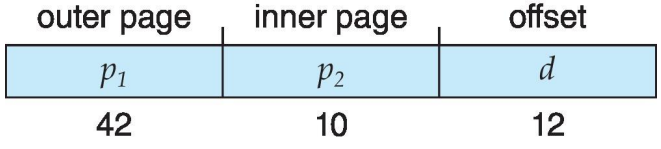
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table[2]
- Known as **forward-mapped page table**

Address-Translation Scheme



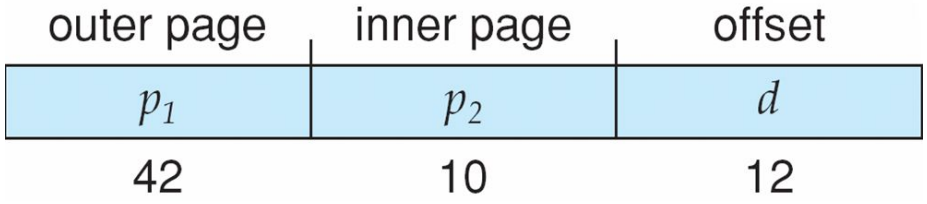
64-bit Logical Address Space

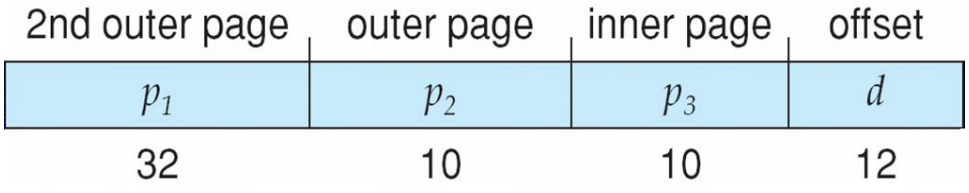
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    4 And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

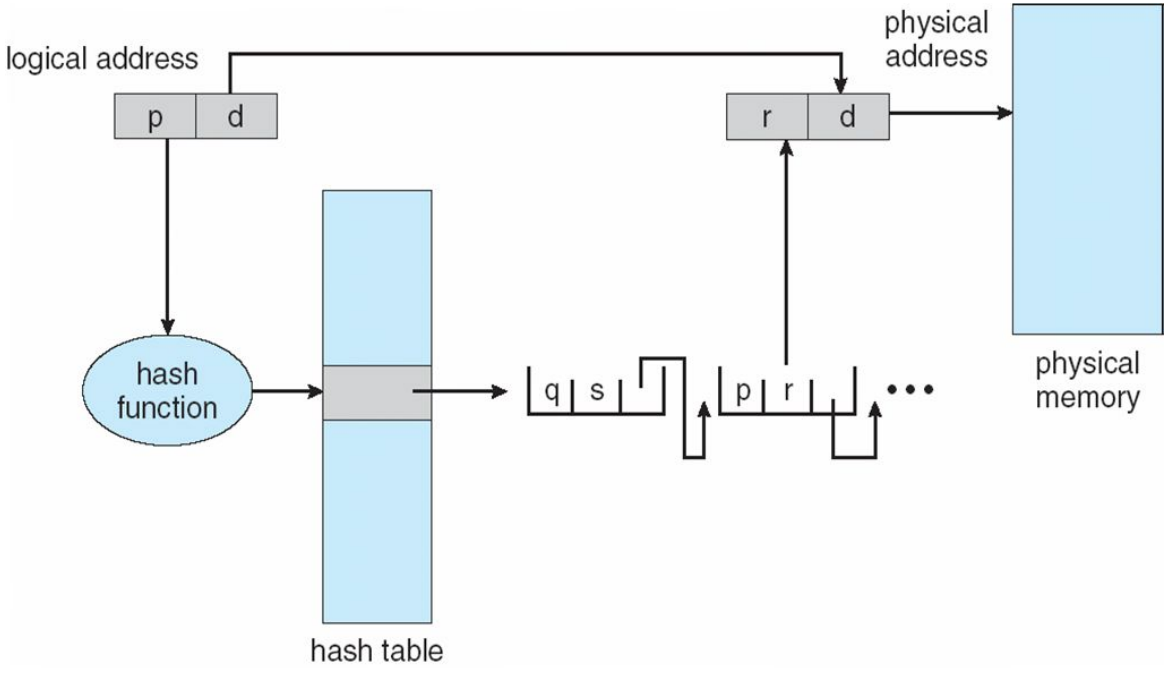




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual 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 virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

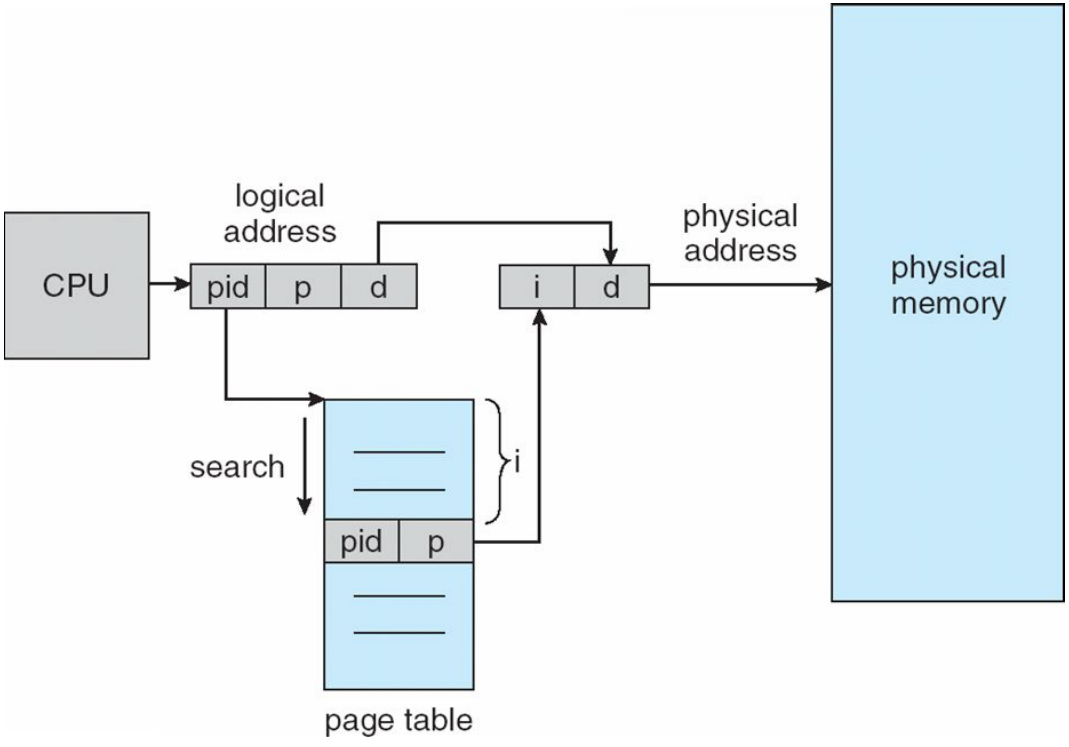
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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, but 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



Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

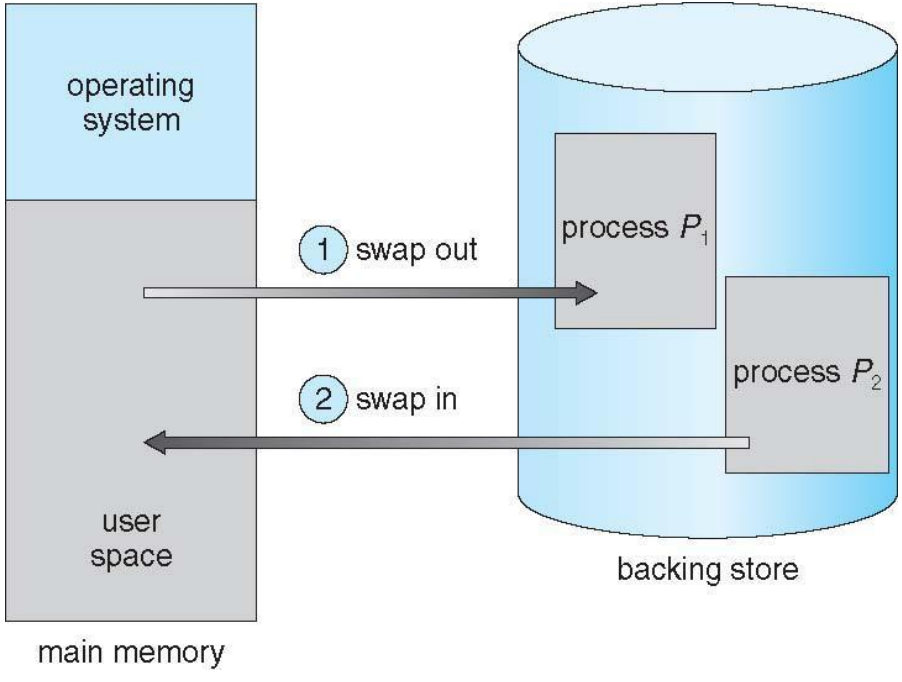
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of 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 2000 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
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

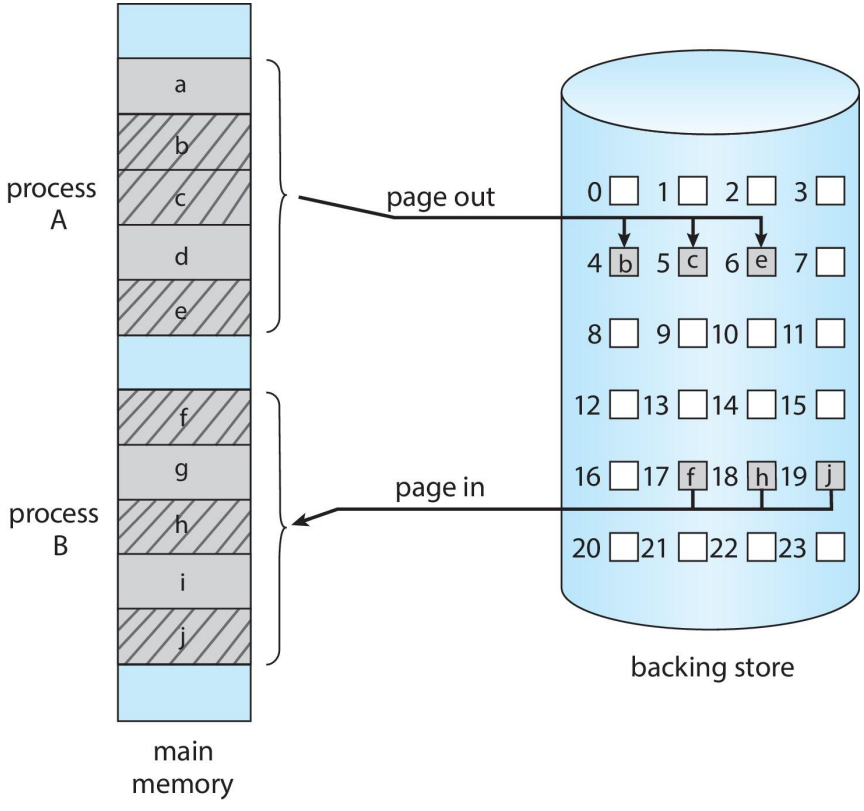
Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

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
  - Both OSes support paging as discussed below

Swapping with Paging



Example: The Intel 32 and 64-bit Architectures

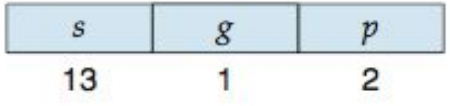
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

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**))

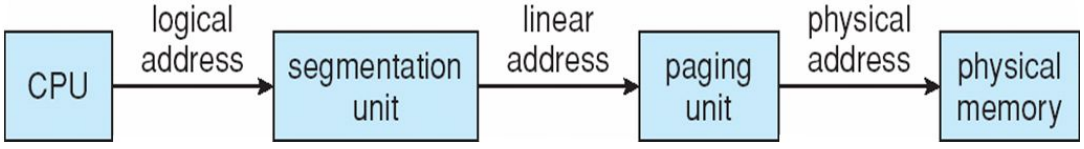
Example: The Intel IA-32 Architecture (Cont.)

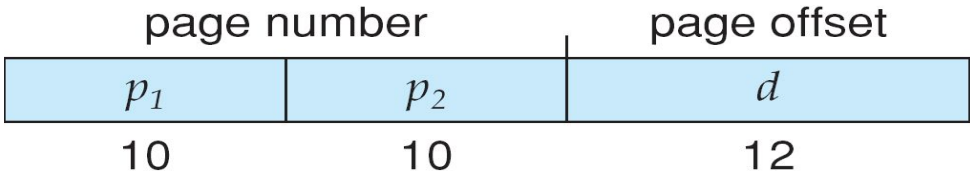
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



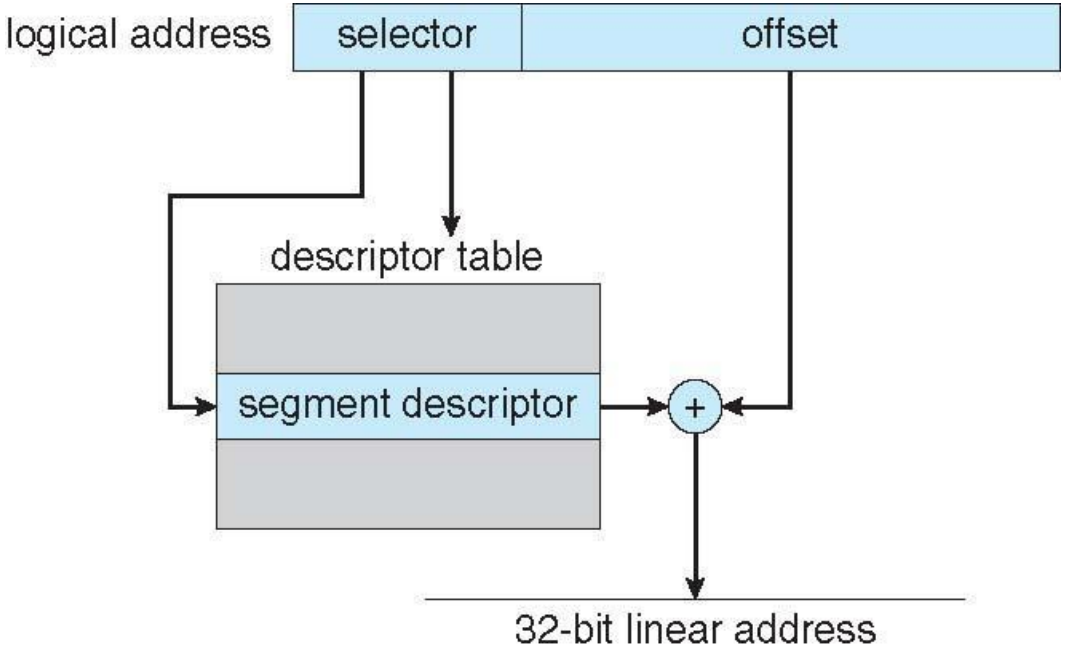
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB
    -

Logical to Physical Address Translation in IA-32

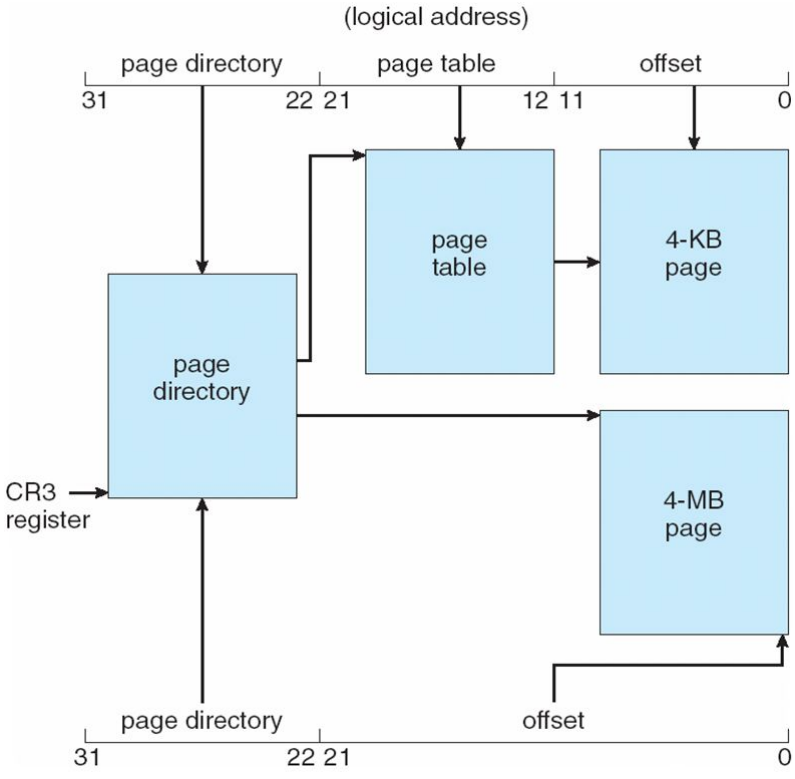




Intel IA-32 Segmentation

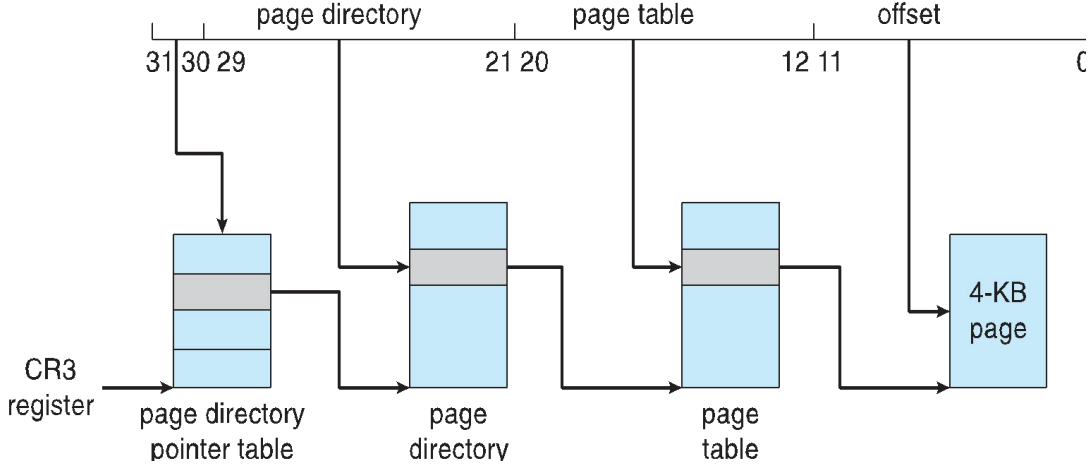


Intel IA-32 Paging Architecture



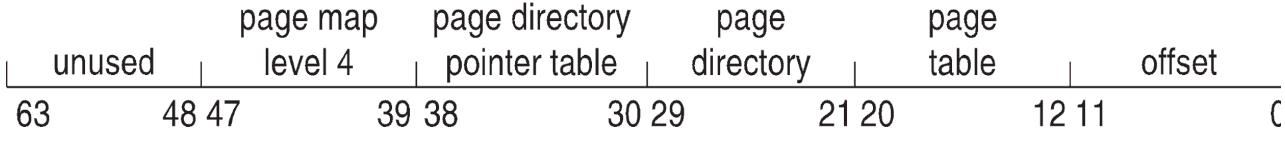
Intel IA-32 Page Address Extensions

- 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



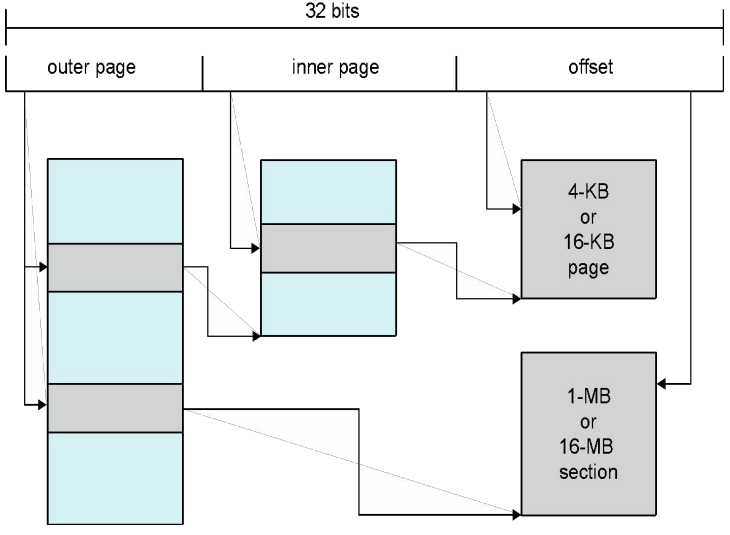
Intel x86-64

- 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



Example: ARM 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



End of Chapter 9