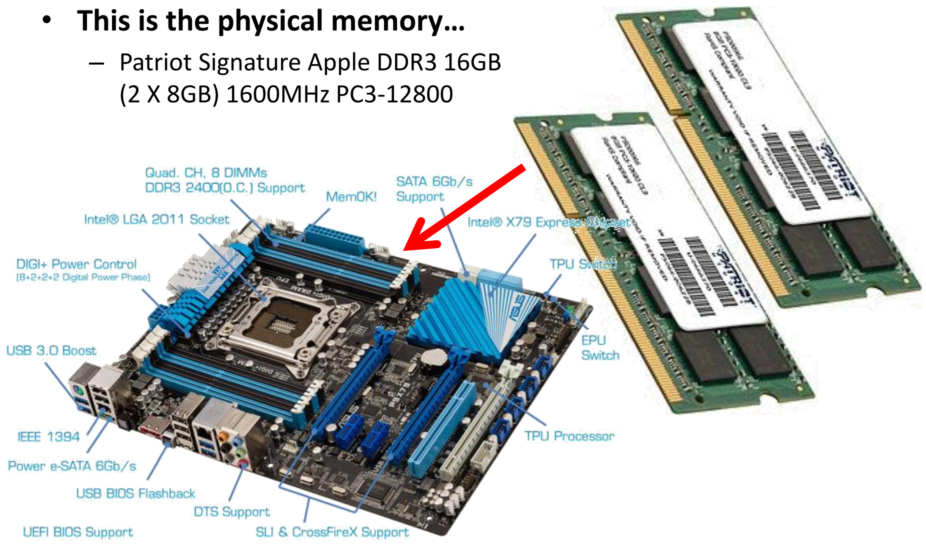
Operating Systems (234123)

Virtual memory

Dan Tsafrir (19/5/2014, 26/5/2014, 2/6/2014)
Partially based on slides by Hagit Attiya

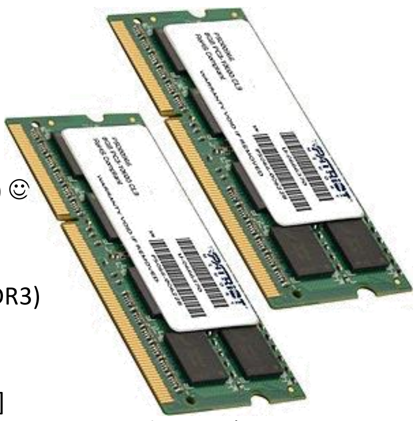
DRAM (dynamic random-access memory)

- **This is the physical memory…**
  - Patriot Signature Apple DDR3 16GB (2 X 8GB) 1600MHz PC3-12800



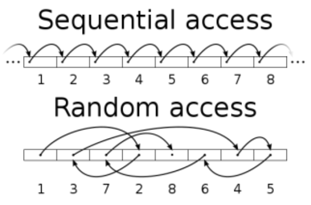
DRAM (dynamic random-access memory)

- **This is the physical memory…**
  - Patriot Signature Apple DDR3 16GB (2 X 8GB) 1600MHz PC3-12800
  - Price
    - May 2013: $134 (~ $8 per 1 GB)
    - May 2014: $159 (~ $10 per 1GB) ☺
  - Bandwidth
    - 1600 (MHz; transfers per sec)
    - x 64 (bit per bus transfer, for DDR3)
    - / 8 (bits per byte)
    - = 1600 x 10^6 x 64 / 8 [B/sec]
    - = 12,800 MB/sec = 12.8 [GB/sec] (= meaning of the "PC3-12800"; GB=10^9, in this case)
  - Latency
    - A handful of 100s of CPU cycles (≈ 100s of ns)
    - (Assuming not cached)



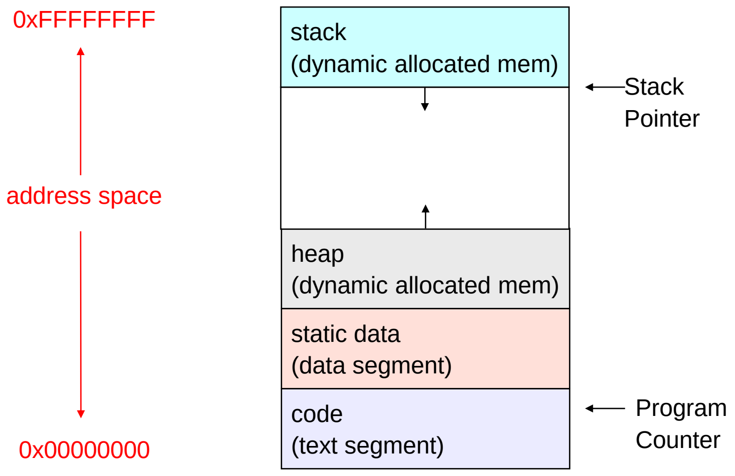
Reminder: in comparison to HDD…

	HDD (hard disk drive)	DRAM	times faster / better
availability	non-volatile	volatile	
sequential access bandwidth	≈ 100 MB/s	≈ 10 GB/s	> x100
random access bandwidth (reading 4KB chunks at a time)	≈ 1 MB/s	≈ 10 GB/s	> x10,000
latency	≈ 5 ms	≈ 100 ns	> x50,000



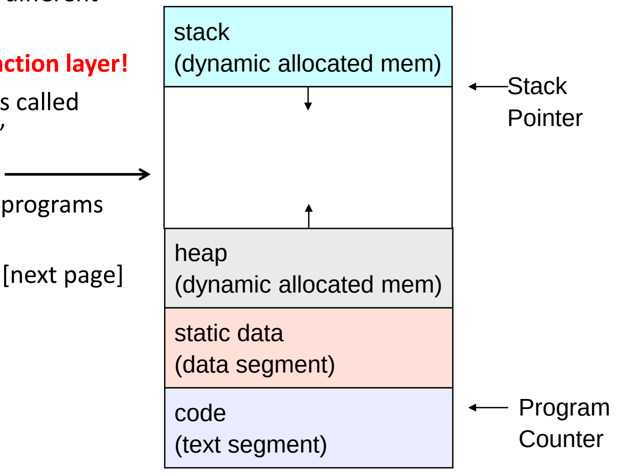
Reminder

- **This is how a program believes its memory looks like…**



But how can this be?

- **It's impossible for all processes to share these addresses**
  - ⇒ They translate to different locations, so…
  - ⇒ **There's an abstraction layer!**
  - ⇒ This abstraction is called "**virtual memory**"
    - Provides this illusion to all programs
    - And more. Specifically… [next page]



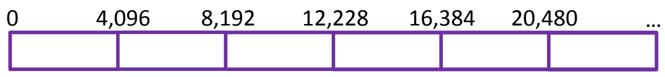
Virtual memory (vmem) – motivation

- **Per-program Illusion of contiguous memory**
  - Programmers need not worry about *where* data is placed exactly
  - The use the ideal address space from 2 slides ago
- **Isolation between processes**
  - Processes can concurrently run on the same processor
  - Yet vmem prevents them from accessing the memory of one another
  - (But still allows for convenient sharing when required)
- **Dynamic growth**
  - Can add memory to process's heap/stack at runtime, as needed
- **Illusion of large memory => memory overcommitment**
  - DRAM often: (i) one of the most costly parts; (ii) the bottleneck resource
  - Vmem size can be bigger than physical memory size
  - Sum of vmem spaces (across all processes) can be >= physical
- **Access control**
  - Decide if individual memory chunks can be read / written / executed

HOW VMEM WORKS, IN PRINCIPLE

Virtual memory – terminology

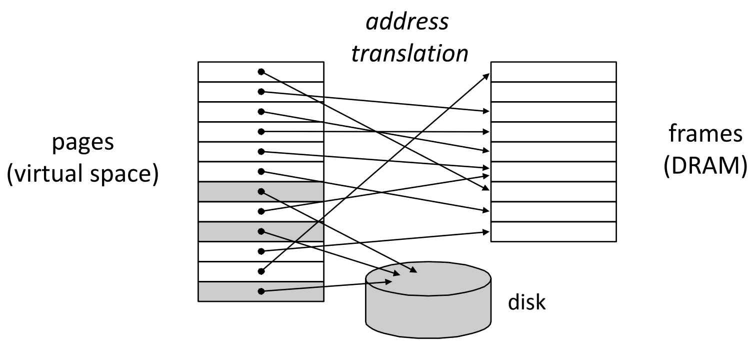
- **Virtual address (VA)**
  - Used by the program/programmer
  - “Ideal” = contagious & as big is we’d like
- **Physical address (PA)**
  - The real, underlying physical memory address
  - Completely abstracted away by OS/HW
- **Memory (virtual & physical) is divided into fixed size blocks**
  - “**Page**” = chunk of contagious data (in virtual or physical space)
  - “**Frame**” = physical memory exactly big enough to hold one page
  - |page| = |frame| = 2^k (bytes)
  - Typically, k = 12, namely a page (and frame) size is 4KB
- **Pages & frames are always aligned on 4KB boundaries**
  - Both in physical and virtual memory spaces: 0, 4KB, 8KB, 12KB, 16KB, …



Virtual memory – basic idea

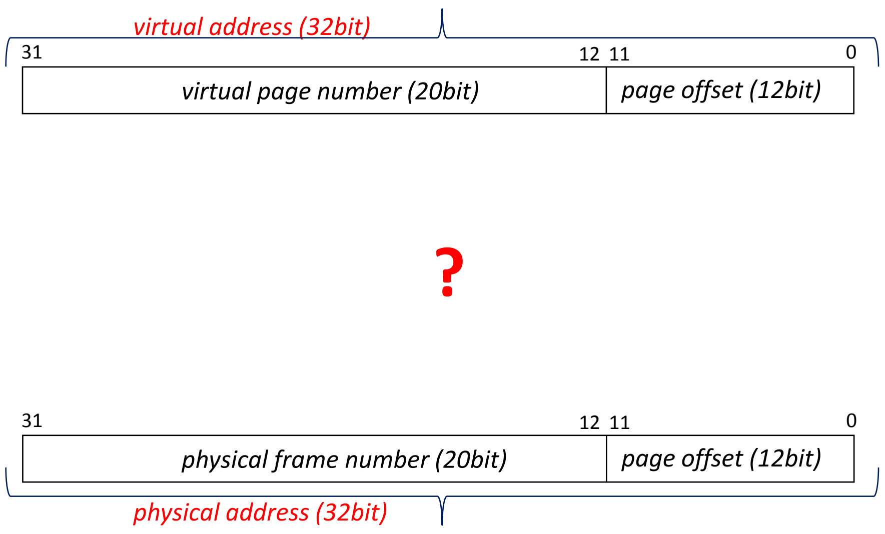
- **“Map” pages to frames, such that VA space is contiguous**
  - Pages can be “mapped” into (associated with) arbitrary frames at arbitrary locations
- **Pages can reside in memory, or on disk**
  - Hence, we achieved the aforementioned memory **overcommitment**
- **All programs are written using VAs**
  - And somehow VAs are seamlessly translated into PAs
  - As we will see later, **translation is a HW/SW mechanism**, whereby
    - OS sets the VA=>PA mappings
    - HW does on-the-fly translation from VA to PA

Per-process virtual memory simplistic illustration

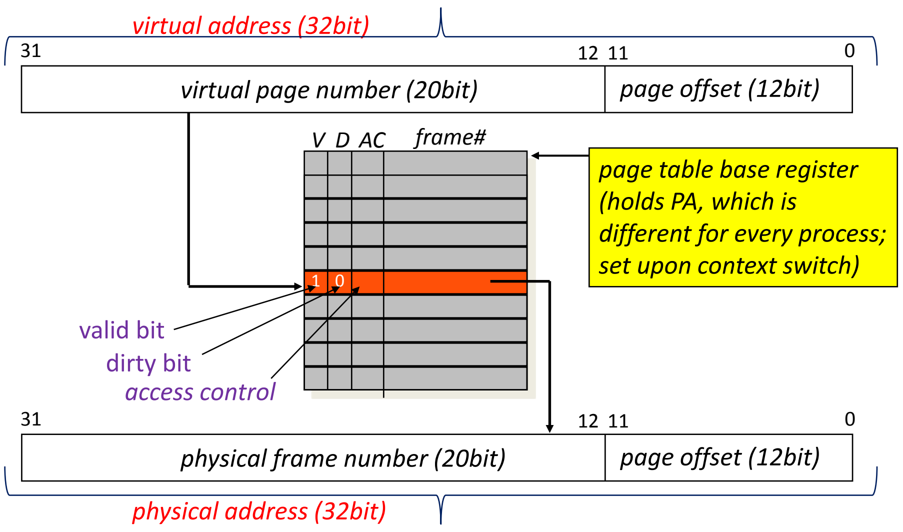


- **Memory may serve as a “cache” for secondary storage (disk);**
- **Immediate advantages**
  1. Illusion of contiguity & of having more physical memory
  2. Program’s actual location unimportant
  3. Dynamic growth, isolation, & sharing are easy to obtain

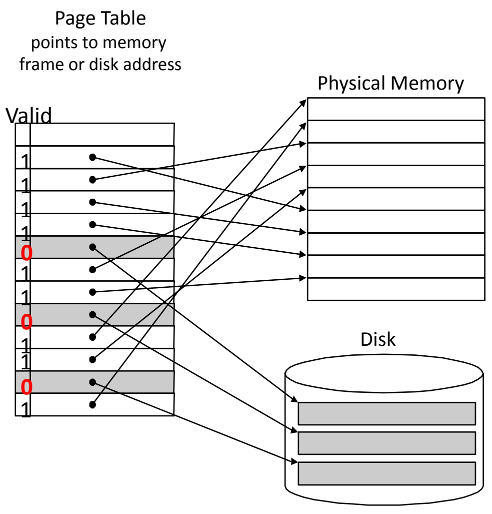
How to map (assume 32bit address)?



Use per-process "page table" (in DRAM)


virtual address (32bit)
31
12 11
0
virtual page number (20bit)
page offset (12bit)
V D AC frame#
1 0
page table base register (holds PA, which is different for every process; set upon context switch)
valid bit
dirty bit
access control
31
12 11
0
physical frame number (20bit)
page offset (12bit)
physical address (32bit)

Per-process page table



Upon (each & every) memory access

- **If (valid == 1)**
  - Page is in main memory @ PA stored in table
  - Data is readily available & can be used
- **else**
  - Suspend process
  - Fetch page from disk
  - Resume process, which will re-execute faulting instruction
    - Now it'll succeed
- **Access Control**
  - R=read-only, R/W=read/write, X=execute
  - If access type incompatible with specified access rights
    $\Rightarrow$ protection violation fault $\Rightarrow$ interrupt $\Rightarrow$ signal

Major page fault

- **Page not in memory $\Rightarrow$ need to retrieve it from disk**
  1. CPU detects the situation (valid bit = 0)
     - But it cannot remedy the situation on its own
     - CPU doesn't communicate with disks; OS does that
  2. CPU generates interrupt and transfers control to the OS
     - Invoking the OS page-fault handler
  3. OS regains control & initiates I/O read operations
     - To read missing page from disk to DRAM
     - Possibly need to write victim page to disk (if no room & dirty)
  4. OS suspends process & context switches to another
     - It will take milliseconds for I/O operations to complete
  5. Upon read completion, OS makes suspended process runnable again
     - It'll soon be chosen for execution
  6. When process is resumed, faulting operation is re-executed
     - Now it will succeed because the page is there

Minor page fault

end of 1st lecture

- **Not all page faults are major…**
  - That is, not all page faults take milliseconds to handle
- **Sometimes the info accessed by the program is in DRAM**
  1. Yet the 'valid' bit is off, or
  2. The 'valid' bit is on, but there's an access violation
- **For example**
  1. A memory page is found in the "buffer cache" of the OS
     - Caching previously read files, possibly by other processes
     - All read/write ops of all processes go through the buffer cache
  2. COW (copy-on-write)
     - Used for, e.g., implementing fork()
     - In child, map all pages to parent memory space, but for reading
     - When child writes => page fault => create a private copy for it
- **Minor page faults are much faster to handle**
  - Processes usually not suspended due to minor page faults

Related concepts

- **"Page in" & "page out" a chunk of data**
  - Page in ⇔ copy page from disk to DRAM (= read)
  - Page out ⇔ copy page from DRAM to disk (=write)
- **The mmap() system call**
  - (Mmap = memory map)
  - Map a given file into the virtual memory space of the process
    - A file becomes an "array of bytes" (backed by disk), and gets a VA
  - With the right mmap() flags, reading/writing from/to the array translates to reading/writing from/to the file
  - A mmap()ed file is said to be a "memory-mapped file"

Related concepts

- **Anonymous pages**
  - Heap/stack pages; not file
  - Can be allocated by mmap (MAP_ANONYMOUS)
    - Namely, can implement malloc with mmap, which can be invoked whenever malloc runs out of memory
    - Typically, however, malloc is implemented using the system calls brk() and sbrk()
- **Named pages**
  - Backed by a file (via mmap)
- **The mmap system call can also map file pages "privately"**
  - Using the MAP_PRIVATE flag
  - Meaning, if the page is changed, this doesn't affect the underlying file
    - There will be a copy-on-write

Related concepts

- **Q/A**
  - Q: do we need to page out named pages?
    A: only if they are "dirty"
  - Q: when we read() a file page (with the read syscall), is it named?
    A: no, as changes to the memory buffer will not affect the file;
    need to write() to the file in order to make a change
  - Q: can anonymous pages reside on disk?
    A: yes [see next slide]

Related concepts

- **Swap space**
  - Disk area (file) where anonymous pages are written, if the OS decides they have no room in DRAM
  - Page is said to be “swapped out” when this occurs (and “swapped in” for the reverse operation)
  - Swap area contains anonymous pages (including mmap anonymity)
- **Swapping vs. paging**
  - In the olden days, “swapping in/out” referred to the entire memory of a process (not just to a certain page)
  - Nowadays people typically use them interchangeably
  - http://en.wikipedia.org/wiki/Paging makes the following distinction
    - Page in = transfer page from anywhere on disk to DRAM
    - Swap in = transfer a page from swap-space to DRAM
    - But we do not use this distinction

Related concepts

- **Demand-paging**
  - OS reads a page from disk into DRAM only if the process attempts to access it (and, hence, a page fault occurs)
  - That is, OS pages data in only via page faults (+ prefetching)
  - It follows that a process begins execution with most of its pages not residing in physical memory, and page faults occur until most of the data it uses is located in DRAM
  - Also called “lazy” loading
  - Q: what’s the benefit?
    A: reading is costly, & with demand-paging we only read what we need
- **Readahead prefetching (anticipatory paging)**
  - read() does prefetching when identifying sequential access
  - Page fault handler does the same
  - Complements demand-paging in an attempt to minimize page faults

Related concepts

- **Working set**
  - $WS_P(k)$ = Pages accessed by P in the last k accesses

time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
page	2	6	1	5	7	7	7	7	5	1	6	2	3	4	4	4	3
WS(k=3)	2	6 2	1 6 2	5 1 6	7 5 1	7 5	7	7	5 7	1 5 7	6 1 5	2 6 1	3 2 6	4 3 2	4 3	4	3 4

  - For a fixed value of k, smaller $WS_P(k)$ indicates more locality
  - If the current working set is not found in memory then…
- **Thrashing**
  - When we've overcommitted too much memory and there isn't enough physical memory, the system might enter a state of thrashing, that is
  - Virtual memory is in a constant state of paging, rapidly exchanging data between memory & disk
  - Nearly nothing else is done in the system (causes performance to degrade or collapse)

Did we achieve our goals?

- **Illusion of contiguous memory**
  - Yes: virtual memory is contiguous by definition
- **Illusion of large memory (possibly bigger than physical mem)**
  - Yes: chunks that don't fit into physical memory reside on disk
- **Dynamic growth**
  - Yes: heap & stack can grow at runtime by mapping more VAs to PAs, in order
- **Isolation between processes**
  - Yes: same VAs point to different PAs; as long as we keep PAs disjoint on a per-process basis, the processes are isolated
- **Memory overcommitment**
  - Yes: using the disk, $\sum_{i=1}^{n} vmem_i$ (for n processes) can be > physical size
- **Access control**
  - Yes: HW enforces PTE (= page table entry) bits; e.g., it will reject a write to a page that is marked 'read-only'

But how does it perform?

- **(1) Temporal locality helps ("temporal" relates to time)**
  - Typically, during a given time interval, a process uses only a fraction of its memory, over and over again
  - So it's fine to keep currently unused parts on disk
  - As long as the working set is mostly in memory
- **(2) Asynchronous I/O helps**
  - Writes are non-blocking: when writing a page to disk, we don't need to block the associated process
  - When reading stuff, we can run other processes
- **(3) Demand-paging helps**
  - Pages fetched from secondary memory only upon the first page fault, rather then, e.g., upon file open (we bring only what we need)

But how does it perform?

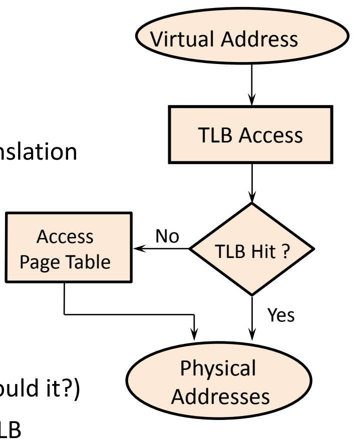
- **(4) Special locality helps ("special" relates to space)**
  - When reading page P from disk, we typically employ the readahead optimization, namely, we also bring some additional file pages that come immediately after P, even though they weren't accessed yet
  - As we've learned, "locality principle" suggests these pages would be used soon
    - Locality = special locality + temporal locality
    - Locality principle contends that most programs exhibit special and temporal locality when utilizing the memory
  - So some pages don't induce a page fault when accessed for the first time and are simply there

But how does it perform?

- **(5) Making VA=>PA translations fast helps**
  - The TLB (translation lookaside buffer) is a very small, very fast HW structure that caches recently used VA=>PA mappings
  - (e.g., 32-64-128 entries)
  - Given a VA, HW first searches for its translation in the TLB; and only if it's not there HW access the in-memory page table
  - Accessing the TLB takes very little time (e.g., a cycle)
  - Even though the TLB is very small, locality principle typically ensures it is rather effective
    - Special locality: since we work in 4KB page granularity, lots of nearby accesses fall in the same page
    - Temporal locality: same pages are used repeatedly
  - Lots of workloads (though certainly not all) approach 100% TLB hit rate

VA => PA translation with TLB

- **Page table resides in memory**
  - Each translation requires a memory access
  - Might be required for each load/store!
- **TLB**
  - Cache recently used PTEs => speeds up translation
  - TLB access time comparable to L1
  - Typically:
    - HW fills TLB automatically by reading the page table on its own (no SW involvement)
    - OS can invalidate TLB entries (when should it?)
    - Processes are completely unaware of TLB



But how does it perform?

- **(6) Using an intelligent page replacement policy helps**
  - When we need to evict a page from memory to disk
    - Page replacement policy decides which page it'll be
  - Also called "page reclamation"
  - Goal
    - Minimize number of future page faults
    - Minimize price of paging (evicting dirty pages costs more)
  - Typically done via a daemon process ("swapper") that runs in the background
    - Start: when number of free frames drops below some threshold
    - Stop: when number of free frames exceeds some threshold

Page reclamation algorithms

- Belady (optimal off-line; if we know the future; theoretical)
  - Greedily page out page accessed furthest in the future
- Could be FIFO (first-in first-out)
  - Simplest (no need to update upon each mem ref), but ignores usage
- LRU (least recently used) is better
  - But typically too wasteful (updated upon each mem ref)
- Second-chance
  - Set per-page "was it referenced?" bit
    - Can be done by HW or SW (how?)
  - Page out pages with bit=0 only, FIFO order
  - When traversed, if bit=1, set it to 0, and push the associated page to end of the list (in FIFO terms, page becomes newest)
- Clock
  - More efficient/popular variant of second-chance
  - Pages are cyclically ordered (no FIFO); search clockwise for first page with bit=0; set bit=0 for pages that have bit=1; periodically turn off 1s

Page reclamation algorithms

- **NRU (not recently used)**
  - More sophisticated LRU approximation
  - HW or SW maintains per-page ‘referenced’ & ‘modified’ bits
  - Periodically (a few clock interrupts), SW turns ‘referenced’ off
    - Though if the modified bit is on, it stays on
  - Replacement algorithm partitions pages to
    - Class 0: not referenced, not modified
    - Class 1: not referenced, modified
    - Class 2: referenced, not modified
    - Class 3: referenced, modified
  - Choose at random a page from the lowest class for removal
  - Underlying principles (order is important):
    - Prefer keeping referenced over unreferenced
    - Prefer keeping modified over unmodified

Hierarchical translation (radix tree)

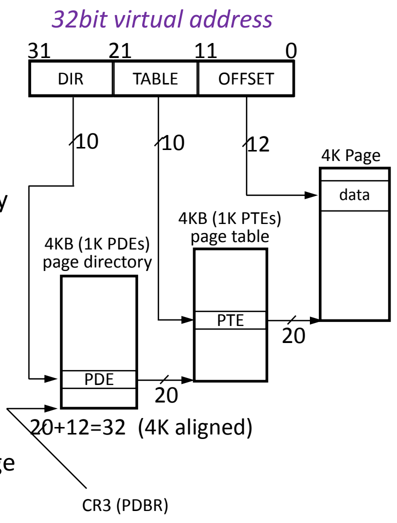
HOW IT WORKS IN X86

32bit x86 address translation

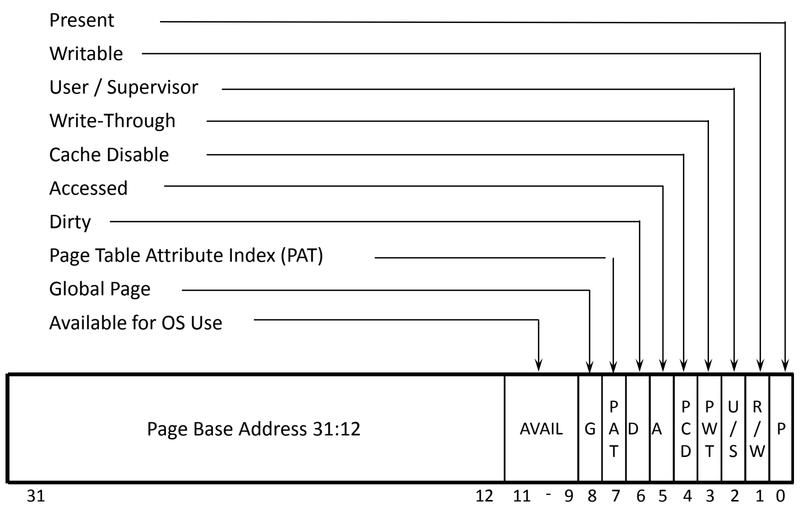
- **32bit address means 2^32 = 4GB address space**
  - There are 2^20 = 1MB pages (1MB pages x 4KB per-page = 4GB)
- **The job of the x86 virtual memory subsystem**
  - Translate 20 bits (= virtual page #) to 20 bits (= physical frame #)
  - Pages are 4KB-aligned, so it's enough to identify them with 20bits
- **Every process has a "page directory" (4KB page)**
  - Holds 1024 PDEs (page-directory entries)
    - Each PDE is comprised of 4 bytes (= 32 bits; 1024 x 4 = 4KB)
  - Each PDE contains
    - Bits: **P**resent? **A**ccessed? **D**irty? **W**ritable? **U**ser? Cache **D**isabled? …
    - 20bit physical page frame number:
      - Where to find the corresponding "page table"
- **Every PDE points to a 4KB "page table"**
  - Holds 1024 PTEs (page-table entries) with same bits + 20bit frame#
  - Frame# points to a program's memory page

2-level hierarchy

- **CR3 register points to page-directory**
  - Physical address set on context switch
  - Per process (but threads share it)
- **DIR (10 bits)**
  - Index of P**D**E (4bytes) in page-directory array (there are 1024 PDEs)
  - Each PDE holds 20bit of 4KB-aligned physical frame# of a 4KB page table
- **TABLE (10 bits)**
  - Index of P**T**E (4bytes) in page-table array (there are 1024 PTEs)
  - Each PTE holds 20 bit of 4KB-aligned physical frame# of a 4KB "regular" page
- **OFFSET (12 bits)**
  - Offset within the selected 4KB page



4KB-page PTE format



Combining user/supervisor & global page bits

- **Problem: when a system call happens…**
  - OS must run and use its own internal data structures
  - But we don't want every system call to induce a context switch
- **Solution: map OS data structures to address space of all processes**
  - Use the "user/supervisor bit" to indicate that only the OS ("ring 0") can access this memory area, whereas user code ("ring 3") cannot
  - Further, set the "global page" bit in the PTEs associated with the OS memory, which would then leave the corresponding TLB translations valid across context switches

Why hierarchical?

- **The alternative: one linear page table**
  - Requires 2^20 (PTEs) x 4 (bytes per PTE) = 4MB
  - Could be wasteful (recall, it's a per-process overhead)
  - And of course it'll be much, much worse (⇔ impossible) for 64bit addresses (discussed later)
- **The hierarchical translation allows us to avoid wasting memory**
  - Page tables (2nd-level) are allocated on-demand only
  - Far less "internal fragmentation" when memory space is sparse
    - Internal fragmentation
      = when parts of an allocation unit remain unused
    - (Is there no internal fragmentation when using hierarchical translation?)

HW/OS cooperate

- **HW defines data structures**
  - Structure of hierarchy, PTE bits, etc.
- **OS determines most of the content of page directories & page tables**
  - It explicitly sets the values of the PDEs and PTEs
- **HW does the "table walk" automatically**
  - If VA=>PA translation not found in the TLB
    - Called "TLB miss"
  - HW knows where to find the page directory (using CR3)
  - HW walks the tables, hierarchically, until it reaches the data page
  - It inserts the VA=>PA translation to the TLB
    - When page-faulting operation resumes, there will be a TLB hit
- **HW also responsible for setting bits**
  - Accessed & dirty bits (Why? Recall that OS emulate this behavior)
  - OS is responsible for turning these bits off

HW/OS cooperate

end of 2nd lecture (1st bullet)

- **HW populates TLB**
  - Whenever TLB miss occurs
- **OS invalidates TLB entries**
  - E.g., upon context switch (if HW doesn't support PID in TLB; nowadays it typically does, but the number of PIDs is limited)
- **OS is also responsible for synchronizing between TLBs**
  - Each core has its own TLB
  - TLB shootdown: when, e.g., one core invalidates PTE or changes access, OS must sync TLBs on other cores

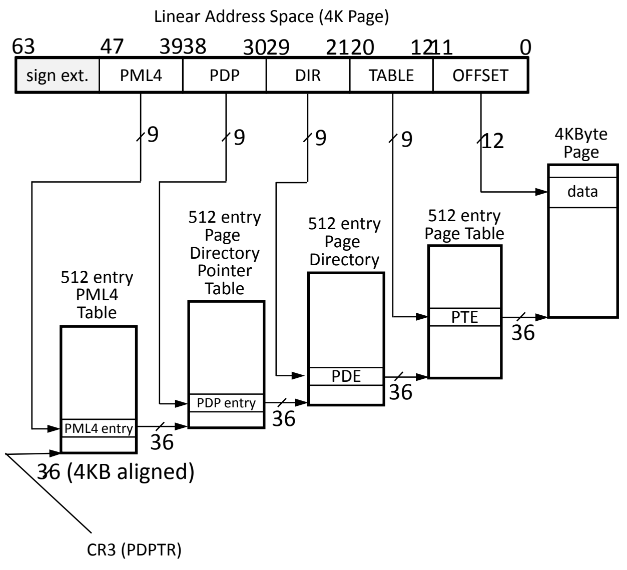
Locality helps address translation too

- **Recall CPU reads data from DRAM in cache-line resolution**
  - Cache-line size = 32 bytes in x86 32bit
  - So whenever the CPU reads one PTE
  - It actually inserts another 7 PTEs to the cache (to L2, but not to L1)
    - 32bytes / 4bytes = 8 PTEs
  - The eight PTEs reflect contiguous virtual space (special locality)
  - If/when CPU encounter a TLB miss on one of the 7, it'll find them in L2 and will not have to read it from DRAM
  - For x86 64 bit, PTE is twice as long (8 bytes), but cache line is twice as long too (64 bytes)

64bit x86 address translation

- **64bit address means 2^64 = 16 Exabyte address space**
  - 2^20≈ 10^6 = Megabyte
  - 2^30 ≈ 10^9 = Gigabyte
  - 2^40 ≈ 10^12 = Terabyte
  - 2^50 ≈ 10^15 = Petabyte
  - 2^60 ≈ 10^18 = Exabyte (= a billion GBs)
  - …DRAM can't be that big; 64 bits are thus much more than needed
- **In practice, current x86_64 HW uses "only" 48 bits => 256 TB**
  - Still more than enough
- **48bit address reflects a 4-level hierarchy, divides into 5 parts**
  - 9bits X 4 levels + 12bits offset
  - Each PTE is 8 bytes (rather than 4, to be able to hold the wider address)
- **The job of the x86 64bit virtual memory subsystem**
  - Translate 36 bits (= virtual page #) to 36 bits (= physical frame #)
  - As before, pages are 4KB-aligned

4-level hierarchy



More details:
http://webcourse.cs.technion.ac.il/234123/Spring2013/ho/WCFiles/ppc64vm-zhang2009.pdf
http://webcourse.cs.technion.ac.il/234123/Spring2013/ho/WCFiles/ppc64vm-peng1995.pdf

A different HW/SW virtual memory implementation

64-BIT POWER-PC (PPC) VIRTUAL MEMORY

HW-support for virtual memory can be….

- **Implemented completely differently**
  - Not carved in stone
  - It's the HW vendor that decides
- **Intel defined in one way for the x86 architecture**
  - Radix tree page table
- **IBM defined it differently for the POWER architecture**
  - Hashed page table
  - Which we're going to describe next…

3 address types: effective => virtual => real

- **Effective**
  - Each process uses 64-bit “effective” addresses
  - Effective addresses aren’t unique per-process
  - More or less equivalent to x86 “virtual” addresses
  - Get translated to PPC “virtual” addresses
- **Virtual**
  - A huge 80-bit address space
  - All processes live in and share this (single) space
  - Namely, if two processes have a page with the same virtual address
    - Then it’s the same page (= a shared page)
  - *Not* equivalent to x86 virtual addresses
  - Get translated into physical (“real”) address
- **Physical (a.k.a. real)**
  - 62-bit

PPC Segments

- **Effective & virtual spaces are partitioned into contiguous segments of 256MB (= 2^28 = 2^16*2^12 = 64K*4KB pages)**
  - Each segment is contiguous in the per-process effective memory space
  - Each segment is contiguous in the single, huge virtual space
  - Segments are 256MB-aligned, and can be private or shared (why?)
- **How many segments can there be in effective space?**
  - Effective space size is 2^64; so can be 2^(64-28) = 2^36 segments

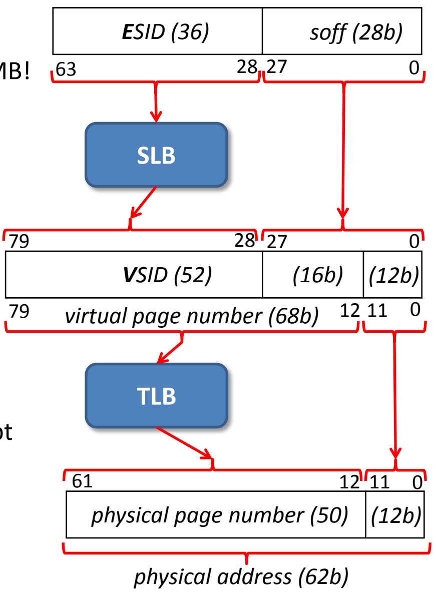
ESID = effective segment ID (36b)	*page number (16b)*	*off (12b)*
63 … 28	27 … 12	11 … 0
	segment offset (28b)	

- **How many segments can there be in virtual space?**
  - Effective space size is 2^80; so can be 2^(80-28) = 2^52 segments

VSID = virtual segment ID (52b)	*page number (16b)*	*off (12b)*
79 … 28	27 … 12	11 … 0
	segment offset (28b)	

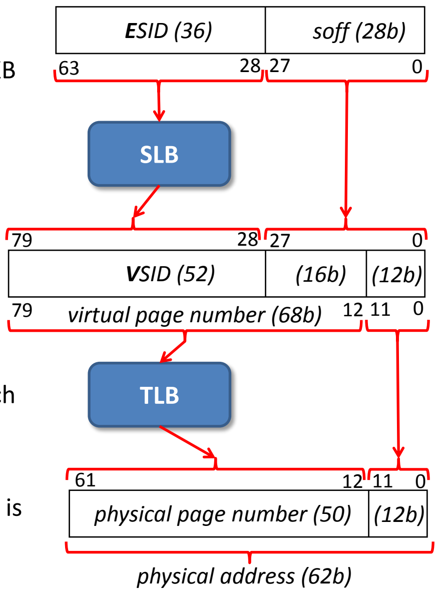
PPC SLB (segment lookaside buffer)

- **SLB is a fast, small HW cache**
  - O(32) entries, each points to 256MB!
  - O(1) cycles to access
- **HW searches SLB**
  - To find ESID=>VSID mapping
  - Each STE (segment table entry) contains ESID & VSID info
- **OS explicitly manages SLB**
  - Maintains ESID=>VSID for all segments of the process
- **Upon SLB miss**
  - HW raises "segment fault" interrupt
  - OS will then insert right STE
- **Upon context switch**
  - OS invalidates SLB (not shared)



PPC TLB (translation lookaside buffer)

- **TLB is a (less) fast HW cache**
  - O(1024) entries, each points to 4KB
  - O(10) cycles to access
- **HW searches TLB**
  - To find VPN=>PPN mapping
  - Each PTE (page table entry) contains VPN & PPN info
- **Shared by all processes**
  - Since virtual space is shared
  - Unlike in x86
  - Not invalidated upon context switch
- **Managed by HW & OS**
  - Upon miss, HW populates TLB
  - By "walking the page table", which is in fact the "HTAB"…



PPC HTAB (hash table)

- **PPC PTE (page table entry)**
  - Contains VPN (the "tag"; why do we need it?) & PPN
    - Unlike x86 PTE that only contains PPN
  - |PTE| = 16 bytes = 128 bit > |VPN (68bit)| + |PPN (50bit)| = 118bit
- **PTEG (page table entry group)**
  - Contains 8 PTEs
  - 8 * 16 = 128 bytes = |cache line|
- **HTAB**
  - At boot time, OS allocates in DRAM the "HTAB" array
    - Holds *k* PTEGs, where *k* is configurable
  - With a DRAM size of a handful of GBs:
    - It is recommended to set |HTAB| = O(10MB–100MB)
      => *k* = O(100,000 – 1,000,000)
      => HTAB points to O(800,000 – 8,000,000) 4KB pages
      => HTAB provides coverage of 3GB – 30GB of DRAM
  - OS saves HTAB's size & base in "SDR1" (storage description register)

PPC HTAB (hash table)

- **Upon TLB miss**
  - HW hashes VPN (modulo *k*)
    - Recall that HW knows about HTAB location and size via SDR1
    - The hash function is well-known and documented in the PPC spec
  - HW accesses HTAB and gets the so called "primary" PTEG of the VPN
  - HW searches for the VPN in the 8 PTEs populating the primary PTEG
  - If found, HW puts VPN=>PPN in TLB and re-executes operation
  - Otherwise, HW uses a secondary well-known hash function to obtain the "secondary" PTEG
  - If VPN found, HW puts VPN=>PPN in TLB and re-executes operation
  - Otherwise, HW triggers page-fault interrupt
  - OS will then resolve the page fault and put the appropriate PTE in one of the associated two PTEGs (primary or secondary)
  - After interrupt is handled, HW will re-execute the operation; now it'll find the VPN in one of the 2 PTEGs

PPC HTAB (hash table)

- **HW searches for VPN=>PPN translation as follows:**
  - if(VPN found in TLB)
    - Get PPN from TLB
  - else if(VPN found in primary_PTEG = HTAB[hash1(VPN) % k])
    - Get PPN from primary_PTEG
  - else if(VPN found in secondary_PTEG = HTAB[hash2(VPN) % k])
    - Get PPN from secondary_PTEG
  - else
    - Generate page-fault interrupt
    - OS will place appropriate PTE in either primary or secondary PTEGs
    - Next time, faulting operation will succeed reloading

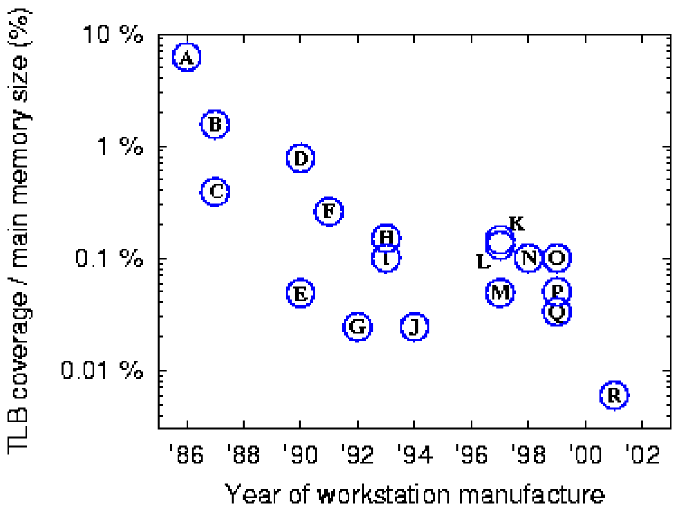
PPC ERAT (effective to real translation)

- **A small, fast HW cache**
  - O(128) entries
  - Quicker than 1-cycle to access (on the critical path)
  - Translates from effective to real (physical)
  - Analogous to x86 TLB (L1)
  - Updated to hold the most recent effective=>physical mappings used
    - On a LRU basis
  - If hit, don’t need to go through the SLB/TLB process
    - In most cases, obviates the need to do costly SLB=>TLB=>HTAB translations

Actually, PPC uses TLB hierarchy

- **ERAT**
  - TLB L1
- **TLB**
  - TLB L2
- **HTAB**
  - TLB L3
- **Only if not found in all 3 levels**
  - Go to OS, which has all information
- **Likewise, Intel & AMD**
  - Introduced a (bigger, slower) TLB 2
- **The reason…**
  - DRAM is getting bigger
  - TLB L1 remains roughly the same size

TLB coverage drops exponentially



http://static.usenix.org/event/osdi02/tech/navarro.html

How to increase TLB converge?

- **Superpages (also called huge pages): page size > 4KB**
  - Different sizes supported
    - Specifics vary among different architectures
    - From a few MBs to a few GBs!
      (E.g., Intel supports 2MB, 4MB, 1GB)
- **TLB hierarchy**
  - Like the caches L1, L2, …
  - Architectures can also support TLB-L1 (size = a few dozens), TLB-L2 (size = a few hundreds)
  - TLB-L2 is bigger, but is slower