CS24: Introduction to Computing Systems

Spring 2015

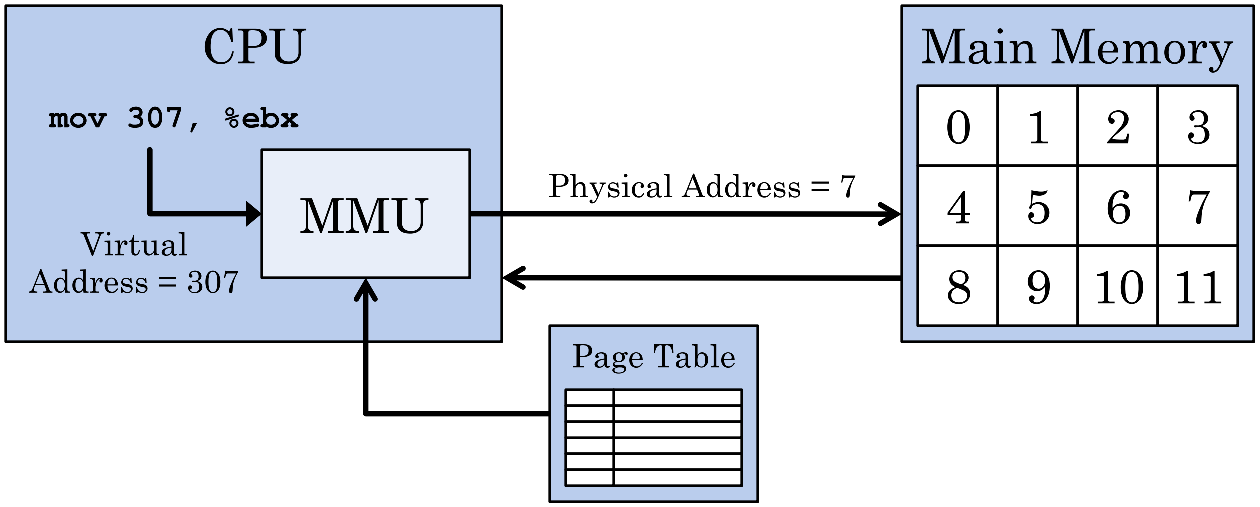
Lecture 23

Last Time: Virtual Memory

- Began to focus on how to virtualize *memory*
- Instead of directly addressing physical memory, introduce a level of indirection
  - Programs use *virtual* addresses, not physical addresses
- Divide memory up into pages of size P, $P = 2^p$
  - Choose a page size much larger than 1 byte or 1 word
  - (makes mapping table smaller; other reasons too)
- Map each virtual page to a physical page frame
  - Mapping is specified using a page table
  - Each page table entry maps one virtual page to one physical page

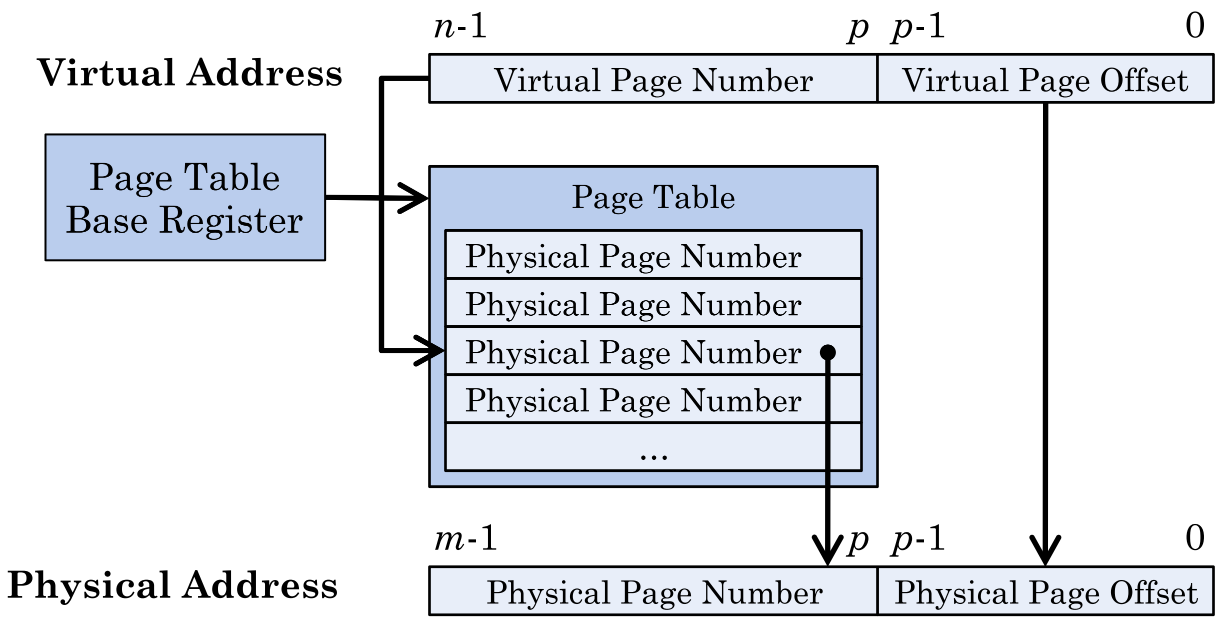
Last Time: Virtual Addressing (2)

- Performing this address translation in software would be horribly slow…
- CPU provides *hardware* support for virtual memory and address translation
  - CPU has a Memory Management Unit (MMU) that performs this address translation on the fly
  - The MMU uses a page table to perform translation



Address Translation

- Page table is indexed with virtual page number
  - Page table entry contains the physical page number
  - Combine physical page number with virtual page offset to get physical address
- Start of page table specified in a control register
  - MMU uses this address to look up page table entries



Virtual Memory: Issues

- This approach already simplifies many things
  - e.g. process isolation, context switches, shared memory
- Our virtual memory is still limited:
  - Must share limited main memory amongst <u>all</u> processes, whether running or stopped
- We know not all processes are always active…
  - e.g. a process can be stopped and resumed
  - e.g. a process may only be using a portion of its instructions and data in memory
- Redefine our notion of "virtual memory" to be an array of *N* contiguous bytes <u>stored on disk</u>
- Main memory becomes a cache for the local disk
  - Main memory is Level 4 (L4) cache, disk is L5 memory

Virtual Memory as a Cache

- Apply same concepts as hardware caches: update page table entries to track more details
  - Is a page of memory actually allocated for a specific virtual page number?
  - Is the virtual page cached in DRAM main memory?
  - (similar to extra details stored in cache lines)

- Now, the memory available to all processes can be *much larger* than actual size of main memory
  - Use our caching techniques to get a virtual memory much larger than main memory, but at [close to] DRAM speeds

Virtual Memory as a Cache (2)

- DRAM is cache for virtual memory stored on disk
  - DRAM access latency is 50-100ns
  - Disk access latency is 8-30ms (!!!)
  - A virtual memory cache-miss is <u>very</u> expensive
- However, disk throughput is 50+ MB/second
  - Reading the first byte of data from disk is very slow…
  - Reading second and subsequent bytes is *much* faster
- Example: transfer unit of 1 byte
  - Cache-miss penalty is 8-30ms
- How about a transfer unit of *100KB?!*
  - Cache-miss penalty is 8-30ms + 2ms
  - Not that much more expensive to retrieve!

Virtual Memory: Strategies

- Choose a large page size (e.g. 4KB or 8KB), to compensate for large disk access latency
  - Retrieving additional bytes doesn't add much time to the cost of a cache miss
  - Using a much larger transfer-unit size will reduce the number of cache misses we have in the first place
- Use sophisticated replacement policies in DRAM cache to avoid cache misses as much as possible!
  - Example: DRAM cache is fully associative
- Use write-back policy instead of write-through, when a program writes to a cached DRAM page
  - Don't write a modified page to disk until it is actually evicted from main memory
  - *(One reason why OS needs to be shut down cleanly…)*

Page Table Entries

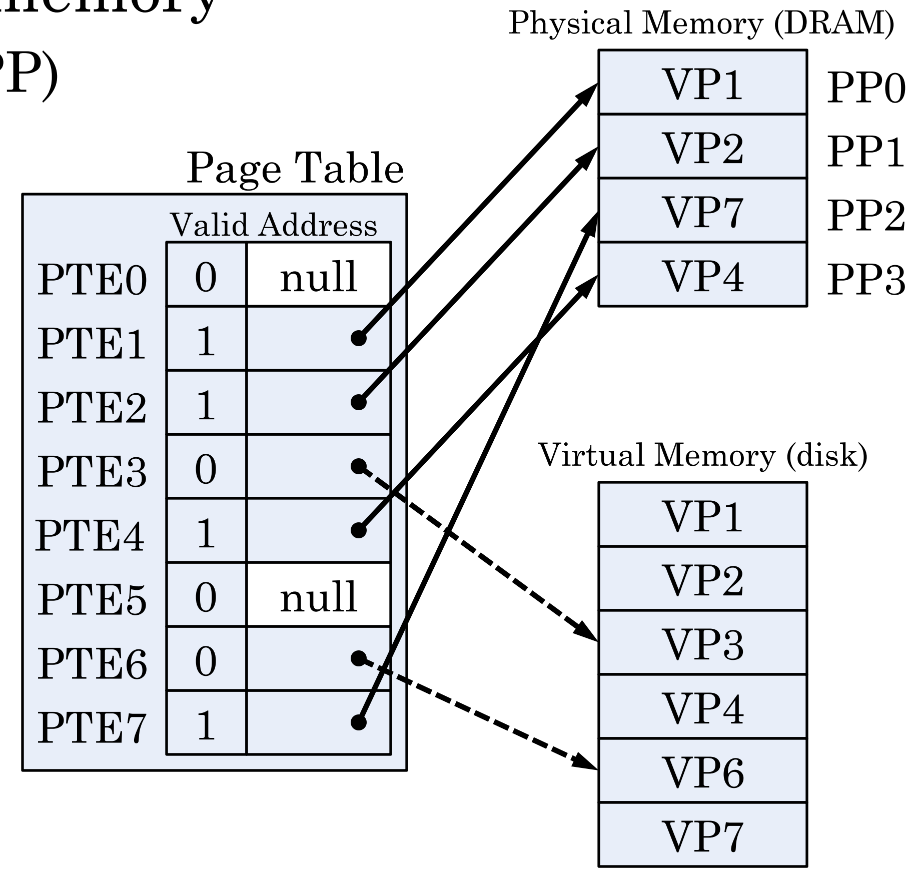
- Page table entries must be enhanced to support this virtual memory model
- Include a *valid* bit, along with each address
- *valid* = 1 means page is cached in DRAM memory
  - Corresponding address is the start of the page in DRAM physical memory
- *valid* = 0 means the page is not cached in DRAM
  - If stored address is 0, the virtual page is not allocated
  - If address is not 0, the virtual page is allocated, and the address is the page's location on disk
- IA32 page-table entries call this the *present* bit, i.e. "Is the page present in main memory?"

Processor and Operating System

- *A very important detail about virtual memory facility:*
- The processor provides *some* hardware support for virtual memory
  - Address translation is performed in hardware, using the Memory Management Unit
  - MMU must use a page table to perform address translation
- The processor cannot <u>completely</u> support virtual memory in hardware, on its own!
  - What to do when a virtual page is not in main memory? CPU has no idea how virtual memory pages are cached.
- The operating system must also provide support for many virtual memory operations
  - CPU does as much as it can on its own…
  - CPU invokes an exception when it needs the kernel's help!

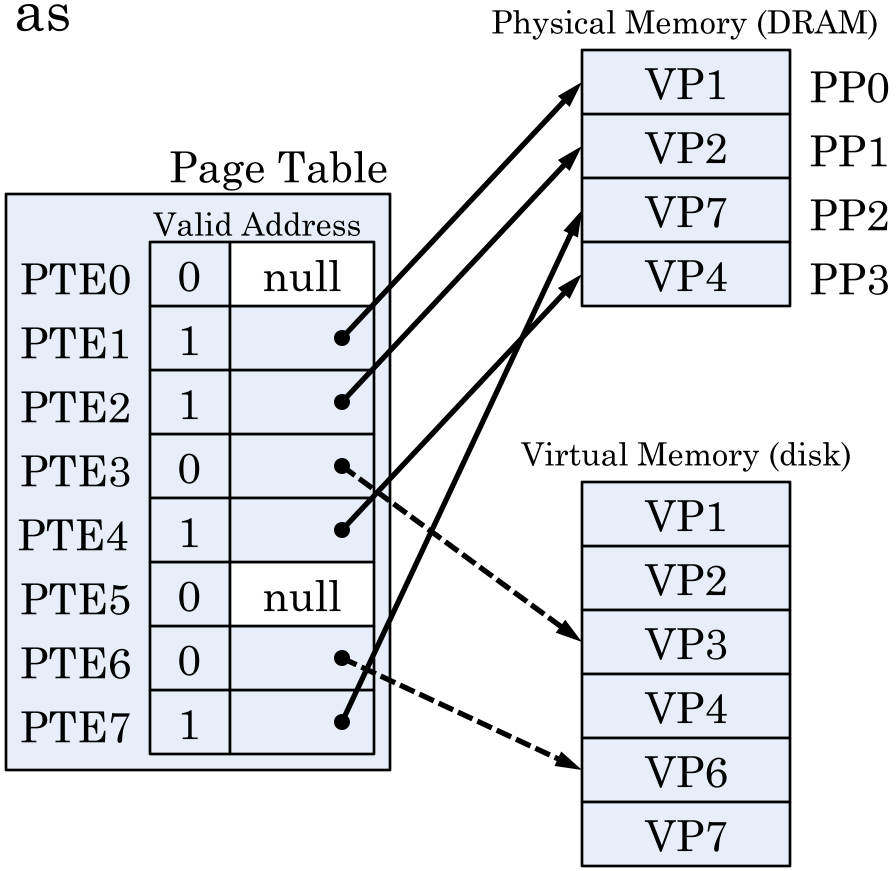
Virtual Memory Example

- Example: small virtual memory
  - 4 physical page frames (PP)
  - 8 virtual pages (VP)
- Some virtual pages are cached in DRAM:
  - VP1, VP2, VP4, VP7
- Some virtual pages are only stored on disk
  - VP3, VP6
- Two virtual pages have not been allocated
  - VP0, VP5



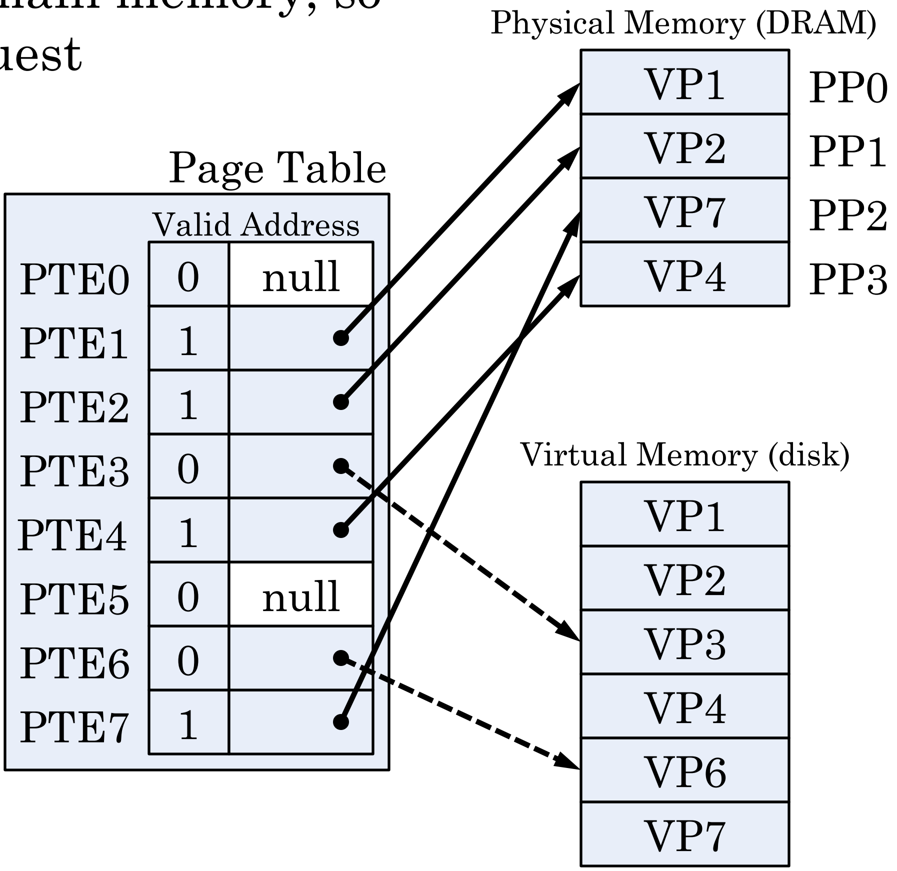
Virtual Memory Accesses

- Program accesses a word in Virtual Page 2
  - MMU looks in page table for VP2 (PTE2)
  - Uses virtual page number as index into Page Table
- Virtual page 2 is stored in physical page 1…
  - Valid flag is 1: page is cached in DRAM
  - MMU sends physical address to DRAM main memory, and retrieves the data value



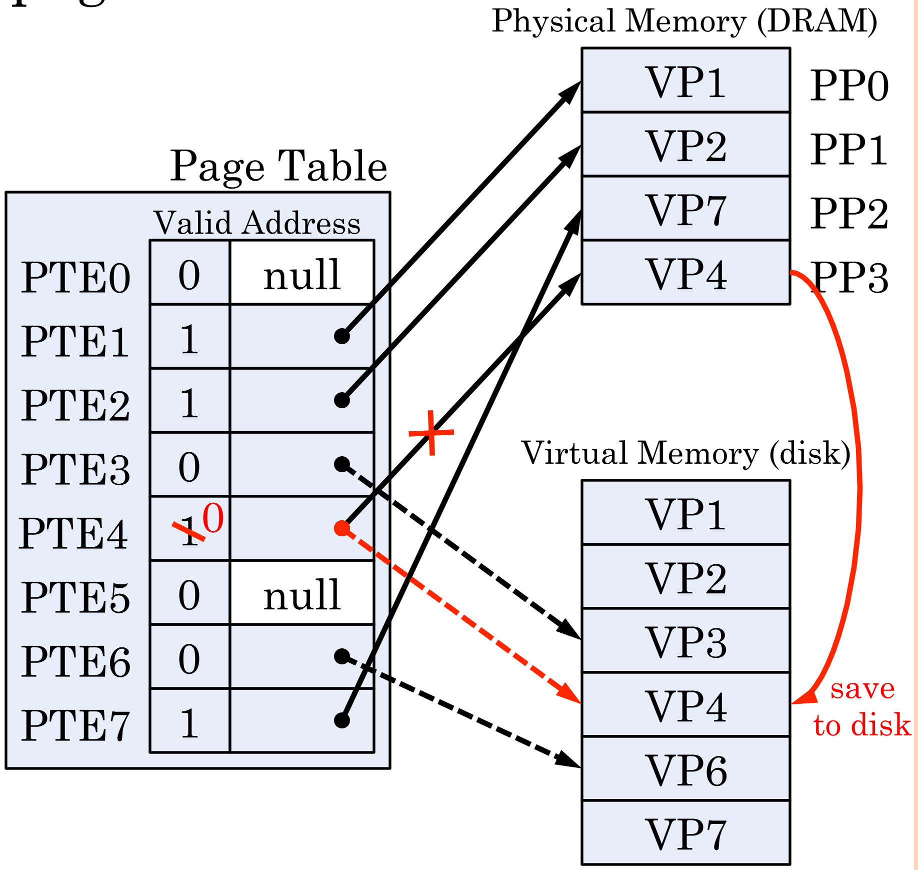
Virtual Memory Accesses (2)

- Now, program accesses a word in Virtual Page 6
  - MMU looks in page table for VP6, but the valid flag is 0!
  - Virtual page isn't cached in main memory, so MMU cannot satisfy the request
- CPU causes a <u>page fault</u> exception
  - It's up to the kernel to resolve the problem!
  - Kernel provides a handler for resolving page faults
- The <u>kernel</u> manages the page table, memory pages
  - Processor simply provides support for virtual memory



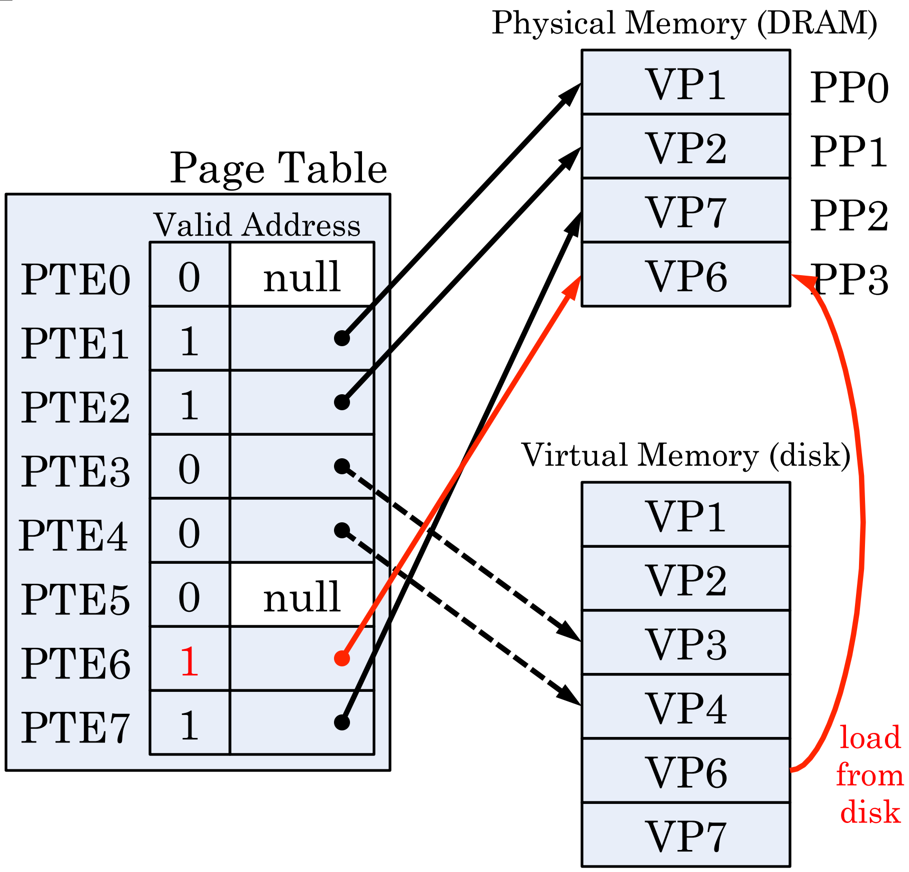
VIRTUAL MEMORY ACCESSES (3)

- CPU generates a page fault, and kernel's handler is invoked
- Page-table entry for virtual page 6 has a non-null address…
  - The kernel can load VP6 from disk into memory
- …but first, some other page must be evicted!
- Kernel selects VP4 as the victim page
  - If VP4 is changed, must also write it back to disk
- Kernel updates PTE4 to show page isn't cached



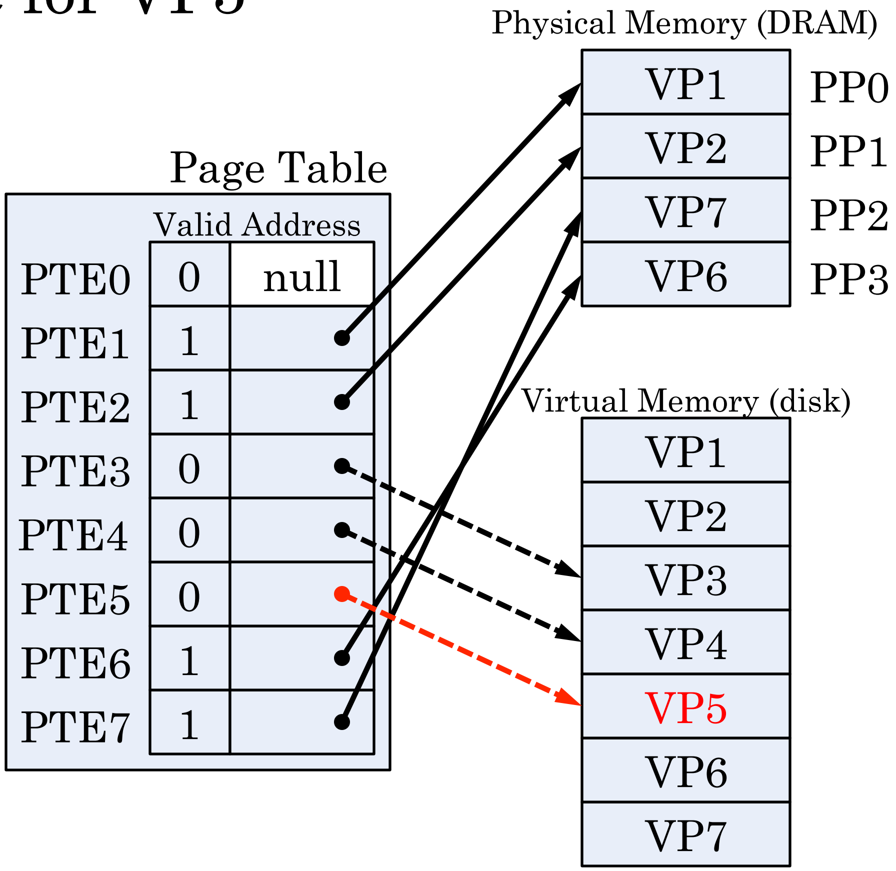
Virtual Memory Accesses (4)

- Now kernel loads virtual page 6 into physical page 3
  - Update PTE6 to point to physical page 3 in DRAM memory
- Finally, kernel returns from the page-fault handler
- Since this is a fault:
  - CPU resumes executing the instruction that caused the fault
- Program repeats the access to virtual page 6
  - This time, MMU finds that PTE6 is valid
  - MMU performs address translation, and retrieves value from physical page 3



Allocating Virtual Memory

- If a process needs more memory, kernel can allocate another virtual memory page
- Example: allocate a page for VP5
  - Make room on the disk for virtual page 5
  - Update PTE5 to point to VP5 on disk
  - *(**Note:** no <u>requirement</u> that pages on disk be kept in order…)*
- When program begins using VP5, it can be swapped into physical memory



Virtual Memory as a Cache

- DRAM main memory is a cache to virtual memory stored on disk
- Caching terminology is slightly different
  - Virtual memory was invented in 1960s, before SRAM caches were needed between the CPU and DRAM!
  - Caches have *blocks*, but virtual memory has *pages* (virtual blocks) and *frames* (physical blocks)
  - Caches have *cache sets* and *cache lines*, but virtual memory has *page tables* and *page-table entries*
  - Caches have *cache-hits* and *cache-misses*, but virtual memory has *page-hits* and *page-faults*
- Nearly identical concepts, but terminology is slightly different

Virtual Memory as a Cache (2)

- Like caches, virtual memory depends on *locality*
  - When programs exhibit good locality, we can get a virtual memory much larger than physical memory, but with same performance as physical memory
- Processes typically reference a specific set of memory pages as they run
  - Called the *working set* of the process, or the *resident set*, since these pages are kept in memory
- If a program has good locality:
  - The working set of pages will usually fit entirely within memory, yielding good performance

Virtual Memory as a Cache (3)

- If a program's working set doesn't fit within memory, *thrashing* will occur
  - Pages are constantly swapped in and out of main memory
  - The program runs *very* slowly, since many accesses incur the 8-30ms disk latency penalty
- If a program runs very slowly, and you see the disk is constantly being accessed, it is probably thrashing ☹
- UNIX provides a **`getrusage()`** function
  - "get resource usage"
  - Reports several details about virtual memory system, such as the number of page faults
  - Also reports number of voluntary and involuntary context-switches, number of signals received, etc.
  - Can monitor how well-behaved your program is! ☺

Virtual Memory and Disk Files

- Once virtual memory is disk-based, we can provide some new features
- Loading programs and libraries into memory:
  - Programs and libraries are stored as binary files containing instructions and data
- When we want to start a new program:
  - Kernel allocates a contiguous set of virtual pages large enough to hold the program's code and data
  - Then, the kernel updates the process' page table, pointing these new pages to the binary file stored on disk
  - The new page-table entries are marked as *valid* = 0
    - i.e. the pages are not yet cached in DRAM memory
- As the program runs, virtual memory system loads program's code and data into memory automatically

Virtual Memory and Disk Files (2)

- This technique is called memory mapping
- Can map a disk file into memory with the UNIX **`mmap()`** function

```
void * mmap(void *start, size_t length, int prot,
                  int flags, int fd, off_t offset)
```

  - Maps a specific portion of the file **`fd`** into memory
  - Portion starts at **`offset`** and contains **`length`** bytes
- Function works by allocating virtual memory pages, and pointing page table entries to disk file
- Instead of manually performing file I/O, simply perform reads and writes against a memory area
  - Virtual memory system automatically loads and saves data to the file stored on disk

Pages and Permissions

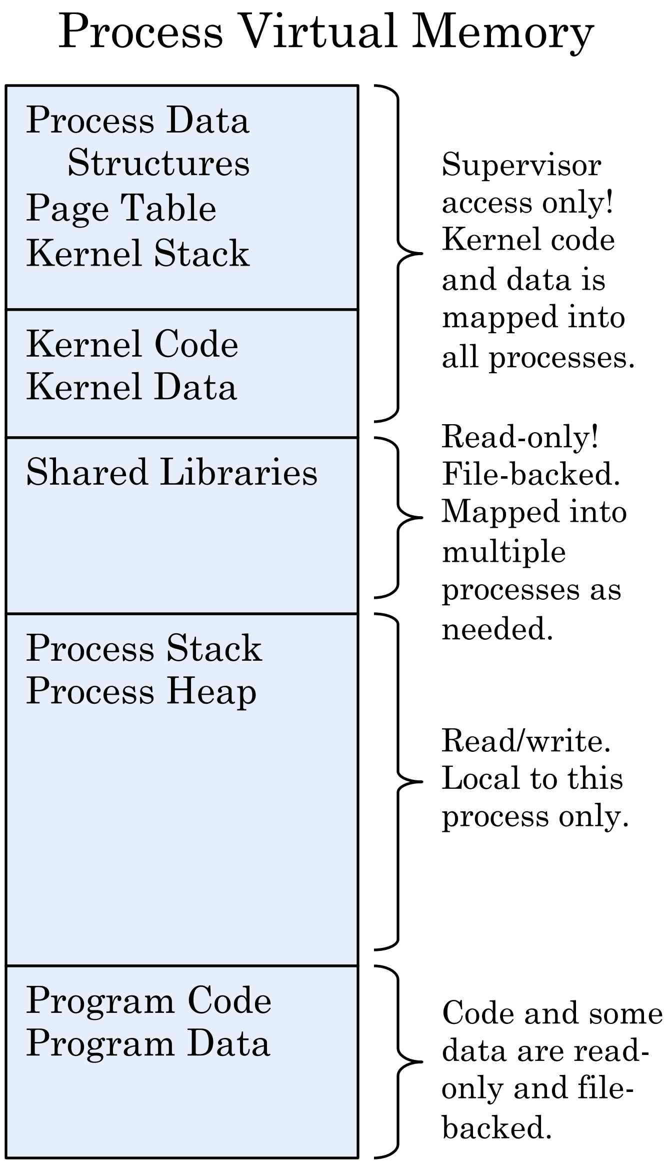
- We can use the virtual memory system to load programs and read-only data into memory…
  - Really don't want to let programs modify their code or read-only data. Or shared libraries, etc.
- Add permission bits to each page table entry, that control read/write access, etc.
- IA32 has two permission bits:
  - Read/Write flag controls whether code can write to the page, or only read from it
  - User/Supervisor flag controls whether code must have a higher privilege level to access the page

Page Table

valid	R/W	SUP	Physical Page Number
valid	R/W	SUP	Physical Page Number
valid	R/W	SUP	Physical Page Number
valid	R/W	SUP	Physical Page Number
valid	R/W	SUP	…

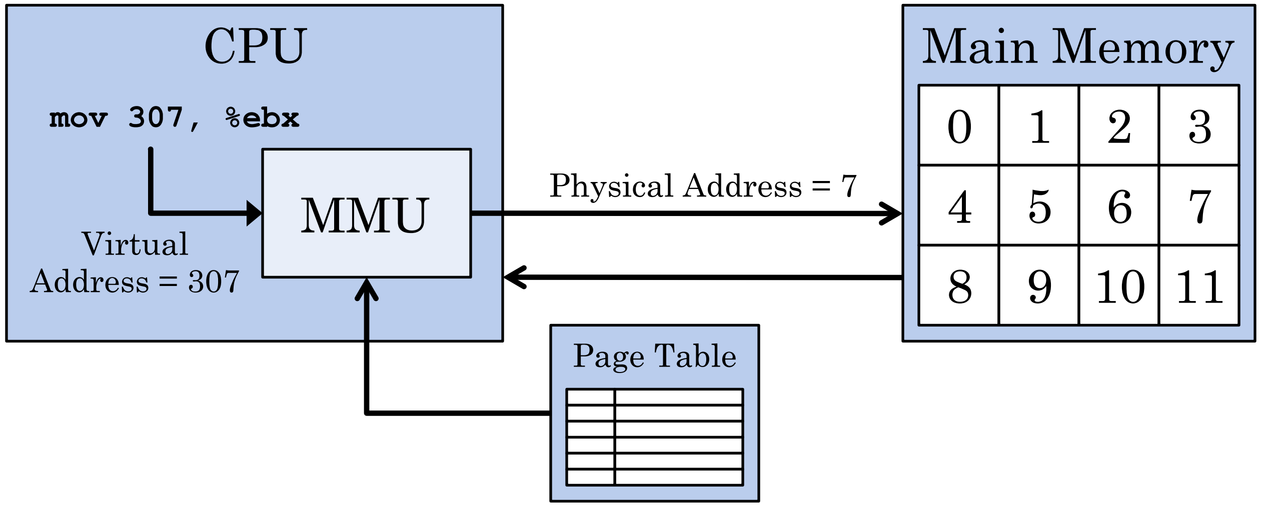
Pages and Permissions (2)

- Kernel uses these virtual memory features to load and run programs
  - Load program and shared library code from disk automatically
  - Share kernel and library code across multiple processes
  - Isolate address space of processes
- If code tries to access a page in a way disallowed by permissions, CPU raises a *general protection fault*
- The kernel has a handler for general protection faults
  - Typical response: terminate the process! ☺


Process Virtual Memory
Process Data Structures
Page Table
Kernel Stack
Kernel Code
Kernel Data
Shared Libraries
Process Stack
Process Heap
Program Code
Program Data
Supervisor access only! Kernel code and data is mapped into all processes.
Read-only! File-backed. Mapped into multiple processes as needed.
Read/write. Local to this process only.
Code and some data are read-only and file-backed.

Implementation Issues

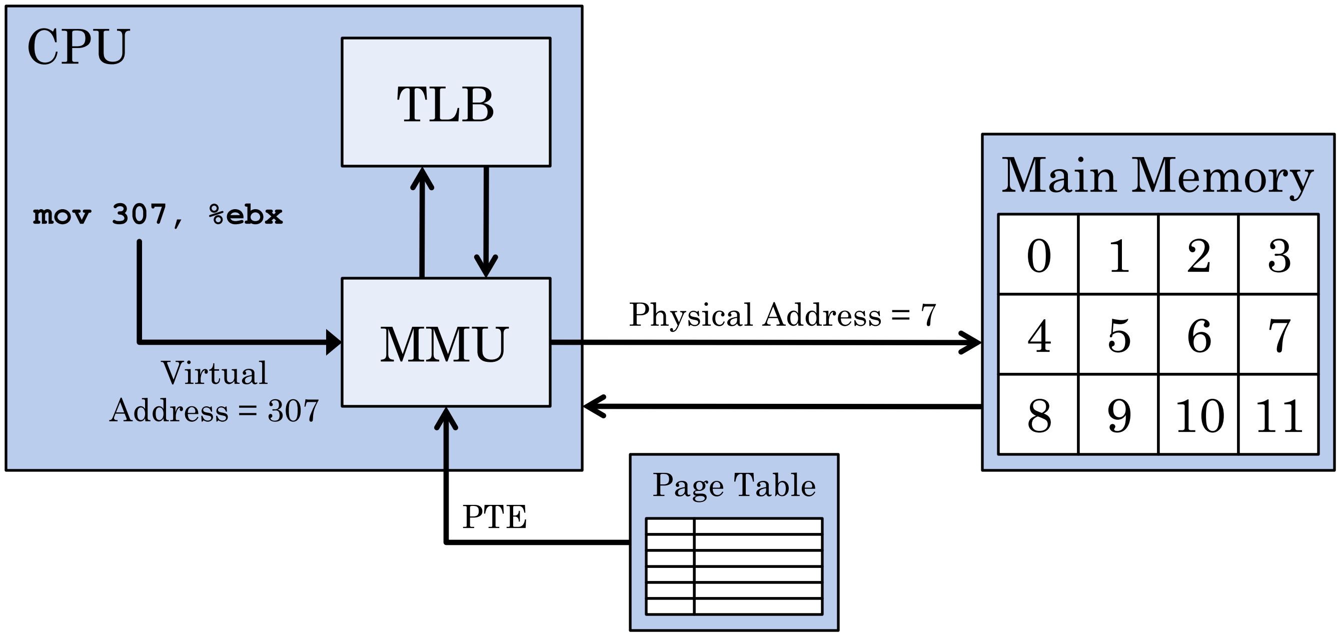
- Glossed over a couple of implementation issues
- Issue 1: Where do page tables live?
  - Showed page tables as separate from main memory



- In reality, page tables also live in main memory
- Why would this be a problem?
  - *Hint: How long does it take to access main memory?*

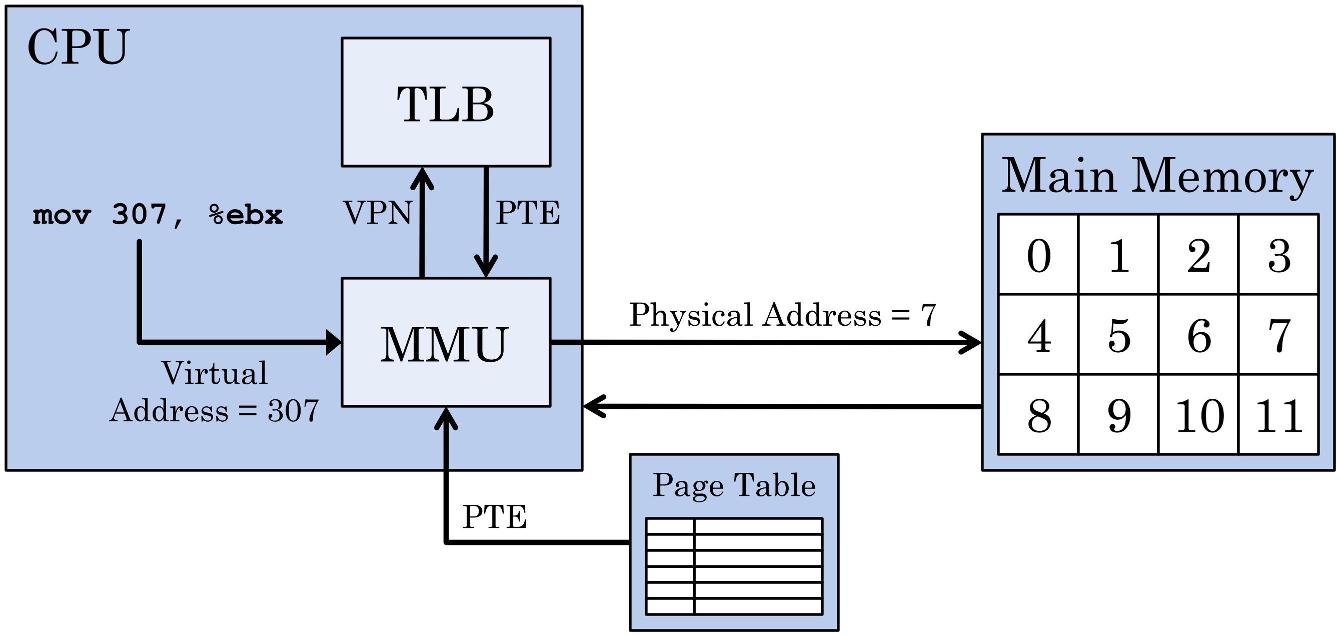
Implementation Issues (2)

- If page tables live in main memory, address translation would frequently incur DRAM access overhead of 50-100ns!
  - If you're lucky, part of page table is in L1 or L2 cache
- CPUs frequently use a Translation Lookaside Buffer (TLB) to cache page table entries
  - Don't even want to incur L1 or L2 penalty, if possible!



TRANSLATION LOOKASIDE BUFFER

- Translation lookaside buffer stores one page table entry per cache line
  - Input is virtual page number, output is page table entry
- TLB often has a high degree of associativity
  - Maximize chance that TLB contains needed page table entry!

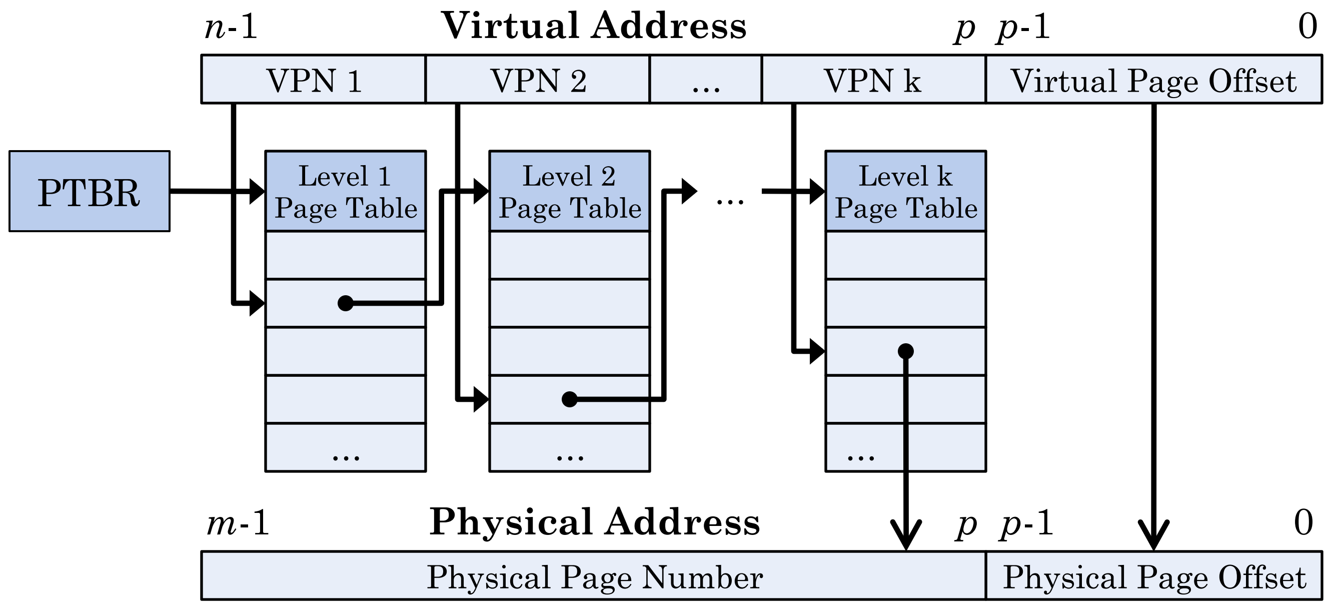


Implementation Issues (3)

- Issue 2: How large are page tables?
  - Showed the processor using a single page table
  - Real-world system doesn't work that way!
- Given a 32-bit address space, and pages of 4KB:
  - Page offsets take 12 bits, page numbers are 20 bits
- To map each virtual page to a physical page, how many page table entries?
  - $2^{20} = 1{,}048{,}576$
- If each page table entry is 32 bits, how big is the entire page table?
  - Page table for this address space would be 4MB!
- Every process would require a 4MB page table
  - *Even bigger trouble for larger address spaces!*

Multi-Level Page Tables

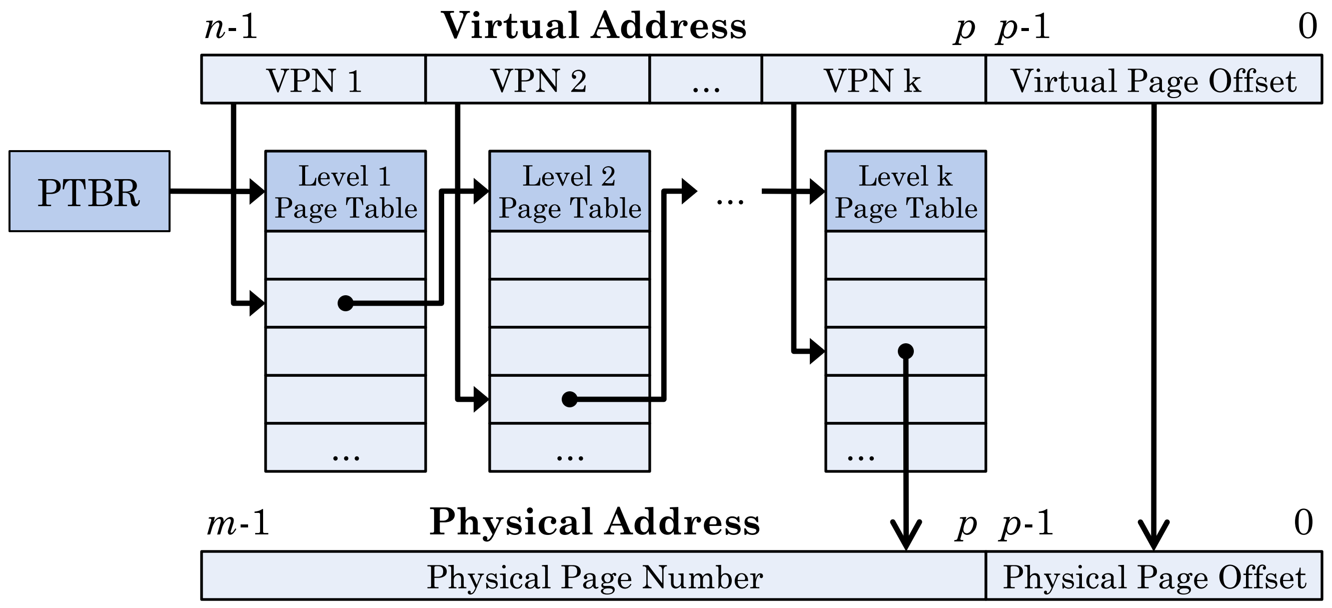
- Instead of a single page table, introduce a *hierarchy* of page tables



- Idea: typically, most of a process' virtual address-space will be unallocated
  - Hierarchy of page tables will be relatively sparse

Multi-Level Page Tables (2)

- Instead of a single page table, introduce a *hierarchy* of page tables



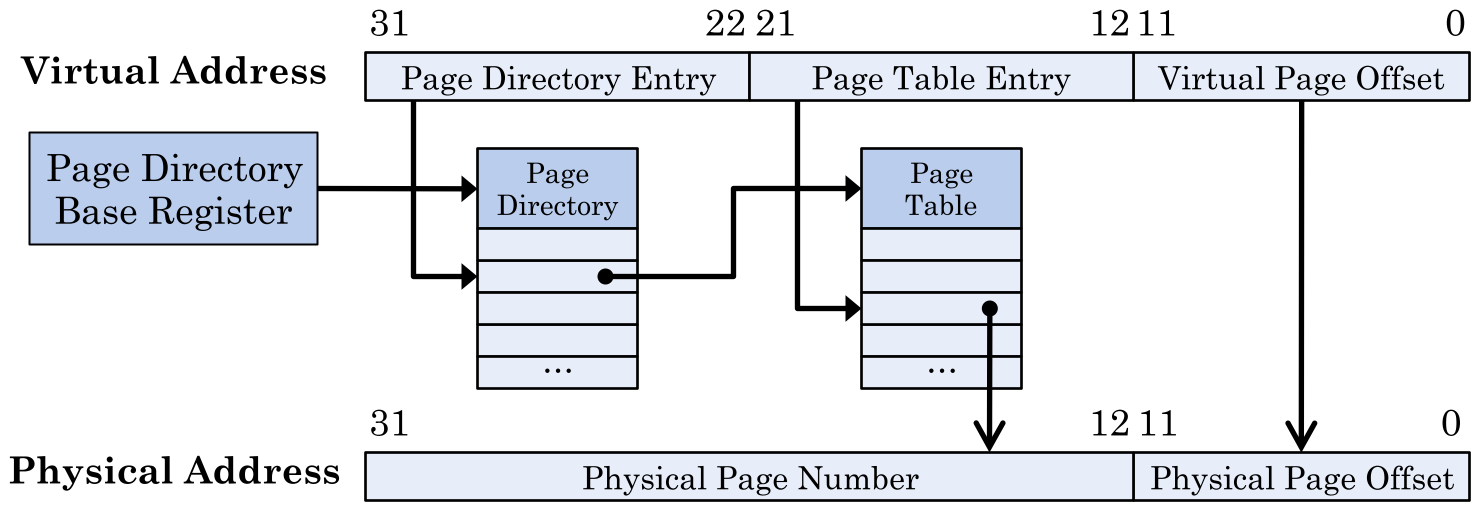
- If an entire page table at level i is empty:
  - Don't allocate a page for the empty page-table
  - Corresponding table entry at level i – 1 is null

IA32 Virtual Memory Support

- Intel Pentium-family processors provide hardware support for virtual memory
- Virtual and physical address spaces are 32 bits
- Pages are 4KB in size ($2^{12} = 4096$)
  - Pages are identified by topmost 20 bits in address
- Pentium-family processors implement a two-level page table hierarchy
- Level 1 is the *page directory*
  - Entries in the page directory refer to page tables, as long as the page table is not empty
- Level 2 contains *page tables*
  - Entries map virtual pages to physical pages

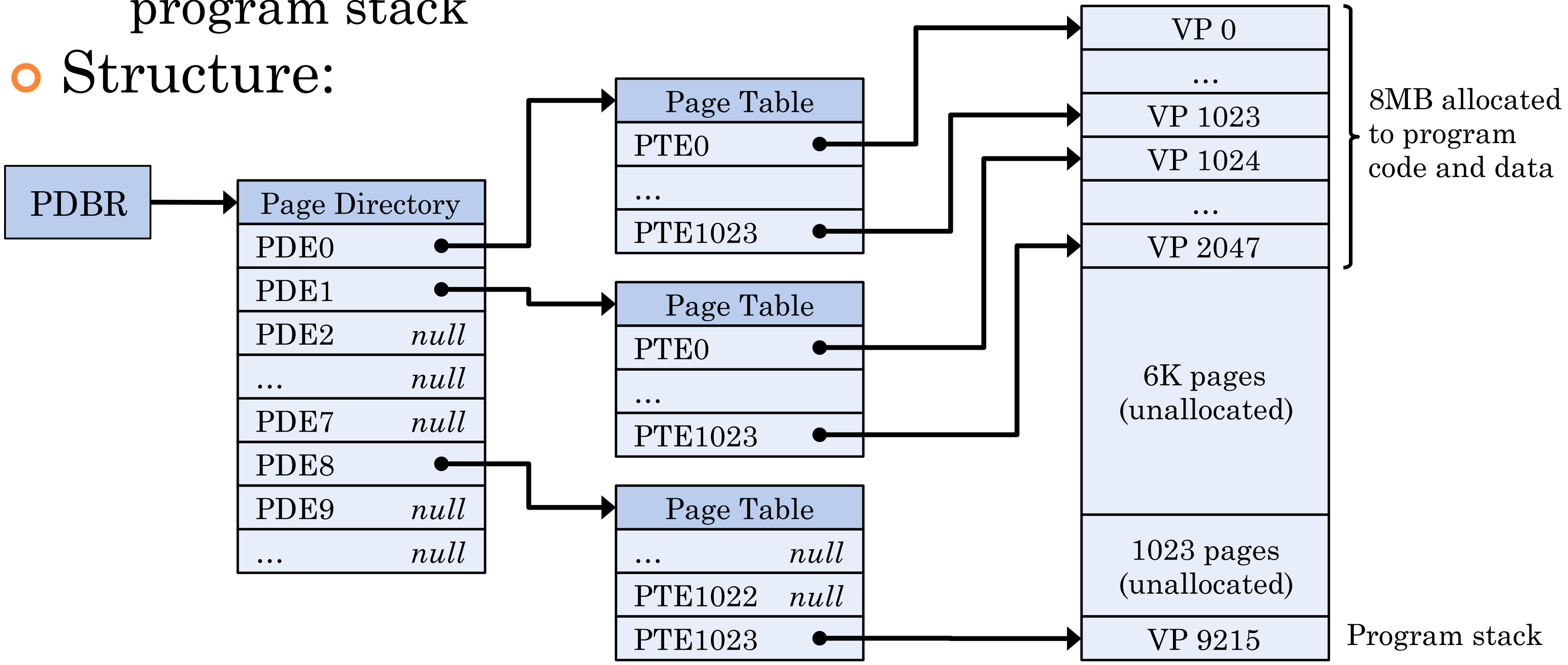
IA32 Virtual Memory Support (2)

- IA32 page-table hierarchy:
  - Top 10 bits of virtual address is the page directory entry
  - Next 10 bits is used for page table entry
  - Bottom 12 bits used for the virtual page offset



Multi-Level Page Table Example

- Program with this memory layout:
  - Bottom 2K pages allocated to program code and data (4KB page size × 2K = 8MB program and data size)
  - 6K pages left unallocated, for program heap to use
  - 1023 unallocated pages, plus 1 page allocated to program stack
- Structure:



Aside: Core i7 and Page Tables

- Intel Core i7 implements a <u>four</u>-level hierarchy
  - 48 bit address space – again, page tables would be too large if not broken into a hierarchy
  - Preserves the 4KB page size of IA32
- Core i7 page-table hierarchy level names:
  - Level 1 = page global directory
  - Level 2 = page upper directory
  - Level 3 = page middle directory
  - Level 4 = page table
- Each level uses 9 bits of virtual address
  - Core i7 page table entries are 64 bits
  - Page tables still work out to be 4KB in size
- Virtual page offset is bottommost 12 bits
- See CS:APP2e §9.7.1 for more details on Core i7

Next Time

- Continue discussion of IA32 virtual memory
  - More hardware implementation details
  - How Linux uses IA32 virtual memory facilities