PROCESS VIRTUAL MEMORY

CS124 – Operating Systems

Winter 2013-2014, Lecture 18

Programs and Memory

- Programs perform many interactions with memory…
  - Accessing variables stored at specific memory locations
  - Calls to functions that reside at specific memory locations
- Source code usually doesn't include absolute addresses
  - Rather, programs use symbols to refer to variables, functions, etc.
- At some point between compiling and running a program, addresses must be assigned to functions and variables
- Ultimately, the OS must specify how executable programs must be laid out in memory
  - The OS is responsible for loading a binary program and running it
  - The OS is responsible for resolving references to shared libraries
- Specification is called **application binary interface** (ABI)

Programs and Memory (2)

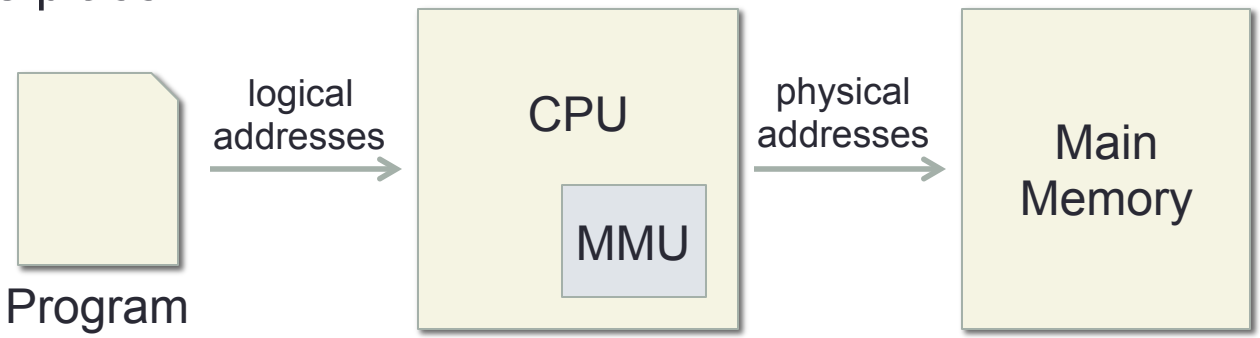
- If the locations of functions and data can be set at compile time, the compiler can generate **absolute code**
  - Code that contains absolute addresses of functions and data
- If a program's location can vary from invocation to invocation, compiler must generate **relocatable code**
  - The final binding of symbols to their addresses occurs at load time
  - The loader updates the image of the binary program in memory, based on where the program will actually be loaded
  - To support this, programs must include details of what symbols are used in the program, and where they are referenced from
- Programs can also be compiled to produce **position independent code**
  - All accesses are relative to the start of the binary in memory
  - The program determines its starting address at runtime

Programs and Memory (3)

- UNIX Executable and Linkable Format (ELF) supports both absolute and relocatable binary programs
- On Linux:
  - “Executable object files” are positioned at an absolute starting address of 0x08048000
  - “Relocatable object files” (.o files produced during compilation) include extra details specifying the locations of function and memory accesses within the binary file
  - Shared libraries are compiled with position-independent code
- Windows Portable Executable (PE) format also supports both absolute and relocatable programs
  - Windows programs are compiled relative to a preferred base address (absolute position)
  - If a program can’t be loaded at its preferred base address for some reason, it can be relocated at runtime (called “rebasing”)

Programs and Memory (4)

- The addresses used by programs are **virtual addresses** (aka “logical addresses”)
  - Range of addresses a program uses is called virtual address space
- The computer memory receives **physical addresses**
- The processor translates virtual addresses to physical addresses via the Memory Management Unit (MMU)
  - The nature of the processor’s MMU governs how this translation takes place



Why Virtual Memory?

- Virtual memory has several benefits
- Frequently talk about process isolation:
  - A process should be protected from access or manipulation by other processes on the system, unless the process specifically allows collaboration with other processes
- By running processes in separate address spaces, it is not possible for processes to access each other's data
  - (…unless the processes arrange to do so, of course)
  - Each process' virtual address space is mapped to a separate region of the computer's physical address space
  - Only the kernel can directly manipulate this mapping; processes are not allowed to do so

Why Virtual Memory? (2)

- Virtual memory also greatly simplifies the design of the operating system's application binary interface (ABI):
  - If every process has its own virtual address space, different processes can use the same virtual addresses without a problem
- All binary programs can be laid out using the same basic structure and pattern
- e.g. Linux:
  - Program text always starts at virtual address 0x08048000
  - Process stack always grows downward from 0xc0000000

- Without virtual addressing, this would be impossible

Why Virtual Memory? (3)

- A third benefit for virtual memory is that it allows the OS to move processes from memory to disk, and vice versa
  - Process is called **swapping**
- A program must be in main memory to run, but processes aren't always ready to run…
  - When a process is blocked or suspended, OS can swap it from memory to a **backing store** (e.g. a hard disk)
  - Most or all of the process' context is saved to disk
- When the scheduler switches to a new process, the dispatcher checks to see if the process is in memory
  - If not, the dispatcher can swap in the process and begin running it
  - If there isn't enough room, the dispatcher can swap out other processes from memory to the backing store

Swapping

- With swapping, the total memory used by all processes can exceed the total physical memory in the system
  - Allows more programs to be run on the system at once, especially if many of these programs are usually blocked on I/O
  - (e.g. user applications waiting for user input)
- Unfortunately, swapping tends to take a lot of time
  - Even though the backing store is usually a fast disk, still much slower than main memory
- **Standard swapping** involves moving *entire processes* into or out of physical memory
  - For a large process, can easily take several seconds or more!
- Operating systems don't generally use standard swapping
  - Instead, focus on swapping *portions* of processes out of memory
  - Much faster than swapping entire processes out of memory

Swapping (2)

- Mobile processors generally have virtual memory support
- Mobile OSes usually don't implement swapping
  - Don't have a large backing store to use for swapping
  - Usually have a small flash memory with a limited number of writes
- Generally, when OS needs more memory for a process, it asks (or forces) other processes to relinquish memory
  - e.g. if a process is taking up too much memory, the OS kills it
  - iOS tends to be aggressive in reclaiming memory from processes
- Android will write application state to flash memory before killing a process, so that it can be restarted quickly
- In general, mobile application developers must be more careful about efficient memory (and other resource) usage

MMU: Relocation

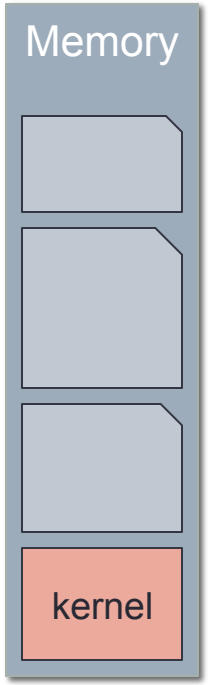
- A simple strategy for the memory management unit: relocate all virtual addresses by a constant amount
  - A **relocation register** holds a constant, which is added to logical addresses to generate physical addresses
  - *phys_addr* = *virt_addr* + *relocation*
- Additionally, can use a **limit register** to enforce the upper bound on the process' virtual address space

```
if virt_addr ≥ limit:
    raise fault
else:
    phys_addr = virt_addr + relocation
```

- Interaction with these registers is protected: only the kernel is allowed to read and write these registers

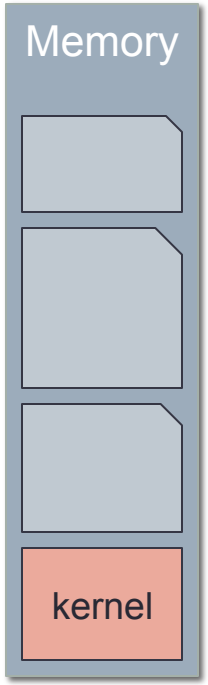
MMU: Relocation (2)

- Using this strategy:
  - Programs can be compiled with absolute addresses, e.g. starting at address 0 (or some other common starting point)
  - When a program is executed, the OS kernel can choose values for the corresponding process' relocation and limit registers based on the program's memory requirements
- Processes are isolated from each other, and from the kernel
- Using this kind of address translation mechanism gives us a **contiguous memory allocation** scheme
  - Each process occupies a single contiguous region of physical memory
  - Processes occupy adjacent regions of memory



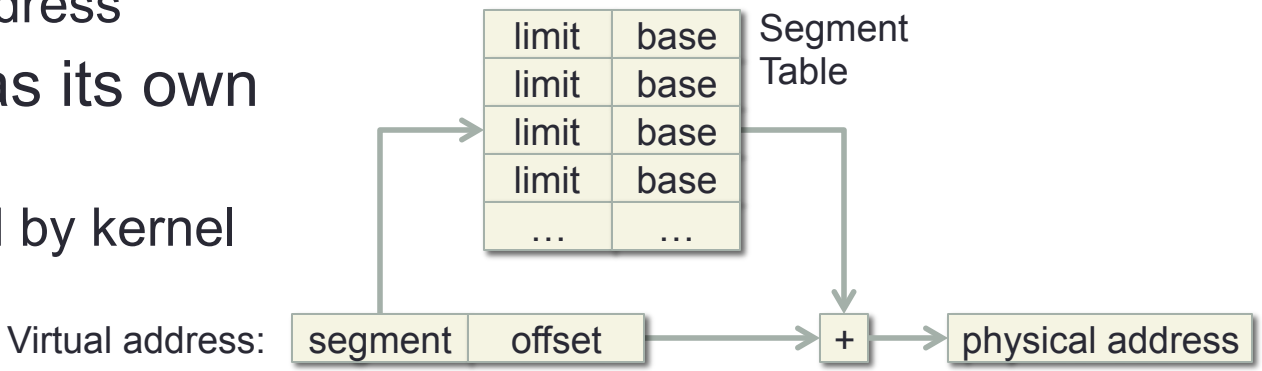
MMU: Relocation (3)

- Contiguous memory allocation mechanism is prone to fragmentation of physical memory
  - As processes terminate and other processes are started, must determine where to place each process in physical memory
  - Placement strategies are same as always, e.g. best fit, first fit, …
  - (OS can use another strategy too; in a moment…)
- This mechanism also requires standard swapping
  - Not really possible to swap out parts of a process
- Does this virtual addressing mechanism allow shared memory areas?
  - *phys_addr = virt_addr + relocation*
- Not without great difficulty:
  - Two processes could be given overlapping regions, but it would complicate other parts of process management



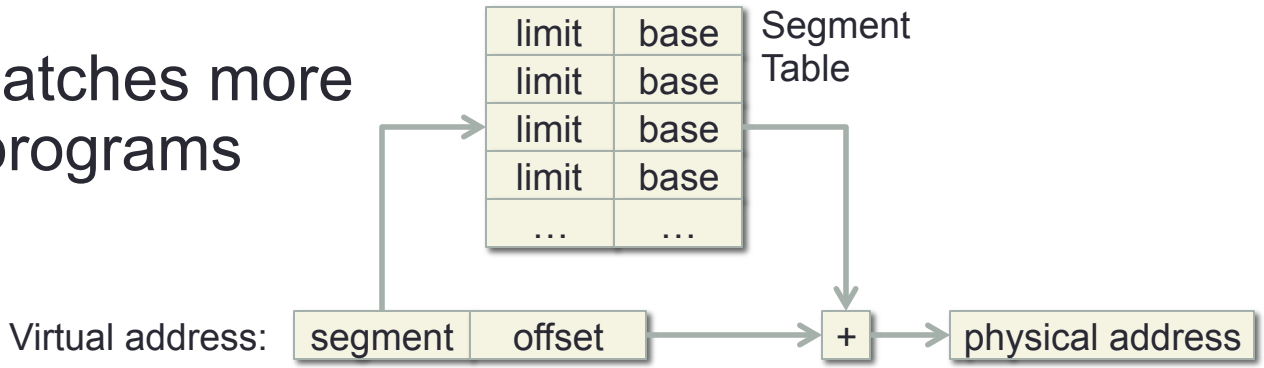
MMU: Segmentation

- A more advanced virtual address mapping technique is called **segmentation**
- Virtual addresses also include a **segment number**
  - Virtual address = segment number + offset within segment
- Segment number used to find an entry in **segment table**
- Similar mechanism to the relocation register:
  - Virtual offset is checked against the segment's limit; if limit is exceeded, MMU generates a fault
  - Otherwise, virtual address is added to the segment's base value to get a physical address
- Each process has its own segment table
  - Only manipulated by kernel



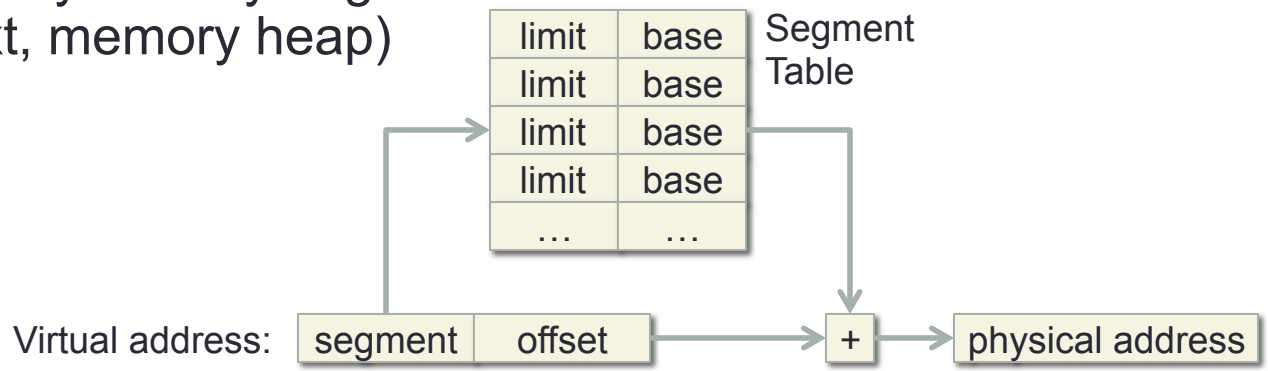
MMU: Segmentation (2)

- Segments can specify additional characteristics:
  - Read-only vs. read-write, executable code vs. data
  - MMU can also enforce these constraints
- Programs frequently contain different kinds of data
  - Program text (read-only, executable)
  - Constants/read-only data (read-only, not executable)
  - Global variables (read-write, not executable)
  - Memory heap
  - Program stack (or stacks, for multithreaded programs)
  - Shared libraries
- Segmentation matches more closely to what programs actually require



MMU: Segmentation (2)

- Segmentation allows a program's virtual memory to be non-contiguous in physical memory
  - Reduces physical memory fragmentation issues somewhat, but it will still be an issue over time
- Also supports shared memory areas very easily
  - Multiple processes can have segments with the same base and limit values, allowing access to the same physical memory area
- Still not particularly great for swapping processes
  - Could swap individual segments, but segments can easily be very large (e.g. program text, memory heap)

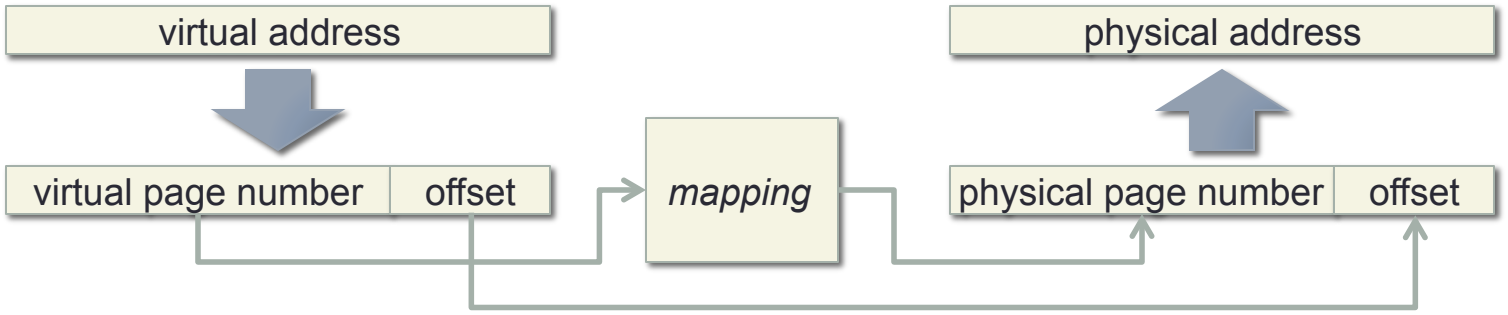


Compaction

- Both relocation register and segmented memory models can suffer from external fragmentation of physical memory
- OS can mitigate this by **compacting** physical memory:
  - Move programs within physical memory to create a single contiguous area of free memory
  - A program's code and data can be moved within physical memory, then the base address(es) can be adjusted to reflect new location
- Increases the number of processes that a system can run
- Clearly has a time impact on system performance
  - Particularly when large programs or data areas must be moved
- OS can perform compaction when system load is lighter
  - e.g. via a low-priority kernel thread
- Or, just force compaction when it can't be avoided!

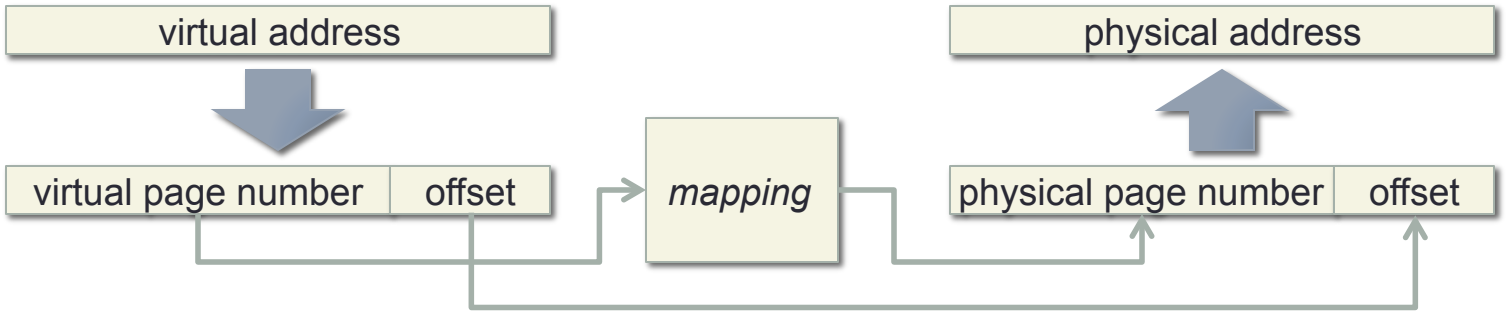
MMU: Paging

- **Paging** is the most common technique for mapping virtual addresses to physical addresses
  - Physical and virtual memory are divided into fixed-size blocks of a particular size, e.g. 4KB, 8KB, etc.
- Blocks of physical memory are called **frames**
- Blocks of virtual memory are called **pages**
- Every virtual page is mapped to a corresponding frame in physical memory



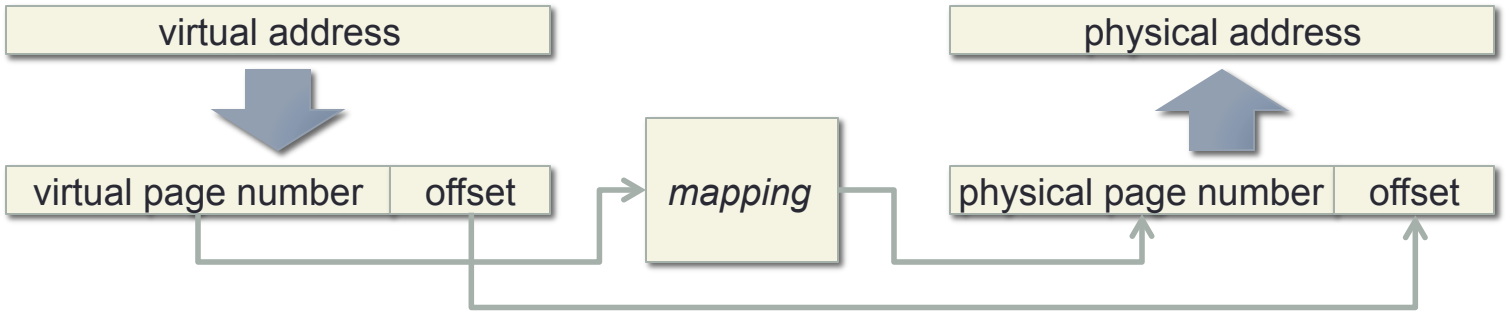
MMU: Paging (2)

- Paging causes no external fragmentation whatsoever
  - Memory is always allocated or released in page-size chunks
- Will have some limited amount of internal fragmentation
  - e.g. a process needs less than one page of space, but receives a whole page
- Motivates choice of a page size that is relatively small, but still large enough to make swapping reasonably efficient
  - Most allocations require a larger number of pages, reducing actual internal fragmentation costs



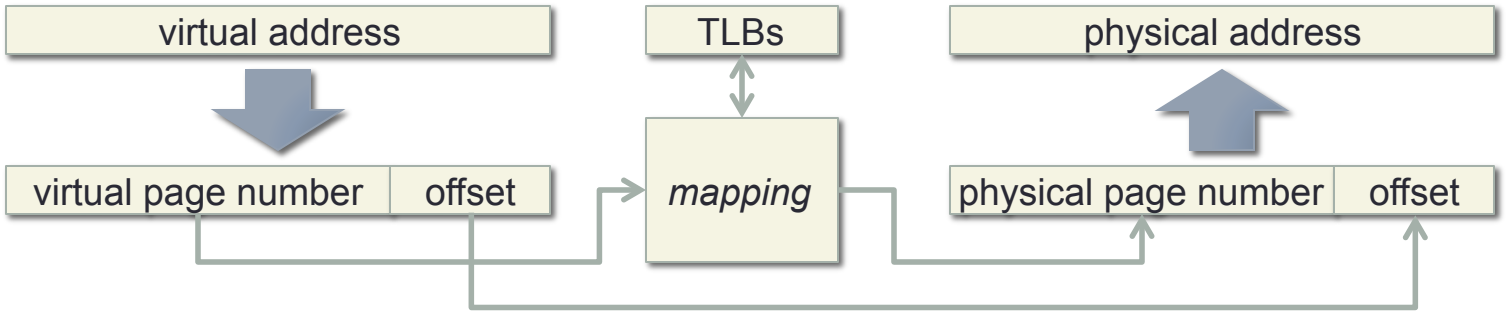
MMU: Paging (3)

- Some systems have a very small mapping of virtual pages to physical pages
  - Store this mapping in dedicated registers
- Example: DEC PDP-11
  - Address space: 16 bits (64KB)
  - Page size: 8KB (13 bits of addresses); only 8 pages total
  - Virtual to physical page mapping is stored in 8 registers



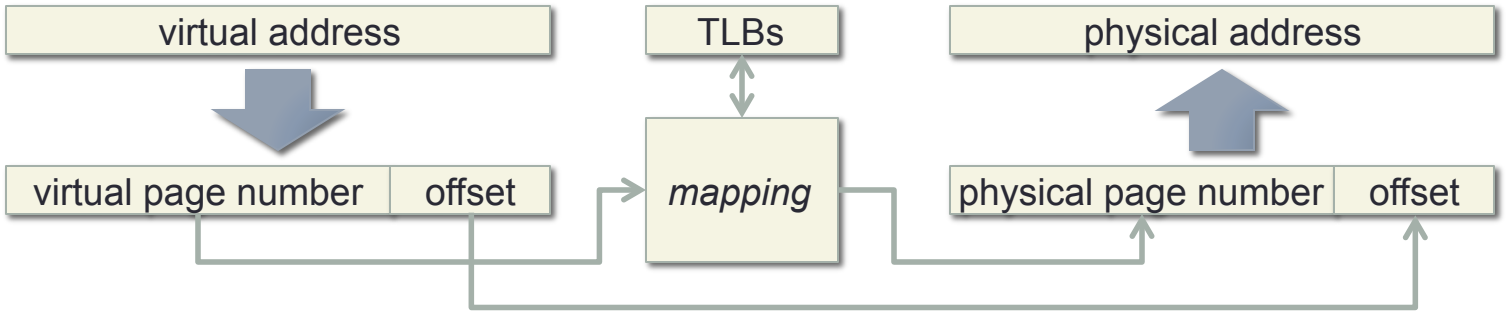
MMU: Paging (4)

- Most modern systems must support very large mappings in a **page table**
  - An entry for every virtual page, specifying the corresponding physical page
- This page table is stored in main memory…
  - Page table must be consulted for every memory access, including both code and data access
  - Main memory is very slow, e.g. ~100 clocks per access
- MMUs include Translation Lookaside Buffers (TLBs) to maximize performance of address translation



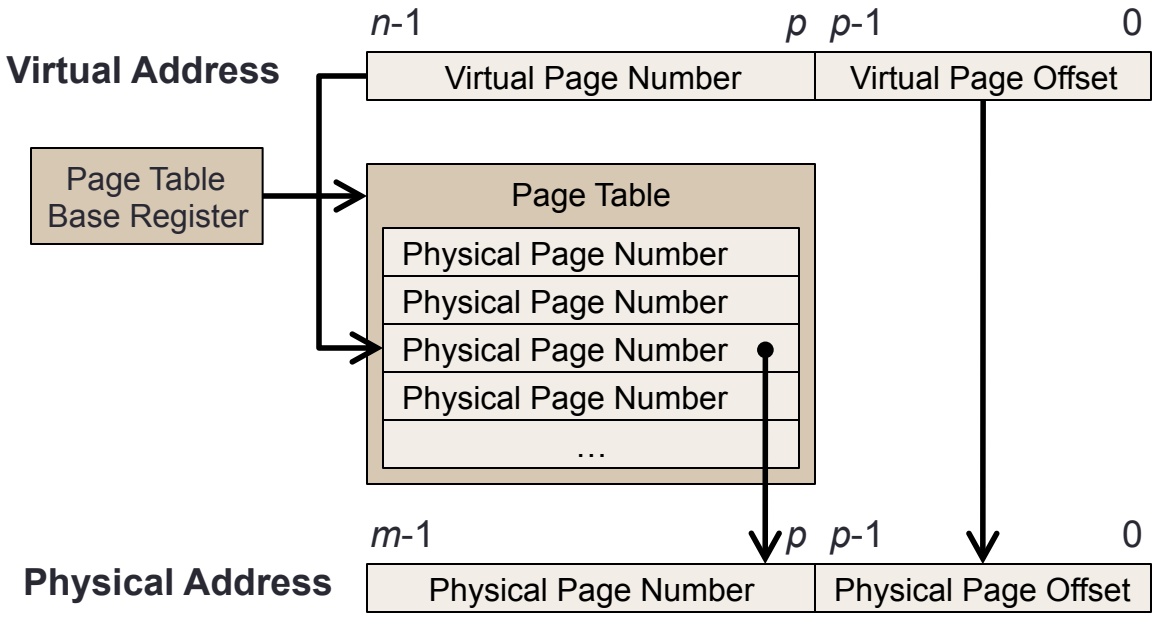
MMU: Paging (5)

- During address translation, the MMU checks the TLBs to see if the mapping is already cached
  - Frequently will be, if the program has good locality
- If not, the MMU suffers a **TLB miss**
  - Must look at the actual page table in memory to complete the address translation
- TLB misses can be resolved by hardware or by software
  - If page table format is simple, hardware can look up the information
  - Some CPUs can also fire a "TLB miss" interrupt to allow the OS kernel to resolve the TLB miss (much slower, obviously…)



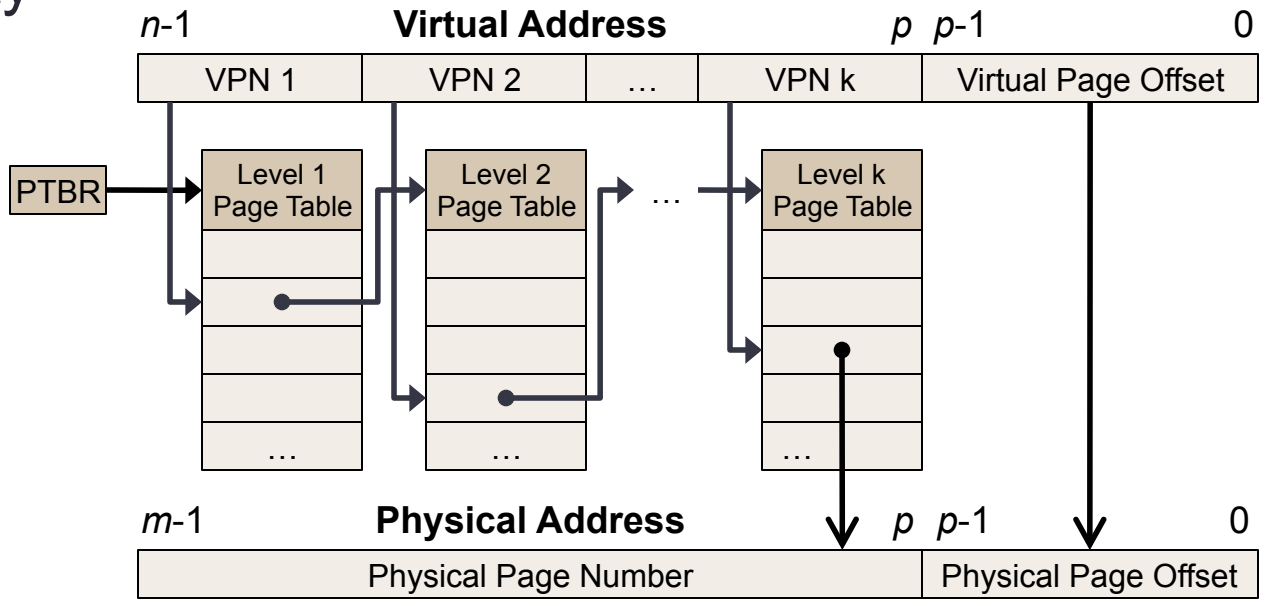
Simple Page Tables

- Simplest page table holds one entry for each virtual page
- As the size of the address space grows, page table also becomes prohibitively large
- Example: IA32 address space = 32 bits (4GB)
  - Page size is 4KB; 1048576 entries in page table!
  - Entries are 32 bits; table takes up 4MB



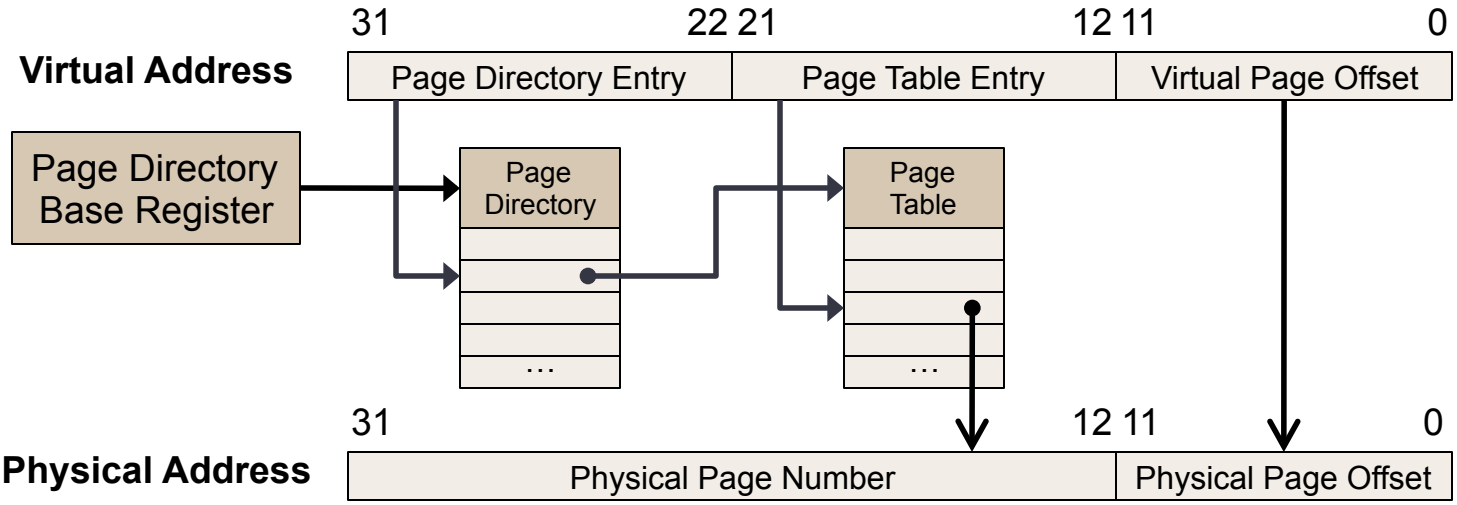
Hierarchical Paging

- To support larger address spaces, many systems use **hierarchical paging**
  - Page table is a sparse data structure
- Virtual page number is broken into parts – each part is used to index a page table at a different level
- If a memory area is unused, the corresponding page table entries are empty



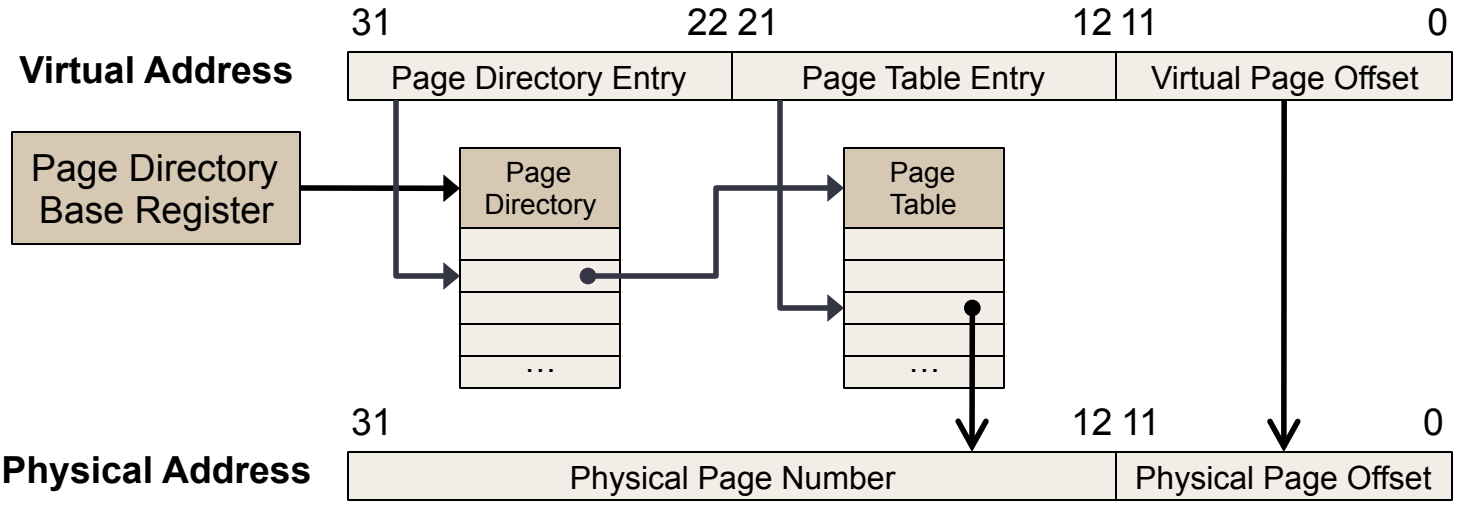
Hierarchical Paging (2)

- Example: IA32 has 32-bit (4GB) address space
  - 4KB pages; 12 bits of address are offset within page
  - 20 bits of address specify virtual/physical page number
- IA32 has a two-level page table hierarchy
  - Top 10 bits are used to index into the page directory
  - Next 10 bits are used to index into a second-level page table



Hierarchical Paging (3)

- Similarly, x86-64 has 48-bit address space
  - Also 4KB pages; 12 bits of address are offset within page
  - 36 bits of address specify virtual/physical page number
- x86-64 has a four-level page table hierarchy
  - Each level has 9 bits to specify the page-table index
- Problem: as address space grows, number of accesses for page-table lookup will clearly become prohibitive…



Hashed Page Tables

- Some processors support **hashed page tables**
  - Virtual page number is hashed to find corresponding physical page
- Obvious issue: hash collisions in the page table
  - Must have some way of resolving hash collisions, e.g. overflow buckets or open addressing
  - Must store both the virtual page number and the corresponding physical page number to resolve collisions
- Systems with large address spaces can use **clustered page tables**
  - Entries in the hash table hold multiple virtual/physical mappings
  - e.g. a clustered page table might hold 16 virtual/physical mappings instead of just one
- Usually requires kernel intervention to resolve TLB misses
  - More advanced CPUs can do this in microcode on the processor

Inverted Page Tables

- Another solution to the “large page table” problem is to use **inverted page tables**
  - Instead of using a table that stores the virtual-to-physical mapping, store a physical-to-virtual mapping instead
- With traditional page tables, each process has its own page table…
- With inverted page tables, the entire system has one page table containing the mappings of all processes
- Problem: processes use virtual addresses…
  - Very difficult to find a process’ virtual-to-physical mapping in the inverted page table
  - Frequently, such systems use a hashing mechanism on top of the inverted page table, to find the appropriate records
  - (PowerPC and UltraSPARC processors use this approach.)

Inverted Page Tables (2)

- Inverted page tables have a second problem:
  - Each physical page is mapped to one virtual page…
  - Providing shared memory on such systems is complicated
- A simple solution for a single-processor system:
  - When the kernel dispatches to the current process, check if it has any shared memory areas it is using
  - If so, update the inverted page table to reflect that the physical pages are owned by the current process
  - When another process is scheduled, update the page table to show the pages as owned by the next process
- Various other solutions to this problem as well…

Next Time

- Continue discussion of virtual memory and paging