CS162
Operating Systems and Systems Programming
Lecture 9

Address Translation

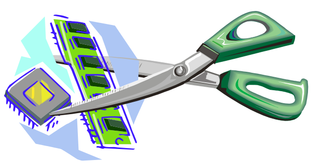
February 25, 2013
Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

Virtualizing Resources

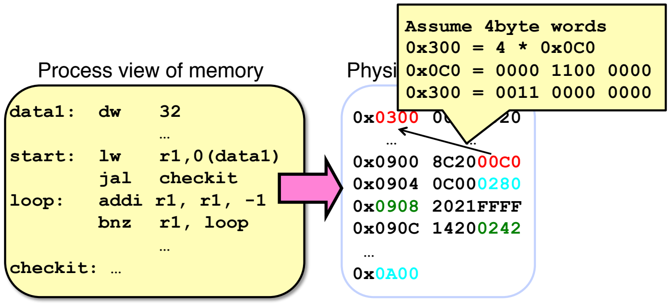


- Physical Reality: Processes/Threads share the same hardware
  - Need to multiplex CPU (CPU Scheduling)
  - Need to multiplex use of Memory (Today)
- Why worry about memory multiplexing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different processes use the same memory
  - Probably don't want different processes to even have access to each other's memory (protection)

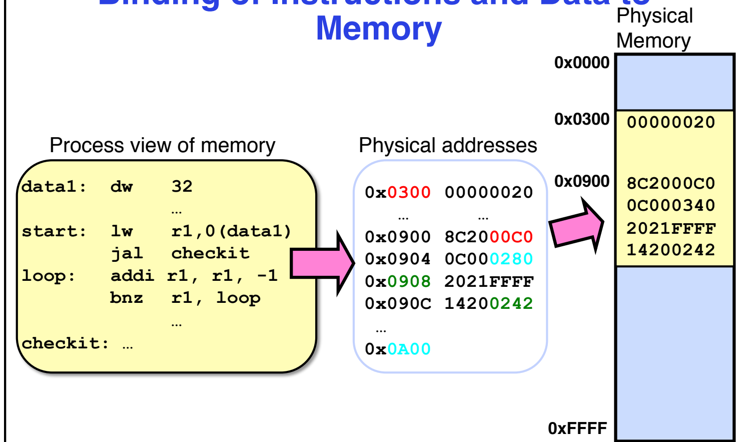
Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc)
    - Kernel data protected from User programs
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, process uses virtual addresses, physical memory uses physical addresses

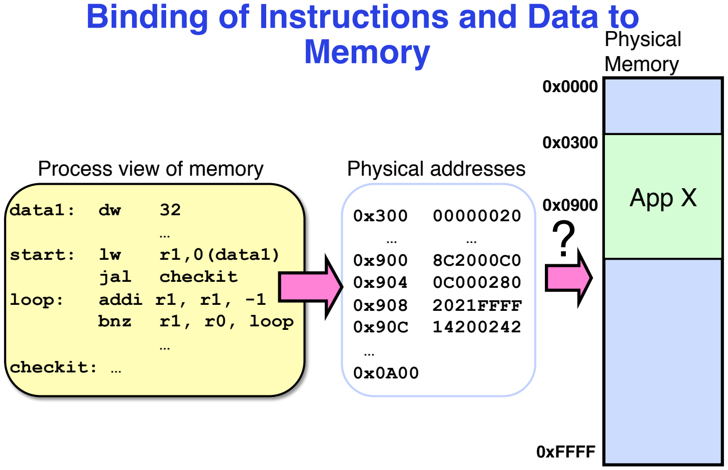
Binding of Instructions and Data to Memory



Binding of Instructions and Data to Memory

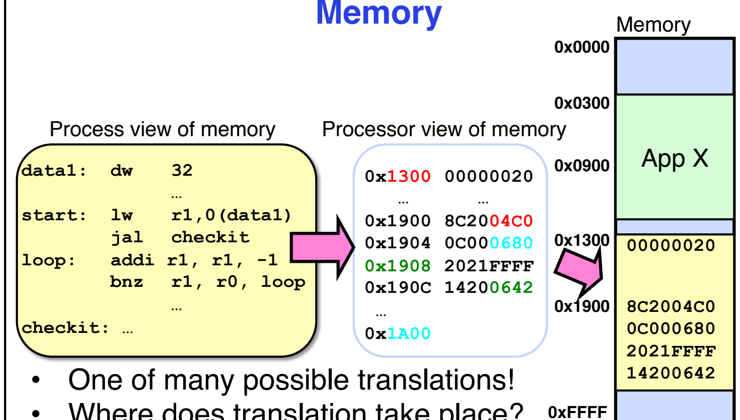


Binding of Instructions and Data to Memory



Need address translation!

Binding of Instructions and Data to Memory

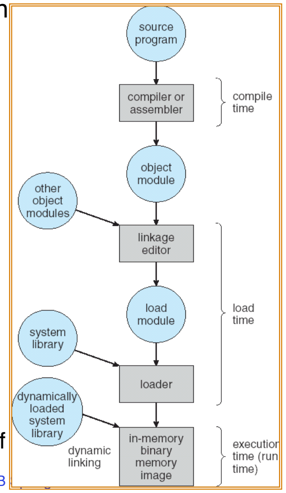


- One of many possible translations!
- Where does translation take place?

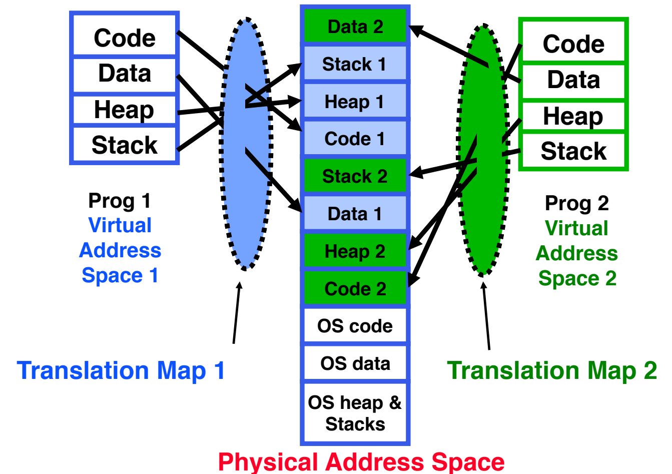
Compile time, Link/Load time, or Execution time?

Multi-step Processing of a Program for Execution

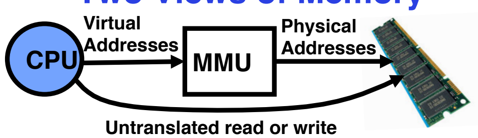
- Preparation of a program for execution involves components at:
  - Compile time (i.e., "gcc")
  - Link/Load time (unix "ld" does link)
  - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Example of General Address Translation



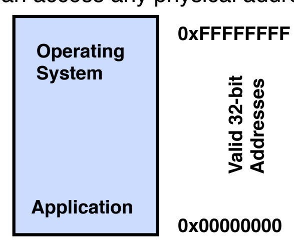
Two Views of Memory



- Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

Uniprogramming (MS-DOS)

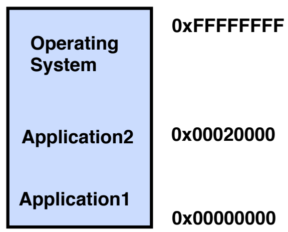
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (First Version)

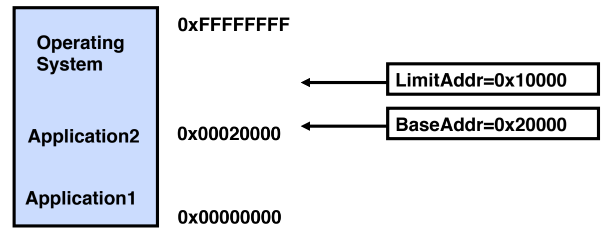
- Multiprogramming without Translation or Protection
  – Must somehow prevent address overlap between threads



  – Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    » Everything adjusted to memory location of program
    » Translation done by a linker-loader
    » Was pretty common in early days
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

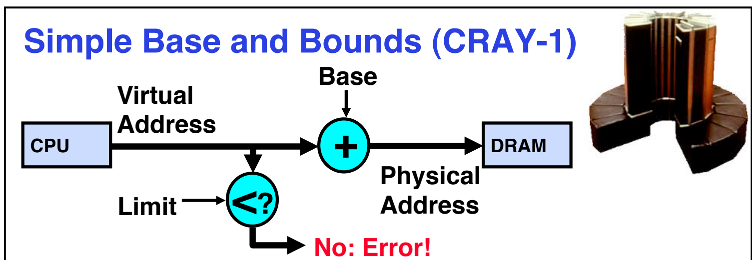
Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



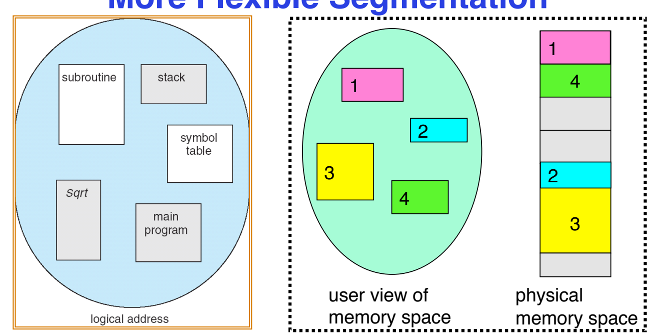
  – Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
    » If user tries to access an illegal address, cause an error
  – During switch, kernel loads new base/limit from TCB (Thread Control Block)
    » User not allowed to change base/limit registers

Simple Base and Bounds (CRAY-1)



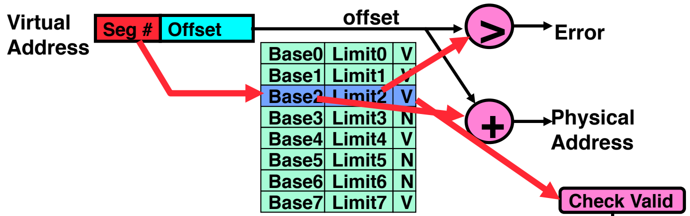
- Could use base/limit for dynamic address translation (often called "segmentation") – translation happens at execution:
  – Alter address of every load/store by adding "base"
  – Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  – Program gets continuous region of memory
  – Addresses within program do not have to be relocated when program placed in different region of DRAM

More Flexible Segmentation



- Logical View: multiple separate segments
  – Typical: Code, Data, Stack
  – Others: memory sharing, etc
- Each segment is given region of contiguous memory
  – Has a base and limit
  – Can reside anywhere in physical memory

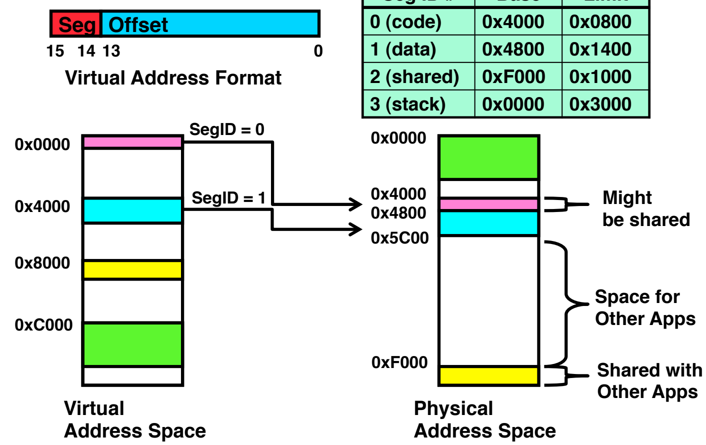
Implementation of Multi-Segment Model



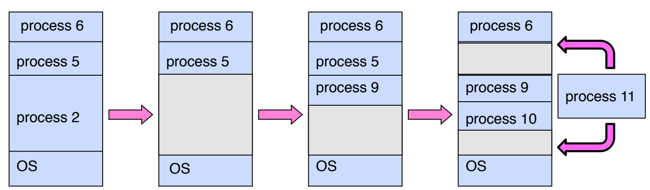
- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: mov [es:bx],ax.
- What is "V/N" (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addresses)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



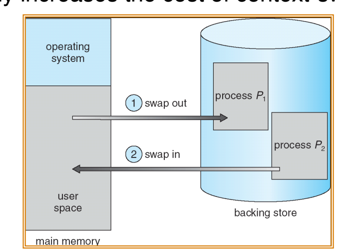
Issues with Simple Segmentation Method



- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

Schematic View of Swapping

- Q: What if not all processes fit in memory?
- A: Swapping: Extreme form of Context Switch
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching



- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  – External: free gaps between allocated chunks
  – Internal: don't need all memory within allocated chunks

Administrivia

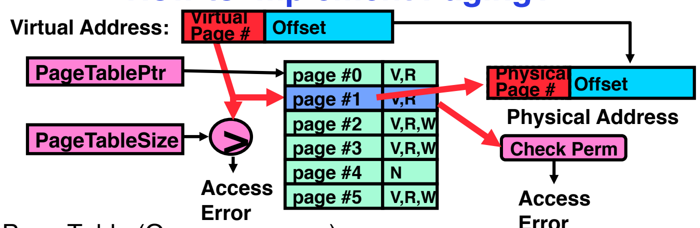
- Midterm date is still pending until campus provides a time and room
  – Requested Monday 3/11 or Wednesday 3/13
- Midterm review session TBA
- Project 1 due tomorrow Tue 2/26 (`submit proj1-code`) at 11:59PM
  – Design doc (`submit proj1-final-design`) and group evals (Google Docs form) due Wed 2/27 at 11:59PM

5min Break

Paging: Physical Memory in Fixed Size Chunks

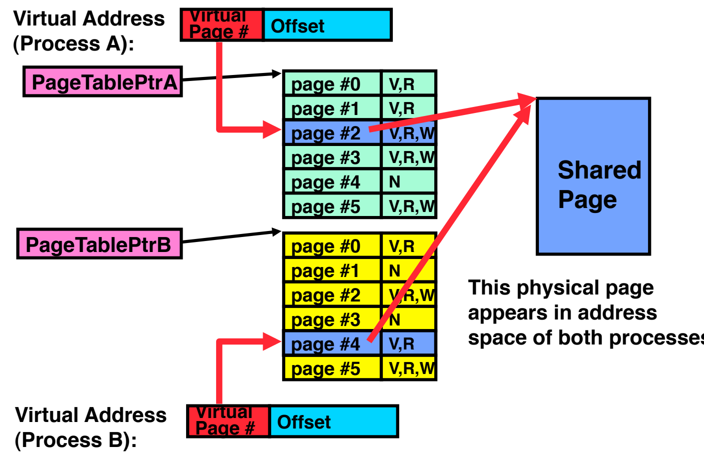
- Solution to fragmentation from segments?
  – Allocate physical memory in fixed size chunks ("pages")
  – Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation: 00110001110001101 … 110010
    » Each bit represents page of physical memory 1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  – No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  – Consequently: need multiple pages/segment

How to Implement Paging?



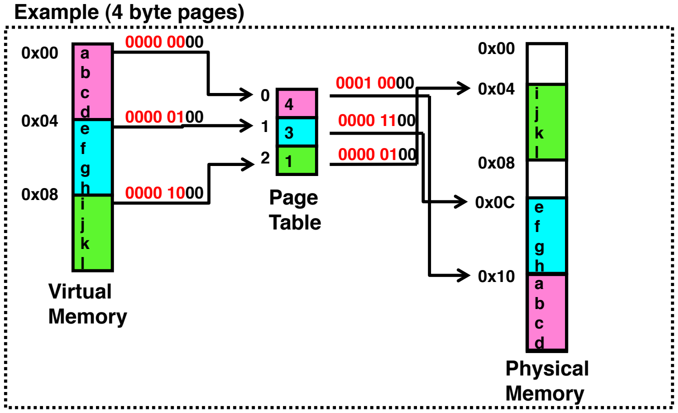
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



Simple Page Table Example

Example (4 byte pages)

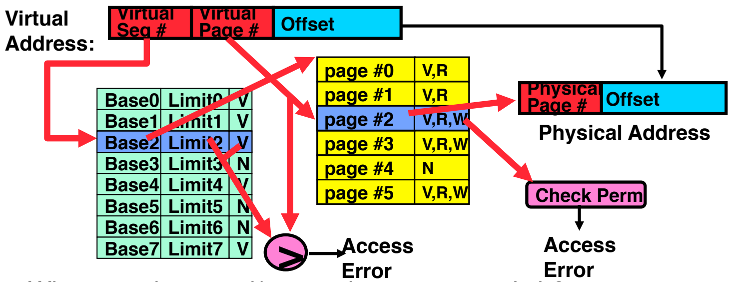


Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

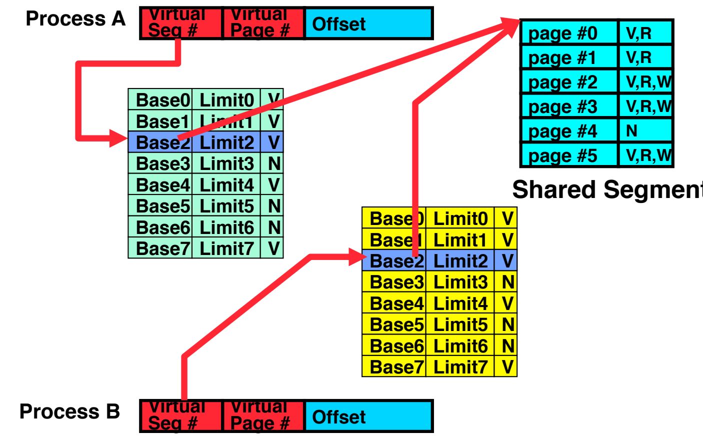
Multi-level Translation

- What about a tree of tables?
  – Lowest level page table⇒memory still allocated with bitmap
  – Higher levels often segmented
- Could have any number of levels. Example (top segment):

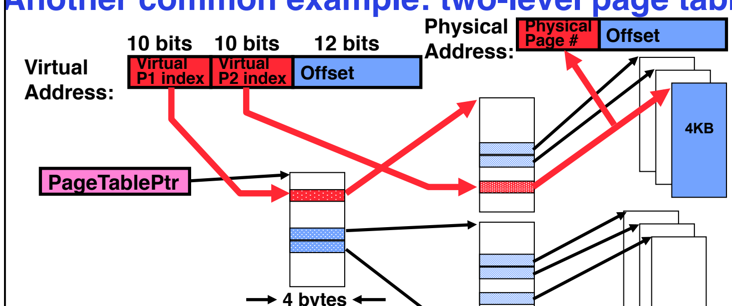


- What must be saved/restored on context switch?
  – Contents of top-level segment registers (for this example)
  – Pointer to top-level table (page table)

What about Sharing (Complete Segment)?



Another common example: two-level page table



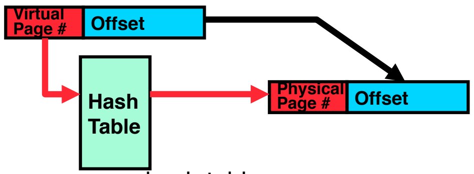
- Tree of Page Tables
- Tables fixed size (1024 entries)
  – On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  – Don't need every 2^{nd}-level table
  – Even when exist, 2^{nd}-level tables can reside on disk if not in use

Multi-level Translation Analysis

- Pros:
  – Only need to allocate as many page table entries as we need for application – size is proportional to usage
    » In other words, sparse address spaces are easy
  – Easy memory allocation
  – Easy Sharing
    » Share at segment or page level (need additional reference counting)
- Cons:
  – One pointer per page (typically 4K – 16K pages today)
  – Page tables need to be contiguous
    » However, previous example keeps tables to exactly one page in size
  – Two (or more, if >2 levels) lookups per reference
    » Seems very expensive!

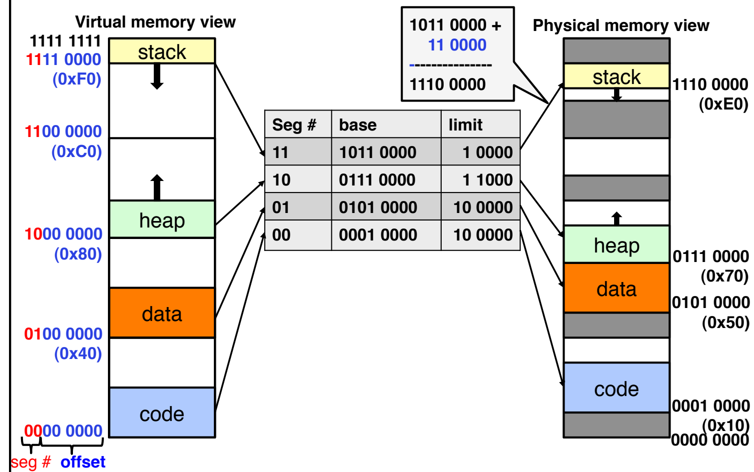
Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



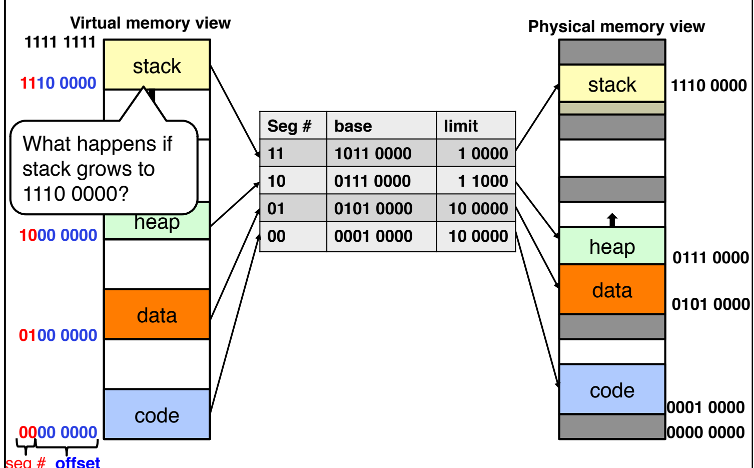
- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Summary: Address Segmentation



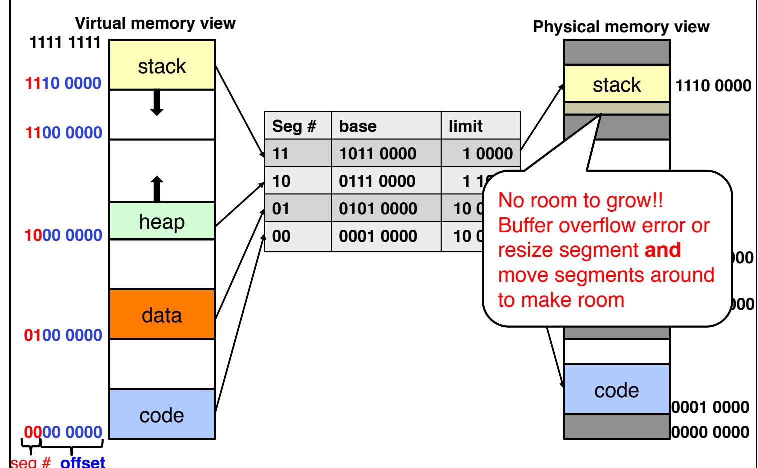
Seg #	base	limit
11	1011 0000	1 0000
10	0111 0000	1 1000
01	0101 0000	10 0000
00	0001 0000	10 0000

Summary: Address Segmentation



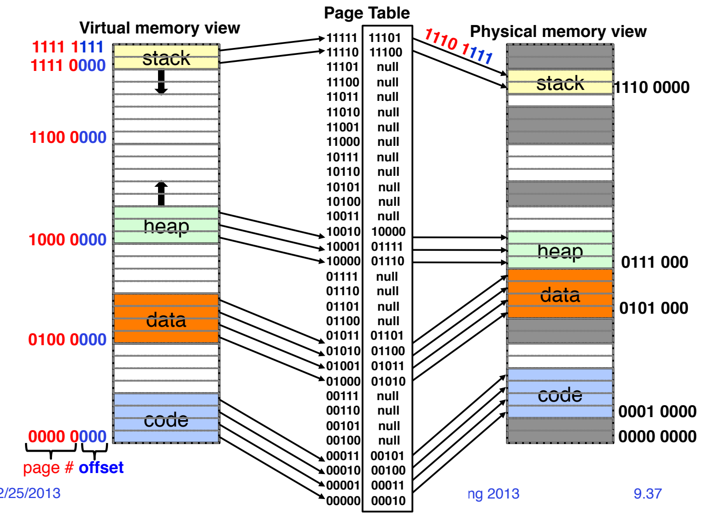
Seg #	base	limit
11	1011 0000	1 0000
10	0111 0000	1 1000
01	0101 0000	10 0000
00	0001 0000	10 0000

Recap: Address Segmentation

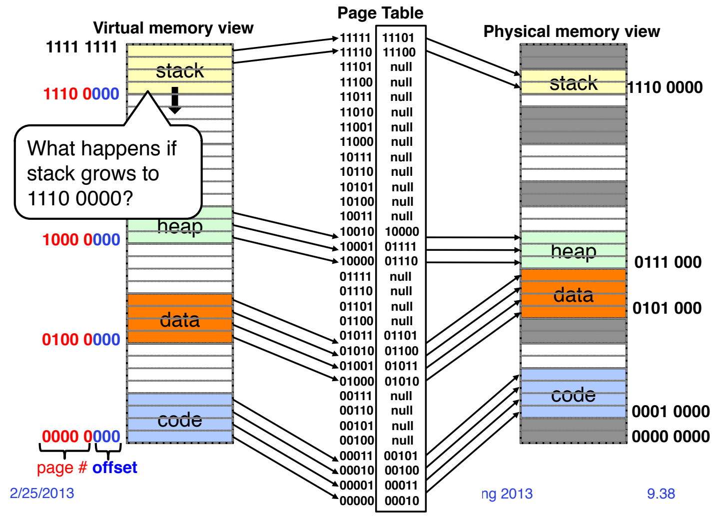


Seg #	base	limit
11	1011 0000	1 0000
10	0111 0000	1 1
01	0101 0000	10 0
00	0001 0000	10 0

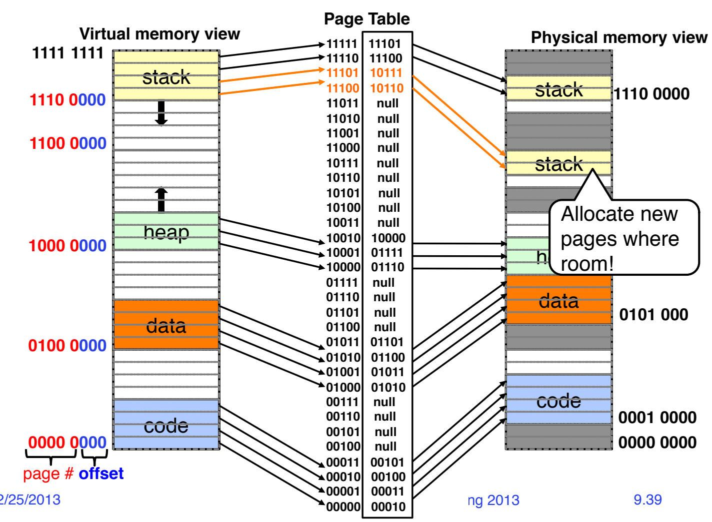
Summary: Paging



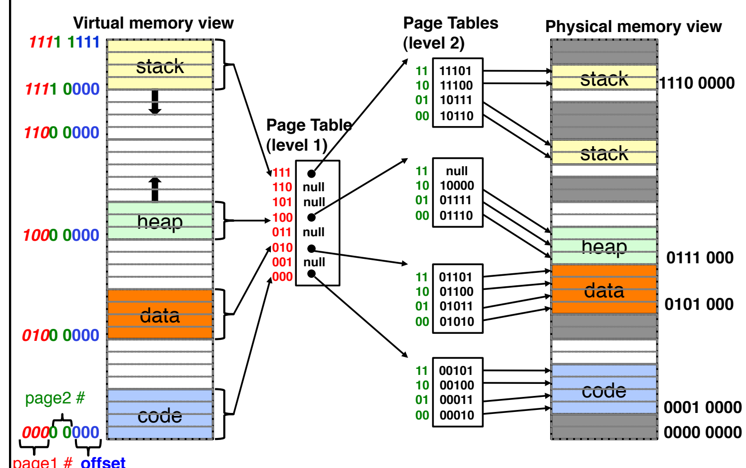
Summary: Paging



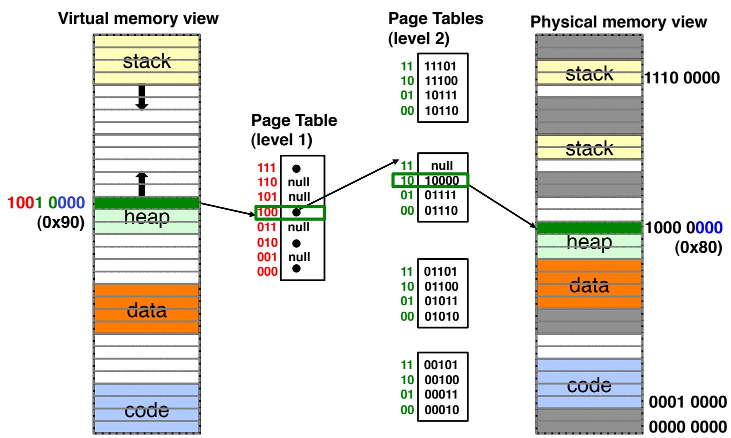
Summary: Paging



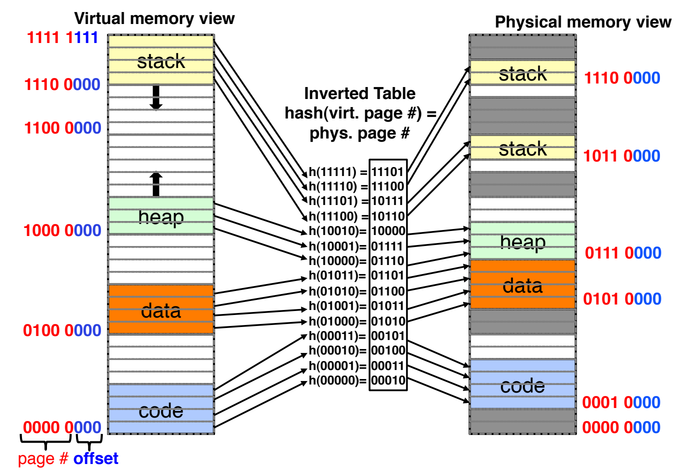
Summary: Two-Level Paging



Summary: Two-Level Paging



Summary: Inverted Table



Address Translation Comparison

	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory
Paged segmentation	Table size ~ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex

Summary

- Memory is a resource that must be multiplexed
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- Simple Protection through segmentation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table: size of page table related to physical mem. size