Memory Management

Dave Eckhardt
de0u@andrew.cmu.edu

Synchronization

- Project 2
  - I called sys_minclone() and something bad happened!
  - You all saw the change notice on the bboard, right?
- “Pop Quiz”
  - What does “ld” do?
- Outline
  - ~ Chapter 9 (with occasional disagreement)
  - Also read Chapter 10

Who emits addresses?

- Program counter (%eip): code area
  - Straight-line code
  - Loops, conditionals
  - Procedure calls
- Stack pointer (%esp, %ebp): stack area
- Registers: data/bss/heap

Initialized how?

- Program counter
  - Set to “entry point” by OS program loader
- Stack pointer
  - Set to “top of stack” by OS program loader
- Registers
  - Code segment (“immediate” constants)
  - Data/BSS/heap
  - Computed from other values

Birth of an Address

```
int k = 3;
int foo(void) {
  return (k);
}

int a = 0;
int b = 12;
int bar (void) {
  return (a + b);
}
```

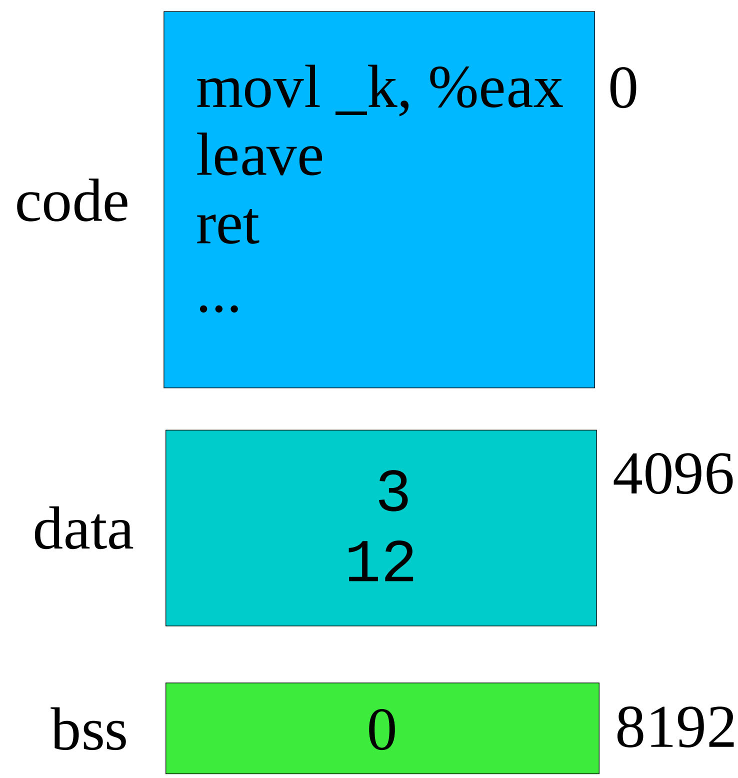
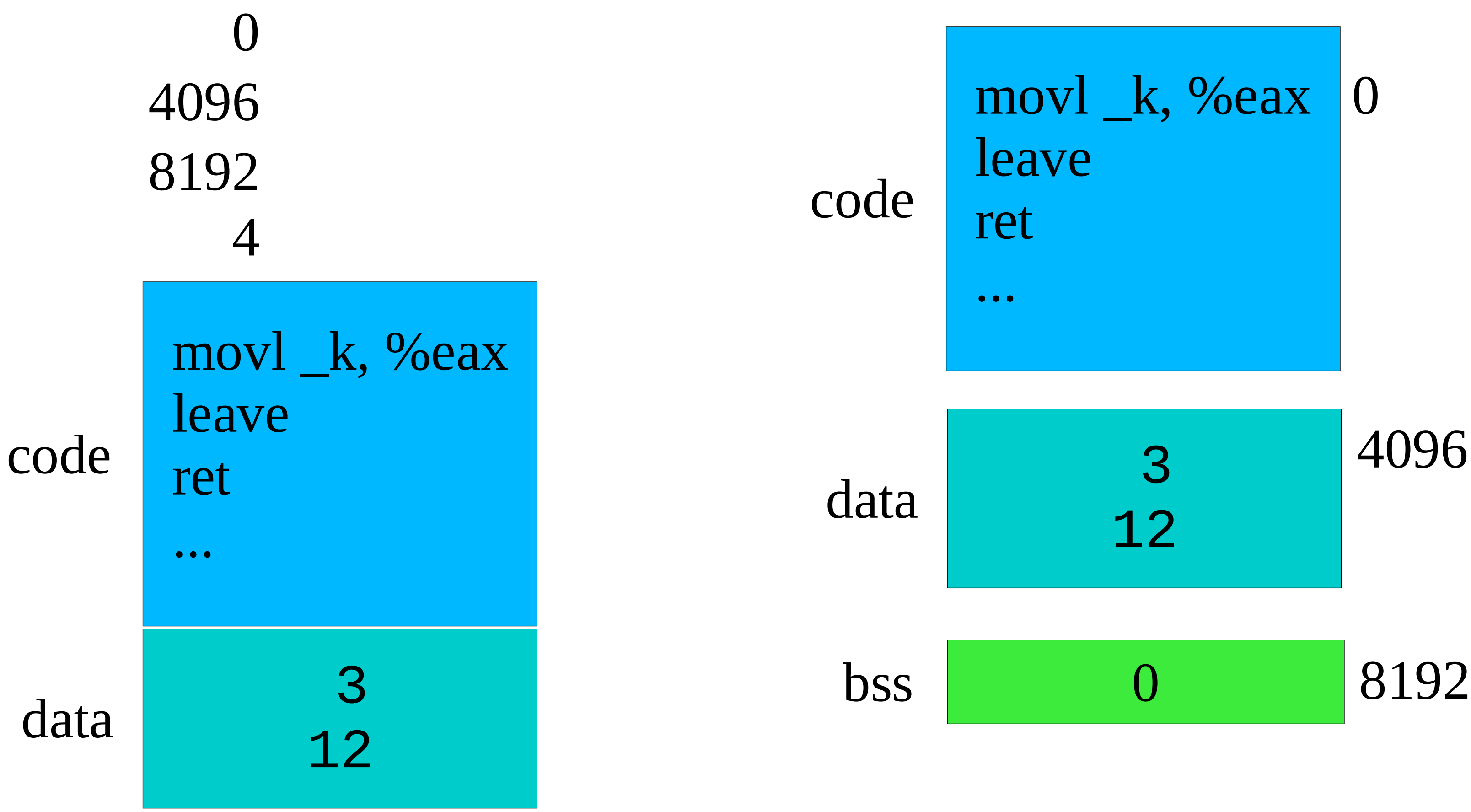


Image File vs. Memory Image



Multi-file Programs?

- “Link editor” combines into one image file
  - Unix “link editor” called “ld”
- Memory range allocation?
  - Each file uses same memory map!
- Linker can “fix up”
  - relocation directive
    - address, bit field
    - reference type
    - symbol name

Logical vs. Physical Addresses

- Logical address
  - According to programmer, compiler, linker
- Physical address
  - Where your program ends up in memory
  - They can't ***all*** be in the same place!
- How to reconcile?
  - Relocate “one last time”?
  - Use hardware!

Static Linking

- Must link a program before running
  - User program
  - Necessary library routines
- Duplication on disk
  - ***Every*** program uses printf()!
- Duplication in memory
- Hard to patch every printf()

Dynamic Linking

- Defer "final link" as much as possible
  - The instant before execution
- Program startup invokes "shared object loader"
  - Locates library files
  - Includes in address space
  - Links, often incrementally
    - Self-modifying "stub" routines

“Shared libraries”

- Extension/optimization of dynamic linking
- Basic idea
  - Why have N copies of printf() in memory?
  - Allow processes to share memory pages
    - “Intelligent” mmap()
  - Must avoid address-map conflicts
    - Library issued an address range
    - Position-independent code

Swapping

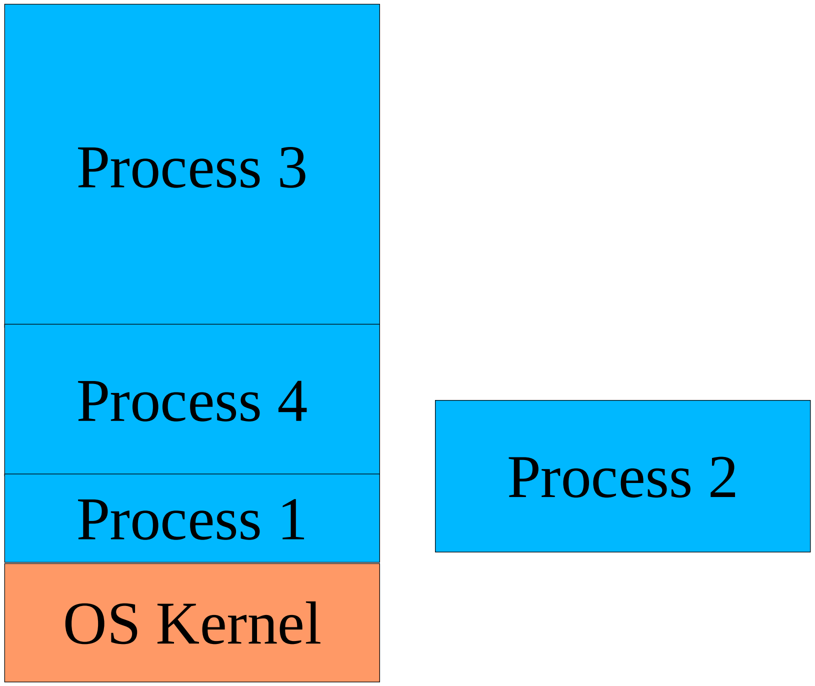
- Multiple user processes
  - Sum of memory demands exceeds system memory
  - Don't want say “no” too early
    - Allow each process 100% of system memory
- Take turns
  - Temporarily evict process(es) to disk
    - Not runnable
    - Blocked on ***implicit*** I/O request

Swapping vs. CPU Scheduling

- Textbook claims
  - ***Dispatcher*** notices swapped-out process
    - Just before resuming execution!
    - Implication: huge stalls
- Two-level scheduling process
  - CPU scheduler schedules in-core processes
  - Swapper decides when to evict/reinstate
    - Cannot swap a process with pending DMA

Contiguous Memory Allocation

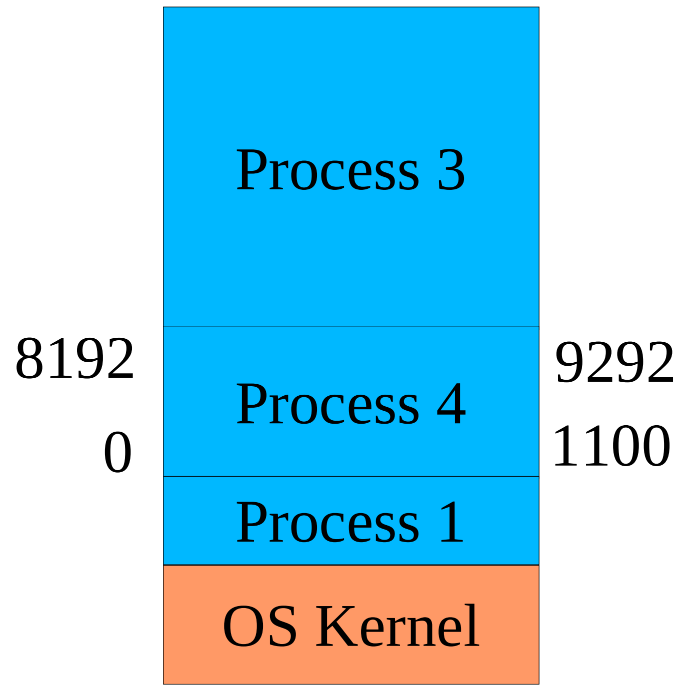
- Goal: share system memory among processes
- Approach: concatenate in memory
- Two new CPU registers
  - Memory base
  - Memory limit



Mapping & Protecting Regions

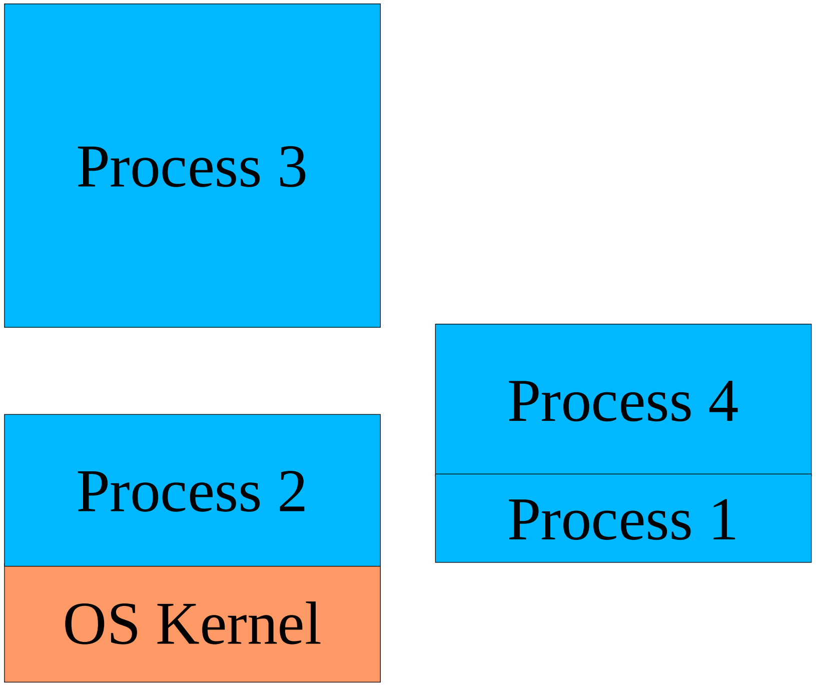
- Program uses ***logical*** addresses
- Memory Management Unit (MMU) maps to ***physical*** addresses

```
If V < limit
  P = base + V;
Else
  ERROR
```



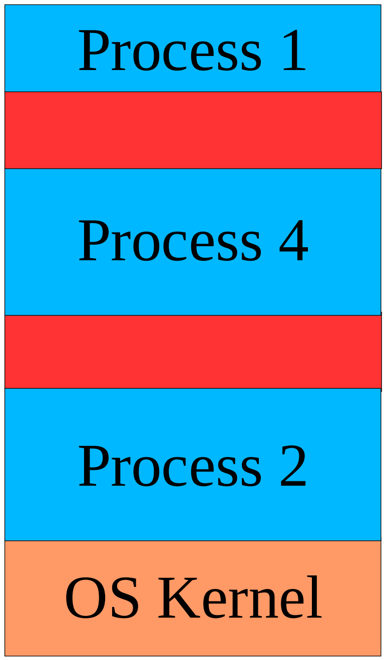
Allocating Regions

- Swapping out creates "holes"
- Swapping in creates ***smaller*** holes
- Various policies
  - First fit
  - Best fit
  - Worst fit



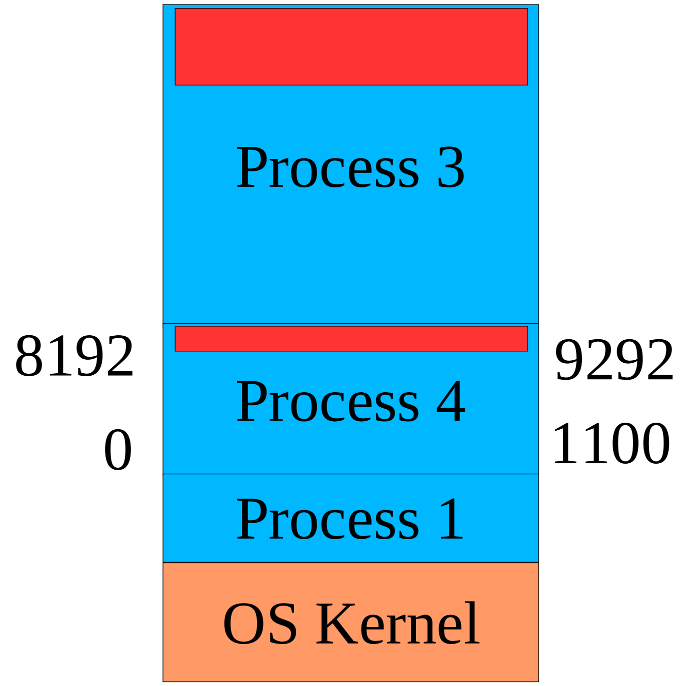
Fragmentation

- External fragmentation
  - Scattered holes can't be combined
    - Without costly "compaction" step
  - Some memory is unusable



Fragmentation

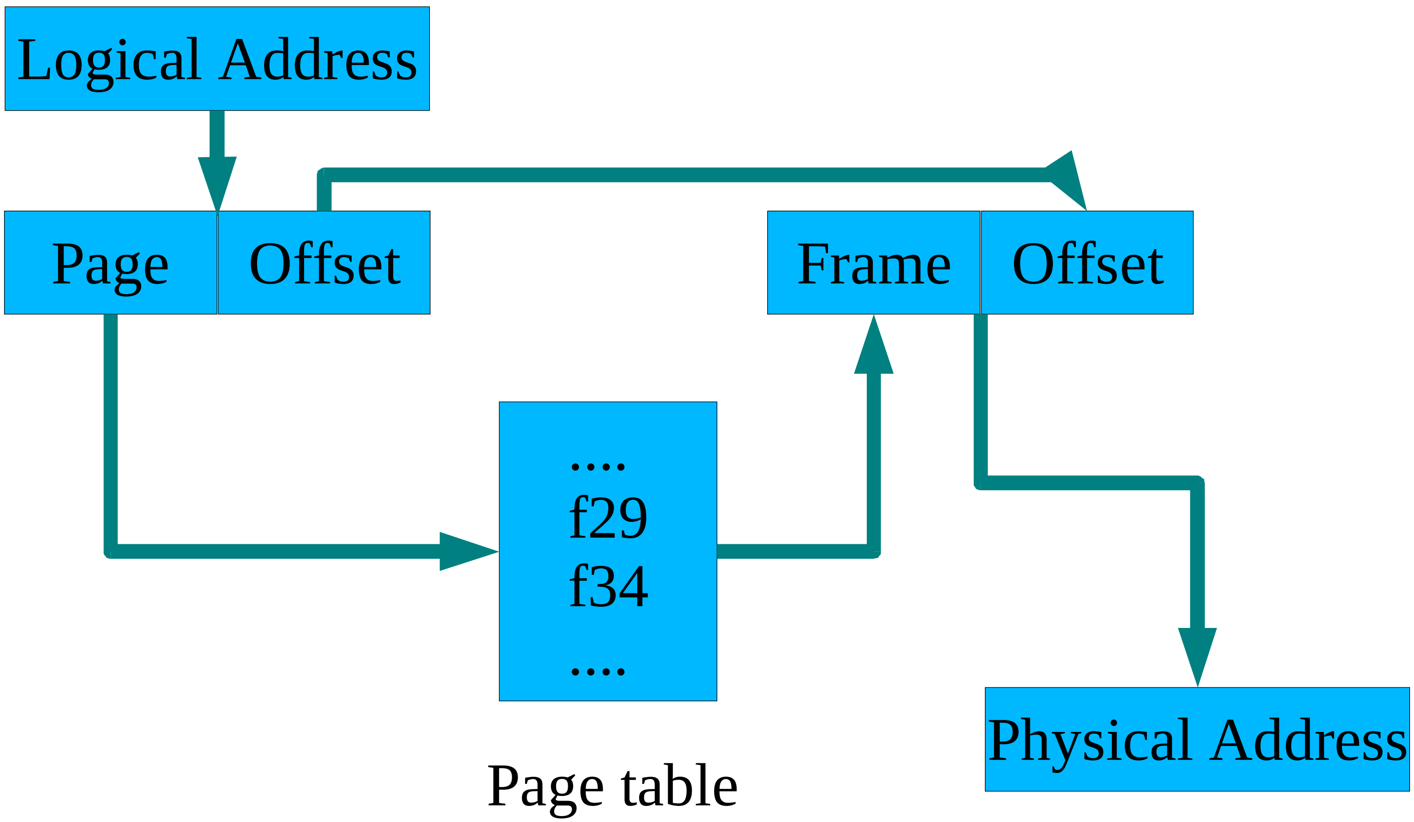
- Internal fragmentation
  - Allocators often round up
    - 8K boundary (***some*** power of 2!)
  - Some memory is wasted ***inside*** each segment



Paging

- Solve two problems
  - External memory fragmentation
  - Long delay to swap a whole process
- Divide memory more finely
  - ***Page*** = small logical memory region (4K)
  - ***Frame*** = small physical memory region
- Any page can map to any frame

Paging – Address Mapping



Paging – Address Mapping

- User view
  - Memory is a linear array
- OS view
  - Each process requires N frames
- Fragmentation?
  - ***Zero*** external fragmentation
  - Internal fragmentation: maybe average ½ page

Bookkeeping

- One page table for each process
- One frame table
  - Manage free pages
  - Remember who owns a page
- Context switch
  - Must install process page table

Hardware Techniques

- Small number of pages?
  - “Page table” can be a few registers
- Typical case
  - Large page tables, live in memory
  - Processor register: Page Table Base Register
  - Double trouble?
    - Program requests memory access
    - Processor makes ***two*** memory accesses!

Translation Lookaside Buffer (TLB)

- Problem
  - Cannot afford double memory latency
- Observation - “locality of reference”
  - Program accesses “nearby” memory
- Solution
  - Cache virtual-to-physical ***mappings***
    - Small, fast on-chip memory
    - Don't forget context switch!

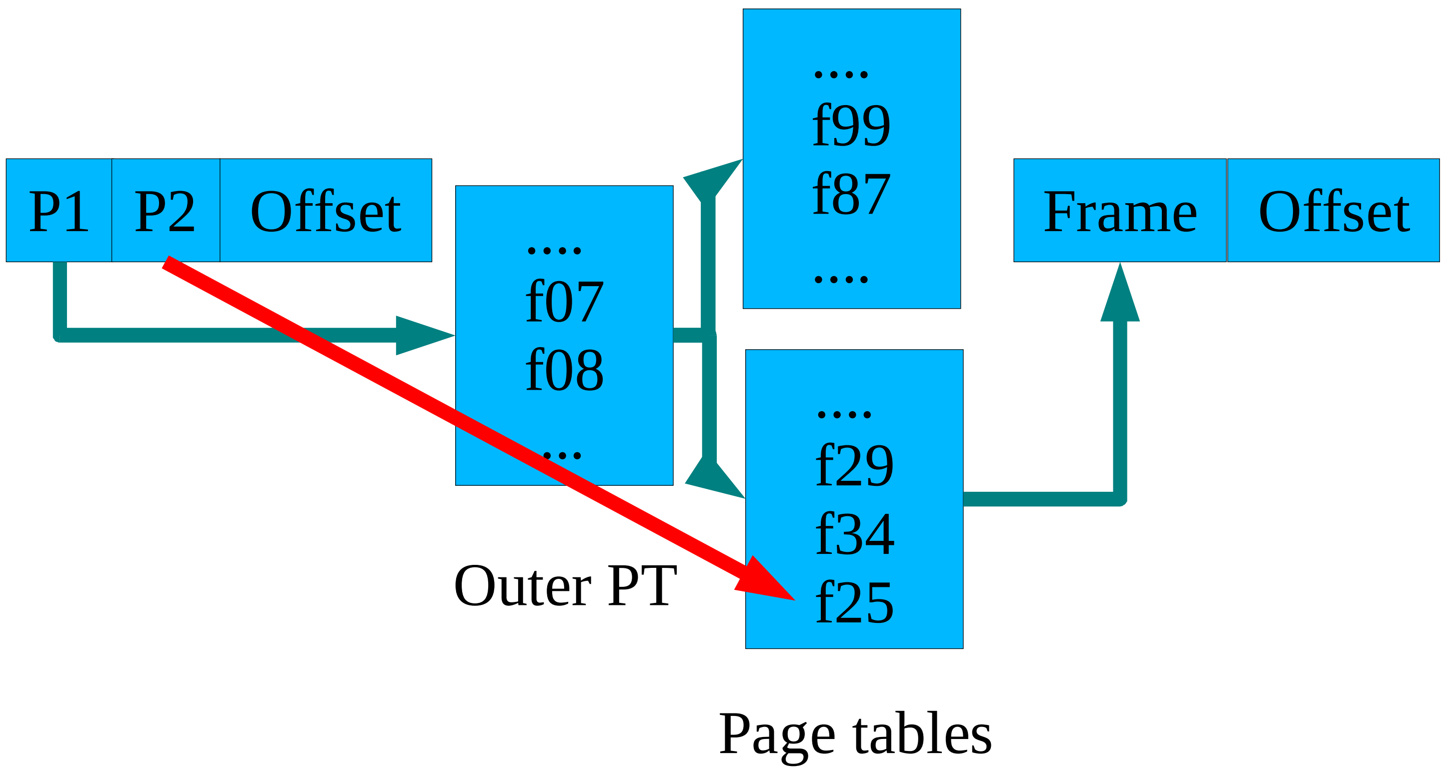
Page Table Entry (PTE) mechanics

- PTE flags
  - Protection
    - Read/Write/Execute bits
  - Valid bit
  - Dirty bit
- Page Table Length Register (PTLR)
  - Programs don't use entire virtual space
  - On-chip register detects out-of-bounds reference
    - Allows small PTs for small processes

Page Table Structure

- Problem
  - Assume 4 KByte pages, 4 Byte PTEs
  - Ratio: 1000:1
    - 4 GByte virtual address (32 bits) -> 4 MByte page table
      - Per process!
- Solutions
  - Multi-level page table
  - Hashed page table
  - Inverted page table

Multi-level page table



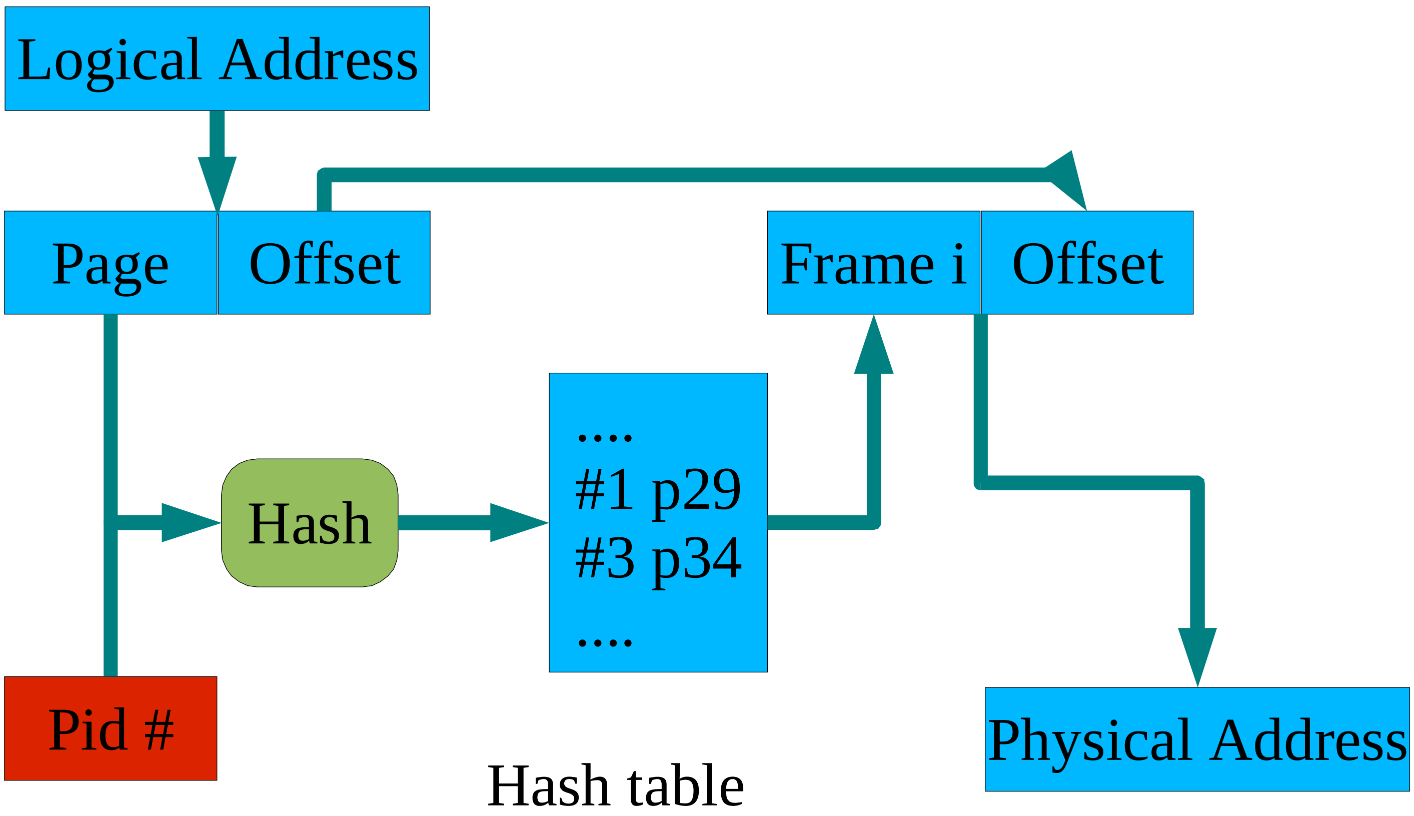
Hashing & Clustering

- Hashed Page Table
  - PT is “just” a hash table
    - Bucket chain entries: virtual page #, frame #, next-pointer
  - Useful for sparse PTs (64-bit addresses)
- Clustering
  - Hash table entry is a miniature PT
    - e.g., 16 PTEs
    - Entry can map 1..16 (aligned) pages

Inverted page table

- Problem
  - Page table size depends on virtual address space
  - N processes * large fixed size
- Observation
  - Physical memory (# frames) is a boot-time constant
  - No matter how many processes!
- Approach
  - One PTE per frame, maps (process #, page#) to index

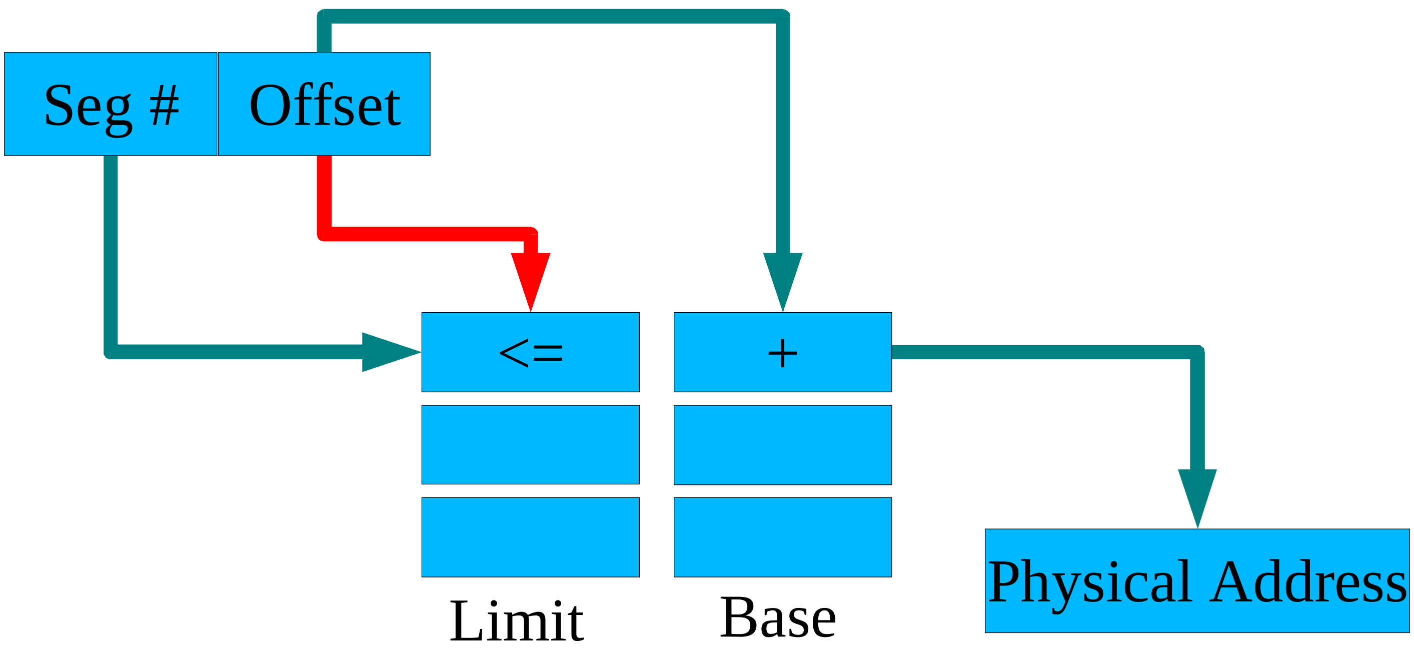
Inverted Page Table



Segmentation

- Physical memory is (mostly) linear
- Is virtual memory linear?
  - Typically a set of regions
    - “Module” = code region + data region
    - Region per stack
    - Heap region
- Why do regions matter?
  - Natural protection boundary
  - Natural ***sharing*** boundary

Segmentation: Mapping



Segmentation + Paging

- 80386 (does it ***all***!)
  - Processor address directed to one of six segments
    - CS: Code Segment, DS: Data Segment
      - CS register holds 16-bit selector
    - 32-bit offset within a segment -- CS:EIP
  - Table maps selector to segment descriptor
  - Offset fed to segment descriptor, generates linear address
  - Linear address fed through segment's page table
    - 2-level, of course

Is there another way?

- Could we have ***no*** page tables?
- How would hardware map virtual to physical?

Software TLBs

- Reasoning
  - We need a TLB for performance reasons
  - OS defines each process's memory structure
    - Which memory ranges, permissions
  - Why impose a semantic middle-man?
- Approach
  - TLB miss generates special trap
  - OS ***quickly*** fills in correct v->p mapping

Software TLB features

- Mapping entries can be computed many ways
  - Imagine a system with one process memory size
    - TLB miss becomes a matter of arithmetic
- Mapping entries can be locked in TLB
  - Great for real-time systems
- Further reading
  - http://yarchive.net/comp/software_tlb.html

Summary

- Processes emit virtual addresses
  - segment-based or linear
- A magic process maps virtual to physical
- No, it's ***not*** magic
  - Address validity verified
  - Permissions checked
  - Mapping may fail temporarily (trap handler)
  - Mapping results cached in TLB