

**15-410**

*“...We are Computer Scientists!...”*

**Virtual Memory #1**  
**Sep. 23, 2009**

**Dave Eckhardt**

**Garth Gibson**

# Outline

## Text

- Chapters 8, 9
- Reminder: reading list on class “Schedule” page

## The Problem: logical vs. physical

## Contiguous memory mapping

## Fragmentation

## Paging

- Type theory
- A sparse map

# Logical vs. Physical

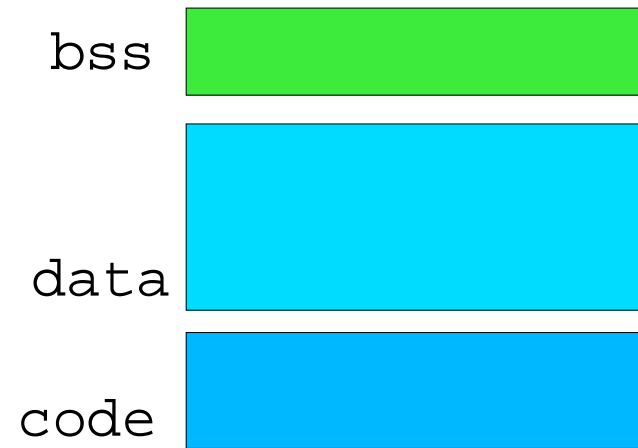
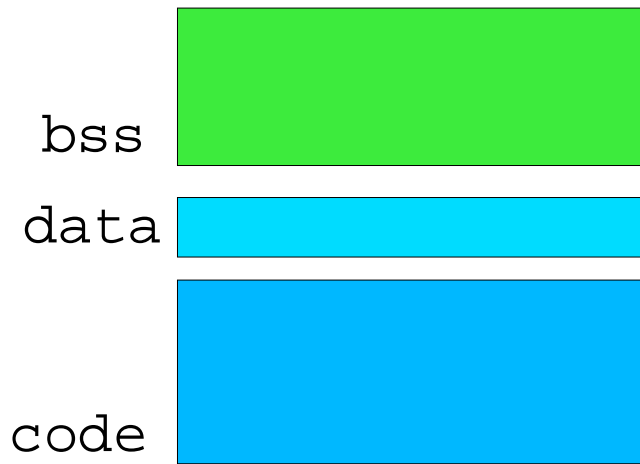
## “It's all about address spaces”

- Generally a complex issue
  - IPv4 ⇒ IPv6 is mainly about address space exhaustion

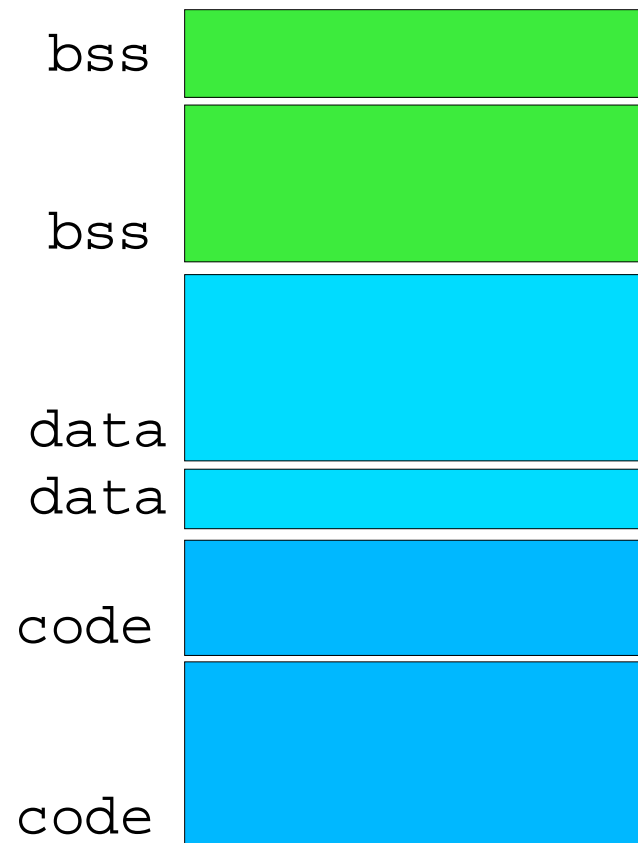
## 213 review (?)

- Combining .o's changes addresses
- But what about *two* programs?

# Every .o uses same address space





# Linker Combines .o's, Changes Addresses



# What About *Two* Programs?

stack  FFFFFFF000


stack  FFFFE000

bss  00010300

bss  00010300

data  00010200

data  00010100

code  00010000

code  00010000

# Logical vs. Physical Addresses

## Logical address

- Each program has its own *address space* ...
  - fetch: address  $\Rightarrow$  data
  - store: address, data  $\Rightarrow$  .
- ...as envisioned by programmer, compiler, linker

## Physical address

- Where your program ends up in memory
- They can't *all* be loaded at 0x10000!

# Reconciling Logical, Physical

## Programs could *take turns* in memory

- Requires swapping programs out to disk
- Very slow

## Could run programs at addresses other than linked

- Requires using linker to “relocate one last time” at launch
- Done by some old mainframe OSs
- Slow, complex, or both

## We are computer scientists!

# Reconciling Logical, Physical

## Programs could *take turns* in memory

- Requires swapping programs out to disk
- Very slow

## Could run programs at addresses other than linked

- Requires using linker to “relocate one last time” at launch
- Done by some old mainframe OSs
- Slow, complex, or both

## We are computer scientists!

- Insert a level of indirection

# Reconciling Logical, Physical

## Programs could *take turns* in memory

- Requires swapping programs out to disk
- Very slow

## Could run programs at addresses other than linked

- Requires using linker to “relocate one last time” at launch
- Done by some old mainframe OSs
- Slow, complex, or both

## We are computer scientists!

- Insert a level of indirection
- Well, get the ECE folks to do it for us

# Type Theory

## Physical memory behavior

- fetch: address  $\Rightarrow$  data
- store: address, data  $\Rightarrow$  .

## Process thinks of memory as...

- fetch: address  $\Rightarrow$  data
- store: address, data  $\Rightarrow$  .

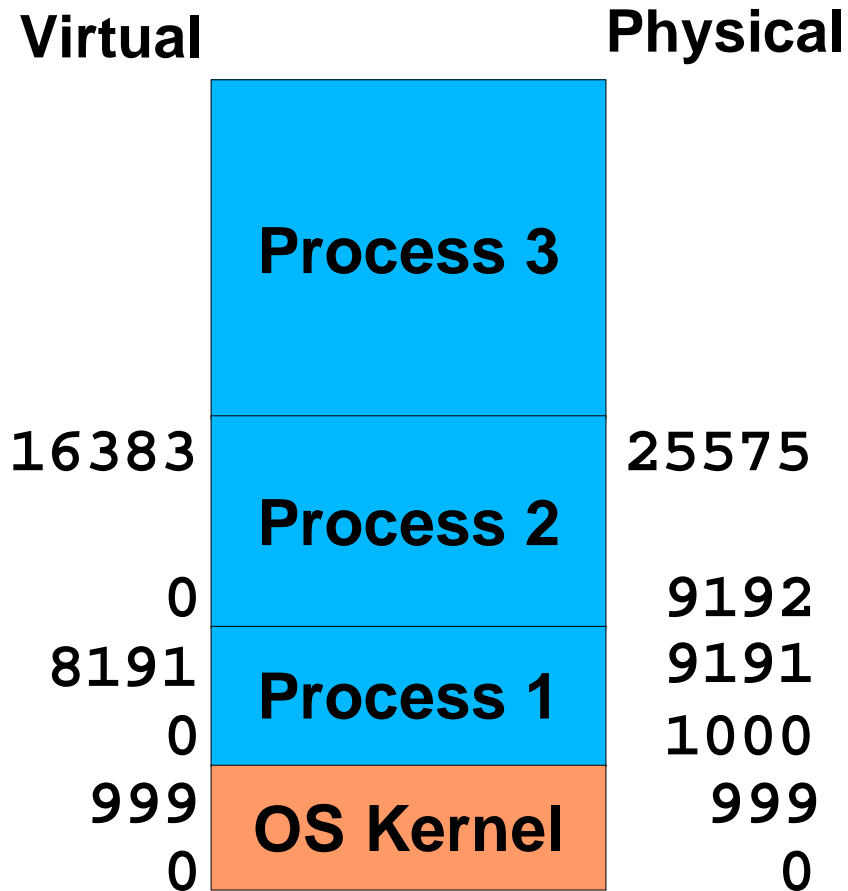
## Goal: each process has “its own memory”

- process-id  $\Rightarrow$  fetch: (address  $\Rightarrow$  data)
- process-id  $\Rightarrow$  store: (address, data  $\Rightarrow$  .)

## What *really* happens

- process-id  $\Rightarrow$  map: (virtual-address  $\Rightarrow$  physical-address)
- Machine does “map o fetch” and “map o store”

# Simple Mapping Functions



**P1**

If  $V > 8191$  *ERROR*

Else  $P = 1000 + V$

**P2**

If  $V > 16383$  *ERROR*

Else  $P = 9192 + V$

**Address space  $\equiv$**

- Base address
- Limit

# Contiguous Memory Mapping

## Processor contains two *control registers*

- Memory base
- Memory limit

## Each memory access checks

```
If V < limit
```

```
    P = base + V;
```

```
Else
```

```
    ERROR /* what do we call this error? */
```

## During context switch...

- Save/load user-visible registers
- Also load process's base, limit registers

# Problems with Contiguous Allocation

## How do we *grow* a process?

- Must increase “limit” value
- Cannot expand into another process's memory!
- Must move entire address spaces around
  - Very expensive

## Fragmentation

- New processes may not fit into unused memory “holes”

## Partial memory residence

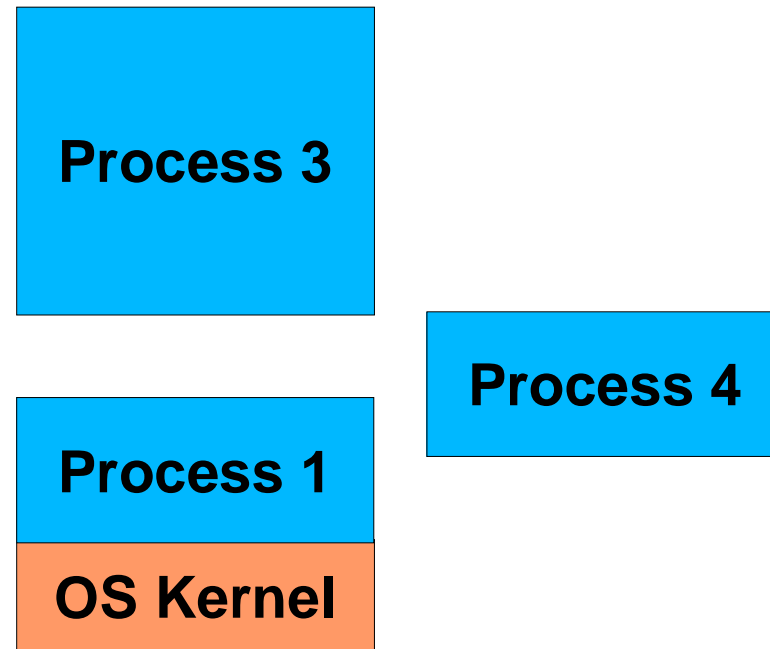
- Must *entire* program be in memory at same time?

# Can We Run Process 4?

**Process exit creates  
“holes”**

**New processes may be  
too large**

**May require moving entire  
address spaces**



# Term: “External Fragmentation”

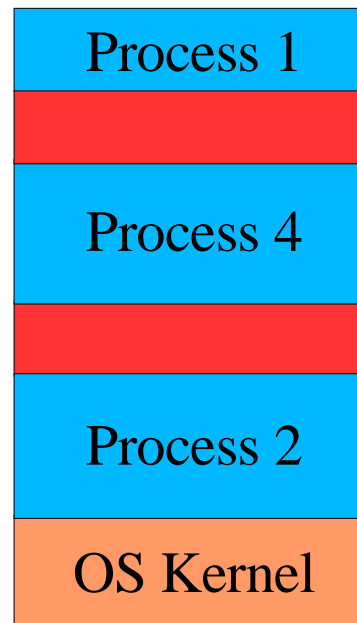
**Free memory is small  
chunks**

**Doesn't fit large objects**

**Can “disable” lots of  
memory**

**Can fix**

- **Costly “compaction”**
  - **aka “Stop & copy”**



# Term: “Internal Fragmentation”

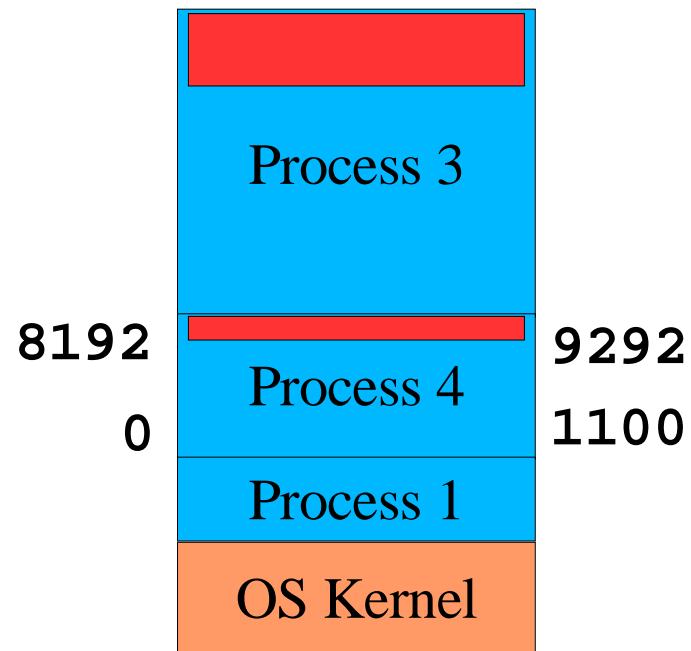
## Allocators often round up

- 8K boundary (*some* power of 2!)

Some memory is wasted *inside* each segment

Can't fix via compaction

Effects often non-fatal



# Swapping

## Multiple user processes

- Sum of memory demands  $>$  system memory
- Goal: Allow *each process* 100% of system memory

## Take turns

- Temporarily evict process(es) to disk
  - Not runnable
  - Blocked on *implicit* I/O request (e.g., “swapread”)
- “Swap daemon” shuffles process in & out
- Can take *seconds* per process
  - Modern analogue: laptop suspend-to-disk
- Maybe we need a better plan?

# Contiguous Allocation $\Rightarrow$ Paging

## Solves multiple problems

- Process growth problem
- Fragmentation compaction problem
- Long delay to swap a whole process

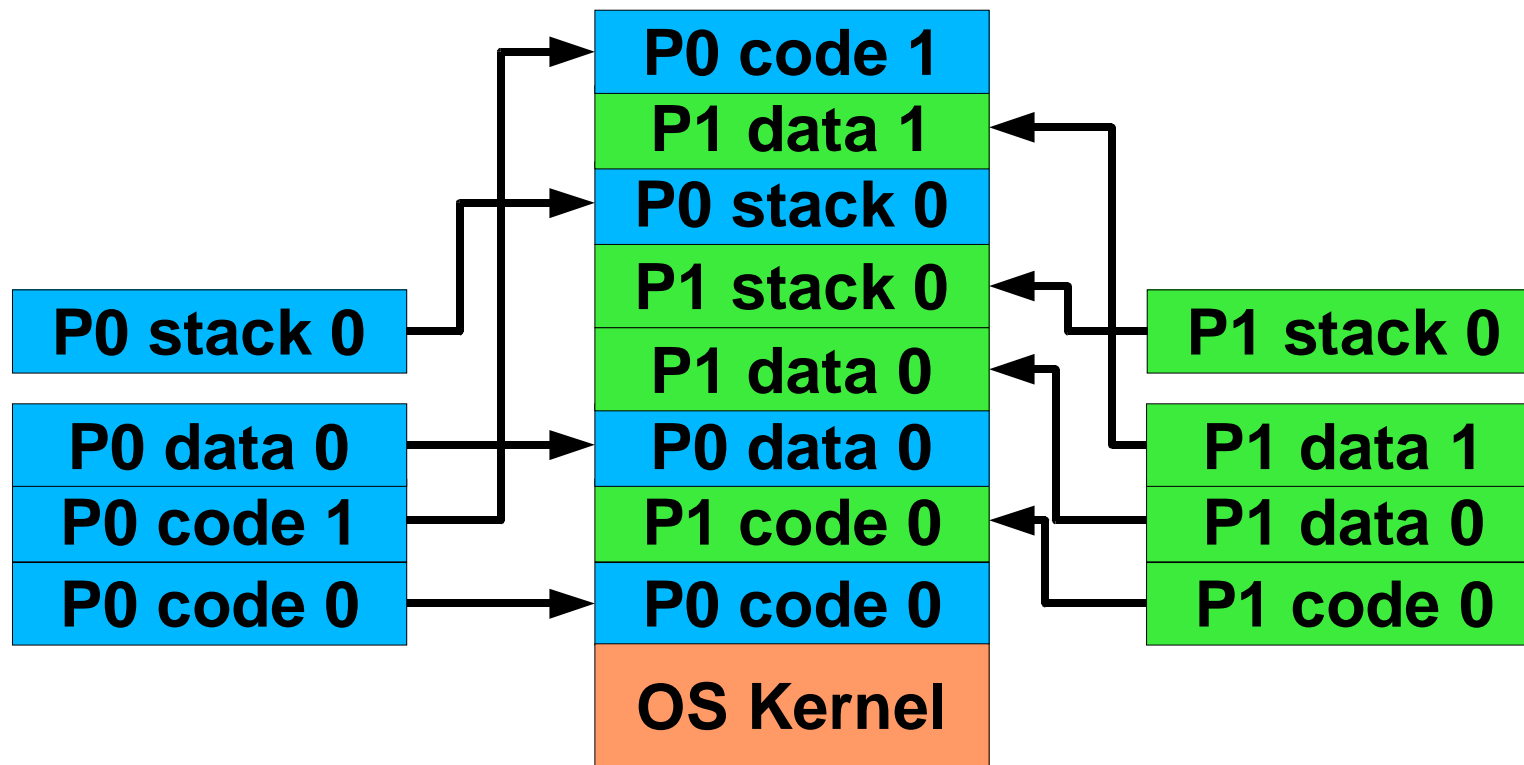
## Approach: divide memory more finely

- *Page* = small region of *virtual* memory (1/2K, 4K, 8K, ...)
- *Frame* = small region of *physical* memory
- [I will get this wrong, feel free to correct me]

## Key idea!!!

- Any page can map to (occupy) any frame

# Per-process Page Mapping



# Problems Solved by Paging

## Process growth problem?

- Any process can use any free frame for any purpose

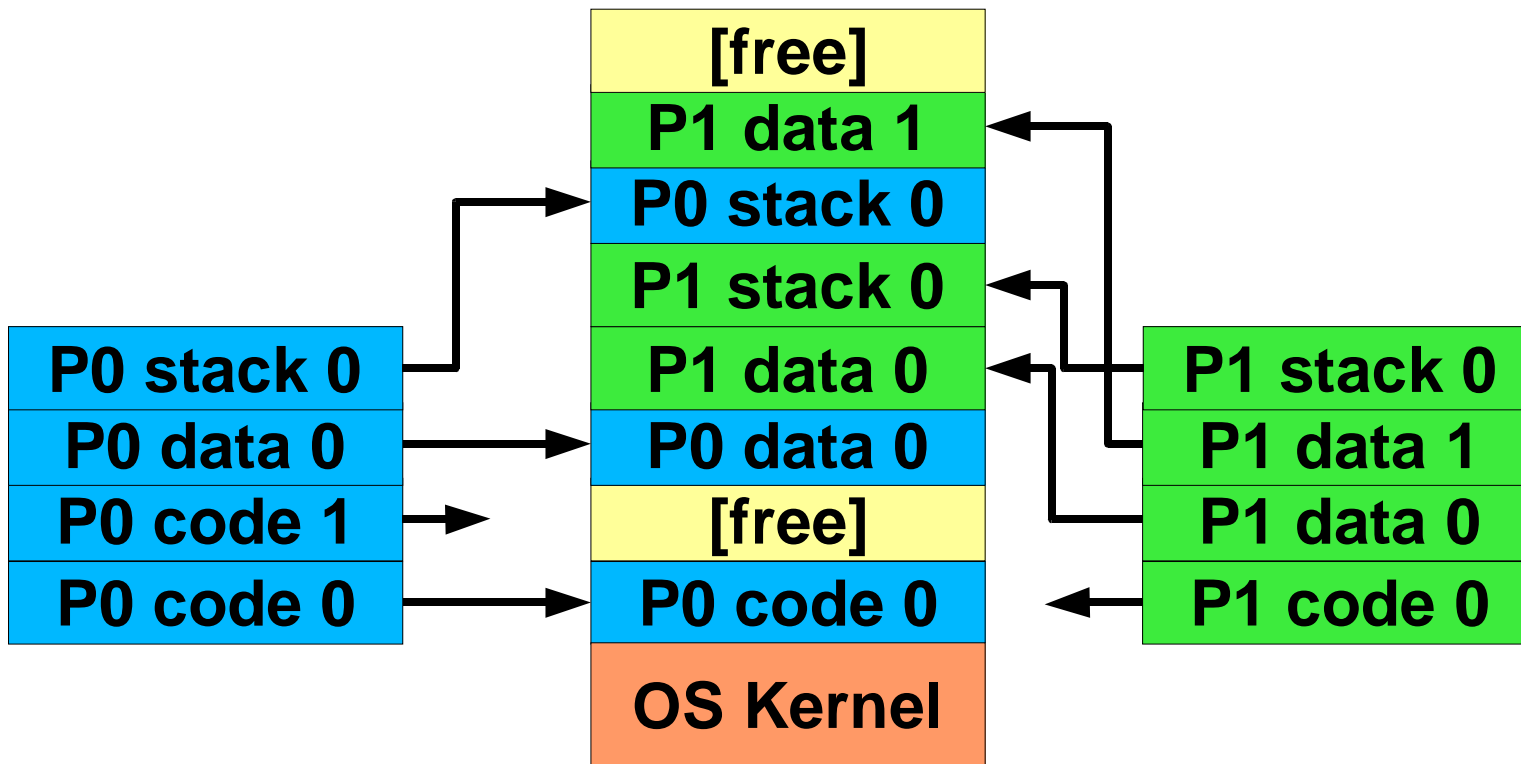
## Fragmentation compaction problem?

- Process doesn't need to be contiguous, so don't compact

## Long delay to swap a whole process?

- Swap *part* of the process instead!

# Partial Residence



# Must Evolve Data Structure Too

## Contiguous allocation

- Each process was described by (base,limit)

## Paging

- Each *page* described by (base,limit)?
  - Pages typically one size for whole system
- Ok, each *page* described by (base address)
- Arbitrary page  $\Rightarrow$  frame mapping requires some work
  - Abstract data structure: “map”
  - Implemented as...

# Data Structure Evolution

## Contiguous allocation

- Each process was described by (base,limit)

## Paging

- Each *page* described by (base,limit)?
  - Pages typically one size for whole system
- Ok, each *page* described by (base address)
- Arbitrary page  $\Rightarrow$  frame mapping requires some work
  - Abstract data structure: “map”
  - Implemented as...
    - » Linked list?
    - » Array?
    - » Hash table?
    - » Skip list?
    - » Splay tree?????

# “Page Table” Options

## Linked list

- $O(n)$ , so  $V \Rightarrow P$  time gets longer for large addresses!

## Array

- Constant time access
- Requires (large) contiguous memory for table

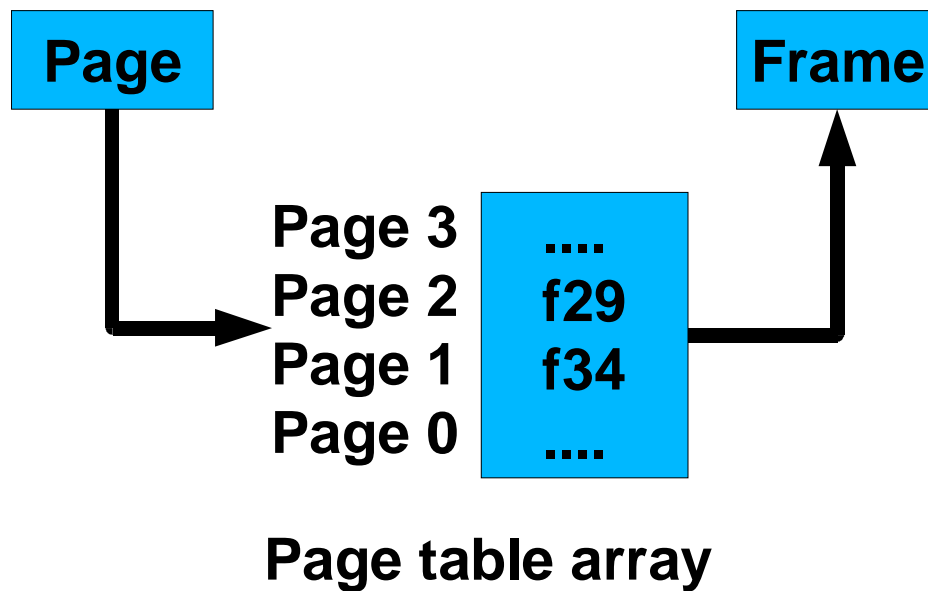
## Hash table

- Vaguely-constant-time access
- Not really bounded though

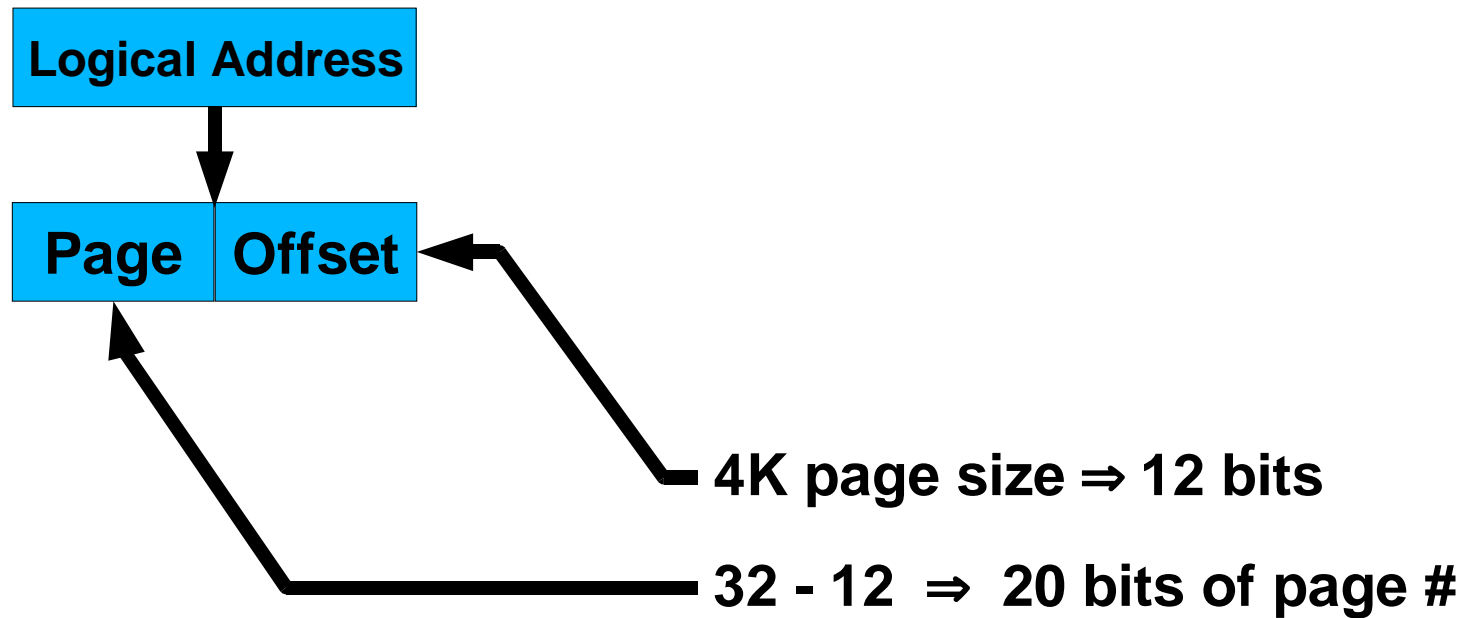
## Splay tree

- Excellent amortized expected time
- *Lots* of memory reads & writes possible for one mapping
- Not yet demonstrated in hardware

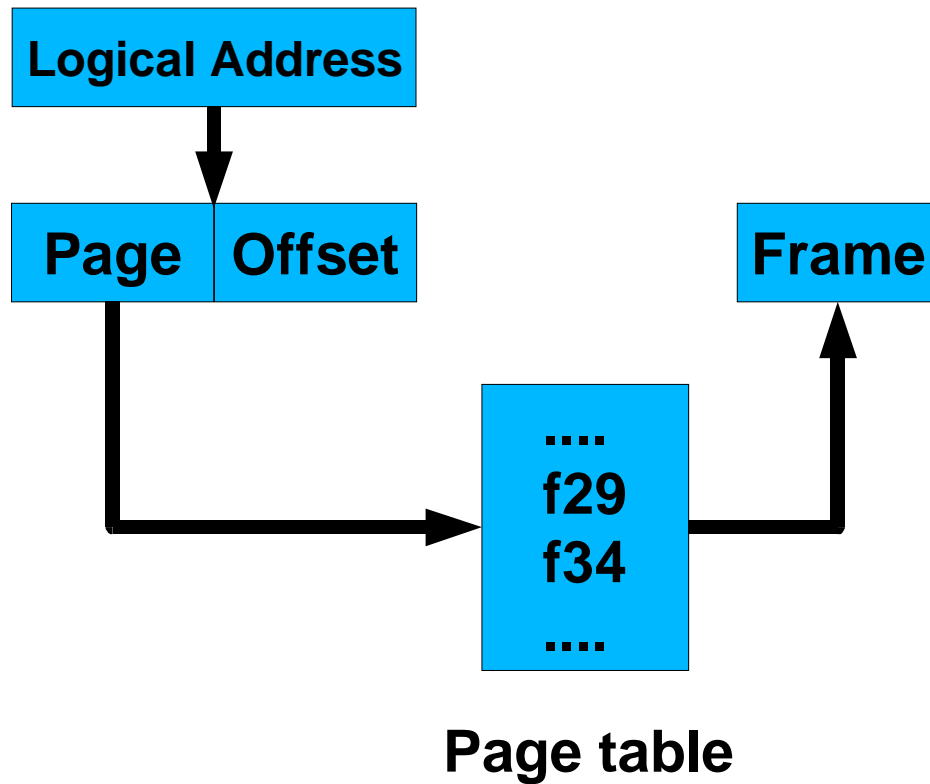
# “Page Table”: Array Approach



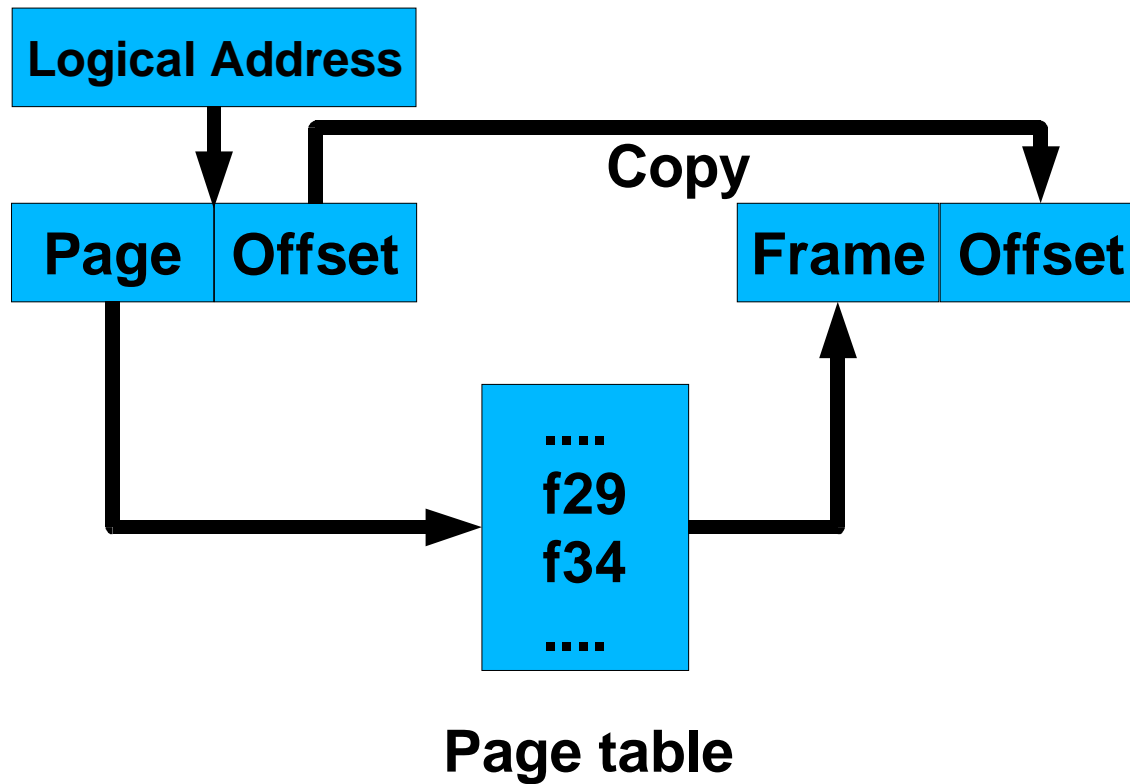
# Paging –Address Mapping



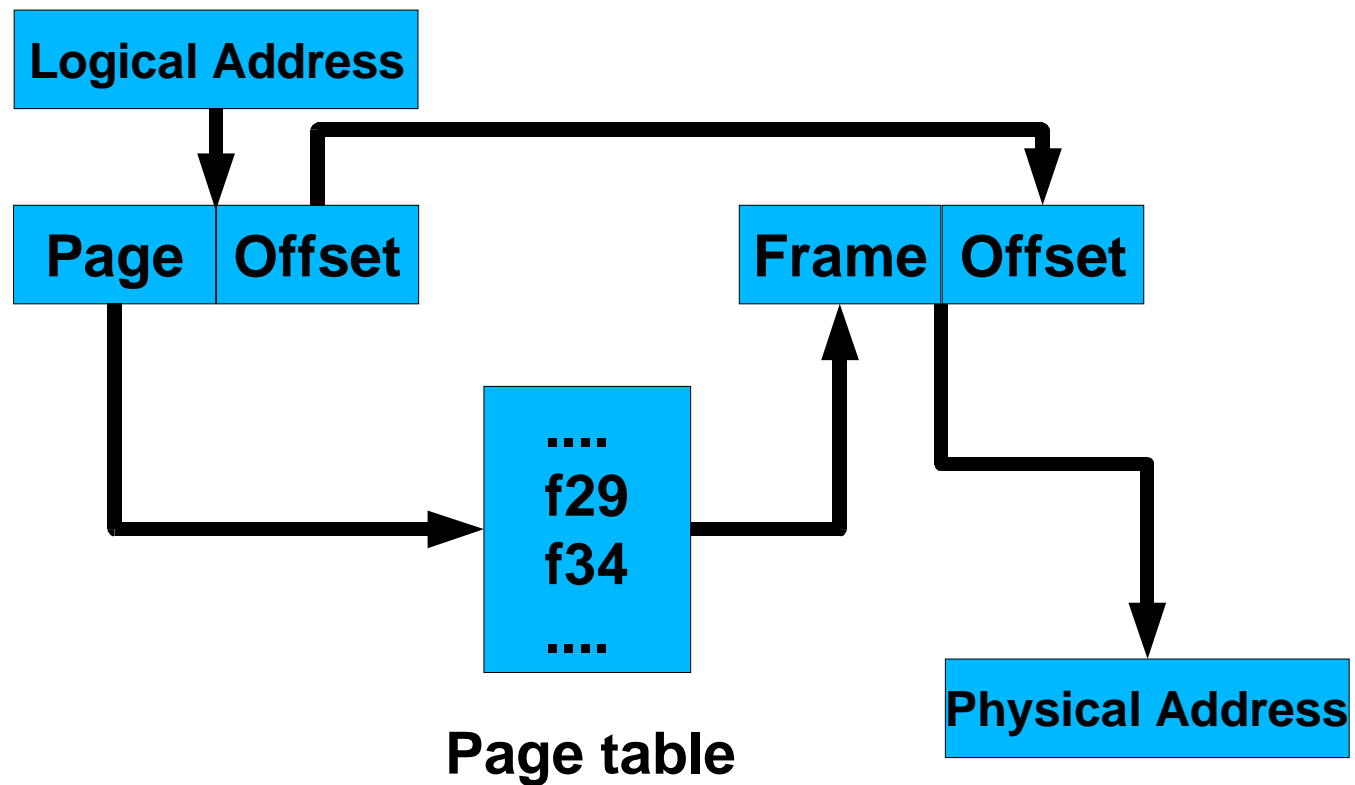
# Paging –Address Mapping



# Paging – Address Mapping



# Paging – Address Mapping



# Paging –Address Mapping

## User view

- Memory is a linear array

## OS view

- Each process requires N frames, located anywhere

## Fragmentation?

- **Zero** external fragmentation
- Internal fragmentation: average  $\frac{1}{2}$  page per region

# Bookkeeping

**One “page table” for each process**

**One global “frame table”**

- **Manages free frames**
- **(Typically) remembers who owns each frame**

**Context switch**

- **Must “activate” switched-to process's page table**

# Hardware Techniques

## Small number of pages?

- Page “table” can be a few registers
- PDP-11: 64k address space
  - 8 “pages” of 8k each –8 registers

## Typical case

- Large page tables, live in memory
  - Processor has “Page Table Base Register” (names vary)
  - Set during context switch

# Double trouble?

## Program requests memory access

- `MOVL (%ESI), %EAX`

## Processor makes *two* memory accesses!

- Splits address into page number, intra-page offset
- Adds page number to page table base register
- *Fetches page table entry (PTE) from memory*
- Concatenates frame address with intra-page offset
- *Fetches program's data from memory into %eax*

## Solution: “TLB”

- Not covered today

# Page Table Entry Mechanics

## PTE conceptual job

- Specify a frame number

# Page Table Entry Mechanics

## PTE conceptual job

- Specify a frame number

## PTE flags

- Valid bit
  - Not-set means access should generate an exception
- Protection
  - Read/Write/Execute bits
- Reference bit, “dirty” bit
  - Set if page was read/written “recently”
  - Used when paging to disk (later lecture)
- Specified by OS for each page/frame
  - Inspected/updated by hardware

# Page Table Structure

## Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
  - 4 GByte virtual address (32 bits) ⇒ \_\_\_\_\_ page table

# Page Table Structure

## Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
  - 4 GByte virtual address (32 bits)  $\Rightarrow$  4 MByte page table
  - *For each process!*

# Page Table Structure

## Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
  - 4 GByte virtual address (32 bits)  $\Rightarrow$  4 MByte page table
  - *For each process!*

## One Approach: Page Table Length Register (PTLR)

- (names vary)
- Many programs don't use entire virtual space
- Restrict a process to use entries 0...N of page table
- On-chip register detects out-of-bounds reference ( $>N$ )
- Allows small PTs for small processes
  - (as long as stack isn't far from data)

# Page Table Structure

## Key observation

- Each process page table is a *sparse mapping*
- Many pages are not backed by frames
  - Address space is sparsely used
    - » Enormous “hole” between bottom of stack, top of heap
    - » Often occupies 99% of address space!
  - Some pages are on disk instead of in memory

# Page Table Structure

## Key observation

- Each process page table is a *sparse mapping*
- Many pages are not backed by frames
  - Address space is sparsely used
    - » Enormous “hole” between bottom of stack, top of heap
    - » Often occupies 99% of address space!
  - Some pages are on disk instead of in memory

## Refining our observation

- Page tables are not randomly sparse
  - Occupied by *sequential memory regions*
  - Text, rodata, data+bss, stack
- “Sparse list of dense lists”

# Page Table Structure

**How to map “sparse list of dense lists”?**

**We are computer scientists!**

- ...?

# Page Table Structure

**How to map “sparse list of dense lists”?**

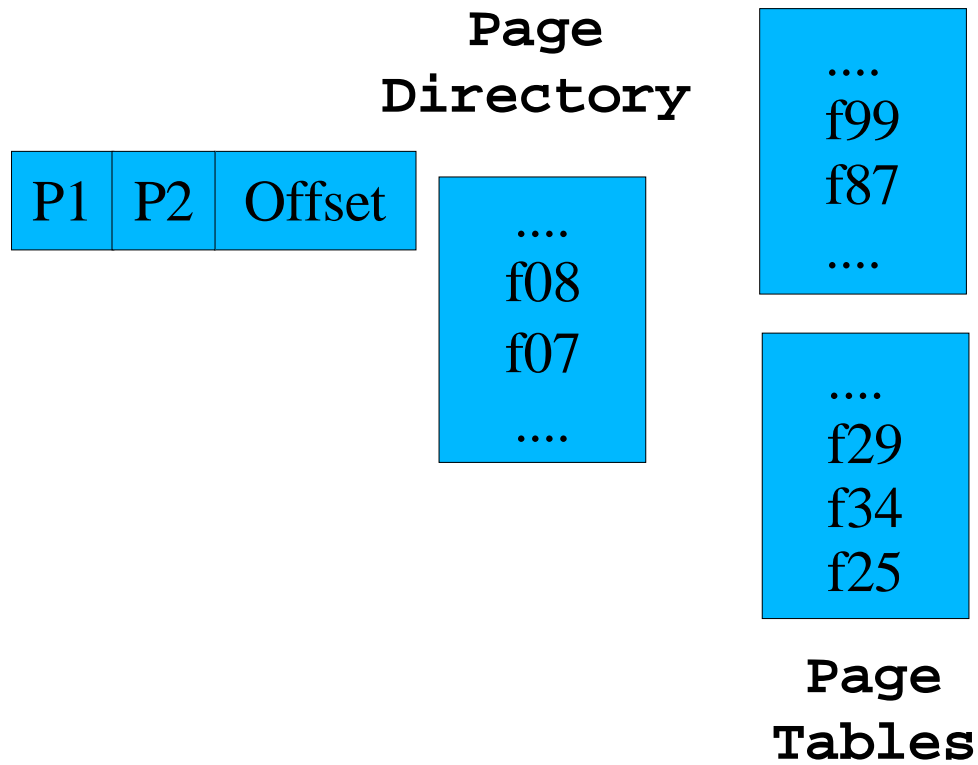
**We are computer scientists!**

- Insert a level of indirection
- Well, get the ECE folks to do it for us

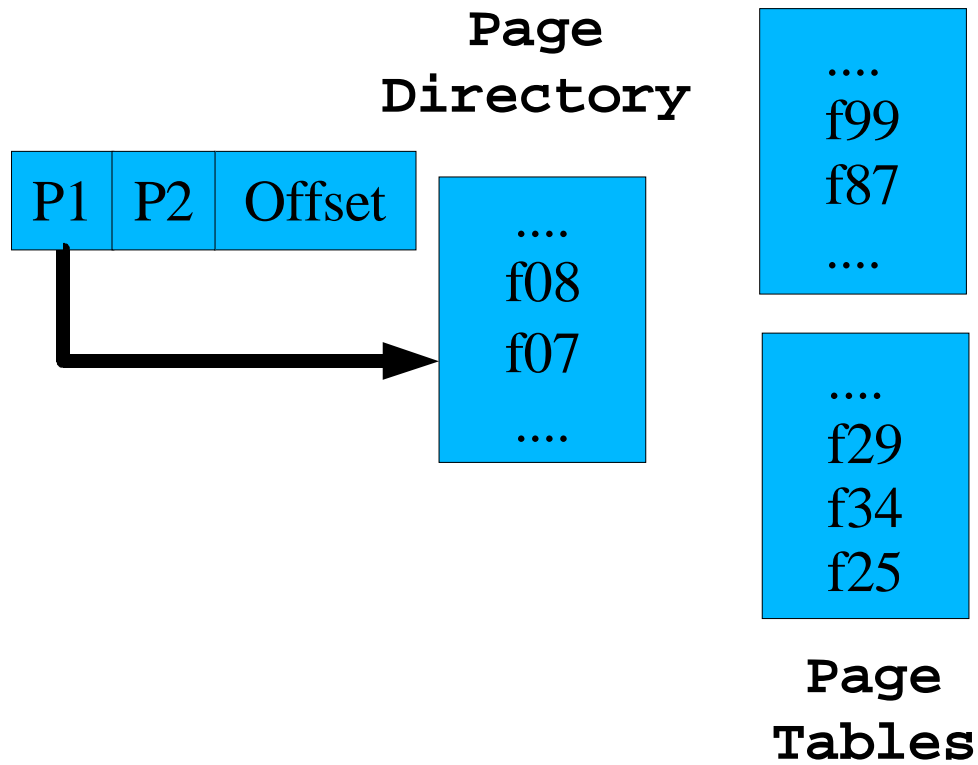
**Multi-level page table**

- “Page directory” maps large chunks of address space to...
- ...Page tables, which map pages to frames
- Conceptually the same mapping as last time
  - But the implementation is a two-level tree, not a single step

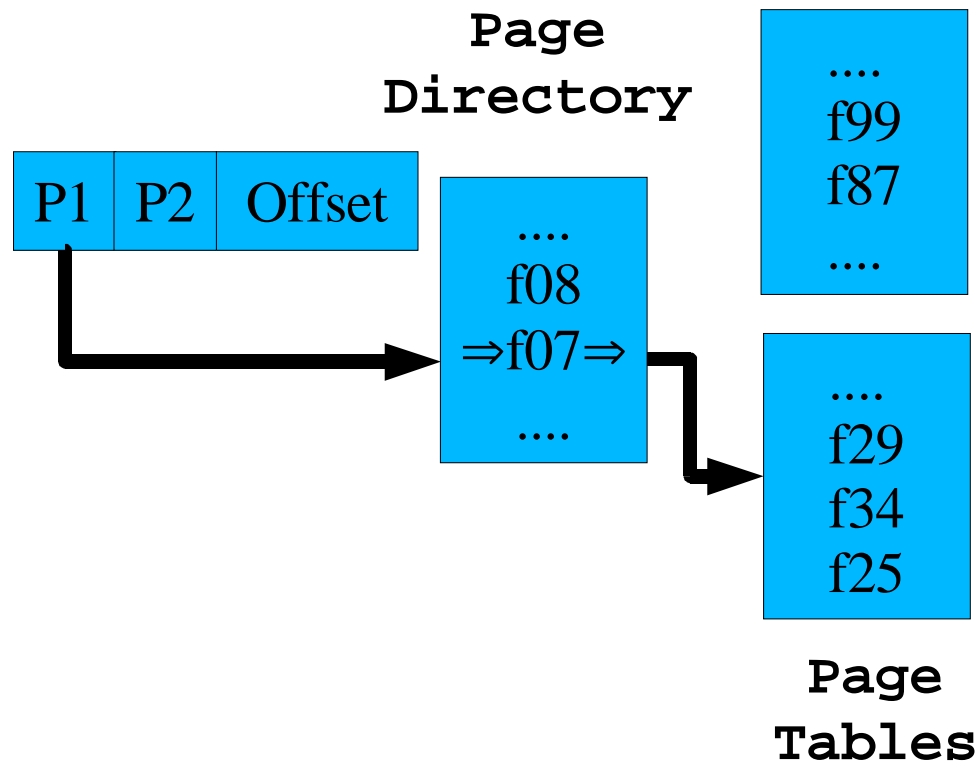
# Multi-level page table



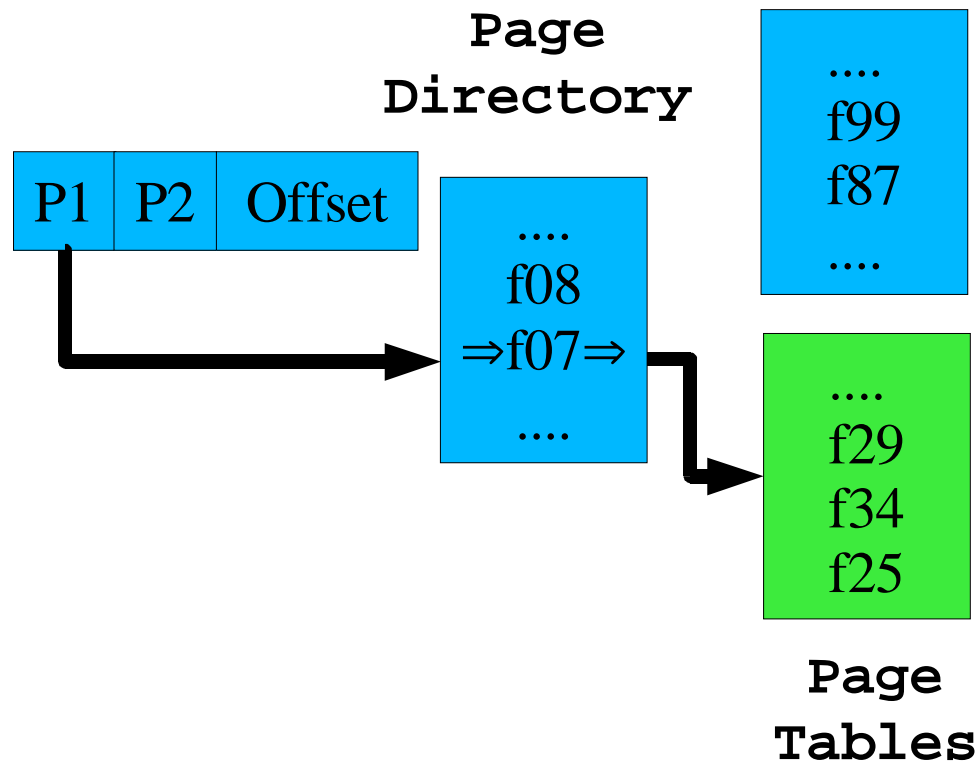
# Multi-level page table



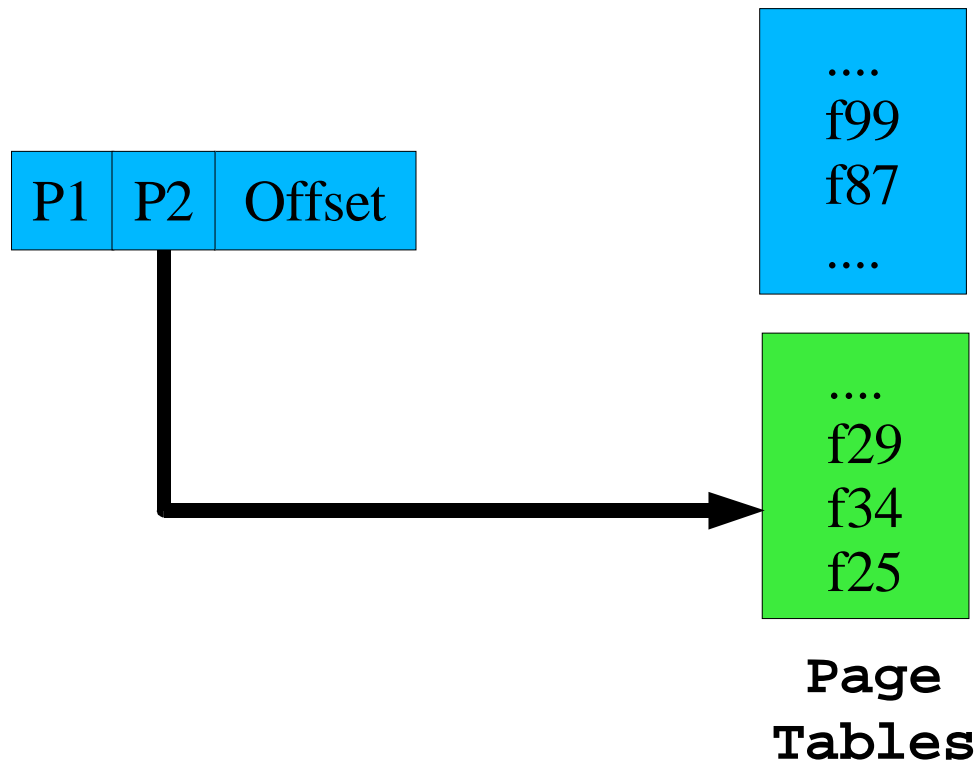
# Multi-level page table



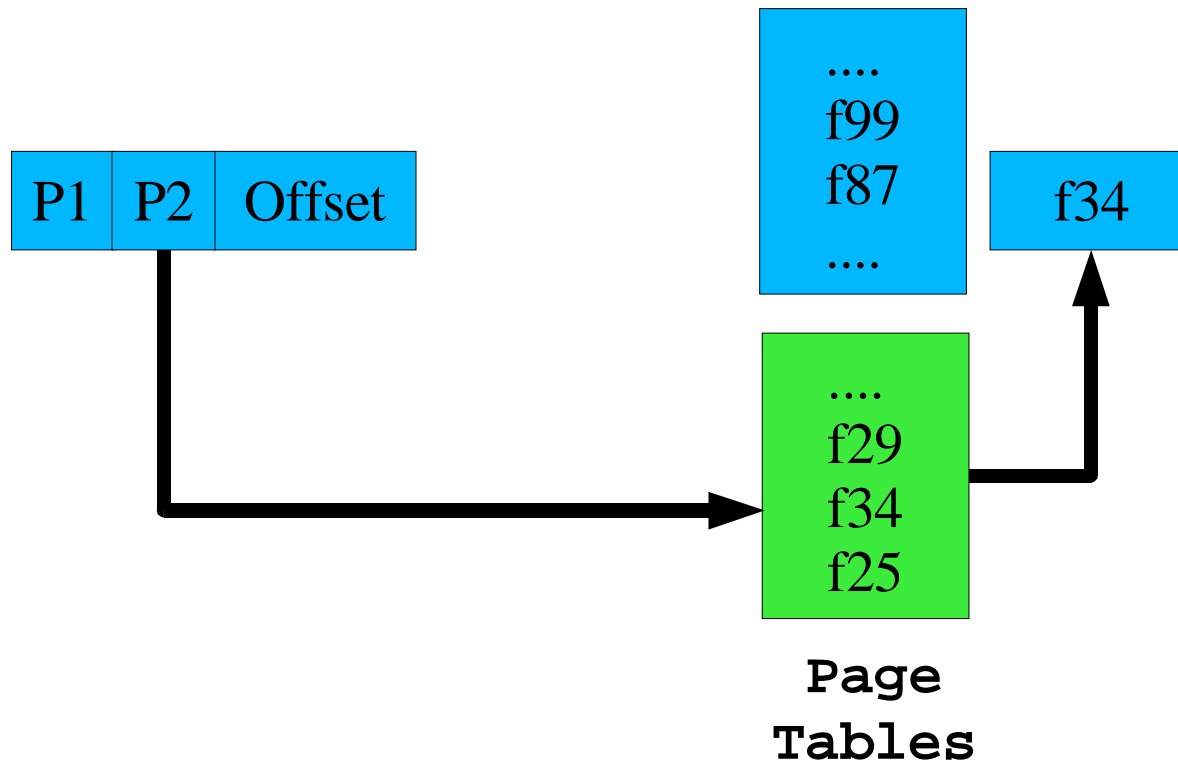
# Multi-level page table



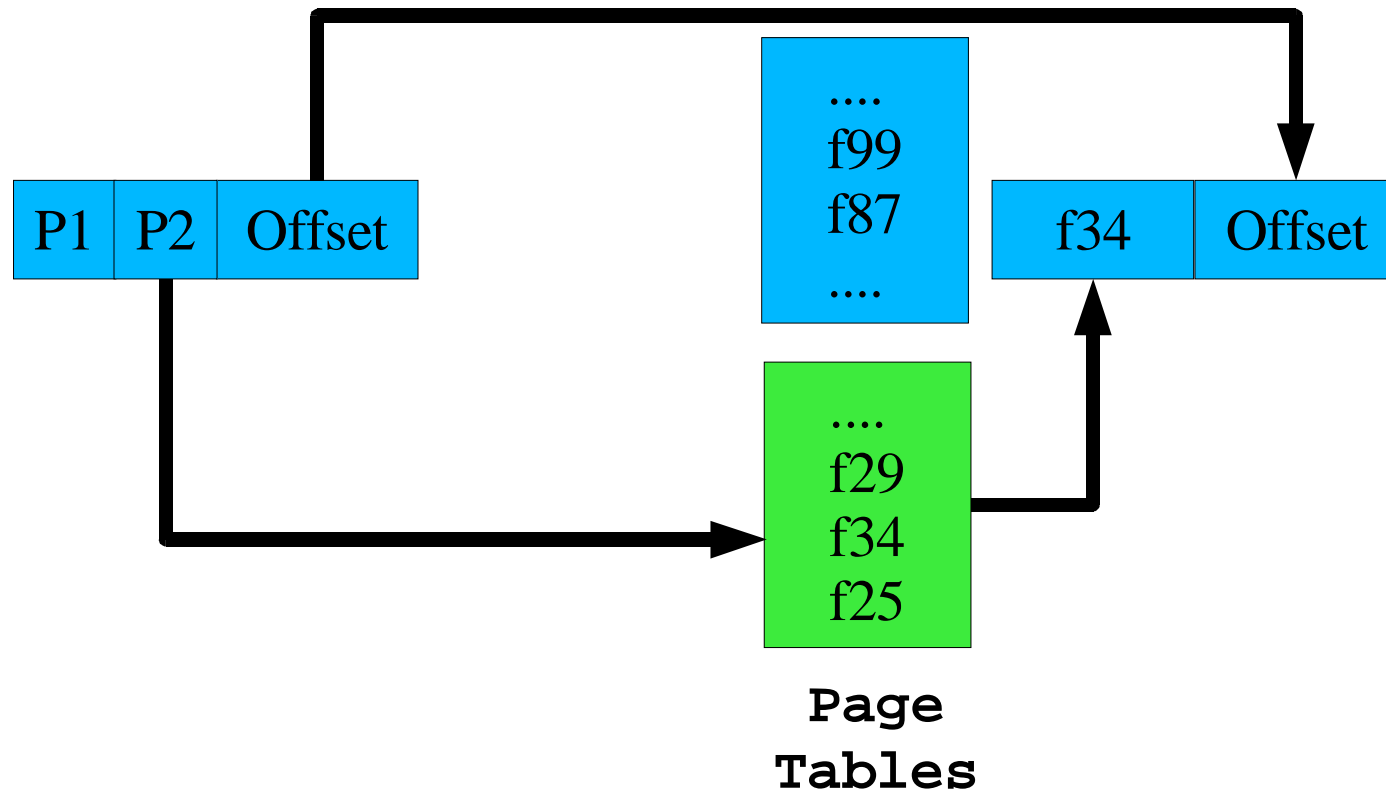
# Multi-level page table



# Multi-level page table



# Multi-level page table



# Sparse Mapping?

## Assume 4 KByte pages, 4-byte PTEs

- Ratio: 1024:1
  - 4 GByte virtual address (32 bits)  $\Rightarrow$  4 MByte page table

## Now assume page *directory* with 4-byte PDEs

- 4-megabyte page table becomes 1024 4K page tables
- Plus one 1024-entry page directory to point to them
- Result: \_\_\_\_\_

# Sparse Mapping?

## Assume 4 KByte pages, 4-byte PTEs

- Ratio: 1024:1
  - 4 GByte virtual address (32 bits)  $\Rightarrow$  4 MByte page table

## Now assume page *directory* with 4-byte PDEs

- 4-megabyte page table becomes 1024 4K page tables
- Plus one 1024-entry page directory to point to them
- Result: 4 Mbyte + 4Kbyte (this is better??)

# Sparse Mapping?

## Assume 4 KByte pages, 4-byte PTEs

- Ratio: 1024:1
  - 4 GByte virtual address (32 bits)  $\Rightarrow$  4 MByte page table

## Now assume page *directory* with 4-byte PDEs

- 4-megabyte page table becomes 1024 4K page tables
- Plus one 1024-entry page directory to point to them
- Result: 4 Mbyte + 4Kbyte (this is better??)

## *Sparse* address space...

- ...means most page tables contribute nothing to mapping...
- ...most page tables would contain only “no frame” entries...
- ...replace those PT's with “null pointer” in page directory.
- Result: *empty* 4GB address space specified by 4KB directory

# Sparse Address Space?

## Address space mostly “blank”

- Reads & writes should fail

## “Compress” out “the middle”

- Sparse address space should use a small mapping structure
- Fully-occupied address space can justify a larger mapping structure

```
stack
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
-no-
data
code
```

# Sparse Mapping!

## “Sparse” page directory

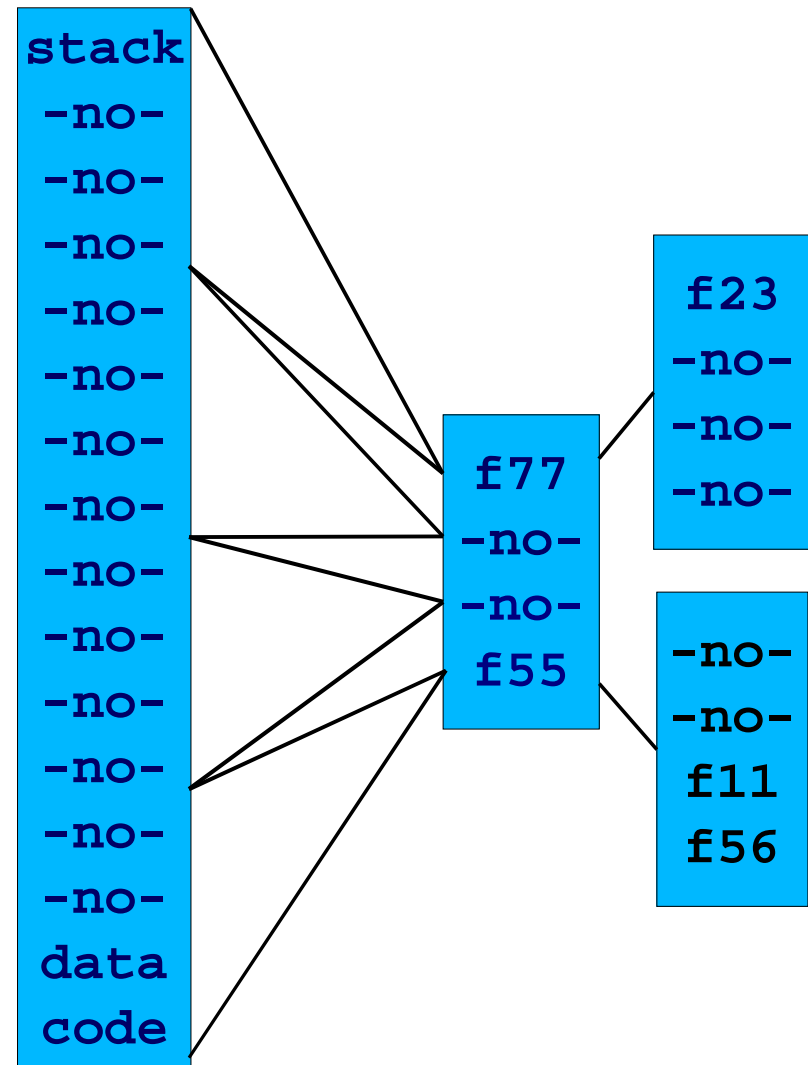
- Pointers to non-empty PT's
- “Null” instead of empty PT

## Common case

- Need 2 or 3 page tables
  - One or two map code & data
  - One maps stack
- Page directory has 1024 slots
  - 2-3 point to PT's
  - Remainder are “not present”

## Result

- 2-3 PT's, 1 PD
- Map entire address space with 12-16Kbyte, not 4Mbyte



# Segmentation

**Physical memory is (mostly) linear**

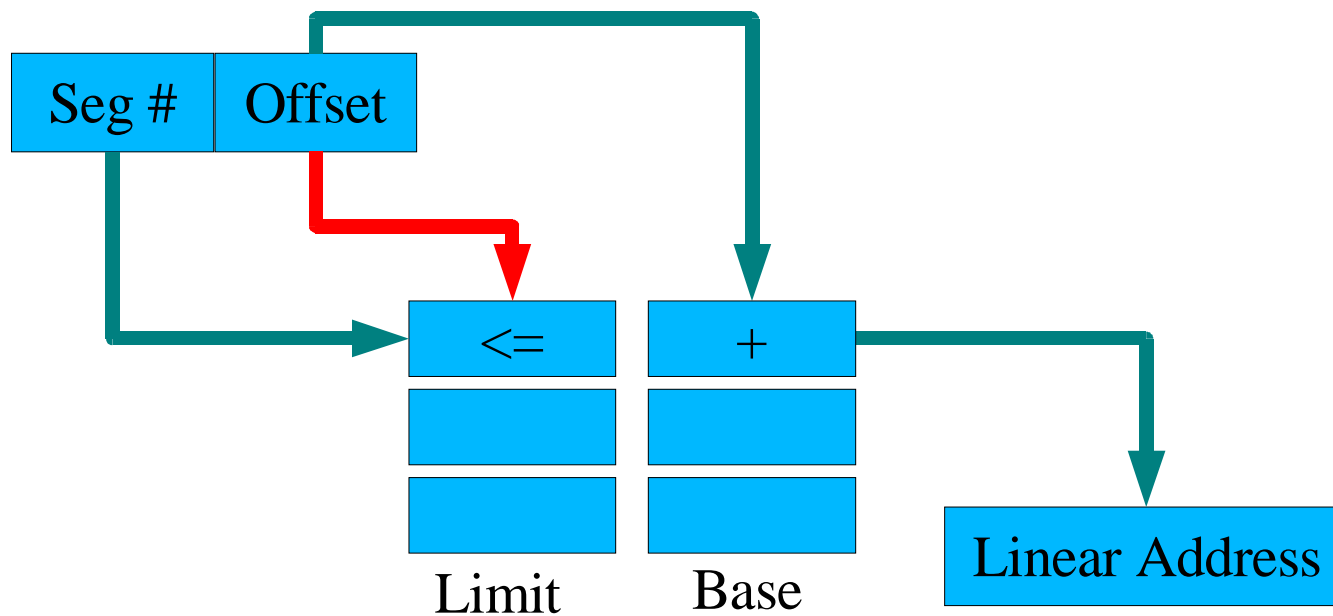
**Is virtual memory linear?**

- 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
  - 32-bit offset within a segment -- CS:EIP
- Descriptor table maps selector to segment descriptor
- Offset fed to segment descriptor, generates linear address
- Linear address fed through page directory, page table
- See textbook!

# x86 Type Theory

**Instruction  $\Rightarrow$  segment selector**

- [PUSHL implicitly specifies selector in %SS]

**Process  $\Rightarrow$  (selector  $\Rightarrow$  (base,limit) )**

- [Global,Local Descriptor Tables]

**Segment, within-segment address  $\Rightarrow$  “linear address”**

- CS:EIP means “EIP + base of code segment”

**Process  $\Rightarrow$  (linear address high  $\Rightarrow$  page table)**

- [Page Directory Base Register, page directory indexing]

**Page Table: linear address middle  $\Rightarrow$  frame address**

59 **Memory: frame address + offset  $\Rightarrow$  ...**

# 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 (trap handler)

## Data structures determined by access patterns

- Most address spaces are *sparsely allocated*

# Quote

**Any problem in Computer Science can be solved by an extra level of indirection.**

**-Roger Needham**