15-410
"...We are Computer Scientists!..."

Virtual Memory #1
Oct. 3, 2007

Dave Eckhardt

Roger Dannenberg
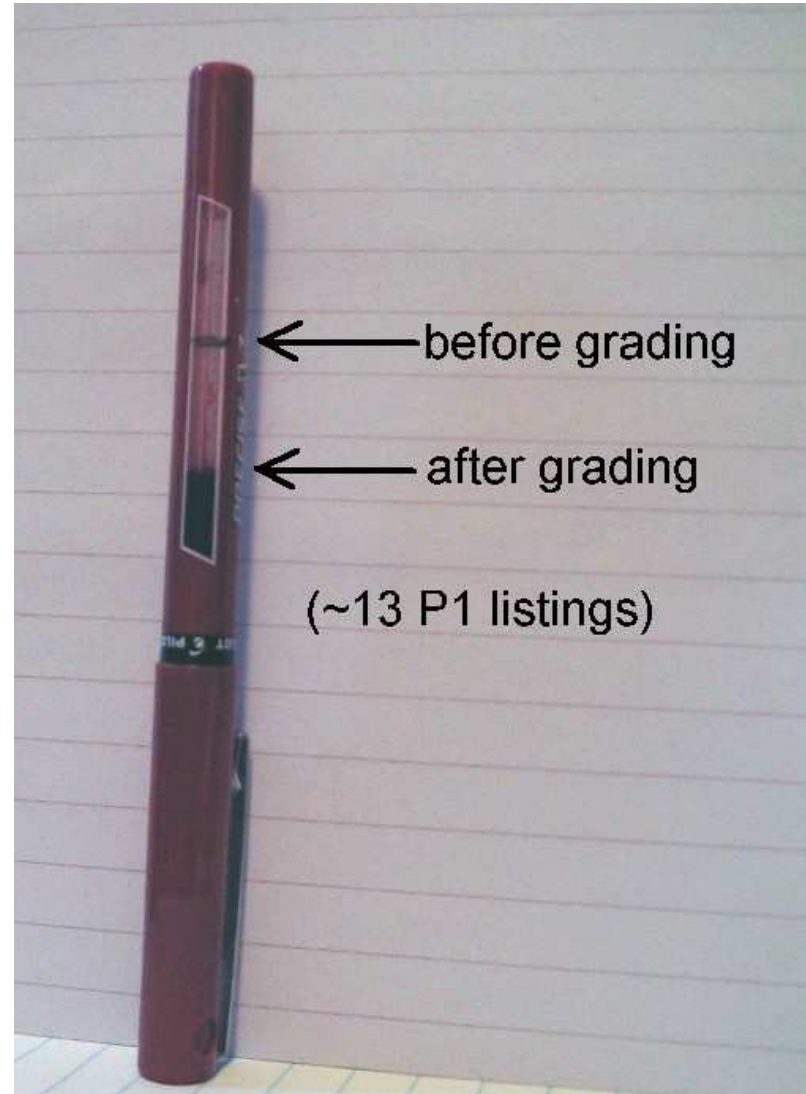
Synchronization

Milo's grading experiment

- How much ink does a grader ink when a grader inks P1?

Synchronization

Milo's grading experiment

- How much ink does a grader ink when a grader inks P1?

before grading

after grading

(~13 P1 listings)

Synchronization

Mid-term *definitely* Tuesday the 9th

- You've received mail asking you to declare conflicts
- Please do so right away. Thanks!

Homework 1

- Out soon
- Goal: study aid for mid-term exam
  - (We'll release solutions @ deadline, for exam study)

4

Outline

Text

- **Chapters 8, 9**

The Problem: logical vs. physical

Contiguous memory mapping

Fragmentation

Paging

- **Type theory**
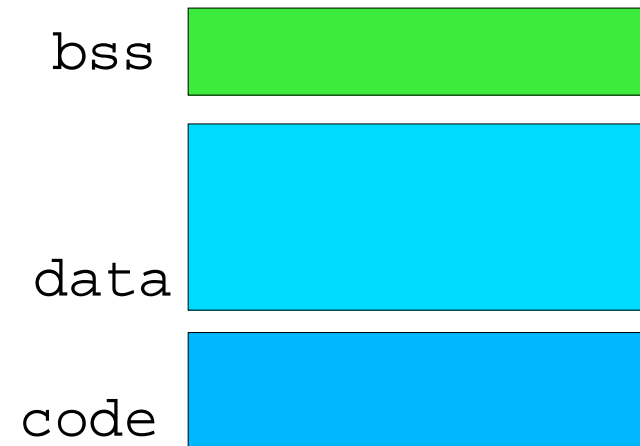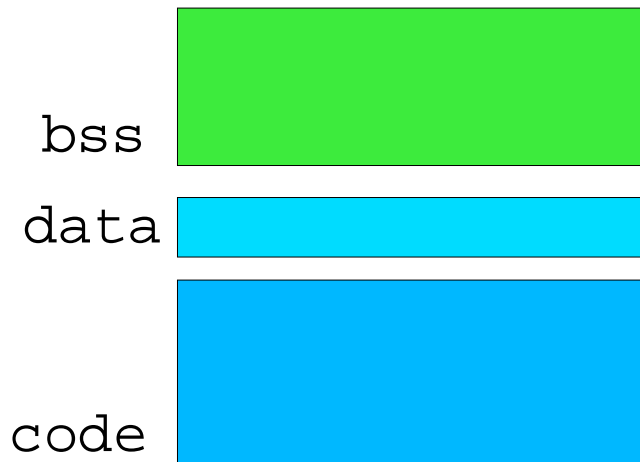- **A sparse map**

5

Logical vs. Physical

It's all about address spaces

- **Generally a complex issue**
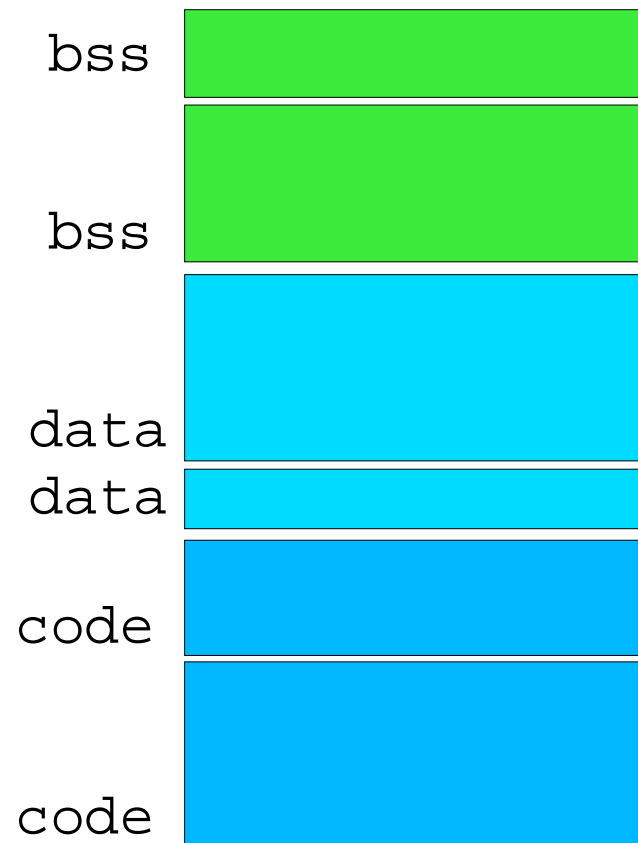   - **IPv4 $\Rightarrow$ IPv6 is mainly about address space exhaustion**

Review

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

6

Every .o uses same address space

bss

data

code

bss

data

code

Linker Combines .o's, Changes Addresses

bss

bss

data
data

code

code

What About *Two* Programs?

stack ▮ FFFFF000

stack ▮ FFFFE000

bss ▮ 00010300

bss ▮ 00010300

data ▮ 00010200

data ▮ 00010100

code ▮ 00010000

code ▮ 00010000

9

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!

10

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!

11

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

12

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

13

Type Theory

Physical memory behavior

- fetch: address ⇒ data

- store: address, data ⇒ .

Process thinks of memory as...

- fetch: address ⇒ data

- store: address, data ⇒ .

Goal: each process has "its own memory"

- process-id ⇒ fetch: (address ⇒ data)

- process-id ⇒ store: (address, data ⇒ .)

What *really* happens

- process-id ⇒ (virtual-address ⇒ physical-address)

14

Simple Mapping Functions

Virtual

Physical

Process 3

16383   25575

Process 2

0   9192

8191   9191

Process 1

0   1000

999   999

OS Kernel

0   0

P1

If V > 8191 *ERROR*

Else P = 1000 + V

P2

If V > 16383 *ERROR*

Else P = 9192 + V

Address space ≡

- Base address
- Limit

15

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**

16

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"
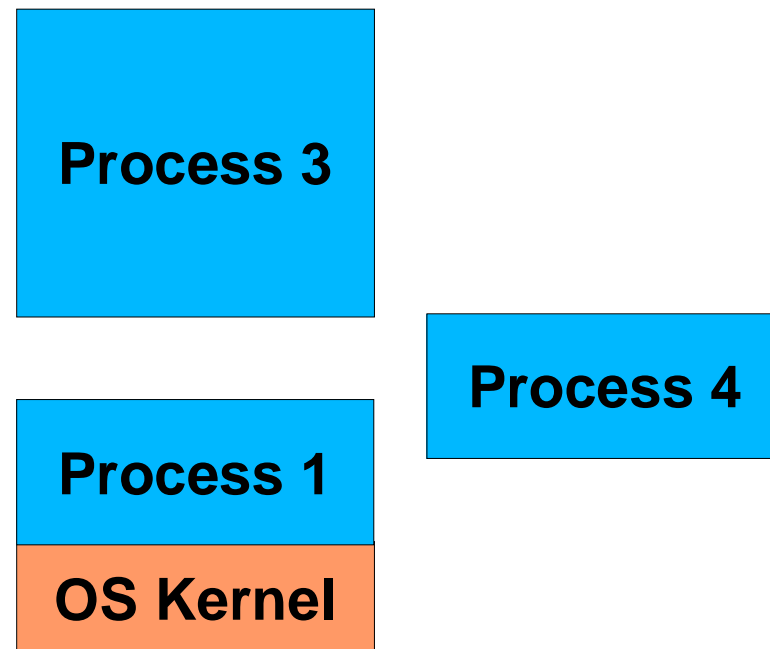
Partial memory residence

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

17

Can We Run Process 4?

Process exit creates "holes"

New processes may be too large

May require moving entire address spaces

Process 3

Process 4

Process 1

OS Kernel
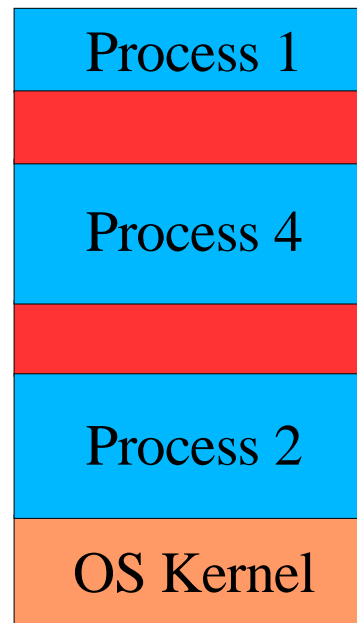
Term: "External Fragmentation"

Free memory is small chunks

Doesn't fit large objects

Can "disable" lots of memory

Can fix

- **Costly "compaction"**
  - **aka "Stop & copy"**

Process 1
Process 4
Process 2
OS Kernel

19

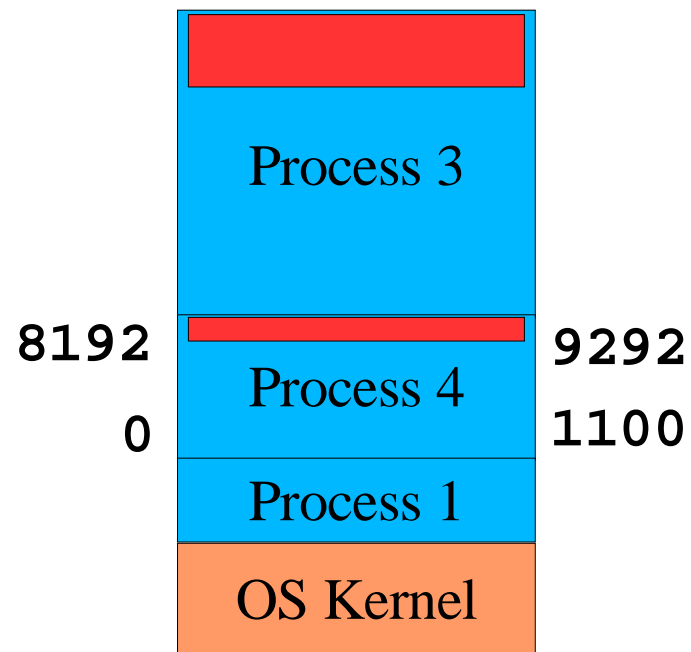# Term: "Internal Fragmentation"

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

8192

9292

0

1100

Process 3

Process 4

Process 1

OS Kernel

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

21

Contiguous Allocation ⇒ Paging

Solve multiple problems

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

Divide memory more finely

- *Page* = small region of *virtual* memory (½K, 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

22

Per-process Page Mapping

P0 code 1

P0 stack 0

P0 data 0

P0 code 1

P0 code 0

P0 code 1
P1 data 1
P0 stack 0
P1 stack 0
P1 data 0
P0 data 0
P1 code 0
P0 code 0
OS Kernel

P1 stack 0

P1 data 1
P1 data 0
P1 code 0
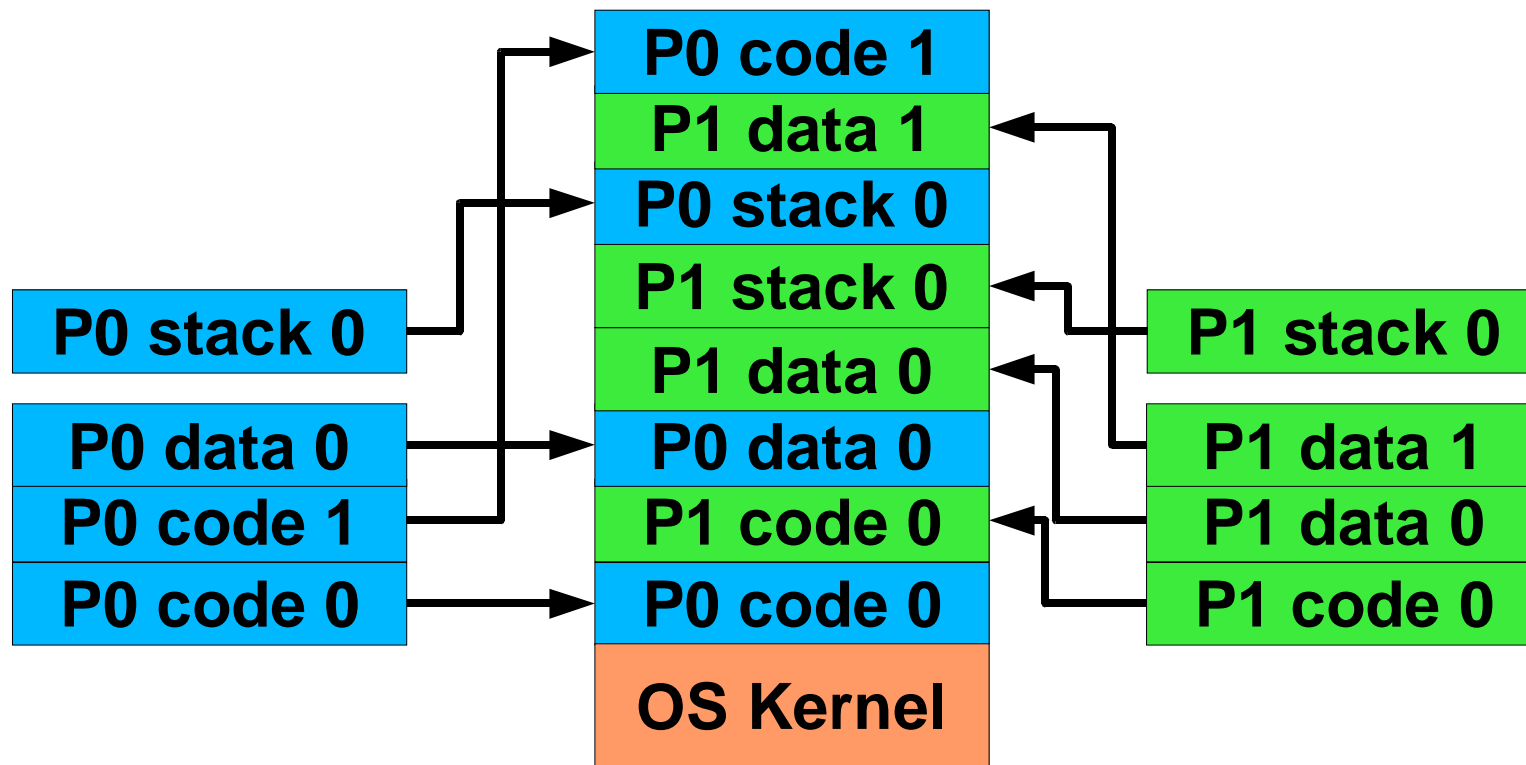
Problems Solved by Paging

Process growth problem?

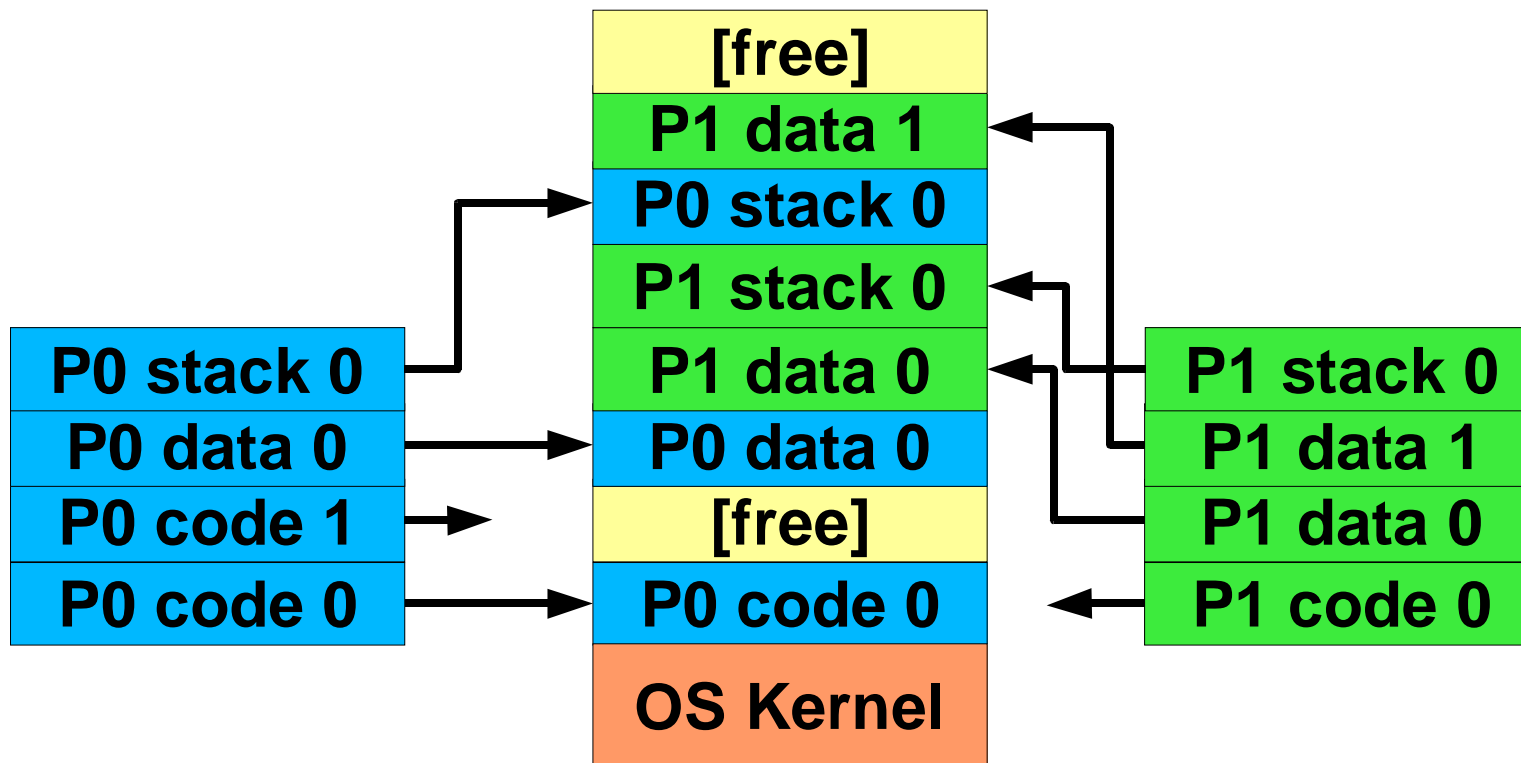- Any process can use any free frame for any purpose

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!

24

Partial Residence

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...**

26

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 ⇒ 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
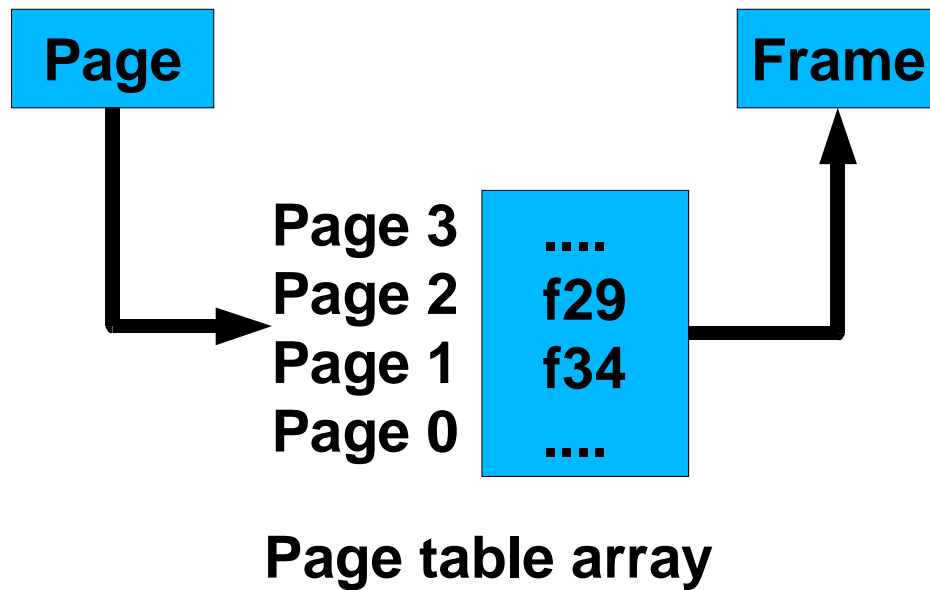
Hash table

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

Splay tree

- Excellent amortized expected time
- *Lots* of memory reads & writes possible for one mapping
- Probably impractical

28

Page Table Array

Page

Frame

Page 3   ....
Page 2   **f29**
Page 1   **f34**
Page 0   ....

Page table array

Paging –Address Mapping
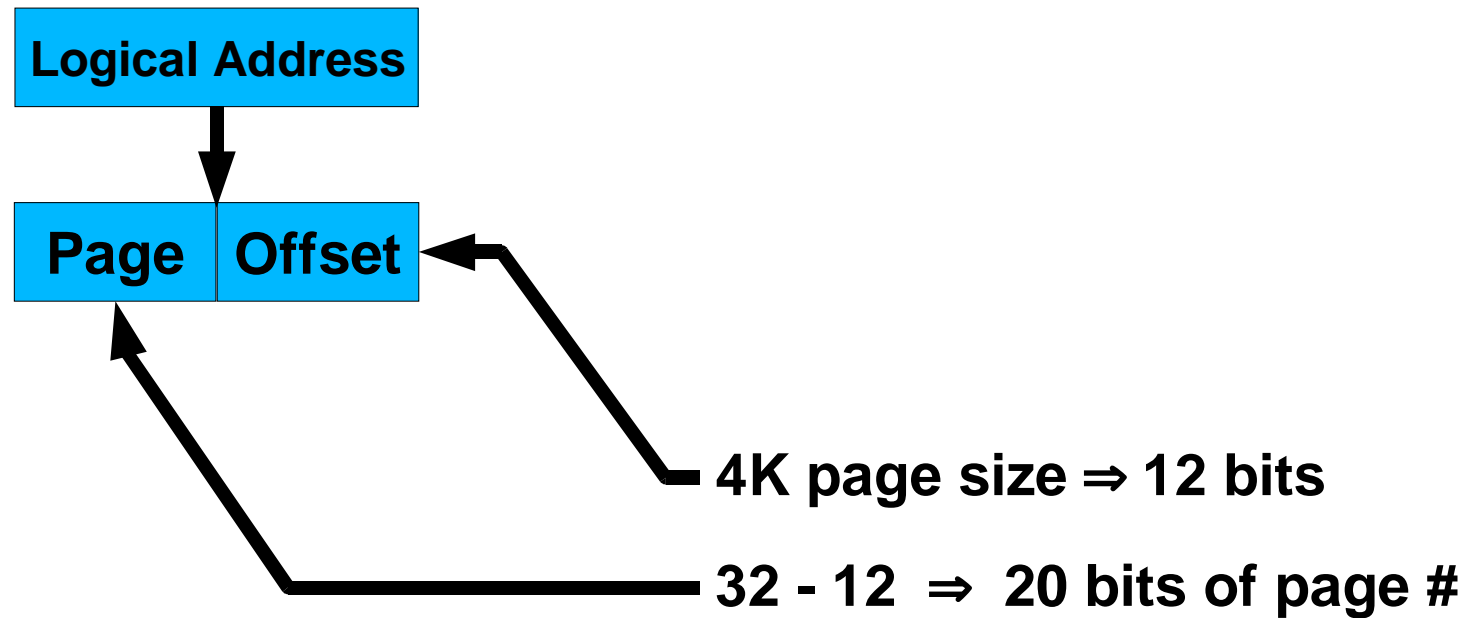
Logical Address

Page | **Offset**

4K page size ⇒ 12 bits

32 - 12 ⇒ 20 bits of page #

Paging –Address Mapping

Logical Address

Page | Offset

Frame

....
f29
f34
....

Page table

Paging –Address Mapping

Logical Address

Page | Offset

Copy

Frame | Offset

....
f29
f34
....

Page table

Paging –Address Mapping

Logical Address

Page	**Offset**

Frame	**Offset**

....
f29
f34
....

Page table

Physical Address
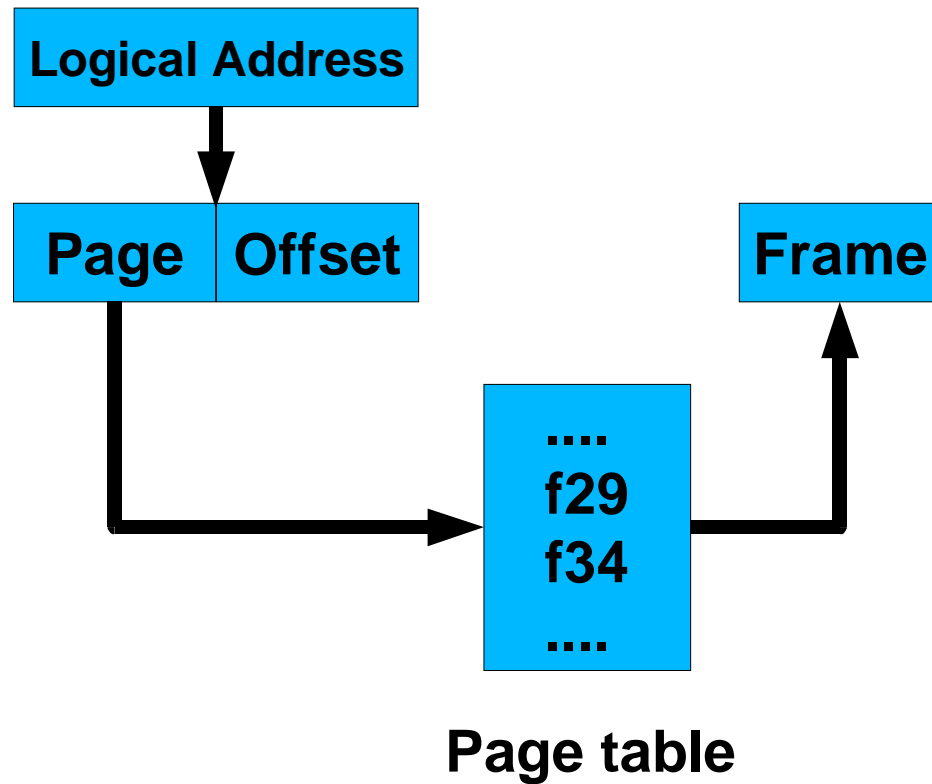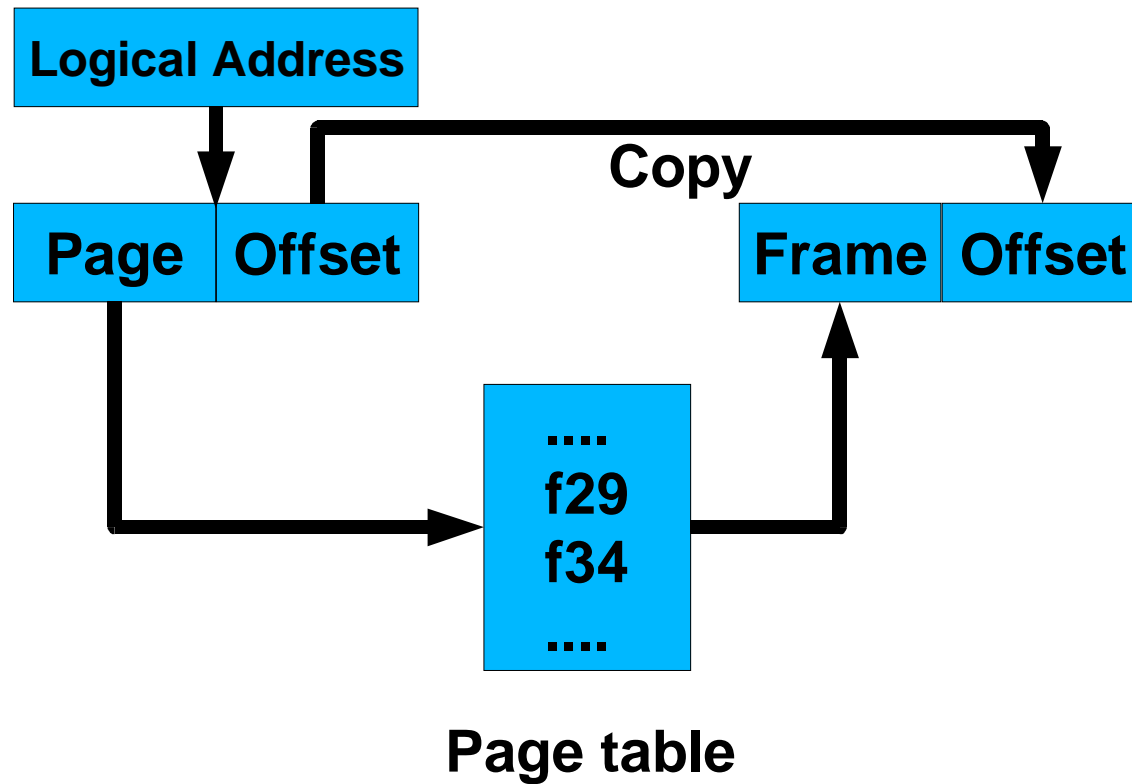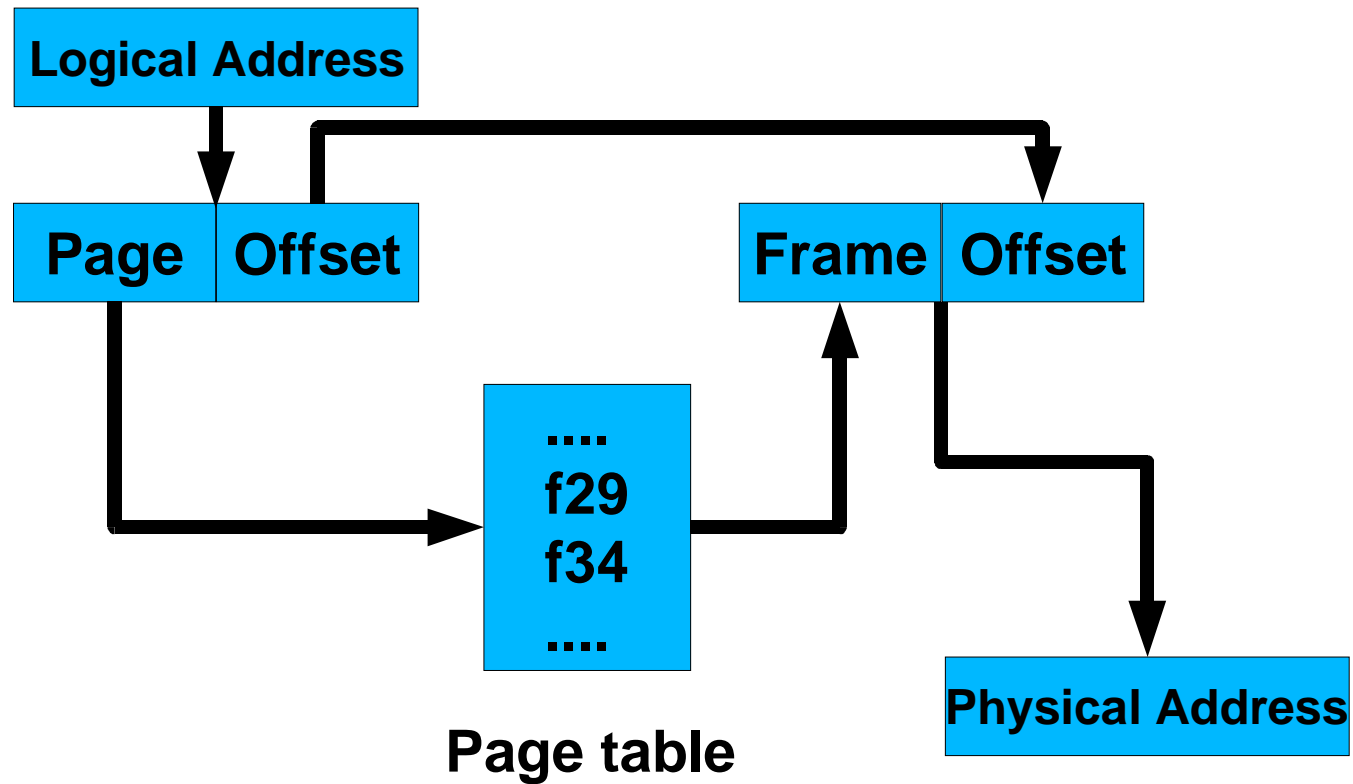
Paging – Address Mapping

User view

- **Memory is a linear array**

OS view

- **Each process requires N frames**

Fragmentation?

- *Zero* **external fragmentation**
- **Internal fragmentation: average ½ 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

35

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
  - Where?
    - » Processor has "Page Table Base Register" (names vary)
    - » Set during context switch

36

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*

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
- Dirty bit
  - Set means page was written to "recently"
  - Used when paging to disk (later lecture)
- Specified by OS for each page/frame

39

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

41

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**

42

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"**

43

Page Table Structure

How to map "sparse list of dense lists"?

We are computer scientists!

- ...?

Page Table Structure
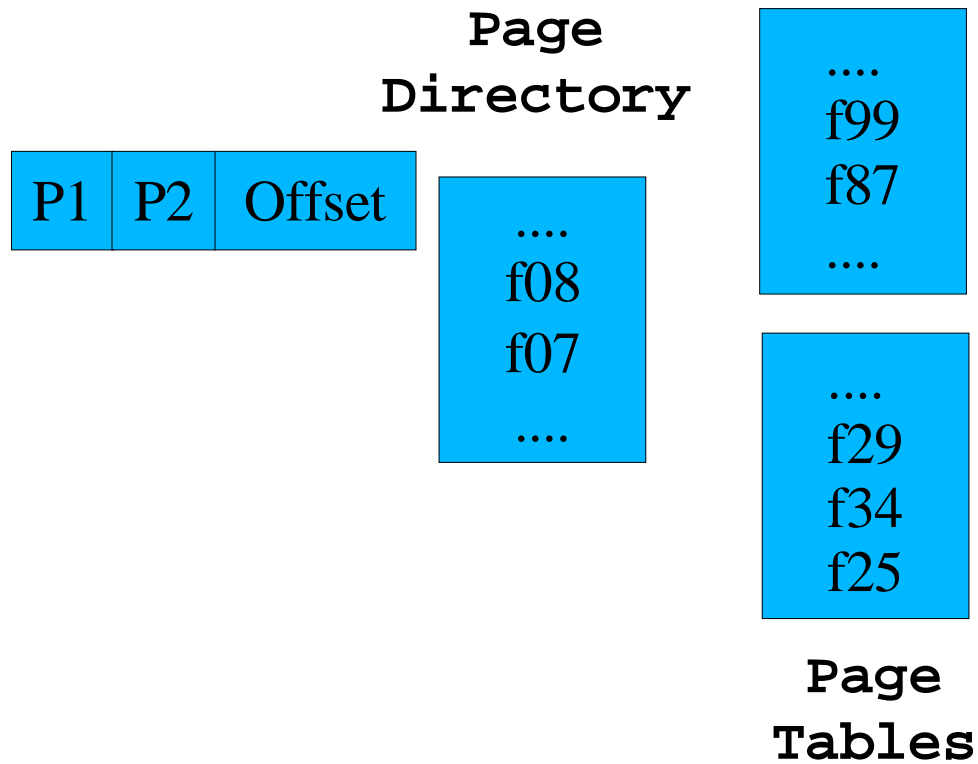
How to map "sparse list of dense lists"?

We are computer scientists!

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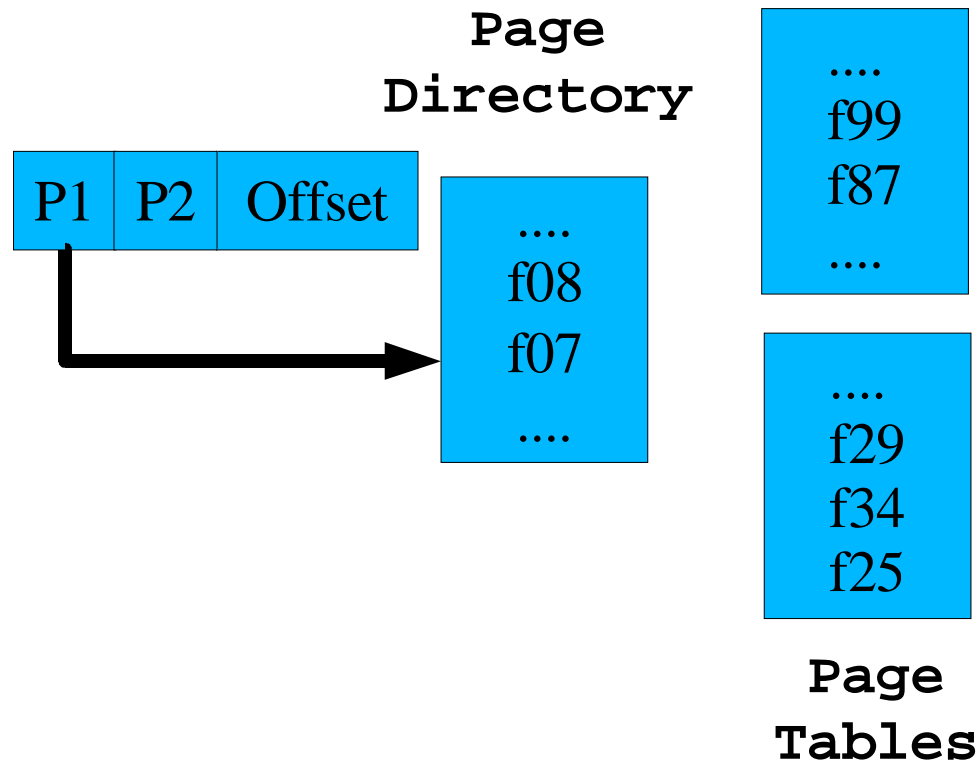- Well, get the ECE folks to do it for us

Multi-level page table

- Page directory maps large chunks of address space to...
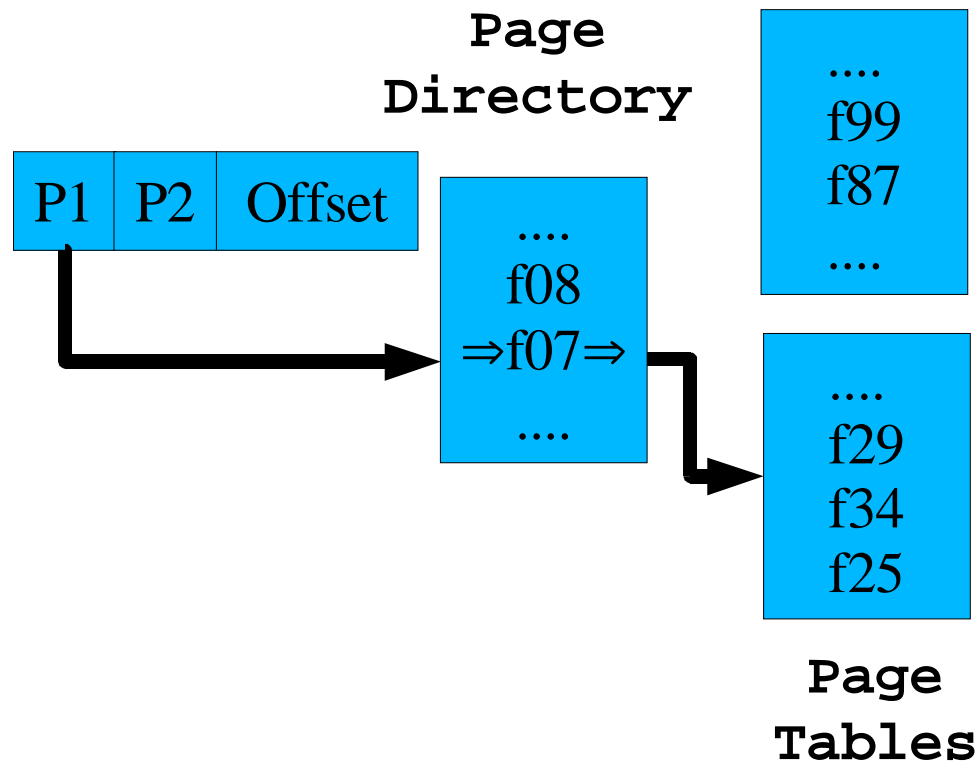- ...Page tables, which map pages to frames

Multi-level page table

P1	P2	Offset

Page Directory

....
f08
f07
....

Page Tables

....
f99
f87
....

....
f29
f34
f25

Multi-level page table

Page Directory

P1	P2	Offset

....
f08
f07
....

Page Tables

....
f99
f87
....

....
f29
f34
f25

Multi-level page table

Page Directory

P1 | P2 | Offset

....
f08
⇒f07⇒
....

....
f99
f87
....

....
f29
f34
f25

Page Tables

Multi-level page table

Page Directory

| P1 | P2 | Offset |

....
f08
⇒f07⇒
....

....
f99
f87
....

....
f29
f34
f25

Page Tables

Multi-level page table

P1	P2	Offset

....
f99
f87
....

....
f29
f34
f25

**Page
Tables**

Multi-level page table

P1 | P2 | Offset

....
f99
f87
....

f34

....
f29
f34
f25

Page Tables

Multi-level page table

P1	P2	Offset

....
f99
f87
....

f34	Offset

....
f29
f34
f25

Page Tables

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 P*D*Es

- **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??)**

53
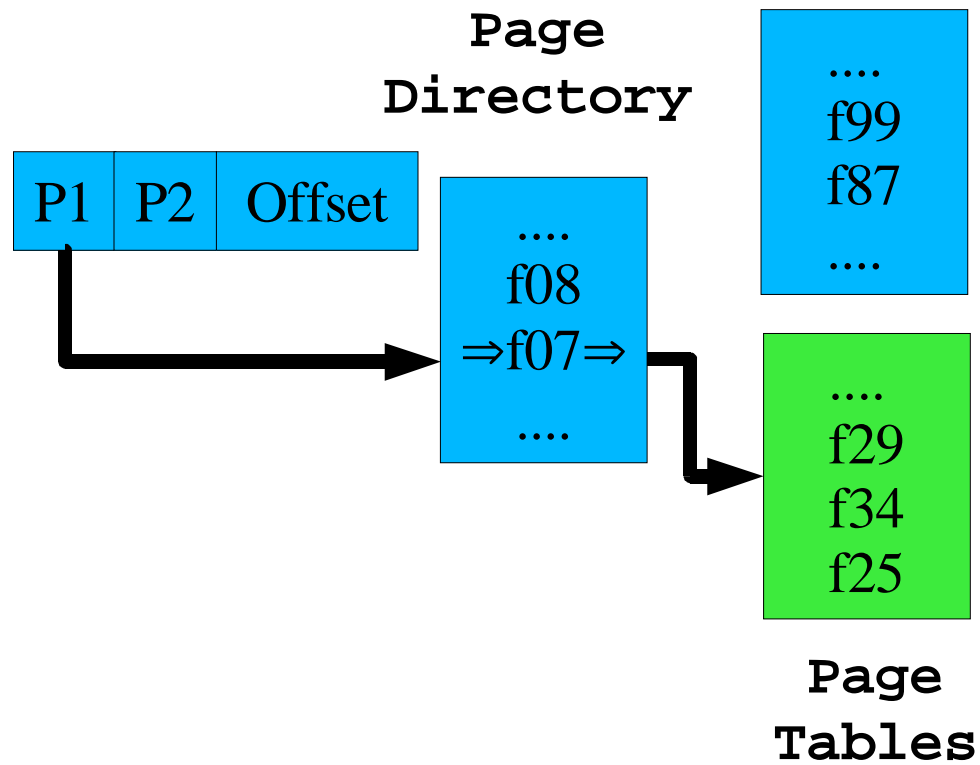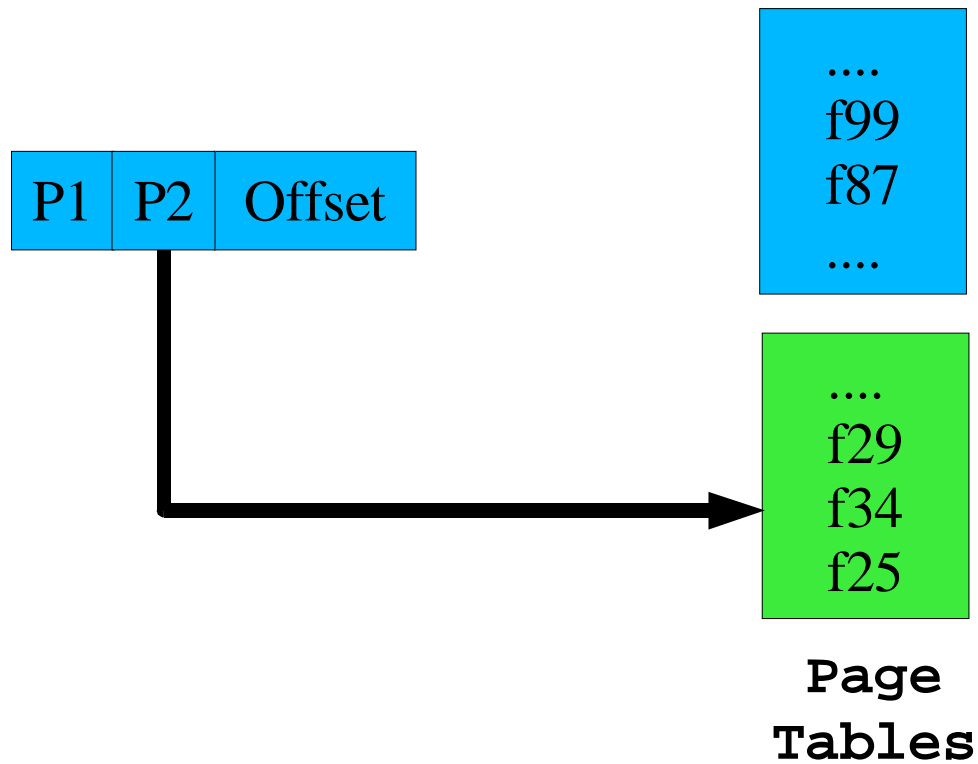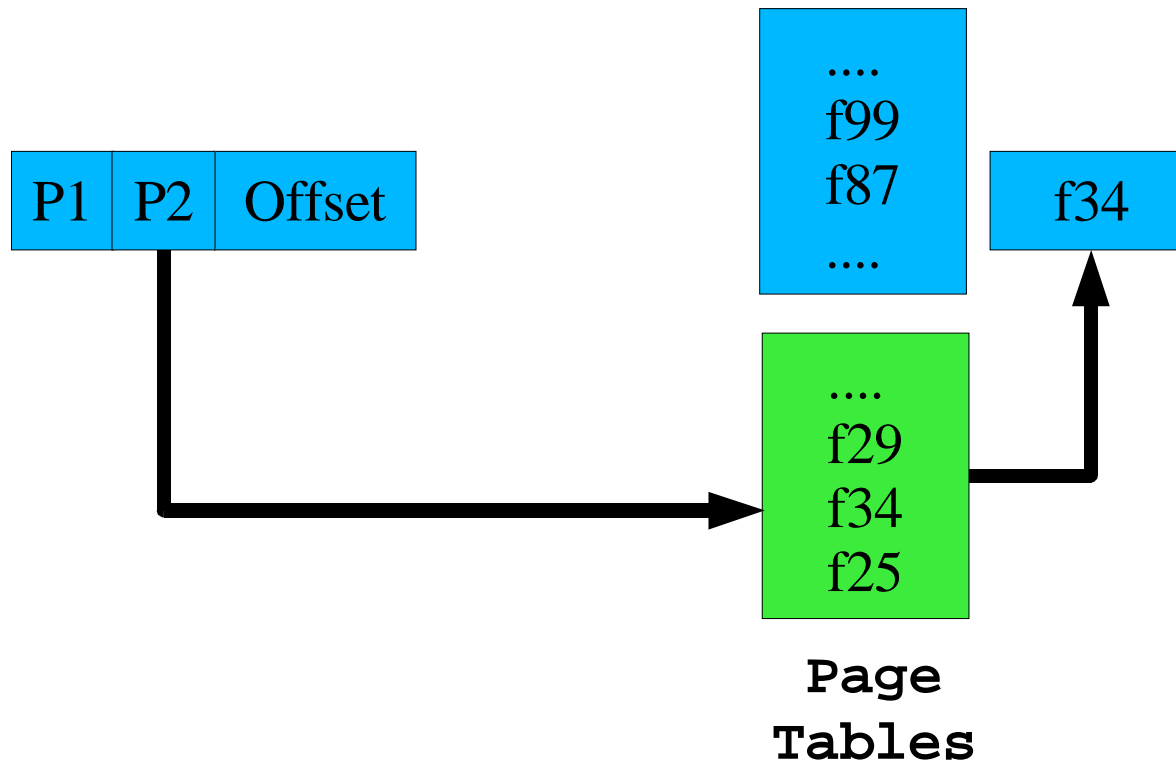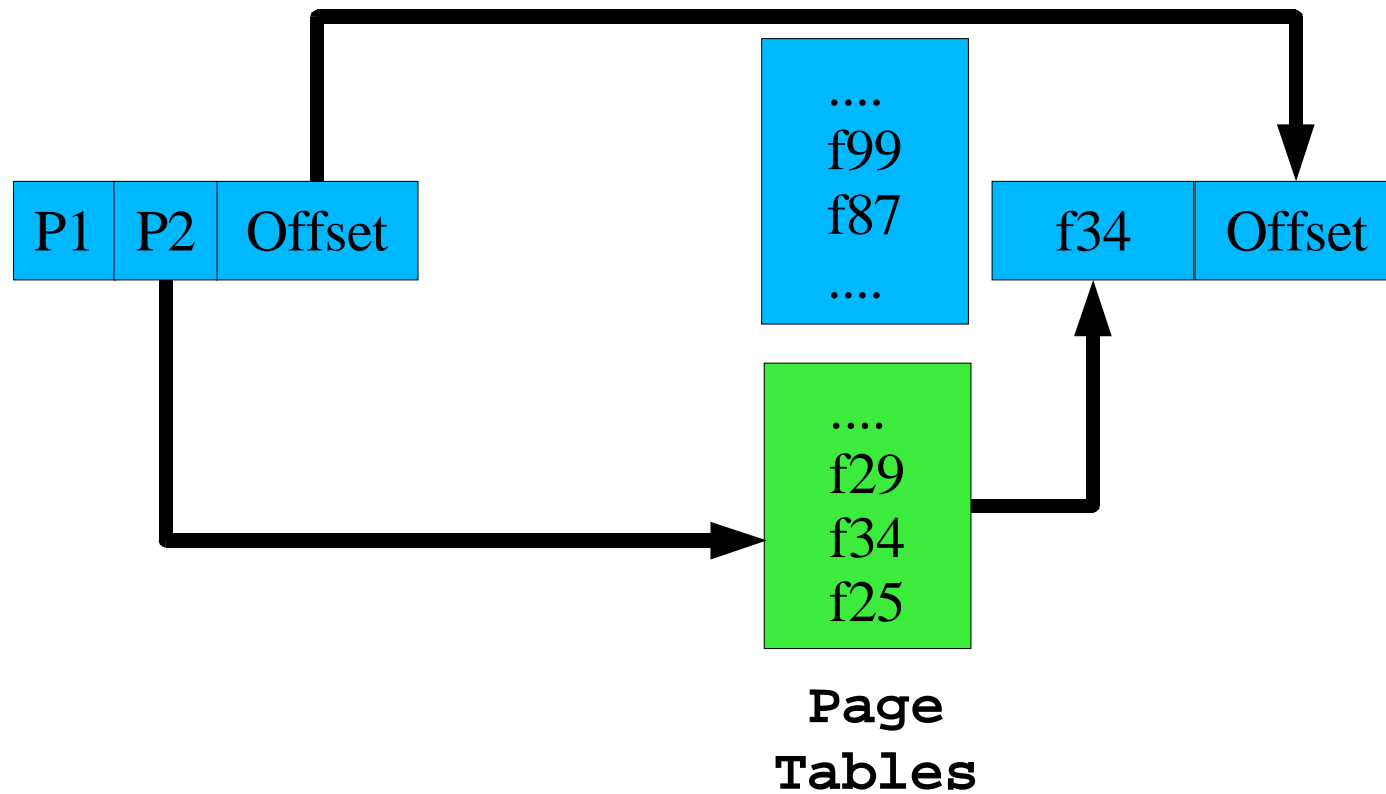
Sparse Mapping?

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 P*D*Es

- **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...**
- **...would all be full of "empty" entries...**
- **...so just use a "null pointer" in page directory instead.**
- **Result: *empty* 4GB address space specified by 4KB directory**
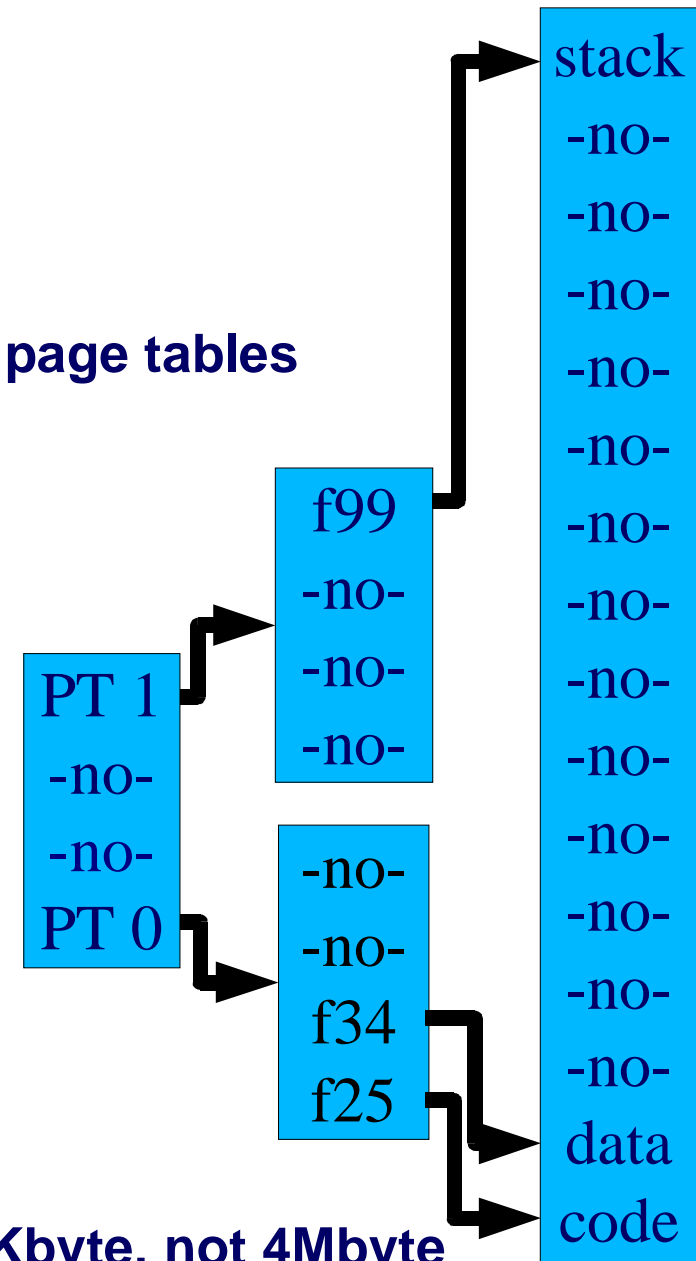
54

Sparse Mapping?

Sparsely populated page directory

- Contains pointers only to non-empty page tables

Common case

- Need 2 or 3 page tables
  - One or two map code, data
  - One maps stack
- Page directory has 1024 slots
  - Two are filled in with valid pointers
  - Remainder are "not present"

Result

- 2-3 page tables
- 1 page directory
- Map entire address space with 12-16Kbyte, not 4Mbyte

| stack |
| -no- |
| -no- |
| -no- |
| -no- |
| -no- |
| -no- |

f99
-no-
-no-
-no-

PT 1
-no-
-no-
PT 0

-no-
-no-
f34
f25

-no-
-no-
-no-
-no-
-no-
-no-
data
code

Segmentation

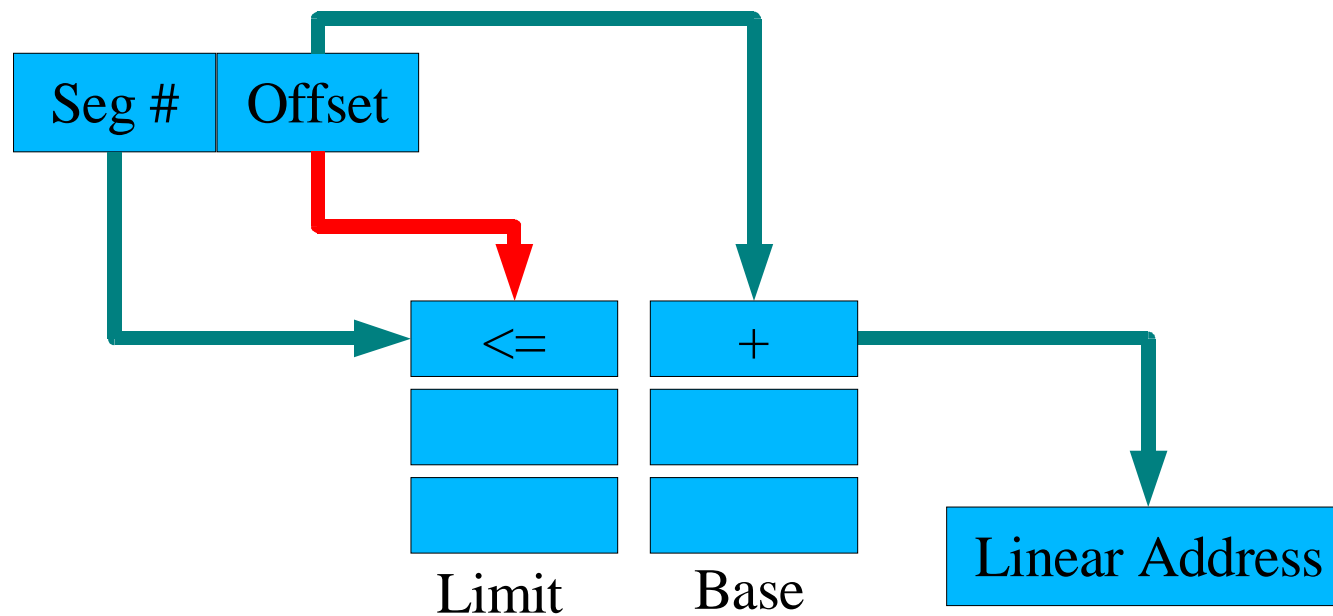**Physical memory is (mostly) linear**

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

56

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

58

x86 Type Theory

Instruction ⇒ segment selector

- [PUSHL implicitly specifies selector in %SS]

Process ⇒ (selector ⇒ (base,limit))

- [Global,Local Descriptor Tables]

Segment, in-segment address ⇒ linear address

- CS:EIP means "EIP + base of code segment"

Process ⇒ (linear address high ⇒ page table)

- [Page Directory Base Register, page directory indexing]

Page Table: linear address middle ⇒ frame address

Memory: frame address + offset ⇒ ...

59

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*

60

Quote

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

—Roger Needham