15-410

"...The only way to win is not to play..."

Virtual Memory #1
Feb. 23, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Mid-term

- Wednesday, 19:00, 7500 Wean
- Does not cover today's lecture

Final Exam list posted

- You must notify us of conflicts in a timely fashion

Summer internship with SCS Facilities?

Outline

The Problem: logical vs. physical

Contiguous memory mapping

Fragmentation

Paging

- **Type theory**
- **Several mapping functions**

TLB

Logical vs. Physical

It's all about address spaces

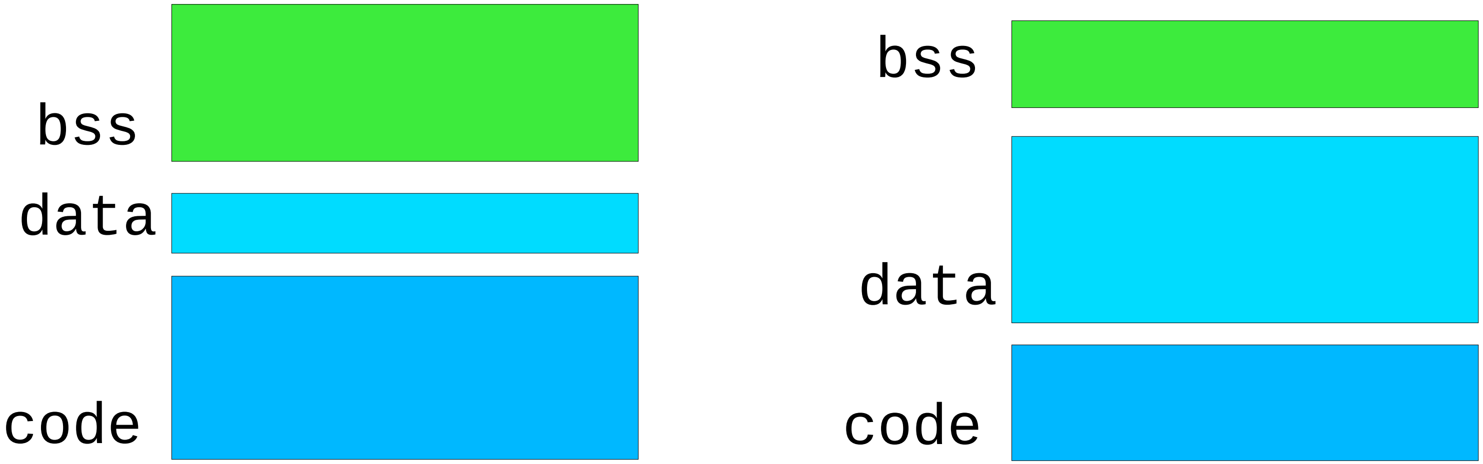
- Generally a complex issue
  - IPv4 $\Rightarrow$ IPv6 is mainly about address space exhaustion

Review

- Combining .o's changes addresses

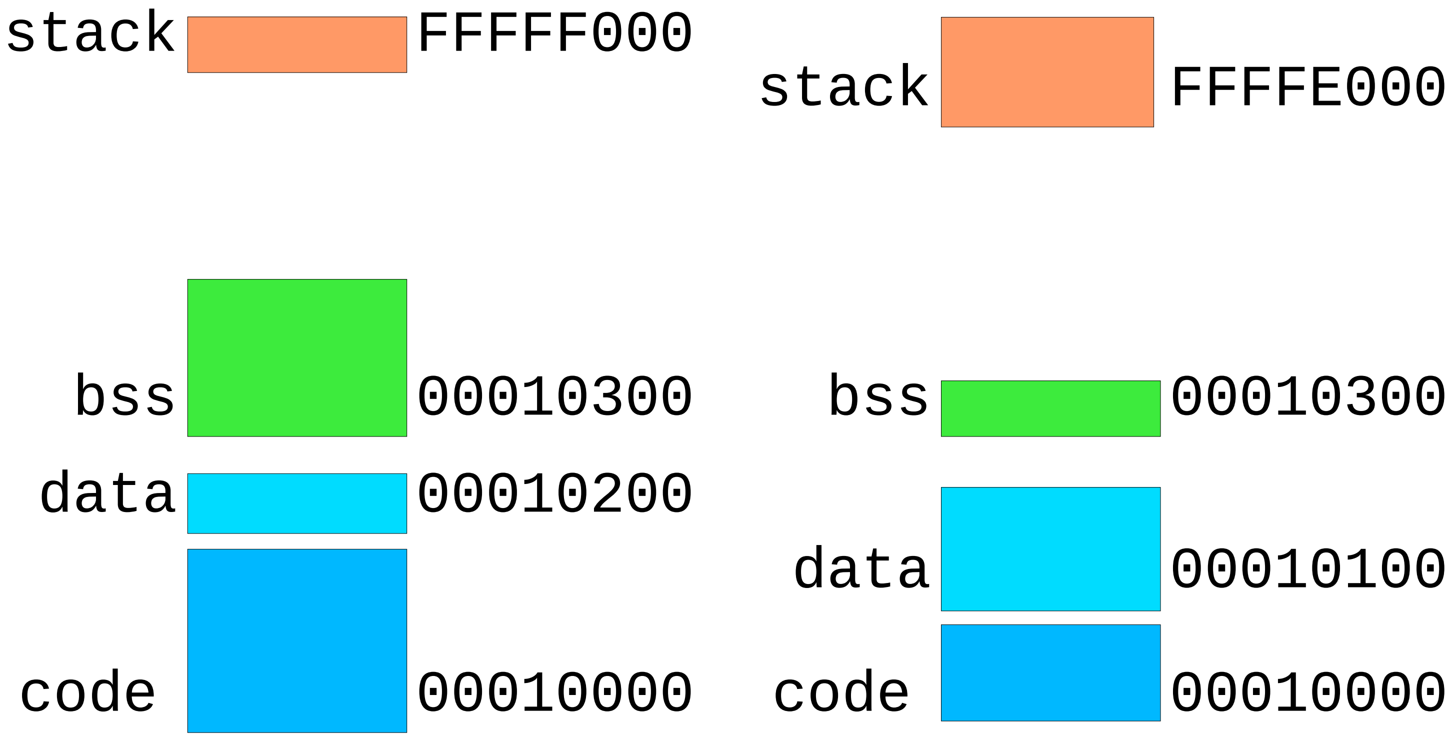
But what about *two* programs?

Every .o uses same address space



Combining .o's Changes Addresses

What about *two* programs?



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

Ok, load programs at *different* addresses

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

Programs can *take turns* in memory

- Requires swapping programs out to disk
- Very slow

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 ⇒ data
- store: address, data ⇒ .

Process thinks of memory as...

- fetch: address ⇒ data
- store: address, data ⇒ .

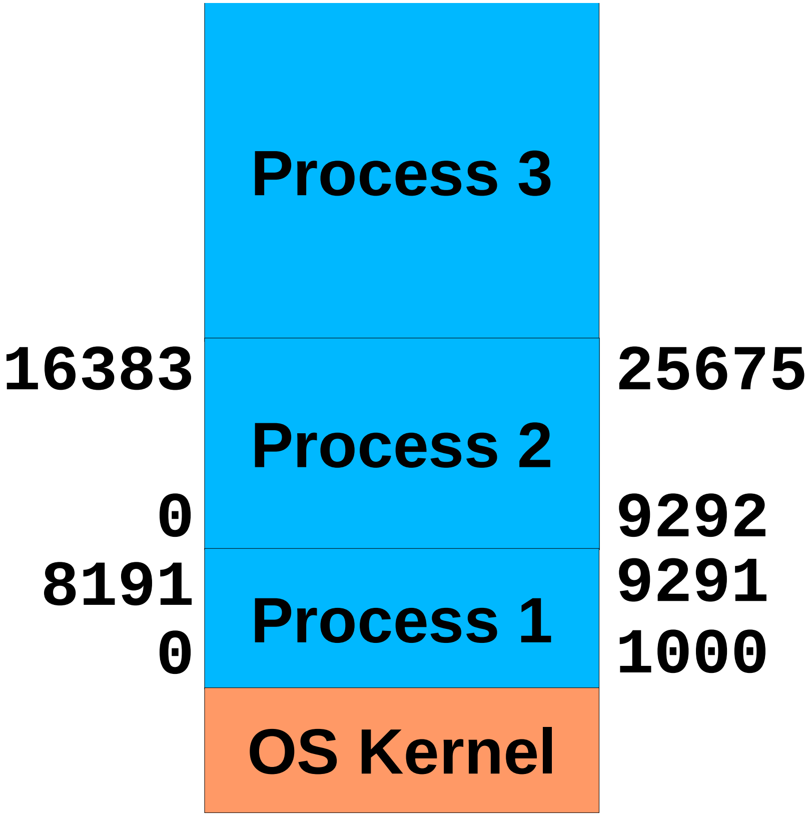
Goal: each process has "its own memory"

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

What *really* happens

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

Simple Mapping Functions



P1

If V > 8191 *ERROR*

Else P = 1000 + V

P2

If V > 16383 *ERROR*

Else P = 9292 + V

Address space ===

- 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? */
```

Context switch

- **Save/load registers**
- **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

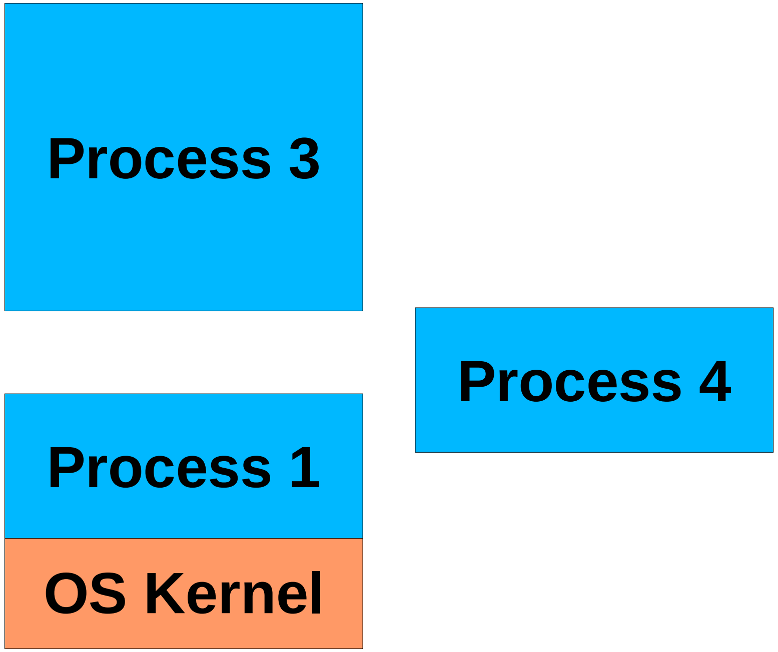
- Must *entire* program be in memory at same time?

Can We Run Process 4?

Process exit creates "holes"

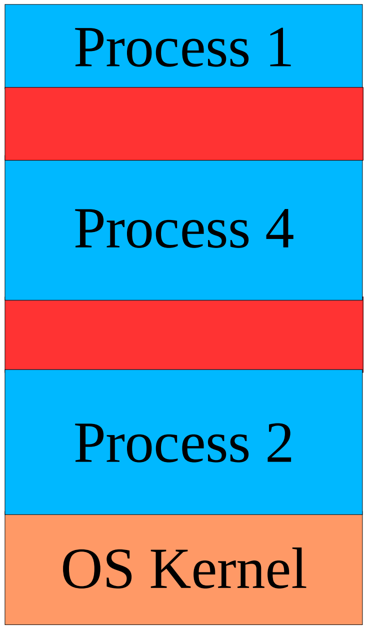
New processes may be too large

May require moving entire address spaces



External Fragmentation

- Free memory in small chunks
- Doesn't fit large objects
- Can disable lots of memory
- Can fix
  - Costly "compaction"



Internal Fragmentation

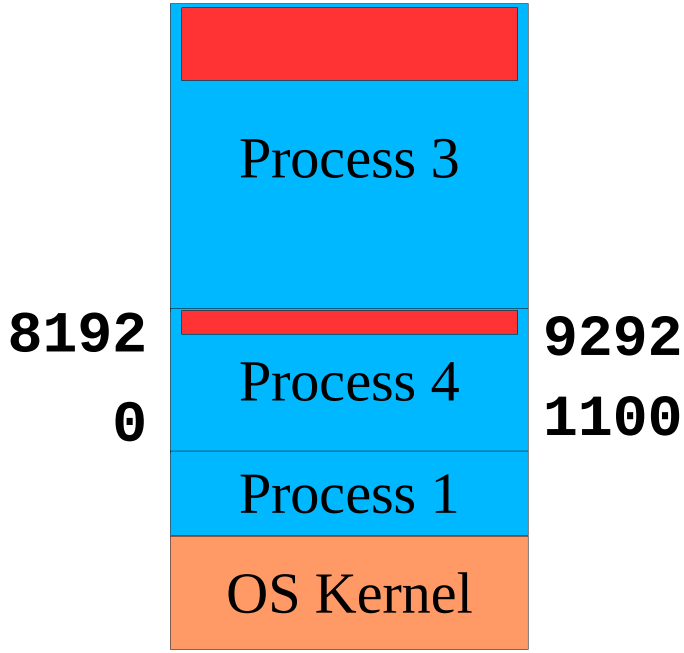
Allocators often round up

- **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

Contiguous Allocation ⇒ Paging

Solve multiple problems

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

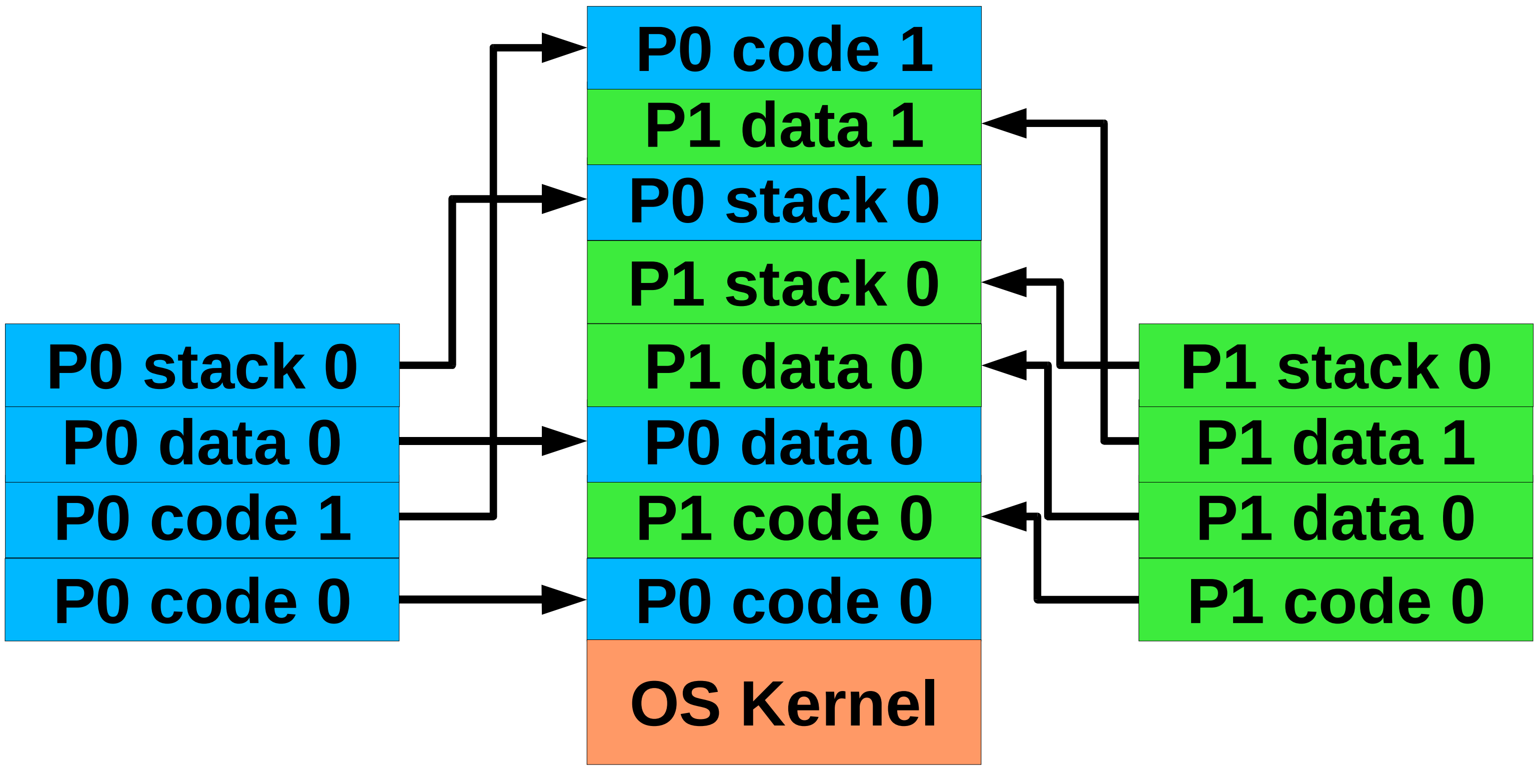
Divide memory more finely

- *Page* = small region of *virtual* memory (4K)
- *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



Benefits of Paging

Process growth problem

- Any process can use any free frame for any purpose

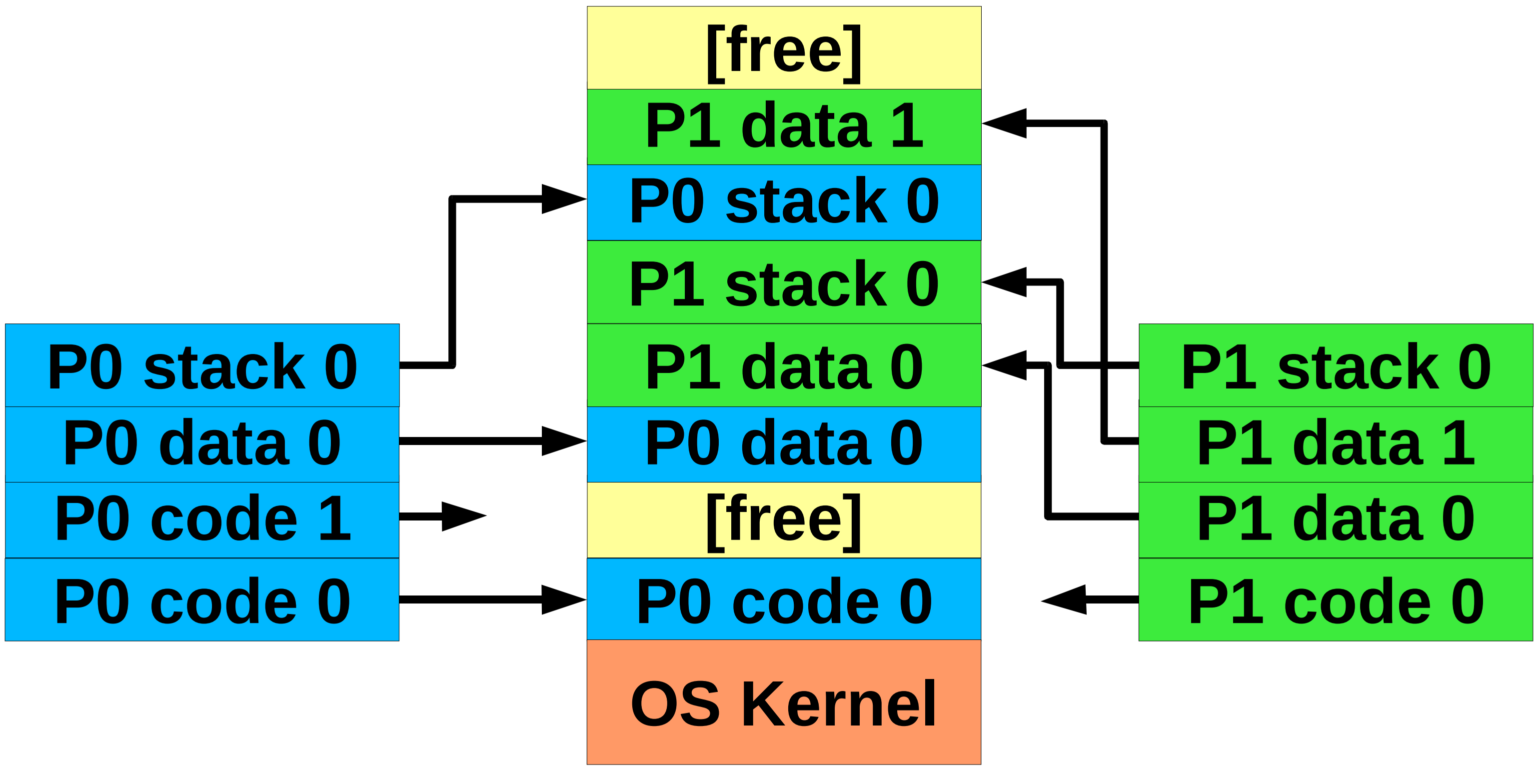
Fragmentation compaction problem

- Process doesn't need to be contiguous

Long delay to swap a whole process

- Swap *part* of the process instead!

Partial Residence



New Data Structure

Contiguous allocation

- Each process described by (base,limit)

Paging

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

Page Table Options

Linked list

- V⇒P time gets longer for large addresses!

Array

- Constant time access
- Requires contiguous memory for table

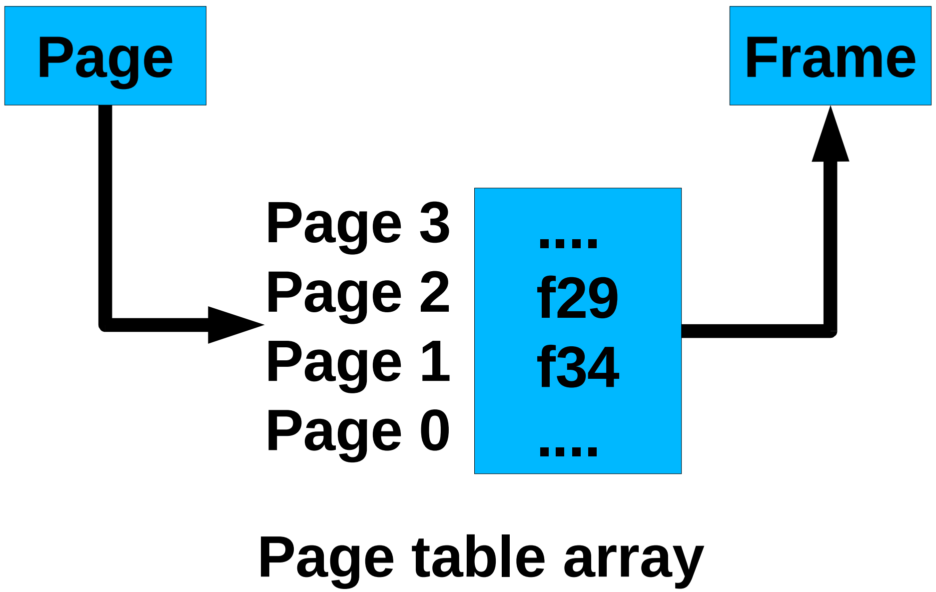
Hash table

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

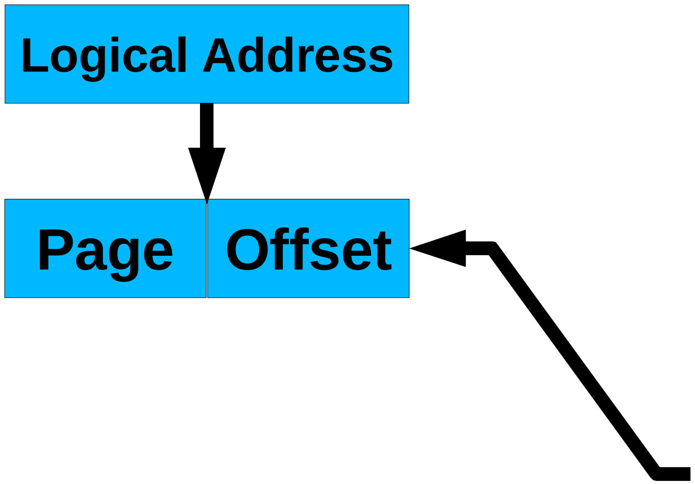
Splay tree

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

Page Table Array

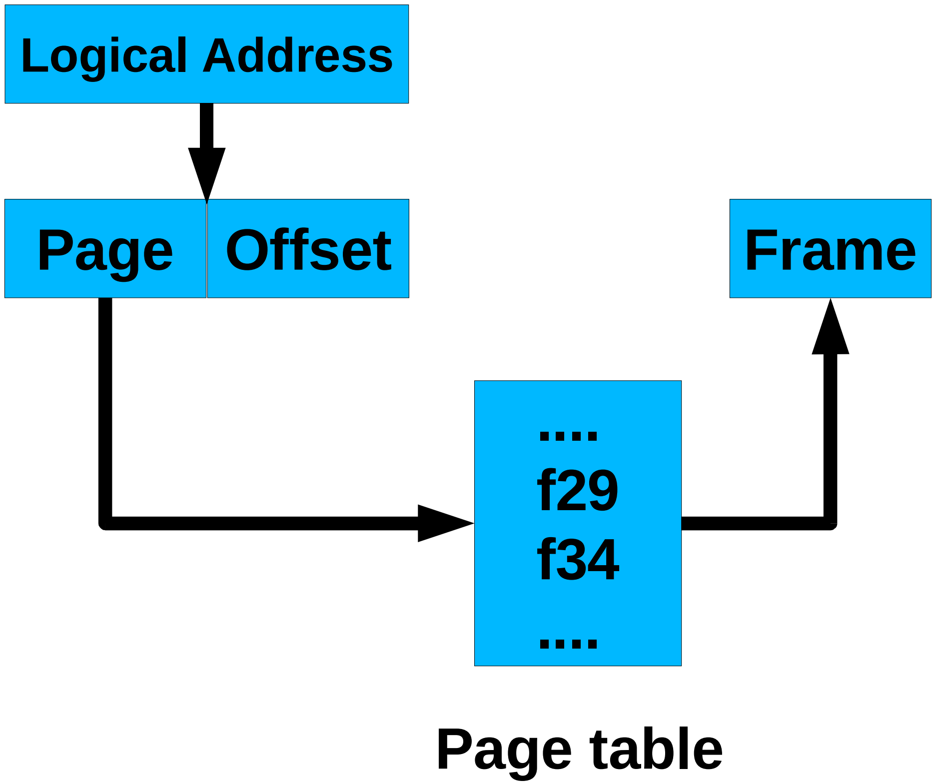


Paging – Address Mapping

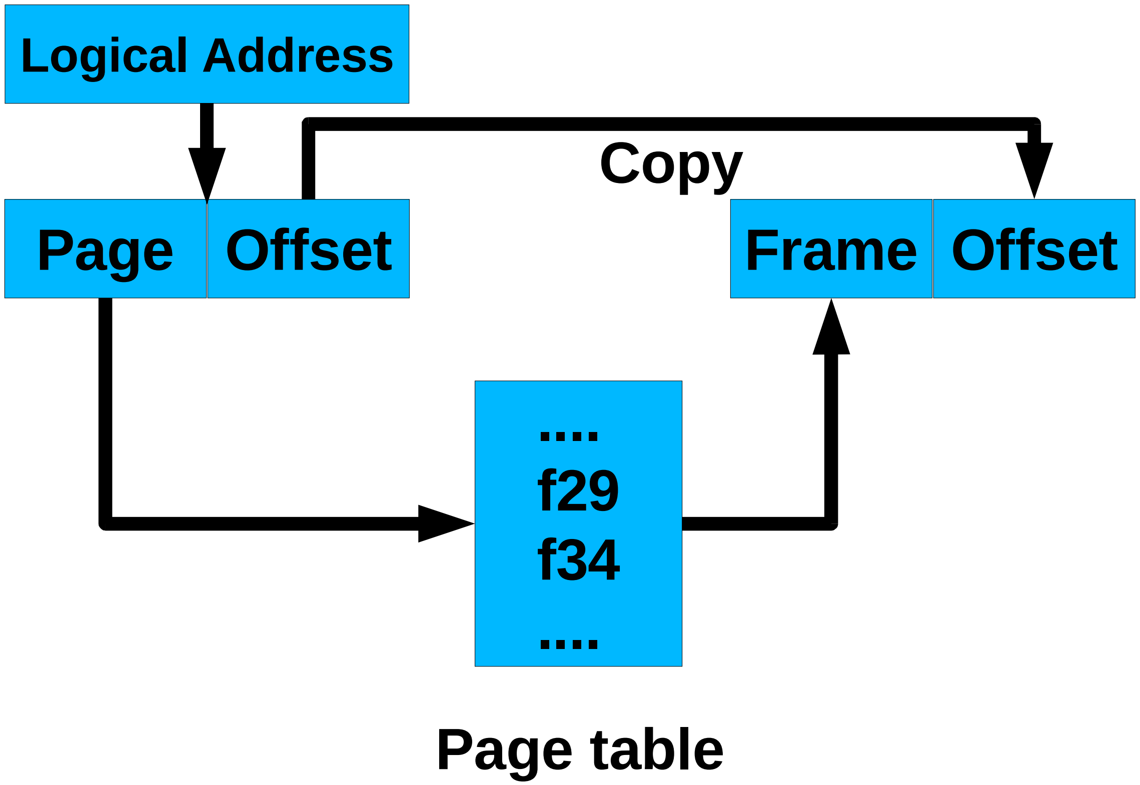


4K page size $\Rightarrow$ 12 bits

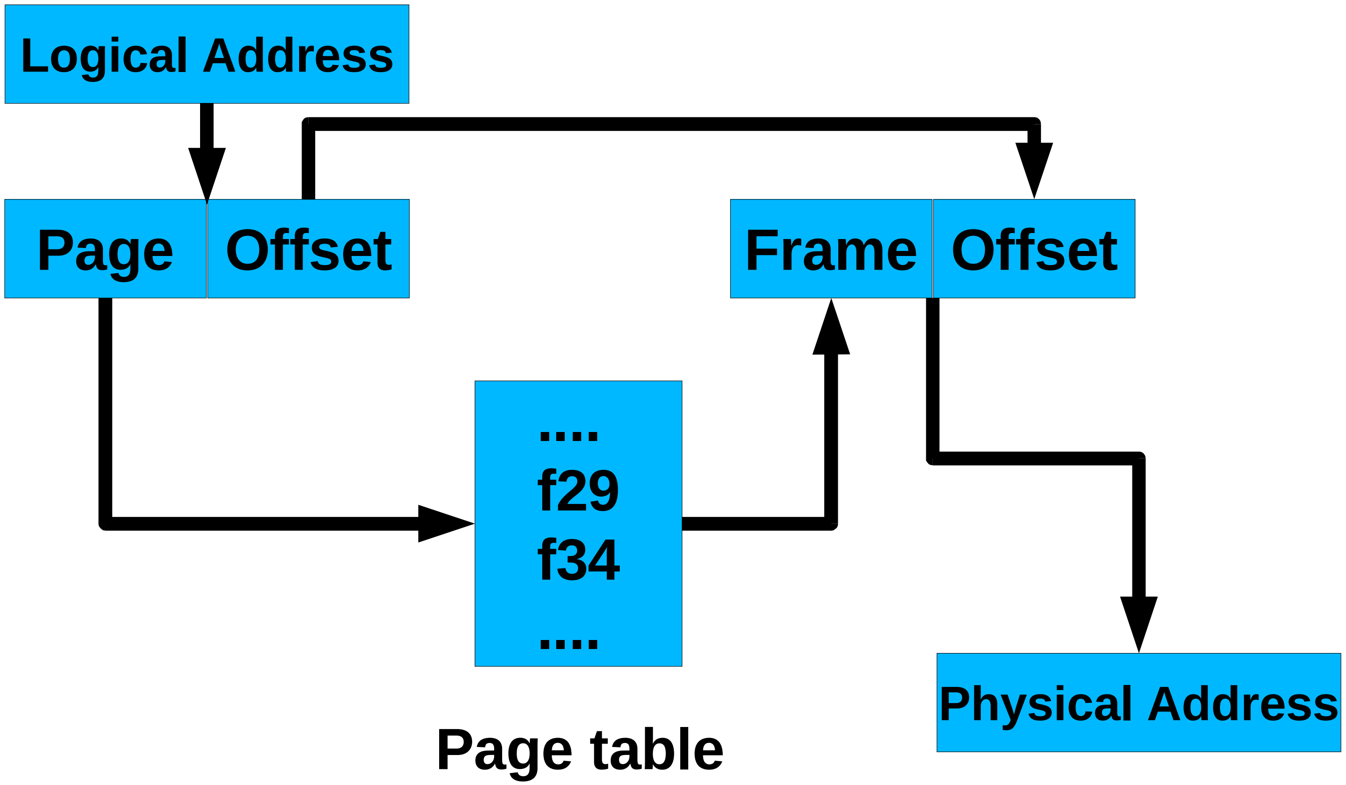
Paging – Address Mapping



Paging – Address Mapping



Paging – Address Mapping



Paging – Address Mapping

User view

- Memory is a linear array

OS view

- Each process requires N frames

Fragmentation?

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

Bookkeeping

One page table for each process

One frame table

- Manages free frames
- Remembers who owns a frame

Context switch

- Must “activate” process's page table

Hardware Techniques

Small number of pages?

- "Page table" can be a few registers

Typical case

- Large page tables, live in memory
  - Where? Processor has "Page Table Base Register"

Double trouble?

Program requests memory access

Processor makes *two* memory accesses!

- **Split address into page number, intra-page offset**
- **Add to page table base register**
- ***Fetch page table entry (PTE) from memory***
- **Add frame address, intra-page offset**
- ***Fetch data from memory***

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: 1024:1
  - 4 GByte virtual address (32 bits) ⇒ 4 MByte page table
  - ***For each process!***

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***
- Page tables are not randomly sparse
  - Occupied by ***sequential memory regions***
  - Text, rodata, data+bss, stack

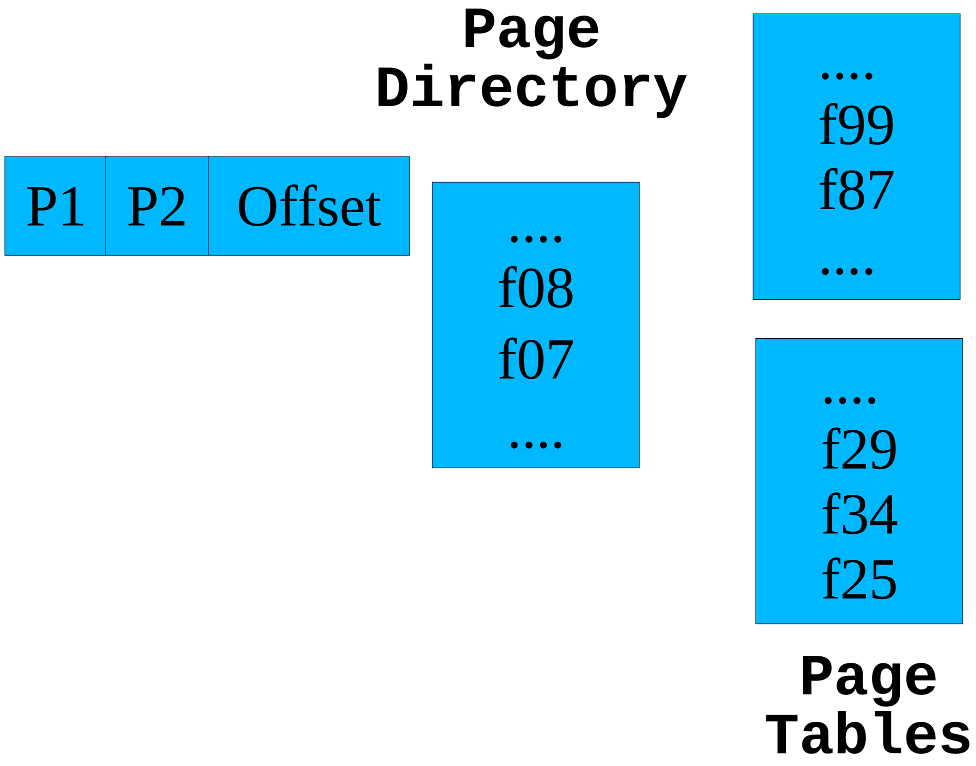
We are computer scientists!

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

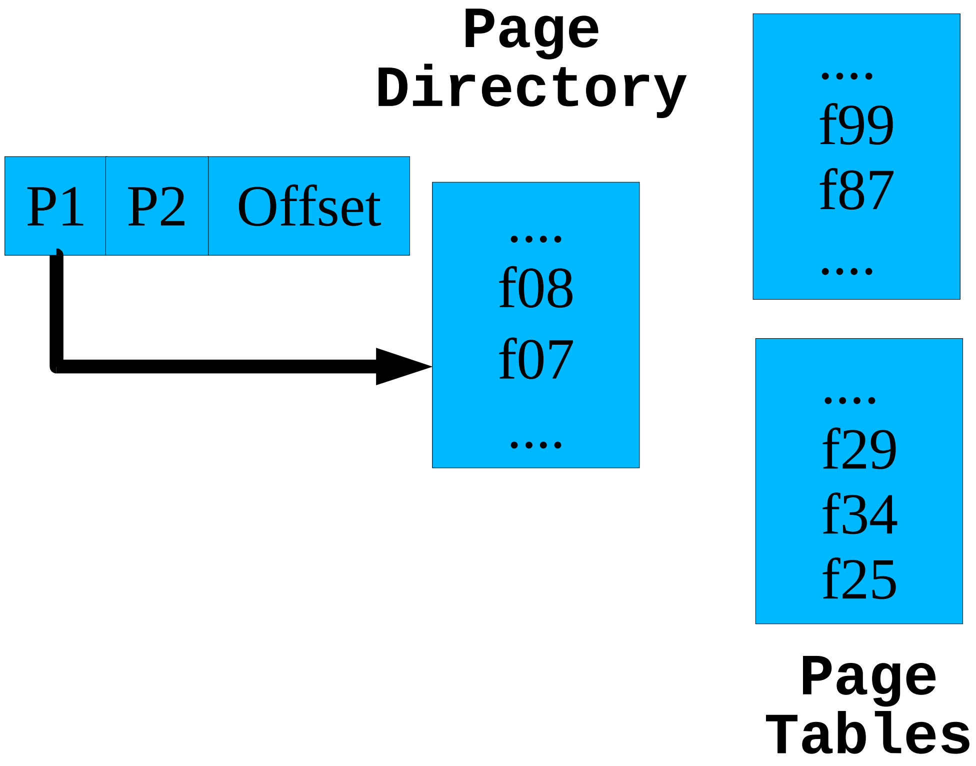
Multi-level page table

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

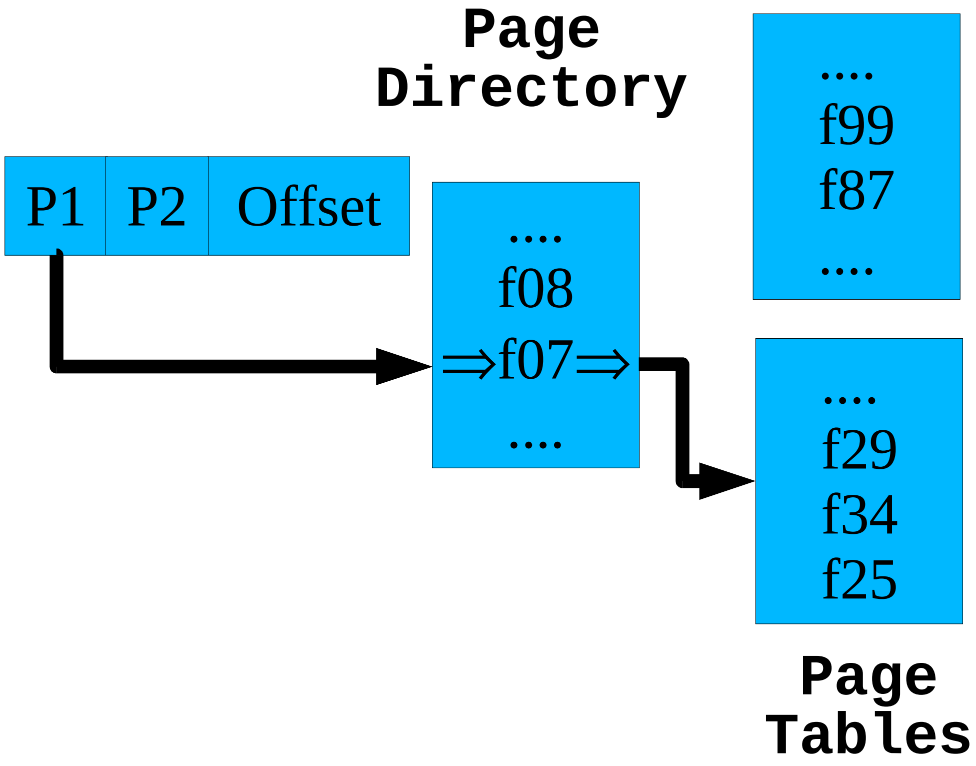
Multi-level page table



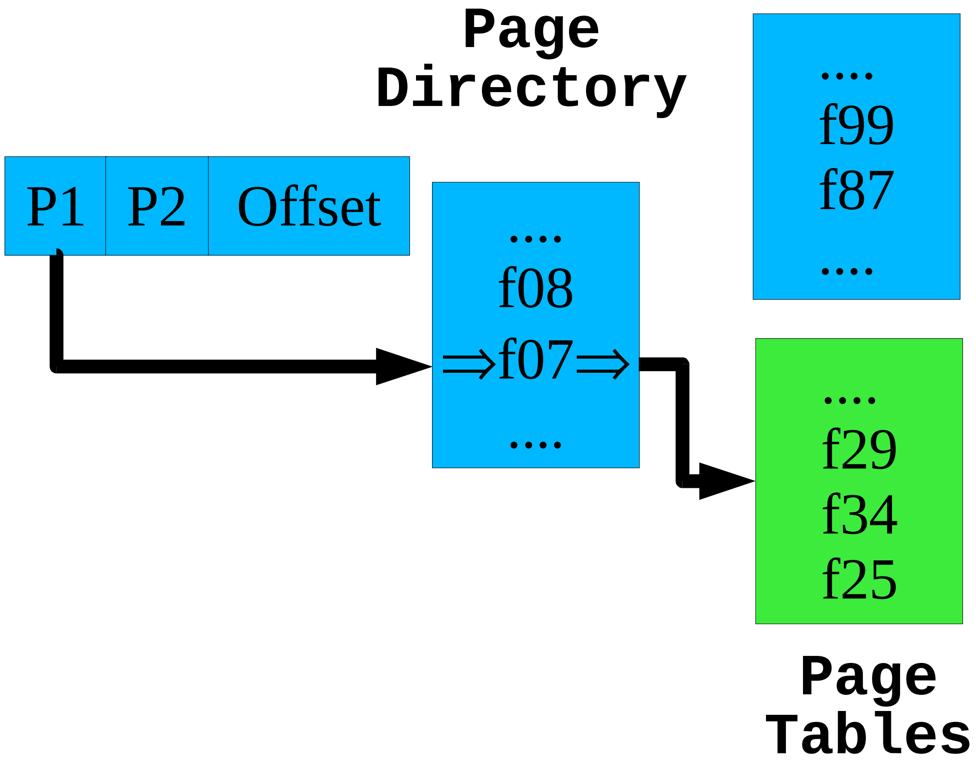
Multi-level page table



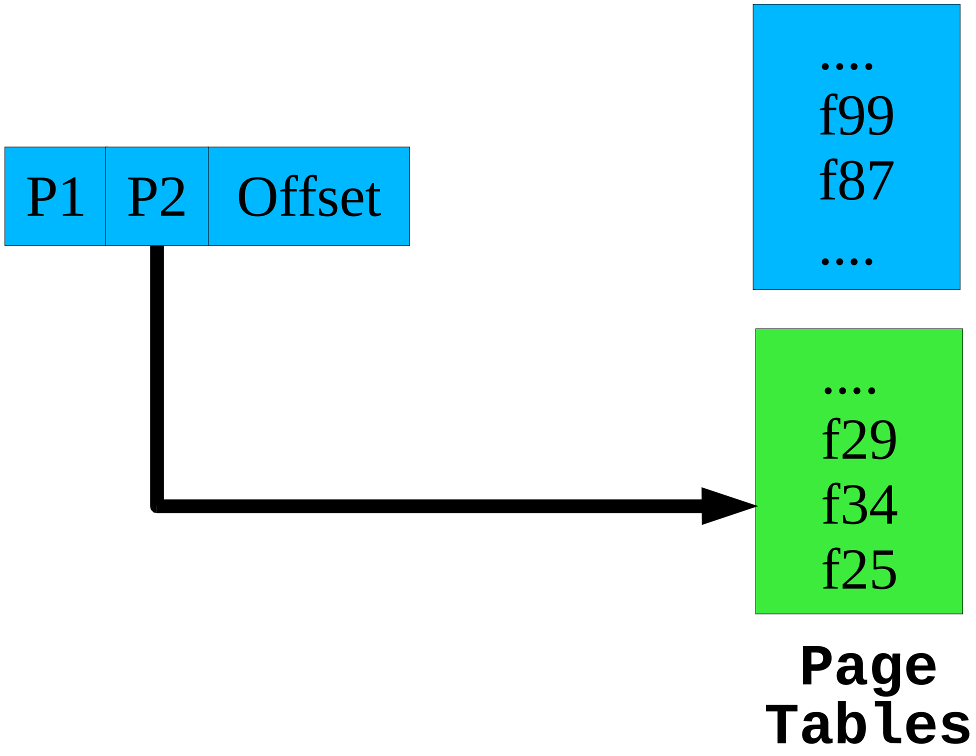
Multi-level page table



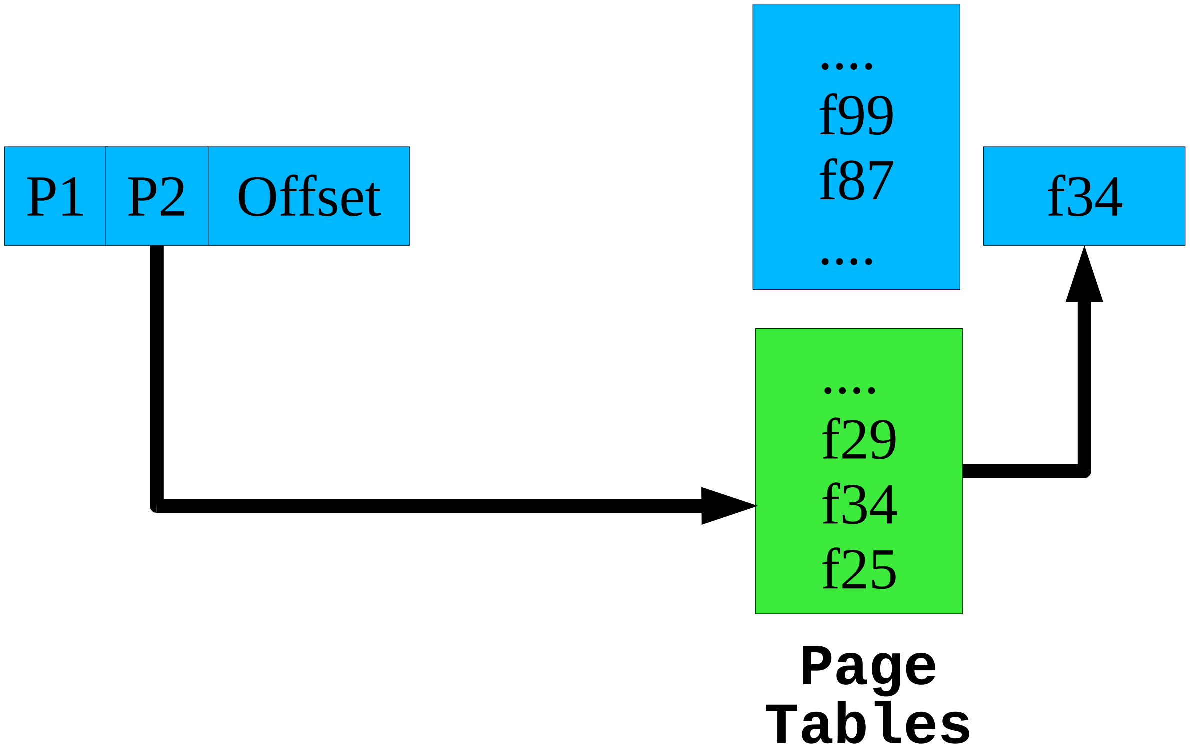
Multi-level page table



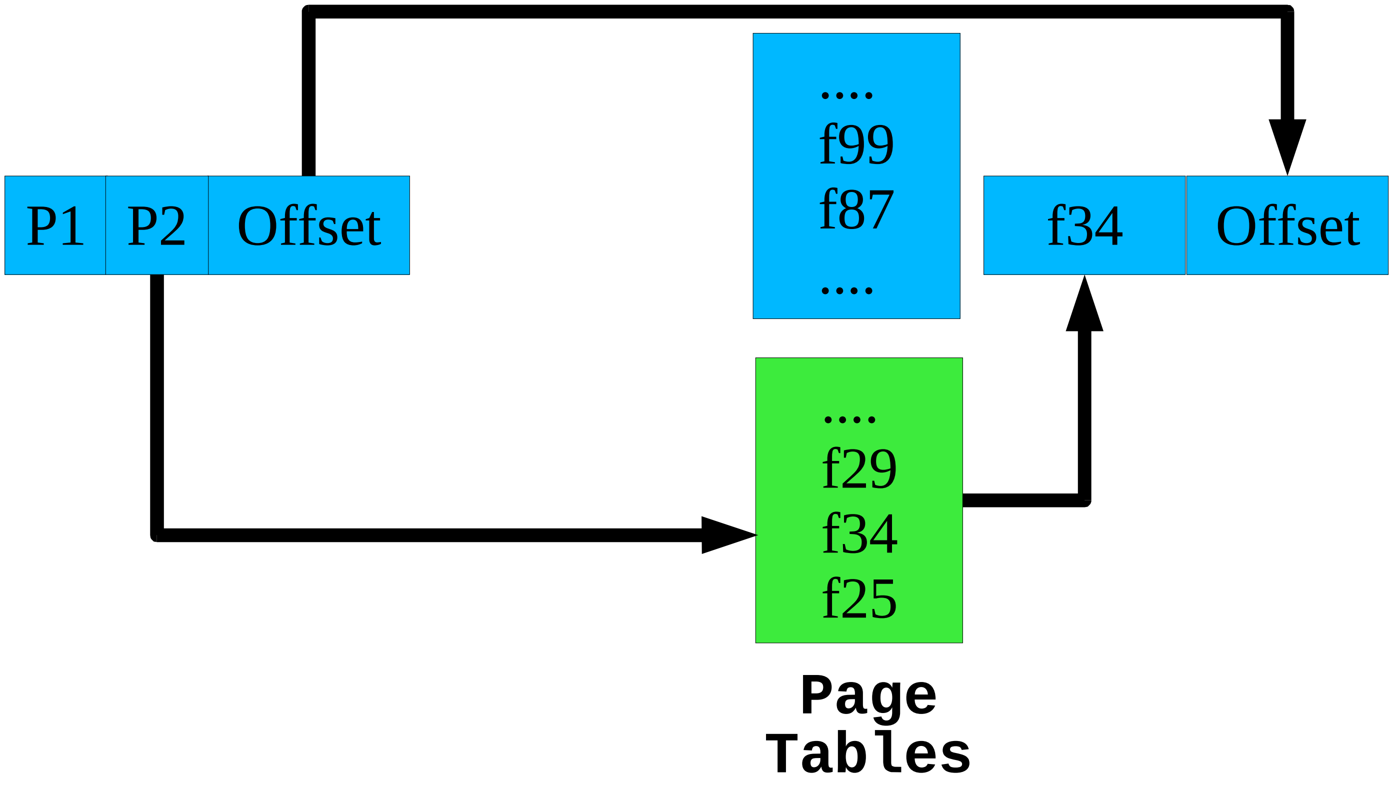
Multi-level page table



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**
- **Single 1024-entry (4Kbyte) page directory can cover them**

Sparse address space...

- **...means most page tables contribute nothing to mapping...**
- **...all of them are full of "empty" entries**

Sparse Mapping?

Page directory can be sparse

- 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
  - Three are filled in with valid pointers
  - Remainder are "not present"

Result

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

Segmentation

Physical memory is (mostly) linear

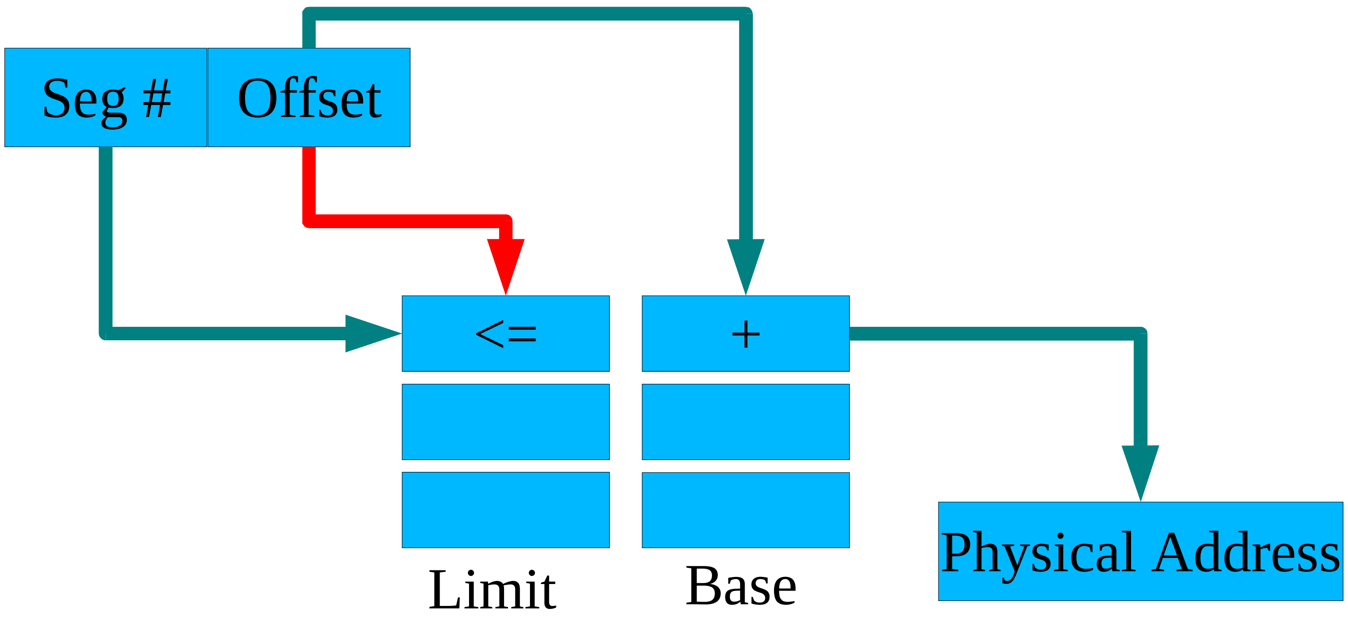
Is virtual memory linear?

- 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
    - CS register holds 16-bit selector
  - 32-bit offset within a segment -- CS:EIP
- Descriptor table maps selector to segment descriptor
- Offset fed to segment descriptor, generates linear address
- If linear address not in TLB...
- Linear address fed through page directory, page table

x86 Type Theory

Instruction ⇒ segment selector

- [PUSHL specifies selector in %SS]

Process ⇒ (selector ⇒ (base,limit))

- [Global,Local Descriptor Tables]

Segment, address ⇒ linear address

TLB: linear address ⇒ physical address or...

Process ⇒ (linear address high ⇒ page table)

- [Page Directory Base Register, page directory indexing]

Page Table: linear address middle ⇒ frame address

Memory: frame address, offset ⇒ ...

Is there another way?

That seems *really complicated*

- **Is that hardware monster really optimal for every OS and program mix?**
- **"The only way to win is not to play?"**

Could we have *no* page tables?

How would hardware map virtual to physical???

Software-loaded 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 contains small number of mappings
- OS knows the rest
- 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

Data structures determined by access patterns

- Most address spaces are *sparsely allocated*