15-410

"...The cow and Zaphod..."

Virtual Memory #2
Feb. 22, 2006

Dave Eckhardt

Bruce Maggs

Last Time

Mapping problem: logical vs. physical addresses

Contiguous memory mapping (base, limit)

Swapping – taking turns in memory

Paging

- **Array mapping page numbers to frame numbers**
- **Observation: typical table is *sparsely occupied***
- **Response: some sparse data structure (e.g., 2-level array)**

Swapping

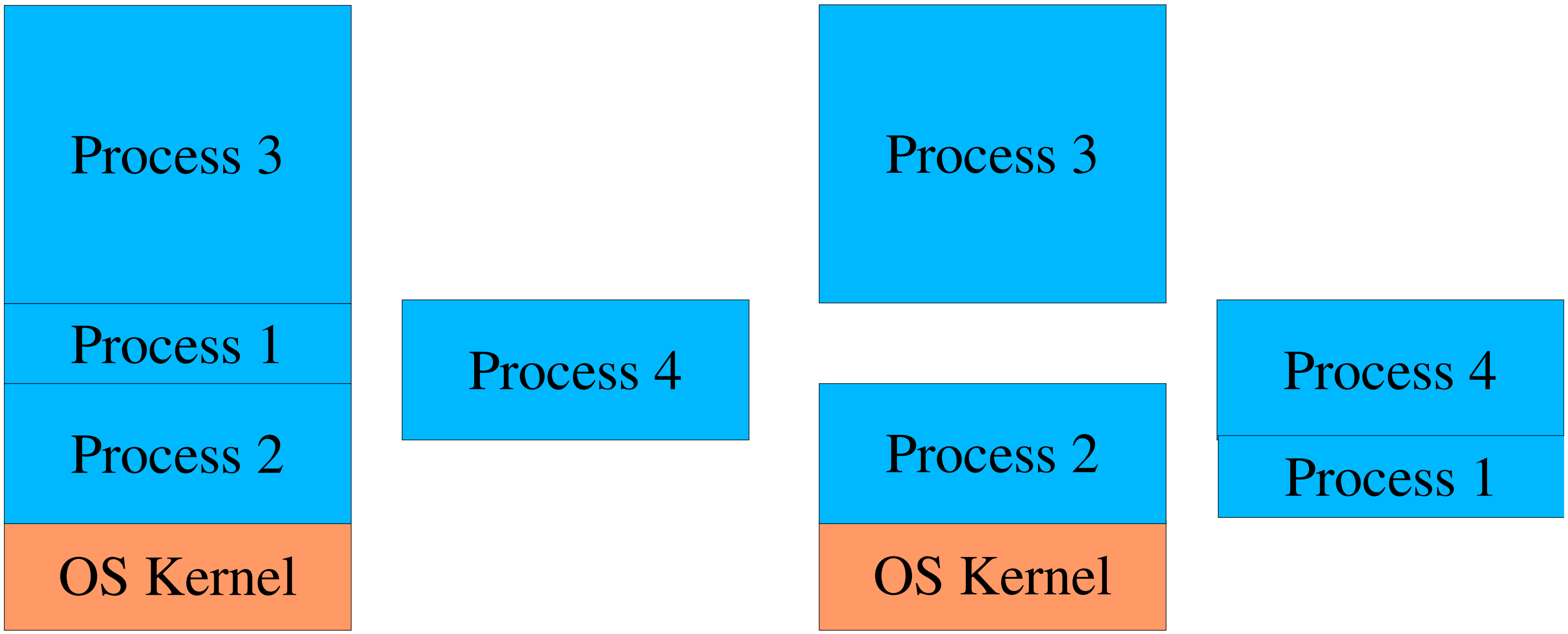
Multiple user processes

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

Take turns

- Temporarily evict process(es) to disk
- “Swap daemon” shuffles process in & out
- Can take *seconds* per process
- Creates *external fragmentation* problem

External Fragmentation ("Holes")



Benefits of Paging

Process growth problem

- Any process can use any free frame for any purpose

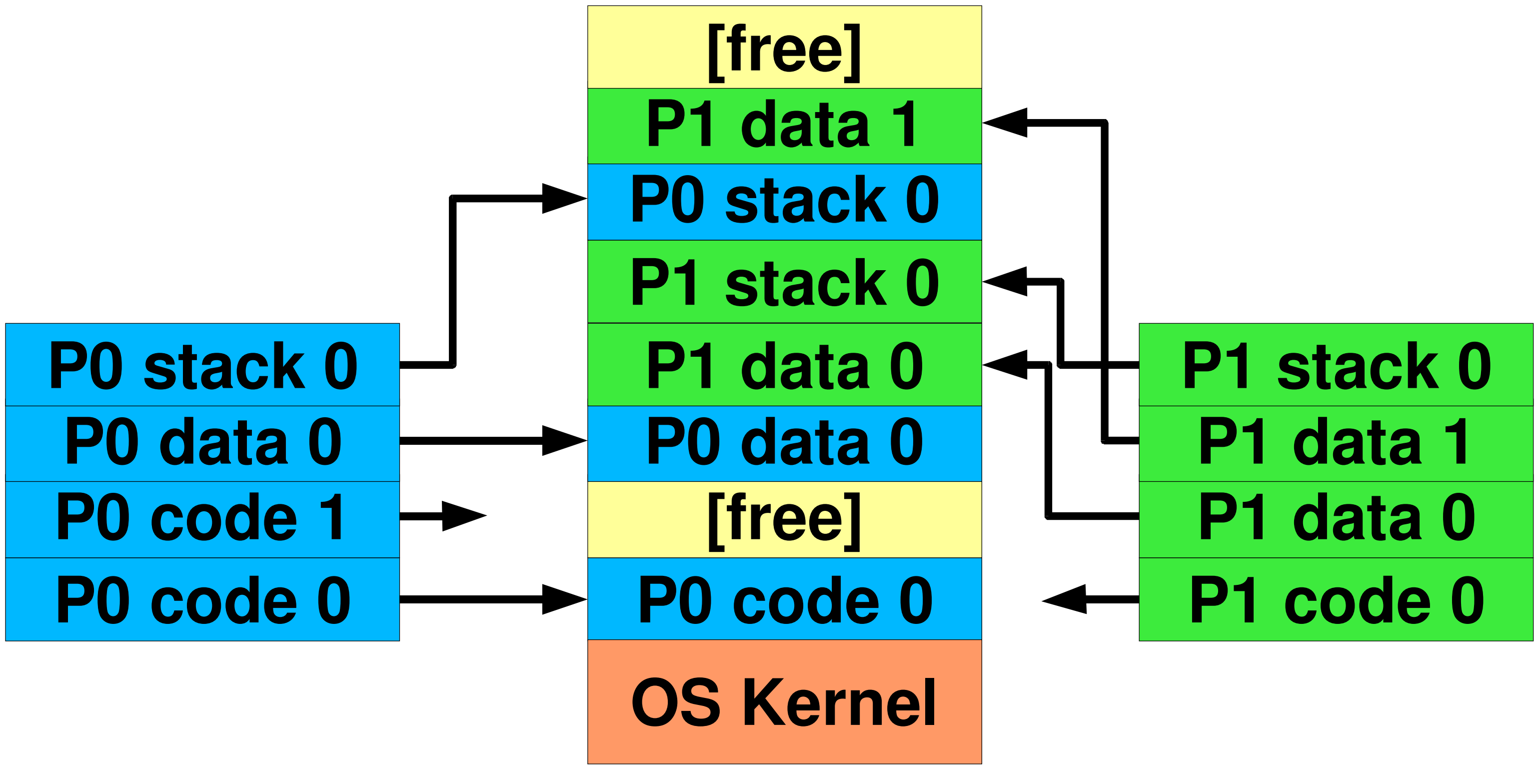
Fragmentation compaction problem

- Process doesn't need to be contiguous

Long delay to swap a whole process

- Swap *part* of the process instead!

Partial Residence



Page Table Entry (PTE) flags

Protection bits – set by OS

- Read/write/execute

Valid/Present bit – set by OS

- Frame pointer is valid, no need to fault

Dirty bit

- Hardware sets 0⇒1 when data stored into page
- OS sets 1⇒0 when page has been written to disk

Reference bit

- Hardware sets 0⇒1 on any data access to page
- OS uses for page eviction (below)

Outline

Partial memory residence (demand paging) in action

The task of the page fault handler

Big speed hacks

Sharing memory regions & files

Page replacement policies

Partial Memory Residence

Error-handling code not used by every run

- No need for it to occupy memory for entire duration...

Tables may be allocated larger than used

```
player players[MAX_PLAYERS];
```

Computer can run *very* large programs

- Much larger than physical memory
- As long as “active” footprint fits in RAM
- Swapping can't do this

Programs can launch faster

- Needn't load whole program before running

Demand Paging

Use RAM frames as a cache for the set of all pages

- Some pages are fast to access (in a RAM frame)
- Some pages are slow to access (in a disk "frame")

Page tables indicate which pages are "resident"

- Non-resident pages have "present=0" in page table entry
- Memory access referring to page generates *page fault*
  - Hardware invokes page-fault exception handler

Page fault – Reasons, Responses

Address is invalid/illegal – deliver *software exception*

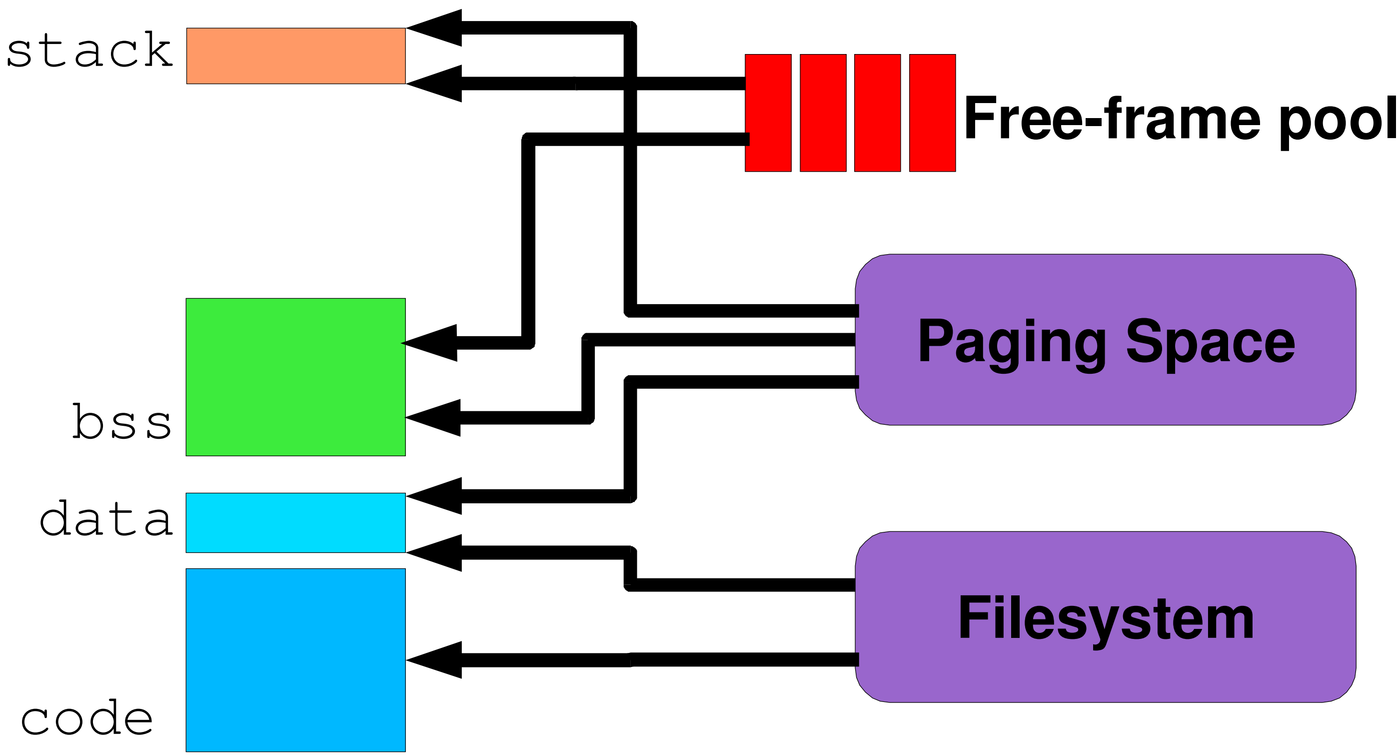
- Unix – SIGSEGV
- Mach – deliver message to thread's exception port
- 15-410 – kill thread

Process is growing stack – give it a new frame

"Cache misses" - fetch from disk

- Where on disk, exactly?

Satisfying Page Faults



Page fault story - 1

Process issues memory reference

- TLB: miss
- PT: "not present"

Trap to OS kernel!

- Processor dumps trap frame onto kernel stack (x86)
- Transfers via "page fault" interrupt descriptor table entry
- Runs trap handler

Page fault story – 2

Classify fault address

- Illegal address $\Rightarrow$ deliver an ouch, else...

Code/rodata region of executable?

- Determine which sector of executable file
- Launch read() from file into an unused frame

Previously resident r/w data, paged out

- "somewhere on the paging partition"
- Queue disk read into an unused frame

First use of bss/stack page

- Allocate a frame full of zeroes, insert into PT

Page fault story – 3

Put process to sleep (for most cases)

- Switch to running another

Handle I/O-complete interrupt

- Fill in PTE (present = 1)
- Mark process runnable

Restore registers, switch page table

- Faulting instruction re-started transparently
- *Single instruction may fault more than once!*

Memory Regions vs. Page Tables

What's a poor page fault handler to do?

- Kill process?
- Copy page, mark read-write?
- Fetch page from file? Which? Where?

Page table not a good data structure

- Format defined by hardware
- Per-page nature is repetitive
- Not enough bits to encode OS metadata
  - Disk sector address can be > 32 bits

Dual-view Memory Model

Logical

- Process memory is a list of *regions*
- "Holes" between regions are *illegal addresses*
- Per-region methods
  - fault(), evict(), unmap()

Physical

- Process memory is a list of *pages*
- Faults delegated to per-region methods
- Many "invalid" pages can be made valid
  - But sometimes a region fault handler returns "error"
    - » Handle as with "hole" case above

Page-fault story (for real)

Examine fault address

Look up: address ⇒ region

```
region->fault(addr, access_mode)
```

- ***Quickly*** **fix up problem**
- **Or start fix, put process to sleep, run scheduler**

Demand Paging Performance

Effective access time of memory word

- $(1 - p_{miss}) * T_{memory} + p_{miss} * T_{disk}$

Textbook example (a little dated)

- T_{memory} 100 ns
- T_{disk} 25 ms
- $p_{miss} = 1/1{,}000$ slows down by factor of 250
- slowdown of 10% needs $p_{miss} < 1/2{,}500{,}000$!!!

Speed Hacks

COW

ZFOD (Zaphod?)

Memory-mapped files

- **What msync() is *supposed* to be used for...**

Copy-on-Write

fork() produces two *very*-similar processes

- Same code, data, stack

Expensive to copy pages

- Many will never be modified by new process
  - Especially in fork(), exec() case

Share physical frames instead of copying?

- Easy: code pages – read-only
- Dangerous: stack pages!

Copy-on-Write

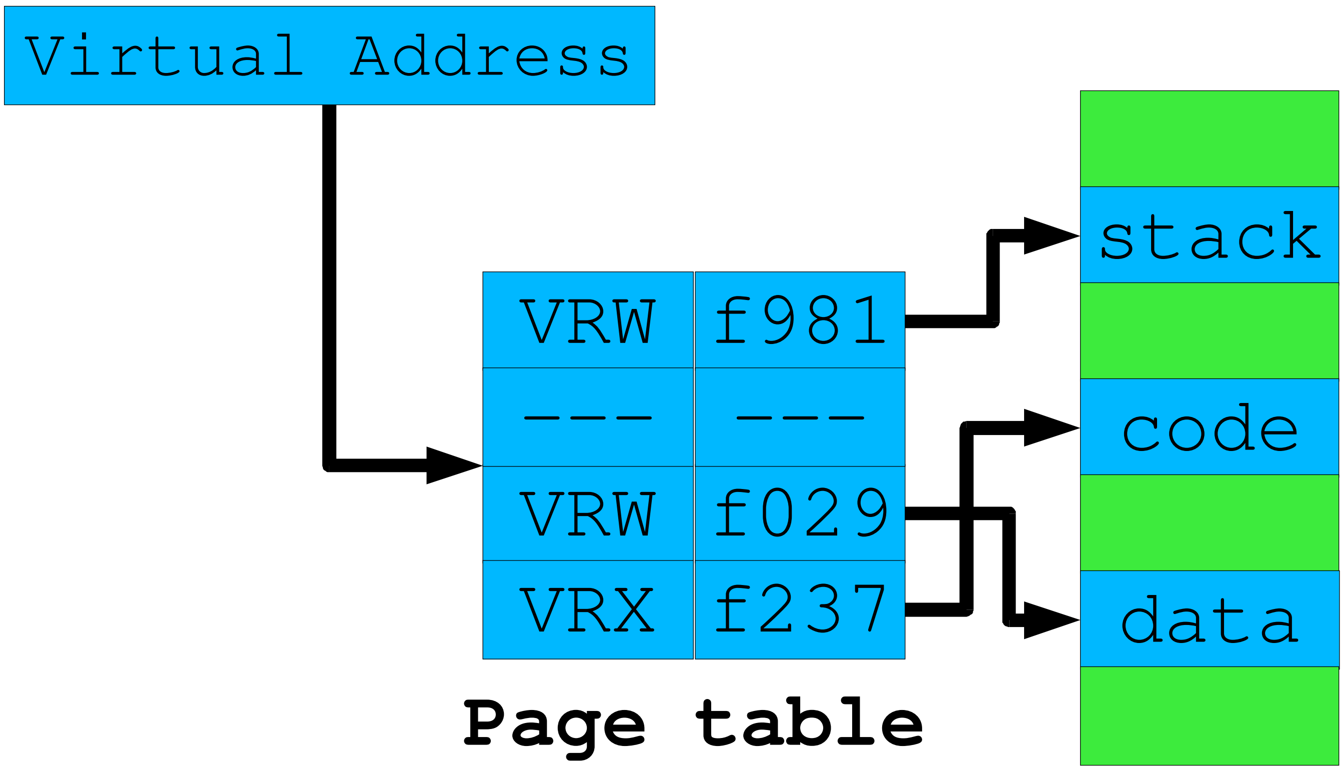
Simulated copy

- Copy page table entries to new process
- Mark PTEs read-only in old & new
- Done! (saving factor: 1024)
  - Simulation is excellent as long as process doesn't write...

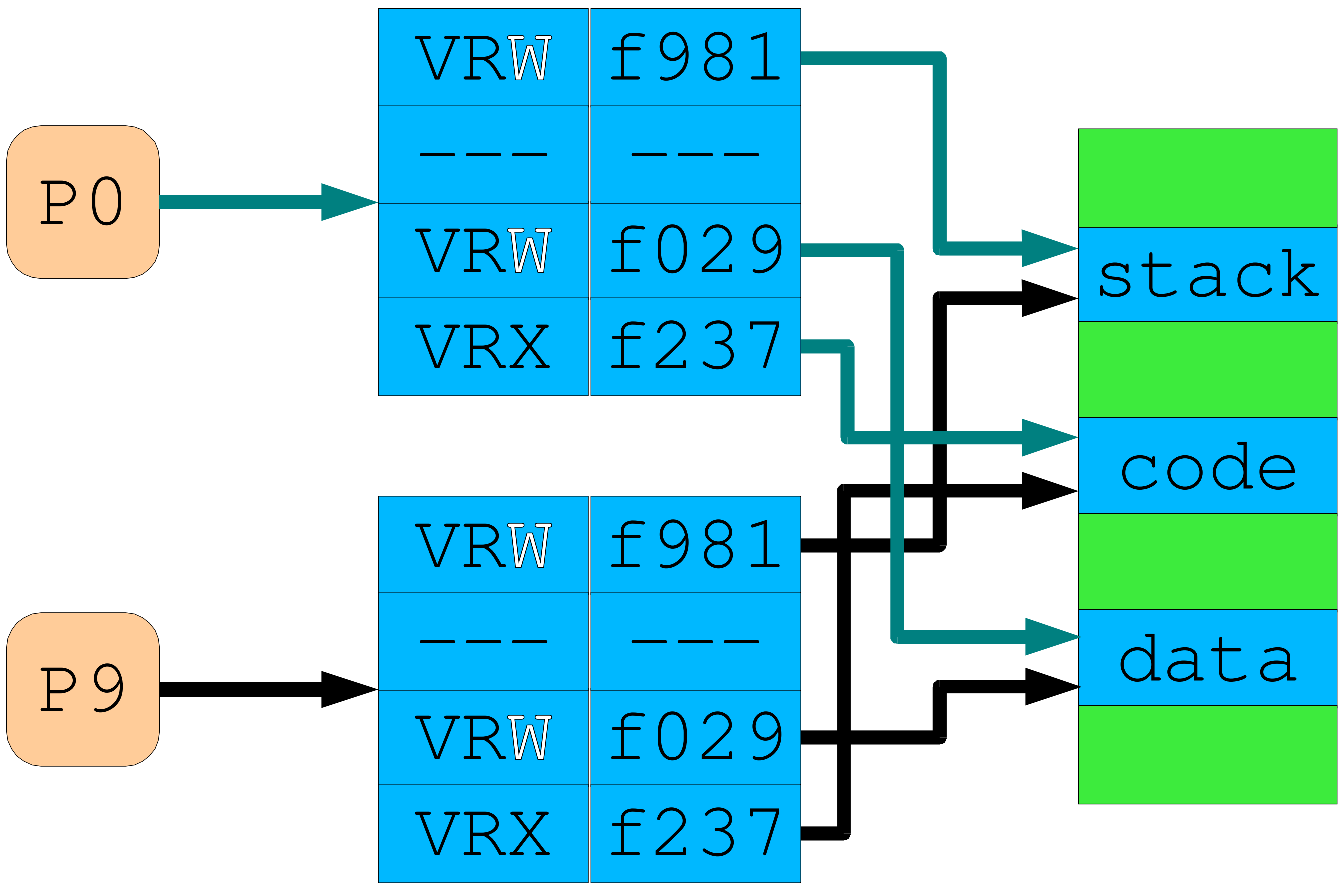
Making it real

- Process writes to page (*Oops! We lied...*)
- Page fault handler responsible
  - Kernel makes a copy of the shared frame
  - Page tables adjusted
    - » ...each process points page to private frame
    - » ...page marked read-write in both PTEs

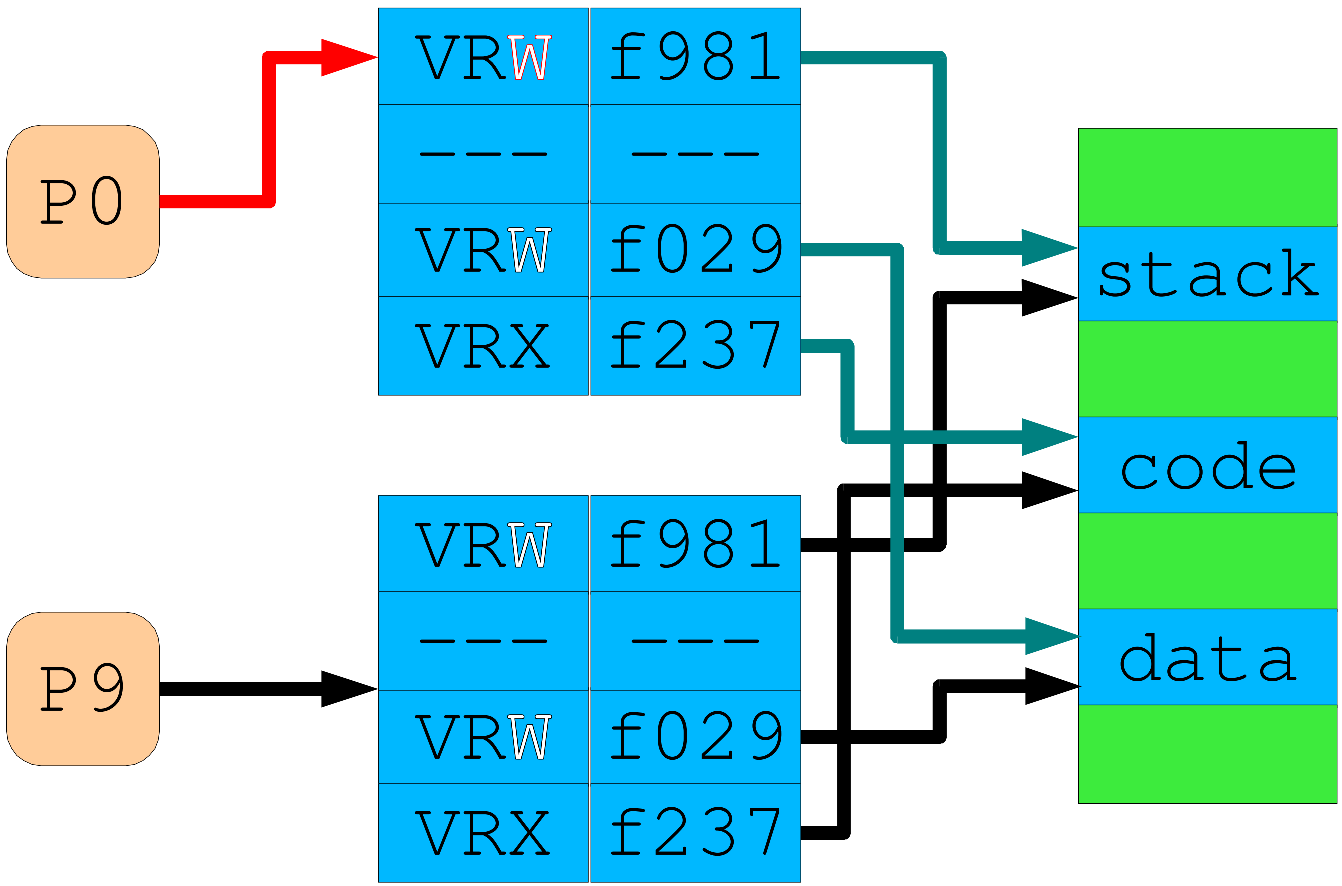
Example Page Table



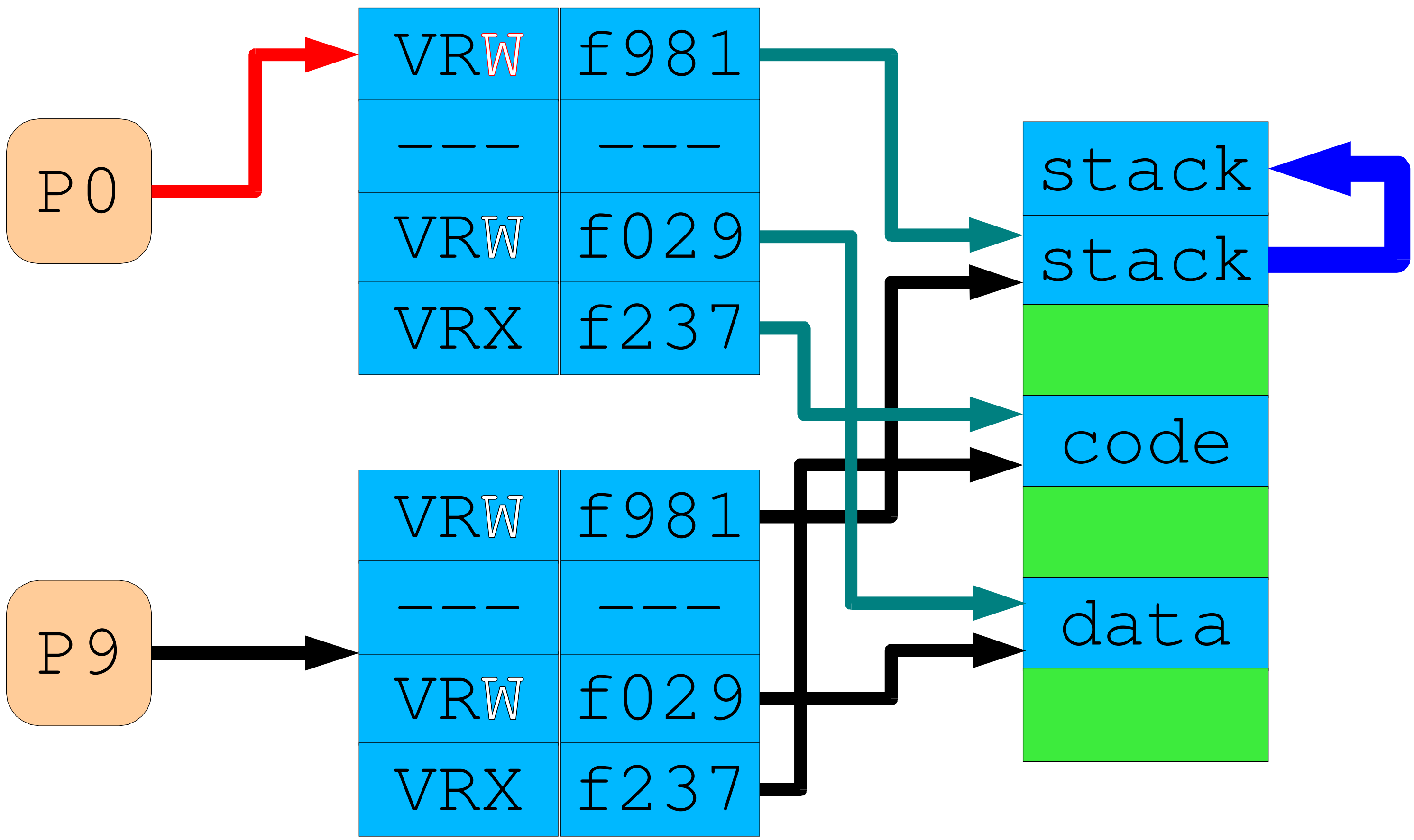
Copy-on-Write of Address Space



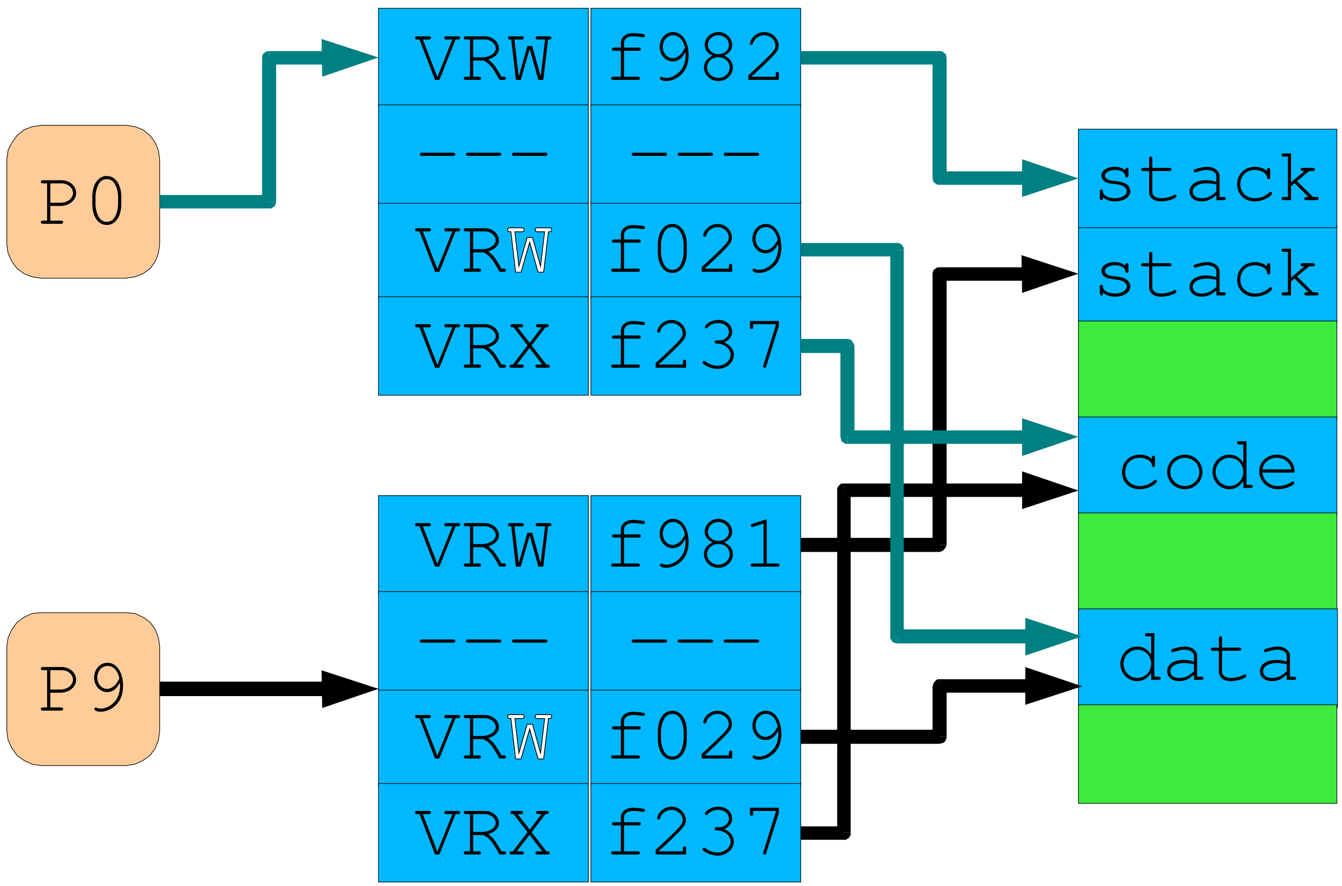
Memory Write ⇒ Permission Fault



Copy Into Blank Frame



Adjust PTE frame pointer, access



Zero Pages

Very special case of copy-on-write

- ZFOD = “Zero-fill on demand”

Many process pages are “blank”

- All of bss
- New heap pages
- New stack pages

Have one *system-wide* all-zero page

- Everybody points to it
- Logically read-write, physically read-only
- Reads are free
- Writes cause page faults & cloning

Memory-Mapped Files

Alternative interface to read(),write()

- mmap(addr, len, prot, flags, fd, offset)
- new memory region presents file contents
- write-back policy typically unspecified
  - unless you msync()...

Benefits

- Avoid serializing pointer-based data structures
- Reads and writes may be much cheaper
  - Look, Ma, no syscalls!

Memory-Mapped Files

Implementation

- Memory region remembers mmap() parameters
- Page faults trigger read() calls
- Pages stored back via write() to file

Shared memory

- Two processes mmap() “the same way”
- Point to same memory region

Summary

Process address space

- Logical: list of regions
- Hardware: list of pages

Fault handler is *complicated*

- Page-in, copy-on-write, zero-fill, ...

Understand definition & use of

- Dirty bit
- Reference bit