15-410

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

Virtual Memory #2
Mar. 1, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Checkpoint 1

- Wednesday 23:59

Final Exam list posted

- You must notify us of conflicts in a timely fashion

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)

TLB – cache of virtual ⇒ physical mappings

Software-loaded TLB

Swapping

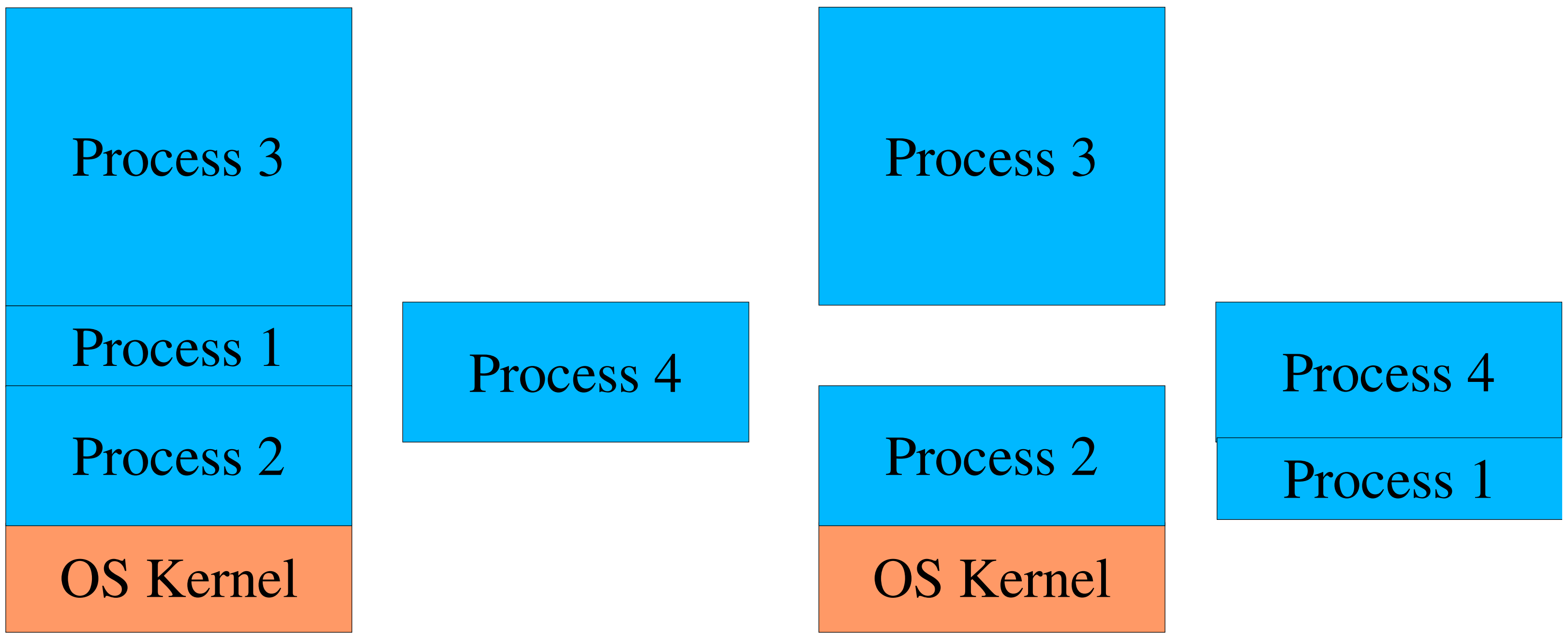
Multiple user processes

- **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

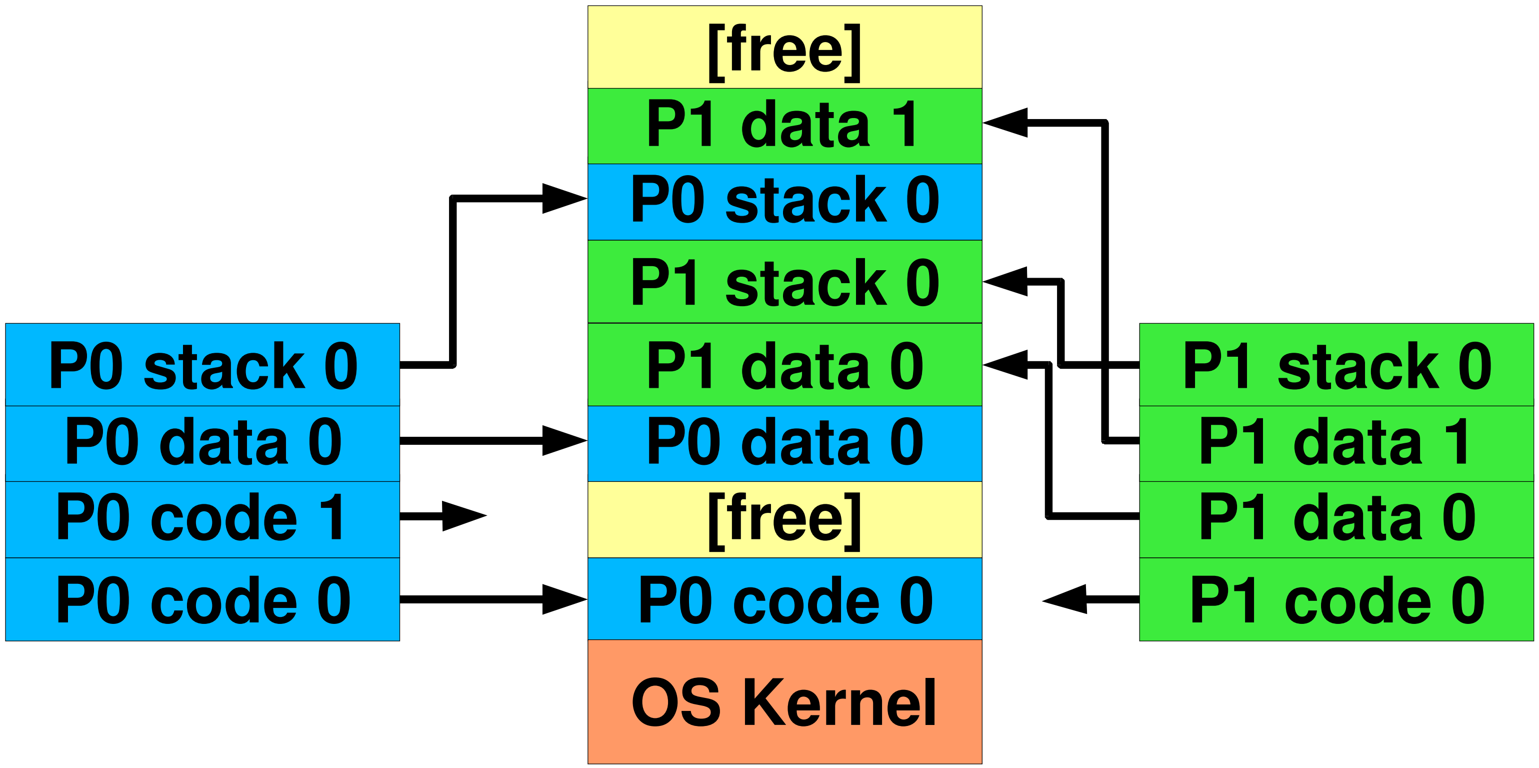
Fragmentation compaction problem

- 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];
```

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

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 - Why?

Address is invalid/illegal – deliver *software exception*

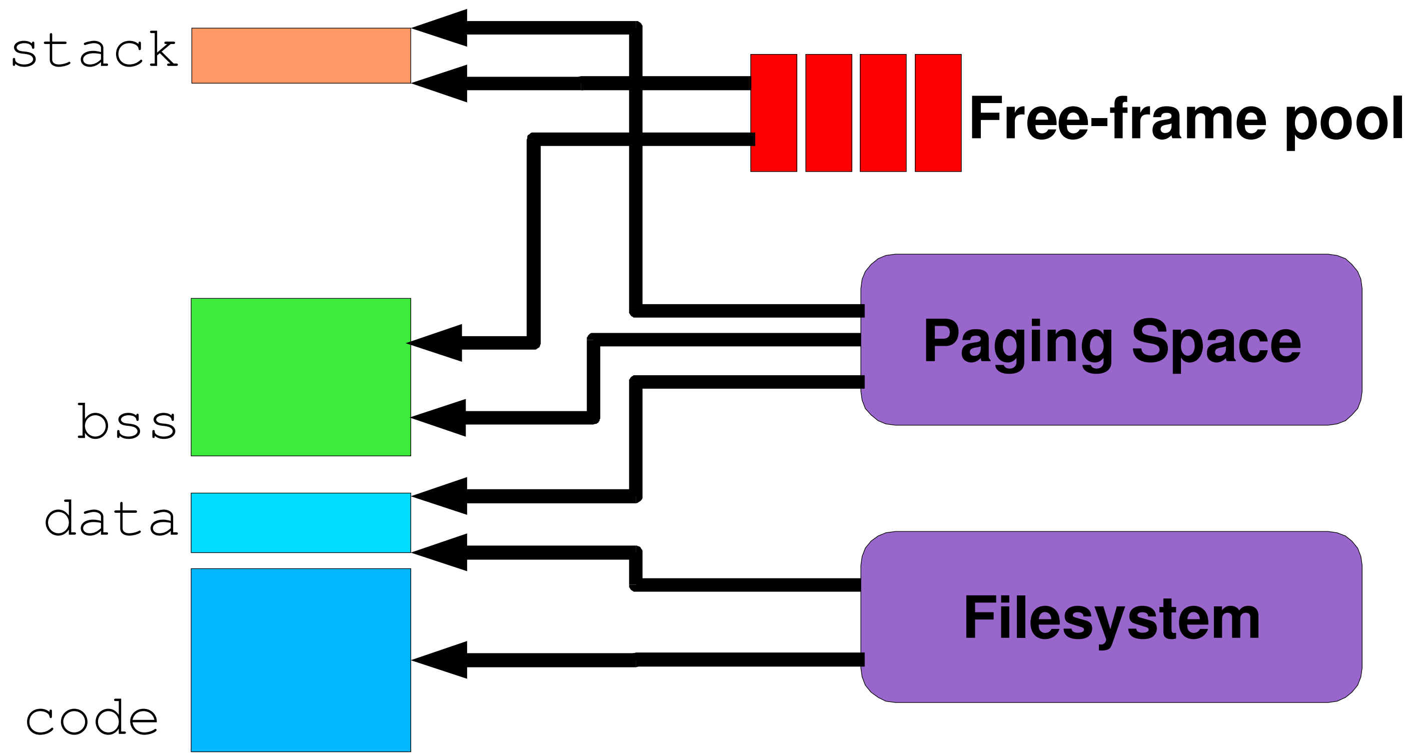
- Unix – SIGSEGV
- Mach – deliver message to thread's exception port
- 15-410 – kill thread

Process is growing stack – give it a new frame

“Cache misses” - fetch from disk

- Where?

Satisfying Page Faults



Page fault story - 1

Process issues memory reference

- TLB: miss (right?)
- PT: “not present”

Trap to OS kernel!

- Dump trap frame
- Transfer via “page fault” interrupt descriptor
- Run trap handler

Page fault story – 2

Classify fault address: legal/illegal

Code/rodata region of executable?

- Determine which sector of executable file
- Launch read() into a blank frame

Previously resident, paged out

- “somewhere on the paging partition”
- Queue disk read into a blank frame

First use of bss/stack page

- Allocate a zero frame, 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 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

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

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

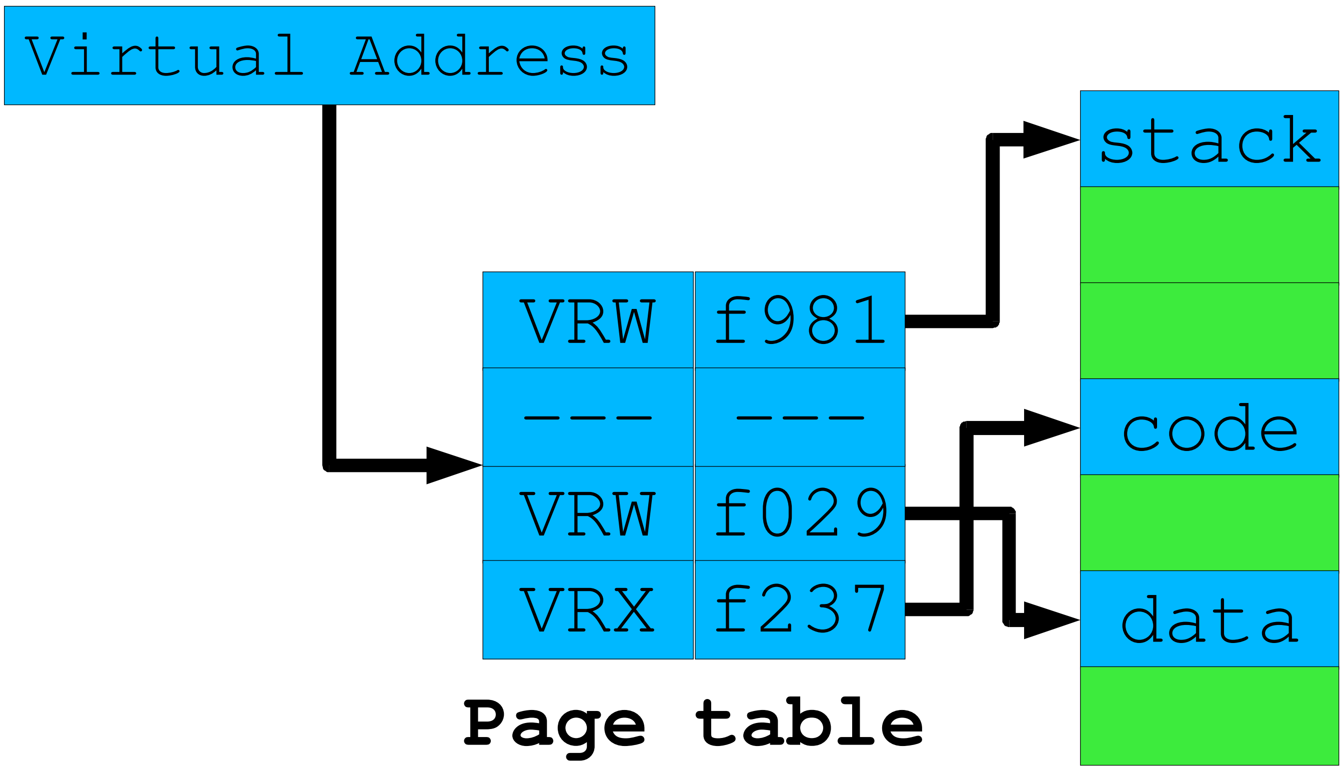
Simulated copy

- Copy page table entries to new process
- Mark PTEs read-only in old & new
- Done! (saving factor: 1024)

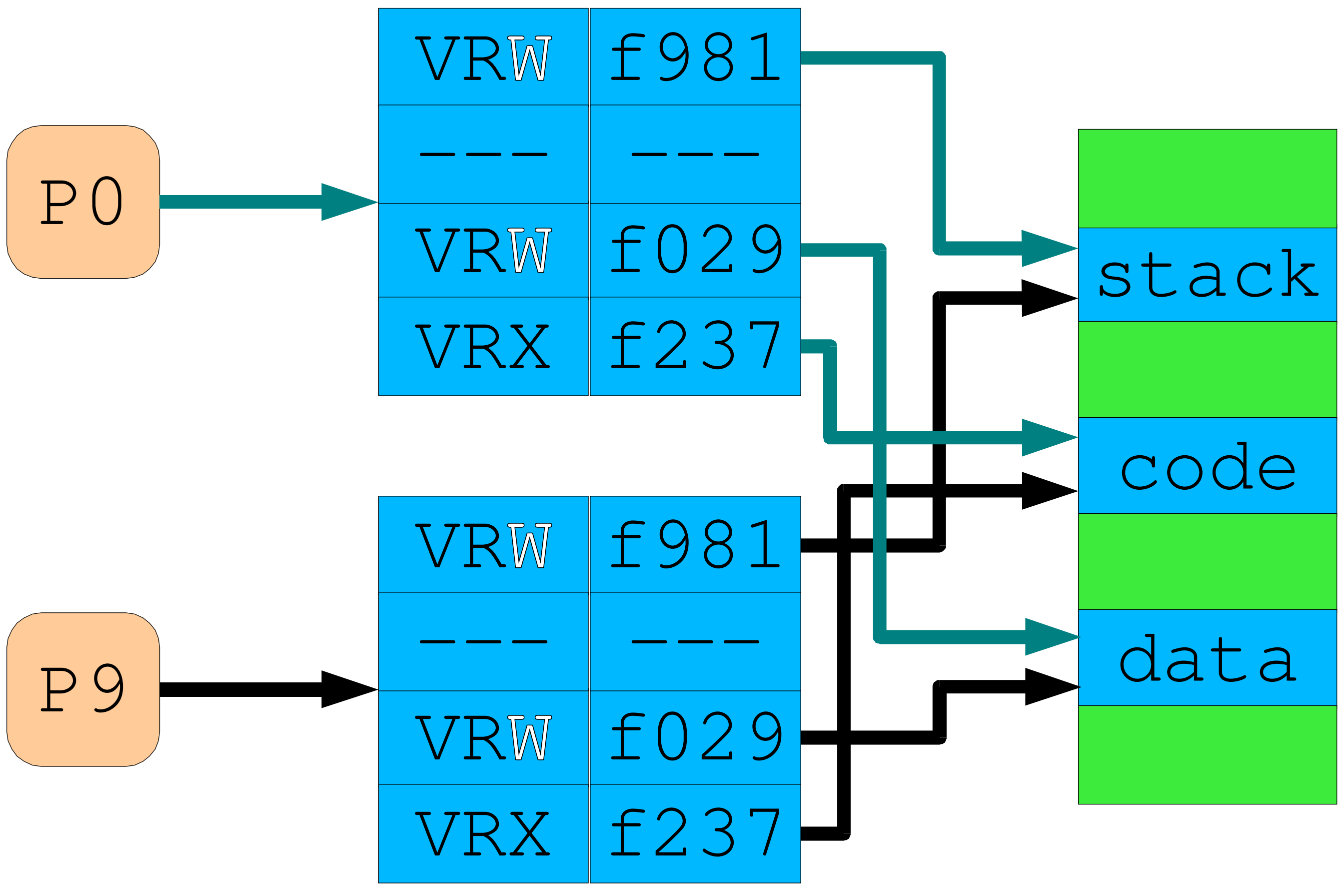
Making it real

- Process writes to page (*oops!*)
- Page fault handler responsible
  - Copy page into empty frame
  - Mark read-write in both PTEs

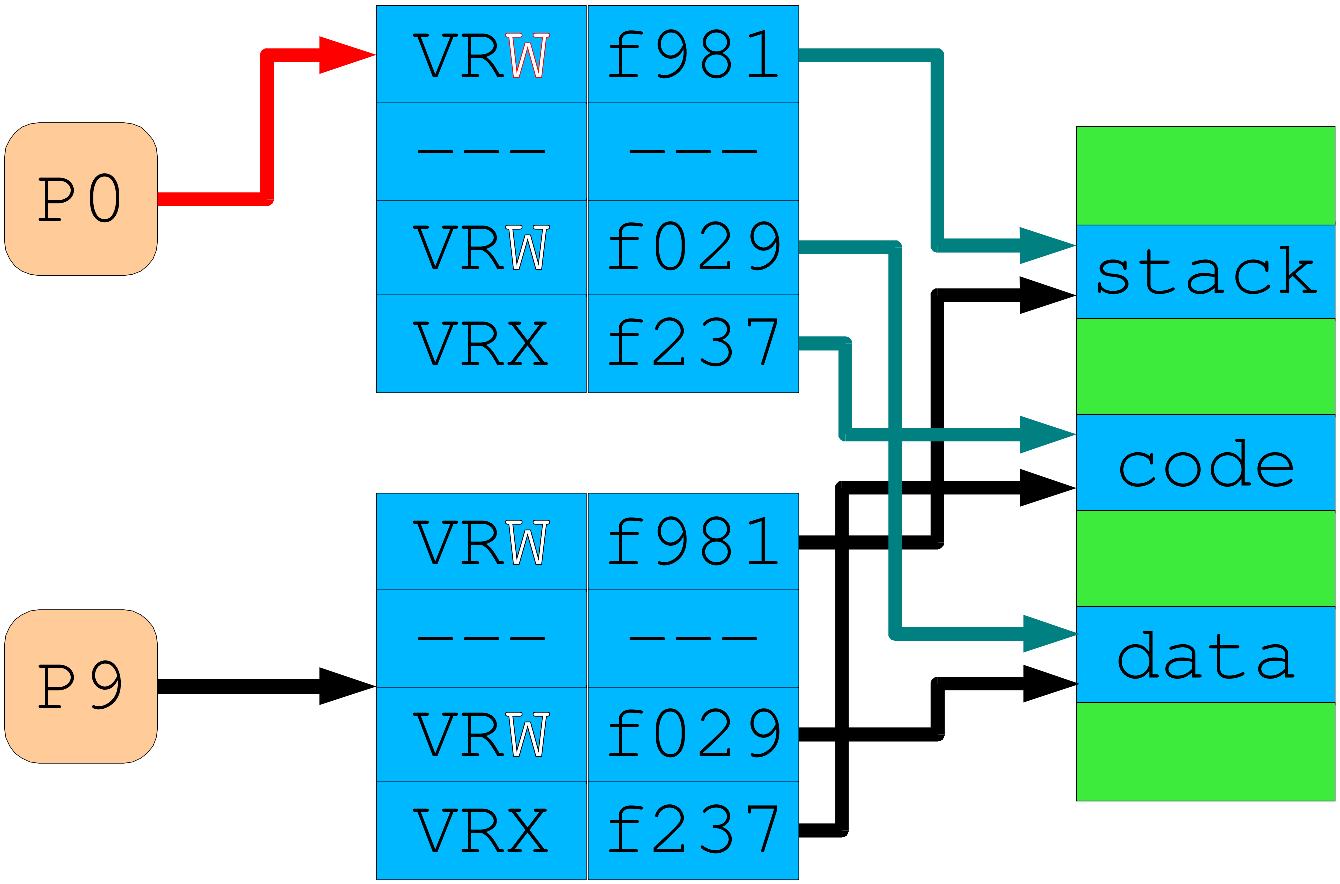
Example Page Table



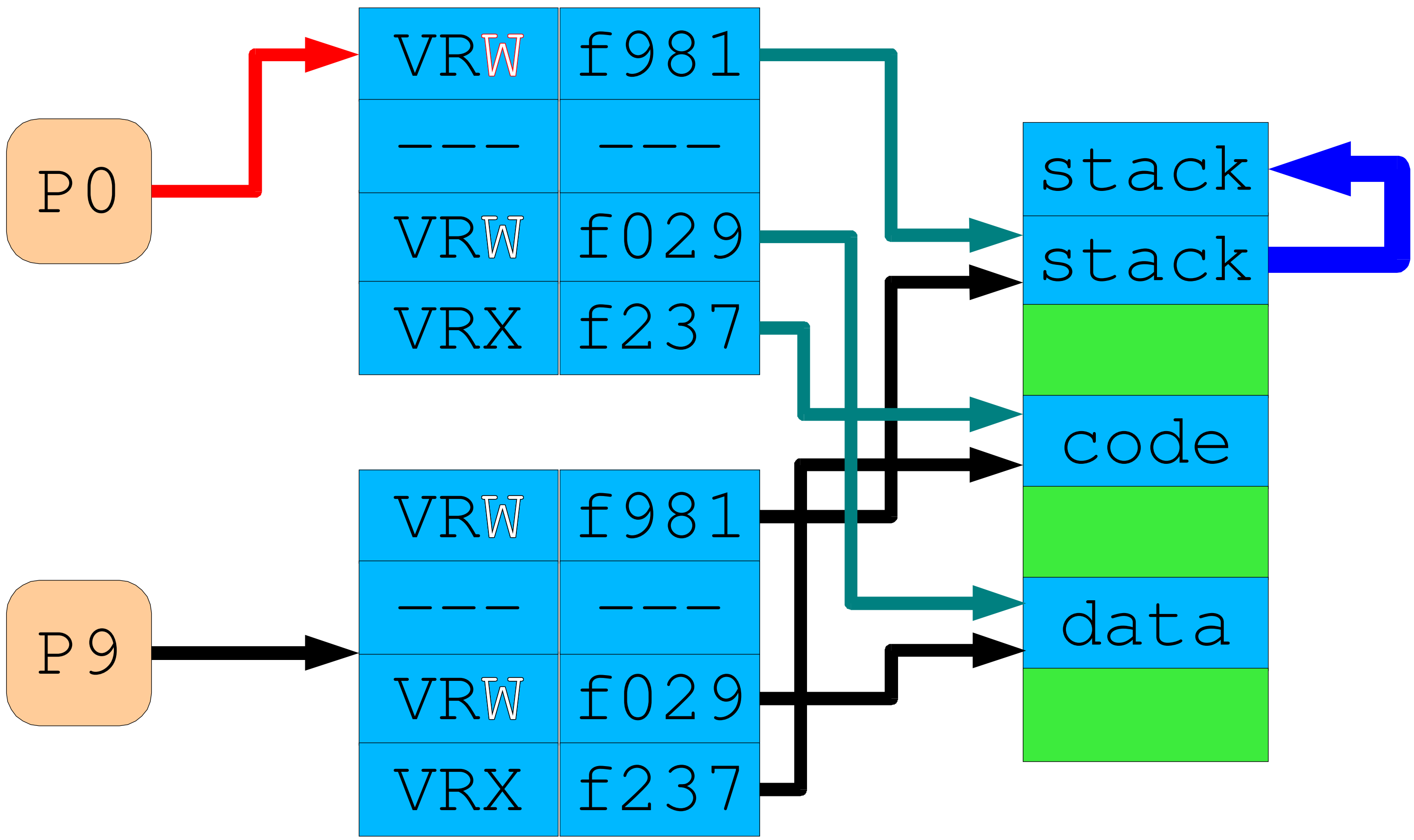
Copy-on-Write of Address Space



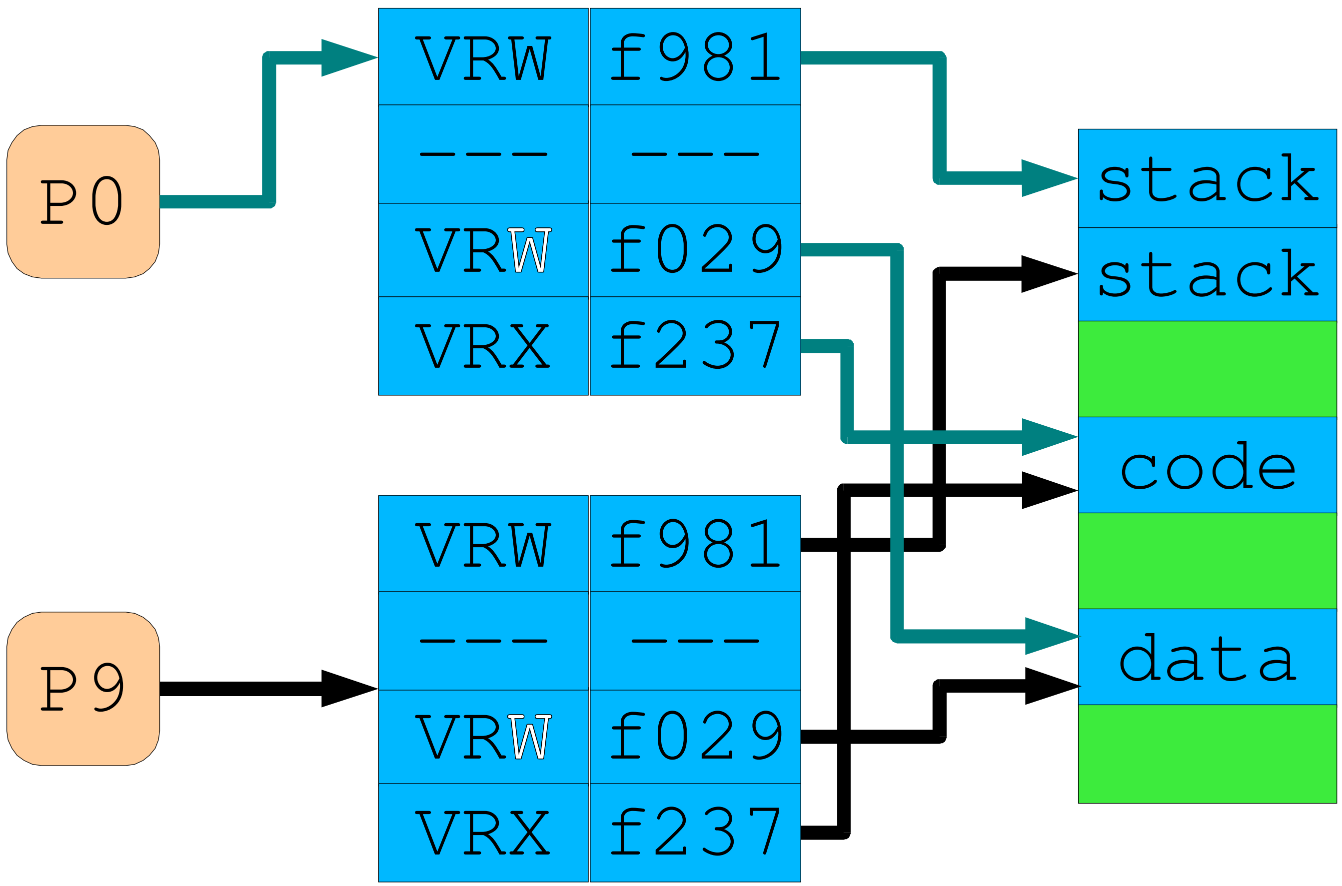
Memory Write ⇒ Permission Fault



Copy Into Blank Frame



Adjust PTE frame pointer, access



Zero pages

Very special case of copy-on-write

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

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

Page Replacement/Page Eviction

Process always want *more* memory frames

- Explicit deallocation is rare
- Page faults are implicit allocations

System inevitably runs out of frames

Solution

- Pick a frame, store contents to disk
- Transfer ownership to new process
- Service fault using this frame

Pick a Frame

Two-level approach

- Determine # frames each process "deserves"
- "Process" chooses which frame is least-valuable

System-wide approach

- Determine globally-least-useful frame

Store Contents to Disk

Where does it belong?

- Allocate backing store for each page
  - What if we run out?

Must we *really* store it?

- Read-only code/data: no!
  - Can re-fetch from executable
  - Saves paging space & disk-write delay
  - File system read() may be slower than paging-disk read
- Not modified since last page-in: no!
  - Hardware typically provides "page-dirty" bit in PTE

Page Eviction Policies

Don't try these at home

- FIFO
- Optimal
- LRU

Practical

- LRU approximation

FIFO Page Replacement

Concept

- Page queue
- Page added to queue when first allocated
- Always evict oldest page (head of queue)

Evaluation

- Cheap
- Stupid
    - May evict old unused startup-code page
    - But *guaranteed* to evict process's favorite page too!

Optimal Page Replacement

Concept

- Evict whichever page will be referenced ***latest***
  - Buy the most time until next page fault

Evaluation

- Requires perfect prediction of program execution
- Impossible to implement

So?

- Used as upper bound in simulation studies

LRU Page Replacement

Concept

- Evict least-recently-used page
- "Past performance *may* not predict future results"

Evaluation

- Would work well
- LRU is computable without fortune teller
- Bookkeeping *very* expensive
  - Hardware must sequence-number every page reference!

Approximating LRU

Hybrid hardware/software approach

- 1 reference bit per page table entry
- OS sets reference = 0 for all pages
- Hardware sets reference=1 when PTE is used
- OS periodically scans
  - reference == 1 ⇒ "recently used"

"Second-chance" (aka "clock") algorithm

- Use stupid FIFO to choose victim pages
- Skip victims with reference == 1 (somewhat-recently used)

Clock Algorithm

```
static int nextpage = 0;
boolean reference[NPAGES];

int choose_victim() {
  while (reference[nextpage])
    reference[nextpage] = false;
    nextpage = (nextpage+1) % NPAGES;
  return(nextpage);
}
```

Page Buffering

Problem

- Don't want to evict pages only when there is a fault
- Must wait for disk write before launching disk read

"Assume a blank page..."

- Page fault handler can be fast

"page-out daemon"

- Scan system for dirty pages
  - Write to disk
  - Clear dirty bit
  - Page can be instantly declared blank later

Frame Allocation

How many frames should a process have?

Minimum

- Examine worst-case instruction
  - Can multi-byte instruction cross page boundary?
  - Can memory parameter cross page boundary?
  - How many memory parameters?
  - Indirect pointers?

Frame Allocation

Equal

- Every process gets same # frames
  - "Fair"
  - Probably wasteful

Proportional

- Larger processes get more frames
  - Probably the right approach
  - Encourages greediness

Thrashing

Problem

- Process *needs* N pages
- OS provides N-1, N/2, etc.

Result

- Every page OS evicts generates “immediate” fault
- More time spent paging than executing
- Paging disk constantly busy
  - Denial of “paging service” to other processes

Working-Set Model

Approach

- Determine necessary # pages
- If unavailable, start swapping

How to measure?

- Periodically scan process reference bits
- Combine multiple scans (see text)

Evaluation

- Expensive

Page-Fault Frequency

Approach

- Thrashing == “excessive” paging
- Adjust per-process frame quotas to balance fault rates
  - Fault rate “too low”: reduce frame quota
  - Fault rate “too high”: increase frame quota

What if quota increase doesn't help?

- Start swapping

Program optimizations

Locality depends on data structures

- Arrays encourage sequential accesss
- Random pointer data structures scatter references

Compiler & linker can help

- Don't split a routine across two pages
- Place helper functions on same page as main routine

Effects can be *dramatic*

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