15-410

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

"...The only winning move is not to play..."

Virtual Memory #2.5

Oct. 12, 2007

Dave Eckhardt

Roger Dannenberg

Synchronization

Project 3 Checkpoint 1 –Monday in class!

- Meet downstairs in 5th floor cluster
  - Probably Wean 5207
  - Look for e-mail on when your group should arrive
- Demo a program of our choice
- Talk to us about your progress
- Attendance is mandatory (you have no conflict...)

Experimental P2 survey

Checkpoint 2 date

Outline

Last time

- Meaning of PTE flags
- Partial memory residence (demand paging) in action
- The task of the page fault handler

Today

- Big speed hacks
- The mysterious TLB
- Sharing memory regions & files

Upcoming

- Page replacement policies

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...

Copy-on-Write

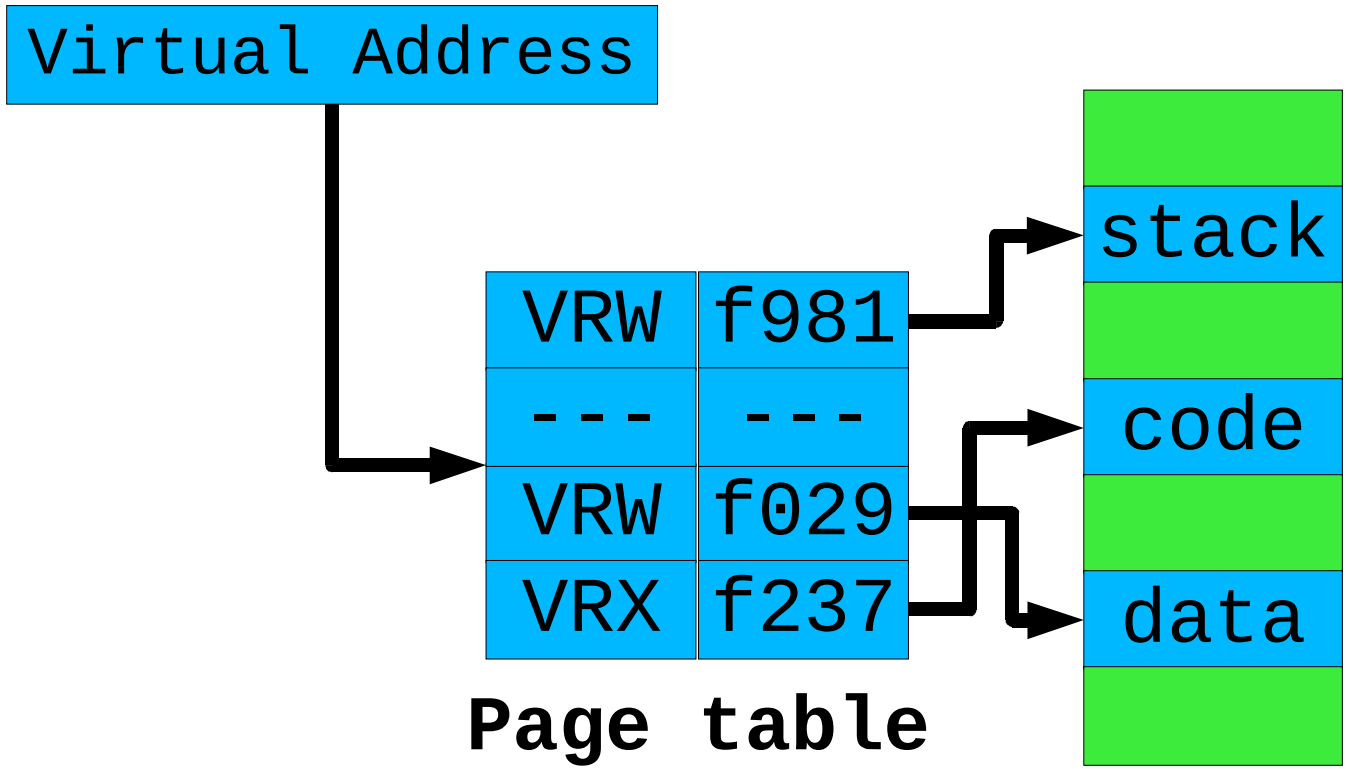
Simulated copy

- 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...

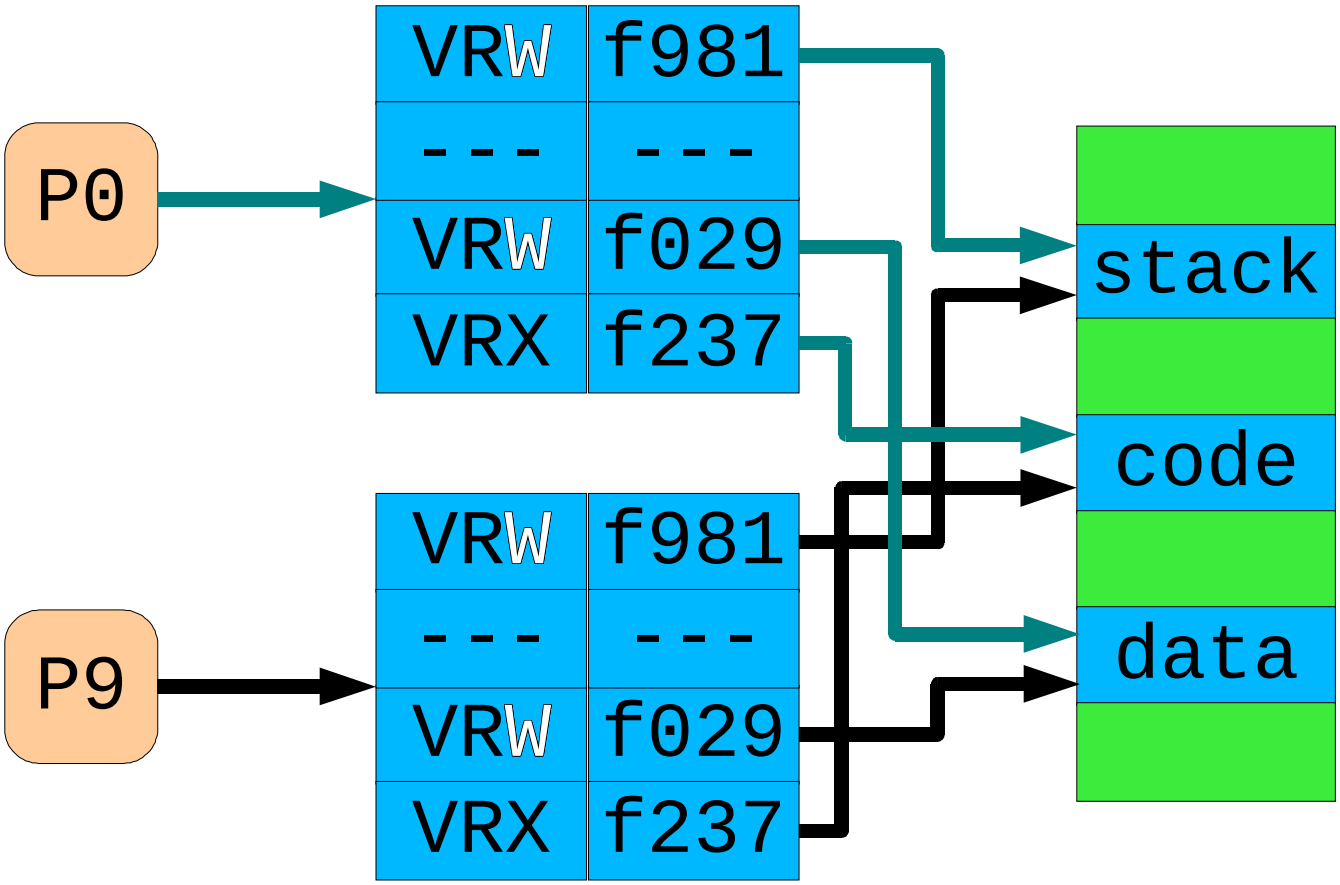
Making it real

- 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

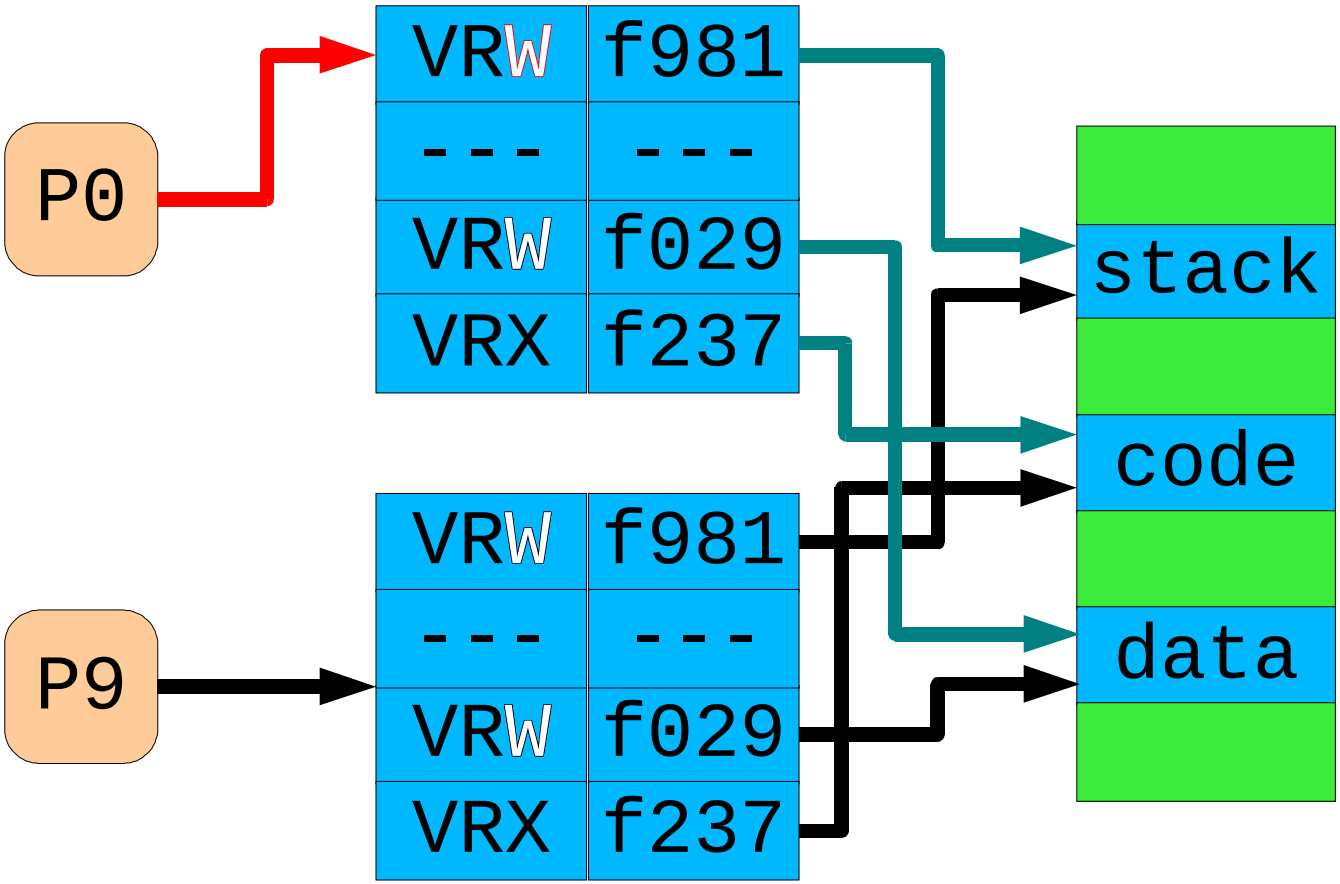
Example Page Table



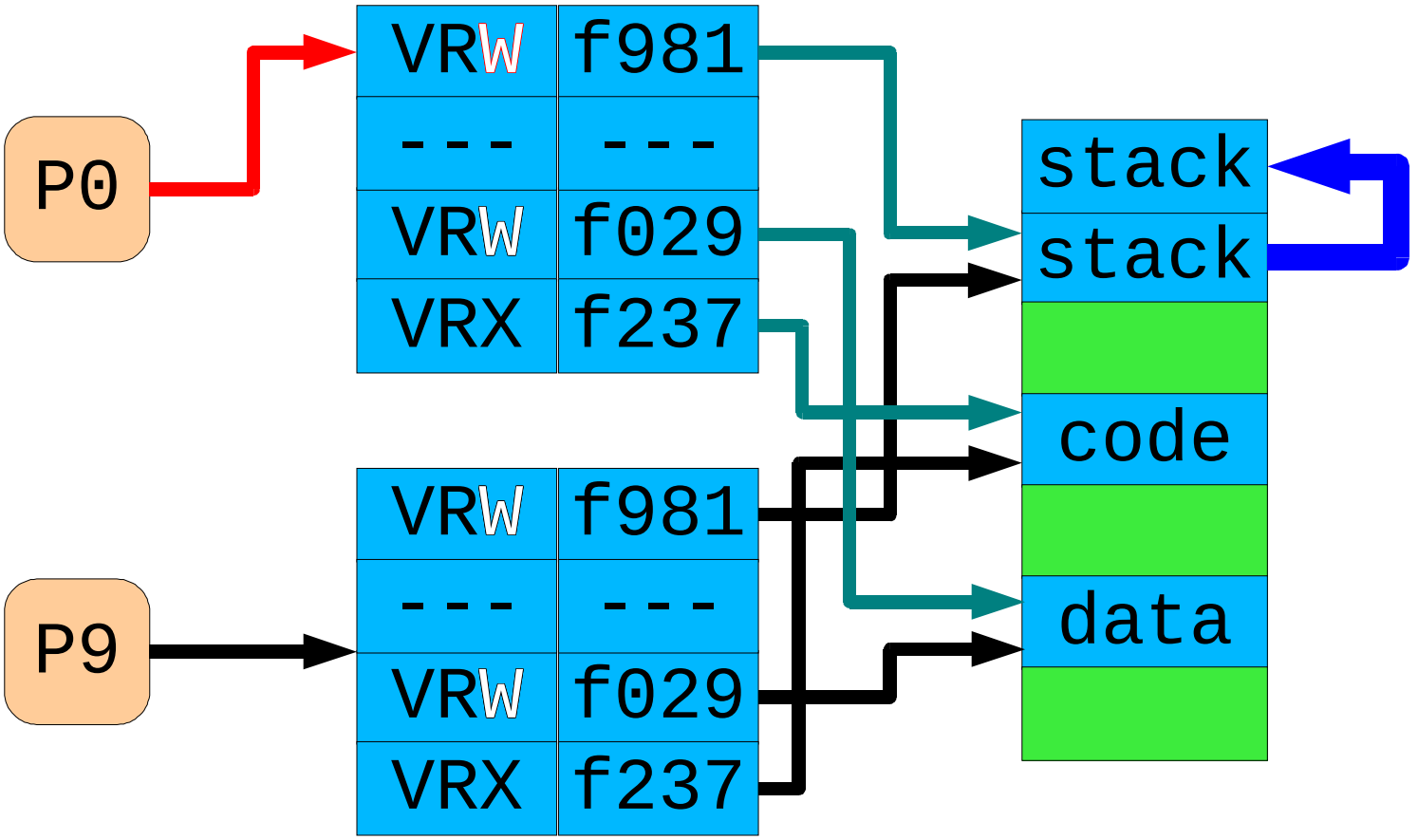
Copy-on-Write of Address Space



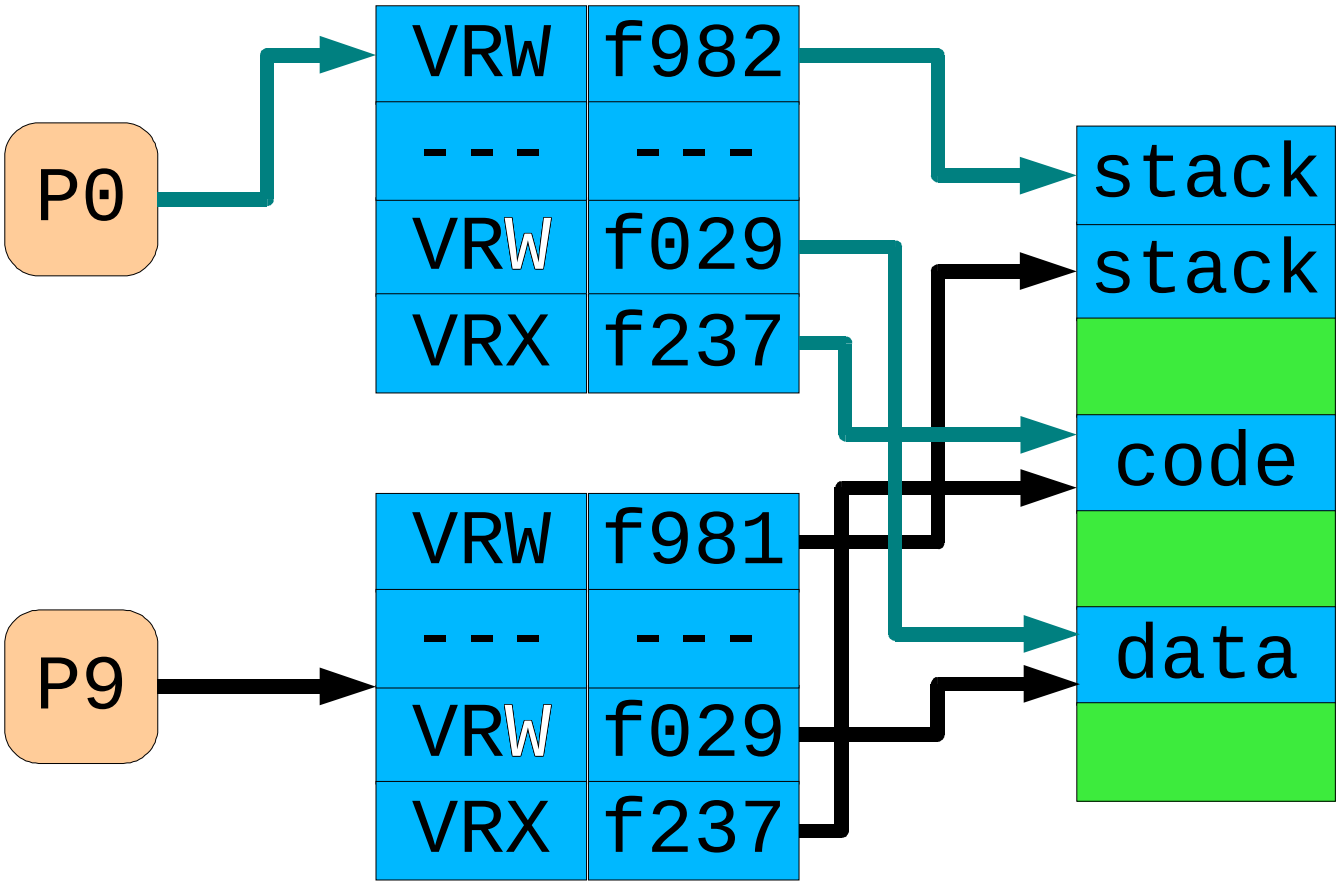
Memory Write ⇒ Permission Fault



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 frame

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

Double Trouble? *Triple* 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***

Can be worse than that...

- **x86 Page-Directory/Page-Table**
  - ***Three* physical accesses per virtual access!**

Translation Lookaside Buffer (TLB)

Problem

- Cannot afford double/triple memory latency

Observation - "locality of reference"

- Program often accesses "nearby" memory
- Next instruction often on same page as current instruction
- Next byte of string often on same page as current byte
- ("Array good, linked list bad")

Solution

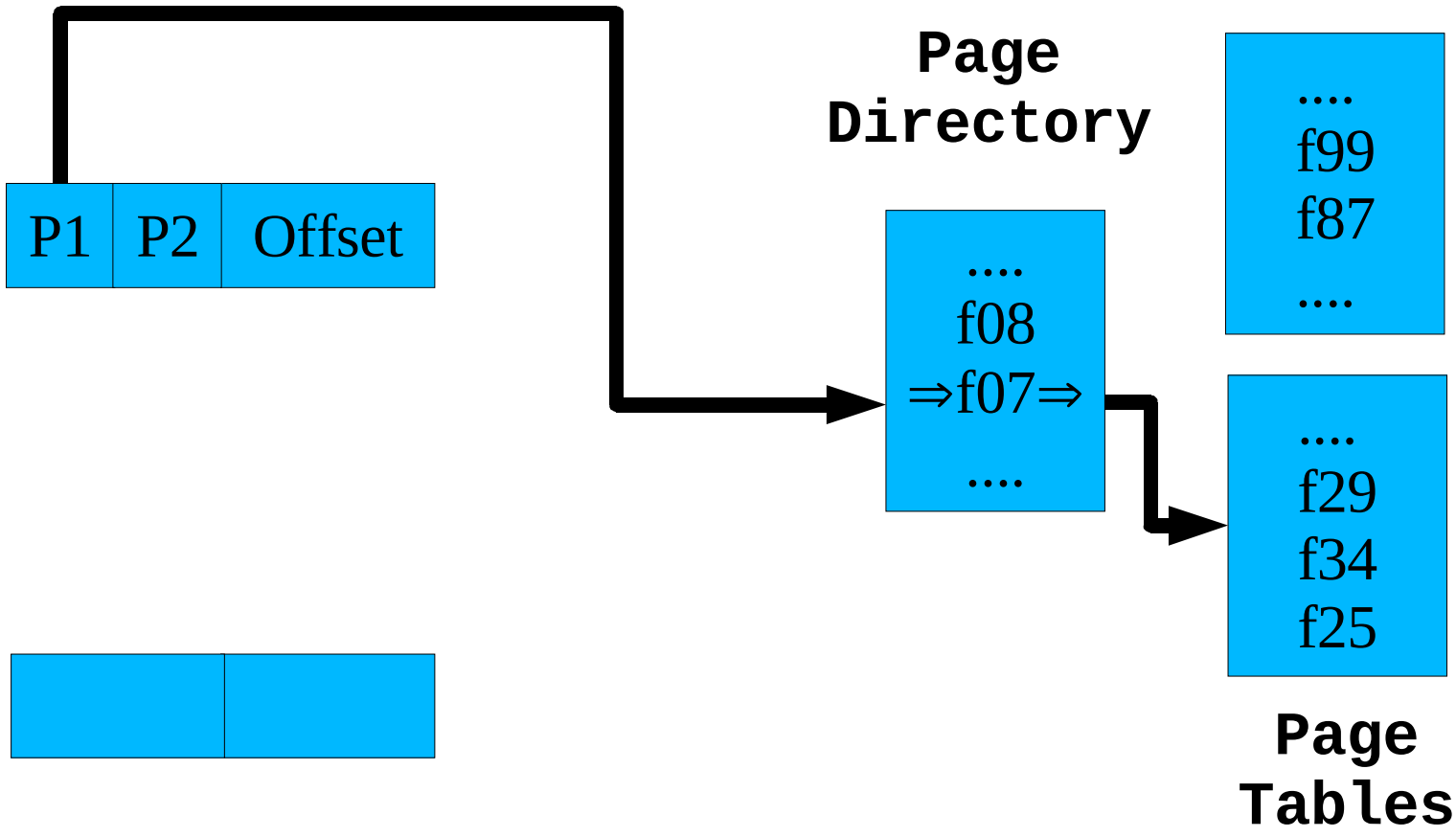
- Page-map hardware caches virtual-to-physical *mappings*
  - Small, fast on-chip memory
  - "Free" in comparison to slow off-chip memory

Simplest Possible TLB

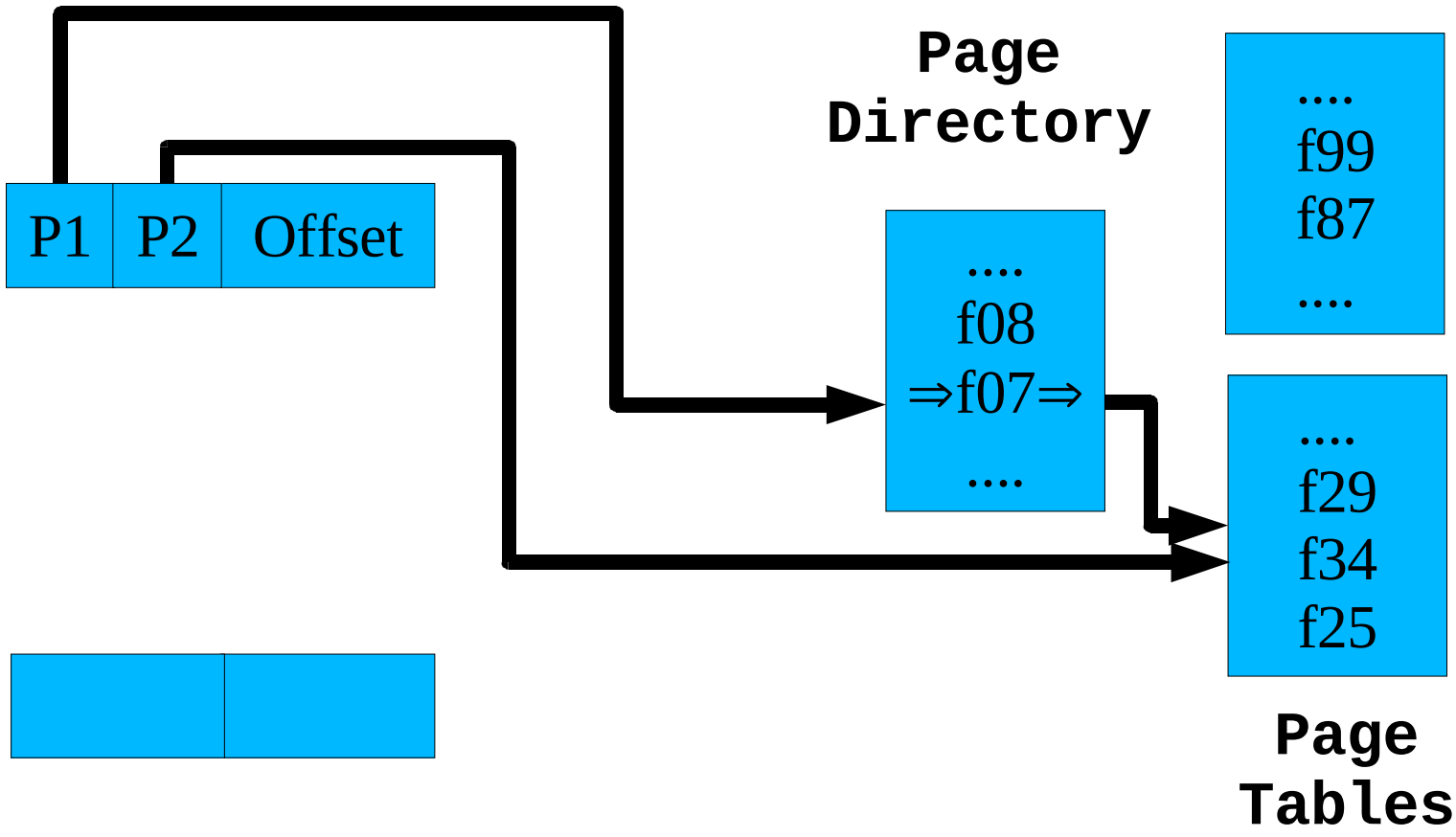
Approach

- Remember the most-recent virtual-to-physical translation
  - (obtained from, e.g., Page Directory + Page Table)
- See if next memory access is to same page
  - If so, skip PD/PT memory traffic; use same frame
  - 3X speedup, cost is two 20-bit registers
    - » "Great work if you can get it"

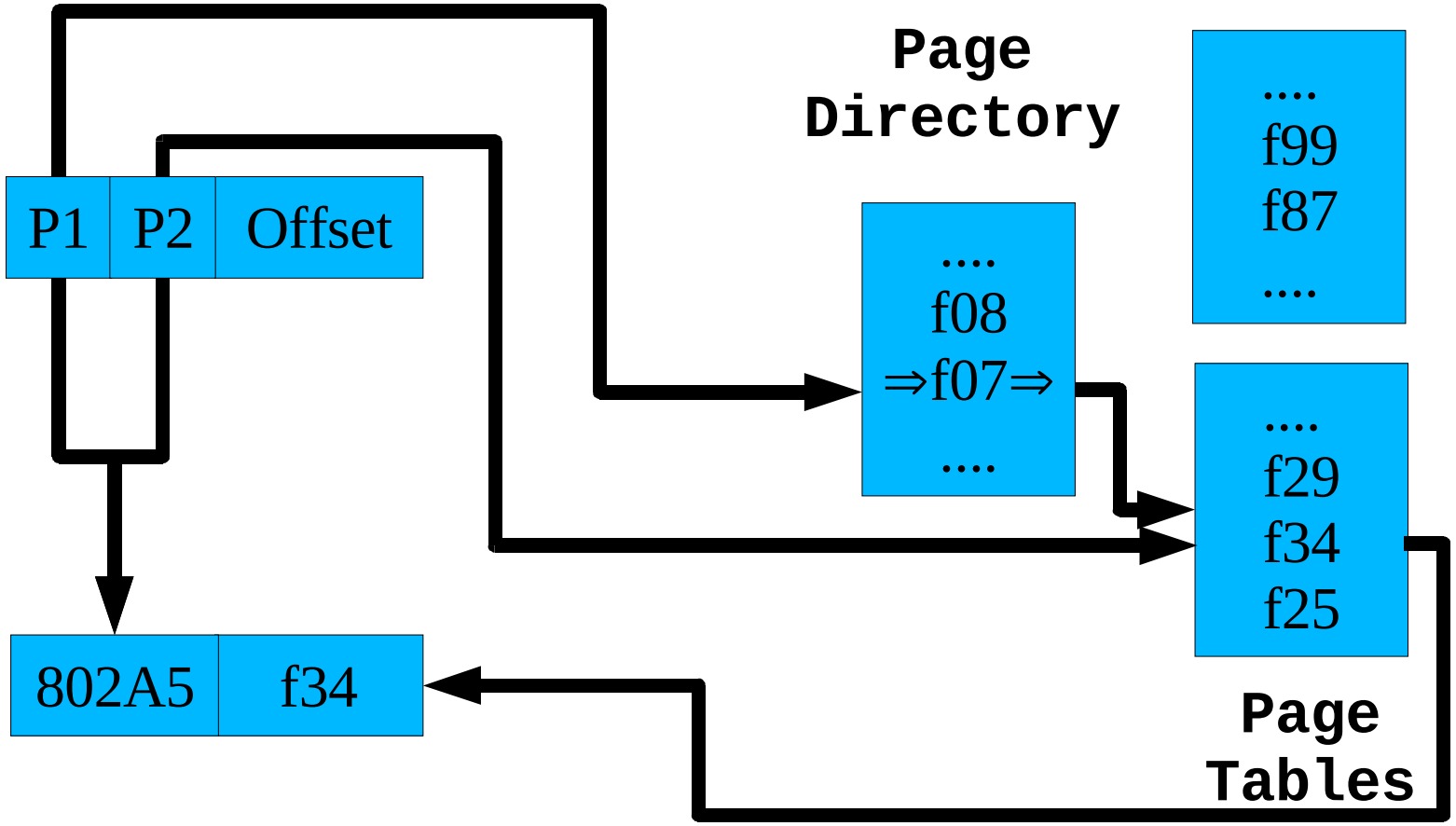
Simplest Possible TLB



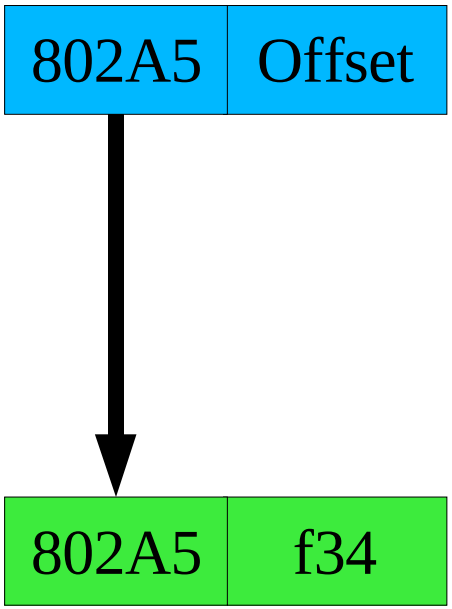
Simplest Possible TLB

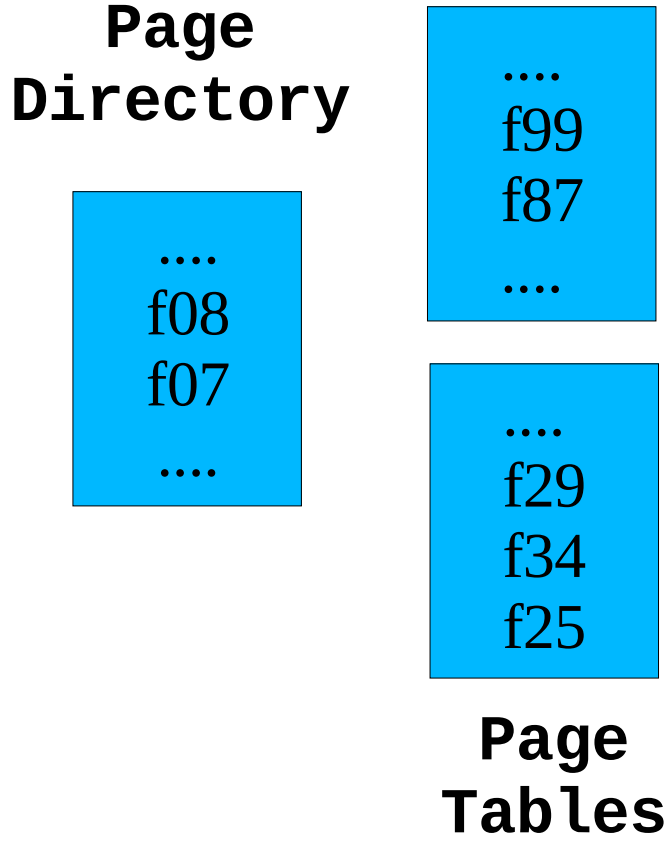


Simplest Possible TLB

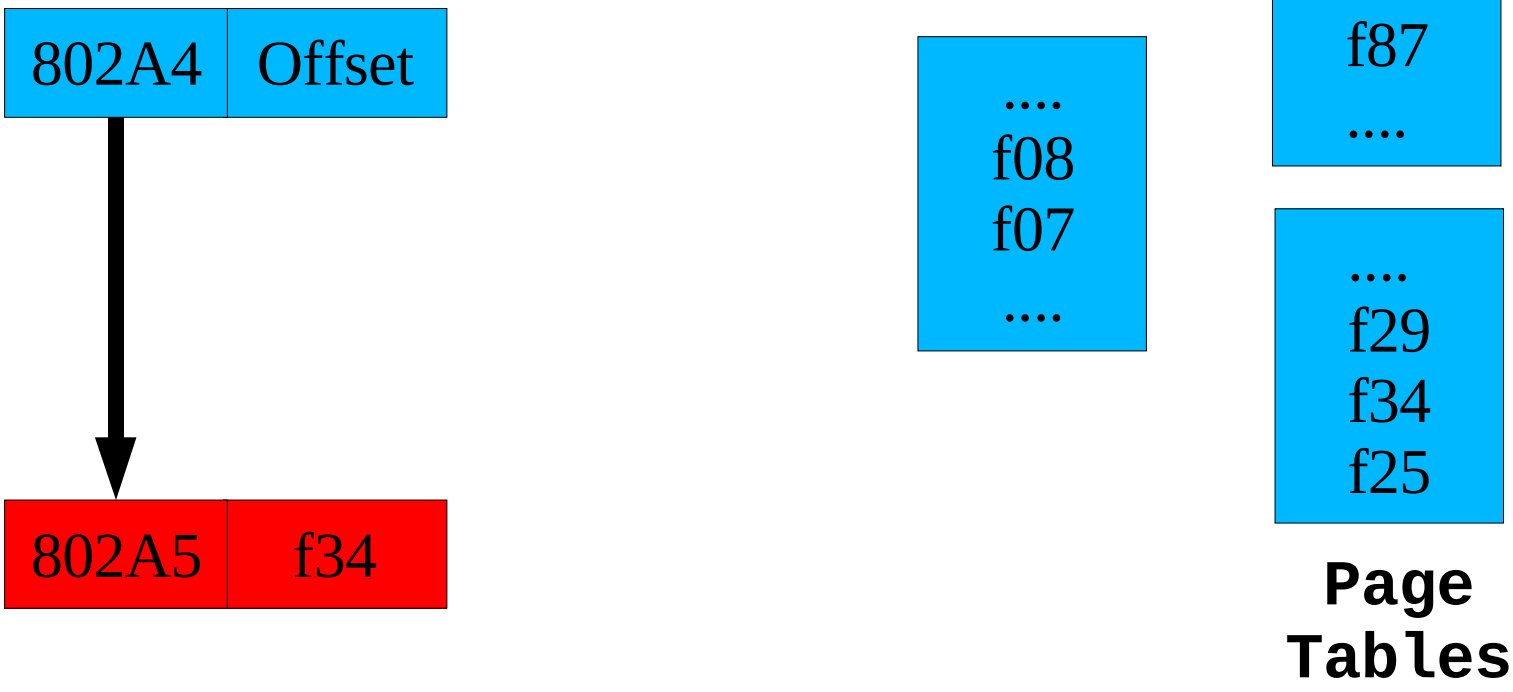


TLB “Hit”

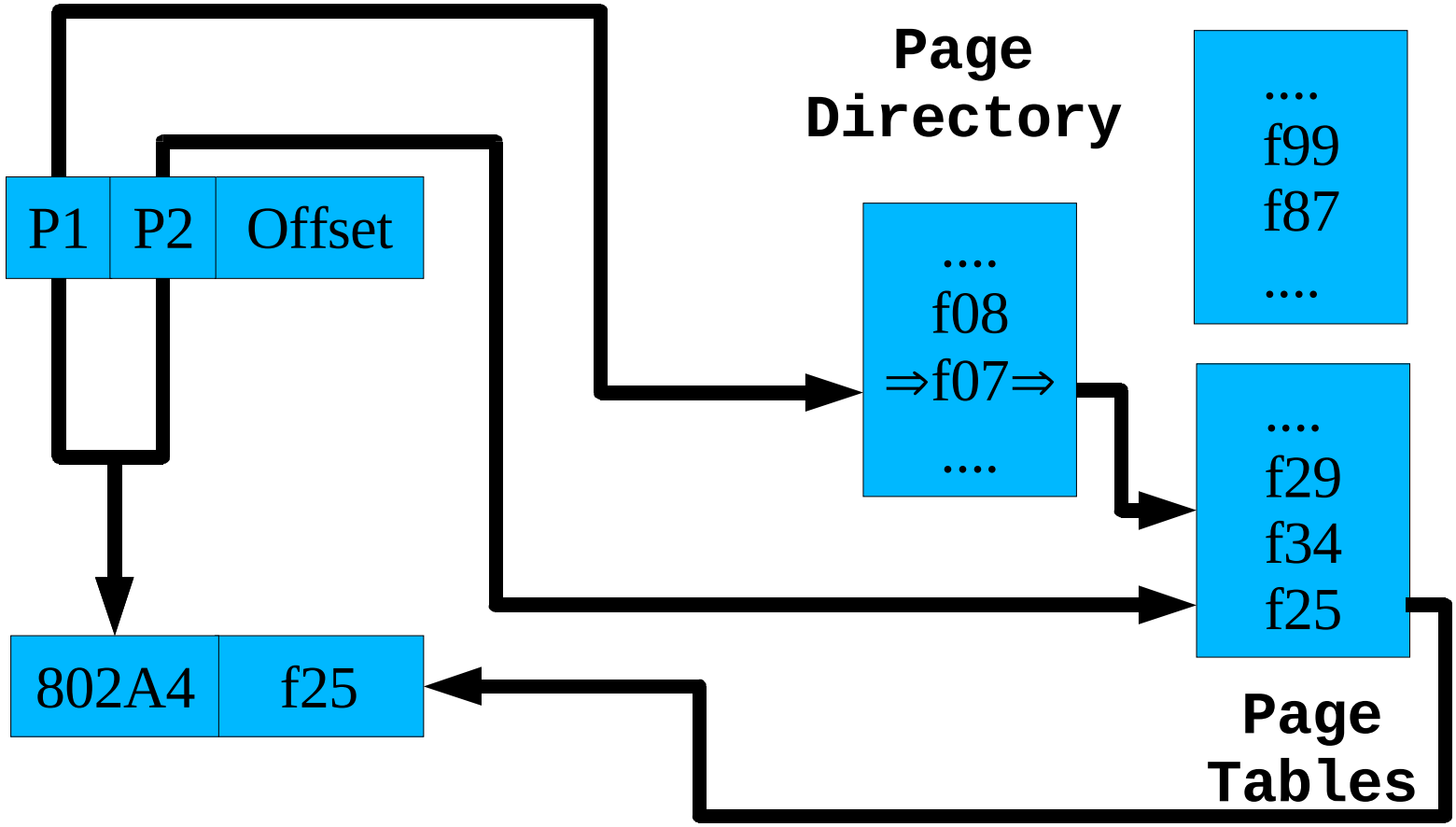

802A5
Offset
802A5
f34


Page Directory
....
f08
f07
....
....
f99
f87
....
....
f29
f34
f25
Page Tables

TLB “Miss”


802A4
Offset
802A5
f34
Page Directory
....
f08
f07
....
....
f99
f87
....
....
f29
f34
f25
Page Tables

TLB “Refill”


Page
Directory
P1
P2
Offset
....
f08
⇒f07⇒
....
....
f99
f87
....
....
f29
f34
f25
802A4
f25
Page
Tables

Simplest Possible TLB

Can you think of a "pathological" instruction?

- **What would it take to "break" a 1-entry TLB?**

How many TLB entries do we need, anyway?

TLB vs. Context Switch

After we've been running a while...

- ...the TLB is “hot” - full of page⇒frame translations

Interrupt!

- Some device is done...
- ...should switch to some other task...
- ...what are the parts of context switch, again?
  - General-purpose registers
  - ...?

TLB vs. Context Switch

After we've been running a while...

- ...the TLB is “hot” - full of page⇒frame translations

Interrupt!

- Some device is done...
- ...should switch to some other task...
- ...what are the parts of context switch, again?
  - General-purpose registers
  - Page Table Base Register (x86 calls it ...?)
  - ...?

TLB vs. Context Switch

After we've been running a while...

- **...the TLB is “hot” - full of page⇒frame translations**

Interrupt!

- **Some device is done...**
- **...should switch to some other task...**
- **...what are the parts of context switch, again?**
  - **General-purpose registers**
  - **Page Table Base Register (x86 calls it ...?)**
  - ***Entire contents of TLB!!***
    - » **(why?)**

x86 TLB Flush

1. Declare new page directory (set %cr3)

- Clears every entry in TLB (whoosh!)
  - Footnote: doesn't clear “global” pages...
    - » Which pages might be “global”?

2. INVLPG instruction

- Invalidates TLB entry of one specific page
- Is that more efficient or less?

x86 Type Theory –Final Version

Instruction ⇒ segment selector

- **[PUSHL specifies selector in %SS]**

Process ⇒ (selector ⇒ (base,limit))

- **[Global,Local Descriptor Tables]**

Segment base, 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?”**

Is there another way?

- **Could we have *no* page tables?**
- **How would the 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 regions, permissions
  - *Lots* of processes share frames of /bin/bash!
- Hardware page-mapping unit imposes its own ideas
- Why impose a semantic middle-man?

Approach

- TLB contains subset 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

- Good idea to lock the TLB-miss handler's TLB entry...
- Great for real-time systems

Further reading

- http://yarchive.net/comp/software_tlb.html

Software TLBs

- PowerPC 603, 400-series (but NOT 7xx/9xx)

TLB vs. Project 3

x86 has a nice, automatic TLB

- Hardware page-mapper fills it for you
- Activating new page directory flushes TLB automatically
- What could be easier?

It's not *totally* automatic

- Something “natural” in your kernel may confuse it...

TLB debugging in Simics

- logical-to-physical (l2p) command
- cpu0_tlb.info, cpu0_tlb.status
  - More bits “trying to tell you something”
- [INVLPG issues with Simics 1. Simics 2, 3 seem ok]

Summary

Big speed hacks

- Copy-on-write, zero-fill on demand

The mysterious TLB

- No longer mysterious

Upcoming

- Sharing memory regions & files
- Page replacement policies