15-410

"...The mysterious TLB..."

Virtual Memory #2
Oct. 1, 2008

Dave Eckhardt

Roger Dannenberg

Synchronization

Reminder: exam conflict mail

- **Please answer promptly**

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

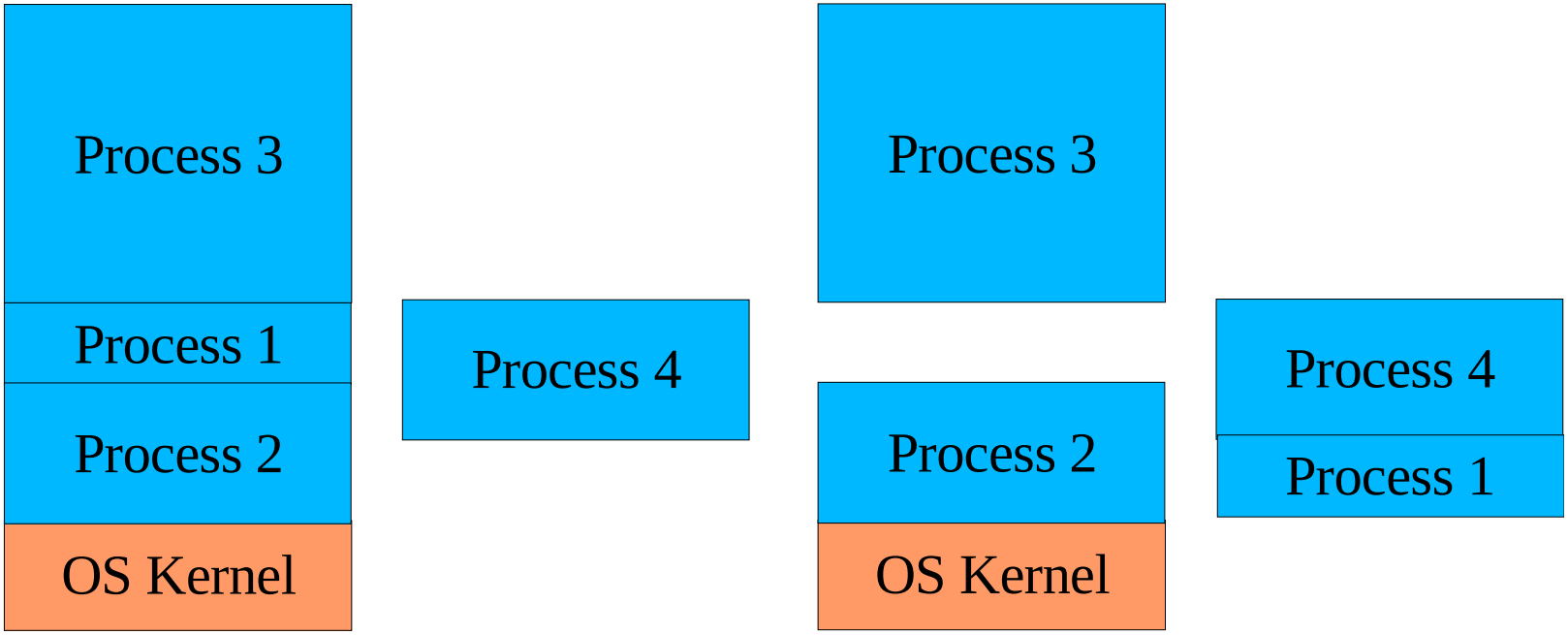
Multiple user processes

- 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

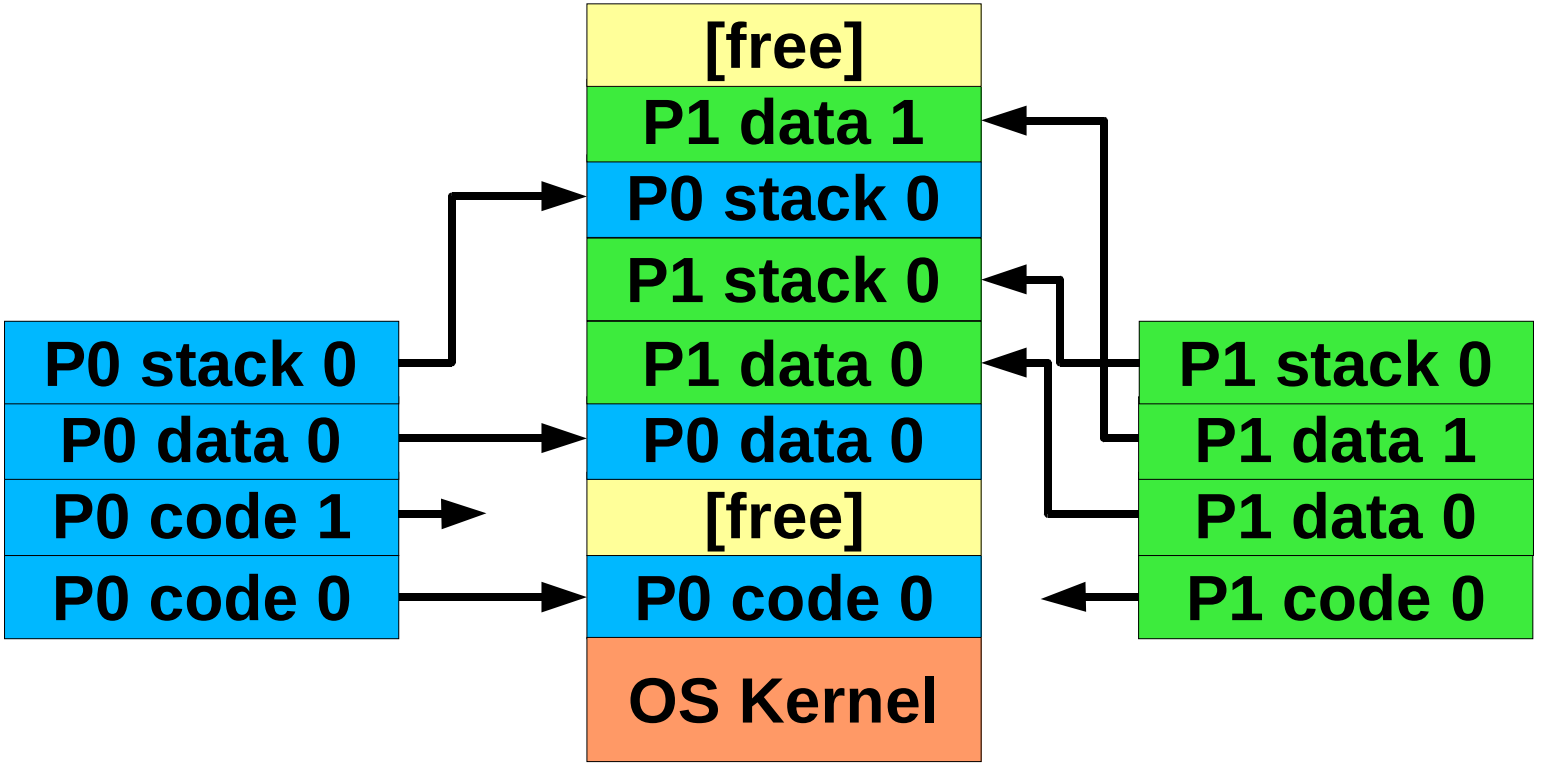
Fragmentation compaction problem

- 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 (later)

Outline

The mysterious TLB

Partial memory residence (demand paging) in action

The task of the page fault handler

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!**
- **x86-64 has a *four-level* page-mapping system**

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

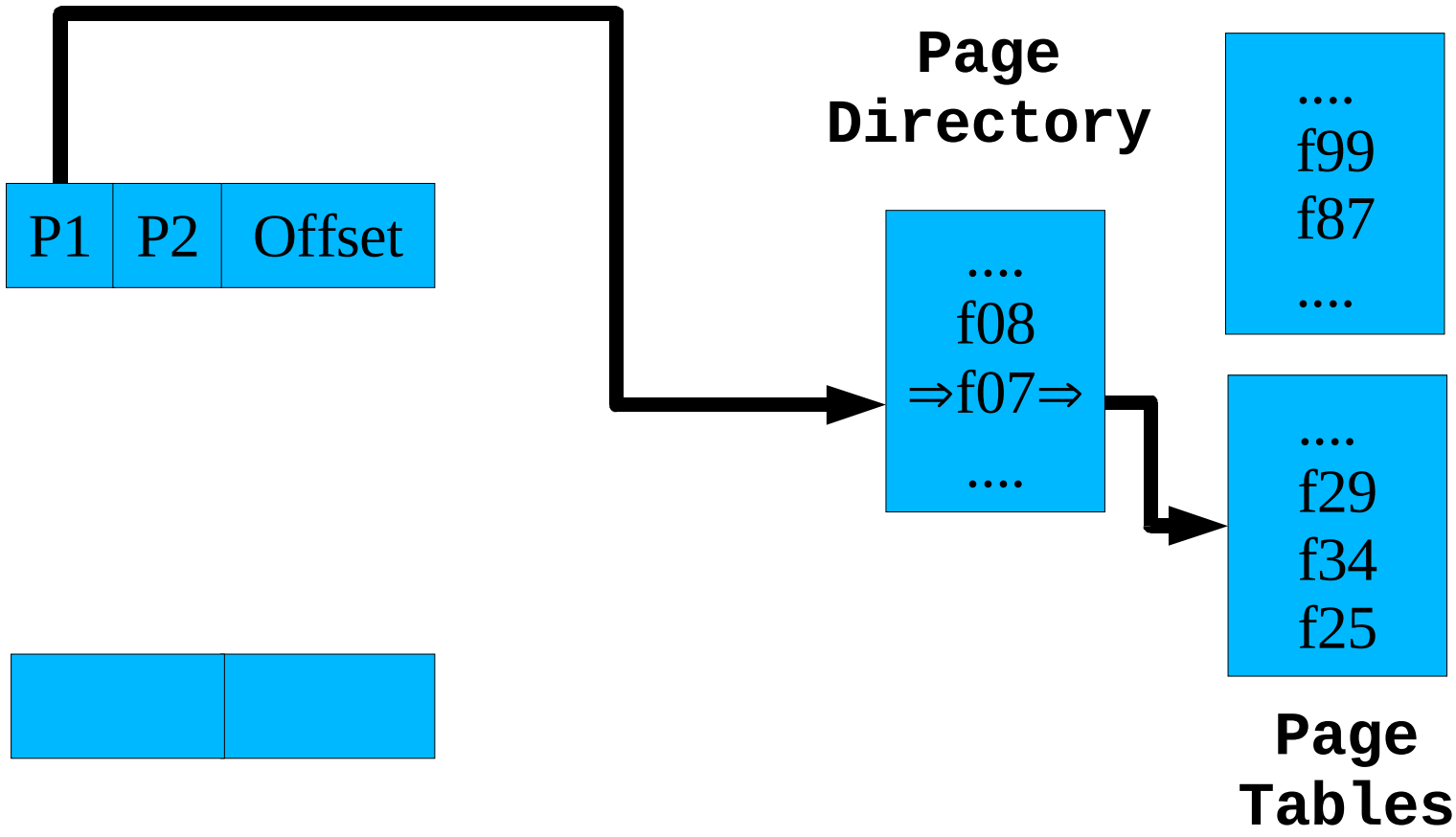
- Page-map hardware caches virtual-to-physical *mappings*
  - Small, fast on-chip memory
  - "Free" in comparison to slow off-chip memory

Simplest Possible TLB

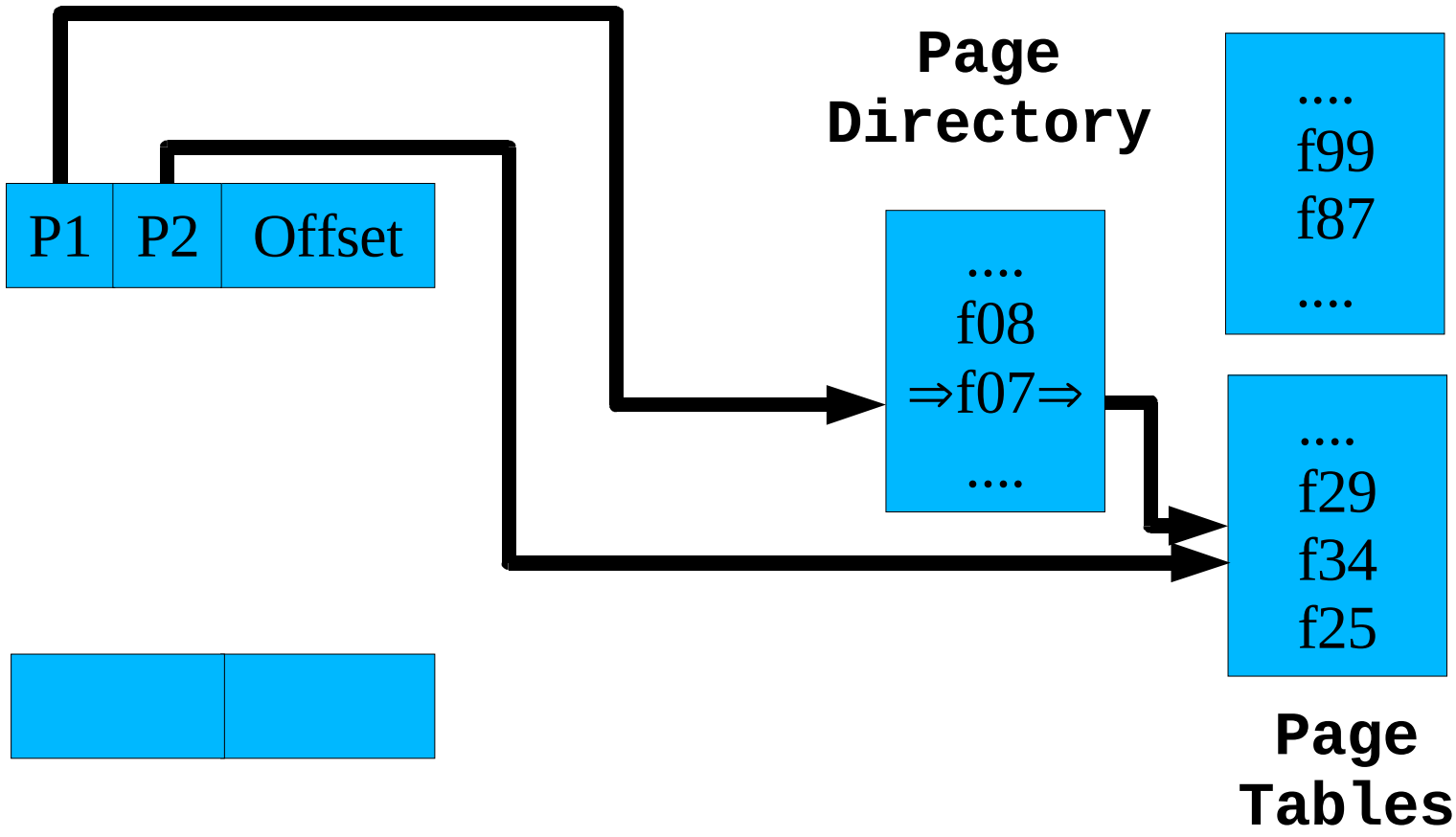
Approach

- 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"

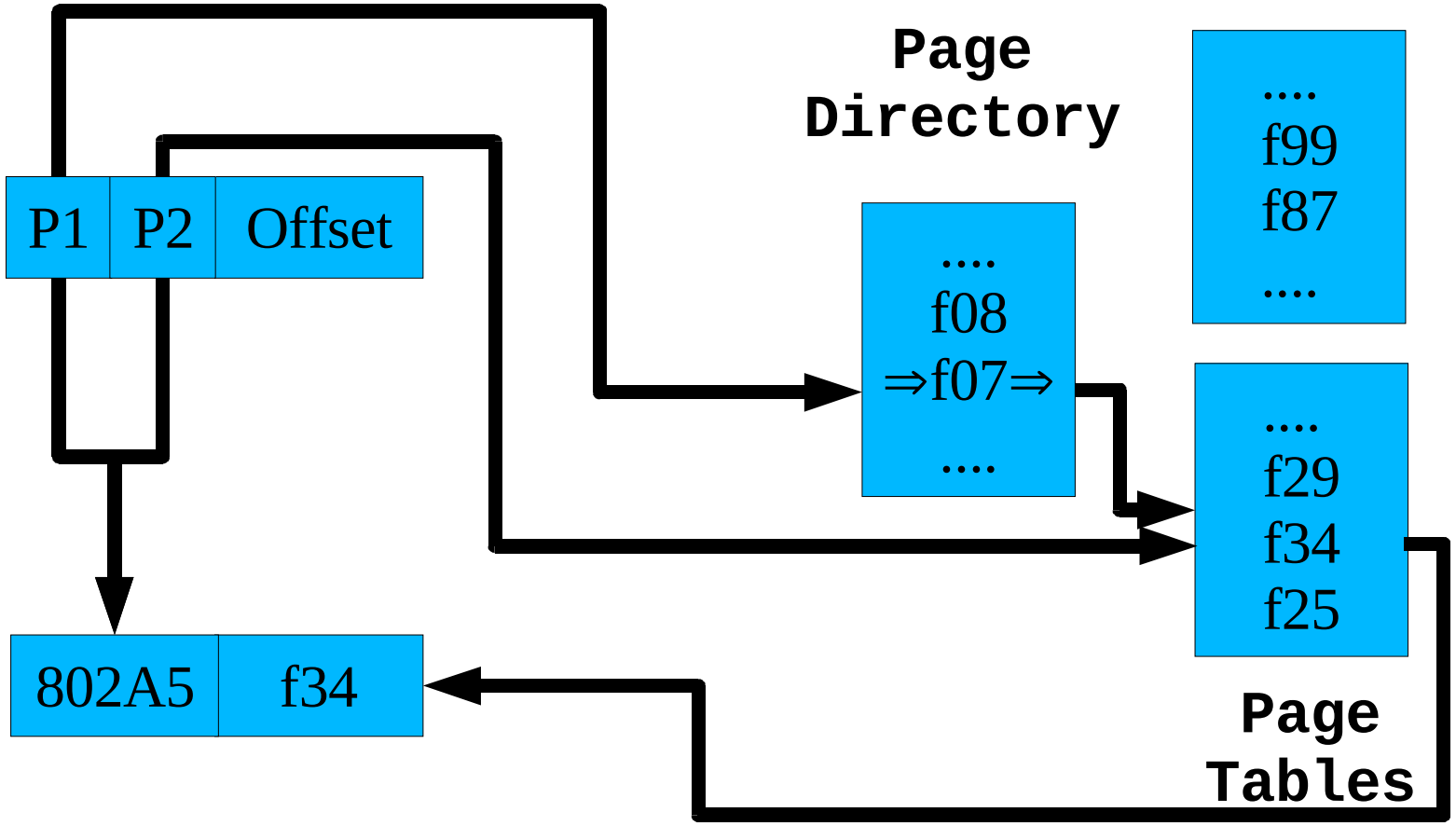
Simplest Possible TLB



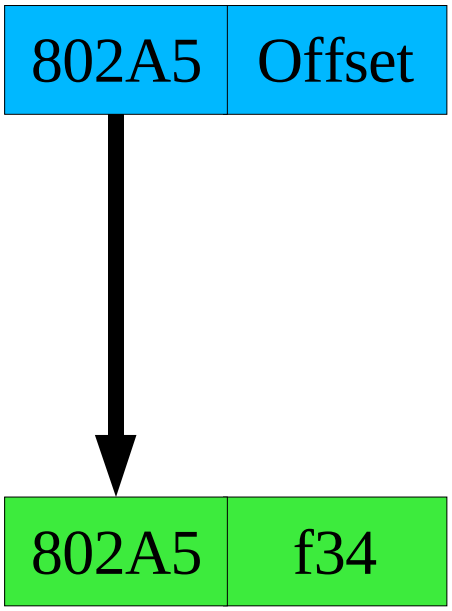
Simplest Possible TLB

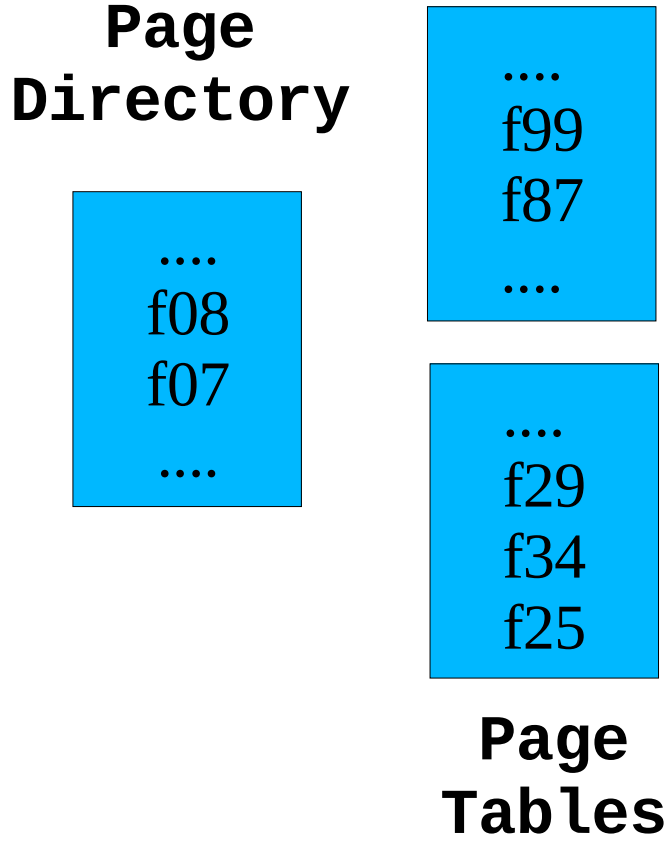


Simplest Possible TLB

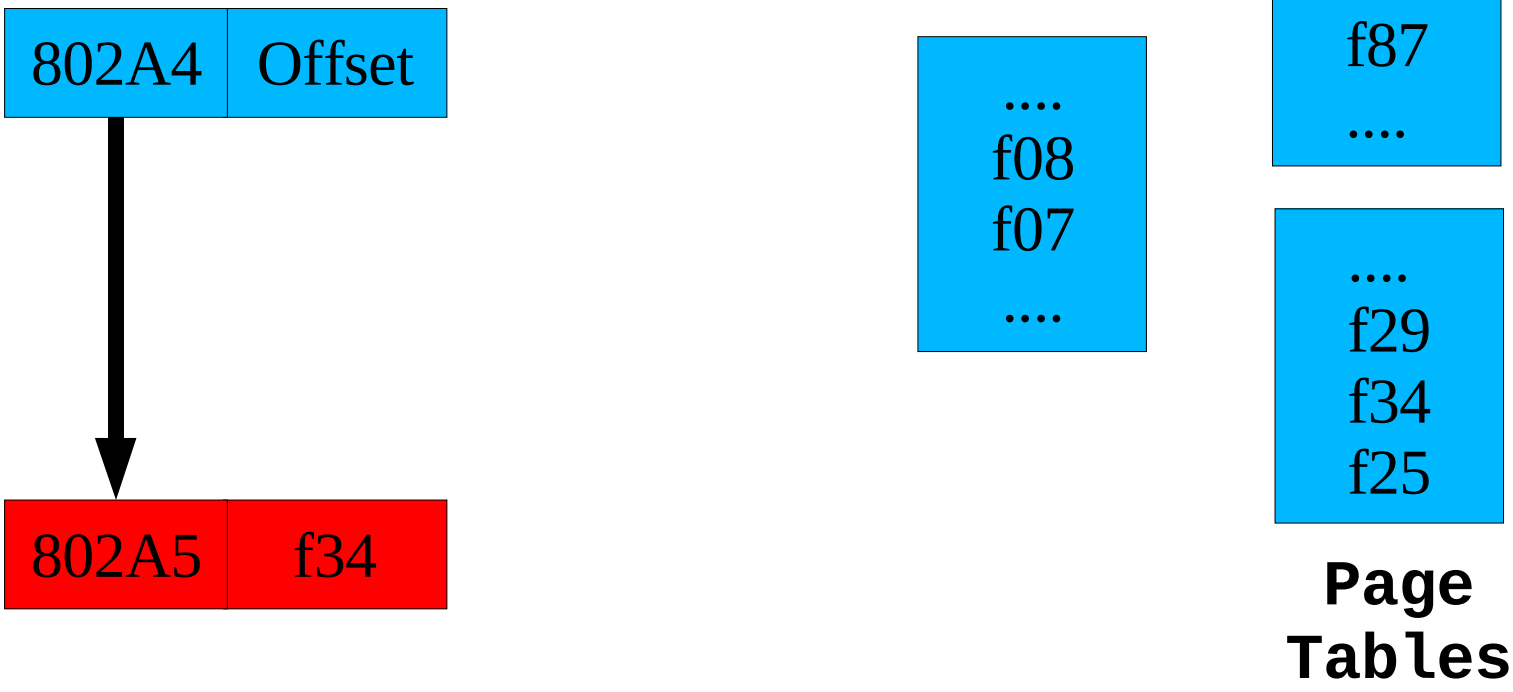


TLB “Hit”

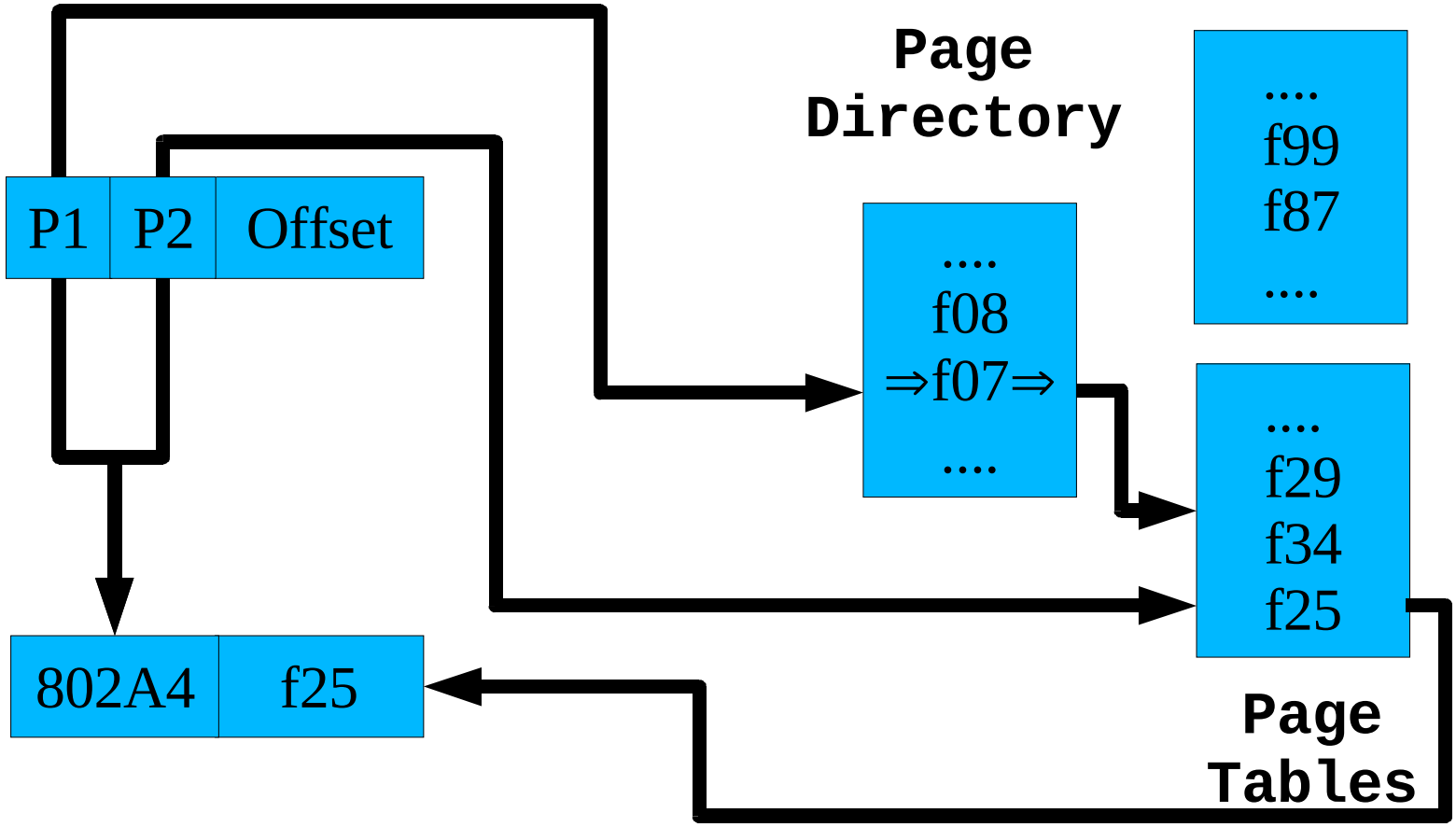

802A5
Offset
802A5
f34


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
  - ...?

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**
  - ***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 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]

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

“Virtual Memory Approach”

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*

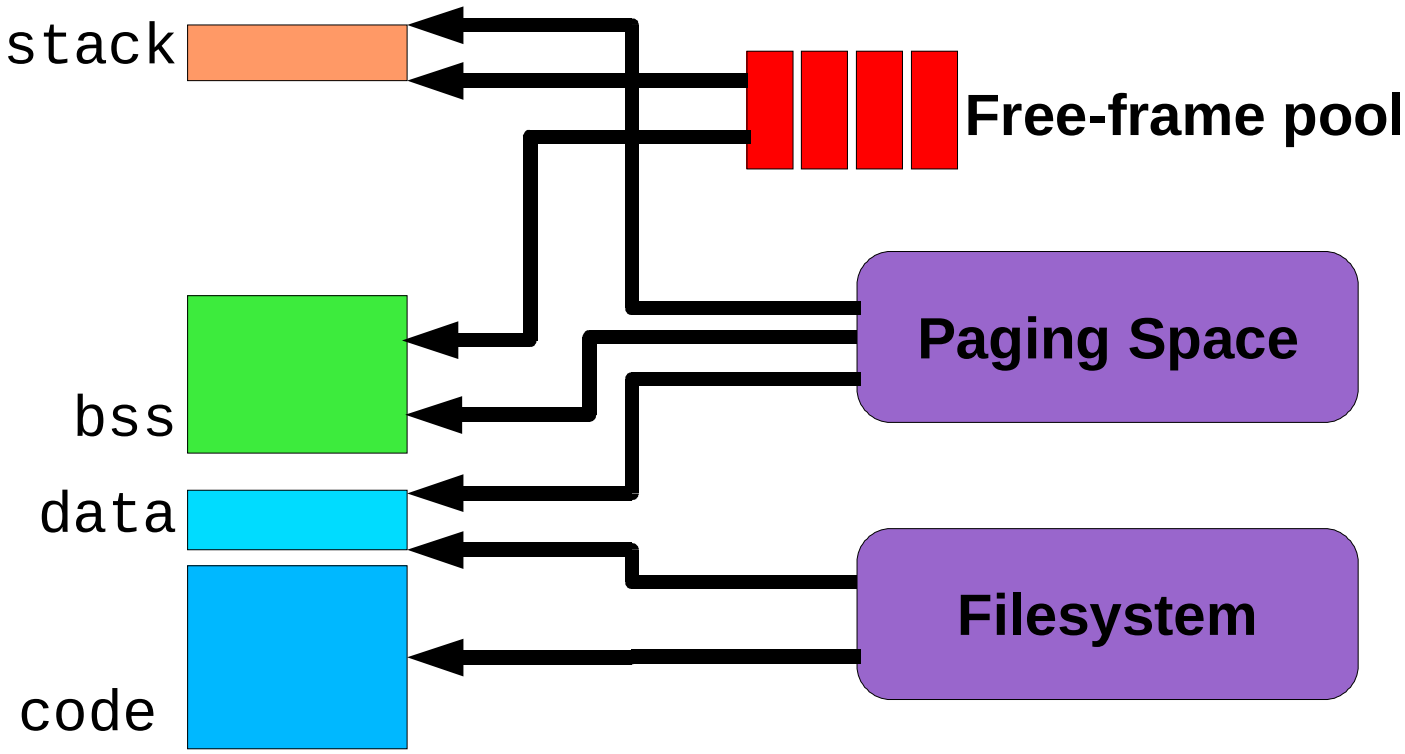
- **Unix –SIGSEGV**
- **Mach –deliver message to thread's exception port**
- **15-410 –kill thread**

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 ⇒ 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**

Summary

The mysterious TLB

- No longer mysterious

Process address space

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

Fault handler is *complicated*

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