# 15-410

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

# Virtual Memory #3
# Feb. 23, 2005

**Dave Eckhardt**

**Bruce Maggs**

# Synchronization

## Mid-term exam

- Monday!

## Homework

- Available online after class
- Due Sunday evening (solutions will be released for study)

## First Project 3 checkpoint

- Next Wednesday during class time
- Meet in Wean 5207
- Show us gettid()
  - Explain which parts are “real”, which are “demo quality”

# Last Time

**Partial memory residence (demand paging) in action**

**Process address space**

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

**Fault handler is *complicated***

- **Page-in, speed hacks (copy-on-write, zero-fill), ...**
- **Shared memory via mmap()**

**Definition & use of**

- **Dirty bit**
- **Reference bit**

# Outline

## Page-replacement policies

- The eviction problem
- Sample policies (theory and practice)
- Page buffering
- Frame Allocation (process page quotas)

## Virtual-memory usage optimizations

## The mysterious TLB

# 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
  - Most OS's: kernel actually does the choosing

## 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
  - But 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
  - Cheap to "store" a page with dirty==0

# Page Eviction Policies

## Don't try these at home

- FIFO
- Optimal
- LRU

## Practical

- LRU approximation

# FIFO Page Replacement

## Concept

- Page queue
- Page added to tail of queue when first given a frame
- Always evict oldest page (head of queue)

## Evaluation

- Fast to “pick a page”
- Stupid
  - Will indeed evict old unused startup-code page
  - But *guaranteed* to eventually 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"
  - ...but it's an important hint!

## Evaluation

- Would probably be reasonably accurate
- LRU is computable without a fortune teller
- Bookkeeping *very* expensive
  - (right?)

# LRU Page Replacement

## Concept

- Evict Least-Recently-Used page
- "Past performance *may* not predict future results"
  - ...but it's an important hint!

## Evaluation

- Would probably be reasonably accurate
- LRU is computable without a fortune teller
- Bookkeeping *very* expensive
  - Hardware must sequence-number every page reference
  - Evictor must scan every page's sequence number

# *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”
- Result:
  - Hardware sloppily partitions memory into “recent” vs. “old”
  - Software periodically samples, makes decisions

# *Approximating* LRU

## "Second-chance" algorithm

- Use stupid FIFO queue to choose victim page
- reference == 0?
  - not "recently" used, evict page, steal its frame
- reference == 1?
  - "somewhat-recently used" - don't evict page this time
  - append page to rear of queue
  - set reference = 0
    - » Process must use page again "soon" for it to be skipped

## Approximation

- Observe that queue is randomly sorted
  - We are evicting not-recently-used, not *least*-recently-used

# *Approximating* LRU

## "Clock" algorithm

- Observe: "Page queue" requires linked list
  - Extra memory traffic to update pointers
- Observe: Page queue's order is essentially random
  - Doesn't add anything to accuracy
- Revision
  - Don't have a queue of pages
  - Just treat memory as a circular array

# 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 after fault happens
- Must wait for disk write before launching disk read...slow...

## "Assume a blank page..."

- Page fault handler can be much faster

## "page-out daemon"

- Scan system for dirty pages
  - Write to disk
  - Clear dirty bit
  - Page can be instantly evicted later
- When, how many?  Indeed...

# Frame Allocation

## How many frames should a process have?

## Minimum allocation

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

# “Fair” Frame Allocation

## Equal allocation

- Every process gets same *number of frames*
  - “Fair” - in a sense
  - Probably wasteful

## Proportional allocation

- Every process gets same *percentage of residence*
  - (Larger processes get more frames)
  - “Fair” - in a different sense
  - Probably the right approach
    - » Theoretically, encourages greediness

# Thrashing

## Problem

- Process *needs* N frames...
  - Repeatedly rendering image to video memory
  - Must be able to have all "world data" resident 20x/second
- ...but 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” Allocation Model

## Approach

- Determine necessary # frames for each process
  - “Working set” - size of frame set you need to get work done
- If unavailable, swap entire process out
  - (later, swap some *other* process out)

## How to measure?

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

## Evaluation

- Expensive
- Can we approximate it?

# Page-Fault Frequency Approach

## Approach

- Thrashing == “excessive” paging
- Adjust per-process frame quotas to balance fault rates
  - System-wide “average page-fault rate” (faults/second)
  - Process A fault rate “too low”: reduce frame quota
  - Process A fault rate “too high”: increase frame quota

## What if quota increase doesn't help?

- If giving you *some* more frames doesn't help, maybe you need *a lot* more frames than you have...
- Swap you out entirely for a while

# Program Optimizations

## Is paging an "OS problem"?

- Can a programmer reduce working-set size?

## Locality depends on data structures

- Arrays encourage sequential accesses
  - Many references to same page
  - Predictable access to next page
- Random pointer data structures scatter references

## Compiler & linker can help too

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

## Effects can be *dramatic*

# 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

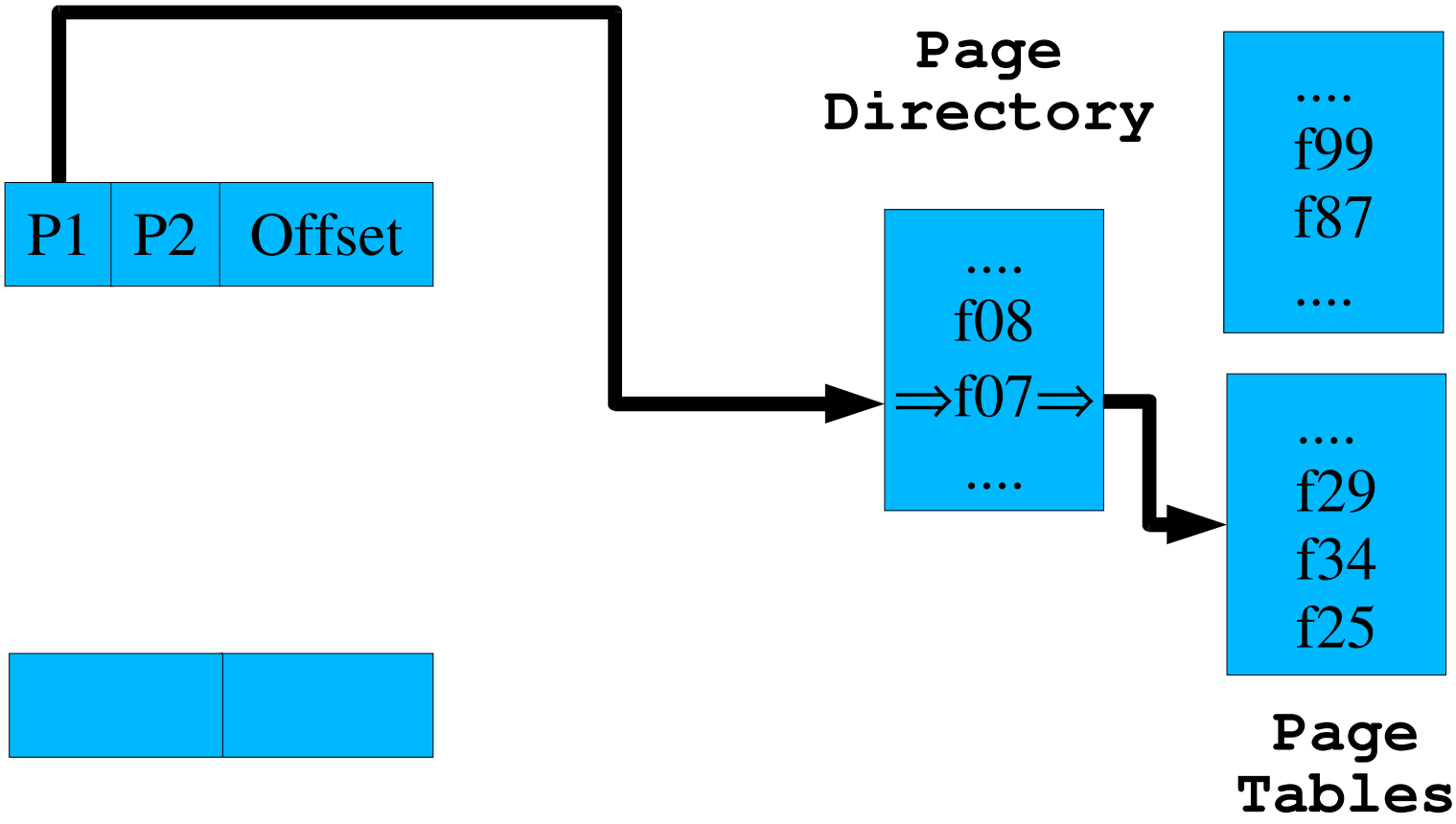
- Page-map hardware caches virtual-to-physical *mappings*
  - Small, fast on-chip memory

# Simplest Possible TLB

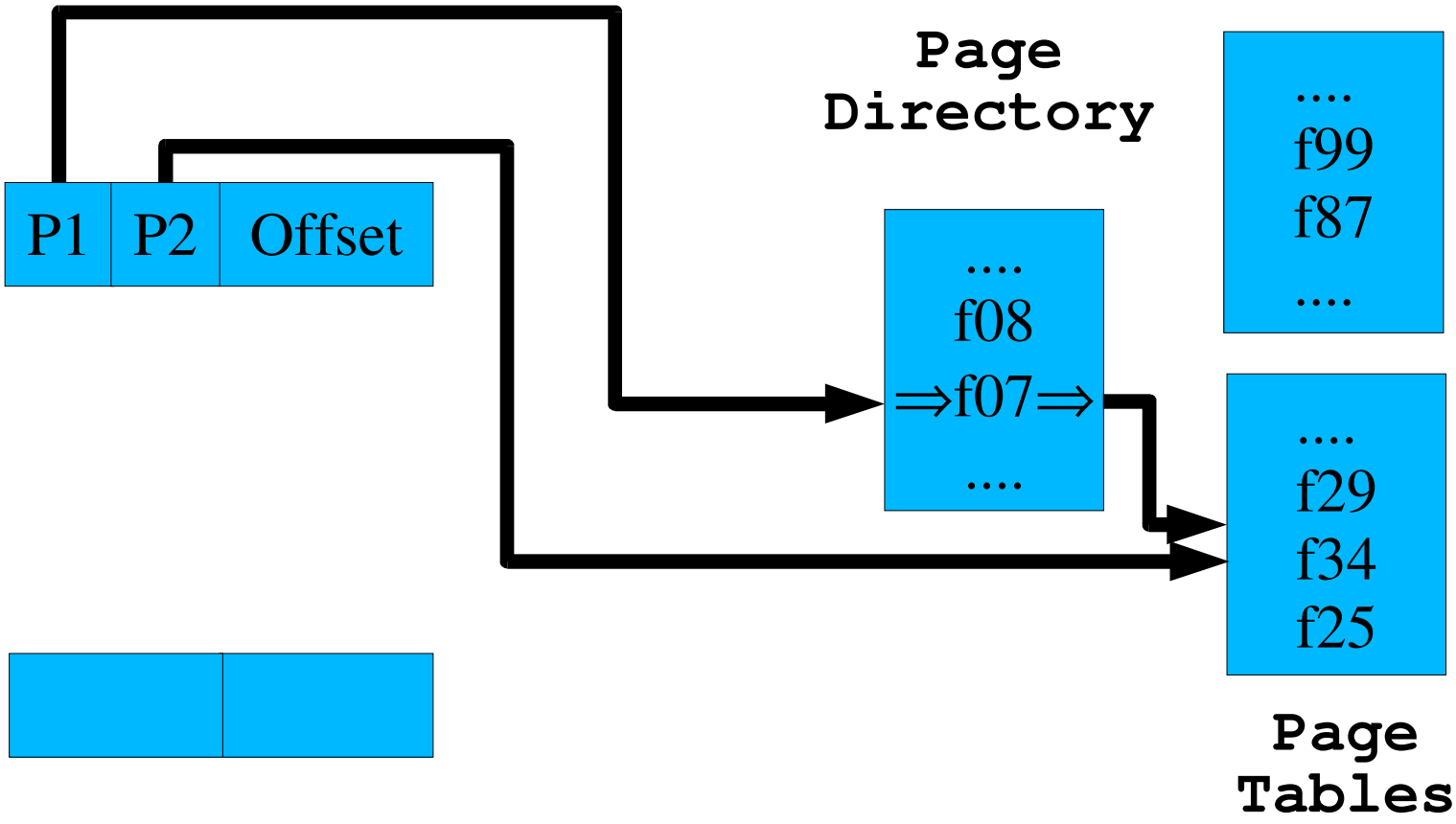
## Approach

- Remember the most-recent virtual-to-physical translation
  - (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"

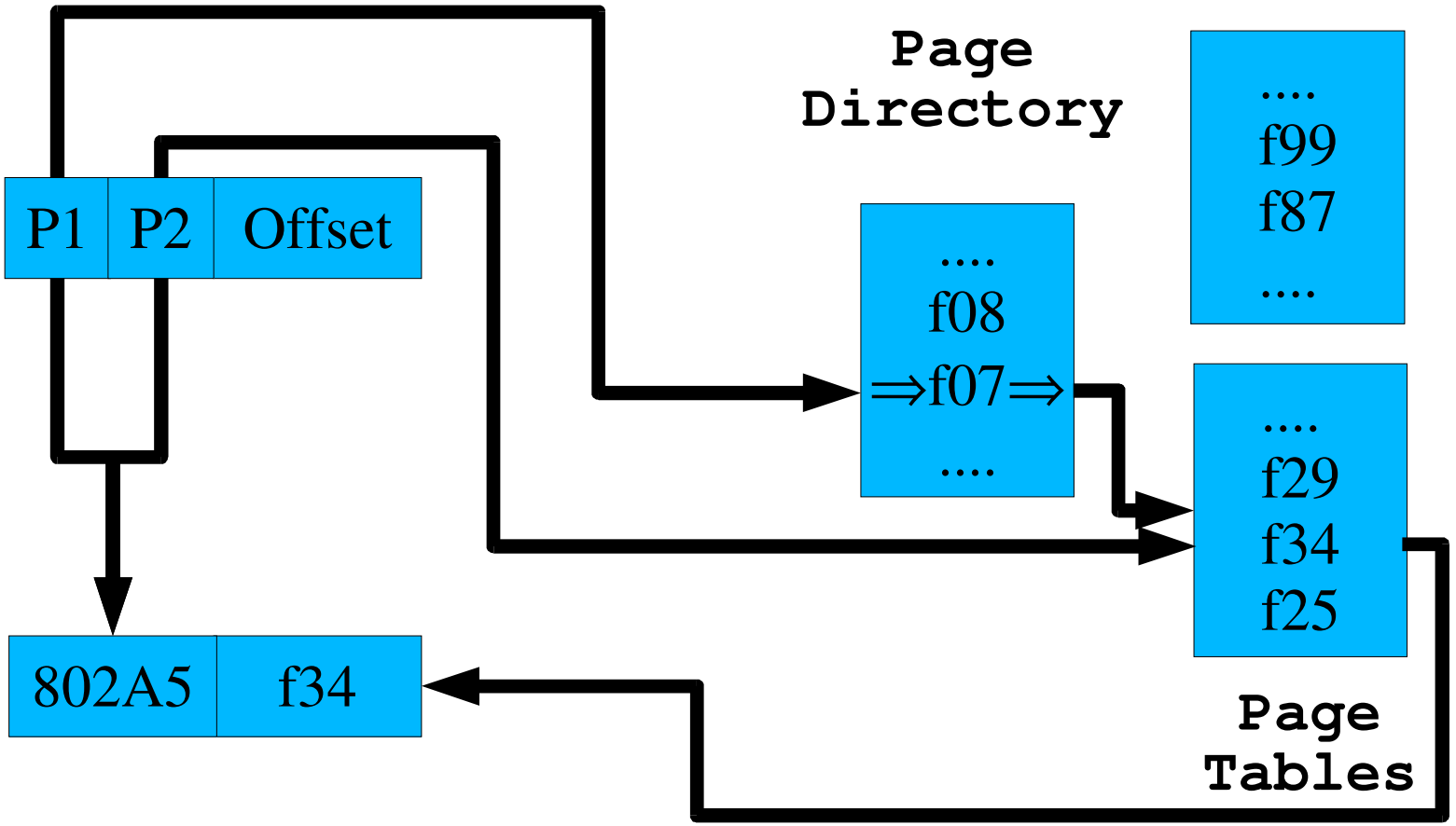
# Simplest Possible TLB



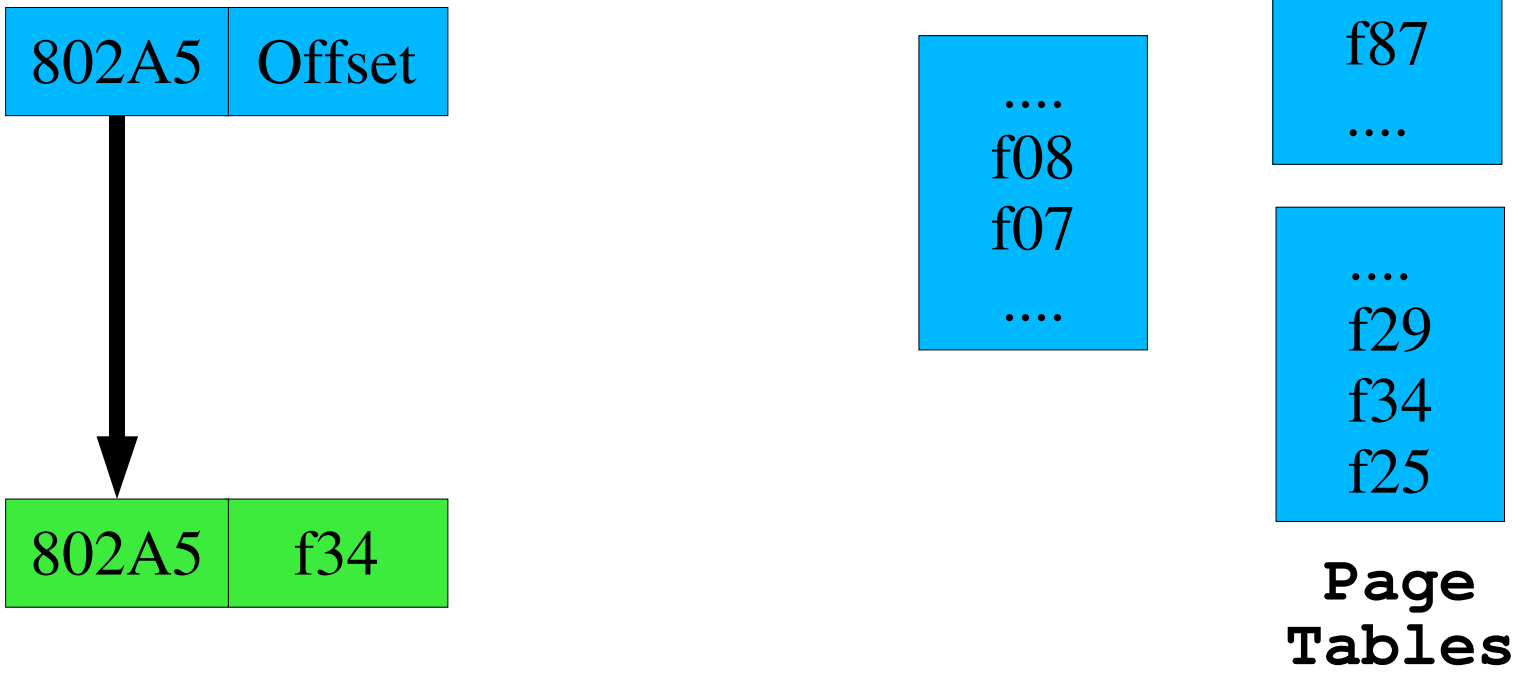
# Simplest Possible TLB



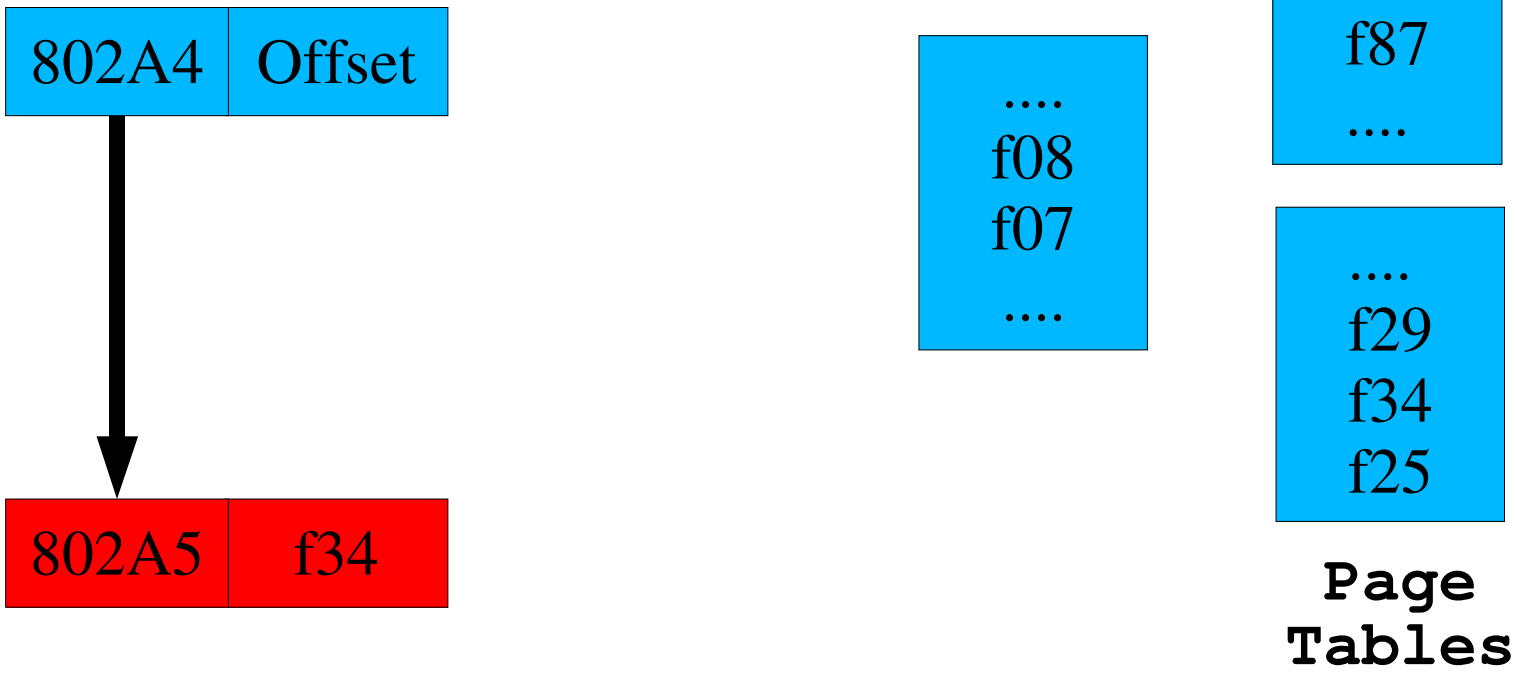
# Simplest Possible TLB



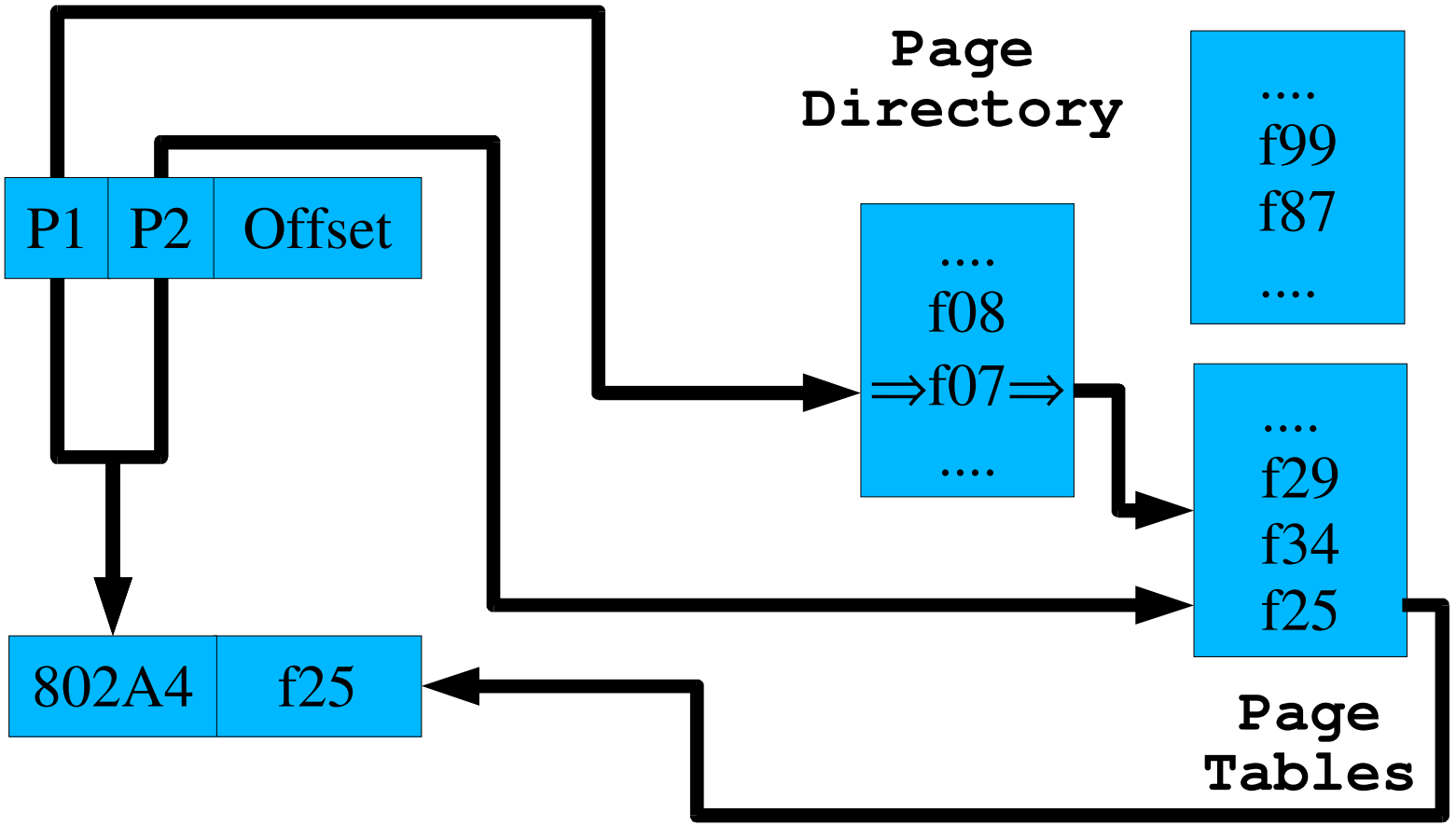
# TLB “Hit”


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 ...?)
  - *Entire contents of TLB!!*
    - » (why?)

# x86 TLB Flush

## 1. Declare new page directory (set %cr3)

- Clears every entry in TLB (whoosh!)
  - Well, not “global” pages...how to use this?

## 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, 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
- Hardware page-mapping unit imposes its own ideas
- 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

- 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 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
- tlb0.info, tlb0.status
  - More bits “trying to tell you something”
- [INVLPG issues with Simics 1. Simics 2?]

# Summary

## Page-replacement policies

- The eviction problem
- Sample policies
  - For real: LRU approximation with hardware support
- Page buffering
- Frame Allocation (process page quotas)

## Virtual-memory usage optimizations

## The no-longer-mysterious TLB