15-410
“...The cow and Zaphod...”

Virtual Memory #3
Feb. 25, 2019

Dave Eckhardt
Brian Raling
Synchronization

Exam tomorrow (Tuesday)!
- 18:30
- Doherty Hall 2210

Homework 1 due tonight!
- Not at midnight!
Synchronization

First Project 3 checkpoint

- Friday during class time
- Meet in Wean 5207
  - If your group number ends with
    - 0-2 try to arrive 5 minutes early
    - 3-5 arrive at 10:42:30
    - 6-9 arrive at 10:59:27

- Preparation
  - Your kernel should be in mygroup/p3ck1
  - It should load one program, enter user space, gettid()
    - Ideally lprintf() the result of gettid()
  - We will ask you to load & run a test program we will name
  - Explain which parts are “real”, which are “demo quality”
Outline

Last time
- The mysterious TLB
- Partial memory residence (demand paging) in action
- The task of the page fault handler

Today
- Fun big speed hacks
- Sharing memory regions & files
- Page replacement policies
Demand Paging Performance

Effective access time of memory word

- \((1 - p_{\text{miss}}) \times T_{\text{memory}} + p_{\text{miss}} \times T_{\text{disk}}\)

Textbook example (a little dated)

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

Virtual Address

<table>
<thead>
<tr>
<th>VRW</th>
<th>f981</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>VRW</td>
<td>f029</td>
</tr>
<tr>
<td>VRX</td>
<td>f237</td>
</tr>
</tbody>
</table>

Page table

stack

code

data
Copy-on-Write of Address Space

P0

<table>
<thead>
<tr>
<th>VRW</th>
<th>f981</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>VRW</td>
<td>f029</td>
</tr>
<tr>
<td>VRX</td>
<td>f237</td>
</tr>
</tbody>
</table>

P9

<table>
<thead>
<tr>
<th>VRW</th>
<th>f981</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>VRW</td>
<td>f029</td>
</tr>
<tr>
<td>VRX</td>
<td>f237</td>
</tr>
</tbody>
</table>

stack

code

data
Memory Write ⇒ Permission Fault
Copy Into Blank Frame

VRW  f981
---  ---
VRW  f029
VRX  f237

stack
stack

code

data
Adjust PTE frame pointer, access

P0

VRW | f982
--- | ---
VRW | f029
VRX | f237

P9

VRW | f981
--- | ---
VRW | f029
VRX | f237

stack
stack
code
data
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
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
Page Replacement/Page Eviction

Processes always want *more* memory frames
- Explicit deallocation is rare
- Page faults are implicit allocations

System inevitably runs out of frames

Solution outline
- 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

Current Research
- ARC (Adaptive Replacement Cache)
- CAR (Clock with Adaptive Replacement)
- CART (CAR with Temporal Filtering)
Page Eviction Policies

Don't try these at home
- FIFO
- Optimal
- LRU

Practical
- LRU approximation

Current Research
- ARC (Adaptive Replacement Cache)
- CAR (Clock with Adaptive Replacement)
- CART (CAR with Temporal Filtering)
- CARTHAGE (CART with Hilarious AppendaGE)
FIFO Page Replacement

**Concept**
- Queue of all pages – named as (task id, virtual address)
- 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
  - Or you can “just” do a doubly-linked-list operation per ref
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 in lookup
- 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 candidate 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 (“second chance”)
  - 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 a fault needs a frame
- Must wait for disk write before launching disk read (slow!)

“Assume a blank page...”
- Page fault handler can be much faster

“page-out daemon”
- Scans system for dirty pages
  - Write to disk
  - Clear dirty bit
  - Page can be instantly evicted later
- When to scan, how many to store? 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*
  - (Everybody 83% resident, 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
- Widespread unhappiness
“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 entirely out)

**How to measure working set?**
- Periodically scan all reference bits of process's pages
- Combine multiple scans (see text)

**Evaluation**
- Expensive
- Can we approximate it?
Page-Fault Frequency Approach

**Approach**

- Recall, “thrashing” == “excessive” paging
- Adjust per-process frame quotas to balance fault rates
  - System-wide “average page-fault rate” (10 faults/second)
  - Process A fault rate “too high”: increase frame quota
  - Process A fault rate “too low”: reduce frame quota

**What if quota increase doesn't help?**

- If giving you *some* more frames didn'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
Summary

Speed hacks

Page-replacement policies

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

Definition & use of

- Dirty bit, reference bit

Virtual-memory usage optimizations