MEMORY: SWAPPING

Shivaram Venkataraman
CS 537, Spring 2019
- Project 2b is out. Due Feb 27\textsuperscript{th}, 11:59
- Project 1b grades are out
LESSONS FROM P2A?

1. Start early!
2. Sketch out a design?
3. Synthesize ideas from various sources:
   - TAs, stackoverflow.com, gdb, discussion videos
4. Handling edge cases, string handling
AGENDA / LEARNING OUTCOMES

Memory virtualization

   How we support virtual mem larger than physical mem?
   What are mechanisms and policies for this?
RECAP
PAGING TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**
   if miss:
   3. calculate addr of **PTE** (page table entry)
   4. read **PTE** from memory, add to TLB
5. extract **PFN** from TLB (page frame num)
6. build **PA** (phys addr)
7. read contents of **PA** from memory
COMBINE PAGING AND SEGMENTATION

Divide address space into segments (code, heap, stack)
   – Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions

<table>
<thead>
<tr>
<th>seg # (4 bits)</th>
<th>page number (8 bits)</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
</table>

Implementation

- Each segment has a page table
- Each segment track base (physical address) and bounds of the page table
Multilevel Page Tables

30-bit address:

- outer page (8 bits)
- inner page (10 bits)
- page offset (12 bits)

base of page directory
How should logical address be structured? How many bits for each paging level?

Goal?
- Each page table fits within a page
- PTE size * number PTE = page size
  Assume PTE size = 4 bytes
  Page size = $2^{12}$ bytes = 4KB

→ # bits for selecting inner page =

Remaining bits for outer page:
- $30 - ___ - ___ = ___$ bits
Problem with 2 levels?

Problem: page directories (outer level) may not fit in a page

Solution:
- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.

How large is virtual address space with 4 KB pages, 4 byte PTEs, (each page table fits in page)

4KB / 4 bytes → 1K entries per level

1 level:
2 levels:
3 levels:
On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

<table>
<thead>
<tr>
<th>ASID</th>
<th>VPN</th>
<th>PFN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>0xbb</td>
<td>0x91</td>
<td>1</td>
</tr>
<tr>
<td>211</td>
<td>0xff</td>
<td>0x23</td>
<td>1</td>
</tr>
<tr>
<td>122</td>
<td>0x05</td>
<td>0x91</td>
<td>1</td>
</tr>
<tr>
<td>211</td>
<td>0x05</td>
<td>0x12</td>
<td>0</td>
</tr>
</tbody>
</table>
INVERTED PAGE TABLE

Only need entries for virtual pages w/ valid physical mappings

Naïve approach:
  Search through data structure <ppn, vpn+asid> to find match
  Too much time to search entire table

Better:
  Find possible matches entries by hashing vpn+asid
  Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB
SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables
If OS traps on TLB miss, OS can use any data structure
   – Inverted page tables (hashing)
If Hardware handles TLB miss, page tables must follow specific format
   – Multi-level page tables used in x86 architecture
   – Each page table fits within a page
SWAPPING
OS goal: Support processes when not enough physical memory
   – Single process with very large address space
   – Multiple processes with combined address spaces
User code should be independent of amount of physical memory
   – Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory
Why does this work?
   – Relies on key properties of user processes (workload) and machine architecture (hardware)
Leverage locality of reference within processes

- **Spatial**: reference memory addresses near previously referenced addresses
- **Temporal**: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory
Leverage *memory hierarchy* of machine architecture
Each layer acts as “backing store” for layer above
SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk
   – Slower, cheaper backing store than memory
Process can run when not all pages are loaded into main memory
OS and hardware cooperate to make large disk seem like memory
   – Same behavior as if all of address space in main memory

Requirements:
   – OS must have mechanism to identify location of each page in address space → in memory or on disk
   – OS must have policy for determining which pages live in memory and which on disk
SWAPPING MECHANISMS

Each page in virtual address space maps to one of three locations:
- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present
- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>present</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>r-w-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>r-w-</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>r-w-</td>
<td>1</td>
</tr>
</tbody>
</table>

What if access vpn 0xb?
VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address
  – if TLB hit, address translation is done; page in physical memory
Else
  ...
  – Hardware or OS walk page tables
  – If PTE designates page is present, then page in physical memory
    (i.e., present bit is cleared)
Else
  – Trap into OS (not handled by hardware)
  – OS selects victim page in memory to replace
    • Write victim page out to disk if modified (add dirty bit to PTE)
  – OS reads referenced page from disk into memory
  – Page table is updated, present bit is set
  – Process continues execution
SWAPPING POLICIES
SWAPPING POLICIES

Goal: Minimize number of page faults
- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions
- Page selection
  **When** should a page (or pages) on disk be **brought into** memory?

- Page replacement
  **Which** resident page (or pages) in memory should be **thrown out** to disk?
Demand paging: Load page only when page fault occurs
- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced
- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems?

Hints: Combine above with user-supplied hints about page references
- User specifies: may need page in future, don’t need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix
Which page in main memory should be selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

**OPT:** Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison
**PAGE REPLACEMENT**

**FIFO**: Replace page that has been in memory the longest
- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement
- Disadvantage: Some pages may always be needed

**LRU**: Least-recently-used: Replace page not used for longest time in past
- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed
  - Does not handle all workloads well
# Page Replacement Example

Page reference string: ABCABDADBCB

<table>
<thead>
<tr>
<th>Metric: Miss count</th>
<th>OPT</th>
<th>FIFO</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three pages of physical memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PAGE REPLACEMENT COMPARISON

Add more physical memory, what happens to performance?

LRU, OPT:
• Guaranteed to have fewer (or same number of) page faults
• Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
• Stack property: smaller cache always subset of bigger

FIFO:
• Usually have fewer page faults
• Belady’s anomaly: May actually have more page faults!
FIFO PERFORMANCE MAY DECREASE!

Consider access stream: ABCDABEABCDE

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?
IMPLEMENTING LRU

Software Perfect LRU
- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU
- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU
- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest
CLOCK ALGORITHM

Hardware
- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

Operating System
- Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear use bits as search
  - Stop when find page with already cleared use bit, replace this page
CLOCK: LOOK FOR A PAGE

Physical Mem:

| 0 | 1 | 2 | 3 | ... |

Use=  Use=  Use=  Use=

clock hand
CLOCK EXTENSIONS

Replace multiple pages at once
  – Intuition: Expensive to run replacement algorithm and to write single block to disk
  – Find multiple victims each time and track free list

Use dirty bit to give preference to dirty pages
  – Intuition: More expensive to replace dirty pages
    Dirty pages must be written to disk, clean pages do not
  – Replace pages that have use bit and dirty bit cleared
SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging
- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging, inverted page tables etc.

Larger address spaces: Swapping mechanisms, policies (LRU, Clock)
NEXT STEPS

Project 2b: Out now

Next class: New module on Concurrency