CS5460: Operating Systems

Lecture 15: Intro to Demand Paging
(Ch. 9)
Important From Last Time

- Translation Lookaside Buffer (TLB)
  - These are key to making virtual memory work in real operating systems
  - Must be fast
  - If TLB is thrashing performance becomes poor

- Multi-level page tables
  - Also key to making virtual memory work in practice
  - Up to 4 levels in real systems

- Cool paging tricks
  - Shared memory
  - Copy on write
  - Memory-mapped files
  - Optimistic memory allocation
Demand Paging

- **Overview**
  - Use disk as “slow” main memory
  - Analogous to cache → DRAM

- **Goal**
  - Memory as large as disk
  - Memory as fast as DRAM (cache!)

- **Basic idea**
  - Special page table mappings indicate “valid mapping, but page not present in RAM right now”
  - “PFN” specifies location on disk
  - Accesses to a “paged out” page generates a page fault
  - OS reloads page, updates mapping, and resumes process
Demand Paging Implementation

- Keep copy of process on disk → some in RAM
- Extend page table to include "present" and "valid"
- Access to "paged out" data → trap
- Inside trap handler:
  - Check to see if page fault due to bogus address or access to page temporarily paged out to disk
  - If latter:
    » Allocate a physical page frame to hold contents (might require another page to be paged out!)
    » Copy data from disk to allocated page frame (slow!)
    » Update page table entry
    » Let process continue (reschedule process, re-execute failed ld/st)
Issues

- **Performance**
  - Need lots of locality → otherwise run at disk speeds
    » If most accesses are to data already in DRAM → works great
    » Spatial locality: Often access “nearby” addresses
    » Temporal locality: Often re-access same addresses again and again
  - Analogous to way in which DRAM backs cache(s)

- Can support virtual address space > size of DRAM

- How do we resume a process?
  - Skip instruction? No!
  - Re-execute instruction? Only if no side effects!
  - Get help from CPU → resumable instructions

- Run other processes / threads while servicing the page fault
Timeline of a Page Fault

1. Trap to operating system
2. Save state in PCB
3. Vector to page fault handler
4. Check in page table if fault address truly is invalid
5. Find/create free page frame
   a. Possibly involves disk write
6. Issue disk read for page
   a. Wait until request queued at disk controller
   b. Wait for seek/latency
   c. Wait for data transfer (DMA)
7. Schedule other processes / threads while waiting
8. Take disk interrupt
9. Update page table
10. Add process to run queue
11. Wait for process to be scheduled next
12. Restore state from PCB
13. Return from OS
Linux page fault handler

1. Read address of fault
2. address > TASK_SIZE && In Kernel Mode
   - Yes: vmalloc_fault
   - No: in_interrupt() || no_mm_context
     - Yes: Fix up page tables
     - No: Can grow nearby region?
       - Yes: pte_present?
         - Yes: Exception handler exist?
           - Yes: Call Exception Handler
           - No: Kernel Bug oops()
         - No: In kernel space?
           - Yes: no_context
             - No: Fault completed
           - No: Permissions OK?
             - Yes: handle_mm_fault
             - No: Fault completed
     - No: expand_stack()? (Valid region?)
       - Yes: good_area:
         - Permissions OK?
           - Yes: Fault completed
           - No: Fault completed
       - No: bad_area:
         - In kernel space?
           - Yes: no_context
             - No: Exception handler exist?
               - Yes: Call Exception Handler
               - No: Kernel Bug oops()
Effective Access Times

What is average access latency?

- L1 cache: 2 cycles
- L2 cache: 10 cycles
- Main memory: 150 cycles
- Disk: 10 ms → 30,000,000 cycles on 3.0 GHz processor

Assume access have following characteristics:
  » 98% handled by L1 cache
  » 1% handled by L2 cache
  » 0.99% handled by DRAM
  » 0.01% cause page fault

What’s the average access latency?
Effective Access Times

- What is average access latency?
  - L1 cache: 2 cycles
  - L2 cache: 10 cycles
  - Main memory: 150 cycles
  - Disk: 10 ms → 30,000,000 cycles on 3.0 GHz processor
  - Assume access have following characteristics:
    » 98% handled by L1 cache
    » 1% handled by L2 cache
    » 0.99% handled by DRAM
    » 0.01% cause page fault
  - Average access latency:
    » \((0.98 \times 2) + (0.01 \times 10) + (0.0099 \times 150) + (0.0001 \times 30,000,000)\) = 1.96 + 0.1 + 1.485 + 3000 = about 3000 cycles / access

- Moral: Need LOW fault rates to sustain performance!
More Issues

- Page selection policy
  - When do we load a page?

- Page replacement policy
  - What page(s) do we swap to disk to make room for new pages?
  - When do we swap pages out to disk?

- How do we handle “thrashing”?
Page Selection Policy

- **Demand paging:**
  - Load page in response to access (page fault)
  - This is the predominant selection policy

- **Pre-paging (prefetching):**
  - Predict what pages will be accessed in near future
  - Prefetch pages in advance of access
  - **Problems:**
    - Hard to predict accurately
    - Mispredictions can cause useful pages to be replaced

- **Overlays**
  - Application controls when pages loaded/replaced
  - Only really relevant now for embedded/real-time systems
**Page Replacement Policies**

- **Random**
  - Works surprisingly well

- **FIFO (first in, first out)**
  - Throw out oldest pages

- **MIN (or OPTIMAL)**
  - Throw out page used farthest in the future

- **LRU (least recently used)**
  - Throw out page not used in longest time

- **NFU/Clock (not frequently used)**
  - Approximation to LRU → do not throw out recently used pages
Important From Today

- **Key ideas**
  - Main memory acts as a cache
  - Cache misses go do disk
  - Only use RAM for pages that have been recently accessed

- **All general-purpose operating systems today use demand paging**
  - You have to understand it to understand Windows, Linux, etc.
  - If these systems didn’t use demand paging, the exec() system call would be extremely slow