Demand Paging

Arvind Krishnamurthy
Spring 2004

Demand Paging

- So far: all of a job's virtual address space must be in physical memory
- Programs don't use all of their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of the code
- Use main memory as a cache for disk
- Bigger virtual address space: illusion of infinite memory
- Allow more programs than will fit in memory to be running at the same time

Demand paging mechanism

- Page table has "present" (valid) bit
  - if present, pointer to page frame in memory
  - if not present, go to disk
- Hardware traps to OS on reference to invalid page
- OS software
  - choose an old page to replace
  - if old page has been modified, write contents back to disk
  - change its page table entry and TLB entry
  - load new page into memory from disk
  - update page table entry
  - continue thread

all this is transparent, OS can run another job in the meantime.

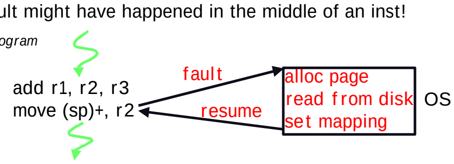
Main Issues

- how to resume a process after a fault?
  - need to save state and resume.
  - process might have been in the middle of an instruction!
- what to fetch?
  - just the needed page or more?
- what to eject?
  - cache always too small, which page to replace?
  - may need to write the evicted page back to the disk
- how many pages for each process?

Problem: resuming process after a fault

- Fault might have happened in the middle of an inst!

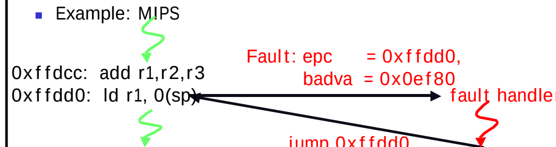
Usr program



- Key constraint: don't want user process to be aware that page fault happened (just like context switching)
- Can we skip the faulting instruction? No.
- Can we restart the instruction from the beginning?
  - Not if it has partial-side effects.
- Can we inspect instruction to figure out what to do?

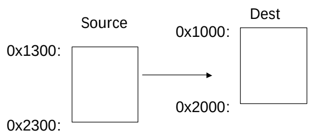
Faulting Instructions

- RISC machines are pretty simple:
  - instructions tend to have 1 memory ref & 1 side effect.
  - thus, only need faulting address and faulting PC.
  - might have to wait for previous loads to complete
- Example: MIPS



CISC Instructions: harder to roll back

- multiple memory references and side effects
- block transfer?



- What happens if there is a page fault while accessing location 0x2000?
  - Cannot restart the instruction from the beginning, need special handling of these situations

Page Replacement Policies

- Random
- FIFO
  - Throw out the oldest page
  - Pros: Low-overhead implementation
  - Cons: May replace the heavily used pages
- Optimal or MIN
  - Replace the page that won’t be used for the longest time
  - Minimal page faults, but offline algorithm
- Least Recently Used
  - Replace page that hasn’t been used for the longest time

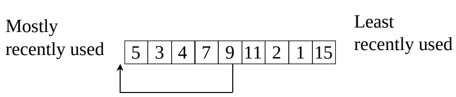
Some Interesting Facts

- More page frames →fewer faults?
  - Consider the following reference string with 4 page frames
  - FIFO replacement
  - **1, 2, 3, 4**, 1, 2, **5, 1, 2, 3, 4, 5**
  - 10 page faults
- Consider the same reference string with 3 page frames
  - FIFO replacement
  - **1, 2, 3, 4**, **1, 2**, **5**, 1, 2, **3, 4,** 5
  - **9 page faults!**
- This is called Belady’s anomaly

Announcements

- Exam on Friday
  - From 1:30 to 2:30
  - Open-notes, open-book exam
  - Sample exam will be posted on website tomorrow
- Next Monday:
  - Submit review for the Multics paper

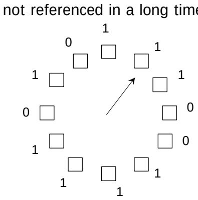
Implementing LRU



- What hardware mechanisms are required to implement LRU?
- Faithful Implementation:
  - Use a timestamp on each reference
  - Keep a list of pages ordered by time of reference
  - Impractical

Clock Algorithm

- Approximate LRU
- Replace some old page, not the oldest unreferenced page
- Arrange physical pages in a circle with a clock hand
  - Hardware keeps “use bit” per physical page frame
  - Hardware sets “use bit” on each reference
  - If “use bit” isn’t set, means not referenced in a long time
- On page fault:
  - Advance clock hand
  - Check “use bit”
  - If “1” clear, go on
  - If “0”, replace page



Clock: Simple FIFO + 2^{nd} Chance

- Will it always find a page or loop infinitely?
  - Even if all use bits are set, it will eventually loop around clearing all use bits ➔ FIFO
- What if hand is moving slowly?
  - Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of use bits set
- One way to view clock algorithm: crude partitioning of pages into two categories: young and old
- Why not partition into more than 2 groups?
  - Could consider "modified bit" and avoid write-backs
  - Can give "more chances"

N^{th} Chance

- Don't throw out until hand has swept by n times
- OS keeps counter per page: number of sweeps
- On page fault: OS checks use bit
  - 1 ➔ clear use and also clear counter
  - 0 ➔ increment counter, if counter equals N, replace page

    Else go on
- How do we pick N?
  - If we pick larger N: better approx to LRU
  - If we pick small N: more efficient, otherwise might have to look a long way to find free page
  - It is a "voodoo constant"

Hardware support for virtual memory

- One extreme:
  - Hardware checks TLB on every reference
  - If TLB entry doesn't exist, hardware checks the page tables in memory
  - If the page exists in memory, hardware loads the TLB with the appropriate page table entry
  - The page tables could also contain "use bits" and "modified bits"
  - Only if the page does not exist in memory, the OS is invoked
- Less hardware support:
  - Hardware checks TLB on every reference
  - If translation doesn't exist, immediately traps into OS
  - Hardware is not aware of page tables and the state associated with pages (use bits and modified bits)
  - OS manages everything and simulates some of these bits

State per page table entry

Many OS's maintain four bits per page table entry:

- **valid** (aka **present**): ok for program to reference this page
- **read-only**: ok for program to read page, but not to modify it (e.g., for catching modifications to code pages)
- **use** (aka **reference**): set when page is referenced, cleared by "clock algorithm"
- **modified** (aka **dirty**): set when page is modified, cleared when page is written to disk

Emulating "modified bit" in software

- BSD Unix started the practice of emulating this bit in software
- Keep two sets of pages:

  (1) Pages user program can access without taking a fault

  (2) Pages in memory
- (2) is a superset of (1)
- Initially mark all pages as "read-only"
  - TLB has "read-only" bit
  - Traps into OS if there is a write to a read-only page
  - OS sets modified bit in its software controlled data structure, marks TLB entry "read-write", resumes program
  - When page comes back in from disk, mark "read-only"

Emulating "use bit"

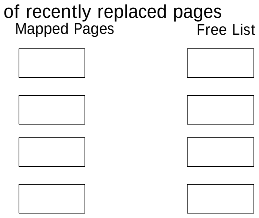
- Exactly the same approach as above:
  - Mark all pages as invalid, even if in memory
  - In other words, lose the corresponding TLB entry to a page when you want to clear the page's use bit
  - On read to this page, trap to OS
  - OS sets "use bit" in its data structures, loads TLB entry as valid, resumes program
  - When clock hand passed by, and when you want to reset use bit
    - Mark page as invalid by invalidating its TLB entry
- Remember that "clock" is just an approximation to LRU
  - Can we do a better approximation since we are trapping into the OS on a page fault (when we collect use information)

VAX-VMS System (Levy-Litman paper)

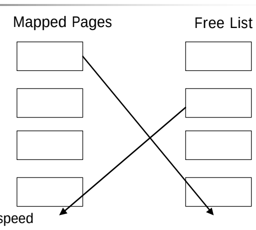
- Historical context: written in 1982
  - VMS designed in 1975
  - Released in 1978
- Motivation for VMS virtual memory design
  - Very large physical and virtual memories
  - Very slow physical devices (no more drums)
- Hardware issues:
  - Single virtual address space shared by OS and current user process
    - All of the user space is accessible
    - OS is just a collection of protected procedure calls
  - Page table organization:
    - User pages in system's address space, allows page tables to be paged
  - Page size small (512 bytes)
  - No use bits: whether page has been referenced or not
  - Divided the address space into four segments
    - Probably not enough segments

VMS OS Software

- Process-local replacement
  - Per-process quota, replace within a process
  - One rogue process cannot bring the system down
- Scanning use bits is costly
  - Use FIFO replacement with a twist
- Use free list as buffer of recently replaced pages



Second Chance List



- On page reference:
  - If mapped, access at full speed
  - Otherwise, page fault:
    - If on second chance list, mark read-write, move first page on FIFO list onto end of second chance list
    - If not on second chance list, bring into memory, move first page on FIFO list onto end of second chance list, replace first page on second chance list
  - 0 pages for second chance list: FIFO
  - If zero page on FIFO list: LRU but page fault on every page reference