UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Virtual Memory

Questions answered in this lecture:

How to run process when not enough physical memory?

When should a page be moved from disk to memory?

What page in memory should be replaced?

How can the LRU page be approximated efficiently?

Motivation

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)

Locality of Reference

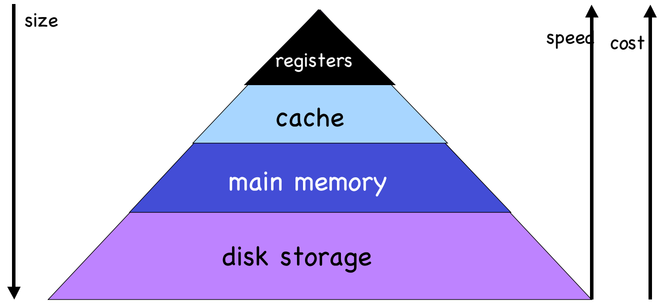
Leverage locality of reference within processes
- Spatial: reference memory addresses near previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:
- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage memory hierarchy of machine architecture

Each layer acts as "backing store" for layer above



Virtual Memory 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 provide illusion of large disk as fast as main memory
- Same behavior as if all of address space in main memory
- Hopefully have similar performance

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

Virtual Address Space 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
  - Trap: page fault

Virtual Memory Mechanisms

Hardware and OS cooperate to translate addresses

First, hardware checks TLB for virtual address
- if TLB hit, address translation is done; page in physical memory

If TLB miss...
- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory

If page fault (i.e., `present` bit is cleared)
- 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

Mechanism for Continuing a Process

Continuing a process after a page fault is tricky
- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
  - When instruction is being fetched
  - When data is being loaded or stored
- Requires hardware support
  - precise interrupts: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

Complexity depends upon instruction set
- Can faulting instruction be restarted from beginning?
  - Example: `move +(SP), R2`
  - Must track side effects so hardware can undo
- Another Example: Early Apollo for 68000

Virtual Memory Policies

OS has two decisions on a page fault

- Page selection
  - When should a page (or pages) on disk be brought into memory?
  - Two cases
    - When process starts, code pages begin on disk
    - As process runs, code and data pages may be moved to disk
- Page replacement
  - Which resident page (or pages) in memory should be thrown out to disk?

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

Page Selection

When should a page be brought from disk into memory?

Request paging: User specifies which pages are needed for process

- Earliest systems: Overlays
- Problems:
  - Manage memory by hand
  - Users do not always know future references
  - Users are not impartial

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
- Advantages: Less work for user
- Problems: Pay cost of page fault for every newly accessed page

Page Selection Continued

Prepaging (anticipatory, prefetching): OS loads page into memory before page is referenced

- OS predicts future accesses (oracle) and brings pages into memory ahead of time
  - How?
  - Works well for some access patterns (e.g., sequential)
- Advantages: May avoid page faults
- Problems?

Hints: Combine demand or prepaging 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

Page Replacement

Which page in main memory should 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

Random: Replace any page at random

- Advantages: Easy to implement
- Works okay when memory is not severely over-committed

Page Replacement Continued

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 (circular buffer)
- Disadvantage: Some pages may always be needed

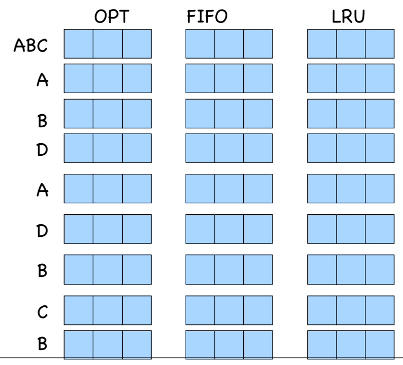
LRU: Replace page not used for longest time in past

- 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

Three pages of physical memory



Page Replacement Comparison

Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
  - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
- FIFO: Add more memory, usually have fewer page faults
  - Belady's anomaly: May actually have more page faults!

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 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 (Second Chance) 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 Algorithm Example

What if clock hand is sweeping very fast?

What if clock hand is sweeping very slow?

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

Add software counter ("chance")
- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment software counter if `use` bit is 0
- Replace when chance exceeds some specified limit

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

What if no Hardware Support?

What can the OS do if hardware does not have `use` bit (or `dirty` bit)?
- Can the OS "emulate" these bits?

Leading question:
- How can the OS get control (i.e., generate a trap) every time `use` bit should be set? (i.e., when a page is accessed?)