UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537 — A. Arpaci-Dusseau
Intro to Operating Systems — Spring 2000

Virtual Memory

Questions answered in these notes:

- How to run process w/ some of address space in physical mem?
- 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?
- How to determine how many processes can fit in memory?

Readings for this topic: Chapter 9

Motivation

Previous approach to memory management

- Required the user process to be completely loaded into memory
- Wasteful because of **locality of reference**

Locality of Reference

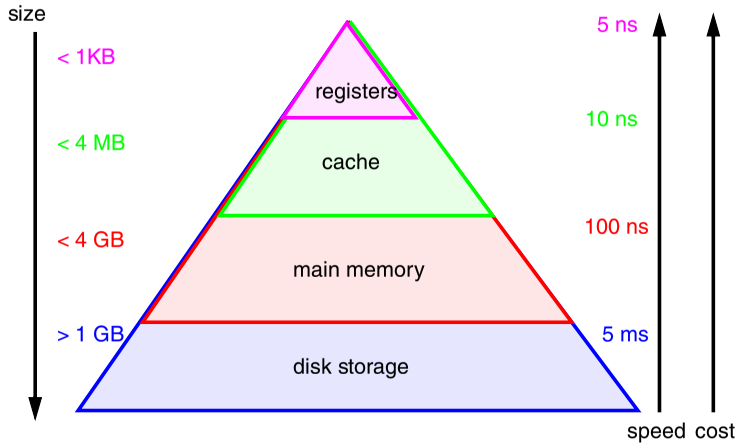
- Programs spend majority of time in small piece of code
- Knuth's estimate: 90% of the time in 10% of the code
- Process only needs small amount of address space at any moment

Keep unreferenced pages on slower, cheaper backing store: disk

- Process runs when not all pages are loaded in memory
- OS and Hardware: Produce illusion of a disk as fast as main memory

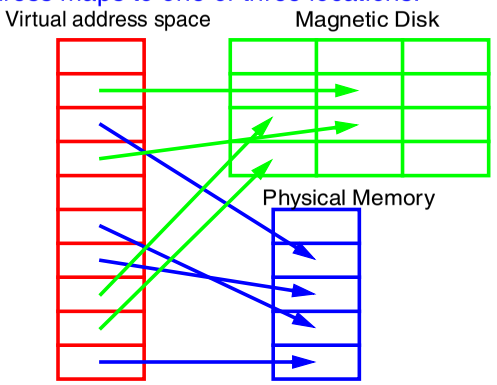
Memory Hierarchy

Levels of memory in computer system



Virtual Address Space

Virtual address maps to one of three locations:



a) Physical memory: Small, fast, expensive
b) Disk (backing store): Large, slow, cheap
c) Nothing (error): Free

Virtual Memory Operation

What happens when reference byte only in backing store?

- Hardware and software cooperate
  - Recognize location of page
  - Choose old page to replace
  - Bring page from disk into main memory

Extend the page tables with an extra bit: `present`

- Page not in memory has `present` bit cleared in page table entry
- If bit is cleared then referencing page results in a trap to OS
  - Trap: **page fault**

When page fault occurs

- Operating system selects page in memory to replace
- Operating system brings page into memory
- Page table is updated, `present` bit is set
- Process continues execution

Continuing Process

Continuing process is tricky

- Page fault may have occurred in middle of instruction
- Want page fault to be transparent to user process

Alternatives

- Can the instruction just be skipped?
- Can the instruction be restarted from the beginning?
  - What about instruction like: MOVE +(SP), R2?
  - bcopy?

Requires hardware support to restart instructions

- Without hardware support, is VM impossible?
  - No: Early Apollo approach for 68000
- Complexity depends upon instruction set

OS Decisions

Page selection

- When to bring page(s) from disk into memory

Page replacement

- Which resident page(s) in memory should be thrown out to disk

Page Selection Algorithms

Demand paging: Load page when page fault for it occurs

- Start up process with no pages loaded
- Wait until it absolutely **must** be in memory

Request paging: User specifies which pages are needed

- Users do not always know best
- Users are not impartial
- Example: Overlays

Prepaging: Load page before it is reference

- When one page is referenced, bring in next one
- Hard to do effectively without an **oracle**
- Example: Sequential read-ahead access patterns

Page Replacement Algorithms

OPT: Throw out page that won't be used for longest time in future

- Best algorithm arises if we can predict the future
- Not practical, but good for comparison

Random: Pick any page at random

- Easy to implement
- Works surprisingly well!

FIFO: Throw out page that has been in memory longest

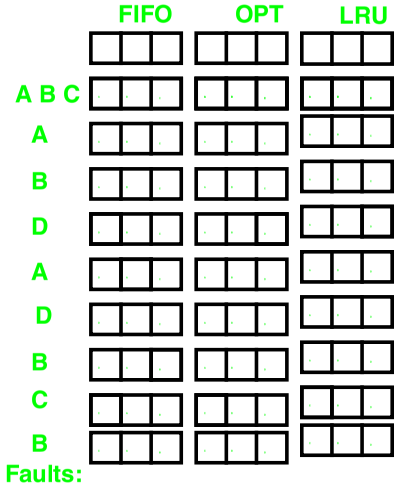
- Fair: All pages receive equal residency

LRU: Throw out the page that hasn't been used in longest time

- Use the past to predict the future
- With locality, LRU approximates OPT

Example of Page Replacement

Reference string: A B C A B D A D B C B:



Implementing LRU

Software Perfect LRU

- Keep ordered list of page references
- Memory reference: Move page to front of list
- Page replacement: Remove page from back of list

Hardware Perfect LRU

- Register for each page
- Memory reference: Store system clock into register
- Page replacement: Scan through pages to find one with oldest clock
- Disadvantage: Slow if a lot of memory pages

In practice, do not implement perfect LRU

- Settle for efficient approximation
- Find an old page, not necessarily the oldest
- LRU is an approximation anyway, so approximate a little more...

Clock Algorithm

Also called "second chance" algorithm (textbook)

Hardware

- Keep `use` (or `reference`) bit for each page frame
- Memory reference: Set bit

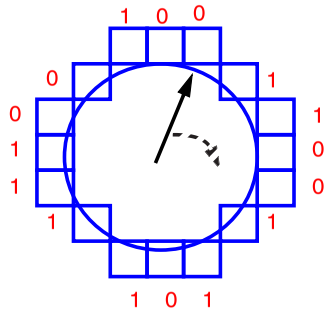
Operating System

- Page replacement: Look for page with use bit cleared
- Traverses all pages, clearing bits over time
- Examine bit to determine how often pages are referenced

Clock Algorithm

OS circulates through physical pages clearing `use` bits

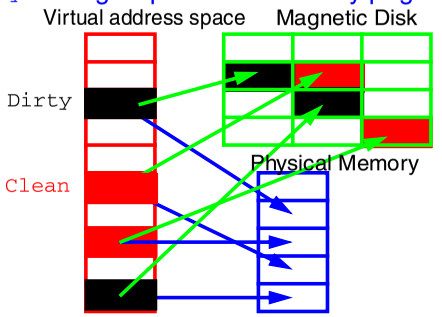
- Stop when find one is found that was zero
- Replace that page



- What if clock hand is sweeping very fast?
- What if clock hand is sweeping very slowly?

Clock Algorithm Extension

Use a `dirty` bit to give preference to dirty pages



- More expensive to throw out dirty pages
  Dirty pages must be written back to disk; Clean pages do not
- Problem: What if clean pages are frequently accessed?
  Example: Code pages
  Incentive: Modify pages to keep in memory

VMStat Example

Solaris uses clock algorithm

`vmstat -s` command gives relevant statistics

number of page faults
pages examined by the clock daemon
revolutions of clock hand
pages freed by the clock daemon

How to Allocate across Processes

Two approaches

- Global replacement
- Per-process replacement

Global replacement

- All pages from all processes are lumped in single replacement pool
- Each process competes with other processes for page frames

- Advantage: Flexibility of allocation, overall performance
- Disadvantage: One process can dominate entire system

How to Allocate across Processes

Per-process replacement

- Each process has a separate pool of pages
  - Fixed number of pages
  - Fixed fraction of physical memory
  - Proportional to size of address space
- Page fault in one process only replaces frame of that process

- Advantage: Relieves interference from other processes
- Disadvantage: Potentially inefficient allocation of resources

Per-user replacement

- Advantage: Users running more processes cannot hog memory
- Disadvantage: Inefficient allocation

Overcommitting Memory

Example:

- Set of processes frequently referencing **33** important pages
- Only **32** frames in physical memory

System repeats cycle

- Reference page not in memory
- Replace a page in memory with newly referenced page

Thrashing

- System reading and writing pages instead of executing useful instructions
- Average memory access time equals disk access time
- Illusion breaks: Memory appears slow as disk rather than disks appearing fast as memory

Add more processes, thrashing gets worse

Average Access Time

Simple calculation

- *H*: Percentage of references that hit page in main memory
- $C_{AccessMemory}$: Cost of referencing page in memory -- 100 ns
- $C_{PageFault}$: Cost of page fault -- 25 ms (25,000,000 ns)

$$HC_{AccessMemory} + (1-H)C_{PageFault}$$

Thrashing Example:

- 1 out of every 33 references misses --> *H=97%*

$$0.97(100) + (0.03)(25000000) \approx 750us$$

Small miss rates lead to unacceptable average access times

Motivation for Solution

System does not know when it has more work than it can handle

Page replacement policies

- Do not indicate that a page must not be thrown out of memory
- Only show which pages are better than others to replace (LRU)

Student's analogy to thrashing: Too many courses

- Solution: Drop a course

Operating system similar solution: Admission Control

- Determine how much memory each process needs
- Long-term scheduling policy:
  - Run only processes whose memory requirements can be satisfied
- What if memory needs of one process are too large?

Working Sets

Informal definition

- Collection of pages the process is referencing frequently
- Collection of pages that must be resident to avoid thrashing

Locality: Use recent requirements to predict future

Formal definition

- Pages referenced by process in last τ seconds of execution
- τ: the working set parameter

Page reference stream:

Interactions

Interaction with page replacement policy

- Ensure that working set remains in memory
- Pages outside working set may be discarded immediately

Interaction with long-term scheduler

- Process not executed unless working set resident in main memory

Balance Set

Examine memory requirements of all processes, not just one

Divide runnable processes into two groups

- Active: Working set is loaded
- Inactive: Working set migrates to disk

Balance set: Sum of working sets across all active processes

Interaction with long-term scheduler

- If balance set exceeds size of memory, move some processes to inactive set
- Policy for returning processes to active set

As working set changes, balance set must change as well

- What happens if the balance set increases too rapidly?

Possible Working Set Implementation

Must constantly update working set information

- What pages have be accessed in the last τ seconds?

Initial plan

- Store capacitor with each memory page
- Charge capacitor on every reference
- Capacitor discharges slowly if page is not referenced
- τ determined by the size of the capacitor

What are some problems with this plan?

Working Set Solution

Leverage use bits

OS maintains *idle time* each page

- Amount of CPU received by process since last access to page
- Periodically scan all resident pages of a process
  - If use bit is on, clear page's idle time
  - If use bit is off, add process' CPU time (since last scan) to idle time
  - Clear use bit
- How frequently to perform scan?
  - Few seconds or few minutes

Unresolved Questions

What should τ be?

- What if it's too large?
- What if it's too small?

What processes should compose balance set?

How do we compute working sets if pages are shared?

- Put jobs sharing pages in same balance set

How much memory in "balanced system"?

- How much is needed to keep the CPU busy?
- With working set, CPU may be idle even with runnable processes

Current Trends

Less critical with personal versus time-shared machines

- One user: Kill jobs when response gets bad
- Many users: OS must arbitrate

Memory is cheap --> Larger physical memory

- Page replacement algorithm is less important
  - Invoked less frequently
- Less hardware support for replacement policies
  - Software emulation of use and dirty bits

Larger page sizes (even multiple page sizes)

- Better TLB coverage
- Smaller page tables, less pages to manage
- More internal fragmentation

Current Trends

Allocation of backing store

- No longer a cache
- Possible: Smaller swap space than memory

Larger virtual address spaces

- 64 bits with 4K page: Max page table size 64K TB
- Sparse address spaces
- Inverted page tables

File I/O using the virtual memory system

- Memory mapped files: mmap()
  - Uses VM system for file caching
  - Page fault handling for reads/writes
- More sequential behavior (prefetching, hints)
  - madvise()

More Current Trends

Allocation of backing store

- No longer a cache

Larger virtual address spaces

- 64 bits with 4K page: Max page table size 64K TB
- Sparse address spaces
- Inverted page tables

File I/O using the virtual memory system

- Memory mapped files
  - Uses VM system for file caching
  - Page fault handling for reads/writes
- More sequential behavior (prefetching, hints)

Advanced Issues

Allocation in large-scale shared memory machines (CC-NUMA)

- Different access times to different pages in memory
- Processes may migrate to different CPUs
- Processes on different CPUs are sharing pages
- Location of page determines memory performance

Page to other types of backing store (not just disk)

- Page to remote memory in clusters
- Requires fast switch-based network (plenty of bandwidth)