Memory Management II
Virtual Memory

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
Background: memory hierarchy

- Levels of memory in computer system

![Memory Hierarchy Diagram]

- **Registers**
  - Size: few cycles
  - Speed: <1 cycle
  - Cost: very low

- **Cache**
  - Size: <100 ns
  - Speed: a few cycles
  - Cost: lower than memory

- **Memory**
  - Size: a few ms
  - Speed: slowest
  - Cost: highest

- **Disk**
  - Size: slowest
  - Speed: a few ms
  - Cost: moderate

---

3/27/13

COMS W4118. Spring 2013, Columbia University. Instructor: Dr. Kaustubh Joshi, AT&T Labs.
Virtual memory motivation

• Previous approach to memory management
  – Must completely load user process in memory
  – One large AS or too many AS ➔ out of memory

• Observation: locality of reference
  – Temporal: access memory location accessed just now
  – Spatial: access memory location adjacent to locations accessed just now

• Implication: process only needs a small part of address space at any moment!
  – Can load programs faster (don’t load everything)
  – Can fit more programs in memory (better utilization)
Linux Address Space Layout

1GB
- **Kernel space**
  - User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault
  - $0x00000000 == TASK_SIZE
  - Random stack offset
  - RLIMIT_STACK (e.g., 8MB)
  - Random mmap offset

3GB
- **Stack (grows down)**
  - Random stack offset
- **Memory Mapping Segment**
  - File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
  - Random mmap offset
- **Heap**
  - program break
  - brk
  - start_brk
  - Random brk offset
- **BSS segment**
  - Uninitialized static variables, filled with zeros.
  - Example: static char *userName;
  - end_data
  - start_data
  - Random bss offset
- **Data segment**
  - Static variables initialized by the programmer.
  - Example: static char *gonzo = “God’s own prototype”;
- **Text segment (ELF)**
  - Stores the binary image of the process (e.g., /bin/gonzo)
  - $0x0008048000
  - $0

Read: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
The Working Set Model

• Working set: set of memory addresses (pages) that the program needs in memory to make progress
  – Often set of pages program accesses in a short period of time

• Why does program need pages in main memory?
  – Instructions can only address main memory and registers
  – Accessed by same instruction
  – Accessed many times
  – Loops access a lot of memory

• Working usually much smaller than full program
  – Program does one thing at a time
  – Code for exception handling rarely accessed
  – Process migrates from one working set to another
  – Working sets may overlap
Locality In A Memory-Reference Pattern
Keeping working sets small

• Small changes to program = big changes to working set
  – Try to preserve locality in high performance code ("cache friendly")
  – Keep accesses related in time also related in space

• Example:
  – int data[1024][1024] of a 2d 1024x1024 byte array
  – Row major: each row is stored in one 4k page

Program1: for (j = 0; j <1024; j++)
    for (i = 0; i < 1024; i++)
        data[i][j] = 0;

Working set: 1024x1024 = 4MB

Program2: for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
        data[i][j] = 0;

Working set = 1024 = 4KB!
Virtual memory idea

• OS and hardware produce illusion of disk as fast as main memory, or main memory as large as disk

• Process runs when not all pages are loaded in memory
  – Only keep referenced pages in main memory
  – Keep unreferenced pages on slower, cheaper backing store (disk)
  – Bring pages from disk to memory when necessary
Virtual memory illustration

Virtual Memory

Page 0
Page 1
Page 2
Page 3

Physical Memory

Page 0
Page 1

Page 2
Page 3

Disk

Page table

3/27/13
COMS W4118. Spring 2013, Columbia University. Instructor: Dr. Kaustubh Joshi, AT&T Labs.
Virtual memory operations

• Detect reference to page on disk
• Recognize disk location of page
• Choose free physical page
  – OS decision: if no free page is available, must replace a physical page
• Bring page from disk into memory
  – OS decision: when to bring page into memory?
• Above steps need hardware and software cooperation
Detect reference to page on disk and recognize disk location of page

- Overload the present bit of page table entries

- If a page is on disk, clear present bit in corresponding page table entry and store disk location using remaining bits

- **Page fault**: if bit is cleared then referencing resulting in a trap into OS

- In OS page fault handler, check page table entry to detect if page fault is caused by reference to true invalid page or page on disk
Steps in handling a page fault

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Reset page table
5. Free frame
6. Restart instruction
Performance of Demand Paging

• Page Fault Rate $0 \leq p \leq 1$
  – if $p = 0$ no page faults
  – if $p = 1$, every reference is a fault

• Effective Access Time (EAT)
  $EAT = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$
Demand Paging Example

• Disparity in memory and disk access times is huge. E.g.,
  – Memory access time = 200 nanoseconds
  – Average page-fault service time = 8 milliseconds

• EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800

• If one out of 1,000 accesses faults, then EAT = 8.2 us, or 40x slower!

• If want performance degradation < 10 percent
  – 200 + 7,999,800 x p < 220, or 7,999,800 x p < 20
  – p < .0000025
  – Less than one page fault in every 400,000 memory accesses
OS decisions

• Page selection
  – When to bring pages from disk to memory?

• Page replacement
  – When no free pages available, must select victim page in memory and throw it out to disk
Page selection algorithms

- **Demand paging**: load page on page fault
  - Start up process with no pages loaded
  - Wait until a page absolutely must be in memory

- **Request paging**: user specifies which pages are needed
  - Requires users to manage memory by hand
  - Users do not always know best
  - OS trusts users (e.g., one user can use up all memory)

- **Prepaging**: load page before it is referenced
  - When one page is referenced, bring in next one
  - Do not work well for all workloads
    - **Difficult to predict future**
Working Sets and Page Fault Rates

- With pure demand paging

- Prepaging tries to smooth out bursts by predicting and fetching in the previous valley
Virtual Memory Gotchas

How to differentiate between access to empty regions vs. access to a not present page?
- Linux, keep a separate data structure to represent valid regions. Called vma (vm_area_struct)
- Could also use PTE bit

How to swap out a shared page mapped by multiple AS?
- Disable swapping (pin)
- Maintain reverse mapping
- Physical page to AS that maps the physical page
- Linux maintains rmap between vmas

Ref: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
Page replacement algorithms

- **Optimal**: throw out page that won’t be used for longest time in future
- **Random**: throw out a random page
- **FIFO**: throw out page that was loaded in first
- **LRU**: throw out page that hasn’t been used in longest time
Ideal curve of # of page faults v.s. # of physical pages
Evaluating page replacement algorithms

• Goal: fewest number of page faults

• A method: run algorithm on a particular string of memory references (reference string) and computing the number of page faults on that string

• In all our examples, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Optimal algorithm

- Throw out page that won’t be used for longest time in future

6 page faults

Problem: difficult to predict future!
First-In-First-Out (FIFO) algorithm

- Throw out page that was loaded in first

1 2 3 4 1 2 5 1 2 3 4 5

1 1 1 1 1 5 5 5 5 4 4
2 2 2 2 2 1 1 1 1 2 2
3 3 3 3 3 2 2 2 2 3 3
4 4 4 4 4 4 4 4 3 3

10 page faults

Problem: ignores access patterns
FIFO algorithm (cont.)

- Results with 3 physical pages

1 2 3 4 1 2 5 1 2 3 4 5

9 page faults

Problem: fewer physical pages → fewer faults!
belady anomaly
Least-Recently-Used (LRU) algorithm

- Throw out page that hasn’t been used in longest time. Can use FIFO to break ties

```
1 2 3 4 1 2 5 1 2 3 4 5
```

8 page faults

Advantage: with locality, LRU approximates Optimal
Implementing LRU: hardware

- A counter for each page
- Every time page is referenced, save system clock into the counter of the page
- Page replacement: scan through pages to find the one with the oldest clock
- **Problem**: have to search all pages/counters!
Implementing LRU: software

• A doubly linked list of pages

• Every time page is referenced, move it to the front of the list

• Page replacement: remove the page from back of list
  – Avoid scanning of all pages

• **Problem**: too expensive
  – Requires 6 pointer updates for each page reference
  – High contention on multiprocessor
LRU: concept vs. reality

• LRU is considered to be a reasonably good algorithm

• Problem is in implementing it efficiently
  – Hardware implementation: counter per page, copied per memory reference, have to search pages on page replacement to find oldest
  – Software implementation: no search, but pointer swap on each memory reference, high contention

• In practice, settle for efficient approximate LRU
  – Find an old page, but not necessarily the oldest
  – LRU is approximation anyway, so approximate more
Clock (second-chance) algorithm

• Goal: remove a page that has not been referenced recently
  – good LRU-approximate algorithm

• Idea
  – A reference bit per page
  – Memory reference: hardware sets bit to 1
  – Page replacement: OS finds a page with reference bit cleared
  – OS traverses all pages, clearing bits over time
Clock algorithm implementation

• Combining FIFO with LRU: give the victim page that FIFO selects a second chance

• Keep pages in a circular list = clock

• Pointer to next victim = clock hand

• To replace a page, OS examines the page pointed to by hand
  – If ref bit == 1, clear, advance hand
  – Else return current page as victim
A single step in Clock algorithm

![Diagram](a.png)
Clock algorithm example

10 page faults

Advantage: simple to implement!
Clock algorithm extension

• Problem of clock algorithm: does not differentiate dirty v.s. clean pages

• Dirty page: pages that have been modified and need to be written back to disk
  – More expensive to replace dirty than clean pages
  – One extra disk write (about 5 ms)
Clock algorithm extension (cont.)

- Use **dirty** bit to give preference to dirty pages

- On page reference
  - Read: hardware sets **reference** bit
  - Write: hardware sets **dirty** bit

- Page replacement
  - reference = 0, dirty = 0 → **victim page**
  - reference = 0, dirty = 1 → **skip** (don’t change)
  - reference = 1, dirty = 0 → reference = 0, dirty = 0
  - reference = 1, dirty = 1 → reference = 0, dirty = 1
  - advance hand, repeat
  - If no victim page found, run **swap daemon** to flush **unreferenced dirty pages** to the disk, repeat
Summary of page replacement algorithms

• **Optimal**: throw out page that won’t be used for longest time in future
  – Best algorithm if we can predict future
  – Good for comparison, but not practical

• **Random**: throw out a random page
  – Easy to implement
  – Works surprisingly well. Why? Avoid worst case
  – Random

• **FIFO**: throw out page that was loaded in first
  – Easy to implement
  – Fair: all pages receive equal residency
  – Ignore access pattern

• **LRU**: throw out page that hasn’t been used in longest time
  – Past predicts future
  – With locality: approximates Optimal
  – Simple approximate LRU algorithms exist (Clock)
Page-Buffering

• Keep pool of free frames, always
  – Frame always available when needed
  – Read page into free frame
  – Select victim to evict and add to free pool
  – When convenient, evict victim

• Keep list of modified pages
  – When disk idle, write pages there and set to non-dirty

• Note and keep free pool contents intact
  – If referenced again before reused, no need to reload from disk
  – Useful if wrong victim frame was selected
Thrashing

• What if we need more pages regularly than we have?
  – Page fault to get page
  – Replace existing frame
  – But quickly need replaced frame back

• Leads to:
  – High page fault rate
  – Lots of I/O wait
  – Low CPU utilization
  – No useful work done

• Thrashing ≡ system busy just swapping pages in and out
Effects of Thrashing

![Graph showing the effects of thrashing on CPU utilization vs. degree of multiprogramming. The graph illustrates that as the degree of multiprogramming increases, CPU utilization increases up to a certain point, after which it decreases due to thrashing.]
Memory-Mapped Files

• Treat files like memory by **mapping** a disk block to a memory page
  – `mmap()` syscall maps file into memory region

• File blocks initially loaded using demand paging
  – Page-sized chunk of the file read into physical page
  – Subsequent accesses to chunk treated as ordinary memory accesses

• Lazily flush writes to disk
  – Periodically, e.g., when pager scans for dirty pages
  – At file close() time
Memory-Mapped Files

• Benefits of memory mapped files
  – Simplify/speed file access compared to read()/write() syscalls
  – Allows several processes to map same file to facilitate memory sharing (useful for binaries)

• Paging and file I/O often tightly intertwined
  – Swapping can use original file as backing store (if not dirty)
  – COW can be used to quickly create “clone” of file
  – Memory mapped files can be used for shared memory

• Some OSes use mmap internally for all I/O
  – Process still does read() and write()
  – Kernel maps file into kernel address space
  – Copies data to and from kernel space and user space
Memory Mapped Files
Paging (or segmentation) and I/O

• DMA devices directly copy data to memory
  – Does I/O device understand paging?
  – Need IOMMU (newer CPUs)
  – Else, OS must program DMA itself using physical addresses
  – Must do permissions checks
  – Pin pages into memory to prevent swapping out while DMA ongoing
Non-Uniform Memory Access

• So far all memory accessed equally

• NUMA – speed of access to memory varies
  – E.g., many system boards containing CPUs and memory, interconnected over a system bus
  – Memory on same board is “fast”, other boards, “slow”

• Allocate memory close to CPU on which thread runs
  – Use processor affinity to keep threads on same CPU
  – E.g.: Solaris “lgroups”
    • Groups of CPU/memory with low latency
    • Scheduler/pager schedule all threads and memory for a process within the lgroup
Current trends in memory management

• Virtual memory is less critical now
  – Personal computer v.s. time-sharing machines
  – Memory is cheap ➔ Larger physical memory

• Virtual to physical translation is still useful
  – “All problems in computer science can be solved using another level of indirection” David Wheeler

• Larger page sizes (even multiple page sizes)
  – Better TLB coverage
  – Smaller page tables, less page to manage
  – Internal fragmentation: not a big problem

• Larger virtual address space
  – 64-bit address space
  – Sparse address spaces

• File I/O using the virtual memory system
  – Memory mapped I/O: `mmap()`
Backup Slides
Problem with LRU-based Algorithms

• LRU ignores frequency
  – Intuition: a frequently accessed page is more likely to be accessed in the future than a page accessed just once
  – Problematic workload: scanning large data set
    • 1 2 3 1 2 3 1 2 3 1 2 3 ...  (pages frequently used)
    • 4 5 6 7 8 9 10 11 12 ...  (pages used just once)

• Solution: track access frequency
  – Least Frequently Used (LFU) algorithm
    • Expensive
  – Approximate LFU: LRU-2Q
Problem with LRU-based Algorithms (cont.)

• LRU doesn’t handle repeated scan well when data set is bigger than memory
  – 4-frame memory with 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

• Solution: Most Recently Used (MRU) algorithm
  – Replace most recently used pages
  – Best for repeated scans
Virtual memory illustration

page 0
page 1
page 2

virtual memory

memory map

physical memory