Chapter 4
Memory Management

4.1 Basic memory management
4.2 Swapping
4.3 Virtual memory
4.4 Page replacement algorithms
4.5 Modeling page replacement algorithms
4.6 Design issues for paging systems
4.7 Implementation issues
4.8 Segmentation

Memory Management

• Want memory that is
  – large
  – fast
  – non volatile

• Memory hierarchy
  – small amount of fast, expensive memory – cache
  – some medium-speed, medium price main memory
  – gigabytes of slow, cheap disk storage

• Memory manager handles the memory hierarchy
Basic Memory Management
Monoprogramming without Swapping or Paging

Three simple ways of organizing memory
- an operating system with one user process

Multiprogramming with Fixed Partitions

- Fixed memory partitions
  - separate input queues for each partition
  - single input queue
Modeling Multiprogramming

CPU utilization as a function of number of processes in memory

Analysis of Multiprogramming System Performance

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval
Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes’ partitions

- Use base and limit values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error

Swapping (1)

Memory allocation changes as
- processes come into memory
- leave memory
Shaded regions are unused memory
Swapping (2)

- Allocating space for growing data segment
- Allocating space for growing stack & data segment

Memory Management with Bit Maps

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list
Memory Management with Linked Lists

Four neighbor combinations for the terminating process X

Swapping: Free space

Key operation: search for a hole (and merge)
- First Fit, Next Fit (worse)
- Best Fit, Worst Fit (largest hole)
- Quick Fit

- Info maintained on Bitmaps, Linked Lists
virtual…

(...)

2. Existing in the mind, especially as a product of the imagination. (American Heritage)

1. Being actually such in almost every respect; "the once elegant temple lay in virtual ruin";
2. Being such in essence or effect though not in actual fact;
"a virtual dependence on charity"; "a virtual revolution"; "virtual reality." (ultralingua.net)

Virtual Memory

• virtual addresses, space - logical/pages
• real addresses, space - physical/frames

Memory Management Unit (MMU)
• translates logical to physical addresses
• *paging*: assign a page (chunk of contiguous logical addresses) to a frame
  – supported by page table
  – variants, costs
Virtual Memory
Paging (1)

The position and function of the MMU

Paging (2)
The relation between virtual addresses and physical memory addresses given by page table
Internal operation of MMU with 16 4 KB pages

- 32 bit address with 2 page table fields
- Two-level page tables
Page Tables (3)

Typical page table entry

TLBs – Translation Lookaside Buffers

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

A TLB to speed up paging
Inverted Page Tables

Comparison of a traditional page table with an inverted page table

Page Replacement Algorithms

• Page fault forces choice
  – which page must be removed
  – make room for incoming page

• Modified page must first be saved
  – unmodified just overwritten

• Better not to choose an often used page
  – will probably need to be brought back in soon
Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable

- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical

Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class
FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory

- Page at beginning of list replaced

- Disadvantage
  - page in memory the longest may be often used

Second Chance Page Replacement Algorithm

- Operation of a second chance
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, A has R bit set
    (numbers above pages are loading times)
The Clock Page Replacement Algorithm

Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time

- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!

- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter
Simulating LRU in Software (1)

<table>
<thead>
<tr>
<th>Page</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3

Simulating LRU in Software (2)

- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)
The Working Set Page Replacement Algorithm (1)

• The working set is the set of pages used by the $k$ most recent memory references
• $w(k,t)$ is the size of the working set at time, $t$

The Working Set Page Replacement Algorithm (2)

The working set algorithm
The WSClock Page Replacement Algorithm

Operation of the WSClock algorithm

Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults

Stack Algorithms

State of memory array, $M$, after each item in reference string is processed
The Distance String

Probability density functions for two hypothetical distance strings

- Computation of page fault rate from distance string
  - the $C$ vector
  - the $F$ vector
Design Issues for Paging Systems
Local versus Global Allocation Policies (1)

- Original configuration
- Local page replacement
- Global page replacement

Local versus Global Allocation Policies (2)

Page fault rate as a function of the number of page frames assigned
Load Control

- Despite good designs, system may still thrash

- When PFF algorithm indicates
  - some processes need more memory
  - but no processes need less

- Solution:
  Reduce number of processes competing for memory
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming

Page Size (1)

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory

- Disadvantages
  - programs need many pages, larger page tables
Page Size (2)

Overhead due to page table and internal fragmentation

overhead = \( \frac{s}{p} + \frac{p}{2} \)

optimal page size \( - p = \sqrt{2se} \)

\( s = \) average process size in bytes

\( p = \) page size in bytes

\( e = 1 \) page entry in bytes

\( e.g., s = 1MB, e = 8B, p = 4KB \)

Separate Instruction and Data Spaces

• One address space
• Separate I and D spaces
Two processes sharing same program sharing its page table

Shared Pages (2)

- Easy with separate Instruction / Data spaces
- Problem: eviction of a process might evict its shared pages - need to know (GC anyone)?
- Share Data pages? cf. Unix’s fork
  - each proc its page table, same pages, readonly
  - when a proc writes: trap. “copy on write”
- Anyway, always need free frames
Cleaning Policy

• Need for a background process, paging daemon
  – periodically inspects state of memory

• When too few frames are free
  – selects pages to evict using a replacement algorithm

• It can use same circular list (clock)
  – as regular page replacement algorithm but with diff ptr

Virtual Memory Interface

• VM usually “transparent” (to whom?)
  Alternative: programmers see memory map
• specify that processes share memory
  – share memory = high bandwidth (problems?)
  – message passing: map/unmap pages
Implementation Issues
Operating System Involvement with Paging

Four times when OS involved with paging
1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages

Page Fault Handling (1)

1. Hardware traps to kernel
   Save PC on Stack, possible instruction state in CPU reg
2. General registers saved
   Assembly routine saves CPU registers, calls OS
3. OS determines which virtual page needed
   Possibly inspect instruction
4. OS checks validity of address, seeks page frame
   If invalid, kill process; if no free frame, choose victim
5. If selected frame is dirty, write it to disk
   Suspend faulting process, run another while I/O completes
Page Fault Handling (2)

6. OS schedules I/O to bring new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored; Program continues

Other Implementation Issues

• Instruction rollback (backup)
• Locking pages in memory (pinning)
• Backing store (static / dynamic)
• Memory Manager at User Level
  – Separation of Policy and Mechanism
Instruction Backup

MOVE.L #6(A1), 2(A0)

16 Bits

| 1000 | MOVE |
| 1002 | 6    |
| 1004 | 2    |

 Opcode
 First operand
 Second operand

An instruction causing a page fault

Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Proc issues call for read from device into buffer
  - while waiting for I/O, another processes starts up
  - has a page fault
  - buffer for the first proc may be chosen to be paged out
- Need to specify some pages locked
  - exempted from being target pages
Backing Store

(a) Paging to static swap area
(b) Backing up pages dynamically

Separation of Policy and Mechanism

Page fault handling with an external pager
Segmentation (1)

- One-dimensional address space with growing tables
- One table may bump into another

Segmentation (2)

Allows each table to grow or shrink, independently
Segmentation (3)

- Segment: Linear sequence of addresses
- Each can grow/shrink independently
- Addresses: (segment #, address within it)
- All segments start at 0 (simpler to gen.)
- Distinct protections, shared libraries easier

Consider *paging* versus *segmentation* issues
- size, fragmentation, relocation…

Segmentation (4)

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>

Comparison of paging and segmentation
Implementation of Pure Segmentation

(a)-(d) Development of checkerboarding
(e) Removal of the checkerboarding by compaction

MULTICS segmentation/paging

- $2^{18}$ segments, each 65536 words long
- each segment a virtual memory (so paged)
  - paging: uniform page size, ability to keep some of it off main memory
  - segmentation: easier programming, protection, sharing - modularity.
- each program has a segment table (itself a paged segment!)
Segmentation with Paging: MULTICS (1)

- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths.

Segmentation with Paging: MULTICS (2)

A 34-bit MULTICS virtual address
MULTICS memory reference

1. segment # points to segment descriptor
2. check if segm page table is in memory, if not a segment fault happens; protection also checked
3. check page table entry for page; if not in memory, page fault.
4. Offset is added, read/store happens

Segmentation with Paging: MULTICS (3)

Conversion of a 2-part MULTICS address into a main memory address
Segmentation with Paging: MULTICS (4)

- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

Segmentation with Paging: Pentium (1)

A Pentium selector
Segmentation with Paging: Pentium (2)

- Pentium code segment descriptor
- Data segments differ slightly

Segmentation with Paging: Pentium (3)

Conversion of a (selector, offset) pair to a linear address
Segmentation with Paging: Pentium (4)

Mapping of a linear address onto a physical address

Segmentation with Paging: Pentium (5)

Protection on the Pentium