Modeling Page Replacement: Stack Algorithms

Reference string: 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

State of memory array, $M$, after each item in reference string is processed.

Design Issues for Paging Systems

- Local page replacement vs. global page replacement
  - How memory be allocated among the competing processes?

(a) Original configuration. (b) Local page replacement. (c) Global page replacement.
Design Issues for Paging Systems (2)

- Local page replacement: static allocation
  - What if the working set of some process grows?
  - What if the working set of some process shrinks?
  - The consideration: thrashing and memory utilization

- Global page replacement: dynamic allocation
  - How many page frames assigned to each process
    - Keep monitoring the working set size
    - Allocating an equal share of available page frames
    - Allocating a proportional share of available page frames
    - Or hybrid allocation, using PFF (page fault frequency)

Page Fault Frequency (PFF)

- PFF: control the size of allocation set of a process
  - when and how much to increase or decrease a process' page frame allocation

- Replacement: what page frames to be replaced

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

° Despite good designs, system may still have thrashing
  • When combined working sets of all processes exceed the capacity of memory

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

° Solution:
  • swap one or more to disk, divide up pages they held
  • reconsider degree of multiprogramming
    - CPU-bound and I/O-bound mixing

Reduce number of processes competing for memory

Page Size (1)

Small page size

° Advantages
  • less unused program in memory (due to internal fragmentation)
  • better fit for various data structures, code sections

° Disadvantages
  • programs need many pages, larger page tables
  • Long access time of page (compared to transfer time)
  • Also Maybe more paging actions due to page faults
Page Size (2)

- Tradeoff: overhead due to page table and internal fragmentation

Where
- $s =$ average process size in bytes
- $p =$ page size in bytes
- $e =$ page entry size in bytes

\[
\text{overhead} = \frac{s \cdot e}{p} + \frac{p}{2}
\]

Optimized/minimized when $f'(p) = 0$

\[
p = \sqrt{2 \cdot s \cdot e}
\]

Separate Instruction and Data Spaces

- What if the single virtual address space is not enough for both program and data?
  - Doubles the available virtual address space, and ease page sharing of multiple processes
  - Both addr. spaces can be paged, each has own page table

Is it an issue in today’s 64-bit machines?
Shared Pages

- How to allow multiple processes share the pages when running the same program at the same time?
  - One process has its own page table(s)

Two processes sharing same program sharing its I-page table

Shared Pages (2)

- What to do when a page replacement occurs to a process while other processes are sharing pages with it?
- How share data pages, compared to share code pages?
- UNIX fork() and copy-on-write
  - Generating a new page table point to the same set of pages, but not duplicating pages until...
  - A violation of read-only causes a trap
Cleaning Policy

- Need for a background process, *paging daemon*
  - periodically inspects state of memory
  - To ensure plenty of free page frames

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

Implementation Issues

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 (as invalidating the cache)

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. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS brings schedules 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
11. Program continues

Instruction Backup

The instruction causing the page fault is stopped part way. After OS has fetched the page needed, it must restart the instruction, but where exactly the page fault was due to?

- The value of the PC at the time of trap depends on which operand faulted and how the CPU's microcode has been implemented, which is hard for OS to tell
- Hidden internal register copies PC

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

16 Bits

1000

1002

1004

MOVE

6

2

Opcode

First operand

Second operand

An instruction causing a page fault, say if PC = 1002; how OS knows the content of 1002 is an opcode or an operand? or where the instruction begins?
**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
  - That process has a page fault
  - If global page replacement, buffer for the first proc may be chosen to be paged out (in particular, partial DMA transfer)
- Need to specify some pages locked
  - Pinning: lock pages engaged in I/O in memory so that they will not be removed (from being target pages)

**Backing Store – Disk Management**

- How to allocate page space on the disk in support of VM?
  - Static swap area (pages copied): adding the offset of the page in the virtual address space to the start of the swap area
  - Dynamic swapping (page no copied, a table-per-process needed)
Segmentation (1)

- Why to have two or more separate virtual address spaces?

How to free a programmer from the issues of expanding and contracting tables?

- One-dimensional address space with growing tables for compiling, one table may bump/interfere into another
Segmentation (2)

- Segments: many independent virtual address spaces
  - A logical entity, known and used by the programmer
  - Two-dimensional memory: the program must supply a two-part address, a segment number and an address with an segment

How about sharing, such as a shared library?

Comparison of Segmentation and Paging

<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>

Modularity
Implementation of Pure Segmentation

- An essential difference of paging and segmentation
  - Segments have different sizes while pages are fixed size!

(a) Segment 4 (7K)
(b) Segment 4 (7K)
(c) (3K)
(d) (3K)
(e) (10K)

What is the key difference of segmentation and swapping?
(a)-(d) Development of checkerboarding (external fragmentation) in physical memory, if segments are small; (e) Removal of the checkerboarding by compaction

Segmentation with Paging: MULTICS

- What if the memory is not large enough for a single segment?

MULTICS (Honeywell 6000)
- Paged segment with word (4B) addressing
- Multi-dim VM up to 2^18 segments, each up to 64K (32-bit) words
- 34-bit virtual address (seg #, page #, page offset)
- Physical memory 16M words (24-bit physical address)
- Page size: 1024 words; or 64 words (64-word alignment)
- 18-bit segment number for page table address

A 34-bit MULTICS virtual address

<table>
<thead>
<tr>
<th>Segment number</th>
<th>Page number</th>
<th>Offset within the page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>
MULTICS Virtual Memory

- One descriptor segment and 36-bit segment descriptors
  - What if the page table of a segment is not in the memory?

**How many page tables?**

Descriptors segment points to page tables. Segment descriptor.

MULTICS Virtual Address $\rightarrow$ Physical Address

MULTICS virtual address

What if the descriptor segment is paged (often it is)?

How to speed up the searching & conversion?

Conversion of a 2-part MULTICS address into a main memory address
MULTICS TLB

Comparison field

<table>
<thead>
<tr>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
</tr>
</tbody>
</table>

*Simplified version* of the MULTICS TLB (LRU replacement), which has 16 most recently referenced pages.

Segmentation with Paging: Pentium (Self-Reading)

- Segmentation in Pentium resembles MULTICS
  - But 16K segments, each up to 1B words
Summary

° Mono-programming -> Multi-programming

° Swapping

° Virtual Memory
  • Paging
  • Page replacement algorithms
  • Design and implementation issues

° Segmentation

° More reading: Chapter 4.1 - 4.9