Segmenting and Paging of Memory

Segmentation and paging are ways of

- partitioning memory for convenient allocation
- reorganizing memory for convenient usage

How?

- relocation via address translation
- protection via matching operations with objects

Result: a logically-organized memory

Implications

Offers possibility that not all pieces need be in memory

- need only piece being referenced
- other pieces can be on disk
- bring pieces in only when needed

Result: illusion that there is more memory than there is

What's needed to support this idea?

- a way to identify whether a piece is in memory
- a way to bring in pieces (from where, to where?)
- relocation (which we have)
From Logical to Virtual Memory

Logical memory becomes virtual memory
- still logical (separate organization from physical)
- memory exists, for all practical purposes

Virtual Memory: illusion of a large physical memory
- keep only portion of logical memory in physical
- rest is kept on disk (larger, slower, cheaper)
- unit of memory is segment or page (or both)

Logical address space becomes virtual address space

Sample Contents of Page Table Entry

<table>
<thead>
<tr>
<th></th>
<th>Valid?</th>
<th>Ref?</th>
<th>Mod?</th>
<th>Prot: rwx</th>
<th>Frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>rwx</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valid: is this entry valid (is page in physical memory)?
Ref: has this page been referenced yet?
Mod: has this page been modified (dirty)?
Prot: what are the allowable operations?
Frame: what frame is this page in (assuming in memory)?

Address Translation and Page Faults

Find entry by using page number to index into page table

If valid bit is off, page fault
- Trap into operating system
- Find page on disk (kept in kernel data structure)
- Read it into a free frame
  - may need to make room: page replacement
- Record frame number in page table entry, set valid
- Retry instruction (return from page-fault trap)

Virtual Memory Based on Paging

Of all pages in virtual memory
- All of them reside on disk
- Some reside in physical memory (which ones?)
Faults under Segmentation/Paging

Virtual address: <segment #, page #, offset>

Index segment # into segment table (to get page table)
  • may get a segment fault

Check bounds (is page # < bound?)
  • may get a segmentation violation

Use page # to index into page table (to get frame #)
  • may get a page fault

Concatenate frame # and offset to get physical address

Problem: Faults are Very Expensive

Disk: 5-6 orders magnitude slower than memory
  • very expensive; but if very rare, tolerable

Example
  • memory access time: 100 nsec
  • disk access time: 10 msec
  • $p = \text{page fault probability}$
  • effective access time: $100 + p \times 10,000,000 \text{ nsec}$
  • if $p = 0.1\%$, effective access time = 10,100 nsec

Principle of Locality

Not all pieces are referenced uniformly over time
  • make sure most referenced pieces in memory
  • if not, thrashing: constant fetching of pieces

References cluster in time/space
  • will be to same or neighboring areas
  • allows prediction based on past

Policies for Virtual Memory

Fetch policy: when to bring in, how many, which ones
  • demand paging vs. prepaging

Placement policy: where to place in memory
  • relevant for segmentation, irrelevant for paging

Replacement policy: which to remove to make room
  • resident set management, replacement

Cleaning policy: when to write out
  • demand cleaning vs. precleaning
Page Replacement

Goal: kick out page outside of locality of reference

Page replacement is about
• which page(s) to kick out
• when to kick them out

How to do it in the cheapest way possible
• least amount of additional hardware
• least amount of software overhead

Basic Page Replacement Algorithms

FIFO: select page that is oldest
• simple: use frame ordering
• doesn’t perform very well (oldest may be popular)

OPT: select page that to be used furthest in future
• optimal, but requires future knowledge
• establishes best case, good for comparisons

LRU: select page that was least recently used
• predict future based on past; works given locality
• costly: time-stamp pages each access, find least

Approximating LRU: Clock Algorithm

Idea: select page that is old and also not recently used
• Clock (aka second change) is approximation of LRU

Hardware support: reference bit
• associated with each frame is a reference bit
• actually, ref bit contained in page table entry

How reference bit is used
• when frame filled with page, bit is set to 0 (by OS)
• if frame is accessed, bit is set to 1 (by hardware)

Another Example

Ref string: 2 3 2 1 5 2 4 5 3 2 5 2

FIFO 2* 2 2 5* 5 5 5 3* 3 3 3
3 faults 1* 1 1 4* 4 4 4 4 2*

OPT 2* 2 2 2 2 4* 4 4 2* 2 2
3 faults 3* 3 3 3 3 3 3 3 3 3 3

LRU 2* 2 2 2 2 2 3* 3 3 3 3
4 faults 3* 3 3 5* 5 5 5 5 5 5 5

Another Example

Ref string: 2 3 2 1 5 2 4 5 3 2 5 2

FIFO 2* 2 2 5* 5 5 5 3* 3 3 3
3 faults 1* 1 1 4* 4 4 4 4 2*

OPT 2* 2 2 2 2 4* 4 4 2* 2 2
3 faults 3* 3 3 3 3 3 3 3 3 3 3

LRU 2* 2 2 2 2 2 3* 3 3 3 3
4 faults 3* 3 3 5* 5 5 5 5 5 5 5

Another Example

Ref string: 2 3 2 1 5 2 4 5 3 2 5 2

FIFO 2* 2 2 5* 5 5 5 3* 3 3 3
3 faults 1* 1 1 4* 4 4 4 4 2*

OPT 2* 2 2 2 2 4* 4 4 2* 2 2
3 faults 3* 3 3 3 3 3 3 3 3 3 3

LRU 2* 2 2 2 2 2 3* 3 3 3 3
4 faults 3* 3 3 5* 5 5 5 5 5 5 5

Another Example

Ref string: 2 3 2 1 5 2 4 5 3 2 5 2

FIFO 2* 2 2 5* 5 5 5 3* 3 3 3
3 faults 1* 1 1 4* 4 4 4 4 2*

OPT 2* 2 2 2 2 4* 4 4 2* 2 2
3 faults 3* 3 3 3 3 3 3 3 3 3 3

LRU 2* 2 2 2 2 2 3* 3 3 3 3
4 faults 3* 3 3 5* 5 5 5 5 5 5 5
**How Clock Works**

Arrange all frames in a circle (clock)

Clock hand points to next frame to consider

Upon page fault, if frame needed, do until frame found
  - if ref bit 0, select this frame
  - else set ref bit to 0
  - advance clock hand
  - if frame was found, break out of loop (else repeat)

If frame found had modified page, must first write out

---

**Example of Clock**

Ref string: 5 9 7 1 9 5 9

Reference page 5: page fault (unavoidable)
  - Hand points to an unreferenced page: use it
  - Advance hand

---

**Example of Clock, continued**

Ref string: 5 9 7 1 9 5 9

Reference page 9: page fault (unavoidable)
  - Hand points to an unreferenced page: use it
  - Advance hand

---

**Example of Clock, continued**

Ref string: 5 9 7 1 9 5 9

Reference page 7: page fault (unavoidable)
  - Hand points to an unreferenced page: use it
  - Advance hand
Example of Clock, continued

Ref string:  5  9  7  1  9  5  9

Reference page 1: page fault (1)
• Hand points to a referenced page: skip it
• Set ref bit to 0, advance hand, try again

Example of Clock, continued

Ref string:  5  9  7  1  9  5  9

Trying to find unreferenced page
• Hand points to a referenced page: skip it
• Set ref bit to 0, advance hand, try again

Example of Clock, continued

Ref string:  5  9  7  1  9  5  9

Trying to find unreferenced page
• Hand points to a referenced page: skip it
• Set ref bit to 0, advance hand, try again

Example of Clock, continued

Ref string:  5  9  7  1  9  5  9

Trying to find unreferenced page
• Hand points to an unreferenced page: use it
• Advance hand
Example of Clock, continued

Ref string: 5 9 7 1 9 5 9

Reference page 9
- Page 9 is already in memory: no page fault
- OS does nothing, but hardware sets ref bit to 1

Example of Clock, continued

Ref string: 5 9 7 1 9 5 9

Reference page 5: page fault (2)
- Hand points to a referenced page: skip it
- Set ref bit to 0, advance hand, try again

Example of Clock, continued

Ref string: 5 9 7 1 9 5 9

Trying to find unreferenced page
- Hand points to an unreferenced page: use it
- Advance hand

Example of Clock, continued

Ref string: 5 9 7 1 9 5 9

Reference page 9
- Page 9 already in memory: no page fault
- OS does nothing, but hardware sets ref bit to 1
### Comparing to OPT, LRU, FIFO

<table>
<thead>
<tr>
<th></th>
<th>FIFO</th>
<th>OPT</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref string</td>
<td>5 9 7 1 9 5 9</td>
<td>5* 5 1* 1 1 1</td>
<td>5* 5 1* 1 1 1</td>
</tr>
<tr>
<td>3 faults</td>
<td>9* 9 9 9 5* 5</td>
<td>7* 7 7 7 9*</td>
<td>7* 1* 1 1 1</td>
</tr>
<tr>
<td>1 faults</td>
<td>9* 9 9 9 9 9</td>
<td>7* 1* 1 1 1</td>
<td>7* 1* 1 1 1</td>
</tr>
<tr>
<td>2 faults</td>
<td>9* 9 9 9 9 9</td>
<td>7* 7 7 5* 5</td>
<td>7* 7 7 5* 5</td>
</tr>
</tbody>
</table>

### Two-Handed Clock

- **Front hand**
  - sets page ref bits to zero
- **Back hand**
  - selects unreferenced pages

**Parameters**
- scan rate
- hand gap

### Resident Set Management

Resident set: pages of process resident in memory
- how big should resident set be? Which pages?
- who provides frame (same process or another)?

**Local**:
- limit frame selection to requesting process
- isolates effects of page behavior on processes
- inefficient: some processes have unused frames

**Global**:
- select any page frame (from any process)
- efficient: resident sets grow/shrink accordingly
- no isolation: process can negatively affect another

### Multiprogramming Level

- Smaller resident sets implies more processes in memory
- increases multiprogramming level
- good for processor utilization
- however, beyond some point, thrashing occurs

Resident set should contain the working set
A Process’s Phases of Locality

Process memory reference pattern is a cycle
- period of locality: few faults outside working set
- locality change: many faults, change in working set

Denning’s Working Set Model

Working set: \( W(t, \Delta) \)
- pages referenced during last delta (process time)

Add/remove pages according to \( W(t, \Delta) \)
If working set cannot be in memory, swap process out
Problem: difficult to implement
- must timestamp pages in working set
- must determine if timestamp older than \( t - \Delta \)
- must determine \( \Delta \)

Monitor Page Fault Frequency

If frequency too high, working set not present
- give process more frames; if none, swap out
If frequency too low, resident set has too many pages
- take away page frames
Problem: no concept of transitions in locality

Page Size: How Large Should It Be?

Waste due to internal fragmentation: \( p/2 \) per process
- small \( p \) reduces waste: use small page size

Page table size: proportional to number of pages \( M/p \)
- large \( p \) reduces page table size

TLB hit rate: proportional to \( p \times \) number of entries
- large \( p \) increases hit rate

Given reduction in memory costs, favor large page sizes
- VAX (’70s technology): 512B; today: 8KB-32KB
Managing Disk Space

Disk is used for swapping/paging

**Swapping:** allocate region for entire process
- variable allocation: e.g., use first fit
- within region, can transfer individual pages
- determine address by region start + page number

**Paging (assuming no per-process regions)**
- on page out, find a free block to hold page
- fixed allocation: any free block will do
- must keep track of each page address on disk

Paging Daemons

On page fault, there may be no free pages
- must make room by paging something out
- in mean time, process must wait for pageout/pagein

Better strategy: always maintain a pool of free pages
- faulting process must only wait for page in

Paging daemon periodically replenishes free page pool
- when pool goes below threshold, replenish
- if freed page is accessed, remove from pool

Summary

VM is efficient because of principle of locality

\[ \text{OPT} \geq \text{LRU} \geq \text{Clock} \geq \text{FIFO} \]

**Goal:** keep working set in memory

If working set cannot be resident, swap process out

Other factors
- use large page size
- maintain free pool of page frames