Lecture - XV

Memory Management & Virtual Memory

Tevfik Koşar

University at Buffalo
October 25th, 2011

Roadmap

- Main Memory Management
  - Segmentation
- Virtual Memory
  - Demand Paging
  - Page Faults
  - Page Replacement
  - Page Replacement Algorithms
  - Performance of Demand Paging
User’s View of a Program

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables
  - global variables
  - common block
  - stack
  - symbol table
  - arrays
Logical View of Segmentation

Segmentation Architecture

• Logical address consists of a two tuple:
  \(<\text{segment-number, offset}>\),
• Segment table - maps two-dimensional physical addresses; each table entry has:
  - base - contains the starting physical address where the segments reside in memory
  - limit - specifies the length of the segment
• Segment-table base register (STBR) points to the segment table’s location in memory
• Segment-table length register (STLR) indicates the length (limit) of the segment
• segment addressing is \( d (\text{offset}) < \text{STLR} \)
**Segmentation Architecture (Cont.)**

- **Protection.** With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

---

**Address Translation Architecture**

[Diagram showing address translation process with CPU, segment table, and physical memory connections.]
Exercise

Consider the following segment table:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>219</td>
<td>600</td>
</tr>
<tr>
<td>1</td>
<td>2300</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1327</td>
<td>580</td>
</tr>
<tr>
<td>4</td>
<td>1952</td>
<td>96</td>
</tr>
</tbody>
</table>

What are the physical addresses for the following logical addresses?

a. 1, 100

b. 2, 0

c. 3, 580
Solution

Consider the following segment table:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>219</td>
<td>600</td>
</tr>
<tr>
<td>1</td>
<td>2300</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1327</td>
<td>580</td>
</tr>
<tr>
<td>4</td>
<td>1952</td>
<td>96</td>
</tr>
</tbody>
</table>

What are the physical addresses for the following logical addresses?

a. 1, 100
illegal reference (2300+100 is not within segment limits)

b. 2, 0
physical address = 90 + 0 = 90

c. 3, 580
illegal reference (1327 + 580 is not within segment limits)

Sharing of Segments

![Diagram of sharing of segments]
Virtual Memory

Background

- **Virtual memory** - separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated
  (1 ⇒ in-memory and legal, 0 ⇒ not-in-memory or invalid)
- Initially valid-invalid bit is set to 0 on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid-invalid bit in page table entry is 0 ⇒ page fault
Page Table When Some Pages Are Not in Main Memory

Transfer of a Paged Memory to Contiguous Disk Space
Page Fault

- If there is ever a reference to a page not in memory, first reference will trap to OS ⇒ page fault
- OS looks at another table (in PCB) to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory. ==> page-in
- Get an empty frame.
- Swap (read) page into the new frame.
- Set validation bit = 1.
- Restart instruction

Steps in Handling a Page Fault
What happens if there is no free frame?

- Page replacement - find some page in memory, but not really in use, swap it out
  - Algorithms (FIFO, LRU ..)
  - performance - want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers - only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process
Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it 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

Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

```
  1  4  5
  2  1  3  9 page faults
  3  2  4
```
First-In-First-Out (FIFO) Algorithm

• Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
• 3 frames (3 pages can be in memory at a time per process)

```
1 4 5
2 1 3 9 page faults
3 2 4
```

• 4 frames

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

FIFO Replacement - Belady’s Anomaly
- more frames ⇒ more page faults
FIFO Illustrating Belady’s Anomaly

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

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

- Memory access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out
- Swap Page Time = 10 msec = 10,000 microsec
- EAT = ?

\[
EAT = (1 - p) \times 1 + p \times (10,000 + \frac{1}{2} \times 10,000) \\
= 1 + 14,999 \times p \quad \text{(in microsec)}
\]

- What if 1 out of 1000 memory accesses cause a page fault?

- What if we only want 30% performance degradation?
Summary

• Main Memory Management
  - Segmentation
• Virtual Memory
  - Demand Paging
  - Page Faults
  - Page Replacement
  - Page Replacement Algorithms
  - Performance of Demand Paging

• Next Lecture: Virtual Memory - II
• Reading Assignment: Chapter 9 from Silberschatz.

Acknowledgements

• “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne

• “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings

• “Modern Operating Systems” book and supplementary material by A. Tanenbaum

• R. Doursat and M. Yuksel from UNR