Chapter 7
Memory Management
Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Basic blocks of memory management
  - Paging
  - Segmentation
The need for memory management

• Memory is cheap today, and getting cheaper
  – But applications are demanding more and more memory, there is never enough!

• Memory Management, involves swapping blocks of data from secondary storage.

• Memory I/O is slow compared to a CPU
  – The OS must cleverly time the swapping to maximise the CPU’s efficiency
Memory Management

Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.
Memory Management Requirements

- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation
Requirements: Relocation

• The programmer does not know where the program will be placed in memory when it is executed,
  – it may be swapped to disk and return to main memory at a different location (relocated)

• Memory references must be translated to the actual physical memory address
# Memory Management Terms

Table 7.1 Memory Management Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td><em>Fixed</em>-length block of main memory.</td>
</tr>
<tr>
<td>Page</td>
<td><em>Fixed</em>-length block of data in secondary memory (e.g. on disk).</td>
</tr>
<tr>
<td>Segment</td>
<td><em>Variable-length</em> block of data that resides in secondary memory.</td>
</tr>
</tbody>
</table>
Addressing

Figure 7.1 Addressing Requirements for a Process
Requirements: Protection

- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time
Requirements: Sharing

• Allow several processes to access the same portion of memory
• Better to allow each process access to the same copy of the program rather than have their own separate copy
Requirements: Logical Organization

- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes
- Segmentation helps here
Requirements: Physical Organization

• Cannot leave the programmer with the responsibility to manage memory

• Memory available for a program plus its data may be insufficient
  – Overlaying allows various modules to be assigned the same region of memory but is time consuming to program

• Programmer does not know how much space will be available
Partitioning

• An early method of managing memory
  – Pre-virtual memory
  – Not used much now
• But, it will clarify the later discussion of virtual memory if we look first at partitioning
  – Virtual Memory has evolved from the partitioning methods
Types of Partitioning

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation
Fixed Partitioning

• Equal-size partitions (see fig 7.3a)
  – Any process whose size is less than or equal to the partition size can be loaded into an available partition

• The operating system can swap a process out of a partition
  – If none are in a ready or running state
Fixed Partitioning Problems

• A program may not fit in a partition.
  – The programmer must design the program with overlays

• Main memory use is inefficient.
  – Any program, no matter how small, occupies an entire partition.
  – This is results in *internal fragmentation*. 
Solution – Unequal Size Partitions

• Lessens both problems
  – but doesn’t solve completely
• In Fig 7.3b,
  – Programs up to 16M can be accommodated without overlay
  – Smaller programs can be placed in smaller partitions, reducing internal fragmentation
Placement Algorithm

• Equal-size
  – Placement is trivial (no options)

• Unequal-size
  – Can assign each process to the smallest partition within which it will fit
  – Queue for each partition
  – Processes are assigned in such a way as to minimize wasted memory within a partition
Fixed Partitioning

(a) One process queue per partition
(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning
Remaining Problems with Fixed Partitions

• The number of active processes is limited by the system
  – I.E limited by the pre-determined number of partitions

• A large number of very small process will not use the space efficiently
  – In either fixed or variable length partition methods
Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
Dynamic Partitioning
Example

- *External Fragmentation*
- Memory external to all processes is fragmented
- Can resolve using *compaction*
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4
Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often
Dynamic Partitioning

• First-fit algorithm
  – Scans memory from the beginning and chooses the first available block that is large enough
  – Fastest
  – May have many processes loaded in the front end of memory that must be searched over when trying to find a free block
Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory
Figure 7.5  Example Memory Configuration before and after Allocation of 16-Mbyte Block
Buddy System

• Entire space available is treated as a single block of $2^U$

• If a request of size $s$ where $2^{U-1} < s \leq 2^U$
  – entire block is allocated

• Otherwise block is split into two equal buddies
  – Process continues until smallest block greater than or equal to $s$ is generated
Example of Buddy System

<table>
<thead>
<tr>
<th>Request 100 K</th>
<th>Request 240 K</th>
<th>Request 64 K</th>
<th>Request 256 K</th>
<th>Release B</th>
<th>Release A</th>
<th>Request 75 K</th>
<th>Release C</th>
<th>Release E</th>
<th>Release D</th>
</tr>
</thead>
<tbody>
<tr>
<td>128K</td>
<td>128K</td>
<td>C = 64K</td>
<td>C = 64K</td>
<td>C = 64K</td>
<td>64K</td>
<td>128K</td>
<td>128K</td>
<td></td>
<td>1M</td>
</tr>
<tr>
<td>256K</td>
<td>B = 256K</td>
<td>B = 256K</td>
<td>D = 256K</td>
<td>D = 256K</td>
<td>256K</td>
<td>D = 256K</td>
<td>D = 256K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.6  Example of Buddy System
Tree Representation of Buddy System

Figure 7.7  Tree Representation of Buddy System
Relocation

- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution
  - Swapping
  - Compaction
Addresses

• Logical
  – Reference to a memory location independent of the current assignment of data to memory.

• Relative
  – Address expressed as a location relative to some known point.

• Physical or Absolute
  – The absolute address or actual location in main memory.
Figure 7.8  Hardware Support for Relocation
Registers Used during Execution

- **Base register**
  - Starting address for the process

- **Bounds register**
  - Ending location of the process

- **These values are set when the process is loaded or when the process is swapped in**
Registers Used during Execution

• The value of the base register is added to a relative address to produce an absolute address

• The resulting address is compared with the value in the bounds register

• If the address is not within bounds, an interrupt is generated to the operating system
Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called *pages*
- The chunks of memory are called *frames*
Paging

- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page
### Processes and Frames

<table>
<thead>
<tr>
<th>Frame number</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A.0</td>
</tr>
<tr>
<td>1</td>
<td>A.1</td>
</tr>
<tr>
<td>2</td>
<td>A.2</td>
</tr>
<tr>
<td>3</td>
<td>A.3</td>
</tr>
<tr>
<td>4</td>
<td>D.0</td>
</tr>
<tr>
<td>5</td>
<td>D.1</td>
</tr>
<tr>
<td>6</td>
<td>D.2</td>
</tr>
<tr>
<td>7</td>
<td>C.0</td>
</tr>
<tr>
<td>8</td>
<td>C.1</td>
</tr>
<tr>
<td>9</td>
<td>C.2</td>
</tr>
<tr>
<td>10</td>
<td>C.3</td>
</tr>
<tr>
<td>11</td>
<td>D.3</td>
</tr>
<tr>
<td>12</td>
<td>D.4</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
### Page Table

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

**Process A** page table

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

**Process B** page table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**Process C** page table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

**Process D** page table

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Free frame list**

---

**Figure 7.10** Data Structures for the Example of Figure 7.9 at Time Epoch (f)
Segmentation

• A program can be subdivided into segments
  – Segments may vary in length
  – There is a maximum segment length

• Addressing consists of two parts
  – a segment number and
  – an offset

• Segmentation is similar to dynamic partitioning
Logical Addresses

Relative address = 1502

0000010111011110

User process
(270 bytes)

(a) Partitioning

Logical address =
Page# = 1, Offset = 478

0000010111011110

Page 0

Page 1

Page 2

(b) Paging
(page size = 1K)

(c) Segmentation

Logical address =
Segment# = 1, Offset = 752

0001001011110000

Segment 0
750 bytes

Segment 1
1950 bytes

Internal fragmentation

Figure 7.11 Logical Addresses
Paging

16-bit logical address

6-bit page #  10-bit offset

0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1 0

Process page table

0 0 0 1 0 1
0 0 0 1 1 0
0 1 1 0 0 1

16-bit physical address

0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 0
Figure 7.12  Examples of Logical-to-Physical Address Translation