Chapter 3
Memory Management
No Memory Abstraction

Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.
Multiple Programs Without Memory Abstraction

Figure 3-2. Illustration of the relocation problem.
Figure 3-3. Base and limit registers can be used to give each process a separate address space.
Swapping (1)

Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.
Swapping (2)

Figure 3-5. (a) Allocating space for growing data segment. (b) Allocating space for growing stack, growing data segment.
Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.
Memory Management with Linked Lists

Figure 3-7. Four neighbor combinations for the terminating process, X.
Virtual Memory – Paging (1)

Figure 3-8. The position and function of the MMU – shown as being a part of the CPU chip (it commonly is nowadays). Logically it could be a separate chip, was in years gone by.
Figure 3-9. Relation between virtual addresses and physical memory addresses given by page table.
Figure 3-10. The internal operation of the MMU with 16 4-KB pages.
Figure 3-11. A typical page table entry.
Paging implementation issues:

- The mapping from virtual address to physical address must be fast.
- If the virtual address space is large, the page table will be large.
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>

Figure 3-12. A TLB to speed up paging.
Multilevel Page Tables

Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.
Inverted Page Tables

Figure 3-14. Comparison of a traditional page table with an inverted page table.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Page Replacement Algorithms

- Optimal page replacement algorithm
- Not recently used page replacement
- First-In, First-Out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement
- Working set page replacement
- WSClock page replacement
Second Chance Algorithm

Figure 3-15. Operation of second chance.
(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.
The Clock Page Replacement Algorithm

Figure 3-16. The clock page replacement algorithm.
**LRU Page Replacement Algorithm**

![Figures 3-17](image)

**Figure 3-17.** LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.
Simulating LRU in Software

Figure 3-18. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Figure 3-19. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time $t$. 
Figure 3-20. The working set algorithm.
When the hand comes all the way around to its starting point there are two cases to consider:

- At least one write has been scheduled.
- No writes have been scheduled.
The WSClock Page Replacement Algorithm (2)

Figure 3-21. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.
The WSClock Page Replacement Algorithm (3)

Figure 3-21. Operation of the WSClock algorithm. (c) and (d) give an example of $R = 0$. 

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
## Summary 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 approximation of LRU</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>

Figure 3-22. Page replacement algorithms discussed in the text.
Figure 3-23. Local versus global page replacement.

(a) Original configuration. (b) Local page replacement. (c) Global page replacement.
Local versus Global Allocation Policies (2)

Figure 3-24. Page fault rate as a function of the number of page frames assigned.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Separate Instruction and Data Spaces

Figure 3-25. (a) One address space. (b) Separate I and D spaces.
Shared Pages

Figure 3-26. Two processes sharing the same program sharing its page table.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Figure 3-27. A shared library being used by two processes.
Page Fault Handling (1)

- The hardware traps to the kernel, saving the program counter on the stack.
- An assembly code routine is started to save the general registers and other volatile information.
- The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed.
- Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access.
Page Fault Handling (2)

- If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
- When page frame is clean, operating system looks up the disk address where the needed page is, schedules a disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables updated to reflect position, frame marked as being in normal state.
Page Fault Handling (3)

- Faulting instruction backed up to state it had when it began and program counter reset to point to that instruction.
- Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
- This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.
Instruction Backup

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

16 Bits

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Opcode
First operand
Second operand

Figure 3-28. An instruction causing a page fault.
Figure 3-29. (a) Paging to a static swap area.
Figure 3-29. (b) Backing up pages dynamically.
Separation of Policy and Mechanism (1)

Memory management system is divided into three parts:

- A low-level MMU handler.
- A page fault handler that is part of the kernel.
- An external pager running in user space.
Separation of Policy and Mechanism (2)

Figure 3-30. Page fault handling with an external pager.
A compiler has many tables that are built up as compilation proceeds, possibly including:

- The source text being saved for the printed listing (on batch systems).
- The symbol table – the names and attributes of variables.
- The table containing integer, floating-point constants used.
- The parse tree, the syntactic analysis of the program.
- The stack used for procedure calls within the compiler.
Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.
Figure 3-32. A segmented memory allows each table to grow or shrink independently of the other tables.
### Implementation of Pure Segmentation

<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>

Figure 3-33. Comparison of paging and segmentation.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Figure 3-34. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.
Figure 3-35. The MULTICS virtual memory. (a) The descriptor segment points to the page tables.
Figure 3-35. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
When a memory reference occurs, the following algorithm is carried out:

- The segment number used to find segment descriptor.
- Check is made to see if the segment’s page table is in memory.
  - If not, segment fault occurs.
  - If there is a protection violation, a fault (trap) occurs.
Segmentation with Paging: MULTICS (7)

- Page table entry for the requested virtual page examined.
  - If the page itself is not in memory, a page fault is triggered.
  - If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
- The offset is added to the page origin to give the main memory address where the word is located.
- The read or store finally takes place.
Figure 3-36. A 34-bit MULTICS virtual address.
Segmentation with Paging: MULTICS (9)

Figure 3-37. Conversion of a two-part MULTICS address into a main memory address.
Figure 3-38. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Figure 3-39. A Pentium selector.

Segmentation with Paging: The Pentium (1)
Figure 3-40. Pentium code segment descriptor. Data segments differ slightly.
Figure 3-41. Conversion of a (selector, offset) pair to a linear address.

Segmentation with Paging: The Pentium (3)

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
Figure 3-42. Mapping of a linear address onto a physical address.
Segmentation with Paging: The Pentium (5)

Figure 3-43. Protection on the Pentium.