No Memory Abstraction

Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.
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 3-8. 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.
Structure of Page Table Entry

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.
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
The Clock Page Replacement Algorithm

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
- \( R = 0 \): Evict the page
- \( R = 1 \): Clear R and advance hand

Figure 3-16. The clock page replacement algorithm.
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 $t$. 
### 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.
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.
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.
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.
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.
Segmentation (2)

Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.

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