Chapter 3
Memory Management

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.

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

Figure 3-5. (a) Allocating space for growing data segment. (b) Allocating space for growing stack, growing data segment.
Memory Management with Bit Maps

- (a) Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- (b) Corresponding bit map (shaded region – 0 in the bitmap)
- (c) Same information as a list

Memory Management with Linked Lists

Before X terminates
(a) A X B becomes A B
(b) A X becomes A
(c) X B becomes A B
(d) X becomes

After X terminates

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.

Paging (2)

Figure 3-9. Relation between virtual addresses and physical memory addresses given by page table.

Paging (3)

Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Structure of Page Table Entry

- Caching disabled
- Modified
- Present/absent
- Page frame number
- Referenced Protection

Typical page table entry
Speeding Up Paging

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

- Multilevel Page Tables work (save space) only because we do not need to keep all pages in the memory:
  - if we used 1 level for page table and x+y bits, then the page table would be $2^{x+y}$ entries. If to use 2-level paging with x bits for virtual page number, y bits for first level, then we would need $2^x * 2^{2y}$ length for all page tables (no savings). However, if from the 1st level only $z << 2^x$ entries were used on average at the given moment of time, then we would need $2^x + z*2^y$ space for 2-level page table.

- Examples of paging:
  - 0-level paging: MIPS R3000 (page table not kept in memory but in TLB cache in CPU)
  - 1-level paging: DEC PDP-11
  - 2-level paging: DEC VAX
  - 3-level paging: SUN SPARC
  - 4-level paging: Motorola 68

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
Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical

Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random
  - from lowest numbered non-empty class

FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory the longest may be often used

Second Chance Algorithm

- Pages sorted in FIFO order.
- 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

- 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
Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Simulating LRU in Software

<table>
<thead>
<tr>
<th>Page</th>
<th>R bits for pages 0-9 (MSB)</th>
<th>R bits for pages 10-19 (MSB)</th>
<th>R bits for pages 20-29 (MSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>2</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>3</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>5</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
</tbody>
</table>

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).

Working Set Page Replacement (1)

\[ w(k, t) \]

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 \( t \).

Working Set Page Replacement (2)

- At least one write has been scheduled.
- No writes have been scheduled.

The WSClock Page Replacement Algorithm (1)

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.
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>

Local versus Global Allocation Policies (1)

<table>
<thead>
<tr>
<th>Age</th>
<th>(a) Original configuration. (b) Local page replacement. (c) Global page replacement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0</td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
</tr>
<tr>
<td>A3</td>
<td>A3</td>
</tr>
<tr>
<td>A4</td>
<td>A4</td>
</tr>
<tr>
<td>A5</td>
<td>A5</td>
</tr>
<tr>
<td>B0</td>
<td>B0</td>
</tr>
<tr>
<td>B1</td>
<td>B1</td>
</tr>
<tr>
<td>B2</td>
<td>B2</td>
</tr>
<tr>
<td>B3</td>
<td>B3</td>
</tr>
<tr>
<td>B4</td>
<td>B4</td>
</tr>
<tr>
<td>B5</td>
<td>B5</td>
</tr>
<tr>
<td>B6</td>
<td>B6</td>
</tr>
<tr>
<td>C1</td>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
<td>C3</td>
</tr>
</tbody>
</table>

Local versus Global Allocation Policies (2)

![Figure 3-24. Page fault rate as a function of the number of page frames assigned.](image)

Separate Instruction and Data Spaces

![Figure 3-25. (a) One address space. (b) Separate I and D spaces.](image)
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, and 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)
```

1000         MOVE         Opcode
1002         6            First operand
1004         2            Second operand

Figure 3-28. An instruction causing a page fault.
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.
A segmented memory allows each table to grow or shrink independently of the other tables.

Comparison of paging and segmentation.

Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

The MULTICS virtual memory. (a) The descriptor segment points to the page tables.

The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

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.

Segmentation with Paging: MULTICS (8)

- Address within the segment
  - Segment number: 18
  - Page number: 6
  - Offset within the page: 10

Figure 3-36. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (9)

- Conversion of a two-part MULTICS address into a main memory address.

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

Segmentation with Paging: The Pentium (1)

- Pentium selector.

Figure 3-39. A Pentium selector.

Segmentation with Paging: The Pentium (2)

- Pentium code segment descriptor.
  - Data segments differ slightly.
**Segmentation with Paging: The Pentium (3)**

- Selector
- Offset
- Descriptor
- Base address
- Limit
- Other fields

32-bit linear address

Figure 3-41. Conversion of a (selector, offset) pair to a linear address.

**Segmentation with Paging: The Pentium (4)**

- Linear address
- Page
- Offset

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.