Memory Management

- Real memory
- Logical memory
- Address spaces
- Virtual memory

Chapter 7 - Physical Memory

7.1 Preparing a Program for Execution
   - Program Transformations
   - Logical-to-Physical Address Binding

7.2 Memory Partitioning Schemes
   - Fixed Partitions
   - Variable Partitions

7.3 Allocation Strategies for Variable Partitions

7.4 Dealing with Insufficient Memory
**Policy versus Mechanism**

- Software systems are seldom "monolithic" entities but are composed of separate modules
  - Modules reflect different functionalities
  - Code and data are separate entities
  - Each module has its own local address space

---

**Policy versus Mechanism**

- Modules are then linked into one combined object
  - This load module has again a private address space
  - Module and library references (external) are resolved

- The load module is finally loaded into the real (physical) memory, either completely or partially (virtual)
  - Mapping of logical and physical addresses and relocation should be supported by hardware features
  - Virtual memory needs hardware support, efficiency depends on exploitation of statistical characteristics and caching
Transformation of Programs

\[ S_1 \rightarrow O_1 \]
\[ S_2 \rightarrow O_2 \]
\[ \vdots \]
\[ S_n \rightarrow O_n \]

\[ L_{\text{temp}} \rightarrow L_{\text{phys}} \]

\( S_1 \) - Source Program
\( O_1 \) - Object Module
\( L_{\text{temp}} \) - relocatable Load Module
\( L_{\text{phys}} \) - Load Module

Program Execution

- **program transformations**
  - Step 1: Translation (compilation, assembly)
    * asm, c, gcc, for, ada, mod, java, ...
  - Step 2: Linking
    * link, bind,
  - Step 3: Loading
    * load
  - Step 4: Execution
    * run, exec, …
Transformation of Programs

**Source Program**

- source module 1
- source module 2
- ... 
- source module n

**Object Module**

- object module 1
- object module 2
- ... 
- object module n

**Load Module**

- load module (in secondary memory)

**Library Module**

- library module

**Linking**

- Modules are developed and translated independently
- External references are left open at translation time and are entered into an “external reference table”
- When the modules are linked into one loadable module the external references need to be resolved
- Two methods are common:
  - Dynamic chaining
  - Transfer Vector
**Address binding**

- assign physical addresses = relocation
- static binding
  - programming time
  - compilation time
  - linking time
  - loading time
- dynamic binding
  - execution time

Compare Figures 7-2 and 7-4

**Relocation**

- Assignment of real memory
- Programs make use of logical address space
  - Code/Data spaces
  - Segments
- Real memory is linear
  - divided into segments
  - subdivided into page frames
- Relocation – Load
  - Static (link – bind early)
  - Dynamic (link – bind late)
  - Hardware supported – MMU and HW caches
**Binding Time**

- **Programming-time**
  - programmer's responsibility
- **Translation-time**
  - Compiler/Assembler
- **Link-time**
  - Linker/Binder
- **Run-time**
  - Hardware support
  - Operating system – VM

- (Interpreter + Run-time system $\rightarrow$ “JIT”)
Static Relocation

\[
\text{loop} \\
w := M[ic]; \quad \{\text{fetch instruction}\} \\
oc := \text{Opcode}(w); \\
adr := \text{Address}(w); \\
ic := ic + 1; \\
\text{case } oc \text{ of} \\
\quad 1: \text{reg} := \text{reg} + M[adr]; \quad \{\text{plus}\} \\
\quad 2: M[adr] := \text{reg}; \quad \{\text{store}\} \\
\quad 3: ic := adr; \quad \{\text{branch}\} \\
\quad : \\
\quad : \\
\text{end} \\
\text{end } \{\text{loop}\}
\]
**Dynamic Relocation**

```plaintext
loop
  w := M[NL_map(ic)];  \{ fetch instruction\}
  oc := Opcode(w);
  adr := Address(w);
  ic := ic + 1;
  case oc of
    1 : reg := reg + M[NL_map(adr)];  \{plus\}
    2 : M[NL_map(adr)] := reg;    \{store\}
    3 : ic := adr;               \{branch\}
  end
end \{loop\}
```

**Address binding**

- how to implement dynamic binding
  - perform for each address at run time:
    \[ pa = \text{address\_map}(la) \]
  - simplest form of address\_map:
    relocation register
    \[ pa = la + RR \]
  - more general form: page/segment table (Chapter 8)
Fig 7-3b: storage reference with dynamic relocation (using relocation register)

NL_map: {relocatable (virtual) addresses} ⇒ {real (physical) storage addresses}

Address Relocation

relocation register

CPU

Main storage

Data transfer
(read/write)

logical address

physical address

storage reference with dynamic relocation (using relocation register)

NL_map: {relocatable (virtual) addresses} ⇒ {real (physical) storage addresses}

Address Relocation

limit register

base register

segment

Main storage

logical address

physical address
Fig 7-4: dynamic relocation

Assembler/Compiler

P

goto A;
A: . . .

50

JMP 50

100

JMP 150

1100

other program modules

Linker/Binder

Loader

1000

other program modules

NL_map

1150

other program modules

binding of external references

Program source

call SR_1
call SR_1
call SR_2
call SR_1
call SR_1

call NIL
call NIL
call NIL
call NIL
call NIL

call SR_2
call SR_2
call SR_2
call SR_2
call SR_2

SR_1:
SR_2:

external symbol table

SR_1:
SR_2:

chaining method

indirect addressing
Partitions

• Assignment of memory in “partitions”
• Storage space is divided into a number of separate spaces
  – Code/Data spaces
  – Segments
• Determination of partition space is difficult but crucial for performance
  – divided into fixed size or variable size segments
  – Can lead to memory fragmentation
• Partition strategies
  – Fixed size – simple but can be wasteful
  – Variable size – more economical, might need compaction

Memory partitioning schemes

• fixed partitions
  – single-program systems: 2 partitions (OS/user)
  – multi-programmed: partitions of different sizes
• how to assign processes to partitions Fig 7-5
  – FIFO for each partition
    * some partitions may remain unused
  – single FIFO
    * more complex but more flexible
• limitations of fixed partitions
  – program size limited to largest partition
  – internal fragmentation
Variable partitions

- memory not partitioned a priori
- each request is allocated portion of free space
- memory: sequence of variable-size blocks
  - some are occupied, some are free (holes)
  - external fragmentation occurs
- adjacent holes must be coalesced to prevent increasing fragmentation

Figure 7-6
Fig 7-5b: scheduling of processes into partitions

memory partitions

queue

processes

One common queue for all partitions

Fig 7-6a: hole coalescing

queue

Release( A )

B A' C
Fig 7-6d: hole coalescing

queue

Release( A )

B
A
C

B'

Fig 5-20: hole recombination on release

Release(A)

(a)

B
A'
C

B
C'

B
A
C

B
A
C

B
A
C

B'

B'
Linked list implementation

- type/size tags at the beginning of each block
- links (pointers) kept inside holes
- how to check neighbors of released block $b$:
  - right neighbor: use size of $b$
  - left neighbor:
    - uses sizes to find first hole to the right
    - follow predecessor ptr to first hole on left
    - check if neighbor

Figure 7-7(a)

- holes must be sorted by physical address
Linked list implementation

- better solution: replicate tags at end of block
  - checking neighbors of released block b:
    * right neighbor: use size of b as before
    * left neighbor: check adjacent tags
  - blocks need not be sorted

Figure 7.7(b)
Bit map implementation

- memory divided into fix-size blocks
- each block represented by a 0/1 bit in a binary string:
  the bit map
- can be implemented as char or int array
- operations use bit masks
  - release: \( B[i] = B[i] \& \text{'}11011111' \)
  - allocation: \( B[i] = B[i] \mid \text{'}11000000' \)
  - search: repeatedly check left-most bit and shift mask right:
    \( \text{TEST} = B[i] \& \text{'}10000000' \)

Dealing with insufficient memory

- memory compaction
  - how much and what to move?
  - Figure 7-9
- swapping
  - temporarily move process to disk
  - requires dynamic relocation
- overlays
  - allows programs larger than physical memory
  - programs loaded as needed according to calling structure
    (Figure 7-10)
The Buddy system

- compromise between fixed and variable
- fixed number of possible hole sizes, e.g., $2^i$
- request $n$ bytes:
  - find smallest hole size such that $n \leq 2^i$
  - if not empty, allocate hole, else consider larger holes:
    * recursively split each hole into two buddies until smallest possible hole is created; allocate it and place other holes on appropriate lists
- release block: recursively coalesce buddies

Example: Figure 7-9

Fig 7-9: the buddy system

H

(a)

(b)
Fig 7-9: the buddy system

memory fragmentation

|D| > |H_1|, \quad |D| > |H_2|
|D| < |H_1| + |H_2|
Allocation strategies

- given a request for n bytes, find hole
- constraints:
  - maximize memory utilization (minimize external fragmentation)
  - minimize search time
- first fit: simplest
- next fit: improve distribution of holes
- best fit: avoid breaking up large holes
- worst fit: avoid leaving tiny hole fragments
- first fit is generally the best choice

Fig 7-10: memory compaction

Initial state

<table>
<thead>
<tr>
<th>P1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>2</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total compaction

<table>
<thead>
<tr>
<th>P1</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>20</th>
<th>9</th>
</tr>
</thead>
</table>

Partial compaction

<table>
<thead>
<tr>
<th>P1</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>11</th>
<th>9</th>
</tr>
</thead>
</table>

Minimal move

<table>
<thead>
<tr>
<th>P1</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>11</th>
</tr>
</thead>
</table>
Overlays

- Observation: memory requirements are not static but vary over time
  - Due to use of data structures
  - Due to execution profile (proc calls, loops, etc)

- Provide the programmer an instrument to control the use of storage space
  - re-allocate memory
  - dynamically reassign memory space

- Question: how good a job can an (average) programmer do, or is it better to do it form him/her/it?

Fig 7-11: program overlays
Measures of memory utilization

• 50% rule:
  \[ \#\text{holes} = 0.5 \times p \times \#\text{full_blocks} \]
  – \( p \): probability of inexact match (remaining hole)
  – in practice: \( p = 1 \), i.e., 1/3 of memory are holes

• but how much memory is wasted
  (hole size ≠ block size)

Measures of memory utilization

• what is the unused fraction of space?
  – utilization depends on \( k = \text{hole_size} / \text{block_size} \):
    \[ \text{unused_memory} = k / (k + 2) \]
  – intuition:
    * \( k \to \infty \): unused memory \( \to 1 \) (100% empty)
    * \( k = 1 \): unused memory \( \to 1 / 3 \) (50% rule)
    * \( k \to \infty \): unused memory \( \to 0 \) (100% full)
  – what determines \( k \)?
    * block size \( b \) relative to total memory size \( M \)
    * determined experimentally
      \[ b \leq M / 10: \quad k = 0.22: \quad f = 0.1 \]
      \[ b = M / 3: \quad k = 2: \quad f = 0.5 \]

• conclusion: \( M \) must be large relative to \( b \)