Memory Management

- Address binding
  - Linking, loading
  - Logical vs. physical address space
- Memory partitioning
- Paging
- Segmentation

- Reading: Silberschatz, Chapter 8

Memory Management

- Observations:
  - Process needs at least CPU and memory to run.
  - CPU context switching is relatively cheap.
  - Swapping memory in/out from/to disk is expensive.

- Need to subdivide memory to accommodate multiple processes!

- How do we manage this memory?
Requirements for Memory Management

- **Relocation**
  - We do not know a priori where memory of process will reside.

- **Protection**
  - No uncontrolled references to memory locations of other processes.
  - Memory references must be checked at run-time.

- **Sharing**
  - Data portions and program text portions.

- **Logical organization**
  - Take advantage of semantics of use.
  - Data portions (read/write) vs. program text portions (read only).

- **Memory hierarchy**
  - RAM vs. secondary storage
  - Swapping

Preparing a Program for Execution

- **compiler**: translates symbolic instructions, operands, and addresses into numerical values.
- **linker**: resolves external references; i.e. operands or branch addresses referring to data or instructions within some other module
- **loader**: brings program into main memory.
Address Binding

- Compile-time binding:
  - branch A
  - assembler
  - branch 150

- Load-time binding (static relocation):
  - branch A
  - assembler
  - branch 50
  - linker
  - branch 150
  - loader
  - branch 1150

- Execution-time binding (dynamic relocation):
  - branch A
  - assembler
  - branch 50
  - linker
  - branch 150
  - loader
  - branch 150
  - MMU

Dynamic Loading, Dynamic Linking

- Dynamic Loading:
  - load routine into memory only when it is needed
  - routines kept on disk in relocatable format

- Load-time Linking:
  - postpone linking until load time.

- Dynamic Linking:
  - postpone linking until execution time.
    - call x
    - x: stub
    - load routine
    - link x to location of routine
    - Problem: Need help from OS!
Logical vs. Physical Address Space

- **Logical address**: address as seen by the process (i.e. as seen by the CPU).
- **Physical address**: address as seen by the memory.

**Example:**

**load time:**

- Branch A
- Assembler: 50
- Branch 50: 100
- Branch 150: 1100
- Loader: 1150

**execution time:**

- Branch A
- Assembler: 50
- Branch 50: 100
- Branch 150: 1100
- MMU: 1150

logical = physical

logical = physical

Logical vs. Physical Memory Space

- **Logical address**: address as seen by the process (i.e. as seen by the CPU).
- **Physical address**: address as seen by the memory.

```
<table>
<thead>
<tr>
<th>process</th>
<th>base</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>28</td>
<td>1000</td>
</tr>
<tr>
<td>P_2</td>
<td>1028</td>
<td>3000</td>
</tr>
<tr>
<td>P_3</td>
<td>5034</td>
<td>250</td>
</tr>
</tbody>
</table>
```
Swapping

Simple Method: Fixed Partitioning

- Partition available memory into regions with fixed boundaries.

Equal-size Partitions

<table>
<thead>
<tr>
<th>OS</th>
<th>8MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
</tbody>
</table>

Unequal-size Partitions

<table>
<thead>
<tr>
<th>OS</th>
<th>8MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>2MB</td>
<td></td>
</tr>
<tr>
<td>4MB</td>
<td></td>
</tr>
<tr>
<td>8MB</td>
<td></td>
</tr>
<tr>
<td>12MB</td>
<td></td>
</tr>
<tr>
<td>16MB</td>
<td></td>
</tr>
</tbody>
</table>

- Problem: Internal Fragmentation.
Simple Method: Dynamic Partitions

- Partitions can be of variable length and number.
- Process is allocated exactly as much memory as requested.

External Fragmentation

Job queue:  
- P_1: 100kB  
- P_2: 256kB  
- P_3: 256kB  
- P_4: 512kB

Available memory: 1024kB

- Solution?  
  - Compaction  
  - Paging
Allocation Strategies

- General schemes for allocating variable-sized blocks of main storage in systems without paging hardware.
- Two commands:

  request_mainstore(int size, char ** base_addr)
  - If there is a hole large enough, allocate size units of that hole (if there are several holes, choice which one to pick defined by placement policy)
  - If no sufficiently big hole available:
    - temporarily block request
    - deallocate one or more used blocks (swapping, choice defined by replacement policy)
    - Compaction

  release_mainstore(int size, char * base_addr)

Placement Policies

- **first-fit**: search for first hole that is big enough
- **best-fit**: search for smallest hole that is big enough
- **worst-fit**: search for largest hole and see if it fits.

Which policy performs best?
Administration of Available Space

- List of available holes: Instead of using separate storage area for data structure, use hole space itself.
- Alternatives: Buddy scheme, others.

![Diagram of available space management](image)

---

Paging

- Contiguous allocation causes (external) fragmentation.
- Solution: Partition memory blocks into smaller subblocks (pages) and allow them to be allocated non-contiguously.

- Simple partitioning:
  - Logical memory
  - Physical memory

- Paging:
  - Logical memory
  - Physical memory
Basic Operations in Paging Hardware

- CPU
- p, d, f
- Page table
- Memory Management Unit
- Physical memory

Internal Fragmentation in Paging

- Example:
  - Logical memory: 13300B
  - Page size: 4kB
  - Physical memory: 4084 bytes wasted!

- Last frame allocated may not be completely full.
- **Average internal fragmentation** per block is typically **half frame size**.
- Large frames vs. small frames:
  - Large frames cause more fragmentation.
  - Small frames cause more overhead (page table size, disk I/O)
Implementation of Page Table

- Page table involved in every access to memory. Speed very important.
- Page table in registers?
  - Example: 1MB logical address space, 2kB page size; needs a page table with 512 entries!
- Page table in memory?
  - Only keep a page table base register that points to location of page table.
  - Each access to memory would require two accesses to memory!
- Cache portions of page table in registers?
  - Use translation lookaside buffers (TLBs): typically a few dozens entries.
  - Hit ratio: Percentage of time an entry is found. Hit ratio must be high in order to minimize overhead.

Multilevel Paging

- Problem: Page tables can become very large!
  - Example: 32-bit address space (4GB) and 4kB page size needs page table with $2^{30}$ entries!
- Solution: Page the page table itself!
- Two-level paging:
  - logical address: 32 bit page size 4kB
- Operation:

  ![Diagram of page table access]

- Three-level paging (SPARC), four-level paging (68030), ...
### Segmentation

- Users think of memory in terms of segments (data, code, stack, objects, ...).
- Data within a segment typically has uniform access restrictions.

#### Paging:

![Diagram of paging](image)

#### Segmentation:

![Diagram of segmentation](image)

### Segmentation Hardware

![Diagram of segmentation hardware](image)
Advantages of Segmentation

- Data in a segment typically semantically related
- Protection can be associated with segments
  - read/write protection
  - range checks for arrays
- Data/code sharing

Solution: Paged Segmentation

- Example: MULTICS

\[
\begin{array}{c|c}
\text{segment number} & \text{offset} \\
\hline
18\text{bit} & 16\text{bit} \\
\end{array}
\]

Problem: 64kW segments -> external fragmentation!
Solution: **Page the segments.**

\[
\begin{array}{c|c|c}
\text{segment number} & \text{page} & \text{page offset} \\
\hline
18\text{bit} & 6\text{bit} & 10\text{bit} \\
\end{array}
\]

Problem: need 2^18 segment entries in segment table
Solution: **Page the segment table.**

\[
\begin{array}{c|c|c|c|c}
\text{page} & \text{page} & \text{page} & \text{page offset} \\
\hline
8\text{bit} & 10\text{bit} & 6\text{bit} & 10\text{bit} \\
\end{array}
\]