Chapter 11

Memory Management

Main memory is a resource that must be allocated and deallocated

Memory Management Techniques determine:

- Where and how a process resides in memory
- How addressing is performed

Binding:

identifiers $\rightarrow$ (virtual) relative addresses $\rightarrow$ physical addresses
Memory Management Techniques

1) Single Contiguous               5) Paging
2) Overlays                      6) Demand Paging
3) Fixed (Static) Partitions      7) Segmented
4) Relocation (Dynamic) Partitions 8) Segmented / Demand Paging

For each technique, observe:
- Algorithms
- Advantages / Disadvantages
- Special Requirements

I. Single Contiguous

While (job is ready) Do
  If (JobSize <= MemorySize)
    Then Begin
      Allocate Memory
      Load and Execute Job
    End
  Else Error
I. Single Contiguous...

😊 **Advantages:**
- Simplicity
- No special hardware

😊 **Disadvantages:**
- CPU wasted
- Main memory not fully used
- Limited job size

II. Overlays

- Programs can be sectioned into modules
- Not all modules need to be in main memory at the same time

```
   A
  / \
 /   \
B     E
/     /
C     D
```

- Programmer specifies which modules can overlay each other
- Linker inserts commands to invoke the loader when the modules are referenced
- The "parent" must stay in memory
- Used in DOS as an alternative to Expanded Memory.
Illustration of Overlays

Process: A B C D E
Memory: 40K 30K 10K 10K 40K

Without Overlays

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>70</td>
<td>130</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Overlays

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advantages:
- Reduced memory requirements

Disadvantages:
- Overlap map must be specified by programmer
- Programmer must know memory requirements
- Overlapped modules must be completely disjoint
Fixed (Static) Partitioning with Absolute Translation

- Earliest attempt at multiprogramming
- Partition memory into fixed sized areas:

```
<table>
<thead>
<tr>
<th>0M</th>
<th>16M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition #1</td>
<td>6M</td>
</tr>
<tr>
<td>Partition #2</td>
<td>2M</td>
</tr>
<tr>
<td>Partition #3</td>
<td>8M</td>
</tr>
</tbody>
</table>
```

- Each partition can hold ONE process
- Code generated using an ABSOLUTE address (relative to 0, 6M, or 8M in picture)
- Queue of processes waiting for each partition
Fixed (Static) Partitioning with Absolute Translation

Fig. 7.5 Fixed partition multiprogramming with absolute translation and loading.

Fixed (Static) Partitioning with Absolute Translation...

Fig. 7.7 An extreme example of poor storage utilization in fixed partition multiprogramming with absolute translation and loading. Jobs waiting for partition 3 are small and could "fit" in the other partitions. But with absolute translation and loading, these jobs may run only in partition 3. The other two partitions remain empty.
Fragmentation - Definitions

*Fragmentation* is a situation in which the free cells in main memory are not contiguous.

**Internal fragmentation:**
A situation in which free memory cells are within the area allocated to a process

**External fragmentation:**
A situation in which free memory cells are not in the area allocated to any process

Fixed Partition Fragmentation

<table>
<thead>
<tr>
<th>Partition</th>
<th>Job Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>20K</td>
<td>20K</td>
</tr>
<tr>
<td>35K</td>
<td>30K</td>
</tr>
<tr>
<td>25K</td>
<td>10K</td>
</tr>
<tr>
<td>30K</td>
<td>30K</td>
</tr>
</tbody>
</table>

External fragmentation: 35K partition
Internal fragmentation: 25-10 => 15K wasted inside 25K partition
Fixed Partitioning with Absolute Translation: Pros/Cons

😊 **Advantages:**
- Simplicity
- Multiprogramming now possible
- Works with *any* hardware (8088, 68000, etc)

😊 **Disadvantages:**
- Job Size $\leq$ Max Partition Size $\leq$ MM Size
- Storage wasted due to *internal fragmentation*:
  - process size $<$ partition size
- Storage wasted due to *external fragmentation*:
  - A partition may be idle because none of the jobs assigned to it are being run
- Once compiled a job can *only* be executed in designated partition
Fixed (Static) Partitions with Relative Address Translation

- Allows process to run in any free partition
- ALL Code generated using addresses relative to zero

Illustration:
Let:
B denote base (absolute) address of a partition
L denote partition length

QTP: Would Pointers work?
Multiprogramming Protection

Fixed partitions with relative addressing supports multiprogramming protection

=> Ensure that one process does not access memory space dedicated to another process

Method:
Each relative address is compared to the bounds register

```
B
B + L
```

```
Bounds Reg
B + L
```

```
“Virtual” Address
```

```
Base Reg
B
```

```
OK
```

```
Error: Illegal Address
```

CS3204 - Arthur
Fixed Partitioning with Relative Addressing: Pros/Cons

Advantage compared to absolute addressing:
- Dynamic allocation of programs to partitions improves system performance

Still some disadvantages:
- Partition sizes are fixed at boot time
- Can't run process larger than largest partition
- Partition selection algorithm affects system performance
- Still has internal and external fragmentation

IV. Dynamic Partitions

Consider following scenario (100K memory):

1. Job 1 arrives; size= 22 K
2. Job 2 arrives; size= 24 K
3. Job 3 arrives; size= 30 K
4. Job 4 arrives; size= 10 K
5. Job 1 terminates
6. Job 3 terminates
7. Job 5 arrives; size= 12 K

Where should job 5 be put?
Partition Selection Algorithms

- Implementation requires a free block table
- Sorting table in a particular manner results in a specific selection algorithm:

1) First Fit -- Table sorted by location, searched top to bottom
2) Best Fit -- Table sorted by size (ascending) [don't break up big blocks]
3) Worst Fit -- Table sorted by size (descending) [break up big blocks]
4) Next Fit

Where does Job 5 Go?
First Fit

<table>
<thead>
<tr>
<th>Free List Table - First Fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start addr</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>e</td>
</tr>
</tbody>
</table>

7. Job 5 arrives; size=12K
Where does Job 5 Go?

Best Fit

Free List Table - Best Fit

<table>
<thead>
<tr>
<th>Start addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>14</td>
</tr>
<tr>
<td>a</td>
<td>22</td>
</tr>
<tr>
<td>c</td>
<td>30</td>
</tr>
</tbody>
</table>

7. Job 5 arrives; size=12K

Worst Fit

Free List Table - Worst Fit

<table>
<thead>
<tr>
<th>Start addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>30</td>
</tr>
<tr>
<td>a</td>
<td>22</td>
</tr>
<tr>
<td>e</td>
<td>14</td>
</tr>
</tbody>
</table>

7. Job 5 arrives; size=12K
Where does Job 5 Go?
Next Fit

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>FREE - 22 K</td>
</tr>
<tr>
<td>b</td>
<td>IN USE (J2) - 24 K</td>
</tr>
<tr>
<td>c</td>
<td>FREE - 30 K</td>
</tr>
<tr>
<td>d</td>
<td>IN USE (J4) - 10 K</td>
</tr>
<tr>
<td>e</td>
<td>FREE - 14 K</td>
</tr>
</tbody>
</table>

Free List Table - Next fit

<table>
<thead>
<tr>
<th>Start addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>22</td>
</tr>
<tr>
<td>c</td>
<td>30</td>
</tr>
<tr>
<td>e</td>
<td>14</td>
</tr>
</tbody>
</table>

7. Job 5 arrives; size=12K

Dynamic Partitions

Requires two OS operations:

- **Allocation:**
  
  Form a partition from a free partition of ample size

- **Deallocation:**
  
  Return partition to free table and *merge* where possible

CS3204 - Arthur
**Merge Example**

Suppose b becomes free

<table>
<thead>
<tr>
<th>Free List Table - First Fit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start addr</strong></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>e</td>
</tr>
</tbody>
</table>

What does Free List Table look like?

<table>
<thead>
<tr>
<th>Free List Table - Best Fit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start addr</strong></td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

What does Free List Table look like?
### Merge Example

Suppose b becomes free

#### Free List Table - *Worst Fit*

<table>
<thead>
<tr>
<th>Start addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>30</td>
</tr>
<tr>
<td>a</td>
<td>22</td>
</tr>
<tr>
<td>e</td>
<td>14</td>
</tr>
</tbody>
</table>

What does Free List Table look like?

---

### Merge Example

Suppose b becomes free

#### Free List Table - *Next fit*

<table>
<thead>
<tr>
<th>Start addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>22</td>
</tr>
<tr>
<td>c</td>
<td>30</td>
</tr>
<tr>
<td>e</td>
<td>14</td>
</tr>
</tbody>
</table>

What does Free List Table look like?
What if we cannot find a big enough hole for an arriving job?

Suppose a 35K job arrives?
Suppose a 90K job arrives?

What do you do?

<table>
<thead>
<tr>
<th>Free</th>
<th>22 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>24 K</td>
</tr>
<tr>
<td>Free</td>
<td>30 K</td>
</tr>
<tr>
<td>Free</td>
<td>10 K</td>
</tr>
<tr>
<td>Free</td>
<td>14 K</td>
</tr>
</tbody>
</table>

Compaction

Shuffle jobs to create larger contiguous free memory

<table>
<thead>
<tr>
<th>Job A 15 K</th>
<th>FREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job B 20 K</td>
<td>FREE</td>
</tr>
<tr>
<td>Job C 7K</td>
<td>FREE</td>
</tr>
<tr>
<td>A</td>
<td>15K</td>
</tr>
<tr>
<td>B</td>
<td>20K</td>
</tr>
<tr>
<td>C</td>
<td>7K</td>
</tr>
<tr>
<td>FREE</td>
<td>58K</td>
</tr>
</tbody>
</table>

Now 40 K job can run

QTP: How about pointers?
Pros/Cons of Dynamic Partitions

😊 **Advantages:**
- Efficient memory usage

😊 **Disadvantages:**
- Partition Management
- Compaction or external fragmentation
- Internal fragmentation (if blocks composing partitions are always allocated in fixed sized units -- e.g. 2k)

Multiple Segment Relocation registers

[Diagram of CPU, Code Register, Stack register, Data Register, Memory Address Register, and Primary Memory interconnected with arrows and labels: (Generated Address), Relative Address, Memory Address Register, Primary Memory]