Memory Management, Early Systems

- Single-User Contiguous Scheme
- Fixed Partitions
- Dynamic Partitions
- Deallocation
- Relocatable Dynamic Partitions

Single User Configurations
  ↓
  Fixed Partitions
  ↓
  Dynamic Partitions
  ↓
  Relocatable Dynamic Partitions
Single-User Contiguous Scheme

- Each program loaded in its entirety into memory and allocated as much contiguous memory space as needed.

- If program was too large -- it couldn’t be executed.

- Minimal amount of work done by Memory Manager.

- Hardware needed: 1) register to store base address; 2) accumulator to track size of program as it is loaded into memory.
Algorithm to Load a Job in a Single-user System

1. Store first memory location of program into base register
2. Set program counter equal to address of first memory location
3. Load instructions of program
4. Increment program counter by number of bytes in instructions
5. Has the last instruction been reached?
   If yes, then stop loading program
   If no, then continue with step 6
6. Is program counter greater than memory size?
   If yes, then stop loading.
   If no, then continue with step 7
7. Load instruction in memory
8. Go to step 3.
Fixed (Static) Partitions

- Attempt at multiprogramming using fixed partitions
  - one partition for each job
  - size of partition designated by reconfiguring the system
  - partitions can’t be too small or too large.

- Critical to protect job’s memory space.

- Entire program stored contiguously in memory during entire execution.

- Internal fragmentation is a problem.
Algorithm to Load a Job in a Fixed Partition

1. Determine job’s requested memory size
2. If job_size > size of largest partition then reject job
   - Print appropriate message
   - Go to step 1 to handle next job
Else continue with step 3
3. Set counter to 1
4. Do while counter <= number of partitions in memory
   - If job_size > mem_partition_size (counter)
     - then counter = counter + 1
   - Else
     - If mem_partition_status (counter) = “free”
       - then load job into mem_partition(counter)
       - change mem_partition_status(counter) to “busy”
       - go to step 1
     - Else counter = counter + 1
   End do
5. No partition available at this time, put job in waiting queue
6. Go to step 1
**Simplified Fixed Partition Memory Table (Table 2.1)**

<table>
<thead>
<tr>
<th>Partition size</th>
<th>Memory address</th>
<th>Access</th>
<th>Partition status</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>200K</td>
<td>Job 1</td>
<td>Busy</td>
</tr>
<tr>
<td>25K</td>
<td>300K</td>
<td>Job 4</td>
<td>Busy</td>
</tr>
<tr>
<td>25K</td>
<td>325K</td>
<td></td>
<td>Free</td>
</tr>
<tr>
<td>50K</td>
<td>350K</td>
<td>Job 2</td>
<td>Busy</td>
</tr>
</tbody>
</table>
Table 2.1: Main memory use during fixed partition allocation of Table 2.1. Job 3 must wait.
Dynamic Partitions

• Available memory kept in contiguous blocks and jobs given only as much memory as they request when loaded.

• Improves memory use over fixed partitions.

• Performance deteriorates as new jobs enter the system
  – fragments of free memory are created between blocks of allocated memory (external fragmentation).
Dynamic Partitioning of Main Memory & Fragmentation (Figure 2.2)
Dynamic Partition Allocation Schemes

• **First-fit**: Allocate the *first* partition that is big enough.
  – Keep free/busy lists organized by memory location (low-order to high-order).
  – Faster in making the allocation.

• **Best-fit**: Allocate the *smallest* partition that is big enough
  – Keep free/busy lists ordered by size (smallest to largest).
  – Produces the smallest leftover partition.
  – Makes best use of memory.
**First-Fit Allocation Example**  
*(Table 2.2)*

<table>
<thead>
<tr>
<th>Job</th>
<th>Internal size</th>
<th>Job size</th>
<th>Status</th>
<th>Internal fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>20K</td>
<td>10K</td>
<td>Busy</td>
<td>20K</td>
</tr>
<tr>
<td>J4</td>
<td>5K</td>
<td>5K</td>
<td>Busy</td>
<td>5K</td>
</tr>
<tr>
<td>J2</td>
<td>30K</td>
<td>20K</td>
<td>Busy</td>
<td>30K</td>
</tr>
<tr>
<td>J3</td>
<td>30K</td>
<td>10K</td>
<td>Free</td>
<td></td>
</tr>
</tbody>
</table>

**Job List**

<table>
<thead>
<tr>
<th>Job</th>
<th>Internal size</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>10K</td>
</tr>
<tr>
<td>J2</td>
<td>20K</td>
</tr>
<tr>
<td>J3</td>
<td>30K*</td>
</tr>
<tr>
<td>J4</td>
<td>10K</td>
</tr>
</tbody>
</table>

- **Memory Location**: 10240  
- **Memory Block Size**: 30K  
- **Job Number**: J1  
- **Job Size**: 10K  
- **Status**: Busy  
- **Internal Fragmentation**: 20K

- **Memory Location**: 40960  
- **Memory Block Size**: 15K  
- **Job Number**: J4  
- **Job Size**: 10K  
- **Status**: Busy  
- **Internal Fragmentation**: 5K

- **Memory Location**: 56320  
- **Memory Block Size**: 50K  
- **Job Number**: J2  
- **Job Size**: 20K  
- **Status**: Busy  
- **Internal Fragmentation**: 30K

- **Memory Location**: 107520  
- **Memory Block Size**: 20K  
- **Job Number**: J3  
- **Job Size**: 10K  
- **Status**: Free  
- **Internal Fragmentation**: 10K

**Total Available:** 115K  
**Total Used:** 40K
## Best-Fit Allocation Example

*(Table 2.3)*

**Job List**

<table>
<thead>
<tr>
<th>Job</th>
<th>Job Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>10K</td>
</tr>
<tr>
<td>J2</td>
<td>20K</td>
</tr>
<tr>
<td>J3</td>
<td>30K</td>
</tr>
<tr>
<td>J4</td>
<td>10K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Memory Block Size</th>
<th>Job Number</th>
<th>Job Size</th>
<th>Status</th>
<th>Internal Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>40960</td>
<td>15K</td>
<td>J1</td>
<td>10K</td>
<td>Busy</td>
<td>5K</td>
</tr>
<tr>
<td>107520</td>
<td>20K</td>
<td>J2</td>
<td>20K</td>
<td>Busy</td>
<td>None</td>
</tr>
<tr>
<td>10240</td>
<td>30K</td>
<td>J3</td>
<td>30K</td>
<td>Busy</td>
<td>None</td>
</tr>
<tr>
<td>56230</td>
<td>50K</td>
<td>J4</td>
<td>10K</td>
<td>Busy</td>
<td>40K</td>
</tr>
</tbody>
</table>

**Total Available:** 115K  **Total Used:** 70K
First-Fit Algorithm

1. Set counter to 1
2. Do while counter <= number of blocks in memory
   
   If job_size > memory_size(counter)
   
   then counter = counter + 1
   
   else
   
   load job into memory_size(counter)
   
   adjust free/busy memory lists
   
   go to step 4

   End do

3. Put job in waiting queue
4. Go fetch next job
# First-Fit Memory Request (Table 2.4)

<table>
<thead>
<tr>
<th>Before request</th>
<th>After request</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Beginning address</td>
<td>Memory block size</td>
</tr>
<tr>
<td>4075</td>
<td>105</td>
</tr>
<tr>
<td>5225</td>
<td>5</td>
</tr>
<tr>
<td>6785</td>
<td>600</td>
</tr>
<tr>
<td>7560</td>
<td>20</td>
</tr>
<tr>
<td>7600</td>
<td>205</td>
</tr>
<tr>
<td>10250</td>
<td>4050</td>
</tr>
<tr>
<td>15125</td>
<td>230</td>
</tr>
<tr>
<td>24500</td>
<td>1000</td>
</tr>
</tbody>
</table>
Best-Fit Algorithm

1. Initialize mem_block(0) = 99999
2. Compute initial_mem_waste = memory_block(0) – job_size
3. Initialize subscript = 0
4. Set counter to 1
5. Do while counter <= number of blocks in memory
   If job_size > mem_size(counter)
   Then counter = counter + 1
   Else
   mem_waste = mem_size(counter) – job_size
6. If subscript = 0
   Then put job in waiting queue
   Else
   load job into mem_size(subscript)
   adjust free/busy memory lists
7. Go fetch next job

If initial_mem_waste > mem_waste
Then subscript = counter
   initial_mem_waste = mem_waste
   counter = counter + 1
End do
## Best-Fit Memory Request (Table 2.5)

<table>
<thead>
<tr>
<th>Before request</th>
<th>After request</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Beginning</strong></td>
<td><strong>Beginning</strong></td>
</tr>
<tr>
<td>address</td>
<td>address</td>
</tr>
<tr>
<td>4075</td>
<td>4075</td>
</tr>
<tr>
<td>5225</td>
<td>5225</td>
</tr>
<tr>
<td>6785</td>
<td>6785</td>
</tr>
<tr>
<td>7560</td>
<td>7560</td>
</tr>
<tr>
<td>7600</td>
<td>7600</td>
</tr>
<tr>
<td>10250</td>
<td>10250</td>
</tr>
<tr>
<td>15125</td>
<td>15125</td>
</tr>
<tr>
<td>24500</td>
<td>24500</td>
</tr>
</tbody>
</table>
# Best-Fit vs. First-Fit

**First-Fit**
- Increases memory use
- Memory allocation takes more time
- Reduces internal fragmentation

**Best-Fit**
- More complex algorithm
- Searches entire table before allocating memory
- Results in a smaller “free” space (sliver)
Release of Memory Space: Deallocation

• Deallocation for fixed partitions is simple
  – Memory Manager resets status of memory block to “free”.

• Deallocation for dynamic partitions tries to combine free areas of memory whenever possible
  – Is the block adjacent to another free block?
  – Is the block between 2 free blocks?
  – Is the block isolated from other free blocks?
Algorithm to Deallocation Memory Blocks

If job_location is adjacent to 1+ free blocks
Then
If job_location is between 2 free blocks
Then merge all 3 blocks into 1 block
mem_size(counter-1) =
mem_size(counter-1) + job_size + mem_size(counter+1)
Set status of mem_size(counter+1)
to null entry
Else
merge both blocks into one
mem_size(counter-1) =
mem_size(counter-1) + job_size
Else
search for null entry in free memory list
enter job_size and beginning_address in entry slot
set its status to “free”
## Case 1: Joining 2 Free Blocks

<table>
<thead>
<tr>
<th>Before</th>
<th>Deallocation</th>
<th>After</th>
<th>Deallocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning address</td>
<td>Memory block size</td>
<td>Status</td>
<td>Beginning address</td>
</tr>
<tr>
<td>4075</td>
<td>105</td>
<td>Free</td>
<td>4075</td>
</tr>
<tr>
<td>5225</td>
<td>5</td>
<td>Free</td>
<td>5225</td>
</tr>
<tr>
<td>6785</td>
<td>600</td>
<td>Free</td>
<td>6785</td>
</tr>
<tr>
<td>7560</td>
<td>20</td>
<td>Free</td>
<td>7560</td>
</tr>
<tr>
<td>(7600)</td>
<td>(200)</td>
<td>(Busy)$^1$</td>
<td>*7600</td>
</tr>
<tr>
<td>*7800</td>
<td>5</td>
<td>Free</td>
<td>10250</td>
</tr>
<tr>
<td>10250</td>
<td>4050</td>
<td>Free</td>
<td>15125</td>
</tr>
<tr>
<td>15125</td>
<td>230</td>
<td>Free</td>
<td>24500</td>
</tr>
<tr>
<td>24500</td>
<td>1000</td>
<td>Free</td>
<td></td>
</tr>
</tbody>
</table>

$^1$ This block is busy.
### Case 2: Joining 3 Free Blocks

<table>
<thead>
<tr>
<th>Before</th>
<th>Deallocation</th>
<th>After</th>
<th>Deallocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning address</td>
<td>Memory block size</td>
<td>Status</td>
<td>Beginning address</td>
</tr>
<tr>
<td>4075</td>
<td>105</td>
<td>Free</td>
<td>4075</td>
</tr>
<tr>
<td>5225</td>
<td>5</td>
<td>Free</td>
<td>5225</td>
</tr>
<tr>
<td>6785</td>
<td>600</td>
<td>Free</td>
<td>6785</td>
</tr>
<tr>
<td>7560</td>
<td>20</td>
<td>Free</td>
<td>7560</td>
</tr>
<tr>
<td>(7600)</td>
<td>(200)</td>
<td>(Busy)(^1)</td>
<td>*</td>
</tr>
<tr>
<td>*7800</td>
<td>5</td>
<td>Free</td>
<td>10250</td>
</tr>
<tr>
<td>10250</td>
<td>4050</td>
<td>Free</td>
<td>15125</td>
</tr>
<tr>
<td>15125</td>
<td>230</td>
<td>Free</td>
<td>24500</td>
</tr>
<tr>
<td>24500</td>
<td>1000</td>
<td>Free</td>
<td></td>
</tr>
</tbody>
</table>
Case 3: Deallocation of an Isolated Block

<table>
<thead>
<tr>
<th>Busy List Before</th>
<th>Busy List After</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Beginning address</strong></td>
<td><strong>Beginning address</strong></td>
</tr>
<tr>
<td><strong>Memory block size</strong></td>
<td><strong>Memory block size</strong></td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td><strong>Status</strong></td>
</tr>
<tr>
<td>7805</td>
<td>7805</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Busy</td>
<td>Busy</td>
</tr>
<tr>
<td>*8805</td>
<td>*</td>
</tr>
<tr>
<td>445</td>
<td>(null entry)</td>
</tr>
<tr>
<td>Busy</td>
<td></td>
</tr>
<tr>
<td>9250</td>
<td>9250</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Busy</td>
<td>Busy</td>
</tr>
</tbody>
</table>
Relocatable Dynamic Partitions

- Memory Manager relocates programs to gather all empty blocks and compact them to make 1 memory block.

- Memory compaction (garbage collection, defragmentation) performed by OS to reclaim fragmented sections of memory space.

- Memory Manager optimizes use of memory & improves throughput by compacting & relocating.
Compaction Steps

• Relocate every program in memory so they’re contiguous.

• Adjust every address, and every reference to an address, within each program to account for program’s new location in memory.

• Must leave alone all other values within the program (e.g., data values).
Original Assembly Language Program
(Figure 2.4)

A: EXP 132, 144, 125, 110 ;the data values

BEGIN: MOVEI 1,0 ;initialize register 1
MOVEI 2,0 ;initialize register 2

LOOP: ADD 2,A(1) ;add (A + reg 1) to reg 2
ADDI 1,1 ;add 1 to reg 1
CAIG 1,4-1 ;is reg 1 > 4-1?
JUMPA LOOP ;if not, go to Loop

MOVE 3,2 ;if so, move reg 2 to reg 3
IDIVI 3,4 ;divide reg 3 by 4,
;remainder to register 4
EXIT ;end

END
Assembly Language Program Loaded into Memory (Figure 2.4)

```
A: EXP 132,144,125,110

BEGIN: MOVEI 1,0
MOVEI 2,0
LOOP: ADD 2,A(1)
ADDI 1,1
CAIG 1,4-1
JUMPA LOOP
MOVE 3,2
IDIVI 3,4
EXIT
```

000000

END
Program in Memory During Compaction & Relocation

- Free list & busy list are updated
  - free list shows partition for new block of free memory
  - busy list shows new locations for all relocated jobs

- **Bounds register** stores highest location in memory accessible by each program.

- **Relocation register** contains value that must be added to each address referenced in program so it can access correct memory addresses after relocation.
Memory Before & After Compaction
(Figure 2.5)
Contents of relocation register & close-up of Job 4 memory area (a) before relocation & (b) after relocation and compaction (Figure 2.6)
More Overhead is a Problem with Compaction & Relocation

- Timing of compaction (when, how often) is crucial.

- Approaches to timing of compaction:
  1. Compact when certain percentage of memory is busy (e.g., 75%).
  2. Compact only when jobs are waiting.
  3. Compact after certain amount of time.
Key Terms

- address
- best-fit memory allocation
- bounds register
- compaction
- deallocation
- dynamic partitions
- external fragmentation
- first come first served
- first-fit memory allocation
- fixed partitions
- internal fragmentation
- K
- multiprogramming
- relocatable dynamic partitions
- relocation
- relocation register