Memory Management

Dr. Tanir Ozcelebi

by courtesy of
Igor Radovanović &
Rudolf Mak
REMINDER: OS as resource manager

- **Process:** An executing program
- **Resource:** Anything that is needed for a process to run
  - Memory
  - Space on a disk
  - CPU
- “An OS creates resource abstractions”
- “An OS manages resource sharing”
REMINDER: Resource sharing

- Space- vs. time-multiplexed sharing
- To control sharing, must be able to isolate resources
- OS usually provides mechanism to isolate, then selectively allows sharing
  - How to isolate resources?
  - How to be sure that sharing is acceptable?
REMINDER: Logical OS Organization

- All modules interact to coordinate their activities
- Examples:
  - Virtual Memory
    - PM –MM to coordinate scheduling activity with memory allocation
  - For performance improvement: FM –DM & FM-MM
    - Information from storage device read prior to request by a thread
User/programmer perspective

Ideally a programmer would like to have memory that
• is non-volatile
• is private (protected)
• has infinite capacity
• has zero access time
• is simple to use
• is cheap

Such a memory clearly does not exist
• but virtualization combined with a hierarchical hardware organization can help!
Storage Hierarchies

- Analogy with the office storage hierarchy
  - Worker’s desktop
  - File folder
  - File cabinet
  - Warehouse

- Upper-layers: info more frequently used
- Lower layers: info less frequently used
Memory Hierarchy (cnt’d)

- CPU register memory
  - fast access (1 clk cycle)
  - but limited in size
- Primary (executable memory)
  - direct access (a few clk cycles),
  - relatively larger,
  - access time is of less importance
- Secondary memory (storage devices)
  - I/O operations required for access,
  - very cheap / very large,
  - data stored for longer period of time,
  - access speed is slow (3 orders of magnitude slower than the CPU register memory)
Contemporary Memory Hierarchy & Dynamic Loading

Primary (Executable)
- CPU Registers
- L1 Cache Memory
- L2 Cache Memory
- "Main" Memory

Secondary
- Rotating Magnetic Memory
- Optical Memory
- Sequentially Accessible Memory

Larger storage → Faster access
Exploiting the Hierarchy

- Place frequently-used info high and infrequently-used info low in the hierarchy
- Reconfigure as process changes phases (process switch)
- Upward moves are (usually) *copy* operations
  - Require allocation in upper memory
  - Image exists in both higher & lower memories
- Updates are first applied to upper memory
- Downward move is (usually) *destructive*
  - Destroy image in upper memory
  - Update image in lower memory
  - should be avoided when the copy in lower memory is still consistent
Memory manager functional requirements

- Classic memory managers
  - Abstraction
    - Main memory appears to be larger than the physical machine mem.
    - Memory as a large array of contiguously addressed bytes
    - Abstract set of logical addresses to reference physical primary mem.
  - Allocation
    - Allocate primary memory to processes as requested
  - Isolation
    - Exclusive use of memory by processes
  - Memory Sharing
    - Two types:
      - Main memory shared by many processes
      - Block of primary memory shared by many processes
Memory manager non-functional requirements

• Performance requirements
  – Minimum main memory access time
  – Maximum main memory size
  – Cost-effectiveness
The Address Space Abstraction

- Instead of a set of physical primary memory addresses, processes are allocated a set of logical primary memory addresses (process address space).
- Memory manager has to bind logical primary memory addresses to physical primary memory addresses.
Virtual and Physical address

- **Decouple**
  - addresses (names) used by a *process* to identify *objects*; from addresses used by *memory* to identify *storage locations*

- **Objects in different processes**
  - can have the same name (logical address),
  - but, must be stored at different physical locations
Binding issues

At which moment?

- **static**, i.e., during translation of program source to executable load module, or upon activation of a process (loading)
  - at programming time
  - at compile time
  - at link time
  - at load time

- **dynamic**, i.e., during execution of the instructions of a running process (as late as possible).
  - typically in load and store instructions for data
  - typically in jump and branch instructions for flow of control
Static binding

- **Compile time**
  - Isolated addresses of relocatable object modules

- **Link time**
  - Addresses of the load module stored in the file in the secondary memory
    - Program instructions
    - Data
    - Stack

- **Load time**
  - Binding logical to physical addresses
    - Static binding
Program transformation (static binding)

```
int i
i = ... f()
```

- **Source module**
  - `int i`
  - `i = ... f()`

- **Object module**
  - `store 20 branch f`

- **Load module (secondary memory)**
  - `store 120 branch 0`

- **Load module (main memory)**
  - `store 1120 branch 1000`
Dynamic binding

- immediately before executing an instruction
- translation using an address map
  - \texttt{physical\_address = address\_map (logical\_address)}
  - hardware support for efficiency
Program transformation (dynamic binding)

int \( i \)

\( i = \ldots \)

\( f(\; ) \)

source module

object module

load module (secondary memory)

load module (main memory)

address_map

execution
Early systems

- Single-user configuration
- Multi-user configurations
  - Fixed partitions
  - Dynamic partitions
  - Relocatable Dynamic partitions
Single-user configuration

• Each program loaded in its entirety into memory and allocated as much contiguous memory space as needed
  – If a program does not fit in the memory, it cannot be executed

• Hardware needed
  – register to store base address
  – accumulator to track size of program as it is loaded into memory (limit)

• Once in main memory, program stays there until execution ends

• No multiprogramming supported
Early systems

- Single-user configuration
- Multi-user configurations
  - Fixed partitions
  - Dynamic partitions
  - Relocatable Dynamic partitions
Fixed partitions

- 1 process/partition
  - Issue: protection of the task’s memory space

- Disadvantage:
  - entire program is stored contiguously in memory during entire execution
    - Problem: Internal fragmentation

- Design issue: Partition size
  - Choosing the right partition size was a problem
    - Too small – long turnaround time
    - Too big – wasted memory

- Partition allocation (e.g. first-fit)
Fixed partition  
-an example-  

Process List:

<table>
<thead>
<tr>
<th>Process</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>30K</td>
</tr>
<tr>
<td>P2</td>
<td>50K</td>
</tr>
<tr>
<td>P3</td>
<td>30K</td>
</tr>
<tr>
<td>P4</td>
<td>25K</td>
</tr>
</tbody>
</table>

Conclusion: Process 3 has to wait even though 70K of free space is available.

Original State

<table>
<thead>
<tr>
<th></th>
<th>100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td>25K</td>
</tr>
<tr>
<td>Partition 2</td>
<td>25K</td>
</tr>
<tr>
<td>Partition 3</td>
<td>50K</td>
</tr>
</tbody>
</table>

After Process Entry

<table>
<thead>
<tr>
<th></th>
<th>50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td></td>
</tr>
<tr>
<td>Partition 2</td>
<td>Process 4 (25K)</td>
</tr>
<tr>
<td>Partition 3</td>
<td></td>
</tr>
<tr>
<td>Partition 4</td>
<td>Process 2 (50K)</td>
</tr>
</tbody>
</table>
Early systems

- Single-user configuration
- Multi-user systems
  - Fixed partitions
  - Dynamic partitions
    - allocation
    - deallocation
    - data structures
  - Relocatable Dynamic partitions
Dynamic partitions

- Partition size not fixed but determined based on the size of a process

- Advantage:
  - less memory waste
    - no internal fragmentation

- Disadvantages:
  - Memory waste not solved completely
    - Performance deteriorates as new processes enter and old processes exit the system
      - External fragmentation
  - Programs still have to be stored contiguously
Dynamic partitions - an example -

First-fit allocation scheme

Initial job allocation (a)

Job List
J1 10k
J2 15k
J3 20k
J4 50k

Operating System
10k
Job 1(10k)
20k
Job 2(15k)
35k
Job 3(20k)
55k
Job 4(50k)
105k

Job 1 ends

Job 2(15k)
35k
Job 3(20k)
55k
Job 4 ends

Job 4(50k)
105k

After Job 1 and Job 4 have finished (b)

Job 5 (5k) arrives

Operating System
10k
20k
Job 2(15k)
35k
Job 3(20k)
55k
Job 6 (30k) arrives

Operating System
10k
20k
Job 2(15k)
35k
Job 3(20k)
55k
Job 6(30k)
85k

After Job 5 and Job 6 have entered (c)
Dynamic partitions -an example-

First-fit allocation scheme (cont’d)

<table>
<thead>
<tr>
<th>Job 3 ends</th>
<th>Operating System</th>
<th>Job 5 (5k)</th>
<th>10k</th>
<th>15k</th>
<th>20k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 2 (15k)</td>
<td></td>
<td>35k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job 6 (30k)</td>
<td></td>
<td>55k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Job 3 has finished (d)</td>
<td></td>
<td>85k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job 7 (10k) arrives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 7 (10k)</td>
</tr>
<tr>
<td>Job 8 (10k)</td>
</tr>
<tr>
<td>After Job 7 has entered (e)</td>
</tr>
<tr>
<td>Job 8 (30k)</td>
</tr>
<tr>
<td>105k</td>
</tr>
</tbody>
</table>

Job 8 has to wait

Conclusion:
Job 8 has to wait even though there is 35K of free memory space available
Dynamic partition -allocation schemes-

- Allocation schemes issues: decrease time, increase utilization

- First-fit
  - Allocate the first partition that is big enough
  - Advantage:
    - Faster in making the allocation
  - Disadvantage:
    - Waste of memory

- Best-fit
  - Allocate the smallest partition fitting the requirements
  - Advantages:
    - Produces the smallest leftover partition
    - Makes best use of memory when it is possible to allocate a new job
  - Disadvantage:
    - Takes more time

Q: How to organize free/busy partition list in both cases?
First-fit allocation - an example -

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Memory block size</th>
<th>Process number</th>
<th>Process size</th>
<th>Status</th>
<th>Internal fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10240</td>
<td>30K</td>
<td>P1</td>
<td>10K</td>
<td>Busy</td>
<td>20K</td>
</tr>
<tr>
<td>40960</td>
<td>15K</td>
<td>P4</td>
<td>10K</td>
<td>Busy</td>
<td>5K</td>
</tr>
<tr>
<td>56320</td>
<td>50K</td>
<td>P2</td>
<td>20K</td>
<td>Busy</td>
<td>30K</td>
</tr>
<tr>
<td>107520</td>
<td>20K</td>
<td></td>
<td></td>
<td>Free</td>
<td></td>
</tr>
</tbody>
</table>

Total Available: 115K  Total Used: 40K
### Best-fit allocation -an example-

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Memory block size</th>
<th>Process number</th>
<th>Process size</th>
<th>Status</th>
<th>Internal fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>40960</td>
<td>15K</td>
<td>P1</td>
<td>10K</td>
<td>Busy</td>
<td>5K</td>
</tr>
<tr>
<td>107520</td>
<td>20K</td>
<td>P2</td>
<td>20K</td>
<td>Busy</td>
<td>None</td>
</tr>
<tr>
<td>10240</td>
<td>30K</td>
<td>P3</td>
<td>30K</td>
<td>Busy</td>
<td>None</td>
</tr>
<tr>
<td>56230</td>
<td>50K</td>
<td>P4</td>
<td>10K</td>
<td>Busy</td>
<td>40K</td>
</tr>
</tbody>
</table>

**Total Available:** 115K  
**Total Used:** 70K
Exercise

- The table of free memory spaces is given below. Show how this table will look like after allocating a process that occupies 200 spaces using the **best-fit algorithm**.

<table>
<thead>
<tr>
<th>Before request</th>
<th>After request</th>
</tr>
</thead>
<tbody>
<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>

Small free space
- sliver -
Allocation schemes

- First-Fit
- Best-Fit
- Worst-Fit
- Next-Fit (rotating First-Fit)
  - Faster search, slightly worse memory utilization than First-Fit
- Optimized schemes
  - Statistics on avg process size etc. needed
Early systems

- Single-user configuration
- Multi-user systems
  - Fixed partitions
  - Dynamic partitions
    - allocation
    - deallocation
    - data structures
  - Relocatable Dynamic partitions
Release of memory space - deallocation -

- Occurs when process terminates or becomes suspended

- 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?
    - If yes, combine the 2 blocks
  - Is the block between 2 free blocks?
    - If yes, combine those 3 blocks
  - Is the block isolated from other free blocks?
    - If yes, put it in the table entry
# Deallocation - an example -

<table>
<thead>
<tr>
<th>Beginning address</th>
<th>Memory block size</th>
<th>Status</th>
<th>Beginning address</th>
<th>Memory block size</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>4075</td>
<td>105</td>
<td>Free</td>
<td>4075</td>
<td>105</td>
<td>Free</td>
</tr>
<tr>
<td>5225</td>
<td>5</td>
<td>Free</td>
<td>5225</td>
<td>5</td>
<td>Free</td>
</tr>
<tr>
<td>6785</td>
<td>600</td>
<td>Free</td>
<td>6785</td>
<td>600</td>
<td>Free</td>
</tr>
<tr>
<td>7560</td>
<td>40</td>
<td>Free</td>
<td>7560</td>
<td>40</td>
<td>Free</td>
</tr>
<tr>
<td>(7600)</td>
<td>(200)</td>
<td>(Busy)¹</td>
<td>(7600)</td>
<td>(200)</td>
<td>(Busy)¹</td>
</tr>
<tr>
<td>*7800</td>
<td>5</td>
<td>Free</td>
<td>10250</td>
<td>4050</td>
<td>Free</td>
</tr>
<tr>
<td>10250</td>
<td>4050</td>
<td>Free</td>
<td>15125</td>
<td>230</td>
<td>Free</td>
</tr>
<tr>
<td>15125</td>
<td>230</td>
<td>Free</td>
<td>24500</td>
<td>1000</td>
<td>Free</td>
</tr>
</tbody>
</table>

¹: Indicates a block that is currently marked as busy.
Early systems

- Single-user configuration
- Multi-user systems
  - Fixed partitions
  - Dynamic partitions
    - allocation
    - deallocation
    - data structures
  - Relocatable Dynamic partitions
Dynamic partitions - data structures -

- **Linked lists**
  - each segment of contiguous memory is equipped with
    - some additional data stating its status (occupied or free)
    - the starting address of the next segment

- **Bitmap** (assumes that memory is handed out in blocks)
  - For each block, there is a separate bit that indicates whether it is occupied or free.

- **Buddy system** (combination of fixed and variable size partitions)
  - All partitions have a size that is a power of two.
  - Each partition of size $2^m$ can be dynamically split in two partitions of size $2^{(m-1)}$, called buddies.
Early systems

- Single-user configuration
- Multi-user configurations
  - Fixed partitions
  - Dynamic partitions
  - Relocatable Dynamic partitions
    - compaction
    - swapping
Relocatable Dynamic partitions

- Solves problem of both internal and external fragmentation
  - Compaction and relocation (increased throughput)
    - avoid memory waste
    - makes insufficient memory problems less frequent

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

Q:
- When should compaction be done?
  - When a certain percent of memory becomes busy (e.g. 75%),
  - When there are processes pending, or
  - After a prescribed amount of time

Note: Compaction cannot be done if the relocation is static
Compaction - an example -

- Memory before and after compaction

<table>
<thead>
<tr>
<th>Job</th>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 1</td>
<td>10k 18k 30k 62k</td>
</tr>
<tr>
<td>Job 4</td>
<td>10k 18k 30k 62k</td>
</tr>
<tr>
<td>Job 2</td>
<td>10k 18k 30k 62k</td>
</tr>
<tr>
<td>Job 5</td>
<td>10k 18k 30k 62k</td>
</tr>
</tbody>
</table>

Now Job 6 Can be accommodated

| Job 1  | 10k 18k 30k 62k |
| Job 4  | 10k 18k 30k 62k |
| Job 2  | 10k 18k 30k 62k |
| Job 5  | 10k 18k 30k 62k |
| Job 6  | 10k 18k 30k 62k |

Now Job 6 Can be accommodated

| Job 1  | 10k 18k 30k 62k |
| Job 4  | 10k 18k 30k 62k |
| Job 2  | 10k 18k 30k 62k |
| Job 5  | 10k 18k 30k 62k |
| Job 6  | 10k 18k 30k 62k |

Now Job 6 Can be accommodated

| Job 1  | 10k 18k 30k 62k |
| Job 4  | 10k 18k 30k 62k |
| Job 2  | 10k 18k 30k 62k |
| Job 5  | 10k 18k 30k 62k |
| Job 6  | 10k 18k 30k 62k |
Memory compaction (strategies)

(a) initial state; (b) complete compaction
(c) partial compaction; (d) minimal data movement
Memory compaction (remarks)

- Very costly
  - Require each word of memory to be read from and written into memory
    - May take several seconds

- Contemporary systems
  - Virtual memory techniques instead of compaction
Relocation

• Introduces overhead

Q:
• Who keeps track of how each job has moved from its original storage area?

A:
• Special-purpose registers
  – Relocation register
    • Contains value to be added to each address referenced in the program
  – Bounds register
    • Stores highest (lowest) location of memory accessible by each program
Relocation -an example-

- Job loaded into memory location 30K (30720)
- It requires memory block of 32K (32768)
- At memory location 31744 an instruction: LOAD 4, ANSWER
  - Answer is the value 37
Swapping

• Changes process status between active and suspended

• Why?
  – because there is insufficient memory to activate a new process

• What?
  – OS selects a process to swap-out and replaces it with another process from the secondary storage
  – only those parts of process space that are not already on disk
    • typically code is not modified
    • benefits from hardware support that registers modification

• Where to (on secondary memory)?
  – either to an arbitrary file (introduces file management overhead)
  – or to a special partition on disk, the swap space
    • more efficient low-level I/O
Conclusions

- Single-user configuration
- Fixed partitions
- Dynamic partitions
- Relocatable Dynamic partitions

Comments:
- All require that the entire program
  - be loaded into memory
  - be stored contiguously
  - remain in memory until a process is completed
- Severe restrictions to the process size