Chapter 7
Memory Management
Seventh Edition
William Stallings
I cannot guarantee that I carry all the facts in my mind. Intense mental concentration has a curious way of blotting out what has passed. Each of my cases displaces the last, and Mlle. Carère has blurred my recollection of Baskerville Hall. Tomorrow some other little problem may be submitted to my notice which will in turn dispossess the fair French lady and the infamous Upwood.

— THE HOUND OF THE BASKERVILLES, Arthur Conan Doyle
# Memory Management Terms

<table>
<thead>
<tr>
<th>Frame</th>
<th>A fixed-length block of main memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td>A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.</td>
</tr>
<tr>
<td>Segment</td>
<td>A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).</td>
</tr>
</tbody>
</table>
Memory Management Requirements

Memory management is intended to satisfy the following requirements:

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization
Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program.

- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization.

- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting.
  - may need to relocate the process to a different area of memory.
Addressing Requirements

Figure 7.1  Addressing Requirements for a Process
Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection
Sharing

- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection
- Mechanisms used to support relocation support sharing capabilities
Logical Organization

- Memory is organized as linear

Programs are written in modules

- modules can be written and compiled independently
- different degrees of protection given to modules (read-only, execute-only)
- sharing on a module level corresponds to the user’s way of viewing the problem

- Segmentation is the tool that most readily satisfies requirements
Physical Organization

Cannot leave the programmer with the responsibility to manage memory

Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

*overlapping* allows various modules to be assigned the same region of memory but is time consuming to program
Memory Partitioning

- Memory management brings processes into main memory for execution by the processor
  - involves virtual memory
  - based on segmentation and paging
- Partitioning
  - used in several variations in some now-obsolete operating systems
  - does not involve virtual memory
Table 7.2
Memory Management Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Partitioning</td>
<td>Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.</td>
<td>Simple to implement; little operating system overhead.</td>
<td>Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.</td>
</tr>
<tr>
<td>Dynamic Partitioning</td>
<td>Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.</td>
<td>No internal fragmentation; more efficient use of main memory.</td>
<td>Inefficient use of processor due to the need for compaction to counter external fragmentation.</td>
</tr>
<tr>
<td>Simple Paging</td>
<td>Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.</td>
<td>No external fragmentation.</td>
<td>A small amount of internal fragmentation.</td>
</tr>
<tr>
<td>Simple Segmentation</td>
<td>Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.</td>
<td>No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.</td>
<td>External fragmentation.</td>
</tr>
<tr>
<td>Virtual Memory Paging</td>
<td>As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.</td>
<td>No external fragmentation; higher degree of multiprogramming; large virtual address space.</td>
<td>Overhead of complex memory management.</td>
</tr>
<tr>
<td>Virtual Memory Segmentation</td>
<td>As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.</td>
<td>No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.</td>
<td>Overhead of complex memory management.</td>
</tr>
</tbody>
</table>
Fixed Partitioning

- Equal-size partitions
  - any process whose size is less than or equal to the partition size can be loaded into an available partition

- The operating system can swap out a process if all partitions are full and no process is in the Ready or Running state
Disadvantages

- A program may be too big to fit in a partition
  - program needs to be designed with the use of overlays
- Main memory utilization is inefficient
  - any program, regardless of size, occupies an entire partition
  - *internal fragmentation*
    - wasted space due to the block of data loaded being smaller than the partition
Unequal Size Partitions

- Using unequal size partitions helps lessen the problems
- Programs up to 16M can be accommodated without overlays
- Partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation
Memory Assignment

Figure 7.3 Memory Assignment for Fixed Partitioning
Disadvantages

- The number of partitions specified at system generation time limits the number of active processes in the system.
- Small jobs will not utilize partition space efficiently.
Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM’s mainframe operating system, OS/MVT
Effect of Dynamic Partitioning

Figure 7.4 The Effect of Dynamic Partitioning
**Dynamic Partitioning**

**External Fragmentation**
- Memory becomes more and more fragmented
- Memory utilization declines

**Compaction**
- Technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- Free memory is together in one block
- Time consuming and wastes CPU time
# Placement Algorithms

<table>
<thead>
<tr>
<th>Placement Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best-fit</strong></td>
<td>- choose the block that is closest in size to the request</td>
</tr>
<tr>
<td><strong>First-fit</strong></td>
<td>- begins to scan memory from the beginning and chooses the first available block that is large enough</td>
</tr>
<tr>
<td><strong>Next-fit</strong></td>
<td>- begins to scan memory from the location of the last placement and chooses the next available block that is large enough</td>
</tr>
</tbody>
</table>
Memory Configuration Example

Figure 7.5  Example Memory Configuration before and after Allocation of 16-Mbyte Block
Buddy System

Comprised of fixed and dynamic partitioning schemes

Space available for allocation is treated as a single block

Memory blocks are available of size $2^K$ words, $L \leq K \leq U$, where

- $2^L = \text{smallest size block that is allocated}$
- $2^U = \text{largest size block that is allocated}; \text{generally} \ 2^U \text{is the size of the entire memory available for allocation}$
### Buddy System Example

<table>
<thead>
<tr>
<th>1 Mbyte block</th>
<th>1 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request 100 K</td>
<td>A = 128K</td>
</tr>
<tr>
<td>Request 240 K</td>
<td>A = 128K</td>
</tr>
<tr>
<td>Request 64 K</td>
<td>A = 128K</td>
</tr>
<tr>
<td>Request 256 K</td>
<td>A = 128K</td>
</tr>
<tr>
<td>Release B</td>
<td>A = 128K</td>
</tr>
<tr>
<td>Release A</td>
<td>128K</td>
</tr>
<tr>
<td>Request 75 K</td>
<td>E = 128K</td>
</tr>
<tr>
<td>Release C</td>
<td>E = 128K</td>
</tr>
<tr>
<td>Release E</td>
<td>512K</td>
</tr>
<tr>
<td>Release D</td>
<td>1M</td>
</tr>
</tbody>
</table>

**Figure 7.6 Example of Buddy System**
Figure 7.7 Tree Representation of Buddy System
Addresses

Logical
• reference to a memory location independent of the current assignment of data to memory

Relative
• address is expressed as a location relative to some known point

Physical or Absolute
• actual location in main memory
Relocation

Figure 7.8  Hardware Support for Relocation
Partition memory into equal fixed-size chunks that are relatively small.

Process is also divided into small fixed-size chunks of the same size.

**Pages**
- chunks of a process

**Frames**
- available chunks of memory
Assignment of Process to Free Frames

Figure 7.9 Assignment of Process Pages to Free Frames
Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address
Data Structures

Figure 7.10  Data Structures for the Example of Figure 7.9 at Time Epoch (f)
Logical Addresses

Relative address = 1502
0000010111011110

(a) Partitioning

Logical address = Page# = 1, Offset = 478
0000010111011110

(b) Paging
(page size = 1K)

Logical address = Segment# = 1, Offset = 752
0000010101111000

(c) Segmentation

Segment 0
750 bytes

Segment 1
1950 bytes

Figure 7.11  Logical Addresses
Logical-to-Physical Address Translation - Paging

(a) Paging
Segmentation

- A program can be subdivided into segments
  - may vary in length
  - there is a maximum length

- Addressing consists of two parts:
  - segment number
  - an offset

- Similar to dynamic partitioning

- Eliminates internal fragmentation
Logical-to-Physical Address Translation - Segmentation

![Diagram of logical-to-physical address translation using segmentation]

- 16-bit logical address
- 4-bit segment #
- 12-bit offset
- Process segment table
- 16-bit physical address
Security Issues

If a process has not declared a portion of its memory to be sharable, then no other process should have access to the contents of that portion of memory.

If a process declares that a portion of memory may be shared by other designated processes then the security service of the OS must ensure that only the designated processes have access.
Buffer Overflow Attacks

- Security threat related to memory management
- Also known as a buffer overrun
- Can occur when a process attempts to store data beyond the limits of a fixed-sized buffer
- One of the most prevalent and dangerous types of security attacks
Buffer Overflow Example

```c
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strcmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1%s, str2%s, valid%d
", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs
### Buffer Overflow Stack Values

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Before gets(str2)</th>
<th>After gets(str2)</th>
<th>Contains Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . .</td>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>bfffffff4</td>
<td>34fcffbf 4 . . .</td>
<td>34fcffbf 3 . . .</td>
<td>argv</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>01000000 . . . .</td>
<td>01000000 . . . .</td>
<td>argc</td>
</tr>
<tr>
<td>bfffffffec</td>
<td>c6bd0340 . . . @</td>
<td>c6bd0340 . . . @</td>
<td>return addr</td>
</tr>
<tr>
<td>bfffffff8</td>
<td>08fcffbf . . . .</td>
<td>08fcffbf . . . .</td>
<td>old base ptr</td>
</tr>
<tr>
<td>bfffffff4</td>
<td>00000000 . . . .</td>
<td>01000000 . . . .</td>
<td>valid</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>80640140 . d . @</td>
<td>00640140 . d . @</td>
<td></td>
</tr>
<tr>
<td>bfffffff8</td>
<td>54001540 . t . @</td>
<td>4e505554 N P U T</td>
<td>str1[4-7]</td>
</tr>
<tr>
<td>bfffffff8</td>
<td>53544152 S T A R</td>
<td>42414449 B A D I</td>
<td>str1[0-3]</td>
</tr>
<tr>
<td>bfffffff4</td>
<td>00850408 . . . .</td>
<td>4e505554 N P U T</td>
<td>str2[4-7]</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>30561540 O V . @</td>
<td>42414449 B A D I</td>
<td>str2[0-3]</td>
</tr>
</tbody>
</table>

### Memory Address Column
- bfffffff4: Initial memory address
- bfffffff0: Argument pointer
- bfffffffec: Return address
- bfffffff8: Old base pointer
- bfffffff4: Valid pointer
- bfffffff0: Data buffer
- bfffffff8: String buffer 1
- bfffffff4: String buffer 2
- bfffffff0: String buffer 2
Defending Against Buffer Overflows

- Prevention
- Detecting and aborting
- Countermeasure categories:

**Compile-time Defenses**
- aim to harden programs to resist attacks in new programs

**Run-time Defenses**
- aim to detect and abort attacks in existing programs
Memory Management

- one of the most important and complex tasks of an operating system
- needs to be treated as a resource to be allocated to and shared among a number of active processes
- desirable to maintain as many processes in main memory as possible
- desirable to free programmers from size restriction in program development
- basic tools are paging and segmentation (possible to combine)
  - paging – small fixed-sized pages
  - segmentation – pieces of varying size