CS3701 - Operating Systems

Chapter 7 - Memory Management

Department of Computer Science

College of Computer Engineering and Science

Prince Sattam bin Abdulaziz University

AY - 2025/2026

Prepared by Dr. Mahdi Khemakhem, Reviewed by Dr. Essra Aldessouky

Updated on January 22, 2026

Outline

1. Background
1.1 Overview
1.2 Memory Protection
1.3 Address Binding
1.4 Logical vs. Physical Address Space
1.5 Dynamic Loading and Linking
1.6 Key Takeaways

2. Contiguous Memory Allocation
2.1 Overview
2.2 Variable Partition
2.3 Fragmentation
2.4 Key Takeaways

3. Paging
3.1 Overview
3.2 Address Translation
3.3 Paging Example
3.4 Internal Fragmentation
3.5 Key Takeaways

4. Swapping Overview

Outline (Contd.)

4.1 Swapping with Paging
4.2 Valid – Invalid Bit Scheme
4.3 Key Takeaways

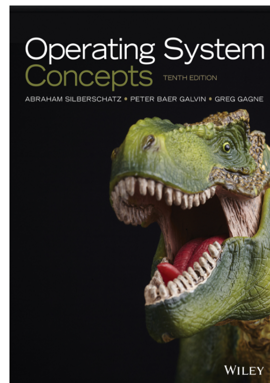
5. Exercise

Textbook Reference

Required Reading

Refer to Chapter 9 of the textbook:

- **Title:** *Operating System Concepts*
- **Authors:** Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
- **Publisher:** John Wiley & Sons
- **Edition:** 10th Edition (2018)



Outline

1. Background
1.1 Overview
1.2 Memory Protection
1.3 Address Binding
1.4 Logical vs. Physical Address Space
1.5 Dynamic Loading and Linking
1.6 Key Takeaways

2. Contiguous Memory Allocation

3. Paging

4. Swapping Overview

5. Exercise

Background

Program Execution

Program must be brought (from disk) into memory and placed within a process to be executed

- **Main memory and registers** are only storage CPU can access directly
- **Memory unit only sees a stream of:**
  - **addresses + read requests**, or
  - **addresses + data and write requests**
- **Register access** is done in one CPU clock cycle (or less)
- **Main memory** can take many cycles, causing a **stall**[a]
- **Cache** sits between main memory and CPU registers to reduce stalls by keeping frequently accessed data closer to the CPU
- *Protection of memory required* to ensure correct operation

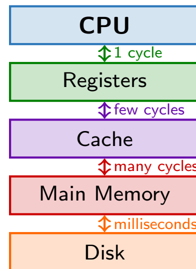


Figure 1: Memory Hierarchy

[a] A stall occurs when the CPU has to wait for data to be fetched from memory before it can proceed with execution.

Protection

- **To separate memory spaces of different processes, the OS must:**
  - **Determine the range of legal addresses** that the process may access
  - **Ensure that every process cannot access memory outside its allocated range**
- **We can provide this protection by using a pair of base and limit registers define the logical address space of a process**

Base and Limit Registers

- **Base register**: Starting address
- **Limit register**: Size of address space
- **Legal address range:** $[\text{base}, \text{base} + \text{limit})$

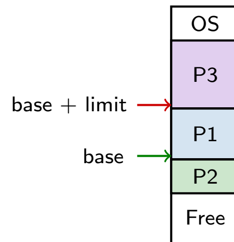


Figure 2: Base and Limit

Hardware Address Protection

- **CPU must check every memory access** generated for a process in user mode to **be sure** it is **between base and limit** for that process
- The instructions to **load the base and limit registers** are **privileged**
- **Only the OS can load these registers**

Protection Mechanism

$$\textbf{base} \leq \textbf{address} < \textbf{base} + \textbf{limit}$$

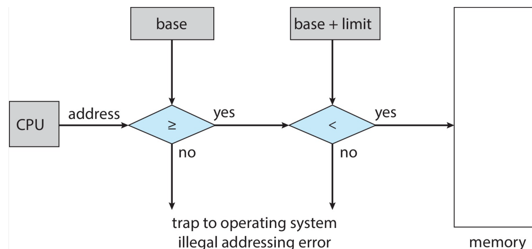


Figure 3: Hardware Address Protection with Base and Limit Registers

Address Binding

Address Binding

Mapping of one address space to another at different stages of a program's life cycle

Address Binding can occur at three different main stages:

- **Compile time:** Address binding from **symbolic** addresses (e.g., variable names in source code) to **relocatable** addresses (e.g., 14 bytes)
- **Load time:** Binding from **relocatable** addresses to **absolute** addresses (e.g., 74014)
- **Execution time:** Binding from *logical* addresses to *physical* addresses (e.g., 34014)

MMU Definition

The Memory-Management Unit (MMU) is a hardware device that at run time maps virtual to physical address

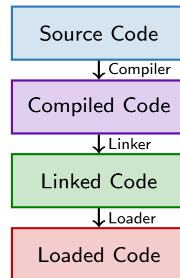


Figure 4: Address Binding Process

Logical vs. Physical Address Space

Central Concept

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Logical address space** – set of all logical addresses generated by the CPU for a program ⇒ **can be deduced from the process size**
- **Physical address** – address seen by the memory unit
- **Physical address space** – set of all physical addresses corresponding to the memory unit ⇒ **can be deduced from the memory size**

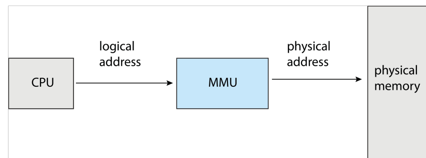


Figure 5: Address Spaces

Dynamic Loading

Dynamic Loading Concept

The entire program does NOT need to be in memory to execute. Only the routine (function) that is called needs to be in memory.

- **Routine is not loaded until it is called**
- **Better memory-space utilization**; unused routine is never loaded
- **All routines kept on disk in relocatable load format**
- Useful when large amounts of code that are infrequently used
- **No special support from the operating system is required**
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

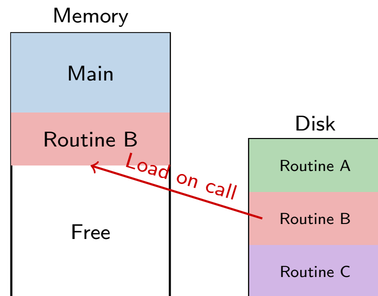


Figure 6: Dynamic Loading

Dynamic Linking

Dynamic Linking Concept

Linking postponed until execution time to allow a program to be shared in memory among several processes

- **Static linking** – linking done at *load time* ⇒ all code in memory when process starts. Larger executable, faster execution, duplicated code
- **Dynamic linking** – linking postponed until *execution time* ⇒ routine called is **linked as needed**. Smaller executable, shared libraries, version flexibility
- Operating system must provide **dynamic linking support**
- Operating system must **keep track of all loaded modules** and **resolve references** when a module is loaded
- **Shared libraries** – allow several processes to share a single copy of a library in memory

Key Takeaways

Key Takeaways

- **Memory management** is essential for efficient and secure operation of an operating system
- **Base and limit registers** provide a simple yet effective way to protect memory spaces of different processes
- **Address binding** is **the process of mapping logical to physical addresses** and can occur at different stages of a program's life cycle
- **Memory-Management Unit (MMU)** is a hardware component that facilitates the mapping of virtual to physical addresses
- **Logical address space** allows for abstraction and flexibility in memory management
- **Physical address space** represents the actual memory locations in the hardware
- **Dynamic loading and linking** optimize memory usage and enable code sharing among processes

Outline

1. Background

2. Contiguous Memory Allocation
2.1 Overview
2.2 Variable Partition
2.3 Fragmentation
2.4 Key Takeaways

3. Paging

4. Swapping Overview

5. Exercise

Contiguous Allocation

Contiguous Allocation

Contiguous allocation is one **early method** where each process is contained in a single contiguous section of memory

- Main memory usually divided into two partitions:
  - **Resident operating system**, usually held in low memory with interrupt vector
  - **User processes** then held in high memory
- **Single contiguous allocation** – each process is contained in a single contiguous section of memory
- **Memory management** is done by keeping track of:
  - **Allocated memory blocks**
  - **Free memory blocks**

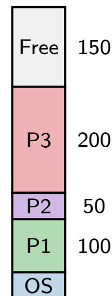


Figure 7: Contiguous Allocation

Variable Partition

Multiple-Partition Allocation

Variable-partition allocation – **memory is divided** into a number of **partitions** of **various sizes**; each partition may contain exactly one process

- **Hole** – block of available memory; holes of various size are scattered throughout memory
- **When a process arrives, it is allocated memory from a hole large enough to accommodate it**
- **If the hole is larger than needed**, it is **split** into two partitions:
  - **One allocated to the process**
  - **One remaining as a smaller hole**
- **When a process terminates, its memory is released to form a larger hole**

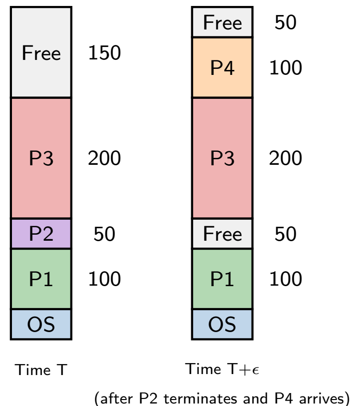


Figure 8: Variable Partitions

Dynamic Storage-Allocation Problem

The Problem

How to satisfy a request of size *n* from a list of free holes?

1. **First-fit**:
   - Allocate the *first hole* that is big enough
   - Fast search
2. **Best-fit**:
   - Allocate the *smallest hole* that is big enough
   - Must search entire list, unless ordered by size
   - Produces the smallest leftover hole
3. **Worst-fit**:
   - Allocate the *largest hole*
   - Must also search entire list
   - Produces the largest leftover hole

⇒**First-fit** and **best-fit** better than **worst-fit** in terms of **speed** and **storage utilization**, respectively.

Example

Ordered free holes in memory: 10KB, 20KB, 4KB, 27KB, 18KB, 9KB

Request: 17KB

- **First-fit:** 20KB
- **Best-fit:** 18KB
- **Worst-fit:** 27KB

Fragmentation

Fragmentation Definition

Fragmentation is a state of memory **wastage** that occurs when **free memory** is *broken into small pieces* and is **unable to satisfy a request** for memory allocation.

Two types of fragmentation:

- **External Fragmentation**
- **Internal Fragmentation**

- **External Fragmentation** – total memory space exists to satisfy a request, but it is *not contiguous*, *e.g., 100KB request cannot be satisfied with holes of 50KB + 50KB*
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but *not being used*. *e.g., 48KB request allocated in a 50KB hole, 2KB internal fragmentation*

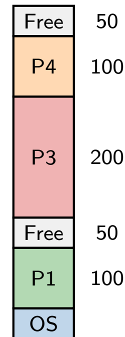


Figure 9: Fragmentation Types

Fragmentation Solutions

Two possible solutions to the external-fragmentation problem are:

1. **Compaction**
   - Shuffle memory contents to place all free memory together in one large block
   - Compaction is possible only if relocation is *dynamic*, and is done at execution time
   - Can be expensive in terms of time
2. **Paging**
   - Another possible solution to the external-fragmentation problem is to permit the *logical address space of processes to be noncontiguous*
   - Thus allowing a process to be allocated physical memory wherever such memory is available
   - **This is the strategy used in paging, the most common memory-management technique for computer systems (see next section)**

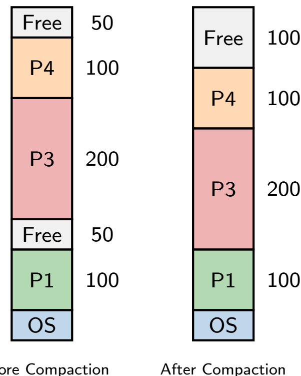


Figure 10: Compaction Example

Key Takeaways

Key Takeaways

- **Contiguous memory allocation** is a simple memory management technique where each process is allocated a single **contiguous** block of memory
- **Variable partition allocation** allows memory to be divided into partitions of varying sizes, accommodating processes of different memory requirements
- **Dynamic storage-allocation problem** involves finding suitable free memory blocks (holes) to satisfy memory requests, with strategies like **first-fit**, **best-fit**, and **worst-fit**
- **Fragmentation** is a memory wastage issue that can be classified into **external** and **internal** fragmentation, affecting memory utilization
- **Compaction** is a technique to address **external fragmentation** by rearranging memory contents to create larger contiguous free memory blocks
- **Paging** is a memory management technique that allows **noncontiguous** allocation of memory, effectively addressing external fragmentation but still subject to internal fragmentation! (to be discussed in the next section)

Outline

1. Background

2. Contiguous Memory Allocation

3. Paging
3.1 Overview
3.2 Address Translation
3.3 Paging Example
3.4 Internal Fragmentation
3.5 Key Takeaways

4. Swapping Overview

5. Exercise

Paging

Paging Concept

Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

- **Avoids external fragmentation**
- **Avoids problem of varying sized memory chunks**
- **Divide physical memory into fixed-sized blocks called frames**
  $\Rightarrow$ *Size is power of 2, between* $2^9 = 512$ *bytes and* $2^{24} = 16$ *Mbytes*
- **Divide logical memory into blocks of same size called pages**
- **To run a program of size N pages, need to find N free frames and load program**
- **Keep track of all free frames using a free-frame list**
- **Set up a page table to translate logical to physical addresses**
- **Still have Internal fragmentation!** – *last page may not be completely full*

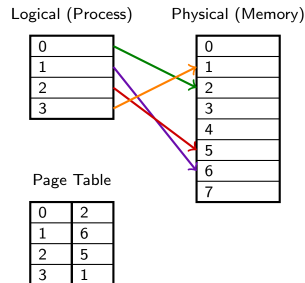


Figure 11: Paging

Address Translation Scheme

Address generated by CPU is divided into two parts:

- **Page number (p)** – used as an index into a **page table** which contains *base address* of each page in physical memory
- **Page offset (d)** – combined with **base address** to define the *logical address* that is sent to the **MMU** for **physical memory access**
- **For given logical address space 2^m and page size 2^n then:**
  - **The high-order $m - n$ bits of a logical address designate the page number**
  - **The n low-order bits designate the page offset**

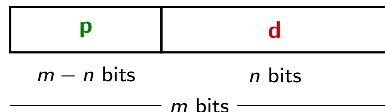


Figure 12: Logical Address Format

Example

- $m = 16$ (64KB address space)
- $n = 10$ (1KB page size)
- Page number: 6 bits
- Offset: 10 bits
- Number of pages: $2^6 = 64$

⇒ **Same format used for physical address: [$k - n$ bits for frame number (f) + n bits for offset (d)], where the physical address space (memory size) is 2^k and the frame size is 2^n**

Paging Hardware

Figure 13: Paging Hardware: Address Translation Mechanism

Free Frame List

free-frame list
14
13
18
20
15

page 0
page 1
page 2
page 3
new process

13
14
15
16
17
18
19
20
21

(a)Before Allocation

free-frame list
15

page 0
page 1
page 2
page 3
new process

0	14
1	13
2	18
3	20

new-process page table

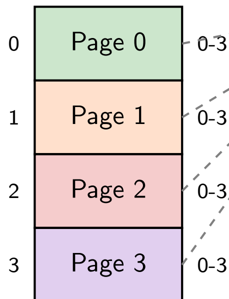
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21

(b)After Allocation

Figure 14: Free Frame List Example

Paging Example

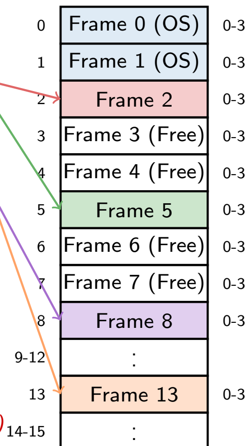
Logical Address Space
(Process: 16 bytes)



Page Table

Page	Frame
0	5
1	13
2	2
3	8

Physical Memory
(64 bytes)



Binary Addresses

Log. Base	Phy. Base
00	0101
01	1101
10	0010
11	1000

Configuration:
Page/Frame Size: 4 = 2^2 bytes
Process: 16 = 2^4 bytes (4 pages)
Logical Address: 4 bits (2 for page, 2 for offset)
Physical Memory: 64 = 2^6 bytes (16 frames)
Physical Address: 6 bits (4 for frame, 2 for offset)

Calculating Internal Fragmentation

Calculation Example

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation = 2,048 - 1,086 = **962 bytes**

- **Worst case fragmentation** = 1 frame – 1 byte
- **On average fragmentation** = $\frac{1}{2}$ frame size
- So **small frame sizes desirable**?
- But each page table entry takes memory to track
- **Page sizes growing over time**
  - Solaris supports two page sizes – 8 KB and 4 MB

Internal Fragmentation Trade-off

Trade-off: Small pages ⇒ Less internal fragmentation but larger page tables

Key Takeaways

Key Takeaways

- **Paging** is a memory management technique that allows **noncontiguous** allocation of memory, effectively addressing external fragmentation
- **Logical address space** is divided into **pages**, and **physical memory** is divided into **frames** of the same size
- **Address translation** is performed using a **page table** that maps **page numbers** to **frame numbers**
- **Internal fragmentation** can occur in paging systems, with worst-case fragmentation being one frame minus one byte, and average fragmentation being half the frame size
- **Trade-off** exists between page size and memory efficiency, as smaller pages reduce internal fragmentation but increase page table size
- **Paging hardware** includes components like the **MMU** and **free frame list** to facilitate efficient memory management

Outline

1. Background

2. Contiguous Memory Allocation

3. Paging

4. Swapping Overview
4.1 Swapping with Paging
4.2 Valid – Invalid Bit Scheme
4.3 Key Takeaways

5. Exercise

Swapping with Paging

- **Swapping** is *moving processes* between **main memory** and **backing store** (disk) to *increase the degree of multiprogramming*
- **With paging**, *individual pages* can be swapped rather than entire processes
- **When a page is needed** that is not in memory, a *page fault* occurs (**see next chapter**)
- **Operating system** must *bring the required page into memory* from backing store (e.g., disk)
- **If no free frame is available**, the OS must *select a frame to evict* (using a page replacement algorithm) and *swap out* the page in that frame to backing store
- **The OS then updates the page table** to reflect the changes and *resumes the execution* of the process

Schematic View of Swapping in Paging

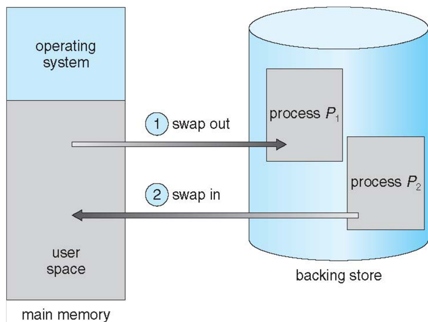


Figure 15: Schematic View of Swapping

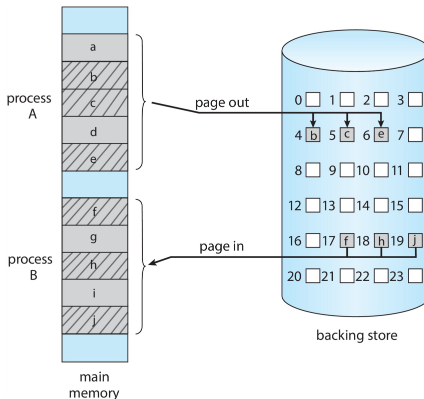


Figure 16: Swapping in Paging Example

Valid – Invalid Bit Scheme

- **Each page table entry** has a *valid – invalid bit* to *indicate whether the page is in memory*
- **Valid bit (1)** indicates that the *page is in memory* and can be accessed
- **Invalid bit (0)** indicates that the *page is not in memory* (may be on disk or not allocated)
- **When a page is referenced**, the OS checks the valid – invalid bit
  - If *valid*, the OS proceeds with address translation and memory access
  - If *invalid*, a *page fault* occurs, and the OS must *bring the page into memory* from backing store

Valid – Invalid Bit Example

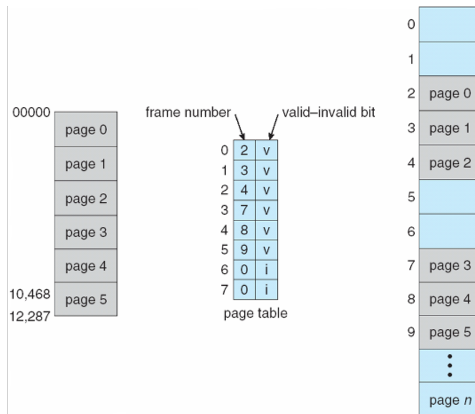


Figure 17: Valid – Invalid Bit Example

Key Takeaways

Key Takeaways

- **Swapping** is a memory management technique that involves moving processes between **main memory** and **backing store** (disk) to increase the degree of multiprogramming
- **With paging**, individual pages can be swapped rather than entire processes, allowing for more efficient memory usage
- **Page faults** occur when a process tries to access a page that is not currently in memory, prompting the OS to bring the required page from backing store (**see next chapter**)
- **Page replacement algorithms** are used to select which page to evict from memory when no free frames are available (**see next chapter**)
- **Valid – invalid bit scheme** is employed in page tables to indicate whether a page is currently in memory or not, facilitating efficient handling of page faults

Outline

1. Background
2. Contiguous Memory Allocation
3. Paging
4. Swapping Overview
5. Exercise

Exercise

Paging System Analysis

Consider a paging system with the following parameters:

- Logical address space = $\mathbf{2^x}$ bytes
- Physical address space = $\mathbf{2^y}$ bytes
- Page size = Frame size = $\mathbf{2^z}$ bytes

Questions:

a) How many bits are needed for a page number?

b) How many bits are needed for the page/frame offset?

c) How many bits are needed for a frame base address?

d) List all byte addresses in page 4 (in binary) when $x = 5$, $z = 2$.

e) List all byte addresses in frame 3 (in binary) when $y = 6$, $z = 1$.

f) Calculate the internal fragmentation for a 405-byte process when $x = 3$, $y = 6$, $z = 2$.

Exercise Solution - Part 1

Solution to Questions a, b, c

a) Page number bits:

- Number of pages $= \frac{2^x}{2^z} = 2^{x-z} \Rightarrow (x-z)$ **bits**

b) Page/Frame offset bits:

- Offset identifies position within a page/frame $\Rightarrow z$ **bits**

c) Frame base address bits:

- Number of frames $= \frac{2^y}{2^z} = 2^{y-z}$ (frame number needs $y-z$ bits)
- Frame base address = Frame number × Frame size = actual physical address
- Frame base address bits = y **bits** (not $y-z$)

Exercise Solution - Part 2

Solution to Questions d, e

d) Page 4 byte addresses ($x = 5$, $z = 2$):

- Page size $= 2^2 = 4$ bytes, Logical address space $= 2^5 = 32$ bytes (5 bits)
- Page 4 base address $= 100_2 \Rightarrow$ Contains 4 addresses (1 for each byte)
- **Binary (5-bit):** $\mathbf{10000}_2$, $\mathbf{10001}_2$, $\mathbf{10010}_2$, $\mathbf{10011}_2$

e) Frame 3 byte addresses ($y = 6$, $z = 1$):

- Frame size $= 2^1 = 2$ bytes, Physical address space $= 2^6 = 64$ bytes (6 bits)
- Frame 3 base address $= 00011_2 \Rightarrow$ Contains 2 addresses (1 for each byte)
- **Binary (6-bit):** $\mathbf{000110}_2$, $\mathbf{000111}_2$

Exercise Solution - Part 3

Solution to Question f

f) Internal fragmentation (405-byte process, $x = 3$, $y = 6$, $z = 2$):

- Page size $= 2^2 = 4$ bytes
- Process size $= 405$ bytes
- Pages needed $= \lceil \frac{405}{4} \rceil = \lceil 101.25 \rceil = 102$ pages
- Total allocated space $= 102 \times 4 = 408$ bytes
- Internal fragmentation $= 408 - 405 =$ **3 bytes**

Answer: Internal fragmentation = **3 bytes**

End of Chapter 7