Memory Management

Ajeeta Asthana

April 2025

1 Introduction

Memory Management in an operating system is the process of coordinating and optimizing the use of computer memory to ensure efficient and secure execution of programs and processes. It involves allocating memory to running programs, keepign track of which memory locations are used by which processes, and freeing up memory when its no longer needed.

1.1 Keys functions of memory management:

1.1.1 Allocation:

The operating system determines which processes get memory and how much they get. This can involve dividing memory into blocks (paging) or logically organized sections (segmentation)

1.2 Protection:

Ensures that processes cannot access memory that belongs to other processes, preventing data corruption or system crashes.

1.3 Sharing:

Allows multiple processes to share memory resources, like libraries or data, for efficiency.

1.4 Swapping:

Moves processes between main memory (RAM) and secondary storage (disk) to manage memory constraints and allow more processes to run concurrently.

1.5 Virtual Memory:

Creates the illusion of a large, continuous memory space even if physical memory is limited, using techniques like paging and swapping.

1.5.1 Methods and Techniques:

Paging: Divides memory into fixed-size blocks called pages and maps them to frames in RAM, allowing processes to run even with fragmented memory.

Segmentation: Divides memory into logical sections like code, data, and stack which can be more efficient for handling different types of data.

Swapping: Moves processes between RAM and disk to manage memory and allow more processes to run.

Virtual Memory: Extends the available memory by using disk space as an extension of RAM.

Compaction: Rearranges memory to eliminate small free spaces, making it easier to allocate large blocks.

1.5.2 Importance:

Efficient Resource Utilization: Ensures that memory is used effectively and efficiently, preventing waste and improving system performance.

Multi-programming and Multi-tasking: Enables multiple processes to run concurrently by managing memory allocation and deallocation.

Program Protection Prevents processes from interfering with each other's memory space, ensuring syatem stability.

Large Program Support: Virtual memory allows programs to be larger than available physical memory.

```
#include <iostream>
#include <memory>
#include <vector>
#include <string>

class Resource {
private:
    std::string name;
    int* data;
    size_t size;

public:
    //Constructor
    Resource(std::string n, size_t s) : name(n), size(s) {
        std::cout << "Constructor: Allocating " << name << " with " << size << " elements\n'
        data = new int[size];
        for(size_t i = 0; i < size; i++) {
            data[i] = 0;
        }
    }

    // Destructor
    ~Resource() {
```

```
    std::cout << "Desctructor: Releasing " << name << "\n";
    delete[] data;
}

// Copy Constructor
Resource(const Resource& other) : name(other.name + "_copy"), size(other.size) {
    std::cout << "Copy Constructor: Creating " << name << "\n";
    data = new int[size];
    for(size_t i=0; i<size; i++) {
        data[i] = other.data[i];
    }
}

// Move constructor
Resource& operator=(const Resource& other) {
    std::cout << "Copy assignment for " << name << "\n";
    if (this != &other) {
        delete[] data;
        name = other.name + "_copy";
        size = other.size;
        data = new int[size];
        for(size_t i=0; i<size; ++i) {
            data[i] = other.data[i];
        }
    }
    return *this;
}

// Move assignment
Resource& operator=(Resource&& other) noexcept {
    std::cout << "Move assignment for " << name << "\n";
    if(this != &other) {
        delete[] data;
        name = std::move(other.name);
        size = other.size;
        data = other.data;
        other.data = nullptr;
        other.size = 0;
    }
    return *this;
}

void setValue(size_t index, int value) {
    if (index < size) {
        data[index] = value;
    }
```

```
    }

    int getValue(size_t index) const {
        return (index < size) ? data[index] : -1;
    }

    std::string getName() const {
        return name;
    }
};

// Function to demonstrate different memory management technique
void memoryManagementDemo() {
    std::cout << "\n === Manual Memory Management === \n";
    {
        //Manual Allocation with new
        Resource* res1 = new Resource("Manual Resource", 5);
        res1->setValue(0, 100);

        // Use the resource
        std::cout << "Value at index 0: " << res1->getValue(0) << "\n";

        //Manual deallocation with delete
        delete res1;
    }

    std::cout << "\n ==== RAII with Stack Allocation ==== \n";
    {
        // Stack allocation ( RAII - destructor called automatically)
        Resource res2("StackResource", 3);
        res2.setValue(1, 200);
        std::cout << "Value at index 1: " << res2.getValue(1) << "\n";
    }

    std::cout << "\n ==== Smart Pointers === \n";
    {
        // unique_ptr - exclusive ownership
        std::unique_ptr<Resource> res3 = std::make_unique<Resource>("UniqueResource", 4);
        res3->setValue(2, 300);
        std::cout << "Value at index 2: " << res3->getValue(2) << "\n";

        // shared_ptr - shared ownership
        std::shared_ptr<Resource> res4 = std::make_shared<Resource>("SharedResource", 6);
        {
            std::shared_ptr<Resource> res5 = res4;
            res5->setValue(3, 400);
```

```
            std::cout << "Reference count: " << res4.use_count() << "\n";
        }
        // res5 is destroyed, but res4 still exists
        std::cout << "Reference count: " << res4.use_count() << "\n";
        std::cout << "Value at index 3: " << res4->getValue(3) << "\n";

        // weak_ptr - non-owning reference
        std::weak_ptr<Resource> res6  = res4;
        if(auto locked = res6.lock()) {
            std::cout << "Weak ptr locked: " << locked->getName() << "\n";
        }
    }

    std::cout << "\n === Move Semantincs === \n";
    {
        Resource res7("Original Resource", 2);
        res7.setValue(0, 500);

        // Move resource (efficient transfer of ownsership)
        Resource res8 = std::move(res7);
        std::cout << "Value after move: " << res8.getValue(0) << "\n";

        // Original resource is now in a valid but unspecified state
        // Typically, its now empty or null
    }
}
```