NUST
NATIONAL UNIVERSITY
OF SCIENCES & TECHNOLOGY

Parallel and Distributed Processing (CS - 435)

Assignment 1

Dynamic Memory Allocation

Name	*M. Saad Farrukh*
Reg. No	*373188*
Section	*BEE-13 D*
Instructor	*Dr. Attique Dawood*

1 Problem Statement

1) Create at least one instance of processes named **A, B** and **C**.

2) A, B and C will allocate **20%**, **10%** and **5%** of (maximum) RAM, respectively.

3) Make a memory map showing where A, B and C reside in memory.

4) You might not have enough memory to allocate a large chunk of RAM. You can come up with something like one instance of B and two instances of C [B,C1,C2]. Or maybe [C1,C2,C3]. Maybe even [A1,A2,A3] if you have a lot of free RAM.

2 Software Used

- ***Operating System:*** Windows 11
- ***Code Editor:*** Visual Studio Code
- ***Compiler:*** MinGW-w64

3 Additional Note

I am using **7 GB** of RAM for this assignment.

4 Code

1. System Setup & Definitions

Purpose: Handles platform compatibility and defines memory constraints.

Key Points:

- Uses **#ifdef _WIN32** to maintain the portability across different operating system platforms.
- Defines **GB** for easier memory calculations (1 GB = 1024 × 1024 × 1024 bytes).
- Sets **MAX_RAM** to simulate a **7 GB memory limit** for the system.

```
#ifdef _WIN32
#include <windows.h>
#include <process.h>
#else
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#endif

#include <iostream>
#include <cstdlib>
#include <vector>

#define GB (1024L * 1024 * 1024)  // 1 GB in bytes
#define MAX_RAM ((size_t)(7L * GB)) // Simulate 4GB max RAM
```

---

2. Memory Allocation & Tracking

Purpose: Allocates memory dynamically, writes data to enforce allocation, and maintains a memory map.

Key Points:

- Uses **malloc()** to allocate a specific memory size.
- Writes data to memory in 4KB steps (**mem[i] = rand() % 256;**) to ensure actual RAM allocation.

- Sleeps (**Sleep(2000)** or **sleep(5)**) to keep allocated memory visible before freeing it.
- Stores allocations in **std::vector<MemorySegment>** for tracking.
- Frees allocated memory after the delay.

```
// Struct to store memory segment info
struct MemorySegment {
    const char* processName;
    void* startAddr;
    size_t size;
};

std::vector<MemorySegment> memoryMap;

void allocateMemory(const char* processName, size_t size) {
    std::cout << processName << " is attempting to allocate "
              << (size / (1024 * 1024)) << " MB...\n";

    // Allocate memory
    char* mem = (char*)malloc(size);
    if (mem) {
        std::cout << processName << " successfully allocated "
                  << (size / (1024 * 1024)) << " MB.\n";
        memoryMap.push_back({processName, mem, size});

        //  Memory Utilization
        for (size_t i = 0; i < size; i += 4096) {
            mem[i] = rand() % 256;             // Write random data
        }
    } else {
        std::cerr << processName << " failed to allocate memory!\n";
    }

    // Simulate processing delay
#ifdef _WIN32
    Sleep(2000);      // Sleep for visibility
#else
    sleep(5);
#endif

    if (mem) {        // Free allocated memory
        free(mem);
        std::cout << processName << " has freed its memory.\n";
    }
}
```

3. Memory Map Printing Function

Purpose: Displays memory allocations before program exits.

Key Points:

- Prints a table containing process name, allocated memory address, and memory size.
- Helps visualize memory usage before the program terminates.

```
// Print the memory map
void printMemoryMap() {

    std::cout << "\nMemory Map:\n";
    std::cout << "--------------------------------------\n";
    std::cout << "| Process | Start Address | Size (MB) |\n";
    std::cout << "--------------------------------------\n";

    for (const auto& segment : memoryMap) {
        std::cout << "| " << segment.processName << " | " <<
segment.startAddr << " | "
                  << (segment.size / (1024 * 1024)) << " MB |\n";
}
    std::cout << "--------------------------------------\n";
}
```

---

4. Process (Thread) Creation & Execution

Purpose: Runs memory allocation in parallel using threads (Windows) or processes (Linux/macOS).

Key Points:

- Defines **aSize**, **bSize**, and **cSize** as **20%**, **10%**, and **5%** of **MAX_RAM**, respectively.
- Uses **_beginthread()** on Windows for multithreading.
- Uses **fork()** on Unix-based systems to create child processes.
- Waits for all processes/threads to finish before proceeding.

```
int main() {
    // Fix: Explicitly use size_t to avoid truncation
    size_t aSize = (size_t)(0.2 * MAX_RAM);  // 20% of RAM
    size_t bSize = (size_t)(0.1 * MAX_RAM);  // 10% of RAM
    size_t cSize = (size_t)(0.05 * MAX_RAM); // 5% of RAM

    std::cout << "Starting memory allocation for processes A, B, and
C...\n";

#ifdef _WIN32
    _beginthread([](void* p) { allocateMemory("Process A", *(size_t*)p); },
0, &aSize);
    _beginthread([](void* p) { allocateMemory("Process B", *(size_t*)p); },
0, &bSize);
    _beginthread([](void* p) { allocateMemory("Process C", *(size_t*)p); },
0, &cSize);

    Sleep(5000);  // Allow threads to complete execution
#else
    if (fork() == 0) { allocateMemory("Process A", aSize); exit(0); }
    if (fork() == 0) { allocateMemory("Process B", bSize); exit(0); }
    if (fork() == 0) { allocateMemory("Process C", cSize); exit(0); }

    wait(NULL);
    wait(NULL);
    wait(NULL);
#endif

    printMemoryMap();

    std::cout << "All processes have completed execution.\n";
    return 0;
}
```

---

5 Calculations

Given that, in my case ***MAX_RAM*** = 7 GB, we calculate:

- aSize = 0.2 * 7 GB = 1.4 GB or approximately 1433 MB
- bSize = 0.1 * 7 GB = 0.7 GB or approximately 716 MB
- cSize = 0.05 * 7 GB = 0.35 GB or approximately 358 MB

6 Obtained Results

The results obtained from multi-threaded programs aren't deterministic, so 2 of the obtained results are mentioned below.

Result 01

Figure 1 Task Manager Report for Result 1

```
PS C:\Users\AZAN LAPTOP STORE\Downloads\SEMESTER 08\Parallel and Distributed Computing\Assignment> ./assign01
.exe
Starting memory allocation for processes A, B, and C...
Process A is attempting to allocate 1433 MB...
Process A successfully allocated 1433 MB.
Process C is attempting to allocate 358 MB...
Process B is attempting to allocate 716 MB...
Process C successfully allocated 358 MB.
Process B successfully allocated 716 MB.
Process A has freed its memory.
Process C has freed its memory.
Process B has freed its memory.
```

Figure 2 Code Output for Result 1

```
Memory Map:
--------------------------------------
| Process | Start Address | Size (MB) |
--------------------------------------
| Process A | 0x160d020   |1433 MB |
| Process C | 0x2e213020  |358 MB  |
| Process B | 0x37128020  |716 MB  |
--------------------------------------
```

Figure 3 Unorganized Memory Map for Result 1

Result 02

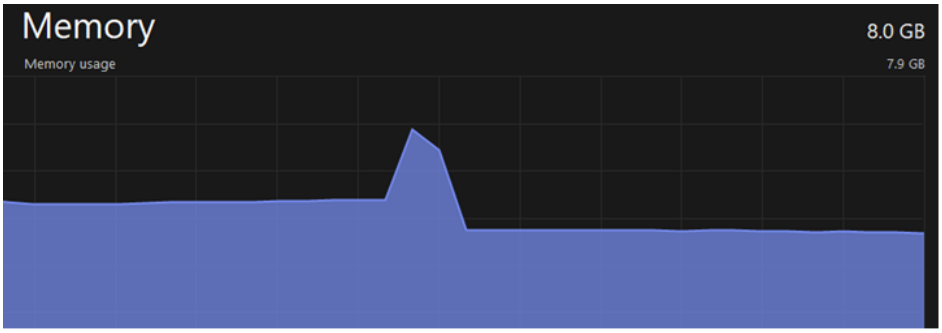

Figure 4 Task Manager Report for Result 2

```
PS C:\Users\AZAN LAPTOP STORE\Downloads\SEMESTER 08\Parallel and Distributed Computing\Assignment> ./assign01
.exe
Starting memory allocation for processes A, B, and C...
Process A is attempting to allocate 1433 MB...
Process C is attempting to allocate 358 MB...
Process B is attempting to allocate 716 MB...
Process A successfully allocated 1433 MB.
Process C successfully allocated 358 MB.
Process B successfully allocated 716 MB.
Process A has freed its memory.
Process B has freed its memory.
Process C has freed its memory.
```

Figure 5 Code Output for Result 2

```
Memory Map:
--------------------------------------
| Process | Start Address | Size (MB) |
--------------------------------------
| Process A | 0x167d020  |1433 MB |
| Process C | 0xa58c020  |358 MB |
| Process B | 0x20b9a020 |716 MB |
--------------------------------------
```

Figure 6 Unorganized Memory Map for Result 2

Additional

If multi-threading isn't used and program is designed to run sequentially, then the memory allocation and deallocation pattern can be observed as follows.

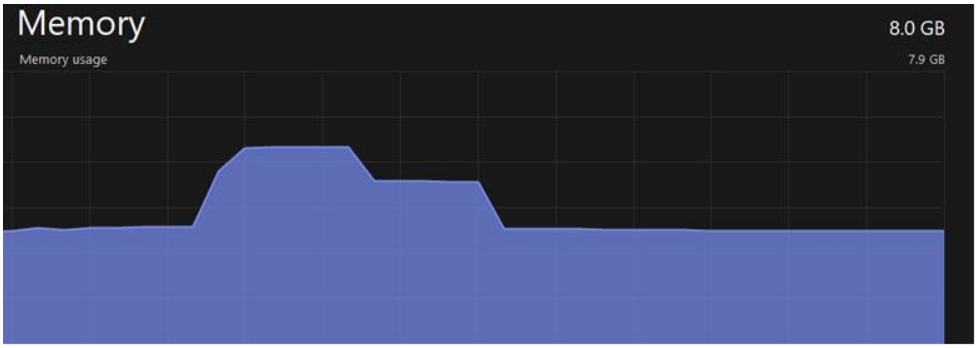

Figure 7 Task Manager Report for Sequential Processing

7 Memory Map

Higher Memory Address	
Stack Segment local_var = 10;	Local variables
Heap Segment Process C: ~ 358 MB (5%) Process B: ~ 716 MB (10%) Process A: ~ 1433 MB (20%)	Dynamically allocated memory, grows upward
Uninitialized (BSS) int uninit_var;	Global uninitialized variables
Initialized **Data** Segment int global_var = 42;	Global/static initialized variables
Code / Text Segment) Function memory_map();	Program instructions Text segment with executable code
Lower Memory Address	

8 Conclusion

- Parallel threads/processes allocate 20%, 10%, and 5% of simulated **7GB RAM**.
- **Memory Utilization** instead of Memory Allocation for better observations.
- A **memory map** logs allocation details.
- OS optimizations affect observed RAM usage.
- Need to optimize **sleep time** smartly.