Operating System
DA2
Deeptanshu Bhattacharya
22BLC1244

Multi Threaded Resource Allocation for Virtual Machines using Game Theory

1. Introduction

This document explains the implementation of a **multithreaded resource allocation system forVirtual Machines (VMs)** using **Game Theory (Nash Equilibrium)**. The goal is to allocate CPU,RAM, and GPUs fairly while ensuring system efficiency.

2.Code:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

// Mutex for synchronization
mutex resourceLock;

// VM Structure: Holds CPU, RAM, GPU requests
struct VM {
    int id;
    double cpuRequest;
    double ramRequest;
    double gpuRequest;
};

// Function to allocate resources using Nash Equilibrium principles
void allocateResources(vector<VM> &vms, double &cpuAvailable, double
&ramAvailable, double &gpuAvailable) {
    bool equilibriumReached = false;
    while (!equilibriumReached) {
        lock_guard<mutex> lock(resourceLock); // Ensure only one thread modifies resources

        double totalCPURequest = 0, totalRAMRequest = 0, totalGPURequest = 0;

        // Calculate total resource demand
        for (const auto &vm : vms) {
            totalCPURequest += vm.cpuRequest;
            totalRAMRequest += vm.ramRequest;
            totalGPURequest += vm.gpuRequest;
        }

        // If demand exceeds supply, proportionally reduce requests
        double cpuScale = (totalCPURequest > cpuAvailable) ? (cpuAvailable / totalCPURequest) : 1.0;
        double ramScale = (totalRAMRequest > ramAvailable) ? (ramAvailable / totalRAMRequest) : 1.0;
        double gpuScale = (totalGPURequest > gpuAvailable) ? (gpuAvailable / totalGPURequest) : 1.0;

        // Apply proportional adjustments to requests
        equilibriumReached = true;
        for (auto &vm : vms) {
            double adjustedCPU = vm.cpuRequest * cpuScale;
            double adjustedRAM = vm.ramRequest * ramScale;
            double adjustedGPU = vm.gpuRequest * gpuScale;

            // If the VM's request changes significantly, equilibrium is not reached
            if (fabs(vm.cpuRequest - adjustedCPU) > 0.01 ||
                fabs(vm.ramRequest - adjustedRAM) > 0.01 ||
                fabs(vm.gpuRequest - adjustedGPU) > 0.01) {
                equilibriumReached = false;
            }

            vm.cpuRequest = round(adjustedCPU);
            vm.ramRequest = round(adjustedRAM);
            vm.gpuRequest = round(adjustedGPU);
        }

        // Update available resources
        cpuAvailable = 0;
        ramAvailable = 0;
        gpuAvailable = 0;
        for (const auto &vm : vms) {
            cpuAvailable += vm.cpuRequest;
            ramAvailable += vm.ramRequest;
            gpuAvailable += vm.gpuRequest;
        }
    }
}

// Thread function for each VM
void vmThread(VM &vm, vector<VM> &vms, double &cpuAvailable, double
&ramAvailable, double &gpuAvailable) {
    allocateResources(vms, cpuAvailable, ramAvailable, gpuAvailable);
}

int main() {
    // Taking user input for resources
    int numVMs;
    double totalCPU, totalRAM, totalGPUs;

    cout << "Enter number of VMs: ";
    cin >> numVMs;
    cout << "Enter total CPU units: ";
    cin >> totalCPU;
    cout << "Enter total RAM units: ";
    cin >> totalRAM;
    cout << "Enter total GPU units: ";
    cin >> totalGPUs;

    vector<thread> vmThreads;
    vector<VM> vms;

    srand(time(0)); // Seed random number generator

    // Creating VMs with random resource requests
    for (int i = 0; i < numVMs; i++) {
        double cpuReq = (rand() % static_cast<int>(totalCPU / 2)) + 1;
        double ramReq = (rand() % static_cast<int>(totalRAM / 2)) + 1;
        double gpuReq = (rand() % static_cast<int>(totalGPUs / 2)) + 1;

        vms.push_back({i + 1, cpuReq, ramReq, gpuReq});
    }

    // Launching threads for each VM
    for (int i = 0; i < numVMs; i++) {
        vmThreads.push_back(thread(vmThread, ref(vms[i]), ref(vms), ref(totalCPU), ref(totalRAM), ref(totalGPUs)));
    }

    // Joining threads
    for (auto &t : vmThreads) {
        t.join();
    }

    // Display final allocations
    cout << "\nFinal VM Resource Allocations (Nash Equilibrium Achieved):\n";
    for (const auto &vm : vms) {
        cout << "VM " << vm.id << " -> CPU: " << vm.cpuRequest
             << ", RAM: " << vm.ramRequest
             << ", GPU: " << vm.gpuRequest << endl;
    }

    return 0;
}
```

INPUT/ OUTPUT for 2 Senerios:

The program takes user input for total available resources (CPU, RAM, GPU), number of Vms.

Input Format

1. User enters the total available resources:

- Number of Virtual Machines (numVMs)
- Total CPU units (totalCPU)
- Total RAM units (totalRAM)
- Total GPU units (totalGPUs)

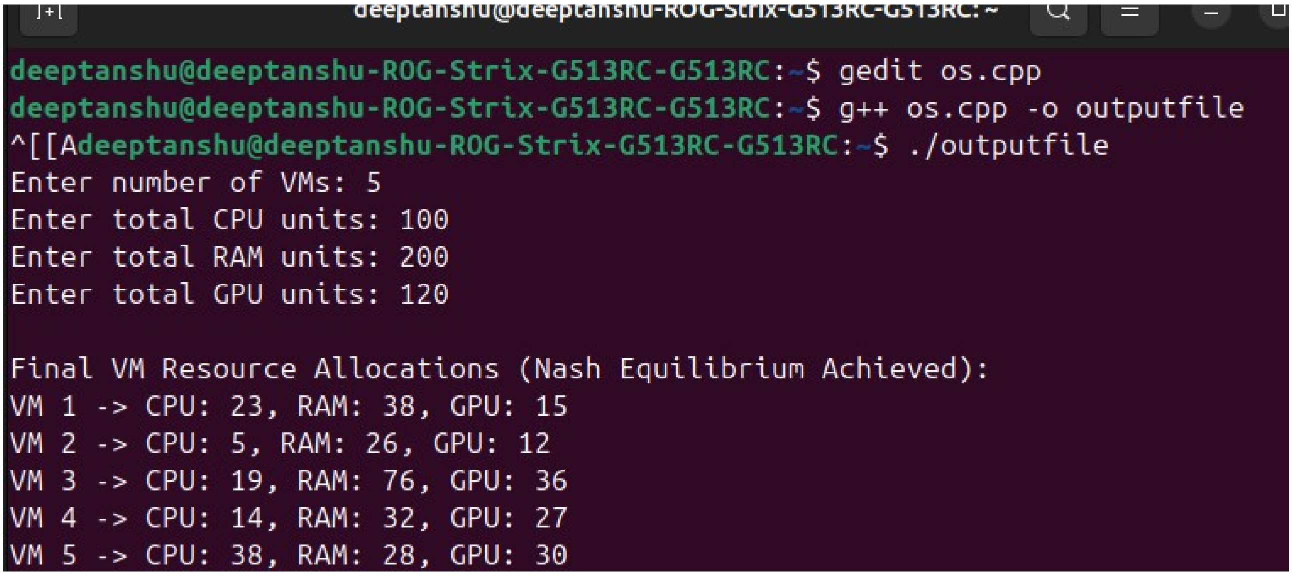
2. User enters resource requests and priority for each VM:

- CPU required (cpuReq)
- RAM required (ramReq)
- GPU required (gpuReq)

Output Format

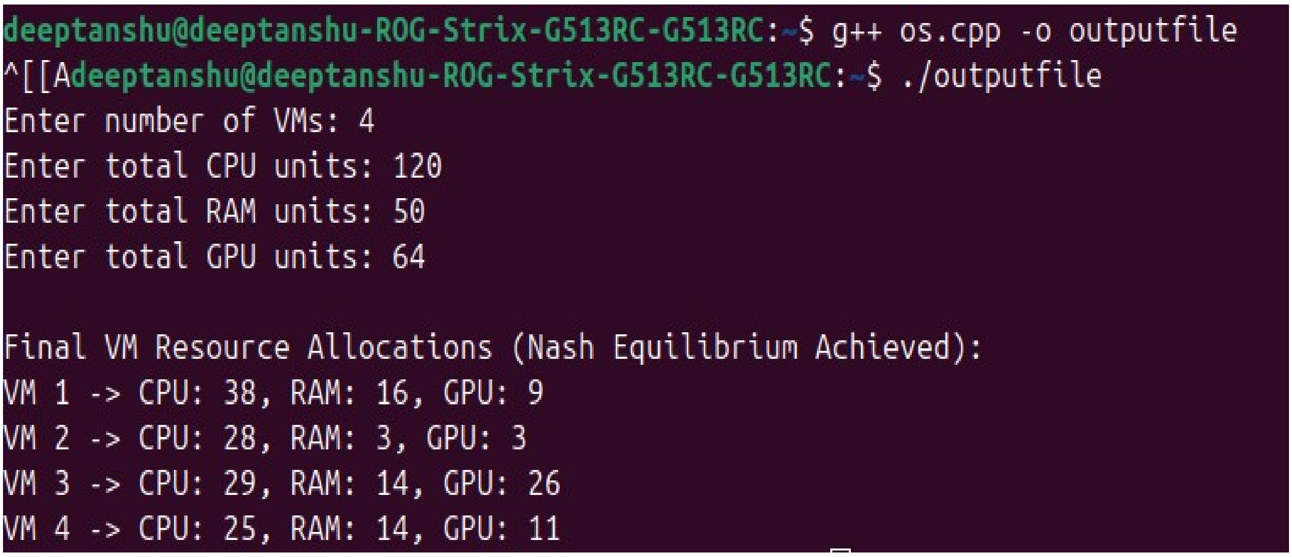
1.Threads are created for each VM.

- Each thread tries to allocate resources.
- The first VMs in sorted order (higher priority) get resources.
- Lower-priority VMs may be denied due to resource shortages.



```
deeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~
deeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~$ gedit os.cpp
deeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~$ g++ os.cpp -o outputfile
^[[Adeeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~$ ./outputfile
Enter number of VMs: 5
Enter total CPU units: 100
Enter total RAM units: 200
Enter total GPU units: 120

Final VM Resource Allocations (Nash Equilibrium Achieved):
VM 1 -> CPU: 23, RAM: 38, GPU: 15
VM 2 -> CPU: 5, RAM: 26, GPU: 12
VM 3 -> CPU: 19, RAM: 76, GPU: 36
VM 4 -> CPU: 14, RAM: 32, GPU: 27
VM 5 -> CPU: 38, RAM: 28, GPU: 30
```



```
deeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~$ g++ os.cpp -o outputfile
^[[Adeeptanshu@deeptanshu-ROG-Strix-G513RC-G513RC:~$ ./outputfile
Enter number of VMs: 4
Enter total CPU units: 120
Enter total RAM units: 50
Enter total GPU units: 64

Final VM Resource Allocations (Nash Equilibrium Achieved):
VM 1 -> CPU: 38, RAM: 16, GPU: 9
VM 2 -> CPU: 28, RAM: 3, GPU: 3
VM 3 -> CPU: 29, RAM: 14, GPU: 26
VM 4 -> CPU: 25, RAM: 14, GPU: 11
```

Conclusion:

This program efficiently allocates **CPU, RAM, and GPU** resources to **multiple Virtual Machines(VMs)** using a priority-based multithreaded system.

• User inputs define the t**otal available resources** and the **number of VMs.**
• **VMs** get resources first, ensuring **fair allocation** based on **Nash Equilibrium.**
• If resources are exhausted, resources are made to **scale down to rounded integer.**

This **multithreaded, Nash-based VM resource allocation system effectively manages computational resources in virtualized environments. It ensures fairness, efficiency, and flexibility**