Memory management

Abstract

In a multiprogramming computer, the operating system resides in a part of memory and the rest is used by multiple processes. The task of subdividing the memory among different processes is called memory management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Our project is a memory manager simulation that allocates and deallocates memory to process requests. There are two separate programs. The first is a console for entering memory requests, and the second is the memory manager. They communicate via an essage passing system.

- In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the program currently being executed.
- In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

Requirements

1. Dynamic Memory Management
2. Pool allocation
3. Fixed-size block allocation
4. Buddy allocations
5. Slab allocations
6. Stack allocation
7. Garbage collection
8. Out of memory
9. Relocation
10. Protection
11. Sharing
12. Logical Organization
13. Physical Organization

Requirements we are working on

1. MEMORY ALLOCATOR-Achintya Misra

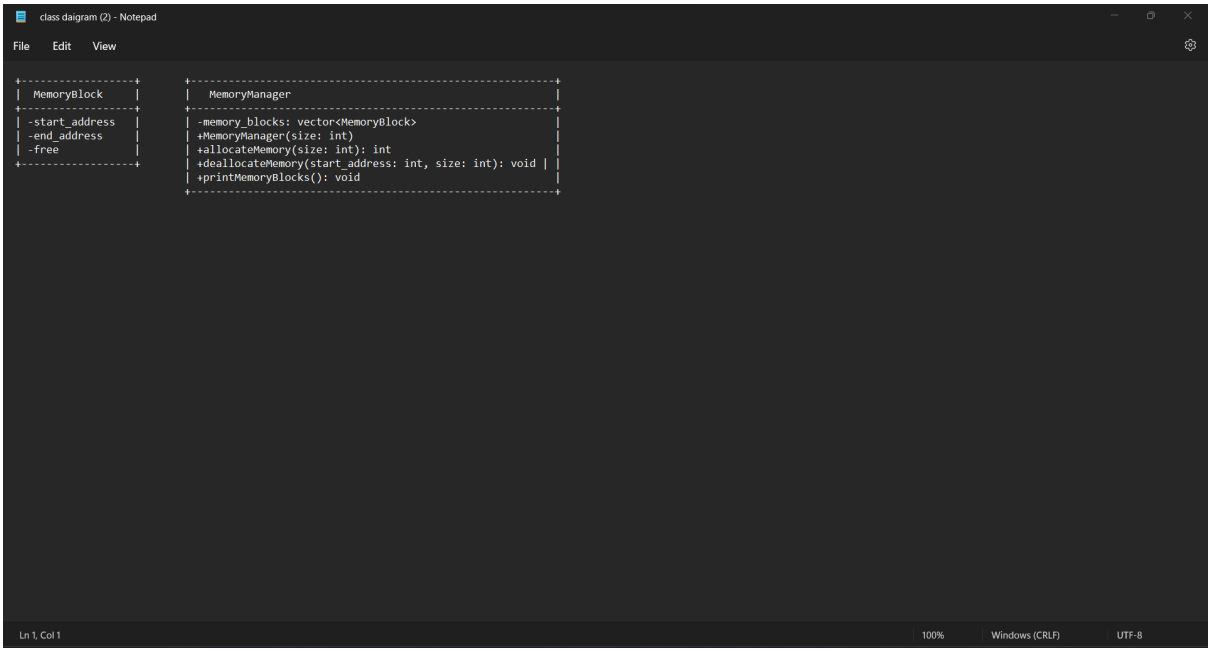
2. MEMORY DEALLOCATOR- Sidharth Thakur

3. DATA STRUCTURES-Shaurya Nagpal

Header files used

1. iostream
2. vector
3. algorithm

CLASS DIAGRAM



OBJECT DIAGRAM

The MemoryBlock class defines the structure of each memory block that will be managed by the MemoryManager class.

The MemoryManager class contains a vector of MemoryBlock objects, representing the available memory blocks.

The MemoryManager class has a constructor that takes an integer size as input to initialize the vector of memory blocks.

The MemoryManager class has a allocateMemory method that takes an integer size as input and returns the starting address of the allocated memory block if it is available, or -1 if not.

The MemoryManager class has a deallocateMemory method that takes a starting address and size as input, updates the corresponding memory block, and merges adjacent free memory blocks if any.

The MemoryManager class has a printMemoryBlocks method that prints the details of each memory block in the vector.

The object diagram shows an instance of the MemoryManager class with a vector containing one MemoryBlock object representing the entire available memory block of size 100.

WORK DONE TILL NOW/CODE TILL 26TH FEBRUARY

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// define a memory block struct
struct MemoryBlock {
    int start_address;
    int end_address;
    bool free;
};

// define a memory manager class
class MemoryManager {
private:
    vector<MemoryBlock> memory_blocks; // vector to store memory blocks
public:
    MemoryManager(int size) {
        // initialize the memory blocks with the given size
        memory_blocks.push_back({0, size-1, true});
    }

    // allocate memory for a process
    int allocateMemory(int size) {
        // find a free memory block that can accommodate the requested size
        auto it = find_if(memory_blocks.begin(), memory_blocks.end(),
[&](const MemoryBlock& block) {
            return block.free && (block.end_address - block.start_address + 1)
>= size;
        });

        if (it == memory_blocks.end()) {
            // no free memory block found
            return -1;
        }

        // update the found memory block and return the starting address of
the allocated memory
        MemoryBlock& block = *it;
        int start_address = block.start_address;
        block.start_address += size;
        block.free = (block.end_address - block.start_address + 1) > 0;
        return start_address;
    }

    // deallocate memory for a process
    void deallocateMemory(int start_address, int size) {
        // find the memory block containing the specified starting address
```

```
        auto it = find_if(memory_blocks.begin(), memory_blocks.end(),
[&](const MemoryBlock& block) {
            return block.start_address == start_address;
        });

        if (it == memory_blocks.end()) {
            // no memory block found with the specified starting address
            return;
        }

        // update the memory block
        MemoryBlock& block = *it;
        block.start_address -= size;
        block.free = true;

        // merge adjacent free memory blocks
        if (it != memory_blocks.begin() && (it-1)->free) {
            (it-1)->end_address = block.end_address;
            memory_blocks.erase(it);
            it = it-1;
            block = *it;
        }
        if (it != memory_blocks.end()-1 && (it+1)->free) {
            block.end_address = (it+1)->end_address;
            memory_blocks.erase(it+1);
        }
    }

    // print the memory blocks
    void printMemoryBlocks() {
        for (const MemoryBlock& block : memory_blocks) {
            cout << "Start Address: " << block.start_address << ", End
Address: " << block.end_address << ", Free: " << block.free << endl;
        }
    }
};

int main()
{
    MemoryManager manager(100); // create a memory manager with 100 units of
memory

    int pid1 = 1, pid2 = 2;
    int size1 = 20, size2 = 30;

    // allocate memory for process 1 and print the memory blocks
    int start_address1 = manager.allocateMemory(size1);
    if (start_address1 != -1) {
        cout << "Allocated " << size1 << " units of memory for process " <<
pid1 << " starting from address " << start_address1 << endl;
        manager.printMemoryBlocks();
    } else {
```

```
        cout << "Unable to allocate memory for process " << pid1 << endl;
    }

    // allocate memory for process 2 and print the memory blocks

}
```

In this code we have not used DSA yet as we were not able to make up our mind whether which requirement to work on

By next time, we will add template classes using api as we were facing issues in requirements

<u>TEAM MEMBERS</u>

Team member 1: Achintya Misra(211166)

Team member 2: Sidharth Thakur(211157)

Team member 3:Shaurya Nagpal(211155)