Week 5: Memory Management

Headers and Namespace

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <vector>
#include <string>

using namespace std;
```

- **Headers**: These include standard C++ libraries for input-output operations (`iostream`), file handling (`fstream`), string stream handling (`sstream`), and data structures (`map`, `vector`, `string`).
- **Namespace**: `using namespace std;` allows us to use standard library functions without needing to prefix them with `std::`.

Enum for Instruction Types

```
enum InstructionType { ADD, SUB, LOAD, STORE, UNKNOWN };
```

- **Enum**: Defines possible instruction types for better code readability and maintainability.

ALU Class

```
class ALU {
public:
    int performOperation(const string& opcode, int operand1, int operand2) {
        if (opcode == "ADD") return operand1 + operand2;
        if (opcode == "SUB") return operand1 - operand2;
        if (opcode == "LOAD") return operand2;
        if (opcode == "STORE") return operand1;
        return 0;
    }
};
```

- **ALU Class**: Implements an Arithmetic Logic Unit (ALU) to perform operations based on the opcode.
- **performOperation**: Handles different operations like ADD, SUB, LOAD, STORE based on the provided opcode.

Registers Class

```
class Registers {
public:
    map<string, int> regs;
    Registers() {
        regs["R0"] = 0; regs["R1"] = 4; regs["R2"] = 9; regs["R3"] = 10;
    }
    int get(const string& reg) { return regs[reg]; }
    void set(const string& reg, int value) { regs[reg] = value; }
    void display(ostream& outputStream) {
        for (const auto& reg : regs) {
            outputStream << reg.first << ": " << reg.second << " ";
        }
        outputStream << endl;
    }
};
```

- **Registers Class**: Manages the general-purpose registers.
- **get**: Retrieves the value of a register.
- **set**: Sets the value of a register.
- **display**: Displays all register values, outputting them to the specified output stream.

Memory Class

```
class Memory {
public:
    vector<int> memorySpace;
    Memory(int size) : memorySpace(size, 0) {}
    int read(int address) {
        if (address < 0 || address >= memorySpace.size()) {
            cout << "Memory read error: Address out of bounds" << endl;
            return -1;
        }
        return memorySpace[address];
    }
    void write(int address, int value) {
        if (address < 0 || address >= memorySpace.size()) {
            cout << "Memory write error: Address out of bounds" << endl;
            return;
        }
        cout << "Writing value " << value << " to memory address " << address << endl;
        memorySpace[address] = value;
    }
    void display(ostream& outputStream) {
        for (int i = 0; i < memorySpace.size(); ++i) {
            outputStream << "Address " << i << ": " << memorySpace[i] << " ";
        }
        outputStream << endl;
    }
};
```

- **Memory Class**: Manages the memory.
- **read**: Reads a value from a given memory address.
- **write**: Writes a value to a given memory address.
- **display**: Displays all memory values, outputting them to the specified output stream.

CPU Class

```
class CPU {
public:
    int programCounter;
    vector<int> instructionMemory;
    Registers registers;
    ALU alu;
    Memory memory;
    CPU() : programCounter(0), memory(25) {} // Initialize with memory size 25
    void loadProgram(const vector<int>& program) {
        instructionMemory = program;
    }
    void executeProgram(ostream& outputStream) {
        while (programCounter < instructionMemory.size()) {
            int instruction = instructionMemory[programCounter];
            outputStream << "Fetching instruction at address " << programCounter << ": " <<
            programCounter++;
            decodeAndExecute(instruction, outputStream);
        }
    }

private:
    void decodeAndExecute(int instruction, ostream& outputStream) {
        int opcode = (instruction >> 6) & 0x03;
        int reg1 = (instruction >> 3) & 0x07;
        int reg2 = instruction & 0x07;
        string opcodeStr = getOpcodeString(opcode);
        outputStream << "Decoding instruction: " << instruction << " as (" << opcodeStr << '
        int operand1 = registers.get("R" + to_string(reg1));
        int operand2 = registers.get("R" + to_string(reg2));
        outputStream << "Operands: " << "operand1 = " << operand1 << ", operand2 = " << oper
        int result = alu.performOperation(opcodeStr, operand1, operand2);
        if (opcodeStr == "LOAD") {
            int value = memory.read(operand2);
            registers.set("R" + to_string(reg1), value);
            outputStream << "Loaded value " << value << " into R" << reg1 << endl;
        } else if (opcodeStr == "STORE") {
            memory.write(operand2, operand1);
            outputStream << "Stored value " << operand1 << " at memory address " << operand2
        } else {
            registers.set("R" + to_string(reg1), result);
            outputStream << "Executing instruction: " << instruction << " (" << opcodeStr <<
            outputStream << "Updated R" << reg1 << " to " << result << endl;
        }
        outputStream << "Current Register States: ";
        registers.display(outputStream);
        outputStream << "Current Memory State: ";
        memory.display(outputStream);
        outputStream << endl;
    }

    string getOpcodeString(int opcode) {
        switch (opcode) {
            case 0: return "ADD";
            case 1: return "SUB";
            case 2: return "LOAD";
            case 3: return "STORE";
            default: return "UNKNOWN";
        }
    }
};
```

- **CPU Class**: Orchestrates the execution of instructions.
- **programCounter**: Tracks the address of the next instruction to execute.
- **loadProgram**: Loads the program (machine code) into instruction memory.
- **executeProgram**: Fetches, decodes, and executes instructions in a loop, outputting the results to the specified output stream.
- **decodeAndExecute**: Decodes the opcode, fetches operands, performs operations, and updates registers or memory.
- **getOpcodeString**: Converts numeric opcode to its string representation.

Assembler Function

```
vector<int> assemble(const string& assemblyCode) {
    map<string, int> opcodes = {{"ADD", 0}, {"SUB", 1}, {"LOAD", 2}, {"STORE", 3}};
    map<string, int> registers = {{"R0", 0}, {"R1", 1}, {"R2", 2}, {"R3", 3}};
    istringstream iss(assemblyCode);
    string line;
    vector<int> machineCode;
    while (getline(iss, line)) {
        istringstream linestream(line);
        string opcode, reg1, reg2;
        linestream >> opcode >> reg1 >> reg2;
        int machineInstruction = (opcodes[opcode] << 6) | (registers[reg1] << 3) | registers
        machineCode.push_back(machineInstruction);
    }
    return machineCode;
}
```

- **assemble**: Converts assembly code to machine code using predefined opcode and register mappings.

Main Function

```
int main() {
    CPU cpu;
    string assemblyCode;

    // Load assembly code from a file
    ifstream inputFile("input.txt");
    if (inputFile.is_open()) {
        string line;
        while (getline(inputFile, line)) {
            assemblyCode += line + "\n";
        }
        inputFile.close();
        cout << "Input loaded from input.txt" << endl;
    } else {
        cout << "Unable to open input.txt" << endl;
        return 1;
    }

    cout << "Sample Assembly Code:\n" << assemblyCode << endl;

    // Convert assembly to machine code
    cout << "\nAssembling code...\n";
    vector<int> machineCode = assemble(assemblyCode);
    cout << "Converted Machine Code:\n";
    for (int code : machineCode) {
        cout << code << " ";
    }
    cout << endl;

    // Display initial register states
    cout << "\nInitial Register States:\n";
    cpu.registers.display(cout);

    // Load and execute program
    cpu.loadProgram(machineCode);
    cout << "\nExecuting program...\n";

    // Redirect output to both console and file
    ofstream outputFile("output.txt");
    if (outputFile.is_open()) {
        ostringstream outputBuffer;
        cpu.executeProgram(outputBuffer);

        // Write buffer to file
        outputFile << outputBuffer.str();
        outputFile.close();

        cout << "Output saved in output.txt" << endl;
        cout << outputBuffer.str(); // Display buffer content to console
    } else {
        cout << "Unable to open output.txt" << endl;
    }

    // Display final register states
    cout << "Final Register States:\n";
    cpu.registers.display(cout);

    return 0;
}
```

Explanation

1. **Initialize the CPU and Assembly Code String**:

```
CPU cpu;
string assemblyCode;
```

   - Creates an instance of the CPU class.
   - Declares a string variable to hold the assembly code read from the input file.

2. **Load Assembly Code from a File**:

```
ifstream inputFile("input.txt");
if (inputFile.is_open()) {
    string line;
    while (getline(inputFile, line)) {
        assemblyCode += line + "\n";
    }
    inputFile.close();
    cout << "Input loaded from input.txt" << endl;
} else {
    cout << "Unable to open input.txt" << endl;
    return 1;
}
```

- Opens the input file `input.txt`.
- Reads each line of the file and appends it to the `assemblyCode` string.
- Closes the input file and displays a message indicating that the input was loaded.
- If the file cannot be opened, it displays an error message and exits the program.

3. **Display the Loaded Assembly Code**:

```
cout << "Sample Assembly Code:\n" << assemblyCode << endl;
```

- Outputs the loaded assembly code to the console for verification.

4. **Convert Assembly Code to Machine Code**:

```
cout << "\nAssembling code...\n";
vector<int> machineCode = assemble(assemblyCode);
cout << "Converted Machine Code:\n";
for (int code : machineCode) {
    cout << code << " ";
}
cout << endl;
```

- Calls the `assemble` function to convert the assembly code into machine code.
- Outputs the resulting machine code to the console.

5. **Display Initial Register States**:

```
cout << "\nInitial Register States:\n";
cpu.registers.display(cout);
```

- Displays the initial state of the registers using the `display` method of the `Registers` class.

6. **Load and Execute the Program**:

```
cpu.loadProgram(machineCode);
cout << "\nExecuting program...\n";
```

- Loads the machine code into the CPU's instruction memory.
- Outputs a message indicating that the program execution is starting.

7. **Redirect Output to Both Console and File**:

```
ofstream outputFile("output.txt");
if (outputFile.is_open()) {
    ostringstream outputBuffer;
    cpu.executeProgram(outputBuffer);
    // Write buffer to file
    outputFile << outputBuffer.str();
    outputFile.close();

    cout << "Output saved in output.txt" << endl;
    cout << outputBuffer.str(); // Display buffer content to console
} else {
    cout << "Unable to open output.txt" << endl;
}
```

- Opens the output file `output.txt`.
- Executes the program, capturing the output in an `ostringstream` buffer.
- Writes the captured output to the output file and closes it.
- Displays a message indicating that the output was saved.
- Outputs the captured buffer content to the console.

8. **Display Final Register States**:

```
cout << "Final Register States:\n";
cpu.registers.display(cout);
```

- Displays the final state of the registers using the `display` method of the `Registers` class.

9. **Return**:

```
return 0;
```

- Returns 0 to indicate that the program executed successfully.