***Phase 1 Implementation Of Our Operating System***

**Introduction**

The Project Phase 1 implements a simple operating system (OS) that simulates basic memory management, instruction execution, and input/output handling on a virtual machine. The OS operates by loading programs into memory, executing them using a basic instruction set, and interacting with input and output devices. This project demonstrates how an OS manages memory, processes instructions, and handles interrupts.

The primary objective is to construct a virtual machine with memory management and a CPU that can read instructions and data from files, execute the instructions, and manage input/output operations.

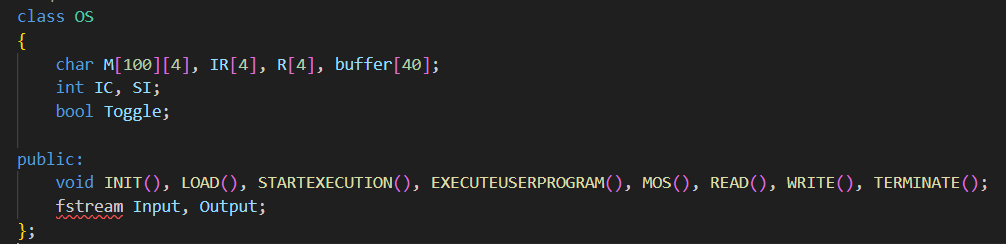
**Variable Descriptions**

The OS class uses several variables and arrays to manage the simulated machine:

* **M[100][4]**: Represents the main memory, with 100 words, each consisting of 4 bytes.
* **IR[4]**: Instruction Register, holds the current instruction being executed.
* **R[4]**: General Purpose Register, used to store temporary data during operations.
* **buffer[40]**: A temporary buffer for reading input from files, representing one card (40 bytes).
* **IC**: Instruction Counter, points to the memory address of the next instruction to execute.
* **SI**: Service Interrupt, manages system calls for operations like reading, writing, and halting.
* **Toggle**: A boolean variable used for conditional jumps.

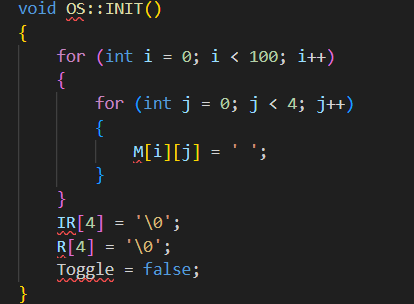
### Program Implementation

#### Class Definition:-

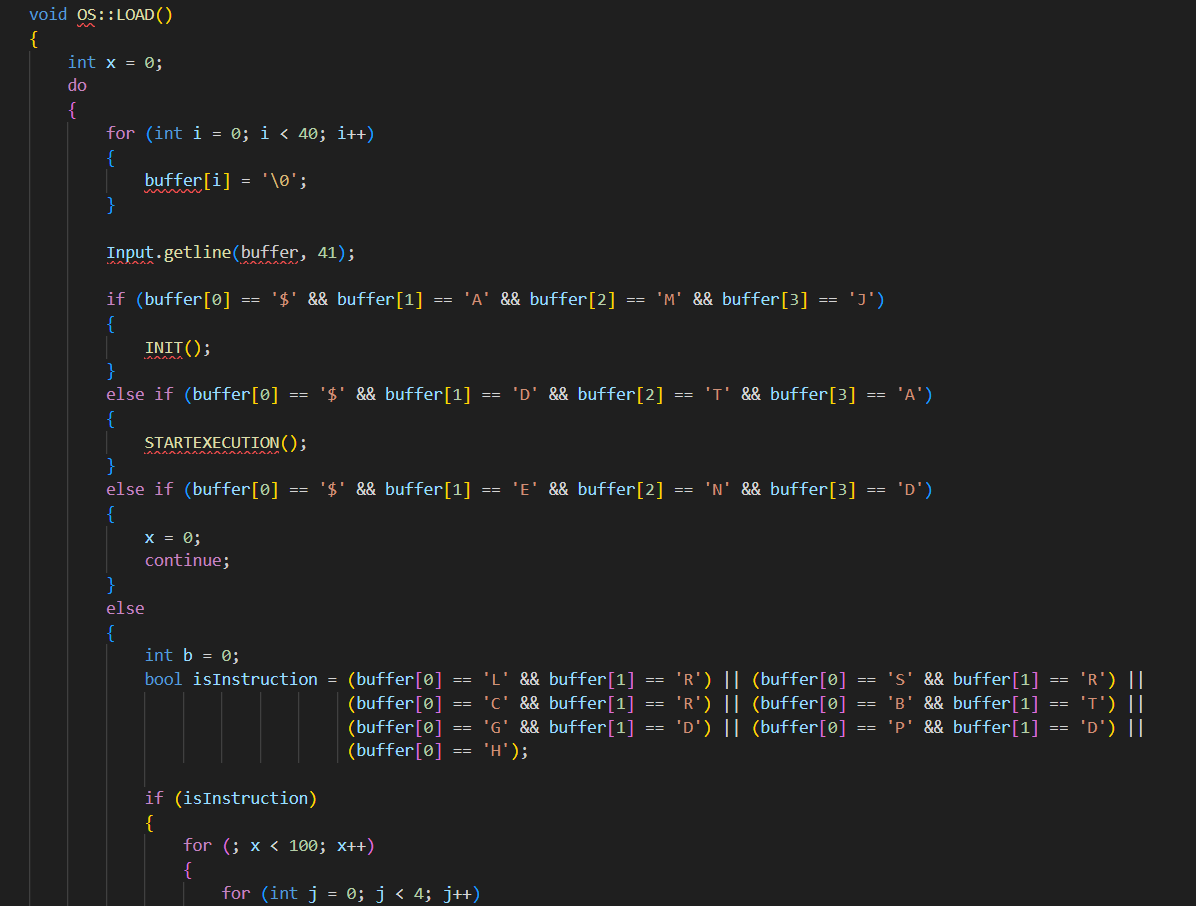


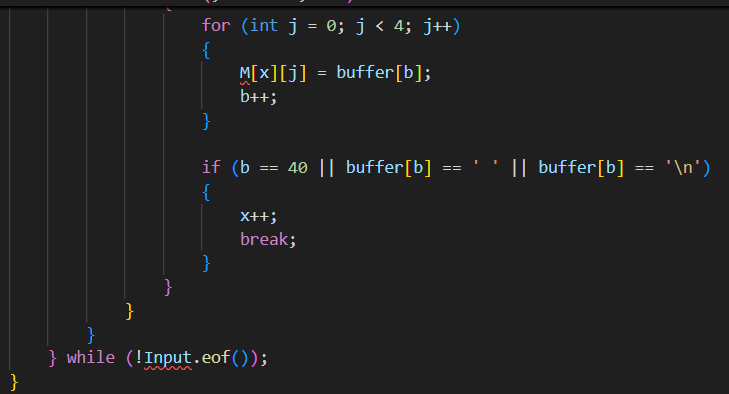
This section defines the OS class, which encapsulates the entire functionality of the virtual operating system. It includes memory (M), registers (IR, R), and control variables (IC, SI, Toggle). Additionally, file streams for input and output are defined.

## *INIT Function:-*

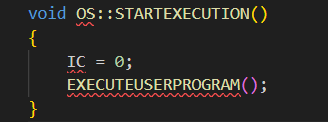


The INIT function initializes memory and registers to empty values. All memory locations are set to spaces, and the IR and R registers are cleared. The Toggle is set to false (used for conditional jumps).





The LOAD function is responsible for loading the job into memory. It reads lines from the input file, identifies control cards ($AMJ, $DTA, $END), and appropriately processes them. Instructions are loaded into memory sequentially, and execution begins when the $DTA card is encountered.  
  
*STARTEXECUTION Function:-*



This function starts program execution by initializing the IC to 0 (the first instruction) and calling the EXECUTEUSERPROGRAM function to begin the instruction cycle.

## *EXECUTE USER PROGRAM Function:-*

This core function fetches and decodes instructions from memory. It handles basic operations like loading, storing, comparing, and branching. It also manages I/O operations through service interrupts (SI).

**Theoretical Background and Importance**

The system simulates a simple multiprogramming OS, focusing on memory management, instruction execution, and I/O operations. This phase of the project helps in understanding the following:

* **Memory Management**: Divides memory into blocks and simulates loading and executing instructions.
* **Instruction Set Simulation**: Simulates common instructions like load, store, compare, and branch.
* **Service Interrupts (SI)**: Introduces I/O operations, essential for modern operating systems.

## *MOS Function:-*

The **MOS (Master Mode)** function is called when a service interrupt (SI) is raised. It checks the value of SI and triggers the appropriate system service, such as READ, WRITE, or TERMINATE. In this particular case, only the TERMINATE service is implemented (when SI = 3), which halts the program execution.

**Theoretical Importance**:

* **Master Mode Operation**: The MOS function simulates how the OS manages system-level services by handling interrupts. It provides a layer of abstraction where high-level operations such as I/O requests are managed. In a real-world OS, this type of mechanism ensures that the CPU can handle user processes as well as system calls.

## *READ Function:-*

The **READ** function simulates the process of fetching data from an input device (in this case, a file) and loading it into memory. It reads one "card" (40 bytes) at a time, starting from the memory location specified in the instruction register (IR).

* **Buffering**: The buffer is used to temporarily store the incoming data before it is written to memory.
* **Memory Loading**: The data from the buffer is transferred into blocks of memory starting at the location specified in the instruction.

**Theoretical Importance**:

* **I/O Simulation**: This function simulates an essential feature of operating systems—handling input from external devices like keyboards, card readers, etc. The READ operation shows how an OS fetches data from a source and transfers it to memory, which is essential for program execution.

## *WRITE Function:-*

The **WRITE** function handles the output operation. It reads 40 bytes of data (one block) from memory starting from the location specified in the instruction, transfers it to a buffer, and writes it to the output file.

**Theoretical Importance**:

* **Output Handling**: This function highlights the OS's role in output management, allowing the system to communicate results back to the user via an output device, such as a file or a printer. It's crucial in providing feedback from the system to the user after executing the program.

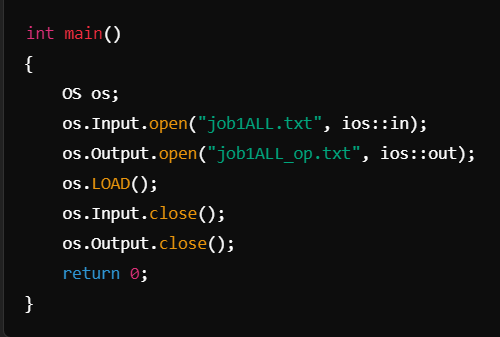
## *TERMINATE Function:-*

The **TERMINATE** function handles the halting of program execution. It simply writes two blank lines to the output file to indicate that the job has completed and resets SI to 0.

**Theoretical Importance**:

* **Job Termination**: Termination is a critical part of the job lifecycle in operating systems. The OS needs to gracefully stop processes and clean up resources, ensuring system stability.

#### Main Function:-



The **main** function is the entry point of the program. It initializes the OS object, opens the input and output files, loads the program, and then closes the files after execution.

* **job1ALL.txt**: This file contains the input program and data in the form of control cards, program instructions, and data cards.
* **job1ALL\_op.txt**: This file will store the output produced by the program.

**Theoretical Importance**:

* **File Handling**: The main function showcases how operating systems handle files, read programs, and write outputs. File handling is crucial for managing user data and programs in any operating system.

### Input and Output Handling with Error Management

The **input and output operations** are crucial aspects of the operating system. They simulate reading data from an input source (like a file or a card reader) and writing the output to a destination (such as a file or a printer). The code for these sections is implemented using file handling functions in C++, specifically focusing on simulating the **READ** and **WRITE** operations as described in the instruction set. In this phase, the input and output errors need to be handled as per the specifications outlined by our Teacher Mrs. **Archana Burujwale.**

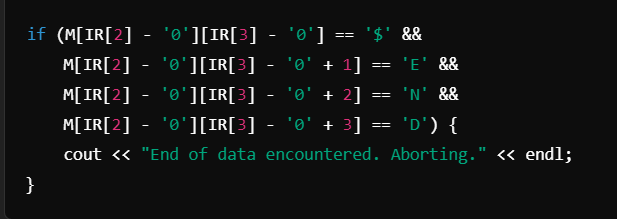
### Input Section (READ Operation)

The **READ** operation simulates reading from a data card into memory. In the code, this is performed when the instruction GD (Get Data) is encountered. The function reads a block of data (40 bytes) from the input file and stores it in memory.

#### Error Handling in the Input Section

According to the PowerPoint document, the following errors must be handled in the input section:

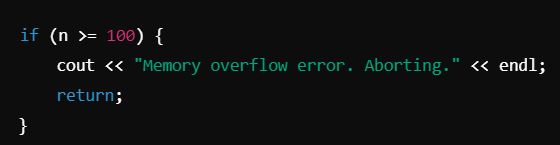
1. **Out of Data Error**: This error occurs when the program tries to read beyond the available data in the input file. To handle this, the program should check if the $END control card is encountered during the read operation, indicating that no more data is available.
   * In the **READ** function, an end-of-data check is performed using



*  This code checks if the string $END appears in memory, signaling the end of the data section, and if encountered, the system halts further reading and aborts the program.

 **Memory Overflow**: This error occurs if the program attempts to load more data into memory than is available. The PowerPoint mentions that memory should not exceed 100 words. If an attempt is made to load beyond this limit, the program should abort with an appropriate error message.

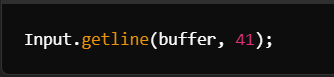
* The current implementation does not explicitly handle memory overflow in the **READ** function. To manage this, we would need to add a check before loading data:



#### How the Input is Handled in the Code

When the **READ** function is called (upon encountering a GD instruction), it performs the following:

* Initializes the buffer to store incoming data from the input file.
* Reads a block of data (40 bytes) into the buffer using



* Transfers the data from the buffer to memory, starting at the memory location specified by the operand in the instruction (IR[2] and IR[3]).

This simulates how an OS reads input data from an external source (like a card reader or disk) and stores it in memory for further processing by the program.

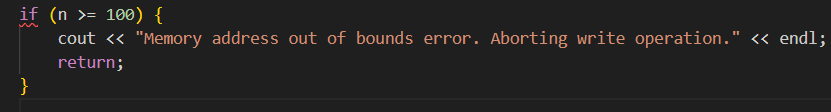
### Output Section (WRITE Operation)

The **WRITE** operation is invoked when a PD (Print Data) instruction is encountered. This function retrieves a block of data (40 bytes) from memory and writes it to the output file.

#### Error Handling in the Output Section

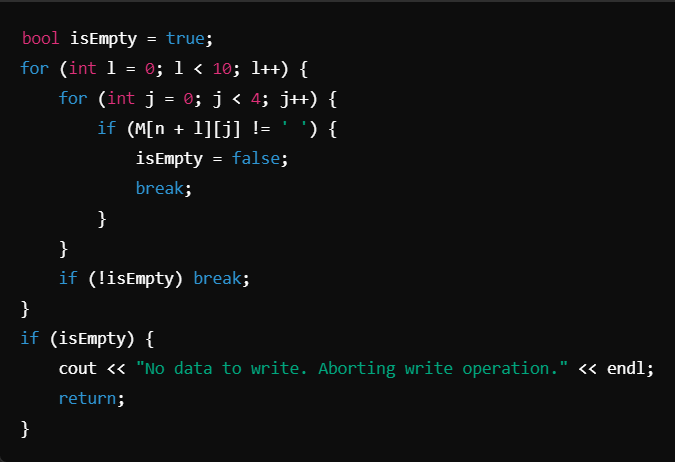
Similar to the input section, the PowerPoint document outlines certain errors that should be managed in the output section:

1. **Out of Memory Error**: This error occurs when the program attempts to write data from a memory location that is invalid or out of bounds. To handle this, the memory location must be validated before retrieving the data for writing.
   * In the **WRITE** function, we can add a check to ensure the memory address is within valid bounds (i.e., less than 100):



**No Data to Write Error**: This occurs if the program attempts to write from memory locations that do not contain valid data. In the current implementation, empty memory locations are initialized to spaces (' '), which could be checked before writing.

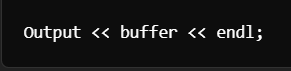
* To handle this error, a check can be added to ensure that the memory location contains valid data before writing to the output file:



#### How the Output is Handled in the Code

The **WRITE** function performs the following:

* Initializes the buffer to empty.
* Retrieves a block of data from memory, starting at the memory location specified in the IR.
* Writes the data to the output file using



This simulates how the OS writes data to an external device (such as a printer or a display) by transferring information from memory to the output.

**Error Handling Based on the PowerPoint Requirements**

The PowerPoint document mentions the various types of errors that must be handled, specifically focusing on data limits, memory bounds, and proper handling of control cards. Based on the document's specifications​(OS\_Course Project Phase…), here's a list of errors that need to be addressed and how they can be managed in the code:

1. **Out of Data**:
   * Handled in the **READ** function by checking if $END is encountered, signaling the end of data.
2. **Memory Overflow**:
   * Needs to be checked in both the **READ** and **WRITE** functions to ensure memory limits are respected.
3. **Out of Memory**:
   * This error occurs when trying to write from or read to a memory location that is out of bounds. It can be handled by ensuring memory locations are within the valid range (0-99).
4. **No Data to Write**:
   * Ensures that memory locations contain valid data before attempting to write them to the output. Empty memory (initialized with spaces) should not be written to the output.

**Conclusion on Input and Output Error Handling**

In this simple OS simulation, **input and output operations** are essential in demonstrating how data is transferred between the system's memory and external devices. Error handling is crucial to ensure that the program operates within the defined limits (memory size, input data availability) and gracefully handles issues like **out of data** or **memory overflow**. The project successfully simulates basic OS input/output mechanisms while adhering to the constraints outlined in the assignment.

By implementing these checks and error handling, the system ensures reliability and prevents unexpected crashes due to invalid memory accesses or missing data, which is a fundamental responsibility of any operating system.

### Conclusion

This first phase lays the groundwork for developing a more robust, multiprogramming OS. By simulating memory operations, instruction execution, and I/O handling, it provides an excellent base for introducing advanced OS concepts like process scheduling, concurrency, and deadlock management in the next phase of the project. Understanding these basic components is crucial for comprehending how modern operating systems work under the hood.

## 