Parallel & Distributed Computing [Y1]

Semester Project

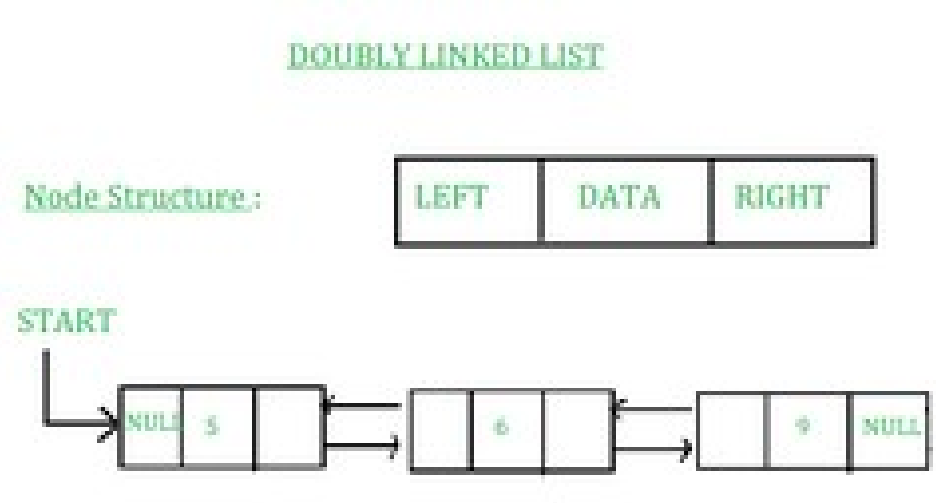
University *of* Management *and* Technology

HAMAD SALEEM
F2021376065
MIAN MUHAMMAD BILAL
[F2021408054]
MUHAMMAD TALHA
[F2021408057]

Doubly Link List

Introduction:

A doubly linked list is a bi-directional linked list. So, you can traverse it in both directions. Unlike singly linked lists, its nodes contain one extra pointer called the **previous pointer**. This pointer points to the **previous node.**



Explanation:

This project demonstrates the implementation of a doubly linked list (D_LinkedList) in C++. It supports standard operations such as insertion, deletion, and display. Additionally, OpenMP is used to parallelize data collection during the display operation.

Implementation Details

1. **Node Class**:
    - Represents a node in the doubly linked list.
    - Contains integer `data`, and pointers to the next (`next`) and previous (`prev`) nodes.
2. **D_LinkedList Class**:
    - Manages the head of the list.
    - **InsertAtBegin(int d)**: Inserts a node at the beginning of the list.
    - **InsertAtEnd(int d)**: Inserts a node at the end of the list.
    - **InsertAfter(int f, int d)**: Inserts a node with data `d` after the first node with data `f`.
    - **InsertBefore(int f, int d)**: Inserts a node with data `d` before the first node with data `f`.
    - **DeleteFromBegin()**: Deletes the node at the beginning of the list.
    - **DeleteFromEnd()**: Deletes the node at the end of the list.
    - **DeleteList()**: Deletes all nodes in the list.

- **Display()**: Displays the list contents. Uses OpenMP to parallelize data collection for display.

3. **Parallel Display with OpenMP**:
    - Counts the number of nodes.
    - Collects data from each node in parallel using OpenMP.
    - Displays the collected data sequentially.

Main Function

- Demonstrates various operations on the doubly linked list:
    - Insertions at the beginning and end.
    - Insertion after and before a specific node.
    - Deletion from the beginning and end.
    - Complete deletion of the list.
    - Displays the list after each operation.

Code:

```
#include <iostream>
#include <omp.h>
using namespace std;
class D_LinkedList{
private:
   class Node{
   public:
      int data;
      Node *next;
      Node *prev;
      Node(int data){
         this->data = data;
         this->next = nullptr;
         this->prev = nullptr;
      }
   };
   Node *head;
public:
   D_LinkedList(){
      head = nullptr;
   }
   void InsertAtBegin(int d){
      Node *newNode = new Node(d);
      if (head != nullptr){
         newNode->next = head;
         head->prev = newNode;
      }
```

```
    head = newNode;
}
void InsertAtEnd(int d){
    Node *newNode = new Node(d);
    if (head == nullptr){
        head = newNode;
        return;
    }
    Node *temp = head;
    while (temp->next != nullptr){
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
void InsertAfter(int f, int d){
    Node *temp = head;
    while (temp != nullptr && temp->data != f){
        temp = temp->next;
    }
    if (temp == nullptr){
        cout << "Node with data " << f << " not found." << endl;
        return;
    }
    Node *newNode = new Node(d);
    newNode->next = temp->next;
    newNode->prev = temp;
    if (temp->next != nullptr){
        temp->next->prev = newNode;
    }
    temp->next = newNode;
}
void InsertBefore(int f, int d){
    Node *temp = head;
    while (temp != nullptr && temp->data != f){
        temp = temp->next;
    }
    if (temp == nullptr){
        cout << "Node with data " << f << " not found." << endl;
        return;
    }
    Node *newNode = new Node(d);
```

```
    newNode->prev = temp->prev;
    newNode->next = temp;
    if (temp->prev != nullptr){
        temp->prev->next = newNode;
    }
    else
    {
        head = newNode;
    }
    temp->prev = newNode;
}
void DeleteFromBegin(){
    if (head == nullptr){
        cout << "List is already empty." << endl;
        return;
    }
    Node *temp = head;
    head = head->next;
    if (head != nullptr){
        head->prev = nullptr;
    }
    delete temp;
}
void DeleteFromEnd()
{
    if (head == nullptr){
        cout << "List is already empty." << endl;
        return;
    }
    Node *temp = head;
    while (temp->next != nullptr){
        temp = temp->next;
    }
    if (temp->prev != nullptr){
        temp->prev->next = nullptr;
    }
    else{
        head = nullptr;
    }
    delete temp;
}
void DeleteList()
```

```
    {
        while (head != nullptr)
        {
            DeleteFromBegin();
        }
    }
    void Display(){
        Node *temp = head;
        int count = 0;
        // First, count the number of nodes
        while (temp != nullptr){
            count++;
            temp = temp->next;
        }
        int *data = new int[count];
        temp = head;
// Parallelize the data collection
#pragma omp parallel for
        for (int i = 0; i < count; ++i){
            data[i] = temp->data;
            temp = temp->next;
        }
        // Display the collected data
        for (int i = 0; i < count; ++i){
            cout << data[i] << "->";
        }
        cout << "null" << endl;
        delete[] data;
    }
};
int main(){
    D_LinkedList l;
    cout << "After InsertAtBegin" << endl;
    l.InsertAtBegin(5);
    l.InsertAtBegin(4);
    l.InsertAtBegin(3);
    l.InsertAtBegin(2);
    l.InsertAtBegin(1);
    l.Display();
    cout << "\nAfter InsertAtEnd" << endl;
    l.InsertAtEnd(6);
    l.InsertAtEnd(7);
```

```
    l.Display();
    cout << "\nAfter InsertAfter" << endl;
    l.InsertAfter(3, 999);
    l.Display();
    cout << "\nAfter InsertBefore" << endl;
    l.InsertBefore(3, 111);
    l.Display();
    cout << "\nAfter DeleteFromBegin" << endl;
    l.DeleteFromBegin();
    l.Display();
    cout << "\nAfter DeleteFromEnd" << endl;
    l.DeleteFromEnd();
    l.Display();
    cout << "\nAfter DeleteList" << endl;
    l.DeleteList();
    l.Display();
    return 0;
}
```

MESI implementation

Introduction:

This Python program simulates the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol using five caches and a shared main memory. The MESI protocol ensures that data consistency between the caches and the main memory is maintained. The program allows for reading and writing operations to the caches, updating the states of cache blocks accordingly.

Constants

- `NUM_CACHES`: Number of caches (5).
- `ARRAY_SIZE`: Size of each cache and the main memory array (10).
- Cache block states: `MODIFIED`, `EXCLUSIVE`, `SHARED`, `INVALID`.

Main Components

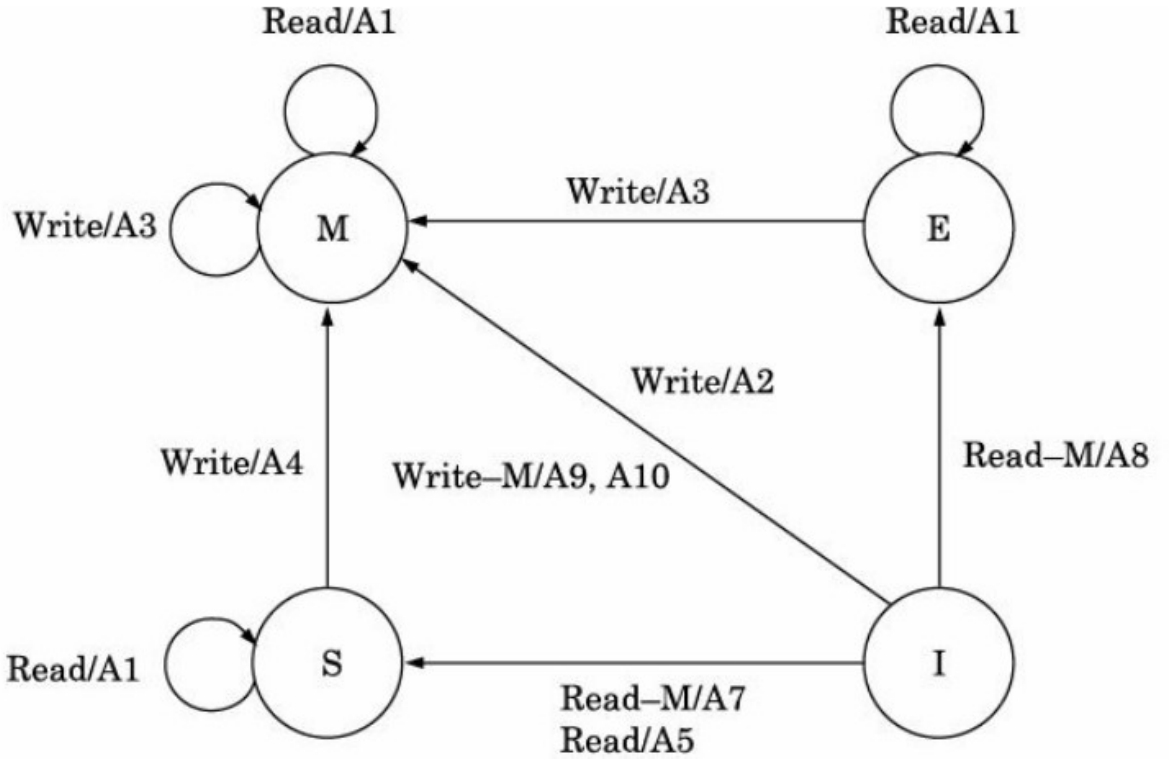
1. **Main Memory**:
    - Represented as a list (`shared_memory`) of size `ARRAY_SIZE` initialized to zero.
2. **Caches and States**:
    - `cache`: A 2D list representing the data in each cache.
    - `cache_state`: A 2D list representing the state of each block in each cache, initialized to `INVALID`.
3. **Functions**:

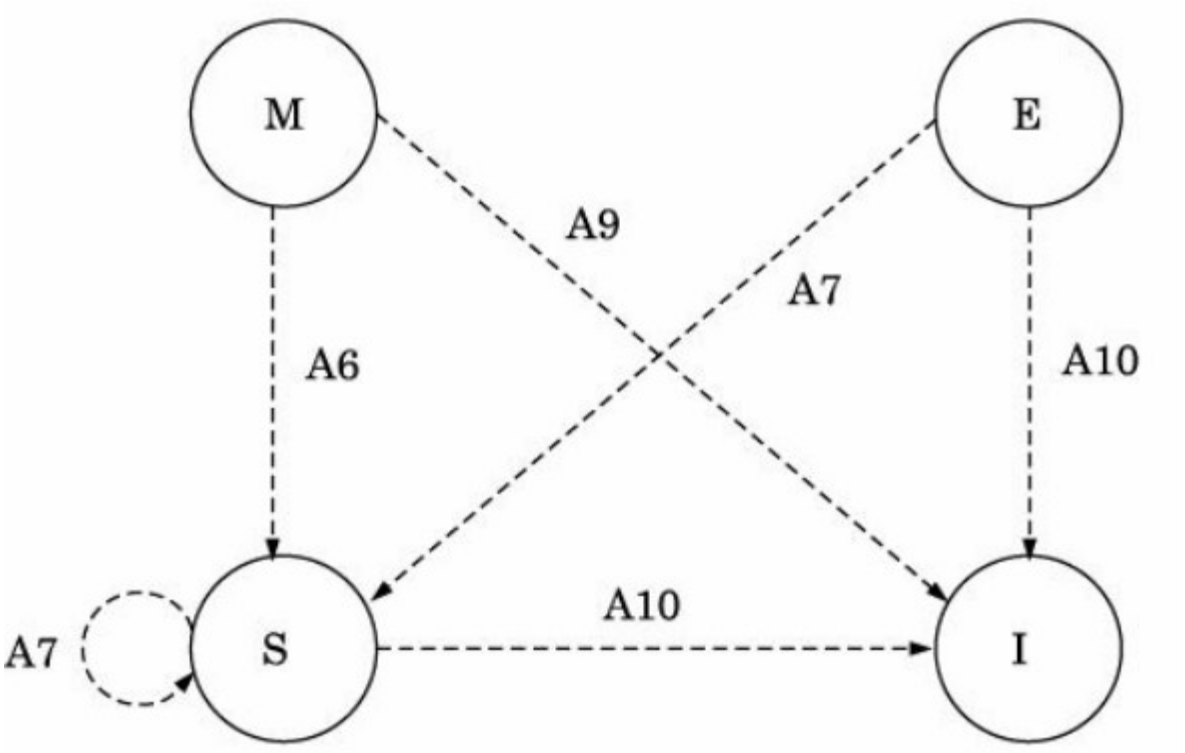
- `display_system_state()`: Displays the current state of the main memory and all caches.
- `read_data(cache_id, index)`: Handles read operations from the specified cache and index, updating states as per MESI protocol.
- `write_data(cache_id, index, data)`: Handles write operations to the specified cache and index, updating states and broadcasting invalidations to other caches as per MESI protocol.

4. **User Interaction**:
    - The `main()` function provides a command-line interface for selecting a cache, performing read or write operations, and updating the cache states.

MESI Protocol Implementation

1. **Read Operation**:
    - If the block is in `MODIFIED`, `EXCLUSIVE`, or `SHARED` state, data is read directly from the cache.
    - If the block is in `INVALID` state, data is requested from other caches or main memory:
        - If found in another cache, the block state in the requesting cache is updated to `SHARED`.
        - If not found, data is read from the main memory and the block state is set to `EXCLUSIVE`.
2. **Write Operation**:
    - If the block is in `MODIFIED` or `EXCLUSIVE` state, data is written to the cache and main memory, and the block state is set to `MODIFIED`.
    - If the block is in `SHARED` state, the state is changed to `MODIFIED`, data is written to the cache and main memory, and other caches' states for that block are set to `INVALID`.
    - If the block is in `INVALID` state, data is requested from other caches or main memory, the state is updated to `MODIFIED`, and data is written to the cache and main memory, invalidating the block in other caches.
3. **Parallel Programming**:
    - The program uses OpenMP (Open Multi-Processing) for parallel programming to improve performance. Specifically, it employs the `#pragma omp parallel for` directive in the `display_system_state()` function to parallelize the process of collecting data from the linked list.


Read/A1
Read/A1
Write/A3
Write/A3
M
E
Write/A2
Write/A4
Write–M/A9, A10
Read–M/A8
Read/A1
S
I
Read–M/A7
Read/A5


M
E
A9
A7
A6
A10
A7
S
A10
I

Code:

```
import parallel
NUM_CACHES = 5
ARRAY_SIZE = 10
# Define cache block states
MODIFIED = 'M'
EXCLUSIVE = 'E'
SHARED = 'S'
INVALID = 'I'
# Main memory (shared memory)
shared_memory = [0] * ARRAY_SIZE
# Cache arrays and states
cache = [[None] * ARRAY_SIZE for _ in range(NUM_CACHES)]
cache_state = [[INVALID] * ARRAY_SIZE for _ in range(NUM_CACHES)]
def display_system_state():
    print("Main Memory:")
    print(shared_memory)
    print("\nCaches:")
    for i in range(NUM_CACHES):
        print(f"Cache {i}:")
        print(cache[i])
        print(f"Cache State:")
        print(cache_state[i])
```

```
    print("\n")
def read_data(cache_id, index):
    current_state = cache_state[cache_id][index]
    if current_state == MODIFIED or current_state == EXCLUSIVE or
current_state == SHARED:
        print(f"Cache {cache_id} read from index {index}")
        return cache[cache_id][index]
    elif current_state == INVALID:
        print(f"'Cache {cache_id} read miss for index {
            index}. Requesting data from other caches or main memory...'")
        for other_cache_id in range(NUM_CACHES):
            if other_cache_id != cache_id and cache_state[other_cache_id][index]
!= INVALID:
                # Update current cache
                cache[cache_id][index] = cache[other_cache_id][index]
                cache_state[cache_id][index] = SHARED
                print(f"Cache {cache_id} updated from Cache {other_cache_id}")
                return cache[cache_id][index]
        # If data not found in any other cache, read from main memory
        cache[cache_id][index] = shared_memory[index]
        cache_state[cache_id][index] = EXCLUSIVE
        print(f"Cache {cache_id} read from Main Memory")
        return cache[cache_id][index]
```

```
def write_data(cache_id, index, data):
    current_state = cache_state[cache_id][index]
    if current_state == MODIFIED or current_state == EXCLUSIVE:
        print(f"Cache {cache_id} write to index {index} with data {data}")
        cache[cache_id][index] = data
        shared_memory[index] = data
        cache_state[cache_id][index] = MODIFIED
        for other_cache_id in range(NUM_CACHES):
            if other_cache_id != cache_id:
                cache_state[other_cache_id][index] = INVALID
        print(
            f"Cache {cache_id} updated and broadcasted invalidation to other caches")
    elif current_state == SHARED:
        print(f"Cache {cache_id} write to index {index} with data {data}")
        cache[cache_id][index] = data
        shared_memory[index] = data
        cache_state[cache_id][index] = MODIFIED
        for other_cache_id in range(NUM_CACHES):
            if other_cache_id != cache_id:
                cache_state[other_cache_id][index] = INVALID
        print(
```

```
            f"Cache {cache_id} updated and broadcasted invalidation to other caches")
    elif current_state == INVALID:
        print(f"'Cache {cache_id} write miss for index {
              index}. Requesting data from other caches or main memory...'")
        for other_cache_id in range(NUM_CACHES):
            if other_cache_id != cache_id and cache_state[other_cache_id][index] != INVALID:
                # Update current cache
                cache[cache_id][index] = cache[other_cache_id][index]
                cache_state[cache_id][index] = SHARED
                print(f"Cache {cache_id} updated from Cache {other_cache_id}")
                cache[cache_id][index] = data
                shared_memory[index] = data
                cache_state[cache_id][index] = MODIFIED
                for other_cache_id in range(NUM_CACHES):
                    if other_cache_id != cache_id:
                        cache_state[other_cache_id][index] = INVALID
                print(
                    f"Cache {cache_id} updated and broadcasted invalidation to other caches")
                return
        # If data not found in any other cache, read from main memory
        cache[cache_id][index] = data
        shared_memory[index] = data
```

```
        cache_state[cache_id][index] = MODIFIED
        for other_cache_id in range(NUM_CACHES):
            if other_cache_id != cache_id:
                cache_state[other_cache_id][index] = INVALID
        print(
            f"Cache {cache_id} updated and broadcasted invalidation to other
caches")



def main():
    while True:
        display_system_state()
        cache_id = int(input("Select a cache (0-4) or 'q' to quit: "))
        if cache_id == 'q':
            break
        if cache_id < 0 or cache_id >= NUM_CACHES:
            print("Invalid cache selection. Please select a cache between 0 and
4.")
            continue
        index = int(
            input(f"Select an index to read/write (0-{ARRAY_SIZE - 1}): "))
        if index < 0 or index >= ARRAY_SIZE:
            print(f"'Invalid index. Please select an index between 0 and {
                ARRAY_SIZE - 1}.'")
```

```
            continue
        action = input("Select action: 'r' for read, 'w' for write: ").lower()
        if action == 'r':
            read_data(cache_id, index)
        elif action == 'w':
            data = int(input("Enter data to write: "))
            write_data(cache_id, index, data)
        else:
            print("Invalid action. Please select 'r' for read or 'w' for write.")
if __name__ == "__main__":
    main()
```