Parallel & Distributed Computing [Y1]

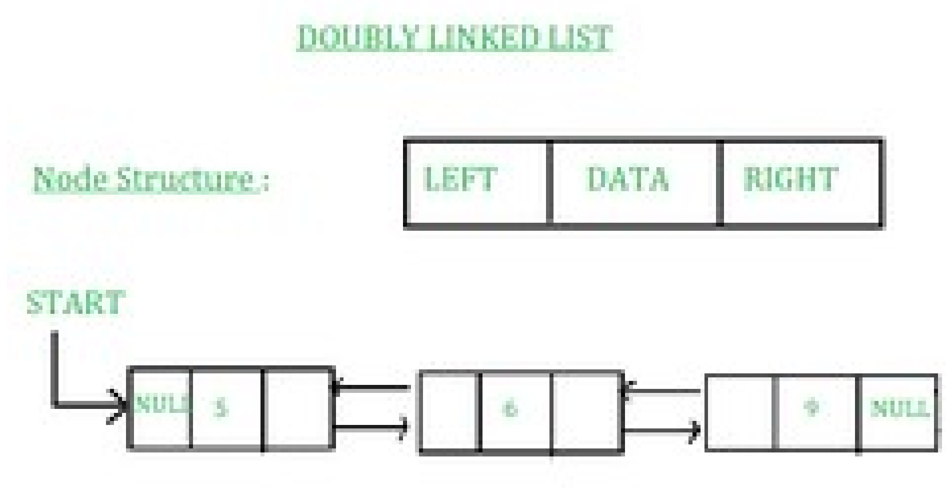
Semester Project

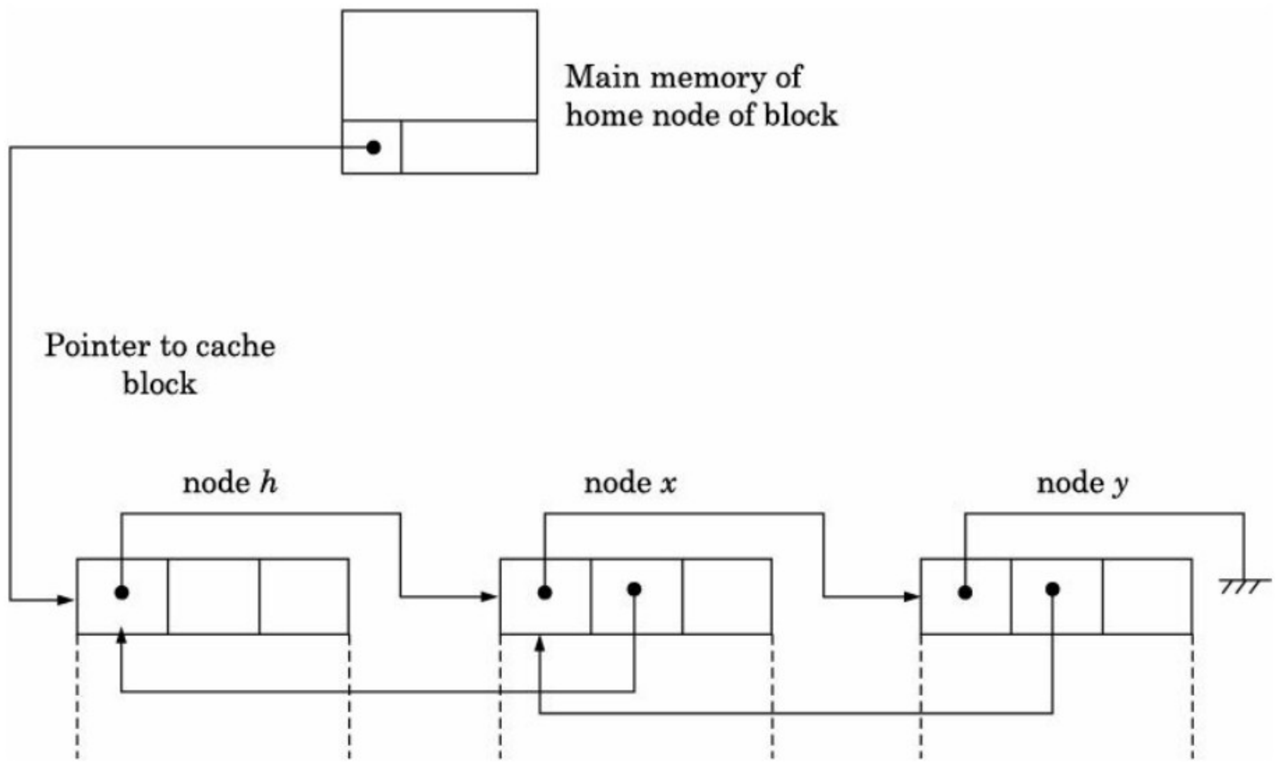
HAMAD SALEEM
F2021376065
MIAN MUHAMMAD BILAL
[F2021408054]
MUHAMMAD TALHA
[F2021408057]

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.

Structures and Initialization

1. **CacheBlock**: This structure represents a cache block, containing the actual data (`data`) and pointers to the previous and next cache blocks (`prev` and `next`). This allows the cache blocks to form a doubly linked list.
2. **DirectoryEntry**: This structure represents a directory entry for a node. It contains a pointer to the head of the linked list of cache blocks (`head`).
3. **HomeNode**: This structure represents the home node, which contains an array of directory entries (`directory`) and the number of nodes in the system (`numNodes`).
4. **createCacheBlock(int data)**: This function creates a new cache block with the specified data, initializes its pointers to NULL, and returns a pointer to the new block.
5. **initializeHomeNode(int numNodes)**: This function initializes the home node with the specified number of nodes. It allocates memory for the home node and its directory entries, sets the `head` pointer of each directory entry to NULL, and returns a pointer to the home node.

Read Procedure

readProcedure(HomeNode homeNode, int nodeIndex, int data):

1. **Check Local Cache**: The function starts by checking if the requested data is already present in the local cache (linked list of cache blocks) for the specified node (`nodeIndex`). It traverses the linked list starting from the head.
2. **Fetch from Home Node**: If the data is not found in the local cache, the function simulates fetching the data from the home node's main memory. It creates a new cache block with the requested data.
3. **Insert at Head**: The new cache block is inserted at the head of the linked list for the specified node. If the list is not empty, the new block's `next` pointer is set to the current head, and the current head's `prev` pointer is updated to point to the new block. The `head` pointer of the directory entry is then updated to point to the new block.

Write Procedure

writeProcedure(HomeNode homeNode, int nodeIndex, int data):

1. **Check Local Cache**: Similar to the read procedure, this function starts by checking if the data to be written is already present in the local cache for the specified node. It traverses the linked list starting from the head.
2. **Update Data**: If the data is found in the local cache, it is simply updated.

3. **Fetch and Update**: If the data is not found, the function simulates fetching the data from the home node's main memory and creating a new cache block with the updated data.
4. **Insert at Head**: The new cache block is inserted at the head of the linked list for the specified node in the same manner as in the read procedure.

Display Cache Blocks

displayCacheBlocks(HomeNode homeNode, int nodeIndex):

- This function prints the data in the cache blocks for a specified node. It traverses the linked list of cache blocks starting from the head and prints the `data` of each block.

Main Function

main():

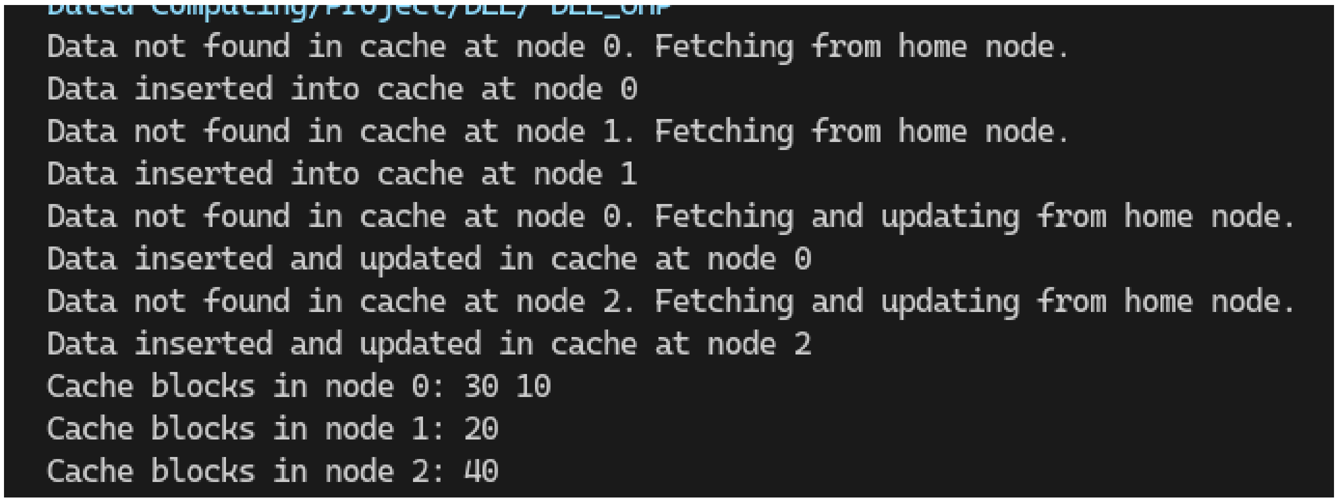
1. **Initialize Home Node**: The home node is initialized with 3 nodes.
2. **Simulate Operations**: The program simulates read and write operations on the cache blocks for different nodes.
3. **Display Cache Blocks**: It prints the cache blocks for each node.
4. **Free Memory**: The dynamically allocated memory for the cache blocks and the home node is freed.

Code:

```
#include <stdio.h>

#include <stdlib.h>

typedef struct CacheBlock {

    int data;

    struct CacheBlock* prev;

    struct CacheBlock* next;

} CacheBlock;

typedef struct {

    CacheBlock* head;

} DirectoryEntry;

typedef struct {

    DirectoryEntry* directory;

    int numNodes;

} HomeNode;
```

```
CacheBlock* createCacheBlock(int data) {
    CacheBlock* newBlock = (CacheBlock*)malloc(sizeof(CacheBlock));
    newBlock->data = data;
    newBlock->prev = NULL;
    newBlock->next = NULL;
    return newBlock;
}
HomeNode* initializeHomeNode(int numNodes) {
    HomeNode* homeNode = (HomeNode*)malloc(sizeof(HomeNode));
    homeNode->directory = (DirectoryEntry*)malloc(numNodes * sizeof(DirectoryEntry));
    for (int i = 0; i < numNodes; i++) {
        homeNode->directory[i].head = NULL;
    }
    homeNode->numNodes = numNodes;
    return homeNode;
}

void readProcedure(HomeNode* homeNode, int nodeIndex, int data) {
    DirectoryEntry* entry = &homeNode->directory[nodeIndex];
    CacheBlock* block = entry->head;
    while (block != NULL) {
        if (block->data == data) {
            printf("Data found in cache at node %d\n", nodeIndex);
            return;
        }
        block = block->next;
    }
  printf("Data not found in cache at node %d. Fetching from home node.\n", nodeIndex);
    CacheBlock* newBlock = createCacheBlock(data);
    if (entry->head != NULL) {
        entry->head->prev = newBlock;
    }
    newBlock->next = entry->head;
    entry->head = newBlock;
    printf("Data inserted into cache at node %d\n", nodeIndex);
}
void writeProcedure(HomeNode* homeNode, int nodeIndex, int data) {
    DirectoryEntry* entry = &homeNode->directory[nodeIndex];
    CacheBlock* block = entry->head;
    while (block != NULL) {
        if (block->data == data) {
            printf("Data found in cache at node %d. Updating.\n", nodeIndex);
            block->data = data;  // Update the data
            return;
        }
        block = block->next;
    }
    printf("Data not found in cache at node %d. Fetching and updating from home node.\n", nodeIndex);
    CacheBlock* newBlock = createCacheBlock(data);
    if (entry->head != NULL) {
        entry->head->prev = newBlock;
    }
    newBlock->next = entry->head;
    entry->head = newBlock;
    printf("Data inserted and updated in cache at node %d\n", nodeIndex);
}
void displayCacheBlocks(HomeNode* homeNode, int nodeIndex) {
    DirectoryEntry* entry = &homeNode->directory[nodeIndex];
    CacheBlock* block = entry->head;
    printf("Cache blocks in node %d: ", nodeIndex);
    while (block != NULL) {
        printf("%d ", block->data);
        block = block->next;
    }
    printf("\n");
}
int main() {
    int numNodes = 3;
    HomeNode* homeNode = initializeHomeNode(numNodes);
    readProcedure(homeNode, 0, 10);
    readProcedure(homeNode, 1, 20);
    writeProcedure(homeNode, 0, 30);
    writeProcedure(homeNode, 2, 40);
    displayCacheBlocks(homeNode, 0);
    displayCacheBlocks(homeNode, 1);
    displayCacheBlocks(homeNode, 2);
    for (int i = 0; i < numNodes; i++) {
        CacheBlock* block = homeNode->directory[i].head;
        while (block != NULL) {
            CacheBlock* temp = block;
            block = block->next;
            free(temp);
        }
    }
    free(homeNode->directory);
    free(homeNode);
  return 0;
}
```

Output:



```
Data not found in cache at node 0. Fetching from home node.
Data inserted into cache at node 0
Data not found in cache at node 1. Fetching from home node.
Data inserted into cache at node 1
Data not found in cache at node 0. Fetching and updating from home node.
Data inserted and updated in cache at node 0
Data not found in cache at node 2. Fetching and updating from home node.
Data inserted and updated in cache at node 2
Cache blocks in node 0: 30 10
Cache blocks in node 1: 20
Cache blocks in node 2: 40
```

MESI implementation

Introduction:

The provided code simulates the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol in a multiprocessor system using OpenMP for parallel processing. The code demonstrates how processors interact with a shared memory and manage their cache states to ensure data consistency.

Explanation:

Initialization

1. **Main Memory and Cache Arrays**:
    - `shared_array[ARRAY_SIZE]`: Represents the main memory initialized to zero.
    - `cache[NUM_PROCS][ARRAY_SIZE]`: Represents the cache for each of the `NUM_PROCS` processors.
    - `cache_state[NUM_PROCS][ARRAY_SIZE]`: Stores the state of each cache line for each processor, initialized to `INVALID`.

Parallel Section

2. **Parallel Processing with OpenMP**:
    - The code uses OpenMP to create `NUM_PROCS` threads, simulating parallel execution.
    - Each thread, identified by `thread_id`, processes a specific portion of the `shared_array`.

Cache Access and MESI States

3. **Cache State Check and Data Handling**:
    - **INVALID State**:
        - If the cache state is `INVALID`, the processor fetches the data from main memory into its cache and changes the state to `EXCLUSIVE`.

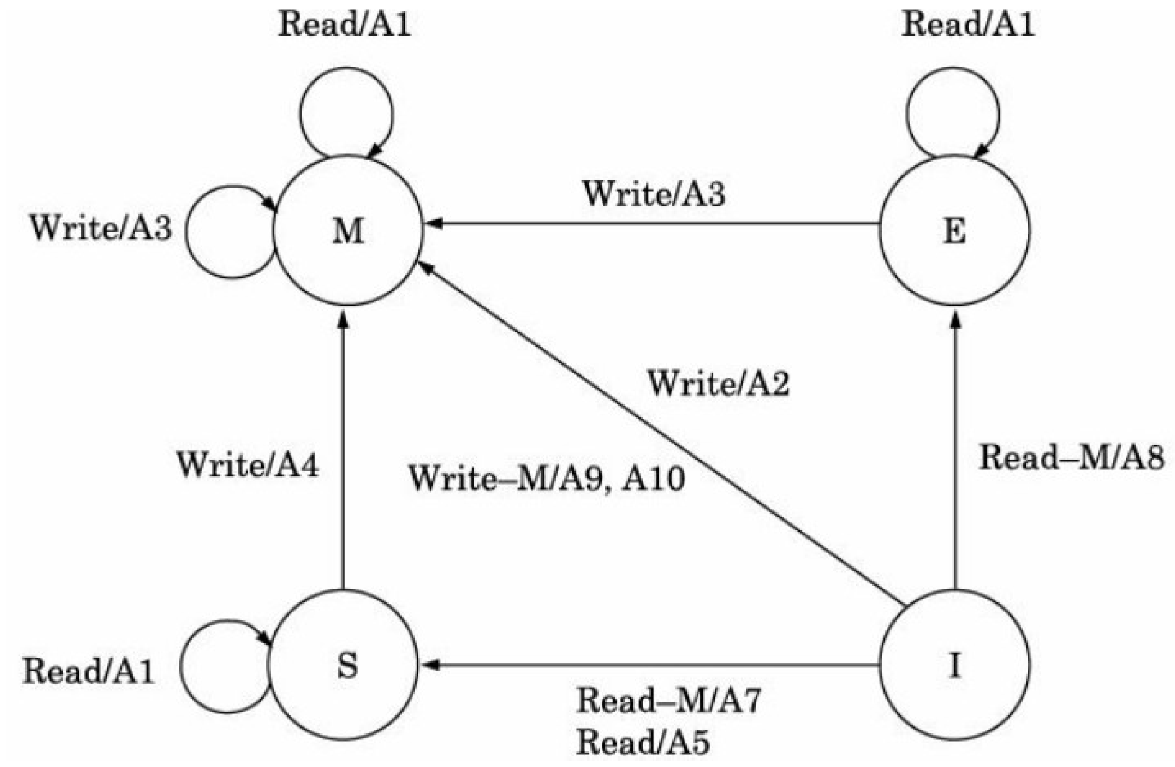
- **SHARED State**:
  - If the cache state is `SHARED`, the processor reads the data from its cache without needing to fetch it from the main memory.
- **Data Modification**:
  - The processor performs a computation (incrementing the data by the thread ID) and updates the cache state to `MODIFIED`.
- **Main Memory Update**:
  - The modified data is written back to the main memory to maintain consistency.

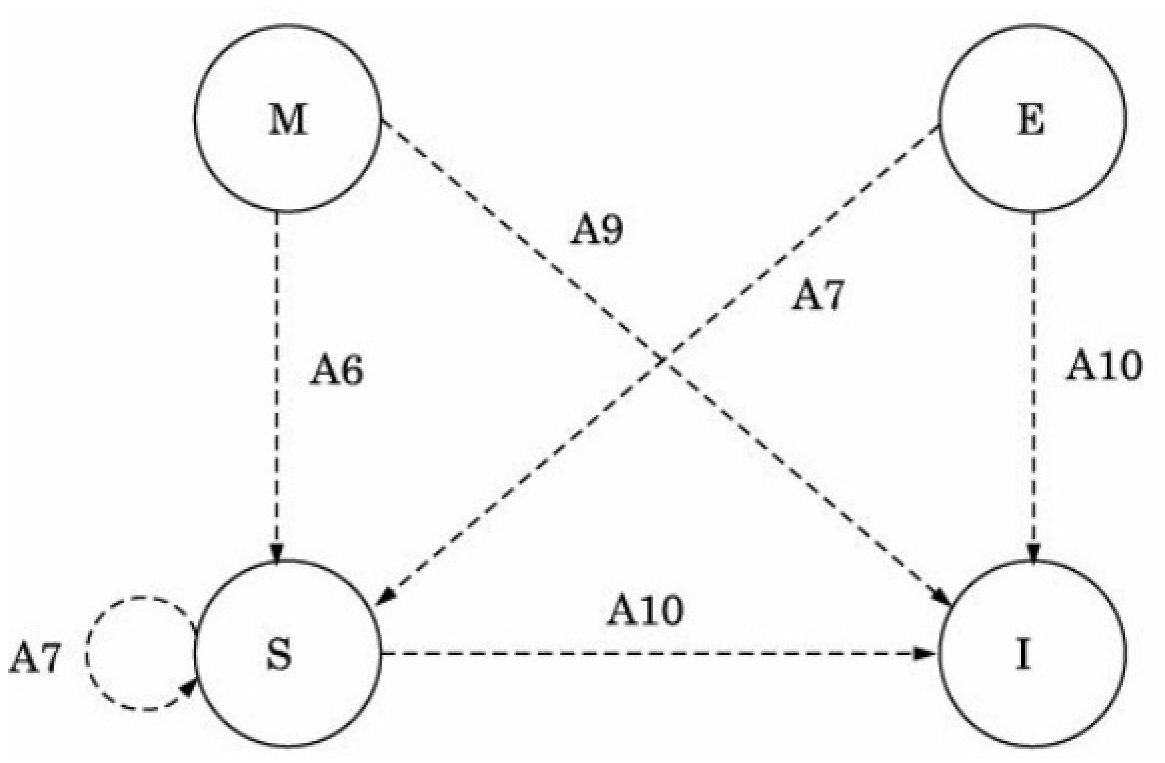
Final Cache States

4. **Output**:
   - The code prints the final state of the cache for each processor, showing how the data and states have changed after the parallel execution.

Summary

- The code simulates a simplified MESI protocol.
- It ensures data consistency across multiple processors by managing cache states.
- The parallel section demonstrates how each processor accesses and modifies data.
- The final cache states are printed to show the outcome of the simulation.

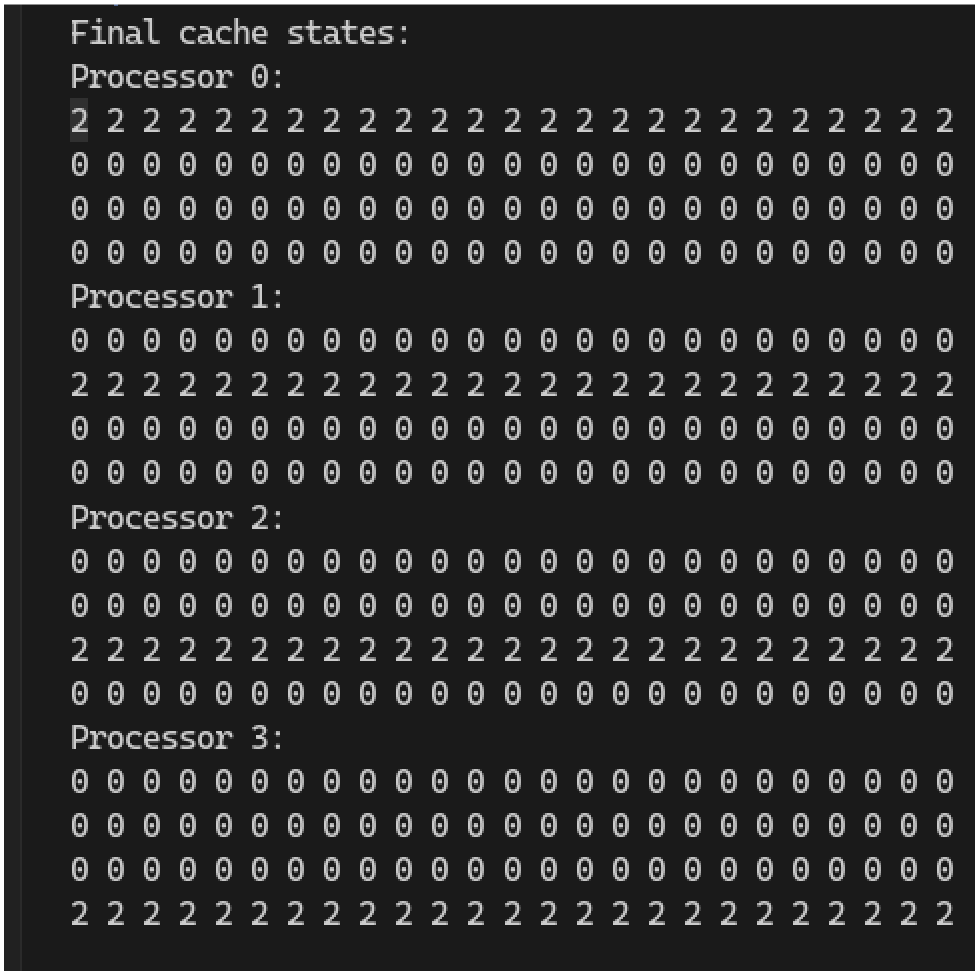




Code:

```
#include <omp.h>

#include <stdio.h>

#define NUM_PROCS 4

#define ARRAY_SIZE 100

#define INVALID 0

#define SHARED 1

#define MODIFIED 2

#define EXCLUSIVE 3

int main()

{

    // Array to be shared among processors

    int shared_array[ARRAY_SIZE] = {0}; // Initialize main memory with
zeros

    int cache[NUM_PROCS][ARRAY_SIZE];
    int cache_state[NUM_PROCS][ARRAY_SIZE]; // Cache line state
    for (int i = 0; i < NUM_PROCS; ++i)
    {
        for (int j = 0; j < ARRAY_SIZE; ++j)
        {
            cache_state[i][j] = INVALID;
        }
    }
#pragma omp parallel num_threads(NUM_PROCS)
    {
        int thread_id = omp_get_thread_num();
        for (int i = thread_id * (ARRAY_SIZE / NUM_PROCS); i < (thread_id
+ 1) * (ARRAY_SIZE / NUM_PROCS); ++i)
        {
            if (cache_state[thread_id][i] == INVALID)
            {
                // Read data from main memory into cache
                cache[thread_id][i] = shared_array[i];
                cache_state[thread_id][i] = EXCLUSIVE; // Change state to
exclusive
            }
            else if (cache_state[thread_id][i] == SHARED)
            {
            }
ID)
            cache[thread_id][i] += thread_id;
            cache_state[thread_id][i] = MODIFIED; // Change state to modified
            shared_array[i] = cache[thread_id][i]; // updating the main memory
        }
    }
printf("Final cache states:\n");
    for (int i = 0; i < NUM_PROCS; ++i)
    {
        printf("Processor %d:\n", i);
        for (int j = 0; j < ARRAY_SIZE; ++j)
        {
            printf("%d ", cache_state[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:



```
Final cache states:
Processor 0:
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Processor 1:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Processor 2:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Processor 3:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```