C Implementation of Maekawa's Algorithm

By Group 5

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 10

typedef struct {
    int id;
    int timestamp;
} Request;

typedef struct {
    int id;
    int quorum[MAX];
    int quorum_size;

    bool voted;             // Has this process granted vote?
    int voted_for;          // Process id it voted for
    bool in_cs;             // Is in critical section?
    int request_ts;         // Timestamp of its request
    int grants_received;    // Count of grants received
} Process;

Process processes[MAX];
int N;

// Simple global logical clock
int logical_clock = 0;

void send_request(int pid);
void send_release(int pid);
void enter_cs(int pid);
void exit_cs(int pid);

// Compare timestamps
bool higher_priority(int ts1, int id1, int ts2, int id2) {
    if (ts1 < ts2) return true;
    if (ts1 == ts2 && id1 < id2) return true;
    return false;
}

// Initialize quorum manually (example for N=4)
void init_quorum() {
```

```
    // Example quorum sets (must intersect)
    processes[0].quorum[0] = 0;
    processes[0].quorum[1] = 1;
    processes[0].quorum_size = 2;

    processes[1].quorum[0] = 1;
    processes[1].quorum[1] = 2;
    processes[1].quorum_size = 2;

    processes[2].quorum[0] = 2;
    processes[2].quorum[1] = 3;
    processes[2].quorum_size = 2;

    processes[3].quorum[0] = 3;
    processes[3].quorum[1] = 0;
    processes[3].quorum_size = 2;
}

// Send REQUEST to quorum members
void send_request(int pid) {
    logical_clock++;
    processes[pid].request_ts = logical_clock;
    processes[pid].grants_received = 0;

    printf("\nProcess %d sending REQUEST (ts=%d)\n", pid, logical_clock);

    for (int i = 0; i < processes[pid].quorum_size; i++) {
        int member = processes[pid].quorum[i];

        if (!processes[member].voted) {
            processes[member].voted = true;
            processes[member].voted_for = pid;
            processes[pid].grants_received++;
            printf("Process %d sends GRANT to %d\n", member, pid);
        } else {
            int current = processes[member].voted_for;

            if (higher_priority(processes[pid].request_ts, pid,
                                processes[current].request_ts, current)) {

                printf("Process %d sends INQUIRE to %d\n", member, current);

                processes[current].grants_received--;
                processes[member].voted_for = pid;
                processes[pid].grants_received++;
                printf("Process %d sends GRANT to %d (after YIELD)\n", member, pid);

            } else {
```

```
                printf("Process %d sends FAILED to %d\n", member, pid);
            }
        }
    }

    if (processes[pid].grants_received == processes[pid].quorum_size) {
        enter_cs(pid);
    }
}

void enter_cs(int pid) {
    processes[pid].in_cs = true;
    printf(">>> Process %d ENTERS Critical Section\n", pid);
}

void exit_cs(int pid) {
    printf("<<< Process %d EXITS Critical Section\n", pid);
    processes[pid].in_cs = false;
    send_release(pid);
}

// Send RELEASE to quorum members
void send_release(int pid) {
    for (int i = 0; i < processes[pid].quorum_size; i++) {
        int member = processes[pid].quorum[i];

        if (processes[member].voted_for == pid) {
            processes[member].voted = false;
            processes[member].voted_for = -1;
            printf("Process %d releases vote from %d\n", pid, member);
        }
    }
}

int main() {

    printf("Enter number of processes (max 4 recommended): ");
    scanf("%d", &N);

    for (int i = 0; i < N; i++) {
        processes[i].id = i;
        processes[i].voted = false;
        processes[i].voted_for = -1;
        processes[i].in_cs = false;
        processes[i].grants_received = 0;
    }

    init_quorum();
```

```
    // Simulate requests
    send_request(0);
    send_request(1);

    exit_cs(0);
    send_request(1);

    return 0;
}
```

OUTPUT: