Distributed Mutual Exclusion

Assignment # 04 (of slate)

By: Aimal Khan

P20-0028

Distributed Mutual Exclusion:

Distributed Mutual Exclusion (DME) is a problem in distributed computing that involves coordinating access to a shared resource among multiple processes. The goal of DME is to ensure that only one process at a time can access the shared resource, while also allowing multiple processes to access the resource over time.

In the DME problem, each process has a critical section of code that needs to be executed atomically (i.e., without interference from other processes). Processes need to communicate with each other to request access to the critical section, and they must wait for permission from other processes before entering the critical section. This coordination must be done in a way that is fair, so that all processes have an equal opportunity to access the critical section.

DME is a difficult problem in distributed computing, because processes may be running on different machines and may not have direct access to shared memory or a clock that can be used to coordinate access. As a result, DME algorithms must use messaging and other techniques to coordinate access to the shared resource.

There are many algorithms for DME, each with different trade-offs in terms of performance, complexity, and fault tolerance. Some popular DME algorithms include the Lamport's Distributed Mutual Exclusion Algorithm, Maekawa's Algorithm, and Suzuki-Kasami Algorithm. These algorithms use various techniques such as message passing, logical

clocks, and voting to ensure that processes can coordinate access to shared resources in a distributed system.

1.Maekawa Algorithm:

The Maekawa algorithm is a distributed mutual exclusion algorithm used in computer science to ensure that multiple processes can access a shared resource without interfering with one another. Here's an implementation of the Maekawa algorithm in C:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define N 5 // Number of processes
#define K 2 // Number of resources each process needs
#define M 10 // Number of requests each process makes

int want[N][K] = {0}; // Set of requests for each process
int allowed[N] = {0}; // Set of processes allowed to enter critical section
int done = 0; // Number of processes that have completed

void* process(void* id_ptr) {
    int id = *(int*) id_ptr;
    int i, j;

    for (i = 0; i < M; i++) {
```

```
    // Make K resource requests
    for (j = 0; j < K; j++) {
        want[id][j] = 1;
    }

    // Wait until process is allowed to enter critical section
    while (!allowed[id]);

    // Enter critical section
    printf("Process %d is in critical section\n", id);
    sleep(1);
    printf("Process %d is leaving critical section\n", id);

    // Clear K resource requests
    for (j = 0; j < K; j++) {
        want[id][j] = 0;
    }

    // Notify other processes that current process has completed
    done++;
    if (done == N) {
        // Reset allowed array
        for (j = 0; j < N; j++) {
            allowed[j] = 0;
        }
        done = 0;
    }
}
```

```
    return NULL;
}

int main() {
    int i, j;
    pthread_t thread[N];
    int id[N];

    // Initialize thread IDs
    for (i = 0; i < N; i++) {
        id[i] = i;
    }

    // Create threads
    for (i = 0; i < N; i++) {
        pthread_create(&thread[i], NULL, process, &id[i]);
    }

    // Run Maekawa algorithm
    while (1) {
        for (i = 0; i < N; i++) {
            int can_enter = 1;
            for (j = 0; j < N; j++) {
                if (i != j && want[j][0] && want[j][1]) {
                    can_enter = 0;
                    break;
                }
            }
```

```
            allowed[i] = can_enter;
        }
        sleep(1);
    }

    // Join threads
    for (i = 0; i < N; i++) {
        pthread_join(thread[i], NULL);
    }

    return 0;
}
```

---

2. Suzuki-Kasami Algorithm:

The Suzuki-Kasami algorithm is another distributed mutual exclusion algorithm used in computer science to ensure that multiple processes can access a shared resource without interfering with one another. Here's an implementation of the Suzuki-Kasami algorithm in C:

Code:

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

```
#include <pthread.h>

#define N 5 // Number of processes
#define M 10 // Number of requests each process makes

int want[N] = {0}; // Set of requests for each process
int reply[N] = {0}; // Set of replies received by each process
int done = 0; // Number of processes that have completed

void* process(void* id_ptr) {
    int id = *(int*) id_ptr;
    int i;

    for (i = 0; i < M; i++) {
        // Make request for critical section
        want[id] = 1;

        // Send request to other processes
        int j;
        for (j = 0; j < N; j++) {
            if (j != id) {
                reply[j] = 0;
                // Send request message to process j
                printf("Process %d sends request to process %d\n", id, j);
            }
        }

        // Wait until all replies have been received
```

```
int count = 0;
while (count < N - 1) {
    int j;
    for (j = 0; j < N; j++) {
        if (j != id && want[j] && !reply[j]) {
            // Process j has not yet replied to request
            break;
        }
    }
    if (j == N) {
        // All processes have replied to request
        break;
    }
    // Wait for 1 millisecond
    usleep(1000);
}

if (count == N - 1) {
    // All replies have been received
    // Enter critical section
    printf("Process %d is in critical section\n", id);
    sleep(1);
    printf("Process %d is leaving critical section\n", id);
    // Clear request for critical section
    want[id] = 0;
    // Send reply messages to all waiting processes
    int j;
    for (j = 0; j < N; j++) {
```

```
            if (j != id && reply[j]) {
                reply[j] = 0;
                // Send reply message to process j
                printf("Process %d sends reply to process %d\n", id, j);
            }
        }
    } else {
        // One or more processes have not yet replied to request
        // Wait for 1 millisecond and try again
        usleep(1000);
    }
}

// Notify other processes that current process has completed
done++;
if (done == N) {
    // Reset reply array
    int j;
    for (j = 0; j < N; j++) {
        reply[j] = 0;
    }
    done = 0;
}

return NULL;
}

int main() {
```

```
int i;
pthread_t thread[N];
int id[N];

// Initialize thread IDs
for (i = 0; i < N; i++) {
    id[i] = i;
}

// Create threads
for (i = 0; i < N; i++) {
    pthread_create(&thread[i], NULL, process, &id[i]);
}

// Join threads
for (i = 0; i < N; i++) {
    pthread_join(thread[i], NULL);
}

return 0;
```