BHARATIYA VIDYA BHAVANS
SARDAR PATEL INSTITUTE OF TECHNOLOGY
MUNSHI NAGAR, ANDHERI (WEST), MUMBAI - 400 058
AUTONOMOUS INSTITUTE AFFILIATED TO MUMBAI UNIVERSITY

Adwait Purao- 2021300101

TE Comps B - Batch B

DISTRIBUTED COMPUTING
EXPERIMENT 7

Aim:

Implementation of Mutual Exclusion Algorithm.

Theory:

Mutual Exclusion:
Mutual exclusion is a critical property in concurrency control, addressing race conditions to ensure the orderly execution of processes.
It mandates that a process cannot access its critical section while another concurrent process is actively present or executing in its critical section.
This requirement dictates that only one process is allowed to execute the critical section at any given instance.

Requirements of Mutual Exclusion Algorithm:

- **No Deadlock:** Ensures that sites do not endlessly wait for a message that will never arrive.
- **No Starvation:** Guarantees that every site seeking to execute the critical section has the opportunity to do so within a finite time, preventing indefinite waiting.

- **Fairness:** Ensures that each site receives a fair chance to execute the critical section, with requests processed in the order of their arrival.
- **Fault Tolerance:** Recognizes failures independently and allows the system to continue functioning without disruption in the event of a failure.

Solution to Distributed Mutual Exclusion:

In distributed systems, shared variables and local kernels are impractical for implementing mutual exclusion. Instead, message passing emerges as a viable solution.
Three distinct approaches based on message passing are employed to implement mutual exclusion in distributed systems:

1. **Token-Based Algorithm:**
    - Involves sharing a unique token among all sites.
    - A site with the token gains access to its critical section.
    - Utilizes sequence numbers to order critical section requests.
    - Ensures mutual exclusion with a unique token.
    - Example: Suzuki–Kasami Algorithm.

2. **Non-Token Based Approach:**
    - Requires communication between sites to determine which sites should execute the critical section next.
    - Utilizes timestamps to order critical section requests.
    - Resolves conflicts using timestamps.
    - Maintains a logical clock updated according to Lamport's scheme.
    - Example: Ricart–Agrawala Algorithm.

3. **Quorum-Based Approach:**
    - Instead of seeking permission from all sites, each site requests permission from a subset called a quorum.
    - Any two quorums contain a common site responsible for ensuring mutual exclusion. Example: Maekawa's Algorithm.

Challenges in Distributed Mutual Exclusion:

1. **Communication Delays:**
    - Unlike in a centralized system, communication delays between nodes can significantly impact the efficiency of distributed mutual exclusion algorithms.
    - Delays in message transmission and processing times introduce complexities that must be considered in designing DME protocols.

2. **Node Failures:**
    - Distributed systems are prone to node failures, making it essential to design algorithms that can handle node crashes and recover gracefully.
    - Ensuring mutual exclusion in the presence of failures adds an extra layer of complexity to DME protocols.

3. **Scalability:**
    - Distributed systems often scale dynamically, with nodes joining or leaving the network.
    - Scalability is a crucial consideration in DME, as the algorithm must adapt to changes in the number of participating nodes while maintaining correctness and efficiency.

Ricart-Agrawala Algorithm:

The Ricart-Agrawala Algorithm is a permission-based solution for mutual exclusion in a distributed system, proposed by Glenn Ricart and Ashok Agrawala.

Working of the Algorithm:

1. **Requesting Permission:**
   When a process wants to enter a critical section, it sends a request to all other processes in the system. The request contains the process's ID and the current timestamp.

2. **Granting Permission:**
   Upon receiving a request, a process can do one of three things:
    - If it is not interested in the critical section, it sends a reply granting permission.
    - If it is in the critical section, it does not reply and instead queues the request.
    - If it has already sent a request and is waiting to enter the critical section, it compares the timestamp of its request with the incoming request. If its own request is older (i.e., has a smaller timestamp), it queues the incoming request. Otherwise, it sends a reply granting permission and queues its own request.

3. **Entering the Critical Section:**
   A process enters the critical section only after it has received a reply from every other process in the system.

4. **Exiting the Critical Section:**
   After exiting the critical section, the process sends a reply to all the requests in its queue.

Properties of the Algorithm:

- **Mutual Exclusion:** At any given time, only one process can execute the critical section.
- **Freedom from Deadlock:** Since every request is eventually replied to, there are no circular waits, and hence, no deadlocks.
- **Freedom from Starvation:** Every request is eventually granted permission to enter the critical section.
- **Fairness:** The algorithm is fair in the sense that it gives priority to the process that made the request first.

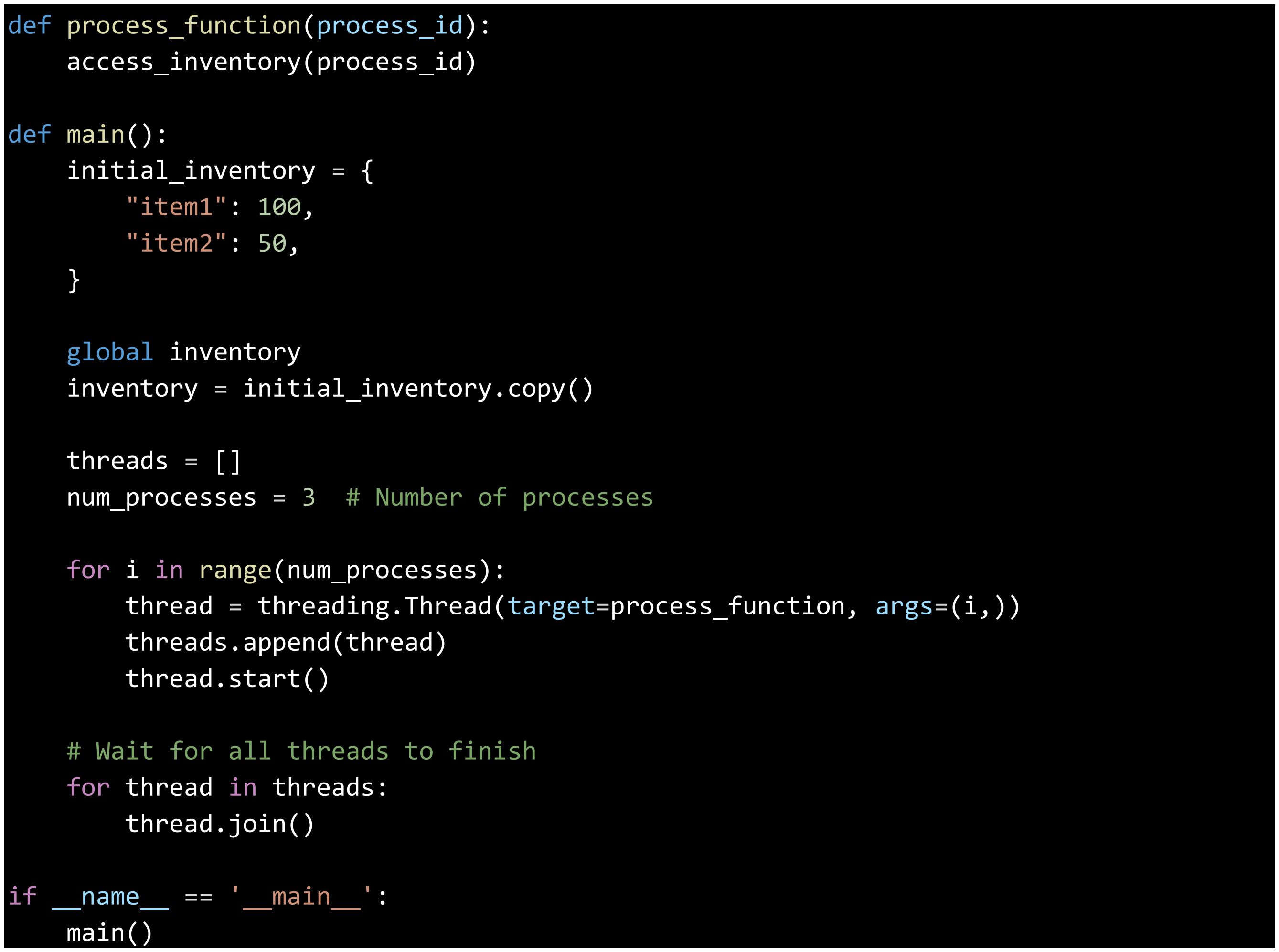
Limitations of the Algorithm:

- **Communication Overhead:** The algorithm requires 2(N-1) messages per entry and exit into the critical section, where N is the number of processes. This can lead to high communication overhead in systems with a large number of processes.
- **Single Point of Failure:** If a process fails, it can cause other processes to wait indefinitely.
- **Lack of Concurrency:** The algorithm does not allow for concurrency, i.e., two non-conflicting processes cannot be in their critical sections at the same time.

Despite these limitations, the Ricart-Agrawala Algorithm is widely used due to its simplicity and effectiveness in ensuring mutual exclusion in distributed systems.

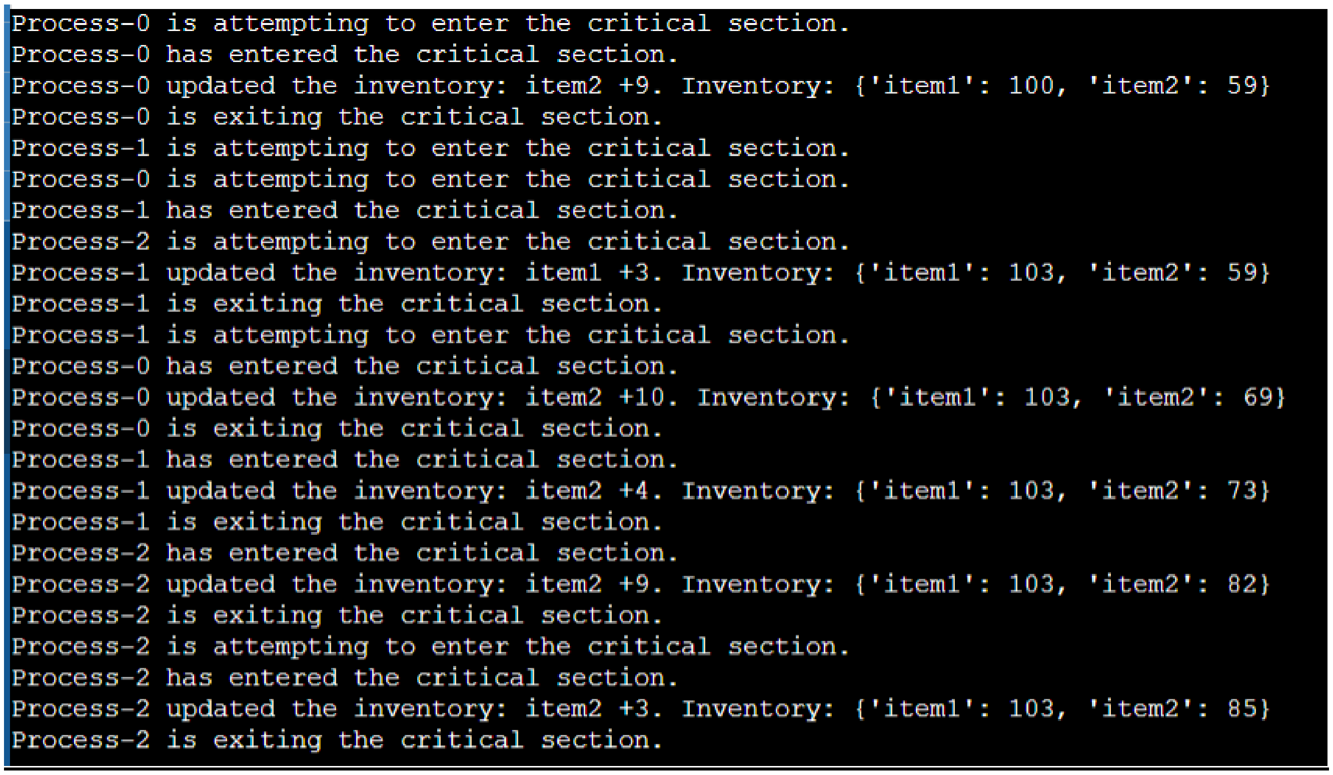
Code:

```python
import threading
import time
import random

# Shared variables
inventory = {}
iterations = 2
lock = threading.Lock()  # Lock for mutual exclusion

# Ricart-Agrawala variables
request_queue = []  # Queue to hold the requests for the critical section
in_critical_section = False  # Flag to indicate if a process is in the critical
section
highest_sequence_number = 0  # The highest sequence number seen so far

def access_inventory(process_id):
    global inventory, lock, request_queue, in_critical_section,
highest_sequence_number

    for _ in range(iterations):
        print(f"Process-{process_id} is attempting to enter the critical section.")
        sequence_number = highest_sequence_number + 1
        request_queue.append((sequence_number, process_id))
        request_queue.sort()  # Sort the queue based on sequence numbers

        while request_queue[0][1] != process_id or in_critical_section:
            pass

        in_critical_section = True

        with lock:
            print(f"Process-{process_id} has entered the critical section.")
            item = random.choice(list(inventory.keys()))  # Random item selection
            quantity = random.randint(1, 10)  # Random quantity

            if item in inventory:
                inventory[item] += quantity
            else:
                inventory[item] = quantity

            print(f"Process-{process_id} updated the inventory: {item} +{quantity}.
Inventory: {inventory}")
            print(f"Process-{process_id} is exiting the critical section.")

        in_critical_section = False
        request_queue.remove((sequence_number, process_id))  # Remove this process's
request from the queue
```



```python
def process_function(process_id):
    access_inventory(process_id)

def main():
    initial_inventory = {
        "item1": 100,
        "item2": 50,
    }

    global inventory
    inventory = initial_inventory.copy()

    threads = []
    num_processes = 3  # Number of processes

    for i in range(num_processes):
        thread = threading.Thread(target=process_function, args=(i,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to finish
    for thread in threads:
        thread.join()

if __name__ == '__main__':
    main()
```

Output:



```
Process-0 is attempting to enter the critical section.
Process-0 has entered the critical section.
Process-0 updated the inventory: item2 +9. Inventory: {'item1': 100, 'item2': 59}
Process-0 is exiting the critical section.
Process-1 is attempting to enter the critical section.
Process-0 is attempting to enter the critical section.
Process-1 has entered the critical section.
Process-2 is attempting to enter the critical section.
Process-1 updated the inventory: item1 +3. Inventory: {'item1': 103, 'item2': 59}
Process-1 is exiting the critical section.
Process-1 is attempting to enter the critical section.
Process-0 has entered the critical section.
Process-0 updated the inventory: item2 +10. Inventory: {'item1': 103, 'item2': 69}
Process-0 is exiting the critical section.
Process-1 has entered the critical section.
Process-1 updated the inventory: item2 +4. Inventory: {'item1': 103, 'item2': 73}
Process-1 is exiting the critical section.
Process-2 has entered the critical section.
Process-2 updated the inventory: item2 +9. Inventory: {'item1': 103, 'item2': 82}
Process-2 is exiting the critical section.
Process-2 is attempting to enter the critical section.
Process-2 has entered the critical section.
Process-2 updated the inventory: item2 +3. Inventory: {'item1': 103, 'item2': 85}
Process-2 is exiting the critical section.
```

<u>**Conclusion:**</u>

In summary, we successfully illustrated the practical application of mutual exclusion in distributed systems, specifically by implementing the Ricart-Agrawala Algorithm. This algorithm, employing a permission-based approach, effectively guarantees that only one process can access its critical section at any given time, preventing race conditions and ensuring system correctness.
However, it's crucial to acknowledge that while the Ricart-Agrawala Algorithm is effective, it comes with challenges such as communication overhead and a single point of failure. Therefore, when choosing a mutual exclusion algorithm, it's essential to consider the specific requirements and constraints of the system. In conclusion, we comprehended the pivotal role of mutual exclusion in upholding the integrity and reliability of distributed systems.