1 Shared Memory Model

Suppose there are \( n \) processors \( P_1, P_2, \ldots, P_n \) and \( m \) shared registers \( R_1, R_2, \ldots, R_m \). An execution is modeled by configurations and events. Here the configuration is a vector consisting of \((q_1, q_2, \ldots, q_n, r_1, r_2, \ldots, r_m)\) where \( q_i \) is the state of processor \( P_i \) and \( r_j \) is the value in shared memory register \( R_j \).

**Definition 1** An initial configuration is a configuration in which every processor is in initial state and every register has some fixed initial value.

In any algorithm, we have to initialize the state of processors and the values of shared memory registers.

**Definition 2** Event: An event is a single data operation (Example: Read/Write) by a processor modifying one or more registers.

**Definition 3** Execution: An execution fragment is an alternating sequence of configurations and events: \( C_0 e_0 C_1 e_1 \ldots C_k e_k \ldots \) where for each \( i \), \( C_{i+1} \) is the result of applying \( e_i \) to \( C_i \) (that is, \( e(C_i) = C_{i+1} \)). An execution is an execution fragment starting at an initial configuration.

**Definition 4** Schedule: A schedule \( \sigma \) is defined as a sequence of processor IDs.

**Definition 5** Given a schedule we define \((C, \sigma)\) to be the execution fragment starting at configuration \( C \) and the result of applying the events in \( \sigma \). If \( \sigma \) is finite \( \sigma(C) \) denotes the final configuration after applying \( \sigma \) to \( C \).

Commutativity for operations: When two processors perform read or when two processors are working on different registers then Commutativity holds. For example: P (read X), Q (read X) will lead to same final configuration as Q (read X), P (read X). Commutativity can hold for writes if the processors write to separate registers. Example: P (write X), Q (write Y) will lead to same final configuration as Q (write Y), P (write X). Commutativity can hold for a combination of reads and writes if the two processors have not written to the same register at any time during either’s execution and, either has not read from a register the other will write or already wrote. Another assumption for commutativity to work is: reads and writes by the same processor can happen in any order. Consider the schedule: P (write X), Q (read Y), P (read Y). This will be equivalent to Q (read Y), P (read Y), P (write X).
Definition 6 A configuration \( C' \) is said to be reachable from \( C \) if there exists a \( \sigma \) such that \( C' = \sigma(C) \).

Definition 7 A configuration is said to be reachable if it is reachable from some initial configuration.

1.1 Mutual Exclusion in Shared Memory Model

To achieve mutual exclusion, there are 4 stages a processor cycles through during the course of its execution:

Entry \( \rightarrow \) Critical Section (CS) \( \rightarrow \) Exit \( \rightarrow \) Remainder \( \rightarrow \) (back to) Entry \( \rightarrow \) ...

Entry is the code executed when trying to gain access to critical section. Entry code is a part of the mutual exclusion algorithm.

Critical is the code in Critical Section (CS) which is a part of the processor’s code.

Exit is the code executed after exiting critical section. It is also a part of the mutual exclusion algorithm.

Remainder is the processor’s own code other than critical section.

The following are the properties of an algorithm which solves the mutual exclusion problem:

Definition 8 Admissibility: We say that an execution \( \alpha \) is admissible if for every processor \( P \) and configuration \( C_i \), if \( P \) is in Entry, CS, or Exit section in \( C_i \) then there exists some \( j \geq i \) such that \( e_j \) is an event by \( P \). (means, \( P \) should not get stuck forever in any of the three sections)

Definition 9 Safety property for Mutual Exclusion: In every execution, in every configuration \( C_i \) of that execution, at most one processor is in CS.

Definition 10 Liveness: If a processor enters the Entry section, it will eventually reach the CS.

Definition 11 No Deadlock: In every admissible execution for every configuration \( C_i \), if processor \( P \) is in entry section in \( C_i \) then there exists some processor \( Q \) which will be in critical section in \( C_j \) for \( j \geq i \).

Definition 12 No Starvation: In every admissible execution, for every configuration \( C_i \), if \( P \) is in Entry in \( C_i \) then for some \( j \geq i \), \( P \) will be in CS in \( C_j \).
In the following section, we shall discuss an algorithm that is used to solve the mutual exclusion problem in Distributed Systems.

**Lamport’s Bakery Algorithm**

This algorithm is used to achieve mutual exclusion in distributed systems. Consider a distributed system with \( n \) processors: \( P_1, P_2, ..., P_n \). There are two shared arrays:

- **NUM**: An integer array of size \( n \) where \( i \)th entry contains the waiting ticket of \( P_i \).
- **CHOOSE**: A boolean array of size \( n \) where \( i \)th entry is 1 (meaning TRUE) if \( P_i \) is in the process of choosing a ticket and 0 (meaning FALSE) otherwise.

Then, the following algorithm will be used by any processor \( P_i \) in order to achieve mutual exclusion:

1. \(< Entry >\) \ CHOOSE\[i\]=1
2. \( \text{NUM}[i] = \text{MAX}(\text{NUM}[1], \text{NUM}[2], ..., \text{NUM}[n]) + 1 \)
3. \( \text{CHOOSE}[i] = 0 \)
4. for \( j = 1 \) to \( n \) do
5. begin
6. wait until \( \text{CHOOSE}[j]=0 \)
7. wait until \( \text{NUM}[j]=0 \) or \( (\text{NUM}[j], \text{ID}_j) \geq (\text{NUM}[i], \text{ID}_i) \)
8. end
9. \(< CS >\)
10. \(< Exit >\) \( \text{NUM}[i] = 0 \)
11. \(< REM >\)

Note: The tuples in step 7 are compared using Lexicographical order. \( ID_i \) is the identifier of processor. Example: ID of \( P_1 \) is 1.

Suppose the algorithm did not include choose array and uses only tickets then it will be possible for two processes to enter critical situation simultaneously. Consider the following scenario:

In the following example both \( P_1 \) and \( P_0 \) enter critical section at same time. Using choose array would have prevented \( P_1 \) from reading \( \text{NUM}[0] \) as 0 which would in turn prevent it from entering critical section.
Lemma: If processor $P_i$ is in the CS then for all $j \neq i$, either:

1. $\text{NUM}[j] = 0$ or
2. $(\text{NUM}[j], ID_j) \geq (\text{NUM}[i], ID_i)$

Proof: Let $P_i$ exit the loop at time $t$. Since $P_i$ has to pass the second wait statement (line 7) in each iteration in order to exit the loop, the statement of Lemma holds at time $t$ (moment where $P_i$ exited the loop). After this point of time, the value of $\text{NUM}[i]$ remains constant while $P_i$ is in Critical Section. So, we need to consider only changes in $\text{NUM}[j]$ from $t$ to the time $t'$ when $P_i$ exits the CS. Given any processor $P_j$ where $i \neq j$, it can fall into one of the two scenarios:

1. Case 1: $P_j$ has not made any attempt to enter critical section from $t$ to $t'$.
   So, $\text{NUM}[j] = 0$

2. Case 2 (a): $P_j$ got its ticket before $t$ at, say, $w$. But $P_i$ got its ticket before $w$ so that $(\text{NUM}[j], ID_j) \geq (\text{NUM}[i], ID_i)$. Since neither $\text{NUM}[i]$ and $\text{NUM}[j]$ change between $t$ and $t'$ the statement of lemma holds in this case. Note that even if $P_j$ and $P_i$ have the same ticket (which can happen in rare cases), only one of them will be able to exit the loop. Since $P_i$ was able to exit the loop that means, $ID_j > ID_i$ and processor ID’s remain constant all the time. So, the condition holds in this case also from $t$ to $t'$.

Case 2 (b): $P_j$ got its ticket after $t$ at, say, $s$. But $\text{NUM}[j]$ will be greater than $\text{NUM}[i]$ because $\text{NUM}[j]$ will be set to $\text{MAX}(\text{NUM}[1], \text{NUM}[2], ..., \text{NUM}[n]) + 1$ and $\text{MAX}(\text{NUM}[1], \text{NUM}[2], ..., \text{NUM}[n]) \geq \text{NUM}[i]$. So, in this case also the lemma is true because from $t$ to $s$ $\text{NUM}[j] = 0$ and from $s$ to $t'$ $(\text{NUM}[j], ID_j) \geq (\text{NUM}[i], ID_i)$. $\text{NUM}[j]$ will not change in the interval between $s$ and $t'$.

Lemma: If $P_i$ is in Critical Section then $\text{NUM}[i] \neq 0$

Theorem 1 Lamport’s Bakery Algorithm provides mutual exclusion.

Proof: (Proof is by contradiction) Suppose two processes $P_i$ and $P_j$ are in critical section at the same time. By Lemma 2 $\text{NUM}[i] \neq 0$ and $\text{NUM}[j] \neq 0$. But by Lemma 1 it follows that
(NUM[j], ID_j) ≥ (NUM[i], ID_i) and (NUM[i], ID_i) ≥ (NUM[j], ID_j). This leads to a contradiction. So, no two processes can be in CS at same time.