Definitions for a Shared-Memory System

- $n$ processes $p_0, p_1, ..., p_{n-1}$ which are modeled as state-machines (deterministic for now). Each process can have a local memory (decoded into the state of the state-machine).

- $m$ shared registers $R_0, ..., R_{m-1}$, registers can be of different types, e.g., R/W, RMW, CAS and more.

Configuration A configuration is the state of the system at a given time: $C = <q_0, ..., q_{n-1}, r_0, ..., r_{m-1}>$ where $q_i$ is the local state of process $p_i$, and $r_j$ is the value of register $R_j$. The part $< r_0, ..., r_{m-1} >$ is denoted $\text{mem}(C)$.

Initial configuration The system has an initial configuration in which each process is in its initial state and each registers has an initial value.

Event An event is an atomic step by one process $p_i$:

1. A register is chosen according to the local state of $p_i$
2. One operation is invoked on this register
3. $p_i$ changes its local state according to the returned value (any local computation is considered part of the local transition function).

Execution segment An execution segment is a sequence $C_0, \phi_1, C_1, \phi_2, C_2, ...$ so that if the event $\phi_k$ occurs from configuration $C_{k-1}$ then the new configuration is $C_k$. From each configuration, every process has one step that it can perform.

Admissible execution An admissible execution is an infinite execution segment where every process performs in infinite number of steps.

Schedule A schedule is a sequence of process identifiers in the order that they will take steps.

Given an initial configuration $C$ and a schedule $\sigma$, there is one execution defined, denoted $\text{exec}(C, \sigma)$.
Mutual Exclusion

The problem of mutual exclusion is writing two pieces of code call Entry and Exit such that given a code for the Critical Section and the Remainder, no two processes can execute the critical section at the same time. We also have liveness conditions which will be formalized next. We assume that the critical section and the remainder do not use any variables used by the entry and exit codes. We also require that the critical section is finite, i.e., a process that is in the critical section reaches the exit code after a finite number of steps.

Formal requirements of a mutual exclusion algorithm:

**Mutual Exclusion** In every configuration of every execution there is at most one process in the critical section.

This is a safety property.

**No-deadlock** In every admissible execution, if a process is in the entry in code configuration, then some process is in the critical section at some later configuration (perhaps a different process).

**No-Starvation** (No-Lockout) In every admissible execution, if a process is in the entry in some configuration, then this process is in the critical section at some later configuration.

**Unobstructed-Exit** In every admissible execution, if a process is in the exit in some configuration, then this process is in the remainder in some later configuration.

These are liveness properties.
Bakery Algorithm

Algorithm 10 The bakery algorithm: code for processor $p_i$, $0 \leq i \leq n - 1$.

Initially $Number[i] = 0$ and $Choosing[i] = false$, for $i, 0 \leq i \leq n - 1$

<Entry>
1: $Choosing[i] := true$
2: $Number[i] := max(Number[0], \ldots, Number[n-1]) + 1$
3: $Choosing[i] := false$
4: for $j := 0$ to $n - 1 (\neq i)$ do
5: wait until $Choosing[j] = false$
6: wait until $Number[j] = 0$ or $(Number[j], j) > (Number[i], i)$
(Critical Section)
(Exit):
7: $Number[i] := 0$
(Remainder)

Consider an execution $\alpha$.

Lemma 1 In every configuration $C$ of $\alpha$, if $p_i$ is in the critical section, and there is a $k \neq i$ for which $Number[k] \neq 0$, then $(Number[k], k) > (Number[i], i)$.

Lemma 2 If $p_i$ is in the critical section then $Number[i] > 0$ (easy proof.)

Theorem 3 The bakery algorithm satisfies mutual exclusion and no-starvation.

Proof: Mutual exclusion follows from Lemmas 1 and 2. No-starvation is obtained by showing that every process that enters the critical section before $p_i$ and then returns to the entry, gets a larger number than $p_i$. 

3