Critical section protocols for multiple processes

**Batch:** B.Tech III year

**Instructor:** Rahul Muthu

DA-IICT
The algorithm is called *bakery algorithm* and is like a token system, used in large establishments like banks and railway reservation counters.
The algorithm is called *bakery algorithm* and is like a token system, used in large establishments like banks and railway reservation counters.

The processes are denoted by $P_0, \ldots, P_{n-1}$. 
The algorithm is called *bakery algorithm* and is like a token system, used in large establishments like banks and railway reservation counters.

The processes are denoted by $P_0, \ldots, P_{n-1}$.

Each process upon entering the entry section, chooses a *number*, which is one more than the highest number allotted to other processes.
Each process has an associated boolean variable called *choosing* which indicates that it is in the entry section and has requested a *number* and is in the process of being allotted one.
Each process has an associated boolean variable called *choosing* which indicates that it is in the entry section and has requested a *number* and is in the process of being allotted one.

A process upon reaching the exit section resets its *number* to 0.
Each process has an associated boolean variable called `choosing` which indicates that it is in the entry section and has requested a `number` and is in the process of being allotted one.

A process upon reaching the exit section resets its `number` to 0.

`number` and `choosing` of the set of processes are stored in two arrays of length \( n \) each.
Each process has an associated boolean variable called *choosing* which indicates that it is in the entry section and has requested a *number* and is in the process of being allotted one.

A process upon reaching the exit section resets its *number* to 0.

*number* and *choosing* of the set of processes are stored in two arrays of length $n$ each.

Thus the number of $P_i$ is stored in array position $number[i]$, and the boolean variable indicating whether it is waiting for allotment of a number is stored in array position $choosing[i]$. 
Each process has an associated boolean variable called `choosing` which indicates that it is in the entry section and has requested a `number` and is in the process of being allotted one.

A process upon reaching the exit section resets its `number` to 0.

`number` and `choosing` of the set of processes are stored in two arrays of length $n$ each.

Thus the number of $P_i$ is stored in array position `number[i]`, and the boolean variable indicating whether it is waiting for allotment of a number is stored in array position `choosing[i]`.

The `number` array is initialised to 0 in all positions, and the `choosing` array is initialised to `false` in all positions.
Among the waiting processes, one with the lowest *number* is selected to execute its critical section.
Among the waiting processes, one with the lowest *number* is selected to execute its critical section.

Due to concurrent execution of the various processes in a multiprogramming environment, multiple processes may get the same number.
Selecting the next process

- Among the waiting processes, one with the lowest *number* is selected to execute its critical section.
- Due to concurrent execution of the various processes in a multiprogramming environment, multiple processes may get the same number.
- If this situation happens then the tie is broken using *process IDs* among processes with the lowest *number*. Process IDs being unique and from a totally ordered set, determine the order exactly.
Among the waiting processes, one with the lowest *number* is selected to execute its critical section.

Due to concurrent execution of the various processes in a multiprogramming environment, multiple processes may get the same number.

If this situation happens then the tie is broken using *process IDs* among processes with the lowest *number*. Process IDs being unique and from a totally ordered set, determine the order exactly.

A process which enters the exit section sets its *number* to 0, and only processes with nonzero values of *number* are considered for execution of critical section.
Code for Bakery Algorithm

repeat
  choosing\[i\] ← true;
  number\[i\] ← max(number[0], \ldots, number[n - 1]) + 1;
  choosing\[i\] ← false;
  for \(j \leftarrow 0 \text{ to } n - 1;\)
    do begin
      while choosing\[j\] do no-op;
      while number\[j\] \neq 0 and
      (number\[j\], j) < (number\[i\], i) do no-op;
    end;
    CRITICAL SECTION
    number\[i\] ← 0;
  REMAINDER SECTION
  until false;
The value of *number* allotted to a process does not change until it is reset to 0 in the exit section, or its coming to the entry section a second time.
Proof that the algorithm works: mutual exclusion

1. The value of *number* allotted to a process does not change until it is reset to 0 in the exit section, or its coming to the entry section a second time.

2. Also, a process $P_i$ in its entry section always waits for the *number* of another process $P_j$ to be set, unless it is 0, before checking whether it has a higher priority to execute its critical section.
Proof that the algorithm works: mutual exclusion

1. The value of *number* allotted to a process does not change until it is reset to 0 in the exit section, or its coming to the entry section a second time.

2. Also, a process $P_i$ in its entry section always waits for the *number* of another process $P_j$ to be set, unless it is 0, before checking whether it has a higher priority to execute its critical section.

3. This is done by making the comparison between ($number[i], i$) and ($number[j], j$).
1. The value of *number* allotted to a process does not change until it is reset to 0 in the exit section, or its coming to the entry section a second time.

2. Also, a process $P_i$ in its entry section always waits for the *number* of another process $P_j$ to be set, unless it is 0, before checking whether it has a higher priority to execute its critical section.

3. This is done by making the comparison between ($number[i], i$) and ($number[j], j$).

4. Due to the invariance of these numbers in one iteration of the entire code, two processes can never be in their critical sections at the same time. Thus mutual exclusion is ensured.
Proof continued: progress and bounded waiting

The progress condition is always maintained, because the terms we compare to obtain the next process are from a totally ordered set of distinct elements, which always has a unique minimum.
The progress condition is always maintained, because the terms we compare to obtain the next process are from a totally ordered set of distinct elements, which always has a unique minimum.

Also, a process never waits for more than \( n - 1 \) other processes to execute their critical sections, between its requesting and executing its own critical section. This ensures bounded waiting.