Summary

Introduction:

There are many examples of different algorithms that appear to be so similar that it makes sense to view them as variants
of each other and to classify them as a family of algorithms. There are many examples of different algorithms that appear to be so similar that it makes sense to view them as variants
of each other and to classify them as a family of algorithms.

Bakery Algorithms:

The Bakery algorithm is a critical section algorithm for N ≥ 2 processes, introduced by Lamport in 1974 [12]. It is an intriguing algorithm: simple and short enough to be grasped in a glance, and yet subtle and involving important aspects of concurrency. The Bakery algorithm is a critical section algorithm for N ≥ 2 processes, introduced by Lamport in 1974 . It is an intriguing algorithm: simple and short enough to be grasped in a glance, and yet subtle and involving important aspects of concurrency. . The **algorithm** is not a baker's recipe, but a method to serve the costumers: upon entry a customer takes a number from a dispenser machine (which dispenses its numbers in increasing order), and is then served after those clients with smaller numbers have been served. We note that this practical ticket dispenser algorithm has only a limited value as the intuitive paradigm for Lamport's algorithm, because accesses to the dispenser need some mutual-exclusion mechanism whereas the Bakery algorithm does not rely on any assumed mutual exclusion, and its registers need not even be atomic. There are many algorithm which base on shared memory mode communication mode and message passing mode. One of them called mimic algorithm by used the shared memory mode communication within message passing mode and read write operation. .They Three paper is discussed and compare with other in this paper.
First bakery algorithm

The processes are numbered 0, 1, ..., N-1. Each process i has an integer variable num[i], initially 0, that is readable by all processes but writeable by process i only.
When process i is thinking, num[i] equals zero. When process i becomes hungry, it sets num[i] to a value higher than the num of every other

process; this operation is assumed to be atomic in this simplified algorithm.

Then process i scans the other processes in order. For each process j, process i waits until num[j] is either zero or greater than num[i]. After going past every process, process i enters the critical section. Upon leaving the critical section, process i zeroes num[i].

The entry and exit codes for process i are:

```
entry(i) {
   i.1: num[i] := max( num[0], num[1], ... , num[N-1] ) + 1 ;
         // i.1 is atomic
        for p := 0 to N-1 do  {      // p is local to process i
   i.2:   while num[p] != 0 and num[p] < num[i] do no-op ;
        }
}

exit(i) {
   i.3: num[i] := 0 ;
}
```

progress

Suppose process i becomes hungry at time t_1 , i.e. executes i.1. We need to show that it eventually eats. The only places where process i can get stuck indefinitely are the while loops i.4 and i.5. So it suffices to show that this does not happen.

Rather than arguing at the level of the statements, it is instructive to take a more abstract view. With respect to a hungry process i, we can divide the processes into four groups:

- a process j is in group 1 if in j.3..6 (i.e. num[j] is nonzero).
- a process j is in group 2 if in j.2 and started executing j.2 before process i finished executing i.2 (i.e. num[j] is zero but it may be assigned a value less than or equal to num[i]).
- a process j is in group 3 if either it is in j.1 or it is in j.2 and it started executing j.2 after process i finished executing i.2 (i.e. num[j] is zero and if it is changed it will be assigned a value greater than num[i]).

A process j in group 1 with (num[i],i)<(num[j],j) holding will not eat before process i eats (because j will not get past process i). Thus the group 1 processes are queued in their timestamp order.

A process j in group 2 eventually joins group 1 (because the execution of j.2 cannot be blocked by other processes). It can join at any place in the ``timestamp queue'' of group 1, i.e., it does not have to join at the end.

A process j in group 3 may eventually join group 1, but it will only do so behind process i.

If process j is at the head of group 1's queue, then eventually either j will eat or another process (from group 2) will get ahead of it.

Let F_1 be the number of processes in group 1 ahead of process i. Let F_2 be the number of processes in group 2. Then F_1+F_2 keeps decreasing, and so process i eventually eats.

A better metric is the lexicographically ordered two-tuple (F_2, F_1). If F_2 is non-zero, then eventually either F_2 decreases and F_1 increases or F_1 decreases. In either case the metric decreases, and it cannot go below (0,0).