4. Mutual Exclusion

Assumption: When Pi has the token and Po requests for the token and Pi is not in its critical section, Pi can send a token message directly to Po (without doing a broadcast, i.e., by a simple message).

1. For Po: When it requires to enter the critical section, it sends a "request token" message to all processes Pi, i ≠ 0. It waits for the process Pi to finish its critical section and when it gets the token, it runs its critical section code. On exiting from critical section, it sends token to Pi.

For Pi (i ≠ 0): (a) If Pi has token, and wants to run the critical section code, it enters its critical section.

On finishing execution of critical section code, Pi sends token to Pi(i+x) mod (n+1) where x is modular operation.
(b) If Pi does not want to execute critical section, it forwards token to Pi(i+x) mod (n+1).

There are n+1 processes numbered 0 ... n, n+1.
We assume that when Po does not want to execute critical section, its behaviour is same as token ring.
2. At P₀
   Use Best Effort Broadcast (BeB)

   Upon event <init>
   CSReq = false;
   Send<token, P₁₀>  // Initially token is at P₀

   Upon event <Execute Critical Section>
   CSReq = true;
   Be₀ ← CS
   be₀ Broadcast <"Request Token">

   Upon event <token received>
   if (CSReq = true)
     trigger <executeCS>  // Run CS
     wait until CS finishes
   CSReq = false
   Send<token, P₁₀>  // Send token to process P₁₀

This is checked for the case when the token comes from P₁₀ but P₀ did not want the token.
AT Pi (i ≠ 0)

upon event <Token received>
  if (CSReqV = true)
    trigger <execute CS>
    Wait until CS finishes
  CSReqV = false;
  if (message "request Token" from Po is received)
    Send <Token, Po> 
  else
    Send <Token, P(i+1)∪(i+1)>

upon event <init>
CSReqV = false;

upon event <Execute Critical Section>
CSReqV = true;

3. ME1: The token is with only one process at a time. So only one process can execute its critical section. So ME1 guaranteed.

ME2: The process with the token is eligible to execute critical section. At any time there is one process with the token. So ME2 guaranteed.

4. ME3 is verified at Po because when it requests for the token, the currently executing Po process Pi currently running its critical section will eventually exit its critical section and send it to the process Po. Po has to wait at most 1 Critical section in the worst case. √
In the worst case process Pi, i ≠ 0 has to wait infinitely. This happens if no requests for the token before the token can ever make a full cycle. That is if the token can never make a full cycle, then there is at least one process which has to wait for will be starved.

Hence ME3 is not guaranteed.

Second Part: Improvement of the algo

At Pi

1. upon event <init>
   CSreq = false;
   Send <token, P_i> 0
   token.lastprocen = 0;
   Send <token, P_i>

2. upon event <Execute Critical Section>
   CSreq = true;
   bcastBroadcast <"Request Token”>;

3. upon event <token received>
   if (CSreq = true)
      trigger <execute CS>
      wait until CS finishes
   CSreq = false
   Send <token, (token.lastprocen + 1) i>(m+1)

At $P_i$ ($i \neq 0$)

Upon event <token received>
   if ($CSReq = \text{true}$)
      trigger $\langle \text{execute CS} \rangle$
      Wait until CS finishes
      $CSReq = \text{false}$;

   token lastProcen = $i$;

   if (Message "Request token" from $P_0$)
      send $\langle \text{token, } P_0 \rangle$;
   else
      send $\langle \text{token, } P(i+1) \rangle$; (at $i+1$);

Upon event <init>
   $CSReq = \text{false}$;

Upon event <Execute CS>
   $CSReq = \text{true}$;

$ME_1$ is guaranteed since there is only one token, and only one process has the token, so only one process can execute its Critical Section.

$ME_2$ is guaranteed since there is one process with the token and it is eligible to enter its Critical Region.

$ME_3$ is also guaranteed in this case: Assume $P_0$ requests token infinitely often. After exiting from its Critical Region once, this algorithm guarantees that the token is sent to another Process $P_i$. This process may or may not execute its Critical Section. In the mean time if $P_0$ requests for critical
section, $P_i$ sends the token to $P_0$. But after $P_0$ finishes, the token goes to $P_{k+1}$. So even if $P_0$ requests infinitely often, the token goes around the ring, although it might be interleaved by $P_{k+1}$.

For example, the worst case is the following (showing tokens received at $P_0$):

$P_0, P_1, P_0, P_2, P_0, P_3, P_0, P_4, \ldots, P_0, P_0, P_0, P_0, P_1, \ldots$

So no process is starved. MT3 is guaranteed.

**Fault Tolerance**

In the second version, the lines of code are numbered.

When checkpointing before sending message:

When a process sends the token to any other process.

We assume that failure detection is perfect and correct processes can learn about failed processes.

Now, with message logging we can handle failures:

Case I: If failure of $P_i$ occurs after $P_j$ sends token to $P_i$. In this case, $P_i$ is correct so $P_i$ detects that $P_i$. However, just after $P_i$ finishes, the token goes to $P_{k+1}$.
Fault Tolerance

In the message logging case, we assume that processes recover. That is if the group which has the token recovers soon so that the algorithm can proceed. This assumption is done because we cannot take the token from a failed process.

Now, if we have a failure detector and the process with the token detects that a process Pk has failed, Pk will not send the token to Pk even if it was Pk’s turn.

If Pk fails with the token then we have a system halt.

In the improved algorithm line 5 will change (as we will need to log a message either before or after the sending of the token).

Line 14 will change and there will be a message logged before or after line 14. If it is before line 14, then the process logs the message before sending the token. Hence if the process fails just before sending the token, it can recover and continue checkpoint.

However, if the message logging is done after the sending and the process fails just before sending the token & then it fails and the system loses the token since even when the process recovers, it is in an indeterminate state as it did not checkpoint the state.
Similarly for line 21 and 23.

If checkpoint is taken before more token sending, and the process fails before the sending, then it can recover and continue.

However, if the process fails before the send and the checkpoint was to be taken after the send, then the token to be lost depending on what state the process recovers to: i.e., earlier checkpoint (after send) assumes that token is with another process and the token is lost.

If the failure of any process occurs when it executes its critical section, the token is lost. However, if failure occurs before just before the send, or just after the send, then the token is lost only in the case when the checkpoint is taken after the send.

Suppose token is at P0. And P1 is executing code line 19. And just after that, it fails. No matter if the checkpoint was taken before or after the sending of the token, in this case, the token is lost since P0 and P1 know that the token is not with them and when P1 recovers, it recovers without losing the information (since it did not reach the checkpoint stage) and recovers to a state in which it does not have the token.