04_01_Mutual_Exclusion

1. BackGround

1. The need for mutual exclusion arises when a resource can only be accessed by one process at a time.
2. **Critical Section(CS)**: in order to access the resource, a process executes a critical section (CS)

2. Model Structure:

1. There are n processes:

$$P_i, i = 0, \ldots, n-1$$

running on multiple processors in a connected network;

1. Each process P_i has a CS, which takes a finite amount of time to execute;
2. **At most one** of the P_i is allowed to be executing its CS at the same time (mutual exclusion);
3. When a process P_i requests entry to its CS, it is **guaranteed to enter it within finite time** (no starvation or deadlock）

2.1. Central Solution：

Assign a **single node** the task of granting the processes exclusive access to their CSs.

Pros

single point of failure+ potential performance bottleneck

2.2. Classification of Distributed Solution

2.2.1. Token-based:

there is a **single distinguished message,** the **token,**the possession of which allows a process to execute its CS

main issues:

1. the prevention of starvation
2. the prevention of deadlock

2.2.2. Assertation-based:

request **permission from all or part of the other processes,** and based on their replies, it may conclude that it is the only one with the right to access its CS

2.3. Requirements

- no deadlock
- no starvation
- fairness (e.g. requests granted in some time order)

3. Assertation-Based Mutual-Exclusion Algorithms

3.1. Lamport’s mutual-exclusion algorithm

Assumption

1. All links are FIFO

Preparation

1. Messages with **timestamp**=[scalar logical time, id of sending processor]
2. Each process with a **queue** of requests ordered according to timestamp
3. Message type: Request, Reply, Release

Procedure

1. A process wishing to enter its CS **broadcasts a timestamped REQUEST** message to all processes, including itself
2. When a process **receives a REQUEST message**,

a. it enters the request into its **queue Q** of requests ordered according to timestamp,

b. **and** sends back **a REPLY message**

3. A process is allowed to enter its CS when

   a. it has received **a REPLY message from every process**

   b. **and** when its own request is **at the head** of its request queue

4. When a process leaves its CS, it **sends a RELEASE message to all processes**, which then **remove** the request from their request queues.

Understanding

Receive all REPLY guarantee a process will know the total sequence that should obey to enter the CS

Implementation

Know the total sequence

Implementation:

I. Broadcasting a REQUEST message

```
no_replies ← 0
T ← current timestamp
broadcast (request;T,i)
```

II. Receiving a REQUEST message

```
upon receipt of (request;T,j) do
    enqueue(Q,(T,j))
    send (reply) to P_j
```

III. Receiving a REPLY message

```
upon receipt of (reply) do
    no_replies ← no_replies + 1
    Conditional_CS
```

IV. Receiving a RELEASE message

```
upon receipt of (release) do
    Q ← tail(Q)
    Conditional_CS
```

V. Procedure Conditional_CS

```
if ((no_replies=n) and (head(Q)=(*,i)))
then
    Critical Section
    broadcast (release)
```

> Lamport's mutual-exclusion algorithm

Complexity

3(n-1) messages **per CS** invocation

Optimization

REPLY and RELEASE can be combined into a single one

Attention

FIFO channel requirements must be met

3.2. Ricart's and Agrawala's mutual-exclusion algorithm

Assumption

Preparation

1. Messages with **timestamp**=[scalar logical time, id of sending processor]
2. Each process with a **queue** of requests ordered according to timestamp
3. Message type: Request, Reply

Procedure

1. A process wishing to enter its CS **broadcasts a timestamped REQUEST** message to all processes, including itself
2. When a process **receives a REQUEST message**,
    a. **REPLY immediately** if a request has a higher precedence than own **REQUEST** or itself do not have a request
    b. else **defer** REPLY until it finishes its own CS
3. A process is allowed to enter its CS when
    a. it has received **a REPLY message from every process**
    b. **and** when its own request is **at the head** of its request queue
4. When a process leaves its CS, it will send **REPLY** to the deferment request.

Understanding

1. The deferment guarantee the later request will not obtain the full **REPLY**,
2. a **REQUES**T can only get enough **REPLY** until the process with higher timesteps has finished

Implementation

```
Requesting access to the CS in process Pi:
  broadcast(REQUEST,ts,i) to all processes

Receiving a request in process Pi from process Pj:
  upon receipt of (REQUEST,ts,j) do
    if  (  (Pi not in CS) and
           ((Pi not requesting) or (ts,i)>(ts,j))  ) then
              send(REPLY,ts') to Pj
    else defer REPLY
```

Complexity

2(n-1) messages per CS invocation (RELEASE is not needed anymore)

Optimization

In order to reduce the message complexity, one may try to reduce the size of the **request set** of processes to which a process sends its requests and from whom it needs permission to enter its CS.

need the request sets of two processes to have a non-empty intersection

Attention

FIFO channel is not neccessary

3.3. Model of Request Set

Definition

Request set of a process is to which it sends its **REQUESTs** and from whom it needs **REPLYs**

Requirement of Request Set

1. any two request sets have a **non-empty intersection (M1)**

Desirable features of request sets:

1. every process is **in its own request set (M2)**
2. all request sets have **the same size (M3)**
3. every process is contained in **the same number of request sets (M4)**

Lemma 1

The minimum size of the request sets is of order $o(\sqrt{n})$

Proof: See slides

Create Methods

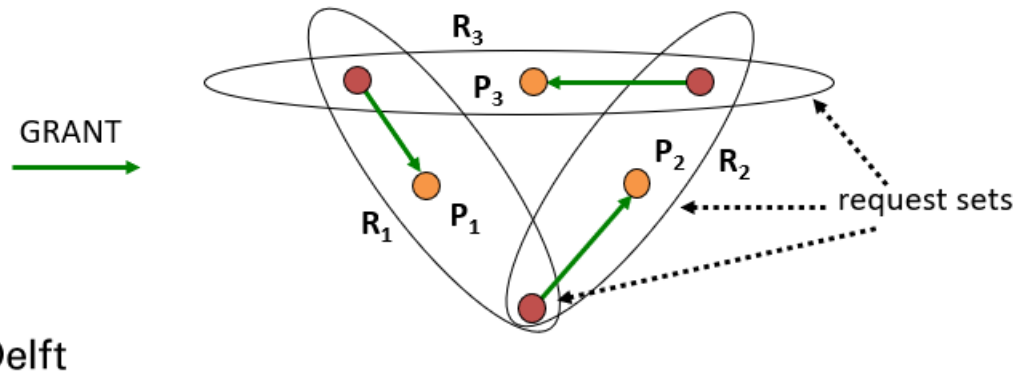
Understanding

The non-empty intersection guarantee for two process, there is always at least agent can help them to determine the sequence of them entering into CS

3.4. Maekawa's mutual-exclusion algorithm

Background

After using request set, it may occur deadlocks:



Request_Set_Deadlock.png

Preparation

1. Request set R (the intersection of any two request sets is non-empty)
2. Ordered Queue of request
3. Message Types: **Request (Carry scalar timestamps), Grant, Release, Postpone, Inquire, Relinquish**

Procedure

1. wants to enter its CS → multicasts a timestamped **REQUEST message to the members of R**
2. a process enter CS → received a **GRANT** from all processes in R
3. finishing its CS → sends **Release** Message
4. Receives **REQUEST message** →
   a. when it **has not sent** a **GRANT** to another process without having received the corresponding RELEASE message → replies with a **GRANT**
   b. otherwise compares the timestamps of the two requests
      i. the timestamp of the new request is the later → **queues** the new request in a queue Q + sends the requesting process a **POSTPONE message**
      ii. Otherwise → **INQUIRES** with the process to whom it has sent a GRANT message
5. receives an **INQUIRE message** →
   a. waits until either it has obtained a **GRANT** from every process in R → **completes its CS** + **replies with a RELEASE** message
   b. until it has received a **POSTPONED message** → gives the permission back with a **RELINQUISH message**
6. receiving a **RELINQUISH message** →
   a. **enqueues** the corresponding CS request
   b. and sends a **GRANT message** to the process with the oldest request it knows of
7. receiving a **POSTPONE message** → just **waiting**
8. P_i **can enter its CS** when it has received a GRANT from all processes in its **request set**

Implementation

I. Multicasting a REQUEST message

```
no_grants ← 0
T ← current timestamp
for all j ∈ R do
    send (request; T, i) to P_j
```

II. Receiving a REQUEST message

```
upon receipt of (request; T, j) do
    if (¬granted) then
        current_grant ← (T, j)
        send (grant) to P_j
        granted ← true
    else
        insert (Q, (T, j))
        (V, k) ← head (Q)
        if (current_grant < (T, j)) or ((V, k) < (T, j)) then
            send (postponed) to P_j
        else
            if (¬inquiring) then
                inquiring ← true
                l ← current_grant.node
                send (inquire; i) to P_l
```

III. Receiving a GRANT message

```
upon receipt of (grant) do
    no_grants ← no_grants+1
    if (no_grants = |R|) then
        postponed ← false
        Critical Section
        for all j ∈ R do
            send (release) to P_j
```

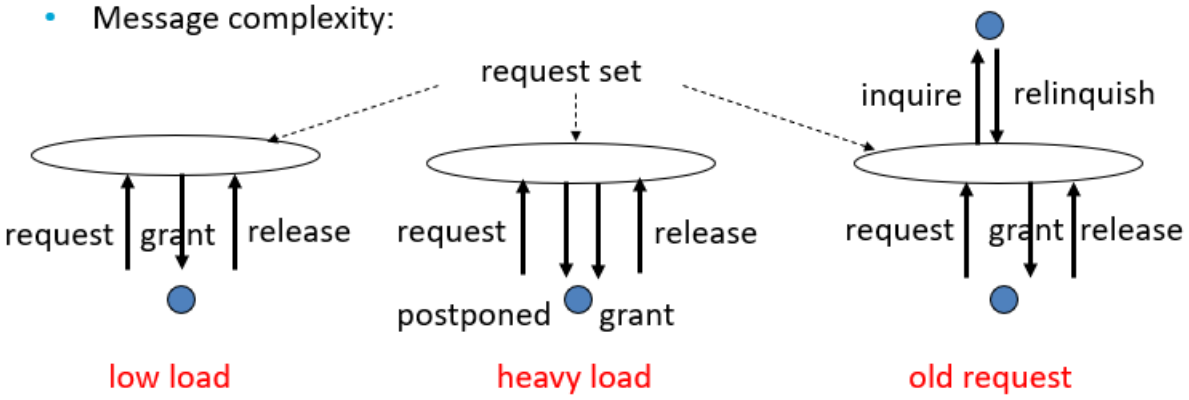
VI. Receiving a RELEASE message
upon receipt of (release) **do**
granted ← false
inquiring ← false
if (not_empty(Q)) **then**
current_grant ← head(Q)
dequeue(Q)
j ← current_grant.node
send (grant) **to** P_j
granted ← true

IV. Receiving an INQUIRE message
upon receipt of (inquire; j) **do**
wait until ((postponed) **or** (no_grants = |R|))
if (postponed) **then**
no_grants ← no_grants−1
send (relinquish) **to** P_j

V. Receiving a RELINQUISH message
upon receipt of (relinquish) **do**
inquiring ← false
granted ← false
insert(Q, current_grant)
current_grant ← head(Q)
dequeue(Q)
j ← current_grant.node
send (granted) **to** P_j
granted ← true

Complexity

- Message complexity:



> Maekawa_Complexity

Comparision

In Algorithm 1.5.2.

the receiving process can enter its CS as far as the process sending the REPLY is concerned. In particular, a process can send multiple REPLY messages.

In Algorithm 1.5.4.

the sending process gives the process to which it sends a GRANT exclusive access, and will only send a GRANT to another process after it has received a RELEASE (or a RELINQUISH) from the first

3.5. A generalized mutual-exclusion algorithm

Preparation

Every Process P_i has three sets:

1. a **request set R_i: request and obtain grants for entering CS**
2. an **inform set I_i**: notify these processes
    a. of requests for entry to the CS
    b. but also of releases of the CS
3. a **status set S_i**: maintain information about these processes

Process P_j is in the **inform set** of process P_i $\Leftrightarrow$ process P_i is in the **status set** of process P_j

Correctness Condition

Preparea

- **Scalar Logic Times** are maintained with ids as tie breakers
- Ever process maintain:
    - an ordered **request queue** with requests not yet granted
    - a variable **CSSTAT** that indicates the process in the status set to which permission has beeng granted or Null
- P_i can enter its CS when it has received a GRANT from all processes in R_i
    - Ricar-Agrawala type: A process acquires **permission from all processes in its request set minus its inform set** "as far as I am concerned, you can enter your CS"
    - Maekawa type: a procecss acquires permission from **all process in its inform set** "I grant you exclusive access"

Correctness Conditions

1. every process P_i is in its own inform set I_i (and so, in its own status set S_i)
2. a process's inform set I_i is a subset of its request set R_i
3. for every two processes:
    a. either the **intersection of their inform sets** is nonempty (Maekawa-like condition)
    b. or they **are in each other's request sets** (Ricart-Agrawala-like condition)

Implementation

- **Requesting access** to the CS in process P_i:
  send(REQUEST,ts,i) to all processes in R_i

- **Receiving a request** in process P_i from process P_j:

```
on receipt of (REQUEST,ts,j) do
    if (CSSTAT = NULL) then                /* so no exclusive access granted */
        send(GRANT) to Pj
        if ( Pj is in Si ) then CSSTAT = j  /* exclusive grant */
    else
        enter (ts,j) into the request queue
```

From this picture, a process can send multiple GRANTs, but only one to a process in its status set (indicated in CSSTAT)

Releasing the CS in process P_i:
send(RELEASE) to all processes in I_i

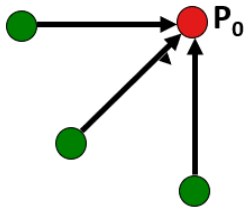
- **Receiving a release** message in process P_i (from a process in S_i):

```
upon receipt of (RELEASE) do   /* from a process in status set */
    CSSTAT = NULL
    repeat until ((request queue = empty) or (CSSTAT ≠ NULL))
        send(GRANT) to process Pj at head of request queue and
                remove it from this queue
        if ( Pj in Si ) then CSSTAT = j
```

Two Scenario

- A centralized algorithm

In the **centralized algorithm**:



- Suppose P_0 is the special process (also does requests)
- **What are I_i** and **R_i** ?
- $I_i = R_i = \{0\}$ for i=0,1,2, …,n: process P_0 gets all requests and releases
- **What is S_0** ?
- $S_0 = \{0,1,2,\ldots,n\}$: process P_0 keeps track of the status of all processes
- **What is S_i** for i=1,2,…,n ?
- S_i = empty for i=1,2,…,n

- In **Ricart-Agrawala**'s algorithm:
  - $R_i = \{0,1,\ldots,n-1\}$: every process sends its requests to (and receives grants from) all processes
  - $I_i = \{i\}$: processes do not send releases
  - $S_i = \{i\}$: processes only keep track of their own status

4. Token-Based Algorithms

4.1. Suzuki's and Kasami's broadcast-based mutual-exclusion algorithm

Assumption

1. If a process gets the token, no matter how many times it has request for entering into critical section, it will finish all the tasks at one time

Preparation

1. Processes
   a. maintain an **array N**: each position represents the **number** of the last request of each process our process knows about
   b. update **N** when they receive a request
2. Token: contains an **array TN** about the **number** of each process's last request that was granted

1. Message:
   a. contains sender

b. contains the number of the request (help other process updates its N)

Procedure

1. When a process **leaves its CS**,:
   a. it updates its own component of **TN**
   b. it scans **N** and **TN from its own entry onwards**, and **sends the token** to the first process P_i with **N[i]>TN[i]**
   c. if for all processes ***N[i] = TN[i]***, then it **keeps** the token until a request arrives

Understanding

This algorithm provides an un-deadlocked way to transfer the token.

1. token record the last request has been finished by each process and each position is maintained only by process itself
2. by comparing the number, the request that has not been dealt with will received a token

Implementation

Assumed: a process enters its CS only once after it has received the token, If it is allowed to do so multiple times when it holding the token, it should check first whether it is holding the token in code fragment I

I. Broadcasting a REQUEST message

```
N[i] ← N[i] + 1
broadcast (request; i, N[i])
```

II. Receiving a REQUEST message

```
upon receipt of (request; j, r) do
   N[j] ← r
    if  ((token_present) and (not in CS)) and
(N[j]>TN[j]) then
      token_present ← false
      send (token; TN) to P_j
```

III. Receiving the token

```
upon receipt of (token; TN) do
   token_present ← true
   Critical Section
TN[i] ← N[i]
for j=i+1 to n, 1 to i-1 do
   if (N[j] > TN[j]) then
      token_present ← false
      send (token; TN) to P_j
      break
```

Suzuki_implementation

Suzuki_implementation_2.png

Complexity

After optimization

n messages per CS invocation: n-1 message, 1 token

Variation

Using a Queue in the token: when a process finish CS

1. extend the queue through local knowledge, choose the head from the queue,
2. choose the head of the queue, send it
3. it does not have another choice to send, keep it

will be optimal when some channel has a long delay

4.2. Singhal's multicast-based mutual-exclusion algorithm

Tries to optimize Suzuki-Kasami's algorithm by only sending requests to processes **that may have the token**

Assumption

1. The token **starts in process** P_1
2. If a process gets the token, no matter how many times it has request for entering into critical section, it will finish all the tasks at one time

Preparation

1. State of processes:
    a. **R**: requesting the token
    b. **E**: executing the CS
    c. **H**: outside the CS, holding the token, and not aware of any requests
    d. **O**: other (own state)
2. Process:
    a. maintains an array **N** with **request numbers**
    b. maintains an array **S** with the **states of the processes** in the system
3. Token:
    a. carries an array **TN** with **request numbers**
    b. carries an array **TS** with **process states**
    c. initialization: all ***TN[i] = 0***, all $TS[i] = 0$
4. Initialization of the state arrays:
    a. In P_1: S[1] = H /* holding the token */
    S[j] = O, j=2,3,…,n
    b. In P_j: S[j] = R, j=1,2,…,i-1 /* **previous ones may have token** */
    S[j] = O, j=i, i+1,…,n

So, **initial values** in the state arrays:

P_1:	H	O	O	O	...	O	O
P_2:	R	O	O	O	...	O	O
P_3:	R	R	O	O	...	O	O
...							
P_N:	R	R	R	R	...	R	O

>If+

> Singhal mutext initialize array

Procedure

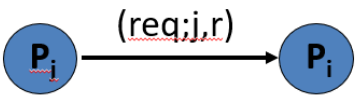
Implementation

Assumed: a process enters its CS only once after it has received the token, If it is allowed to do so multiple times when it holding the token, it should check first whether it is holding the token in code fragment I

- **Requesting access** to the CS

```
S[i] := R                                   /* set own state to requesting */
N[i] := N[i]+1                              /* increment request number */
for j=1 to i-1, i+1 to n do                 /* send request to those processes */
    if ( S[j] = R ) then                    /* that may have the token */
        send(request;i,N[i]) to P_j
```

- **Receiving a request**




```
upon receipt of (request;j,r) do
    N[j] := r
    case S[i] of                          /* depending on local state do */
      E,O:   S[j] := R                    /* record process as requesting */
      R:     if (S[j] ≠ R) then
                 S[j] := R                /* record process as requesting */
                 send(request;i,N[i]) to Pj   /* send request after all */
      H:     S[j] := R;  S[i] := O        /* if holding the token, */
             TS[j] := R; TN[j] := r
             send(token) to Pj            /* send it to requesting process */
```

- **Receiving the token**

local process more up-to-date than token

Question: why ">" and not "≥" ?

```
upon receipt of (token) do
    S[i] := E
    Critical Section
    S[i] := O; TS[i] := O          /* own state */
    for j=1 to n do                /* update information on other processes */
      if (N[j] > TN[j]) then TN[j] := N[j];   TS[j] := S[j]
      else                   N[j]   := TN[j]; S[j] := TS[j]
    if (S[j]=O for all j) then S[i] := H  /* nobody requesting */
    else send(token) to some Pj with S[j] = R
```

Understanding

1. Different from the previous algorithm, the TS and TN in this part is only used to synchronize the current knowledge about other process, more specifically, use TN and N to compare the knowledge's timeliness in TS and S.

Correctness

When at some point in the execution of the algorithm

1. no process is requesting or is in its CS,
2. and no messages are in transit,
3. then the **state of the system** with respect to the values of the arrays N and S **is identical to the initial state**, up to a **permutation** of the process ids

Complexity

1. low contention: per CS: n/2 REQUEST messages and one token transfer
2. high contention: n messages per CS

4.3. Raymond's token-based algorithm

Assumption

1. If a process gets the token, no matter how many times it has request for entering into critical section, it will finish all the tasks at one time

Preparation

1. Spanning tree:
    a. root: the process currently **holding the token** is the **root**
    b. processes only know their neighbours in the tree (only know whether their neighbours need token or not)
2. Processes:
    - maintains a variable **holder**
        - its own id if it holds the token
        - the id of its neighbor **that is on the path to the root** (its parent in the current tree)
    - **request_queue**:
        - **its own id** if it has a request pending
        - **the id of any neighbor** (at most once)

Procedure

1. a process that wants to enter its CS **appends its id to its request_queue, and sent to parent**
2. a process that receives a request
    a. **appends** the id of its request_queue
    b. sending a request to parent
3. in both cases**, it only sends a request** (with its own id) to its holder **if it hasn't** done so already
4. A process that releases its CS
    a. sends the token to the process at the had of its request_queue
    b. stops being root
5. A process that receives the token :
    a. becomes root
    b. enter its CS if it is at the head of its request_queue
    c. or sent the token to another process at the head of its queue sends a request to the same process if its request_queue is still no empty

Understanding

1. the FIFO queue and the re-sent of the request when the queue is not empty guarantee that the root always knows all request.

Implementation

I. Procedure `assign_token()`

```
if ( (holder = self) and (request_queue ≠ empty) ) then
    holder ← head_of(request_queue)
    asked ← false
    if (holder = self) then
        CS
    else send(token) to holder
```

II. Procedure `do_request`

```
if ( (holder ≠ self) and (request_queue ≠ empty) and (asked = false) )
then
    send (request) to holder
    asked ← true
```

III. Requesting access to the CS

```
enqueue(self,request_queue);assign_token();do_request()
```

IV. Receiving a REQUEST message

```
enqueue(sender,request_queue);assign_token();do_request()
```

V. Receiving the token

```
holder ← self;assign_token();do_request()
```

VI. Releasing the CS

```
assign_token();do_request()
```

> Raymond mutex implementation.png

Complexity

- **Worst-case message complexity under low contention**:
  - **2(D-1)** with **D** the diameter of the tree (request + token)
  - **D = N** when the tree is linear
  - **D = log N** for random trees
- **Message complexity under high contention**:
  - token does a **tree traversal**: travels **twice** across every link
  - token does one step in response to one request message
  - so 4 messages for every CS invocation

4 (

> Raymond mutex complexity.png

1.7. Summary

Assertion-based

1.	**Lamport**:	basic algorithm (request-reply-release)
2.	**Ricart-Agrawala**:	small optimization (defer replies)
3.	**Maekawa**:	optimization with request sets
4.	**Generalized**:	generalizes 2. and 3.

Token-based

1.	**Suzuki-Kasami**:	basic algorithm (requests to everybody)
2.	**Singhal**:	optimization (requests only to potential token holders)
3.	**Raymond:**	optimization (use a spanning tree)

> summary_mutext