Resource Allocation

Resource Allocation : Schedule

The Critical-Section Problem

- Synchronization Hardware
- Semaphores
- Algorithms

Distributed Mutual Exclusion Algorithms

- Central Algorithm
- Lamport's Algorithm
- Ricart-Agrawala Algorithm
- Maekewa's Algorithm
- Suzuki-Kasami Algorithm
- Raymond's Algorithm

Background

Concurrent access to shared data may result in data inconsistency.
Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
Shared-memory solution to bounded-butter problem (Chapter 4) allows at most *n* – 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.

- Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

Producer process

```
item nextProduced;

while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Bounded-Buffer

Consumer process

```
item nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Bounded Buffer

The statements

counter++;
counter--;

must be performed *atomically*.

Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

The statement "**count++**" may be implemented in machine language as:

register1 = counter

register1 = register1 + 1

counter = register1

The statement "**count—**" may be implemented as:

register2 = counter

register2 = register2 – 1

counter = register2

Bounded Buffer

If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

Interleaving depends upon how the producer and consumer processes are scheduled.

Assume **counter** is initially 5. One interleaving of statements is:

producer: **register1 = counter** (*register1 = 5*)

producer: **register1 = register1 + 1** (*register1 = 6*)

consumer: **register2 = counter** (*register2 = 5*)

consumer: **register2 = register2 – 1** (*register2 = 4*)

producer: **counter = register1** (*counter = 6*)

consumer: **counter = register2** (*counter = 4*)

The value of **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

n processes all competing to use some shared data

Each process has a code segment, called *critical section*, in which the shared data is accessed.

Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes.

Software Solutions

In addition to **mutual exclusion**, prevent **mutual blocking**:

1. Process outside of its CS must not prevent other processes from entering its CS.

2. Process must not be able to repeatedly reenter its CS and **starve** other processes (*fairness*)

3. Processes must not block each other forever **(deadlock)**

4. Processes must not repeatedly yield to each other ("after you"-- "after you" **livelock**)

Shared Memory : Peterson's Mutual Exclusion Algorithm

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 6.2 The structure of process P_i in Peterson's solutio

Cooperation

Problems with software solutions:

- Difficult to program and to verify
- Processes loop while waiting (busy-wait)
- Applicable to only to critical problem: *Competition* for a resource

Cooperating processes must also *synchronize*

Classic generic scenario:

Producer → Buffer → Consumer

Bakery Algorithm

Critical section for n processes

Before entering its critical section, process receives a number.

Holder of the smallest number enters the critical section.

If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.

The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

Notation <≡ lexicographical order (ticket #, process id #)

- $(a,b) < c,d)$ if $a < c$ or if $a = c$ and $b < d$
- max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i - 0, …, n – 1

Shared data

boolean Entering[n];
int Number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (integer j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /* nothing */ }
    }
}

unlock(integer i) {
    Number[i] = 0;
}
```

Bakery Algorithm

```
Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Synchronization Hardware

Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;}
```

Shared data:

```
boolean lock = false;
```

Process P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section }
```

Synchronization Hardware

Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Shared data (initialized to false):

```
boolean lock;
boolean waiting[n];
```

Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
        critical section
    lock = false;
        remainder section
}
```

Semaphores

Semaphore *S* – integer variable

can only be accessed via two indivisible (atomic) operations

wait (*S*):

while $S \leq 0$ **do** ***no-op;***

S--;

signal (*S*):

S++;

Critical Section of *n* Processes

Shared data:

semaphore mutex; //initially *mutex* = 1

Process *Pi:*

```
do {
    wait(mutex);

        critical section

    signal(mutex);

    remainder section

} while (1);
```

Semaphore Implementation

Define a semaphore as a record

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

Assume two simple operations:

- **block** suspends the process that invokes it.
- **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

Semaphore operations now defined as

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```

Semaphore as a General Synchronization Tool

Execute *B* in P_j only after *A* executed in P_i

Use semaphore *flag* initialized to 0

Code:

P_i	P_j
⋮	⋮
A	*wait(flag)*
signal(flag)	*B*

Deadlock and Starvation

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	*wait*(Q);
wait(Q);	*wait*(S);
⋮	⋮
signal(S);	*signal*(Q);
signal(Q)	*signal*(S);

Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

Counting semaphore – integer value can range over an unrestricted domain.

Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.

Can implement a counting semaphore *S* as a binary semaphore.

Classical Problems of Synchronization

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem

Shared data

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

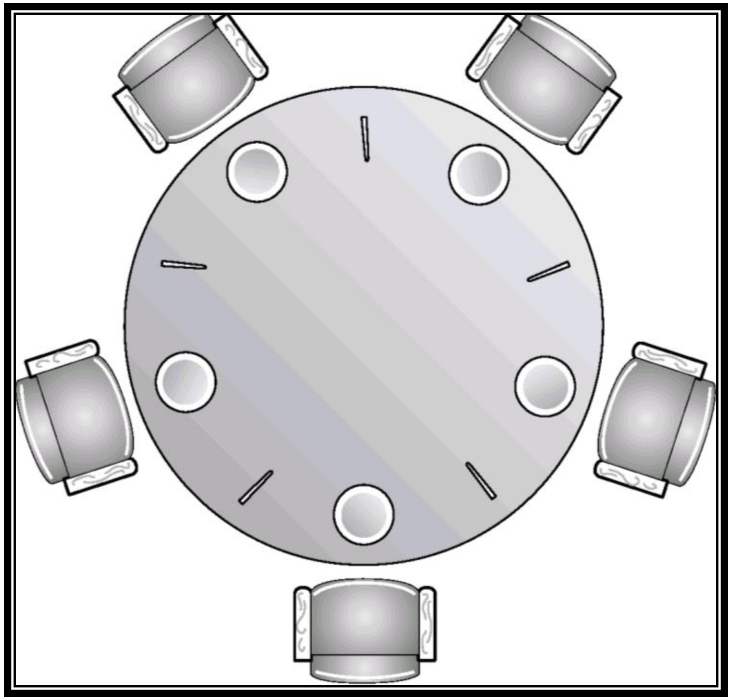
Bounded-Buffer Problem Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

Bounded-Buffer Problem Consumer Process

```
do {
    wait(full)
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (1);
```

Dining-Philosophers Problem



Shared data

semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

Philosopher *i*:

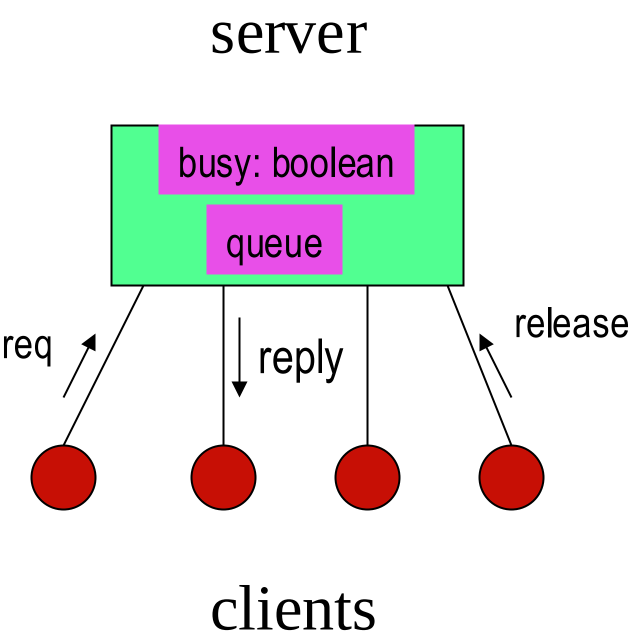
```
do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
        ...
        eat
        ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        ...
        think
        ...
} while (1);
```

Why mutual exclusion?

Some appplications are:

Resource sharing
Avoiding concurrent update on shared data
Controlling the grain of atomicity
Medium Access Control in Ethernet
Collision avoidance in wireless broadcasts

Centralized Solution



Client

do true $\rightarrow$

send request;

reply received $\rightarrow$ enter CS;

send release;

<other work>

od

Server

do request received **and** not busy $\rightarrow$ send reply; busy:= true

□ request received **and** busy $\rightarrow$ enqueue sender

□ release received **and** queue is empty $\rightarrow$ busy:= false

□ release received **and** queue not empty $\rightarrow$ send reply

to the head of the queue

od

Comments

Centralized solution is simple.

Leader maintains a pending queue of events

Requests are granted in the order they are received

But the server is a ***single point of failure***. This is BAD.

Can we do better? Yes!

Central MUTEX Algorithm : Client

P_i::
var
v: array$[1..N]$ of integer initially $\forall j : v[j] = 0$;
$inCS$: boolean initially $false$;

To $request$:
$v[i] := v[i] + 1$;
send ($request$, v) to P_0;

Upon $receive(token)$ from P_0:
$inCS := true$;

To $release$:
send $token$ to P_0;
$inCS := false$;

Upon $receive(u)$: // program message
$v := max(v, u.v)$;

Central Coordinator Algorithm

P_0::

var

$reqdone$: array$[1..N]$ of integer initially 0;
$reqlist$: list of $(pid, reqvector)$ initially $null$;
$havetoken$: boolean initially $true$;

Upon $receive(u, request)$:
$append(reqlist, u)$;
if $havetoken$ **then** $checkreq()$;

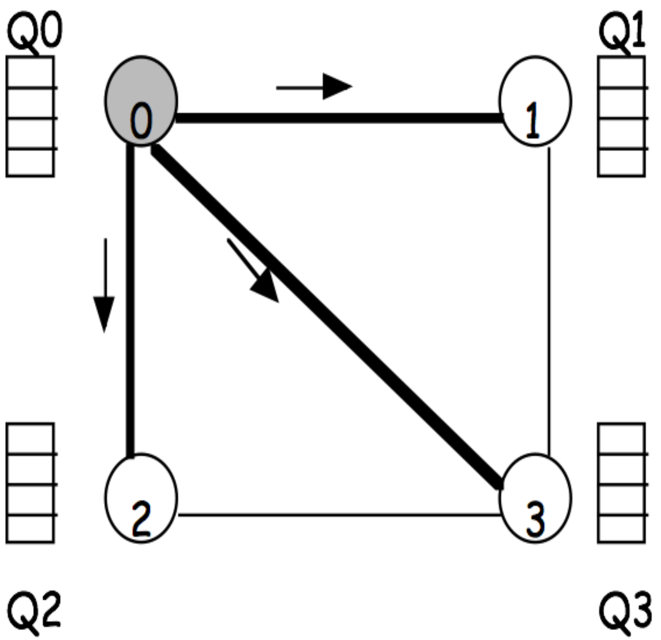
Upon $receive(u, token)$:
$havetoken := true$;
$checkreq()$;

$checkreq()$:
$eligible := \{w \in reqlist \mid \forall j : j \neq w.p : w.v[j] \leq reqdone[j]\}$;
if $eligible \neq \{\}$ **then**
$w := first(eligible)$;
$delete(reqlist, w)$;
$reqdone[w.p] := reqdone[w.p] + 1$;
send $token$ to $P_{w.p}$
$havetoken := false$;
endif

```java
// Centralized mutex algorithm
public class CentMutex extends Process implements Lock {
        . . .
    public synchronized void requestCS() {
        sendMsg(leader, "request");
        while (!haveToken) myWait();
    }
    public synchronized void releaseCS() {
        sendMsg(leader, "release");
        haveToken = false;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("request")) {
            if (haveToken){
                sendMsg(src, "okay");
                haveToken = false;
            }
            else  pendingQ.add(src);
        } else if (tag.equals("release")) {
            if (!pendingQ.isEmpty()) {
                int pid = pendingQ.removeHead();
                sendMsg(pid, "okay");
            } else haveToken = true;
        } else if (tag.equals("okay")) {
            haveToken = true;    notify();
        }
    }
}
```

Decentralized solution 1 : Lamport's Algorithm

1. Broadcast a timestamped ***request*** *to all.*
2. Request received → enqueue it in **local Q**.
   Not in CS → send ***ack***, else postpone sending ***ack*** until exit from CS.
3. Enter CS, when
   (i) You are at the head of your Q
   (ii) You have received ***ack*** from all
4. To exit from the CS,
   (i) Delete the ***request*** from Q, and
   (ii) Broadcast a timestamped ***release***
5. When a process receives a ***release*** message, it removes the sender from its **Q**.



Lamport's Algorithm

program Lamport's Algorithm

define m: msg

try, done : boolean

Q : queue of msg {Q.i is process i entry}

N : integer

initially try = false {turns true when a process wants CS}

in = false { aprocess enters CS only when in is true}

done = false {turns true when a process wants to exit CS}

1. do try -> m := (i, req, t);
   ∀ j : j !=i send m to j
   enqueue i in Q
   try :=false

2.(m.type=request) -> j:=sender
   enqueue j in Q;
   send(i,ack, t') to j

Lamport's Algorithm

3. (m.type = ack) -> N := N + 1

4. (m.type = release) -> j := m.sender; deque Q.j from Q

5. (N = n-1) and (∀ j != i : Q.j.ts > Q.i.ts) -> in := true
   { process enters CS}

6. in and done -> in := false; N := 0;
   dequeue Q.i from Q;
   ∀ j : j != i send (i, release, t") to j;
   done : = false

Lamport's Algorithm

```
Pᵢ::
 var
      v: depclock; // direct-dependency clock
      q: array[1..N] of integer initially (∞, ∞, ..., ∞);

 request:
      q[i] := v[i];
      send (q[i]) to all processes; // request messages

 release:
      q[i] := ∞;
      send (q[i]) to all processes; // release messages

 receive(u):
      q[u.p] := u.q[u.p];
      if event(u) = request then
           send ack to process u.p; // acknowledge "request"
```

```java
// Lamport's mutual exclusion algorithm
public class LamportMutex extends Process implements Lock {
    public synchronized void requestCS() {
        v.tick();
        q[myId] = v.getValue(myId);
        broadcastMsg("request", q[myId]);
        while (!okayCS())  myWait();
    }
    public synchronized void releaseCS() {
        q[myId] = Symbols.Infinity;
        broadcastMsg("release", v.getValue(myId));
    }
    boolean okayCS() {
        for (int j = 0; j < N; j++){
            if(isGreater(q[myId], myId, q[j], j))        return false;
            if(isGreater(q[myId], myId, v.getValue(j), j))return false;
        }
        return true;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        v.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            q[src] = timeStamp;  sendMsg(src, "ack", v.getValue(myId));
        } else if (tag.equals("release")) q[src] = Symbols.Infinity;
        notify(); // okayCS() may be true now
}     }
```

Analysis of Lamport's algorithm

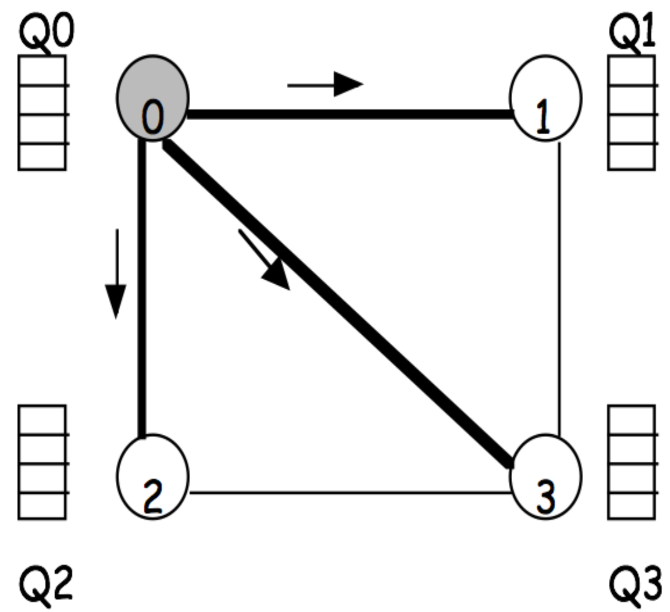
Can you show that it satisfies all the properties (i.e. ME1, ME2, ME3) of a correct solution?

Observation. *Any two processes taking a decision must have* ***identical views*** *of their queues.*

Proof of ME1. *At most one process can be in its CS at any time.*

- j in CS ⇒ Qj.ts.j < Qk.ts.k
- k in CS ⇒ Qk.ts.k < Qj.ts.j

Impossible.



Analysis of Lamport's algorithm

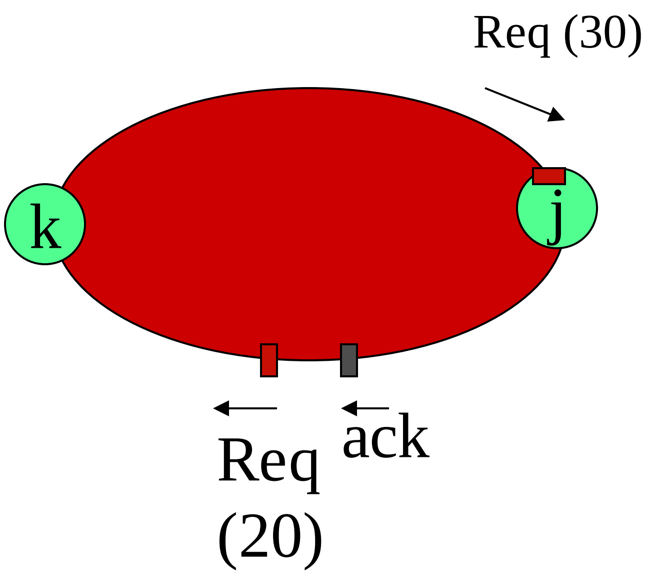
Proof of FIFO fairness.

timestamp of j's request < timestamp of k's request implies j enters its CS before k

Suppose not. So, k enters its CS before j.
So k did not receive j's request.
But k received the ack from j for its own req.

This is impossible **if the channels are FIFO**.

Message complexity = 3(N-1)

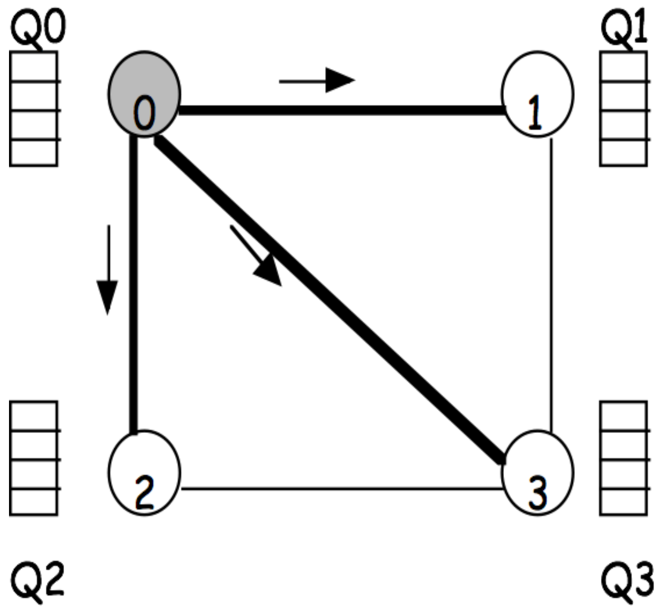


Analysis of Lamport's algorithm

Proof of ME2. *(No deadlock)*
The waiting chain is acyclic.

i waits for j
i is behind j in all queues
j does not wait for I

Proof of ME3. *(progress)*
New requests join the end of the queues, so new requests do not pass the old ones



Analysis of Lamport's algorithm

Proof of FIFO fairness.

timestamp (j) < timestamp (k)

⇒ j enters its CS before k does so

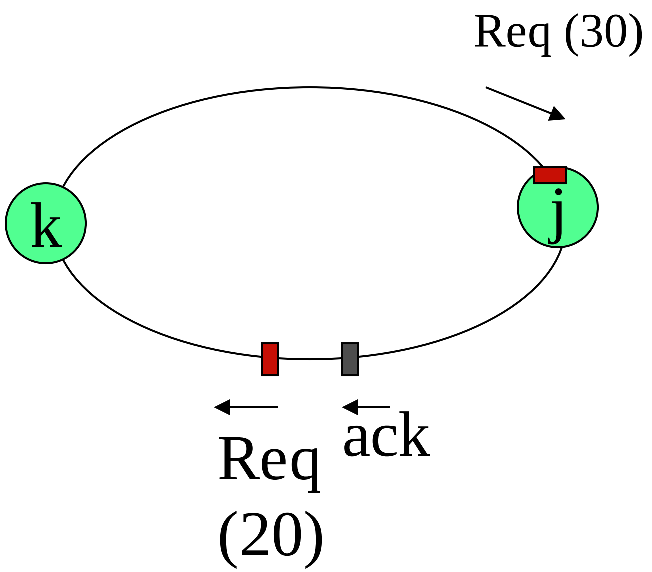
Suppose not. So, k enters its CS before j.
So k did not receive j's request.
But k received the ack from j for its own req.
This is impossible **if the channels are FIFO**.

(*No such guarantee can be given if the channels are not FIFO*)

Message complexity = 3(N-1)
(N-1 requests + N-1 ack + N-1 release)

Ensures that processes enter the critical section in the order of timestamps of their requests

Requires 3(N-1) messages per invocation of the critical section



Decentralized algorithm 2 : Ricart-Agrawala's Algorithm

{Ricart & Agrawala's algorithm}

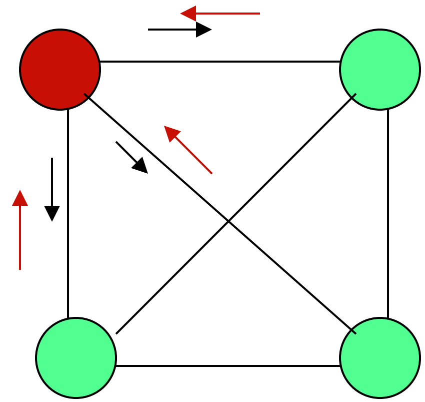
What is new?

1. Broadcast a timestamped ***request*** to all.
2. Upon receiving a request, send ***ack*** if
   - -You do not want to enter your CS, or
   - -You are trying to enter your CS, but your timestamp is higher than that of the sender.

   (If you are in CS, then buffer the request)

3. **Enter CS**, when you receive ***ack*** *from all.*
4. Upon **exit from CS**, send ***ack*** to each pending request before making a new request.

(No release message is necessary)



Ricart Agrawala Algorithm

```
program          Ricart Agrawala Algorithm
define    m: msg
          try, want, in  : boolean
          A : array [0..n-1] of boolean
          t : timestamp
          N : integer { number of acknowledgements}
initially        try = false {turns true when a process wants CS}
                 in = false { aprocess enters CS only when in is true}
                 want = false {turns false when a process exits CS}
                 N = 0
                 A[k] = false { for every k : 0<=k<=n-1}

1. do try                  -> m := (i, req, t);
                            ∀ j : j !=i send m to j
                           try :=false; want := true
```

RA Algorithm

2a. (m.type=request) AND (~want OR m.ts < t)
-> send(i,ack, t') to m. Sender

2b. (m.type=request) AND (want AND m.ts > t)
-> A [sender] := true

3. (m.type = ack) -> N := N + 1

5. (N = n-1) -> in := true
{ process enters CS}
want := false

6. in AND ~want -> in := false; N := 0;
∀ k : send (i, ack, t') to k | A[k]=true;
∀ k : A[k] := false;

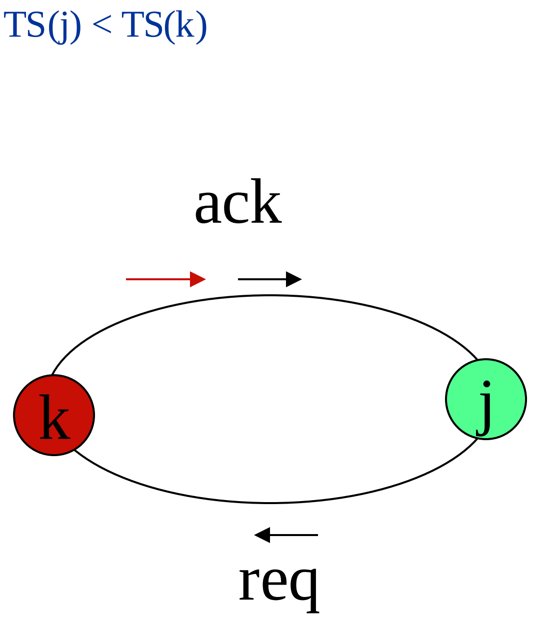
od

Ricart & Agrawala's algorithm

ME1. Prove that at most one process can be in CS.

ME3. Prove that FIFO fairness holds even if channels are not FIFO

Message complexity = 2(N-1)
(N-1 requests + N-1 acks - no release message)



Ricart-Agrawala MUTEX Algorithm

P_i::
var
  $pendingQ$: list of process ids initially $null$;
  $myts$: integer initially ∞;
  $numOkay$: integer initially 0;

$request$:
  $myts := logical\_clock$;
  send $request$ with $myts$ to all other processes;
  $numOkay := 0$;

$receive(u, request)$:
  if $(u.myts < myts)$ **then**
    send $okay$ to process $u.p$;
  else $append(pendingQ, u.p)$;

$receive(u, okay)$:
  $numOkay := numOkay + 1$;
  if $(numOkay = N - 1)$ **then**
    enter_critical_section;

$release$:
  $myts := \infty$;
  for $j \in pendingQ$ **do**
    send *okay* to the process j;
  $pendingQ := null$;

```java
public class RAMutex extends Process implements Lock {
    public synchronized void requestCS() {
        c.tick();
        myts = c.getValue();
        broadcastMsg("request", myts);
        numOkay = 0;
        while (numOkay < N-1)   myWait();
    }
    public synchronized void releaseCS() {
        myts = Symbols.Infinity;
        while (!pendingQ.isEmpty()) {
            int pid = pendingQ.removeHead();
            sendMsg(pid, "okay", c.getValue());
        }
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        c.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            if ((myts == Symbols.Infinity ) || (timeStamp < myts)
                ||((timeStamp == myts)&&(src<myId)))//not interested in CS
                sendMsg(src, "okay", c.getValue());
            else    pendingQ.add(src);
        } else if (tag.equals("okay")) {
            numOkay++;
            if (numOkay == N - 1)  notify(); // okayCS() may be true now
}   }   }
```

Decentralized algorithm 3 : Maekewa's Algorithm

{Maekawa's algorithm}

- First solution with a **sublinear** O(sqrt N) message complexity.
- "Close to" Ricart-Agrawala's solution, but each process is required to obtain permission from only a **subset** of peers

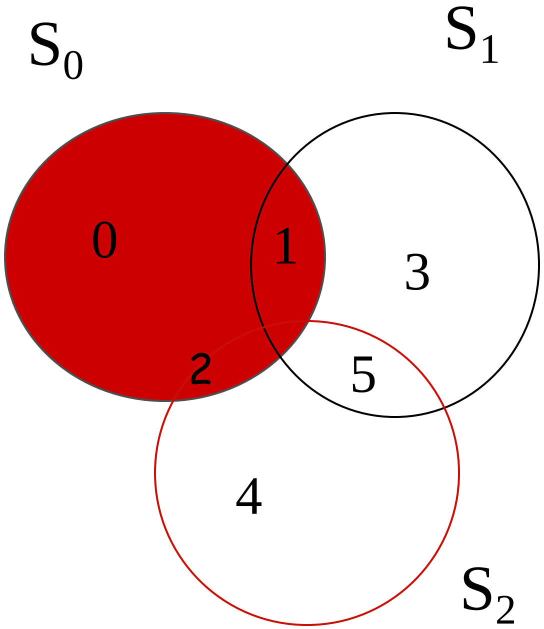
Maekawa's algorithm

With each process i, associate a subset S_i.Divide the set of processes into subsets that satisfy the following two conditions:

$i \in S_i$

$\forall$ i,j : $0 \leq i,j \leq n-1$ :: $S_i \cap S_j \neq \emptyset$

Main idea. Each process i is required to receive permission from S_i only. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.



Maekawa's algorithm

Example. Let there be seven processes 0, 1, 2, 3, 4, 5, 6

S_0	=	{0, 1, 2}
S_1	=	{1, 3, 5}
S_2	=	{2, 4, 5}
S_3	=	{0, 3, 4}
S_4	=	{1, 4, 6}
S_5	=	{0, 5, 6}
S_6	=	{2, 3, 6}

Maekawa's algorithm

Version 1 {Life of process I}

1. Send ***request*** to each process in S_i.

2. Request received → send ***ack*** to process with the if I am not blocked. Thereafter, "**lock**" (i.e. commit) yourself to that process, and keep others waiting.

3. Enter CS if you receive ***ack*** from **each member** in S_i.

4. To exit CS, send ***release*** to every process in S_i.

5. Release received → **unlock** yourself. Then send ack to the next process .

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

Maekawa's algorithm-version 1

ME1. At most one process can enter its critical section at any time.

Let **i** and **j** attempt to enter their Critical Sections
$\mathbf{S_i} \cap \mathbf{S_j} \neq \phi$ there is a process $\mathbf{k} \in \mathbf{S_i} \cap \mathbf{S_j}$
Process **k** will **never** send ack to both.
So it will act as the arbitrator and establishes ME1

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

Maekawa's algorithm-version 1

ME2. No deadlock

Unfortunately deadlock is possible!

From S_0**={0,1,2}**, **0,2** send *ack* to **0**, but **1** sends *ack* to **1**;

From S_1**={1,3,5}**, **1,3** send *ack* to **1**, but **5** sends *ack* to **2**;

Prom S_2**={2,4,5}**, **4,5** send *ack* to **2**, but **2** sends *ack* to **0**;

Now, 0 waits for 1, 1 waits for 2, and 2 waits for 0.

So deadlock **is** possible!

S_0 = {0, 1, 2}

S_1 = {1, 3, 5}

S_2 = {2, 4, 5}

S_3 = {0, 3, 4}

S_4 = {1, 4, 6}

S_5 = {0, 5, 6}

S_6 = {2, 3, 6}

Maekawa's algorithm-Version 2

Avoiding deadlock

If processes receive messages **in increasing order of timestamp**, then deadlock "could be" avoided. But this is too strong an assumption.

Version 2 uses three more messages:

- ***failed***
- ***inquire***
- ***relinquish***

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

Maekawa's algorithm-Version 2

New features in version 2

Send ***ack*** and set lock as usual.

If lock is set and a request with larger timestamp arrives, send ***failed*** *(you have no chance)*. If the incoming request has a lower timestamp, then send ***inquire*** *(are you in CS?) to the locked process*.

- Receive ***inquire*** and at least one ***failed*** message → send ***relinquish***. The recipient resets the lock.

S_0 = {0, 1, 2}

S_1 = {1, 3, 5}

S_2 = {2, 4, 5}

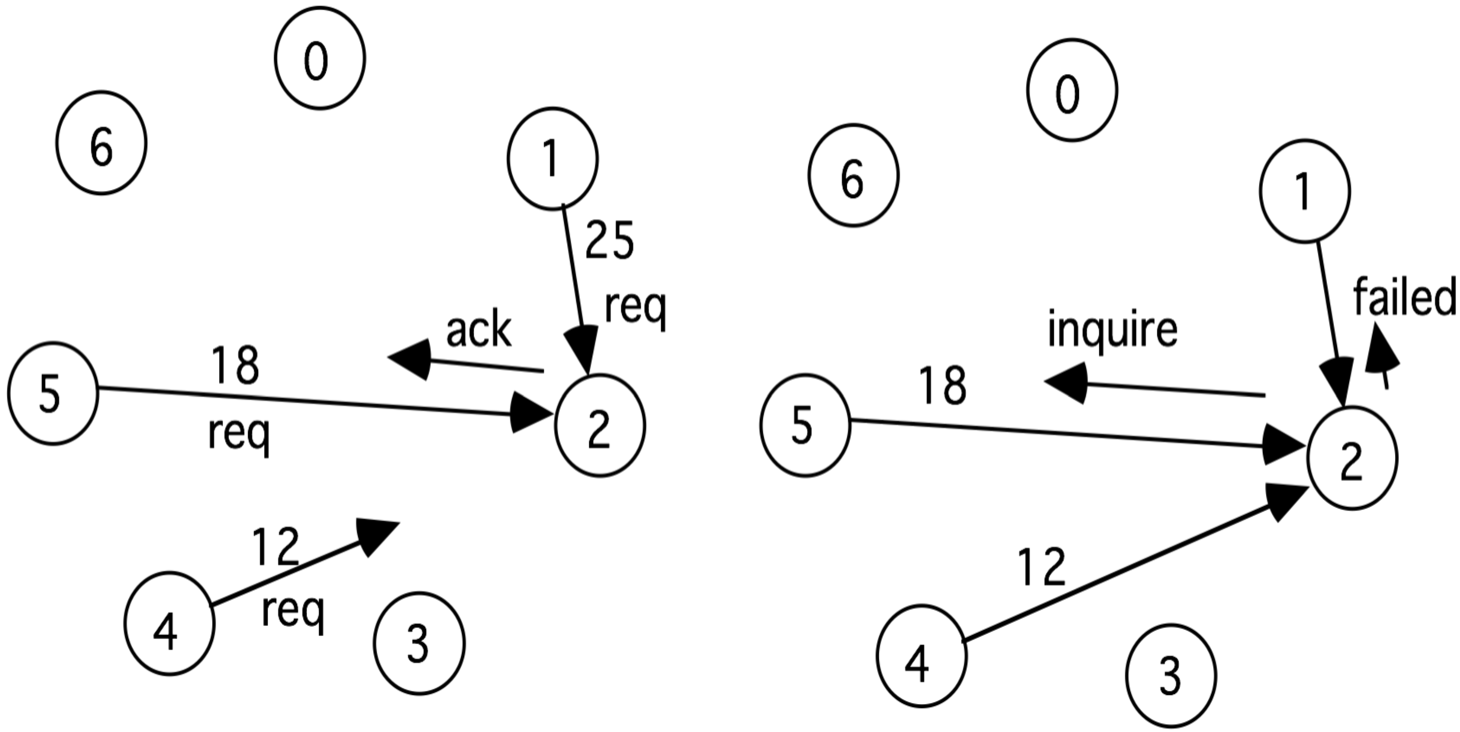
S_3 = {0, 3, 4}

S_4 = {1, 4, 6}

S_5 = {0, 5, 6}

S_6 = {2, 3, 6}

Maekawa's algorithm-Version 2



Comments

Let $K = |S_i|$. Let each process be a member of D subsets. When $N = 7$, $K = D = 3$. When $K=D$, $N = K(K-1)+1$. So K is of the order $\sqrt{N}$

-The message complexity of Version 1 is $3\sqrt{N}$. Maekawa's analysis of Version 2 reveals a complexity of $7\sqrt{N}$

Sanders identified a bug in version 2 …

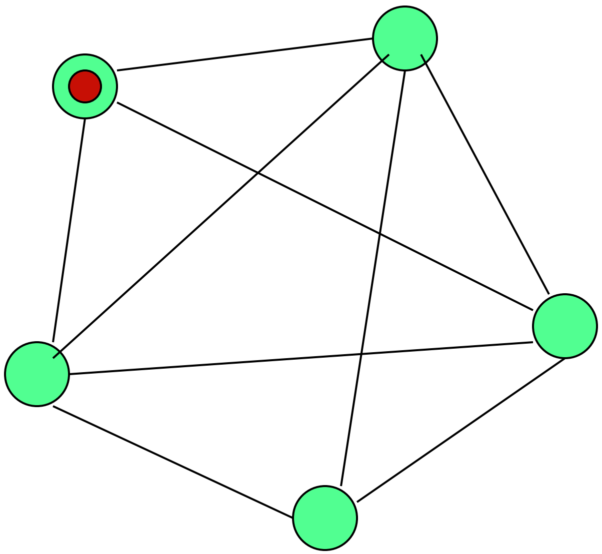
Token-passing Algorithms

Suzuki-Kasami algorithm

Completely connected network of processes

There is **one token** in the network. The owner of the token has the permission to enter CS.

Token will move from one process to another based on demand.



Suzuki-Kasami Algorithm

Process i broadcasts **(i, num)**

Each process maintains

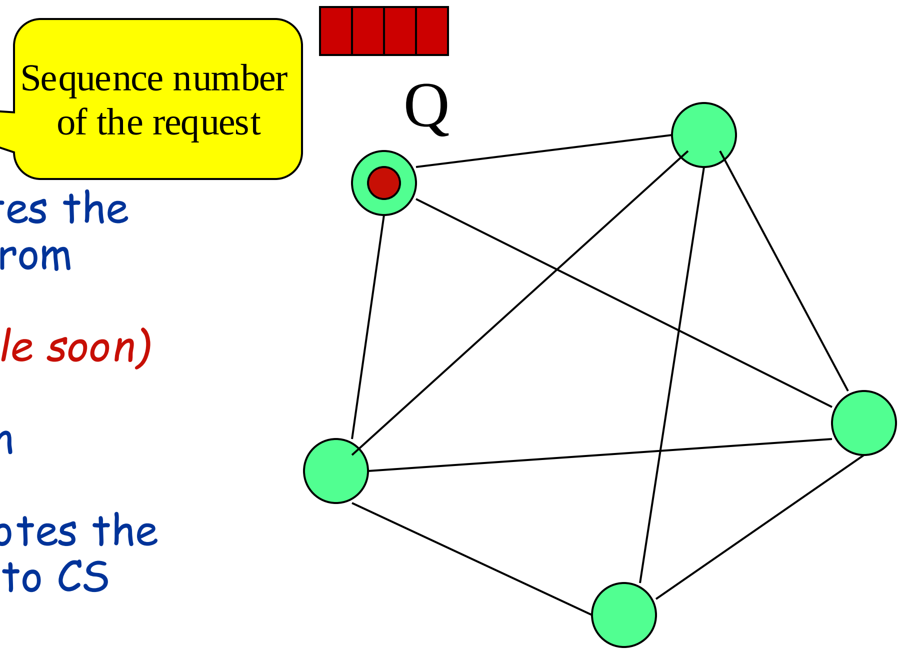
-an array **req**: **req[j]** denotes the sequence no of the *latest request* from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

-an array **last**: **last[j]** denotes the sequence number of *the latest visit* to CS from for process j.

- **a queue Q** of waiting processes



req: array[0..n-1] of integer

last: array [0..n-1] of integer

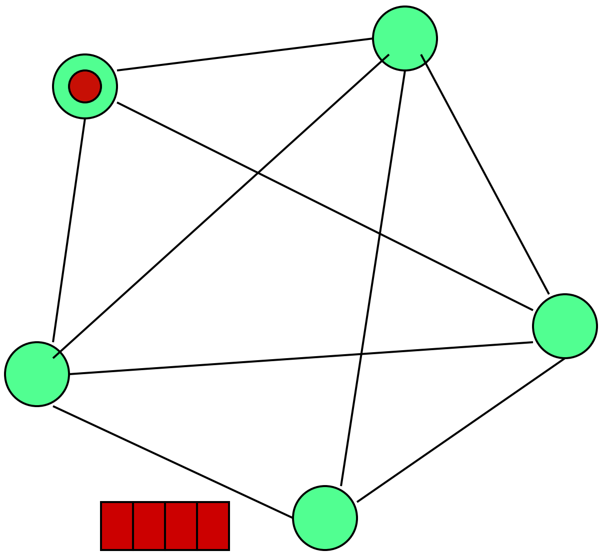
Suzuki-Kasami Algorithm

When a process **i** receives a request **(i, num)** from process **k**, it sets **req[k] to max(req[k], num)** and enqueues the request in its **Q**

When process **i** sends a token to the ***head of Q***, it sets **last[i]** := its own **num**, and passes the array **last**, as well as the ***tail of Q***,

The holder of the token retains process **k** in its Q only if **1+ last[k] = req[k]**

This guarantees the freshness of the request



Req: array[0..n-1] of integer

Last: Array [0..n-1] of integer

Suzuki-Kasami's algorithm

{Program of process j}
Initially, $\forall i$: req[i] = last[i] = 0

*** Entry protocol ***
req[j] := req[j] + 1
Send (j, req[j]) to all
Wait until token (Q, last) arrives
Critical Section

*** Exit protocol ***
last[j] := req[j]
$\forall k \neq j$: $k \notin Q \wedge req[k] = last[k] + 1 \rightarrow$ append k to Q;
if Q is not empty → send (tail-of-Q, last) to head-of-Q **fi**

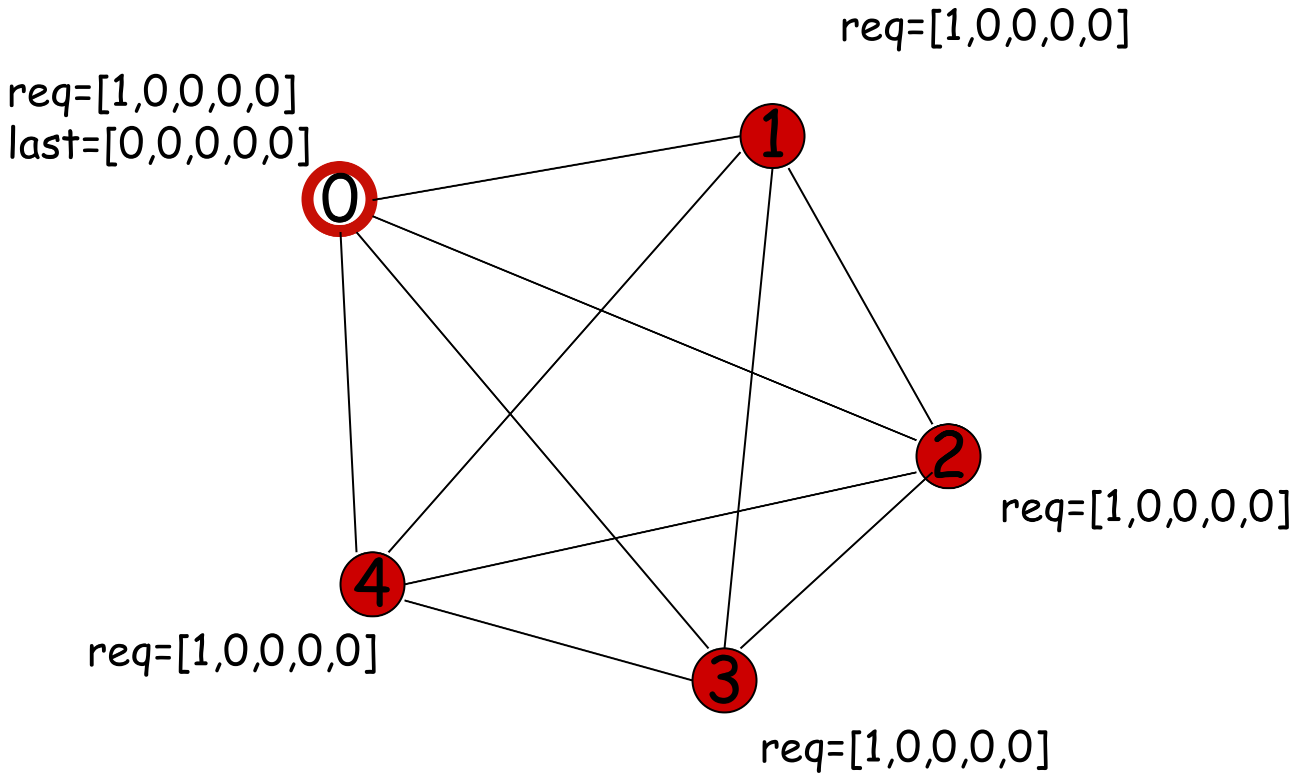
*** Upon receiving a request (k, num) ***
req[k] := max(req[k], num)

Suzuki Kasami's Algorithm

```
Pi::
 var
      v: array[1..N] of integer initially ∀j : v[j] = 0;
      inCS: boolean initially false;
      havetoken: boolean initially false except for P0;
      myreqlist: list of (pid, reqvector) initially null;

 To request:
      v[i] := v[i] + 1;
      send (request, v) to all processes (including itself);

 Upon receive(request, u):
      v := max(v, u.v);
      if (havetoken) then
           append(token.reqlist, u);
           if not inCS then checkreq();
      else append(myreqlist, u);

 receive(u, token):
      inCS := true;
      havetoken := true;
      append(token.reqlist, {u | (u ∈ myreqlist) ∧ (u > token.reqdone)});
      myreqlist := null;

 release:
      inCS := false;
      checkreq();

 receive(u): //program message
      v := max(v, u.v);

 checkreq():
      eligible := {w ∈ token.reqlist such that
                ∀j : j ≠ w.p : w.v[j] ≤ token.reqdone[j]};
      if eligible ≠ {} then
           w := first(eligible) ;
           delete(token.reqlist, w);
           token.reqdone[w.p] := token.reqdone[w.p] + 1;
           send token to Pw.p;
           havetoken := false;
      endif
```

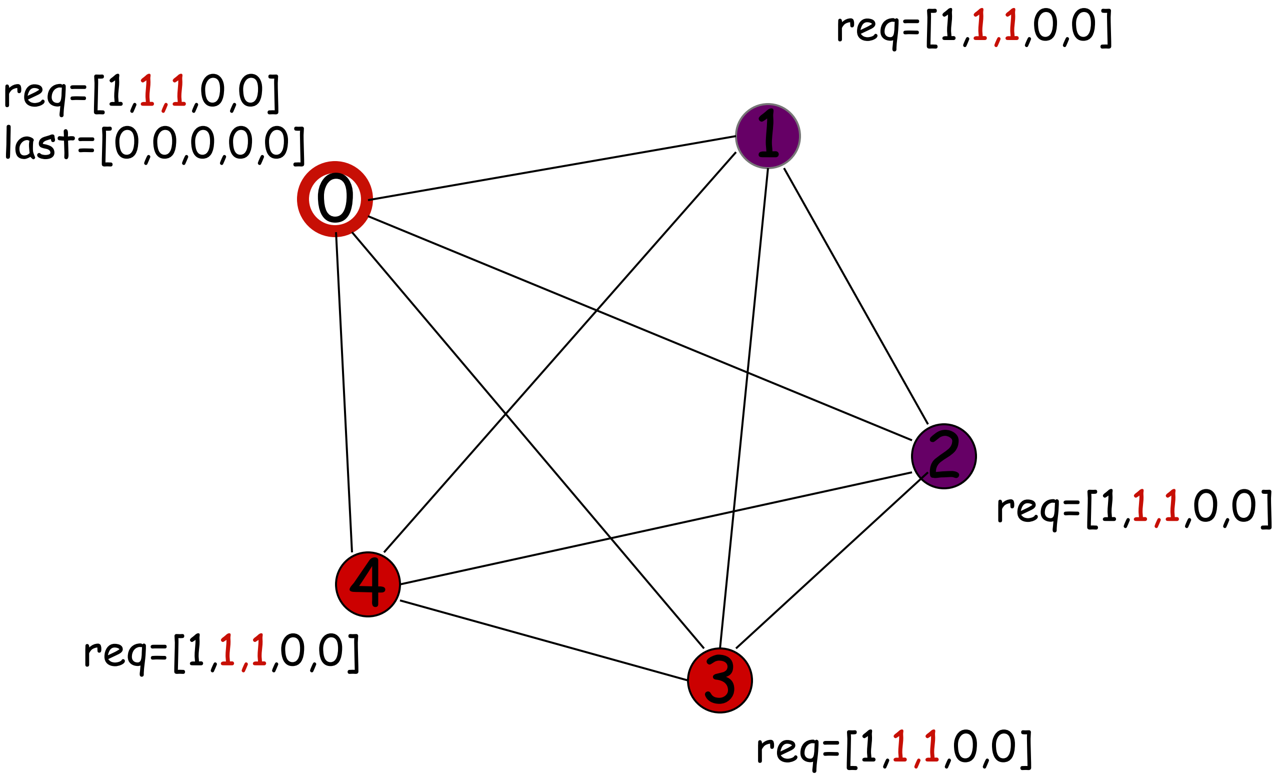
```java
public class CircToken extends Process implements Lock {
    public synchronized void initiate() {
        if (haveToken) sendToken();
    }
    public synchronized void requestCS() {
        wantCS = true;
        while (!haveToken) myWait();
    }
    public synchronized void releaseCS() {
        wantCS = false;
        sendToken();
    }
    void sendToken() {
            . . .
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("token")) {
            haveToken = true;
            if (wantCS)    notify();
            else {
                Util.mySleep(1000);
                sendToken();
            }
        }
    }
}
```

Example



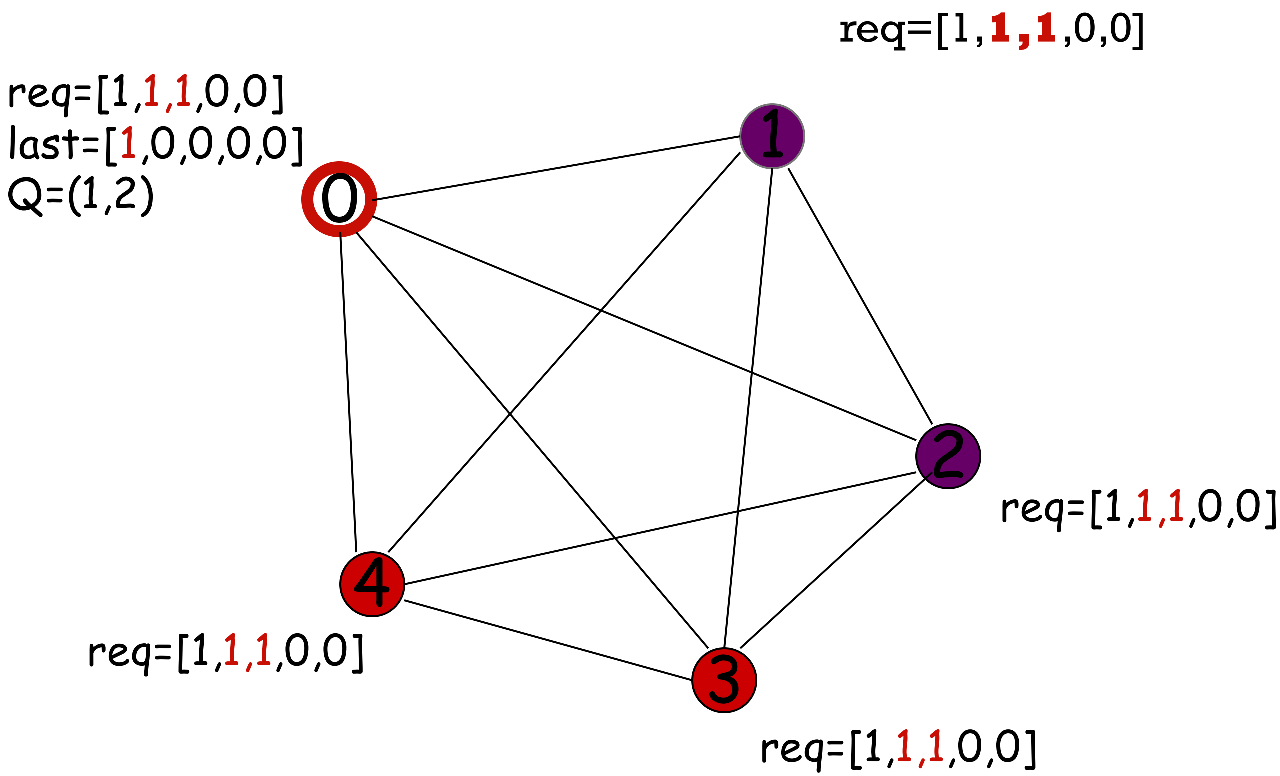
initial state

Example



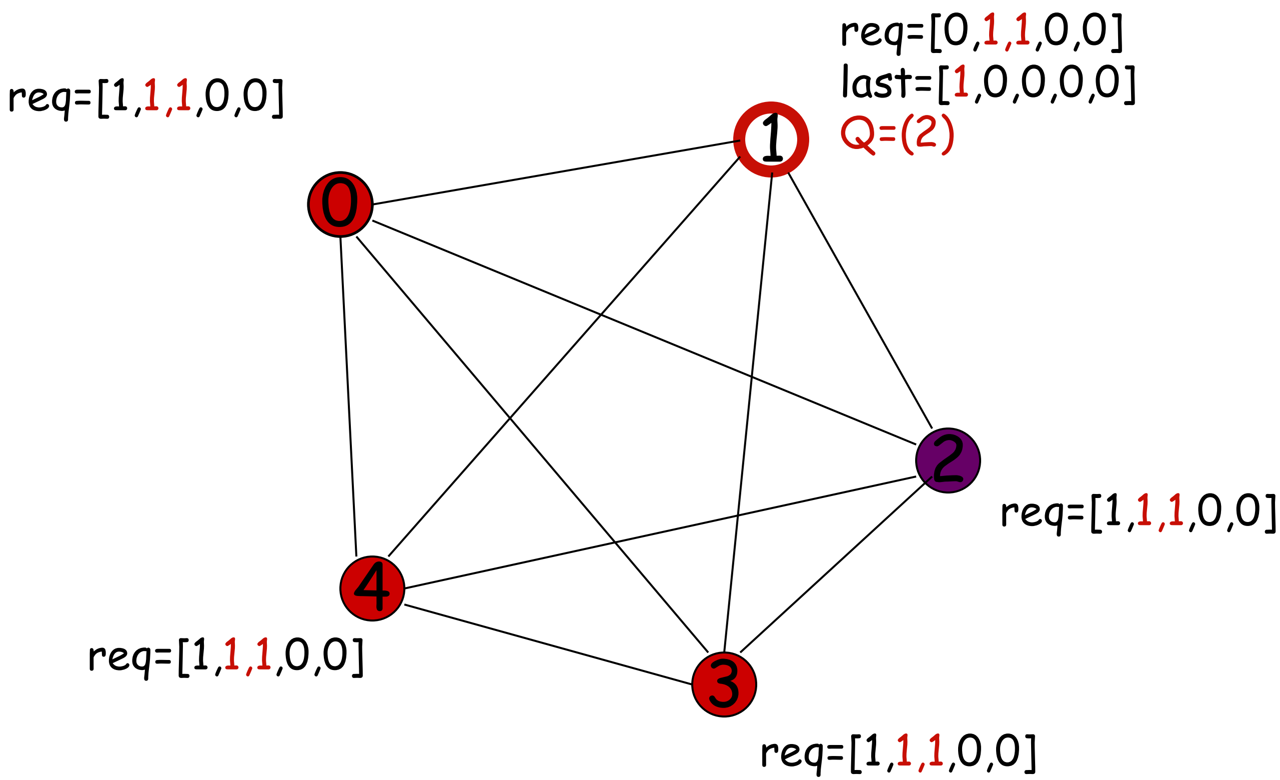
1 & 2 send requests

Example



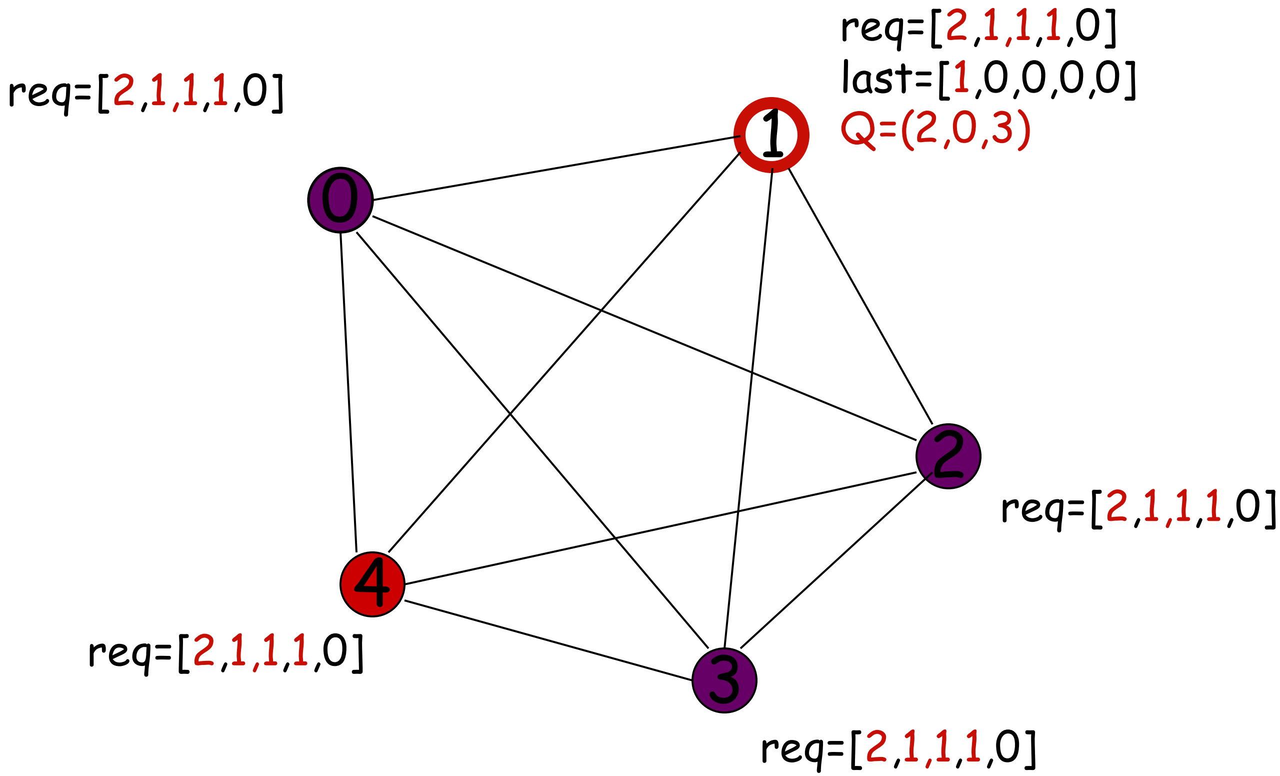
0 prepares to exit CS

Example



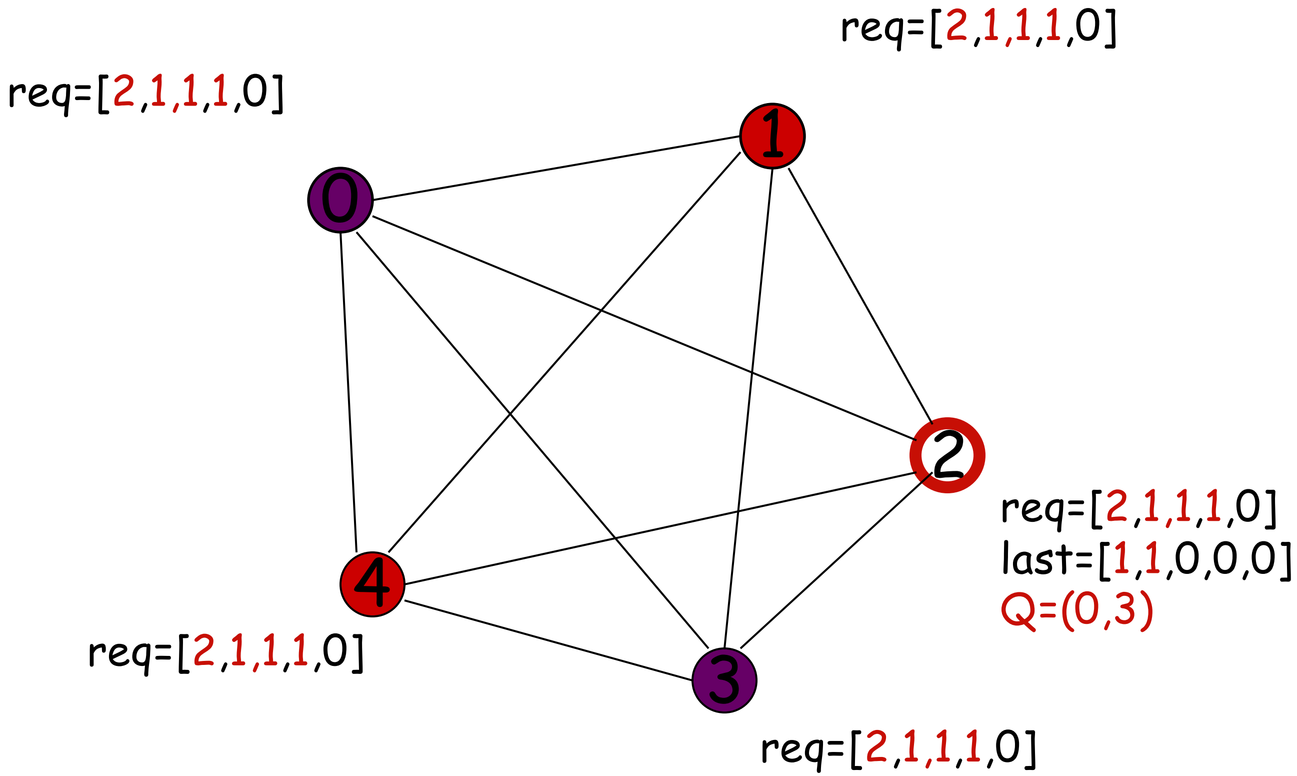
0 passes token (Q and last) to 1

Example



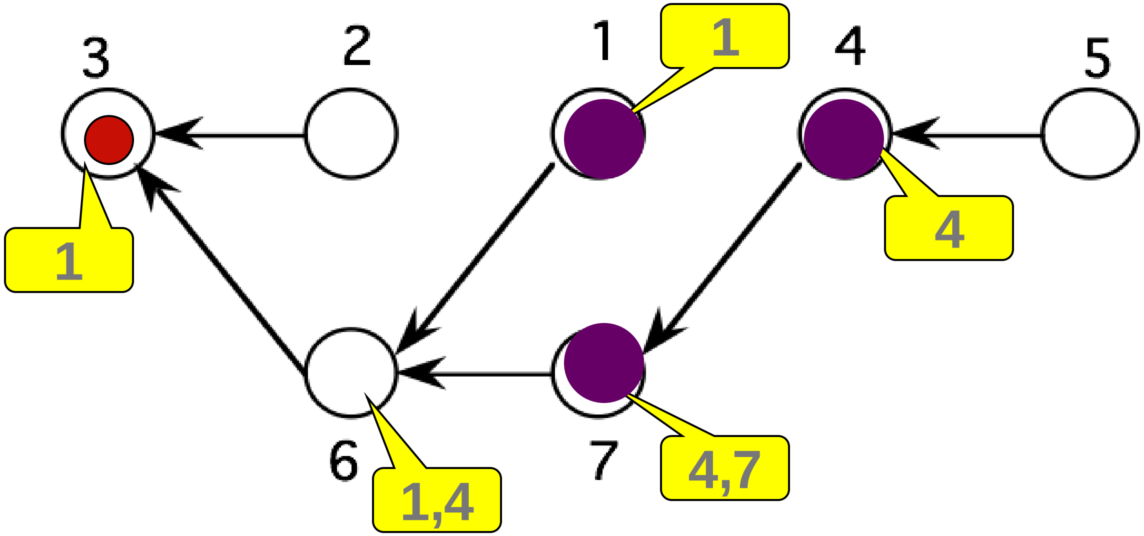
0 and 3 send requests

Example



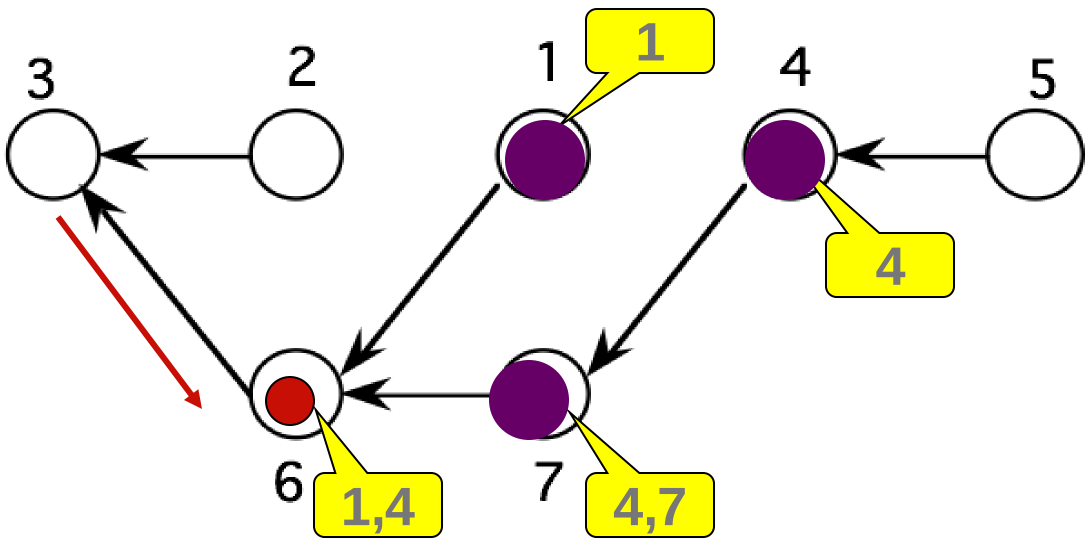
1 sends token to 2

Raymond's tree-based algorithm



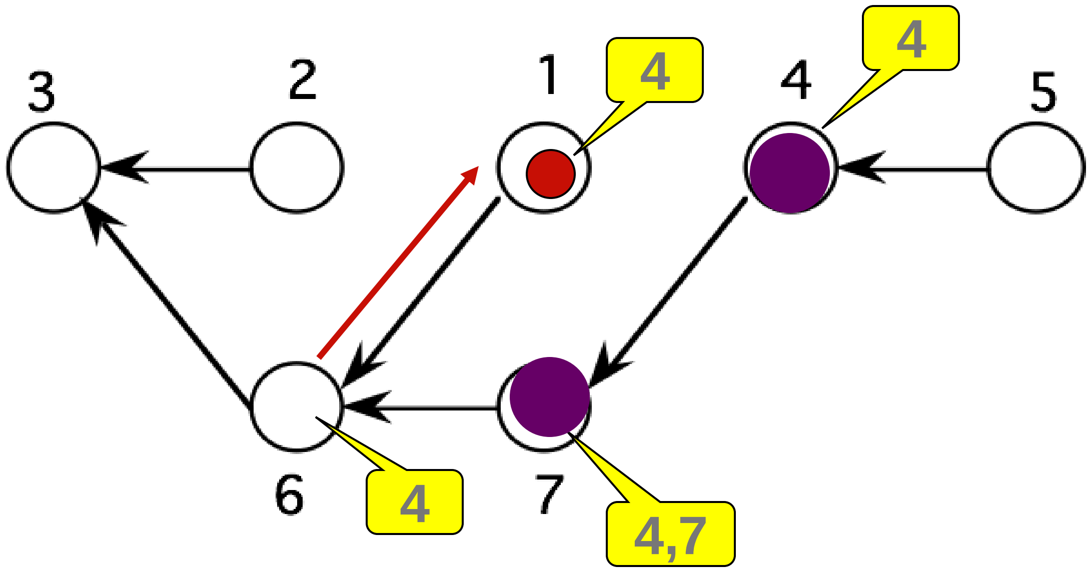
1,4,7 want to enter their CS

Raymond's Algorithm



3 sends the token to 6

Raymond's Algorithm

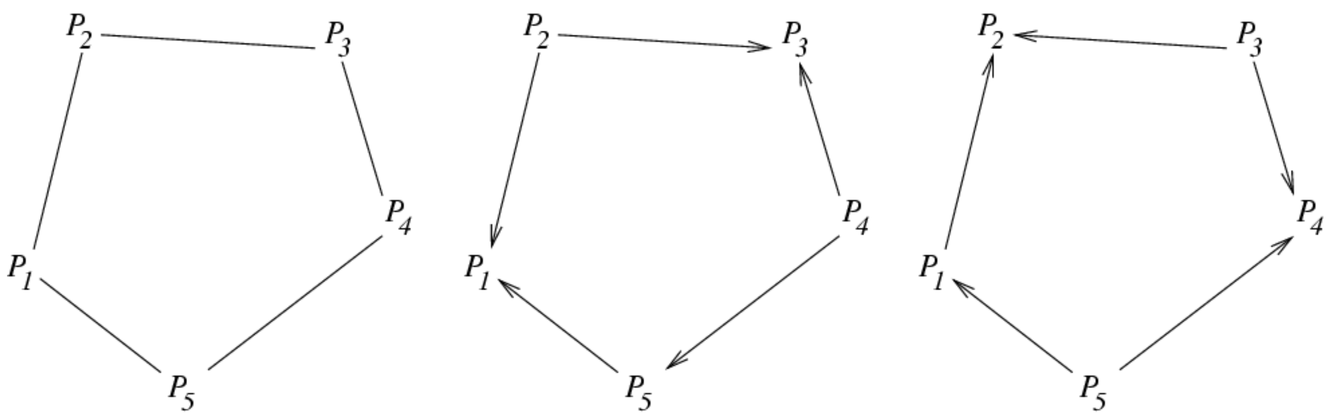


6 forwards the token to 1

The message complexity is O(diameter) of the tree. Extensive Empirical measurements show that the average diameter of randomly chosen trees of size n is O(log n). Therefore, the authors claim the average message complexity to be O(log n)

Dining Philosopher Algorithm

Eating rule: A process can eat only when it is a source
Edge reversal: After eating, reverse orientations of all the outgoing edges



(a) (b) (c)

Dining Philosphers Algorithm

var

$hungry, eating, thinking$: boolean;

$fork(f)$, $request(f)$,$dirty(f)$: boolean;

initially

1. All forks are dirty.
2. Every fork and request token are held by different philosophers.
3. H is acyclic.

To request a fork:

if $hungry$ and $request(f)$ and $\neg fork(f)$ **then**

send request token for fork f;

$request(f) := false$;

Releasing a fork:

if $request(f)$ and $\neg eating$ and $dirty(f)$ **then**

send fork f;

$dirty(f) := false$;

$fork(f) := false$;

Upon receiving a request token for fork f:

$request(f) := true$;

Upon receiving a fork f:

$fork(f) := true$;

```java
public class DinMutex extends Process implements Lock {
    public synchronized void requestCS() {
        myState = hungry;
        if (haveForks()) myState = eating;
        else
            for (int i = 0; i < N; i++)
                if (request[i] && !fork[i]) {
                    sendMsg(i, "Request"); request[i] = false;
                }
        while (myState != eating) myWait();
    }
    public synchronized void releaseCS() {
        myState = thinking;
        for (int i = 0; i < N; i++) {
            dirty[i] = true;
            if (request[i]) { sendMsg(i, "Fork");  fork[i] = false; }
        }
    }
    boolean haveForks() {
        for (int i = 0; i < N; i++)
            if (!fork[i]) return false;
        return true;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("Request")) {
            request[src] = true;
            if ((myState != eating) && fork[src] && dirty[src]) {
                sendMsg(src, "Fork");  fork[src] = false;
                if (myState == hungry){
                    sendMsg(src, "Request"); request[src] = false;
                }
            }
        } else if (tag.equals("Fork")) {
            fork[src] = true; dirty[src] = false;
            if (haveForks()) { myState = eating; notify();   }
}   }   }
```

Drinking Philosophers Algorithm

var

$thirsty, drinking, tranquil$: boolean;
$bot(b)$, $request(b)$,$need(b)$: boolean;
initially
bottle and its request token are held by different philosophers.

To request a bottle b:
if $thirsty$, $request(b)$, $need(b)$, $\neg bot(b)$ **then**
send request token for bottle b;
$request(b) := false$;

To release a bottle b:
if $request(b), bot(b)$ **then**
if $\neg$ [$need(b)$ and $((state = drinking)$ or $fork(f))$] **then**
send bottle b;
$bot(b) := false$;

Upon receiving a request token for bottle b:
$request(b) := true$;

Upon receiving a bottle b:
$bot(b) := true$;