MAEKAWA'S DISTRIBUTED MUTEX ALGORITHM

Reference: 1. Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems,

2. George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012

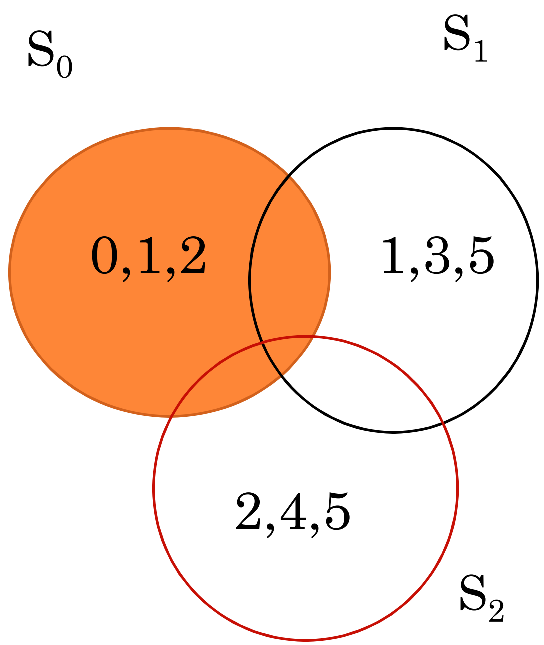
MAEKAWA'S ALGORITHM

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

$i \in S_i$

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

- **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 timestamped ***request*** to each process in $\mathbf{S_i}$.
2. Request received → send ***reply*** to process with the *lowest timestamp*. Thereafter, "**lock**" (i.e. **commit**) yourself to that process, and keep others waiting.
3. Enter CS if you receive an *reply* 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 reply to the next process with the lowest timestamp.

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

$S_i \cap S_j \neq \emptyset$ implies there is a process **k** $\in S_i \cap S_j$

Process **k** will **never** send reply 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! Assume 0, 1, 2 want to enter their critical sections.

From $\mathbf{S_0}$= **{0,1,2}**, **0,2** send *reply* to **0**, but **1** sends *reply* to **1**;

From $\mathbf{S_1}$= **{1,3,5}**, **1,3** send *reply* to **1**, but **5** sends *reply* to **2**;

From $\mathbf{S_2}$= **{2,4,5}**, **4,5** send *reply* to **2**, but **2** sends *reply* to **0**;

Now, 0 waits for 1 **(to send a release),** 1 waits for 2 **(to send a release),** , and 2 waits for 0 **(to send a release),** . So deadlock **is** possible!

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

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

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

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

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

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

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

Maekawa's algorithm-Version 2

Avoiding deadlock

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

Version 2 uses three ***additional*** messages:

- ***failed***
- ***inquire***
- ***yield***

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 ***reply*** and set lock as usual.
- If lock is set and a request with a 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 ***yield***. The recipient resets the lock.

S_0 = {0, 1, 2}

S_1 = {1, 3, 5}

S_2 = {2, 4, 5}

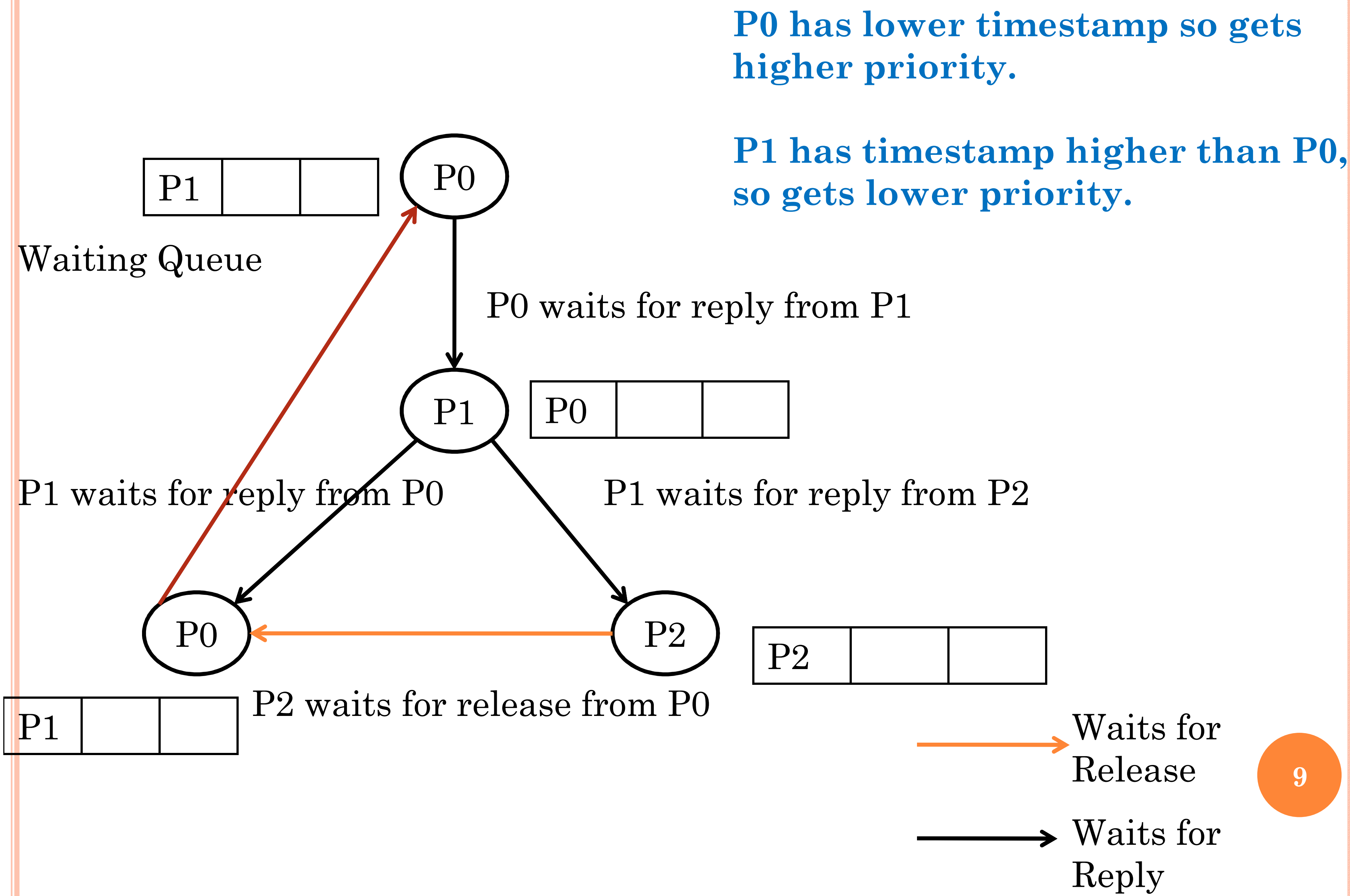
S_3 = {0, 3, 4}

S_4 = {1, 4, 6}

S_5 = {0, 5, 6}

S_6 = {2, 3, 6}

Resolving Deadlock



Maekawa's algorithm-version 1

ME2. No deadlock. Unfortunately deadlock is possible! Assume 0, 1, 2 want to enter their critical sections.

Now, 0 waits for 1 **(to send a release),** 1 waits for 2 **(to send a release),** , and 2 waits for 0 **(to send a release),** . So deadlock **is** possible!

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

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

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

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

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

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

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

low priority **high priority**

i **j** **k**

P1 blocks → P2 blocks → P0 (P0 locked by P1)

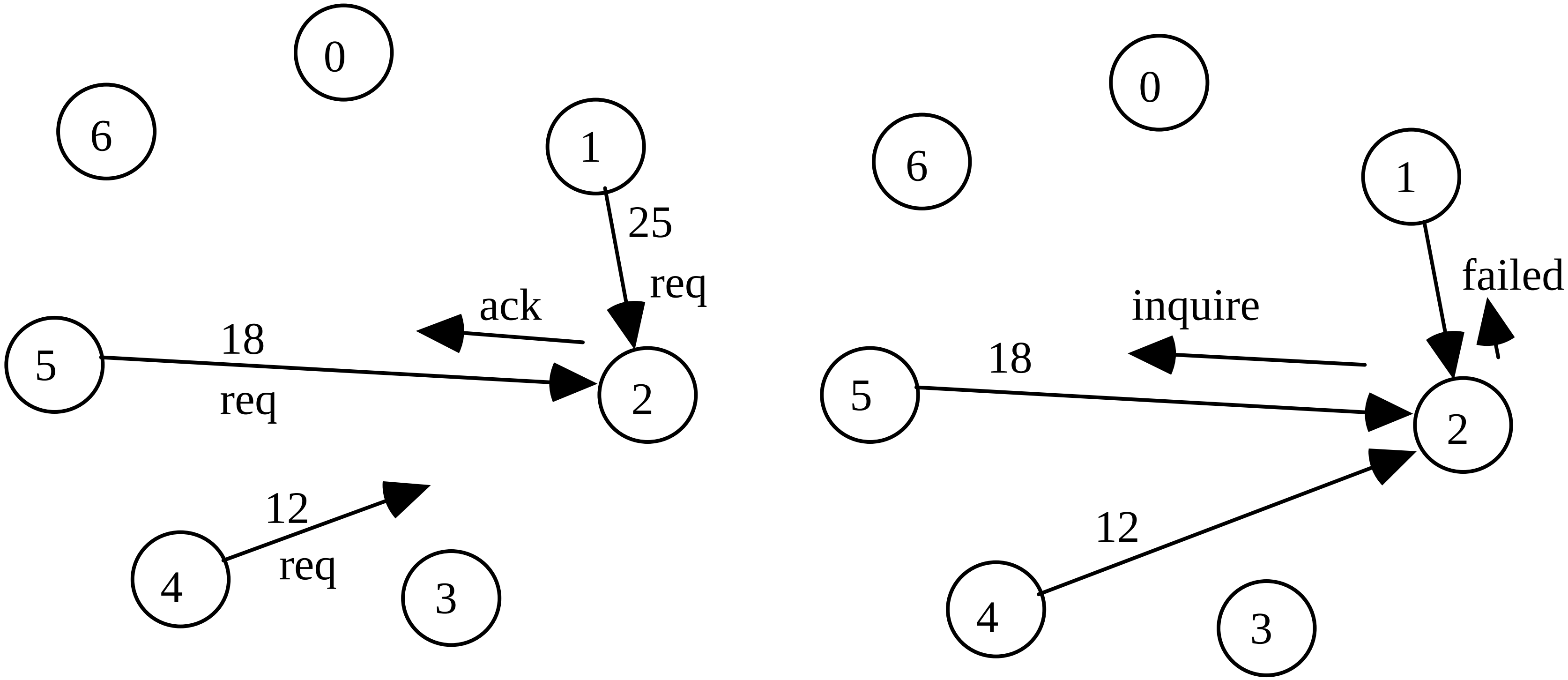
P2 sends failed to P1 ← **P2 sends Inquire to P0** →

P0 sends yield to P2 ←

P2 sends Grant to P0 →

(P0 after receiving grant from P2, P0 replies to its waiting set of process with high priority, say here it is to P1)

Maekawa's algorithm-Version 2



Maekawa's Algorithm

- state = Released, voted = false
- enter() at process P*i*:
  - state = Wanted
  - Multicast Request message to all processes in V*i*
  - Wait for Reply (vote) messages from all processes in V*i* (including vote from self)
  - state = Held
- exit() at process P*i*:
  - state = Released
  - Multicast Release to all processes in V*i*

Maekawa's Algorithm

- When P*i* receives a Request from P*j*:

if (state == Held OR voted = true)
queue Request
else
send Reply to P*j* and set voted = true

- When P*i* receives a Release from P*j*:

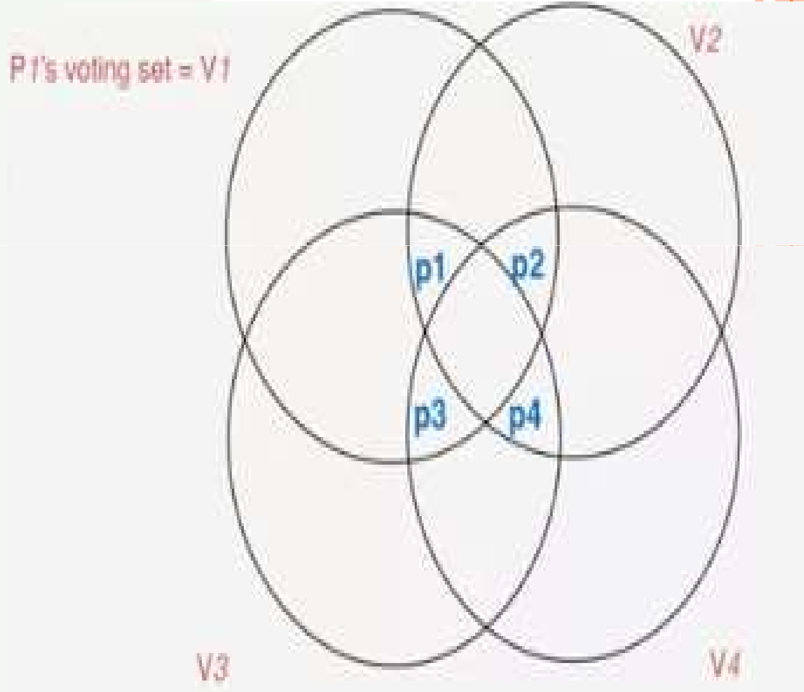
if (queue empty)
voted = false
else
dequeue head of queue, say P*k*
Send Reply *only* to P*k*
voted = true

Safety

- When a process P_i receives replies from all its voting set V_i members, no other process P_j could have received replies from all its voting set members V_j
  - V_i and V_j intersect in at least one process say P_k
  - But P_k sends only one Reply (vote) at a time, so it could not have voted for both P_i and P_j

LIVENESS

- A process needs to wait for at most (*N-1*) other processes to finish CS
- But does not guarantee liveness
- Since can have a *deadlock*
- Example: all 4 processes need access
  - P1 is waiting for P3
  - P3 is waiting for P4
  - P4 is waiting for P2
  - P2 is waiting for P1
  - No progress in the system!
- There are deadlock-free versions



Performance of Maekawa's Algorithm

- Bandwidth
  - $2\sqrt{N}$ messages per enter()
  - $\sqrt{N}$ messages per exit()
  - Better than Ricart and Agrawala's ($2*(N-1)$ and $N-1$ messages)
  - $\sqrt{N}$ quite small. N ~ 1 million => $\sqrt{N}$ = 1K
- Client delay: One round trip time
- Synchronization delay: 2 message transmission times