Embedded Operating System
Mutual exclusion and deadlocks

Graziano Pravadelli

Mutual exclusion

Mutal exclusion

Assumptions:

- *n* processes; each process P_i running on a different processor
- Each process has a critical section to be executed in mutual exclusion

Requirements

- If P_i it is executing its own critical section, there is no other process P_j which is executing its own critical section

Two types of approaches

- Centralized
- Distributed

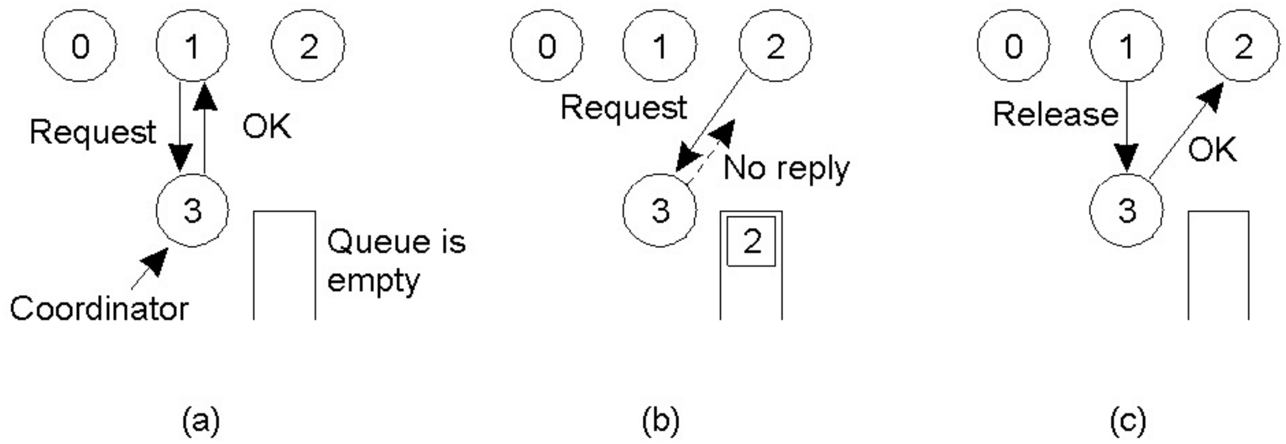
Centralized approach

A process takes care of coordinating accesso to critical section (CS)

- A process that wants to enter the CS sends a ***request*** message to the coordinator
- The coordinator decides which process enters the CS and sends a ***reply*** message
- The process that receives the ***reply*** enters the CS
- When it exits from the CS, it sends a ***release*** message to the coordinator

Three messages for each access in the CS

Centralized approach – Example



Centralized approach – Pros & Cons

Advantages

- Simple (3 messages for each request)
- It guarantees the three conditions (mutual exclusion, progress, limited waiting) without deadlock/starvation
- Suitable for all kinds of resources

Disadvantages

- The coordinator is a critical point
  - Fault tolerance (if it crashes?)
  - Performances (possible bottleneck)

Distributed approach [Ricart & Agrawala 81]

- Based on broadcast and on the total ordering of events in the system
- Given a couple of events the algorithm uniquely defines which has occurred before
- We can use Lamport's timestamps

Distributed approach

When a process P_i wants to enter the CS

- It generates a timestamp TS
- It sends *request (Pi ,CSk ,TS)* to all processes
  - The send of messages is reliable (ACK for each message)

When a process P_j receives a *request* message

- It can
  - send a *reply (OK)* message
  - or put the request in a queue

When a process P_i receives a *reply (OK)* message from all other processes

- It can enter the CS

When P_i leaves the CS

- send a *reply (OK)* message to all queued requests

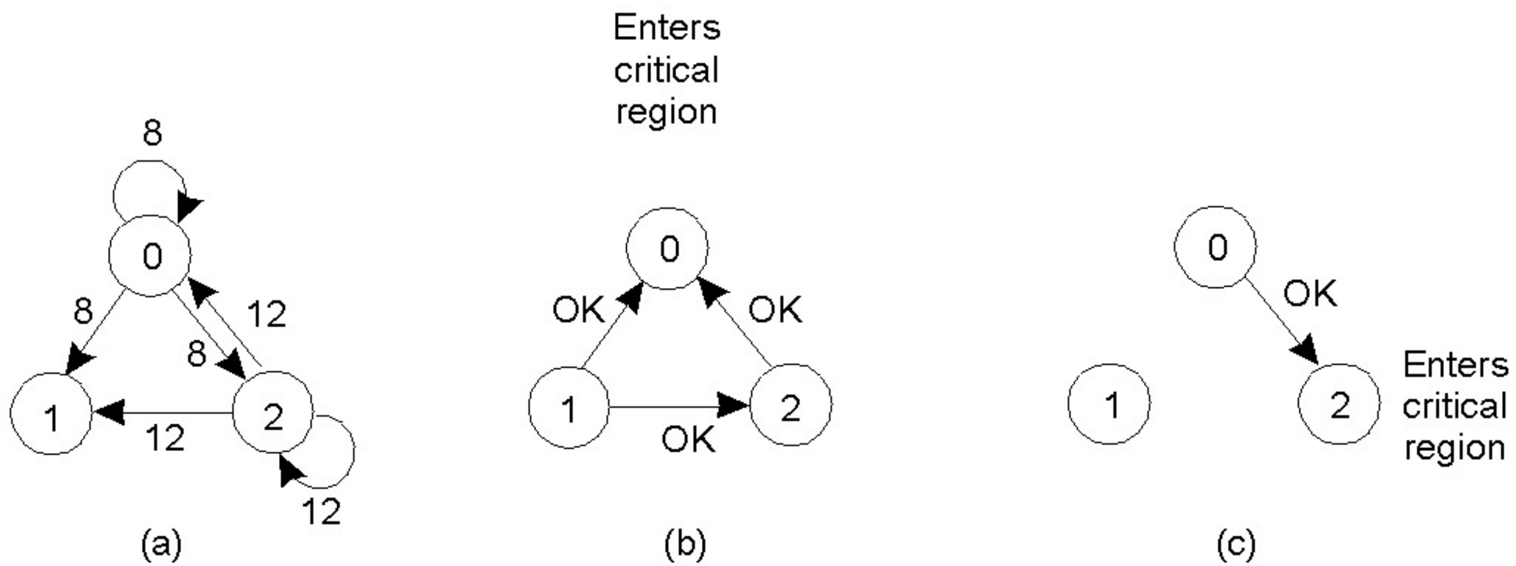
Distributed approach

Does P_j send a *reply* message or not?

When P_j receives *request(P_i, CSk, TS)*

- If P_j is in CS, it enqueues the *reply* (OK) to P_i
- If P_j is not in CS and it does not want to enter CS, it sends *reply* (OK) to P_i
- If P_j wants to enter CS, it compares the timestamp of its own request TS_j with TS:
  - If TS_j>TS, it sends the *reply* (OK) to P_i (P_i has requested before P_j)
  - Otherwise, it enqueues the *reply* (OK)

Distributed approach – Example



Distributed approach – Pros & Cons

Advantages

- Mutual exclusion guaranteed without deadlock/starvation
- 2(n-1) for each access in the CS
- No a single critical point

Disadvantages

- *n* critical points
  - Crash of a process = No access to the CS
- *n* bottlenecks
- Need group communication primitives

Variant

- Instead of enqueuing messages, explicit rejection message
- It acts as ACK and solves the problem of *n* critical points

Token-ring algorithm

Token originally is in process 0

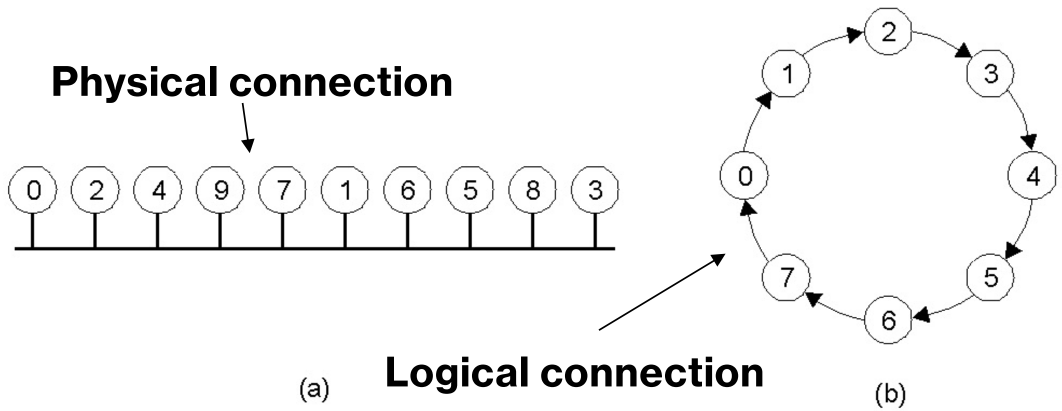
It pass through point-to-point messages from process i to i+1 (module N)

When a process receives the token

- if it wants to enter the CS, it enters the CS. When it leaves the CS, it passes the token to the next process
- if it does not want to enter the CS, it immediately passes the token

Token-ring algorithm – Example

- It assumes a specific logical organization (ring) of processes
- Based on the rotation of a token between various processes



Token-ring algorithm – Pros & Cons

Advantages

- No deadlock/starvation
- The impact of node failure is limited
  - In the event of a crash, it is sufficient to "bypass" (updating the network structure)
  - We notice the crash if the ACK of the token does not arrive

Disadvantages

- Loss of the token
  - How to detect it (long CS vs. lost token)?

Deadlock

Definition

Processes use resources

Sequence of use

- Request
- [Wait]
- Use
- Release

DEADLOCK
A set of processes is in *deadlock* when each process is waiting for an event that can be caused by a process of the same set

Required conditions

Mutual exclusion

- At least one resource must be non-shareable

Hold and Wait

- There must be a process that holds a resource and expects to acquire another resource, held by another

No preemption

- Resources can only be released “voluntarily” by the process that is using them

Circular waiting

- There must be a set of processes that cyclically wait the release of a resource

Deadlock management

- Same options of centralized systems
  - Ostrich algorithm
  - **Static prevention**
    - **Avoid circular waiting**
    - **Distributed**
  - Dynamic prevention based on resource allocation
  - **Detection and recovery**
    - **Centralized**
    - **Distributed**

RAG model

Resource allocation graph (RAG) G(V,E)

- V = nodes
- E = edges

Nodes

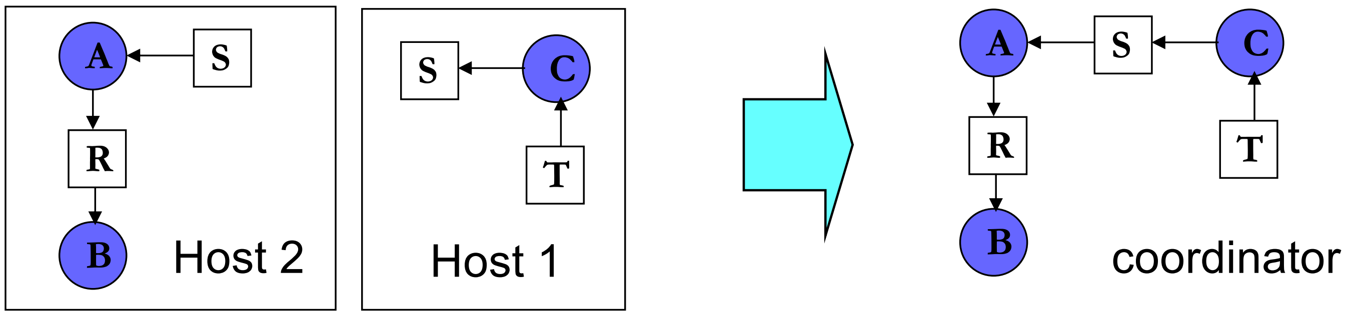
- Circles = processes (CPU, I/O, memory)
- Rectangles = resources
  - In the rectangles there are as many "•" as the instances of the corresponding resource

Edges

- From processes to resources: process requires resource
- From resources to processes: process holds resource

Detection – Centralized approach

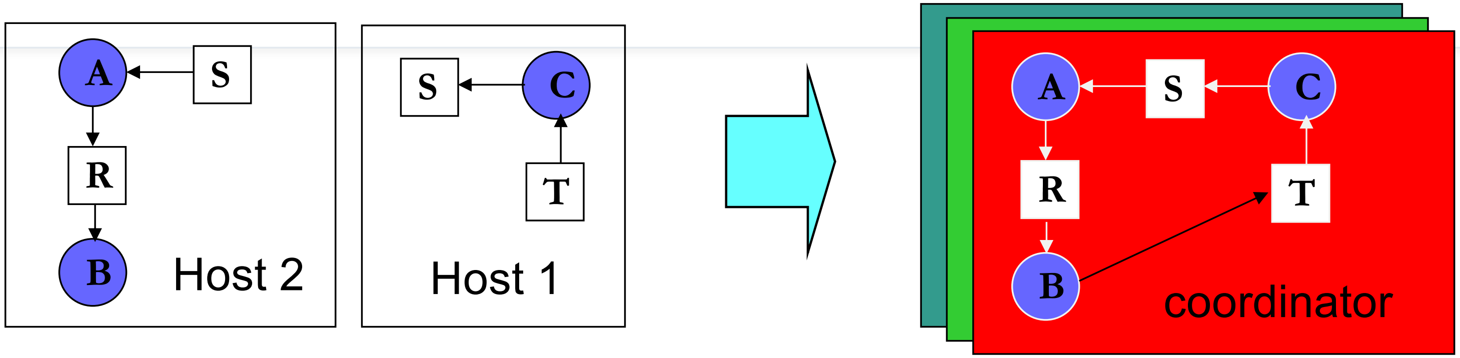
- Emulation of the not distributed case
- A coordinator maintains a global RAG
  - Union of individual RAGs of individual hosts
  - If it finds a cycle, it kills a process



Detection – Centralized approach

- How and when to build the global RAG? (when send it to the administrator?)
  - After each addition/removal of an arc in the local graphs
  - Periodically
  - At the request of the coordinator

False deadlocks – Example



- B releases R and requests T (legal operation)
- What is the order of the messages?
  - Msg1: Host 2 announces to the coordinator that B releases R
  - Msg2: Host 1 announces to the coordinator that B requests T
- If messages arrive in opposite order → false deadlock

False deadlocks – Solution

- Global graph must be updated independently from the order in which messages are received
  - Association of (Lamport) timestamps to modifications of local graphs
  - When the coordinator receives a change with timestamp T, the coordinator itself urges the sending of any other changes in progress with timestamp smaller than T

Detection – Distributed approach
[Chandy&Misra&Haas 83]

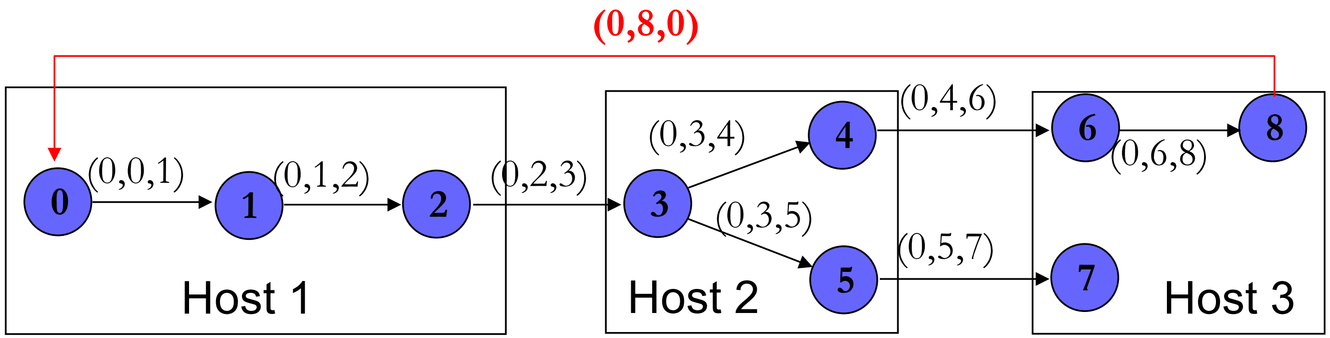
Invoked when a process waits for some resource

- It sends probe messages to processes that hold the resource
  - **Message = (id blocked proc, id sender proc, id destination proc)**
- The probe is propagated to processes that are also waiting
  - If the receiver is waiting, it forwards the probe by updating the sender and receiver
- If the message reaches the sender → deadlock

Recovery

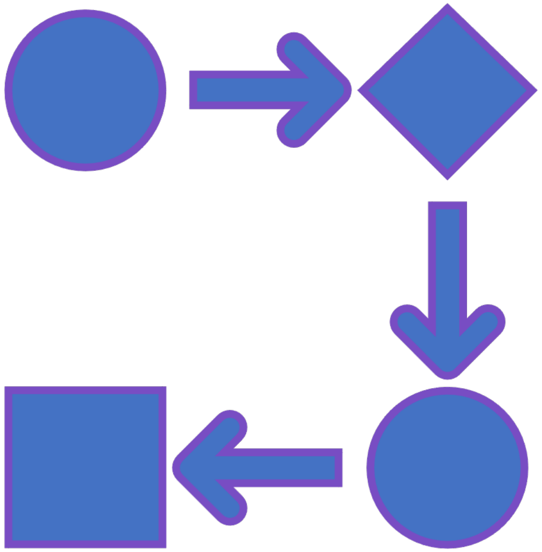
- Killing the probe initiator → possible many kills if many processes start the algorithm at the same time
- Killing the process with the higher id

Distributed approach – Example



Static prevention based on resource ordering

- Preventing the circular waiting condition by definition of a global order among resources in the system
  - E.g.: assigning a single number to each resource
    - A process can request a resource with a number *n* only if it does not hold resources with numbers greater than *n*
- Simple and low-cost implementation



Static prevention based on timestamp

Timestamp used as a priority value for each process

Use of timestamps to regulate the wait of a process

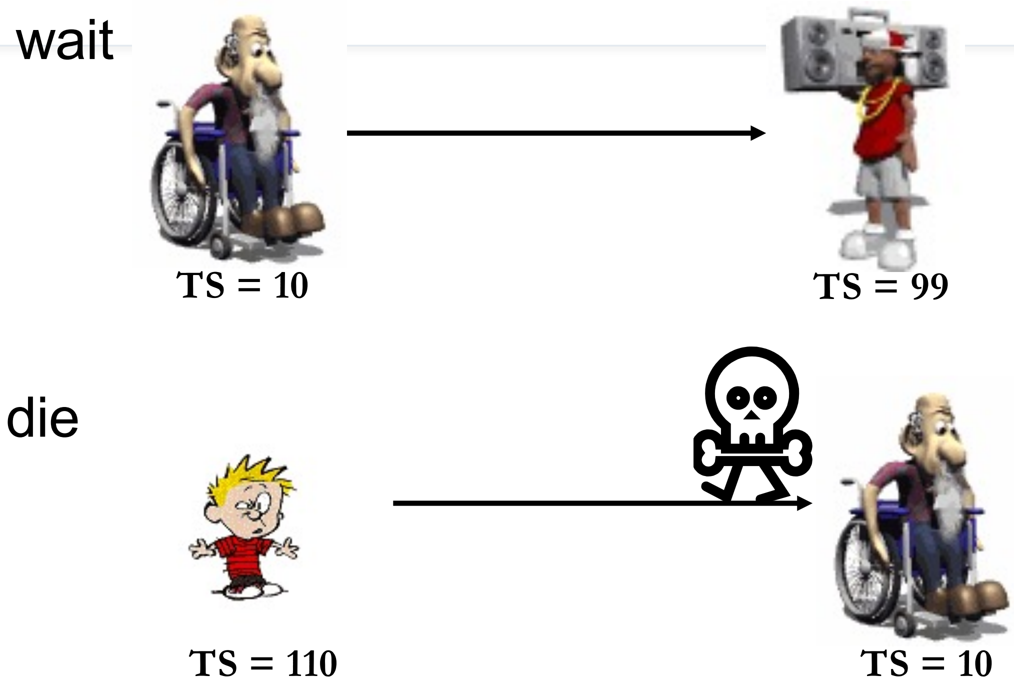
- Non-preemptive scheme **(wait-die)**
- Preemptive scheme **(wound-wait)**

The use of timestamps prevents from starvation

Wait-Die scheme

- If a process P_i needs a resource which is held by a process P_j
  - if $TS(P_i) < TS(P_j)$ ➡ P_i goes on wait (wait)
  - if $TS(P_i) > TS(P_j)$ ➡ rollback of P_i (die)

Wait-die – Example

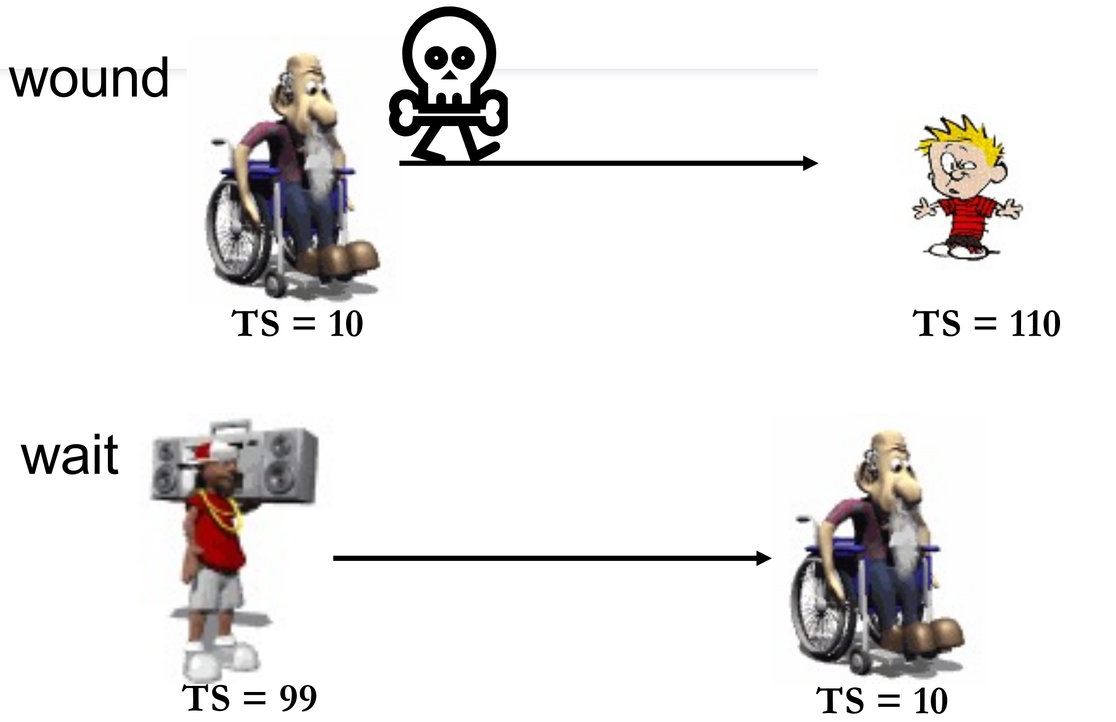


Always in the direction of increasing timestamps
of increasing timestamps

Wound-Wait scheme

- If a process P_i needs a resource which is held by another process P_j
  - if $TS(P_i) < TS(P_j)$ ➡ P_j is wounded and killed (wound)
  - if $TS(P_i) > TS(P_j)$ ➡ it goes on wait (wait)

Wound-wait – Example



Always in the direction of decreasing timestamps

Static prevention based on timestamp

- Wait-die tends to kill many young processes
  - The old respectful man waits
  - The presumptuous young man is repeatedly killed
- Wound-wait kills only once
  - The wise old man pre-empts the young man
  - The young man re-starts and waits patiently