Operating Systems

Deadlock

Lecturer:
William Fornaciari
Politecnico di Milano
william.fornaciari@elet.polimi.it
Homel.deib.polimi.it/fornacia

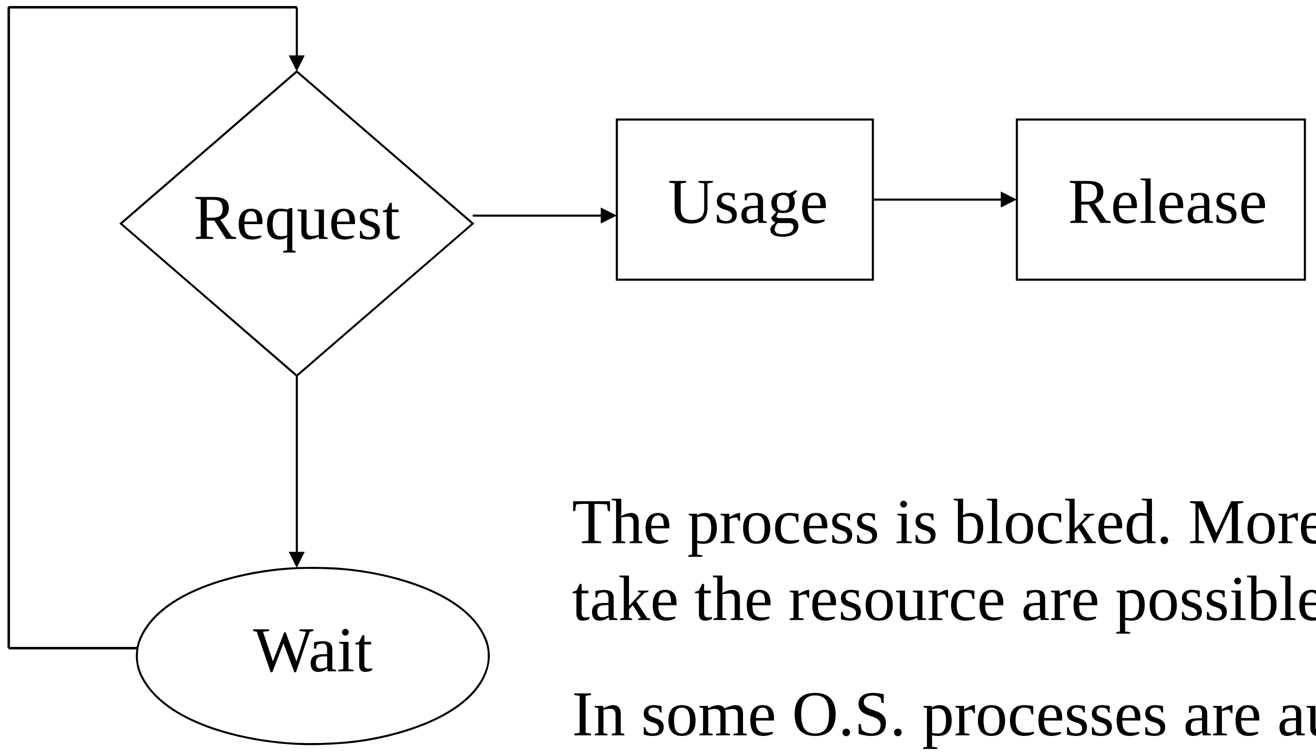
Summary

- What is a resource
- Conditions for deadlock
- Directed graph modeling of deadlock
- Detection, avoidance and prevention
- Starvation
- Deadlock in distributed systems

Resource

- Any hw or sw component (e.g. a disk drive, a license, a locked record) with exclusive access from a process
  - *preemptable*: can be taken away causing no disaster (e.g.memory with process swapping)
  - *non-preemptable*: computation fails if the resource is taken away from its owner (e.g. switching printer use among processes causes garbled output)

Use cycle of a resource



The process is blocked. More attempts to take the resource are possible.

In some O.S. processes are automatically blocked and awakened when the resources become available

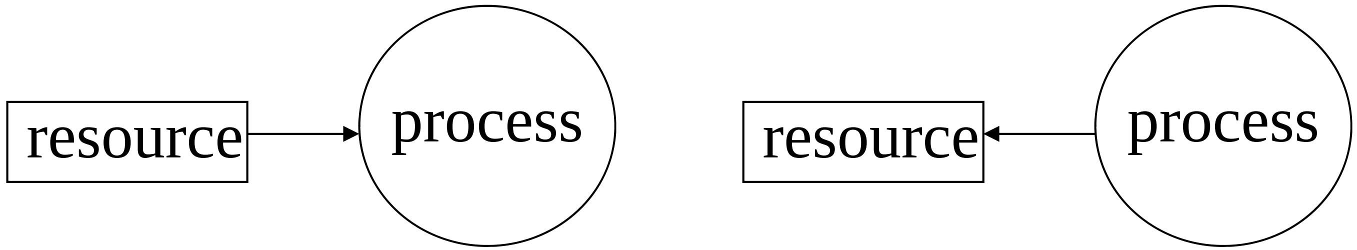
Example of deadlock

- Processes P1, P2 need to use in exclusive manner resources R1 (lpr) and R2 (tape) that are not preemptable
- Possible situation
  - P1 request to use R1 is granted
  - P2 request to use R2 is granted
  - P1 req.to use R2: P1 is blocked waiting for R2
  - P2 req. to use R1: P2 is blocked waiting for R1
  - P1 and P2 remain blocked forever with no chance to modify the situation

Deadlock: definition and conditions

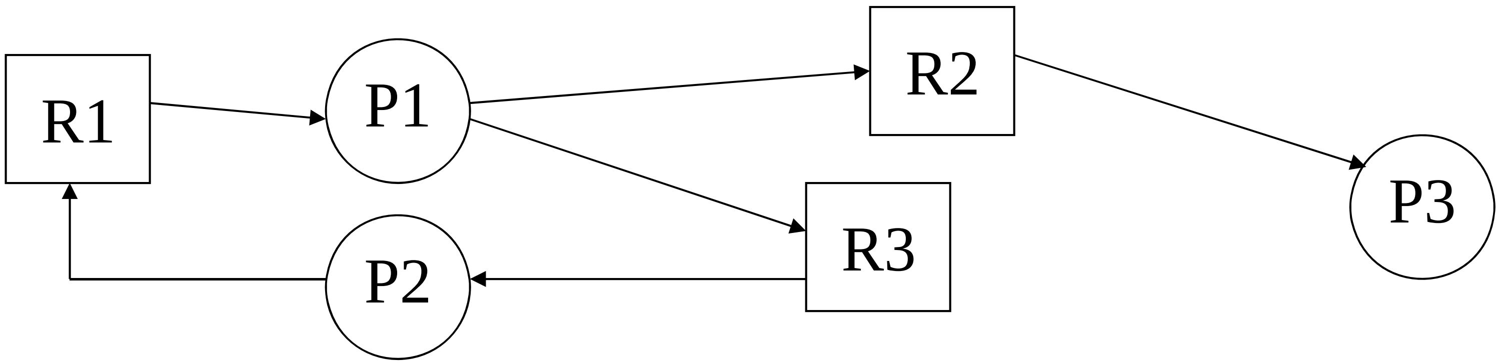
- A set of processes is deadlocked if each of them is waiting for an event that only another process in the set can cause
- Conditions necessary for having deadlock
  - **Mutual exclusion**: resources are either available or assigned to only one process
  - **Hold & Wait**: processes already holding resources can request for new ones
  - **No preemption**: res. previously taken can be released only spontaneously by the holder
  - **Circular wait**: there must be a circular chain of processes, each waiting for the resource held by the next

Directed graph modeling



P holds R   P is blocked, waiting for R

- Directed graph loops are deadlocks
- The presence of deadlocks can be monitored by updating the graph each time a request is issued
- E.g. P1, P2, R1, R3 are in deadlock



Deadlock management

- Ignore the problem
- Detection and recovery
- Dynamic avoidance
- Prevention

Ignore the problem

- Careful tradeoff between the cost of possible damages and deadlock rate. Applicable if DL is infrequent and no *mission-critical* systems
- Ex: in Unix there exists upper bounds on process tables, i-node tables, ... which potentially could cause e.g. endless loops *forking-and-failing*
- Ignoring the problem avoid putting restrictions, such as forbidding dynamic process forking

Detection and Recovery

- No prevention from deadlock occurring, but there exist methods for detecting it and recovering
  - Detection with one resource per type
    - A resource graph is built and updated as a new request comes out. Graph inspection allows to discover deadlock conditions along with pertaining processes and resources
  - Detection with multiple resources per type
    - *n* processes P1…Pn
    - *m* classes of resources with *Ei* cardinality $(1 \leq i \leq m)$

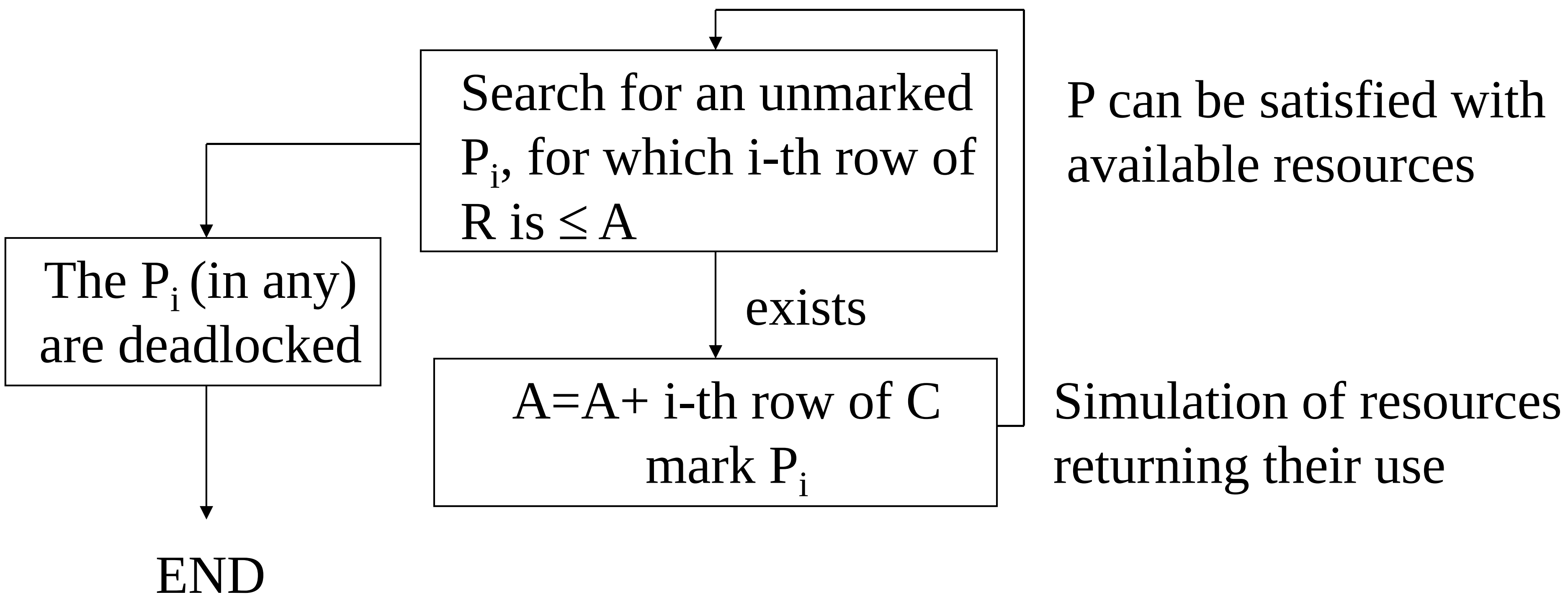
Multiple Res. Detection (1)

- E $[E_1,..E_j,..E_m]$ vector of existing resources
- A $[A_1,..A_j,..A_m]$ vector of available resources
- C $[c_{ij}]_{nxm}$ current allocation matrix. c_{ij} is the # of instances of j class currently held by P_i
- R $[r_{ij}]_{nxm}$ request matrix. r_{ij} is the # of resources of j class requested by P_i
- Every resource is either available or allocated

$$\sum_{i=1}^{n} c_{ij} + A_j = E_j$$

Multiple Res. Detection (2)

- Note: $A \le B$ iff each element of A is less than equal to the corres. of B ($A_i \le B_i$, $\forall i$, $1 \le i \le m$)



Recovery using preemption

- Resources can be taken away from its current process owner (now marked as runnable) and allocated to another process
- Choice of the resource depends on how easily can it be taken back
- Recovery can be frequently impossible
- Frequent manual intervention

Recovery through rollback

- Processes are *checkpointed* periodically. A ceckpoint is a memory image + state of the resources currently allocated to the process
- After deadlock detection, the necessary resources are identified. Each process holding resources is rolled-back to a point in time before their acquisition. This now free resources are then allocated to processes so to solve the deadlock
- Work carried out until checkpoint is lost. Tradeoff between freq of deadlocking and checkpoint “density”

Recovery through killing processes

- Processes belonging to deadlock loop are (incrementally) killed
- Processes not belonging to the deadlock loop, but holding resources necessary to the deadlocked processes, are killed
- Candidates
  - P re-startable without side-effects (e.g. compilers)
  - P incurring in more than one deadlock loop
  - P who did little work

Deadlock avoidance

- Safe state
  - There is no deadlock and it exists a way to satisfy pending requests executing in some order the processes
- Unsafe state
  - Processes can go on, but it is not granted their terminations (different from deadlock)
- In general the system allows requested allocation of resources only if it remain in a safe state

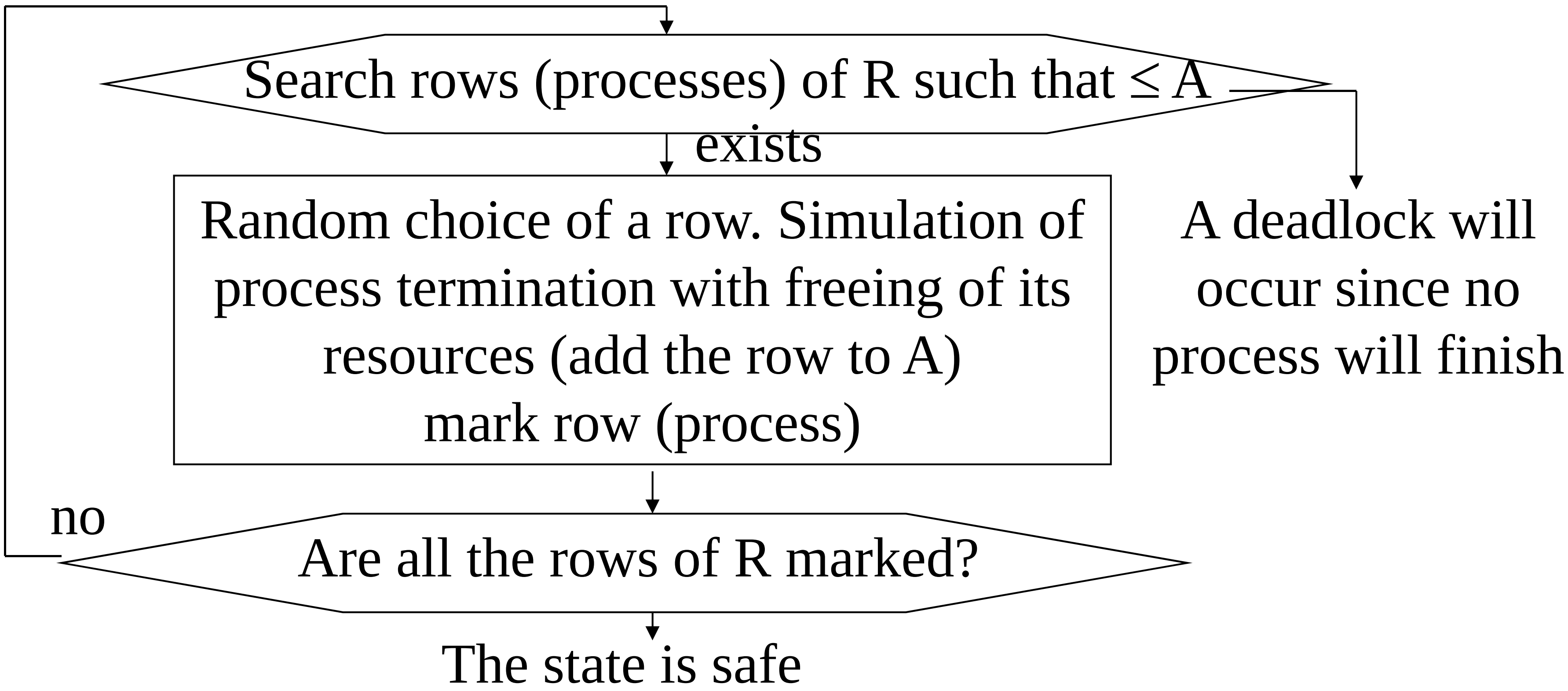
Banker’s algorithm for single resources

- Each P has a given #max of resources to be taken
- Requests are considered as they arrive. If they take to system in an unsafe state, P is moved in a waiting state
- Problems
  - predictability of the necessary resources
  - #Pi can change dynamically
  - availability of resources can vary (e.g. after a fault)

Banker's for multiple resources (safe state)

- E vector existing res. P vector taken res.
- A vector available res. R request matrix

E-P=A



Deadlock prevention

- Attempt to ensure that at least one of the conditions will never be satisfied
  - Mutual exclusion
    - make sharable resources (when possible)
    - spooler: a daemon process is the only manager of a device. It queues the requests
    - not all the devices can be spooled
  - Hold & Wait
    - P must ask before execution the necessary resources, otherwise it is suspended
    - It is hard to know in advance the needs; possible non optimal use of resources due to conservative overbooking
    - variant: before to issue a request, P temporary release those held, then it attempt to take all requests atomically

Deadlock prevention (2)

- No preemption
  - Applicable only in particular not frequent cases
- Circular wait condition
  - Resources are numerically ordered. Each process can hold only one resource at a time
  - The allocation graph is acyclic if the processes issue request according to such ordering: at any time, a P cannot wait for a already assigned resource
  - In a few cases it is possibile to discover a ordering satisfying the need of all processes
  - The use on only one resource at a time makes impossible simple actions like tape-disk copy

Deadlock prevention (3)

- Two-phase locking (DB)
  - ph1: the process tries to lock all records, one at a time
  - ph2: DB record are updated, then locks released
  - if during ph1 some record are busy, locks cumulated are released and ph1 restarted
  - Applicable to processes restartable without side effects (difficult in case of write/read from net)
  - it is easy to predict in advance the resources necessary for DB operations

Starvation

- Some processes, even not involved in a deadlock, never get service
- E.g., in a print manager giving the precedence to the smallest job, processes with big job can be postponed indefinitely even though not blocked
- Typical problem related with priority policies
- Policies like FCFS (First Come First Served) or round-robin prevent it from happening

DL in distributed systems (1)

- Information are scattered over multiple computers
- Possible sources of deadlock
  - **communication**: circularity in trying to send a msg among a set of P (ex for lack of buffers)
  - **resources**: P compete for exclusive access
- Situations similar to those for single processor, only worse

DL in distributed systems (2)

- Managing strategies
  - *Ignore*: always possible
  - *Detection & Recover*: widely used
  - *Prevention*: make it structurally impossible. Applicable especially in trasactional systems
  - *Avoidance*: Careful allocation of resources. Never used, too hard to predict resources requests in advance

DL avoidance in distr. systems

- Distributed deadlock avoidance is impractical
  - Every node must keep track of the global state of the system
  - The process of checking for a safe global state must be mutually exclusive
  - Checking for safe states involves considerable processing overhead for a distributed system with a large number of processes and resources

Detection in distributed systems

- Each site only knows about its own resources
  - Deadlock may involve distributed resources
- Centralized control – one site is responsible for deadlock detection
- Hierarchical control – lowest node above the nodes involved in deadlock
- Distributed control – all processes cooperate in the deadlock detection function

Detection in distributed systems

- Managing strategies
  - normal systems
    - DL detection with process killing
  - transaction based systems
    - DL detection plus abort to restore previous state
- Centralized detection algorithm
  - It exists a coordinator machine, collecting and merging the Res./P allocation graphs of each machine
  - once a DL is detected, it kill off one processes to break it

Distributed detection 2

- Need of updating msgs
  - Each time a graph changes
  - Periodically each process send a message to report any modification (update) of previous msg (e.g. adding of a new arc)
  - Sending triggered by coordinator
- False deadlocks can appear due to msg delays or inconsistency in updating the whole graph
- Need of an (expensive) global time; If a DL is suspected, msgs are sent to the pertaining machines with *timestamping* mechanisms to get the actual up-to-date situation

Detection in distributed systems

- Hierarchical control
  - Sites have a tree organization
  - All the nodes but leaves collect information on the resource allocation of the lowest nodes
  - It is possibile to detect only deadlock in the lowest levels of the root

Distributed detection 3

- Distributed detection algorithm
  - The algorithm is invoked whenever a P has to wait for resources
  - A *probe* msg is generated to be propagated to all P_i holding resources
    - msg=(id of blocked P, id of sender P, id of receiver P)
  - When a msg arrives, the receiver P
    - if P is itself waiting for other Ri, a new msg is sent to the Pi holding Ri maintaining the first field (id_blocked P)
    - if the msg arrives back to the first sender (first field), a waiting loop exists, i.e. a DL

DL Detection: Algorithm Comparison

- Centralized
  - ☺: simple,easy to implement. Optimal resolution due to strategy based on global information
  - ☹: communication overhead. Single point of failure
- Distributed
  - ☺: robust: no single point of failure. Distribution of the work on several nodes
  - ☹: *cahotic* resolution, multiple detections, difficult to implement
- Hierarchical
  - ☺: robust: no single point of failure. If deadlocks are localized, resolution is simplified
  - ☹: difficult system configuration, in some case can be worse than distributed

Distributed DL Resolution

- The P initiating the probe commit suicide
  - If many probe are contemporaneously active it can result in a overkill
- Each P adds its id to the probe
  - Eventually, the first sender obtain a list of the deadlocked processes, and can decided which to send a kill msg

Distributed DL prevention

- **Hold-and-wait** condition can be prevented by requiring that a process request all of its required resource at one time, and blocking the process until all requests can be granted simultaneously
  - Inefficient: long waiting time before freeing and granting of resources
- **Circular wait condition**: Careful ordering of request and grant of resources should avoid presence of loops
- For systems with *global time* associated with *transaction* (T)
  - Each T has a t_{start} different from others
  - A P needing a resource held by another, check its timestamp to see which is larger, if it cannot block suicide

Suicide policies

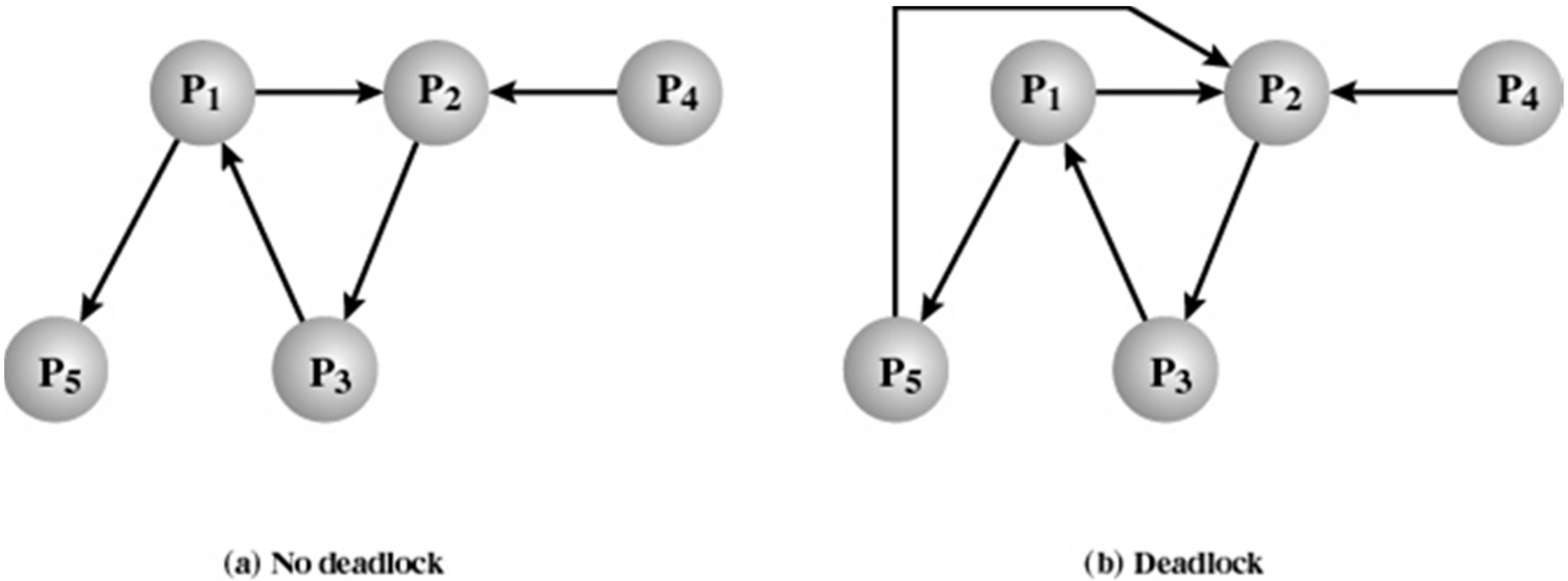
- Wait-die
  - A P can block iff it is older than the P holding Res. it is waiting for. Timestamp always increase and no loops can be generated
    - variant: a P can wait only for younger Pi
    - in general it is better to give priority to older Pi to waste less work already done
    - suitable to transactional systems which can be restarted without side-effects
- Wound-wait
  - older Pi can preempt younger ones. Young Pi can only wait for the old after their restart. Differently from wait die, the younger is not killed ma only turned in a waiting state

DL in message communication: Mutual Waiting

- Deadlock occurs in message communication when each one of a group of processes is waiting for a message from another member of the group and there are no messages in transit
- Typically a P can go on if
  - any of the message it is waiting for arrives
  - when all of them are received (not considered)
- Deadlock of a process set S
  - All the processes of S are blocked, waiting for msgs
  - S contains the entire dependency graph of all processes
  - No messages are in transit among the members of S

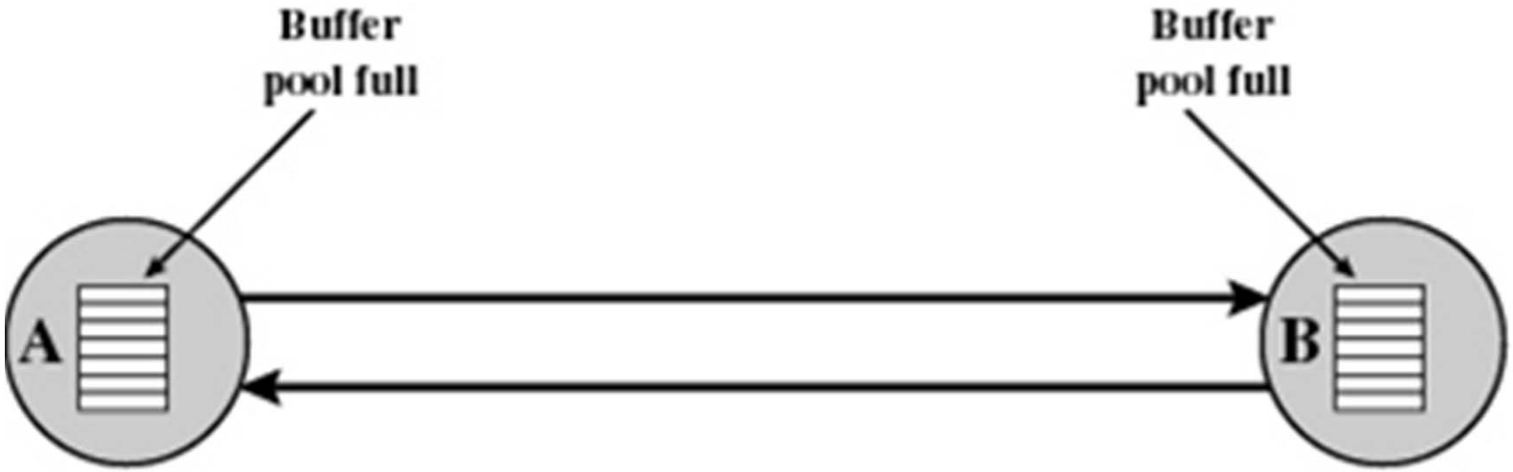
DL in message communication

- Differently from resources, in the case of messages DL occurs if the successors of a process in S belongs to S itself, i.e. there is a "*tie*" in S



DL in message communication: Unavailability of message buffers

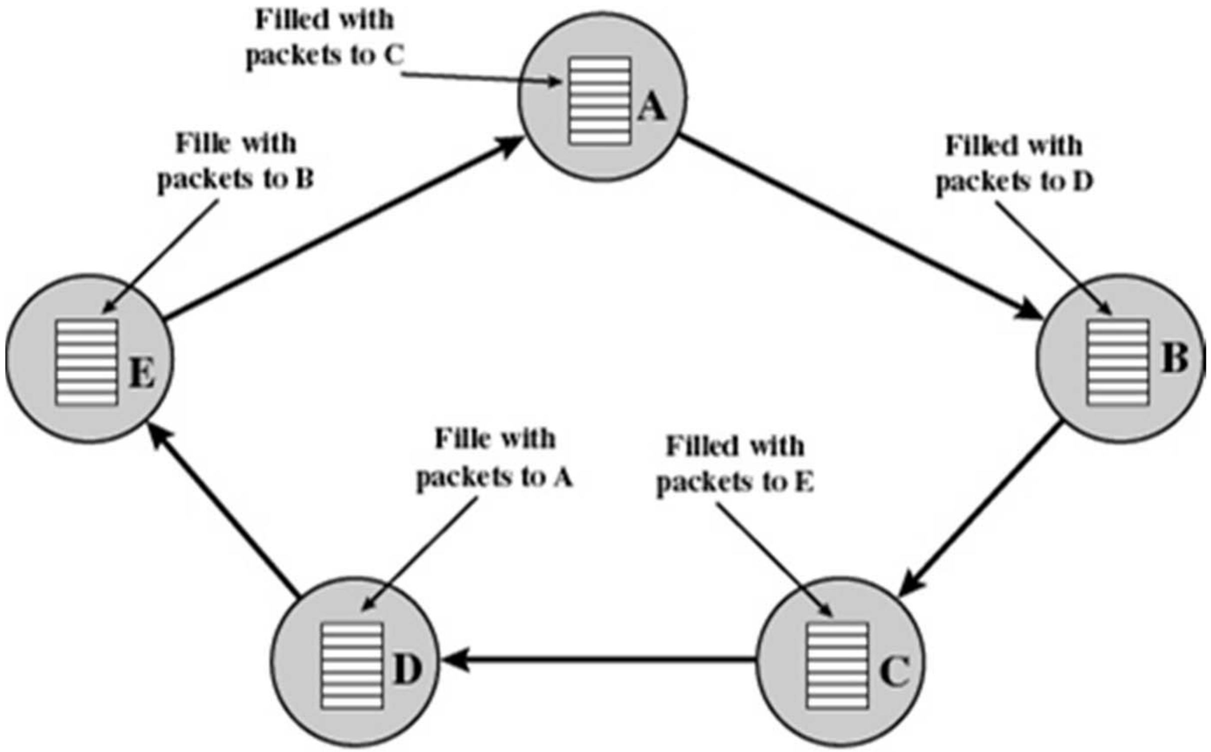
- Direct Deadlock
  - Well known in packet-switching data networks
  - Example: buffer space for A is filled with packets destined for B. The reverse is true at B



(a) Direct store-and-forward deadlock

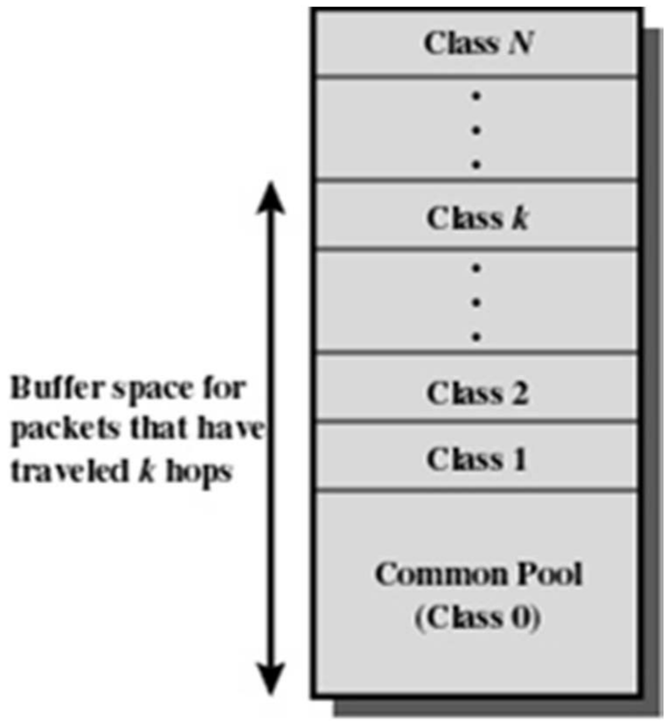
Unavailability of Message Buffers (Indirect Deadlock)

- For each node, the queue to the adjacent node in one direction is full with packets destined for the next node beyond



Prevention using Structured Buffer pool

- Buffers are hierarchically organized, for each class it is defined a minimum number of hops *k* for the packet for being stored
- Heavy load: the buffers are filled incrementally, from 0 upwards
- It can be demonstrated that this solution prevents deadlock



Evaluation of detection algorithms for distr. sys

- Conditions to be verified
  - All DL must be detected in a finite time
  - Absence of false DL (e.g. due to msg delays)
- Performance
  - DL persistency (time between detection and resolution)
  - memory and computational requirements
  - size and count of msg exchanged
- Methods
  - Analytical
  - Empirical
  - Simulation-based

DL in buffered communication

- In general, the use of finite-lenght buffers for inter-process communication is a risk of deadlock
  - If send is not blocking, than the outcoming messages must be stored in a buffer. If it is full the sender will be blocked
  - If two processing are communication via two separate buffers, and both try to send before to receive and the buffers are full, the system is in deadlock
- Possible solution
  - Define upper bounds to the number of messages exchanged between pairs of processes. This allows to allocate a sufficient amount of buffers slots
  - Problems: a priori knowledge, waste of space
  - Use of heuristics

