Deadlock

Definition and modeling

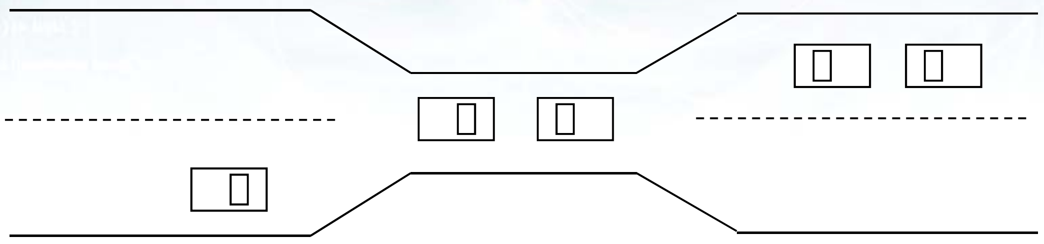
Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
  - Example: P_1 and P_2 each hold one tape drive and each needs another one.
  - Solution with 2 semaphores A and B, initialized to 1

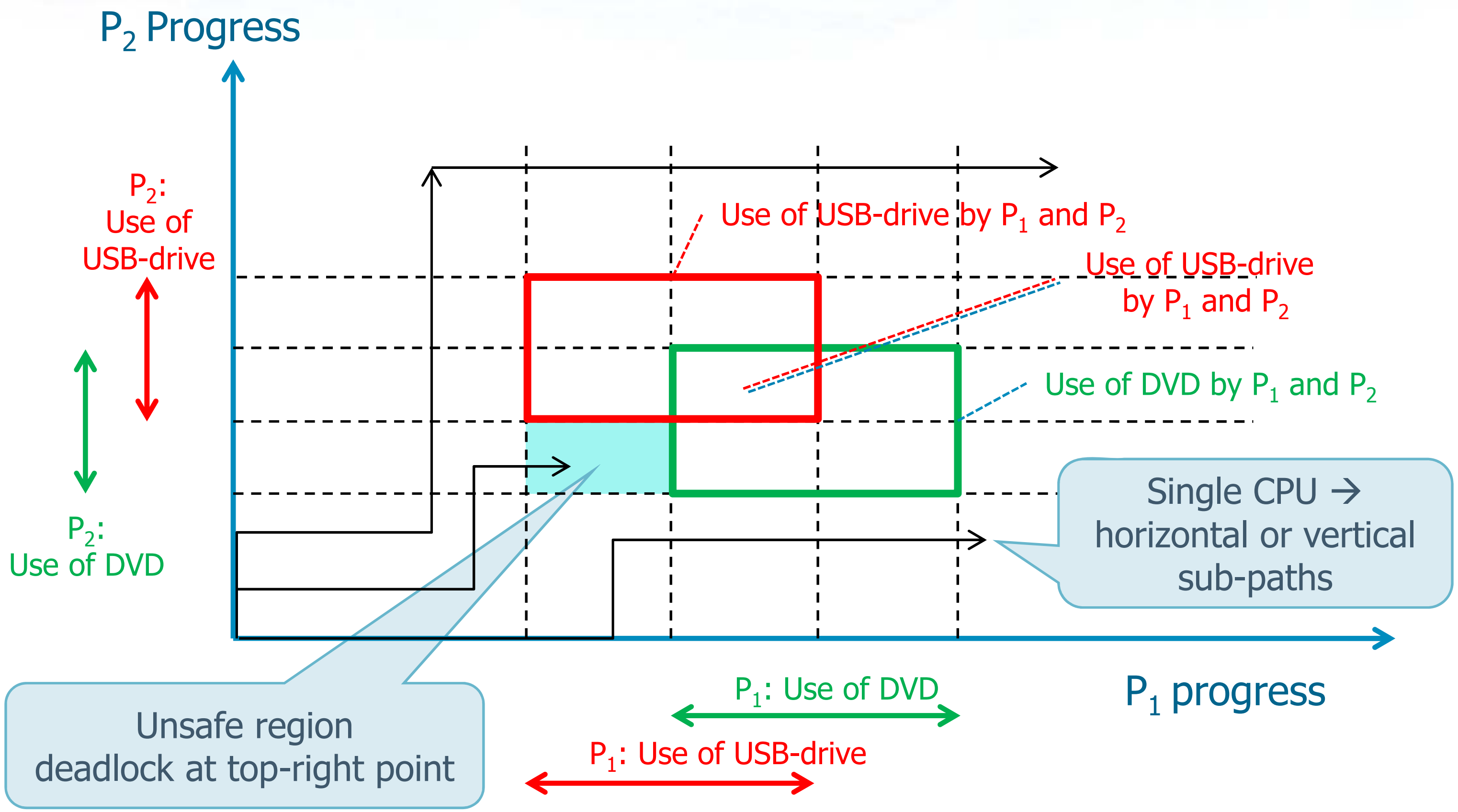
P_1	P_2
wait (A)	wait(B)
wait (B)	wait(A)

Bridge Crossing Example

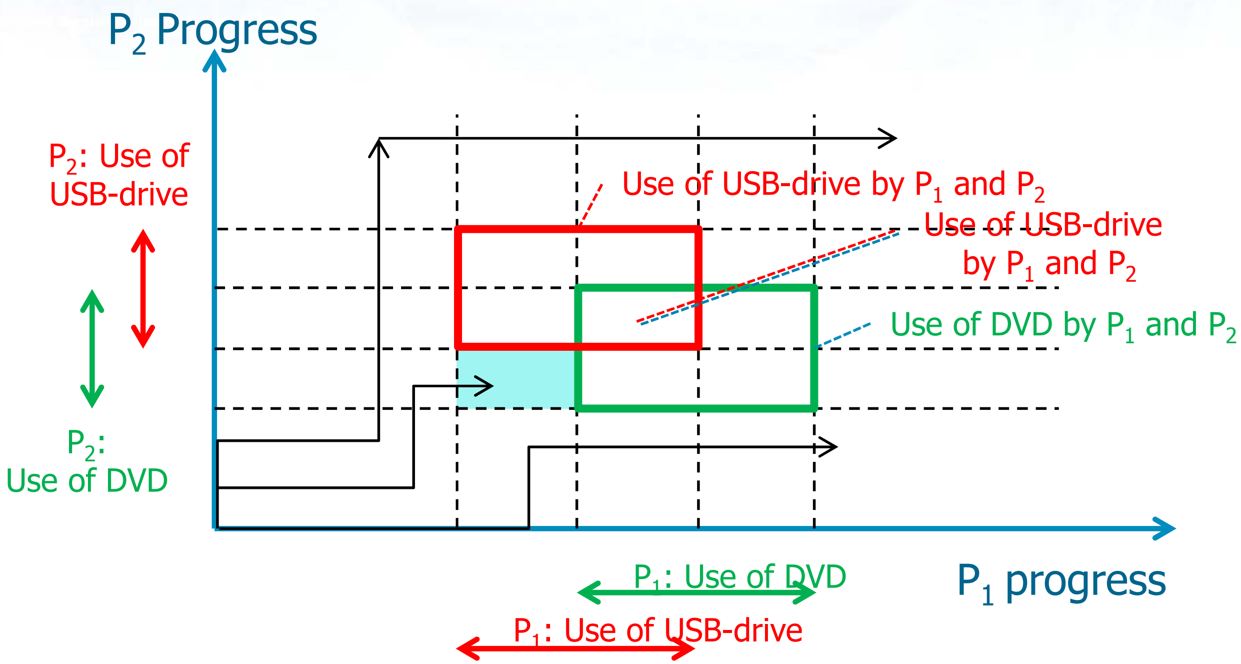


- Traffic on the bridge possible only in one direction.
- Each of the two sections of the bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car rolls back preempting resource it occupies.
- Several cars may have to roll backup if the bridge has place for several cars, and a deadlock occurs.
- Starvation is possible.

Joint progress of two processes



Joint progress of two processes



- Prevention: forbid the existence of an unsafe state
- Avoidance : forbid the entrance into an unsafe state
- Recovery: forbid the residence into an unsafe state

Deadlock

- **Deadlock**
  - A process requires a not available resource, and enter in a waiting state forever
  - The deadlock consists of
    - A set of processes that wait the occurrence of an event that can only be caused by another process of the same set
- Deadlock implies starvation not vice-versa
  - The starvation of a process implies that it waits indefinitely, but the other processes can proceed, not being deadlocked
  - All processes in deadlock are also in starvation

Necessary conditions for occurrence of a deadlock

Conditions	Description
Mutual exclusion	Only one process at a time can use a resource
Hold and wait	A process holding at least one resource is allowed to wait for acquiring additional resources held by other processes
No preemption	A resource can be released only voluntarily by the process holding it, not preempted by the system.
Circular wait	A set of waiting processes $\{P_1, P_2, \ldots, P_n\}$ such that P_1 is waiting for a resource that is held by P_2, P_2 is waiting for a resource that is held by P_3, …, and P_n is waiting for a resource that is held by P_1

Summary

- Modeling techniques
- Management strategies
  - Ignore
    - **Ignore the problem assuming the probability of a deadlock in the system is very low**
      - Method used by many operating systems, including Windows and Unix
      - Less appropriate if concurrency and complexity of the system increase
  - A priori
    - Prevention
    - Avoidance
    - In case of possibility of deadlock
  - A posteriori
    - Detect
    - Recovery
    - In case of deadlock

Modeling

- **Resource-Allocation** Graph G = (V, E)
  - Allows deadlock description and analysis
- The set of vertices V is composed of processes and resources
  - Process set $P = \{P_1, P_2, \ldots, P_n\}$
    - Each process accesses a resource via a standard protocol consisting of
      - Request
      - Utilization
      - Release

Modeling

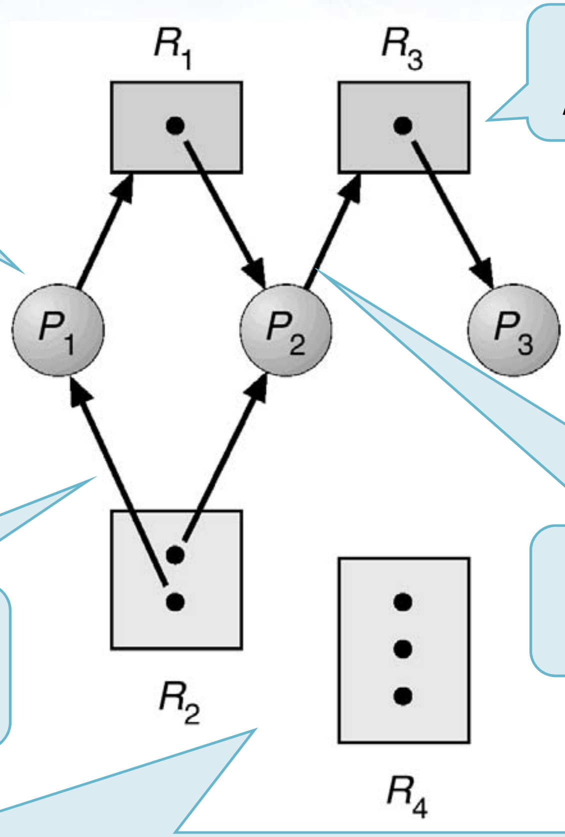
- System resource set $R = \{R_1, R_2, \ldots, R_m\}$
  - The resources are divided into classes (types)
  - Each resource type R_i has W_i instances
  - All instances of a class are **identical**: any instance satisfies a demand for that type of resource
- The set of edges E is composed of
  - Request edges
    - $P_i \rightarrow R_j$, i.e., from a process to a resource type
  - Assignment edge
    - $R_{jk} \rightarrow P_i$, i.e., from a resource to a process

Modeling

Vertices: Processes
P_1, P_2, P_3

Vertices: Resources
An instance of R_1 and R_3

P_1 holds R_2 and is waiting for R_1



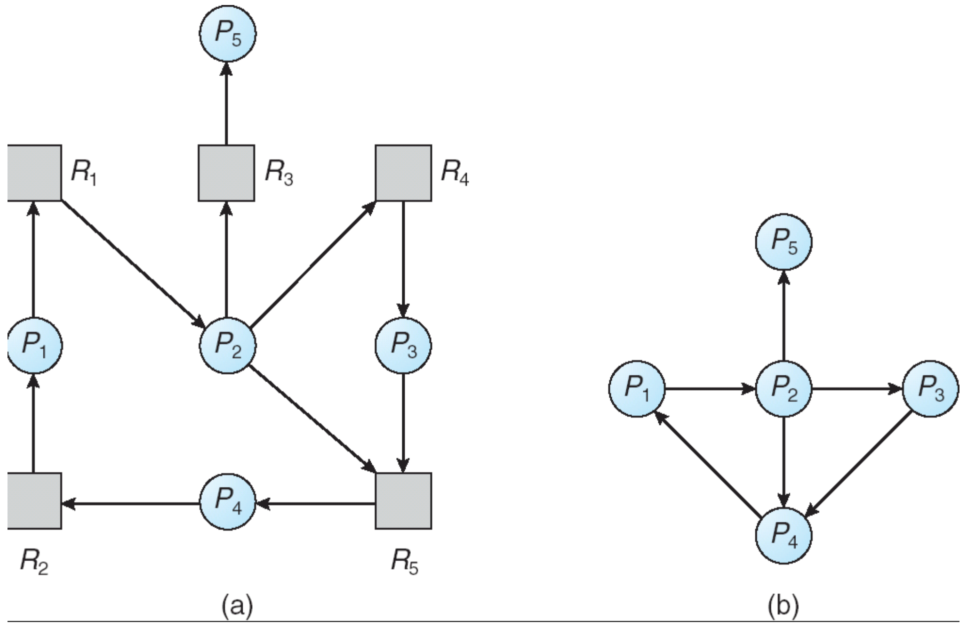
Request edge:
P_2 requests for a R_3 type resource

Assignment edge:
P_1 holds R_2

Vertices: Resources
R_2 and R_4 with 2 and 3 instances, respectively

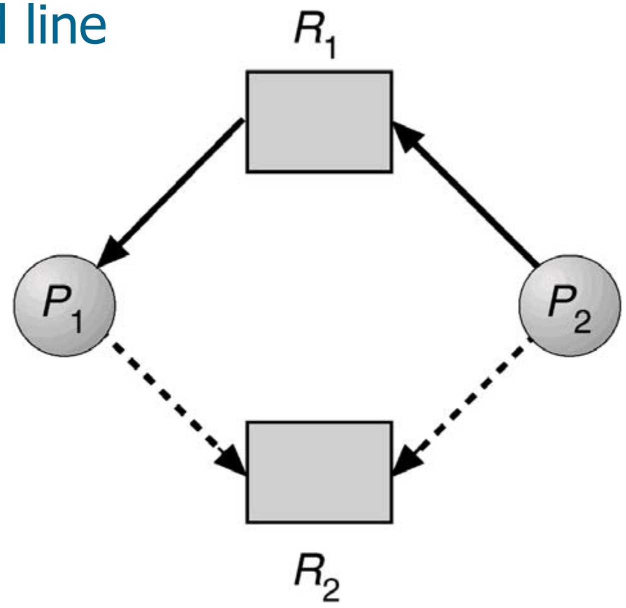
Modeling

- A **resource-allocation** graph can be sometime simplified in a **wait-for graph** by
  - deleting the resource vertices
  - creating the edges between the remaining vertices



Modeling

- Sometimes it is useful to extend the resource-allocation graph to a **claim graph** by
  - adding a claim edge: $P_i \dashrightarrow R_j$, indicates that process P_j **can ask resource R_j in the future**
  - It is represented by dashed line



Detection and recovery techniques

- The system is allowed to enter in a deadlock state, but
  - **Deadlock detection**
    - The system performs a deadlock detection algorithm
  - **Recovery form deadlock**
    - If deadlock has been detected a recovery action is performed

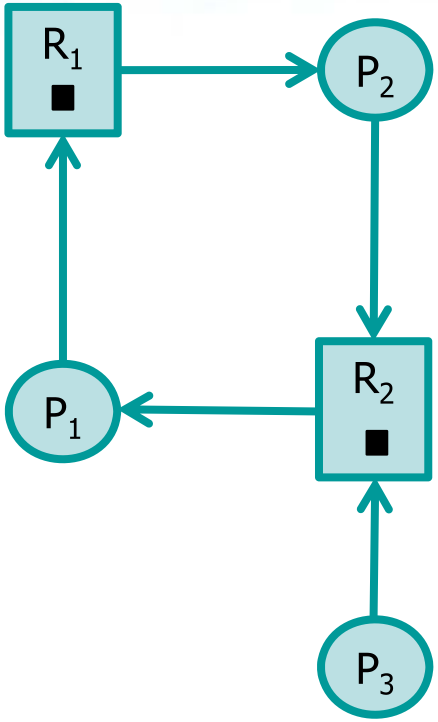
Detection: strategies

- Given an allocation graph deadlock can be detected by checking for for cycles
  - If the graph contains no cycles, then there is no deadlock
  - If the graph contains one or more cycles then
    - Deadlock exist if each type of resource has a single instance
    - Deadlock is possible if the are several instances per resource type
      - The presence of cycles is necessary but not sufficient in the case of multiple instances per resource type

For multiple instances see the Banker's Algorithm

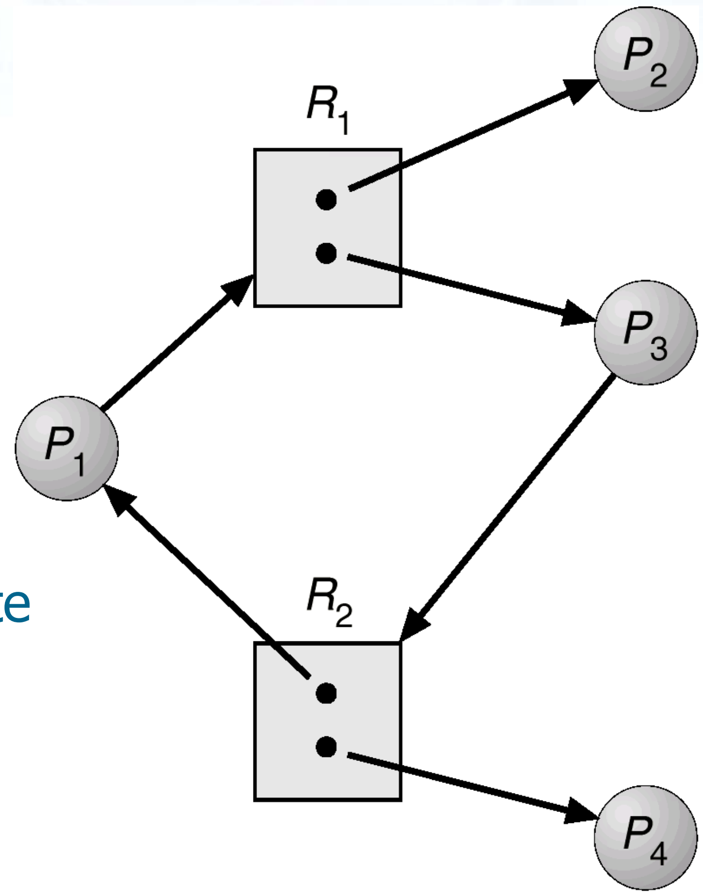
Example

- Processes
  - P_1, P_2, P_3
- Resources
  - R_1 and R_2 with a single instance
- A cycle exists
- Deadlock
  - P_1 waits for P_2
  - P_2 waits for P_1



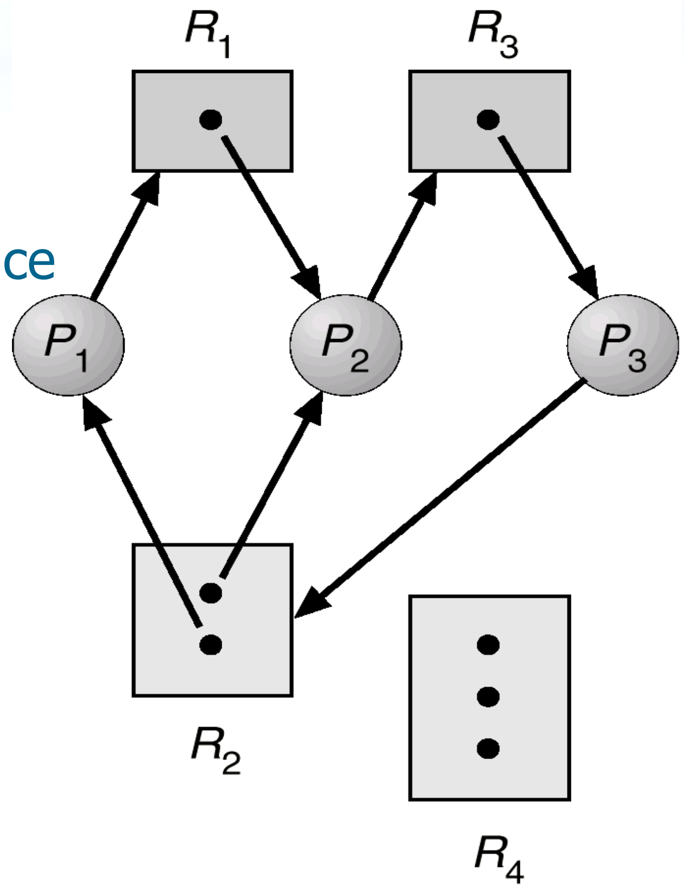
Example

- Processes
  - P_1, P_2, P_3, P_4
- Resources
  - R_1 and R_2 with two instances
- A cycle exists
- No deadlock
  - P_2 and P_4 can terminate
  - P_1 can acquire R_1 and terminate
  - P_3 can acquire R_2 and terminate



Example

- Processes
  - P_1, P_2, P_3
- Resources
  - R_1 and R_3 with an instance
  - R_2 with two instances
  - R_4 with three instances
- Two cycles exist
- Deadlock
  - P_1 waits for R_1
  - P_2 waits for R_3
  - P_3 waits for R_2



Detection: costs

- Detection phase has the high computational cost of finding a cycle in the graph
  - *An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where n is the number processes.*

- When detection is performed?
  - Every time a process makes a request not immediately satisfied
  - At fixed time intervals, e.g., every 30 minutes
  - At variable intervals of time, e.g., when the CPU usage falls below a given threshold

Recovery

- Different strategies are possible for deadlock recovery
  - Terminate all deadlocked processes
  - Terminate a process at a time among the ones in deadlock
    - Select a victim process, re-check the deadlock condition, and possibly iterate
  - Select a deadlocked process and
    - preempt the resources it holds, imposing a rollback, re-check the deadlock condition, and possibly iterate

Recovery

Strategy	Description
Terminate all deadlocked processes	• **Complexity**: **low**, but easy to cause inconsistencies on databases • **Cost**: **much higher than it might be** strictly necessary
Terminate a process at a time among the ones in deadlock	• **Complexity**: **high**, since it is necessary to select the victims with objective criteria (priority, current and future execution time, number of held resources, etc.) • **Cost**: **high**, after each termination must re-check the deadlock condition
Preempt the resources of a deadlocked process at a time	• **Complexity**: rollback is necessary to return the selected process to a safe state • **Cost**: the victim process selection must aim at minimizing the preemption cost

Conclusions

- Detection and recovery operations are
  - logically complex
  - computationally expensive
- In any case, if a process requires many resources, starvation may occur
  - The same process is repeatedly chosen as the victim, incurring repeated rollbacks
    - To avoid starvation the victim selection algorithm should take into account the number of a process rollbacks