Deadlock

Deadlock avoidance techniques

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Deadlock avoidance

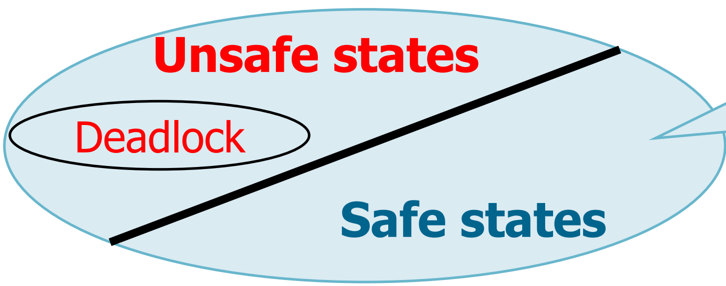
- Deadlock **avoidance** techniques force the processes to provide (a priori) additional information about the requests they will perform during their execution
  - Each process must indicate how many resources of all types it will need to terminate its task
  - This information allows a process scheduling order so that there is the guarantee of no deadlock
    - If granting immediately a requested resource to a process can cause deadlock, the process is forced to wait (by not assigning the resource to the process)

Deadlock avoidance

- The main algorithms
  - differ in the amount and type of information required
    - The simplest model imposes that all processes declare the maximum number of resources of each type that they will need
  - generally reduce the use of resources and the efficiency of the system
  - are based on the concept of **safe state** and **safe sequence**

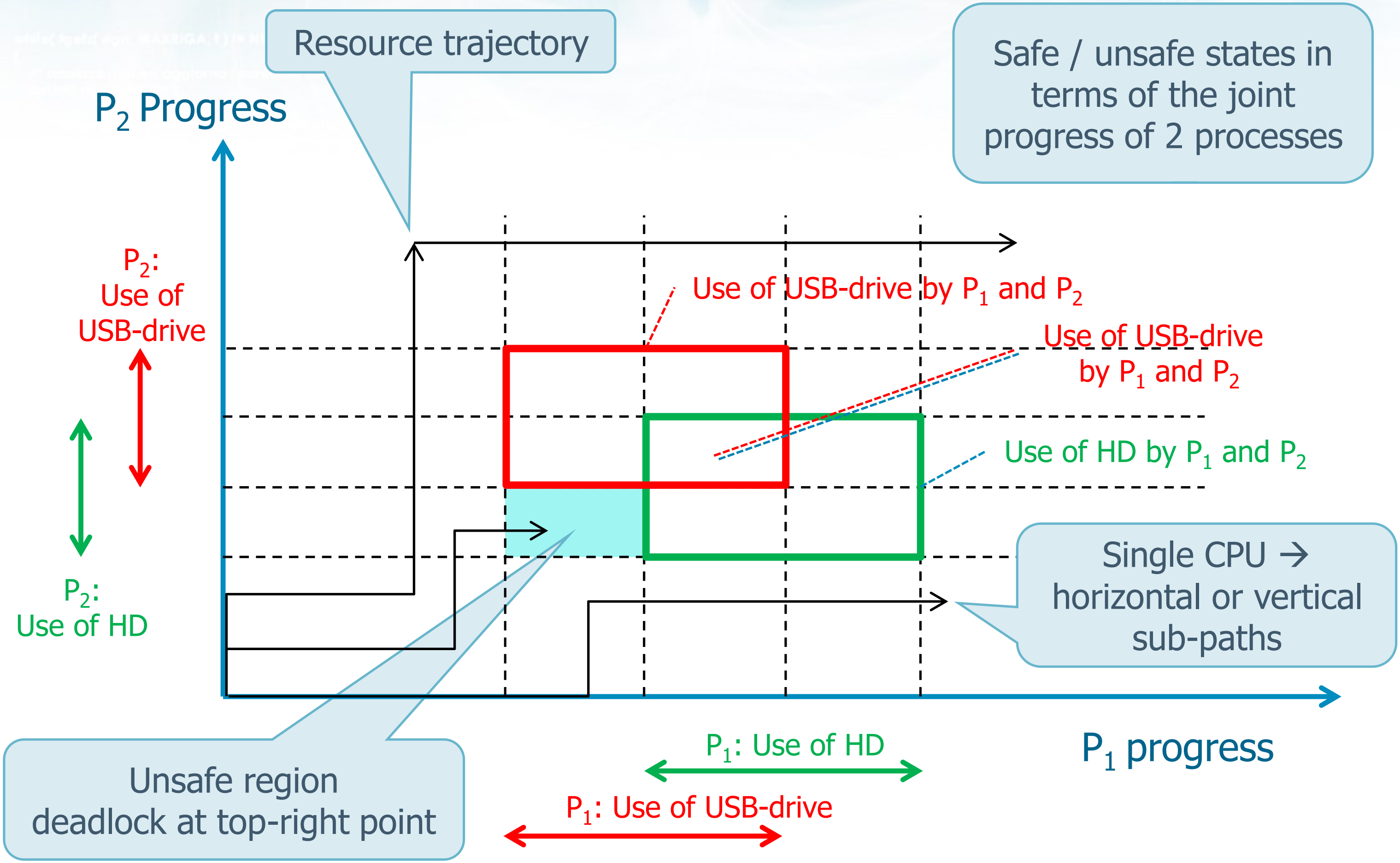
Safe state

Safe state	The system is able to • Allocate the required resources to all processes • Prevent the occurrence of a deadlock • Find a safe sequence
Safe sequence	A sequence of process scheduling $\{P_1, P_2, \ldots, P_n\}$ such that for each requests that could be performed by any P_i, the request can be satisfied by using the currently available resources and the other resources released by processes P_j with $j < i$



A state is said **unsafe** if it is not safe.
An unsafe state is not necessarily a deadlock state. It can leads to a deadlock state in case of standard behavior.

Joint progress of two processes

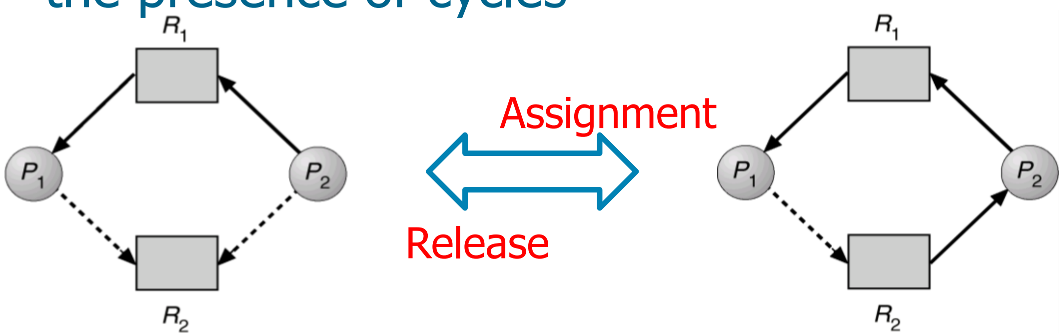


Strategies

- To avoid a deadlock, one must ensure that the system remains always in a safe state
  - Initially the system is in a safe state
  - Each new resource request
    - will be granted **immediately**, if this allows the system to remain in a **safe state**
    - **otherwise**, granting the request will be **delayed**; the process that performed the request is forced to wait
- There are two classes of strategies
  - For resources having unitary instances
  - For resources having multiple instances

Algorithm for resources with a single instance

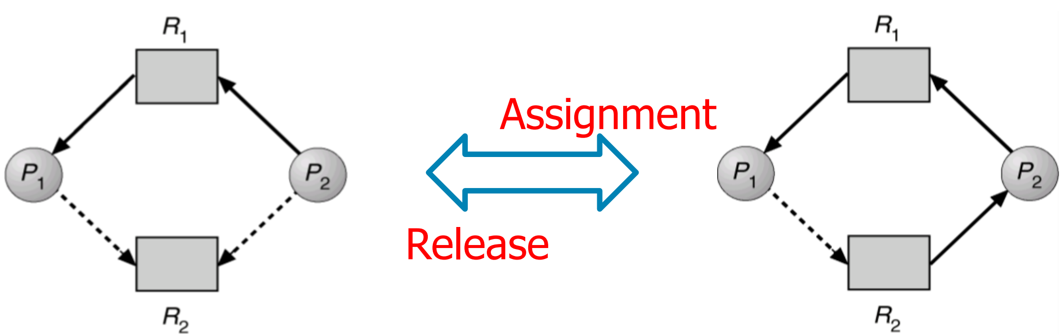
- Based on the determination of cycles, using the claim-for graph
  - All requests must be a priori declared
  - they are represented by claim arcs - - - ->
- At a time a request is performed
  - the corresponding claim arc is transformed into an assignment arc
  - Before the request is satisfied, the algorithm verify the presence of cycles



Algorithm for resources with a single instance

- If no cycle is present, the conversion of the arc is performed and the resource assigned
- Otherwise, the assignment of the requested resource would bring the system into an unsafe state. For this reason it is postponed

- Each time a resource is released
  - the assignment arc is transformed into a claim arc (to manage any subsequent request)



Algorithm for resources with multiple instances

- Verify the state of the system to understand if the available resources are sufficient to complete all processes based on
  - the number of resources available to the system,
  - number of resources allocated, and
  - max number of resource that the process may need
- Each process
  - must declare in advance its maximum number of resources it may need
  - when it requests a resource, it can be blocked for a limited amount of time
  - must guarantee to return an allocated resource in a finite amount of time

Algorithm for resources with multiple instances

- Banker's Algorithm (Dijkstra, [1965])
  - It consists of two parts
    - Verifies that the current state is safe
    - Verifies whether the new request can be immediately granted allowing to system to remain in a safe state
      - Simulates assigning the resource, and controls that a sequence of assignments exists that allows the system to satisfy all requests, possibly delaying the delivery of the resources for some of the requests.
- The algorithm uses the data structures listed in the following slide

Algorithm for multiple instances

Given a set of:

- **n processes P_r**
- **m resources R_c**

Name	Dim.	Content and meaning
finish	[n]	finish[r] initially false (indicates P_r has not compete)
allocation	[n][m]	allocation[r][c]=k P_r owns k instances of R_c
max	[n][m]	max [r][c]=k P_r can ask a maximum of k instances of R_c
need	[n][m]	need[r][c]=k P_r needs k additional instances of R_c $\forall i \forall j$ need[i][j]=max[i][j]-allocation[i][j]
available	[m]	available[c]=k k resources R_c are available

Example

- By applying the banker algorithm, the underlying system is in a safe state?
  - Safe sequence: P_1, P_3, P_0, P_2, P_4

P	finish	allocation	max	need	available
		R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2
P_0	F	0 1 0	7 5 3		3 3 2
P_1	F	2 0 0	3 2 2		
P_2	F	3 0 2	9 0 2		
P_3	F	2 1 1	2 2 2		
P_4	F	0 0 2	4 3 3		

Example

- Can the request of P_1 (1, 0, 2) be satisfied?
  - Yes ...
  - System state evolution …

P	finish	allocation	max	need	available
		R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2
P_0	F	0 1 0	7 5 3	7 4 3	3 3 2
P_1	F	2 0 0	3 2 2	1 2 2	
P_2	F	3 0 2	9 0 2	6 0 0	
P_3	F	2 1 1	2 2 2	0 1 1	
P_4	F	0 0 2	4 3 3	4 3 1	

Example

- The new state is safe or not?
  - Safe sequence: P_1, P_3, P_0, P_4, P_2

P	finish	allocation	max	need	available
		R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2
P_0	F	0 1 0	7 5 3	7 4 3	2 3 0
P_1	F	3 0 2	3 2 2	0 2 0	
P_2	F	3 0 2	9 0 2	6 0 0	
P_3	F	2 1 1	2 2 2	0 1 1	
P_4	F	0 0 2	4 3 3	4 3 1	

Example

Same initial state

- Can the request of P_4 (3, 3, 0) be satisfied?
  - No … there is not availability
- Can the request of P_0 (0, 3, 0) be satisfied?
  - No … the resulting state is not safe

P	finish	allocation	max	need	available
		R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2
P_0	F	0 1 0	7 5 3	7 4 3	2 3 0
P_1	F	3 0 2	3 2 2	0 2 0	
P_2	F	3 0 2	9 0 2	6 0 0	
P_3	F	2 1 1	2 2 2	0 1 1	
P_4	F	0 0 2	4 3 3	4 3 1	

Banker's algorithm

- Verify a set of requests performed by P_i

```
if
   ∀j requests[i][j] <= need [i][j]
   and
   ∀j requests[i][j] <= available[j]
   then
      ∀j available[j]      -= requests[i][j]
      ∀j allocation[i][j] += requests[i][j]
      ∀j need[i][j]         -= requests[i][j]

if the resulting state is safe then
    the assignment is confirmed,
else
    the previous state is restored
```

Banker's algorithm

- Verify whether a state is safe or unsafe

```
1.
   ∀i∀j need[i][j]= max[i][j] - allocation[i][j]
   ∀i finish[i]=false
2.
   Find a process Pi such that
   finish[i]=false AND ∀j need[i][j] <= available[j]
   If no such i is found goto step 4
3.
   ∀j available[j] += allocation[i][j]
   finish[i]=true
   goto step 2
4.
   if ∀i finish[i]=true then
      system is in a safe state
```

Exercise

- Can the request of P_1 (1, 0, 1) be satisfied?
  - Yes …
- Can the request of P_2 (1, 0, 1) be satisfied?
  - No … the resulting state is not safe

P	finish	allocation	max	need	available
		R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2	R_0 R_1 R_2
P_0	F	1 0 0	3 2 2		1 1 2
P_1	F	5 1 1	6 1 3		
P_2	F	2 1 1	3 1 4		
P_3	F	0 0 2	4 2 2		

Exercise

- Are the following states safe or unsafe?

(single resource problems)

P	F	A	M	N	D
P_0	F	3	9		3
P_1	F	2	4		
P_2	F	2	7		

... safe state

P	F	A	M	N	D
P_0	F	4	9		2
P_1	F	2	4		
P_2	F	2	7		

... unsafe state

Banker's algorithm

- Complexity is
  - $O(m \cdot n^2) = O(|R| \cdot |P|^2)$
- It is also based on unrealistic assumptions
  - Processes must specify their demands in advance
    - The necessary resources are not always known
    - Also it is not known when a resource will be used
  - Assumes that the number of resources is constant
    - Resources may increase or decrease due to transient or continuous failures
  - It requires a fixed population of processes
    - The number of active processes in the system increases and decreases dynamically