Deadlock

Deadlock prevention techniques

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

Prevention techniques

- Try to **control** how resources are requested to **prevent** the occurrence of **at least one** of the necessary conditions
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

Mutual exclusion

A deadlock occurs because of "mutual exclusion" when a process is indefinitely waiting a non sharable resource.

Thus, deadlock could be avoided if
1. All resources were shareable (e.g., read-only)
2. A process could not wait for a resource not immediately available

Strategy 1	• Allow only shareable resources. • This strategy is generally considered very restrictive.
Strategy 2	• Inhibit a process to wait for a resource that is not immediately available. • This strategy is considered complex to be implemented

Hold and wait

A deadlock occurs because of a “hold and wait" condition, where a process requests further resources while holding one or more resources.

So a hold and wait condition can be avoided by imposing that a process waits for a resource only when it does not hold others

Request All First (RAF)	A process must acquire all the necessary resources before starting its processing activities • Poor resource usage • Resources may be assigned a long time in advance of their usage
Release Before Request (RBR)	A process can request resources only when it has not previously acquired other resources • Before each new request each process must release the resources already held • Possibility of starvation • Processes requiring many widely used resources may have to "start over" very often

No preemption

A deadlock occurs because no preemption is possible of a resource held by a process

In general is not easy to divert resources from a running process, but a similar effect can be obtained by means of the following strategies:

Allow preemption of resources held by the process itself	• If a process that holds some resources asks for another that cannot be immediately granted, it is forced to release all held resources (preemption). • These resources are added to the list of resources that the process is waiting for. • The process will be awakened only when it can regain its old resources, and additionally the new one.

No preemption

A deadlock occurs because no preemption is possible of a resource held by a process

Allowing preemption of resources owned by another process as long as it is waiting	• If process P asks for a resource that is not immediately available, a search is performed for the process that currently holds it • If a process Q is found, which is waiting for another resource, preempt from Q the resource and assign it to process P • Otherwise, process P goes on the waiting state, so that the resources it hold can be preempted

Both strategies

- are suited for resources whose state can be easily saved and restored (CPU registers, main memory, etc.)
- are not suited for resources whose state cannot be recovered (files, printers, etc.)

Circular wait

A deadlock occurs because of a "circular wait" when a set of processes is waiting for a resource held by another set o processes

To avoid this condition, one can impose a total ordering of all resource classes

Hierarchical Resource Usage (HRU)	• It imposes a total ordering relation between the various types of resources, associating to each of them an integer number. Example: HD = 1, DVD = 5, printers = 12 • Force each process to request resources with an increasing order of enumeration In general, the HRU verification is applied by • Programmer • Operating system. The **`witness`** tool, available in FreeBSD UNIX version, checks the order of the lock acquired by processes

Circular wait

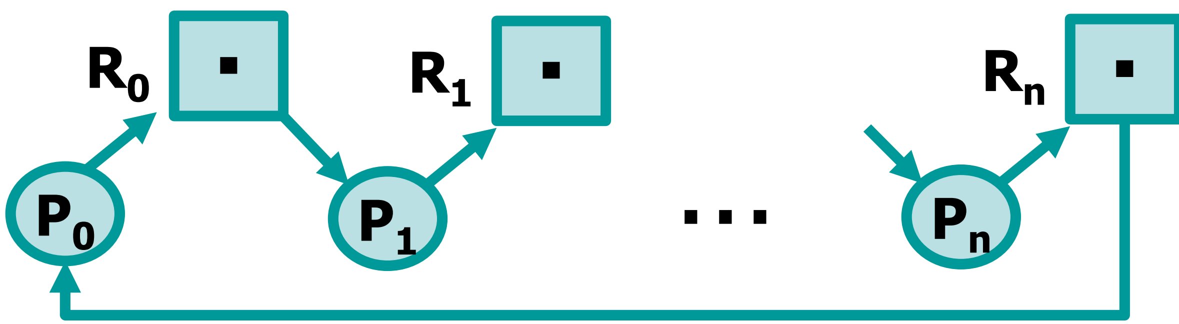
- Let F be the function that imposes a unique order among all classes of system resources R_i
  - Let a process have previously requested an instance of R_{old} resource, and now request a R_{new} instance
  - If $F(R_{new}) > F(R_{old})$
    - The resource is granted
  - If $F(R_{new}) \leq F(R_{old})$
    - The process must release all resources R_i such that $F(R_{new}) \leq F(R_i)$ before getting an instance of R_{new}

Circular wait

- It can be shown that this condition is sufficient to avoid the circular wait
  - That is, if the resources are requested in a certain order, is it true that it is not possible to have a circular wait?
  - We proceed using a demonstration of type "reduction to absurdity", assuming there is a circular wait, i.e., supposing there is a set of processes that
    - They were requested in the specified order, e.g., in increasing numerical order
    - They are in circular wait

Circular wait

- Let's suppose that there exists a set of processes that satisfy the HRU rules and are in circular wait



Request order $R_0,\ldots,R_n$

The order of requests requires that

$$F(R_k) < F(R_{k+1}), \ \forall\, k = 0 \,..\, n-1 \ .$$

This implies

$$F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$$

$$F(R_0) < F(R_0) \ ,$$

which is absurd

Since P_i holds R_i and it has required R_{i-1}
R_i was requested before R_{i-1}
Thus, $F(R_i)>F(R_{i-1})$