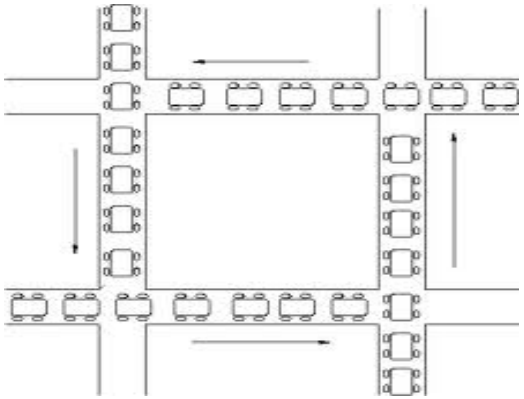
Unit 4.- Deadlocks

Concurrency and Distributed Systems

Teaching Unit Objectives

- Understand the **problems** that can occur with incorrect use of the synchronization mechanisms.
  - Analyse deadlock situations.

- Learn techniques for managing deadlocks:
  - Prevention
  - Avoidance
  - Detection and Recovery
  - No action (to ignore the problem)



Bibliography

- Operating system concepts with Java. Abraham Silberschatz Peter B Galvin; Greg Gagne Hoboken, NJ : John Wiley & Sons cop. 2007 7th ed. ISBN 9780471769071. Chapter 7 – Deadlocks.

- Documents:
  - **System Deadlocks.** Edward G. Coffman Jr., M. J. Elphick y Arie Shoshani. ACM Comput. Surv., 3(2):67–78, 1971. http://dl.acm.org/citation.cfm?id=356588
  - Richard C. Holt. **Some deadlock properties of computer systems**. ACM Comput. Surv., 4(3):179–196, 1972. http://dl.acm.org/citation.cfm?id=356607

- Java concurrency: *Liveness → Deadlock; Starvation and Livelock*

  http://docs.oracle.com/javase/tutorial/essential/concurrency/

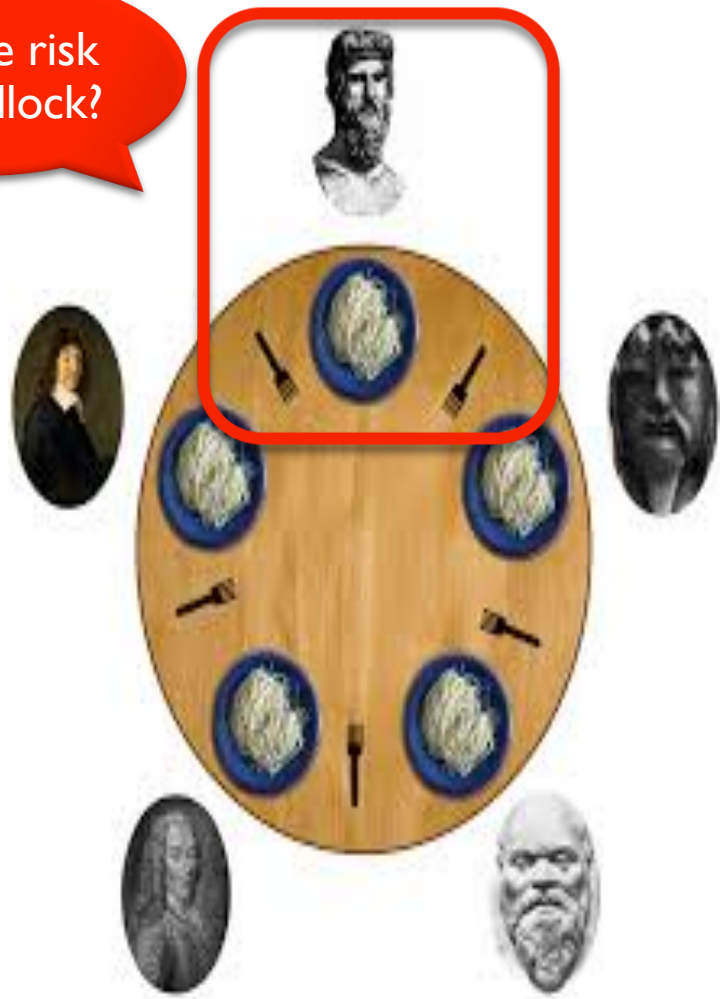
- In Spanish:
  - Chapter 4, course book ("Concurrencia y Sistemas Distribuidos").

Content

- Deadlock: what is it
- Coffman's Conditions
- Graphical representation
  - Resource Allocation Graph
- Solution Strategies

Motivating Example: Dining Philosophers

Is there risk of deadlock?



Think

Eat

```
while(true){
   think();
   takeFork(right);
   takeFork(left);
   eat();
   leaveFork(left);
   leaveFork(right);
}
```

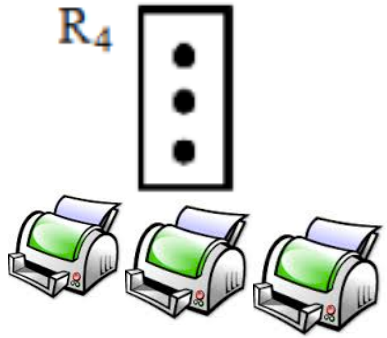
What is a deadlock??

- **Resource**: any physical or logical element that is requested by a thread (e.g. printers, semaphore, …)
  - A resource can have several **instances**
  - Protocol for using the resources:

Request

Usage

Release



Deadlock: there is a set of threads that cannot evolve because they are waiting to each other

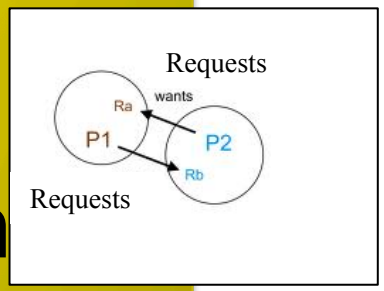
How can you detect a risk of deadlock?

- Using **Coffman's Conditions:**

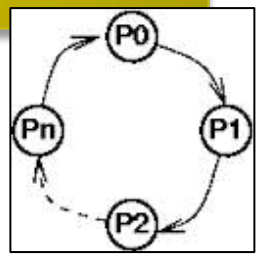
Mutual Exclusion

Hold and Wait

No Preemption



Circular Wait



How can you detect a risk of deadlock?

- **Coffman's Conditions:**
  - **Mutual Exclusion:** While a resource is assigned to a thread, other threads cannot use it.
  - **Hold and wait:** Resources are requested as needed, so we can have a resource assigned and request another not available (and thus wait for it)
  - **No Preemption:** An assigned resource can only be released by its owner (it cannot be expropriated)
  - **Circular wait:** In the group of threads in deadlock, each of them is waiting for a resource that holds another of the group, and so on until closing the circle

Coffman's Conditions Properties

- **Coffman's Conditions**

Mutual Exclusion

Hold and Wait

No Preemption

Circular Wait

If all them hold simultaneously→ ***risk of deadlock***

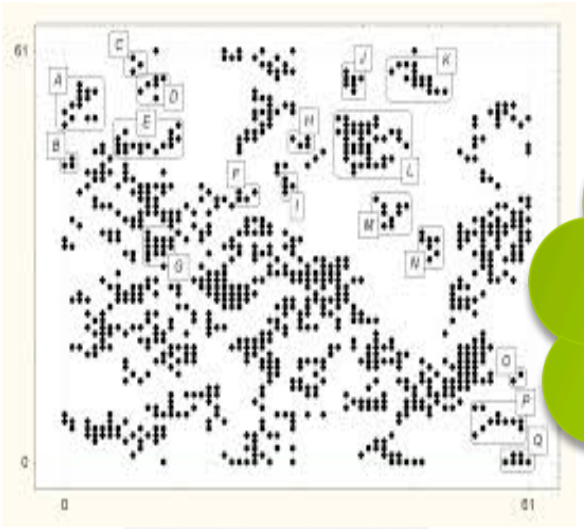
In case of deadlock→ ***they are all true***

They are **necessary**, but not sufficient conditions

Examples

Do Coffman's Conditions hold here?

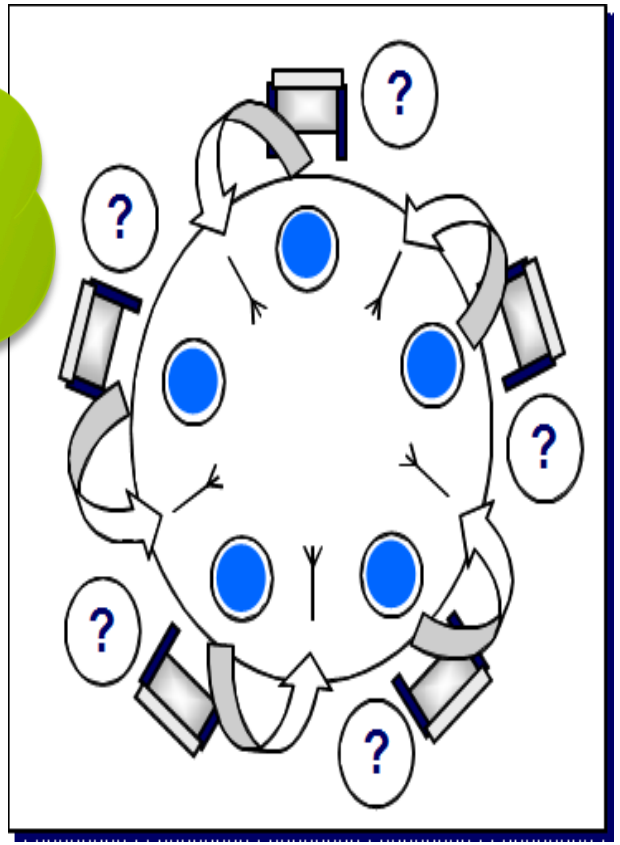
- Ants



→ Think example situations

→ Analyse Coffman conditions

- Dining Philosophers



Examples

- **Ants**
  - Ants A and B occupy neighbouring cells
  - A wants to move where B is (A must wait), and B where A is (B must wait)

- Coffman's Conditions
  - **Mutual Exclusion.-** a cell is not shareable
  - **Resource holding (Hold and wait).-** each ant uses a cell and waits until the target cell gets free
  - **No preemption.-** an ant cannot push another out of his cell
  - **Circular wait.-** A waits to B, and B waits to A

Examples

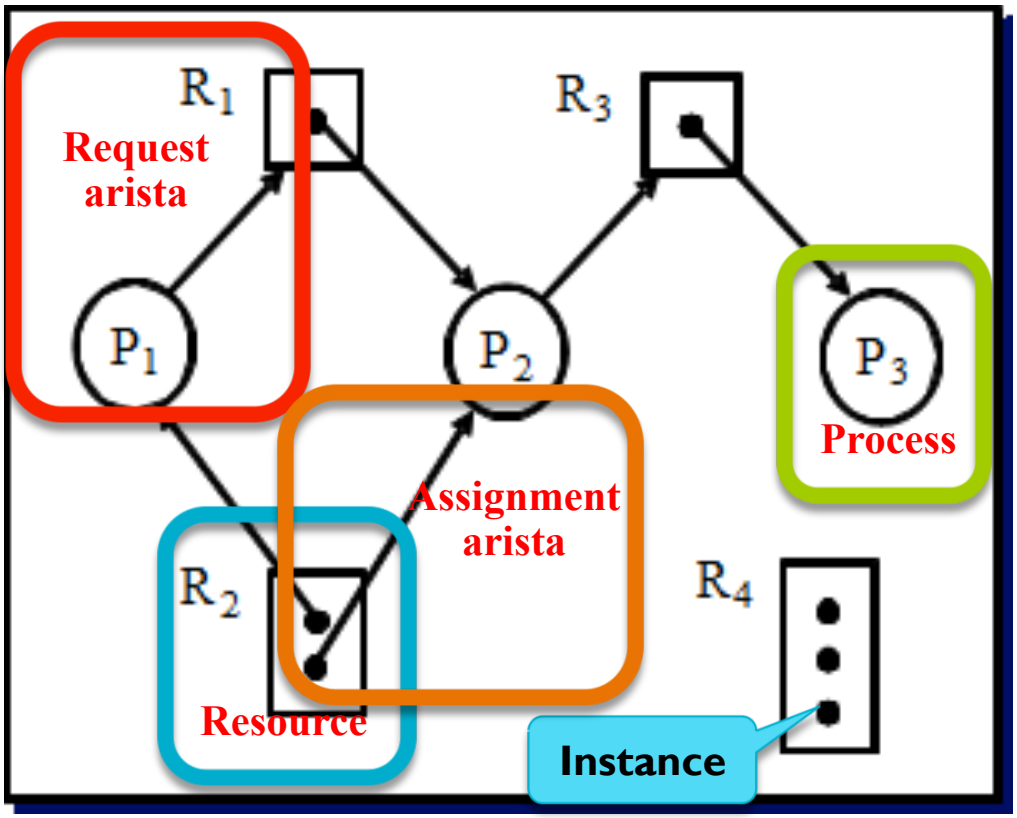
- **Dining Philosophers**
  - They come to the table all at once
    - Everyone has time to pick up his right fork
    - When they try to pick up their left fork, everyone has to wait
- Coffman's Conditions
  - **Mutual Exclusion.-** A fork cannot be employed simultaneously by 2 philosophers
  - **Hold and wait.-** They use (hold) the right fork, and wait for the left one.
  - **No preemption.-** The neighbour's fork cannot be stolen when he is using it.
  - **Circular Wait.-** Each one of the group waits to its neighbour, until close the circle

Content

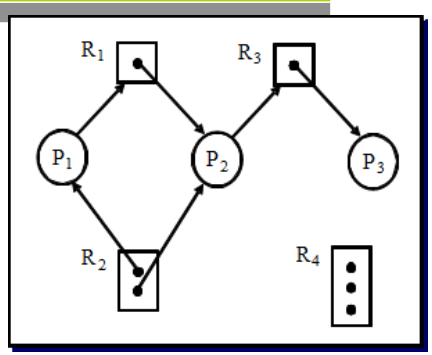
- Deadlock: what is it
- Coffman's Conditions
- Graphical representation
  - Resource Allocation Graph
- Solution Strategies

Graphical Representation: RAG

- RAG = **Resource Allocation Graph** or **Wait-For Graph**
  - Graphical representation of the state of the system



Graphical Representation: RAG

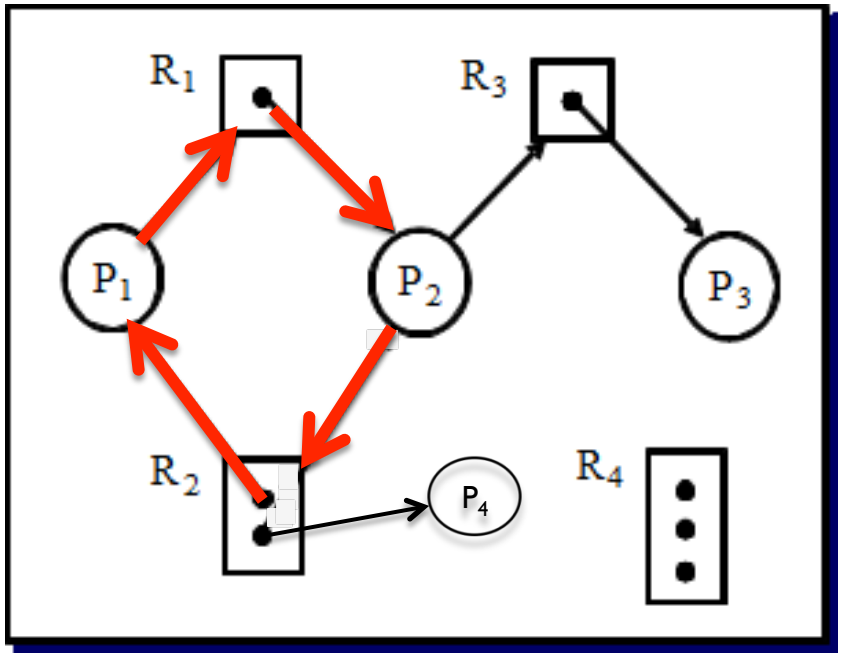


- RAG= **Resource Allocation Graph**
  (or **Wait-For Graph**)

  - Graphical representation of the state of the system
  - A **circle** represents a thread (or process)
  - A **rectangle** represents a resource (and contains as much internal points as instances of the resource)
  - **Assignment arista**: an **assigned resource** is represented with an arrow from a specific instance to the thread that employs it
  - **Request arista**: an **unresolved request** (there are no free instances of the resource, so the thread must wait) is represented with an arrow that goes from the thread to the resource (and not to a specific instance)

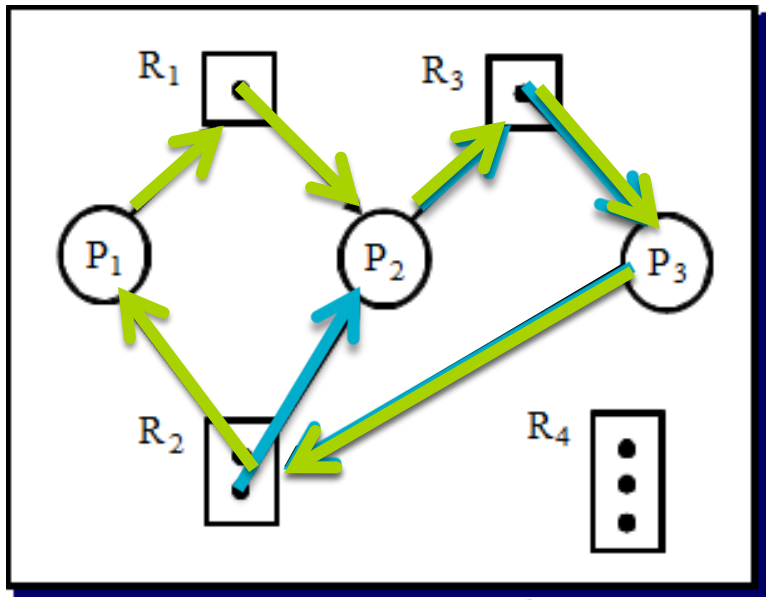
Graphical Representation: RAG

- **Directed cycle** in the RAG → ***risk of deadlock***
  - If all resources of the cycle have just 1 instance → **deadlock**
  - If there is a **safe sequence** → **No deadlock**



Safe sequence: there is an order in which all threads can finish

RAG Examples



Is there a risk of deadlock ? **Yes**

Is there a directed cycle? **Yes**

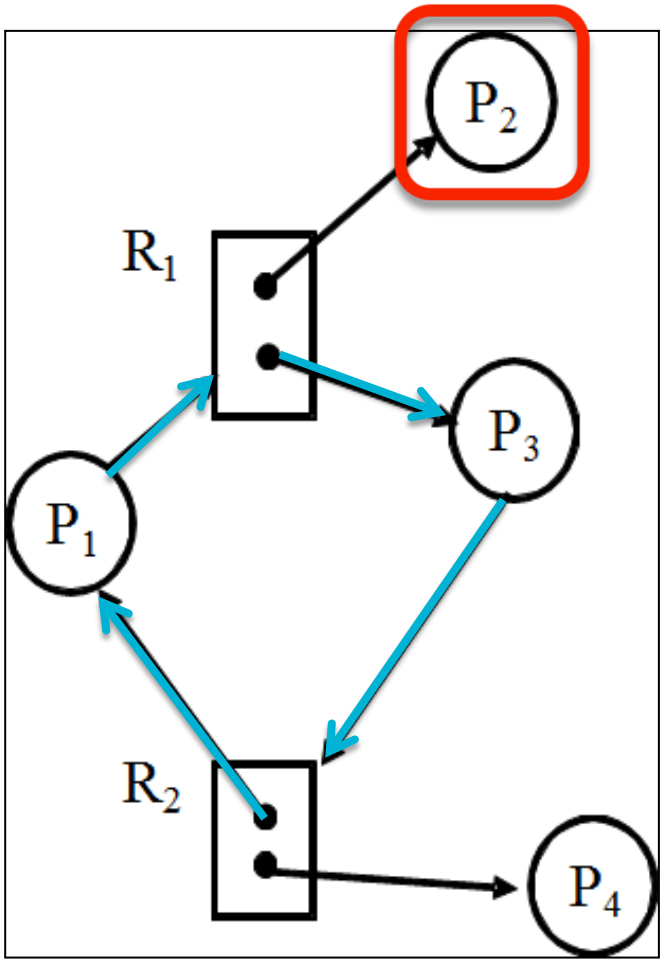
Is there a deadlock? **Yes**

Is there a safe sequence?

- P1 cannot finish
- P2 cannot finish
- P3 cannot finish

There is not any safe sequence

RAG Examples→ Is there a deadlock?

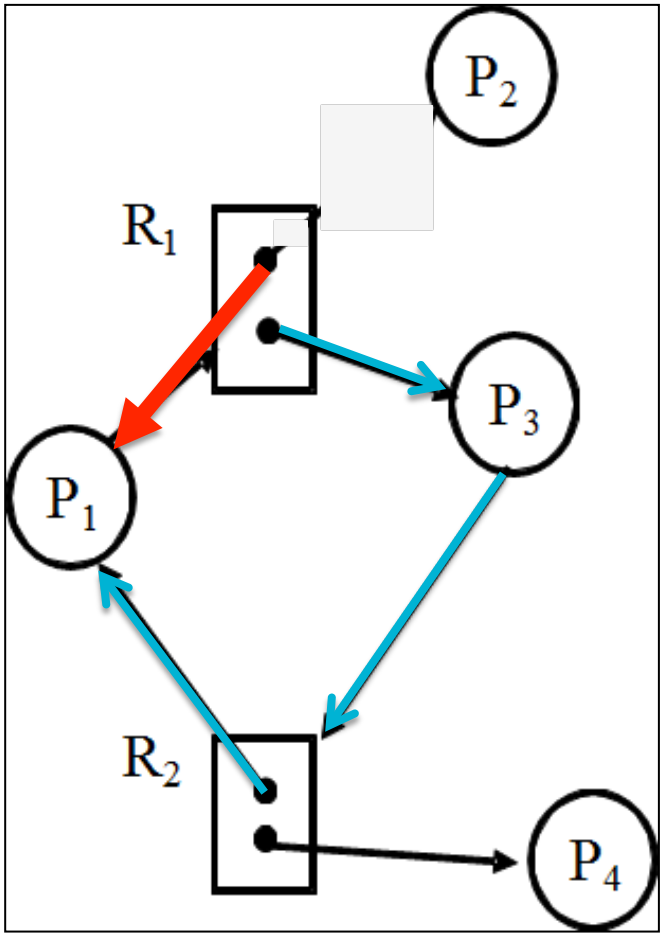


Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

RAG Examples→ Is there a deadlock?



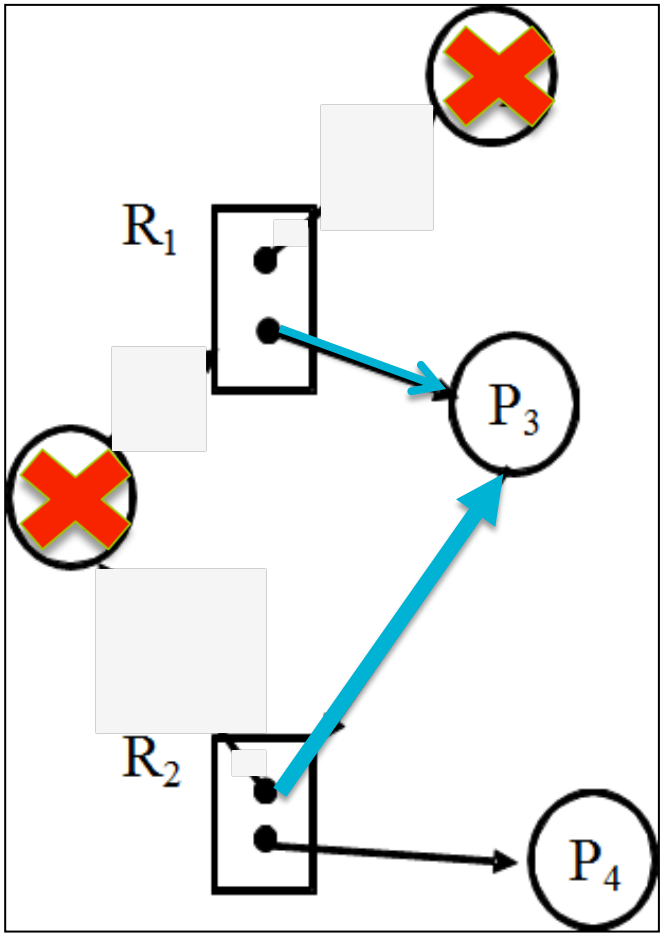
Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

P2,

RAG Examples→ Is there a deadlock?



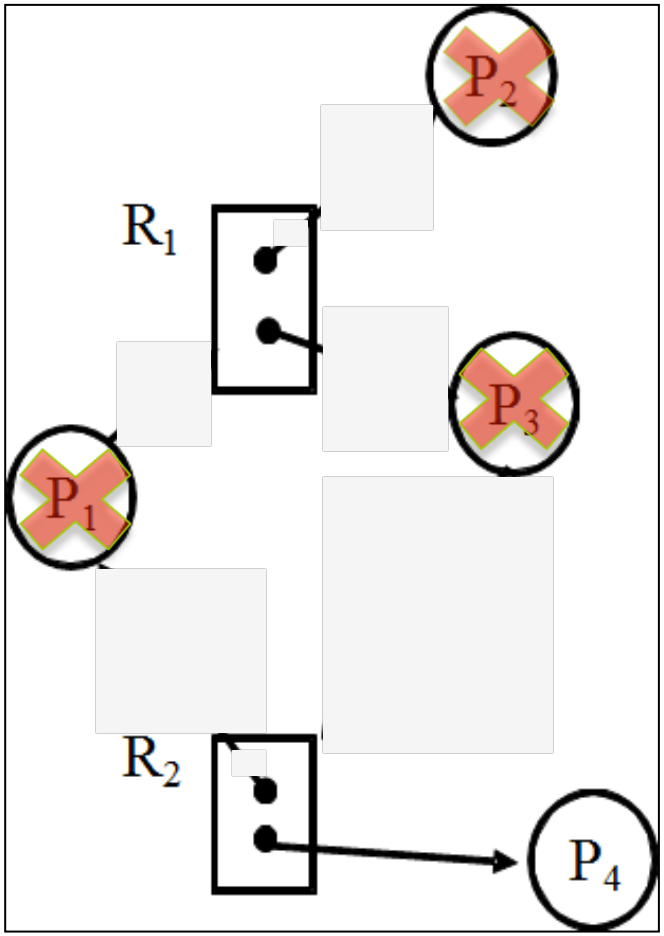
Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

P2, P1,

RAG Examples→ Is there a deadlock?



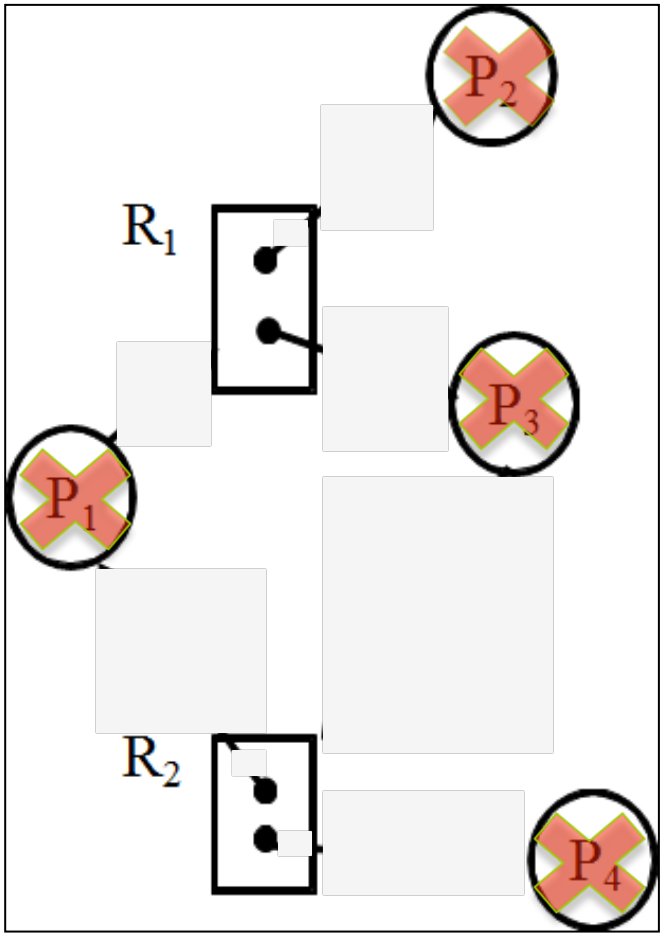
Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

P2, P1, P3

RAG Examples→ Is there a deadlock?



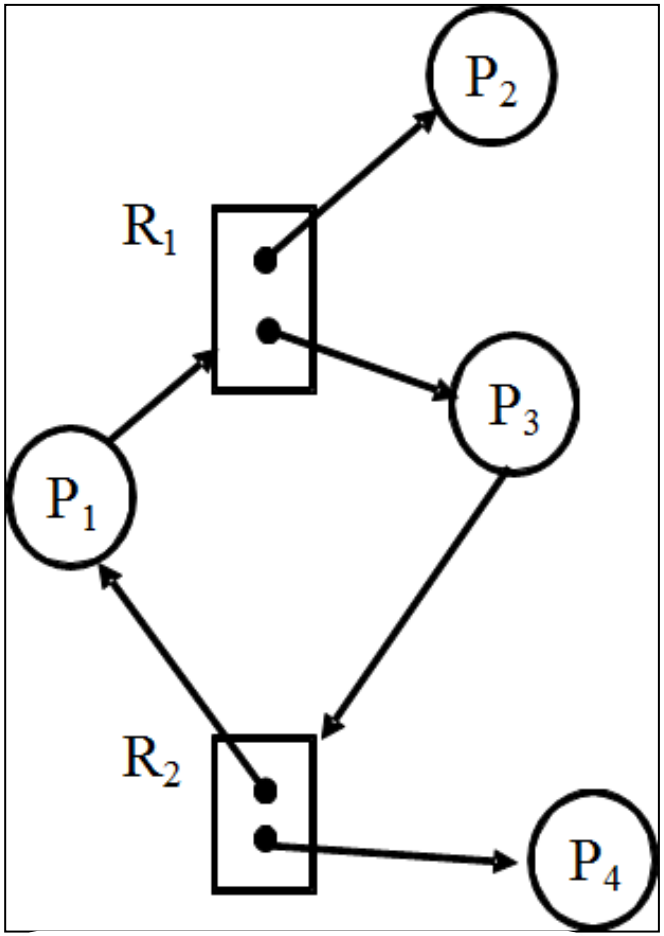
Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

P2, P1, P3, P4

RAG Examples→ Is there a deadlock?



NO deadlock

Is there a risk of deadlock?

Directed Cycle in P1,R1,P3, R2

Safe sequence?

P2, P1, P3, P4

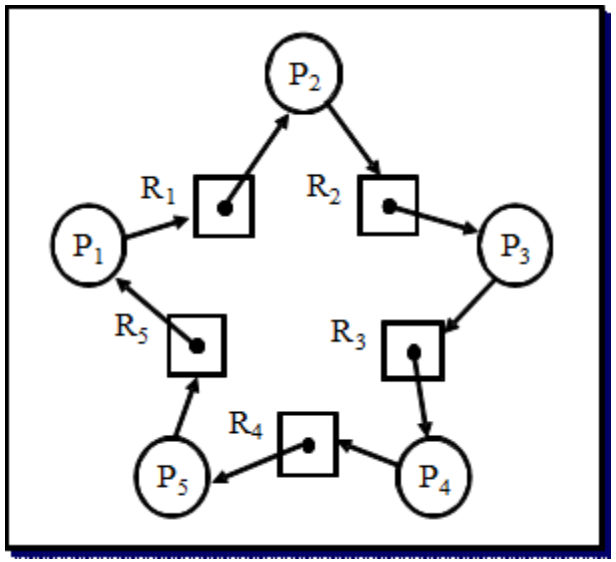
P4, P2, P3, P1

P2, P4, P1, P3

…..

Graphical Representation: RAG

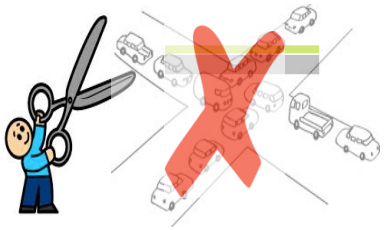
- RAG for the Dining Philosophers example
- There is a cycle, and all resources have one single instance:
  - There is a **deadlock**
- Question:
  - Why do we represent each fork as a resource with a single instance instead of using a single resource 'fork' with 5 instances?



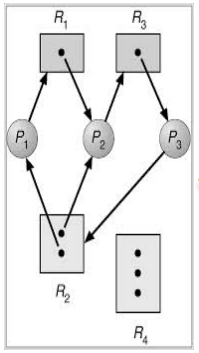
Content

- Deadlock: what is it
- Coffman's Conditions
- Graphical representation
  - Resource Allocation Graph
- Solution Strategies

Solutions to the deadlock problem



- **From best to worst:**
  1. **Prevention**
     - Design a system that breaks any Coffman's Conditions
     - So then deadlock IS NOT possible.



Risk of deadlock!!!

  2. **Avoidance**
     - Using a RAG, dynamically consider every request
       - If a request creates a cycle (*risk of deadlock*), the system denies it *(*this is *The Banker's algorithm)*
     - There cannot be a deadlock, but **this solution implies checking each request.**

Solutions to the deadlock problem

- **From best to worst:**

3. Detection and Recovery

- *Detection*: to monitor the system periodically
- *Recovery*: if there is a deadlock situation, then it aborts one or some of the activities involved in the deadlock

4. Ignore the problem

- This solves nothing, but it is a comfortable and frequent solution
- For example, many Operating Systems employ this solution, such as Unix and Windows

Prevention → How to carry it out?

- Breaking *Coffman's Conditions* → *How?*

Mutual Exclusion
Hold and Wait
No Preemption
Circular Wait

Resources are normally used in mutual exclusion by definition

Difficult to break it

Prevention → How to carry it out?

- Breaking *Coffman's Conditions* → *How?*

Mutual Exclusion
Hold and Wait
No Preemption
Circular Wait

You must avoid that a process holds a resource and waits for another

Concurrency ↓↓

Usage of resorces ↓

Possible thread **starvation**

Solution 1: Request EVERYTHING you might need at the beginning

Solution 2: Request resources in a non-blocking way

Prevention

- **Hold and wait**
  - The usual way of working is requesting resources as we need them, leading to hold and wait.
  - **Solution 1.-** To request from the beginning all that we can ever need
  - **Solution 2.-** To request resources in a non-blocking way
    - If the resource is being used then the requesting thread does not get blocked, but it receives a value that indicates this situation
    - If a thread cannot obtain all needed resources, it releases those that was holding
  - Both solutions:
    - Drastically reduce concurrency
      - Many threads have to wait, or resource reserves have to be retried repeatedly
    - They suppose a low usage of resources
    - They can cause **starvation** of the threads that require many resources that are frequently requested by other threads

Prevention → How to carry it out?

- Breaking *Coffman's Conditions* → *How?*

Mutual Exclusion

Hold and Wait

No Preemption

Circular Wait

A thread can expropriate resources of another thread

Expropriated thread would have to request the resources again

Problem: Livelock

Prevention → How to carry it out?

- Breaking *Coffman's Conditions* → *How?*

Mutual Exclusion

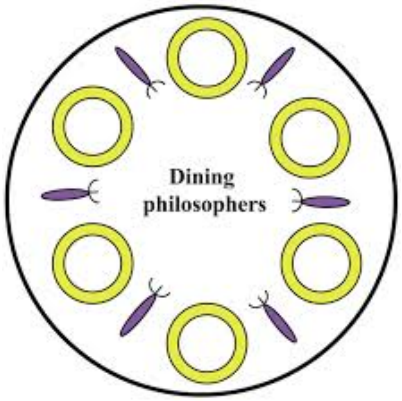
Hold and Wait

No Preemption

Circular Wait

A **total order** among resources is established

Processes must request resources in order



Easiest condition to break

Solution to the examples

- Solutions to the **Dining philosophers** problem (*deadlock prevention*)
  - Asymmetry: breaks "circular wait" condition
    - All philosophers equal except the last one (all pick up their right fork and then their left one, except last philosopher who employs the reverse order)
    - Odd and even (odd philosophers pick up their right fork first and then their left fork; whereas even philosopers do it in reverse order)
  - Each philosopher picks up both forks at once or none of them
    - This breaks "hold and wait" condition
  - Dining room (maximum 4 philosophers sitting at the table simultaneously)
    - This breaks "hold and wait" (at least one philosopher has both forks) and "circular wait"

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Identify the problems that an incorrect use of the synchronization mechanisms can provoke.
    - Specifically, to identify the deadlock problem.
  - Characterize the deadlock situations.
  - Identify the techniques for managing deadlocks.
  - Describe examples of solution strategies to prevent deadlocks.