Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of Resource-Allocation Graphs, having the following properties:

- A set of resource categories, { R_1, R_2, R_3, . . ., R_N}, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

- A set of processes, { P_1, P_2, P_3, . . ., P_N }

- **Request Edges** - A set of directed arcs from P_i to R_j, indicating that process P_i has requested R_j, and is currently waiting for that resource to become available.

- **Assignment Edges** - A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i, and that P_i is currently holding resource R_j.

Resource-Allocation Graph(cntd)

- Note that a request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges originates from a particular instance dot within the box.)
- The resource-allocation graph shown

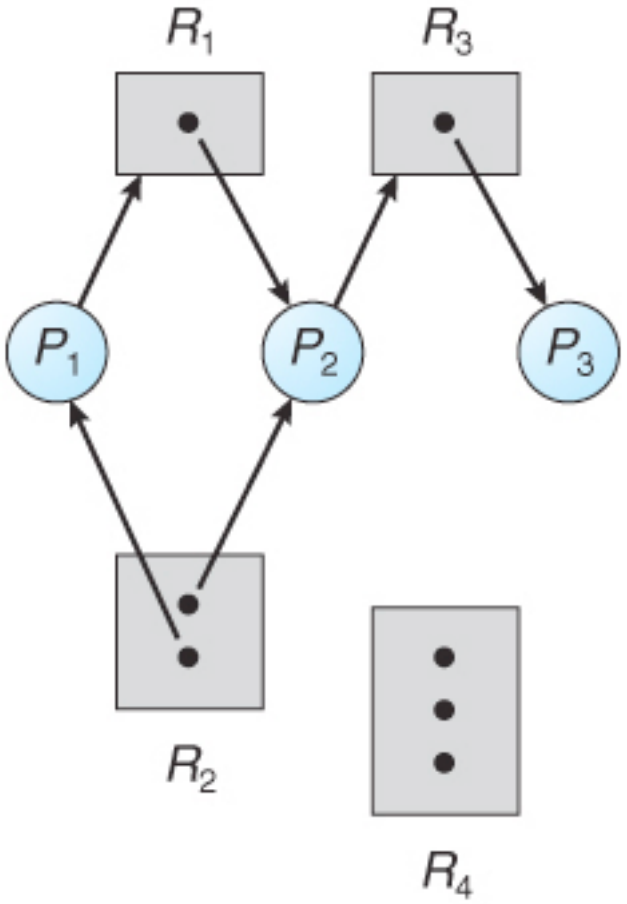
depicts the following situation.

- **The sets P, R, and E:**

P={P_1,P_2,P_3}

R= {R_1,R_2,R_3,R4}

E={p_1->R_1,P_2->R_3,R_1->P_2,R_2->P_1,R_2->P_2,R_3->P_3}



Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Resource-Allocation Graph(cntd)

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.)

- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.

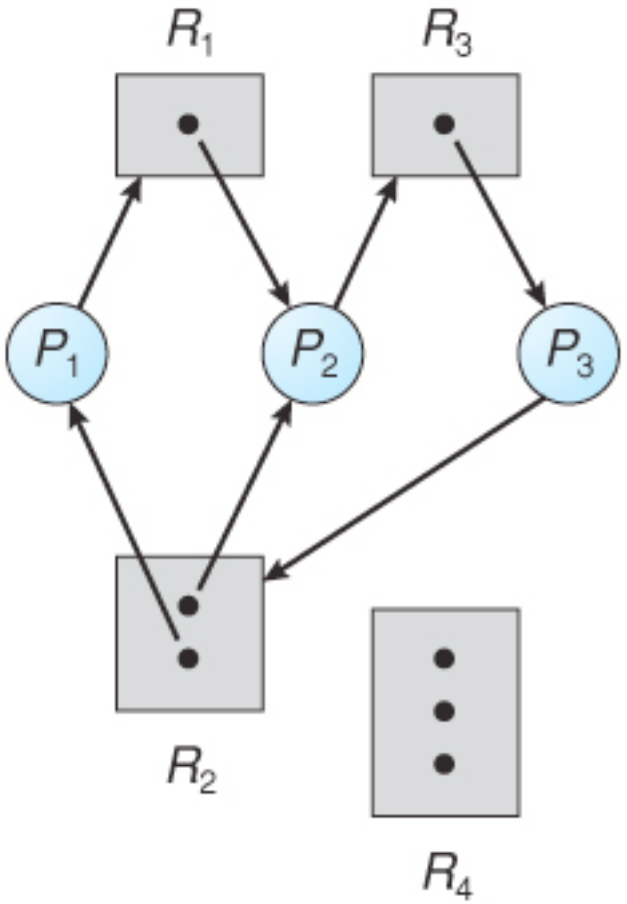
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.

Resource-Allocation Graph(cntd)

- Lets consider the resource-allocation graph given on previous slide. Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge P_3 —>R_2 is added to the graph.

- At this point, two minimal cycles exist in the system:
- P_1->R_1->P_2->R_3->P_3->R_2->P_1
- P_2->R_3->P_3->R_2->P_2

- Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

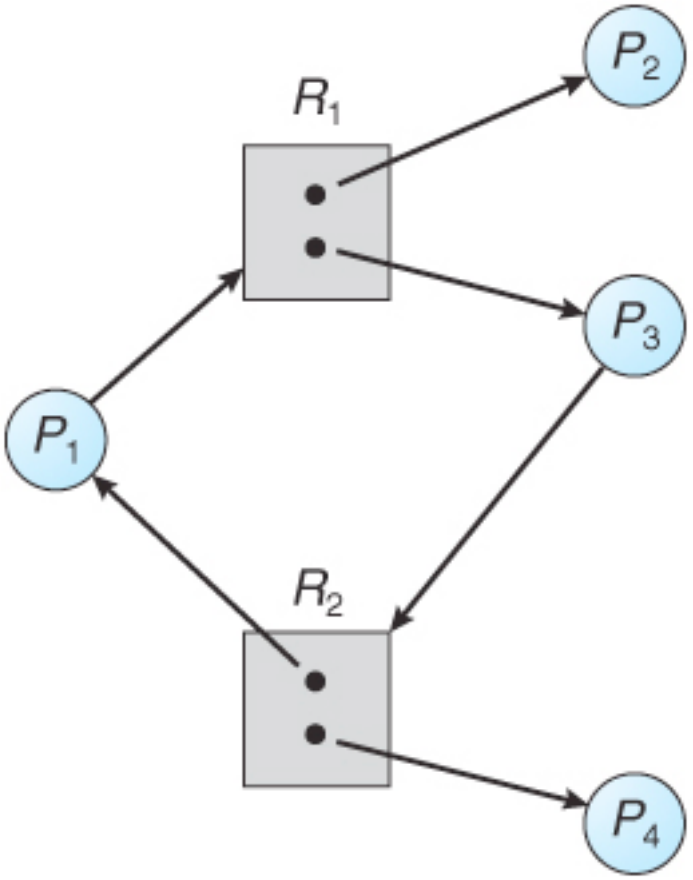


Resource-Allocation Graph(cntd)

- Now consider the resource-allocation graph shown. In this example, we also have a cycle.

- P1->R1->P3->R2->P1

- However, there is no deadlock. Observe that process P4 may release its instance of resource type R?. That resource can then be allocated to P3, breaking the cycle.



The Dining-Philosophers problem

- Multiple resources (Dijkstra 1968)

- “Five philosophers sit around a table, which is set with 5 plates (one for each philosopher), 5 chopsticks, and a bowl of rice. Each philosopher alternately thinks and eats. To eat, he needs the two chopsticks next to his plate. When finished eating, he puts the chopsticks back on the table, and continues thinking.”

- Philosophers are processes, and chopsticks are resources.

The Dining-Philosophers problem

Problems with dining philosophers

- The system may deadlock: if all 5 philosophers take up their left chopstick simultaneously, the system will halt (unless one of them puts one back)

- A philosopher may starve if her neighbors have alternating eating patterns

Simple Solution to Dining philosopher's Problem

- Simple solution to the dining philosopher problem is to restrict the number of philosophers allowed access to the table.

- If there are **N** chopsticks but only **N-1** philosophers allowed to compete for them, at least one will succeed, even if they follow a rigid sequential protocol to acquire their chopsticks.

- This solution is implemented with an integer semaphore, initialized to N-1. This solution avoid deadlock a situation in which all of the philosophers have grabbed one chopstick and are deterministically waiting for the other, so that there is no hope of recovery.

- However, they may still permit *starvation*, a scenario in which at least one hungry philosopher never gets to eat.

Simple Solution to Dining philosopher’s Problem

- Starvation occurs if the solution allow an individual to eat repeatedly, thus keeping another from getting a chopstick.

- The starving philosopher runs, perhaps, but doesn't make progress. Under some notions of fairness the solutions given above can be said to be correct.

Arbitrator solution using semaphore

- Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter.

- In order to pick up the forks, a philosopher must ask permission of the waiter.

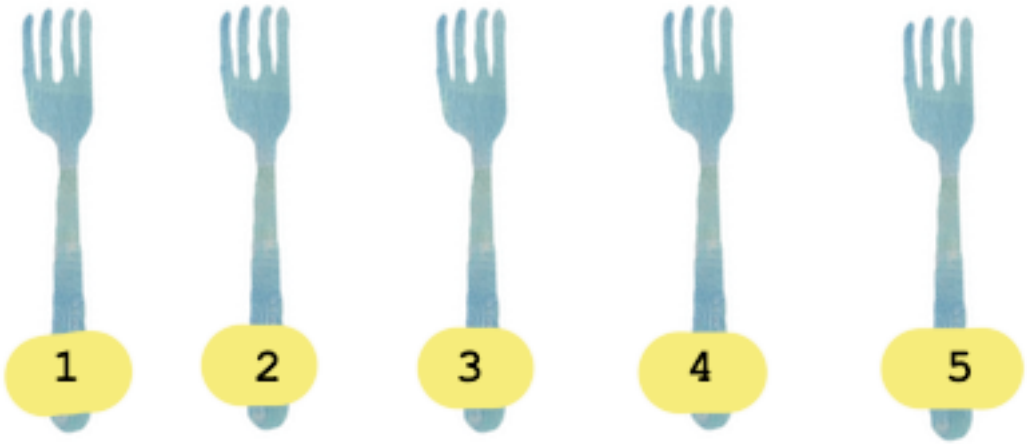
- The waiter gives permission to only one philosopher at a time until he has picked up both his forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex.

- In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Resource Hierarchy solution

- Let’s number the forks:

- Now the rule is, if you’re using two forks, you need to pick up the lower numbered fork first.



Resource Hierarchy solution

Resource Hierarchy solution

- Philosopher#1 picks up fork #1
- Philosopher#2 picks up fork #2
- Philosopher#3 picks up fork #3
- Philosopher#4 picks up fork #4
- Philosopher#5 can’t pick up fork #5! Because he will need two forks and he needs to pick up the lower numbered fork first!

- So fork #5 goes to Philosopher#4 – no deadlock!

Resource Hierarchy solution

- Resource hierarchy avoids deadlocks! But it is slow. Suppose you have forks #3 and #5. Then you decide you need fork #2. Well forks #3 and #5 are larger numbers. So you'll have to:

- put down fork #5
- put down fork #3 (the order you put these down in doesn't matter)
- pick up fork #2
- pick up fork #3

- Wastes a lot of time!