M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Operating Systems

Course Code: CS51

Credits: 3:1:0

Term: September – December 2020

Faculty:
Chandrika Prasad
Vandana S Sardar
Shilpa Hariraj

Deadlocks

The threat of deadlocks always exists in complex software systems.

Here, we will learn to prevent, avoid, detect, and eliminate deadlocks.

Readings: Silberschatz et al., chapter 8

RAMAIAH Institute of Technology

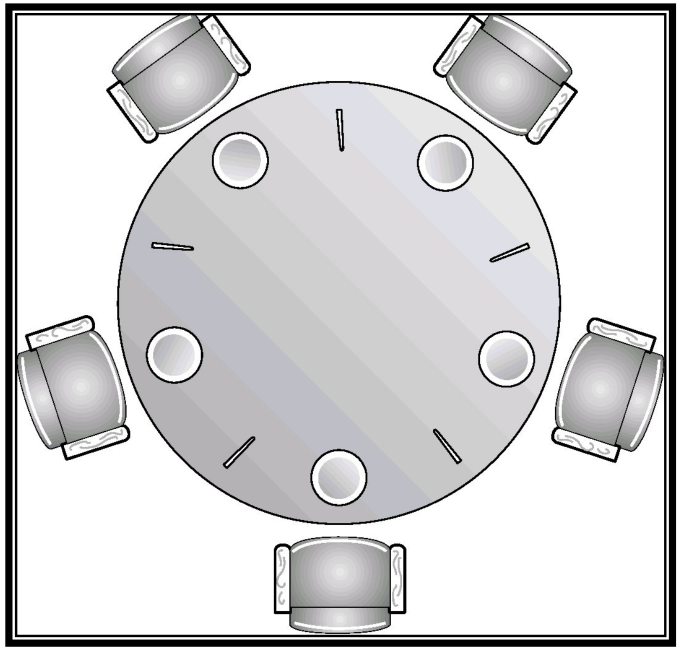
Classic Example: Dining Philosophers

5 philosophers spend their lives eating and thinking.

There are 5 chairs, 5 chopsticks, and 5 bowls of rice.

- When thinking, philosophers do not interact.
- When hungry, a philosopher tries to pick up two chopsticks and start eating.

Chopsticks are shared resources.



RAMAIAH
Institute of Technology

Dining Philosophers Example

Possible solution to Dining Philosophers problem with 5 semaphores initialized to 1:

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
        …
        eat
        …
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
        …
        think
        …
} while ( 1 );
```

Possible sequence of operations:

P0: wait(chopstick[0]);
P1: wait(chopstick[1]);
P2: wait(chopstick[2]);
P3: wait(chopstick[3]);
P4: wait(chopstick[4]);

DEADLOCK!

Deadlock-Free Dining Philosophers

How to prevent deadlock in the semaphore solution?

- Allow only 4 philosophers to sit down at one time
- Pick up both chopsticks together in a critical section
- Asymmetric pickup: some philosophers pick up left first, some pick up right first

Deadlock Analysis: System Model

A system contains

- Finite set of resources
- A set of competing processes

Resources:

- Partitioned into types
- Each type contains some number of identical instances
- Examples: memory space, CPU cycles, files, I/O devices

Processes do not care which instance of a type they get.

Deadlock Analysis: System Model (continued)

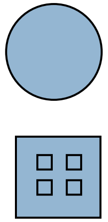
Processes:

- request resources via system calls
  - A process can request any number of resources up to max
  - If requested resource not available, must wait
- use resources
- release the resources

Definition: A set of processes is in a deadlocked state when every process in the set is waiting for an event that can only be caused by another process in the set.

Resource Allocation Graphs

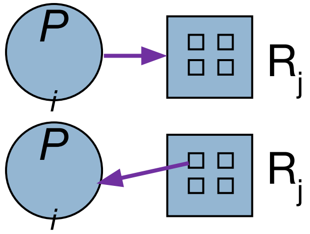
A useful analysis tool. Two node types, two edge types.



Nodes

- Process nodes are represented by circles:
- Resource nodes are represented by boxes:

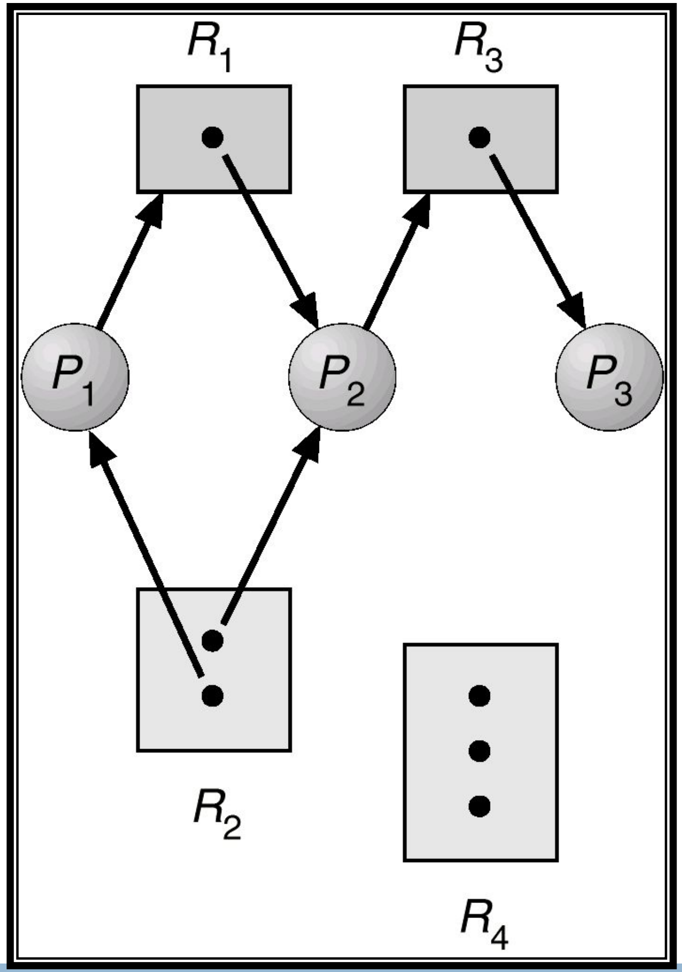
  [# of instances represented in the box]



Edges

- Resource request edge Pi->Rj:
- Resource assignment edge Rj->Pi:

Example Resource Allocation Graph (RAG)



Necessary Conditions for Deadlock

Mutual exclusion for resources

- There is at least one non-sharable resource

Hold and wait

- A process P is holding at least one resource and waiting for additional resources

No preemption of resources

- Once a process P is holding resources, the OS cannot take them away

Circular wait

- There exists a sequence of distinct processes P1, P2, …, Pn such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, …, and Pn is waiting for a resource held by P1.

Resource Allocation Graph and Existence of Deadlocks

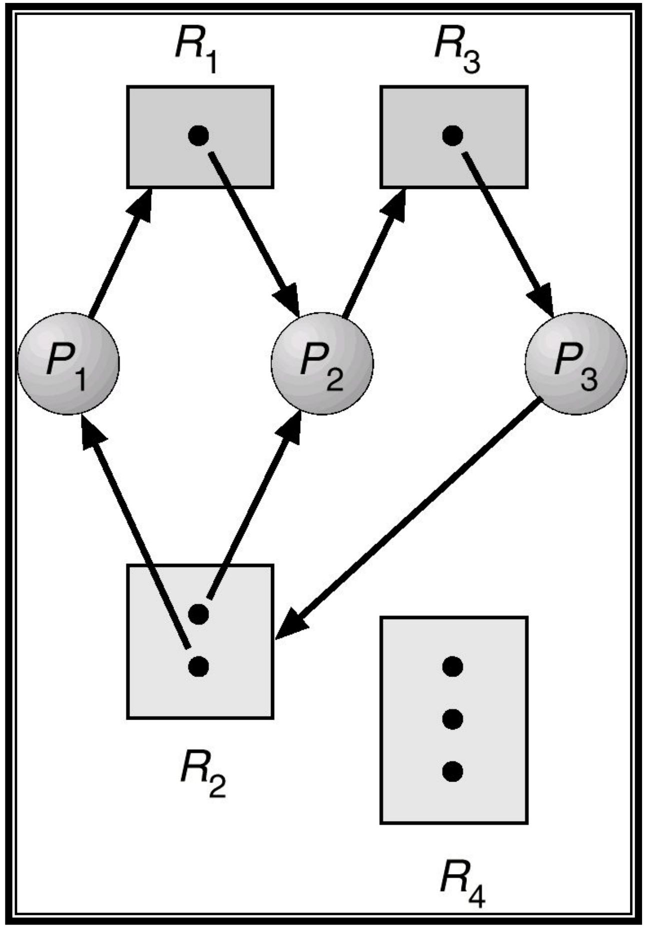
P2->R3->P3->R2->P2 is a deadlock cycle.

So is P1->R1->P2->R3->P3->R2->P1

If no cycle in RAG, then no deadlock.

If there is a deadlock, then there is a cycle in RAG.

If there is a cycle in RAG, then there may or may not be a deadlock.

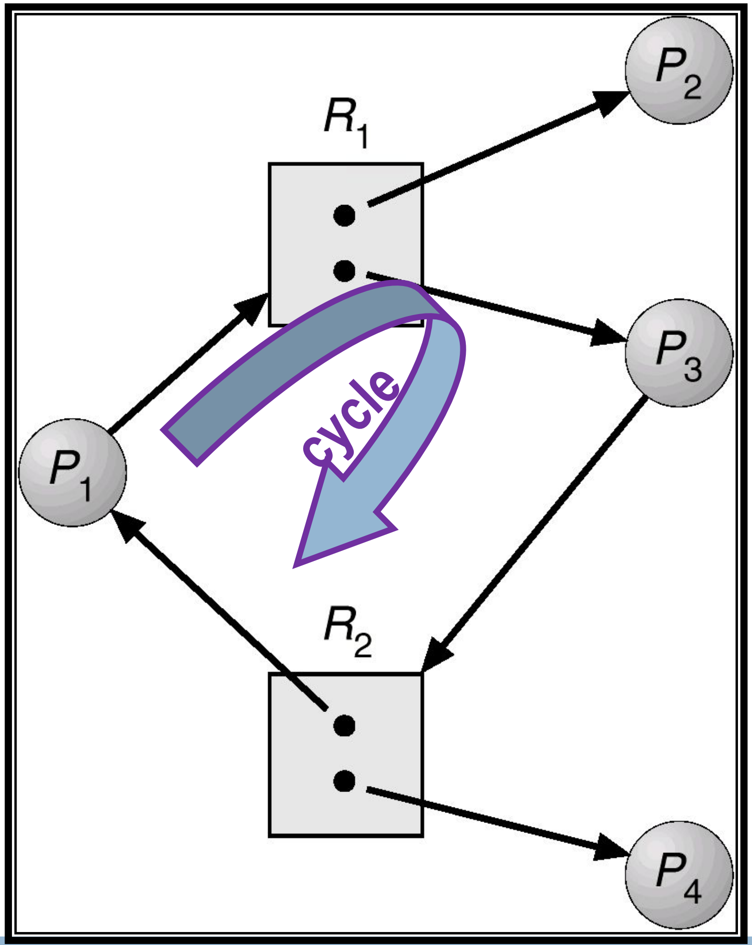


RAMAIAH Institute of Technology

Example: RAG With Cycle But No Deadlock

P1->R1->P3->R2->P1 is a cycle.

But there is no deadlock in this system. WHY?



Because P2 will eventually release its hold onR1. Then P1 will be assigned R1 and the cycle will be broken.

Deadlock Handling

Deadlocks can occur in multitasking systems

How do we deal with them?

We can

- Make sure the system never enters a deadlocked state
- Allow deadlocks but detect them and recover from them
- Ignore the problem altogether

Most systems ignore the problem and leave it to the user to detect the deadlock and restart their processes.

Deadlock Prevention

Take away any one of the necessary conditions

Mutual exclusion

- Some resources are intrinsically non-sharable.

Hold and wait

- Force processes to request all resources at startup time
- Inefficient and can lead to starvation

No preemption

- If Pi requesting Rj causes a wait, preempt all other resources held by Pi and make Pi wait for them too
- OK for CPU and memory, but difficult for I/O devices

Circular wait

- Force an enumeration on resources, and force processes to follow the ordering in their requests
- Inflexibility for the programmers

Deadlock Prevention

Three conditions cannot always be prevented:

- Mutual exclusion (some resources not sharable)
- Hold and wait (advance allocation is too wasteful)
- No preemption (hard to preempt some resource types)

Circular wait is easily prevented BUT,

- Forcing an ordering on resources is inconvenient
- Why should I have to request the tape drive before the printer?

Deadlock Avoidance

The basic idea:

- Watch system resource requests
- For each request, decide whether granting is "safe"
- If safe, grant request
- If not safe, force requesting process to wait

For single-instance resources, there is a simple Resource Allocation Graph algorithm to determine safety.

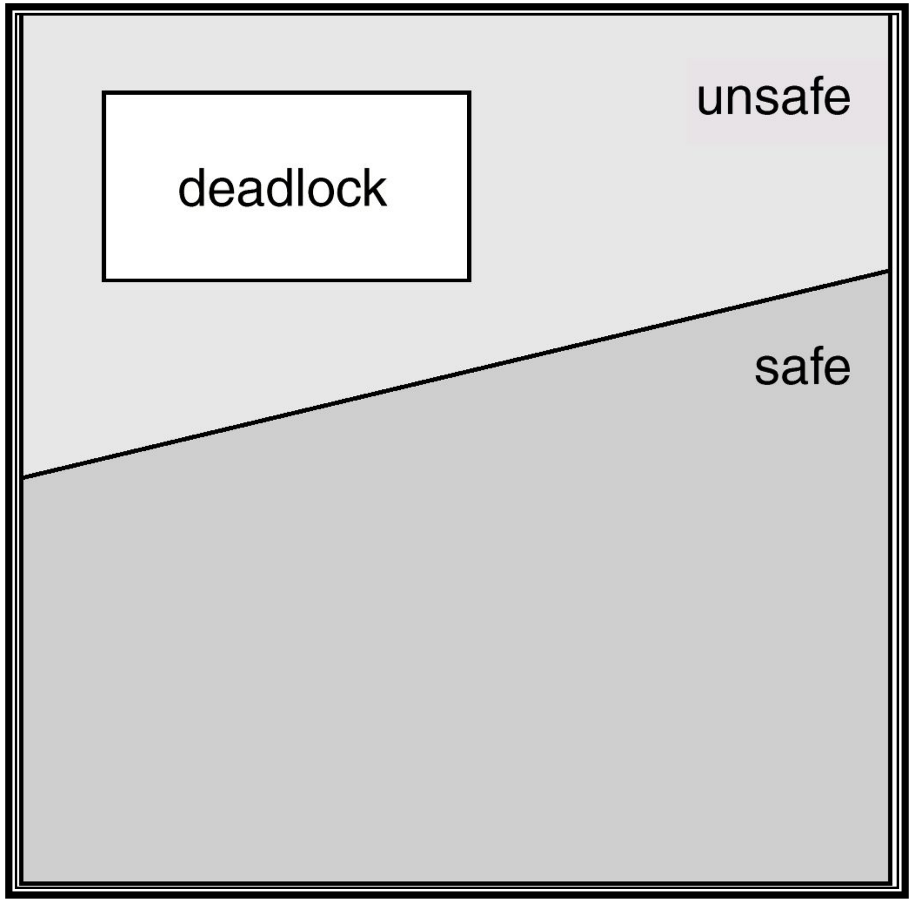
For multiple-instance resources, use the Banker's Algorithm

Relationship between safe, unsafe, and deadlocked states

Safe state: no deadlock

Unsafe state: possible deadlock

Using the safe state algorithm is non-optimal but at least avoids deadlocks.



Resource Allocation Graph Algorithm

Claim edge Pi->Rj:

- Means process Pi may request resource Rj
- Represented by a dashed line

Algorithm:

- When a process requests a resource, claim edges convert to request edges
- When Pi releases Rj, the Pi->Rj assignment edge reverts to a claim edge.
- All resources must be claimed at startup time.

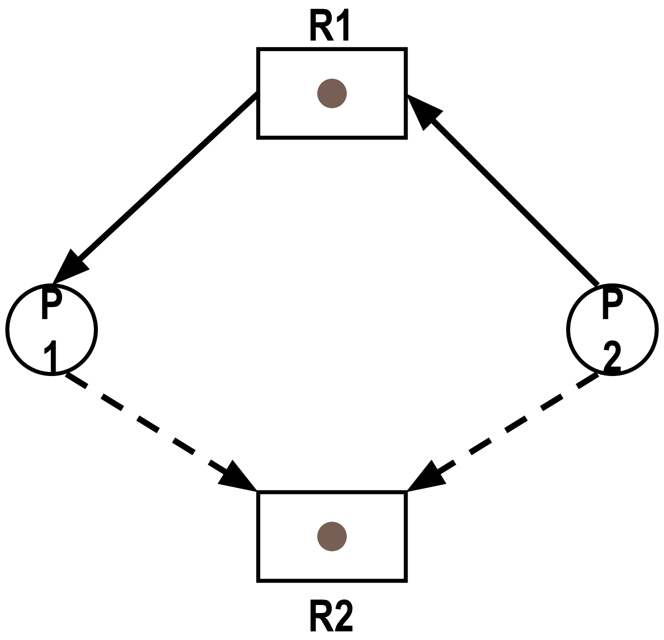
Deadlock Avoidance Resource Allocation Graph: Example

Safe state: no cycles in the resource allocation graph.

Unsafe state: there is a cycle in the resource allocation graph.

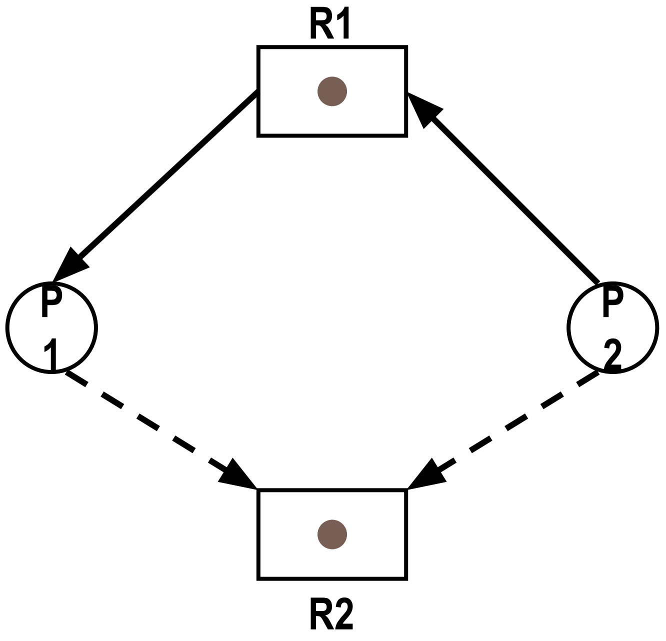
This state is safe.

WHY?



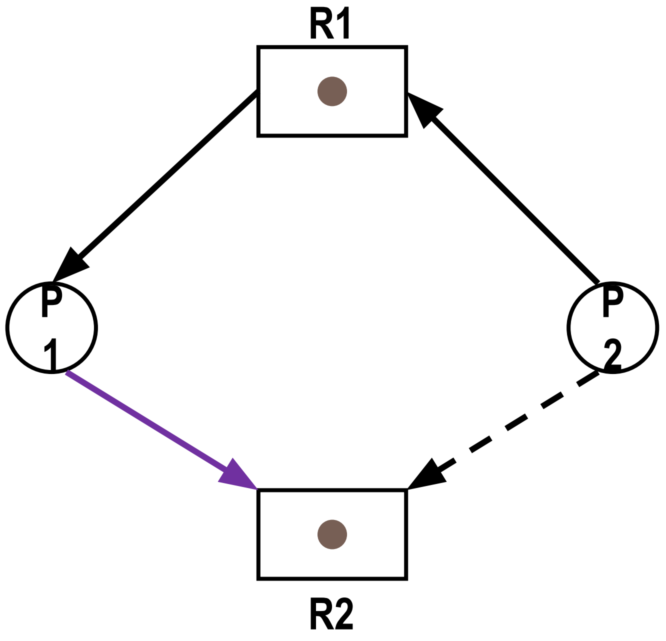
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



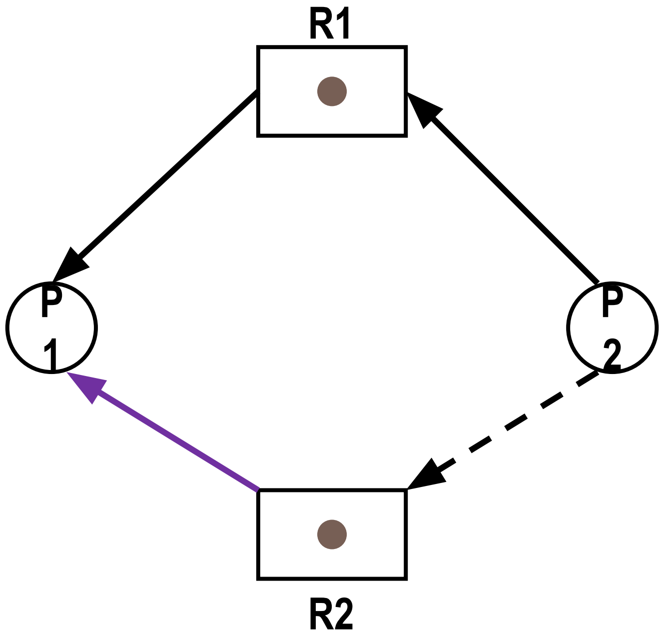
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



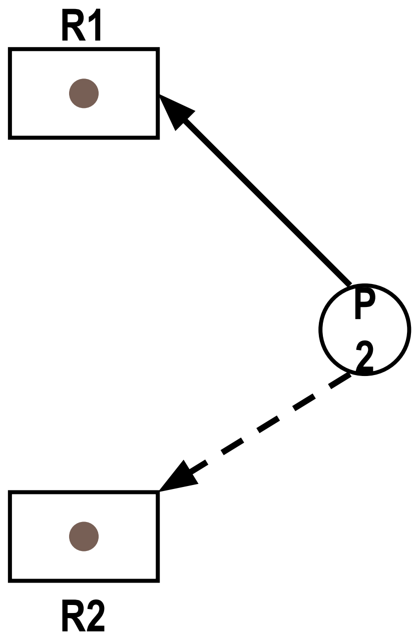
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



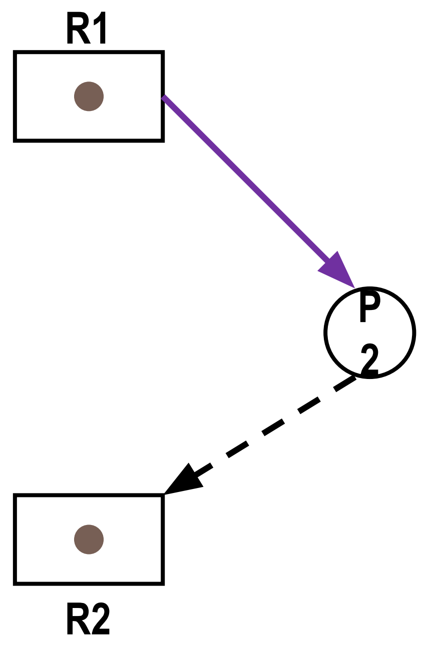
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



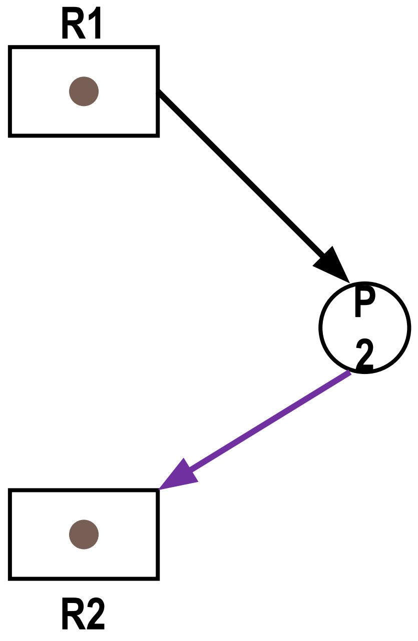
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



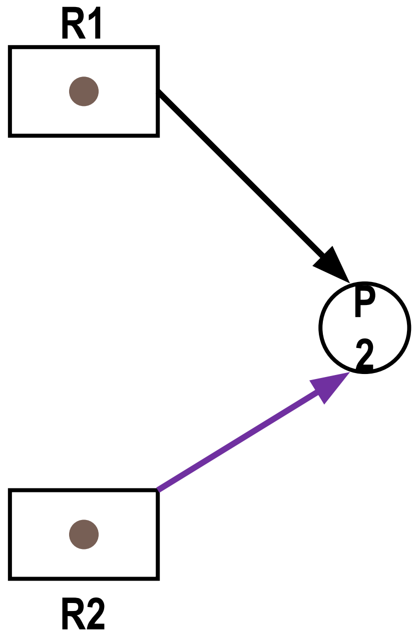
Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



Deadlock Avoidance Resource Allocation Graph: Example

- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.



Deadlock Avoidance Resource Allocation Graph: Example

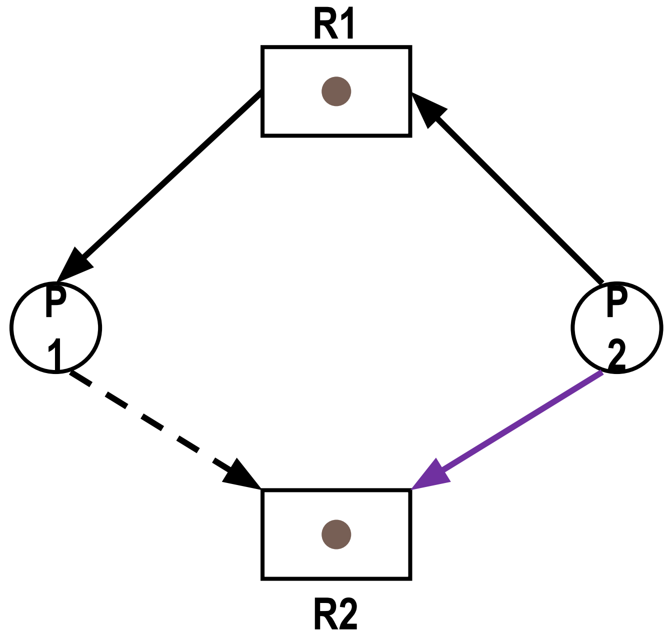
- We can let P1 run.
- P1 may request R2 if necessary.
- If P1 requests R2, then P1 will immediately get R2.
- P1 will then complete.
- Then P2 can run and maybe request R2, get R2, and complete.

Deadlock Avoidance Resource Allocation Graph: Example

But suppose P2 requests R2.

This puts the system in an unsafe state.

WHY?



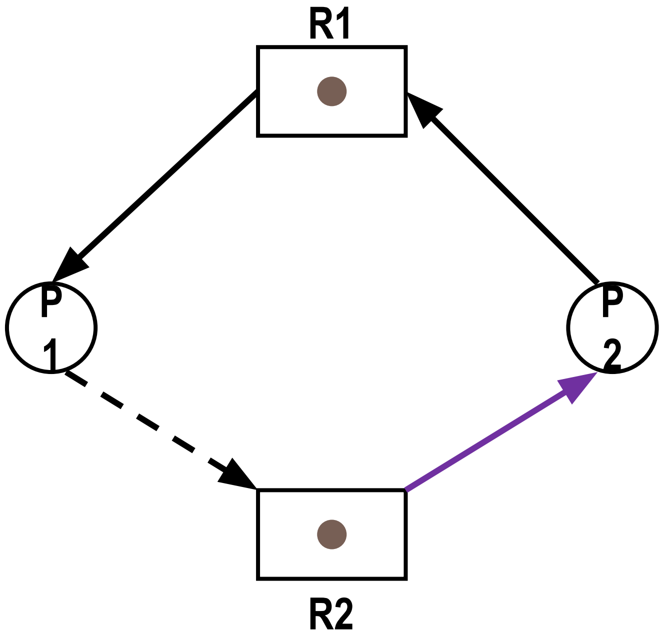
Deadlock Avoidance Resource Allocation Graph: Example

But suppose P2 requests R2.

This puts the system in an unsafe state.

WHY?

If we give R2 to P2



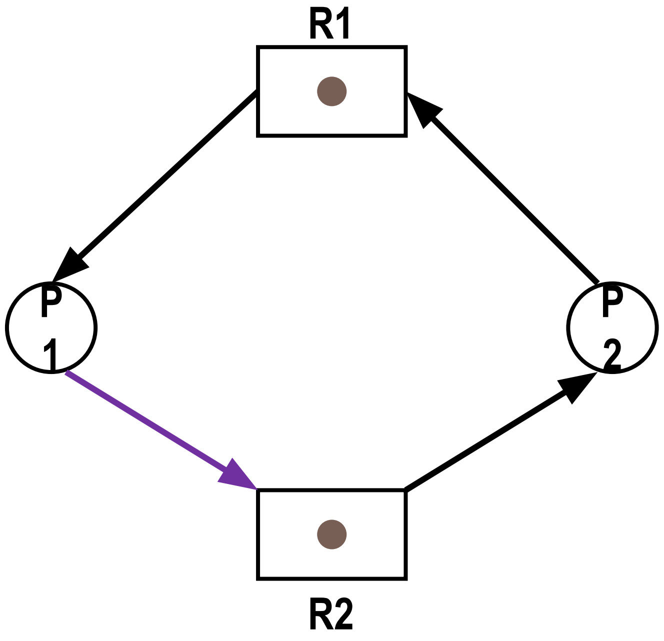
Deadlock Avoidance Resource Allocation Graph: Example

But suppose P2 requests R2.

This puts the system in an unsafe state.

WHY?

If we give R2 to P2 and if P1 requests R2,



Deadlock Avoidance Resource Allocation Graph: Example

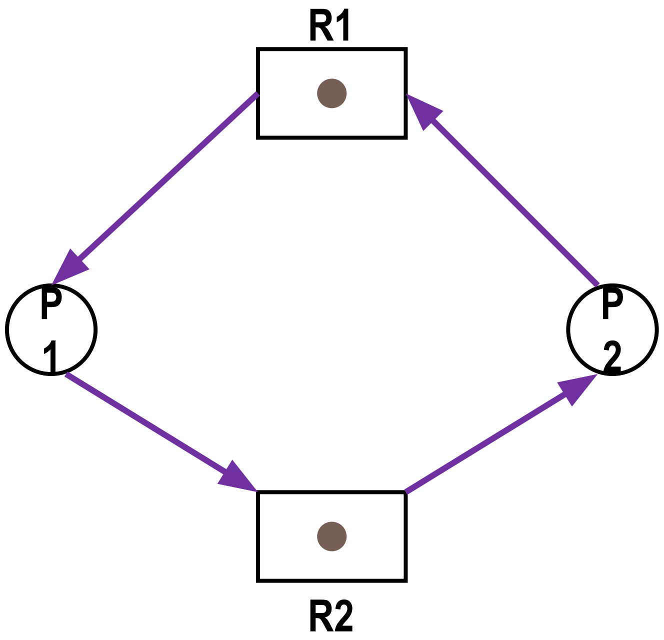
But suppose P2 requests R2.

This puts the system in an unsafe state.

WHY?

If we give R2 to P2 and if P1 requests R2, we will have a deadlock.

Therefore, we should not grant P2's request until P1 completes.



Banker’s Algorithm:

- Tentatively grant each resource request
- Analyze resulting system state to see if it is “safe”.
- If safe, grant the request
- if unsafe refuse the request (undo the tentative grant)
- block the requesting process until it is safe to grant it.

Data Structures for the Banker's Algorithm

Let n = number of processes,

m = number of resource types

Available: Vector of length *m*. If Available [*j*] = *k*, there are *k* instances of resource type R_j currently available

Max: n x m matrix. If *Max* [*i,j*] = *k*, then process P_i will request at most *k* instances of resource type R_j.

Alloc: n x m matrix. If Alloc[*i,j*] = *k* then P_i is currently allocated (i.e. holding) *k* instances of $R_{j.}$

Need: n x m matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

$$Need\,[i,j] = Max[i,j] - Alloc\,[i,j].$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
   Initialize:
   - *Work* := *Available*
   - *Finish* [i] == *false* for *i* = 1,2, …, *n.*
2. Find an *i* such that both:
   - *Finish* [i] == *false*
   - $Need_i \leq Work$

   If no such *i* exists, go to step 4.
3. $Work := Work + Allocation_i$

   (Resources freed when process completes!)

   Finish[i] := *true*

   go to step 2.
4. If *Finish* [i] = true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for P_i .
$Request_i [j] = k$ means process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, error (process exceeded its maximum claim).
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, (resources not available).
3. "Allocate" requested resources to P_i as follows:

   Available := Available - $Request_i$
   $Alloc_i := Alloc_i + Request_i$
   $Need_i := Need_i – Request_i$
   If safe $\Rightarrow$ the resources are allocated to P_i.
   If unsafe $\Rightarrow$ restore the old resource-allocation state and block P_i

Example of Banker's Algorithm

5 processes P_0 through P_4

3 resource types A (10 units), B (5 units), and C (7 units).

Snapshot at time T_0:

	Allocation	***Max***	***Available***
	A B C	***A B C***	***A B C***
P_0	**0 1 0**	**7 5 3**	**3 3 2**
P_1	**2 0 0**	**3 2 2**	
P_2	**3 0 2**	**9 0 2**	
P_3	**2 1 1**	**2 2 2**	
P_4	**0 0 2**	**4 3 3**	

Example (cont)

Need = Max – Allocation

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Now P_1 requests (1,0,2)

Check that Request ≤ Available

(that is, $(1,0,2) \le (3,3,2)) \Rightarrow$ *true*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Deadlock Detection and Recovery

Deadlock prevention/avoidance is complex and underutilizes resources.

Alternative: allow system to deadlock, detect the deadlock, then try to recover.

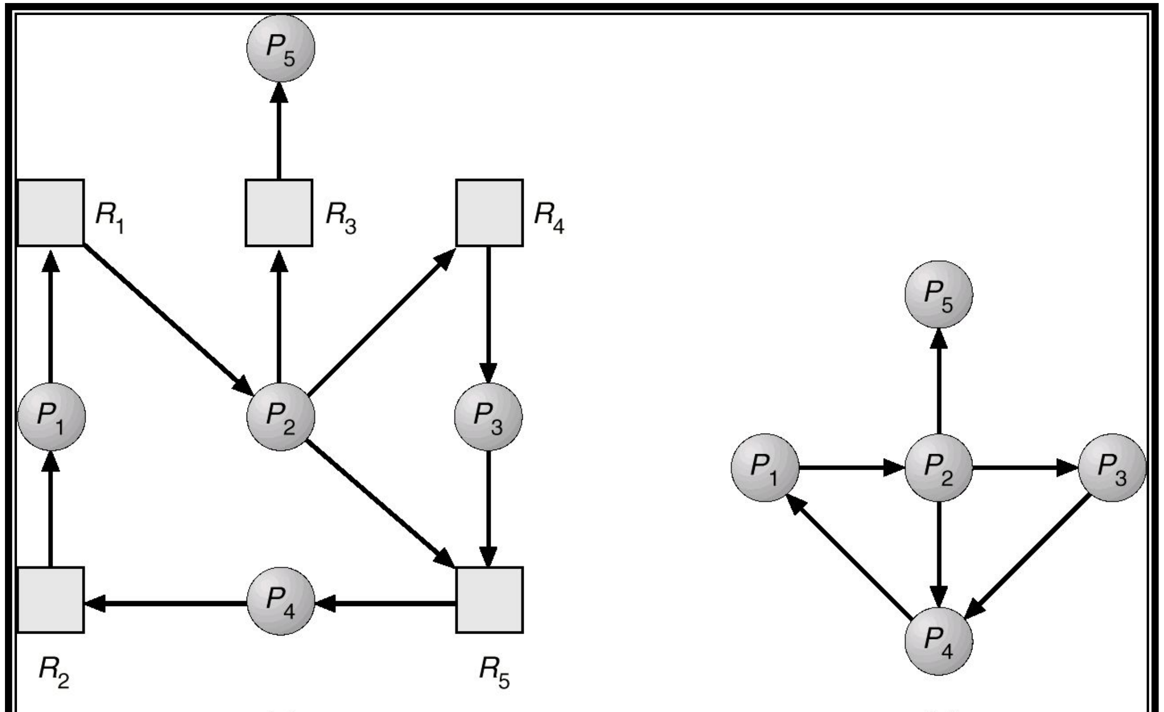
Deadlock detection with single-instance resources:

- A simple resource allocation graph algorithm exists
- Construct “Wait-for” graph and find cycles

Multiple instance resources:

- A more complex algorithm is required

Constructing the Wait-For Graph



(a) (b)

Recovery

Once deadlock is detected, must attempt recovery.

Solution 1: Process termination

- Abort all processes in the deadlock: costly
- Abort one process at a time until deadlock is broken:

  [This requires a policy for selecting which process to abort.]

Solution 2: Resource preemption

- Preempt resources and give to other processes until deadlock is broken

Question and Answer 1

Deadlocks:

How they occur
How to avoid them
How to detect and recover from them

Thank you