Lecture 6
Deadlocks

I233E OPERATING SYSTEMS

RAZVAN BEURAN

Today’s Topics

Deadlocks

- What they are

Mechanisms

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection

Deadlock Model

Resource type

- CPU, memory location, file, I/O device, ...

Resource instance

- One resource item of a given type

Steps to use the resource

- Processes request some resource type
- Request is fulfilled with any instance of the requested type
- Processes use that resource instance, then release it

Sequence: *Request → Use → Release*

Necessary Conditions for Deadlock

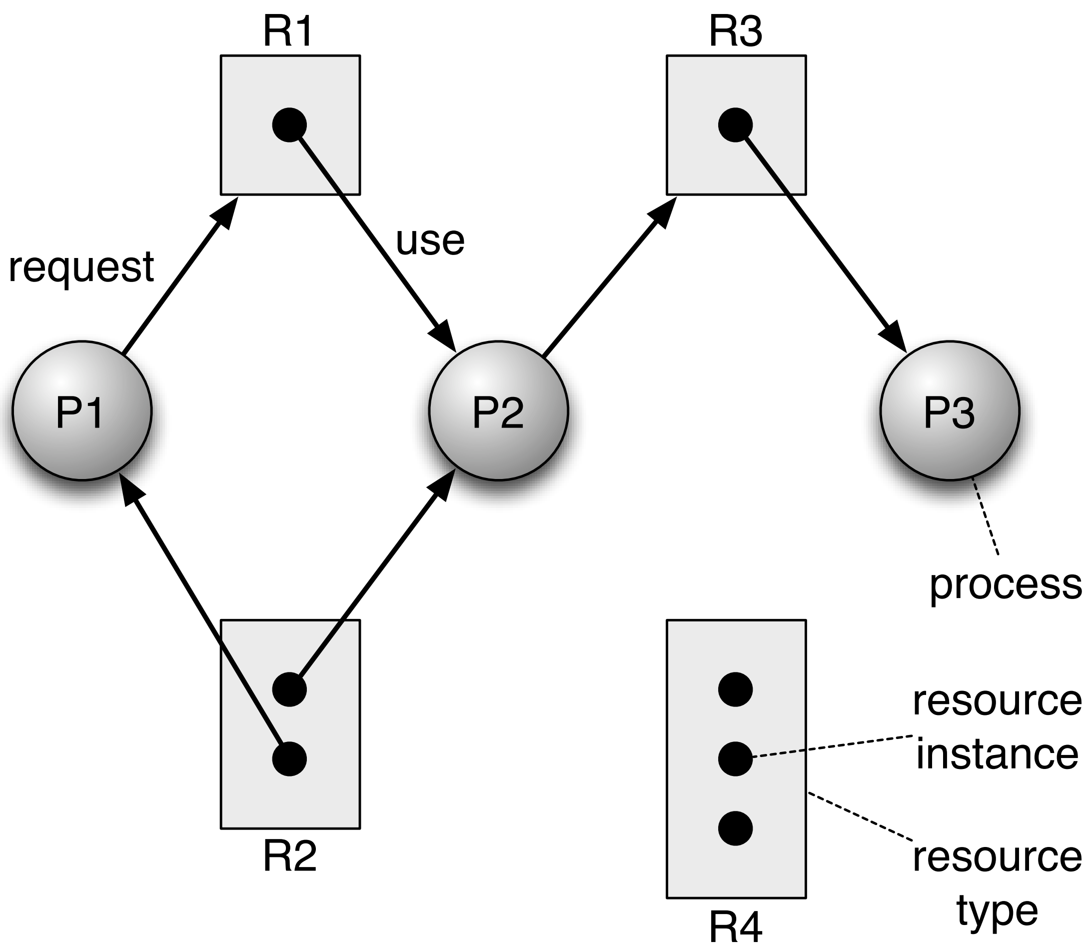
Mutual exclusion
- Some resource is used exclusively

Hold and wait
- Some process holds a resource and waits for others

No preemption
- Resource can only be released voluntarily

Circular wait
- Cycle: `P0` waits for `P1`, which waits for …, which waits for `P0`

Resource Allocation Graph



Graph Interpretation

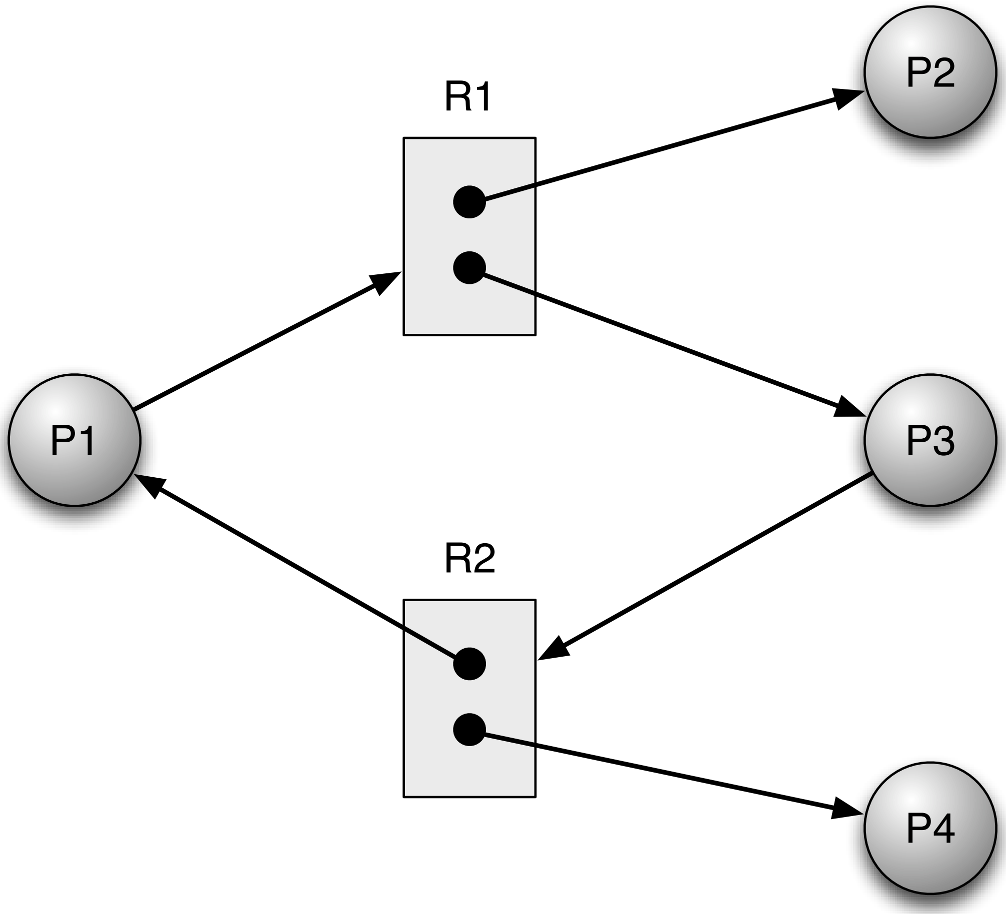
General case

- No cycle → no deadlock can occur
- Cycle means that it is *possible* for a deadlock to occur

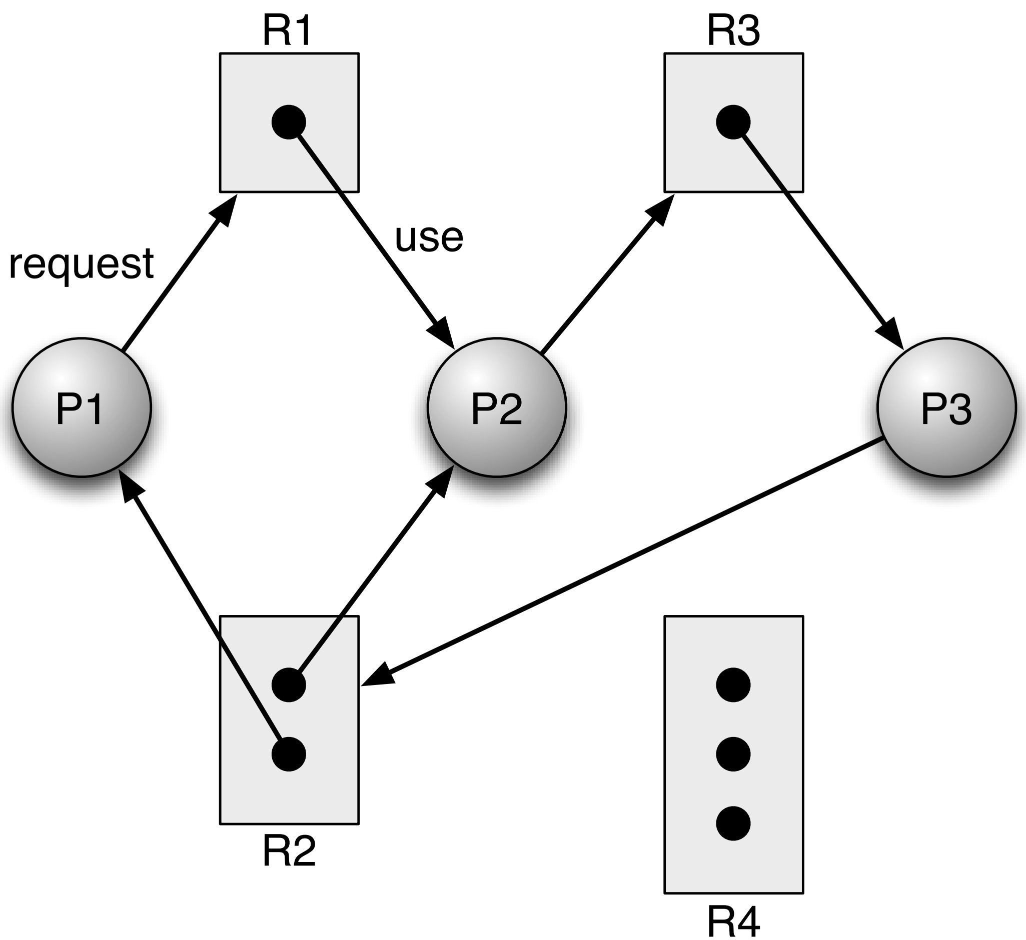
Special case: 1 resource type, 1 resource instance

- Cycle means deadlock

Cycle, but No Deadlock



Deadlock Example



Dealing With Deadlocks

Approach 1

- Prevent / avoid deadlocks
  - Prevent = Remove one condition for deadlock
  - Avoid = Keep out of “unsafe” situations

Approach 2

- When deadlock occurs
  - Detect the deadlock
  - Recover from deadlock

Approach 3

- Ignore the problem... and pray ☺

Deadlock Prevention

Prevention Mechanisms

Idea: Avoid any of the 4 necessary deadlock conditions

Avoid mutual exclusion →

- Allow resources to be shared

Avoid hold and wait →

- Make all requests first, before execution (*grouped request*)

Avoid no preemption →

- Allow forced release of resources by OS

Avoid circular wait →

- Create total ordering on resources, e.g., request resources in *increasing order* of id

Deadlock Avoidance

Avoidance Mechanisms

Idea

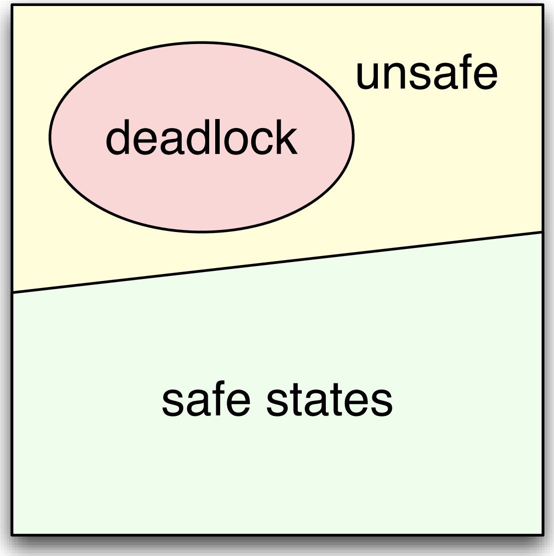
- Get additional information about requests
- Keep track of resource allocation state
- Allow requests only if they are “safe”

Safe state

- There exists some allocation sequence that leads to normal termination

Unsafe state

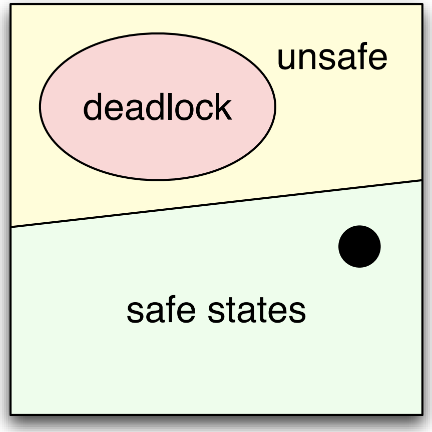
- Every allocation sequence leads to deadlock



Safe vs. Unsafe Allocation (1)

Proc. ID	Max. Needs	Current Holds
P0	10	5
P1	4	2
P2	9	2

Total resource instances = 12



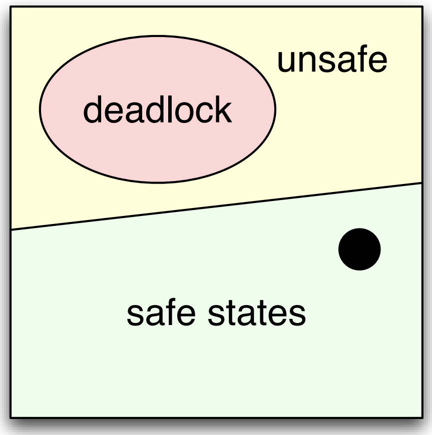
Allocation Sequence #1

Time	P0	P1	P2	Free
t0	5	2	2	3
t1	5	**4**	2	**1**
t2	5	-	2	**5**
t3	**10**	-	2	**0**
t4	-	-	2	**10**
t5	-	-	**9**	**3**
t6	-	-	-	**12**

Safe vs. Unsafe Allocation (2)

Proc. ID	Max. Needs	Current Holds
P0	10	5
P1	4	2
P2	9	2

Total resource instances = 12



Allocation Sequence #2

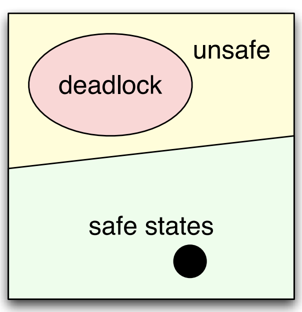
Time	P0	P1	P2	Free
t0	5	2	2	3
t1	5	2	**3**	**2**
t2	5	**4**	3	**0**
t3	5	-	3	**4**
t4	5; **req: 5**	-	3; **req: 6**	4

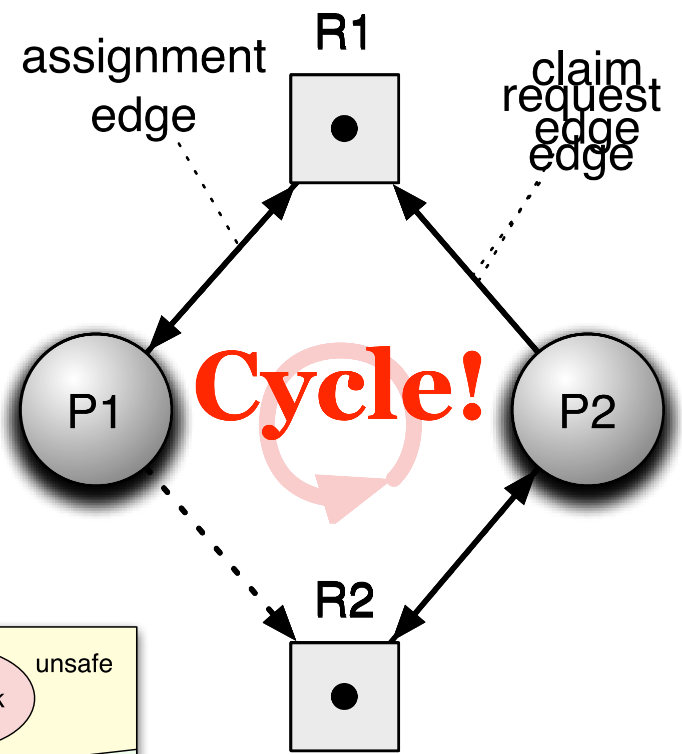
Single-Instance Case

Allocation graph algorithm

- Initially
  - Process “claims” resources (*claim edge*)
- Runtime
  - Resource is requested (*request edge*)
  - Resource is assigned (*assignment edge*)

Unsafe when cycle is created





Banker's Algorithm

System assumptions

- Several resource types
- Several resource instances of each type
- Initially, processes announce max. need for each type

Has 2 parts

- Safety algorithm
  - Detect if current system state is safe
- Resource request algorithm
  - Authorize (or not) resource requests

Banker's Algorithm (cont.)

System assumptions

- *m* resource types; *n* processes

Data structures

- *Available* (vector of size *m*)
  - *Available*[*j*] = # available instances of resource type *j*
- *Max* (*n* x *m* matrix)
  - *Max*[*i,j*] = max. demand of *Pi* for resource type *j*
- *Allocation* (*n* x *m* matrix)
  - *Allocation*[*i,j*] = # instances of type *j* held by *Pi*
- *Need* (*n* x *m* matrix)
  - *Need*[*i,j*] = *Max*[*i,j*] - *Allocation*[*i,j*]

Safety Algorithm

Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

	Allocation A	Allocation B	Allocation C	Max A	Max B	Max C	Available A	Available B	Available C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Safety Algorithm (2)

	Allocation			Need		
	A	*B*	*C*	*A*	*B*	*C*
P0	0	1	0	7	4	3
P1	2	0	0	1	2	2
P2	3	0	2	6	0	0
P3	2	1	1	0	1	1
P4	0	0	2	4	3	1

Available		
A	*B*	*C*
3	3	2

Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

Need := *Max* - *Allocation*

Safety Algorithm (3)

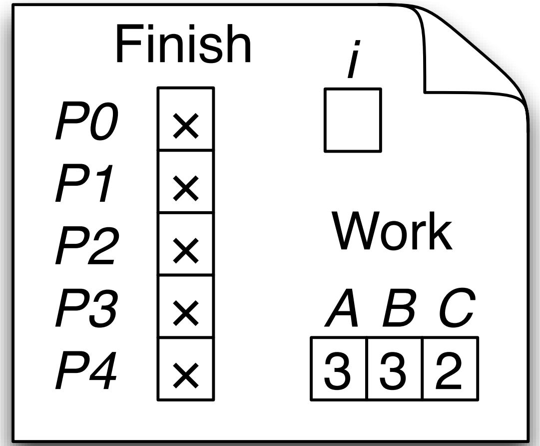
	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true



Safety Algorithm (4)

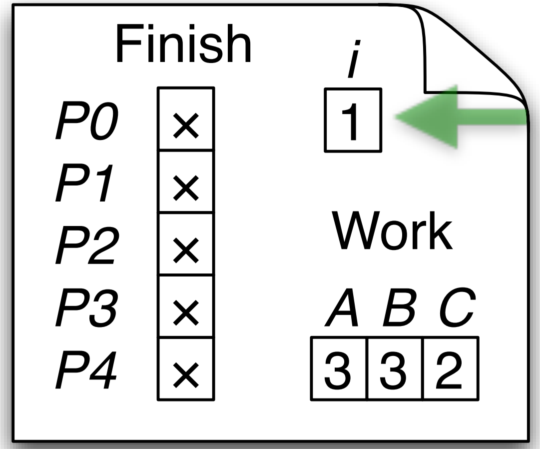
Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			



Safety Algorithm (5)

Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P0	0	1	0	7	4	3	3	3	2
✓ *P1*	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

	Finish
P0	×
P1	○
P2	×
P3	×
P4	×

i
1

Work

A	*B*	*C*
5	3	2

Safety Algorithm (6)

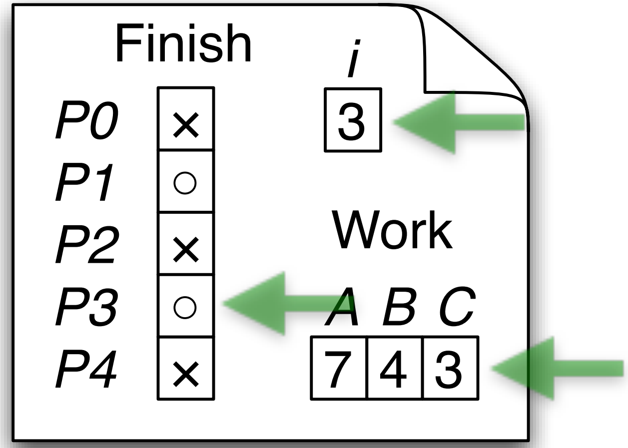
Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P0	0	1	0	7	4	3	3	3	2
✓ *P1*	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
✓ *P3*	2	1	1	0	1	1			
P4	0	0	2	4	3	1			



Safety Algorithm (7)

Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

		Allocation			Need			Available		
		A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
✓	*P0*	0	1	0	7	4	3	3	3	2
✓	*P1*	2	0	0	1	2	2			
	P2	3	0	2	6	0	0			
✓	*P3*	2	1	1	0	1	1			
	P4	0	0	2	4	3	1			

	Finish
P0	○
P1	○
P2	×
P3	○
P4	×

i: 0

Work

A	*B*	*C*
7	5	3

Safety Algorithm (8)

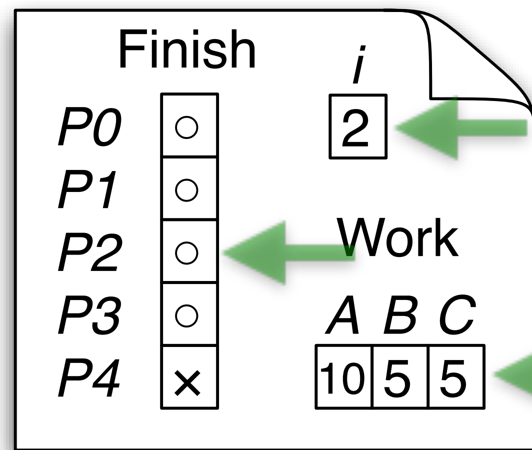
Variables

- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t.
  Finish[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

		Allocation			Need			Available		
		A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
✓	*P0*	0	1	0	7	4	3	3	3	2
✓	*P1*	2	0	0	1	2	2			
✓	*P2*	3	0	2	6	0	0			
✓	*P3*	2	1	1	0	1	1			
	P4	0	0	2	4	3	1			



Safety Algorithm (9)

Variables

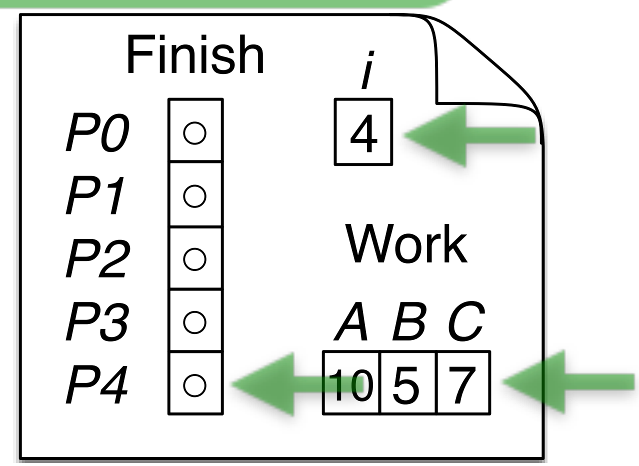
- Define *Work* := *Available*
- Define *Finish*[0…*n*-1] := false

Steps

- While exists *i* s.t. *Finish*[*i*]==false && *Need*[*i*] ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
- System is safe iff. for all *Pj*, *Finish*[*j*] == true

	Allocation			Need		
	A	*B*	*C*	*A*	*B*	*C*
✓ *P0*	0	1	0	7	4	3
✓ *P1*	2	0	0	1	2	2
✓ *P2*	3	0	2	6	0	0
✓ *P3*	2	1	1	0	1	1
✓ *P4*	0	0	2	4	3	1

Available		
A	*B*	*C*
3	3	2



Safety Algorithm (10)

Variables

- Define *Work* := *Available*
- Define *Finish*[0..*n*-1] := false

St

- *Work* := *Work* + *Allocation*[*i*]
- *Finish*[*i*] := true
- System is safe iff.
  for all *Pj*, *Finish*[*j*] == true

A possible allocation sequence (P1::P3::P0::P2::P4) exists → the system is in a **safe state**

	Allocation A	Allocation B	Allocation C	Need A	Need B	Need C
✓ *P0*	0	1	0	7	4	3
✓ *P1*	2	0	0	1	2	2
✓ *P2*	3	0	2	6	0	0
✓ *P3*	2	1	1	0	1	1
✓ *P4*	0	0	2	4	3	1

Available

A	B	C
3	3	2

Finish

P0	○
P1	○
P2	○
P3	○
P4	○

i: 4

Work

A	B	C
10	5	7

Resource Request Algorithm

Request made by process *Pi*

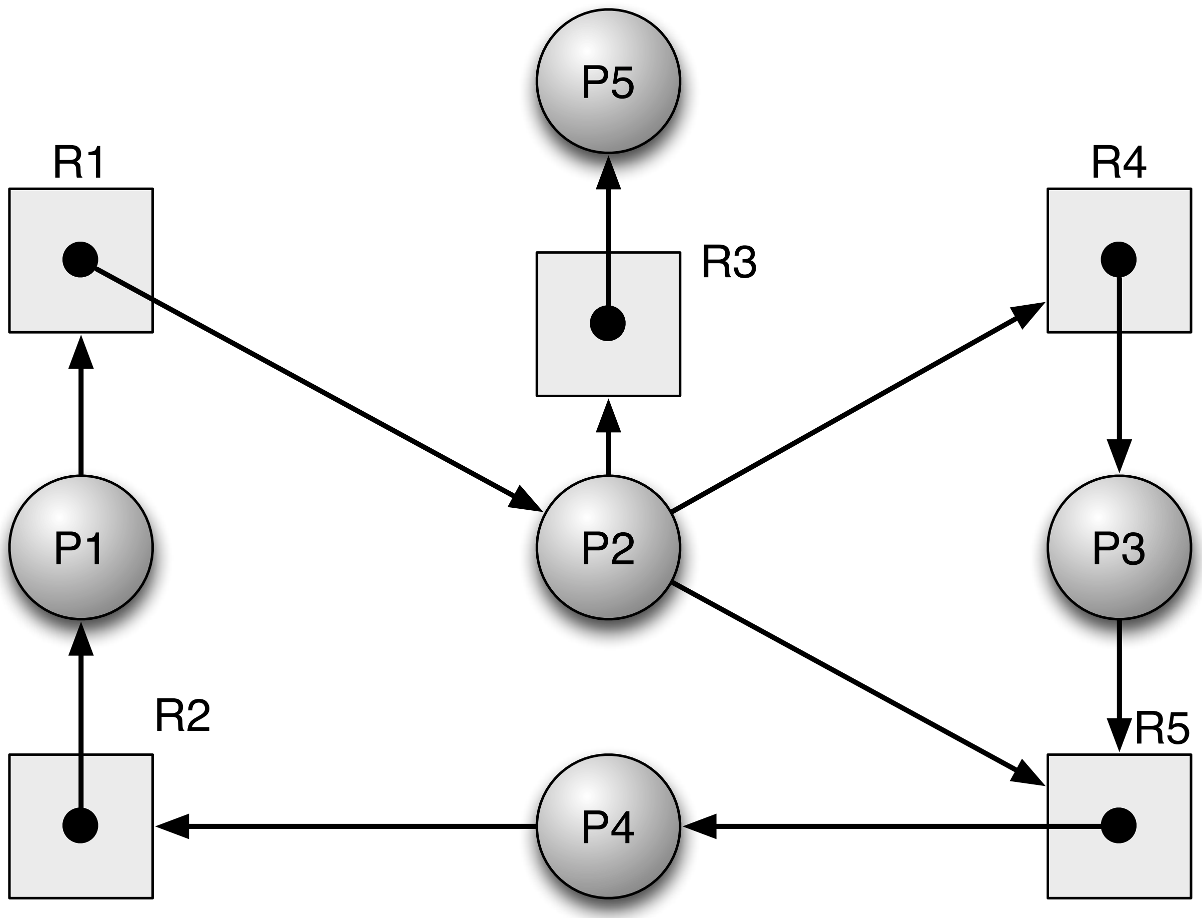
- *Request_Pi*[*j*] = # instances requested of type *j*

Steps

1. If not *Request_Pi* ≤ *Need*[*i*] => **reject** (request more than max.)
2. If not *Request_Pi* ≤ *Available* => *Pi* must **wait**
3. Try the following new states
   - *Available'* := *Available* – *Request_Pi*
   - *Allocation'*[*i*] := *Allocation*[*i*] + *Request_Pi*
   - *Need'*[*i*] := *Need*[*i*] – *Request_Pi*
4. Run safety algorithm with State (Available', Allocation', Need')
   - If **safe** => **accept** to give resources
   - If **unsafe** => *Pi* must **wait**

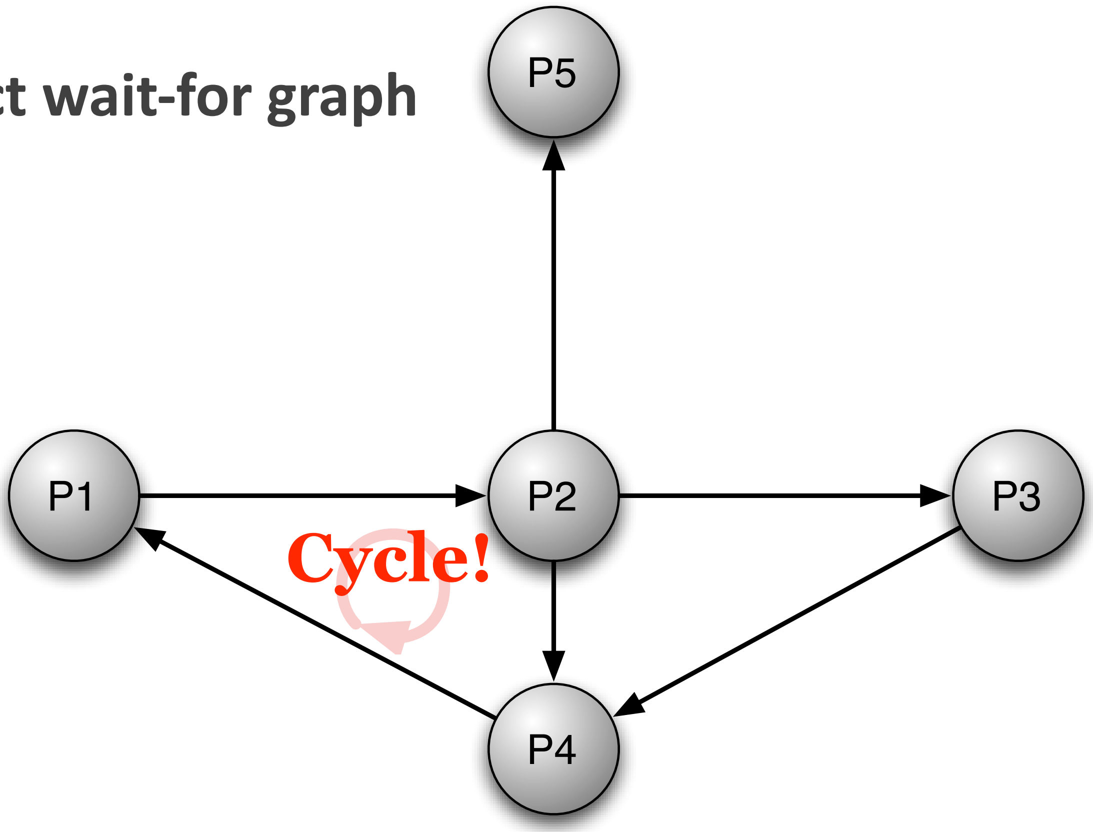
Deadlock Detection

Detection: 1 Inst. / Type



Detection: 1 Inst. / Type (cont.)

Construct wait-for graph



Detection: *m* Inst. / Type

Variables

- Define *Work* := *Available*
- Define *Finish*[*i*] := (*Allocation*[*i*] == 0)

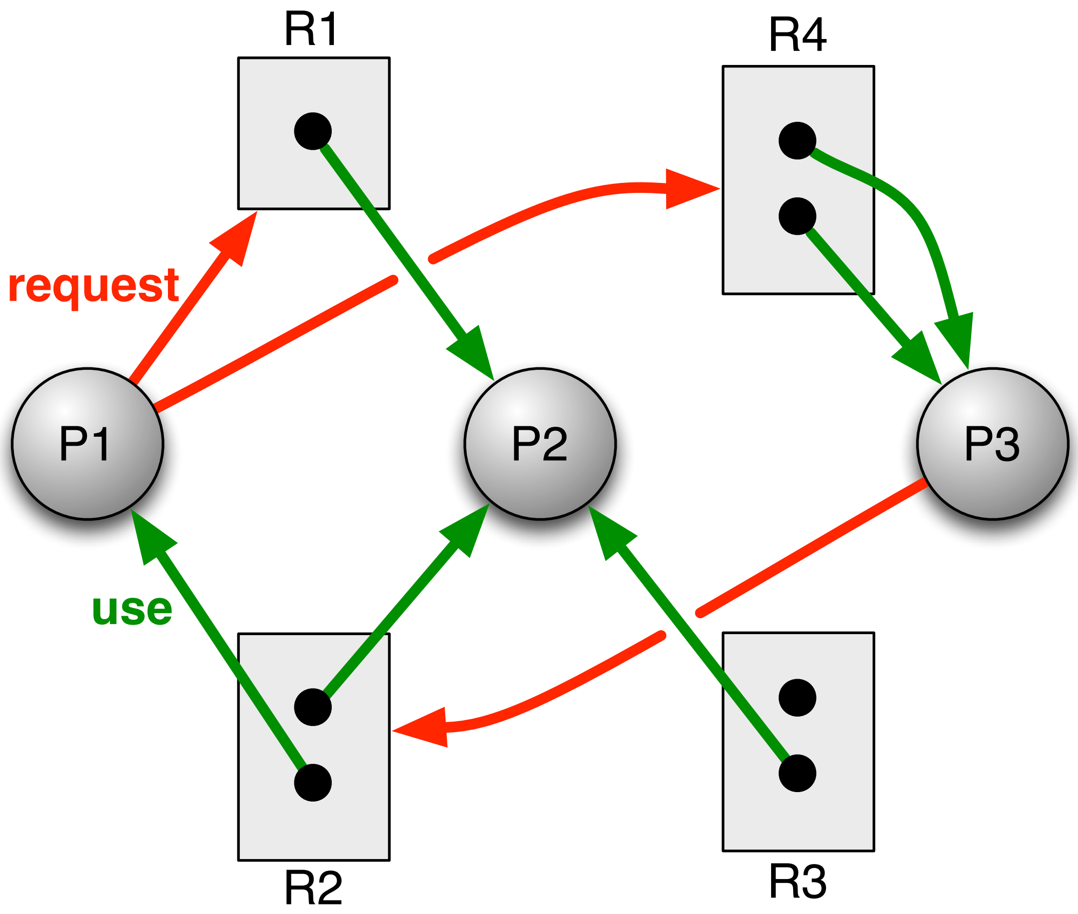
Steps (similar to Banker's safety alg.)

- While exists *i* s.t. *Finish*[*i*]==false && *Request_Pi* ≤ *Work*
  - *Work* := *Work* + *Allocation*[*i*]
  - *Finish*[*i*] := true
  - If exists *Pj* s.t. (*Finish*[*j*] == false) → *Pj* is deadlocked

Notes

- Algorithm assumes processes will not ask for more resources
- If they do, then deadlock must be detected again later

Example: *m* Inst. / Type



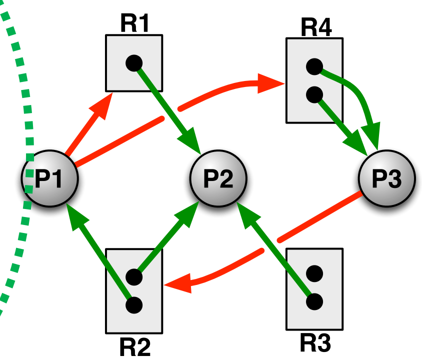
Example: *m* Inst. / Type (cont.)

Initial conditions

- *Allocation* = [P1: [0 1 0 0] P2: [1 1 1 0] P3: [0 0 0 2]]
- *Request* = [P1: [1 0 0 1] P2: [0 0 0 0] P3: [0 1 0 0]]
- *Work* = [0 0 1 0]
- *Finish* = [false false false]

Steps

- If exists *i* s.t. *Finish*[i]==false && *Request_Pi* ≤ *Work* → ***i* = 2**
  - *Work* = *Work* + P2:[1 1 1 0] = [1 1 2 0]; *Finish*[P2]=true
- If exists *i* s.t. ... → ***i* = 3**
  - *Work* = *Work* + P3:[0 0 0 2] = [1 1 2 2] ; *Finish*[P3]=true
- If exists *i* s.t. ... → ***i* = 1**
  - *Work* = *Work* + P1:[0 1 0 0] = [1 2 2 2] ; *Finish*[P1]=true



Recovery From Deadlock

Process termination

- Abort all deadlocked processes
- Abort one process at a time

Resource preemption

- Issues
  - How to select target (process, resource)
  - How to do rollback (e.g., restart)
  - How to avoid starvation

Summary

Deadlock

- Situation when execution cannot continue

Deadlock prevention

- Avoid any of the 4 necessary conditions

Deadlock avoidance

- Safe versus unsafe allocation
- Banker's algorithm: safety alg. + resource request alg.

Deadlock detection

- Similar to Banker's safety algorithm

Next Time

Main memory

Paging

Segmentation