Deadlock

Operating Systems

- Objectives
  - describe deadlock, and forms of prevention, avoidance, detection, and recovery

Contents

1. What is Deadlock?

- An example from US Kansas law:
  - “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

In Picture Form:

1.1. System Deadlock

- A process must request a resource before using it, and must release the resource after finishing with it.

- A set of processes is in a *deadlock state* when every process in the set is waiting for a resource that can only be released by another process in the set.

1.2. Necessary Conditions for Deadlock

- Mutual Exclusion
  - at least one resource must be held in non-shareable mode

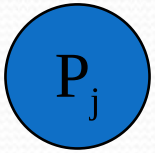
- Hold and Wait
  - a process is holding a resource and waiting for others

continued

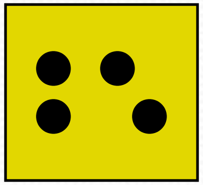
- No Preemption
  - only the process can release its held resource
- Circular Wait
  - $\{P_0, P_1, \ldots, P_n\}$
  - P_i is waiting for the resource held by P_{i+1}; P_n is waiting for the resource held by P_0

1.3. Resource Allocation Graph

- A set of processes $\{P_0, P_1, \ldots\}$



- A set of resource types $\{R_1, R_2, \ldots\}$, together with instances of those types.



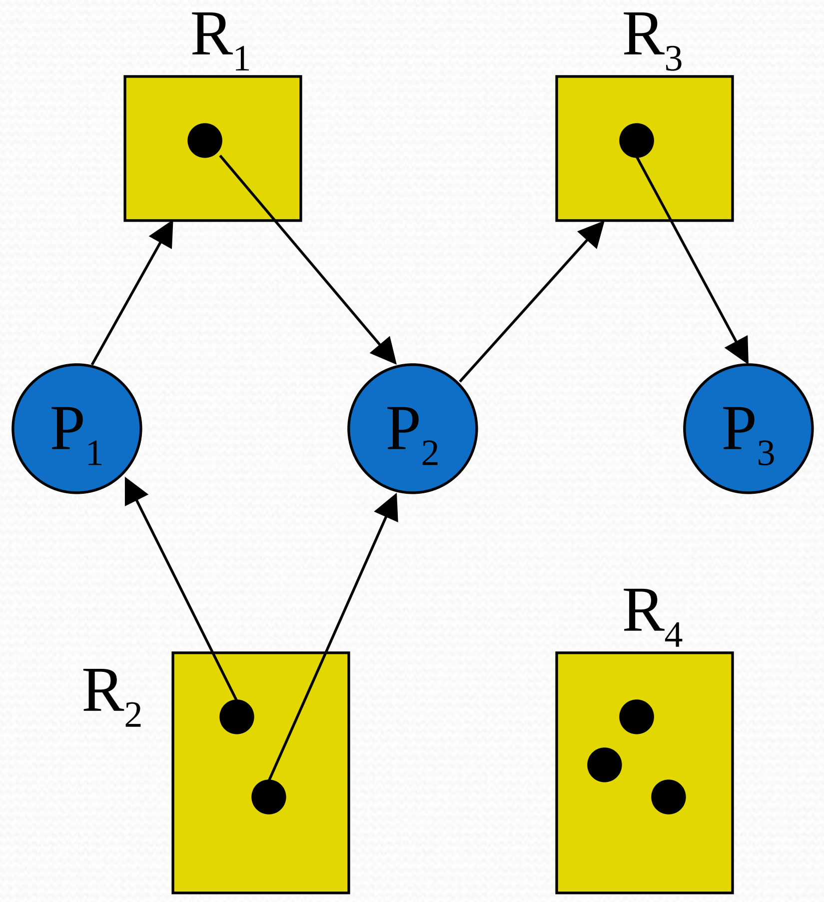
R_k

Edge Notation

- $P_i \rightarrow R_j$
  - process i has requested an instance of resource j
  - called a *request edge*

- $R_j \rightarrow P_i$
  - an instance of resource j has been assigned to process i
  - called an *assignment edge*

Example Graph

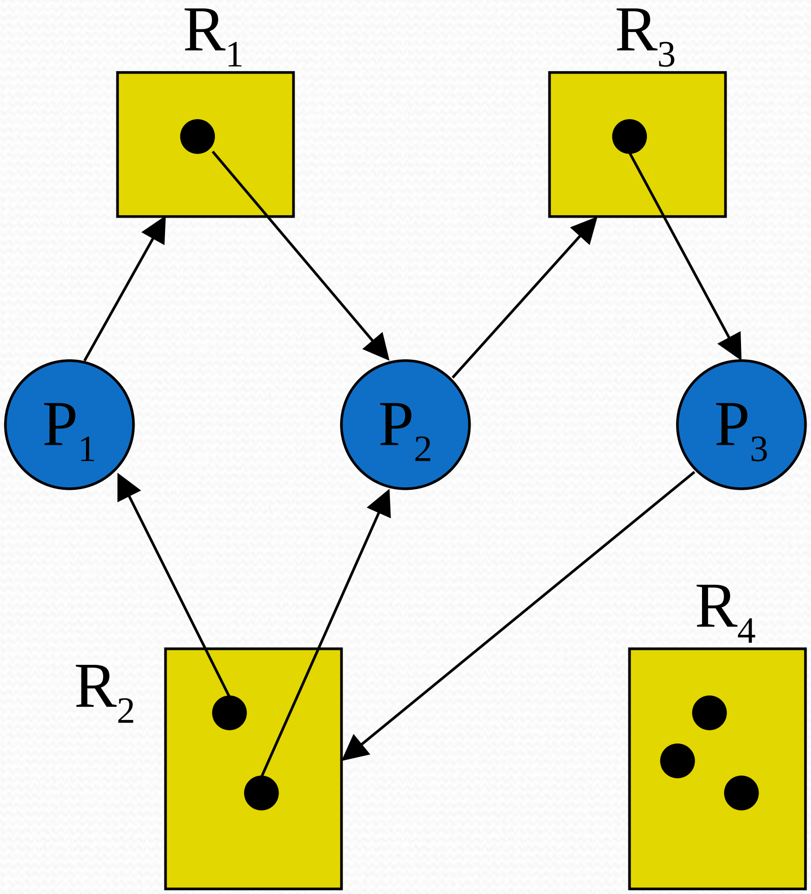


Finding a Deadlock

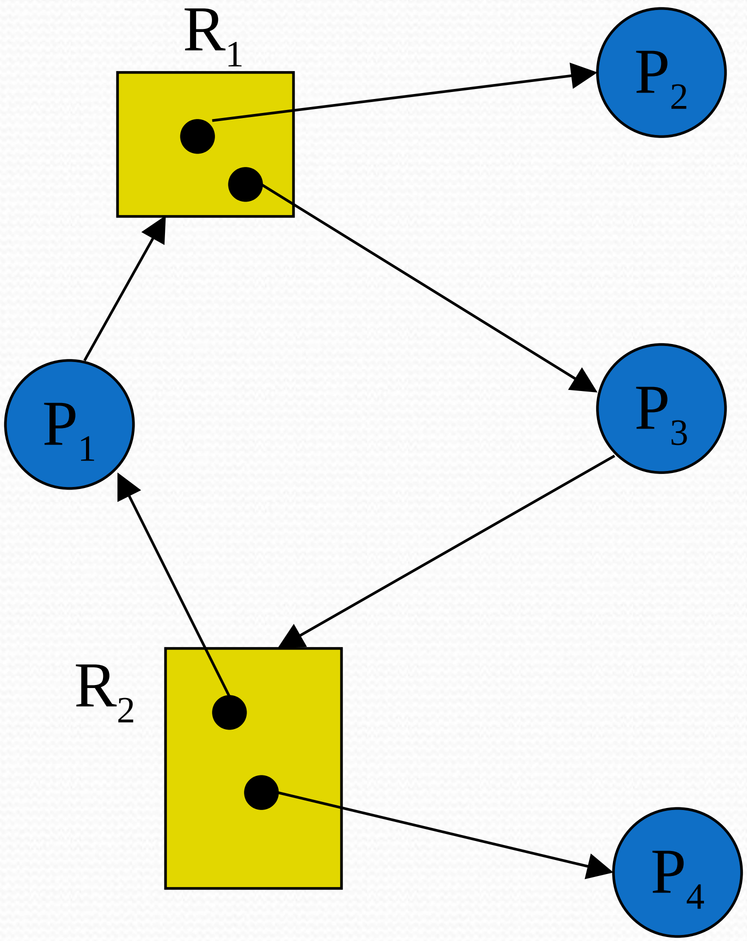
- If the graph has no cycles then there are no deadlock processes.

- If there is a cycle, then there *may* be a deadlock.

Graph with a Deadlock



Graph without a Deadlock



2. Deadling with Deadlocks

- Stop a deadlock ever occuring
  - *deadlock prevention*
    - disallow at least one of the necessary conditions
  - *deadlock avoidance*
    - see a deadlock coming and alter the process/resource allocation strategy

continued

- *Deadlock detection and recovery*

- Ignore the problem
  - done by most OSes, including UNIX
  - cheap solution
  - infrequent, manual reboots may be acceptable

3. Deadlock Prevention

- Eliminate one (or more) of:
  - mutual exclusion
  - hold and wait
  - no preemption (i.e. *have* preemption)
  - circular wait

3.1. Eliminate Mutual Exclusion

- Shared resources do not require mutual exclusion
  - e.g. read-only files

- But some resources cannot be shared (at the same time)
  - e.g. printers

3.2. Eliminate Hold & Wait

- One approach requires that each process be allocated *all* of its resources before it begins executing
  - eliminates the wait possibility

- Alternatively, only allow a process to request resources when it currently has none
  - eliminates the hold possibility

3.3. Eliminate “No Preemption”

- Make a process automatically release its current resources if it cannot obtain all the ones it wants
  - restart the process when it can obtain everything

- Alternatively, the desired resources can be preempted from other waiting processes

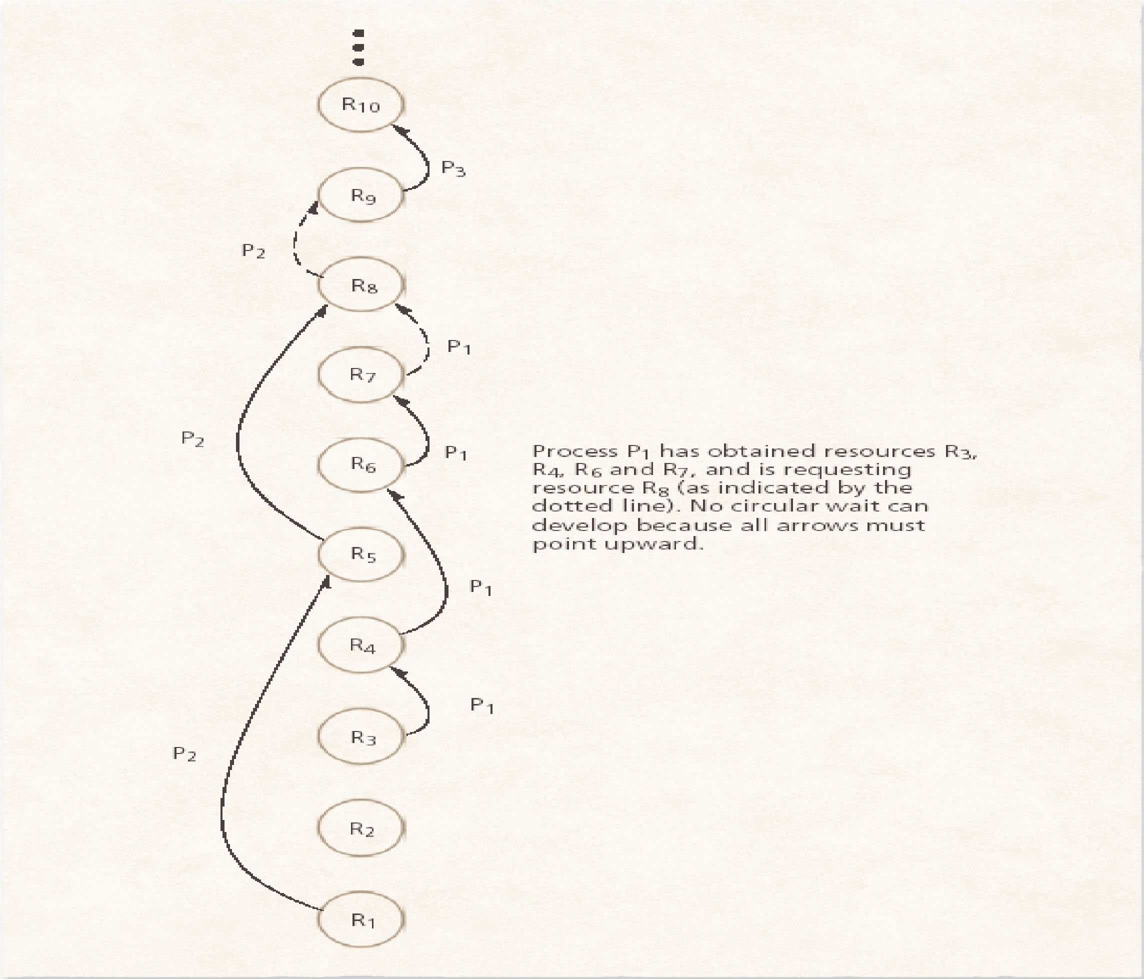
3.4. Eliminate Circular Wait

- Impose a total ordering on all the resource types, and force each process to request resources in increasing order.

- *Another approach*: require a process to release larger numbered resources when it obtains a smaller numbered resource.

3.4 Eliminate Circular Wait

Havender's linear ordering of resources to prevent deadlock



4. Deadlock Avoidance

- In deadlock avoidance, the necessary conditions are untouched.

- Instead, extra information about resources is used by the OS to do better forward planning of process/resource allocation
  - indirectly avoids circular wait

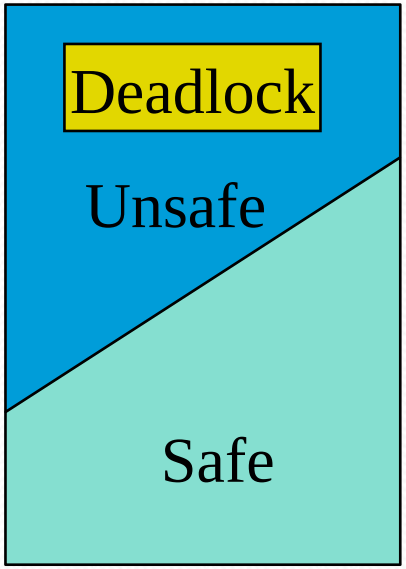
4.1. Safe States

- An OS is in a *safe state* if there is a *safe sequence* of process executions <P_1, P_2, …, P_n>.

- In a safe sequence, each P_i can satisfy its resource requests by using the currently available resources *and* (if necessary) the resources held by P_j ($j < i$)
  - only when P_j has finished

Safe State Implications

- A safe state cannot lead to deadlock.
- An unsafe state *may* lead to deadlock.
- Deadlock is avoided by always keeping the system in a safe state
  - this may reduce resource utilization



Example 1

- Max no. of resources: 12 tape drives
-

	Max needs	Current Allocation
P_0	10	5
P_1	4	2
P_2	9	2

- Currently, there are 3 free tape drives
- The OS is in a *safe* state, since <P_1, P_0, P_2> is a safe sequence.

Example 2

- Same as last slide, but P_2 now has 3 tape drives allocated currently.

	Max needs	Current Allocation
P_0	10	5
P_1	4	2
P_2	9	**3**

- The OS is in an *unsafe* state.

4.2. Using Resource Allocation Graphs

- Assume a resource type only has one instance.

- Add a *claim edge*:
  - $P_i \rightarrow R_j$
  - process P_i may request a resource R_j in the future
  - drawn as a dashed line

continued

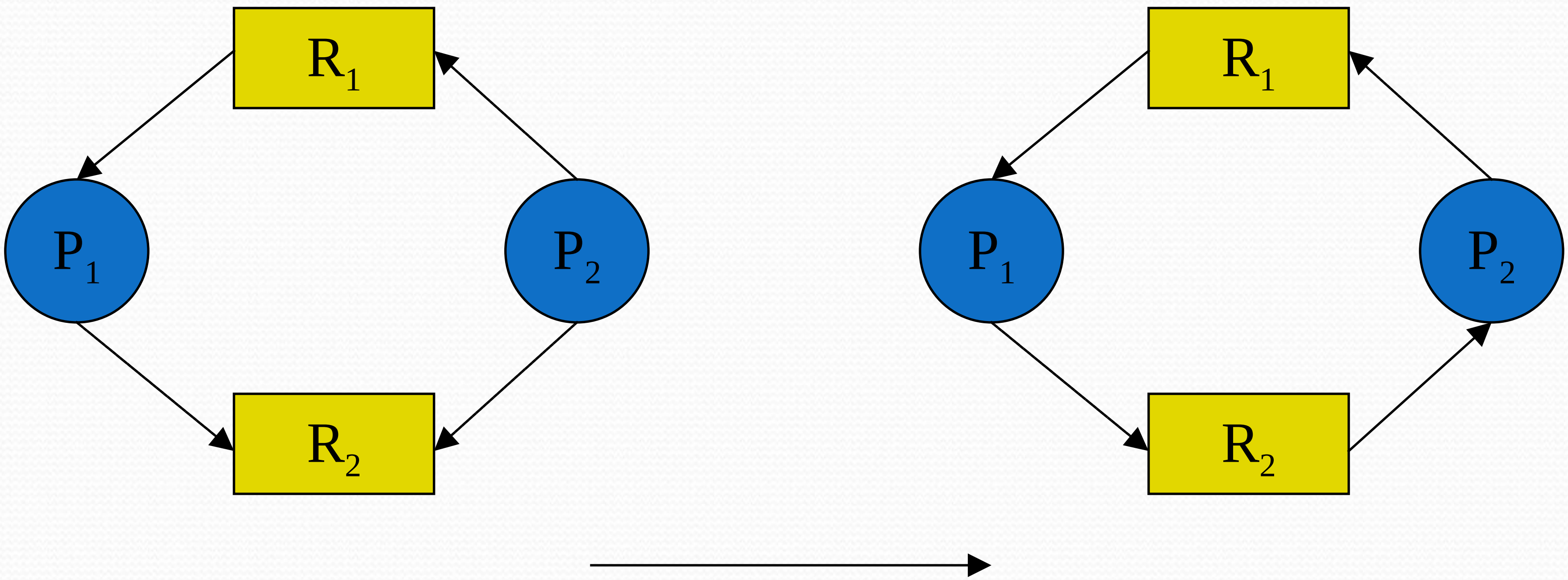
- When the resource is actually requested, the claim edge is changed to a request edge.

- When an assignment is released, the assignment edge is changed back to a claim edge.

continued

- All resources must be claimed before system start-up.
- An unsafe state is caused by a cycle in the resource allocation graph.

Example



R_2 allocation to P_2

creates an unsafe state

4.3. Banker's Algorithm

- Assume that:
  - a resource can have multiple instances
  - the OS has N processes, M resource types
- Initially, each process must declare the maximum no. of resources it will need.
- Calculate a safe sequence if possible.

Banker Data Structures

- `Available[M]`
  - no. of available resource instances for each resource type
  - e.g. `Available[j] == k` means K R_j's
- `Max[N][M]`
  - max demand of each process
  - e.g. `max[i][j] == k` means P_i wants k R_j's

continued

- `Work[M]`
  - no. of resource instances available for work (by all processes)
  - e.g. `Work[j] == k` means K R_j's are available
- `Finish[N]`
  - record of finished processes
  - e.g. P_i is finished if `Finish[i] == true`

continued

- `Allocation[N][M]`
  - no. of resource instances allocated to each process
  - e.g. `Allocation[i][j] == k`
    means P_i currently has k R_j's

- `Need[N][M]`
  - no. of resource instances still needed by each process
  - e.g. `Need[i][j] == k`
    means P_i still needs k R_j's
  - `Need[i][j] == Max[i][j] - Allocation[i][j]`

continued

- `Request[N][M]`
  - no. of resource instances currently requested by each process
  - e.g. `Request[i][j] == k`
    means P_i has requested k R_j's

Vectors

shorthand for referring to the 2D data structures

- `Allocation[i]`
  - resources currently allocated to P_i
- `Need[i]`
  - resources still needed by P_i
- `Request[i]`
  - resources currently requested by P_i

The Safety Algorithm

1. Vector Copy: *Work* := *Available*; *Finish* := *false*
2. Find i such that P_i hasn't finished but could:
   Finish[*i*] == *false*
   Need[*i*] <= *Work*
   If no suitable i, go to step 4.
3. Assume P_i completes:
   Work := *Work* + *Allocation*[*i*]
   Finish[*i*] := *true*
   go to step 2
4. If *for all i Finish*[*i*] == *true* then *Safe-State*

Safety Example

Resource Type	*Instances*
A	10
B	5
C	7

continued

	Allocation	*Max*	*Available*	*Need*
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- The OS is in a *safe state* since $<P_1, P_3, P_4, P_2, P_0>$ is a safe sequence.

Request Resource Algorithm

1 If (Need[*i*] < Request[i]) then *max-error*

2 While (Available < Request[i]) do *wait*

3 Construct a new state by:

$$Available = Available - Request[i]$$
$$Allocation[i] = Allocation[i] + Request\ [i]$$
$$Need[i] = Need[i] - Request\ [i]$$

4 If (*new state is not safe*) then
restore and *wait*

Request Example 1

- At some time, P_1 requests an additional 1 A instance and 2 C instances
  - i.e. `Request[1] == (1, 0, 2)`
- Does this lead to a safe state?
  - `Available >= Request[1]` so continue
  - generate new state and test for safety

	Allocation	*Max*	*Available*	*Need*
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**2 3 0**	7 4 3
P_1	**3 0 2**	3 2 2	**0 2 0**	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 3	

- The OS is in a *safe state* since $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ is a safe sequence.

Further Request Examples

- From this state, P_4 requests a further (3,3,0)
  - cannot be granted, since insufficient resources
- Alternatively, P_0 requests a further (0,2,0)
  - should not be granted since the resulting state is unsafe

Weaknesses in the Banker’s Algorithm

- Weaknesses in the Banker’s Algorithm
  - Requires there be a fixed number of resource to allocate
  - Requires the population of processes to be fixed
  - Requires the banker to grant all requests within “finite time”
  - Requires that clients repay all loans within “finite time”
  - Requires processes to state maximum needs in advance

5. Deadlock Detection

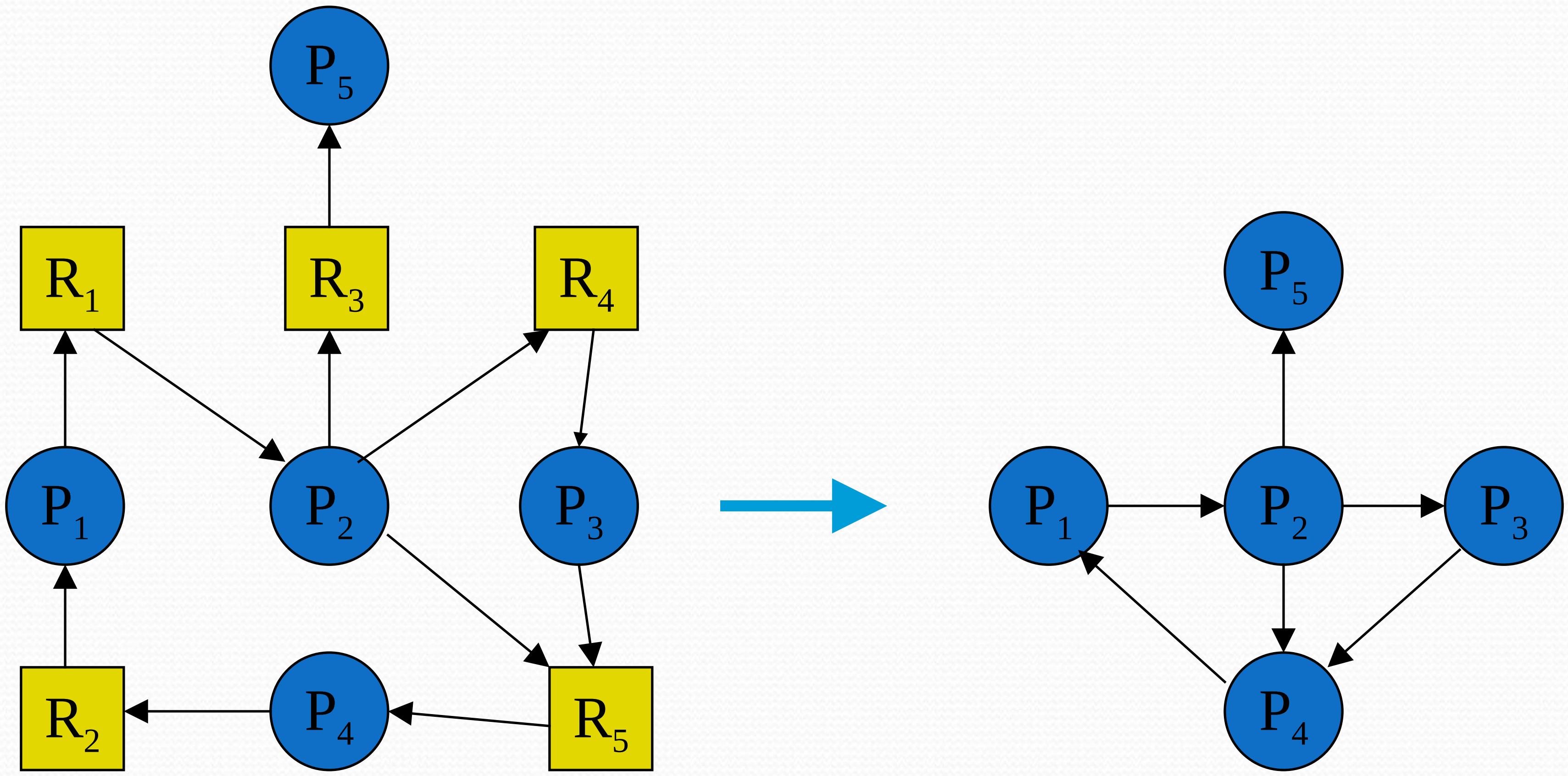
- If there are no prevention or avoidance mechanisms in place, then deadlock may occur.
- Deadlock detection should return enough information so the OS can recover.
- How often should the detection algorithm be executed?

5.1. Wait-for Graph

- Assume that each resource has only one instance.
- Create a wait-for graph by removing the resource types nodes from a resource allocation graph.
- Deadlock exists if and only if the wait-for graph contains a cycle.

Example

Fig. 7.7, p.225



5.2. Banker’s Algorithm Variation

- If a resource type can have multiple instances, then an algorithm very similar to the banker’s algorithm can be used.

- The algorithm investigates every possible allocation sequence for the processes that remain to be completed.

Detection Algorithm

1. Vector Copy: *Work* := *Available*; *Finish* := *false*
2. Find i such that P_i hasn't finished but could:
   Finish[*i*] == *false*
   Request[i] <= *Work*
   If no suitable i, go to step 4.
3. Assume P_i completes:
   Work := *Work* + *Allocation*[*i*]
   Finish[*i*] := *true*
   go to step 2
4. If *Finish*[*i*] == *false* then *Pi is deadlocked*

Example 1

Resource Type	*Instances*
A	7
B	2
C	6

continued

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- The OS is *not* in a deadlocked state since $<P_0, P_2, P_3, P_1, P_4>$ is a safe sequence.

Example 2

- Change P_2 to request 1 C instance

	Allocation			*Request*			*Available*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- The OS *is* deadlocked.

6. Deadlock Recovery

- Tell the operator
- System-based recovery:
  - abort one or more processes in the circular wait
  - preempt resources in one or more deadlocked processes

6.1. Process Termination

- Abort all deadlocked processes, or
- Abort one process at a time until the deadlocked cycle disappears
  - not always easy to abort a process
  - choice should be based on *minimum cost*

6.1. Process Termination

The term *minimum cost* depends on various factors:

- What is the priority of the process
- How long the process has computed and how much longer the process will compute before completing its designated task
- How many and what types of resources the process has used.

6.1. Process Termination

The term *minimum cost* depends on various factors:

- How many more resources the process needs in order to complete
- How many processes will need to be terminated.
- Whether the process is interactive or batch

6.2. Resource Preemption

- Issues:
  - how to select a resource
    (e.g. by using minimum cost)
  - how to rollback the process which has just lost its resources
  - avoiding process starvation