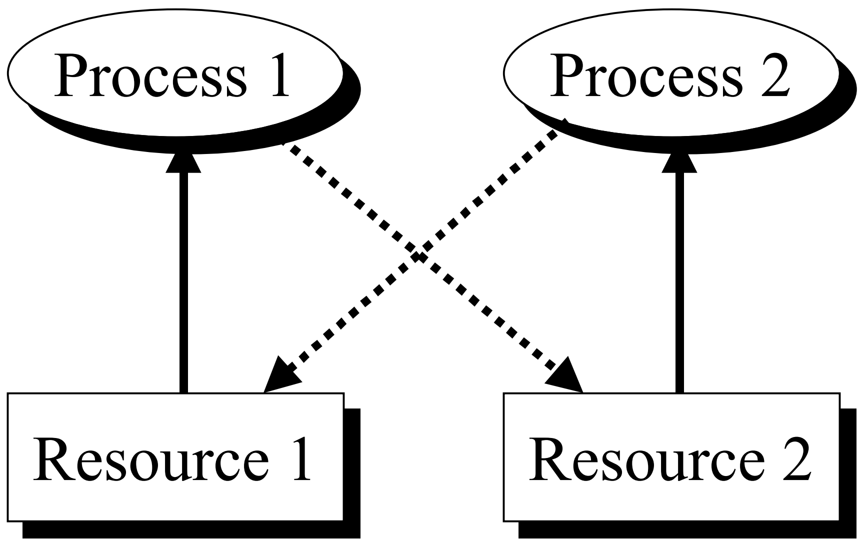
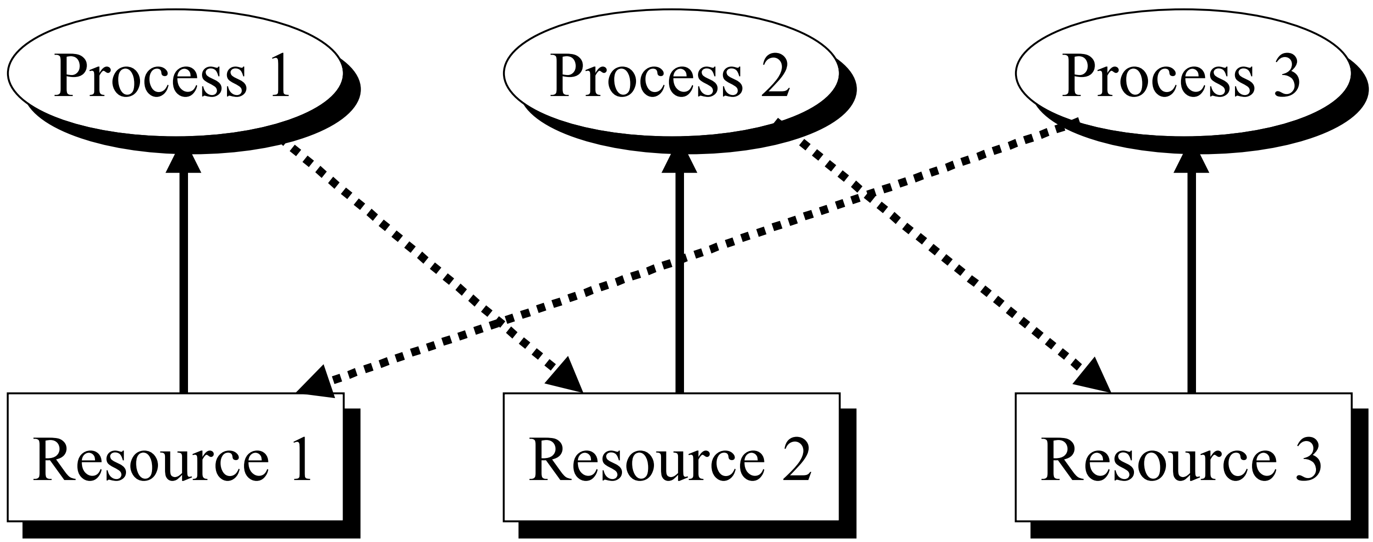
Deadlock

Example



Example


Process 1
Process 2
Process 3
Resource 1
Resource 2
Resource 3

Addressing Deadlock

- Prevention: Design the system so that deadlock is impossible
- Avoidance: Construct a model of system states, then choose a strategy that will not allow the system to go to a deadlock state
- Detection & Recovery: Check for deadlock (periodically or sporadically), then recover
- Manual intervention: Have the operator reboot the machine if it seems too slow

A Model

- $P = \{p_1, p_2, \ldots, p_n\}$ be a set of processes
- $R = \{R_1, R_2, \ldots, R_m\}$ be a set of resources
- c_j = number of units of R_j in the system
- $S = \{S_0, S_1, \ldots\}$ be a set of states representing the assignment of R_j to p_i
  - State changes when processes take action
  - This allows us to identify a deadlock situation in the operating system

State Transitions

- The system changes state because of the action of some process, p_i
- There are three pertinent actions:
  - Request ("r_i"): request one or more units of a resource
  - Allocation ("a_i"): All outstanding requests from a process for a given resource are satisfied
  - Deallocation ("d_i"): The process releases units of a resource

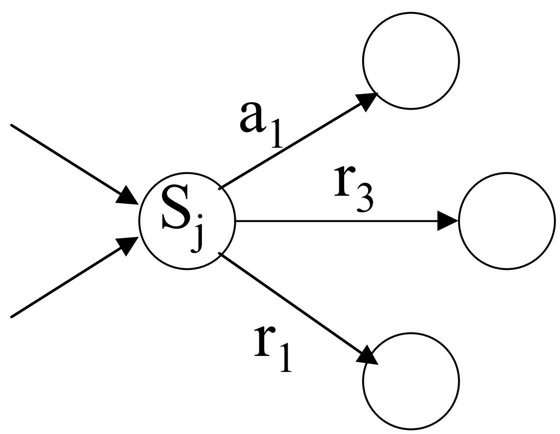
$$S_j \xrightarrow{x_i} S_k$$

Properties of States

- Want to define deadlock in terms of patterns of transitions
- Define: p_i is ***blocked*** in S_j if p_i cannot cause a transition out of S_j

Properties of States

- Want to define deadlock in terms of patterns of transitions
- Define: p_i is *blocked* in S_j if p_i cannot cause a transition out of S_j



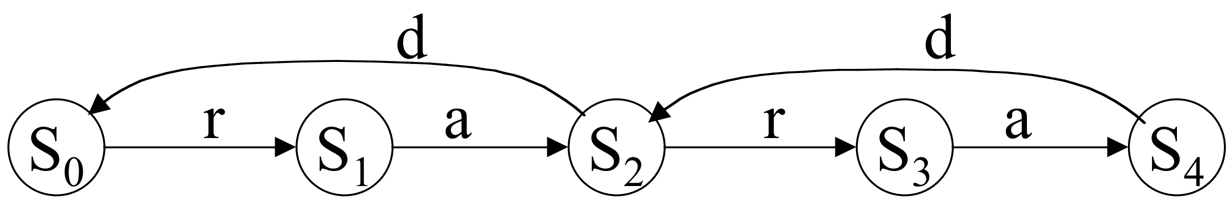
p_2 is blocked in S_j

Properties of States (cont)

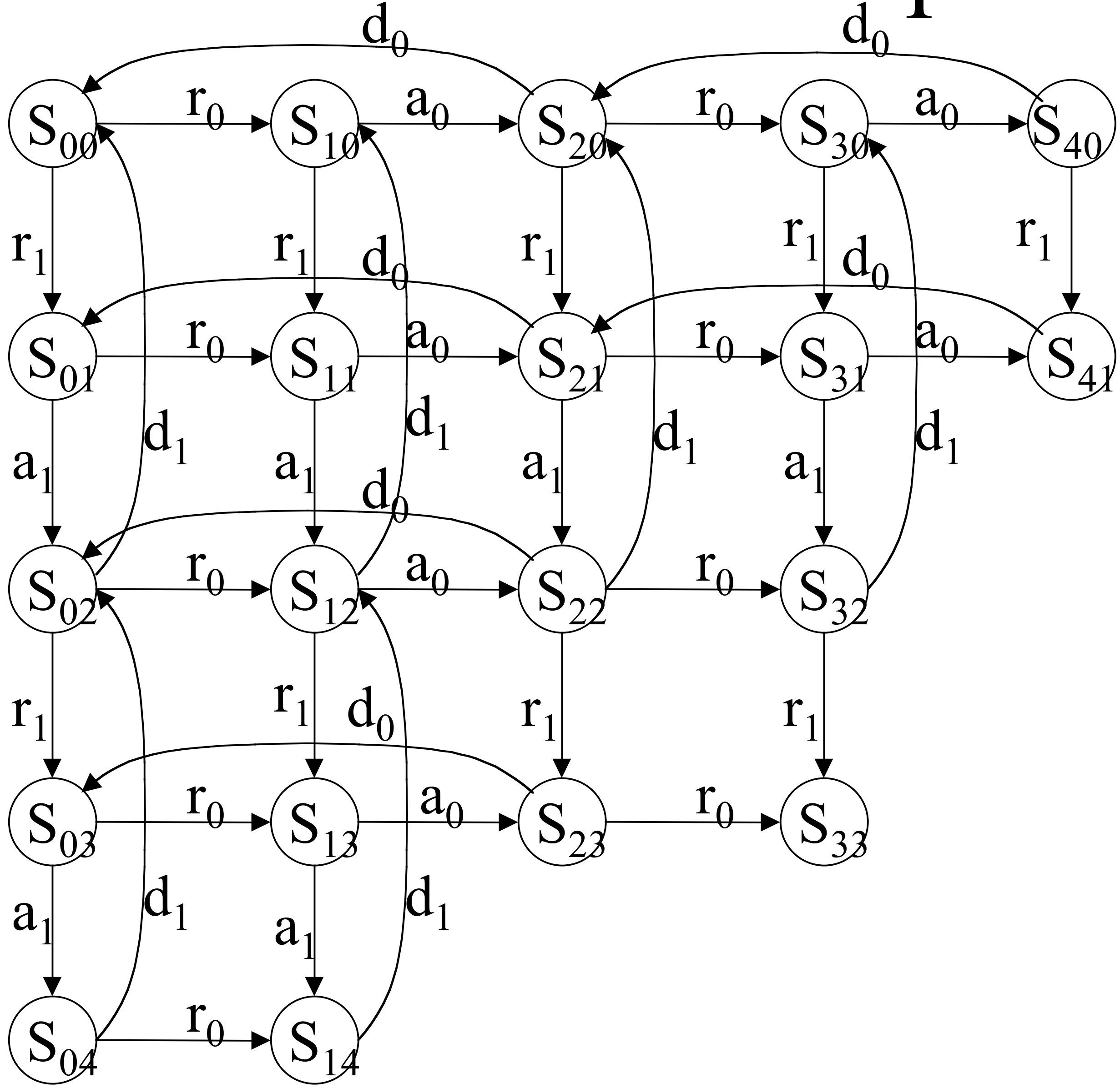
- If p_i is blocked in S_j, and will also be blocked in every S_k reachable from S_j, then p_i is deadlocked
- S_j is called a *deadlock state*

Example

- One process, two units of one resource
- Can request one unit at a time



Extension of Example



Prevention

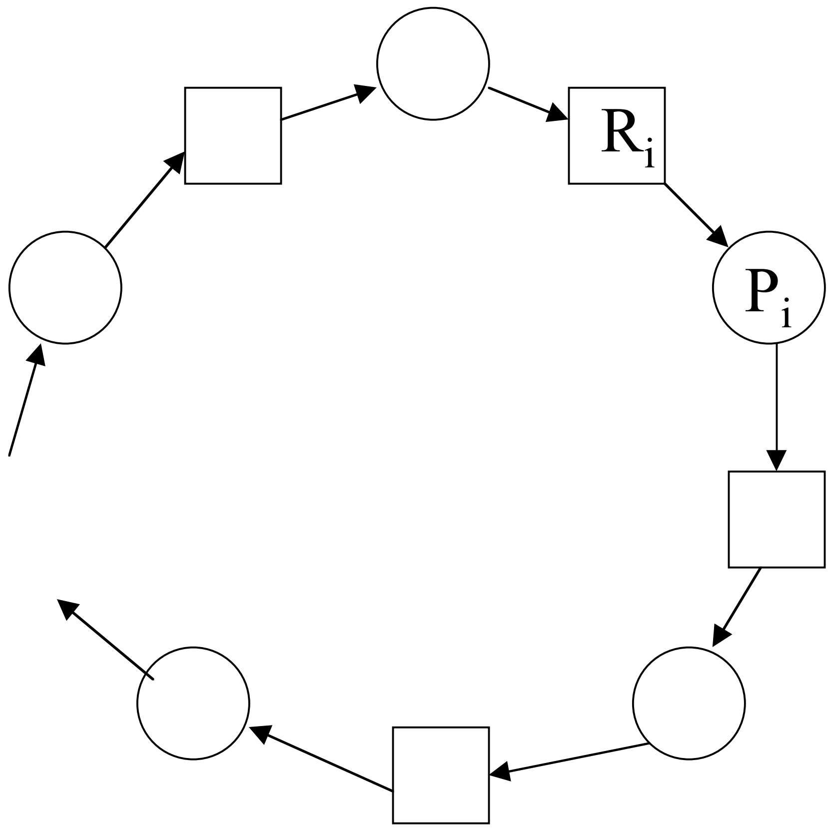
- Necessary conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - Circular waiting
  - No preemption
- Ensure that at least one of the necessary conditions is false at all times
  - Mutual exclusion must hold at all times

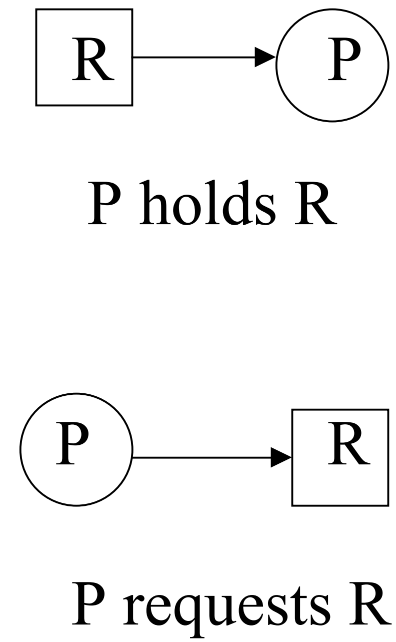
Hold and Wait

- Need to be sure a process does not hold one resource while requesting another
- Approach 1: Force a process to request all resources it needs at one time
- Approach 2: If a process needs to acquire a new resource, it must first release all resources it holds, then reacquire all it needs
- What does this say about state transition diagrams?

Circular Wait

- Have a situation in which there are K processes holding units of K resources





Circular Wait (cont)

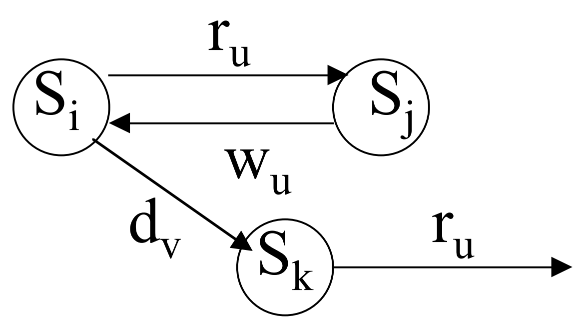
- There is a cycle in the graph of processes and resources
- Choose a resource request strategy by which no cycle will be introduced
- Total order on all resources, then can only ask for R_j if $R_i < R_j$ for all R_i the process is currently holding

Circular Wait (cont)

- There is a cycle in the graph of processes and resources
- Choose a resource request strategy by which no cycle will be introduced
- Total order on all resources, then can only ask for R_j if $R_i < R_j$ for all R_i the process is currently holding
- Here is how we saw the easy solution for the dining philosophers

Allowing Preemption

- Allow a process to time-out on a blocked request -- withdrawing the request if it fails



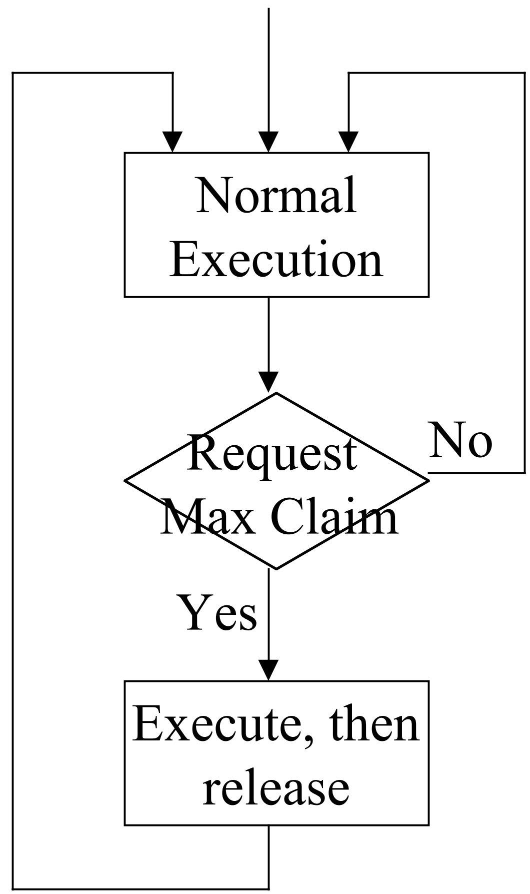
Avoidance

- Construct a model of system states, then choose a strategy that will guarantees that the system will not go to a deadlock state
- Requires extra information -- the *maximum claim* for each process
- Allows resource manager to see the worst case that could happen, then to allow transitions based on that knowledge

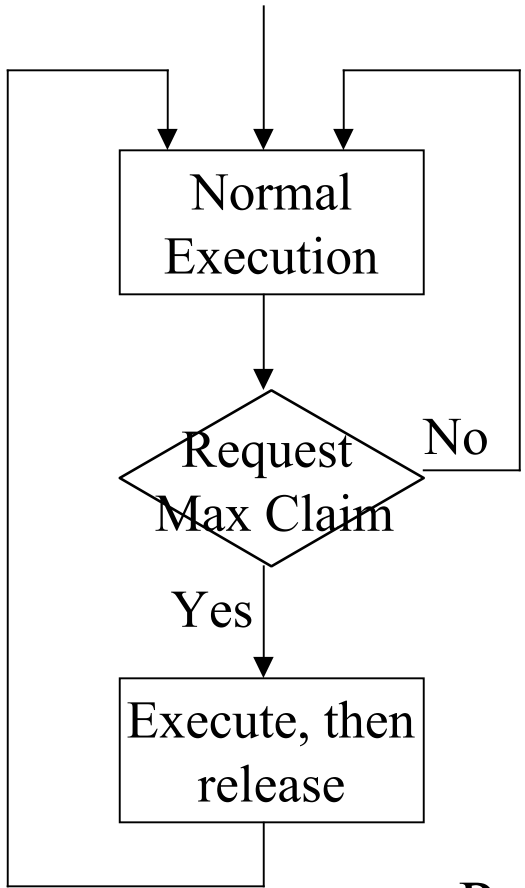
Safe vs Unsafe States

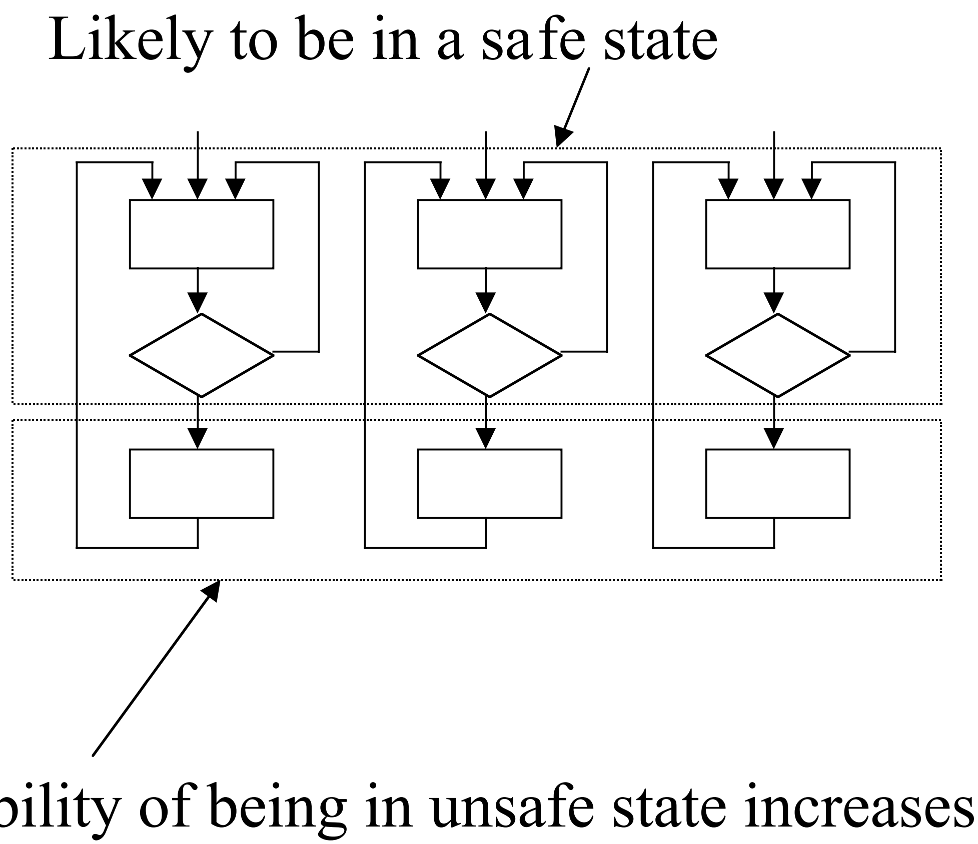
- *Safe state*: one in which there is guaranteed to be a sequence of transitions that leads back to the initial state
  - Even if all exercise their maximum claim, there is an allocation strategy by which all claims can be met
- *Unsafe state*: one in which the system cannot guarantee there is such a sequence
  - Unsafe state *can* lead to a deadlock state if too many processes exercise their maximum claim at once

More on Safe & Unsafe States

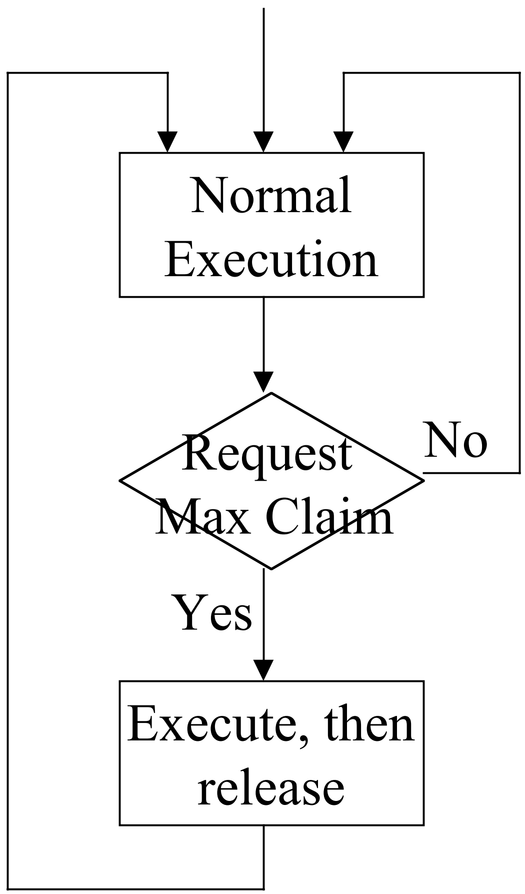


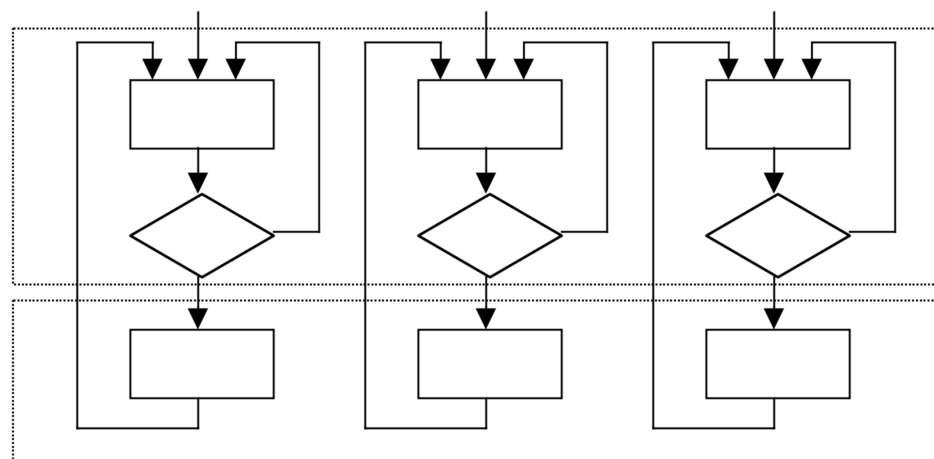
More on Safe & Unsafe States





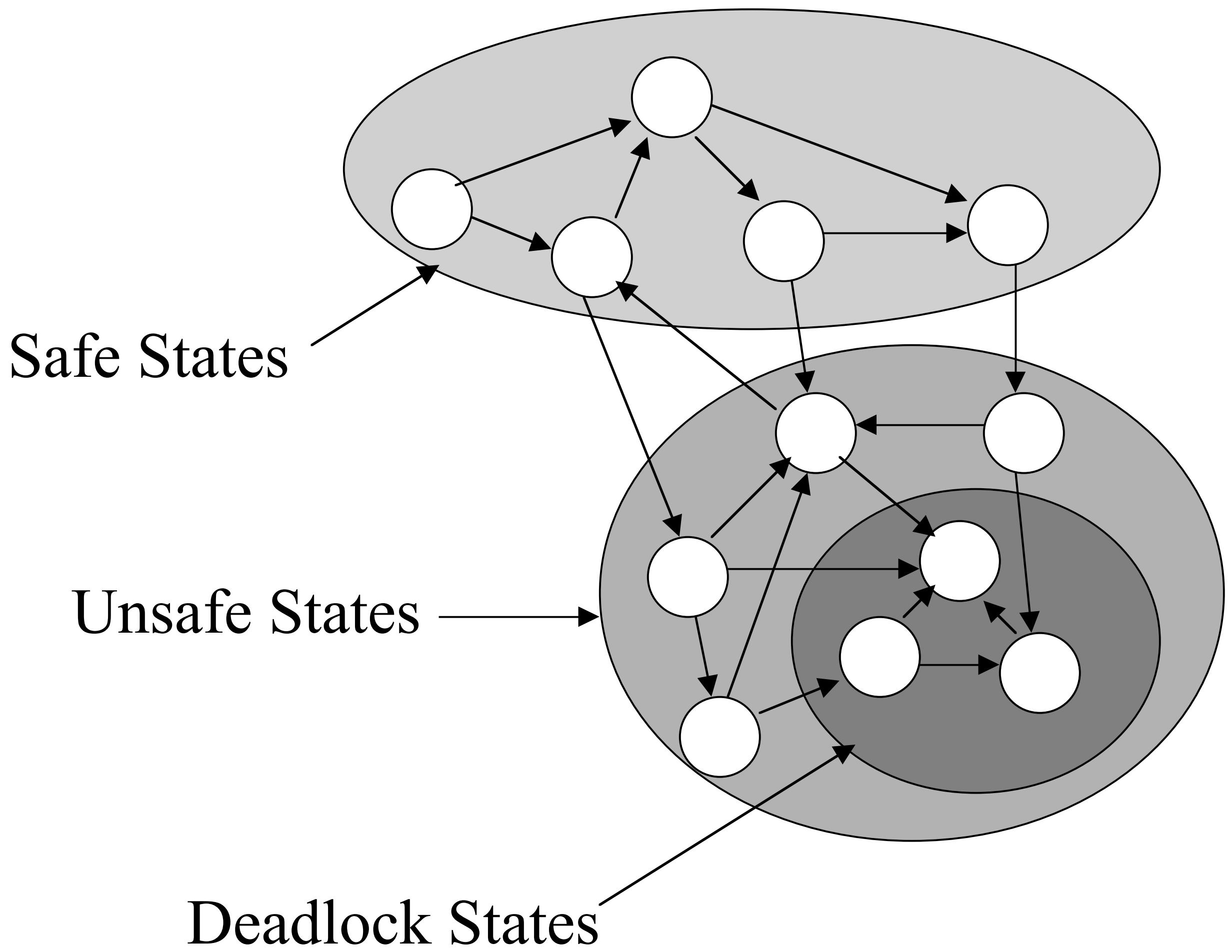
More on Safe & Unsafe States





- Suppose all processes take "yes" branch
- Avoidance strategy is to allow this to happen, yet still be safe

More on Safe & Unsafe States



Banker's Algorithm

- Let maxc[i, j] be the maximum claim for R_j by p_i
- Let alloc[i, j] be the number of units of R_j held by p_i
- Can always compute
  - $avail[j] = c_j - \Sigma_{0 \leq i < n} alloc[i,j]$
  - Then number of available units of R_j
- Should be able to determine if the state is safe or not using this info

Banker's Algorithm

- Copy the alloc[i,j] table to alloc'[i,j]
- Given C, maxc and alloc', compute avail vector
- Find p_i: maxc[i,j] - alloc'[i,j] $\leq$ avail[j] for $0 \leq j < m$ and $0 \leq i < n$.
  - If no such p_i exists, the state is unsafe
  - If alloc'[i,j] is 0 for all i and j, the state is safe
- Set alloc'[i,j] to 0; deallocate all resources held by p_i; go to Step 2

Example

$C = <8, 5, 9, 7>$

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0

Example

C = <8, 5, 9, 7>

- Compute total allocated

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	7	3	7	5

Example

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	7	3	7	5

Example

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	7	3	7	5

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

- Can anyone's maxc be met?

maxc[2,0]-alloc'[2,0] = 5-4 = 1≤1 = avail[0]
maxc[2,1]-alloc'[2,1] = 1-0 = 1≤2 = avail[1]
maxc[2,2]-alloc'[2,2] = 0-0 = 0≤2 = avail[2]
maxc[2,3]-alloc'[2,3] = 5-3 = 2≤2 = avail[3]

Example

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	7	3	7	5

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

- Can anyone's maxc be met?

maxc[2,0]-alloc'[2,0] = 5-4 = 1≤1 = avail[0]
maxc[2,1]-alloc'[2,1] = 1-0 = 1≤2 = avail[1]
maxc[2,2]-alloc'[2,2] = 0-0 = 0≤2 = avail[2]
maxc[2,3]-alloc'[2,3] = 5-3 = 2≤2 = avail[3]

- P_2 can exercise max claim

avail[0] = avail[0]+alloc'[2,0] = 1+4 = 5
avail[1] = avail[1]+alloc'[2,1] = 2+0 = 2
avail[2] = avail[2]+alloc'[2,2] = 2+0 = 2
avail[3] = avail[3]+alloc'[2,3] = 2+3 = 5

Example

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	1	0	3	0
Sum	3	3	7	2

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-3, 5-3, 9-7, 7-2>
= <5, 2, 2, 5>

- Can anyone's maxc be met?

maxc[4,0]-alloc'[4,0] = 5-1 = 4≤5 = avail[0]
maxc[4,1]-alloc'[4,1] = 0-0 = 0≤2 = avail[1]
maxc[4,2]-alloc'[4,2] = 3-3 = 0≤2 = avail[2]
maxc[4,3]-alloc'[4,3] = 3-0 = 3≤5 = avail[3]

Example

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	1	0	3	0
Sum	3	3	7	2

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-7, 5-3, 9-7, 7-5>
= <5, 2, 2, 5>

- Can anyone's maxc be met?

maxc[4,0]-alloc'[4,0] = 5-1 = 4≤5 = avail[0]
maxc[4,1]-alloc'[4,1] = 0-0 = 0≤2 = avail[1]
maxc[4,2]-alloc'[4,2] = 3-3 = 0≤2 = avail[2]
maxc[4,3]-alloc'[4,3] = 3-0 = 3≤5 = avail[3]

- P_4 can exercise max claim

avail[0] = avail[0]+alloc'[4,0] = 5+1 = 6
avail[1] = avail[1]+alloc'[4,1] = 2+0 = 2
avail[2] = avail[2]+alloc'[4,2] = 2+3 = 5
avail[3] = avail[3]+alloc'[4,3] = 5+0 = 5

Example

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	0	0	0	0
Sum	2	1	4	2

C = <8, 5, 9, 7>

- Compute total allocated
- Determine available units

avail = <8-7, 5-3, 9-7, 7-5>
= <6, 2, 5, 5>

- Can anyone's maxc be met?

(Yes, any of them can)

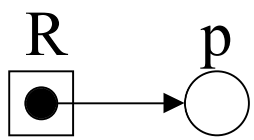
Detection & Recovery

- Check for deadlock (periodically or sporadically), then recover
- Can be far more aggressive with allocation
- No maximum claim, no safe/unsafe states
- Differentiate between
  - Serially reusable resources: A unit must be allocated before being released
  - Consumable resources: Never release acquired resources; resource count is number currently available

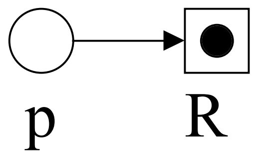
Reusable Resource Graphs (RRGs)

- Micro model to describe a single state
- Nodes = $\{p_0, p_1, \ldots, p_n\} \cup \{R_1, R_2, \ldots, R_m\}$
- Edges connect p_i to R_j, or R_j to p_i
  - (p_i, R_j) is a request edge for one unit of R_j
  - (R_j, p_i) is an assignment edge of one unit of R_j
- For each R_j there is a count, c_j of units Rj
- Number of units of R_j allocated to p_i plus the number requested by p_i cannot exceed c_j

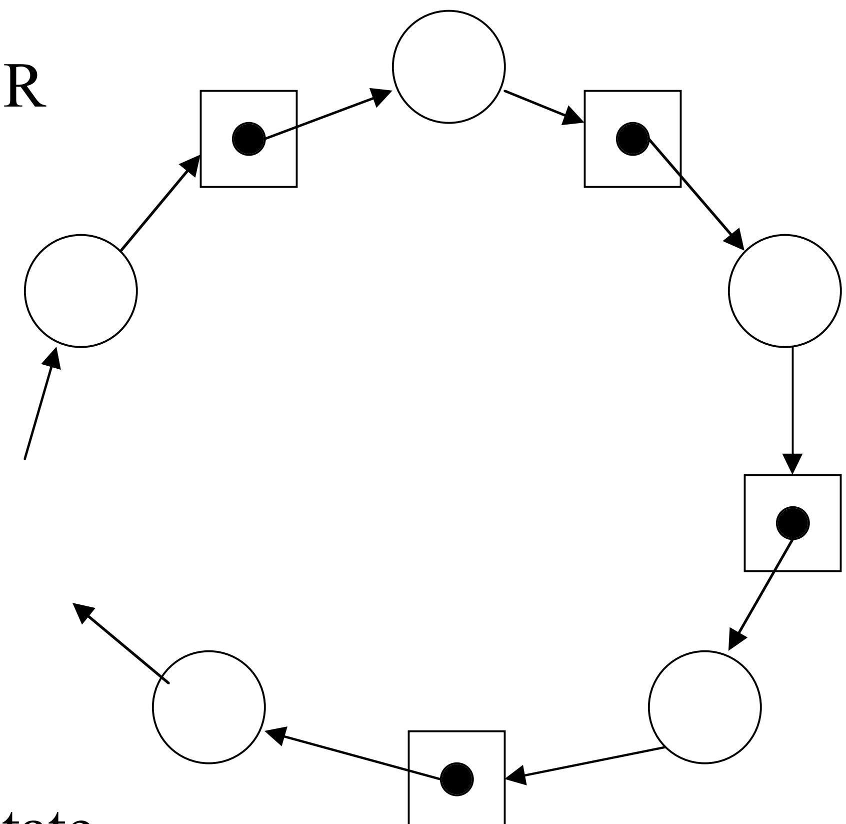
Example



P holds one unit of R

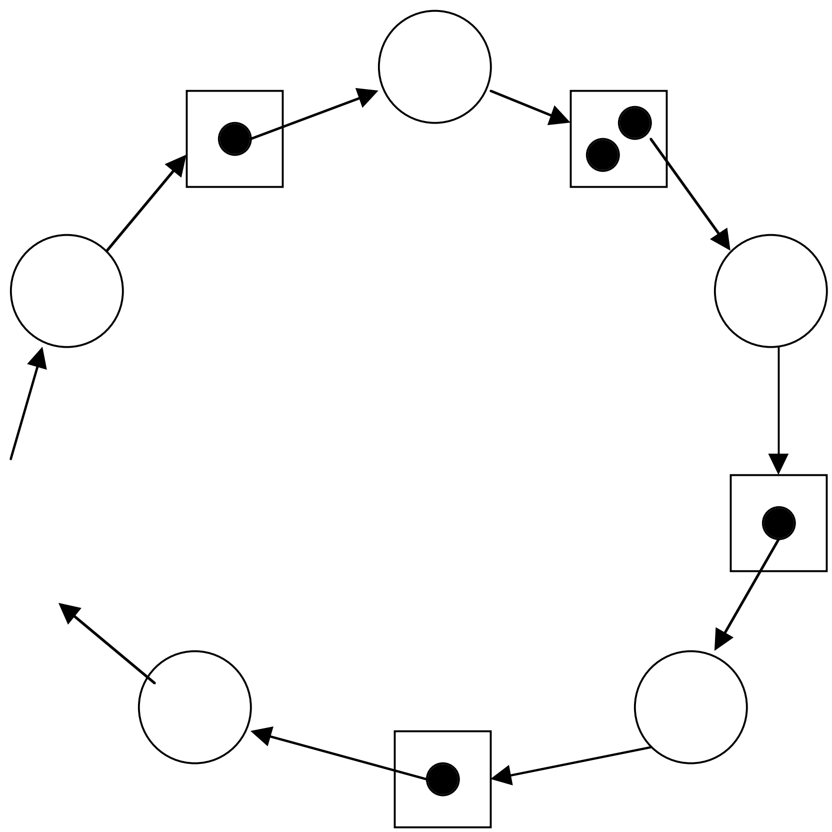


P requests one unit of R

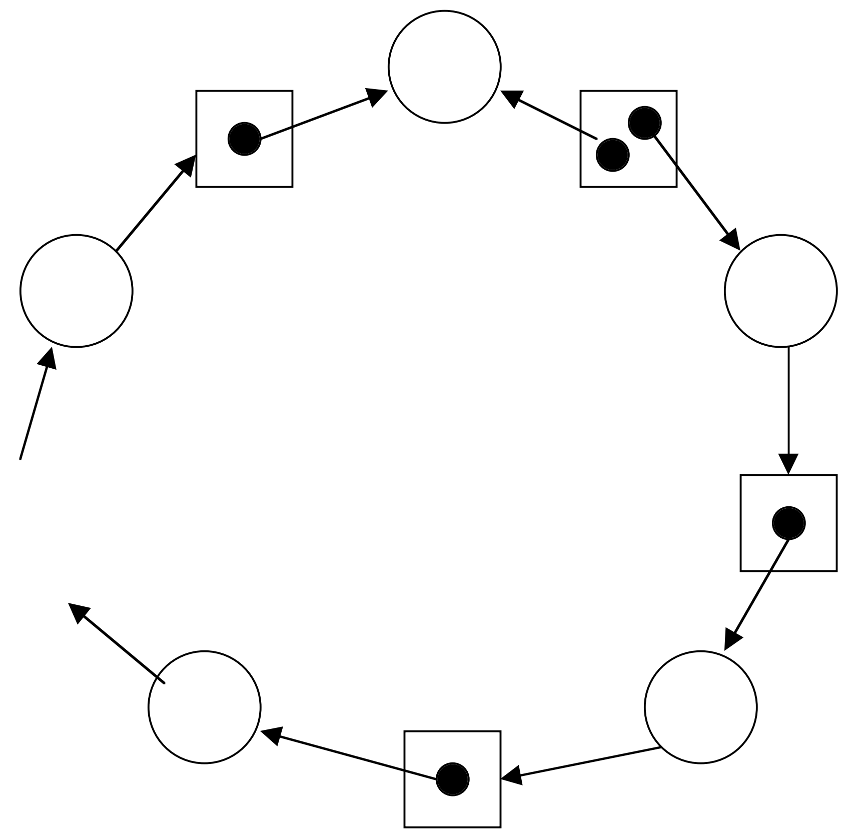


A Deadlock State

Example



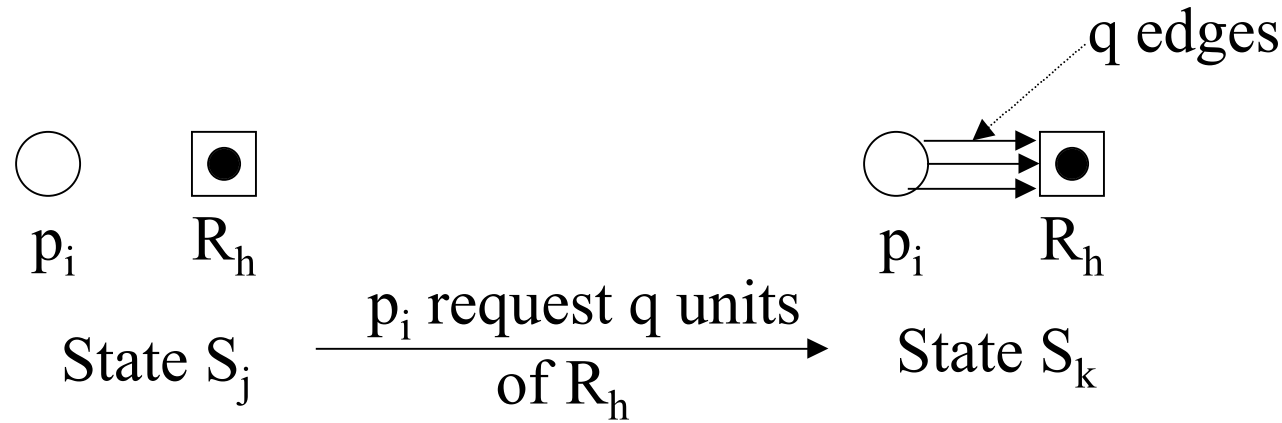
Not a Deadlock State



No Cycle in the Graph

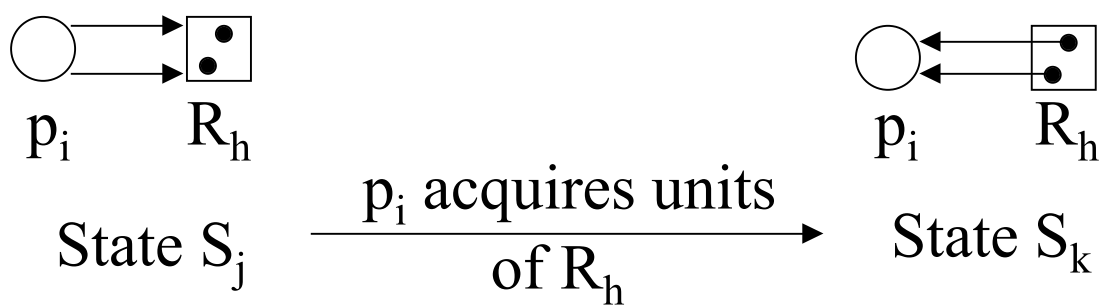
State Transitions due to Request

- In S_j, p_i is allowed to request $q \le c_h$ units of R_h, provided p_i has no outstanding requests.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from S_j by adding q request edges from p_i to R_h



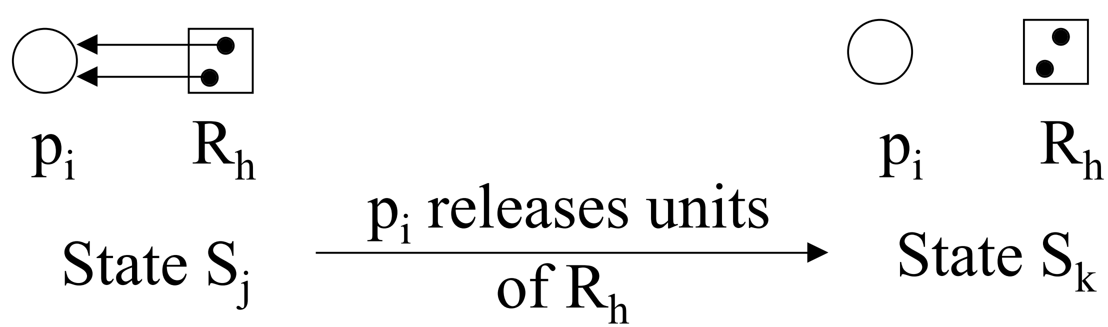
State Transition for Acquire

- In S_j, p_i is allowed to acquire units of R_h, iff there is (p_i, R_h) in the graph, and all can be satisfied.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from S_j by changing each request edge to an assignment edge.

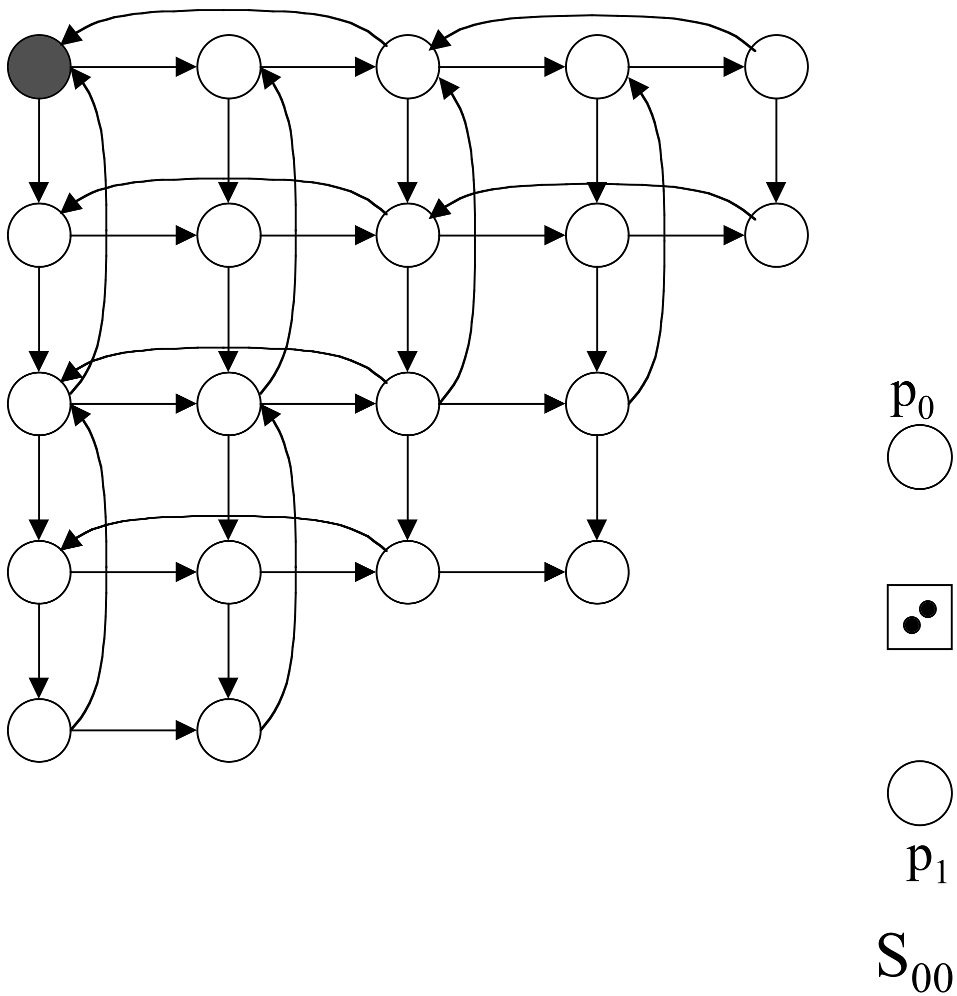


State Transition for Release

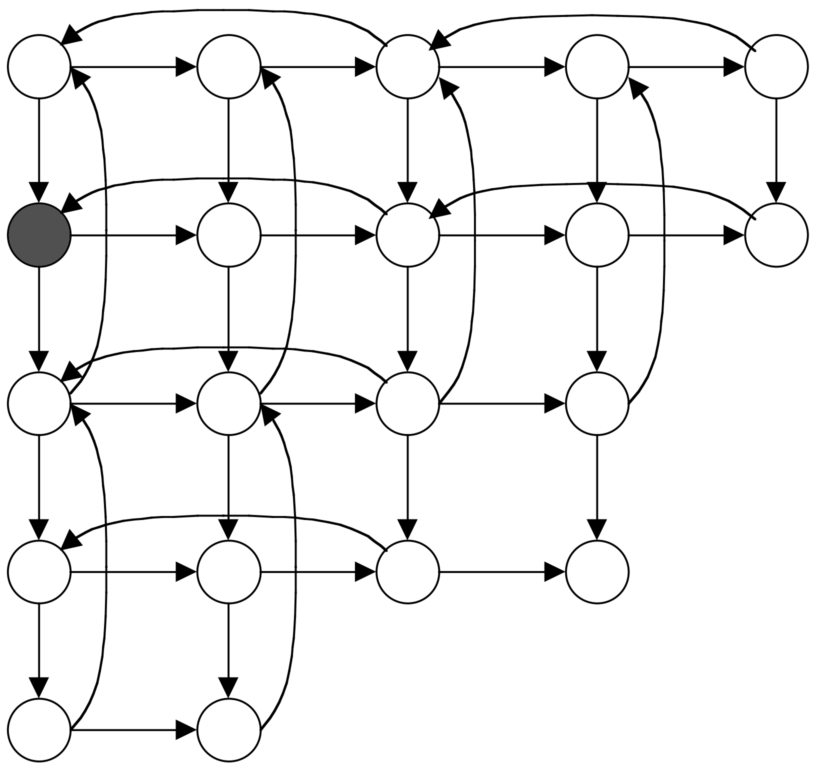
- In S_j, p_i is allowed to release units of R_h, iff there is (R_h, p_i) in the graph, and there is no request edge from p_i.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from S_j by deleting all assignment edges.

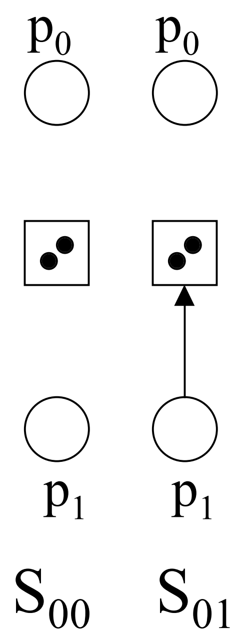


Example

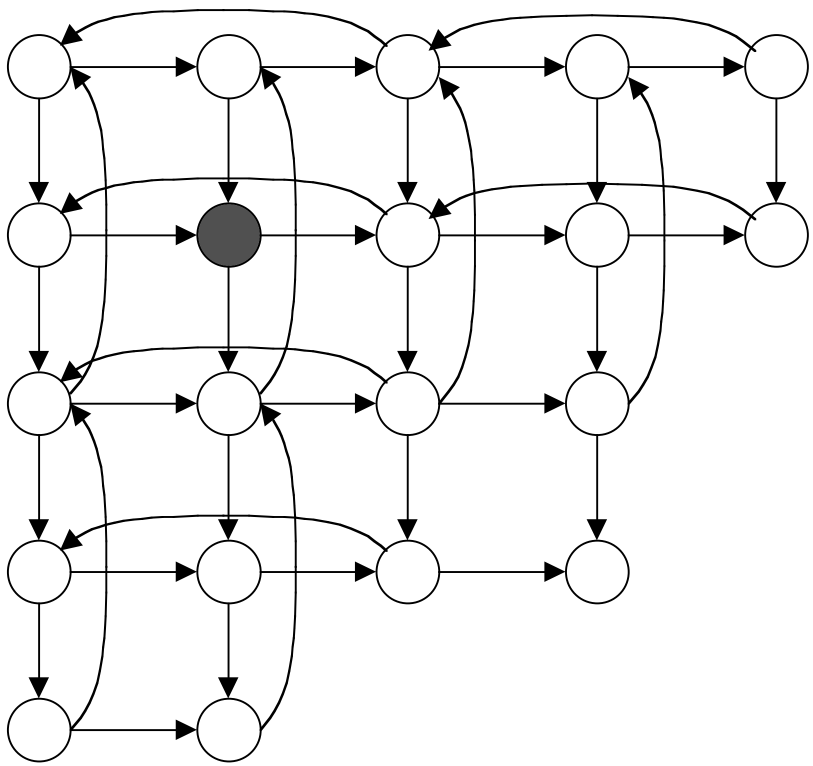


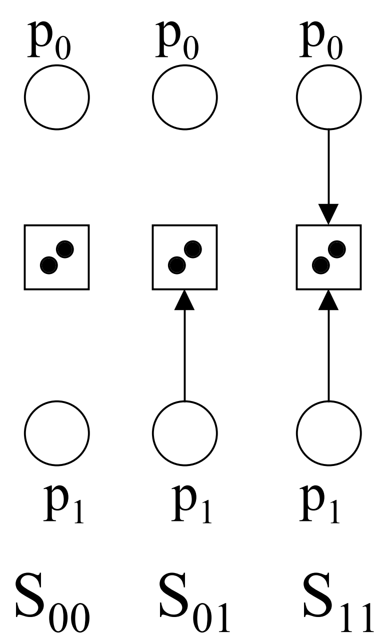
Example



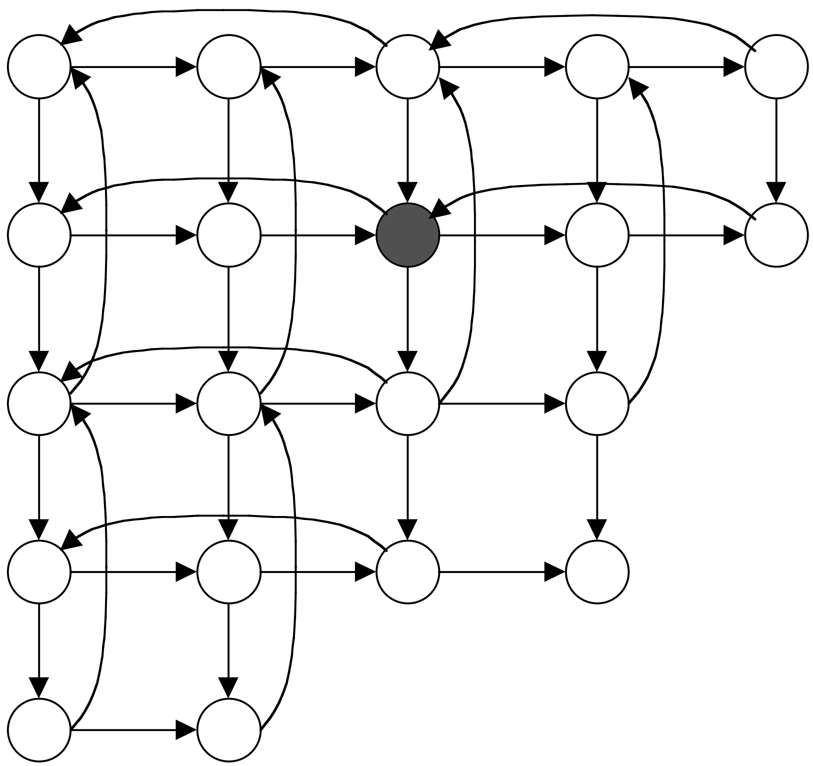


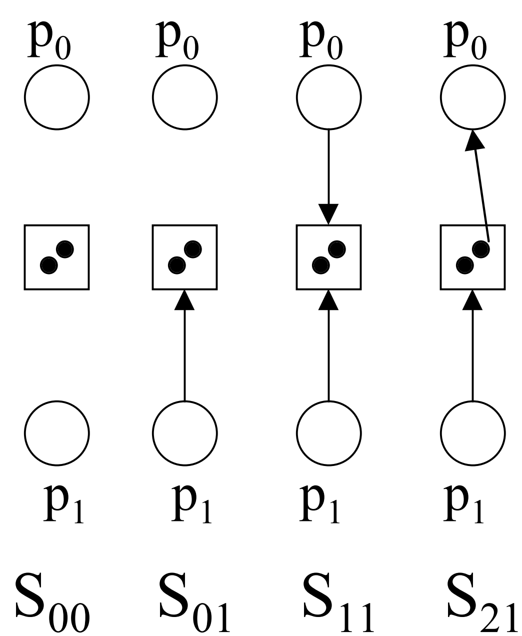
Example



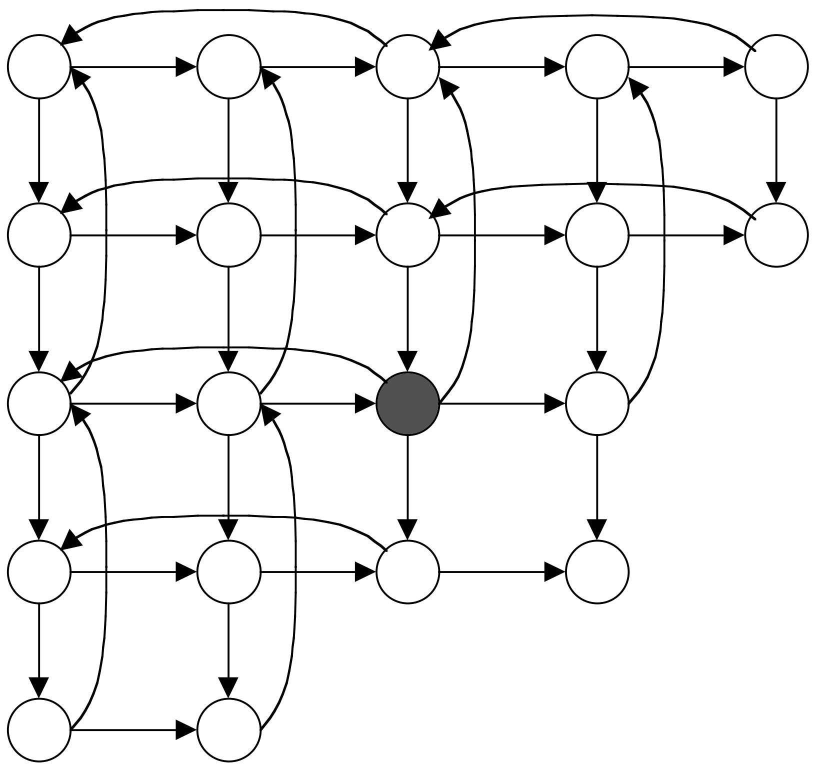


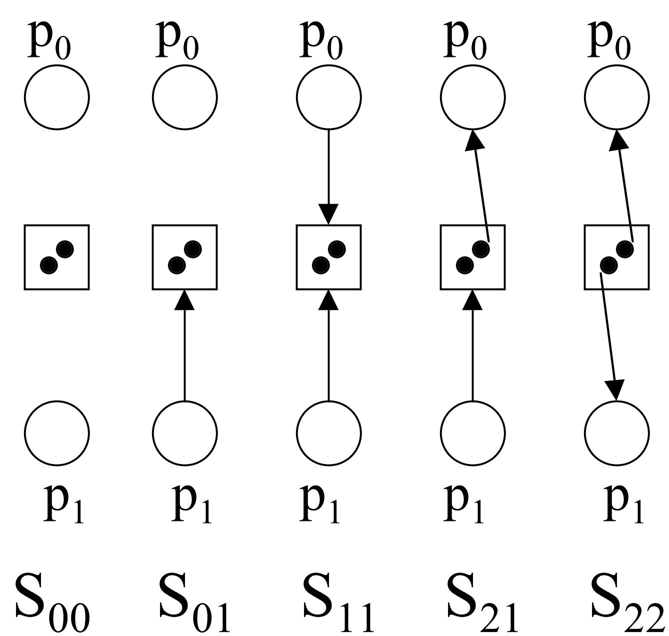
Example



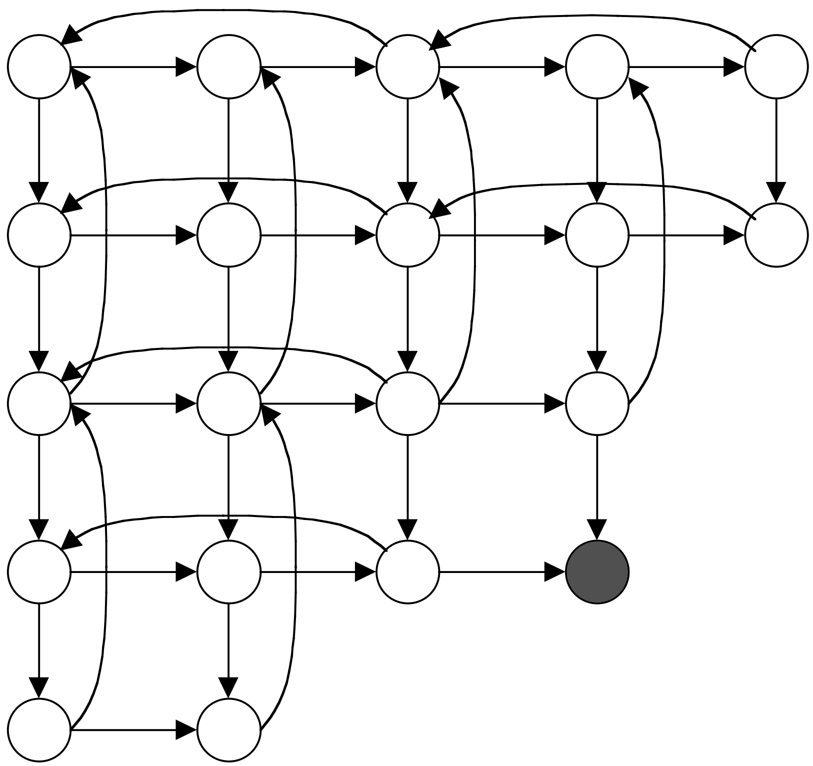


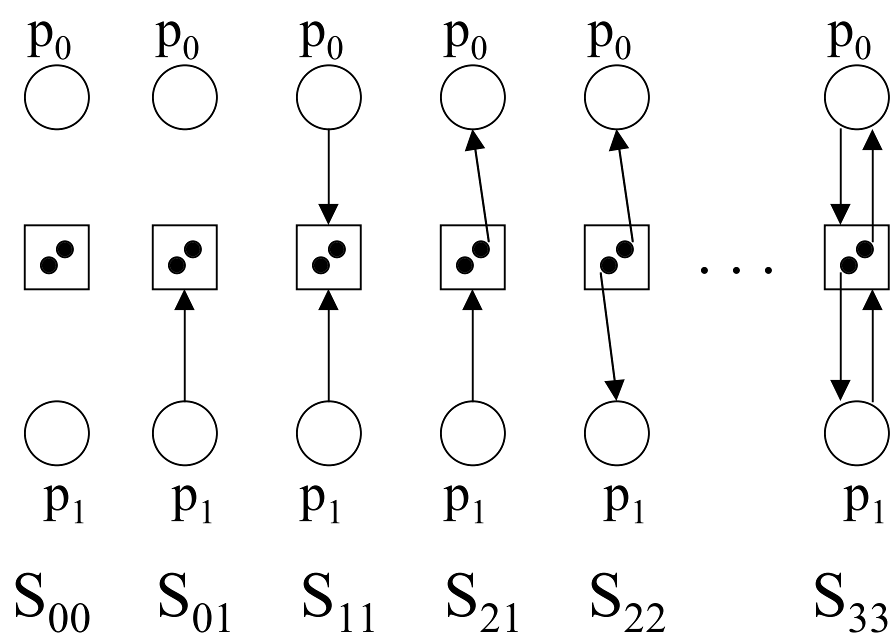
Example





Example



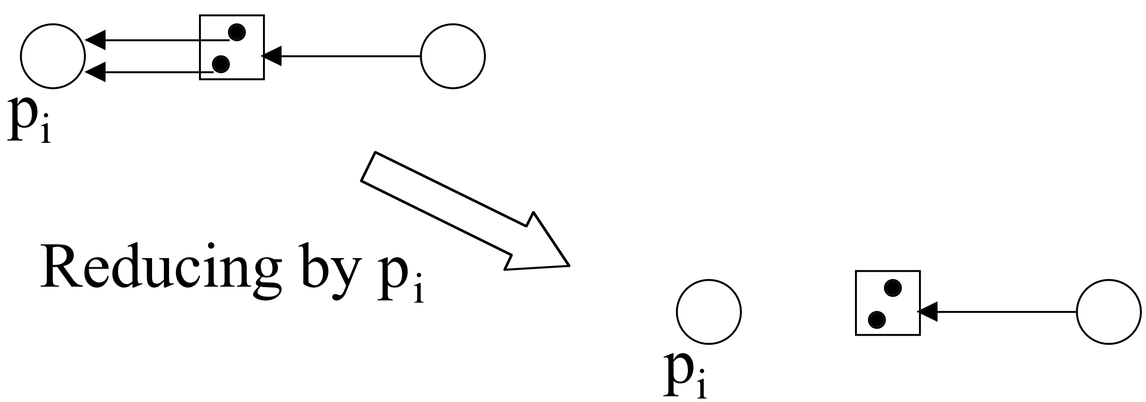


Graph Reduction

- Deadlock state if there is no sequence of transitions unblocking every process
- A RRG represents a state; can analyze the RRG to determine if there is a sequence
- A graph reduction represents the (optimal) action of an unblocked process. Can reduce by p_i if
  - p_i is not blocked
  - p_i has no request edges, and there are (R_j, p_i) in the RRG

Graph Reduction (cont)

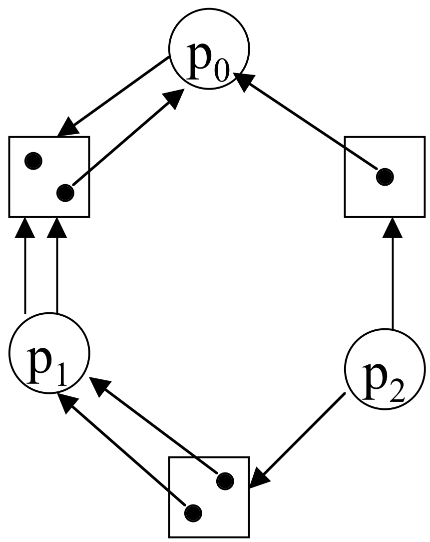
- Transforms RRG to another RRG with all assignment edges into p_i removed
- Represents p_i releasing the resources it holds



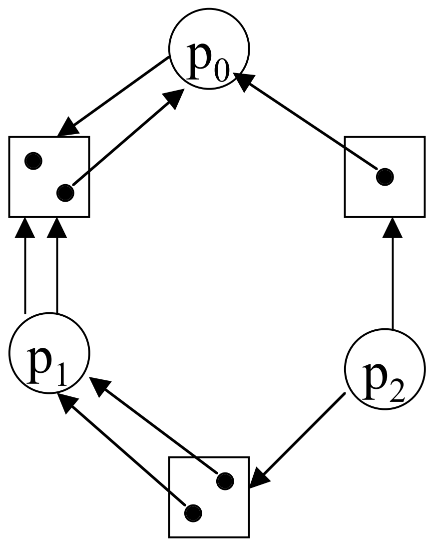
Graph Reduction (cont)

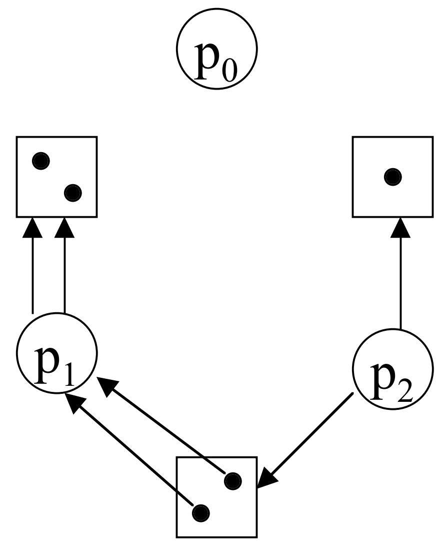
- A RRG is completely reducible if there a sequence of reductions that leads to a RRG with no edges
- ***A state is a deadlock state if and only if the RRG is not completely reducible***.

Example RRG

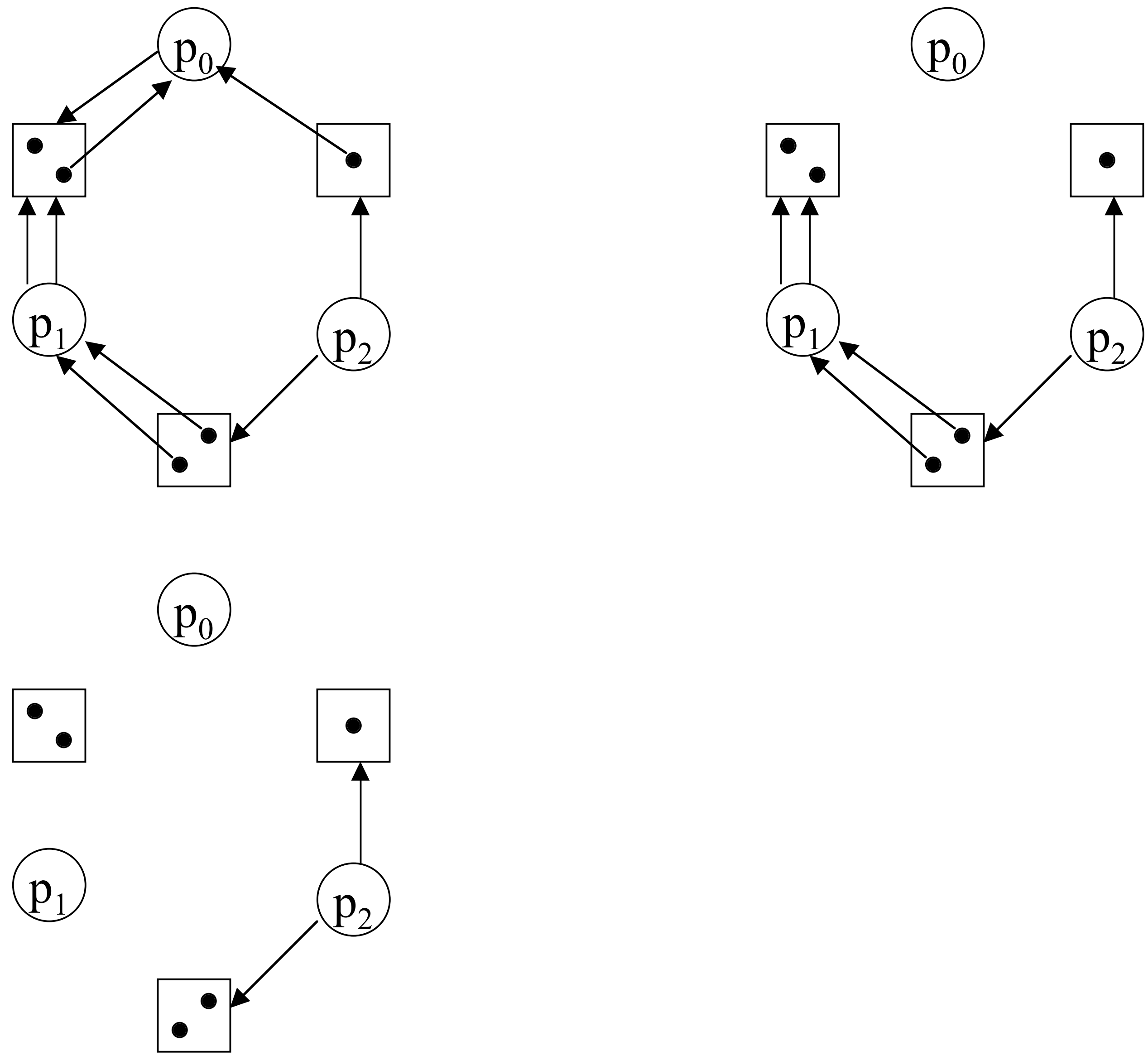


Example RRG

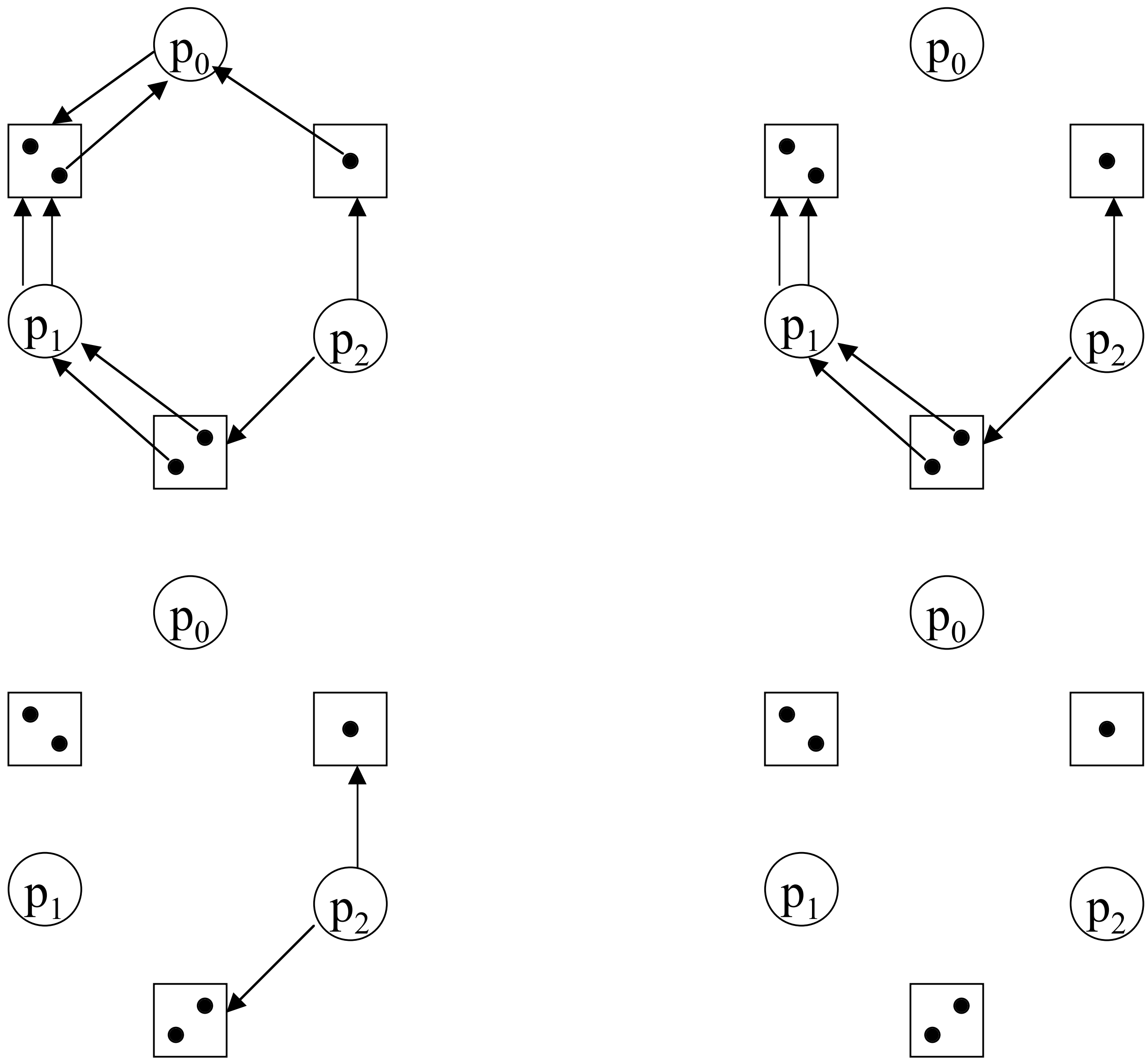




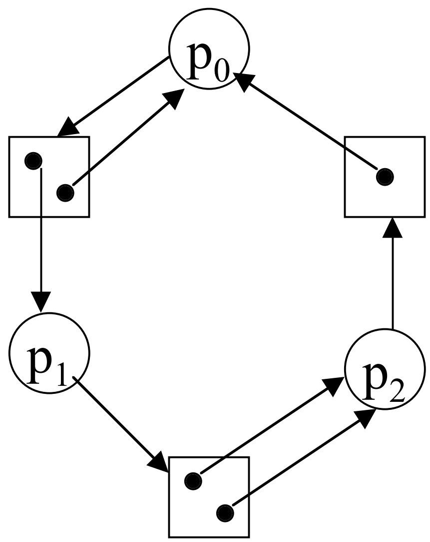
Example RRG



Example RRG



Example RRG

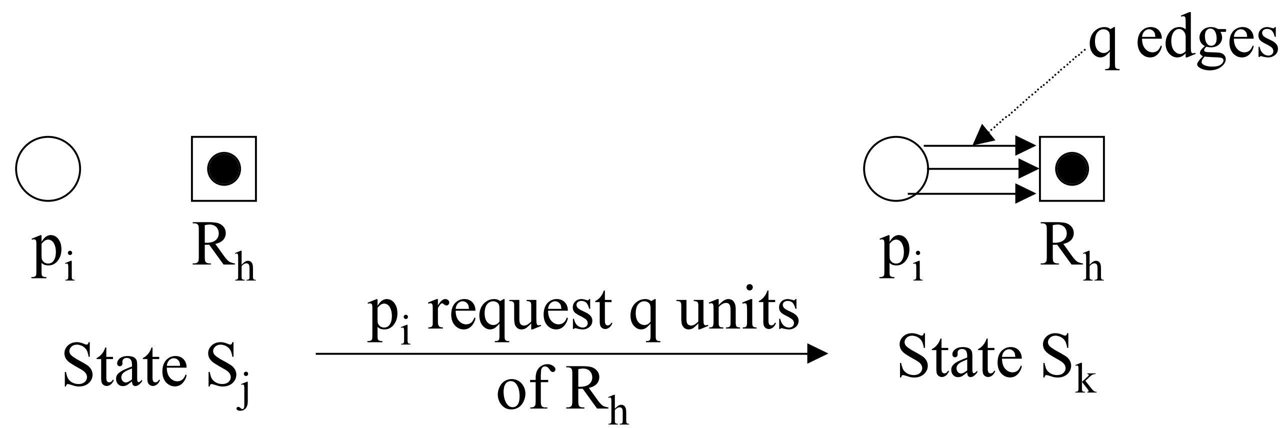


Consumable Resource Graphs (CRGs)

- Number of units varies, have producers/consumers
- Nodes = $\{p_0, p_1, \ldots, p_n\} \cup \{R_1, R_2, \ldots, R_m\}$
- Edges connect p_i to R_j, or R_j to p_i
  - (p_i, R_j) is a request edge for one unit of R_j
  - (R_j, p_i) is an producer edge (must have at least one producer for each R_j)
- For each R_j there is a count, w_j of units R_j

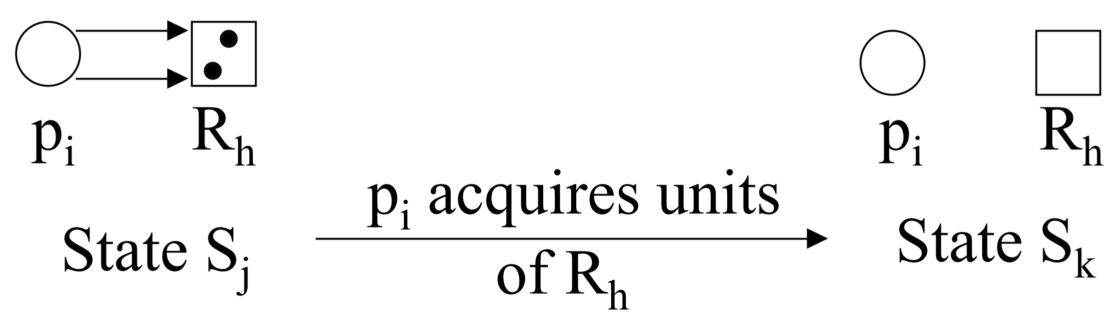
State Transitions due to Request

- In S_j, p_i is allowed to request any number of units of R_h, provided p_i has no outstanding requests.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from Sj by adding q request edges from p_i to R_h



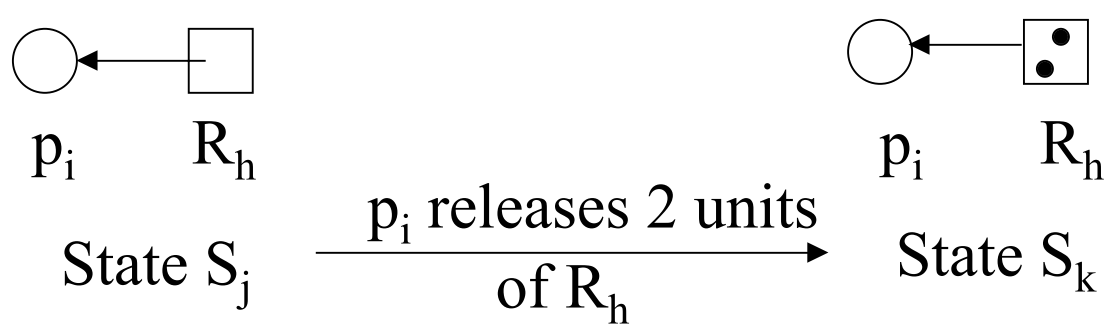
State Transition for Acquire

- In S_j, p_i is allowed to acquire units of R_h, iff there is (p_i, R_h) in the graph, and all can be satisfied.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from Sj by deleting each request edge and decrementing w_h.

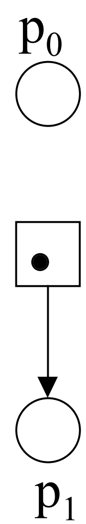


State Transition for Release

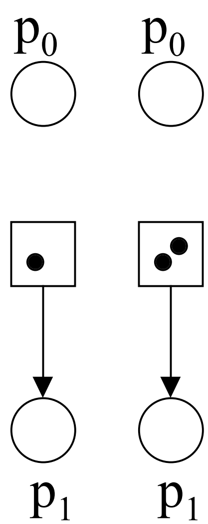
- In S_j, p_i is allowed to release units of R_h, iff there is (R_h, p_i) in the graph, and there is no request edge from p_i.
- $S_j \rightarrow S_k$, where the RRG for S_k is derived from S_j by incrementing w_h.



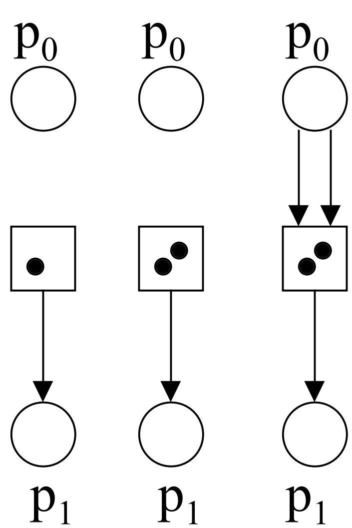
Example



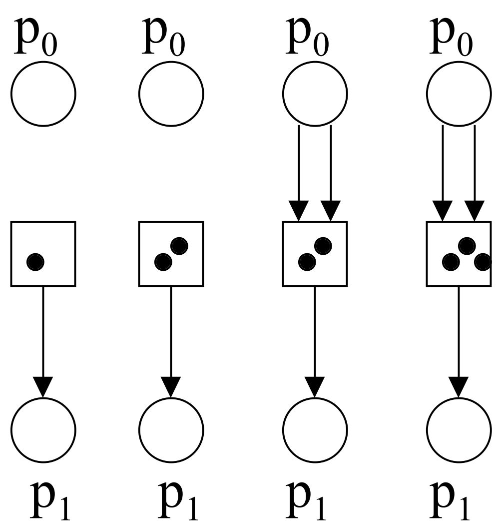
Example



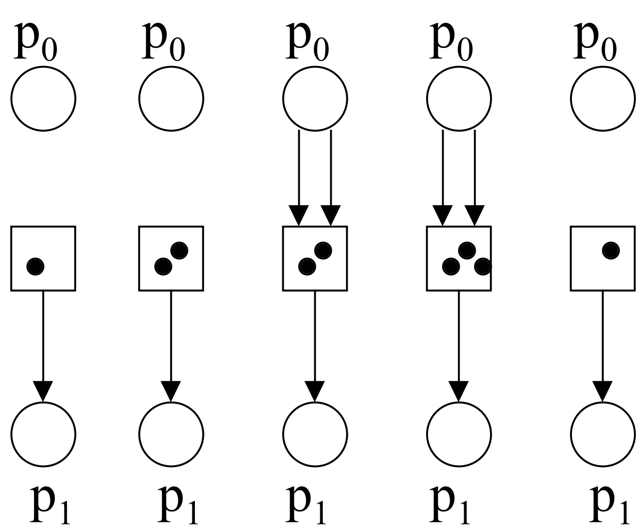
Example



Example



Example

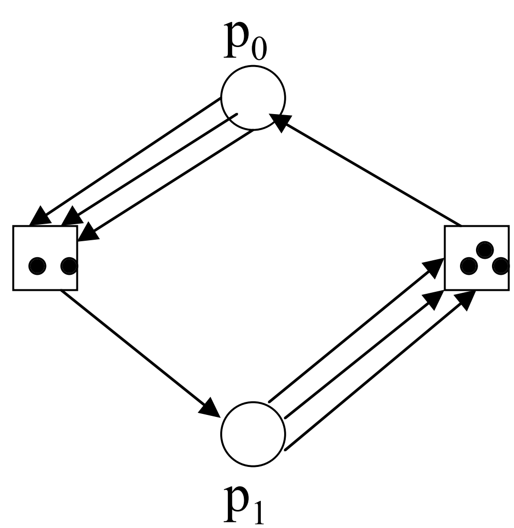


Deadlock Detection

- May have a CRG that is not completely reducible, but it is not a deadlock state
- For each process:
  - Find at least one sequence which leaves each process unblocked.
- There may be *different* sequences for different processes -- not necessarily an efficient approach

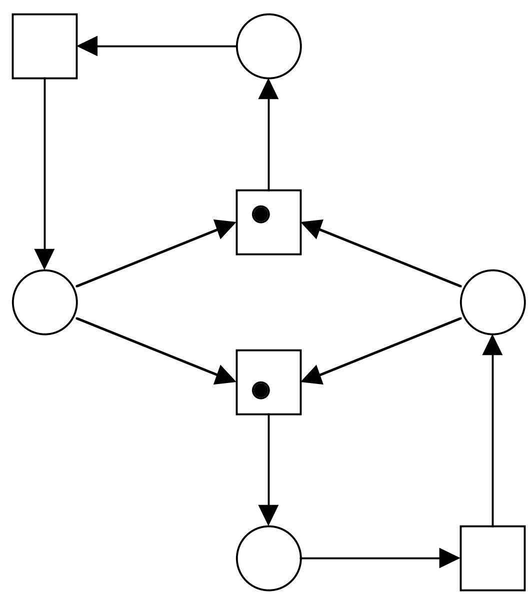
Deadlock Detection

- May have a CRG that is not completely reducible, but it is not a deadlock state
- Only need to find sequences, which leave each process unblocked.



Deadlock Detection

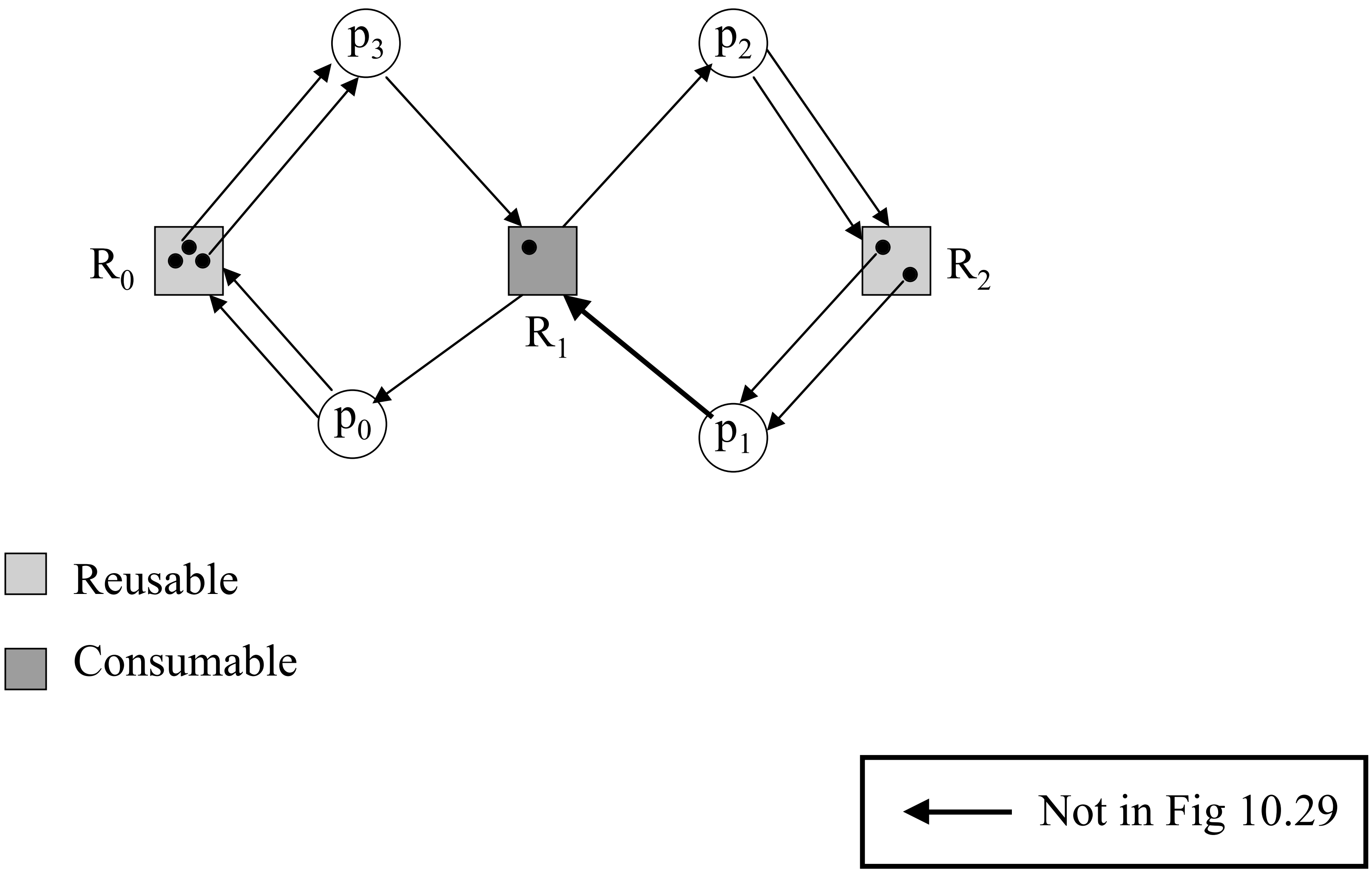
- May have a CRG that is not completely reducible, but it is not a deadlock state
- Only need to find a set of sequences, which leaves each process unblocked.



General Resource Graphs

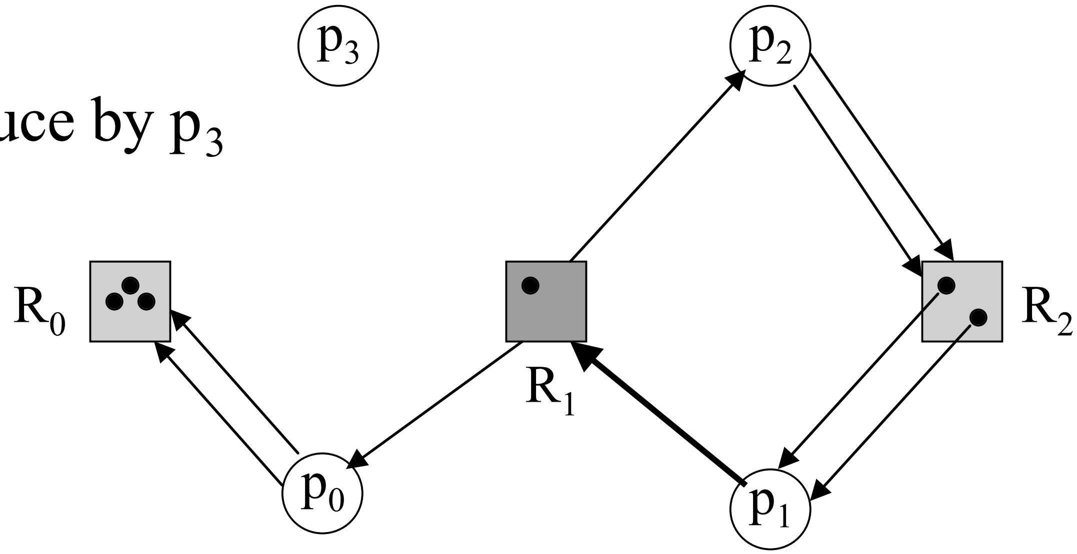
- Have consumable and reusable resources
- Apply consumable reductions to consumables, and reusable reductions to reusables
- See Figure 10.29

GRG Example (Fig 10.29)



GRG Example (Fig 10.29)

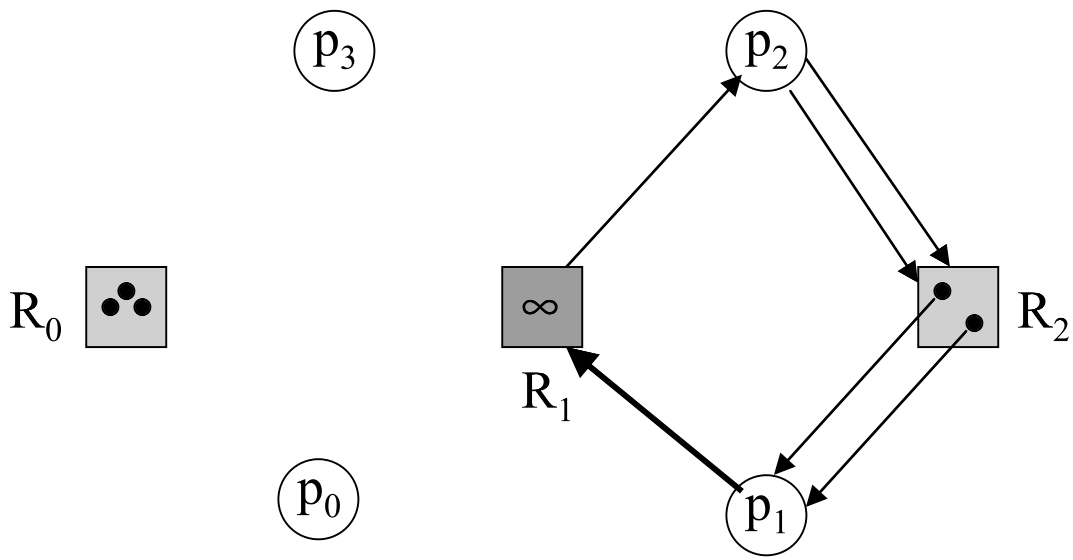
Reduce by p_3



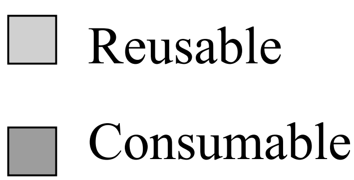
Reusable

Consumable

GRG Example (Fig 10.29)



Reduce by p_0



Recovery

- No magic here
  - Choose a blocked resource
  - Preempt it (releasing its resources)
  - Run the detection algorithm
  - Iterate if until the state is not a deadlock state