6. Deadlocks

6.1 Deadlocks with Reusable and Consumable Resources

6.2 Approaches to the Deadlock Problem

6.3 A System Model

- Resource Graphs
- State Transitions
- Deadlock States and Safe States

6.4 Deadlock Detection

- Reduction of Resource Graphs
- Special Cases of Deadlock Detection
- Deadlock Detection in Distributed Systems

6.5 Recovery from Deadlock

6.6 Dynamic Deadlock Avoidance

- Claim Graphs
- The Banker's Algorithm

6.7 Deadlock Prevention

Deadlocks

- Informal definition: Process is blocked on resource that will never be released.
- Deadlocks *waste resources*
- Deadlocks are *rare*:
  - Many systems ignore them
    - Resolved by explicit user intervention
  - Critical in many real-time applications
    - May cause damage, endanger life

Reusable/Consumable Resources

- Reusable Resources
  - Number of units is “constant”
  - Unit is either free or allocated; no sharing
  - Process requests, acquires, releases units

  Examples: memory, devices, files, tables
- Consumable Resources
  - Number of units varies at runtime
  - Process may create new units
  - Process may consume units

  Examples: messages, signals

Examples of Deadlocks

```
p1: ...                    p2: ...
   open(f1,w);                 open(f2,w);
   open(f2,w);                 open(f1,w);
   ...                          ...
```

- Deadlock when executed concurrently

```
p1: if (C) send(p2,m);     p2: ...
     while(1){...               while(1){...
       recv(p2,m);                recv(p1,m);
       send(p2,m);                send(p1,m);
       ... }                       ... }
```

- Deadlock when C not true

Deadlock, Livelock, Starvation

- *Deadlock:* Processes are blocked
- *Livelock:* Processes run but make no progress
- Both deadlock and livelock lead to *starvation*
- Starvation may have other causes
  - ML scheduling where one queue is never empty
  - Memory requests: unbounded stream of 100MB requests may starve a 200MB request

Approaches to Deadlock Problem

1. Detection and Recovery
   - Allow deadlock to happen and eliminate it
2. Avoidance (dynamic)
   - Runtime checks disallow allocations that might lead to deadlocks
3. Prevention (static)

   Restrict type of request and acquisition to make deadlock impossible

System Model for Deadlock Detection, Avoidance, etc.

- Assumptions:
  - When a process requests a resource, either the request is fully granted or the process blocks
  - No partial allocation
  - A process can only release resources that it holds
- *Resource graph:*
  - Vertices are processes, resources, resource units
  - Edges (directed) represent requests and allocations of resources

System Model: Resource Graph

Resource graph:

- Process = Circle
- Resource = Rectangle with small circles for each unit
- Request = Edge from process to resource class
- Allocation = Edge from resource unit to process

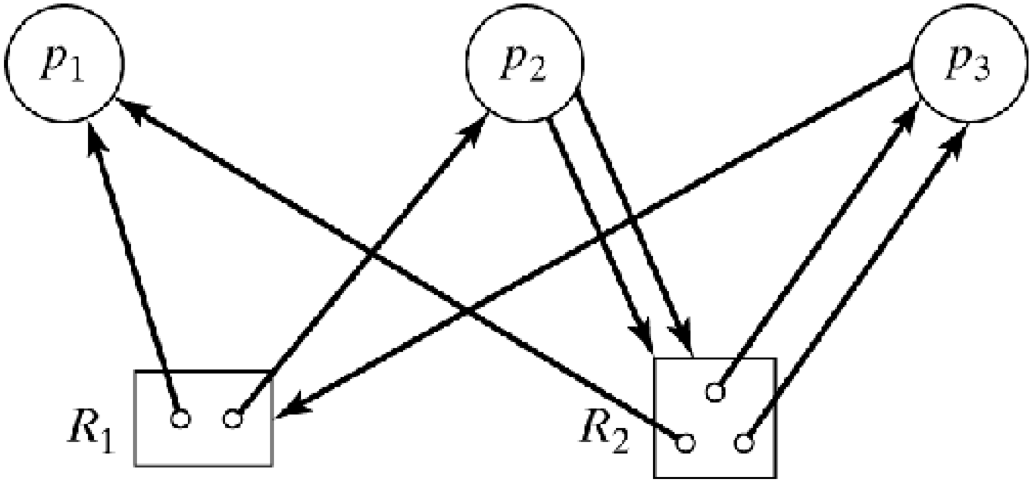


Figure 6-1

System Model: State Transitions

Request: Create new request edge $p_i \rightarrow R_j$

- p_i has no outstanding requests
- number of edges between p_i and R_j cannot exceed total units of R_j

Acquisition: Reverse request edge to $p_i \leftarrow R_j$

- All requests of p_i are satisfiable
- p_i has no outstanding requests

Release: Remove edge $p_i \leftarrow R_j$

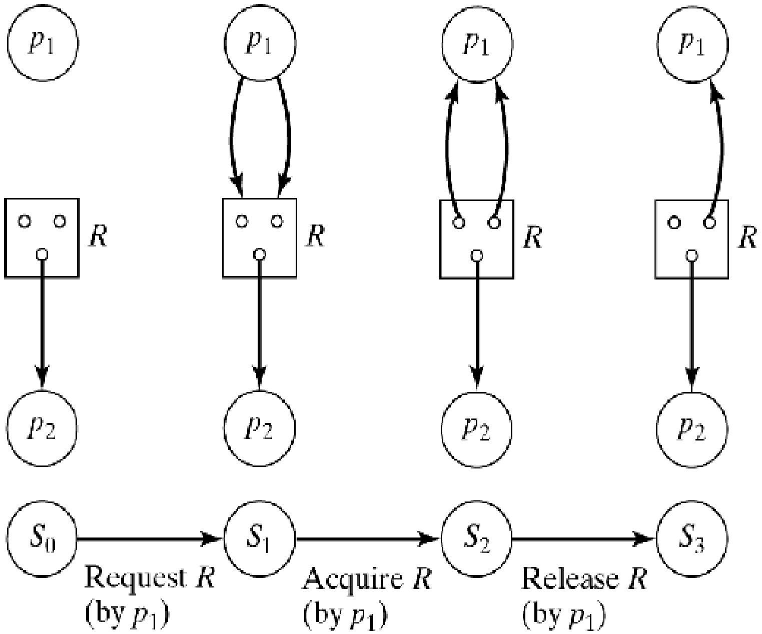


Figure 6-2

System Model

- A process is *blocked* in state S if it cannot request, acquire, or release any resource.
- A process is *deadlocked* in state S if it is currently blocked now and remains blocked in all states reachable from state S
- A state is a *deadlock state* if it contains a deadlocked process.
- State S is a *safe state* if no deadlock state can be reached from S by any sequence of request, acquire, release.

Example

2 processes p_1 , p_2; 2 resources R_1, R_2,

- p_1 and p_2 both need R_1 and R_2
- p_1 requests R_1 before R_2 and releases R_2 before R_1
- p_2 requests R_2 before R_1 and releases R_1 before R_2

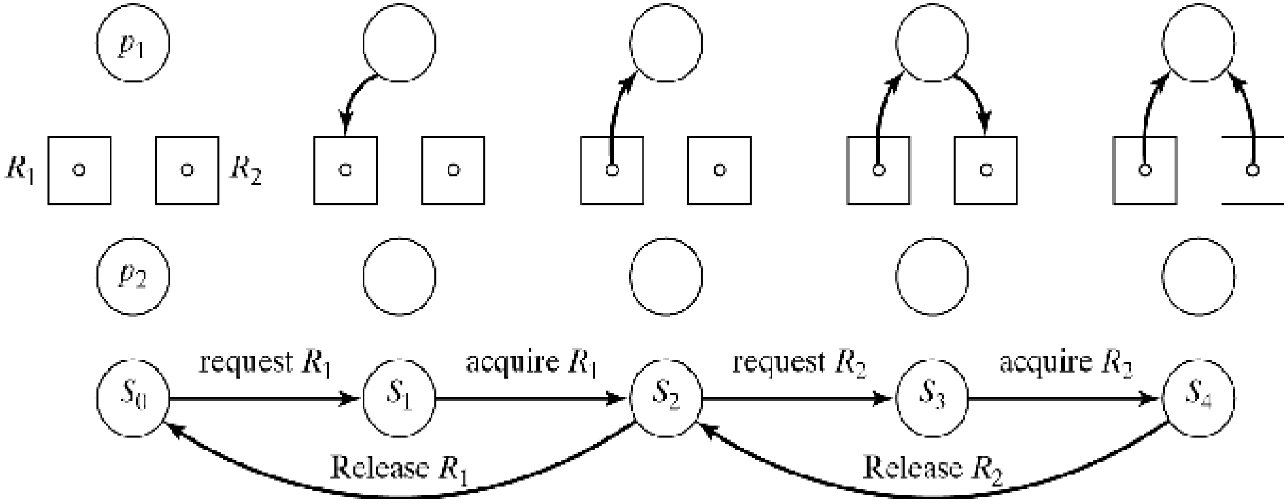


Figure 6-3: Transitions by p_1 only

Example

- p_1 and p_2 both need R_1 and R_2
- p_1 requests R_1 before R_2 and releases R_2 before R_1
- p_2 requests R_2 before R_1 and releases R_1 before R_2

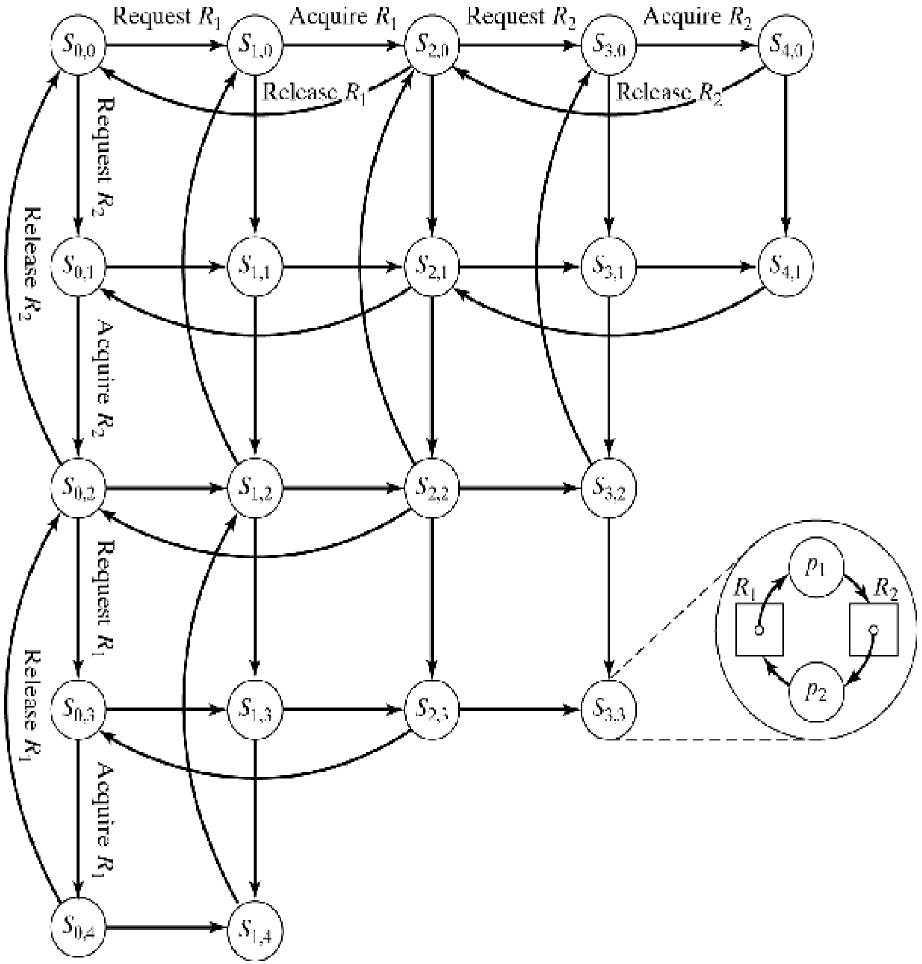


Figure 6-4: Transitions by p_1 and p_2

Deadlock Detection

- *Graph Reduction:* Repeat the following
  1. Select unblocked process ***p***
  2. Remove ***p*** and all request and allocation edges
- Deadlock ⇔ Graph not completely reducible.
- All reduction sequences lead to the same result.

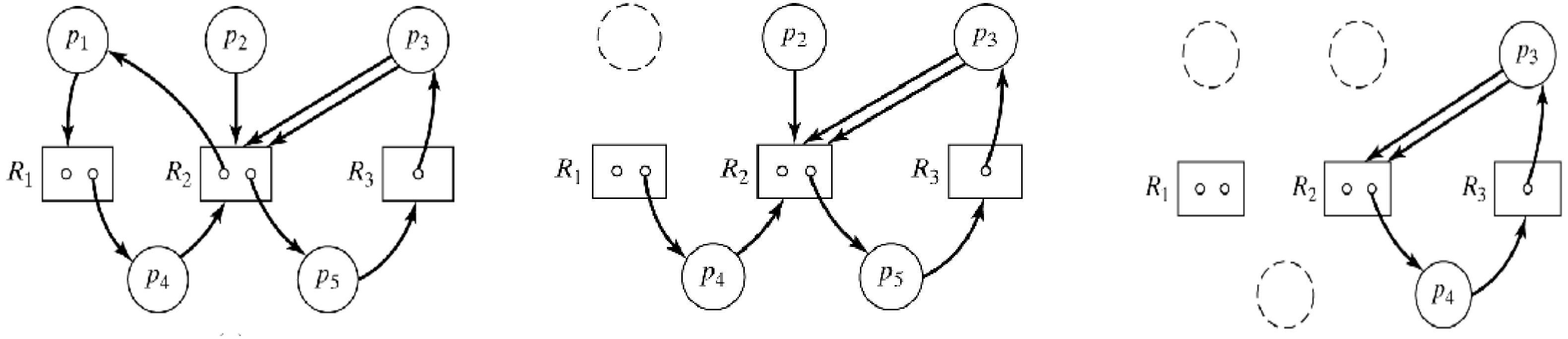


Figure 6-5

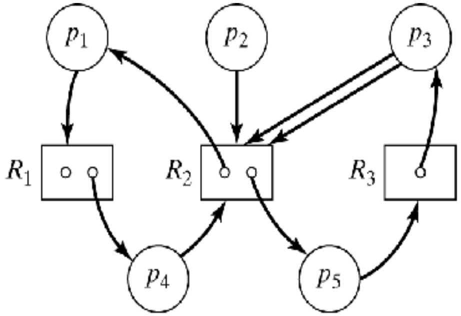
Special Cases of Detection

- Testing for whether a specific process ***p*** is deadlocked:
  - Reduce until ***p*** is removed or graph irreducible
- Continuous detection:
  1. Current state not deadlocked
  2. Next state ***T*** deadlocked only if:
     a. Operation was a request by ***p*** and
     b. ***p*** is deadlocked in ***T***
  3. Try to reduce ***T*** by ***p***

Special Cases of Detection

- Immediate allocations
  - All satisfiable requests granted immediately
  - *Expedient state:* state with no satisfiable request edges
  - If all requests are granted immediately, all states are expedient.

Not expedient (p1->R1)

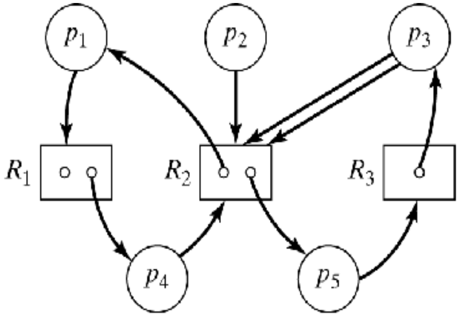


Special Cases of Detection

- Immediate allocations, continued.
  - *Knot* : A set K of nodes such that
    - Every node in K reachable from any other node in K
    - No outgoing edges from any node in K
  - Knot in expedient state ⇒ Deadlock :
  - Reason:
    - All processes in K must have outstanding requests
    - Expedient state means requests not satisfiable

(Remove R2->p1: knot R2,p3,R3,p5)

(Reverse edge p1->R1): expedient state



Special Cases of Detection

- For single-unit resources, cycle ⇒ deadlock
  - Every p must have a request edge to R
  - Every R must have an allocation edge to p
  - R is not available and thus p is blocked
- *Wait-For Graph (wfg):* Show only processes
  - Replace $p_1 \rightarrow R \rightarrow p_2$ by $p_1 \rightarrow p_2$: p_1 waits for p_2

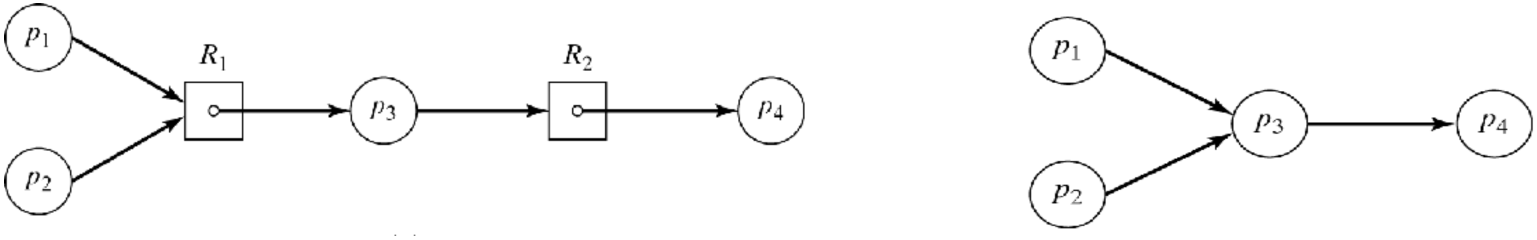


Figure 6-6

Deadlock detection in Distributed Systems

- Central Coordinator (CC)
  - Each machine maintains a local wfg
  - Changes reported to CC
  - CC constructs and analyzes global wfg
- Problems
  - Coordinator is a performance bottleneck
  - Communication delays may cause phantom deadlocks

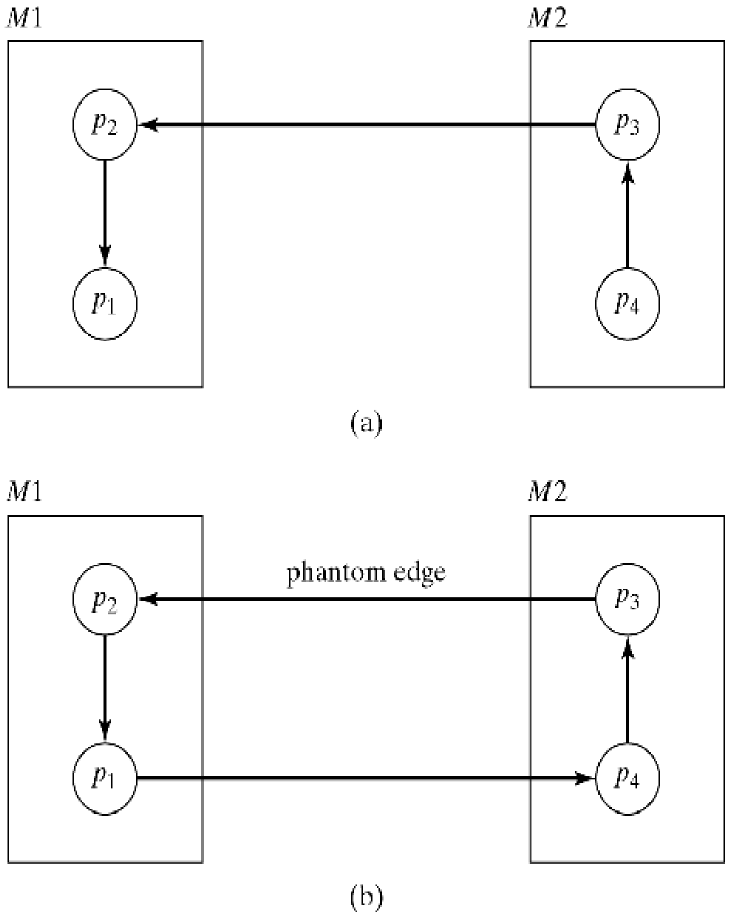


Figure 6-7

Detection in Distributed Systems

- Distributed Approach
  - Detect cycles using probes.
  - If process p_i blocked on p_j , it launches probe $p_i \rightarrow p_j$
  - pj sends probe $p_i \rightarrow p_j \rightarrow p_k$ along all request edges, etc.
  - When probe returns to p_i, cycle is detected

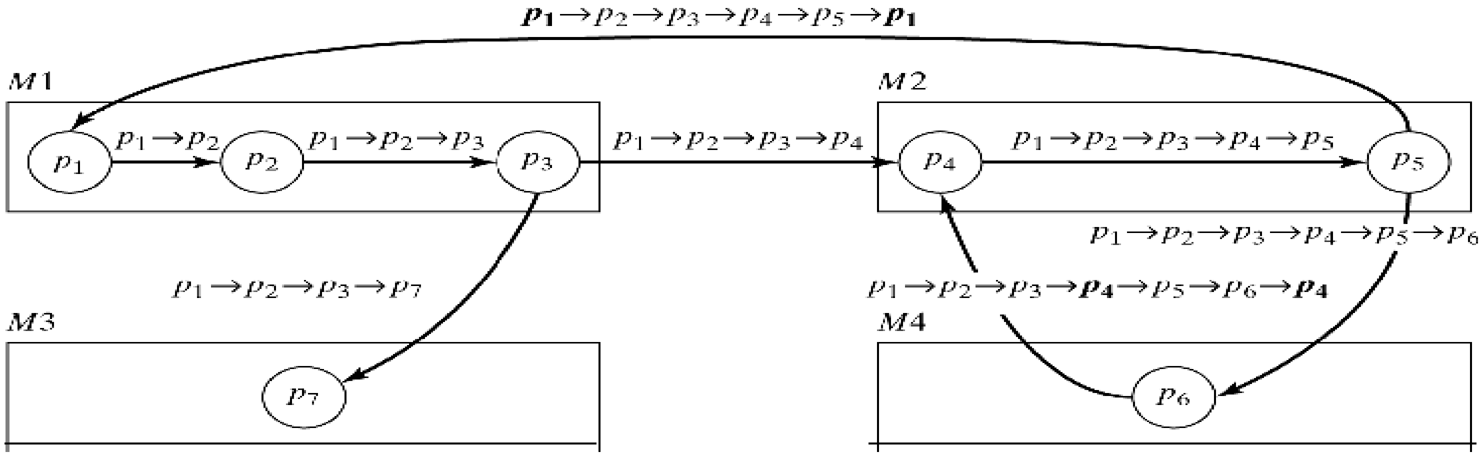


Figure 6-8

Recovery from Deadlock

- Process termination
  - Kill all processes involved in deadlock; or
  - Kill one at a time. In what order?
    - By priority: consistent with scheduling
    - By cost of restart: length of recomputation
    - By impact on other processes: CS, producer/consumer
- Resource preemption
  - Direct: Temporarily remove resource (e.g., Memory)
  - Indirect: Rollback to earlier “checkpoint”

Dynamic Deadlock Avoidance

- Maximum Claim Graph
  - Process indicates *maximum* resources needed
  - *Potential* request edge **$p_i \rightarrow R_j$** (dashed)
  - May turn into *real* request edge

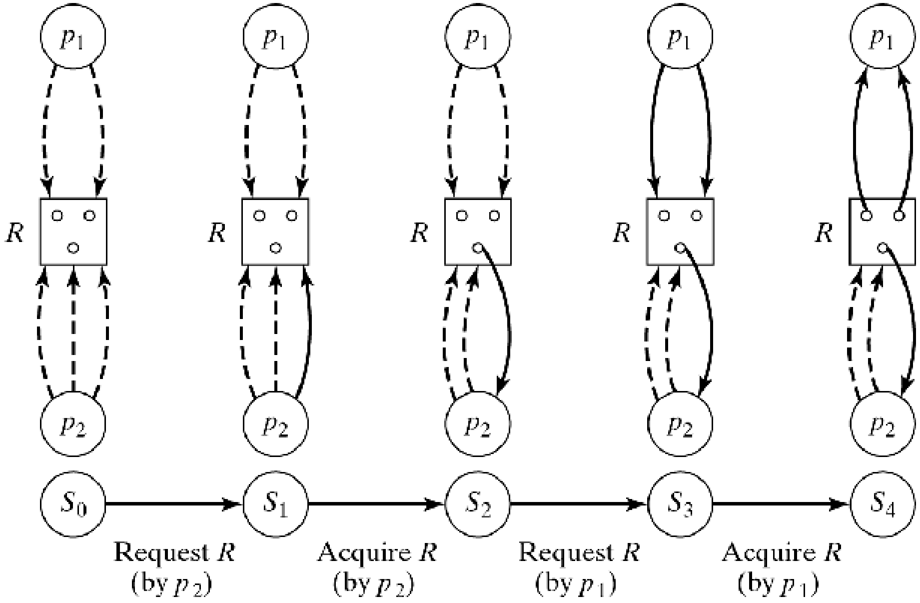


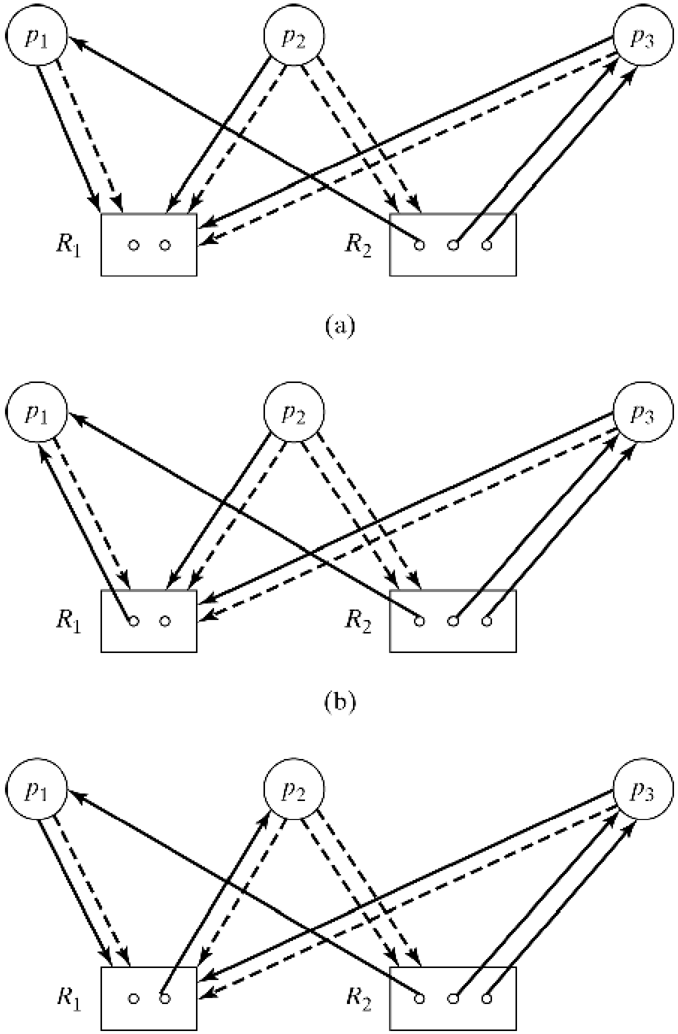
Figure 6-9

Dynamic Deadlock Avoidance

- Theorem: Prevent acquisitions that do not produce a completely reducible graph
  $\Rightarrow$ All state are safe.
- Banker's algorithm (Dijkstra):
  - Given a satisfiable request, $\boldsymbol{p \rightarrow R}$, temporarily grant request, changing $\boldsymbol{p \rightarrow R}$ to $\boldsymbol{R \rightarrow p}$
  - Try to reduce new claim graph, treating claim edges as actual requests.
  - If new claim graph is completely reducible proceed. If not, reverse temporary acquisition $\boldsymbol{R \rightarrow p}$ back to $\boldsymbol{p \rightarrow R}$
- Analogy with banking: resources correspond to currencies, allocations correspond to loans, maximum claims correspond to credit limits

Example of banker's algorithm

- Claim graph (a). Which requests for R1 can safely be granted?
- If p1's request is granted, resulting claim graph (b) is reducible (p1,p3,p2).
- If p2's request is granted, resulting claim graph (c) is not reducible.
- Exercise: what about p3's request?



(a)

(b)

(c)

Figure 6-10

Dynamic Deadlock Avoidance

- Special Case: Single-unit resources
  - Check for cycles after tentative acquisition
    Disallow if cycle is found (cf. Fig 6-11(a))
  - If claim graph contains no *undirected* cycles,
    all states are safe (cf. Fig 6-11(b))
    (Because no *directed* cycle can ever be formed.)

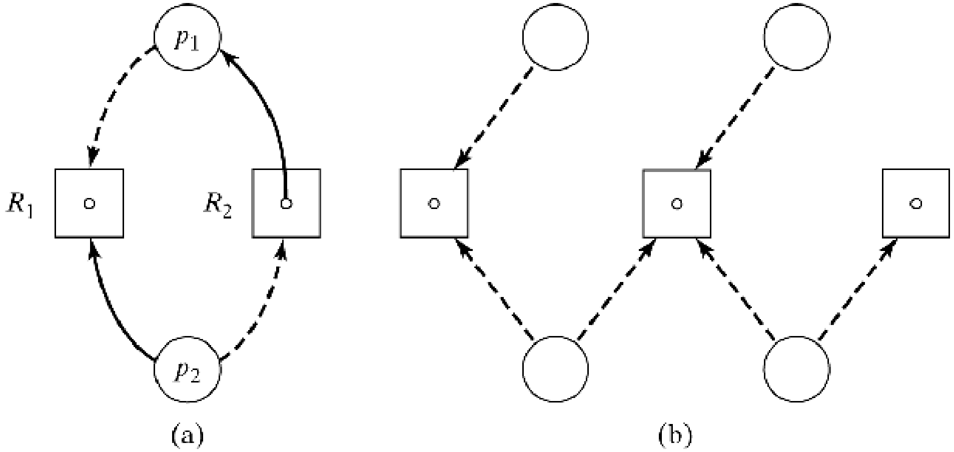


Figure 6-11

Deadlock Avoidance – Another Approach

- Restrict waits to avoid “wait for” cycles.
- Each process has timestamp. Ensure that either
  - Younger process never waits for older process; or
  - Older process never waits for younger process
- When process R requests resource that process H holds (two variants)

1. *Wait/die algorithm:* (Younger process never waits)
   - If R is older than H, R waits
   - If R is younger than H it dies, restarts
2. *Wound/wait algorithm:* (Older process never waits)
   - If R is older than H, resources is preempted (which may mean process is killed, restarted)
   - If R is younger than H, R waits

- Restarted process keeps old timestamp

Comparison of deadlock avoidance schemes

- Wound/wait and wait/die kill processes even when there is no deadlock (more aggressive).
- Wait/die generally kills more processes than wound/wait, but generally at an earlier stage
- Note: Wait/die and Wound/wait are sometimes classified as prevention schemes rather than avoidance schemes

Deadlock Prevention

- Deadlock requires the following conditions:
  - Mutual exclusion:
    - Resources not sharable
  - Hold and wait:
    - Process must be holding one resource while requesting another
  - Circular wait:
    - At least 2 processes must be blocked on each other

Deadlock Prevention

- Eliminate mutual exclusion:
  - Not possible in most cases
  - Spooling makes I/O devices sharable
- Eliminate hold-and-wait
  - Request all resources at once
  - Release all resources before a new request
  - Release all resources if current request blocks
- Eliminate circular wait
  - Order all resources: $\boldsymbol{SEQ(R_i) \neq SEQ(R_j)}$
  - Process must request in ascending order

History

- Originally developed by Steve Franklin
- Modified by Michael Dillencourt, Summer, 2007
- Modified by Michael Dillencourt, Spring, 2009
- Modified by Michael Dillencourt, Winter, 2010