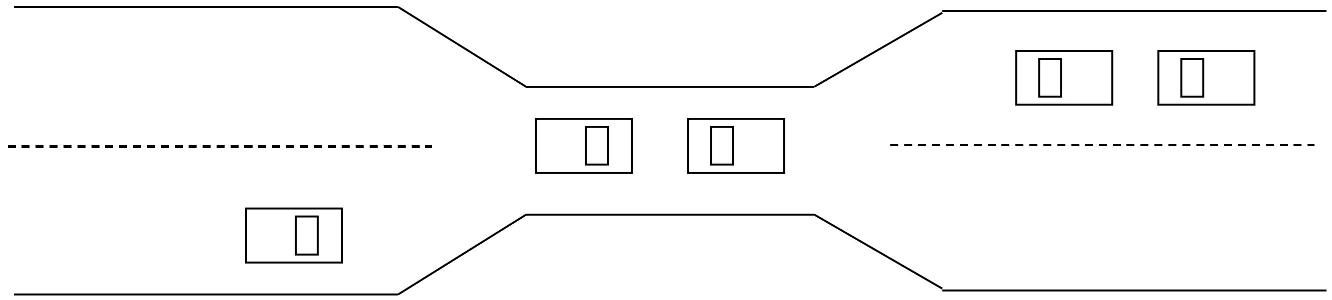
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
  - System has 2 disk drives; P_1 and P_2 each hold one disk drive and each needs another one
  - Semaphores *A* and *B*, initialized to 1, acquired in different orders by two processes



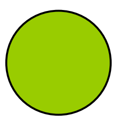
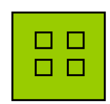
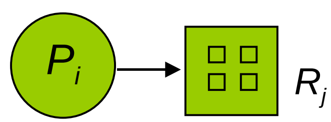
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- A deadlock can be resolved if one car backs up: preempt and rollback
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

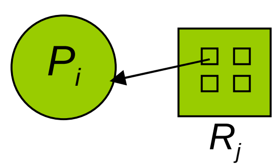
Deadlock Characterization

- Resource types R_1, R_2, . . ., R_m (CPU cycles, memory space, I/O devices, ...)
  - Each resource type R_i has W_i instances.
  - Each process utilizes a resource as follows: request, use, release
- Deadlock can arise if the following four conditions are met simultaneously:
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

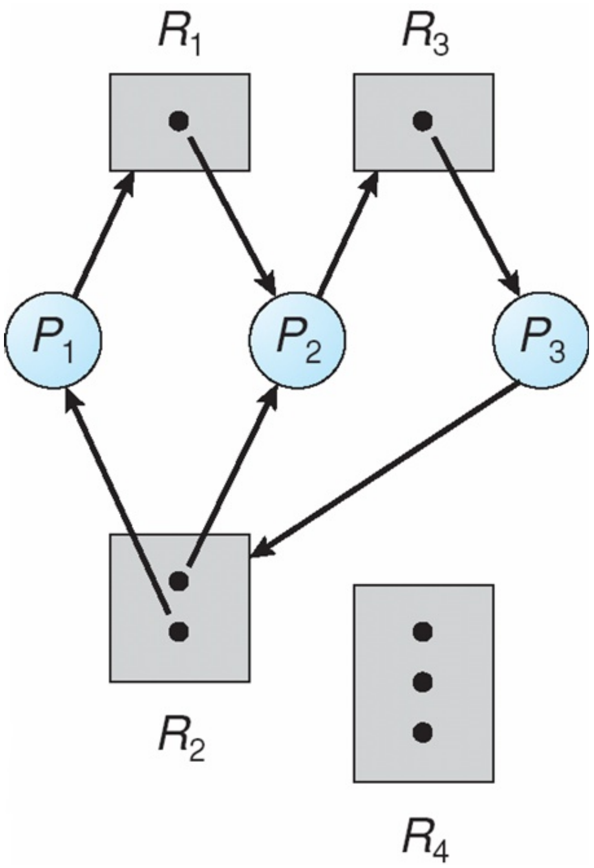
Resource-Allocation Graph

- A set of vertices V and a set of edges E.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
  - Request edge – directed edge $P_1 \rightarrow R_j$
  - Assignment edge – directed edge $R_j \rightarrow P_i$

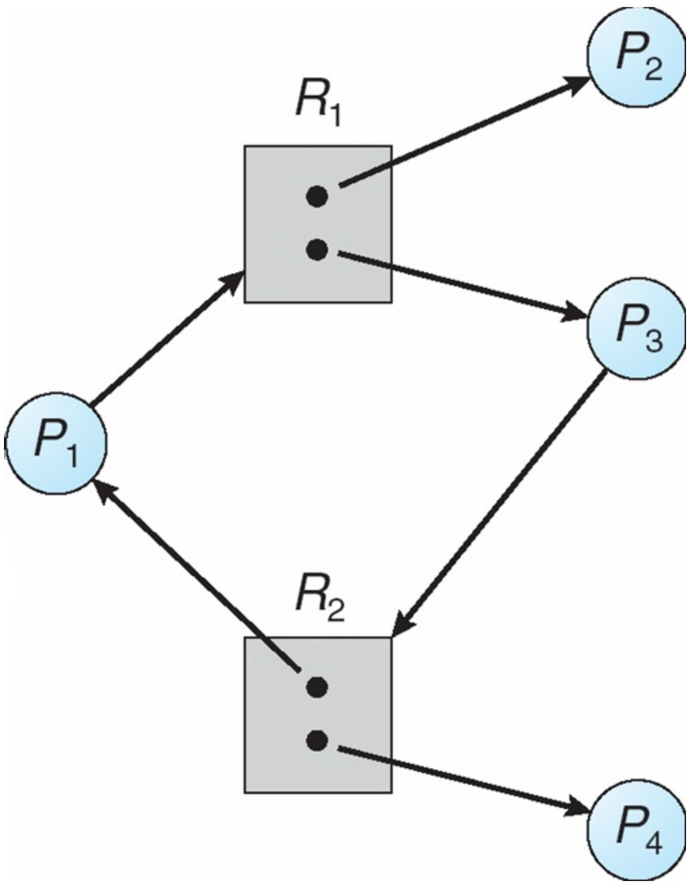
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Deadlock!



Cycle but no deadlock



Basic facts:

- No cycles → no deadlock
- Cycles:
  - only one instance per resource type → deadlock
  - several instances per resource type → possible deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system
    - Used by most operating systems, including UNIX!

Deadlock Prevention

Restrain the ways request can be made

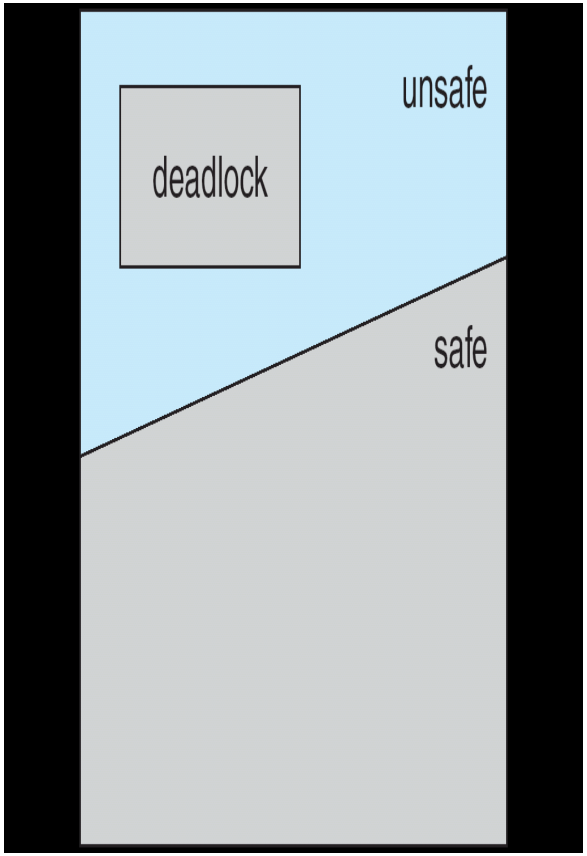
- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **No Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
- Low resource utilization; starvation possible
- **Preemption** – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **No Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

- Requires that the system has some additional a priori information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- A system is in safe state → no deadlocks
- A system is in unsafe state → possibility of deadlock
- Avoidance = ensure that a system will never enter an unsafe state.

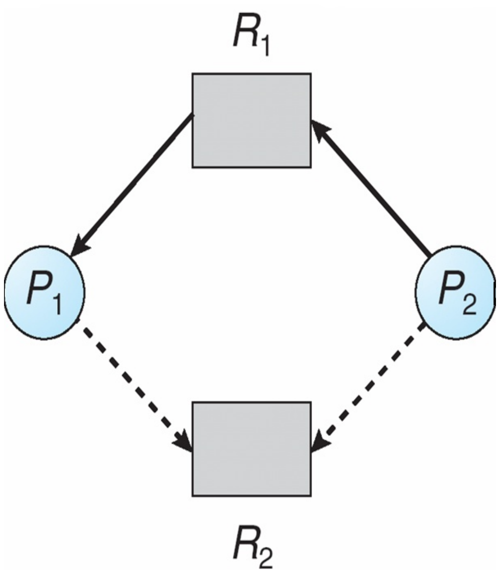


Avoidance algorithms

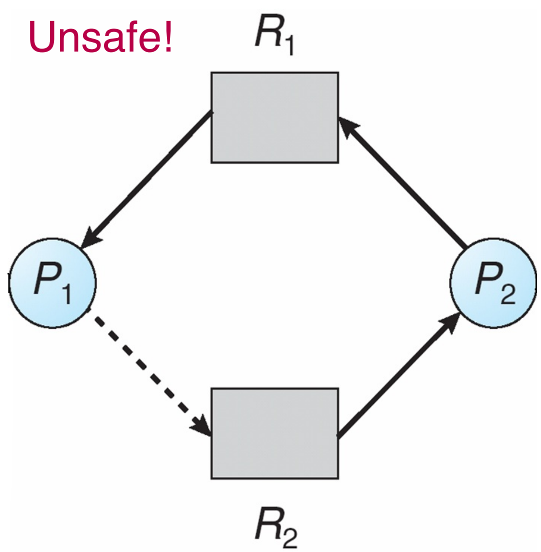
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j (dashed line)
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Unsafe!



Algorithm:

Suppose that process P_i requests a resource R_j

The request is granted only if converting the request edge to an assignment edge does not result in the formation of a cycle

Banker s Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time
- Data structures (n = number of processes, and m = number of resources types)

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

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

Allocation*:* *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *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

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i$ <= *Work*

   If no such *i* exists, go to step 4

3. *Work* = *Work* + $Allocation_i$
   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 P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants *k* instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

If safe → the resources are allocated to Pi

If unsafe → Pi must wait, and the old resource-allocation state is restored

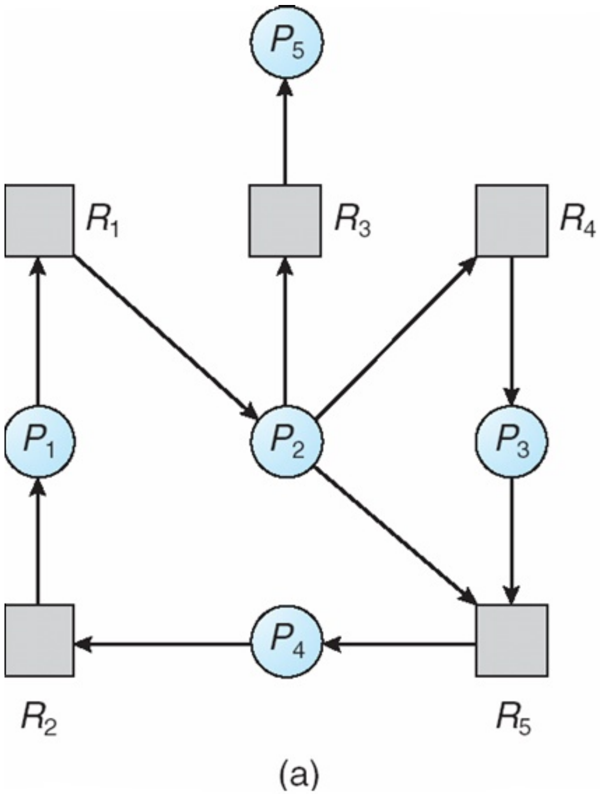
Deadlock Detection

- Allow system to enter deadlock state, run a detection algorithm, use a recovery scheme
- Single instance of each resource:

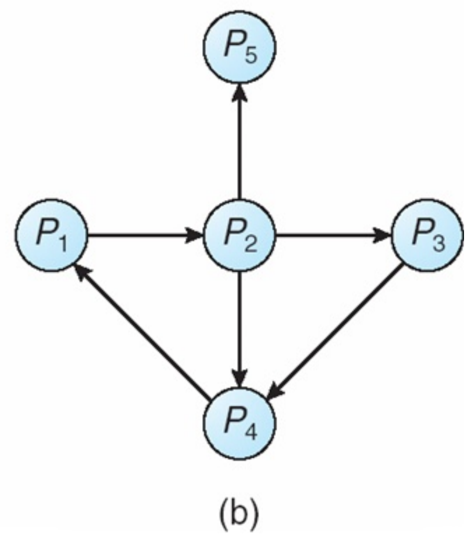
  Maintain *wait-for* graph

  - Nodes are processes
  - $P_i \rightarrow P_j$ iff P_i is waiting for P_j
  - Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
  - An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-allocation graph



Wait-for graph

Several Instances of a Resource Type

Available*:* A vector of length *m* indicates the number of available resources of each type.

Allocation*:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request*:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively Initialize:

   (a) *Work* = *Available*

   (b) For $i = 1,2, \ldots, n$, if $Allocation_i$ /= 0, then *Finish*[i] = false; else *Finish*[i] = *true*

2. Find an index i such that both:

   (a) *Finish*[*i*] == *false*

   (b) $Request_i$ <= *Work*

   If no such i exists, go to step 4

3. *Work* = *Work* + $Allocation_i$
   Finish[*i*] = *true*
   Go to step 2

4. If *Finish*[*i*] == false, for some i, $1 <= i < n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - One for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Recovery from Deadlock

Process termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
    - Priority of the process
    - How long process has computed, and how much longer to completion
    - Resources the process has used
    - Resources process needs to complete
    - How many processes will need to be terminated
    - Is process interactive or batch?

Resource preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor