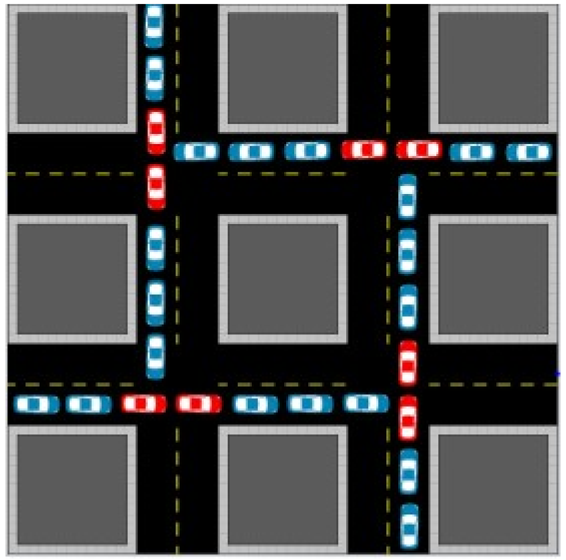
CECS 326
Operating Systems

4-Deadlocks

(Reference: Chapter 8. Operating system Concepts by Silberschatz, Galvin and Gagne)

System Model

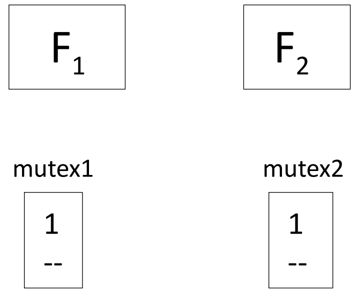
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices, etc.
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - Request
  - Use
  - Release
- Request and release of resources may be made by system calls, such as request() & release() device, open() & close() file, allocate() & free() memory
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set



Deadlock Example 1

- Two files F_1 and F_2, requires mutually exclusive access
- Semaphore solution to enforce mutual exclusion
  - Semaphore mutex1 initialized to 1, for F_1
  - Semaphore mutex2 initialized to 1, for F_2
- Two processes, both need to use F_1 and F_2 as follows. Deadlock may result

```
Process 1:        Process 2:
. . . . . .
wait(mutex1);     wait(mutex2);
wait(mutex2);     wait(mutex1);
Use F1 and F2     Use F1 and F2
signal(mutex2);   signal(mutex1);
signal(mutex1);   signal(mutex2);
. . . . . .
```



Deadlock Example 2

```
void transaction(Account from, Account to, double amount) {
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transaction 1
from = A; lock1 = lock of A
to = B; lock2 = lock of B

Transaction 2
from = B; lock1 = lock of B
to = A; lock2 = lock of A

- Transactions 1 & 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A, i.e.

  Transaction 1 calls transaction(A, B, 25)

  Transaction 2 calls transaction(B, A, 50)
- Can deadlock occur?

Deadlock Example 2 – A Scenario

Transaction 1

```
from = A
to = B
amount = 25
lock1 = get_lock(from) = $lock_A$
lock2 = get_lock(to) = $lock_B$
acquire($lock_A$);
acquire($lock_B$);
    withdraw(A,25);
    deposit(B,25);
release($lock_B$);
release($lock_A$);
```

Transaction 2

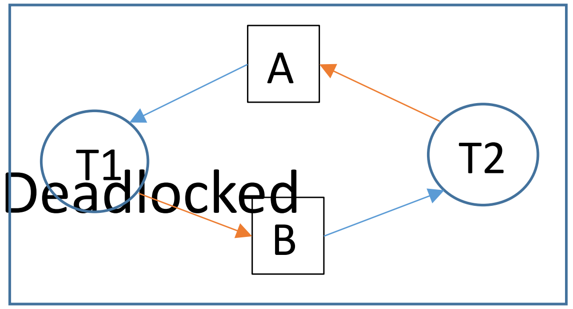
```
from = B
to = A
amount = 50
lock1 = get_lock(from) = $lock_B$
lock2 = get_lock(to) = $lock_A$
acquire($lock_B$);
acquire($lock_A$);
    withdraw(B,50);
    deposit(A,50);
release($lock_A$);
release($lock_B$);
```

Deadlock Characterization

Deadlock can arise if 4 conditions hold simultaneously (referred to as necessary conditions for deadlock)

- Mutual exclusion – at least one resource type in the system that only one process at a time can use it
- Hold and wait – a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption – at least one resource type in the system that, when allocated, can be released only voluntarily by the process holding it, after that process has completed its task
- Circular wait – there exists a set of waiting processes $\{P_0, P_1, \ldots, P_n\}$ 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_n is waiting for a resource that is held by P_0.

Scenario in Example 2: Deadlocked



Resource Allocation Graph

Graph G consists of a set of vertices V and a set of edges E

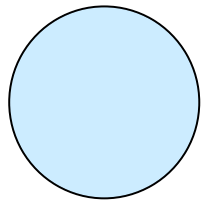
G = (V, E)

- 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

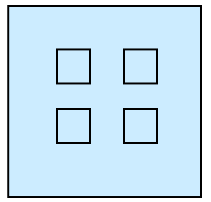
- The edge set includes:
  - request edge – directed edge $P_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph - Graphic Representation

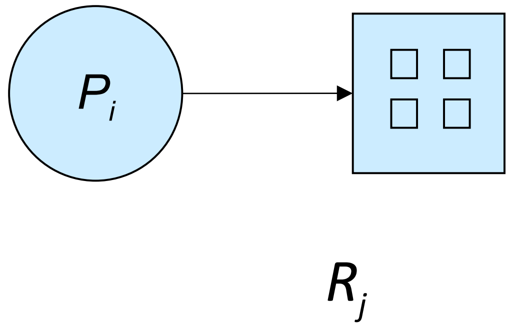
- Process



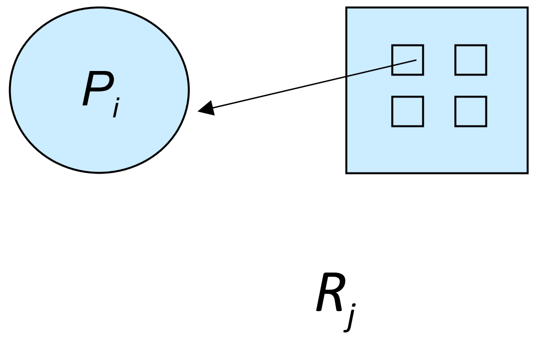
- Resource Type with 4 instances



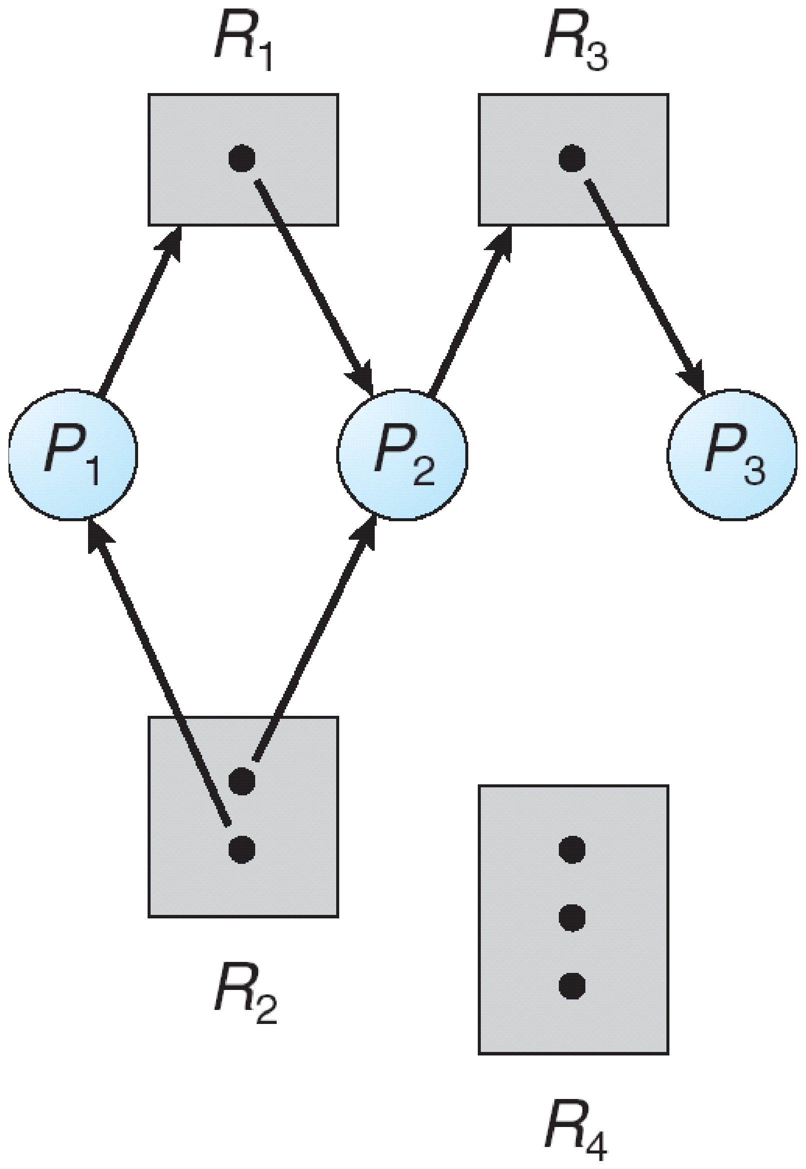
- P_i requests instance of R_j (request edge)



- P_i is holding an instance of R_j (assignment edge)

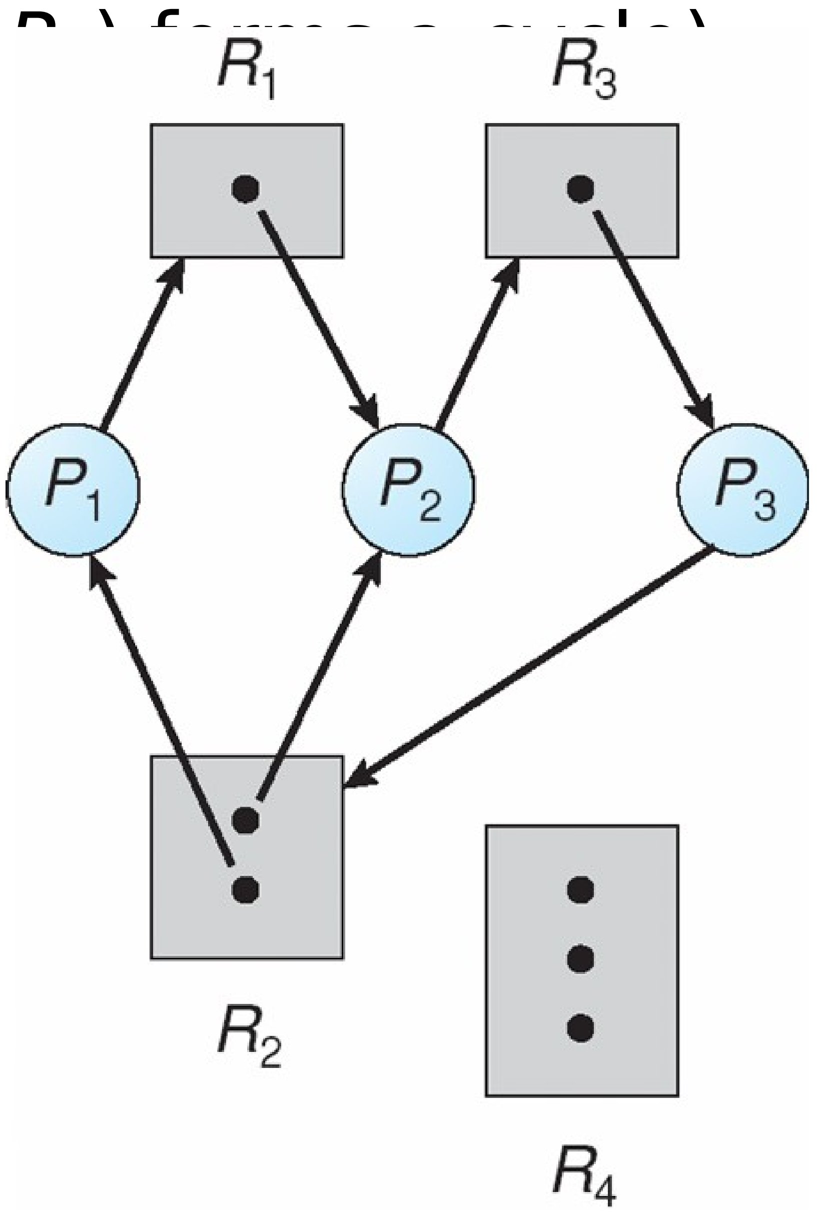


Example of Resource Allocation Graph

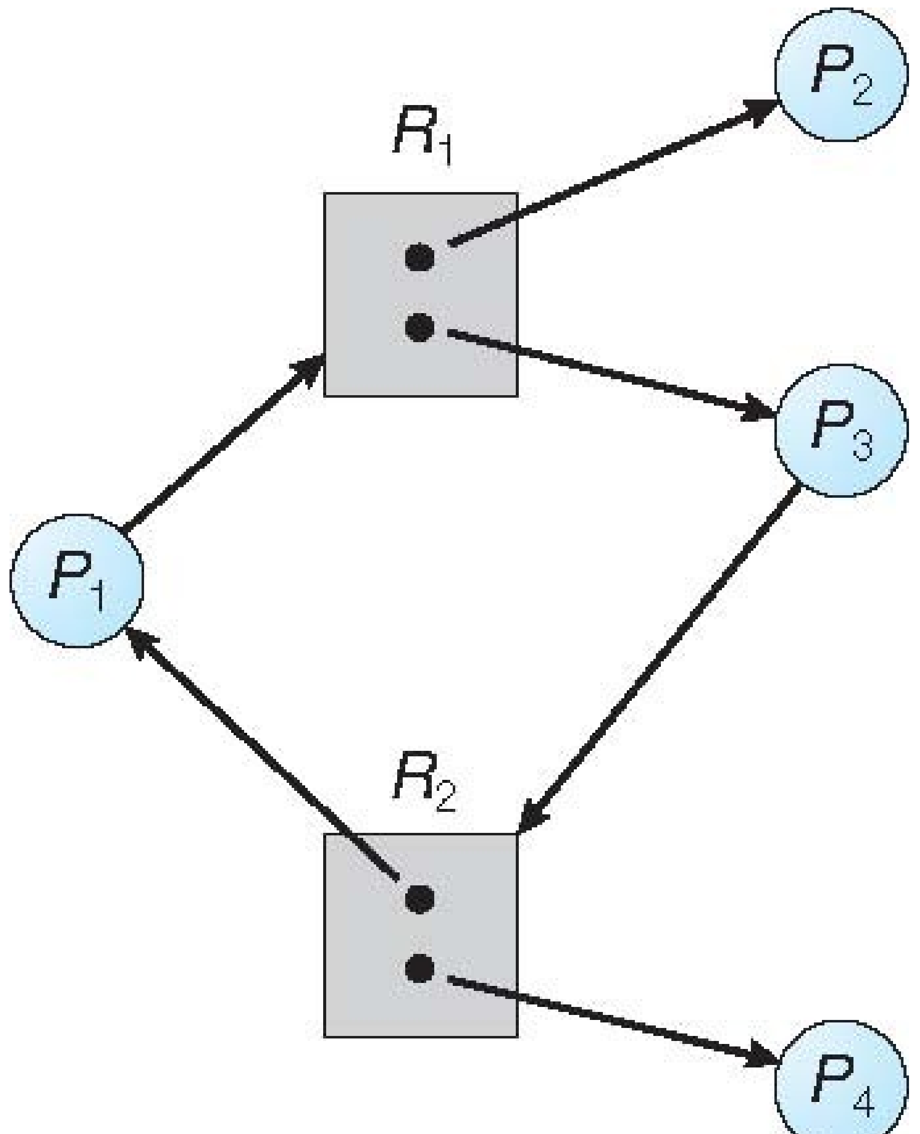


Resource Allocation Graph with a Deadlock

(P_1, R_1, P_2, R_3, P_3,



Graph with a Cycle But No Deadlock



Basic Facts

- If graph contains no cycle ⇒
  - no deadlock (circular wait is a necessary condition for deadlock)
- If graph contains a cycle ⇒
  - If only one instance per resource type, then deadlock
  - If several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
    - Deadlock prevention (by denying possibility of the existence of at least one of the necessary conditions for deadlock)
    - Deadlock avoidance
- Allow the system to enter a deadlock state, recover when deadlock Is detected
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems. Including UNIX

Deadlock Prevention

Restrain the ways that resource request can be made so that one of the following conditions will not hold:

- Mutual Exclusion – this is not required for sharable resources *e.g., read-only files), but must hold for non-sharable resources
- Hold and Wait – guarantee that whenever a process requests a resource, it doesn't 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 allocated to it
  - Low resource utilization, and starvation possible
- No Preemption –
  - A process holding some resources requests another resource that is not immediately available is required to release all that it is currently holding
  - 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
- 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

Deadlock Avoidance – Safe State

- When a process requests an available resource, system must decide if the immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of all the processes in the system 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 $j < i$. That is,
  - 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

Basic Facts

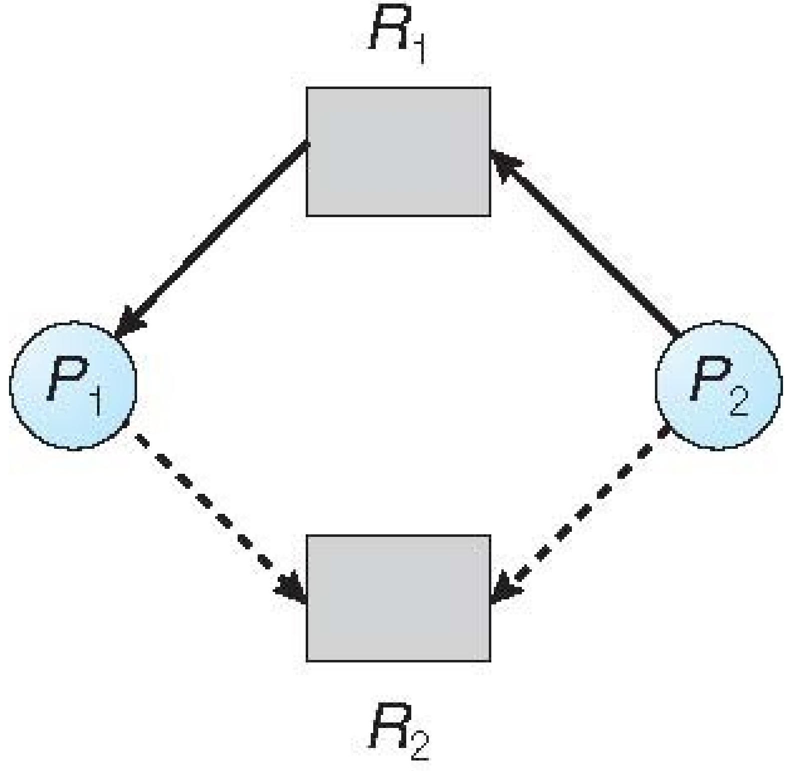
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state

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 Algorithm

- Resources must be claimed a priori in the system
- Claim edge $P_i \rightarrow R_j$ indicates that processes P_i may request resource R_j, represented by a dashed line in the graph
- Claim edge converts to request edge when a process requests a resource

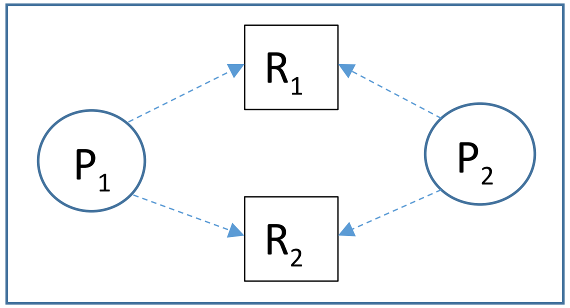


Resource-Allocation Graph Algorithm

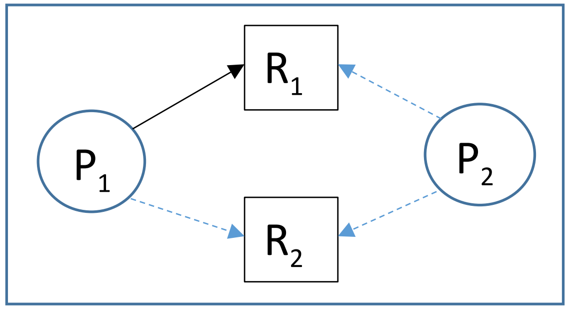
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Facing a request, if converting the request edge to an assignment edge will result in the formation of a cycle in the graph, it is deemed an unsafe state. A request can be granted only if it does not lead to an unsafe state.

Example Applying Resource-Allocation Graph Algorithm

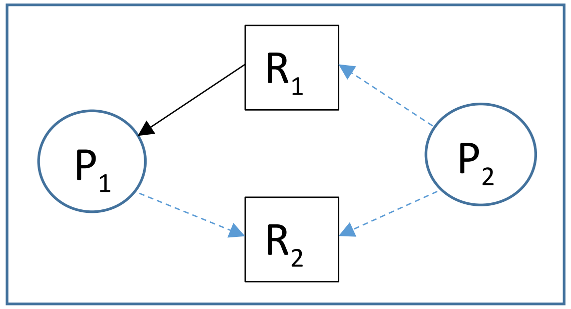
Initial state



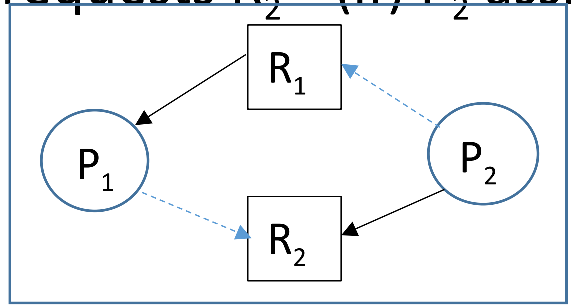
P_1 requests R_1



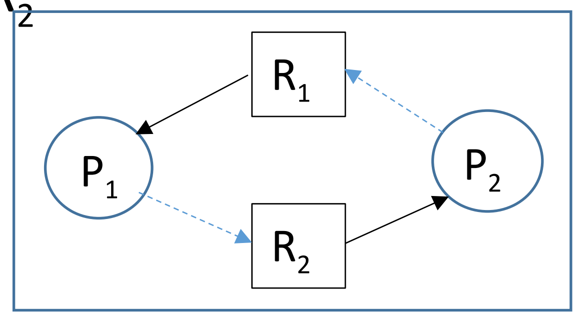
P_1 assigned R_1



P_2 requests R_2



(if) P_2 assigned R_2



Unsafe state!

Banker’s Algorithm

System model

- Multiple instances in a resource type
- 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 for the Banker's Algorithm

Let 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 Checking

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 an ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state, else unsafe

Resource-Request for Process P_i

Request**$_i$* = 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**$_i$**;

    Allocation**$_i$* **=** ***Allocation**$_i$* **+** ***Request**$_i$**;

    Need**$_i$* **=** ***Need**$_i$* **–** ***Request**$_i$**;

  Perform safety checking

  - If safe $\Rightarrow$ the resources are allocated to ***P**$_i$*
  - If unsafe $\Rightarrow$ ***P**$_i$* must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

- Snapshot at time T_0:

	Max		
	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Allocation		
A	B	C
0	1	0
2	0	0
3	0	2
2	1	1
0	0	2

Available		
A	B	C
3	3	2

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

	Work		
	A	B	C
	3	3	2
P_1	2	0	0
	5	3	2
P_3	2	1	1
	7	4	3
P_4	0	0	2
	7	4	5
P_2	3	0	2
	10	4	7
P_0	0	1	0
	10	5	7

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example of Banker's Algorithm (cont.)

- P_1 requests (1,0,2)
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2

	Max		
	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Allocation		
A	B	C
0	1	0
3	0	2
3	0	2
2	1	1
0	0	2

Available		
A	B	C
2	3	0

	Need		
	A	B	C
P_0	7	4	3
P_1	0	2	0
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

W	O	R	K
	A	B	C
	2	3	0

- Executing safety algorithm shows that sequence < ***P_1*, *P_3*, *P_4*, *P_0*, *P_2***> satisfies safety requirement. Hence, grant request of P_1.
- Can request for (3,3,0) by ***P_4*** be granted?
- Can request for (0,2,0) by ***P_0*** be granted?

Example of Banker's Algorithm (cont.)

- Instead of P_1's request, P_4 requests (3,3,0)
- Check that Request $\leq$ Available (that is, $(3,3,0) \leq (3,3,2) \Rightarrow$ true)

	Max		
	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Allocation		
A	B	C
0	1	0
2	0	0
3	0	2
2	1	1
3	3	2

	Available		
	A	B	C
	3	3	2
-	3	3	0
	0	0	2

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	1	0	1

- State will be unsafe since none of the remaining need of ***P_0*, …, *P_4*** can be satisfied with the available. Hence, request for (3,3,0) by ***P_4*** should not be granted!
- Can request for (0,2,0) by ***P_0*** be granted?

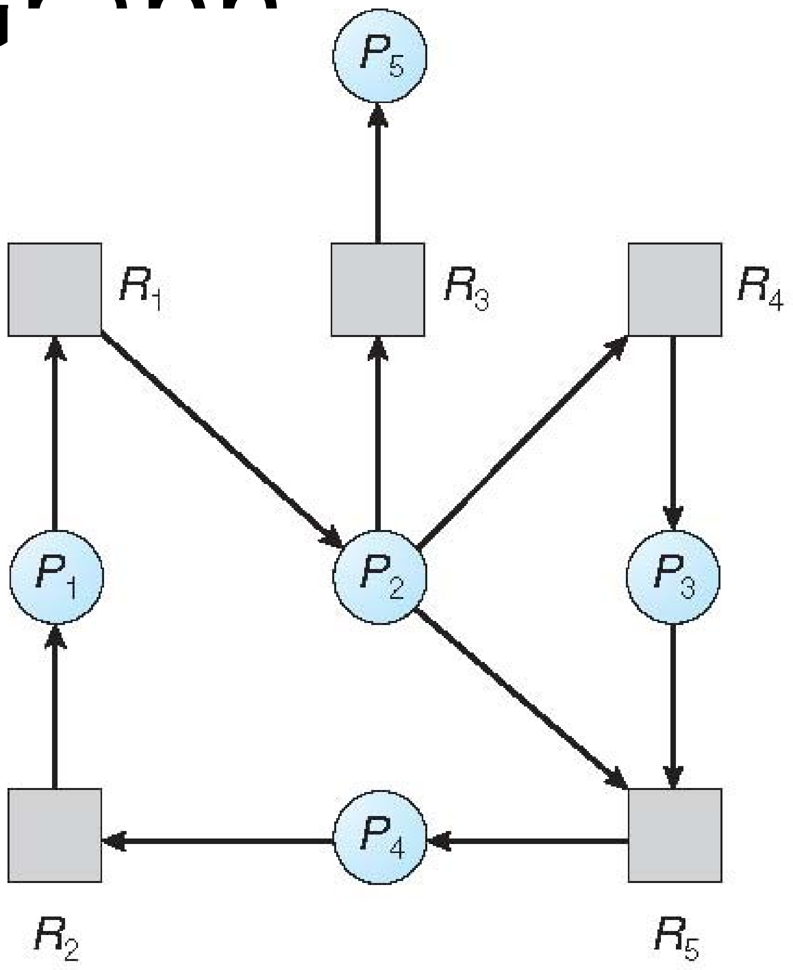
Deadlock Detection

- Allow system to enter deadlock state
- Need
  - Detection algorithm
  - Recovery scheme

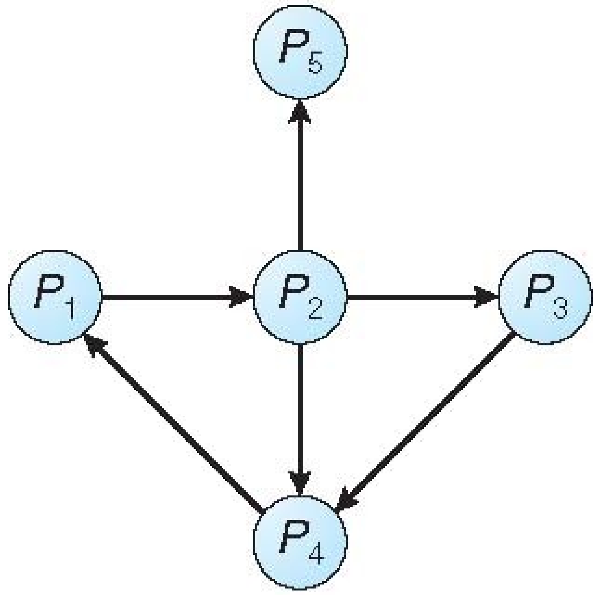
System with Single Instance of Every Resource Type

- Maintain wait-for graph
  - All vertices are processes
  - $\boldsymbol{P_i \rightarrow P_j}$ in graph if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$
- Periodically invoke an algorithm that searches for a cycle in the wait-for graph. If a cycle is found, there exists a deadlock
- An algorithm to detect a cycle in a graph with n vertices requires an order of $\boldsymbol{n^2}$ operations because a directed graph with n vertices can have up to n(n-1) edges, and one may have to traverse all edges to find a cycle

Resource Allocation Graph vs. Wait-for Graph



(a)



(b)

Resource Allocation Graph Corresponding Wait-for Graph

System with Multiple Instances of a Resource Type

Data structures for deadlock detection

- **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.

Deadlock Detection (Graph Reduction) Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if ***Request****$_i$* **≠ 0**, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true and Work = Work + Allocation****$_i$*
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***, $1 \leq$ ***i*** $\leq$ ***n***, then the system is in deadlock state.
   Moreover, if ***Finish*[*i*] == *false***, then ***P****$_i$* is deadlocked

Algorithm requires an order of O(mxn^2) operations to detect whether the system is in deadlocked state

Example of Deadlock Detection Algorithm

- Five processes ***P*₀** through ***P*₄**; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time $\boldsymbol{T_0}$:

	Allocation		
	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2
Sum	7	2	6

Request		
A	B	C
0	0	0
2	0	2
0	0	0
1	0	0
0	0	2

Available		
A	B	C
0	0	0

	Finished?	Work		
		A	B	C
		0	0	0
P_0	TRUE	0	1	0
		0	1	0
P_2	TRUE	3	0	3
		3	1	3
P_3	TRUE	2	1	1
		5	2	4
P_1	TRUE	2	0	0
		7	2	4
P_4	TRUE	0	0	2
		7	2	6

Request ≤ Work?

- Sequence <$\boldsymbol{P_0, P_2, P_3, P_1, P_4}$> will result in ***Finish[i] = true*** for all ***i***, **no deadlock**

Example of Deadlock Detection (cont.)

- ***P_2* requests an additional instance of type *C***

	Allocation		
	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2
Sum	7	2	6

Request		
A	B	C
0	0	0
2	0	2
0	0	1
1	0	0
0	0	2

Available		
A	B	C
0	0	0

	Finished?	Work		
		A	B	C
		0	0	0
P_0	TRUE	0	1	0
		0	1	0
P_1	FALSE			
P_2	FALSE			
P_3	FALSE			
P_4	FALSE			

Request ≤ Work?

- State of system?
  - Can reclaim resources held by process ***P_0***, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes ***P_1, P_2, P_3***, and ***P_4***

Complexity of the Graph Reduction Algorithm

- To find the first row in the Request matrix with the request amount ≤ Work, one may need to traverse through all n rows.
- Comparing a row of the Request matrix with the Work vector of size m, an amount of computation proportional to m is needed.
- Therefore, finding the first row that satisfies Request ≤ Work requires up to an amount of computation proportional to nm, and finding the second such row requires (n-1)m, and so on.

Total amount of work (in the worst case)

$= nm+(n-1)m+(n-2)m+\ldots+2m+1m$

$=[n+(n-1)+(n-2)+\ldots+2+1] \ast m$

$=[n(n+1)/2] \ast m$

$=O(n^2 \ast m)$

When to Apply Deadlock Detection

- 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 or killed?
    - 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.

Deadlock Recovery: Process Termination

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

Deadlock Recovery: Rollback

- Selecting a victim – minimize cost
- Rollback – return to some safe state where a resource is released
- Starvation – the same process may always be picked as victim, can include number of rollback in cost factor