Chapter 10

Deadlock

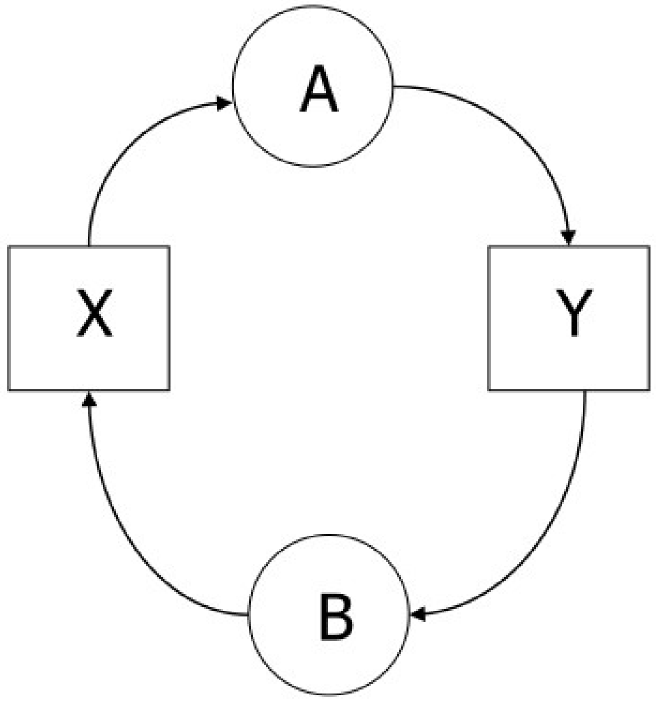
What is Deadlock?

- Two or more entities need a resource to make progress, but will never get that resource
- Examples from everyday life:
  - Gridlock of cars in a city
  - Class scheduling: Two students want to swap sections of a course, but each section is currently full.
- Examples from Operating Systems:
  - Two processes spool output to disk before either finishes, and all free disk space is exhausted
  - Two processes consume all memory buffers before either finishes

Deadlock Illustration

- A requests & receives X
- B requests & receives Y
- A requests Y and blocks
- B requests X and blocks

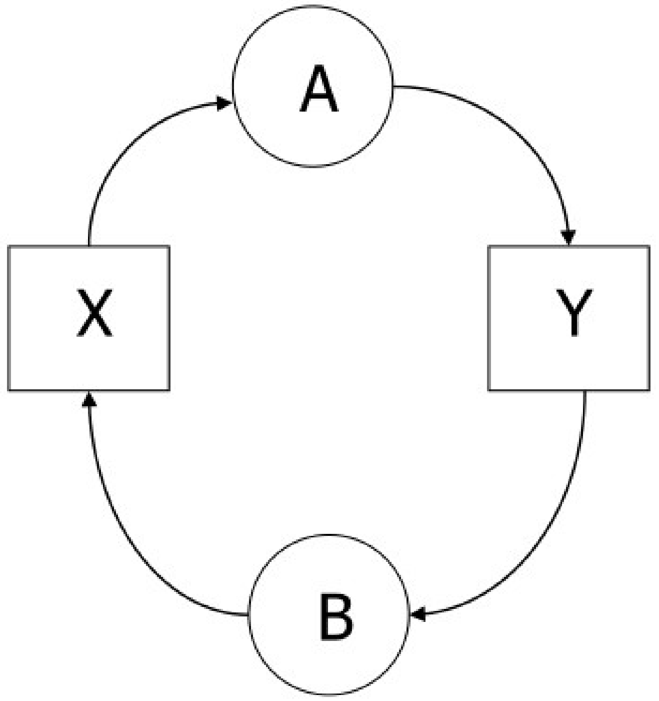
The “Deadly Embrace”



Deadlock Illustration

- A requests & receives X
- B requests & receives Y
- A requests Y and blocks
- B requests X and blocks

The "Deadly Embrace"



Terminology …

- Indefinite postponement
  - Job is continually denied resources needed to make progress

Example: High priority processes keep CPU busy 100% of time, thereby denying CPU to low priority processes

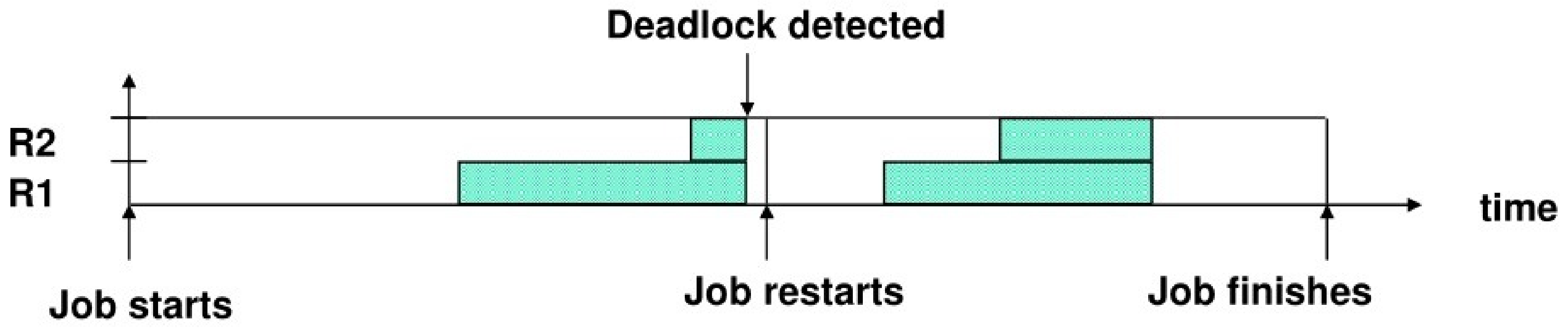
Terminology …

- Indefinite postponement
  - Job is continually denied resources needed to make progress

Example: High priority processes keep CPU busy 100% of time, thereby denying CPU to low priority processes

Three Solutions to Deadlock…

#3: Mr./Ms. Liberal ***(Detection/Recovery)***

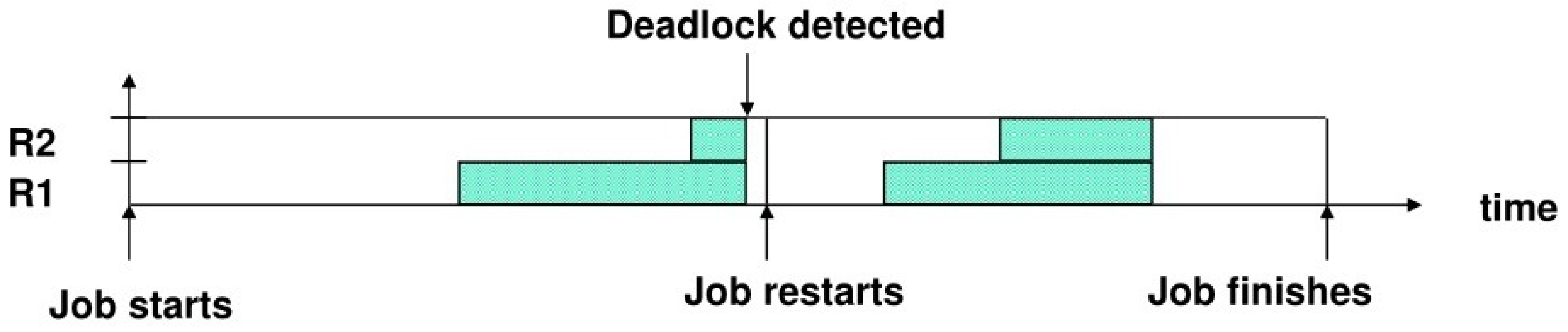


"If it's free, use it -- why wait?"

Good resource utilization, minimal process wait time
Until deadlock occurs….

Three Solutions to Deadlock…

#3: Mr./Ms. Liberal ***(Detection/Recovery)***

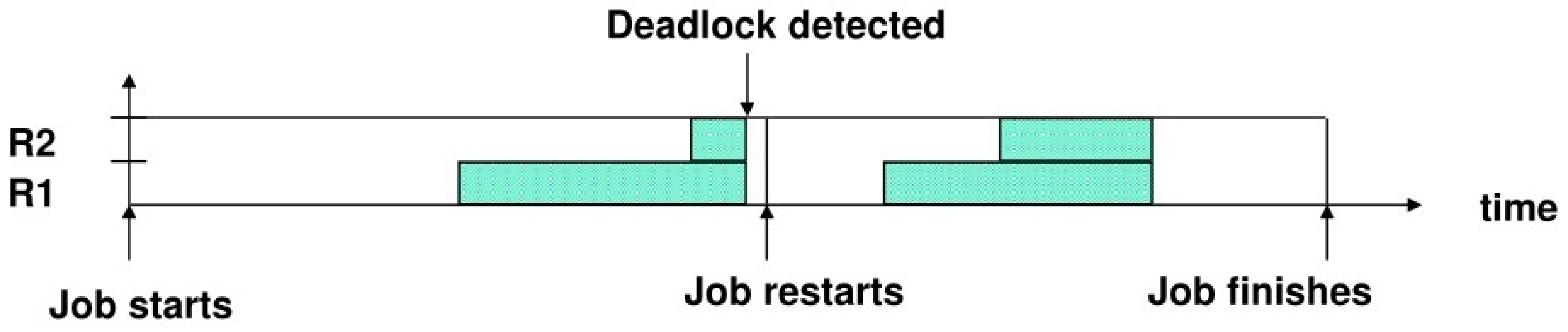


“If it’s free, use it -- why wait?”

Good resource utilization, minimal process wait time
Until deadlock occurs….

Three Solutions to Deadlock…

#3: Mr./Ms. Liberal ***(Detection/Recovery)***



"If it's free, use it -- why wait?"

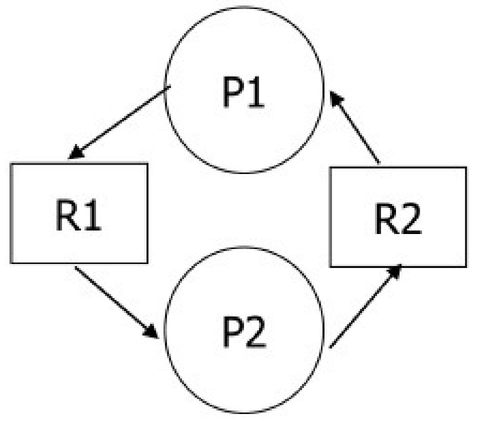
Good resource utilization, minimal process wait time
Until deadlock occurs….

Names for Three Methods on Last Slide

1) Deadlock Prevention
   - Design system so that possibility of deadlock is avoided *a priori*
2) Deadlock Avoidance
   - Design system so that if a resource request is made that *could* lead to deadlock, then block requesting process.
   - Requires knowledge of future requests by processes for resources.
3) Deadlock Detection and Recovery
   - Algorithm to detect deadlock
   - Recovery scheme

4 Necessary Conditions for Deadlock

- Mutual Exclusion
  - Non-sharable resources
- Hold and Wait
  - A process must be holding resources and waiting for others
- No pre-emption
  - Resources are released voluntarily
- Circular Wait



Deadlock Prevention …

- Prevent Circular Wait
  - Order resources and
  - Allow requests to be made only in an increasing order

Deadlock Prevention …

- Prevent Circular Wait
  - Order resources and
  - Allow requests to be made only in an increasing order

Deadlock Prevention …

- Prevent Circular Wait
  - Order resources and
  - Allow requests to be made only in an increasing order

Deadlock Prevention …

- Prevent Circular Wait
  - Order resources and
  - Allow requests to be made only in an increasing order

Preventing Circular Wait

Impose an ordering on Resources:

1	W
2	X
3	Y
4	Z

Process:	A	B	C	D	A	B	C	D
Request:	W	X	Y	Z	X	Y	Z	W

After first 4 requests:

A / W

D / Z

B / X

C / Y

Process D cannot request resource W without voluntarily releasing Z first

Problems with Linear Ordering Approach

(1) Adding a new resource that upsets ordering requires <u>all</u> code ever written for system to be modified!

(2) Resource numbering affects efficiency

=> A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending sequence

Deadlock Avoidance

Unsafe State:

	Current Loan	Max Need
Process 1	8	10
Process 2	2	5
Process 3	1	3

Available = 1

Banker's Algorithm

Taken from Operating System Concepts, 6th Ed, Silberschatz, et al, 2003

- 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.

Definition: Safe State

- State of a system
  - An enumeration of which processes hold, are waiting for, or might request which resources
- Safe state
  - No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.

    or alternatively,
  - No process is deadlocked, and the current state will not lead to a deadlocked state

Deadlock Avoidance

Unsafe State:

	Current Loan	Max Need
Process 1	8	10
Process 2	2	5
Process 3	1	3

Available = 1

Deadlock Avoidance

Unsafe State:

	Current Loan	Max Need
Process 1	8	10
Process 2	2	5
Process 3	1	3

Available = 1

Safe to Unsafe Transition

Current state being safe does not necessarily imply future states are safe

Current Safe State:

	Current Loan	Maximum Need	
Process 1	1	4	
Process 2	4	6	
Process3	5	8	**Available = 2**

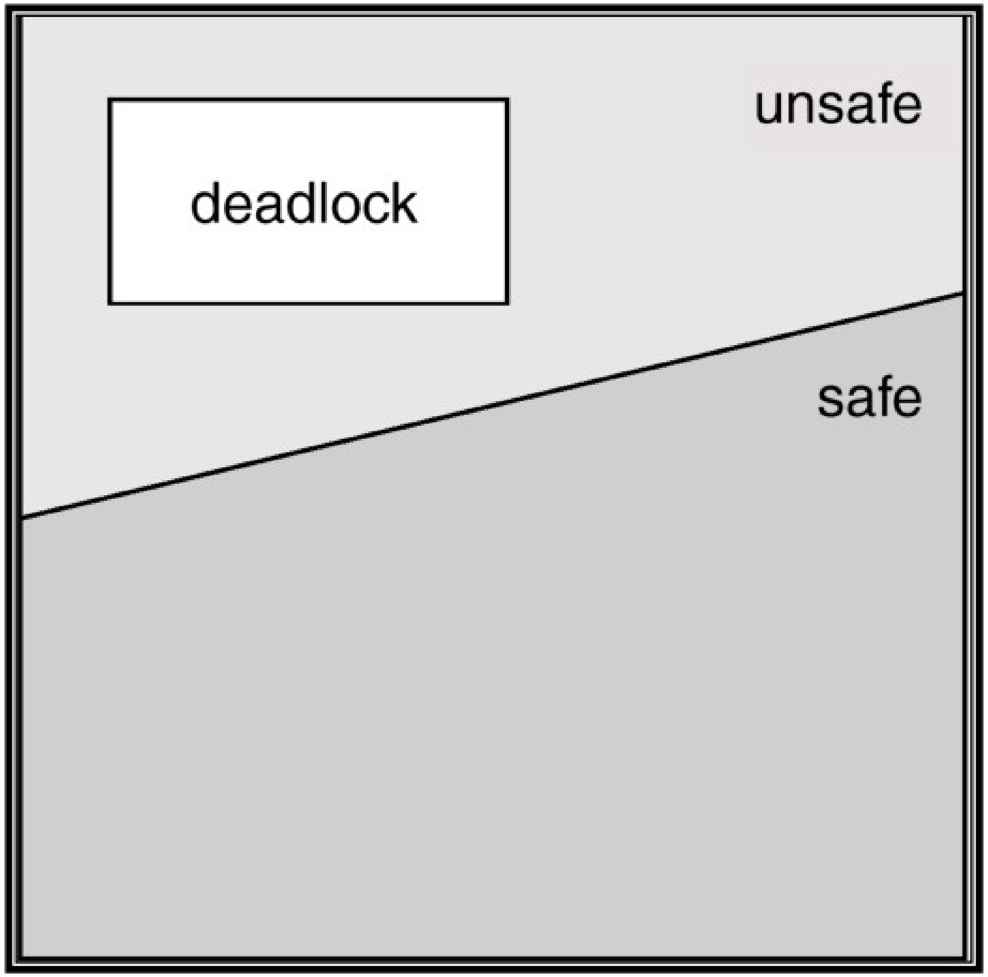
Suppose Process 3 requests and gets one more resource

	Current Loan	Maximum Need	
User1	1	4	
User2	4	6	
User3	6	8	**Available = 1**

Basic Facts

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

Safe, Unsafe , Deadlock State



Banker's Algorithm

Taken from Operating System Concepts, 6th Ed, Silberschatz, et al, 2003

- 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 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 Algorithm

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

   Work = Available

   Finish [*i*] = *false* for *i* = 1,2,3, …, *n*.

```
i=1;
while (i <= n) Do {
    if (!Finish[i]  && $Need_i$ <= Work ) {
        Finish[i] = True;
        Work = Work + $Allocation_i$;
        i  =  1;
    }
    else i++;
}
if  (Finish [i] == true for all i,) return ( SAFE )
else return ( UNSAFE );
```

Safety Algorithm

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

    Work = *Available*

    Finish [*i*] = *false* for *i* = 1,2,3, …, *n.*

```
i=1;
while (i <= n) Do {
    if (!Finish[i]  && $Need_i$ <= Work ) {
        Finish[i] = True;
        Work = Work + $Allocation_i$;
        i  =  1;
    }
    else i++;
}
if  (Finish [i] == true for all i,) return ( SAFE )
else return ( UNSAFE );
```

Resource-Request Algorithm for Process 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 $Request_i$ ⊠ $Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i$ ⊠ *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:

$$Available = Available - Request_{i};$$

$$Allocation_i = Allocation_i + Request_{i};$$

$$Need_i = Need_i - Request_{i;;}$$

- If safe ⇒ the resources are allocated to P_i.
- If unsafe ⇒ 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:

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

Example (Cont.)

- The content of the matrix. *Need* is defined to be *Max – Allocation*.

$$\begin{array}{cccc} & & \underline{\textit{Need}} & \\ & A & B & C \\ P_0 & 7 & 4 & 3 \\ P_1 & 1 & 2 & 2 \\ P_2 & 6 & 0 & 0 \\ P_3 & 0 & 1 & 1 \\ P_4 & 4 & 3 & 1 \end{array}$$

- The system is in a safe state since the sequence

$< P_1, P_3, P_0, P_2, P_4>$ satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

- Check that *Request* ☒ *Available* (that is, (1,0,2) ☒ (3,3,2) [illegible] true.

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

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

Banker's Algorithm: Summary

(+) PRO's:

- Deadlock never occurs.
- More flexible & more efficient than deadlock prevention. (Why?)

(-) CON's:

- Must know max use of each resource when job starts.

  => No truly dynamic allocation
- Process might block even though deadlock would never occur

Deadlock Detection

Allow deadlock to occur, then recognize that it exists

- Run deadlock detection algorithm whenever <u>locked</u> resource is requested
- Could also run detector in background

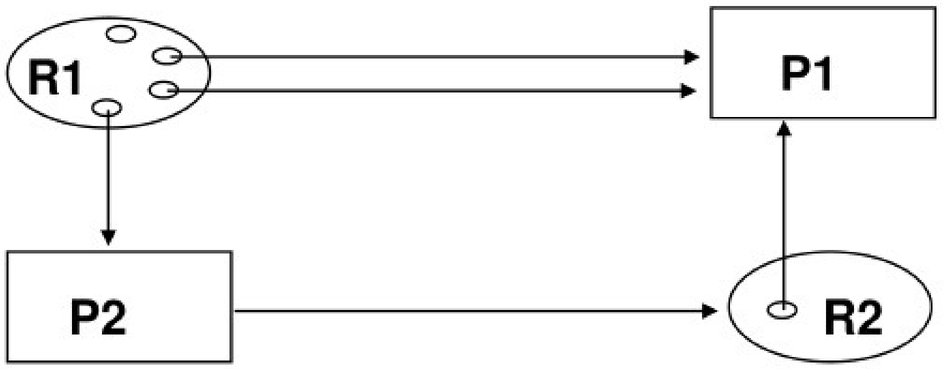
Resource Graphs…

What if there was only 2 available unit of R2 ?

?

Can deadlock occur with multiple copies of just one resource?

Resource Graphs: Example



P1 holds 2 units of R1

P1 holds 1 unit of R2

R1 has a total inventory of 4 units

P2 holds 1 unit of R1

P2 requests 1 unit of R2 (and is blocked)

Operations on Resource Graphs: An Overview

1) Process requests resources: Add arc(s)

2) Process acquires resources: Reverse arc(s)

3) Process releases resources: Delete arc(s)

Graph Reductions

- A graph is <u>reduced</u> by performing operations 2 and 3 (reverse, delete arc)
- A graph is <u>completely reducible</u> if there exists a sequence of reductions that reduce the graph to a set of isolated nodes
- A process P is <u>not</u> deadlocked if and only if there exists a sequence of reductions that leave P unblocked
- If a graph is completely reducible, then the system state it represents is not deadlocked

Operations on Resource Graphs: Details …

3) P releases resources (<u>Delete arc</u>)

<u>Precondition:</u>

- P must have no outstanding requests
- P can release any subset of resources that it holds

<u>Operation:</u>

- Delete one arc directed away from resource for each released resource

Operations on Resource Graphs: Details …

3) P releases resources (<u>Delete arc</u>)

<u>Precondition:</u>

- P must have no outstanding requests
- P can release any subset of resources that it holds

<u>Operation:</u>

- Delete one arc directed away from resource for each released resource

Resource Graphs…

What if there was only 2 available unit of R2 ?

?

Can deadlock occur with multiple copies of just one resource?

Resource Graphs…

What if there was only 2 available unit of R2 ?

?

Can deadlock occur with multiple copies of just one resource?

Resource Graphs…

What if there was only 2 available unit of R2 ?

?

Can deadlock occur with multiple copies of just one resource?

Recovering from Deadlock

Once deadlock has been detected, the system must be restored to a non-deadlocked state

1) Kill one or more processes

- Might consider priority, time left, etc. to determine order of elimination

2) Preempt resources

- Preempted processes must <u>rollback</u>
- Must keep ongoing information about running processes