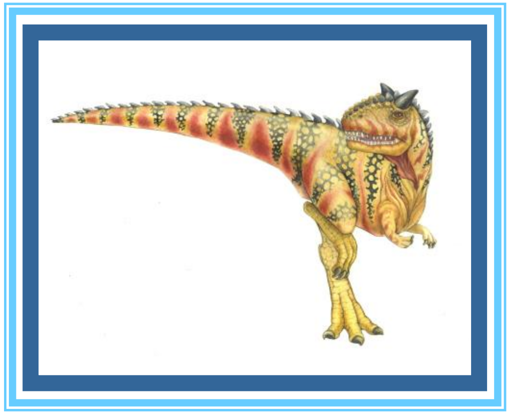
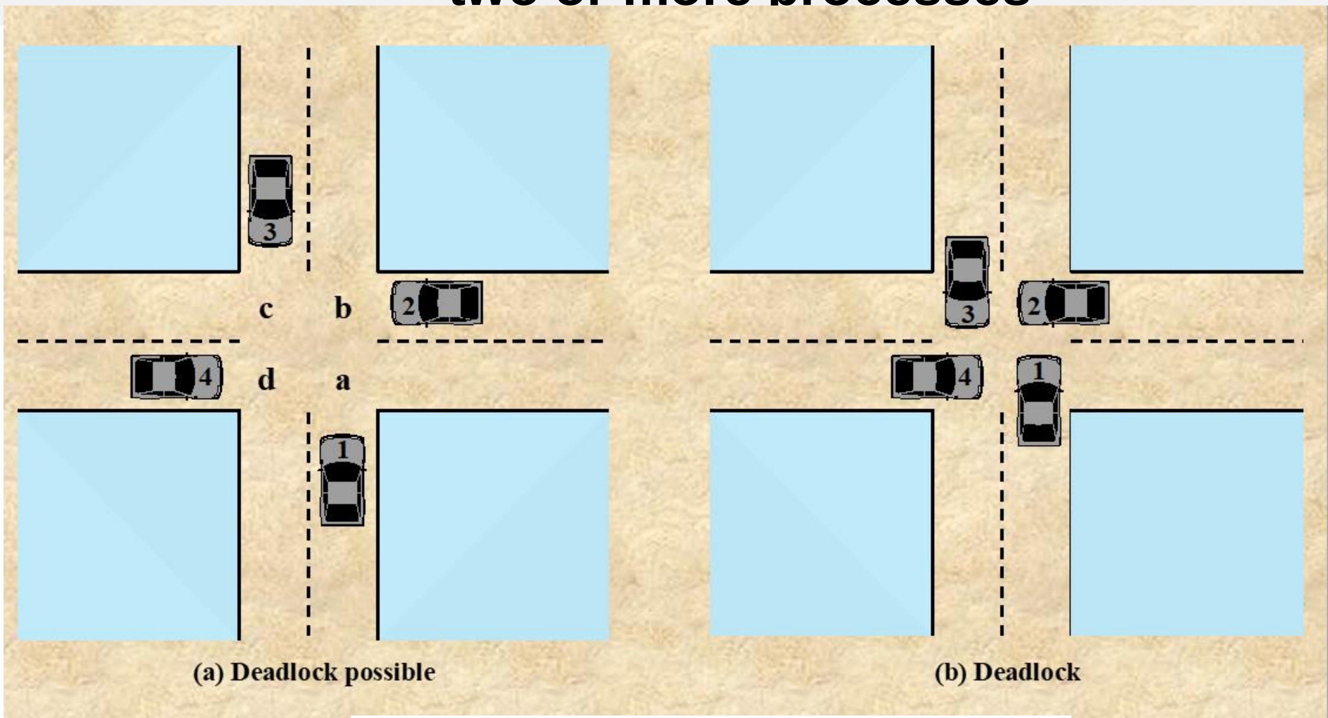
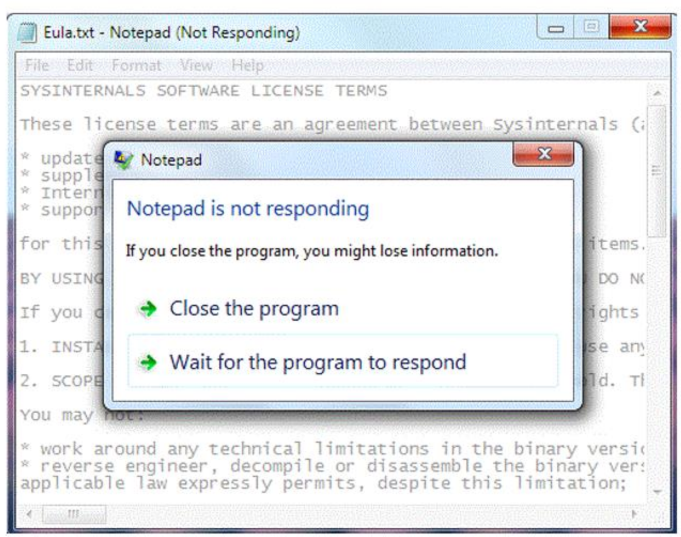
Chapter 8: Deadlocks



Deadlock: All deadlocks involve conflicting needs for resources by two or more processes





Chapter 8: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  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 Characterization

Deadlock can arise if four conditions hold 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 $\{P_0, P_1, \ldots, P_n\}$ 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_n is waiting for a resource that is held by P_0.

Deadlock with Mutex Locks

- **file locks:** When two processes lock different files and attempt to lock each other's file.
- **process synchronization (wait(), mutex())**: Multiple processes or threads waiting on resources in circular dependencies can lead to deadlock.
- **memory management (malloc())**: If memory resources are exhausted and processes need additional memory, deadlocks can occur.

- Difficult to detect and prevent, but techniques like **deadlock detection** algorithms, **resource allocation graphs**, or implementing protocols like the **Banker's Algorithm** can help manage or avoid deadlocks.

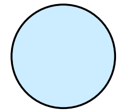
Resource-Allocation Graph

A set of vertices ***V*** and a set of edges ***E***.

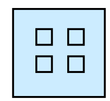
- **V** is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

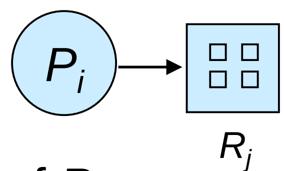
- Process



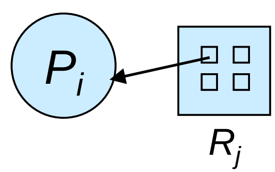
- Resource Type with 4 instances



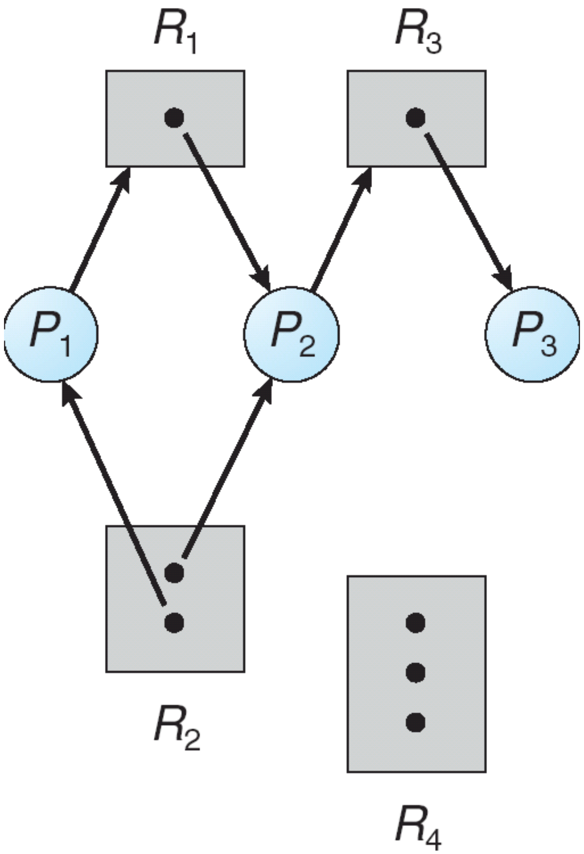
- P_i requests instance of R_j



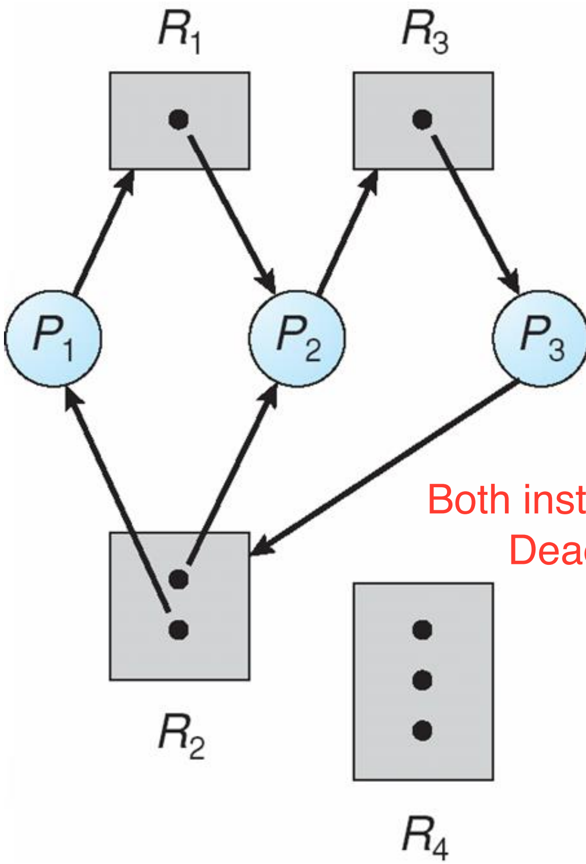
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

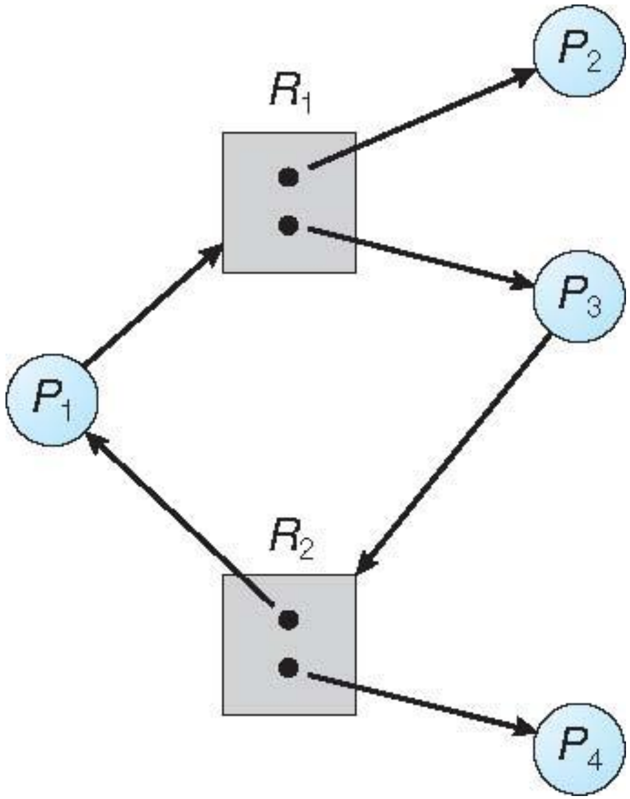


Resource Allocation Graph With A Deadlock



Both instances in R2 are held by P1 and P2.
Deadlock because P3 requests R2,
but no resources available.

Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - 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
  - Deadlock avoidance
- 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 (e.g., read-only files); must hold for non-sharable resources
  - **Example**: If multiple processes want to read from the same file, they can all do so simultaneously without conflict (no mutual exclusion is needed). However, if a process wants to write to the file, mutual exclusion is necessary since writing can’t be done by multiple processes at the same time
  - **Prevention Rule**: Allow sharing of resources when possible (e.g., multiple readers), but enforce mutual exclusion for non-sharable resources (e.g., writing to a file).

Deadlock Prevention

Restrain the ways request can be made

- **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 allocated to it.
  - Low resource utilization; starvation possible:
    - It wastes resources because they are locked up by processes that don't need them right away.
- **Example**: a process wants to print a document and needs both a printer and a scanner. Acquire the scanner first and wait for the printer to become available, possible deadlock if another process holds the printer and waits for the scanner. To prevent this, the process must either:
  - Request both the scanner and printer **at the same time** before starting, or
  - Release the scanner if the printer is not available and retry later.
- **Prevention Rule**: A process must request all its resources at once or hold none while waiting for additional resources.

Deadlock Prevention (Cont.)

- **No 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
  - **Example:** Process holds a **printer** and requests a **scanner**. If the scanner is unavailable, the system will force the process to release the printer. The process will then be put back in a waiting queue until both the **printer** and **scanner** are available at the same time.

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - **Example**: Suppose we have two resources: **R1 (printer)** and **R2 (scanner)**. To avoid deadlock, we assign a total ordering, such as R1 < R2. A process that needs both the printer and the scanner must always request the **printer (R1)** first and then the **scanner (R2)**. If a process is holding the scanner and then tries to request the printer, it will violate the rule, so it won't be allowed to proceed.
  - May avoid any possibility of a circular wait:
    - It may require processes to request resources they don't need yet, which can lead to **low resource utilization**. Also requires careful planning of resource ordering to avoid inefficiencies.

Deadlock Example

```
import threading
import time

# Define two semaphores (representing resources)
resource1 = threading.Semaphore(1)
resource2 = threading.Semaphore(1)

# Thread 1 function
def process_1():
    print("Process 1: Trying to acquire Resource 1...")
    resource1.acquire()
    print("Process 1: Acquired Resource 1")

    # Simulate some work
    time.sleep(1)

    print("Process 1: Trying to acquire Resource 2...")
    resource2.acquire()  # Deadlock happens here
    print("Process 1: Acquired Resource 2")

    # Release resources
    resource2.release()
    resource1.release()
```

```python
# Thread 2 function
def process_2():
    print("Process 2: Trying to acquire Resource 2...")
    resource2.acquire()
    print("Process 2: Acquired Resource 2")

    # Simulate some work
    time.sleep(1)

    print("Process 2: Trying to acquire Resource 1...")
    resource1.acquire()   # Deadlock happens here
    print("Process 2: Acquired Resource 1")

    # Release resources
    resource1.release()
    resource2.release()

# Create two threads
thread1 = threading.Thread(target=process_1)
thread2 = threading.Thread(target=process_2)

# Start both threads
thread1.start()
thread2.start()

# Join the threads to ensure the main program waits for them to finish
thread1.join()
thread2.join()
```

```
Process 1: Trying to acquire Resource 1...Process 2: Trying to acquire Resource 2...

Process 1: Acquired Resource 1Process 2: Acquired Resource 2

Process 1: Trying to acquire Resource 2...Process 2: Trying to acquire Resource 1...
```

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**

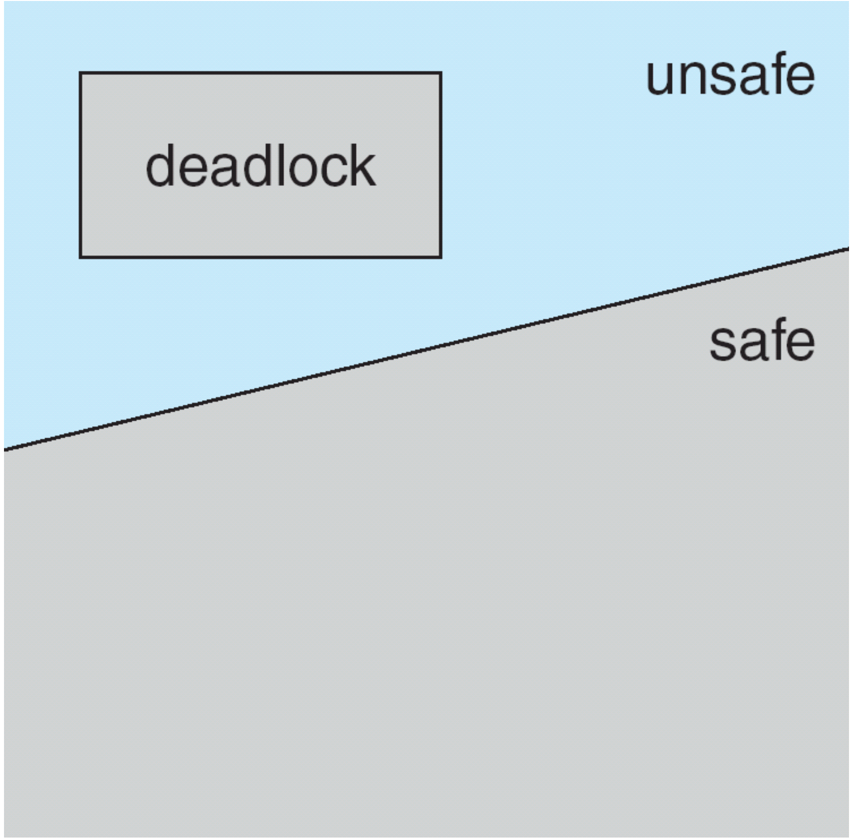
Safe State

- When a process requests an available resource, system must decide if immediate 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 in 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 $j < I$
- 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 $\Rightarrow$ 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



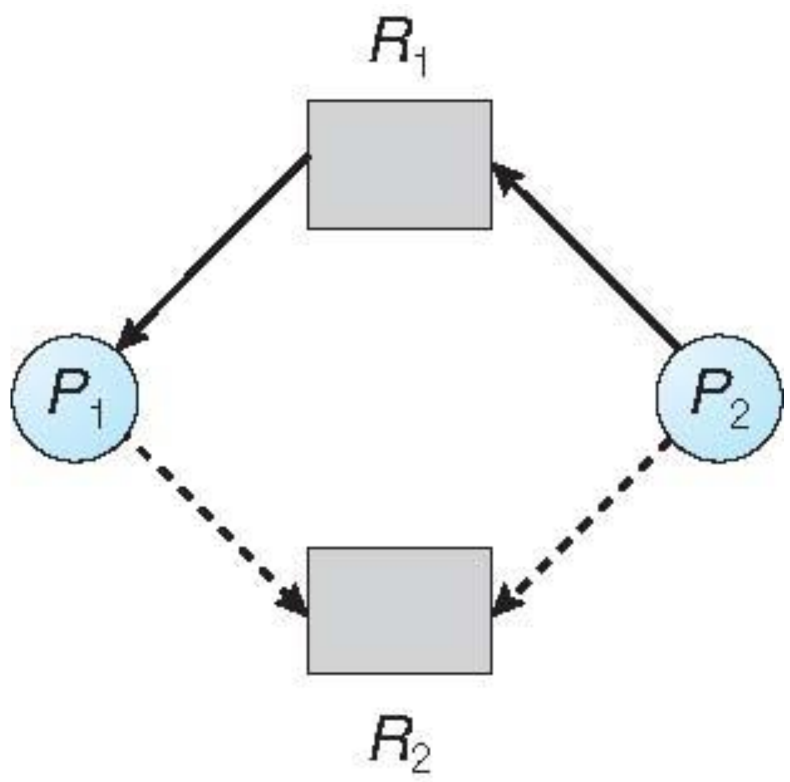
Avoidance Algorithms

- **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

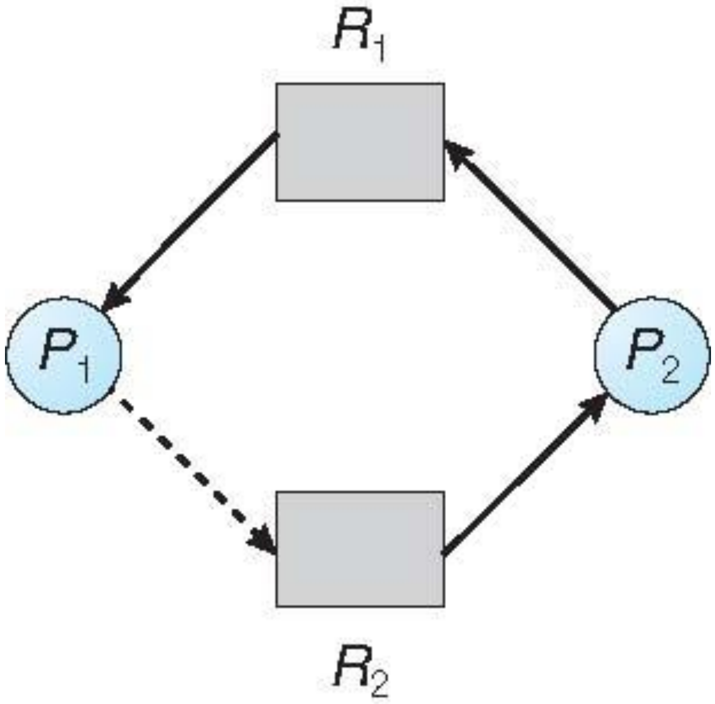
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a **dashed line**
- Claim edge converts to **request** edge when a process requests a resource
- Request edge converted to an **assignment** edge when the resource is allocated to the process
-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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does **not result** in the formation of a **cycle** in the resource allocation graph



Resource Allocation Graph - Algorithm

Draw the "Avoidance Graph"

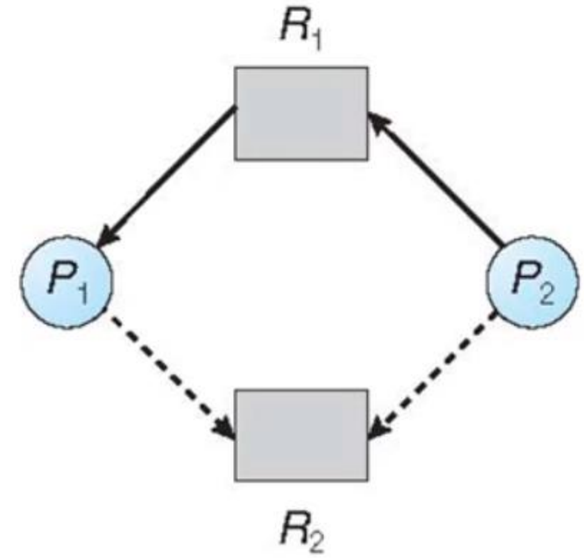
Suppose that process P_i requests a resource R_j

Convert the claim edge ($P_i \rightarrow R_j$) to request edge ($P_i \rightarrow R_j$)

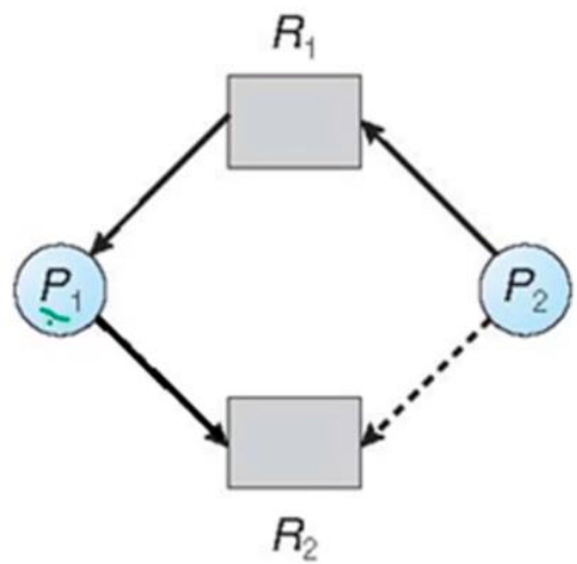
Temporarily convert this request edge ($P_i \rightarrow R_j$) to assignment edge

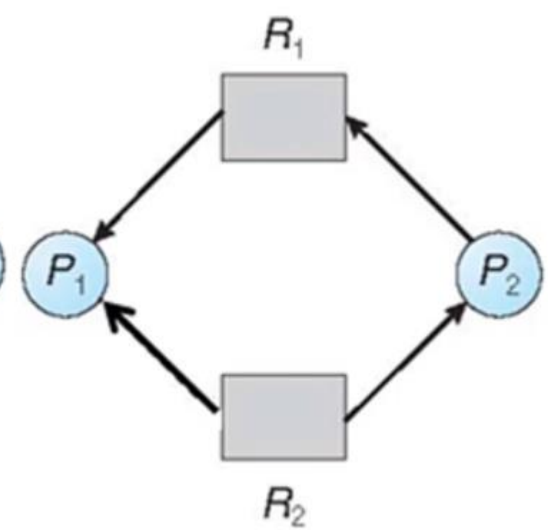
If still there is **"no cycle"** formed; **allocate** the resource

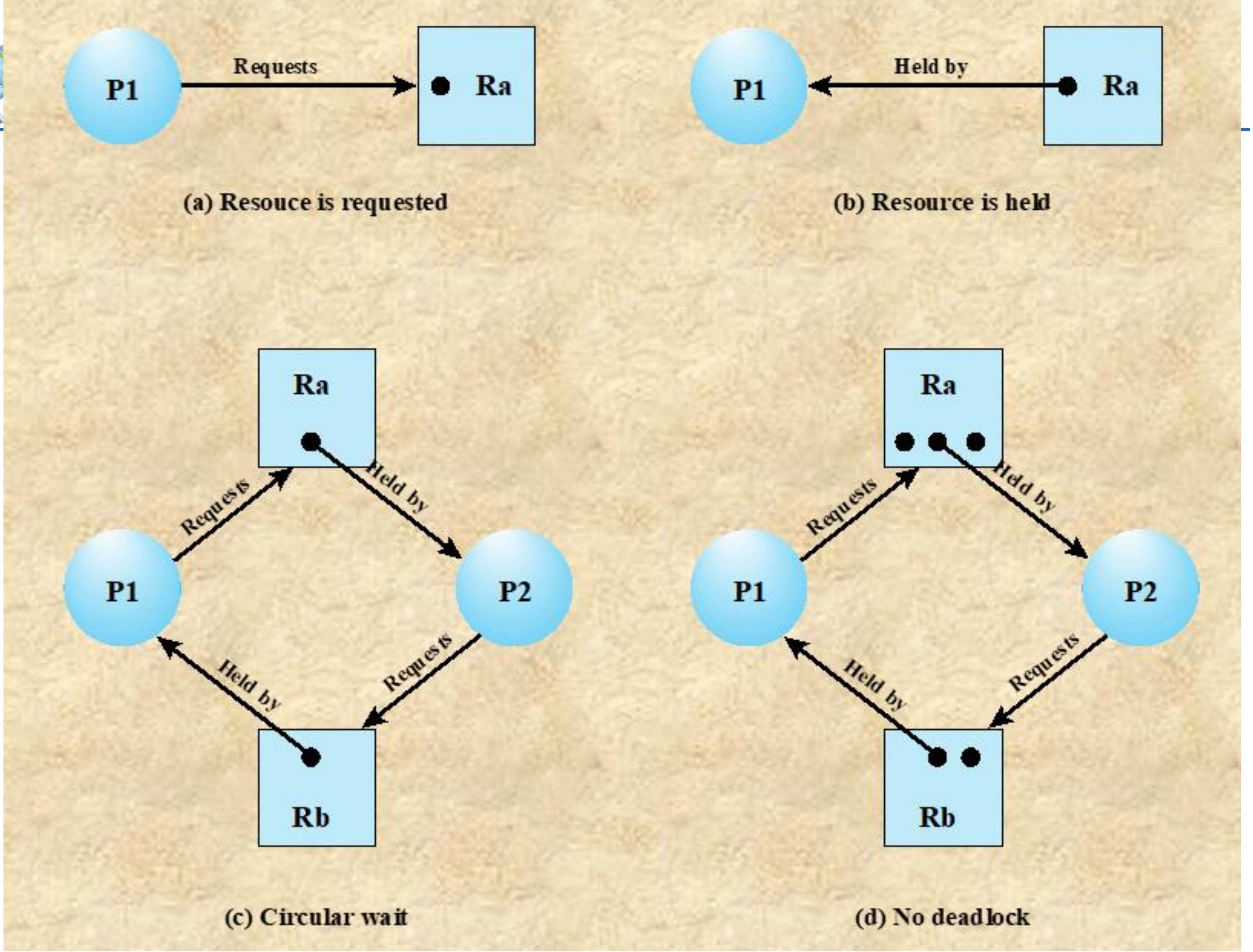
Otherwise revert the change



- P1 requested for R2
- **Safe State**
- **Request Accepted**







P1
Requests
Ra
(a) Resouce is requested
P1
Held by
Ra
(b) Resource is held
Ra
Requests
Held by
P1
P2
Held by
Requests
Rb
(c) Circular wait
Ra
Requests
Held by
P1
P2
Held by
Requests
Rb
(d) No deadlock

Banker's Algorithm

- Multiple instances
- 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
  - When a process (or program) gets all the resources it needs (like memory, files, or a printer), it should **use them and then give them back** within a reasonable amount of time. This ensures that other processes can also get access to those resources and prevents them from waiting forever.

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**: $n \times m$ **matrix**. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ **matrix**. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ **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 an ***i*** such that both:

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

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

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

3. ***Work*** **=** ***Work*** **+** $\boldsymbol{Allocation_i}$
   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 Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

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

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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* (5instances), 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 01

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   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.

After each process finishes,
update Available and check from P0 again.

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C

Finish

0	1	2	3	4

Example 01

Safe Sequence

< P1>

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. *Finish*[*i*] == *false*
   b. $Need_i \leq Work$

   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.

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
5	3	2

Finish

0	1	2	3	4
F	T	F	F	F

Example 01

Safe Sequence

< P1, P3>

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n-1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   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.

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
7	4	3

Finish

0	1	2	3	4
F	T	F	T	F

Example 01

Safe Sequence

< P1, P3, P0>

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   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.

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
7	5	3

Finish

0	1	2	3	4
T	T	F	T	F

Example 01

Safe Sequence

< P1, P3, P0, P2>

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \le Work$

   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.

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
10	5	5

Finish

0	1	2	3	4
T	T	F	T	F

Example 01

Safe Sequence

< P1, P3, P0, P2, P4>

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. *Finish*[*i*] == *false*
   b. $Need_i \le Work$

   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.

Safe State

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
10	5	7

Finish

0	1	2	3	4
T	T	T	T	T

Example 02

P_1 requests (1, 0, 2)

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

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

Safe Sequence

< >

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	~~3 3 2~~ 2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C

Finish

0	1	2	3	4

P_1	2 0 0	3 2 2	1 2 2

Example 02

P_1 requests (1, 0, 2)

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   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.

Safe Sequence

< >

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
2	3	0

Finish

0	1	2	3	4
F	F	F	F	F

Example 02

P_1 requests (1, 0, 2)

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   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.

Safe Sequence

< P1,>

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
5	3	2

Finish

0	1	2	3	4
F	T	F	F	F

Example 02

P_1 requests (1, 0, 2)

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n-1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   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.

Safe Sequence

< P1, P3,>

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
7	4	3

Finish

0	1	2	3	4
F	T	F	T	F

Example 02

P_1 requests (1, 0, 2)

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. *Finish*[*i*] == *false*
   b. $Need_i \le Work$

   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.

Safe Sequence

< P1, P3, P4,>

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
7	4	5

Finish

0	1	2	3	4
F	T	F	T	T

Example 02

P_1 requests (1, 0, 2)

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available* and *Finish*[*i*] = *false* for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. *Finish*[*i*] == *false*
   b. $Need_i \le Work$

   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.

Safe Sequence

< P1, P3, P4, P0, P2>

Safe State, ACCEPTED

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Work

A	B	C
10	5	7

Finish

0	1	2	3	4
T	T	T	T	T

Deadlock Detection

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

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if 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 n^2 operations, where n is the number of vertices in the graph

Consider four friends: **A**, **B**, **C**, and **D**. They invite each other in a cycle as follows:

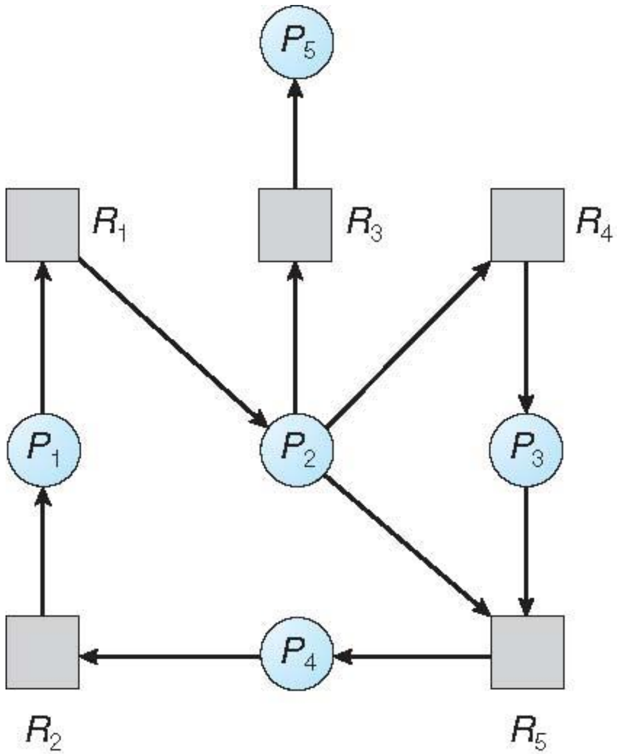
- **Friend A invites Friend B**
- **Friend B invites Friend C**
- **Friend C invites Friend D**
- **Friend D invites Friend A**

This forms a cycle: **A → B → C → D → A**.

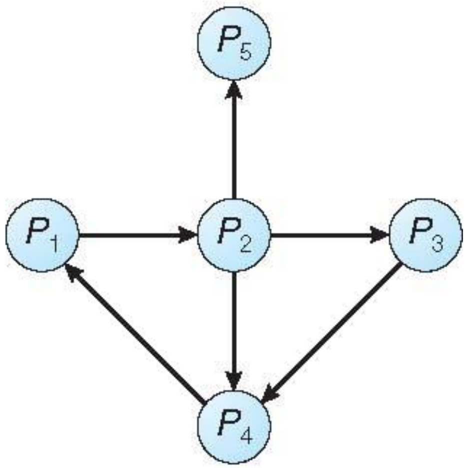
Adjacency Matrix Representation:

	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	1
D	1	0	0	0

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph   Corresponding 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.

Algorithm 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.
  - The algorithm can see the "traffic jam" (the cycle), but it can't point to a single process that caused it.

Recovery from Deadlock: Process Termination

- Abort **ALL** deadlocked processes
- Abort **ONE** process at a time until the deadlock cycle is eliminated
- <u>In which order should we choose to abort?</u>
  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?
     - Interactive processes require continuous interaction with the user, and they need to respond quickly.
     - Batch processes execute without the need for user interaction and typically handle tasks that don't require real-time responses.

Recovery from Deadlock: 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

End of Chapter 8