Deadlocks

Definition

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This permanenr situation is called a deadlock.

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.
  - If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request.
    - If it does not, then the instances are not identical, and the resource type classes have not been defined properly.
  - The number of resources requested may not exceed the total number of resources available in the system.

System Model

- Each process utilizes a resource as follows:
  - **Request:** If the request cannot be granted immediately, the requesting process must wait until it can acquire the resource
  - **Use:** The process can operate on the resource
  - **Release:** The process releases the resource
- A mutex lock may be used to protect critical sections of code i.e., a thread acquires the lock before entering a critical section and releases it upon exiting the critical section.

Mutex locks

```
#include <pthread.h>
pthread_mutex_t mutex;
/* create the mutex lock */
pthread_mutex_init(&mutex,NULL)
   ;
 /* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

We also have pthread_mutex_trylock(&mutex)

If the mutex is available, pthread_mutex_trylock locks the mutex and returns 0.

If the mutex is not available (that is, if it is locked by another thread) pthread_mutex_trylock returns a nonzero error code (EBUSY).

Order of mutexes

```
proc1( ) {
pthread_mutex_lock(&m1);
/* use object 1 */
pthread_mutex_lock(&m2);
/* use objects 1 and 2 */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
}
--------------------------------------------
proc2( ) {
while (1) {
pthread_mutex_lock(&m2);
if (!pthread_mutex_trylock(&m1))
break;
pthread_mutex_unlock(&m2);
}
/* use objects 1 and 2 */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
}
```

Deadlock Example

- The `pthread_mutex_init()` function initializes an unlocked mutex.
- Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively.
- If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`. Two mutex locks created below:

```
/* Create and initialize the mutex locks */
pthread mutex_t first_mutex;
pthread mutex_t second_mutex;
pthread mutex_init(&first_mutex,NULL);
pthread mutex_init(&second_mutex,NULL);
```

- Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks.

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Characterization

Deadlock can arise if four conditions hold <u>simultaneously</u>:

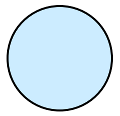
- **Mutual exclusion:** Only one process at a time can use a resource.
- **Hold and wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** □ 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.

Resource-Allocation Graph

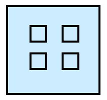
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge** – directed edge $P_i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

Resource-Allocation Graph (Cont.)

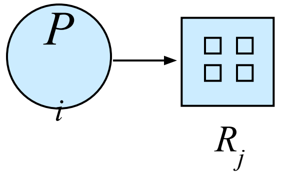
- Process



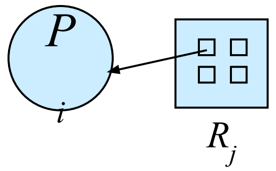
- Resource Type with 4 instances



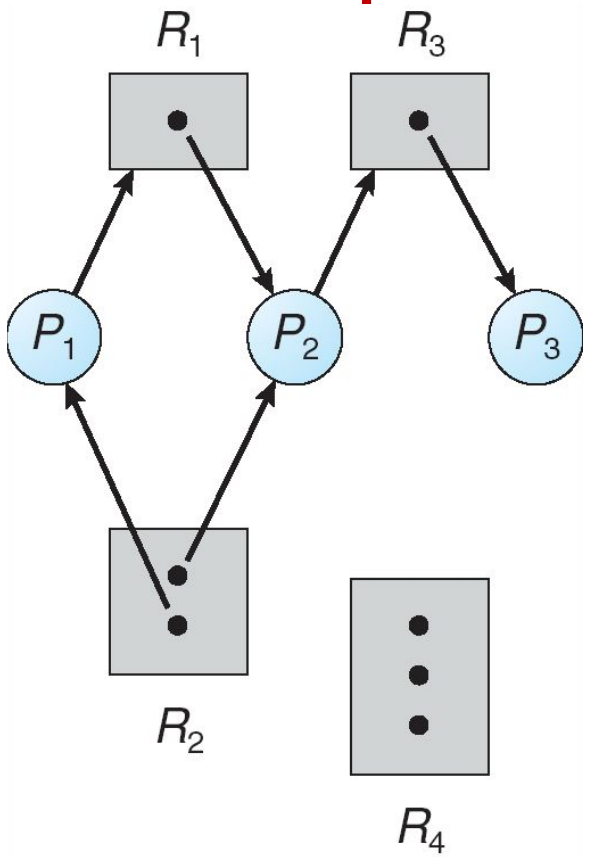
- P_i requests instance of R_j



- P_i is holding an instance of R_j

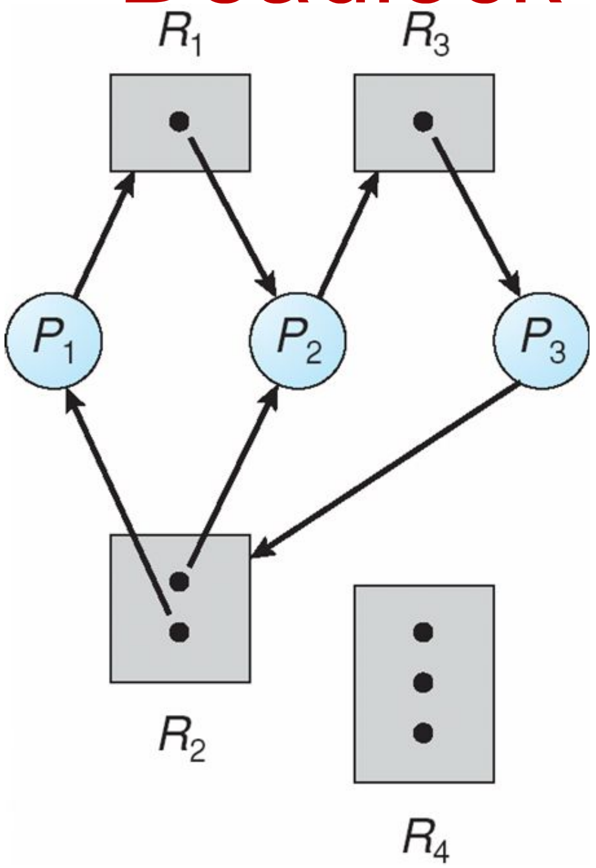


Example of a Resource Allocation Graph



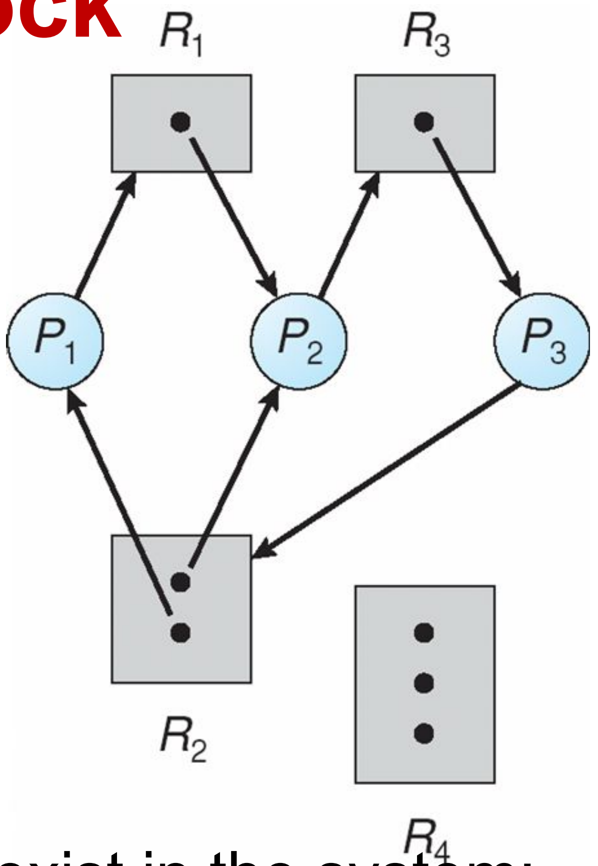
R = {R1, R2, R3, R4}
E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource Allocation Graph With A Deadlock



Given the definition of a resource-allocation graph, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Resource Allocation Graph With A Deadlock



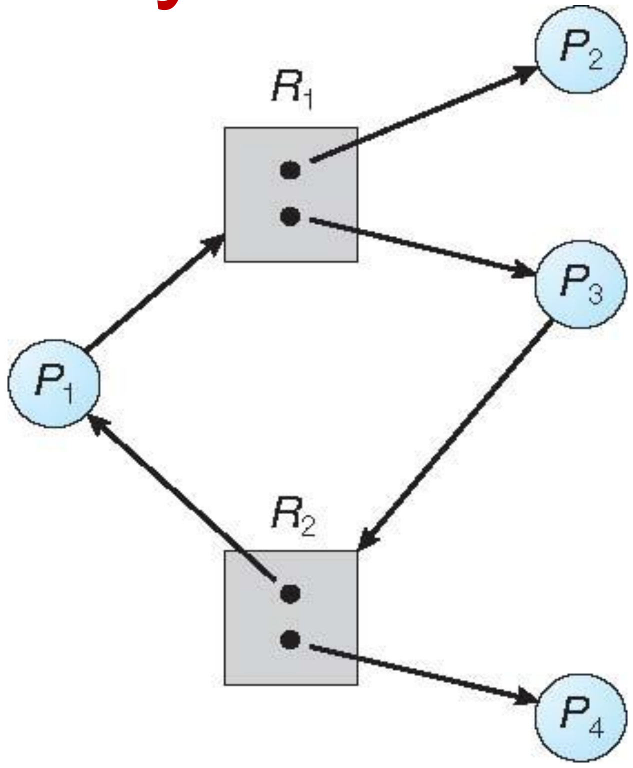
At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

Processes P1, P2, and P3 are deadlocked. Why?

Graph With A Cycle But No Deadlock



We have a cycle: P1 → R1 → P3 → R2 → P1
However, there is no deadlock.
Process P4 may release its instance of resource type R2.
That resource can then be allocated to P3, breaking the cycle.

Basic Facts

- If graph contains no cycles ⇒ no 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

1. Ensure that the system will ***never*** enter a deadlock state
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Methods for Handling Deadlocks

- Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
  - These methods prevent deadlocks by constraining how requests for resources can be made.

Methods for Handling Deadlocks

- Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
  - With this additional knowledge, the operating system can decide for each request whether or not the process should wait.
  - To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Prevention

- **Restrain the ways request can be made**

- **Mutual Exclusion** – Not required for sharable resources; must hold for nonsharable resources
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.
- For example, a mutex lock cannot be simultaneously shared by several processes.

Deadlock Prevention

- **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 <u>none</u>**
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** – How to ensure that this condition never holds
  - 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.
  - Alternatively, if a process requests some resources, we first check whether they are available.
  - If they are, we allocate them.
  - If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
  - If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
  - Otherwise wait

Deadlock Prevention (Cont.)

- **Circular Wait** – How to ensure that this condition never holds
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - Each process can request resources only in an increasing order of enumeration.

Circular Wait

- Let $R = \{R_1, R_2, ..., R_m\}$ *be the set of resource types.*
- *We* assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Formally, we define a one-to-one function $F: R \rightarrow N$, *where N is the* set of natural numbers.
- For example, if the set of resource types *R includes* tape drives, disk drives, and printers, then the function *F might be defined as* follows:

  F(tape drive) = 1

  F(disk drive) = 5

  F(printer) = 12

Circular Wait …

- A process can initially request any number of instances of a resource type —say, R_i .
- After that, the process can request instances of resource type R_j if and only if $F(R_i) < F(R_j)$.
- A process that wants to use the tape drive(1) and printer(12) at the same time must first request the tape drive and then request the printer.
- Or we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.
- If several instances of the same resource type are needed, a single request for all of them must be issued.

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
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);
}
```

Deadlock is possible if two threads simultaneously invoke the transaction() function, transposing different accounts.

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

Livelock

```
/* thread_one runs here */
void *do_work_one(void *param)
{
int done = 0;
while (!done) {
pthread_mutex_lock(&first_mutex);
if
(pthread_mutex_trylock(&second_mutex))
            {
/**
* Do some work
*/
pthread_mutex_unlock(&second_mutex);
pthread_mutex_unlock(&first_mutex);
done = 1;      }
else
pthread_mutex_unlock(&first_mutex);
    }
pthread_exit(0);
}
```

```
/* thread_two runs here */
void *do_work_two(void *param)
{
int done = 0;
while (!done) {
pthread_mutex_lock(&second_mutex);
if
   (pthread_mutex_trylock(&first_mut
   ex)) {
/*** Do some work*/
pthread_mutex_unlock(&first_mutex);
pthread_mutex_unlock(&second_mutex);
done = 1;}
else
pthread_mutex_unlock(&second_mutex);
}
pthread_exit(0);
}
```

Livelock

- It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons.
  - Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails.
- Livelock typically occurs when threads retry failing operations at the same time.
- It thus can generally be avoided by having each thread retry the failing operation at random times.

Deadlock Avoidance

Possible side effects of *preventing deadlocks* are low device utilization and reduced system throughput
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
- 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

Witness

- Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible.
  - One lock-order verifier, which works on BSD versions of UNIX such as FreeBSD, is known as witness.
  - Witness uses mutual-exclusion locks to protect critical sections
  - It works by dynamically maintaining the relationship of lock orders in a system.

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 resource requests that P_i can still make can be satisfied by currently available resources + resources **held by all the P_j, with $j < i$**

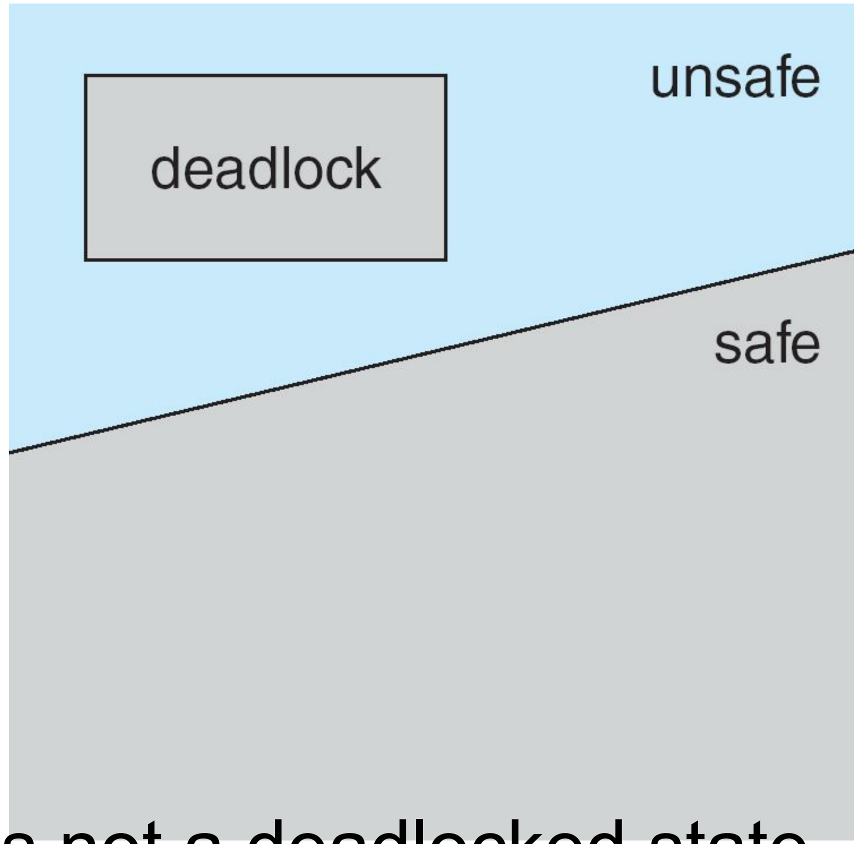
Safe State

- 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.

Safe, Unsafe, Deadlock State



- A safe state is not a deadlocked state.
- A deadlocked state is an unsafe state.
  - ✔ Not all unsafe states are deadlocks.
  - ✔ An unsafe state may lead to a deadlock.

Safe, Unsafe, Deadlock State

- As long as the state is safe, the operating system can avoid unsafe and deadlocked states.
- In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.
  - The behavior of the processes controls unsafe states.

Avoidance Algorithms

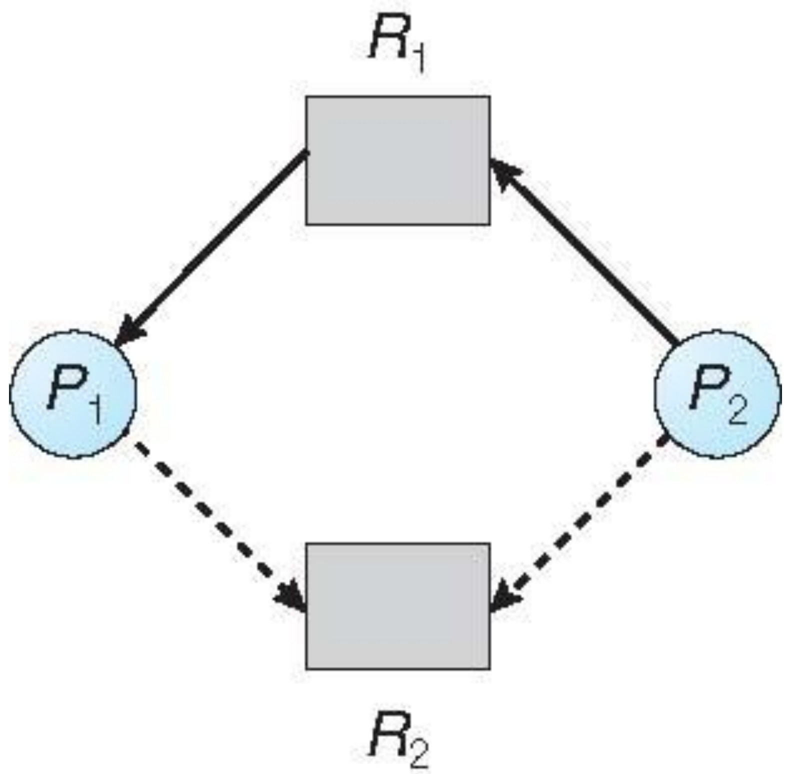
- Single instance of a resource type
  - **Use a resource-allocation graph**

- Multiple instances of a resource type
  - **Use the banker’s algorithm**

Resource-Allocation Graph Scheme

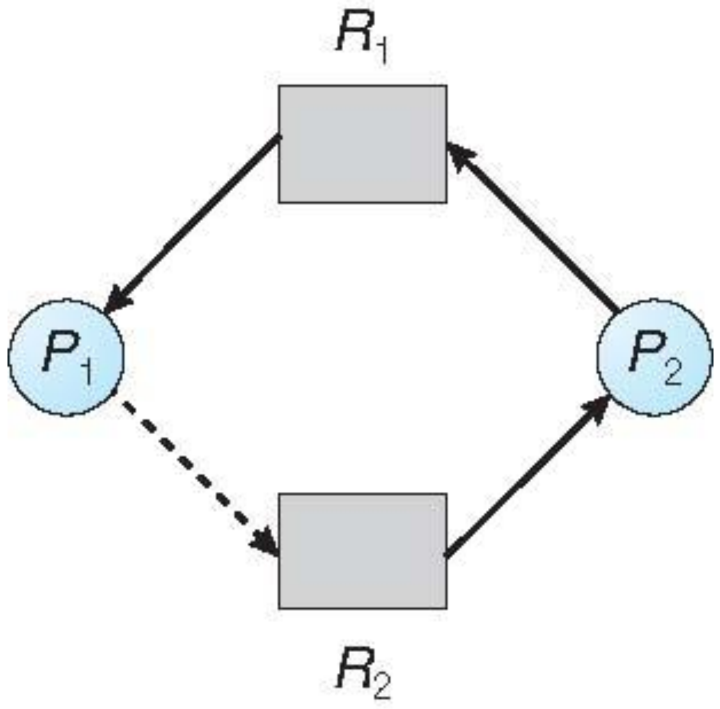
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j at sometime in future; <u>represented by a dashed line</u>
  1. Claim edge converts to request edge when a process requests a resource
  2. Request edge converted to an assignment edge when the resource is allocated 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



- Now suppose that process Pi requests resource Rj.
- Request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.
- Safety is checked by using a cycle-detection algorithm.

Unsafe State In Resource-Allocation Graph



If P1 requests R2, and P2 requests R1, then a deadlock will occur

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- 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

Banker’s Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

Banker’s Algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
  - This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

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

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* = 0, 1, …, *n*- 1**

2. Find an ***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

Notation

- Let X and Y be vectors of length n.
- We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \ldots, n$.
- For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$.
- In addition, $Y < X$ if $Y \leq X$ and $Y = X$.

Resource-Request Algorithm 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**$_i$* ≤ ***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 $\Rightarrow$ 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:

	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*** =***Max – Allocation***

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

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

Example: P_1 Requests (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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 2	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_4, P_0, P_2> 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?

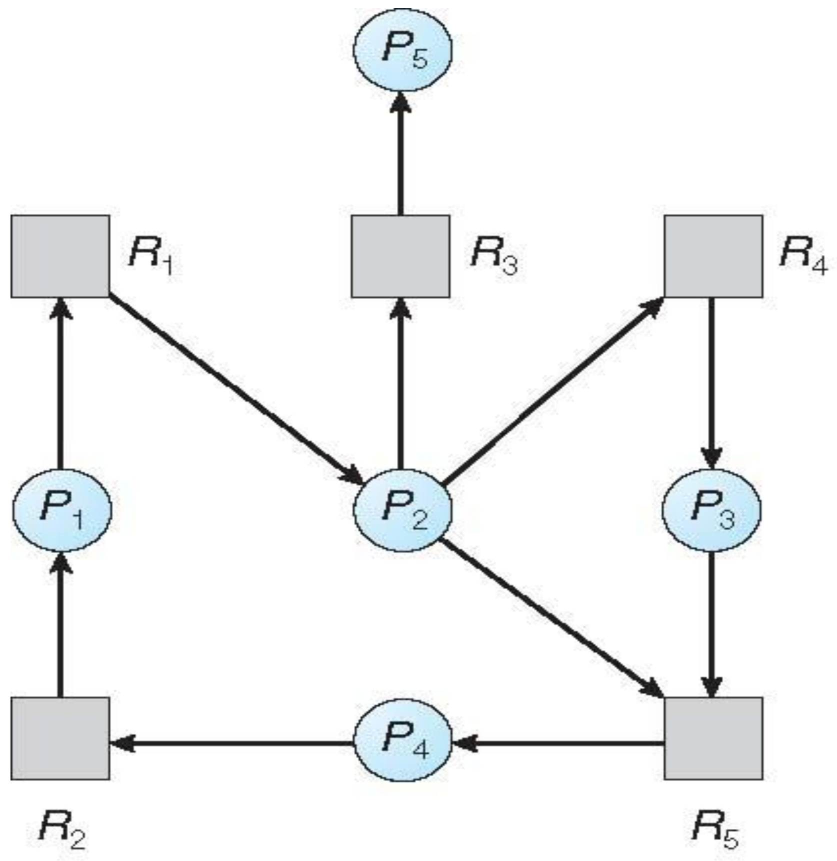
Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.
- So, deadlock Detection algorithm needed
- Recovery algorithm to recover from the deadlock needed

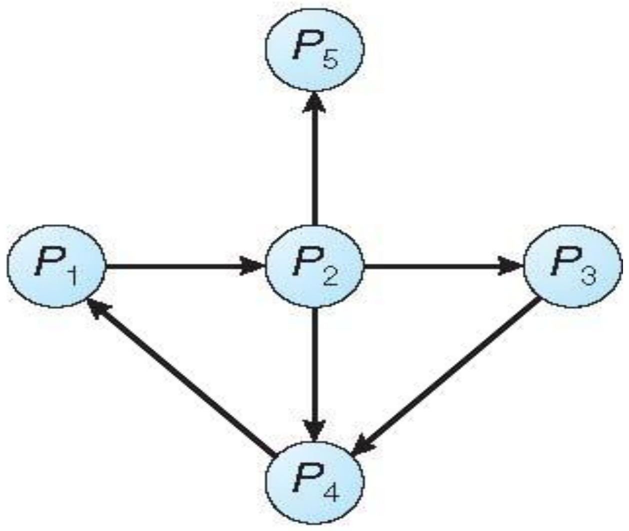
1. Single Instance of Each Resource Type

- Maintain **wait-for** graph (A variant of RA graph)
  - Nodes are processes
  - $\boldsymbol{P_i \rightarrow P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$
- 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 $\boldsymbol{n}^2$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph

2. 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, …, *n***, if ***Allocation*** $_i$ **≠ 0**, then ***Finish*[i] = *false***; otherwise, ***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

O(*m* x *n*²) → $O(m \times n^2)$

Example of 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 ***T*₀**:

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> results in ***Finish[i] = true*** for all ***i* -> *system is not in deadlocked state***

Example (Cont.)

- $\boldsymbol{P_2}$ requests an additional instance of type ***C***

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?

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?
  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?

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