Deadlock

Introduction to Operating Systems

Modeling Resource Contention

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

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **Request**
  - **Use** (exclusive)
  - **Release**

Conditions for Deadlock

- **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 process is holding onto a resource (R) while it is waiting for some other resource that can only be released after R is released

The Circular Wait Problem

A set $\{P_0, P_1, \ldots, P_\mathrm{n}\}$ of waiting processes:

- 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.

The Circular Wait Problem

Dining Philosophers problem:

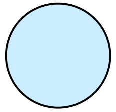
- All philosophers have picked up one chopstick
- Each is waiting for their 2nd chopstick
- But none can be released until one of the philosophers can pick up that 2nd chopstick…

Resource Allocation Graph

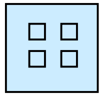
- Vertices are of two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge:** directed edge $P_i \rightarrow R_j$
- **Assignment edge:** directed edge $R_j \rightarrow P_i$

Resource Allocation Graph: Notation

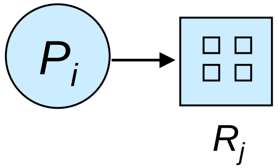
- Process



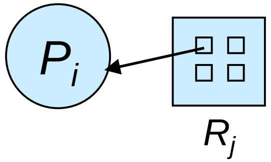
- Resource Type with 4 instances



- P_i requests instance of R_j



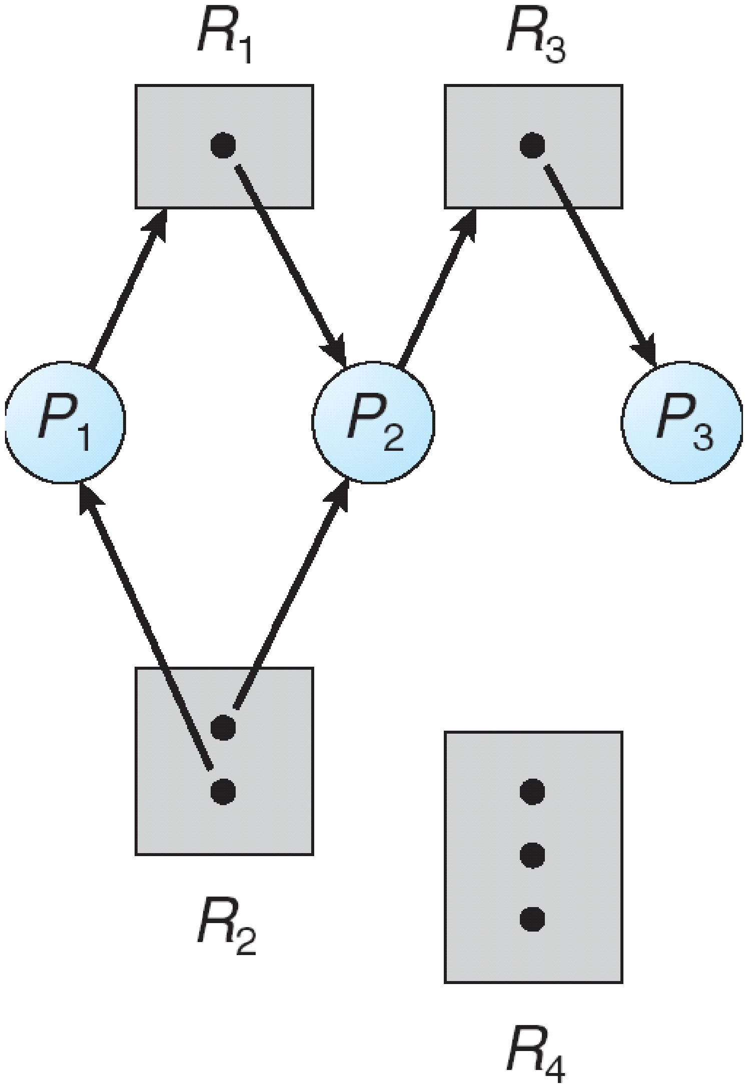
- P_i is holding an instance of R_j



Example: Resource Allocation Graph

- State:
  - P1 has R2 and is waiting for R1
  - P2 has R2 and is waiting for R3
  - P3 has R3

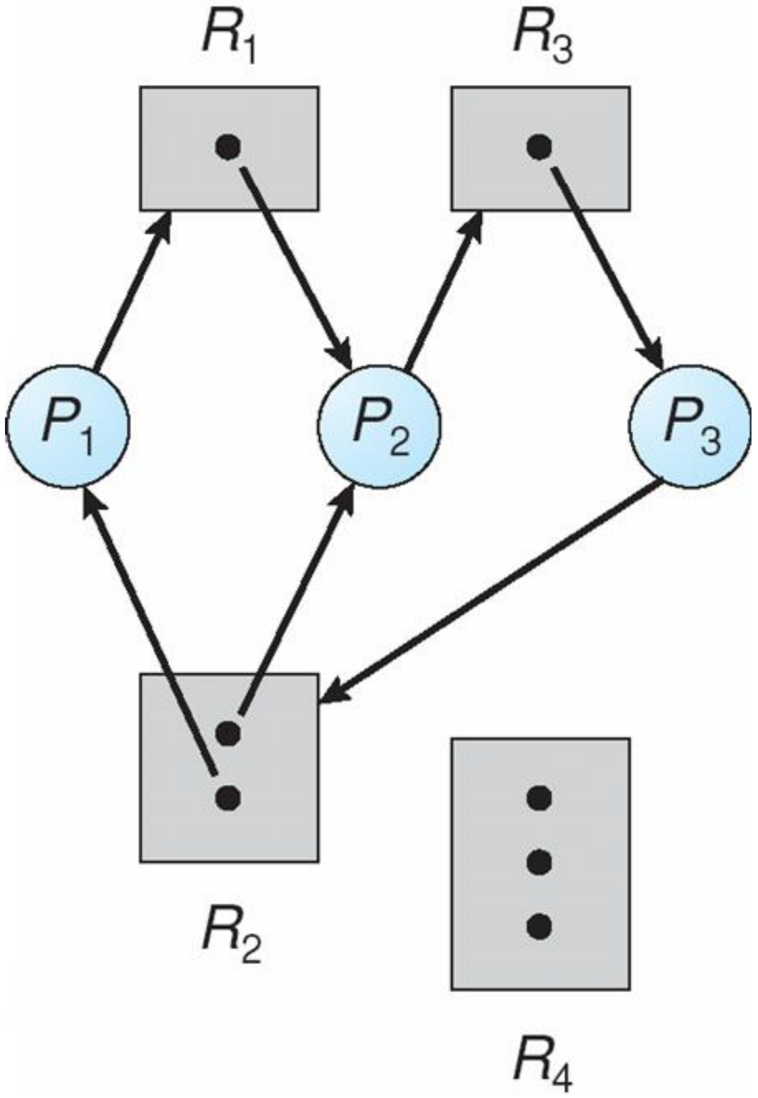
- Assuming no other allocation requests, can all of the processes complete execution?
- Yes!



Example 2: Resource Allocation Graph

- State:
  - P1 has R2 and is waiting for R1
  - P2 has R2 and is waiting for R3
  - P3 has R3 and is waiting for R2

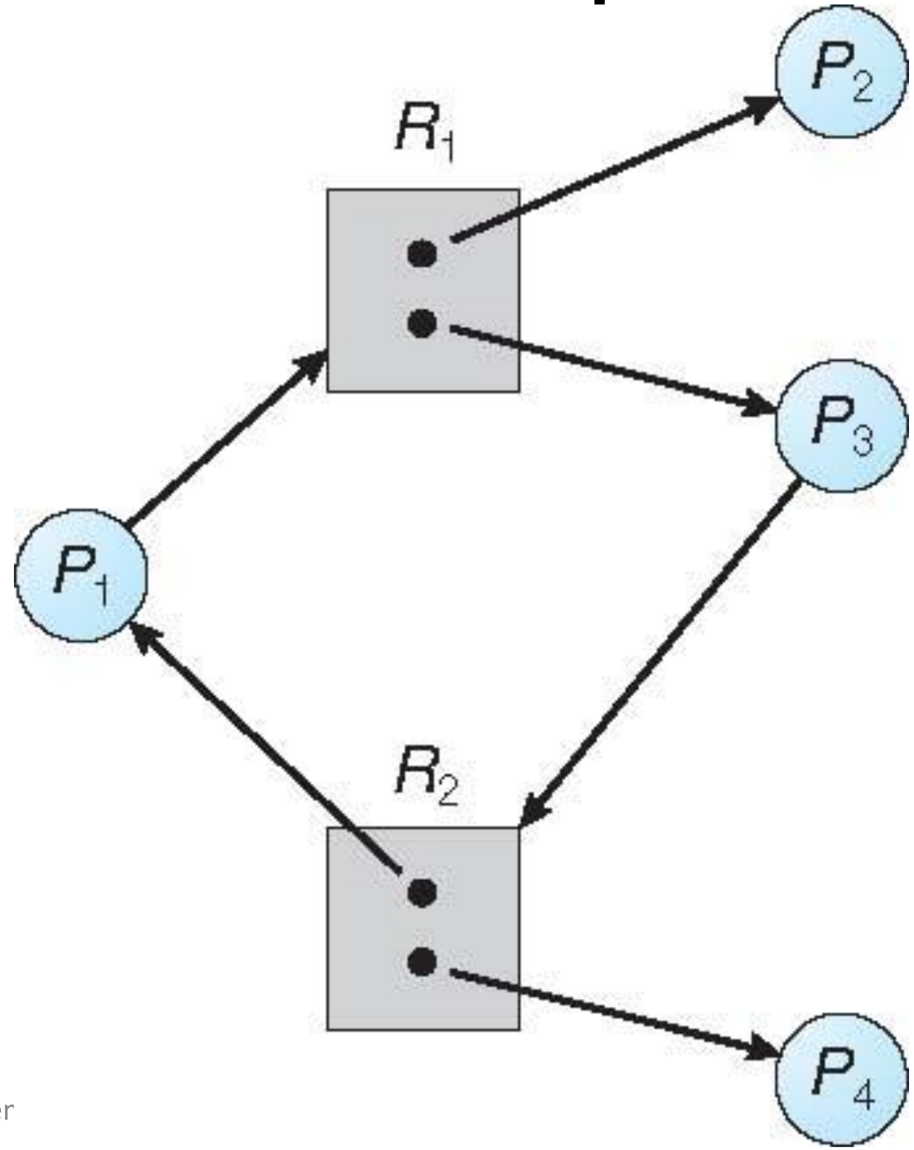
- Assuming no other allocation requests, can all of the processes complete execution?
- No! Everyone is waiting on somebody else



Example 3: Resource Allocation Graph

- State:
  - P1 has R2 and is waiting for R1
  - P2 has R1
  - P3 has R1 and is waiting for R2
  - P4 has R2

- Assuming no other allocation requests, can all of the processes complete execution?
- Yes!



Deadlock

How do we know if we have a deadlock?

- 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

Dealing with 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

Approach: we don’t allow one of the four necessary conditions to hold

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular wait

Deadlock Prevention

Mutual Exclusion

- Do not lock sharable resources (e.g., read-only files)
- But, this does not address non-sharable resources

Deadlock Prevention

Hold and Wait

- Guarantee that whenever a process requests a resource, it does not hold any other resources
- One approach: process must request all resources up front, as a single unit
- Another approach: only allow a process to request resources only when the process has none allocated to it
- Problems: Low resource utilization; starvation possible

Deadlock Prevention

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

Deadlock Prevention

Circular Wait

- Impose a total ordering of all resource types
- Require that each process requests resources in an increasing order of enumeration

- Two processes cannot both block while waiting for resources that are held by the opposite process

Deadlock Example

Prevention:

- Could force total ordering on the locks
- Could force one thread to give up locks when preempted

```
/* 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

Two different transactions execute concurrently:

- Transaction 1 transfers $25 from account A to account B, and
- Transaction 2 transfers $50 from account B to account A

Prevention:

- Could have a total ordering of accounts
- Could require all resources to be allocated simultaneously

```
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 Prevention

- Kernel can take preventative steps
  - Resource utilization could be poor

- Or the application programmer can take explicit steps
  - E.g., ordering of lock operations
  - Dealing with preemption

  - This approach relies on programmers doing the right
    - Generally, this is a bad idea…

Deadlock Avoidance

- Deadlock prevention techniques place a lot of restrictions on what can be done
  - In particular: allocation decisions are made using uniformly applied rules
- Next approach (avoidance): dynamically make allocation decisions on a case-by-case basis
  - Only allow an allocation to proceed if there is no opportunity in the current system for deadlock

Deadlock Avoidance

Process Model:

- Each process must declare up front the maximum number of resources of each type that it ***may need*** to complete execution
- Then, during execution, the process may request that resources as they are actually needed
  - Must respect the declared needs at the start

System State

Three possible situations:

- **Deadlock**: a circular wait has happened
- **Safe**: given the current allocations and the potential allocation of the remaining needs, all processes can complete without deadlock occurring
- **Unsafe**: deadlock has not occurred, but if the right set of needs are requested, then deadlock will happen

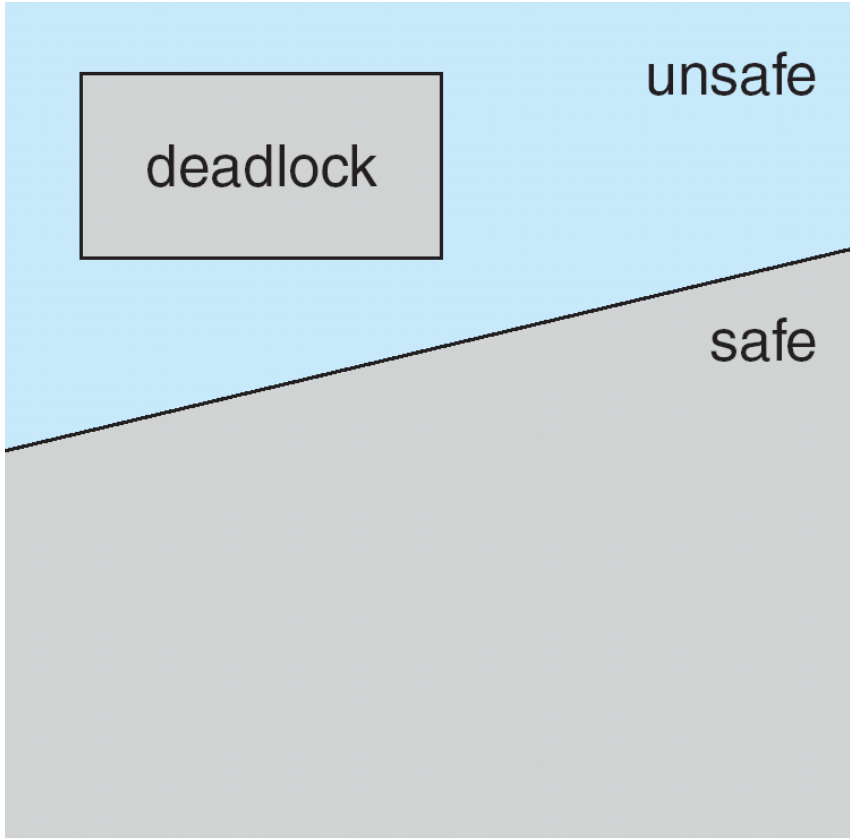
Safe State

- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the executing processes such that:
  - P_1 can allocate its remaining needs from the available resources
  - Each P_i can allocate its remaining needs from the available resources **plus** those currently held by processes $P_1 \ldots P_{i-1}$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished (where $j < i$)
  - P_i can then obtain the needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} is guaranteed to be able to obtain its needed resources, etc.

System State

Three possible situations:

- **Deadlock**: a circular wait has happened
- **Safe**: all processes can complete without deadlock occurring
- **Unsafe**: deadlock has not occurred, but if the right set of needs are requested, then deadlock will happen



System Allocation Algorithm

- Goal: always stay in a safe state
- When a new request is made by a process:
  - Kernel tests whether the new state will be safe or not
  - If safe, then allocation is allowed
  - If unsafe, then the process is placed in a waiting queue until a safe state can be achieved

Avoidance Algorithms

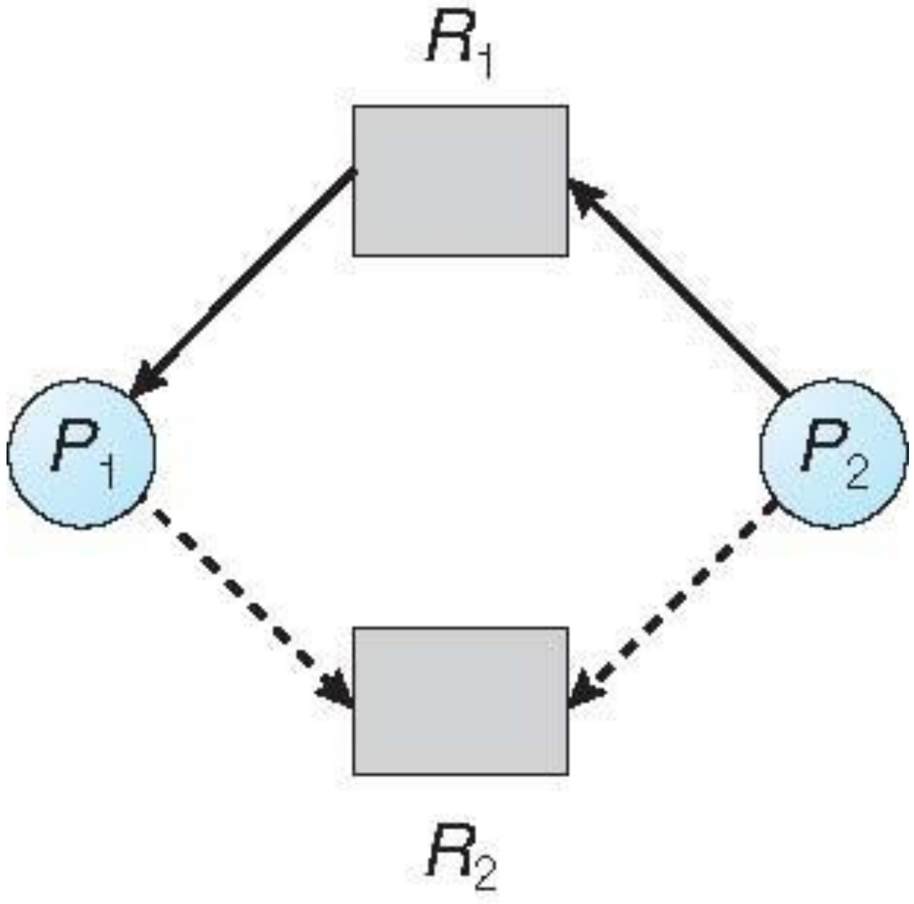
- All resources are single-instance:
  - We can just look at the resource allocation graph to determine whether a cycle can happen
- Multiple instances of some resources:
  - Use the **Banker’s Algorithm** to determine safe vs unsafe

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource.
  - Request edge: $P_i \rightarrow R_j$ solid line
- Request edge converted to an assignment edge when the resource is allocated to the process
  - Assignment edge: $R_j \rightarrow P_i$
- When a resource is released by a process, assignment edge reconverts back to a claim edge
- All resources must be claimed before any allocation requests are made

Resource-Allocation Graph

- P1:
  - Claimed: R2
  - Assigned R1
- P2:
  - Claimed: R2
  - Requested: R1



Two independent questions:

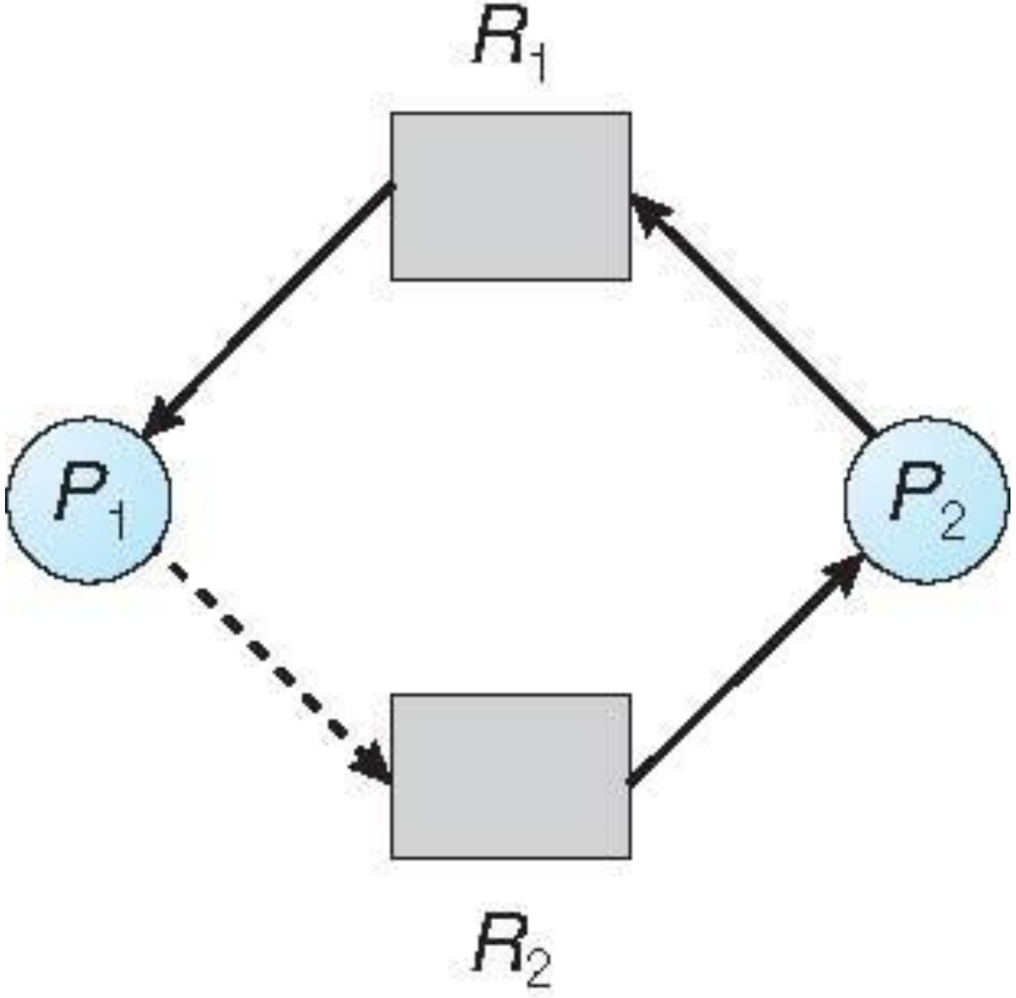
- Should P1 be assigned R2?
- Should P2 be assigned R2?

Resource-Allocation Graph

Assign R2 to P2:

- Now in an unsafe state!
- If P1 then requests R2, we will have deadlock

Conclusion: we should not assign R2 to P2 right now



Resource-Allocation Graph Algorithm

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
- If this is the case, then the process is placed into a waiting queue

Banker's Algorithm

- Multiple instances of resources
- Each process must claim the maximum use of resources before any requests can be made
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them and terminate 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 to be allocated
- **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

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

Banker's Algorithm: Determining Safety

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

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

Otherwise, it is unsafe

- Examples

Using the Banker's Algorithm

$Request_i$ = 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 ***$Request_i$ > $Need_i$*** raise error condition, since process has exceeded its maximum claim
2. If ***$Request_i$ > Available***, ***P_i*** must wait, since the 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

Banker's Example III

Process	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			

https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/

Banker's Example IV

New request by Process 1: 1,0,2

- Will we be in a safe state?

Process	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			

Deadlock Summary

Necessary conditions for deadlock (all must be true):

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular wait

Deadlock Summary

Deadlock Prevention:

- Fixed set of rules that apply to all situations
- Remove one of the necessary conditions
- Simple
- But: can be overly conservative and may not give us good use of the available resources

Deadlock Summary

Deadlock Avoidance:

- Make context-specific decisions on the fly as to whether an allocation request should be granted
- Single instance per resource type:
  - Use allocation graph
  - If an allocation results in a cycle, then do not grant it
- Multiple instances per resource type:
  - Banker's Algorithm
  - If an allocation results in an unsafe state, then do not grant it