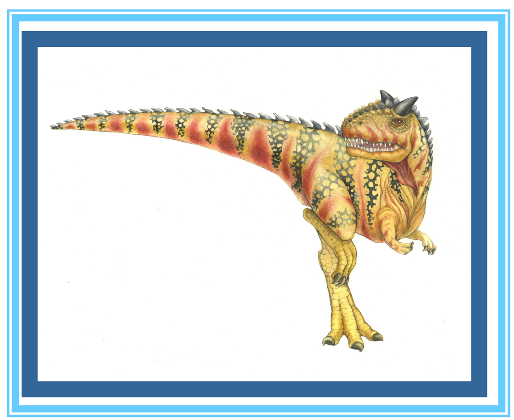
Section 8: Deadlocks



Chapter 7: Deadlocks

- A set of System and Resource Assumptions
- Deadlock Characterization
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

System Model

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

requests/releases

- The request and release of resources done through system calls:
  - open() (request for file) and close() (release of a file)
  - allocate() and free() of memory
  - wait() and signal() for semaphores
  - acquisition and release for mutex locks

What is a Deadlock?

- A set of processes is in a ***deadlock*** state when
  - Every process in the set is waiting for an event that can be caused only by another process in the set
  - The event in question is a resource acquisition or release

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Could happen with semaphores
- S and Q are two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

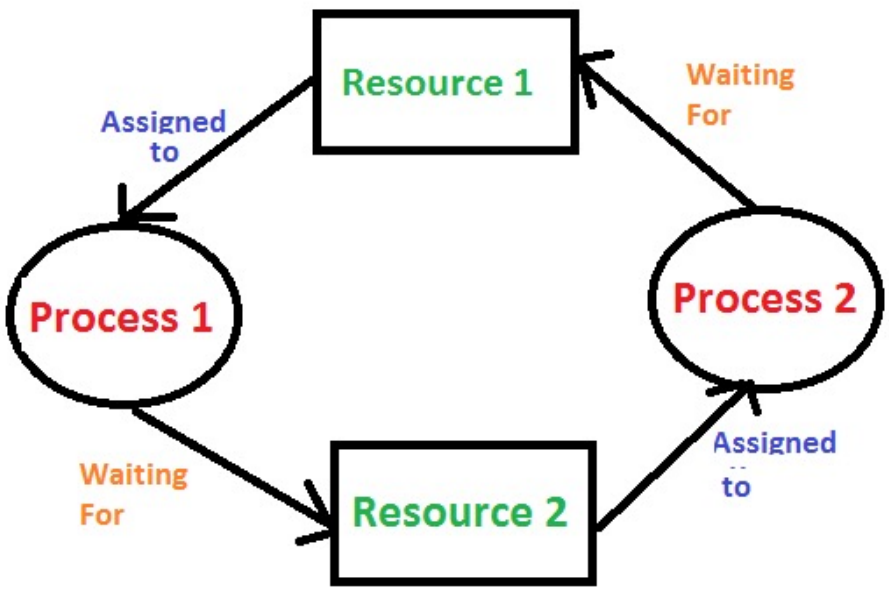
Process deadlock itself

- A deadlock can occur even when there is a single process

- Suppose a programmer replaces signal() with wait(), then the process deadlock itself

```
wait (mutex);
    critical section
wait (mutex);
    remainder
```

Deadlock in graphic



Deadlock with Semaphores

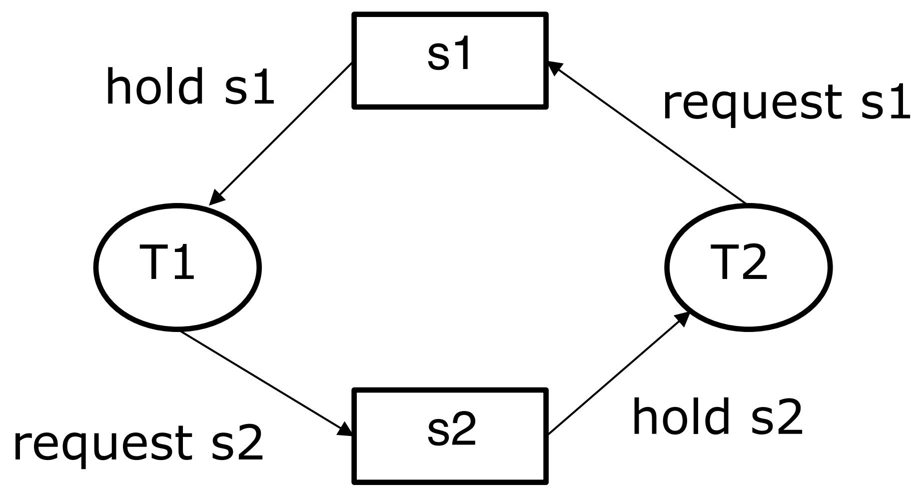
- Data:
  - A semaphore s1 initialized to 1
  - A semaphore s2 initialized to 1
- Two threads T1 and T2
- **T1:**

  wait(s1) success

  wait(s2) blocked
- **T2:**

  wait(s2) success

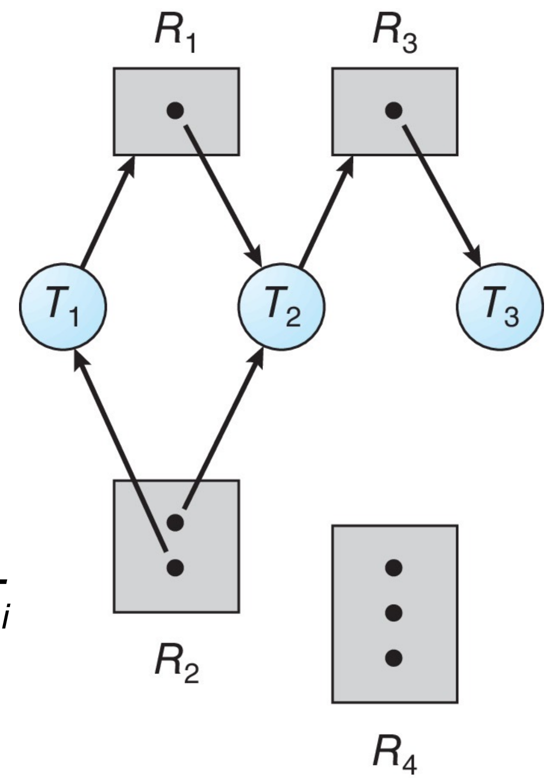
  wait(s1) blocked



Resource-Allocation Graph

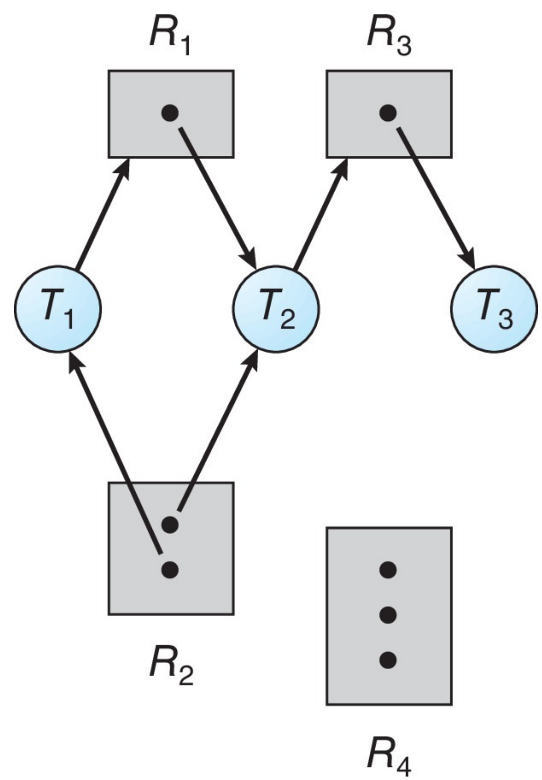
The resource-allocation graph G = (V,E) describes the relationship among the processes and resources in a system in terms of a directed graph

- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Edges are either
  - **request edge** – directed edge $T_i \rightarrow R_j$
  - **assignment edge** – directed edge $R_j \rightarrow T_i$



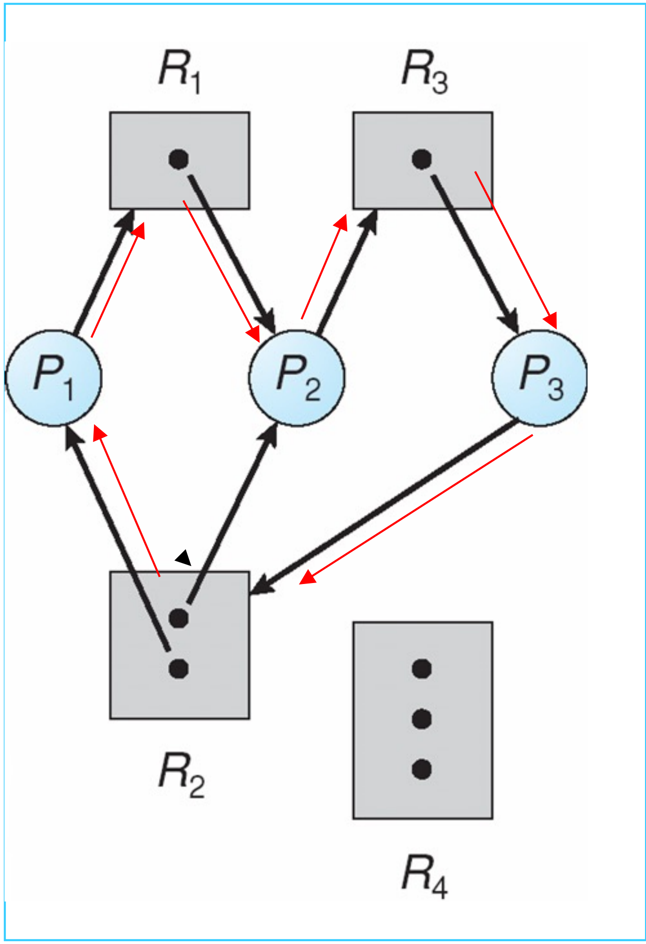
Resource Allocation Graph Example

- The number of dots inside a resource type is the number of occurrence of the corresponding resource:
  - One instance of R1
  - Two instances of R2
  - One instance of R3
  - Three instance of R4
- Edges meaning:
  - T1 holds one instance of R2 and is waiting for an instance of R1
  - T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
  - T3 is holds one instance of R3



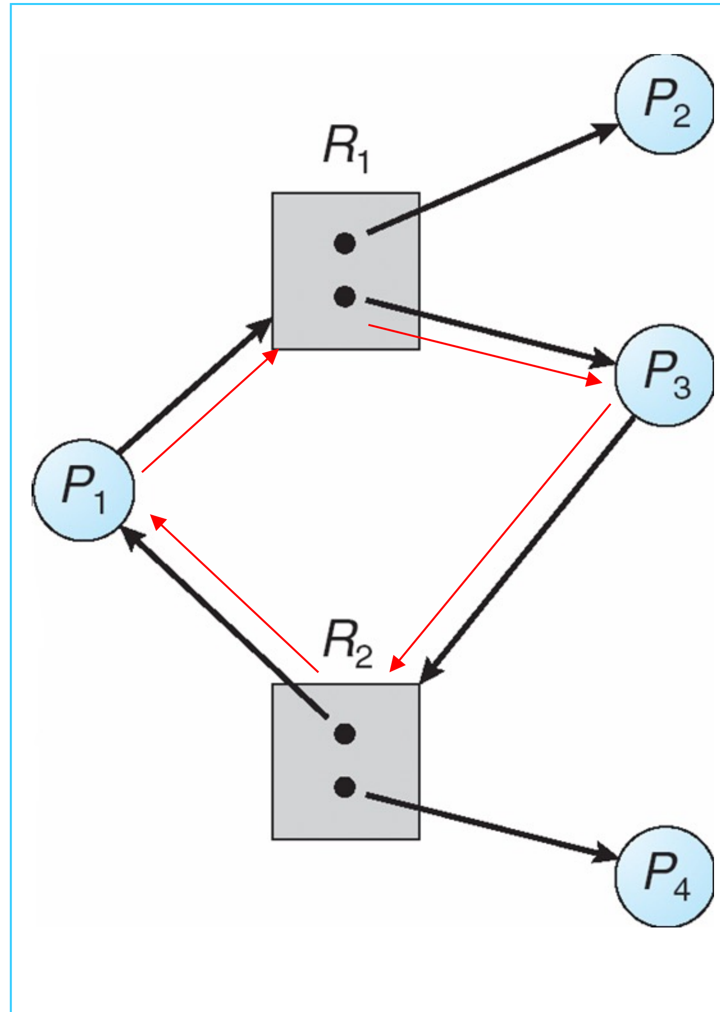
Resource Allocation Graph: Deadlock

- Deadlocked may occur when there are cycles in a resource-allocation graph
- This resource-allocation graph has two cycles:
  - $P_1\ R_1\ P_2\ R_3\ P_3\ R_2\ P_1$
  - $P_2\ R_3\ P_3\ R_2\ P_2$
- In the cycle $P_1\ R_1\ P_2\ R_3\ P_3\ R_2\ P_1$
  - P_1 is waiting for the resource R_1 which is held by P_2
  - P_2 is waiting for resource R_3 which is held by P_3
  - P_3 is waiting for R_2 which is held by P_1
  - The three processes are deadlocked



Graph With a Cycle: No Deadlock

- Resource allocation graph is a picture in time of the state of the resource allocation, this graph keeps changing during the computation
- If the resource allocation graph contains no cycles $\Rightarrow$ no deadlock
- If the resource allocation graph contains a cycle:
  - only one instance per resource type, then deadlock
  - several instances per resource type, then possibility of deadlock



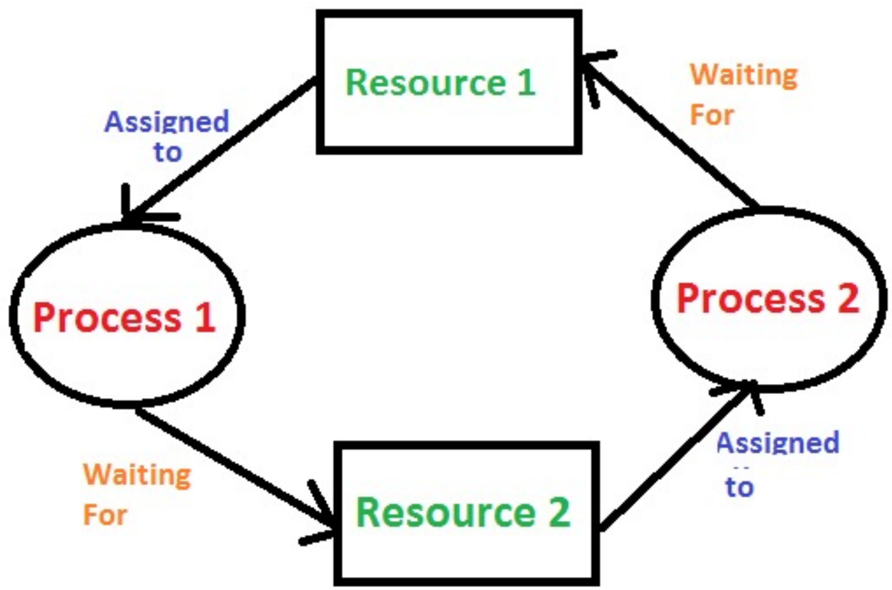
Deadlock Characterization

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.

Example

Here all the four necessary conditions for deadlock exist: mutual exclusion, hold and wait, no preemption, circular wait



Methods for Handling Deadlocks

- We can deal with the deadlock problem in one of three ways:
  1. Use protocols to avoid deadlocks, ensuring that the system will *never* enter a deadlock state: deadlock prevention and deadlock avoidance
  2. Allow the system to enter a deadlock state, detect it, and then recover
  3. Ignore the problem and pretend that deadlocks never occur in the system

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources
  - A printer is a non-sharable resource, cannot be accessed simultaneously by several processes
  - A sharable resource like read-only files can be accessed by several processes
    - As processes don’t need to wait for sharable resources, they cannot be deadlocked

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - 1- require process to request and be allocated all its resources before it begins execution, or 2- allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention

- **Hold and Wait** (continue)– must guarantee that whenever a process requests a resource, it does not hold any other resources
- Ex: a process copies data from a DVD drive to a file on disk, sorts the file and then prints the results to a printer
- First protocol:
  - Require process to request and be allocated all its resources before it begins execution
  - Process initially request the DVD drive, disk file and printer
  - Printer is held for the whole execution though used only at the end
- Second protocol:
  - Processes request resources only when they have none
  - A process may request some resources and use them
  - If additional resources needed, must release all resource currently held
  - Once process has copied from DVD drive to disk, release both resources, then again request disk file and printer to copy disk file to the printer

Hold & Wait Protocols

- Disadvantages for both protocols:
  - **Low resource utilization**:
    - Resources may be allocated but unused for a long period (first protocol)
  - **Starvation is possible**:
    - A process that needs several popular resources may have to wait indefinitely (second protocol)
    - If at least one of the resources that it needs is always allocated to some other process

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
  - Released 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

Circular Wait: Example

- Impose a total ordering F of all resource types:
  - F(tape drive) = 1
  - F(disk drive) = 5
  - F(printer) = 12
- Require that each process requests resources in an increasing order of enumeration
  - Usually tape drive is needed before the printer, so makes sense to define F(tape drive) < F(printer)
- => P_k cannot request a resource R_i when holding resource R_j for j > i.
- Therefore, circular waiting impossible for it will implies that:
  - $F(R_n) < F(R_0) < F(R_1) < F(R_{n-1}) < F(R_n)$
- Deadlock will be avoided on slide 10 if mutex variables are ordered after the first thread acquires a lock:
  - If thread one acquires first mutex, then order is one, two. Second thread must follow this order

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

`first_mutex = 1`
`second_mutex = 5`

code for **`thread_two`** could not be written as follows:

```
/* 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 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

Resource Allocation State

- *Resource-allocation state* is the number of available and allocated resources, and the maximum demand by each process for these resources
- This state (represented in a table) is dynamically updated
- Deadlock-avoidance algorithms examines the resource-allocation state to ensure that there can never be a circular-wait condition

12 resources, 3 available

	Max Needs	Current Needs (hold)
P_0	10	5
P_1	4	2
P_2	9	2

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- A state is safe if the system can allocate resources to each process in some sequence and avoid deadlock
- Formally, the system is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all the processes 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
- If no such sequence exists, then the system is said to be unsafe

Deadlock Avoidance

- Example: a system with 12 magnetic tape drivers and three processes: P_0, P_1, and P_2.
  - At time t_0: Additional information

	Max Needs	Current Needs (hold)
P_0	10	5
P_1	4	2
P_2	9	2

available:
3

  - Is the system in a safe state? **YES!!**
  - The sequence $<P_1, P_0, P_2>$ satisfies the safety condition.

Deadlock Avoidance

- A system can go from a safe state to an unsafe state: at time t_1:

	Max Needs	Current Needs (hold)
P_0	10	5
P_1	4	2
P_2	9	3

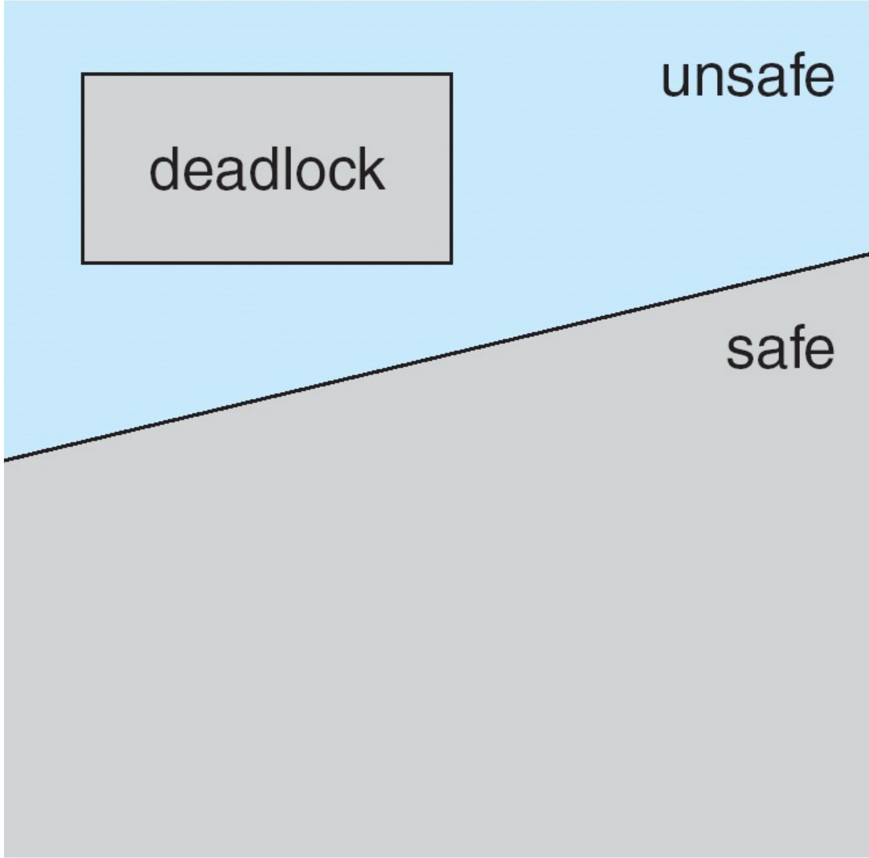
available:
2

Process P_2 requests and is allocated one more tape drive.

- Can we find a safe sequence?
- We cannot , P_1 is safe, but not the two other processes
- The mistake was in granting the request from P_2 for one more tape drive.

Safe, Unsafe, Deadlock State

- 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.



Avoidance algorithms

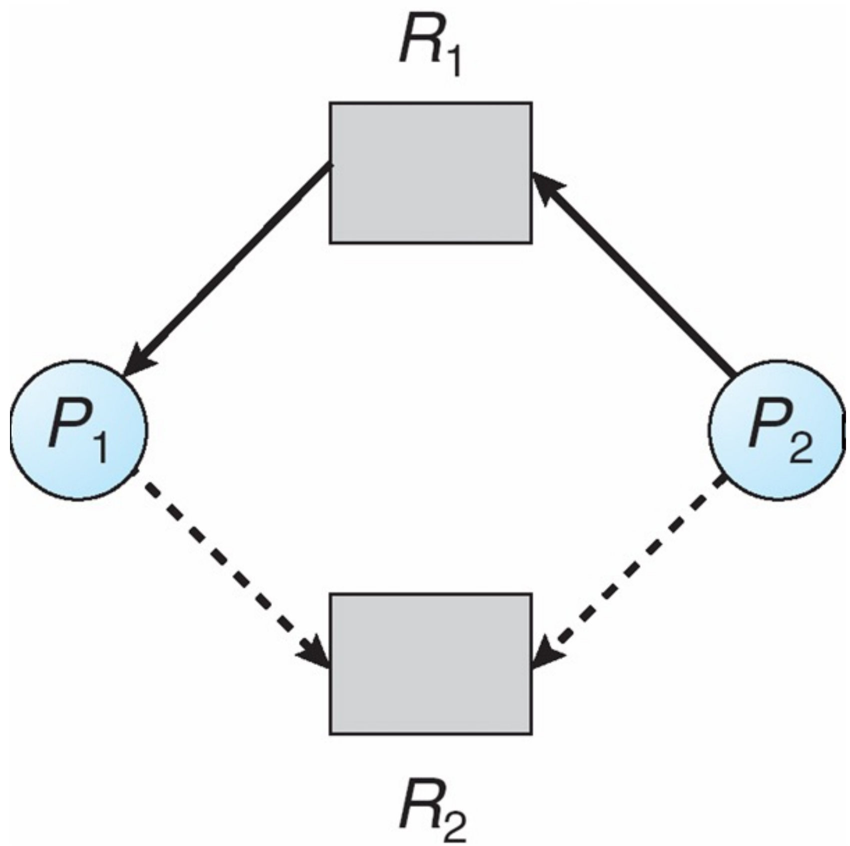
- Avoidance algorithms ensure that the system will always remain in a safe state
- Initially the system is in a safe state
- Whenever a process requests a resource that is currently available, the system must decide
  - whether the resource can be allocated or
  - whether the process must wait

Avoidance algorithms

- Single instance of a resource type
  - Use the resource-allocation graph algorithm
  - In addition to the request and assignment edges, we introduce a new type of edge, called a ***claim edge***
- Multiple instances of a resource type
  - Use the banker's algorithm

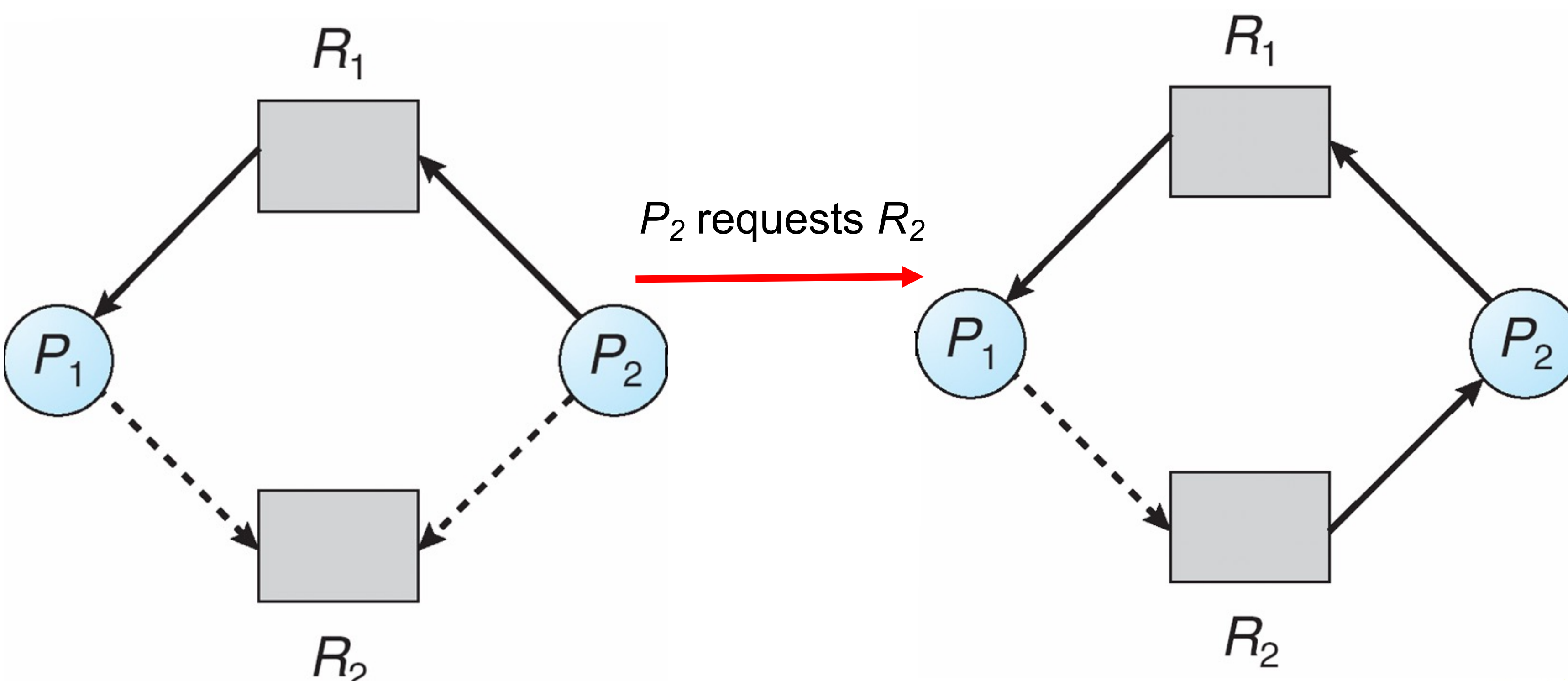
Resource-Allocation Graph Algo

- Claim edge
  - represented by a dashed edge $P_i \to R_j$
  - indicates that process P_j may request resource R_j
- Resources must be claimed *a priori* in the system
  - Before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph

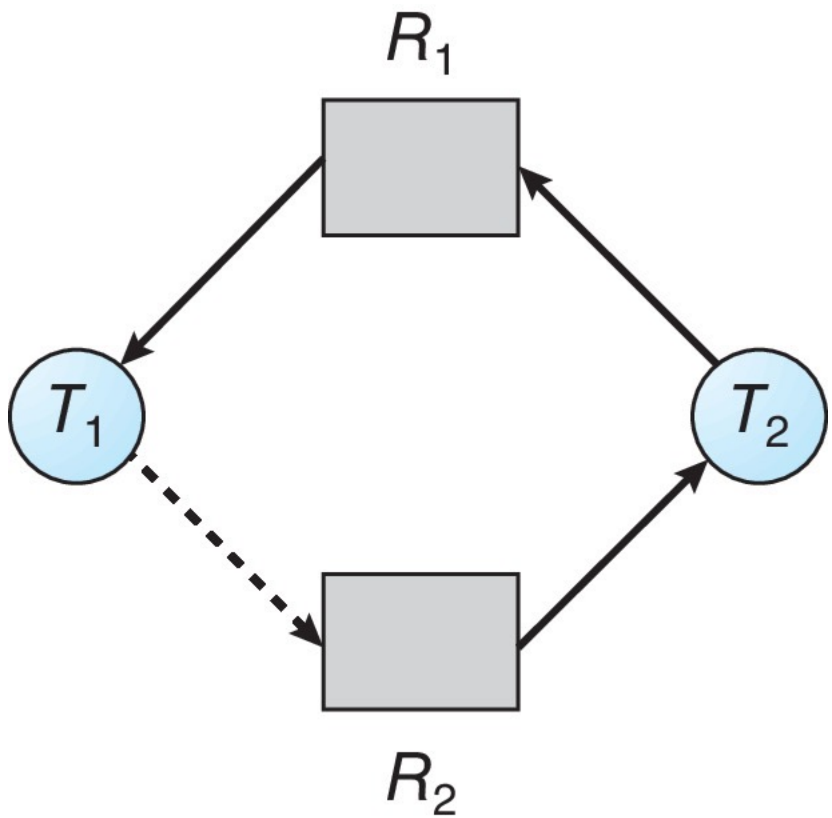


Resource-Allocation Graph Algo

- Claim edge converts to assignment edge when a process requests a resource
- Request granted only if the edge conversion does not result in a cycle in the graph



Unsafe State In Resource-Allocation Graph



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

Banker's Algorithm

- Multiple instances of each resource
- When a process enters the system, must declare the maximum number of instances of each resource it may need
- System must determine whether the allocation of the resources requested by a process will leave the system in a safe state

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* and

   (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

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 $Request_i \leq Need_i$ 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:

   $Available = Available - Request;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

   - 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

- Given:
  1. 5 processes P_0 through P_4;
  2. 3 resource types: *A* (10 instances), *B* (5 instances), and *C* (7 instances)
  3. 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	

The matrix need

Compute the matrix ***Need*** defined as ***Max – Allocation***

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

System is in a safe state

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

- The need from P_1 can be satisfied from the available resources. Once P_1 exits, its allocated resources become available, A = 5, B = 3, C = 2.
- Next need from P_3 can be satisfied, available becomes A = 7, B = 4, C = 3
- Similarly, for P_4
- Thus, the system is in a safe state since the sequence <P_3, P_1, P_4, P_2, P_0> satisfies safety criteria

P_1 makes request (1,0,2)

- Check that request ≤ need (that is, (1,0,2) ≤ (1,2,2) ⇒ true
- Check that request ≤ Available (that is, (1,0,2) ≤ (3,2,2) ⇒ true
- Pretend the request is allocated to P_1, i.e., <u>allocation</u> changes from 2, 0, 0 to 3, 0, 2, (thus <u>available</u> become 2, 3, 0) and check if system in a safe state

	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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Exercises

- 2 processes P_0 ,P_1;

  4 resource types of one instance each

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P_0	1 1 0 0	1 1 0 1	0 0 0 1
P_1	0 0 1 0	0 1 1 1	

- Draw the resource-allocation graph for this system
- Assume a request is made by P_0 to obtain resource D, draw the resource-allocation graph after this request has been made
- Can this request be granted? Explain your answer briefly

Exercises

- Consider the following snapshot of a system of four processes and four resource types:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 1 0 0	0 1 0 1	0 0 0 1
P_1	0 0 1 0	0 0 1 1	
P_2	1 0 0 0	1 1 1 1	
P_3	0 0 0 0	0 0 0 1	

- Draw the matrix Need?
- Is the system in safe state? If you answer yes, prove it by exhibiting a safe sequence. If you answer no, show a scenario where the safety algorithm fails to complete a safe sequence.
- Given the current allocation of resources, if a request from process P_2 arrives for (0,1,0,0), can the request be granted immediately? Explain your answer.

Deadlock Detection

- No effort is made to detect whether the system is in a safe state
- Rather allow the system to perform without constraints
- Periodically detect whether the system is in a deadlock state using a detection algorithm
- Apply some recovery scheme

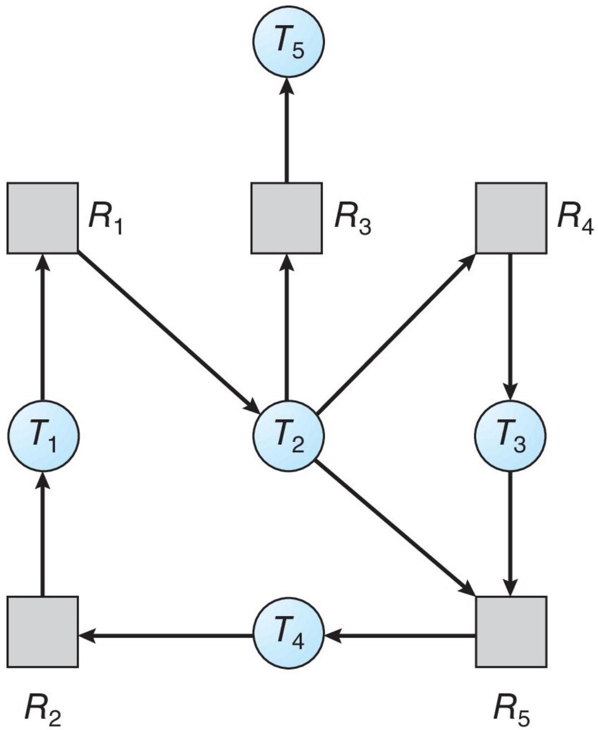
Single Instance of each Resource Type

- Maintain a **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

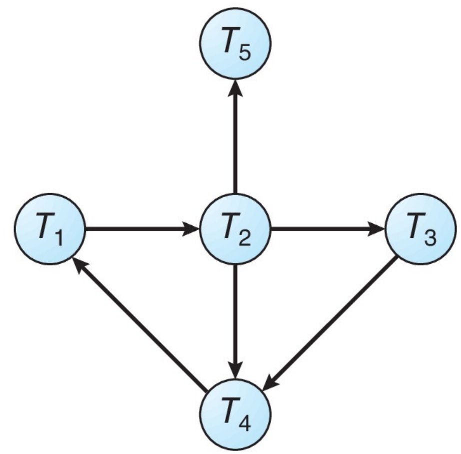
- Periodically invoke an algorithm that searches for a cycle in the wait-for graph. If there is a cycle, then there exists a deadlock

- 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

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.

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 $Allocation_i \neq 0$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***
2. Find an index ***i*** such that both:
   a) ***Finish*[*i*] == *false***
   b) $Request_i \leq Work$

   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] == 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

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Allocation is what is currently allocated to a process, while Request are the resources a process currently wait-for given none is available
- In this current config, the detection algo is run to see whether there is a deadlock

	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	

- Here the system is not in deadlock state as once processes P_0, P_2 exit, they will release enough resource to allow the other processes to complete
- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- Here P_2 make an additional request for an 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

	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	

- 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

Exercises

- Five processes P_0 through P_4; three resource types A (5 instances), *B* (6 instances), and *C* (3 instances)

- Snapshot at time T_0:

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

- Is this system in a deadlock state? Prove your answer by listing an execution sequence of the processes or by pointing to a deadlocked process

Exercise

- Here P_1 make an additional request for an instance of type ***C***

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

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

- State of system?
  - Give an executable process sequence with this new request if one exists
  - Otherwise, point to the deadlock in which the system will be with this request

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

End of Section 8