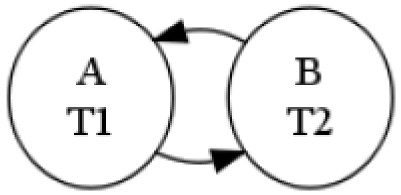
Deadlocks

Process Deadlocks

- *Definition : A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*
- i.e. A deadlock is a situation where a process or a set of processes is blocked, waiting on an event which may be the release of computing resource or other hardware resources, that can only be caused by a member of the set and that will never occur.
- The formation and existence of deadlocks in a system lowers system efficiency.
- Therefore to avoid performance degradation, a system should be deadlock free or deadlocks should be quickly detected and recovered.

Examples

- System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one



- Semaphores A and B each initialized to 1

```
P0                      P1
wait(A)                      wait(B)
wait(B)                      wait(A)
```

Deadlock vs Starvation

- A process is *deadlocked* if it is waiting for an event that will never occur. Typically, more than one process will be involved in a deadlock and the processes are involved in a circular wait.
- Starvation occurs when a process waits for a resource that continually becomes available but is never assigned to that process because of priority or a flaw in the design of a scheduler e.g. the process is ready to proceed but never gets the CPU.

- Two major differences between deadlock and starvation:

  1. In starvation, it is not certain that a process will ever get the requested resource, whereas a deadlocked process is permanently blocked because the required resource never becomes available unless external actions are taken.

  2. In starvation, the resource under contention is in continuous use whereas in deadlock resources are not used as the processes are blocked.

Resources:

Resource is a commodity required by a process to execute.

Under normal operation, a resource allocation proceed like this::

1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.

Resources can be of several types:

- Serially Reusable Resources

  e.g. CPU cycles, memory space, I/O devices, files

  A process acquires, uses and then releases the resource.

- Consumable Resources

  Produced by a process, needed by a process

  e.g. Messages, buffers of information, interrupts

  A process creates, acquires and then use the resource. Resource ceases to exist after it has been used.

Another classification is

- Preemptable - the resource can be taken away from its current owner (and given back later). An example is memory.
- Non-preemptable- the resource cannot be taken away. An example is a printer.

Fundamental causes of deadlocks

The following four conditions are *necessary* for a deadlock to occur.

- Mutual exclusion: A resource can be assigned to at most one process at a time (no sharing).
- Hold and wait: A processing holding a resource is permitted to request another. i.e. 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 i.e. A process must release its resources; they cannot be taken away.
- Circular wait: There must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain.

  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.

System model

System can be modelled using Resource allocation graph. Resource types are denoted by, R_1, R_2, ..., $R_{n\,,}$ and each resource R has W_i instances.

Resource Allocation Graph (RAG)

A set of vertices V and a set of edges E

V is partitioned into 2 types

P = {P1, P2,…,Pn} - the set of processes in the system

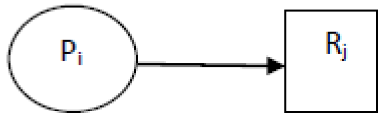
R = {R1, R2,…,Rn} - the set of resource types in the system

Two kinds of edges

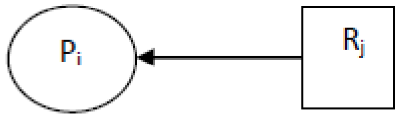
Request edge - Directed edge Pi ---> Rj

Assignment edge - Directed edge Rj ----> Pi

P_i requests an instance of R_j

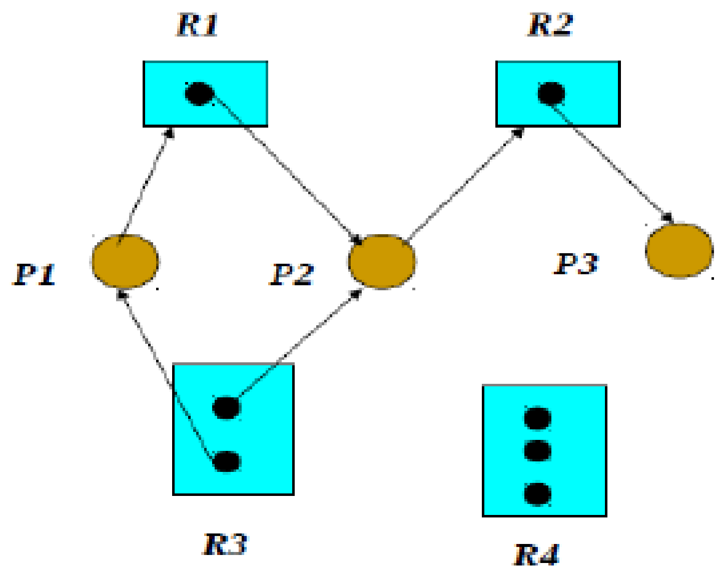


P_i is holding an instance of R_j

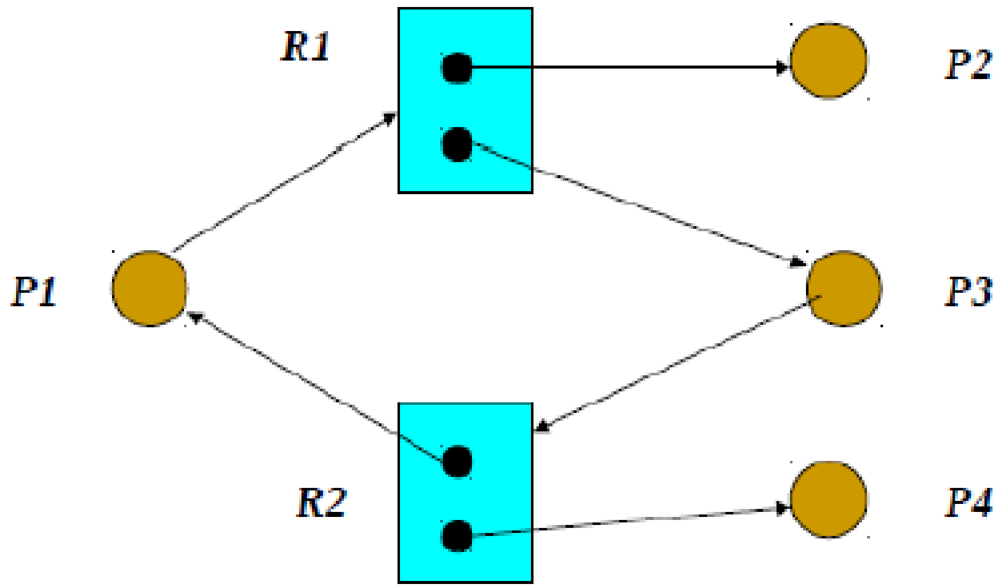


- The request for resources by processes can be modelled using RAG.
- If a resource allocation graph contains no cycles, then no process is deadlocked.
- If a RAG contains a cycle, then a deadlock may exist. Therefore, a cycle means deadlock is *possible*, but not necessarily *present*.
- A cycle is not sufficient proof of the presence of deadlock. A cycle is a *necessary* condition for deadlock, but not a *sufficient* condition for deadlock.

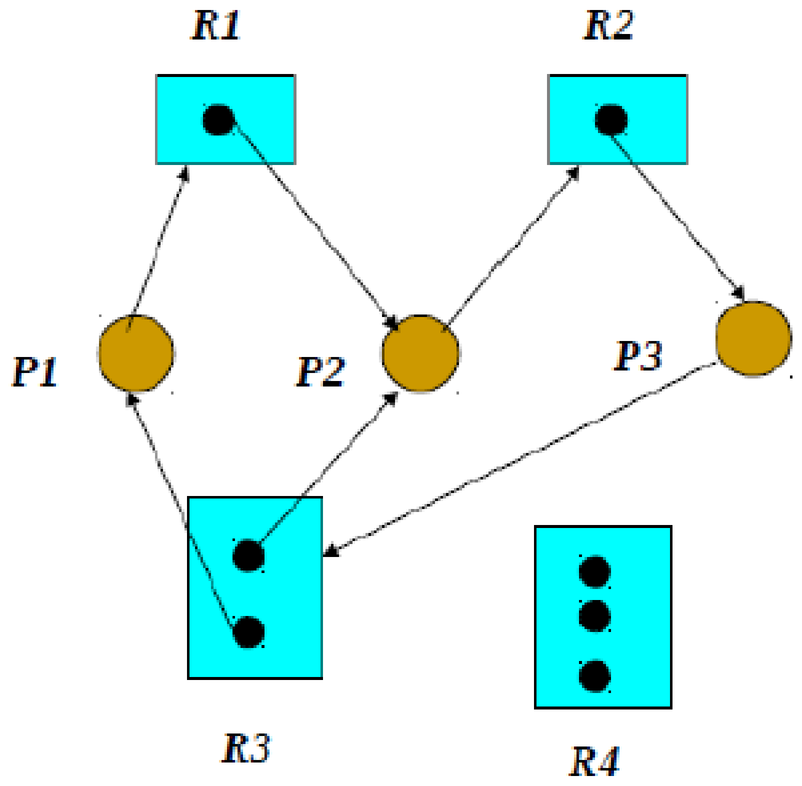
RAG with no cycles



RAG with cycles



RAG with cycles and deadlocks



If RAG contains no cycles then NO DEADLOCK.

If RAG contains a cycle and

- if only one instance per resource type, then deadlock
- if several instances per resource type, possibility of deadlock.

Four *strategies* for handling with deadlocks

- Ignore the problem - Ostrich Algorithm Popular in Distributed Systems.
  - Reasonable if
    - deadlocks occur very rarely
    - cost of prevention is high
  - UNIX and Windows takes this approach
- Deadlock Prevention - Prevent deadlocks by violating one of the 4 necessary conditions.
- Deadlock Avoidance - Avoid deadlocks by carefully deciding when to allocate resources.
- Deadlock Detection and recovery - Detect deadlocks and recover from them

Deadlock Prevention

Restrain the ways request can be made

- Prevent Mutual Exclusion
  - Not required for sharable resources (e.g., read-only files);
  - Must hold for non-sharable resources
  - Prevention not possible, since some devices are intrinsically non-sharable

- Prevent 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

1. Pre-allocate
   - do not pick up one chopstick if you cannot pick up the other
   - for a process that copies data from DVD drive to a file on disk and then prints it from there:
     - request DVD drive
     - request disk file
     - request printer

2. A process can request resources only when it has none

  - request DVD drive and disk file
  - release DVD drive and disk file
  - request disk file and printer (no guarantee data will still be there)
  - release disk file and printer

- Disadvantages: inefficient, possibility of starvation.

- Prevent 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
  - some resources cannot be feasibly preempted (e.g., printers, tape drives)

- Prevent Circular Wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

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

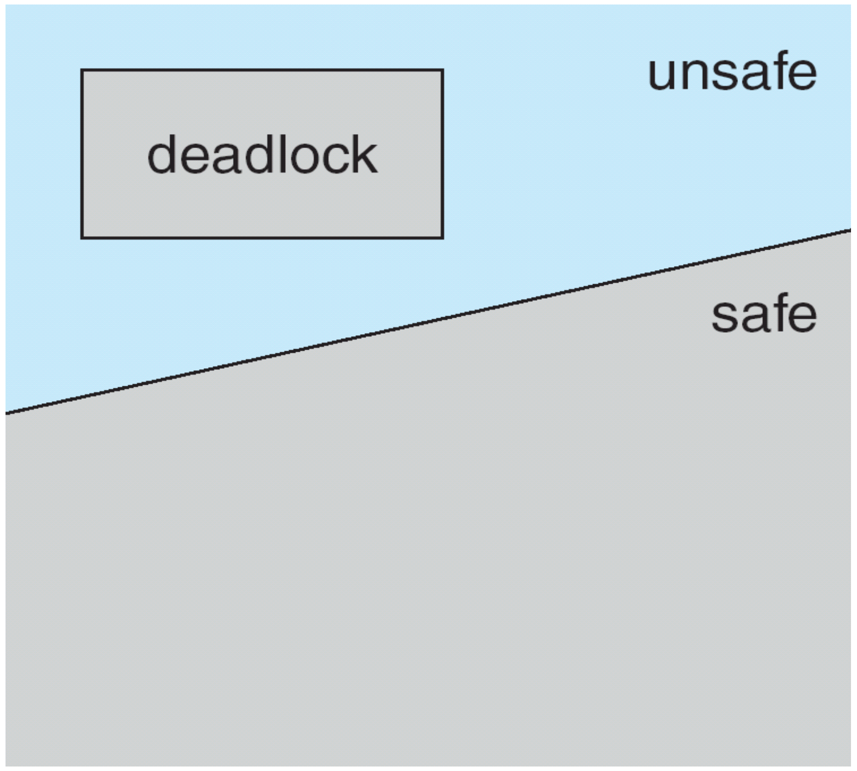
Safe State

- 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 $<P_1, P_2, ..., P_n>$ 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

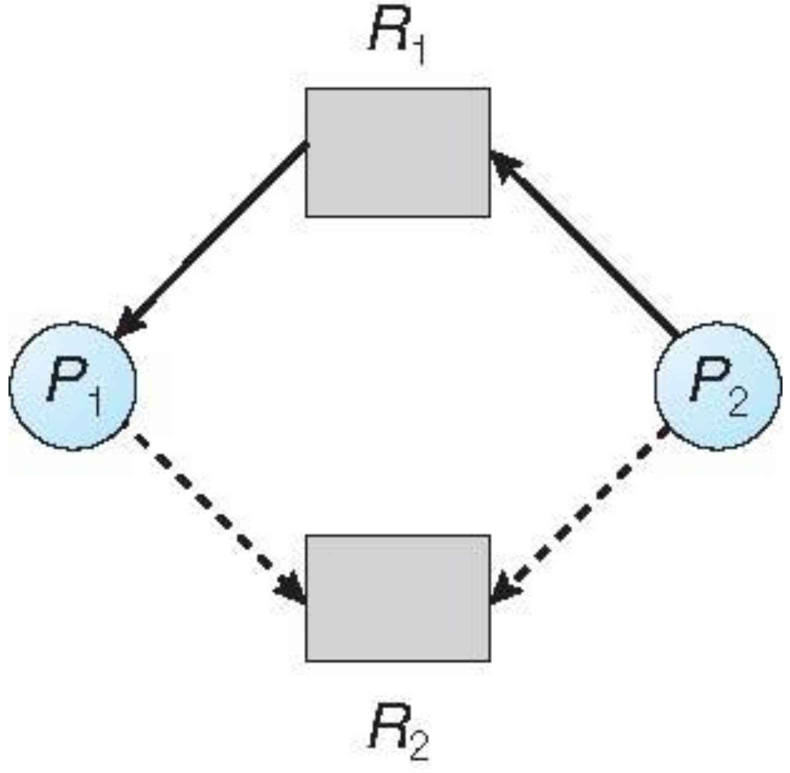
- Single instance of a resource type
  - Use a resource-allocation graph

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

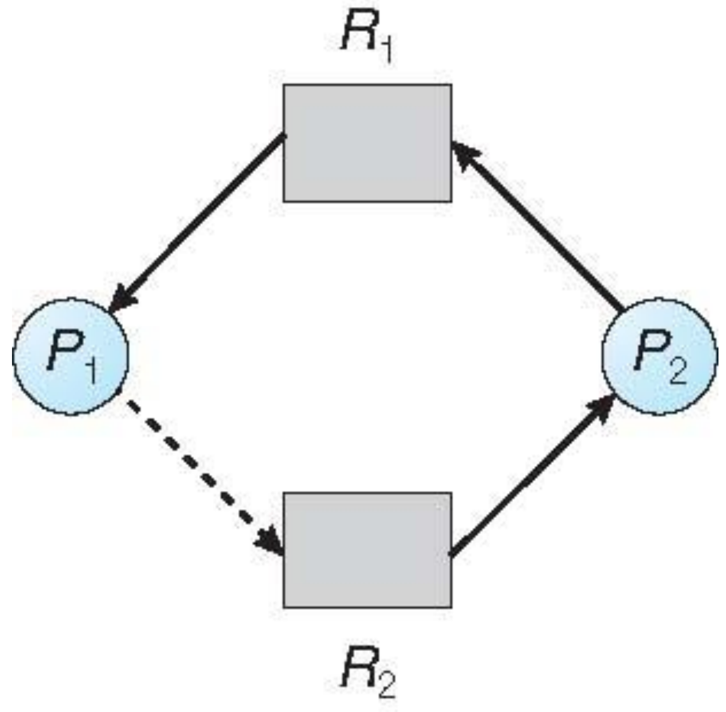
Resource-Allocation Graph Scheme

- **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



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

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) $Need_i \leq Work$
   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**$_i$* = 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**$_i$* $\leq$ ***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**$_i$**;

   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

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

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

- 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 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> 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?

Deadlock Detection

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

- It requires an algorithm which examines the state of the system to determine whether a deadlock has occurred
- It has overhead of run-time cost for maintaining necessary information and executing the detection algorithm and potential losses inherent in recovering from deadlock.

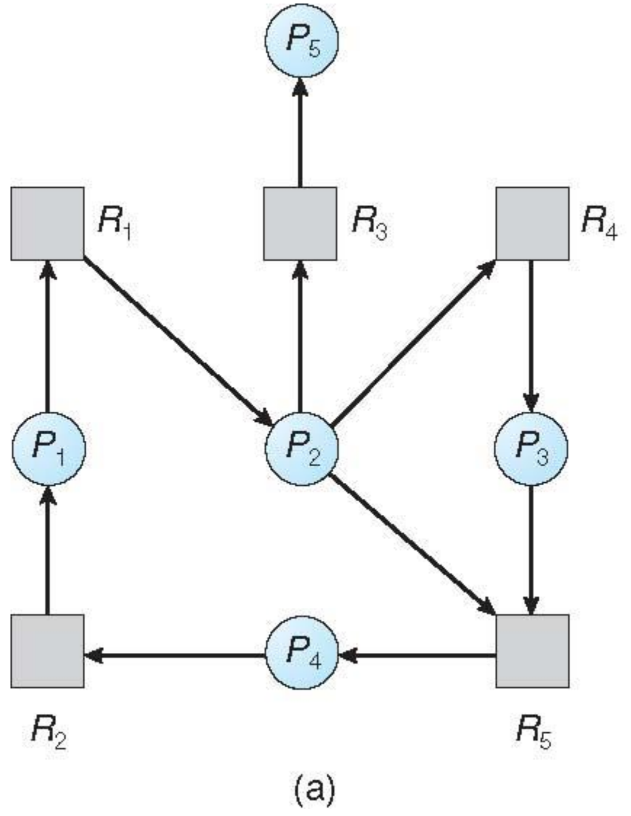
Single Instance of Each Resource Type

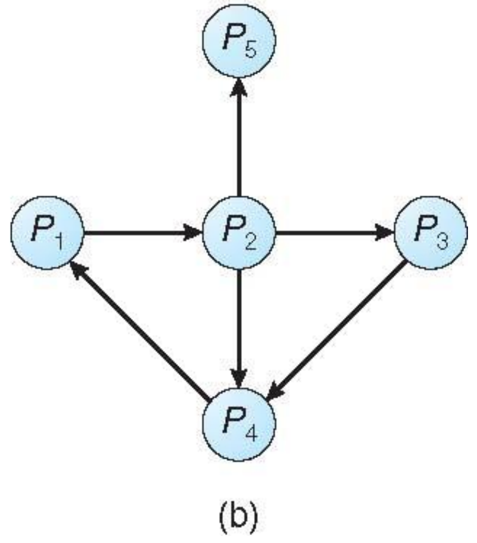
- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i \rightarrow P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$

- 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 ***n*** is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





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 $\boldsymbol{P_i}$ is requesting ***k*** more instances of resource type $\boldsymbol{R_j}$.

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

Detection Algorithm

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

   Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

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**> will result in ***Finish[i] = true*** for all ***i***

Example

- $\boldsymbol{P_2}$ 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?
  - Can reclaim resources held by process $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

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 – criteria for selection to be made - minimize cost
- Rollback – return to some safe state, restart process for that state - some process may lose resources and cannot continue.
- Starvation – same process may always be picked as victim - include number of rollback in cost factor

- Not one solution is best for deadlock handling.
- Better solution can be provided by combining the three approaches namely prevention, avoidance and recovery, allowing the use of optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes and use most appropriate technique for handling deadlocks within each class.