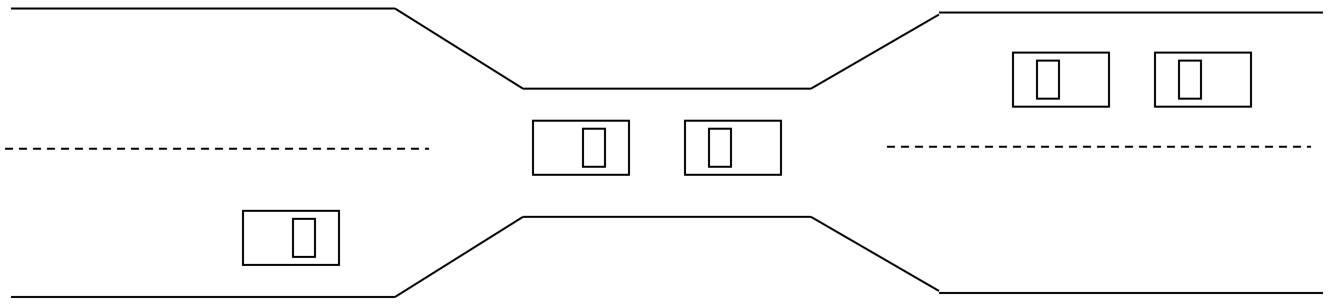
Agenda

- Assignment 3 due November 10
- Feedback for Assignment 2, Midterm on Brightspace
- Seminar on Deep Learning and Music
  - Dr. Sageev Oore, Google Visiting Research Scientist
  - Wednesday 11: 30 am – 1:00 pm
  - ME 107
- Today's lecture
  - Dealing with Deadlocks
    - Prevention
    - Avoidance
    - Detection and Action

Where we are…

- The OS provides three key abstractions
  - Process → CPU
  - Memory (Address) Space → RAM
  - Files → Secondary storage, Network, and Peripheral devices

Deadlock Bridge Crossing Example

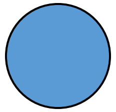


- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

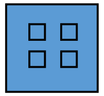
Resource Allocation Graph (RAG)

A set of vertices V and a set of edges E

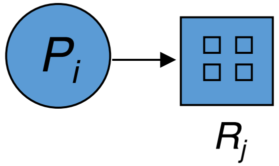
- Process



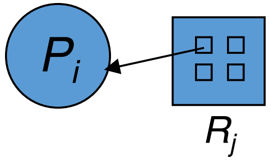
- Resource Type with 4 instances



- P_i requests instance of R_j

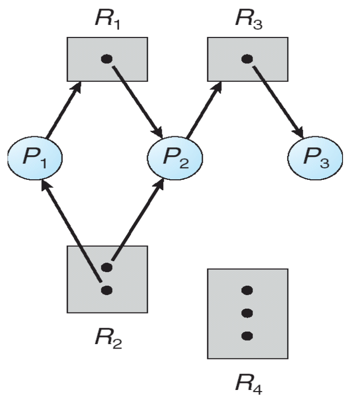


- P_i is holding an instance of R_j

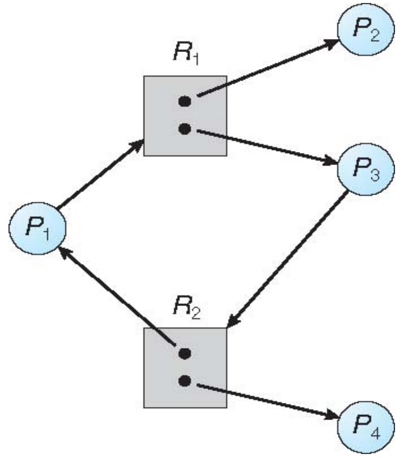


Basic Facts

- 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



Conditions for Deadlock

- **Mutual Exclusion**
  - Only one process may use a resource at a time
  - Processes that request a resource being used are forced to block
- **Hold-and-Wait**
  - A process may attempt to acquire more than one resource.
  - A process may hold allocated resources while awaiting assignment of other resources.
- **No Pre-emption** (no interference from the system)
  - No resource can be forcibly removed from a process holding it
- **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 $P_2, \ldots, P_{n-1}$ 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 Prevention Strategy

Idea: Prevent one of the four conditions from occurring

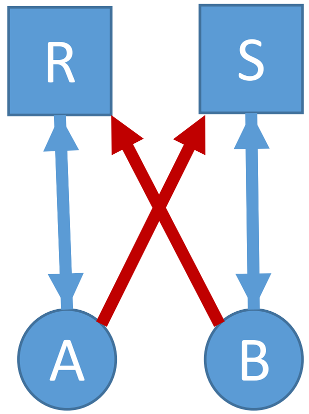
- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Indirect method

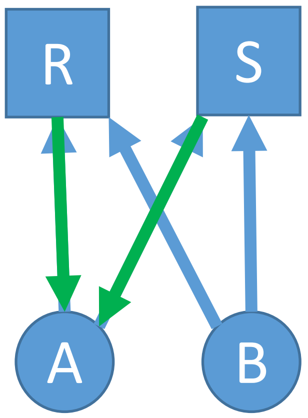
Direct method

Deadlock Condition Prevention

- **Mutual exclusion**
  - In general, nothing to be done
  - Mutual exclusion must be granted for resources that require it
- **Hold and wait**
  - Option 1: require that a process requests and allocates all of its required resources at the start of execution
  - Option 2: processes may not own a resource when requesting a new one
  - Option 3: allow processes to hold only one resource at a time
    - Easy to implement
    - However, some problems may not be solvable this way



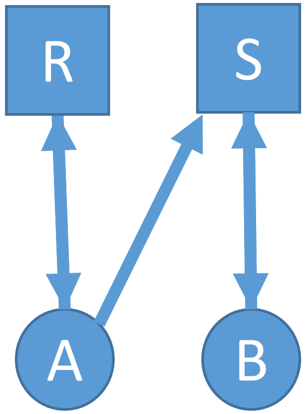
Deadlock Condition Prevention: Hold and Wait



Options 1,2 disadvantages:

- **Starvation:** a process may be held up for a long time waiting for all of its resource request to be filled
- **Low Resource Utilization:** resources allocated to a process may remain unused for a considerable period
- **Practical Problem**: a process may not know in advance all of its resource requirements

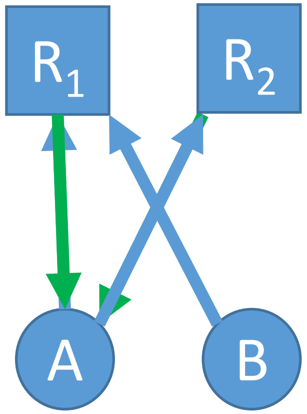
Deadlock Condition Prevention: No Pre-emption



- **Idea**: allow resources to be preempted if they are being held while the process is waiting
  - Option 1: if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource
  - Option 2: if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.
    - Would prevent deadlock only if no two processes possessed the same priority
- **Problem**: some resources cannot/should not be preempted
- Prone to **Starvation** and **Low Resource Utilization**

Deadlock Condition Prevention: Circular Wait

- Idea: allow waiting but prevent circular wait. Can be implemented by
  - defining a linear ordering of resource types
  - Requiring that resources must be acquired in ascending order
- Example: let us associate an index with each resource type
  - Then resource *R i precedes R j in the ordering if i < j .*
  - *Now suppose that two* processes, A and B, are deadlocked because A has acquired R_1 *and requested* R_2, and B has acquired R_2 *and requested* R_1.
  - *This condition is impossible because it* implies *2 < 1 and 1 < 2 .*
- Easy to implement but requires ordering of resources to be hard coded



Deadlock Prevention Strategy

Idea: Prevent one of the four conditions from occurring

Condition	Approach
Mutual Exclusion	Spool Everything
Hold and wait	Request all resources initially
No pre-emption	Take resources away
Circular Wait	Order resources numerically

Indirect method

Direct method

Deadlock Avoidance

- **Motivation**: deadlock prevention strategies lead to inefficient use of resources and inefficient execution of processes.
- Deadlock avoidance
  - Allows more concurrency than prevention
  - Dynamic decisions regarding granting resources and/or starting processes
    - Resources currently available
    - Resources currently in use
    - Future requests and releases of each process
  - Simplest form requires that each process declares the maximum number of resources (of each type) that it may need

Deadlock Avoidance Approaches

Resource Allocation Denial

- Referred to as Banker's algorithm
- Do not grant an incremental resource request if this allocation might lead to deadlock.

Process Initiation Denial

- Do not start a process if its demands might lead to deadlock

Data Structures Used

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. Available resources
  - Available [j] = k means there are k instances of resource type R_j available
- **Max**: n x m matrix. Current requests of each process
  - *Max* [i,j] = k means process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. Current resource allocation to each process
  - Allocation[i,j] = k means P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. Remaining resource needs of processes
  - *Need*[i,j] = k means P_i may need k more instances of R_j to complete its task

    Need [i,j] = *Max*[i,j] – *Allocation* [i,j]

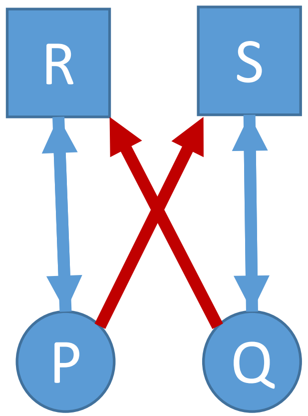
Process Initiation Denial

- All resources are either available or allocated
- No process can request more than the total amount of resources in the system
  - Max[i][j] <= Available[i][j]
- No process is allocated more resources of any type than the process originally claimed to need
  - Allocation[i][j] <= Max[i][j]
- Start a new process P_{n+1} only if
  - (Max[n+1][j] + Sum (Max [i][j])) <= Available [j]
  - for all j, 1<=i<=n

Terminology

- ***State*** of the system reflects the current allocation of resources to processes
- ***Safe state*** is one in which there is at least one sequence of resource allocations to processes that ***does not*** result in a deadlock
  - No cycles in the RAG
- ***Unsafe state*** is one in which there is at least one sequence of resource allocations to processes that ***may*** result in a deadlock
  - Cycles will occur in the RAG
- ***Deadlock state*** is one in which there is at least one sequence of resource allocations to processes that **results** in a deadlock
  - Cycles occur in the RAG

Safe, Unsafe, Deadlocked



- ***Safe state*** : P, Q
- ***Unsafe state :*** *P*
- ***Deadlock state:*** *P, Q*
- ***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.

Algorithms for Determining Safety

- Option 1: Resource Allocation Graph Algorithm
  - Check if admitting the process results in cycles in the graph
  - If yes, don’t admit process
  - If no, admit process
  - This algorithm does not work when resources have ***multiple instances***
- Option 2: Banker’s Algorithm
  - More general algorithm
  - Applicable to resources with multiple instances
  - Similar to algorithms used by banks

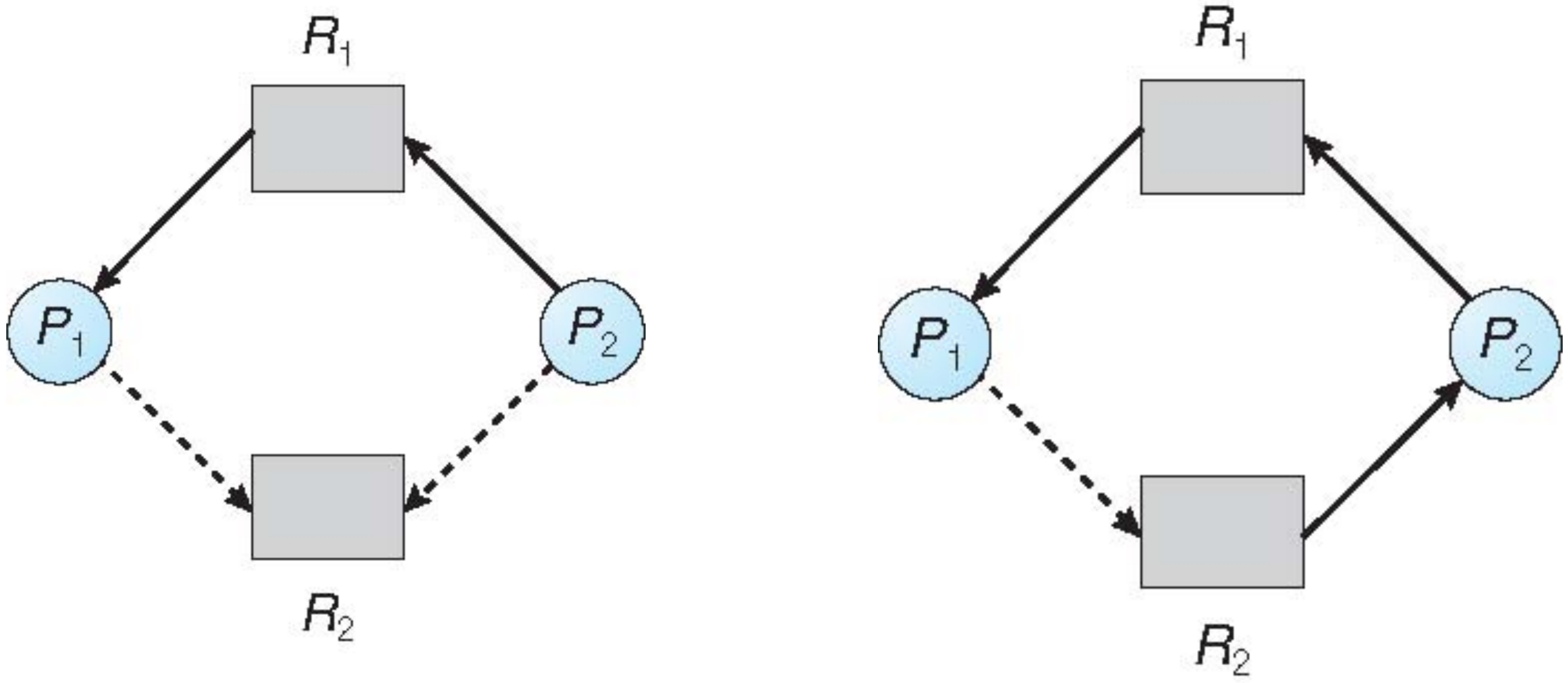
Resource Allocation Graph Scheme

- **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 Scheme

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 of the same resource
- Processes declare
    - maximum number of instances it may need for a particular resource type
    - Sequence of requests of the resources
- Upon a user request for a resource, the system determines whether the allocation of resources will leave the system in a safe state
    - If yes → request is granted
    - Otherwise, process must wait
- When a process gets all its resources it must return them in a finite amount of time
- Used in real-time and embedded systems
- May lead to low utilization and poor user experience!

Data Structures for Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length *m*. Available resources
  - Available [*j*] = *k* means there are *k* instances of resource type R_j available

- **Max***: n x m* matrix. Current requests of each process
  - *Max* [*i,j*] = *k* means process P_i may request at most *k* instances of resource type R_j

- **Allocation***: n* x *m* matrix. Current resource allocation to each process
  - Allocation[*i,j*] = *k* means P_i is currently allocated *k* instances of R_j

- **Need***: n* x *m* matrix. Remaining resource needs of processes
  - *Need*[*i,j*] = *k* means 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

- Step 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
- Step 2: Find an index *i* such that both:
  - *Finish* [*i*] = *false*
  - $Need_i \leq Work$

  If no such *i* exists, go to step 4
- Step 3:
  - $Work = Work + Allocation_i$
  - *Finish*[*i*] = *true*

  go to step 2
- Step 4: the system is in a safe state
  - if Finish[i] == true for all i

Resource Request Algorithm

- **Request** = request vector for process P_i. If
  - $Request_i[j] = k$ means process P_i wants k instances of resource type R_j
- Step 1:
  - If $Request_i \leq Need_i$ go to step 2.
  - Otherwise, raise error condition, since process has exceeded its maximum claim
- Step 2:
  - If $Request_i \leq Available$, go to step 3.
  - Otherwise P_i must wait, since resources are not available
- Step 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 Pi*
- *If unsafe* $\Rightarrow$ *Pi must wait, and the old resource-allocation state is restored*

Example

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	

- The content of the matrix *Need* is defined to be *Max – Allocation*
- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Need
A B C
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

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 1	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 Avoidance

Advantages

- It is not necessary to pre-empt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

Restrictions

- Maximum resource requirement for each process must be stated in advance
- Process under consideration must be independent and with no synchronization requirements
- Fixed number of resources to allocate
- Process holding resources may not exit