# Silberschatz, et al.
# Topics based on Chapter 8

Deadlocks



# Deadlocks

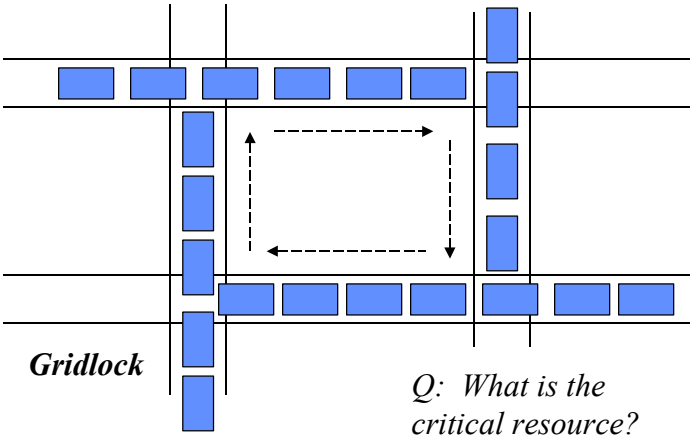
- Deadlocks
  - Definition
  - Prevention
  - Avoidance
  - Detection
- **Definition:**
  A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set

# Deadlock

- Resource model of process and system
  - A system contains a finite number of resources to be distributed among a number of competing processes
  - Resources are partitioned into several types. Each instance of a type is identical to other instances (e.g., CPU, CPU cycles, memory space, files, I/O devices)
  - Processes use resource in only the following sequence
    - Request (if resource is in use then must wait)
    - Use resource
    - Release resource



# Deadlock Example



***Gridlock***

*Q: What is the critical resource?*

# Conditions for Deadlock

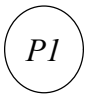
- Four necessary and sufficient deadlock conditions (i.e., all must hold simultaneously)
  - **Mutual exclusion**: At least one resource cannot be shared. Processes claim exclusive control of resources
  - **Hold and wait**: Processes hold resources already allocated to them while waiting for additional resources
  - **No preemption**: Process keeps a resource granted to it until it voluntarily releases the resource
  - **Circular wait**: Circular chain of processes exists in which each holds one or more resources requested by the next process in the chain (implies hold and wait)



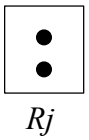
# Strategies for dealing with Deadlock

- Deadlock prevention
  - Construct system in such a way that deadlock cannot happen
- Deadlock avoidance
  - When deadlock could potentially occur, sidestep the deadlock situation
- Deadlock detection/recovery
  - When deadlock occurs, take steps to remove the deadlock situation (e.g., roll back or terminate some process)
- Ignore the problem
  - Don't worry, be happy!

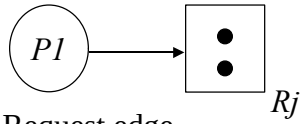
# Resource Allocation Graphs



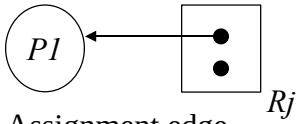

Process




Resource
(2 resource instances in this example)




Request edge

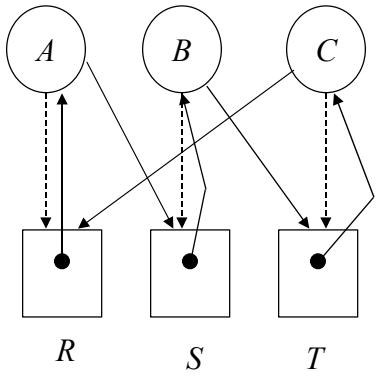



Assignment edge



# Resource Allocation Graph Example

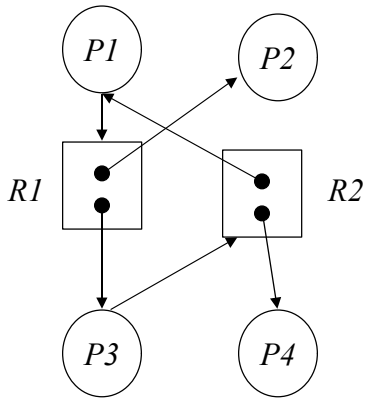
A requests R (granted)
B requests S (granted)
C requests T (granted)
A requests S (wait)
B requests T (wait)
C requests R (deadlock)



# Resource Allocation Graph Example with Multiple Instances

P1 requests R2 (granted)
P2 requests R1 (granted)
P3 requests R1 (granted)
P4 requests R2 (granted)
P1 requests R1 (wait)
P3 requests R2 (wait)

Cycle but no deadlock
(e.g., when P4 completes, P3 can continue)






# Deadlock Detection by Resource Allocation Graph Reduction

- General technique for detecting deadlock: reduction of the resource allocation graph
  - if all of a process' resource requests can be granted, remove the arrows from and to that process (this is equivalent to the process completing and releasing its resources).
  - repeat...
  - if the graph can be reduced by all of its processes then there is no deadlock
  - if not, then the irreducible processes constitute the set of deadlocked processes in the graph

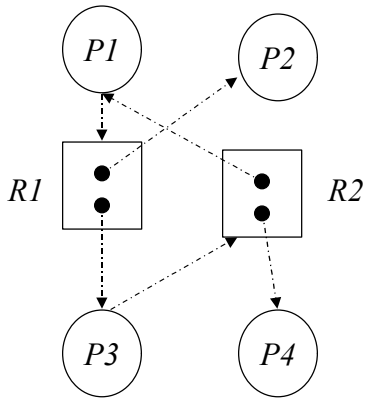
# Reduction of the Resource Allocation Graph

Remove R1-->P2

Remove R2-->P4

Now can grant P3's request for R2 so remove R1-->P3 and P3-->R2

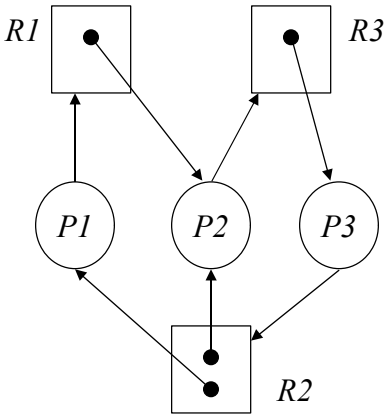
Finally, can grant P1's request for R1 and so remove those arcs






# Deadlock Detection by Resource Allocation Graph Reduction

- Can any arcs be reduced?



# Deadlock Prevention

- Recall the four necessary and sufficient conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Deadlock prevention is achieved by ensuring that at least one condition cannot occur



# Deadlock Prevention by denying Mutual Exclusion

- Perhaps can substitute a sharable resource for a dedicated one in some cases (e.g., readers' access to a file)
- Generally not a useful solution because we need to provide dedicated resources

# Deadlock Prevention by denying Hold and Wait

- Method 1: allocating all the needed resources when starting a process
- Method 2: a process is allowed to request a resource only if it does not hold any resource
- Essentially resource requests are granted by the system on an “all or none” basis
- Problems
  - resources may be left idle for long periods of time
  - starvation possible, especially if a process is requesting several popular resources



# Deadlock Prevention by denying No Preemption

- Method 1: preempting the resources of a process (and then restarting the process or reallocating resources to it) if its request cannot be granted
- Method 2: preempting the resources of a process (and then restarting the process or reallocating its resources) if it holds some resource that is being requested
- Problems
  - State of resource must be saved and restored
  - Hence easy for resources whose state can be saved and restored (e.g., CPU registers, memory space)
  - Difficult for resources whose state cannot be saved and restored (e.g., tape and printers)

# Deadlock Prevention by denying Circular Wait

- Impose total ordering on resource types. Require that process requests resources in an increasing order of enumeration.
- Problems
  - for best efficiency, resource numbers must correspond to expected order of use of resources. If use is out of order, the result is idle resources (waste)
  - Large effect on system programs
    - change in resource numbers may require change in program
    - programmer has to be aware of ordering in structuring program
    - portability of program compromised
- Even so, this method has been used in a number of systems such as IBM MVS and VAX/VMS

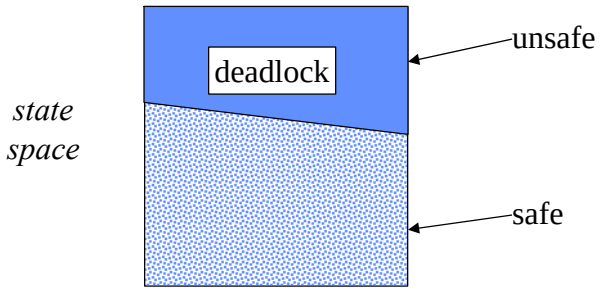


# Deadlock Avoidance

- Main idea
  - Carefully allocate the resources such that the system will not run into deadlock
- More specifically
  - Require additional information about how resources will be requested. Use this information to determine whether to grant an allocation request or to cause the requesting process to wait.
- Costs
  - Run-time overhead of decision making
  - Extra information required from applications

# Deadlock Avoidance
# Safe and Unsafe states

- A system is in a *safe state* if the system can allocate resources to each process (up to its maximum) in some order and let each of them compete successfully (hence, avoiding a deadlock).






# Safe State Example

10 instances of resource, 3 processes. Currently 3 instances of resource are free.

| | current | maximum | needs (max-current) |
|---|---|---|---|
| p1 | 3 | 9 | 6 |
| p2 | 2 | 4 | 2 |
| p3 | 2 | 7 | 5 |

This is a safe state. Can complete by granting resources to p2 then p3 then p1.

Can become unsafe. Grant 2 more instances to p1 initially--no one can now get rest of maximum allocation

Still not deadlocked. That depends on what happens next...

# Deadlock Avoidance
# Safe and Unsafe States

- More formally
  - each process declares a *maximum need*
  - *safe* if there exists a sequence of processes <P1,P2,...,Pn> such that for each Pi in the sequence, 1 <= i <= n, the resources that Pi can still request (i.e., Pi's maximum need - current need) can be satisfied by the currently available resources plus the resources held by all the Pj, j < i
  - *unsafe* state: not safe



# Deadlock Avoidance
# Safe and Unsafe states

- To reiterate, if safe then no deadlock possible. If deadlock, then unsafe. However can be unsafe *without* deadlock.
- Unsafe only implies that some unfortunate sequence of events might lead to a deadlock.
- Note further that a safe state can become unsafe, depending on process behavior.

# Deadlock Avoidance
# Dijkstra's Banker's Algorithm

- Basic notion: don't grant a request if it would cause the state to become unsafe. Instead force the process to wait.
- Safety is checked by repeatedly picking a process that can be allocated all of its resources, and then assuming that the process has received resources and has released them. If this can be done for all processes, then the state is safe.



# Banker's Algorithm

- Given *n* processes and *m* types of resources, define the following data structures:
  - Available: the number of available resources of each type
  - Max: maximum demand of each process on each type resource
  - Allocation: number of resources of each type currently allocated to each process
  - Need: Need[i,j] = Max[i,j] - Allocation[i,j]

Banker's Algorithm

Def: Request[i,j] = The number of resource j requested by Process i

1. **If** Request[i] <= Need[i], **go to** Step 2. Otherwise, report error.
2. **If** Request[i] <= Available, **go to** Step 3. Otherwise, let process i wait: not enough resources available.
3. Pretending to have allocated the request resources to process i:
   Available := Available - Request[i];
   Allocation[i] := Allocation[i] + Request[i];
   Need[i] := Need[i] - Request[i];
4. **If** Safe(Available, Allocation, Need) = false **then**
   Available := Available + Request[i];
   Allocation[i] := Allocation[i] - Request[i];
   Need[i] := Need[i] + Request[i];
   Let Process i wait: allocation may cause deadlock

   **else** let process proceed with allocated resources.



Function Safe(Available, Allocation, Need) : boolean;

1. Work := Available;
   **For** i := 1 **to** n **do** Finish[i] := false;
2. **While** there is an i such that Finish[i] = false **and** Need[i] <= Work
   **do begin**
      Work := Work + Allocation[i];
      Finish[i] := true
   **end**;
3. **If** for all i Finish[i] = true, **then**

      return(true)

   **else**

      return(false);

## Examples of Safe Function

Input of the algorithm

| i | Allocation | | | Need | | |
|---|---|---|---|---|---|---|
| | A | B | C | A | B | C |
| 0 | 0 | 1 | 0 | 7 | 4 | 3 |
| 1 | 3 | 0 | 2 | 0 | 2 | 0 |
| 2 | 3 | 0 | 2 | 6 | 0 | 0 |
| 3 | 2 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 2 | 4 | 3 | 1 |

Available

| A | B | C |
|---|---|---|
| 2 | 3 | 0 |

Trace of the algorithm:

| Work | | | |
|---|---|---|---|
| A | B | C | |
| 2 | 3 | 0 | |
| 5 | 3 | 2 | Finish[1] := true; |
| 7 | 4 | 3 | Finish[3] := true; |
| 7 | 5 | 3 | Finish[0] := true; |
| 10 | 5 | 5 | Finish[2] := true; |
| 10 | 5 | 7 | Finish[4] := true; |

The system is safe.

Safe sequence may not be unique.



## Examples of Safe Function

Input of the algorithm

| i | Allocation | | | Need | | |
|---|---|---|---|---|---|---|
| | A | B | C | A | B | C |
| 0 | 0 | 3 | 0 | 7 | 2 | 3 |
| 1 | 3 | 0 | 2 | 0 | 2 | 0 |
| 2 | 3 | 0 | 2 | 6 | 0 | 0 |
| 3 | 2 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 2 | 4 | 3 | 1 |

Available

| A | B | C |
|---|---|---|
| 2 | 1 | 0 |

Trace of the algorithm:

| Work | | |
|---|---|---|
| A | B | C |
| 2 | 1 | 0 |

No process can finish!

The system is unsafe.

Examples of Safe Function

*P0 requests resources; enough resources are available to meet the request; updating Allocation and Need results in the following:*

Input of the algorithm

| i | Allocation | | | Need | | |
|---|---|---|---|---|---|---|
| | A | B | C | A | B | C |
| 0 | 0 | 3 | 0 | 7 | 2 | 3 |
| 1 | 3 | 0 | 2 | 0 | 2 | 0 |
| 2 | 3 | 0 | 2 | 6 | 0 | 0 |
| 3 | 2 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 2 | 4 | 3 | 1 |

Available

| A | B | C |
|---|---|---|
| 0 | 3 | 0 |

Trace of the algorithm:

| Work | | | |
|---|---|---|---|
| A | B | C | |
| 0 | 3 | 0 | |
| 3 | 3 | 2 | Finish[1] := true; |
| 5 | 4 | 3 | Finish[3] := true; |
| 5 | 4 | 5 | Finish[4] := true; |

Cannot satisfy any other requests, so Safe returns false.

P0 must wait.



# Deadlock Detection

- Must (1) determine that deadlock has occurred and (2) determine how to recover from deadlock
- Deadlock detection: check if a deadlock has happened
  - When do you check?
  - How do you check?
- Deadlock recovery: bring the system back to a non-deadlocked state
  - Invoke when a deadlock has been detected.
  - How does it work?

# Deadlock Detection

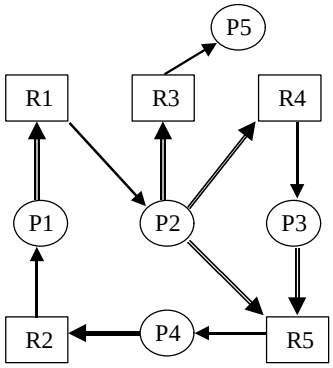
- When do you detect deadlock?
- A deadlock occurs when a process makes a request that cannot be granted.
- Consequently should the system invoke its deadlock detection algorithm whenever ungranted requests are made?
  - One issue is “how soon should a deadlock be detected?
  - Overhead vs. performance. How *many* processes will be affected by deadlock when it happens?
  - Clearly an open-ended system design issue



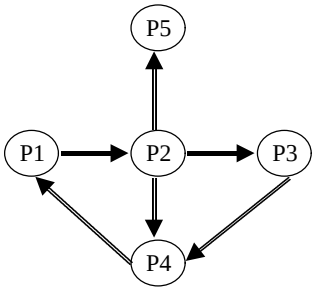
# Deadlock Detection

- How do you detect deadlock?
- If you have only one instance of each resource type, you can use a *wait-for graph*
- Given a resource allocation graph, remove resource nodes and collapse arcs

For the case when each resource type has only one instance, a deadlock exists if and only if there is a cycle in the wait-for graph.




Resource-allocation graph




Wait-for graph



# Detecting Deadlock

- In the wait-for graph, an edge from Pi to Pj implies that Pi is waiting for Pj to release a resource.
- Hence a cycle in the wait-for graph implies deadlock.
- Finding a cycle in a graph is $O(n^2)$ for $n$ processes.

# Detecting Deadlock

- Detecting deadlock when there are multiple instances of a resource cannot use the wait-for graph.
- Use a notion similar to the banker's algorithm safety check
- This algorithm is O($m*n^2$) for $n$ processes and $m$ resources
- Available[m]: number of available resources of each type
- Allocation[n,m]: Number of resources of each type currently allocated to each process
- Request[n,m]: current request of each process for each resource type.



How to detect a deadlock

Function detect_deadlock(Available, Allocation, Request):boolean;

1. Work := Available;
   **For** i := 1 **to** n **do** Finish[i] := false;
2. **While** there is an i such that Finish[i] = false **and**
   Request[i] <= Work
   **do begin**
   Work := Work + Allocation[i];
   Finish[i] := true
   **end**;
3. **If** for all i Finish[i] = true, **then**
   return(false) -- there is no deadlock (now).
   **else**
   return(true); -- there is a deadlock.

# Deadlock Recovery

- Process termination
- Resource preemption (successively preempt resources until deadlock is broken)



# Deadlock Recovery Process Termination

- abort all deadlocked processes
- abort one process at a time until the deadlock cycle is eliminated. (Note that you have to check for deadlock after terminating each process and have to have a metric to decide *which* to terminate
  - what kind of metric?
    - priority of process
    - how long it has computed and how much longer it needs to compute
    - what kind and how many resources does the process have?
    - how many more resources does it need?
    - how many processes will need to be terminated?
    - is this an interactive or a batch process?

# Deadlock Recovery
# Resource Preemption

- selection of victim similar to process termination (these are cost factors); avoid starvation (i.e., avoid always picking the *same* victim)
- rollback of process to some consistent state is necessary (since it no longer possesses resources it once had)



# Combined approach to deadlock handling

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection

  *Allowing the use of the optimal approach for each class of resources in the system*
- Partition resources into hierarchically ordered classes
- Use most appropriate technique for handling deadlocks within each class