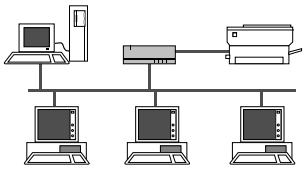


Operating Systems
600.418
Deadlocks

Department of Computer Science
The Johns Hopkins University

Deadlocks

Lecture 4

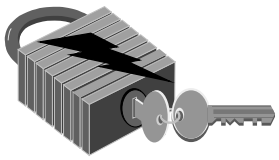
Reading:
- Silberschatz & Galvin
  chapter 7

Additional Reading:
- Stallings
  chapter 6

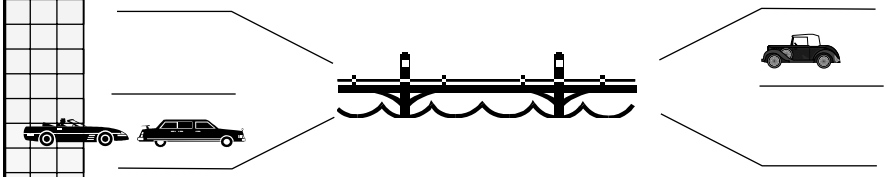
The Deadlock Problem

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.
- **Example**: Semaphores A and B, initialized to 1.

P_0	P_1
wait(A);	*wait*(B);
wait(B);	*wait*(A);



Real-life Example



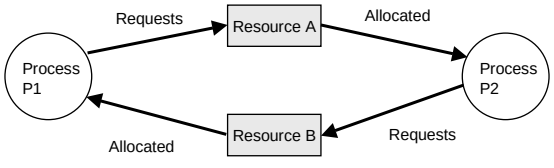
- **Bridge traffic can only be in one direction.**
- **Each entrance of the bridge can be viewed as a resource.**
- **If a deadlock occurs, it can be resolved if one car backs up (resource preemption).**
- **Starvation is possible.**

Conditions for Deadlock

- The following **policy conditions** must be present for a deadlock to be possible:
  - 1. Mutual Exclusion:
    - **Only one process at a time can use a resource.**
  - 2. Hold and Wait:
    - **A process may hold allocated resources while awaiting assignment of other resources.**
  - 3. No Preemption:
    - **A resource can be released only voluntarily by the process holding it (no forced release).**

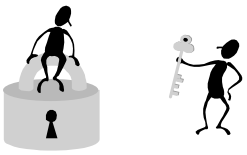
Conditions for Deadlock (cont.)

- In addition, some **specific scenario** must occur for a deadlock to happen:
  - 4. Circular Wait:
    - **A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.**
    - **Alternatively, each process waits for a messages that has to be sent by the next process in the chain.**



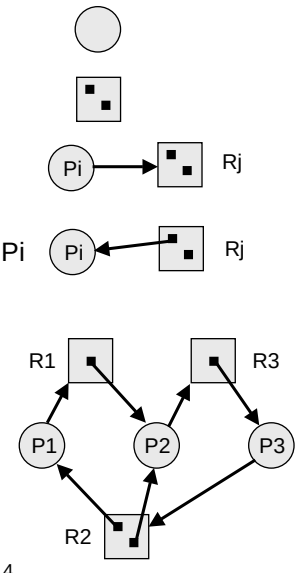
Conditions Analysis

- **Deadlock occurs if and only if the circular wait condition is not resolvable.**
- **The circular wait condition is not resolvable when the first 3 policy conditions hold.**
- Therefore, the four conditions taken together constitute necessary and sufficient conditions for deadlock!



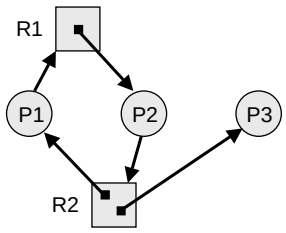
Resource Allocation Graph

- Process
- Resource type with 2 instances
- Pi requests instance of Rj
- An instance of Rj is allocated to Pi
- A graph with a deadlock:



Resource Allocation Graph (cont.)

- A graph with a cycle but without a deadlock:



- If there are no cycles then there is no deadlock.
- If there is a cycle:
  - **If there is only one instance per resource type then there is a deadlock.**
  - **If there is more than one instance for some resource type, there may or may not be a deadlock.**

Methods for Handling Deadlocks

- Deadlock Prevention.
  - Disallow one of the four necessary conditions for deadlock.
- Deadlock Avoidance.
  - Do not grant a resource request if this allocation have the potential to lead to a deadlock.
- Deadlock Detection.
  - Always grant resource request when possible. Periodically check for deadlocks. If a deadlock exists, recover from it.
- Ignore the problem...
  - Makes sense if the likelihood is very low.

The Difference Between Deadlock Prevention and Deadlock Avoidance

- Deadlock Prevention:

  - Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (system design).
  - The goal is to ensure that at least one of the necessary conditions for deadlock can never hold.
- Deadlock Avoidance:

  - The system dynamically considers every request and decides whether it is safe to grant it at this point,
  - The system requires additional apriori information regarding the overall potential use of each resource for each process.
  - Allows more concurrency.

Similar to the difference between a traffic light and a police officer directing traffic.

Deadlock Prevention

Eliminate one of the four conditions:

- Mutual Exclusion:
  - **Well, if we need it then we need it.**
- Hold and Wait:
  - **Require a 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.**
  - **May lead to low resource utilization.**
  - **Starvation is a problem - a process may be held for a long time waiting for all its required resources.**
  - **If it needs additional resources, it releases all of the currently held resources and then requests all of those it needs (the application needs to be aware of all of the required resources).**

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.**
  - **The state of preempted resources has to be saved and later restored. Not practical for many types of resources (e.g. printer).**
- Circular Wait:
  - **Impose a total ordering on all resource types.**
  - **Require each process to request resources only in a strict increasing order.**
  - **Resources from the same resource type have to be requested together.**

Deadlock Avoidance

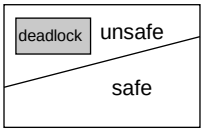
- Maximum requirements of each resource must be stated **in advance** by each process.
- Two approaches:
  - Do not start a process if its maximum requirement might lead to deadlock.
  - Do not grant an incremental resource request if this allocation might lead to deadlock.
- Two algorithms:
  - One instance per resource type - Resource Allocation Graph algorithm.
  - Multiple instances per resource type - The Banker's algorithm.

Safe State

- A system is in a safe state only if there exists a safe sequence of processes P_1, P_2, …, P_n where:
  - **For each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources help by all P_j, *j<i*.**
- If a system is in safe state, there is no deadlock.
- If the system is deadlocked, it is in an unsafe state.
- If a system is in unsafe state, there is a possibility for a deadlock.
- Avoidance: making sure the system will not enter an unsafe state.

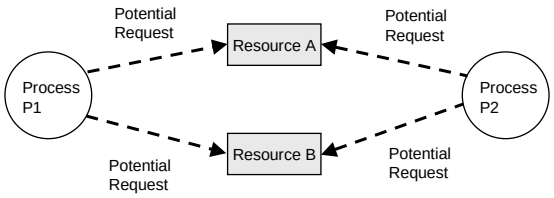
Safe State

- A system is in a safe state only if there exists a safe sequence of processes P_1, P_2, …, P_n where:
  - **For each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources help by all P_j, *j<i*.**
- If a system is in safe state, there is no deadlock.
- If the system is deadlocked, it is in an unsafe state.
- If a system is in unsafe state, there is a possibility for a deadlock.
- Avoidance: making sure the system will not enter an unsafe state.



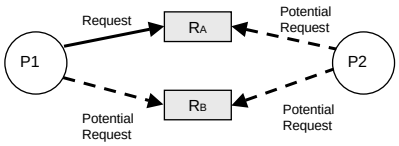
Resource Allocation Graph Algorithm

- Maintains a graph with a directed edge from each process to each resource it might request. (needs apriori knowledge).
- Allocated resource reverses the edge direction.
- Released resource returns the edge to its original direction.
- The algorithm allocates a resource only if it can do that without creating a cycle in the graph.



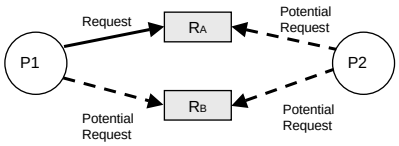
Resource Allocation Graph Algorithm (cont.)

A

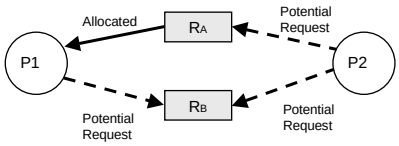


Resource Allocation Graph Algorithm (cont.)

A

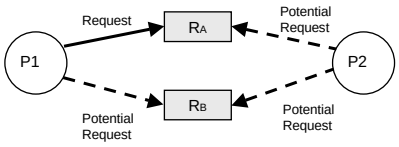


B

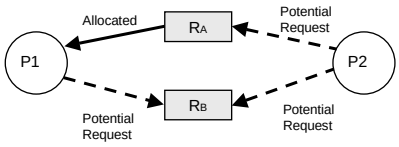


Resource Allocation Graph Algorithm (cont.)

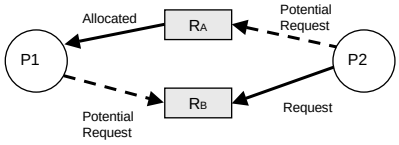
A



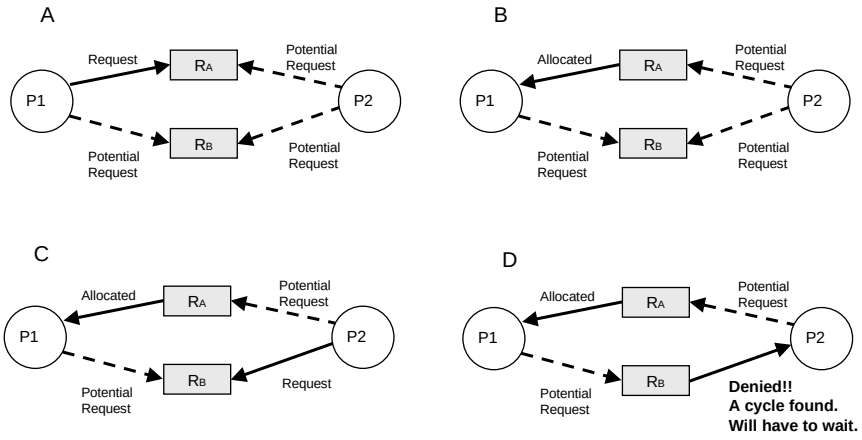
B



C



Resource Allocation Graph Algorithm (cont.)



Note: This operation is done per resource request.
Complexity: finding a cycle in a graph.

The Banker's Algorithm

Handles multiple instances for resource types.

.

n = number of processes*; m* = number of resource types;

Data Structures:

- *Available*: vector [1..m], Available[j] - the number of instances currently available for resource j.
- *Max*: matrix[1..n, 1..m], Max[i, j] - the maximum number of instances of resource j that process i can request at any one time.
- *Allocation*: matrix[1..n, 1..m] - process i currently holds Allocation[i, j] instances of resource j.
- *Need*: matrix[1..n, 1..m] - process i may need additional Need[i, j] instances of resource j.

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

Banker's Algorithm - Safety Procedure

Local Data Structures:

- *Work*: vector [1..m], initialize Work to Available.
- *Finish*: vector[1..n], initialized to *false* for each process i.

1. Find process i such that *Finish*[i] = *false* and $Need_i \leq Work$
   if i exists do
   Work := *Work* + $Allocation_i$
   Finish[i] := *true*
   Go back to 1.
2. (no such i exists)
   if *Finish*[i] = *true* for all i in 1..n, then the system is in a safe state. Otherwise, the processes whose index is false may potentially be involved in a deadlock in the future.

Banker's Algorithm - Resource Request

$Request_i$ - request vector - the number of additional instances for each resource type process i requests at this time.

1. If not ($Request_i \leq Need_i$) raise an error - process i tries to get more resources than what it declared.
2. If not ($Request_i \leq Available$) process i must wait - no sufficient resources at this time.
3. Tentatively allocate the requested resources to process i:
   Available := *Available* - $Request_i$
   $Allocation_i$:= $Allocation_i$ + $Request_i$
   $Need_i$:= $Need_i$ - $Request_i$
4. Check safety of state. If safe, the resources are allocated.
   If not safe then cancel the tentative allocation and process i must wait.

Banker's Algorithm - Example

- 5 processes and 3 resource types A (with 10), B (with 5), and C (with 7 instances).
- A Snapshot:

Available		Allocation	Max
A B C		A B C	A B C
3 3 2	P_0	0 1 0	7 5 3
	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

- Is this a safe state?

Banker's Algorithm - Example

- 5 processes and 3 resource types A (with 10), B (with 5), and C (with 7 instances).
- A Snapshot:

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

$<P_1, P_3, P_0, P_2, P_4>$ is a safe sequence.

Banker's Algorithm - Example (cont.)

- Suppose P_1 now requests (1, 0, 2).

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

$<P_1, P_3, P_0, P_2, P_4>$ is a safe sequence also in this case.

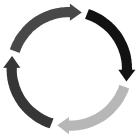
What if P_4 then requests (3,3,0) ?
And if P_0 requests (0,2,0) ?

Deadlock Detection

- The system may enter a deadlock state.
- The system needs:
  - An algorithm that periodically determines whether a deadlock has occurred in the system.
  - A procedure to recover from a deadlock.
- Two algorithms:
  - One instance per resource type.
  - Multiple instances per resource type.

Single Instance for Each Resource Type

- Maintain a wait-for graph:
  - Nodes are processes.
  - If process i is waiting for a resource held by process j then there is an edge from i to j.
  - Exactly the same as a resource allocation graph but optimize it for the search by collapsing edges.
  - Similar optimization could be done also for the resource allocation graph algorithm.
- Periodically invoke an algorithm that searches for cycles in the graph.
- Complexity (in wait-for graph): n^2



Several Instances of a Resource Type

- Similar to the Banker's algorithm safety procedure with the following semantics differences:
  - Replacing *Need* by *Waiting* where *Waiting*$_i$ is the actual vector of resources process i is currently waiting to acquire.
  - May be slightly optimized by initializing *Finish*[i] to *true* for every process i where *Allocation*$_i$ = 0.
  - This is valid because we only care whether there is a deadlock right now. We are optimistic. If processes will need more resources in the future, this might lead to deadlock and we will discover it in the future.
  - Processes with a *false* entry in the end are the ones actually involved in a deadlock at this time.
- Complexity (in wait-for graph): $m * n^2$

Detection Algorithm Usage

When or how often to invoke ?

- In the extreme case - every time a request for resource cannot be granted immediately.
  - This comes with enormous cost (think about complexity).
- Reasonable alternative - periodically. But what period to use?
  - How long are we willing to wait after it happens to be detected.
  - How much system resources are we willing to commit to the detection.

Recovery from Deadlock

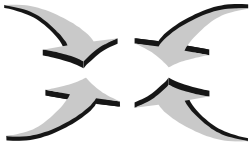
What can the system do if it discovers a deadlock?

- Process Termination
  - **Abort all deadlocked processes:**
    - Fast
    - A lot of process work is lost.
  - **Abort one deadlocked process at a time and check for deadlocks again:**
    - More work to resolve a deadlock.
    - Better in terms of process work.
    - What is a good order to abort processes?
- Resource Preemption
    - what is a good way to select a victim
    - How can we rollback and then recover from preemption?
    - How can we protect from starvation

An Integrated Deadlock Approach

In a practical system, some of the previous approaches can be integrated.

- **Group Resources into a number of different classes and order them.**
- **Use prevention of circular wait to prevent deadlock between resource classes.**
- **Use the best approach for each class to handle deadlocks within each class.**



Homework

There are 4 processes in the system: A, B, C, D.
The system has to allow operation in each of the following 3 cases and only in them.
(1) Any single process.
(2) The pair A and C.
(3) The pair B and D.
The other process in a pair can join even if its buddy is already **operating**.

All semaphores are initialized to 1

Process A	Process B	Process C	Process D
repeat	*repeat*	*repeat*	*repeat*
wait(x);	*wait(y);*	*wait(z);*	*wait(w);*
wait(y);	*wait(z);*	*wait(w);*	*wait(x);*
operate	*operate*	*operate*	*operate*
signal (y);	*signal(z);*	*signal(w);*	*signal(x);*
signal(x);	*signal(y);*	*signal(z);*	*signal(w);*
forever	*forever*	*forever*	*forever*

Homework (cont.)

1.

1.1 Demonstrate that the original code does not meet the requirements.

1.2 The problem in 1.1 could be fixed by adding a single semaphore (e.g. v) that protects each of the double-waits as a critical section. Detail this solution (write the code). What are the limitations of such a solution? Demonstrate.

1.3 Without adding semaphores to the original code, propose a solution that meets the requirements and does not suffer from the weaknesses of the solution proposed in 1.2

2. (only for CS418)

Please modify the reader/writers solution (lecture 3 slide 27) to be a fair solution: Neither readers get priority nor writers get priority. i.e. even if readers keep coming, writers will not wait for ever, and even if writers keep coming, readers will not wait forever. Still several readers should be able to read simultaneously.

You can assume that all semaphores are FIFO semaphores.

The simpler the solution the better. So please come with the simplest possible solution.

Submit homework in class, Wednesday October 18, 2000.
Homework has to be typed (rather than written by hand).