Deadlock

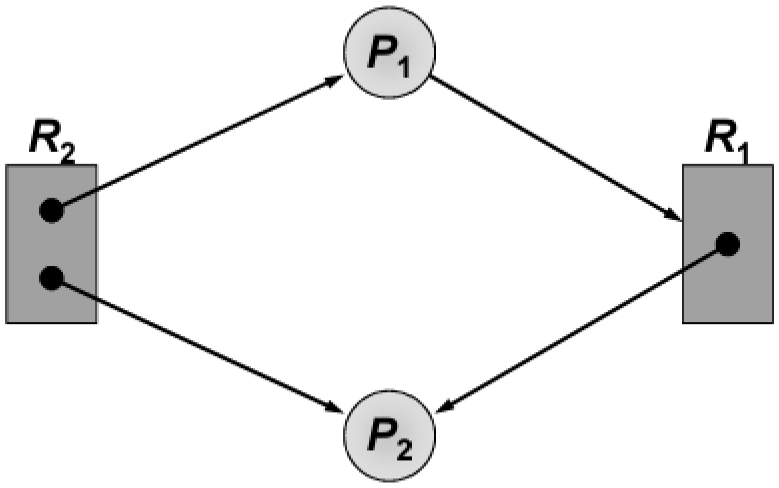
System model

- The system comprises m types of resources $R = \{R_1, R_2, \ldots, R_m\}$
- For each resource type there can be a number of identical instances (units).
- The instances are requested by the processes $P = \{P_1, P_2, \ldots, P_n\}$
- Classification of resources from the viewpoint of a deadlock:
    - Reusable resources

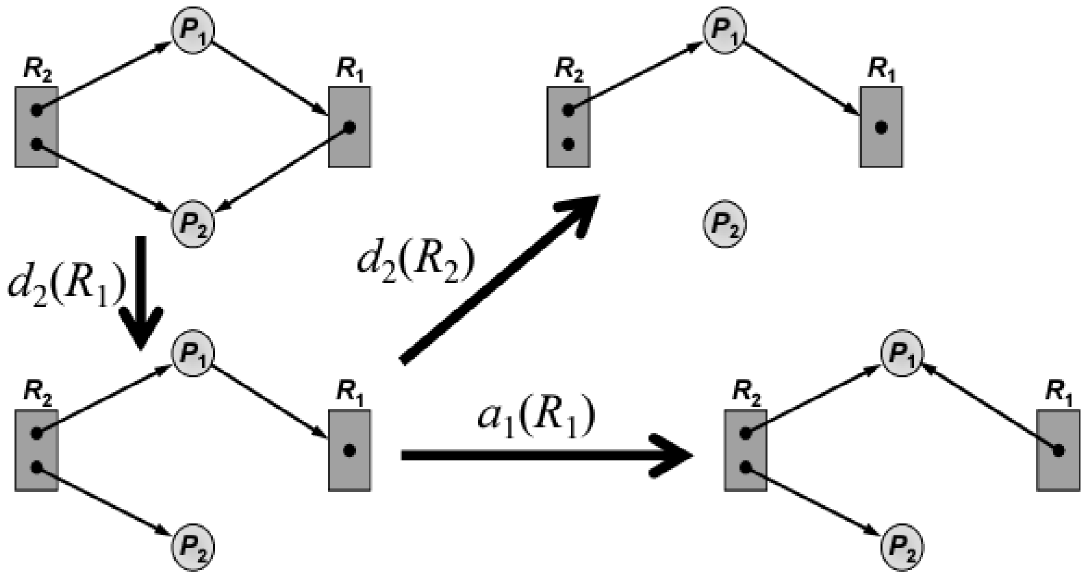
      For the reusable resources the number of unit is fixed and each unit is either allocated or free. A unit can be released only if it has been allocated. A process has to request this resource (which may block), use it and then release it. Examples: memory, I/O channel.
    - Consumable resources

      For the consumable resources there are two types of processes that can utilize it: requesting processes and producing processes. A requesting process never releases allocated units. A producing process can make (produce) more units of that resource without allocating this resource prior. The number of units is thus unbounded and may vary. A requesting process first needs to request this resource (which may block) and then use it. A producing process can only release it. Examples: signal or message sent from one process to another.
- A deadlock can occur if the following conditions are fulfilled:
    - For reusable resources:
        - Mutual exclusion — each resource is either currently assigned to exactly one process or is available.
        - Hold and wait — there is a process which is holding a unit of that resource and at the same time is requesting another unit.
        - No pre-emption — resources previously granted cannot be forcibly taken away from a process.

        - Circular waiting — there must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.
    - For consumable resources:
        - Mutual exclusion — each resource is either currently taken by exactly one process or is available.
        - Hold and wait — a process waiting for a unit cannot release (produce) new units.
        - No pre-emption — resources previously granted cannot be forcibly taken away from a process.
        - Circular waiting — there must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.
- A useful tool in characterizing the allocation of resources to processes is the resource allocation graph.
    - The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. A circle represents a process and a square represents a resource.
    - Within a resource node, a dot is shown for each instance of that resource.
    - A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted. A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted.



- For reusable resources we define three events that can happen to such a resource as:
  - Request (by P_i) — r_i
  - Acquisition (by P_i) — a_i
  - Release (by P_i) — d_i
- We can visualize how the graph changes with those events:



- Definition of a deadlock:

    - A process, say P_i is blocked in a system state σ if it cannot cause a transition out of σ (in other words, the process is incapable of changing the current state of the system, formally: all admissible events are in other processes than P_i).
    - A process, P_i is deadlocked in a system state σ if it is blocked in σ and in each state reachable from σ.

Deadlock detection

- Under the deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected.

Graph based approach

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, and edge from P_i to P_j in a wait-for graph exists if and only if the corresponding resource-allocation graph contains two edges $P_i \to R_q$ and $R_q \to P_j$.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle
- There is a more complicated version of this algorithm for resources with multiple instances that involves knot detection, however I couldn't find much info on it.

Graph reduction

- This is another technique for detecting deadlock.
- There is not much info on it, but from what I understand:

1. Find a process in resource allocation graph that either has no requests, or all requests can be granted, meaning all resources it requested have the appropriate number of instances.
2. Remove that process and all edges connected to/from it.
3. Repeat from step 1
4. If any processes are left, they are deadlocked

- This supposedly works for any kind of resource allocation graph.

Matrix based approach

- This approach is applicable even with multiple instances of a single resource.
- We have n processes $P_1, \ldots, P_n$
- There are m types of resources, for each class (type) of resource there are E_i instances of that resource with $1 \leq i \leq m$.
- $\mathbf{E}$ is the existing resource vector, the ith entry of $\mathbf{E}$ tells us how many instances of the ith resource there are.
- $\mathbf{A}$ is the available resource vector, the ith entry of $\mathbf{A}$ tells us how many instances of the ith resource are currently available (not allocated).
- $\mathbf{C}$ is the current allocation matrix, the ith row of $\mathbf{C}$ tells us how many instances of each resource type the ith process holds.
- $\mathbf{R}$ is the request matrix, the $\mathbf{R}_{ij}$ entry is the number of instances of resource j that P_i wants.
- We define a relation on two vectors $A \leq B$ to hold if and only if $A_i \leq B_i$ for all $i : 1 \leq i \leq m$
- The algorithm marks processes as it goes, all processes initially are said to be unmarked. When the algorithm terminates, any unmarked process is known to be deadlocked.
- Then the algorithm is as follows:
  1. Find an unmarked process P_i for which the ith row of $\mathbf{R}$ is less than or equal to $\mathbf{A}$

    2. If such a process is found, add the ith row of $\mathbf{C}$ to $\mathbf{A}$, mark the process, and go back to step 1.
    3. If no such process exists, the algorithm terminates.
- When the algorithm finishes, all the unmarked process, if any, are deadlocked.

Recovery from deadlock

- Abort of all deadlocked processes
- Abort of one process at a time until the deadlock cycle is eliminated, taking into account:
    - Priority of the process
    - Current computation time computer, and the time necessary to completion
    - Allocated resources
    - Resources necessary to completion

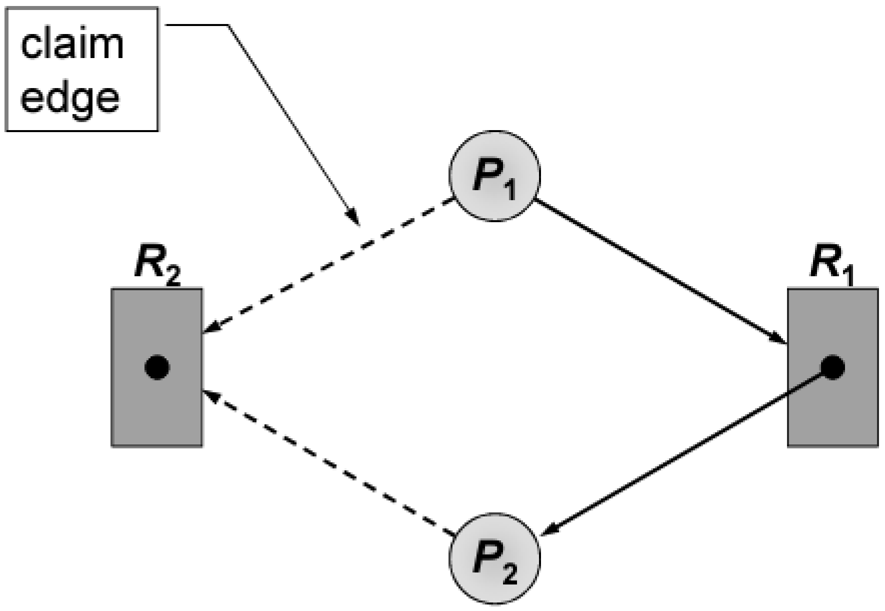
Deadlock avoidance

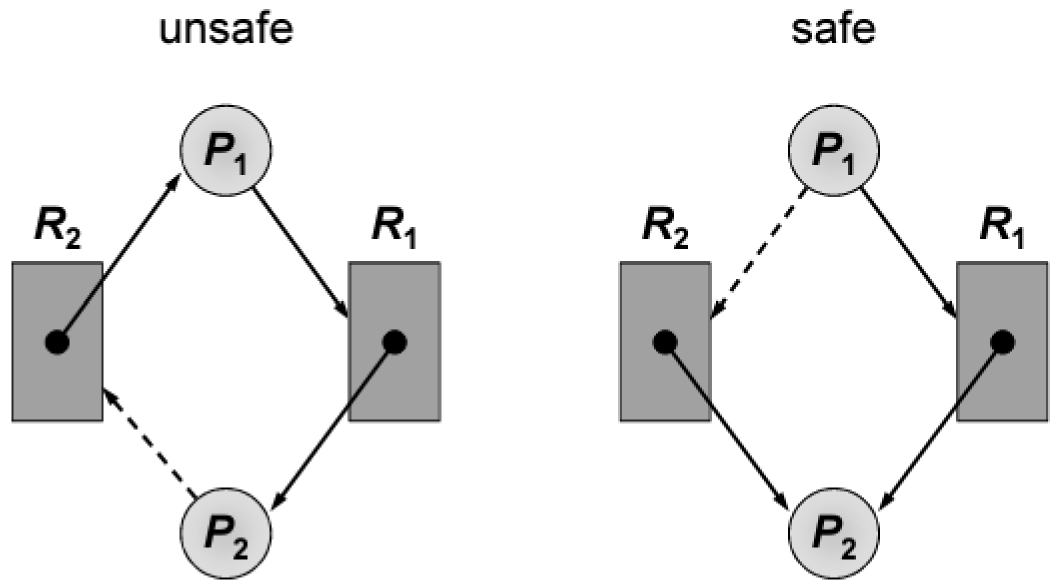
- In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once.
- In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and make the allocation only when it is safe.
- This is called deadlock avoidance. If our system has required information about resource requests it can avoid deadlocks.
- The idea is to preserve a safe state of the system.
- The state of the system is said to be safe if there exists a safe sequence of scheduling of all processes. Let’s break it down what it means. For each process we need to know how many instances of each resource it has and how many instances of each resource it can request at maximum if we decide to run it. If we schedule to run a process we assume it will request the maximum number of instances, and if we don’t have this many instances at the

moment, the sequence that lead to scheduling this process is unsafe. If a scheduled process requests a number of instances that can be supplied then it completes successfully and returns all the resources to our system. So a safe state is a state in which we can schedule all the processes so that when we assume they will request the maximum number of instances of a resource, still all the request will be possible to fulfil.

Graph based approach

- It is possible to determine if the current state is safe using a graph based algorithm. This algorithm is only limited to resources with single instances.
- We use the resource-allocation graph with a modification — we introduce a new type of edge, the claim edge. A claim edge indicates that a process may request a resource at some time in the future. A process can request a resource only if there is a claim edge from that process to that resource, and then that edge turns into a request edge. After the release of a resource the request edge turns back into a claim edge.
- Our current state is safe if and only if there is no cycle or knot in our modified resource allocation graph.





Banker's algorithm

- I could find a lot of information on it, but it seems like the Banker's algorithm is exactly the matrix based algorithm described in the deadlock detection section.

Deadlock prevention

- Deadlock prevention consist of ensuring that at least one of the necessary condition cannot hold:
  - Mutual exclusion — not required for sharable resources; must hold for non-sharable resources.
  - Hold and wait — a process can request a resource only if it does not hold other resources.
  - No pre-emption — a process holding a resources unit can be pre-empted when it requests (and waits for) other resources.
  - Circular waiting — impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.