Deadlock Avoidance

Deadlock Avoidance -Safe state

- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

- A state is *safe if the system can allocate resources to each process (up to its* maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a **safe sequence.**

- A sequence of processes *<P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently* available resources plus the resources held by all *Pj, with j <i.*

Safe state

- If the resources that *Pi needs are not immediately available, then Pi can wait* until all *Pj have finished.*

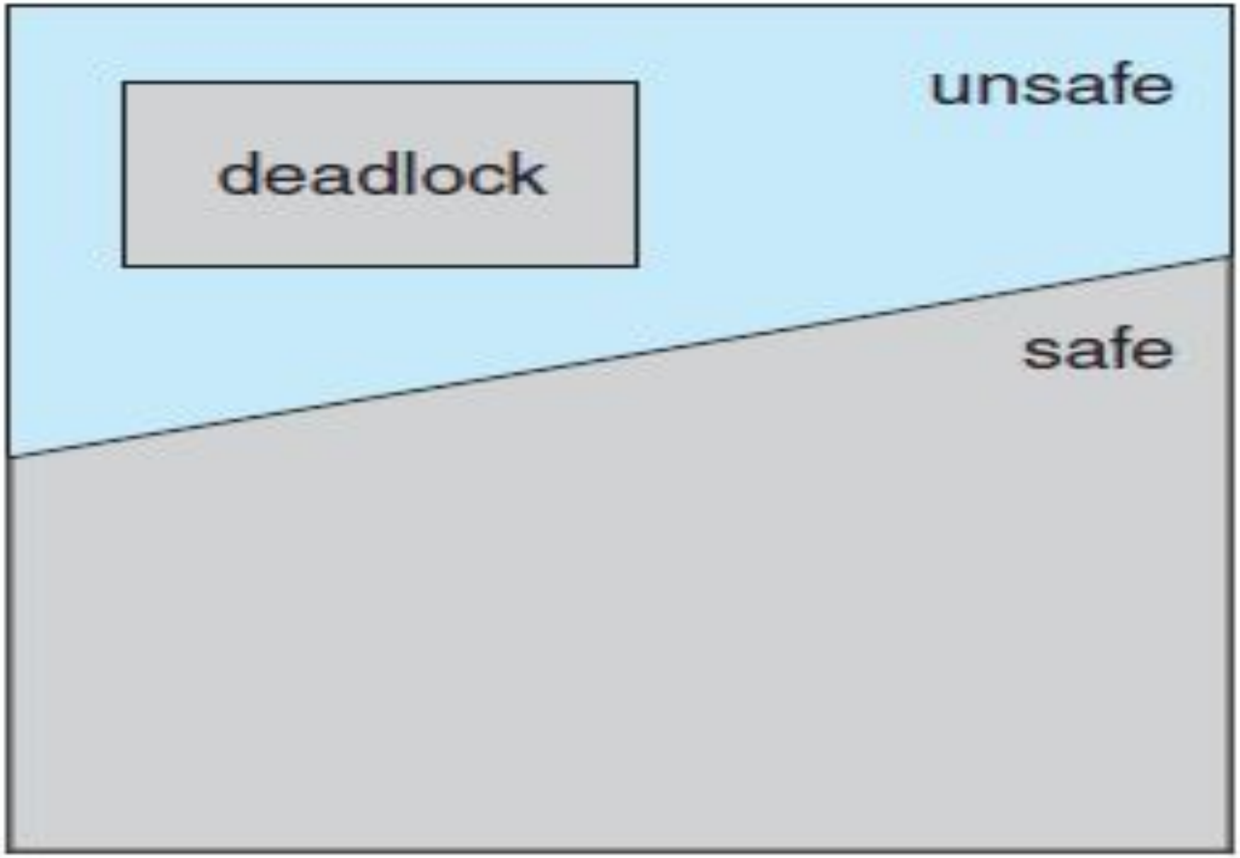
- *When they have finished, Pi can obtain all of its* needed resources, complete its designated task, return its allocated resources, and terminate.

- When *Pi terminates, Pi+1 can obtain its needed resources, and* so on. If no such sequence exists, then the system state is said to be *unsafe.*

Safe state

- A safe state is not a deadlocked state.
- A deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state *may lead to a deadlock.*
- *As long as the state is safe, the* operating system can avoid unsafe (and deadlocked) states.
- In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.

Safe state



Safe, unsafe, and deadlocked state spaces.

Safe state

- Consider a system with twelve magnetic tape drives and three processes: *P0, P1, and P2.*
- *Process P0 requires ten tape drives, process P1* may need four tape drives, and process *P2 may need up to nine tape* drives.
- Suppose that, at time *t0, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus,* there are three free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Safe state

- At time *t0, the system is in a safe state. The sequence <P1, P0, P2> satisfies* the safety condition.

- Process *P1 can immediately be allocated all its tape drives* and then return them (the system will then have five available tape drives).

- Then process *P0 can get all its tape drives and return them (the system will then* have ten available tape drives).

- Finally process *P2 can get all its tape drives* and return them (the system will then have all twelve tape drives available).

Safe state

- A system can go from a safe state to an unsafe state.
- Suppose that, at time *t1, process P2 requests and is allocated one more tape drive. The system is no* longer in a safe state.
- At this point, only process *P1 can be allocated all its tape* drives.
- When it returns them, the system will have only four available tape drives. Since process *P0 is allocated five tape drives but has a maximum of ten,* it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P2 may request six additional tape* drives and have to wait, resulting in a deadlock.
- *If we had made P2 wait* until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Safe state

- By using the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state.
- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.

Deadlock Avoidance Algorithms

1)Resource-Allocation-Graph Algorithm

Used for single instance of each resource types

2)Bankers Algorithm

Used for system with multiple instances of each resource type.

Resource-Allocation-Graph Algorithm

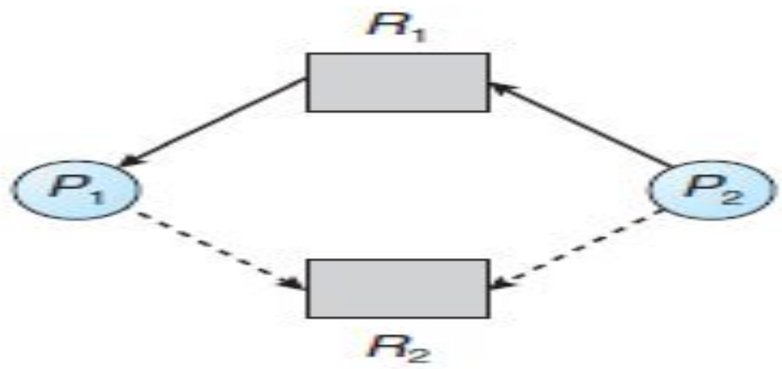
Claim edge

- A claim edge *Pi → Rj indicates that process Pi may request resource Rj at* some time in the future.
- This edge is represented in the graph by a dashed line.
- When process *Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge.*
- *Similarly, when a* resource *Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a* claim edge *Pi → Rj* .

Resource-Allocation-Graph Algorithm

- Suppose that process *Pi requests resource Rj.*
- *The request can be* granted only if converting the request edge *Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation* graph.
- We check for safety by using a cycle-detection algorithm.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state

Resource-Allocation-Graph Algorithm_Example



Resource-allocation graph for deadlock avoidance.

- Suppose that *P2 requests R2.*
- *Although R2 is currently free, we* cannot allocate it to *P2, since this action will create a cycle in the graph.*
- A cycle, indicates that the system is in an unsafe state.
- If *P1* requests *R2, and P2 requests R1, then a deadlock will occur.*

Banker's Algorithm

- **The name was chosen because** the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm

- Let *n is the number of* processes in the system and *m is the number of resource types.*

Data structures used d to implement the banker's algorithm

- **Available.** A vector of length m indicates the number of available resources of each type. If *Available*[j] equals k, then k instances of resource type R_j are available.
- **Max.** An $n \times m$ matrix defines the maximum demand of each process. If *Max*[i][j] equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If *Allocation*[i][j] equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need.** An $n \times m$ matrix indicates the remaining resource need of each process. If *Need*[i][j] equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that *Need*[i][j] equals *Max*[i][j] − *Allocation*[i][j].

Banker's Algorithm

Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.
2. Find an index 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.

Banker's Algorithm

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process 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 the resulting resource-allocation state is safe, the transaction is completed, and process *Pi is allocated its resources.*
- *If the new state* is unsafe, then *Pi must wait for Request*$_i$, *and the old resource-allocation* state is restored.

Banker's Algorithm-Example

To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types *A*, *B*, and *C*. Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	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	

- Suppose now that process *P1 requests one additional instance of resource type A and two instances of* resource type *C, so Request1 = (1,0,2). Check whether this request can be granted* immediately ?