Operating Systems

Deadlocks-Part2

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Outline

- Liveness
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - **Deadlock prevention**
  - **Deadlock avoidance**

- Allow the system to enter a deadlock state and then recover.

- Ignore it and pretend that deadlocks never occur in the system.

Deadlock Prevention

Invalidate ***one of the four*** necessary conditions for deadlock

Deadlock Prevention-Mutual Exclusion

- **Not required for sharable resources** (e.g., read-only files)

- **Must hold** for non-sharable resources

Deadlock Prevention- Hold and Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources.
- Require 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 allocated to it.
- Low resource utilization; starvation possible.

Deadlock Prevention-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.

- Preempted resources are added to the list of resources for which the process is waiting.

- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock Prevention- Circular Wait

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.

Circular Wait

- If:
  `first_mutex = 1`
  `second_mutex = 5`
  code for **`thread_two`** could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Deadlock Avoidance

Requires that the system has some additional

a priori information available.

Deadlock Avoidance

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there **can never be a circular-wait condition**.
- Resource-allocation ***state*** is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- System is in **safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

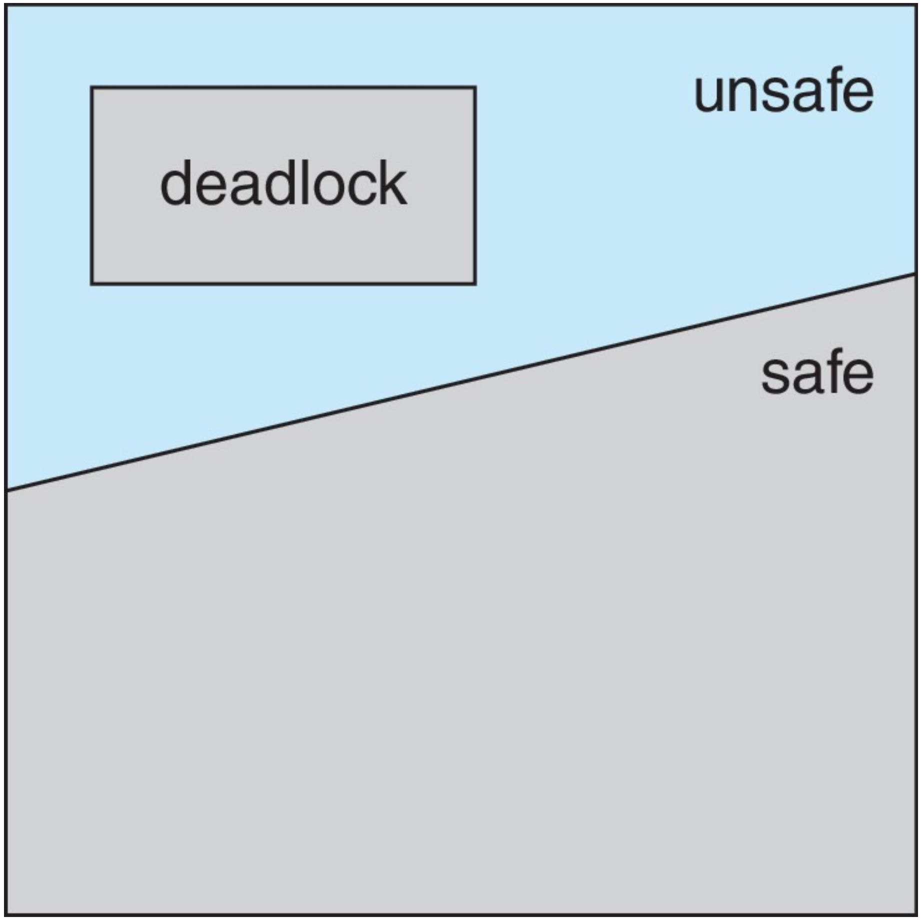
Safe State (cont.)

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

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.

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the Banker's Algorithm

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.

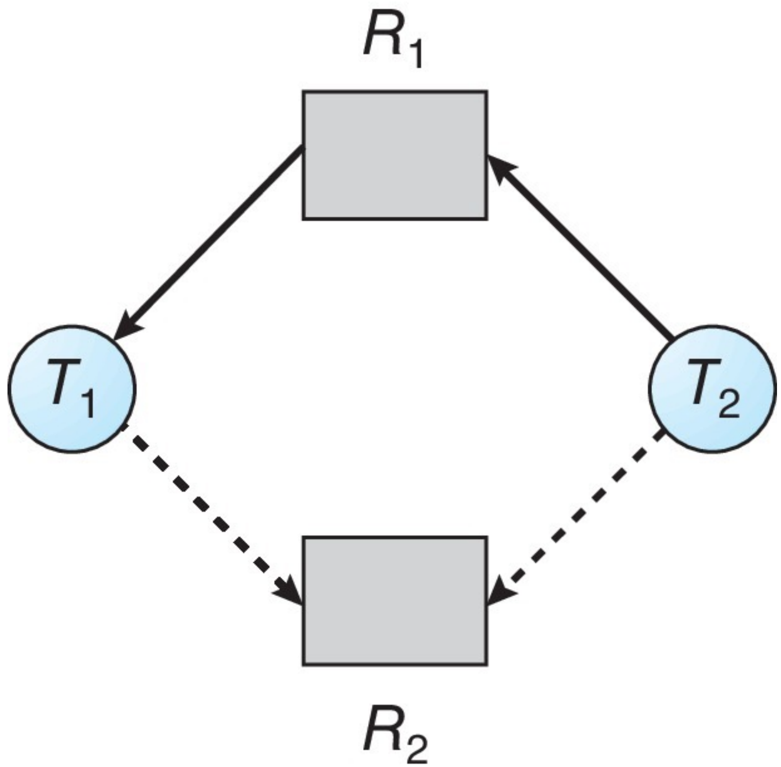
- Request edge converted to an assignment edge when the resource is allocated to the process.

Resource-Allocation Graph Scheme (cont.)

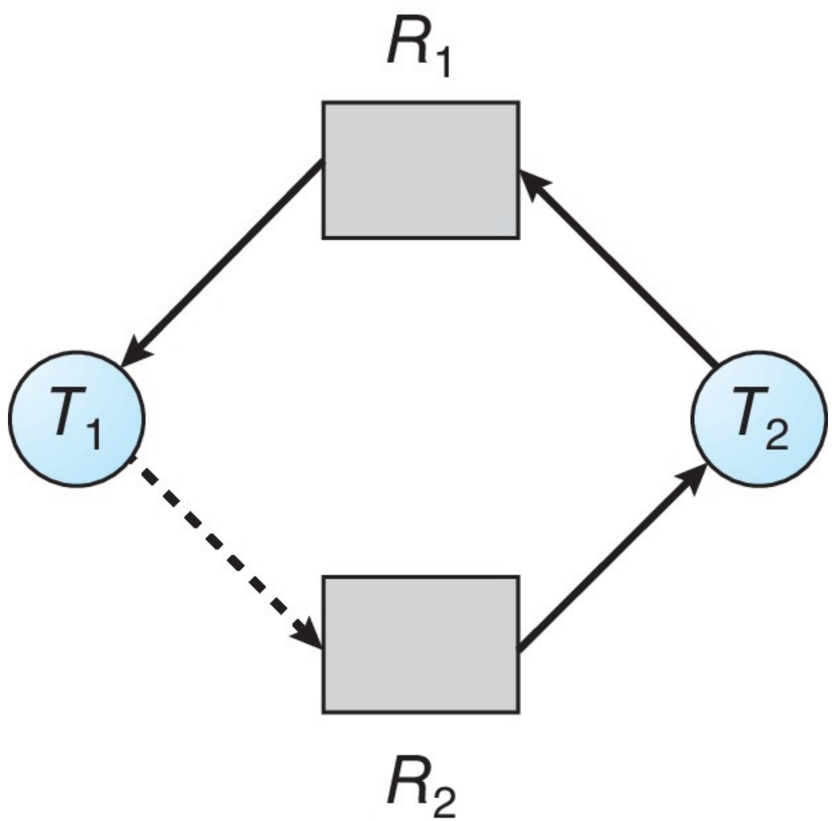
- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed a priori in the system.

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



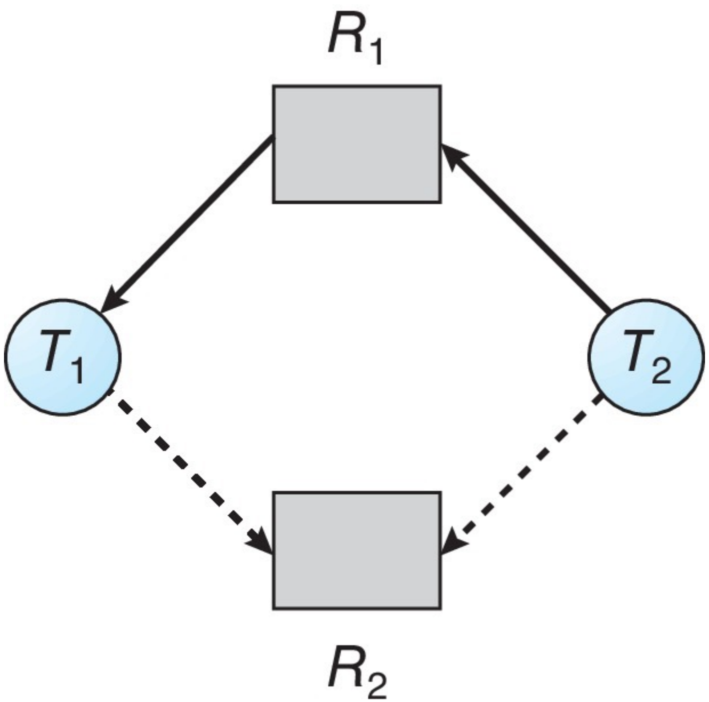
Resource-Allocation Graph Algorithm

- 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.

- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, thread T_i will have to wait for its requests to be satisfied.

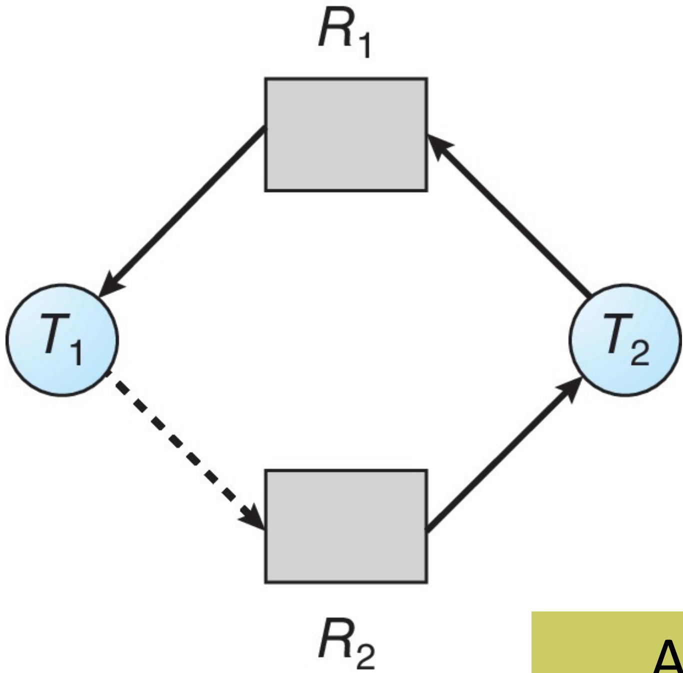
Example of Using the Algorithm



Suppose that T2 requests R2.

Can the request be granted?

Example of Using the Algorithm (Cont.)



Suppose that T_2 requests R_2.

Can the request be granted?

Although R_2 is currently free, we cannot allocate it to T2, since this action will create a cycle in the graph.

A cycle, indicates that the system is in an unsafe state. If T_1 requests R_2, and T_2 requests R_1, then a deadlock will occur.

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

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

- **Available**: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available
- **Max**: *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j

Data Structures for the Banker's Algorithm

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

- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j

- **Need**: n x m matrix. If *Need*[i,j] = k, then 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

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`

2. Find an `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

Resource-Request Algorithm for Process P_i

- Algorithm determines whether requests can be safely granted.
- $\boldsymbol{Request_i}$ = ***request*** vector for process $\boldsymbol{P_i}$
- If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants **k** instances of resource type $\boldsymbol{R_j}$

Resource-Request Algorithm for Process P_i (Cont.)

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since resources are not available.

3. Pretend to allocate requested resources to 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 safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), 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	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

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 2	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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?