Operating Systems

Deadlocks

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

May, 2022

System model

- Resources:
  - Like CPU, memory, files, devices, semaphores, etc.
  - Divided into several types (e.g., L1 and L2 cache)
  - Each type has one or more *identical* instances (e.g., CPU cores)
- Threads (of Processes):
  1. Request resource (If resource cannot be granted immediately, process waits until it can acquire resource.)
  2. Use resource
  3. Release resource

Deadlock conditions

- **Mutual exclusion:** Resources cannot be shared.
- **Hold and wait:** Threads must not release resources just because they are waiting.
- **No preemption:** A resource can only be released voluntarily by the thread holding the resource.
- **Circular wait:** There must exist a set $\{P_0, \ldots, P_{n-1}\}$ of waiting threads such that P_i is waiting for a resource held by $P_{(i+1) \% n}$, $i = 0, \ldots, n-1$.

Resource allocation graph

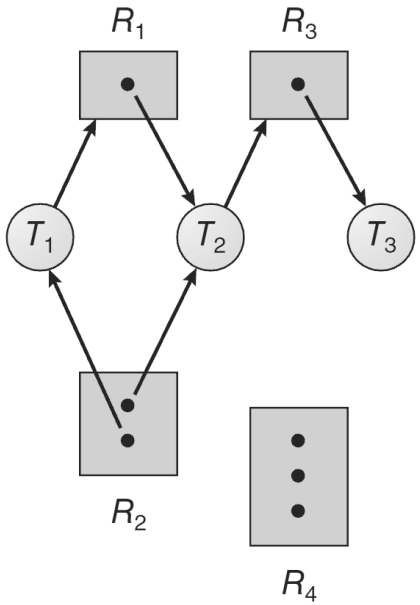
Let $P = \{P_1, \ldots, P_n\}$ and $R = \{R_1, \ldots, R_m\}$ be the set of all active threads and all resource types in the system, respectively.

Now construct the following graph:
The set of vertices defined by $V = P \cup R$. The set of edges defined by $E = E_{req} \cup E_{assign}$, where
$E_{req} =$
$\{P_i \rightarrow R_j \mid P_i$ has requested an instance of R_j and is waiting for it$\}$
$E_{assign} = \{R_j \rightarrow P_i \mid P_i$ holds an instance of $R_j\}$

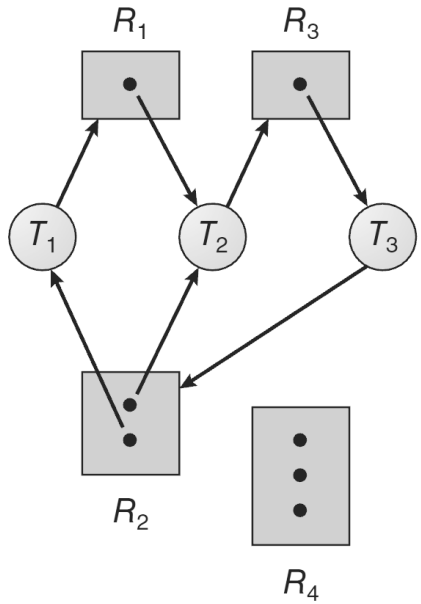
- If the graph does not contain a cycle, no deadlock exists.
- If the graph contains a cycle and each resource node in the cycle has exactly one instance, then deadlock exists.
- Otherwise, deadlock may or may not be present.

Resource-allocation graph

Suppose the thread T_3 requests an instance of resource type R_2.



Resource-allocation graph (before granting the request) with no deadlock



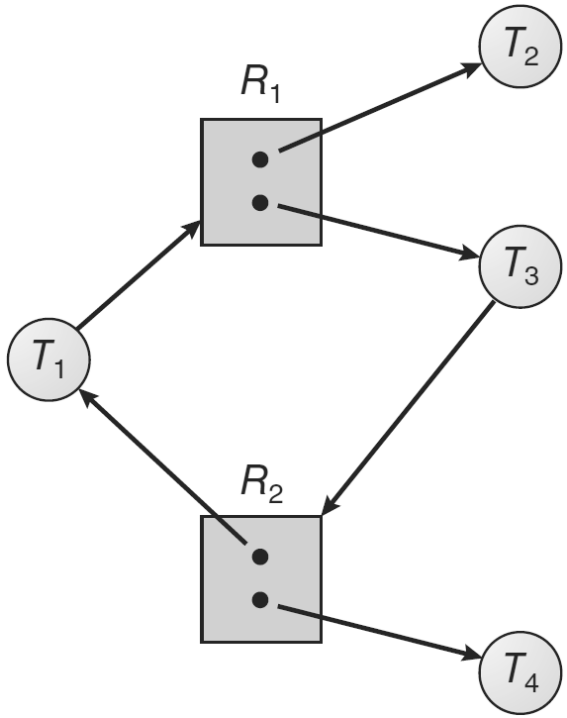
Resource-allocation graph (after granting the request) with deadlock

On granting the request, the following minimal cycles will appear.

- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

Resource-allocation graph

See below a resource-allocation graph with a cycle but no deadlock.



Handling the deadlock

- **Prevention:** System/processes/threads ensure(s) that at least one of the necessary conditions does not hold. Deadlocks are prevented by limiting how requests can be made. The advantages of preventing deadlocks are low device utilization and reduced system throughput.
- **Avoidance:** Processes follow a protocol to ensure that deadlock never happens.
- **Recovery:** Recover to a safe state before the deadlock happened.
- **Ignore deadlock:** System may have to be restarted if deadlock occurs (used in most of the OSs).

Denying mutual exclusion

Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.

However, in reality, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable (e.g., mutex locks).

Denying hold and wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources.

In reality, we cannot prevent deadlocks by denying the hold and wait condition, because then concurrent use of resources will get affected.

Denying no preemption

If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released.

The preempted resources are added to the list of resources for which the thread is waiting.

In reality, we cannot prevent deadlocks by allowing preemption in all kinds of resources (e.g., mutex locks, semaphores).

Denying circular wait

This is a more practical solution because we can ensure that this condition never holds by imposing a total ordering of all resource types and requiring that each thread requests resources in an increasing order of enumeration.

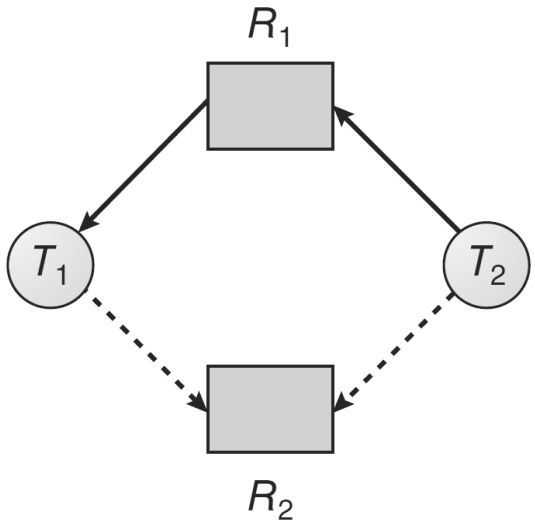
Avoiding deadlocks

We can avoid deadlocks by providing additional information about how resources are to be requested.

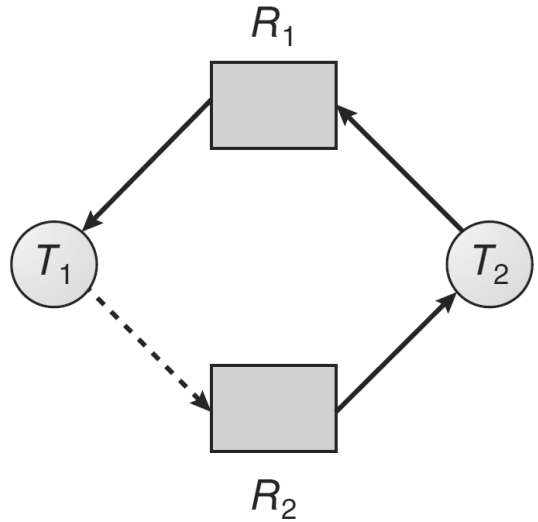
A state is *safe* if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.

Avoiding deadlocks with resource-allocation graph algorithm

In addition to the request (E_{req}) and assignment (E_{assign}) edges already described, we can accommodate a new type of edge, called a claim edge. A claim edge $T_i \rightarrow R_j$ indicates that thread T_i may request resource R_j at some time in the future.



Resource-allocation graph (before granting the request) with no deadlock



Resource-allocation graph (after granting the request) with deadlock

Avoiding deadlocks with Banker's algorithm

The following data structures are used.

- **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 thread. If $Max[i][j]$ equals k, then thread T_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 thread. If $Allocation[i][j]$ equals k, then thread T_i is currently allocated k instances of resource type R_j.
- **Need:** An $n \times m$ matrix indicates the remaining resource need of each thread. If $Need[i][j]$ equals k, then thread T_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j] = Max[i][j] - Allocation[i][j]$. are available.

Safety algorithm

The algorithm for finding out whether a system is in a safe state is stated below.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$
3. If no such i exists, go to step 4.
4. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
5. If $Finish[i] == true$ for all i, then the system is in a safe state.

Note: This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Consider an example

The following system is currently in a safe state.

	Allocation	*Max*	*Available*
	A B C	A B C	A B C
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

Resource-request algorithm

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_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 thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for $Request_i$, and the old resource-allocation state is restored.