Deadlock

Deadlock: situation in which two or more threads are waiting for an event that can only be generated by these same threads. Also known as the deadly embrace.

Example

```
Process A:                Process B:
 printer->Wait();          disk->Wait();
 disk->Wait();             printer->Wait();

 // copy from disk         // copy from disk
 // to printer             // to printer

 printer->Signal();        printer->Signal();
 disk->Signal();           disk->Signal();
```

Deadlock can occur any time threads acquire multiple resources (printers, disks, etc.), perform work, and then release their resources.

Deadlock prevention algorithms check resource requests and resource availability to forestall deadlock.

Deadlock detection finds instances of deadlock when threads stop making progress and tries to recover.

Starvation occurs when a thread waits indefinitely for some resource, but other threads are actually using the resource and making progress. Also known as effective deadlock.

Necessary Conditions for Deadlock

Deadlock can happen only if all the following conditions hold.

1. **Mutual Exclusion** - at least one thread must hold a resource in non-sharable mode, i.e., the resource may only be used by one thread at a time.
2. **Hold and Wait** - at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
3. **No Preemption** - A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
4. **Circular wait** - A set of waiting threads $\{t_1, \ldots, t_n\}$ where t_i is waiting on t_{i+1} ($i = 1$ *to* n) and t_n is waiting on t_1.

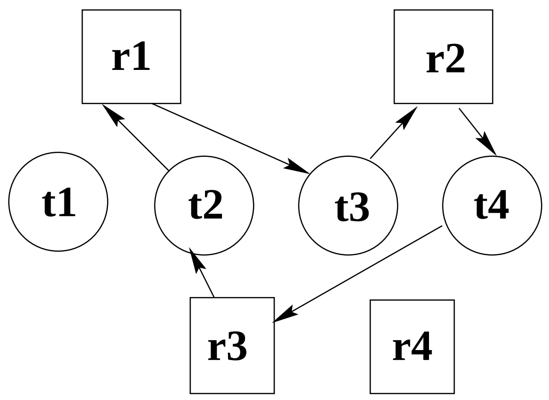
Deadlock Detection Using a Resource Allocation Graph

We define a graph with vertices that represent both resources $\{r_1, \ldots r_m\}$ and threads $\{t_1, \ldots, t_n\}$.

- A directed edge from a thread to a resource, $t_i \rightarrow r_j$ indicates that t_i has requested that resource, but has not yet acquired it.
- A directed edge from a resource to a thread $r_j \rightarrow t_i$ indicates that the OS has allocated r_j to t_i.

$\Longrightarrow$ If the graph has no cycles, no deadlock exists.

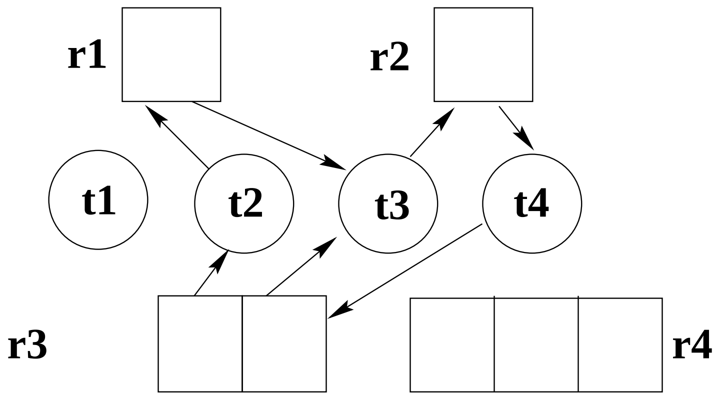
$\Longrightarrow$ If the graph has a cycle, deadlock may exist.

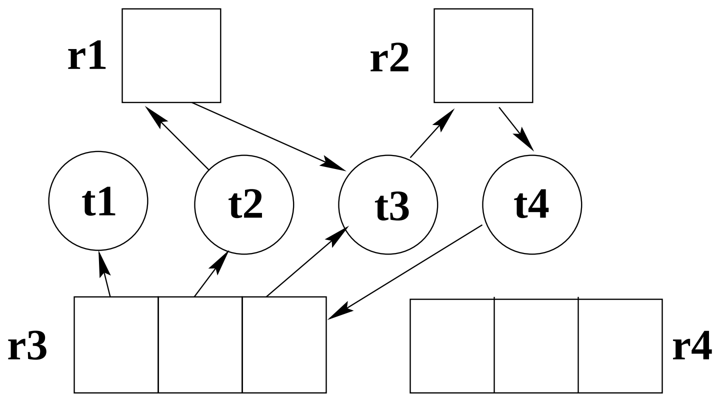


What are the cycles? Is there deadlock?

Deadlock Detection Using a Resource Allocation Graph

What if there are multiple interchangeable instances of a resource?





Then a cycle indicates only that deadlock *may* exist. If any instance of a resource involved in the cycle is held by a thread that is not in the cycle, we may be able to make progress when that thread releases the resource.

Detect Deadlock and Then Correct It

Using the resource allocation graph, scan the graph for cycles, and then break the cycles.

Alternative ways of breaking the cycles:

- Kill all threads in the cycle.
- Kill the threads one at a time, forcing them to give up resources.
- Preempt resources one at a time rolling back the state of the thread holding the resource to the state it was in prior to getting the resource. This technique is common in database transactions.

Detecting cycles takes O(n^2) time, where n is $|T| + |R|$. When should we execute this algorithm?

- Just before granting a resource, check if granting it would lead to a cycle? (Each request is then O(n^2).)
- Whenever a resource request can't be filled? (Each failed request is O(n^2).)
- On a regular schedule (hourly or ...)? (We could be deadlocked a long time before it is detected.)
- When CPU utilization drops below some threshold? (We could be deadlocked a long time before it is detected.)

Deadlock Prevention

Prevent deadlock by ensuring that at least one of the necessary conditions does not hold.

1. **Mutual Exclusion** - make resources sharable (but some resources cannot be shared)
2. **Hold and Wait** - guarantee that a thread cannot hold one resource when it requests another, or make threads request all the resources they need at once and make the thread release all resources before requesting a new set.
3. **No Preemption** - if a thread requests a resource that cannot be immediately allocated to it, then the OS preempts (releases) all the resources that the thread is currently holding. Only when all of the resources are available will the OS restart the thread. (Not all resources can be easily preempted — consider printers.)
4. **Circular wait** - impose an ordering (numbering) on the resources and request them in order.

Deadlock Prevention with Resource Reservation

- Force threads to provide advance information about the maximum resources they may need for the duration of the execution.
- Define a sequence of threads $\{t_1, \ldots, t_n\}$ as *safe* if for each t_i, the resources that t_i can still request can be satisfied by the currently available resources plus the resources held by all $t_j, j < i$.
- A *safe state* is a state in which there is a safe sequence for the threads.
- An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads may not ever use the maximum resources they have declared.
- When a thread requests a resource, the system grants it the resource if the new state is safe. If the new state is unsafe, the thread must wait even if the resources are currently available.
- This algorithm ensures that no circular-wait condition exists.

Example

Threads t_1, t_2, and t_3 are competing for 12 tape drives. Currently, 11 of those drives are allocated to the threads, leaving 1 available.

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

$\Longrightarrow$ The current state is *safe* because there exists a safe sequence, $\{t_1, t_2, t_3\}$ where all process may obtain their maximum number of resources without waiting.

- t_1 can complete with the current resource allocation
- t_2 can complete with its current resources, plus all of t_1's resources, and the unallocated tape drive.
- t_3 can complete with all its current resources, all of t_1 and t_2's resources, and the unallocated tape drive.

$\Longrightarrow$ If t_3 requests one more drive, then it must wait because allocating the drive would lead to an unsafe state. There are now 0 available drives, but each thread might need at least one more drive.

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Summary

Deadlock is a situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set. Approaches to handling deadlock are:

- Detection and recovery - recognize deadlock after it has occurred and break it.
- Avoidance - don't allocate a resource if it would introduce a cycle.
- Prevention - design resource allocation strategies that guarantee that one of the necessary conditions never holds
- Mixed strategy
- Code concurrent programs very carefully. This only helps prevent deadlock over resources managed by the program, not OS resources.
- Ignore the possibility!