**What is a Deadlock?**

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



**How to avoid Deadlocks**

Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock.

1. **Mutual Exclusion**

Resources shared such as read-only files do not lead to deadlocks but resources, such as printers and tape drives, requires exclusive access by a single process.

1. **Hold and Wait**

In this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

1. **No Preemption**

Preemption of process resource allocations can avoid the condition of deadlocks, where ever possible.

1. **Circular Wait**

Circular wait can be avoided if we number all resources, and require that processes request resources only in strictly increasing(or decreasing) order.

**Handling Deadlock**

The above points focus on preventing deadlocks. But what to do once a deadlock has occured. Following three strategies can be used to remove deadlock after its occurrence.

1. **Preemption**

We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

1. **Rollback**

In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

1. **Kill one or more processes**

This is the simplest way, but it works.

**What is a Livelock?**

There is a variant of deadlock called livelock. This is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.

A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

**Conditions Necessary for Deadlock**

All of the following four necessary conditions must hold simultaneously for deadlock to occur:

* **mutual exclusion:** only one process can use a resource at a time.
* **hold and wait:** a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes.
* **no preemption:** a resource can only be released voluntarily by the process holding it.
* **circular wait:** a cycle of process requests exists (i.e., *P*0 is waiting for a resource hold by *P*1 who is waiting for a resource held by *Pj* ... who is waiting for a resource held by *P*(*n*-1) which is waiting for a resource held by *Pn* which is waiting for a resource held by *P*0).

Circular wait implies the hold and wait condition. Therefore, these conditions are not completely independent.

**Resource Allocation Graph Syntax**

A resource allocation graph contains a set of vertices V and a set of edges E.

V is partitioned into two types:

* P = {P1, P2, ..., Pn} is the set of all processes.
* R = {R1, R2, ..., Rm} is the set of all resources.

A *request* is represented by a directed edge from *Pi* to *Rj*.
An *assignment* is represented by a directed edge from *R*j to *Pi*.

* resource type with four instances:

* *Pi* requests an instance of *Rj*

* *Pi* is holding an instance of *Rj*


**Sample Resource Allocation Graphs**

* **resource allocation graph without deadlock:**
	+ *P*1 wants a resource held by *P*2
	+ no process is requesting an instance of *R*4


(regenerated from [OSC8] Fig. 7.2 on p. 288)

* **resource allocation graph with a cycle and deadlock:**


(regenerated from [OSC8] Fig. 7.3 on p. 289)

* **resource allocation graph with a cycle but no deadlock:**

(regenerated from [OSC8] Fig. 7.4 on p. 289)

***Possibility* of Deadlock**


(regenerated from [OSC8] Fig. 7.5 on p. 295)

If a resource allocation graph contains no cycles, then no process is deadlocked.

If a resource allocation graph contains a cycle, then a deadlock may exist.

Therefore, a cycle means deadlock is *possible*, but not necessarily *present*.

A cycle is not sufficient proof of the presence of deadlock. A cycle is a *necessary* condition for deadlock, but not a *sufficient* condition for deadlock.

difference between *necessary* and *sufficient*

* getting a 4.0 GPA is *sufficient* to graduate, but it is not *necessary*
* passing CPS 356 (OS) is *necessary*, but not *sufficient*

**Resource Allocation Graph Summary**

* if a resource allocation graph does not contain a cycle, then there is absolutely no possibility of deadlock
* if a resource allocation graph contains a cycle, then there is the possibility of deadlock
* if each resource type has exactly one instance, then a cycle implies that deadlock has occurred
* if the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred
* if all instances of a resource are allocated to a process in a cycle, then there is deadlock

**Methods for Handling Deadlock**

The following are methods for addressing the possibility of deadlock:

* ensure that the system never enters a deadlocked state:
	+ deadlock prevention
	+ deadlock avoidance
* deadlock detection and recovery: allow the system to enter a deadlocked state, then deal with and eliminate the problem
* ignore the problem: approached used by many operating systems including UNIX and Windows, and the Java VM

**Deadlock Prevention**

Restrain the ways resource requests are made so to prevent one of the [four conditions](http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/deadlock.html%22%20%5Cl%20%22NecessaryConditions) necessary for deadlock.

* **prevent mutual exclusion**
	+ use only sharable resources (e.g., a read-only file)
	+ impossible for practical systems
* **prevent hold and wait**
	+ methods
		- preallocate
			* do not pick up one chopstick if you cannot pick up the other
			* for a process that copies data from DVD drive to a file on disk and then prints it from there:
				1. request DVD drive
				2. request disk file
				3. request printer
			* all system calls requesting resources must proceed all other system calls
		- a process can request resources only when it has none
			* request DVD drive and disk file
			* release DVD drive and disk file
			* request disk file and printer (no guarantee data will still be there)
			* release disk file and printer
	+ inefficient
	+ starvation possible
* **prevent no preemption** (i.e., allow preemption, and permit the OS to take away resources from a process)
	+ when a process must wait, it must release its resources
	+ some resources cannot be feasibly preempted (e.g., printers, tape drives)
* **prevent circular wait**
	+ impose a total ordering on resources
	+ only allow requests in an increasing order

Usually a deadlock prevention approach is simply unreasonable.

**Deadlock Avoidance**

This requires that the system has some information available up front. Each process declares the maximum number of resources of each type which it may need. Dynamically examine the resource allocation state to ensure that there can never be a circular-wait condition.

The system's resource-allocation state is defined by the number of available and allocated resources, and the maximum possible demands of the processes. When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe state*.

The system is in a safe state if there exists a safe sequence of all processes:

Sequence < P1, P2, ... P*n* > is safe for the current allocation state if, for each P*i*, the resources which P*i* can still request can be satisfied by

* the currently available resources plus
* the resources held by all of the P*j*'s, where *j* < *i*.

If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the *possibility* of deadlock.

**Example:** consider a system with 12 magnetic tapes and 3 processes (P0, P1, and P2):

|  |
| --- |
| available = 3 |
| **Process** | **Maximum Needs** | **Holding** | **Needs** |
| P0 | 10 | 5 | 5 |
| P1 | 4 | 2 | 2 |
| P2 | 9 | 2 | 7 |

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

|  |
| --- |
| available = 2 |
| **Process** | **Maximum Needs** | **Holding** | **Needs** |
| P0 | 10 | 5 | 5 |
| P1 | 4 | 2 | 2 |
| P2 | 9 | 3 | 6 |

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

In this scheme, a process which requests a resource that is currently available, may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

**Deadlock Avoidance Algorithms**

Two deadlock avoidance algorithms:

* resource-allocation graph algorithm
* Banker's algorithm

**Resource-allocation graph algorithm**

* only applicable when we only have 1 instance of each resource type
* claim edge (dotted edge), like a *future* request edge
* when a process requests a resource, the claim edge is converted to a request edge
* when a process releases a resource, the assignment edge is converted to a claim edge
* cycle detection: ***O*(*n*²)**

**Banker's Algorithm**

* a classic deadlock avoidance algorithm
* more general than resource-allocation graph algorithm (handles multiple instances of each resource type), but
* is less efficient

**Resource-allocations graphs for deadlock avoidance**



(regenerated from [OSC8] Fig. 7.6 on p. 297)



(regenerated from [OSC8] Fig. 7.7 on p. 297)

**Banker's Algorithm**

We call Banker's algorithm when a request for R is made. Let n be the number of processes in the system, and m be the number of resource types.

Define:

* available[m]: the number of units of R currently unallocated (e.g., available[3] = 2)
* max[n][m]: describes the maximum demands of each process (e.g., max[3][1] = 2)
* allocation[n][m]: describes the current allocation status ( e.g., allocation[5][1] = 3)
* need[n][m]: describes the *remaining* possible need (i.e., need[i][j] = max[i][j] - allocation[i][j])

**Resource-request algorithm**:

Define:

* request[n][m]: describes the current outstanding requests of all processes (e.g., request[2][1] = 3)
1. If request[i][j] <= need[i][j], to to step 2; otherwise, raise an error condition.
2. If request[i][j] > available[j], then the process must wait.
3. Otherwise, *pretend* to allocate the requested resources to P*i* :
4. available[j] = available[j] - request[i][j]
5. allocation[i][j] = allocation[i][j] + request[i][j]

 need[i][j] = need[i][j] - request[i][j]

Once the resources are *allocated*, check to see if the system state is safe. If unsafe, the process must wait and the old resource-allocated state is restored.

**Safety algorithm** (to check for a safe state):

1. Let work be an integer array of length m, initialized to available.
Let finish be a boolean array of length n, initialized to false.
2. Find an i such that both:
	* finish[i] == false
	* need[i] <= work

If no such i exists, go to step 4

1. work = work + allocation[i];
finish[i] = true;
Go to step 2
2. If finish[i] == true for all i, then the system is in a safe state, otherwise unsafe.

Run-time complexity: ***O*(*m* × *n*²)**.

*Example*: consider a system with 5 processes (P0 ... P4) and 3 resources types (A(10) B(5) C(7))

resource-allocation state at time t0:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Allocation** | **Max** | **Need** | **Available** |
|  | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 3 | 3 | 2 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 | 1 | 2 | 2 |  |  |  |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 |  |  |  |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 |  |  |  |

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

< P1, P3, P4, P2, P0 >

Now suppose, P1 requests an additional instance of A and 2 more instances of type C.

request[1] = (1,0,2)

1. check if request[1] <= need[i] (yes)
2. check if request[1] <= available[i] (yes)
3. do pretend updates to the state

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Allocation** | **Max** | **Need** | **Available** |
|  | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 | 3 | 3 | 2 |
| P1 | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 |  |  |  |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 | 0 | 1 | 1 |  |  |  |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 | 4 | 3 | 1 |  |  |  |

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

<P1, P3, P4, P0, P2>

Hence, we immediately grant the request.

Will a request of (3,3,0) by P4 be granted?

Will a request of (0,2,0) by P0 be granted?

**Deadlock Detection**

* requires an algorithm which examines the state of the system to determine whether a deadlock has occurred
* requires overhead
	+ run-time cost of maintaining necessary information and executing the detection algorithm
	+ potential losses inherent in recovering from deadlock

Single instance of each resource type

* wait-graph
* P*i* → P*j* = P*i* > R*q* and R*q* → P*j*
* detect cycle: ***O*(*n*²)**
* overhead: maintain the graph + invoke algorithm

**Resource-allocations graphs for deadlock detection**

resource-allocation graph:



corresponding wait-for graph:



(regenerated from [OSC8] Fig. 7.8 on p. 302)

Multiple instances of a resource type: use an algorithm similar to Banker's, which simply investigates *every* possible allocation sequence for the processes which remain to be completed.

Define:

* available[m]
* allocation[n][m]
* request[n][m]
with their usual semantics.

Algorithm:

1. Let work be an integer array of length m, initialized to available.

Let finish be a boolean array of length n.

For all i, if allocation[i] != 0, then finish[i] = false;
Otherwise finish[i] = true.
2. Find an i such that both
	* finish[i] == false // P*i* is currently *not* involved in a deadlock
	* request[i] <= work

If no such i exists, go to step 4

1. // reclaim the resources of process P*i*
work = work + allocation[i];
finish[i] = true;
Go to step 2
2. If finish[i] == false for some i,
Then the system is in a deadlocked state.
Moreover, if finish[i] == false, then process P*i* is deadlocked.

Run-time complexity: ***O*(*m* × *n*²)**.

Example: consider a system with 5 processes (P0 .. P4) and 3 resources types (A(7) B(2) C(6))

resource-allocation state at time t0:

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Allocation** | **Request** | **Available** |
|  | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P1 | 2 | 0 | 0 | 2 | 0 | 2 |  |  |  |
| P2 | 3 | 0 | 3 | 0 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 1 | 0 | 0 |  |  |  |
| P4 | 0 | 0 | 2 | 0 | 0 | 2 |  |  |  |

Is the system in a deadlocked state?

If not, which sequence results in finish[i] == true for all i ?

< P0, P2, P3, P1, P4 >

Now suppose, P2 requests an additional instance of C:

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Allocation** | **Request** | **Available** |
|  | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P1 | 2 | 0 | 0 | 2 | 0 | 2 |  |  |  |
| P2 | 3 | 0 | 3 | 0 | 0 | 1 |  |  |  |
| P3 | 2 | 1 | 1 | 1 | 0 | 0 |  |  |  |
| P4 | 0 | 0 | 2 | 0 | 0 | 2 |  |  |  |

Is the system in a deadlocked state?    Yes.

If not, which sequence results in finish[i] == true for all i ?

Although we can reclaim the resources held by P0, the number of available resources is insufficient to fulfill the requests of the other processes.

Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

**When should we invoke the detection algorithm?** Depends on:

* how *often* is a deadlock likely to occur
* how *many* processes will be affected by deadlock when it happens

If deadlocks occur frequently, then the algorithm should be invoked frequently.

Deadlocks only occur when some process makes a request which cannot be granted (if this request is the completes a chain of waiting processes).

* **Extreme:** invoke the algorithm every time a request is denied
* **Alternative**: invoke the algorithm at less frequent time intervals:
	+ once per hour
	+ whenever CPU utilization < 40%
	+ disadvantage: cannot determine exactly which process 'caused' the deadlock

**Deadlock Recovery**

How to deal with deadlock:

* inform operator and let them decide how to deal with it manually
* let the system *recover* from the deadlock automatically:
	+ abort or more of the deadlocked processes to break the circular wait
	+ preempt some resources from one or more of the processes to break the circular wait

**Process termination**

Aborting a process is not easy; involves clean-up (e.g., file, printer).

* abort all deadlocked processes (disadvantage: wasteful)
* abort one process at a time until the circular wait is eliminated
	+ disadvantage: lot of overhead; must re-run algorithm after each kill
	+ how to determine which process to terminate? minimize cost
		- priority of the process
		- how long has it executed? how much more time does it need?
		- how many and what type of resources has the process used?
		- how many more resources will the process need to complete?
		- how many processes will need to be terminated?
		- is the process interactive or batch?

**Resource Preemption**

Incrementally preempt and re-allocate resources until the circular wait is broken.

* selecting a victim (see above)
* rollback: what should be done with process which lost the resource?
clearly it cannot continue; must rollback to a safe state (???) => total rollback
* starvation: pick victim only small (finite) number of times; use number of rollbacks in decision

**Summary**

* definition of deadlock
* three methods of addressing deadlock
	+ ensure deadlock never arises
		- deadlock prevention: ensure at least one of the four necessary conditions for deadlock never holds
			* mutual exclusion
			* hold and wait
			* no preemption
			* circular wait
		- deadlock avoidance
			* less stringent, but requires *a priori* information
			* Banker's algorithm
	+ deadlock detection and recovery
		- terminate processes: selecting a victim
		- preempt resources
			* selecting a victim
			* rollback
			* starvation
	+ ignore the problem (most common approach)
* there is no `silver bullet'
* often a combination of approaches should be employed to permit us to use an optimal approach for each class of resources in the system