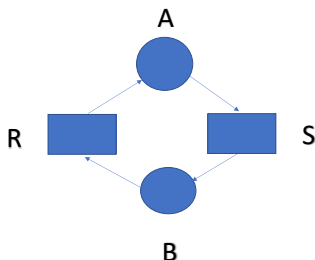


# DEADLOCKS

A computer system has multiple resources and those can be used by one process at a time and for many applications we may need more than one resource at a time. Trouble begins when we have a process (say, A) is using a resource (R) and asking for another resource (S) and simultaneously another process (B) using resource (S) and asking for R. The requirements of these two processes would never be fulfilled, if one of the processes do not release a resource or forced to take away the resource; by the OS), and this situation is known as a *deadlock*.

- The resources can be hardware (devices) or software (database records)
- Deadlocks may be mostly relevant in the context of OS but they may occur in other contexts applicable to a variety of concurrent systems



# RESOURCES

A deadlock is a fight between concurrent processes to acquire and use common *resources*. A computer has many different resources (multiple instances of the same resource as well, say two printers) that a process can acquire. In short, a resource is anything that must be

- acquired,
- used, and
- released over the course of time.

A resource can be *preemptable* – i.e., if we take it away from a process it has no ill-effects (say a page is swapped). A *non-preemptable* resource cannot be taken away without a potential failure – and we deal with deadlock for non-preemptable resource for obvious reasons.

If the resource is not available the process is automatically blocked, and awakened when it becomes available. In other systems, denial leads to wait in a tight loop requesting the resource, then sleeping, then trying again – note that it is as good as blocked as no work can proceed without the requested resource.

Here we assume that when a process is denied a resource request, it is put to sleep.

# RESOURCE AQUISITION

The following code snippets shows the chance of a deadlock for a minor variation in the code asking for the resource. Left hand side showing a deadlock free code and the right hand side code might lead to deadlock.

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

In (a) one process would acquire the 1st resource and get the 2nd as well and carry out the job. However in (b) it may be OK sometimes but it might happen that process A acquires resource 1 and process B acquires resource 2. Each one will now block when trying to acquire the other one.

# DEFINITION and DEADLOCK CONDITIONS

*Deadlock* can be formally defined as **A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.**

The following four conditions must hold for a (resource) deadlock

- I. Mutual exclusion: Each resource is either currently assigned to exactly one process or is available.
- II. Hold-and-wait: Processes currently holding resources that were granted earlier can request new resources.
- III. No-preemption: Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- IV. Circular wait: There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

# DEADLOCK MODELING

*Deadlock* can be modelled using a resource allocation graph (RAG) where a process is represented by a circle and resources by a rectangle. The directed arc

- from a resource to a process indicates that the resource has previously been requested by, granted to, and is currently held by that process.
- from a process to a resource indicates that the process is currently blocked waiting for that resource.

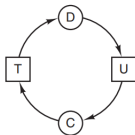
In (c) we see a deadlock: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle. In this example, the cycle is C T D U C.



(a)



(b)



(c)

# DEADLOCK MODELING

contd.

There are three processes A, B and C and three resources R, S and T requesting and releasing them. Sequential execution of the processes does not lead to any deadlock – if the processes are completely CPU bound then that is the best option. However, in reality it is always a mix and a reasonable scheduling, say RR, while have a higher throughput and better resource utilization, might lead to a deadlock.

(Note: Only 1 resource of a type is assumed)

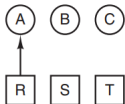
A  
Request R  
Request S  
Release R  
Release S  
(a)

B  
Request S  
Request T  
Release S  
Release T  
(b)

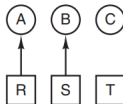
C  
Request T  
Request R  
Release T  
Release R  
(c)

1. A requests R
  2. B requests S
  3. C requests T
  4. A requests S
  5. B requests T
  6. C requests R
- deadlock**

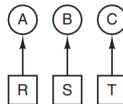
(d)



(e)



(f)



(g)

The OS may run the processes in any suitable order. If it detects an impending deadlock it can simply suspend the process without granting the request until it is safe. Here it could suspend B instead of granting it S. By running only A and C, we would get the requests and releases of (k) instead of (d). This sequence leads to the resource graphs of (i)–(q), which do not lead to deadlock. After step (q), process B can be granted S because A is finished and C has everything it needs. Even if B blocks when requesting T, no deadlock can occur. B will just wait until C is finished.

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S  
no deadlock

(k)



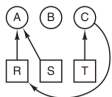
(l)



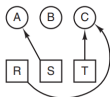
(m)



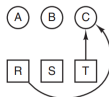
(n)



(o)



(p)



(q)

# DEADLOCK HANDLING

In general, four strategies are used to deal with deadlocks.

1. Just ignore the problem.
2. Detection and recovery. Let them occur, detect them, and take action.
3. Dynamic avoidance by careful allocation of resources; and
4. Prevention, by structurally negating one of the four conditions.

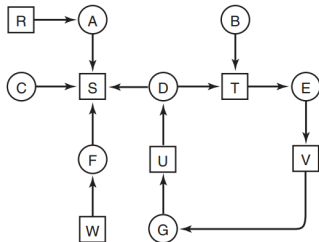
1. IGNORE: We may follow the behaviour of an ostrich (ostrich algorithm) to stick our head in the sand and pretend everything is cool. You may react that what would happen – in reality deadlock is not that frequent than a system crash and the strategy is good for many systems.



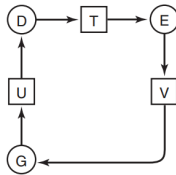
# DEADLOCK HANDLING: DETECTION

2. DETECTION AND RECOVERY: Here we don't try to prevent (as it is costly) – we detect and try for a recovery, A case for 7 processes and 6 resources is shown – a cycle can be detected that shows deadlock.

Formal algorithm may be used where each node (from the lowest level) is taken as a root and with a depth first search it checks if it is a tree. If this property holds for all nodes (by moving the next upper level and repeat checking and finally backtracking to the root), the entire graph is cycle free, so the system is not deadlocked.



(a)



(b)

# DETECTION with multiple resources of each type

A matrix based algorithm to detect deadlock is used.

- 'n' processes  $P_1$  to  $P_n$ ;  $E_i$  resources of class  $i$  ( $1 \leq i \leq m$ ).
- $E$  is the **existing resource vector**; if class 1 is tape drive then  $E_1 = 2$  indicates 2 tape drives.
- $A$  be the **available resource vector**;  $A_1 = 0$  means both the tape drives are available for use.
- $C[i, j]$  and  $R[i, j]$  are the current allocation and request matrices and  $C[i, j]$  is the number of instances of resource  $j$  that are held by process  $P_i$ .
- $R[i, j]$  is the number of instances of resource  $j$  that  $P_i$  wants.
- Every resource is either free or allocated thus  $\sum_{i=1}^n C_{ij} + A_j = E_j$ .
- Let  $A \leq B$  mean that each element of vector A is less than or equal to the corresponding element of vector B. Mathematically,  $A \leq B$  holds iff  $A_i \leq B_i$  for  $1 \leq i \leq m$ .

# DEADLOCK DETECTION :

contd...

The algorithm sets each process as unmarked and then proceeds. At the end unmarked processes indicating deadlock. Note, that the algorithm assumes a worst case scenario that all the acquired resources by a process is released when they exit.


1. Look for an unmarked process,  $P_i$ , for which the  $i_{th}$  row of  $R$  is less than or equal to  $A$ .
2. If such a process is found, add the  $i_{th}$  row of  $C$  to  $A$ , mark the process, and go back to step 1.
3. If no such process exists; terminate.

Resources in existence  
( $E_1, E_2, E_3, \dots, E_m$ )

Resources available  
( $A_1, A_2, A_3, \dots, A_m$ )


Current allocation matrix

Request matrix



$C_{11}$	$C_{12}$	$C_{13}$	$\dots$	$C_{1m}$
$C_{21}$	$C_{22}$	$C_{23}$	$\dots$	$C_{2m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$C_{n1}$	$C_{n2}$	$C_{n3}$	$\dots$	$C_{nm}$

Row n is current allocation  
to process n



$R_{11}$	$R_{12}$	$R_{13}$	$\dots$	$R_{1m}$
$R_{21}$	$R_{22}$	$R_{23}$	$\dots$	$R_{2m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$R_{n1}$	$R_{n2}$	$R_{n3}$	$\dots$	$R_{nm}$

Row 2 is what process 2 needs

# DEADLOCK DETECTION :

contd...

Here we have three processes and four arbitrarily labelled resource classes. From C we see that process 1 ( $P_1$ ) has 1 scanner; process 2 ( $P_2$ ) has 2 tape-drive and 1 blu-ray drive... etc. Each process needs additional resources, as shown by the R matrix.

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied.

- Requests from  $P_1$  and  $P_2$  cannot be satisfied as we have no blu-ray free as well as scanner. However,
- Request from  $P_3$  can be complied – it runs and returns giving  $A = (2\ 2\ 2\ 0)$ .
- Now  $P_2$  can run giving  $A = (4\ 2\ 2\ 1)$  on return – and rest of the process now runs without deadlock

$$E = \begin{matrix} & \begin{matrix} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{Blu-rays} \end{matrix} \\ \begin{matrix} (4 & 2 & 3 & 1) \end{matrix} \end{matrix}$$

$$A = \begin{matrix} & \begin{matrix} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{Blu-rays} \end{matrix} \\ \begin{matrix} (2 & 1 & 0 & 0) \end{matrix} \end{matrix}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

A change in the requirements leads to deadlock; e.g., say process 3 ( $P_3$ ) needs a Blu-ray drive plus two tape drives and the plotter.

- None of the requests can be satisfied, so the entire system will eventually be deadlocked.
- Even if we give process 3 ( $P_3$ ) its two tape drives and one plotter, the system deadlocks when it requests the Blu-ray drive.

Now, with static resource requests known in advance, when to look for them comes up. There may be three strategies.

1. Check every time a resource request is made. This is certain to detect them as early as possible, but it is potentially expensive in terms of CPU time. Or,
2. Check every  $k$  minutes, or
3. Only when the CPU utilization has dropped below some threshold. If enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

# DEADLOCK HANDLING: RECOVERY

After detection the OS should do recovery; if possible. There are three approaches; [note: None are exciting] that can be taken by

1. preemption;
2. rollback; and
3. killing processes

1. In certain cases it may be possible to take back a resource to recover from a deadlock. However, the ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult (may need manual intervention) or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

# DEADLOCK HANDLING: RECOVERY

2. Recovery through rollback: Here we may have processes checkpointed periodically; i.e., the states written in a file so that it can be restarted later. It can be checked at what point the deadlock happened and the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.
3. Killing processes: The crudest but simplest way is to kill a process preferably one of those which creates a cycle. Even a process not in a cycle but holding a lot of resources may be killed to recover from the deadlock. In fact, whenever possible kill a process which can be run from the beginning without having any ill effect. So, simply when the second run has no effect on the first. Say, a compiling process (for a database – this cannot be done always)

# DEADLOCK HANDLING: AVOIDANCE

If we could avoid scenarios leading to deadlock our job is done then. So, avoidance could be an effective strategy

- The system could detect if a grant is safe; possibly not leading to deadlock, and allocate accordingly.
- If certain advance information is available we could formulate algorithm that could take a right choice in allocation of resources and avoid deadlocks.

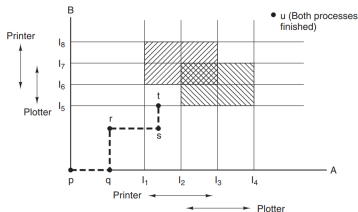
**Resource Trajectories:** In most systems the resources are requested one at a time. Tracing the resource trajectory to grasp the concept of *safety* may be the starting point.

**Trajectory for two processes and two resources:** The horizontal and vertical axes represent the number of instructions executed by processes A and B, respectively. At  $I_1$  A requests a printer; at  $I_2$  it needs a plotter. The printer and plotter are released at  $I_3$  and  $I_4$ , respectively. Process B needs the printer from  $I_6$  to  $I_8$  and the plotter from  $I_5$  to  $I_7$

**Trajectory:** At p both A and B are not executing. A is scheduled (p-q) and running and thereafter B is scheduled and running (q-r). Then A is running (s to t). At u neither A nor B are running.



- A has got Printer ( $I_1$  to  $I_3$ ) and the Plotter ( $I_2$  to  $I_4$ )
- B has got Printer ( $I_6$  to  $I_8$ ) and the Plotter ( $I_5$  to  $I_7$ )
- The 9 cells ( $I_1$  to  $I_4$  for A and  $I_5$  to  $I_8$  for B) denote different scenario. (i) Blank Cell– Only one resource is in use; (ii) 45 deg. slant lines in cells – both using the printer and (iii) 135 deg. slant lines in cells – both using the plotter  
Assume that simultaneous use ((ii) and (iii)) is protected through mutual exclusion. However, the cross hatched cell indicates– DEADLOCK as A is requesting the plotter (already enjoying printer access) and B is requesting printer (already enjoying plotter access). The cell is unsafe and NOT TO BE ENTERED. At point t the only safe thing to do is run process A until it gets to  $I_4$ . Beyond that, any trajectory to u will do.



# DEADLOCK AVOIDANCE: SAFE AND UNSAFE STATES

At any instant the current state is given by E, A, C and R.

- A state is considered safe if a scheduling order exists for which all the processes can run to completion even in the case of simultaneous request for maximum allowable resources.
- Consider a single resource class with 10 instances – (a) shows the present holding (7 in use and 3 are free) and maximum requirement for the processes
- It is a safe state as we can schedule B exclusively and we get scenario (b) and when B completes we reach (c).
- Now we can allow to run process C and reach (d) – finally when C completes we are at (e) – now A can complete.

Has Max		
A	3	9
B	2	4
C	2	7

Free: 3  
(a)

Has Max		
A	3	9
B	4	4
C	2	7

Free: 1  
(b)

Has Max		
A	3	9
B	0	–
C	2	7

Free: 5  
(c)

Has Max		
A	3	9
B	0	–
C	7	7

Free: 0  
(d)

Has Max		
A	3	9
B	0	–
C	0	–

Free: 7  
(e)

# DEADLOCK AVOIDANCE: SAFE AND UNSAFE STATES

We may move from a safe to unsafe state—

- starting state (a) and now assume A has acquired one more instances of the resource (see (b)).
- If B runs now and got the required instances of the resource we arrive at (c) – no resource is now free
- After completion of B we reach (d) – and we are stuck here. As 4 instances are free but A requires 5 more and C requires 5.
- So (b) is unsafe (though (a) was safe. Running A or C after (b) would not help either) – indicating that A's request to get one more instance should not be satisfied. [Note that unsafe does not mean leading to DEADLOCK]

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

# DA: BANKER'S ALGORITHM for A SINGLE RESOURCE

Here, it is checked that granting the request moves you from safe to unsafe state. And the resource request in that case is denied else granted if the safe condition is maintained.

- A, B, C and D are 4 borrowers (processes) to whom the banker (OS) lends some unit (1K each instances of a resource; 22 K is the maximum requirement) of money.
- The Banker knows that the maximum of 22 units may not be required immediately and reserves 10 units only. After granting some units we may have situation (b) with 2 units free.
- It is safe as the banker can delay any request except from C, letting C finishes and releases all 4 units
- with these 4 units the banker can allow either B or D getting the required units.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

# BANKERS ALGORITHM for A SINGLE RESOURCE

## ....contd.

Now consider the situation (c) where B has got 1 more instance of the resource (in comparison to what is shown in situation (b)).

- This scenario is unsafe – if all the customers asks for their maximum loans – the banker could not satisfy [with 1 free unit ] any one of them and we would have a deadlock
- Unsafe state, however, may not lead to deadlock since the customer may not ask for their maximum at a time [a savings bank works this way – if you try to withdraw money all at a time they will never be able to cope up]
- The bankers algorithm checks the resource request if the allocation leads to a unsafe state the request is postponed and the customer (process) is made to wait until other customers releases (repaying their loans) the resources.

The banker's algorithm may be extended for multiple resources each with several instances. However, in practise the algorithm can never be implemented as the processes and the OS do not know the resource requirements in advance

# Deadlock Prevention

Now consider the motto "Prevention is better than a cure" as we may not ignore deadlock– costly to detect and recover and avoidance requires future knowledge. So, if we take action not to comply one of the 4 Coffman conditions for deadlock – we can prevent the same

- i) Attacking the Mutual-Exclusion condition :
  - If no resource is exclusive allocated to a process then we would have no deadlock. For example a data page can be made RO – concurrent processes could use it without a problem. However,
  - What about concurrent access to a printer – output would all jumbled up and useless. However, concurrent processes may produce their output on a spooler and the printer daemon (only) can request the printer once the spooling is complete, But, there may be deadlock in completing the spooling as well

For many resources exclusive access is the only choice and the strategy could allow exclusive access if it is absolutely essential.

- ii) Attacking the Hold-and-wait condition : This is better than the 1st condition as mutual exclusion is the key in many cases. Hold and wait can be avoided if the requirement is that requests for all resources be made before execution. If everything is available then only the process would be started. The problems are
  - Process may not know in advance what are the exact requirements until started [otherwise Banker's Algorithm would be a great success]
  - very poor utilisation of the resources

It may be possible in a batch system with jobs running at regular basis and the requirements are known before hand. Hold-and-wait condition may be enforced in a slightly different way where the process temporarily releasing all the resources before asking for more.

- iii) Attacking no preemption : This is a possibility in certain cases as already mentioned in connection with printer daemon
  - However, virtualisation of concurrency in sharing a printer using spooling (i.e., disk space) may not always prevent deadlock
  - locking the records for database tables or tables used by OS may be a potential threat leading to deadlock.
- iv) Attacking Circular wait condition: The circular wait can be eliminated in several ways.
  - One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.
  - Provide a global numbering of all the resources. Now the rule is this: processes can request whenever they want to, but all requests must be made in numerical order in forward direction (less to more).



- Even with multiple processes with this rule there cannot be any cycle in the RAG and no deadlock.
- There can be a slackened version of the rule like a process may only ask for any high number resources from the current resource – should it require a lower one then it should release the currently using high numbered resources.

It may be noted that for critical resources like process-table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

Condition	Strategy
Mutual exclusion	Spool everything
Hold and wait	Request all resources before you start
No preempton	Take away the resource
Circular wait	Order resources nummerically

# LIVELOCK

This is similar to a scenario when two very polite persons offering each other the right of the narrow way to cross – eager to prove his/her politeness nobody crosses and no progress is made [though the resource – the narrow alley] is available.

- Suppose a process after acquiring a lock releases it when realising that it could not get the next required lock. This is good to avoid deadlock. However,
- if the other process does the same at the same time [rare but possible] then we have livelock

Consider an atomic primitive `try_lock` in which the calling process tests a mutex and either grabs it or returns failure. In other words, it never blocks. Programmers can use it together with `acquire_lock` which also tries to grab the lock, but blocks if the lock is not available.

For a pair of processes (A and B) running concurrently may be in livelock as we see from the example code.

```
void process_A(void) {
    acquire_lock(&resource _1);
    while (try_lock(&resource_2)
           == FAIL) {
        release_lock(&resource_1);
        wait_fixed_time();
        acquire lock(&resource_1);
    }
    use_both_resources( );
    release_lock(&resource_2);
    release_lock(&resource_1);
}
```

```
void process_B(void) {
    acquire_lock(&resource _2);
    while (try_lock(&resource_1)
           == FAIL) {
        release_lock(&resource_2);
        release_lock(&resource_1);
        acquire lock(&resource_2);
    }
    use_both_resources( );
    release_lock(&resource_1);
    release_lock(&resource_12);
}
```

Suppose a UNIX system has 100 process slots. Ten programs are running each requires 12 more slots. After each process creating 9 new processes – now we have our process table full– next fork() – would fail and wait and try and fail – and so on – leading to a livelock

# STARVATION

A problem similar to Deadlock and livelock but not exactly same is starvation. Though the basic principle followed by the OS designer that it would be unbiased and fair to every process it may compromise this principle considering different other factors that generally improves the throughput of the system. Here comes the possibility of starvation. As an example suppose the printer daemon knows the size of the files and started picking up small files only to print. A large file is waiting to be printed in a busy system flooded with the print request of many small files all the time. In such a scenario the large file may never be printed. An obvious solution here is the FCFS algorithm – however FCFS may not be a good choice in some other scenario – thus starvation, though rare in a well designed OS, may happen in some form or other.