DEADLOCKS

EXAMPLES:

- "It takes money to make money".
- You can’t get a job without experience; you can’t get experience without a job.
- The cause of deadlocks: Each process needing what another process has. This results from sharing resources such as memory, devices, links.
- Under normal operation, a resource is attained by:

  a) Request a resource (suspend until available if necessary).
  b) Use the resource.
  c) Release the resource.

DEADLOCK CHARACTERISATION:

NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur:

- **Mutual exclusion**

  One or more than one resource must be held by a process in a non-sharable (exclusive) mode.

- **Hold and Wait**

  A process holds a resource while waiting for another resource.

- **No Preemption**

  There is only voluntary release of a resource - nobody else can make a process give up a resource.

- **Circular Wait**

  Process A waits for Process B waits for Process C .... waits for Process A.

  Can detect this in resource allocation graph.

  A cycle present may mean deadlock is possible, unless several instances of resource are treated as one node.

RESOURCE ALLOCATION GRAPH

- A visual (mathematical) way to determine if a deadlock has, or may occur.

G = (V, E)	The graph contains nodes and edges.
V	Nodes consist of processes = { P1, P2, P3, ...} and resource types { R1, R2, ...}
E	Edges are (Pi, Rj) or (Ri, Pj)

- An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.
- Process is a circle, resource type is square; dots represent number of instances of resource in type. Request points to square, assignment comes from dot.
- If the graph contains no cycles, then no process is deadlocked.
- If there is a cycle, then:

  a) If resource types have multiple instances, then deadlock MAY exist.

  b) If each resource type has 1 instance, then deadlock has occurred.
  <<<SEE FIGURES 7.1, 7.2, 7.3 >>>

HOW TO HANDLE DEADLOCKS – GENERAL STRATEGIES

There are two methods:

- Ensure deadlock never occurs using either

Prevention	Prevent any one of the 4 conditions from happening.
Avoidance	Calculate cycles about to happen.

- Allow deadlock to happen. This requires:

Detection	Know a deadlock has occurred.
Recovery	Regain the resources.

PREVENTION:

Do not allow one of the four conditions to occur.

- **Mutual exclusion:**

  a) Automatically holds for printers and other non-sharables.

  b) Shared entities (read only files) don't need mutual exclusion (and aren’t susceptible to deadlock.)

  c) Prevention not possible, since some devices are intrinsically non-sharable.

- **Hold and wait:**

  a) Collect all resources before execution.

  b) A particular resource can only be requested when no others are being held. A sequence of resources is always collected beginning with the same one.

  c) Utilization is low, starvation possible.

- **No preemption:**

  a) Release any resource already being held if the process can't get an additional resource.

  b) Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.

- **Circular wait:**

  a) Number resources and only request in ascending order.

- EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.
- Prevention is generally the easiest to implement.

AVOIDANCE:

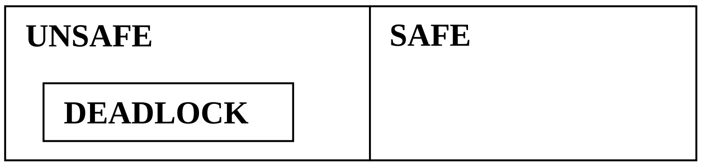
- If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.

- Possible states are:

  - **Deadlock** No forward progress can be made.
  - **Unsafe state** A state that **may** allow deadlock.
  - **Safe state** A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

- The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, not all unsafes are deadlocks.



Only with luck will processes avoid deadlock.

O.S. can avoid deadlock.

- Let's assume a very simple model: each process declares its maximum needs. In this case, algorithms exist that will ensure that no unsafe state is reached.

EXAMPLE:

There exists a total of 12 tape drives. **<<< Example on Page 218. >>>**

The current state looks like this:

Process	Max Needs	Allocated	Current Needs
P0	**10**	**5**	**5**
P1	**4**	**2**	**2**
P2	**9**	**2**	**7**

a) In this example, < p1, p0, p2 > is a workable sequence.
b) Suppose p2 requests and is given one more tape drive. What happens then?

RESOURCE ALLOCATION GRAPH ALGORITHM

- **Claim edge Pi** ---> indicates that process **Pi** may request resource **Rj**; represented by a dashed line.
- Claim edge converts to a request edge when a process requests a resource.
- When a process releases a resource, the assignment edge converts to a claim edge.
- Resources must be claimed a priori in the system.

SAFETY ALGORITHM

- A method used to determine if a particular state is safe. It’s safe if there exists a sequence of processes such that for process I:

$$\text{need} <= \text{available} + \text{allocated}[0] + .. + \text{allocated}[I-1]$$

Let **work** and **finish** be vectors of length **m** and **n** respectively.

```
1.    Initialize  work        = available
      Initialize  finish[i]   = false,    for i = 1,2,3,..n

2.    Find an i such that:
      finish[i] == false   and   need[i] <= work

      If no such i exists, go to step 4.

3.    work          = work   +   allocation[i]
      finish[i]     = true
      goto step 2

4.    if finish[i]   ==  true for all i,    then the system is  in a safe state.
```

- Do the example on **<<< PAGES 222 - 223 >>>**

Consider a system with five processes, P0 → P4 and three resource types, A, B, C. Type A has 10 instances, B has 5 instances, and C has 7 instances. At time T0 the following snapshot of the system is taken.

		←	**Alloc**	**→**		**←**	**Need**	**→**		**←**	**Avail**	**→**
		A	**B**	**C**		**A**	**B**	**C**		**A**	**B**	**C**
P0		**0**	**1**	**0**		**7**	**5**	**3**		**3**	**3**	**2**
P1		**2**	**0**	**0**		**3**	**2**	**2**				
P2		**3**	**0**	**2**		**9**	**0**	**2**				
P3		**2**	**1**	**1**		**2**	**2**	**2**				
P4		**0**	**0**	**2**		**4**	**3**	**3**				

Is the system in a safe state?

- Now try it again with only a slight change in the request by P1.
- P1 requests one additional resource of type A, and two more of type C.
- Request1 = (1,0,2).
- Is Request1 < available?
- Produce the state chart as if the request is granted and see if it's safe.

		←	**Alloc**	**→**		**←**	**Need**	**→**		**←**	**Avail**	**→**
		A	**B**	**C**		**A**	**B**	**C**		**A**	**B**	**C**
P0		**0**	**1**	**0**		**7**	**4**	**3**		**3**	**3**	**2**
P1		**3 #**	**0**	**2 #**		**0**	**2**	**0**				
P2		**3**	**0**	**2**		**6**	**0**	**0**				
P3		**2**	**1**	**1**		**0**	**1**	**1**				
P4		**0**	**0**	**2**		**4**	**3**	**1**				

DEADLOCK DETECTION:

- Need an algorithm that determines if deadlock occurred.
- Also need a means of recovering from that deadlock.

SINGLE INSTANCE OF A RESOURCE TYPE

- Wait-for graph == remove the resources from the usual graph and collapse edges.
- An edge from p(j) to p(i) implies that p(i) is waiting for p(j) to release.

<<< FIGURE 7.7 >>>

SEVERAL INSTANCES OF A RESOURCE TYPE

- Complexity is of order m * n * n.
- We need to keep track of:

available	- records how many resources of each type are available.
allocation	- number of resources of type m allocated to process n.
request	- number of resources of type m requested by process n.

Let **work** and **finish** be vectors of length **m** and **n** respectively.

```
1.    Initialize     work = available
      For   i = 1,2,...n,   if   allocation[i] != 0   then
            finish[i]  =  false;   otherwise,   finish[i]  =  true;

2.    Find an i such that:
       finish[i]  ==  false and request[i]  <=  work

      If no such i exists, go to step 4.

3.    work   =   work  +  allocation[i]
      finish[i]  =  true
      goto step 2

4.    if  finish[i]  ==  false for some i, then the system is in deadlock state.
      IF finish[i]  == false, then    process p[i] is deadlocked.
```

EXAMPLE

- This is an example from **<<< PAGE 226 >>>.**

  We have three resources, A, B, and C. A has 7 instances, B has 2 instances, and C has 6 instances. At this time, the allocation, etc. looks like this:

		←	Alloc	→		←	Req	→		←	Avail	→
		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 there a sequence that will allow deadlock to be avoided? Is there more than one sequence that will work?

- Suppose the Request matrix is changed like this. In other words, the maximum amounts to be allocated are initially declared so that this request matrix results.

		←	Alloc	→		←	Req	→		←	Avail	→
		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 there now a sequence that will allow deadlock to be avoided?

USAGE OF THIS DETECTION ALGORITHM

Frequency of check depends on how often a deadlock occurs and how many processes will be affected.

RECOVERY:

So, the deadlock has occurred. Now, how do we get the resources back and gain forward progress?

PROCESS TERMINATION:

- Could delete all the processes in the deadlock -- this is expensive.
- Delete one at a time until deadlock is broken (time consuming).
- Select who to terminate based on priority, time executed, time to completion, needs for completion, or depth of rollback
- In general, it's easier to preempt the resource, than to terminate the process.

RESOURCE PREEMPTION:

- Select a victim - which process and which resource to preempt.
- Rollback to previously defined "safe" state.
- Prevent one process from always being the one preempted (starvation).

COMBINED APPROACH TO DEADLOCK HANDLING:

- Type of resource may dictate best deadlock handling. Look at ease of implementation, and effect on performance.
- In other words, there is no one best technique.
- Cases include:

  Preemption for memory,

  Preallocation for swap space,

  Avoidance for devices (can extract Needs from process.)