OPERATING SYSTEMS

DEADLOCKS

OPERATING SYSTEM
Deadlocks

What Is In This Chapter?

- What is a deadlock?
- Staying Safe: Preventing and Avoiding Deadlocks
- Living Dangerously: Let the deadlock happen, then detect it and recover from it.

DEADLOCKS

EXAMPLES:

- "It takes money to make money".
- You can't get a job without experience; you can't get experience without a job.

BACKGROUND:

The cause of deadlocks:

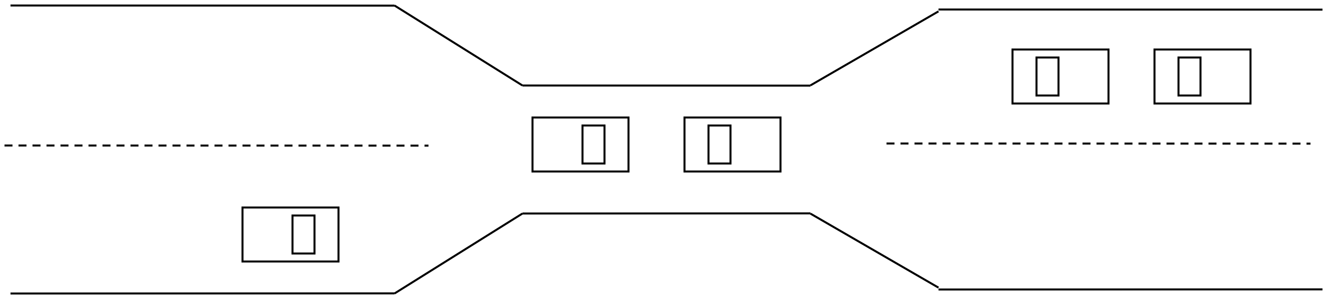
Each process needing what other process has. This results from sharing resources such as memory, devices, links.

Under normal operation, a resource allocations proceed like this::

1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.

DEADLOCKS

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

DEADLOCKS

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.
- If any other process requests this resource, then that process must wait for the resource to be released.

Hold and Wait

- A process holds a resource while waiting for another resource.
- A process simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

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.

DEADLOCKS

RESOURCE ALLOCATION GRAPH

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

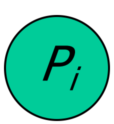
G = (V, E) The graph contains nodes and edges.

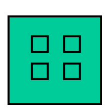
V Nodes consist of processes = { P1, P2, P3, ...} and resource types { R1, R2, ...}

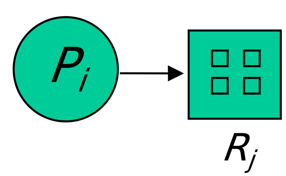
E Edges are (Pi, Rj) or (Ri, Pj)

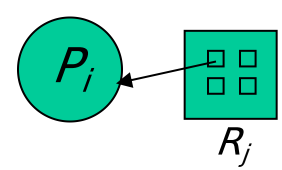
An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.

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.







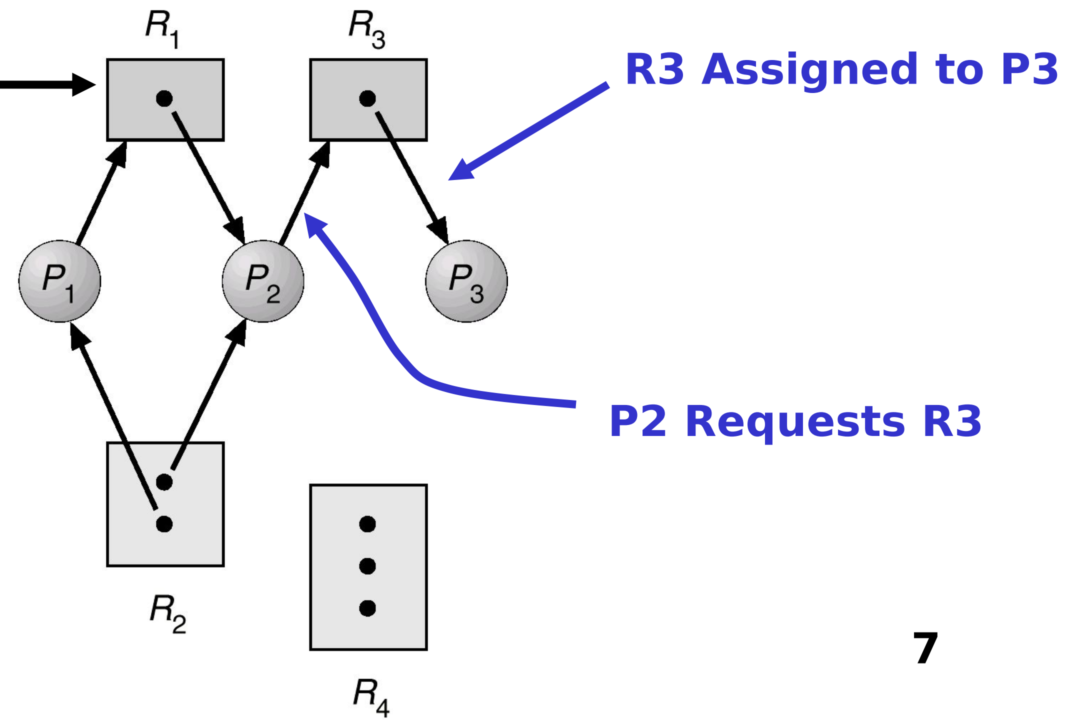


DEADLOCKS

RESOURCE ALLOCATION GRAPH

- 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.

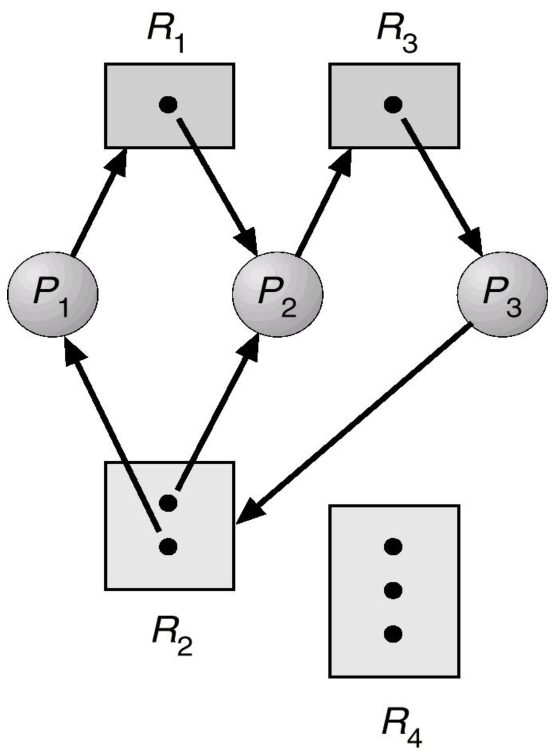
Resource allocation graph



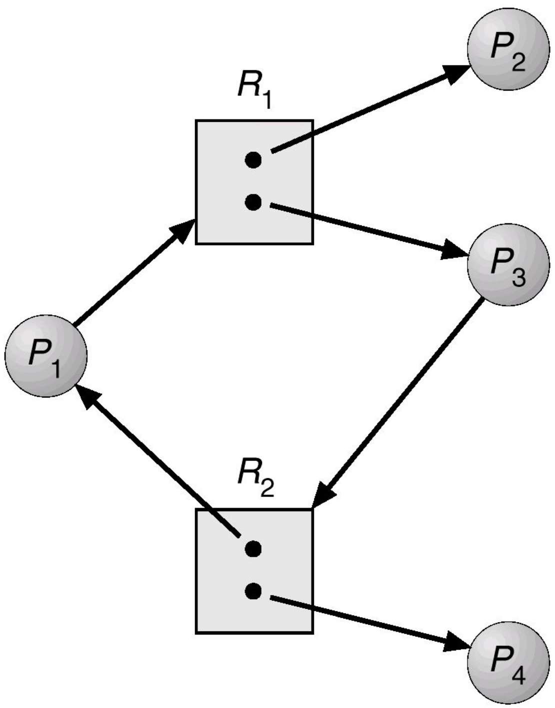
DEADLOCKS

RESOURCE ALLOCATION GRAPH

Resource allocation graph with a deadlock.



Resource allocation graph with a cycle but no deadlock.



DEADLOCKS

HOW TO HANDLE DEADLOCKS – GENERAL STRATEGIES

There are three methods:

1. Ignore Deadlocks: ← **Most Operating systems do this!!**

2. Ensure deadlock never occurs using either

Prevention Prevent any one of the 4 conditions from happening.

Avoidance Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations.

3. Allow deadlock to happen. This requires using both:

Detection Know a deadlock has occurred.

Recovery Regain the resources.

DEADLOCKS

Deadlock 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 holding. A sequence of resources is always collected beginning with the same one.

c) Utilization is low, starvation possible.

DEADLOCKS

Deadlock Prevention

Do not allow one of the four conditions to occur.

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.

b) EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.

c) Prevention is generally the easiest to implement.

DEADLOCKS

Deadlock 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.

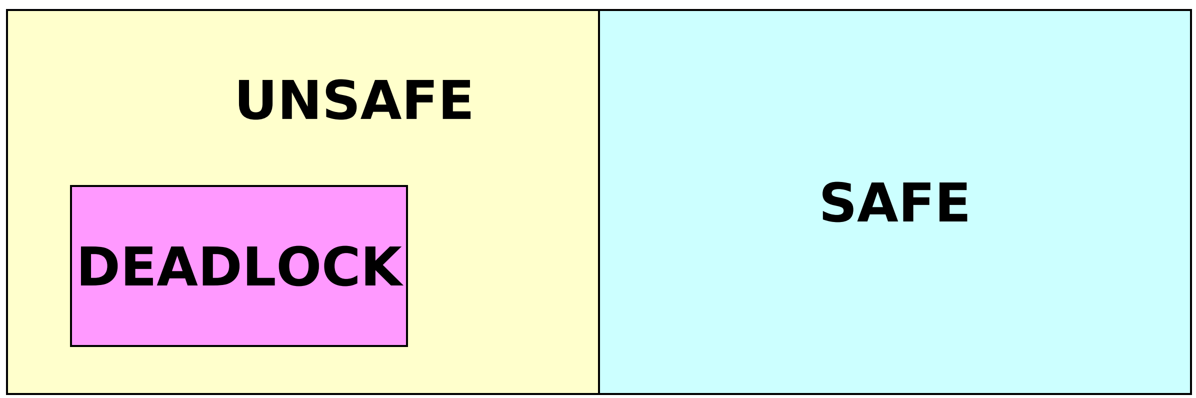
The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

DEADLOCKS

Deadlock Avoidance

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.



Only with luck will processes avoid deadlock.

O.S. can avoid deadlock.

DEADLOCKS

Deadlock Avoidance

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 all the processes, there's a way to avoid deadlock:

The algorithm uses these variables:

Need[I] – the remaining resource needs of each process.

Work - Temporary variable – how many of the resource are currently available.

Finish[I] – flag for each process showing we ' ve analyzed that process or not.

need <= available + allocated[0] + .. + allocated[I-1] **<- Sign of success**

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

Safety Algorithm

1. Initialize work = available
Initialize finish[i]= false,

2. for i = 1,2,3,..n
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.

DEADLOCKS

Deadlock Avoidance

- Process P0, P1 and P2 compete for 12 tape drive, then consider the following case:

Processes	Max(Need)	Current Usage	Could ask for
P0	10	5	5
P1	4	2	2
P2	9	2	7

- Total allocated resources: 9
- Available resources are: 12 – 9 = 3
- At time T0 processes are in safe state, the sequence for which is < P1, P0, P2>
- Because it satisfies the safety conditions and the safe sequences can be calculated as follows:
  - P1 can immediately be allocated to all its resource requirement and then return to the system
  - Now available resources becomes: 3 + 2 = 5
  - P0 can get all its resources and return them to the system. Need of the P0 is 5 and system available resource is 5.
  - Now available resources = 10
  - Now P2 can be allocated required resources (i.e. requirement = 7 and available is = 10) Then after resources can be released and available resource = 10 + 2 = 12.(equals no. of tape drive)

DEADLOCKS

Deadlock Avoidance

Safety Algorithm

Do these examples:

Consider a system with: five processes, P0 – P4, three resource types, A, B, C.

Type A has 10 instances, B has 5 instances, C has 7 instances.

At time T0 the following snapshot of the system is taken.

Max Needs = allocated + can-be-requested

Is the system in a safe state?

	Alloc			Req			Avail		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

DEADLOCKS

Deadlock Avoidance

Safety Algorithm

Do these examples:

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. (We've drawn the chart as if it's granted.

Can the request be granted?

			Alloc				Req				Avail	
		A	B	C		A	B	C		A	B	C
P0		0	1	0		7	4	3		1 #	3	0 #
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				

DEADLOCKS

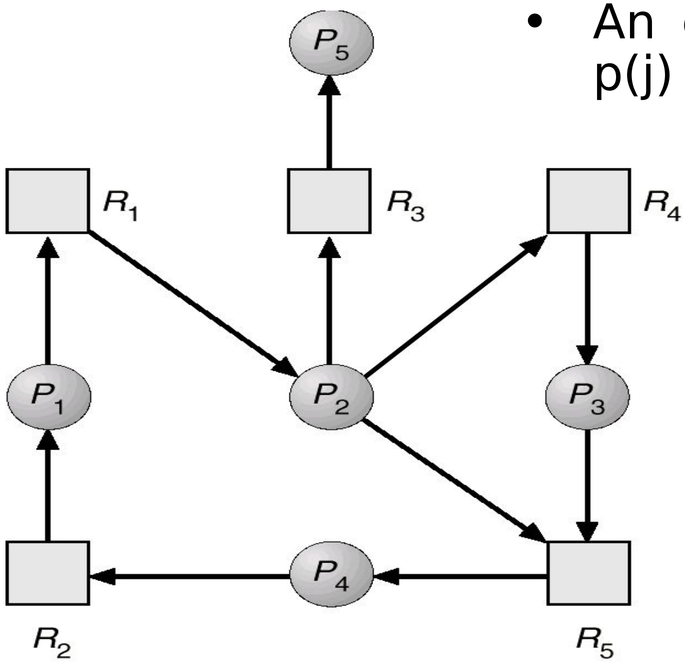
Deadlock Detection

Need an algorithm that determines if deadlock occurred.

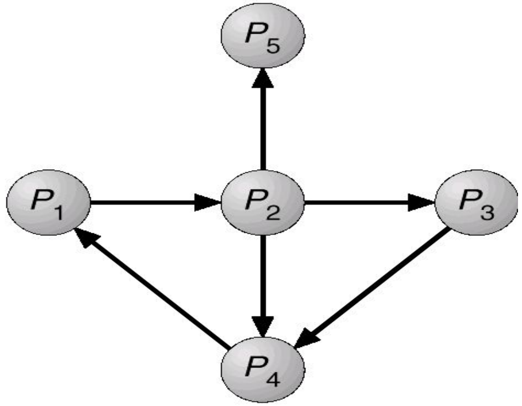
Also need a means of recovering from that deadlock.

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(j) is waiting for p(i) to release.



(a)



(b)

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.

Deadlock Detection

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.

Deadlock Detection

EXAMPLE

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:

Is there a sequence that will allow deadlock to be avoided?

Is there more than one sequence that will work?

			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				

DEADLOCKS

Deadlock Detection

EXAMPLE

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.

Is there now a sequence that will allow deadlock to be avoided?

		“	**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				

USAGE OF THIS DETECTION ALGORITHM

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

Deadlock 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).

Deadlock Recovery

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.)

DEADLOCKS

WRAPUP

In this section we have:

Looked at necessary conditions for a deadlock to occur.

Determined how to prevent, avoid, detect and recover from deadlocks.