3.3 Deadlock

Deadlock occurs when each process in a set of process controls a resource that another process in the set has requested. Each process blocks waiting for its requested resource to become available. The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and *CPU* Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

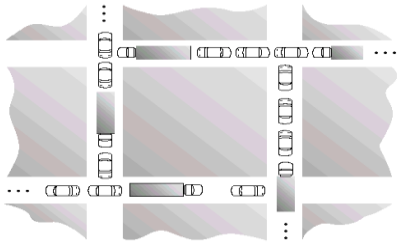
Under the normal mode of operation, a process may utilize a resource in only the following sequence:

- ***Request:*** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- ***Use:*** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- ***Release:*** The process releases the resource.

3.3.1 Conditions of deadlock

- **Mutual Exclusion Condition:** The resources involved are non-shareable.
  Explanation: At least one resource (thread) must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and Wait Condition:** Requesting process hold already allocated resources while waiting for requested resources.
  Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.
- **No-Preemptive Condition:** Resources already allocated to a process cannot be preempted.
  Explanation: A resource cannot be removed from a process using it till completion or released voluntarily by the process holding it.
- **Circular Wait Condition:** The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

As an example, consider the traffic deadlock in the following figure



Consider each section of the street as a resource.

- *Mutual exclusion* condition applies, since only one vehicle can be on a section of the street at a time.
- *Hold-and-wait* condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
- *No-preemptive* condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
- *Circular wait* condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

3.3.2 Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

Mutual Exclusion

- Not required for sharable resources; must hold for non-sharable resources.

Hold and Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources.
- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
- Low resource utilization; starvation possible.

No Preemption

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

- Impose a total ordering of all resource types, and require that each process requests resources in

an increasing order of enumeration.

3.3.3 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the **Banker's algorithm**; so named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

Banker's Algorithm

In this analogy

Customers	≡	processes
Units	≡	resources, say, tape drive
Banker	≡	Operating System

Customers	**Used**	**Max**	
A	0	6	Available Units = 10
B	0	5	
C	0	4	
D	0	7	
	Fig. 1		

In the above figure, we see four customers each of whom has been granted a number of credit units. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	**Used**	**Max**	
A	1	6	Available Units = 2
B	1	5	
C	2	4	
D	4	7	
	Fig. 2		

Safe State: the key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure 2 is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State: Consider what would happen if a request from B for one more unit were granted in above figure 2. We would have following situation

Customers	Used	Max	
A	1	6	Available Units = 1
B	2	5	
C	2	4	
D	4	7	
	Fig. 3		

This is an unsafe state.
If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

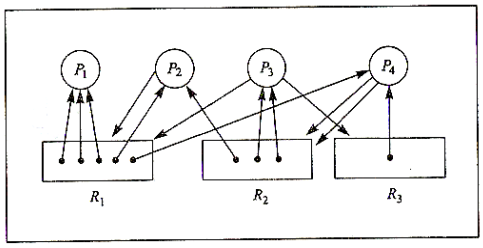
Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

3.3.4 Deadlock detection

- A resource allocation graph (RAG) may be used to model the state of resource allocations and requests.
- Processes are drawn as circles, resources as rectangles. Within a resource rectangle, a dot is drawn for each of that resource.
- For each pending resource request, a direct arrow is drawn from the process to the resource rectangle.
- For each granted request, a direct arrow is drawn from a resource dot to the process.

Resource Allocation Graph



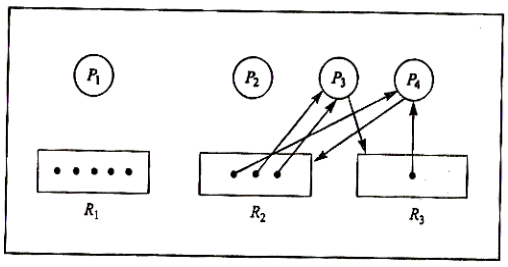
Resource Usage

Process	Current Allocation			Outstanding Requests			Resources Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	3	0	0	0	0	0	0	0	0
P_2	1	1	0	1	0	0			
P_3	0	2	0	1	0	1			
P_4	1	0	1	0	2	0			

- A reduced RAG can be used to determine whether or not deadlock exists. To reduce a RAG, the

arrows associated with each process and each resource are checked.

- If a resource has only arrows pointing away from it (meaning it has no requests pending), erase all its arrows.
- If a process has only arrows pointing toward it (meaning all of its requests have been granted), erase all its arrows.
- If a process has arrows pointing away from it, but for each such request arrow there is an available resource dot (a dot without an arrow leading from it) in the resource the arrow points to, erase all the process's arrows.



- If in checking all the processes, at least one process was found for which its arrows could be erased, go back and reiterate the process. Continue until there are either no arrows remaining or no process can have its arrows removed.
- The system is deadlocked if and only if arrows remain.

3.3.5 Recovery from deadlock

a) Automatic pre-emption

- The OS pre-empts a subset of the allocated resources. the issues to be resolved here include:

i) Selection: i.e. would resources from which processes are to be pre-empted and if the decision is made to pre-empt one resource from a process, should all its other resources also be pre-empted. Other factors include:

- A process's priority
- How long a process has been executing?
- How long a process needs to execute?
- How many resources a process currently holds?
- The size of the different sets of resources that if released , would break the deadlock
- How many processes would be affected?

ii) Consequence: i.e what happens to the processes that have resources pre-empted.

- Can the system maintain state information about processes such that if they are victims of pre-emption, they can be rolled back to an earlier state.
- Maintaining such information can be costly in terms of both time and storage space.
- Rolling a process back to its starting state may not be practical since some input data may be unrecoverable. Unless that data was captured and saved when the process was first run, the ability to process that information will have been lost.

iii) Starvation: if the same process is repeatedly a victim of pre-emption, constant rollbacks could preclude it from terminating.

- Will the selection mechanism preclude the possibility of starvation? This can be solved by using a **pre-emption counter.** Each time a process is rolled back, the pre-emption counter is incremented. Should deadlock occur, no resources would be pre-empted from the process with the highest pre-emption counter.

b) Automatic termination

- Eliminates deadlock by terminating processes.
- All deadlocked processes could be terminated or the system could select a subset of the deadlocked process to terminate.
- Although drastic, terminating all processes can be done quickly on systems where fast resolution is important.
- If only a subset of processes are to be terminated, the same factors encountered in automatic pre-emption may be considered in selecting the process to be terminated.
- Termination of one process may have effects beyond that process. Processes dependent on the terminated process's computation may be adversely affected and partially updated files may be left in inconsistent states.

c) Manual intervention

- Turns the responsibility of resolving the problem over to the system operator. Except that some systems run without a full-time (or even part-time) operator and others must react to a deadlock situation within moments of its detection. This solution isn't practical.

Conclusion: All the mechanisms for dealing with deadlocks presented thus far have had one or more serious drawbacks. There is no good way of dealing with deadlocks. So in many cases, the problem of deadlock is ignores, like an ostrich sticking its head in the sand and hoping the problem will go away!

Revision Questions

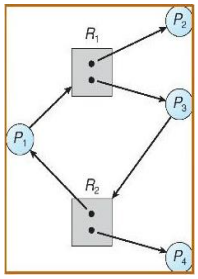
Q1. i) Define **concurrency** as used in interprocess communication.
ii) Describe the techniques used in realising inter-process communication

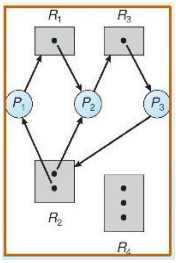
Q2. Clearly explain the concept of *no indivisibility*. Show how it relates to access to critical section.

Q3. Define each of the following resource types and State **one** example of each type.
i) A preemptive resource.
ii) A non preemptive resource.

Q4. List **three** typical computer applications where deadlock is a serious problem.

Q5. Consider the following resource allocation graphs and determine which of them is deadlocked and which one is not.





a

b

Q6. Below are two Process A and Process B. They use blocking send and receive primitives for synchronization and communication. What are the consequences for Process A and B using nonblocking send, blocking receive if communication is considered unreliable?

Process A :
```
send (B, a_message);
receive(B, b_message);
```

Process B:
```
receive (A, a_message);
send(A, b_message);
```

Q9. What is *busy waiting*?