Operating Systems: Internals and Design Principles

Chapter 6 Concurrency: Deadlock and Starvation

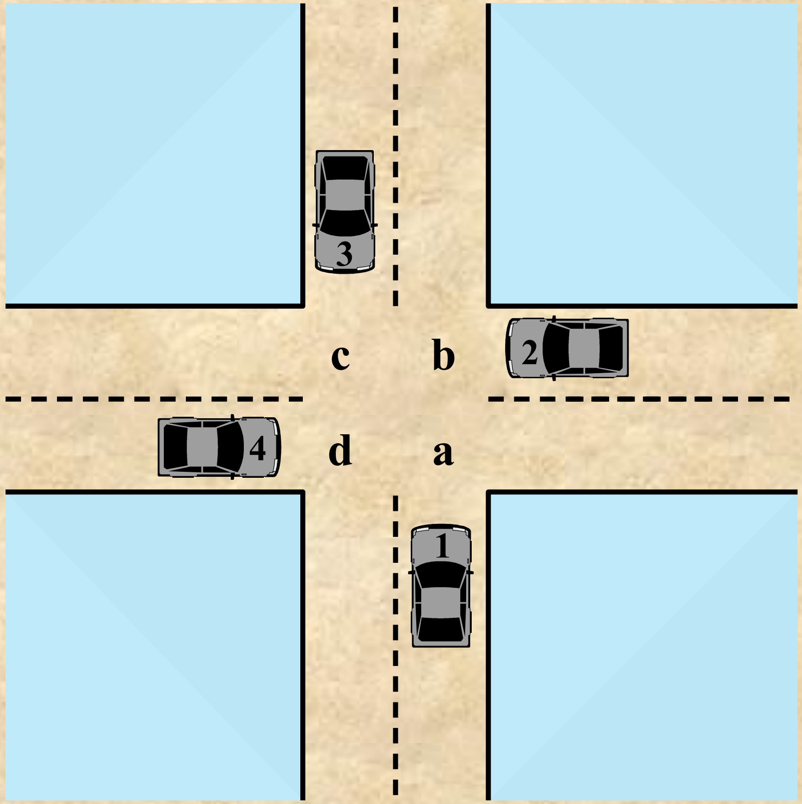
Ninth Edition
By William Stallings

Let's look at an example

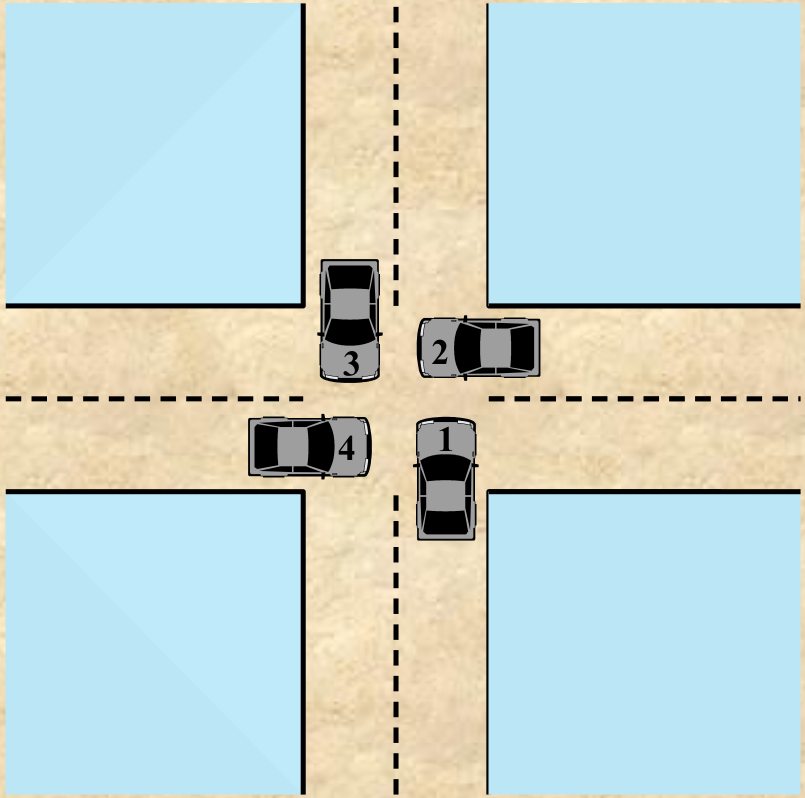
- Suppose there is a narrow staircase.
- Two people, one going down and one going up meets up in the middle.
- Neither can move anywhere except go back.
- Each of them are waiting for another person to move.
  - Deadlock
- Suppose you have slightly wider staircase
  - What happens now?
  - If they try to move to opposite direction at the same time?
    - Livelock
    - Two or more processes or threads continuously change their states in response to each other but make no actual progress.

Deadlock

- The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible



(b) Deadlock

Figure 6.1 Illustration of Deadlock

Let's look at another example

- Suppose we have processes P, Q

P

- Get A
- Get B
- Do work
- Release A
- Release B

Q

- Get B
- Get A
- Do work
- Release B
- Release A

- Joint resource diagram

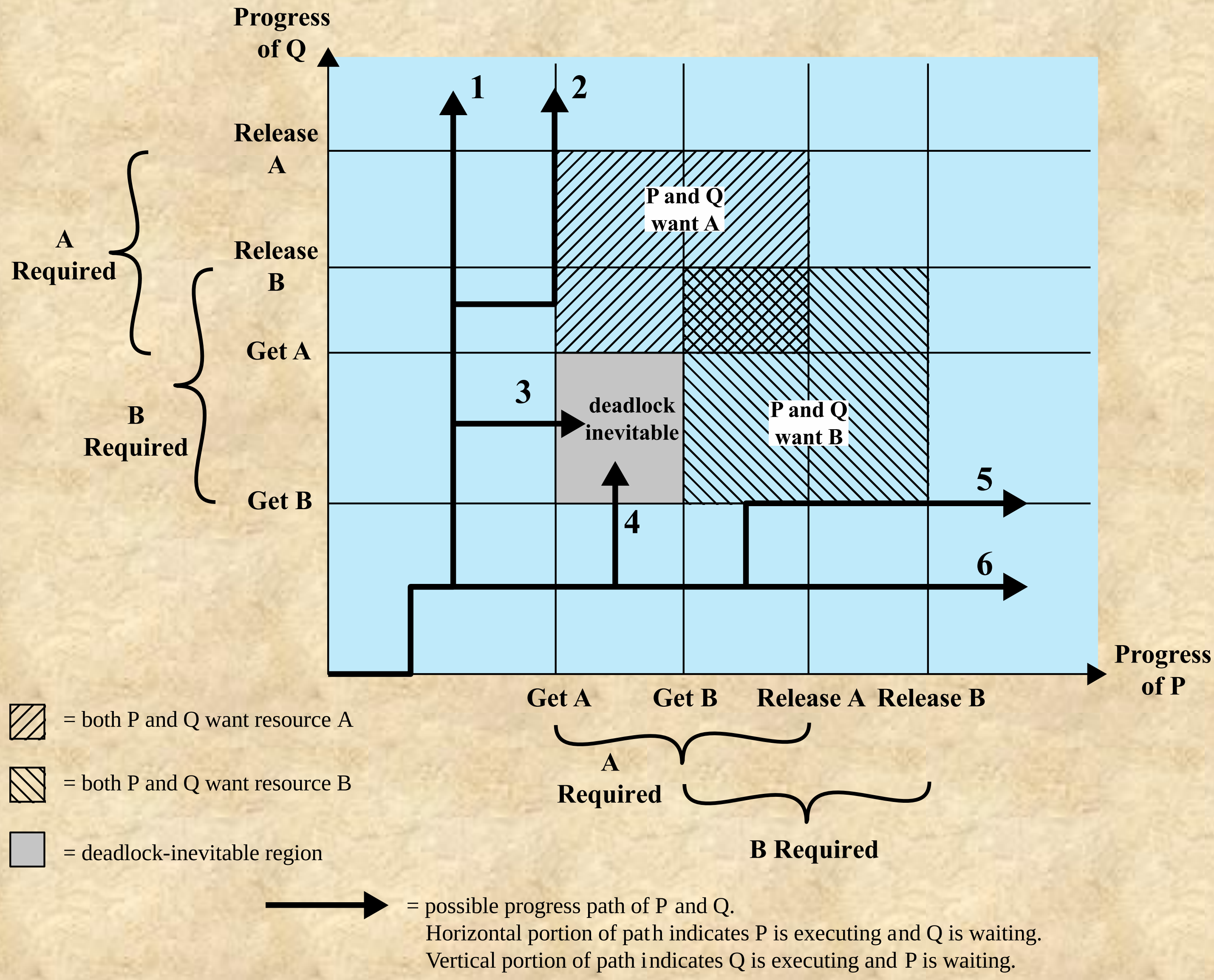


Figure 6.2 Example of Deadlock

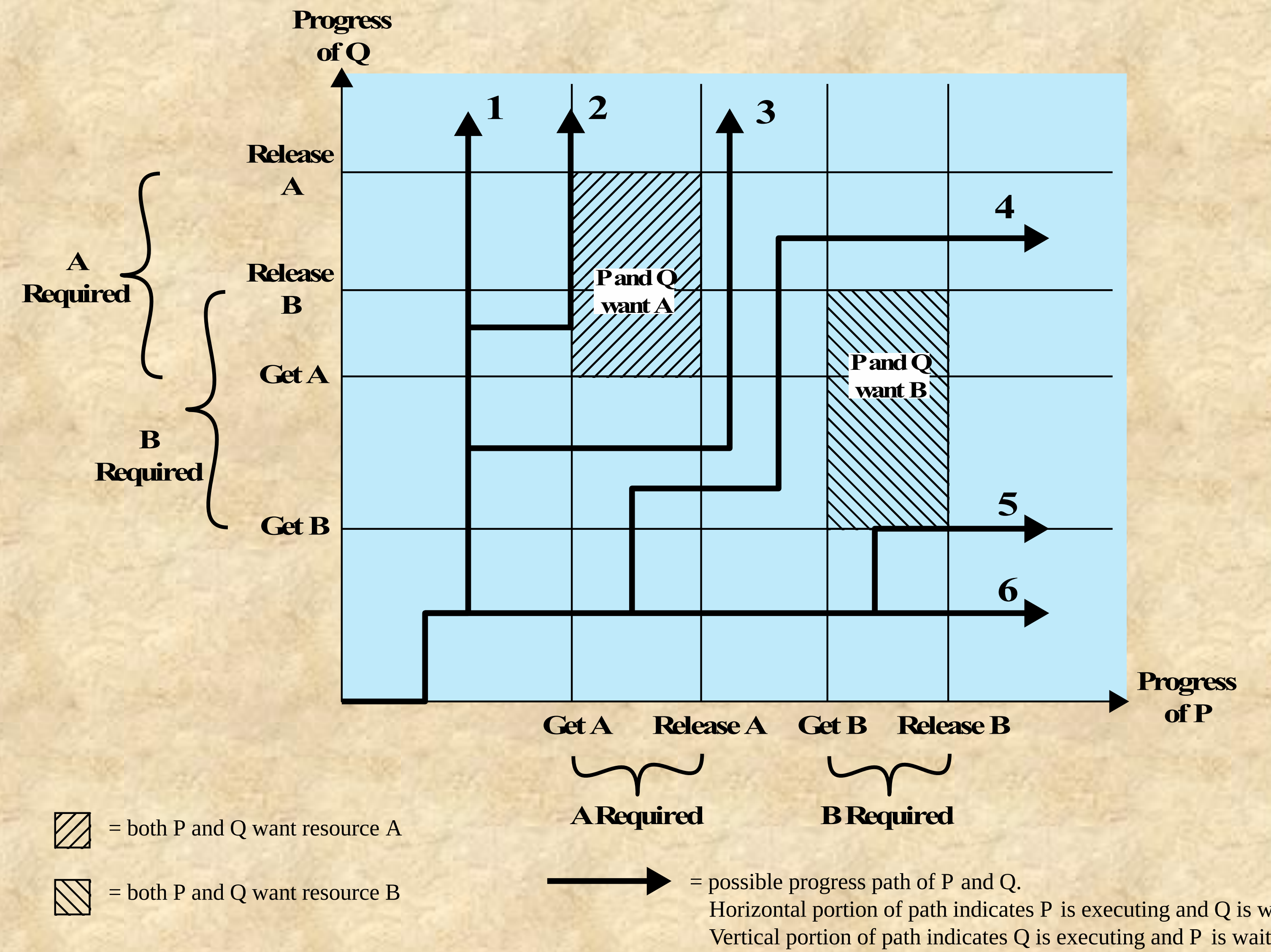


Figure 6.3 Example of No Deadlock

Fatal Regions

- Deadlock only occurs if two processes (or more) creates a path that enters the fatal region.
- What if there are more than 2 processes that shares resources
  - We need joint progress diagrams that uses more than two dimensions.

Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- One that can be created (produced) and destroyed (consumed)
  - Interrupts, signals, messages, and information
  - In I/O buffers

Memory Request example

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:

P1	P2
. . .	. . .
Request 80 Kbytes;	**Request 70 Kbytes;**
. . .	. . .
Request 60 Kbytes;	**Request 80 Kbytes;**

- Deadlock occurs if both processes progress to their second request

Consumable resources

- Can you produce a scenario where attempting access consumable resources by multiple processes ends up in a deadlock?

Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

- Deadlock occurs if the Receive is blocking

Resource categories

- Deadlocks with Reusable Resources: common and straightforward.
- Deadlocks with Consumable Resources:
  - Less typical but possible
  - If the production and consumption of resources are interdependent.
    - Such deadlocks might involve indirect circular waits.

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• Only one process may use a resource at a time • No process may access a resource until that has been allocated to another process	• A process may hold allocated resources while awaiting assignment of other resources	• No resource can be forcibly removed from a process holding it	• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

How to model a deadlock?

- Resource allocation graphs
- Directed graph that depicts a state of the system of resources and processes
- Each process and each resource represented by a node
- Edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.
- Edge directed from a reusable resource node dot to a process indicates a request that has been granted

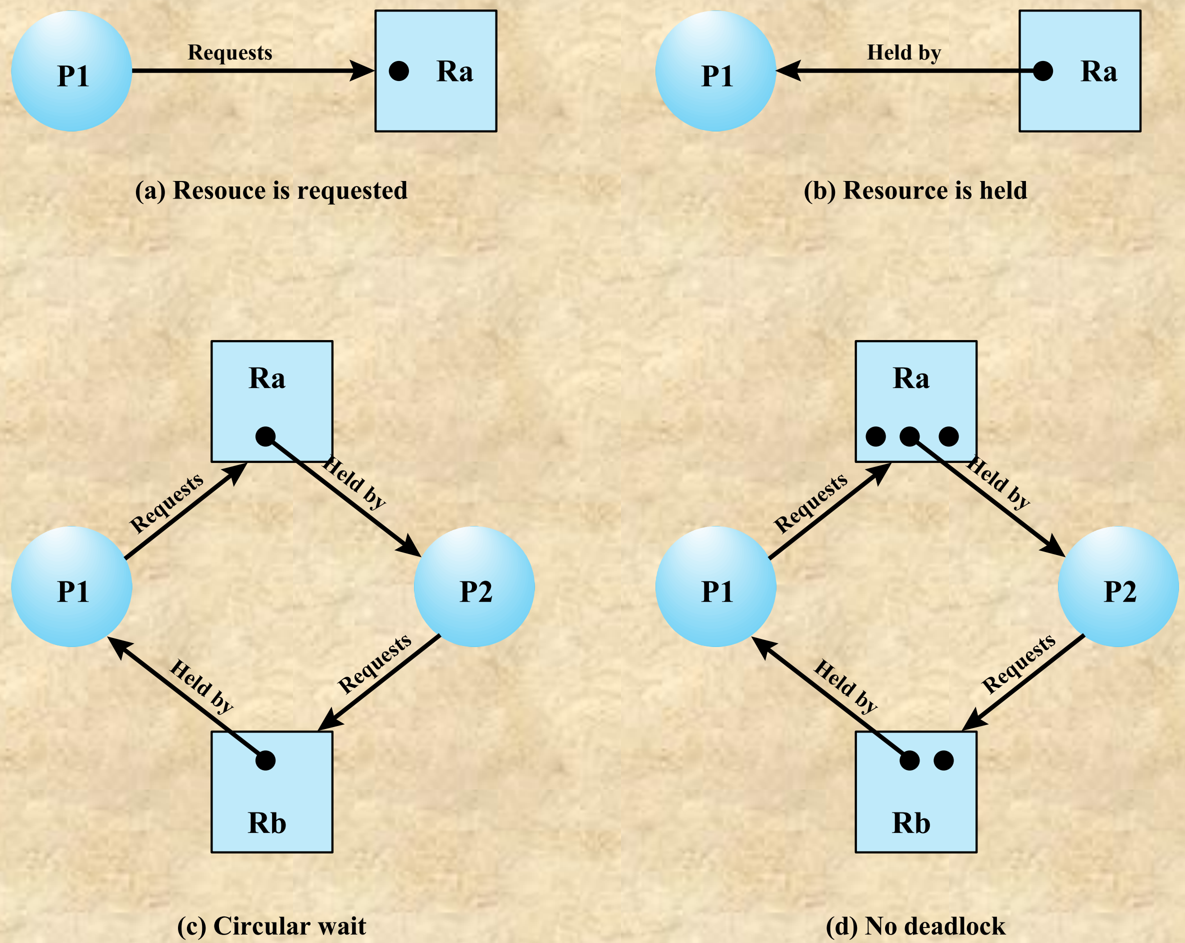


Figure 6.5 Examples of Resource Allocation Graphs

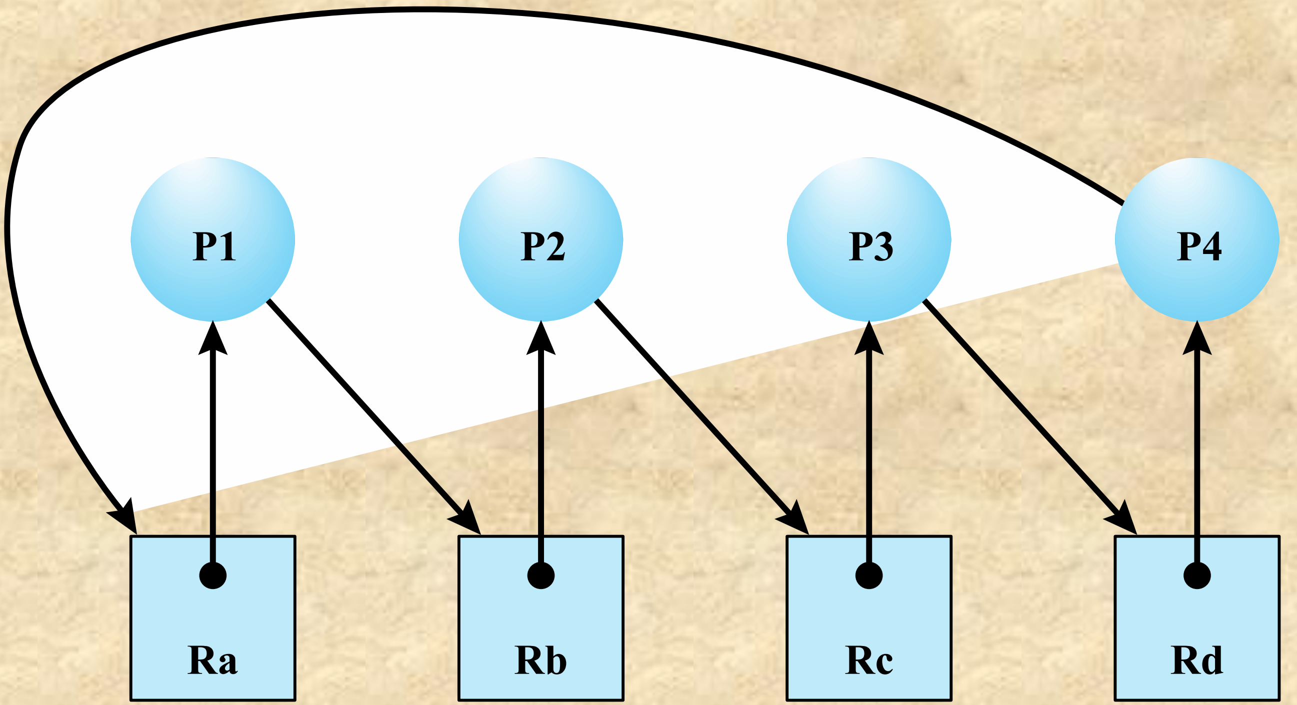


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Deadlock Approaches

- Can you think about an effective way to deal with deadlocks?
- **Deadlock avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock
- **Deadlock prevention**
  - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening
- **Deadlock detection**
  - Grant resource requests, when possible, but periodically check for the presence of deadlock and take action to recover

Difference between deadlock prevention and avoidance

- **Deadlock prevention** involves designing the system and its resource allocation protocols to ensure that at least one of the necessary conditions for deadlock cannot occur.
  - **Using a resource allocation graph with specific rules to avoid circular waits can help prevent deadlocks**
- **Deadlock avoidance** involves dynamically examining the resource allocation state of the system to ensure that each resource request can be granted without leading to a deadlock.
  - **When a process requests a resource, the system temporarily pretends to allocate the resource and checks if this allocation keeps the system in a safe state.**

Deadlock Prevention

- Design a system in such a way that the possibility of deadlock is excluded
- We can remove one of the conditions of deadlock from the system.
- Two main methods:
  - Indirect
    - Prevent the occurrence of one of the three necessary conditions
  - Direct
    - Prevent the occurrence of a circular wait

Deadlock Condition Prevention

- Mutual exclusion
  - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
  - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
  - Even in this case, deadlock can occur if more than one process requires write permission
- Hold and wait
  - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously

Deadlock Condition Prevention

- No Preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources
- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types

Deadlock Condition Prevention

- Most of these prevention strategies are inefficient or unfavorable.

Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- Requires knowledge of future process requests

Two Approaches to Deadlock Avoidance

Deadlock Avoidance

Resource Allocation Denial

- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Process Initiation Denial

- Do not start a process if its demands might lead to deadlock

Resource Allocation Denial

- Referred to as the *banker's algorithm*
- ***State*** of the system reflects the current allocation of resources to processes
- ***Safe state*** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock
- ***Unsafe state*** is a state that is not safe

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Figure 6.7 Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Figure 6.7 Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Figure 6.7 Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – **A**

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Figure 6.7 Determination of a Safe State

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – **A**

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – **A**

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                               /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource allocation algorithm

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found) {                        /* simulate execution of Pk */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

Deadlock Avoidance Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Deadlock Strategies

Deadlock prevention strategies are very conservative

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- Resource requests are granted whenever possible

Deadlock Detection Algorithm

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
- CES algorithm

Advantages:

- It leads to early detection
- The algorithm is relatively simple

Disadvantage

- Frequent checks consume considerable processor time

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Integrated Deadlock Strategy

- Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations
  - Group resources into a number of different resource classes
  - Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes
  - Within a resource class, use the algorithm that is most appropriate for that class

- Classes of resources
  - Swappable space
    - Blocks of memory on secondary storage for use in swapping processes
  - Process resources
    - Assignable devices, such as tape drives, and files
  - Main memory
    - Assignable to processes in pages or segments
  - Internal resources
    - Such as I/O channels

Class Strategies

- Within each class the following strategies could be used:
  - **Swappable space**
    - Prevention of deadlocks by requiring that all of the required resources that may be used be allocated at one time, as in the hold-and-wait prevention strategy
    - This strategy is reasonable if the maximum storage requirements are known
  - **Process resources**
    - Avoidance will often be effective in this category, because it is reasonable to expect processes to declare ahead of time the resources that they will require in this class
    - Prevention by means of resource ordering within this class is also possible
  - **Main memory**
    - Prevention by preemption appears to be the most appropriate strategy for main memory
    - When a process is preempted, it is simply swapped to secondary memory, freeing space to resolve the deadlock
  - **Internal resources**
    - Prevention by means of resource ordering can be used