# CS415 Module 4 Part C - Dealing with Deadlocks

Athens State University

**Outline**

## Contents



## 1 The Concept

**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
- Think about how a four-way stop operates on a highway
- Deadlocks are permanent and have no efficient solution

Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

**Joint Progress Diagram**

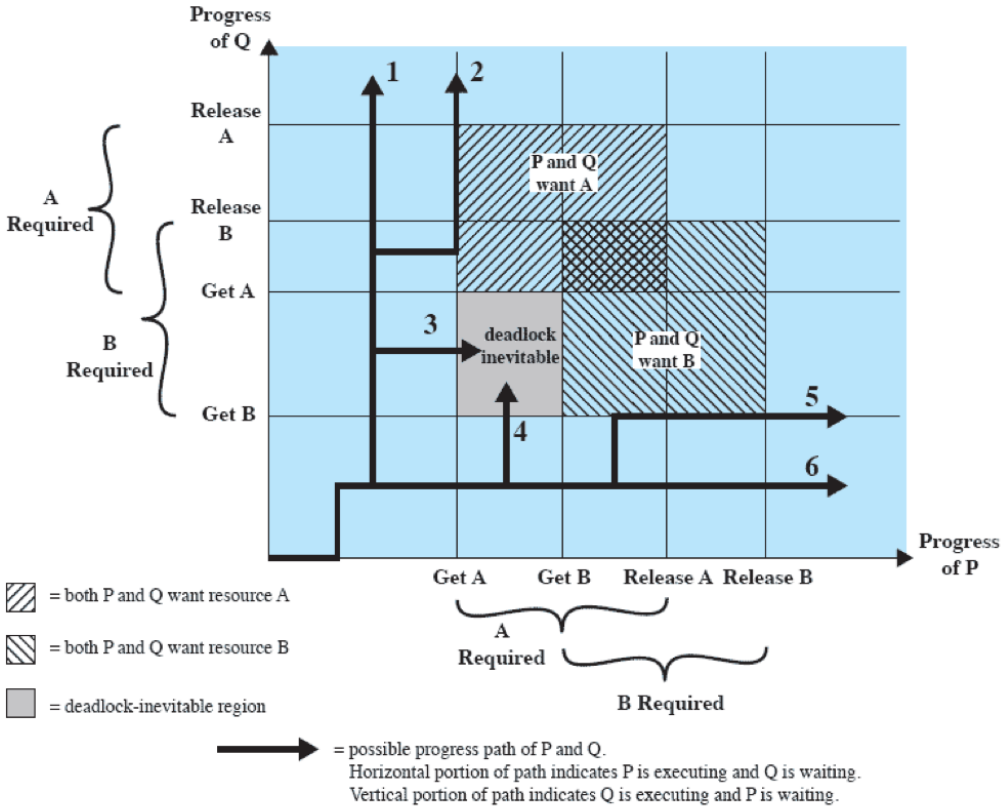


**Figure 6.2 Example of Deadlock**

Let us now look at a depiction of deadlock involving processes and computer resources. The joint progress diagram illustrates the progress of two processes competing for two resources. Each process needs exclusive use of both resources for a certain period of time.

In the figure , the x -axis represents progress in the execution of P and the y –axis represents progress in the execution of Q. The joint progress of the two processes is therefore represented by a path that progresses from the origin in a northeasterly direction. For a uniprocessor system, only one process at a time may execute, and the path consists of alternating horizontal and vertical segments, with a horizontal segment representing a period when P executes and Q waits and a vertical segment representing a period when Q executes and P waits. The figure indicates areas in which both P and Q require resource A (upward slanted lines); both P and Q require resource B (downward slanted lines); and both P and Q require both resources. Because we assume that each process requires exclusive control of any resource, these are all forbidden regions; that is, it is impossible for any path representing the joint execution progress of P and Q to enter these regions.

The figure shows six different execution paths. These can be summarized as follows:

1. Q acquires B and then A and then releases B and A. When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
4. P acquires A and then Q acquires B. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.

5. P acquires A and then B. Q executes and blocks on a request for B. P releases A and B. When Q resumes execution, it will be able to acquire both resources.

6. P acquires A and then B and then releases A and B. When Q resumes execution, it will be able to acquire both resources.

The gray-shaded area of the joint progress diagram , which can be referred to as a fatal region , applies to the commentary on paths 3 and 4. If an execution path enters this fatal region, then deadlock is inevitable. Note that the existence of a fatal region depends on the logic of the two processes. However, deadlock is only inevitable if the joint progress of the two processes creates a path that enters the fatal region.

**Joint Progress Diagram**

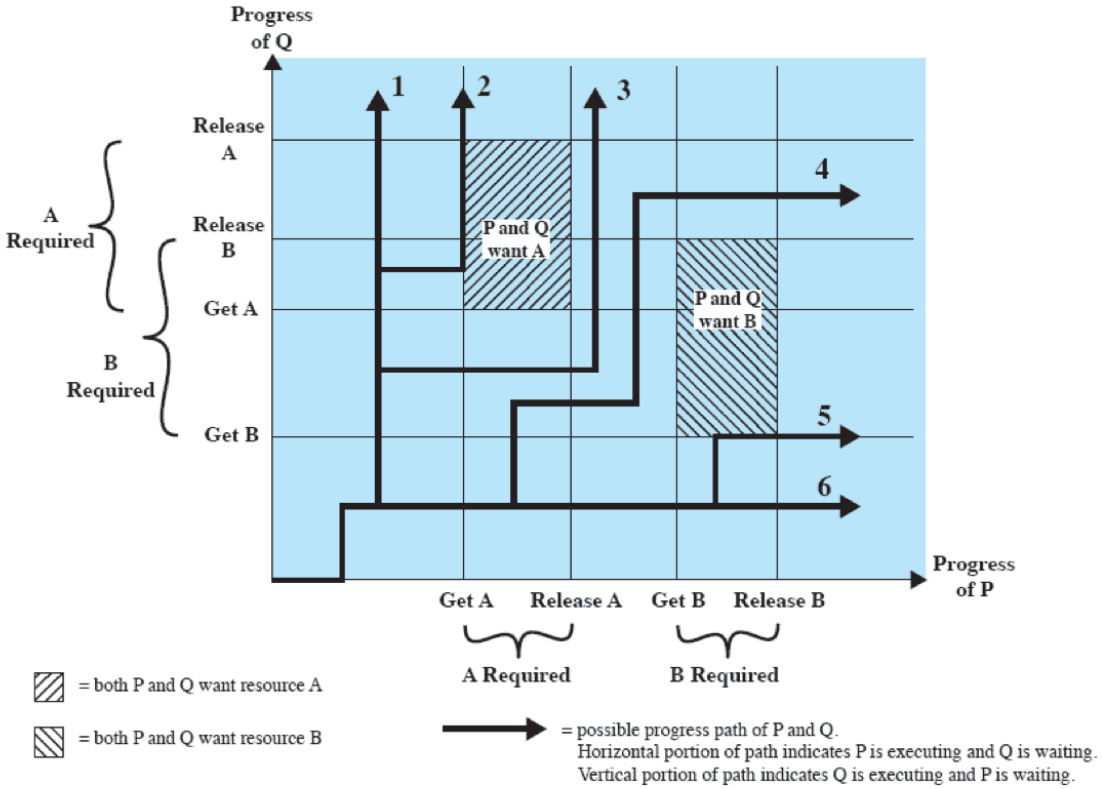


**Figure 6.3 Example of No Deadlock [BACO03]**

Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application.

This situation is reflected in the joint progress diagram in this slide. Some thought should convince you that regardless of the relative timing of the two processes, deadlock cannot occur.

As shown, the joint progress diagram can be used to record the execution history of two processes that share resources. In cases where more than two processes may compete for the same resource, a higher-dimensional diagram would be required. The principles concerning fatal regions and deadlock would remain the same.

# 2 Examples of Deadlocks

**Resource Categories**

**Reusable Resources**

- Can be safely used by only process at a time and is not depleted by the use
- Examples: processors, I/O channels, memory

**Consumable Resources**

- Resources that can be created (produced) and destroyed (consumed)
- Examples: interrupts, signals, messages, information

**Example: Process Resource Contention**

**Process P**

| Step | Action |
|---|---|
| 0 | Request(D) |
| 1 | Lock(D) |
| 2 | Request(T) |
| 3 | Lock(T) |
| 4 | Perform function |
| 5 | Unlock(D) |
| 6 | Unlock(T) |

**Process Q**

| Step | Action |
|---|---|
| 0 | Request(T) |
| 1 | Lock(T) |
| 2 | Request(D) |
| 3 | Lock(C) |
| 4 | Perform function |
| 5 | Unlock(T) |
| 6 | Unlock(D) |

As an example of deadlock involving reusable resources, consider two processes that compete for exclusive access to a disk file D and a tape drive T. Deadlock occurs if each process holds one resource and requests the other. For example, deadlock occurs if the multiprogramming system interleaves the execution of the two processes as follows:

```
p0 p1 q0 q1 p2 q2
```

It may appear that this is a programming error rather than a problem for the OS designer. However, we have seen that concurrent program design is challenging. Such deadlocks do occur, and the cause is often embedded in complex program logic, making detection difficult. One strategy for dealing with such a deadlock is to impose system design constraints concerning the order in which resources can be requested.

**Deadlock Due To Resource Exhaustion**

- Space is available for allocation of 200KB
- Process 1 makes 2 memory requests: request for 80KB followed soon after by a request for 60KB
- Process 2 at the same time makes requests for 70KB with a request soon afterwards for 80KB
- Deadlock occurs if both processes progress to the their second request

**Wait to Receive Deadlock**

- Consider two processes that attempt to receive a message from the other process and then send a message to the other process
- A deadlock occurs if the act of receiving a message is blocking

As an example of deadlock involving consumable resources, consider the following 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.

Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received). Once again, a design error is the cause of the deadlock. Such errors may be quite subtle and difficult to detect. Furthermore, it may take a rare combination of events to cause the deadlock; thus a program could be in use for a considerable period of time, even years, before the deadlock actually occurs.

# 3 Detecting and Avoiding Deadlocks

**The Four Necessary Conditions for Deadlock**

- Mutual Exclusion
  - Only one process may use a resource at a time
- Hold-and-Wait
  - A process may hold allocated resources while awaiting assignment of the others
- No Pre-emption
  - No resource can be forcibly removed from a process holding it
- Circular Wait
  - A closed chain of processes exist so that each process holds at least one resource needed by the next process in the chain

**Three Approaches to Dealing With Deadlock**

- Prevent
  - Adopt a policy that eliminates one of the necessary conditions
- Avoid
  - Make the appropriate dynamic choices based on the current state of resource allocation
- Detect
  - Attempt to detect the presence of deadlock and take corrective action

**Deadlock Prevention**

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - *Indirect*: prevent the occurrence of one of the three required conditions other than circular wait
  - *Direct*: Prevent circular waits

**Deadlock Condition Prevention**

- Mutual Exclusion
  - The OS has to do this, so can't not allow it
- Hold and Wait
  - Require a process request all required resources at one time
- No Preemption
  - If a request for resource is not granted, then force process to let go original reourses
- Circular wait
  - Define a linear ordering of resource types

Mutual Exclusion

In general, the first of the four listed conditions cannot be disallowed. 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

The hold-and-wait condition 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. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require.

There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

No Preemption

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources. This latter scheme would prevent deadlock only if no two processes possessed the same priority.

This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

Circular Wait

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R , then it may subsequently request only those resources of types following R in the ordering. To see that this strategy works, let us associate an index with each resource type. Then resource R i precedes R j in the ordering if $i < j$ . Now suppose that two processes, A and B, are deadlocked because A has acquired R i and requested R j , and B has acquired R j and requested R i . This condition is impossible because it implies $i < j$ and $j < i$ .

As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

## 3.1 Deadlock Avoidance

### Deadlock Avoidance

- Make dynamic decision whether or not a resource request leads to a deadlock; requires knowledge of future process requests
- Two approaches:
  - Resource Allocation Denial: Do not grant request if it might lead to deadlock
  - Process Initiation Denial: Do not start a process if its demands might lead to deadlock

### Resource Allocation Denial: The Banker's Algorithm

- **State**: current allocation of resources to processes
- **Safe state**: A state in which there is at least one sequence of resource allocations that does not lead to deadlock
- **Unsafe state**: A state that is not safe

The strategy of resource allocation denial, referred to as the banker's algorithm. Let us begin by defining the concepts of state and safe state. Consider a system with a fixed number of processes and a fixed number of resources. At any time a process may have zero or more resources allocated to it. The state of the system reflects the current allocation of resources to processes. Thus, the state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation, defined earlier. A safe state is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion). An unsafe state is, of course, a state that is not safe.

### Deadlock Avoidance Logic

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

function allocate_test() {
   if ((alloc[i, *] + request[I]) > claim[i,*])
      total request is greater than claim
   else if (request[*] > available[I])
      suspend process
   else {
      //define new state
      alloc[i,*] = alloc[i,*] + request[*];
      available[*] = available[*] - request[*];
   }
   if (new state is safe)
      permit allocation
   else {
      restore original state
      suspend process
   }
}
```

The main algorithm is shown in `allocate_test`. With the state of the system defined by the data structure state , `request[*]` is a vector defining the resources requested by process `i` . First, a check is made to assure that the request does not exceed the original claim of the process. If the request is valid, the next step is to determine if it is possible to fulfill the request (i.e., there are sufficient resources available). If it is not possible, then the process is suspended. If it is possible, the final step is to determine if it is safe to fulfill the request. To do this, the resources are tentatively assigned to process `i` to form `newstate`.

**Deadlock Avoidance Logic**

```
function safe(state S) {
   int currentavail[m];
   process rest[<number of processes]
   currentavail = available;
   rest = {all processes};
   possible = true;
   while(possible){
     find process in rest claim minus alloc is less than currentavail
     if (found) {
        increment currentavail by alloc for this process
        remove the process from the rest list
     }
     else possible = false
   }
   return (rest == NULL)
}
```

**Deadlock Avoidance Advantages**

- It is not required to preempt and rollback processes as in deadlock detection
- It is less restrictive that deadlock prevention

**Deadlock Avoidance Restrictions**

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and have 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 restricting processes
- Deadlock detection strategies do the opposite
  - Resource requests are granted whenever possible

## 3.2 Deadlock Detection Algorithms

**Deadlock Detection Algorithms**

- A check for deadlock can be as frequently as each resource request or less, depending upon likelihood of deadlock
- Advantages
  - It leads to early detection
  - The algorithm is relatively simple
- Disadvantage
  - Frequent checks consume LOTS of processor time

**Recovery Strategies**

- Abort all deadlocked processes
- Rollback each deadlock process to some previously defined checkpoint and restart all processes
- Kill deadlocked processes until deadlock is, well, dead
- Preempt resources until deadlock is, well, dead

# 4 Key Points

**Key Points**

- What is a deadlock?
- What are four necessary conditions for deadlock?
- How to address deadlocks: prevent, avoid, or detect