In This Module...

- Deadlock: Definition and Examples
- Deadlock: Models
- Deadlock: Algorithms
A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause.
Deadlock Characteristics

- Deadlocks occur when multiple processes are attempting to make exclusive use of multiple resources.
- The resources that are the source of deadlock conflicts may be hardware resources, software resources, or both.
- Example: Process A gets exclusive access to resource R, B gets access to S. Then A requests S and B requests R...
  DEADLOCK!
We categorize resources involved in deadlocks in several ways. First, resources may be consumable or non-consumable. For example...

- Primary memory is non-consumable (assuming the system keeps track of it properly), since after its use by a process it is available for use by other processes.

- Messages from another process are consumable resources. When they arrive and are processed they disappear.
Resources may also be preemptable or non-preemptible. By preemption, we mean a resource can be taken away from a process for a while to satisfy the needs of another process. Later the resource can be returned to the process from which it was preempted. For example

- Primary memory is a preemptable resource in virtual memory systems or systems using two-level scheduling. It can have its contents saved on secondary storage and later restored.
- A tape drive isn’t normally considered to be a preemptable resource, since it would be very costly to share it among several processes.
Type Use of Non-Preemptible Resources

1. A process first requests exclusive use of the resource. If it is available, the process continues. Otherwise, the process blocks until it is available. If multiple units of the resource are available, any one of these is usually acceptable.

2. The process then uses the resource for an arbitrary time.

3. Finally, the process releases the resource, allowing a process waiting on the resource to continue.
Outcomes for Resource Allocation Requests

A resource allocation request may have several possible outcomes:

- The request may be successful, and the process can continue running.
- The resource may be unavailable, and the process will block.
- The resource may be unavailable, and the process receives an error indication.

This last outcome is usually possible only if a process has indicated it does not wish to block if the resource is unavailable.
Let's consider the Win32 system call **WaitForSingleObject**. This call has two parameters:

- A handle to the object (a semaphore, for example).
- A timeout parameter.

If the resource is unavailable, then if

- **timeout** is infinite, the calling thread will block.
- **timeout** = 0, the calling thread will return immediately with an appropriate error indication.
- **timeout** ≠ 0 the thread will wait at most **timeout** milliseconds for the resource to become available.
How Are Resources Requested?

- Resources can be requested explicitly or implicitly. Here are a few UNIX examples.
  - The open system call explicitly requests the use of a file resource.
  - Similarly, the fcntl system call may explicitly request a lock on all or part of a file resource.
  - The fork and exec system calls implicitly request a variety of resources, primarily including the memory required to execute a program.
Coffman, in 1971, identified four necessary and sufficient conditions for there to be a deadlock condition. These conditions are as follows:

- **Mutual exclusion**: each resource is either assigned to a single process or is unavailable.
- **Hold and wait**: processes holding resources may request and wait for additional resources.
- **No preemption**: resources previously allocated cannot be forcibly removed from a process.
- **Circular wait**: there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
Resource Allocation Graphs are directed graphs that can be used to illustrate deadlock conditions.

A resource graph has the following properties:

- Resources are shown as nodes labeled with the resource name. We will use squares to identify resource nodes.
- Processes are labeled with the process identification. We will use circles to identify processes.
- A directed edge from a resource to a process shows the process “owns” the resource.
- An edge from a process to a resource shows the process is requesting use of the resource.
Process A Holds Resource R
Process B Requests Resource S
Deadlock!
Strategies for Dealing with Deadlock

- Ignore the problem completely.
- Detect and recover.
- Prevent deadlock.
- Avoid deadlock.
Ignoring Deadlock
Ignoring Deadlock

- Tanenbaum calls this the “ostrich algorithm.”
- This approach is appropriate if the cost of a deadlock is small compared to the cost of preventing or avoiding them.
- Many UNIX systems ignore deadlocks.
- Most mission-critical systems are designed with consideration for deadlock.
Detection requires that the system maintain a complete “database” of all resources and owner processes and all pending requests.

This database must be in a form usable for an algorithm that can detect cycles in graphs.

For systems with one unit of each resource type, a simple cycle-detection algorithm can be used.
Deadlock Recovery

Techniques for Recovery:

- Preempt resources - but this is not always possible.
- Kill a process, releasing its resources, and restart it from the last checkpoint. A *checkpoint* is a snapshot of the complete environment of a running process, including memory and register contents, input/output state, etc. Checkpoints should be taken at regular intervals, the interval size dependent on the cost of a process losing the result of a computation (for any reason).
- Kill a process which can be run again with no adverse side effects (or changes in the environment, including databases, files, etc.). A good example is a compilation, since we can always use the same source program as input to the compiler.
- Kill any process which breaks the deadlock. Obviously this means a process that has unsaved and potentially unreproducible results, or one that has changed databases or other files. This is the least desirable process to kill.
Deadlock Prevention

To prevent deadlock, we must impose restrictions on processes and resource usage so that deadlocks are structurally impossible.

Recall the four Coffman conditions that are necessary for deadlock:

- **Mutual exclusion**: each resource is either assigned to a single process or is unavailable.
- **Hold and wait**: processes holding resources may request and wait for additional resources.
- **No preemption**: resources previously allocated cannot be forcibly removed from a process.
- **Circular wait**: there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Ensuring that at least one of the conditions will **not** be satisfied will guarantee that deadlock is impossible.

Note that different conditions can be denied for different resource types.
On the surface this doesn’t appear to be viable.

However some resources are never shared, but owned by a single process that then accepts requests for the use of that resource.

For example, recall the techniques used to share a printer among processes. There will never be deadlock involving the printer as a resource.
All processes would need to request all resources before they begin execution.

- Problem 1: this may not be possible, as some resource needs may be determined only at execution time. For example, consider the amount of memory needed by a recursive function. This is clearly dependent on the data. Allocation of an excessive amount of memory to handle the worst case is clearly inappropriate.

- Problem 2: this approach doesn’t lead to efficient use of resources.

- Alternate method: processes must give up resources if a new request would block.
Prevention: No Preemption

- Preempting the use of an assigned resource is either costly or impossible for some resources.

- Examples: Preempting a tape drive is not reasonable. Doing so would require that the position of the current tape be recorded, the tape removed and then replaced by a different tape. After that tape’s use is complete, the original tape would need to remounted and repositioned. Ugh!

- Some resources may also be left with inconsistent states if they are preempted at arbitrary times. Think about the updating of a relational database where multiple tables must be modified. If the preemption occurs before all tables have been consistently updated, then the state of the database is corrupted.
Prevention: No Circular Wait

- Assign each resource a unique numeric code.
- Processes must request resources in increasing order of their numeric codes.
- This technique has been used with success in some systems.
- It is due to Havender, "Standard Allocation Patterns".
Deadlock Avoidance

Q. How do you avoid having people step in your flower garden?

A1. Politely ask each person who might accidentally step on your flowers to please not do so.

A2. Surround your garden with an electrified 10 foot high fence topped by razor wire, 20 feet away from the garden’s boundary.

Q. Which technique do you think will work more effectively to protect your garden?

It is the second approach that is taken by deadlock avoidance: avoid getting into situations that might result in deadlock.
A resource trajectory is a plot of the execution path of processes through regions where they hold mutually-exclusive use of resources.

Displaying such a plot is difficult for more than two processes, but two processes will be sufficient to illustrate the technique we will consider for deadlock avoidance.

To show resource trajectories for two processes, draw a plot with one axis for each process – much like the x and y axes for a Cartesian coordinate system, but the axes correspond to processes.

Each axis has increasing process “progress” (i.e. time) extending upward and to the right of the origin. That is, what would normally be the x coordinate indicates how far the process represented by that axis has progressed in its computation (and similarly for the vertical axis).

In the appropriate regions on each axis draw orthogonal regions in which various resources are used in a mutually-exclusive manner.

Intersecting regions with the same unit of the same resource are forbidden regions, as they indicate that both processes would have mutually-exclusive use of the same resource, which is impossible.
Since progress in each process must be in one direction only, reaching a region in which any process progress is blocked by forbidden regions spells t-r-o-u-b-l-e!

Such regions are called **unsafe** regions.

Regions where at least one process can make progress are called **safe** regions (or states).

Note carefully that unsafe states are not deadlocked states. That is, processes can continue to execute as long as they do not encounter a forbidden region.

Unfortunately, unless we have a clairvoyant system, we cannot see the future, and do not know if there is a forbidden region in our path.

The only safe solution is to avoid any unsafe regions, thereby guaranteeing we will never encounter any forbidden regions.
Suppose we have two processes, named A and B.

We know each process may, at some time during their execution, use resources named R and S, either singly, or in combination.

The processes have executed for some time.

Process A has allocated resource R.

Process B has allocated resource S.

The path the two processes have taken is shown by the dashed blue line on the next slide, with the arrow pointing to their current position.

But we still don’t know the order in which the processes might proceed. In particular, we don’t know what they will do with respect to resource allocation and deallocation.
- Note the yellow region into which the execution of the processes has proceeded. This is **not** deadlock!
- But we don’t know what will happen next.
- If process A soon requests resource S, and process B soon requests resource R, we’ll encounter the situation shown on the next slide.
Resource Trajectories - 5

The diagram illustrates resource trajectories with dimensions S & R, S, and B. The region labeled "UNSAFE" indicates a conflict in resource allocation, marked by a dashed blue line starting from the origin and moving through the specified paths. The trajectories highlight the conditions leading to resource deadlock in the context of operating systems.
This is deadlock; the processes cannot continue execution. The regions colored red are impossible to enter.

But again, note that this is only one of the possible execution sequences that could have occurred.

Process B could have released its claim on resource S which would allow process A to continue execution.

But since we don’t know what will happen, and we do know what might happen could lead to deadlock, we label the yellow region “UNSAFE” and avoid entry into that region during execution.

This avoidance of unsafe regions, or unsafe states, gives rise to the classic algorithm we now consider for its implementation.
Dijkstra’s Banker’s Algorithm

- This deadlock avoidance algorithm is modeled on the actions of a banker making loans to customers.
- The banker has a fixed amount of money available to make loans.
- Each customer must indicate their “maximum claim” (that is, the maximum amount they want to borrow) in advance; this amount can never be exceeded, and can obviously never be larger than the total amount of money possessed by the banker.
- Customers always repay their loans at some point in the future. This may happen before or after they have obtained their maximum loan amount.
- Since the banker has a limit on the amount of money that can be loaned, customers may have to wait before obtaining their loans. But as long as their total loan amount is less than or equal to the total amount they indicated as their maximum claim, the loan will eventually be granted.

Q. How can the banker make safe loans?
The problem formulation we just considered has a single resource – money. But there are multiple units of the resource.

In the algorithm implementation, we assume each process states, in advance of execution, the maximum number of units of the resource it will require.

We'll construct a matrix with one row for each process $i$ showing the number of units allocated to the process, $A_i$, and the maximum claim for the process $M_i$. We'll also include a flag to mark certain processes during execution of the algorithm.

We will also keep track of the total amount of money the banker/system has available to make loans, $c$.

Initially, the banker/system has all the resources, and no allocations have been made to the processes. That is, each $A_i$ is 0.
The Banker’s Algorithm: Allocation

Process $P_j$ asks for $k$ additional units of the resource, and the system has $c$ units available.

1. If $k > c$ go to step 7.

2. Set $c' = c$. Clear the marks on all processes. Allocate $k$ units of the resource to $P_j$: set $A_j = A_j + k$ and $c = c - k$.

3. Search for an unmarked, unblocked process $P_i$ with $M_i - A_i \leq c$. That is, search for a process that could receive its maximum claim.

4. If such a process is found, mark it, set $c = c + A_i$, and repeat from step 3. (Simulate process termination.)

5. If all non-blocked processes are marked the allocation is safe. Set $c = c' - k$, then return, allowing process $P_j$ to continue execution.

6. If any non-blocked processes is not marked, then the proposed allocation is not safe. Set $A_j = A_j - k$ and $c = c'$.

7. Move $P_j$ to the blocked state and indicate it’s waiting on $k$ units of the resource. Return and continue execution with a different process.
Process $P_j$ returns $k$ units of the resource.

1. Set $c = c + k$ and $A_j = A_j - k$.

2. For each process that is blocked waiting on the resource, repeat its previous allocation request. If the allocation is successful, move the process to the ready state.
A Safe State (10 Units Available)

<table>
<thead>
<tr>
<th>Who</th>
<th>Allocated</th>
<th>Max. Claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Barbara</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Bill</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Paula</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

With 10 available units, anyone can ask for and receive their maximum claim, after which they will be required to return it. All other users can then obtain their maximum claims and return them.
Another Safe State (2 Units Available)

<table>
<thead>
<tr>
<th>Who</th>
<th>Allocated</th>
<th>Max. Claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Barbara</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Bill</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Paula</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Bill can get 2 units and then repay his total of 4 units. Then either Barbara or Paula could get their maximum claim and repay. Finally, Andy could get his maximum claim. Since all users can obtain their maximum claims, this is a safe state.
An Unsafe State (1 Unit Available)

<table>
<thead>
<tr>
<th>Who</th>
<th>Allocated</th>
<th>Max. Claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Barbara</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Bill</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Paula</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

With this allocation of resources, there is no process is that is guaranteed to be able to obtain its maximum claim. Of course, we don’t know for sure that any process will make such a claim. Nevertheless, this is an unsafe state, and the Banker’s Algorithm would identify it as such.
Generalizing the single resource type Banker’s Algorithm is easy.

Instead of having $c$ (resources in the bank) and elements of the $A$ and $M$ array (per process allocations and maximum claims) be integers, we replace them with vectors (one-dimensional arrays) with one entry for each resource type.

Now a request by a process specifies a vector indicating the number of units of each resource type it wants to allocate or deallocate.
Disadvantages of the Banker’s Algorithm

Although the Banker’s Algorithm does identify unsafe states, it is not generally used in real systems for a variety of reasons:

- It requires that each process specify, before execution begins, the maximum number of units of each resource it might need. This information might not be available when the process begins.
- Additionally, specifying too large a value for the maximum number of units required may lead to states being classified as unsafe when, in practice, they wouldn’t be.
- The number of resources in modern systems is enormously large. Physical resources, of course, can be easily enumerated, but logical/virtual resources, like locks on data structures or files, may be dynamically created, and thus enumeration of these would be very difficult.
- Even ignoring the problems related to enumerating all resources and identifying maximum requirements for them, the Banker’s Algorithm still needs to be executed for every allocation request or return. This will be very expensive.
- Of course, individual applications could use the Banker’s Algorithm to avoid deadlock among their own processes.
Two-Phase Locking

- Sadly, deadlock prevention and deadlock avoidance are often too “heavy handed” for real systems. But specialized algorithms are available for many specific cases.
- For example, two-phase locking is often used with database systems with multiple concurrent users.
- Multiple locks, both read locks and write locks, can be applied to data (for example, tables in a relational database) prior to carrying out a transaction.
- In the first phase of locking, a process attempts to obtain locks on the individual resources that are required. As long as the locks are successfully obtained, the locking process continues.
- If the required locks are obtained, then the transaction is performed, and finally the locks are released (the second phase), completing the operation.
- If all of the required locks cannot be obtained, all of the locks that were obtained are released, and the first phase is tried again.
Two-Phase Locking Variations and Inadequacies

- In many implementations, there may be a random delay inserted between the lock release and the restarting of the first phase. The purpose of this delay is similar to the delay used in the Ethernet collision exponential backoff algorithm.

- Although the two-phase locking algorithm works well for databases, it is not suitable for all situations that may lead to deadlock.

- It requires that the process be able to anticipate all needed resources and lock them all before beginning work. This is not possible in all situations.

- In particular, in real-time systems and process control systems a process (or task) cannot be terminated partway through because a resource is not available.

- Likewise, operations that cannot be “undone” can’t be performed before the needed resources are locked.
In the next module we’ll begin our study of input and output.

We will examine input/output hardware characteristics, and look at a few devices in some detail.

We will then study the way input/output devices are managed by the operating system and are made available to application programs.