Deadlocks

OPERATING SYSTEMS

Sercan Külcü | Operating Systems | 16.04.2023

Contents

Chapter 7: Deadlocks

1 Introduction

Welcome to this chapter on deadlocks in operating systems! In this chapter, we will discuss the concept of deadlocks and their importance in the context of operating systems.

A deadlock is a situation in which two or more processes are unable to continue executing because each is waiting for one of the others to release a resource. In other words, a deadlock occurs when two or more processes are stuck in a circular wait for resources, and none of them can proceed until the others release the resources they are waiting for.

Understanding and preventing deadlocks is essential in operating systems because deadlocks can cause system failure, which can be costly in terms of time, money, and resources. Thus, this chapter will focus on the different causes of deadlocks and the strategies for preventing them.

In summary, the goals of this chapter are to define deadlocks, explain why understanding and preventing deadlocks are crucial in operating systems, and provide an overview of the strategies for preventing deadlocks.

1.1 Definition of deadlocks

In the context of operating systems, deadlocks refer to a situation where a set of processes are blocked and unable to proceed, as they are waiting for resources that are held by other processes in the set. Deadlocks can have a significant impact on the performance and reliability of operating systems and can result in significant loss of time, resources, and even data. Therefore, it is important to understand the concept of deadlocks, their causes, and the various methods used to prevent, detect, and resolve them.

A deadlock is a situation where a set of processes is blocked and unable to proceed, as they are waiting for resources that are held by other processes in the set. In other words, each process is waiting for a resource that is currently held by another process in the set, and hence none of the processes can proceed. Deadlocks can occur when a set of processes compete for a finite set of resources, and each process requires a resource that is held by another process.

There are four necessary conditions that must be present for a deadlock to occur:

- Resource types and allocation policies: The system must have a finite number of resources that are divided into several types, and the allocation of these resources must follow a certain policy.
- Hold and wait: A process must hold at least one resource and be waiting for additional resources that are currently held by other processes.
- No preemption: Resources cannot be preempted, i.e., they cannot be forcibly removed from a process that is holding them.
- Circular wait: A set of processes must be waiting for resources in a circular chain, where each process is waiting for a resource that is held by the next process in the chain.

Deadlocks can have a significant impact on the performance and reliability of operating systems. Therefore, it is important to understand the concept of deadlocks, their causes, and the various methods used to prevent, detect, and resolve them. In the next chapter, we will discuss the various methods used to prevent, detect, and resolve deadlocks in operating systems.

1.2 Resources

Resources are an important part of any operating system. They are objects to which some process has been granted exclusive access. These resources can take many forms, including hardware devices like printers, scanners, and Blu-ray drives, as well as software objects like data records, files, and other system resources.

Resources can be categorized in several ways. For example, they can be classified as sharable or non-sharable resources. A sharable resource is one that can be used by multiple processes at the same time. For example, a printer can be used by multiple users simultaneously. On the other hand, a non-sharable resource is one that can be used by only one process at a time. For example, a Blu-ray drive can be used by only one process at a time.

Another way to categorize resources is by their availability. Some resources are always available, while others are created dynamically when requested by a process. For example, a printer is always available, while a database record is created dynamically when requested by a process.

A computer system can have many different types of resources, and managing them can be a complex task. To avoid deadlocks, the operating system must carefully manage the allocation and release of resources.

When multiple processes compete for resources, deadlocks can occur. A deadlock is a situation in which two or more processes are waiting indefinitely for each other to release resources. To avoid deadlocks, the operating system must ensure that resources are allocated and released in a way that does not allow a circular wait to occur.

1.2.1 Preemptable resources

In operating systems, resources can be classified as preemptable or non-preemptable. Preemptable resources are those that can be taken away from the process owning them without any negative impact. This is in contrast to non-preemptable resources, which cannot be taken away from the process owning them without causing serious problems.

Memory is a good example of a preemptable resource. A process may be using a certain amount of memory, but if it is preempted, its memory can be swapped out to disk without causing any ill effects. When the process resumes execution, its memory can be swapped back in from disk, and the process can continue where it left off.

Let's consider an example to illustrate this concept further. Imagine a system with 1 GB of user memory, one printer, and two 1-GB processes that each want to print something. Process A requests and obtains the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk.

While Process A is swapped out, Process B requests and obtains the printer. Since the printer is a non-preemptable resource, Process A cannot proceed until it gets the printer back. However, since memory is a preemptable resource, its memory can be swapped out to disk without any ill effects. When Process A is ready to continue, its memory can be swapped back in from disk and it can continue with the printing process.

Preemptable resources are important in operating systems because they allow for efficient use of system resources. By preempting processes that are not currently using their resources, the system can free up those resources for other processes to use. This can lead to better overall system performance and more efficient use of system resources.

In conclusion, preemptable resources are resources that can be taken away from the process owning them without causing any negative impact. Memory is a common example of a preemptable resource in operating systems, and their use is important for efficient use of system resources.

1.2.2 Nonpreemptable resources

In contrast to preemptable resources, nonpreemptable resources are those that cannot be taken away from their current owner without potentially causing failure. A common example of a nonpreemptable resource is a Blu-ray recorder. If a process is in the middle of burning a Blu-ray and the recorder is suddenly taken away and given to another process, the result will likely be a garbled or unusable disc. Nonpreemptable resources must therefore be carefully managed by the operating system to ensure that they are only granted to processes that can safely and reliably complete their tasks.

Managing nonpreemptable resources involves allocating and deallocating them in a way that minimizes the risk of contention and deadlock. This typically requires the use of more sophisticated synchronization mechanisms than those used for preemptable resources. For example, one approach to managing nonpreemptable resources is to use a token-based protocol, in which processes must acquire a token before they can access the resource. The token is passed from process to process, ensuring that only one process at a time has access to the resource. This approach can be effective but can also lead to contention and deadlock if the token becomes unavailable.

Another approach to managing nonpreemptable resources is to use priority-based scheduling. In this approach, processes are assigned priorities based on their resource needs and the urgency of their tasks. Processes with higher priorities are given access to the resource before processes with lower priorities. This approach can be effective but can also lead to starvation if lower-priority processes are never given access to the resource.

Overall, managing nonpreemptable resources is a critical aspect of operating system design and requires careful consideration of the tradeoffs between performance, reliability, and fairness. By carefully managing these resources, operating systems can ensure that processes can complete their tasks without the risk of failure or data corruption.

1.3 Resource acquisition

Resource acquisition is a crucial aspect of managing shared resources in an operating system. Different resources have different requirements, and the way that they are acquired and released must be carefully managed to ensure that no process is left waiting indefinitely for a resource to become available.

For some types of resources, such as records in a database system, it is up to the user processes to manage the usage of resources themselves. In these cases, semaphores or mutexes can be used to control access to the resource.

Each resource is associated with a semaphore or mutex, which is initialized to 1. When a process wants to use the resource, it must first acquire the semaphore or mutex by performing a "down" operation on it. This effectively decrements the semaphore or mutex count and waits until it becomes available if it is currently held by another process. Once the semaphore or mutex is acquired, the process can use the resource.

When the process is finished with the resource, it must release it by performing an "up" operation on the semaphore or mutex. This effectively increments the count and makes the resource available for another process to use.

The use of semaphores and mutexes to manage resource acquisition ensures that multiple processes can access the resource without conflicting with one another. However, it is important to note that this approach can lead to deadlock if processes are not careful to release the resource when they are finished using it. In addition, it can be difficult to manage the acquisition and release of resources in a large and complex system with many different types of resources.

Overall, resource acquisition is a critical aspect of operating system design, and different types of resources require different approaches to ensure efficient and reliable management. By carefully managing resource acquisition, an operating system can ensure that processes can access the resources they need without encountering conflicts or delays.

1.4 Importance of understanding and preventing deadlocks

Deadlocks are one of the most significant problems in operating systems that can lead to system crashes, data loss, and user frustration. As a result, it is crucial to understand and prevent deadlocks in operating systems. In this chapter, we will discuss the importance of understanding and preventing deadlocks in operating systems.

Importance of understanding deadlocks:

- Prevent system crashes: Deadlocks can cause the entire system to crash. Understanding how deadlocks occur can help prevent these crashes and ensure the stability of the operating system.
- Improve system performance: Deadlocks can cause resource contention and delays, resulting in reduced system performance. By understanding how deadlocks occur, we can design systems to avoid them, which can improve system performance.
- Ensure data integrity: Deadlocks can cause data loss or corruption. Understanding how deadlocks occur can help prevent these problems, ensuring the integrity of the data stored in the operating system.

Importance of preventing deadlocks:

- Reduce system downtime: Deadlocks can cause system downtime, which can be costly for businesses. Preventing deadlocks can help reduce system downtime, leading to improved productivity and reduced costs.
- Improve user experience: Deadlocks can cause programs to freeze, leading to a poor user experience. Preventing deadlocks can help ensure that programs run smoothly, providing a better user experience.
- Ensure system reliability: Deadlocks can cause system failures, leading to data loss and other problems. Preventing deadlocks can help ensure system reliability and reduce the risk of data loss or other problems.

In conclusion, understanding and preventing deadlocks is critical for ensuring the stability, performance, and reliability of operating systems. By taking measures to prevent deadlocks, we can reduce system downtime, improve user experience, and ensure data integrity and system reliability.

1.5 Overview of the goals of the chapter

Deadlocks occur when two or more processes are waiting for a resource that is held by another process, and none of the processes can proceed until the resource is released. This results in a circular waiting pattern, which can lead to a system deadlock. Understanding the causes and effects of deadlocks is crucial in ensuring that systems remain reliable and efficient.

The goals of this chapter are to provide a comprehensive overview of deadlocks, including their definition, causes, prevention techniques, and resolution methods. We will examine the necessary conditions for a deadlock to occur and the implications of deadlocks in an operating system. Additionally, we will look at how to detect and avoid deadlocks, along with the advantages and disadvantages of different approaches.

We will also discuss the impact of deadlocks on system performance and reliability.

By the end of this chapter, readers will have a clear understanding of what deadlocks are, the conditions that cause them, and the methods used to prevent and resolve them. This knowledge will help system administrators and developers identify and mitigate potential deadlocks in their systems, leading to improved system reliability and performance.

2 Necessary Conditions for Deadlocks

Welcome to the chapter on "Necessary Conditions for Deadlocks". In this chapter, we will be discussing the necessary conditions that can lead to deadlocks in an operating system. Deadlocks are one of the most critical problems in operating systems and can cause system crashes, data loss, and other serious issues. Therefore, understanding the necessary conditions for deadlocks is essential for any operating system developer.

We will start by defining what deadlocks are and why it is essential to prevent them. Then, we will discuss the four necessary conditions that can lead to deadlocks, which include resource types and allocation policies, hold and wait, no preemption, and circular wait. By understanding these conditions, you can identify and prevent deadlocks in your operating system.

So, let's dive into the chapter and explore the necessary conditions for deadlocks!

2.1 Resource types and allocation policies

In order to understand deadlocks, it is important to understand the types of resources that can be involved in a deadlock situation.

Resources can be classified as either reusable or consumable. Reusable resources, such as printers or communication channels, can be used by multiple processes at the same time. Consumable resources, such as memory or CPU time, are used up as processes run and cannot be shared.

Resource allocation policies determine how resources are allocated to processes. In a system where resources are allocated on a first-come, first-served basis, processes may end up holding resources for longer than necessary, leading to potential deadlock situations.

2.2 Mutual exclusion

Mutual exclusion is a fundamental requirement for concurrent systems that deal with shared resources. It refers to the idea that only one process can access a shared resource at any given time, to prevent conflicting updates and ensure data consistency. The mutual exclusion condition is a key concept in operating systems, as it ensures that only one process can hold a specific resource at any given time, thereby preventing multiple processes from accessing and modifying the same resource simultaneously.

The mutual exclusion condition can be expressed as follows: each resource is either currently assigned to exactly one process, or it is available. This means that a process must first request a resource before it can use it, and only one process can be granted access to the resource at a time. Once the process has finished using the resource, it must release it back to the system so that other processes can use it.

In order to implement mutual exclusion, operating systems typically provide synchronization mechanisms such as semaphores, mutexes, and monitors. These mechanisms ensure that only one process can acquire a lock on a resource at any given time, effectively enforcing mutual exclusion.

The mutual exclusion condition is crucial in preventing race conditions, which can occur when two or more processes access the same shared resource simultaneously, leading to inconsistent results. By enforcing mutual exclusion, operating systems can ensure that each process has exclusive access to a resource when it needs it, preventing conflicts and ensuring data consistency.

2.3 Hold and wait

In addition to the mutual exclusion condition, the hold and wait condition is another necessary condition for a deadlock to occur. This condition can arise when a process is holding onto a resource while also waiting for another resource that is currently being held by another process.

To illustrate this, let's consider a scenario where two processes, A and B, each need two resources to complete their tasks. Process A currently holds resource 1 but needs resource 2, while process B currently holds resource 2 but needs resource 1. If both processes are allowed to hold onto their currently held resources and wait for the other process to release the needed resource, then a deadlock will occur.

The hold and wait condition can be avoided by requiring a process to request and acquire all necessary resources before beginning execution. This can be achieved through various methods such as the banker's algorithm or using non-preemptive resources.

Alternatively, processes can be allowed to release their currently held resources and then reacquire them in a predetermined order. This approach can help avoid deadlocks by preventing a process from holding onto resources while waiting for another resource to become available.

Overall, the hold and wait condition highlights the importance of carefully managing resource allocation to prevent situations where processes are waiting for resources to become available while holding onto resources themselves. By proactively addressing this condition, we can help reduce the likelihood of deadlocks occurring in a system.

2.4 No preemption

The no preemption condition states that resources cannot be taken away from a process without that process voluntarily releasing them. This means that if a process is holding onto a resource and not releasing it, other processes cannot preempt that resource. For example, if process A is holding resource R1 and process B needs resource R1 to complete its execution, process B cannot forcibly take resource R1 from process A. Process A must voluntarily release resource R1 before process B can acquire it.

The no preemption condition is necessary for a deadlock to occur because it prevents the system from resolving a deadlock by forcibly taking resources away from processes. If the system were allowed to forcibly take resources away from processes, it would be possible to break a deadlock by forcibly taking a resource from one process and giving it to another process. However, if the no preemption condition is in effect, the system cannot break a deadlock by forcibly taking resources away from processes.

To illustrate the no preemption condition, consider a system with two processes, P1 and P2, and two resources, R1 and R2. Suppose that process P1 is holding onto resource R1 and waiting for resource R2, which is being held by process P2. Similarly, process P2 is holding onto resource R2 and waiting for resource R1, which is being held by process P1. This situation creates a deadlock, as neither process can proceed without releasing a resource that it is holding.

Now, suppose that the system is allowed to preempt resources from processes. In this case, the system could forcibly take resource R1 from process P1 and give it to process P2, and forcibly take resource R2 from process P2 and give it to process P1. This would resolve the deadlock, as both processes would now have the resources they need to proceed. However, if the no preemption condition is in effect, the system cannot forcibly take resources from processes, and the deadlock remains.

2.5 Circular wait

The circular wait condition is one of the necessary conditions for a deadlock to occur. In a system with multiple processes and resources, circular wait happens when a set of processes are waiting for resources that are held by other processes in a circular chain.

For example, process A is waiting for a resource held by process B, process B is waiting for a resource held by process C, and process C is waiting for a resource held by process A. This creates a circular chain of processes waiting for resources, which prevents any of them from proceeding and leads to a deadlock.

Deadlocks can be avoided by breaking the circular wait condition. One way to do this is to impose an ordering on the resources, such that all processes request resources in the same order. This eliminates the possibility of circular wait, as resources are requested in a specific order, and each process can acquire the resources it needs without having to wait for another process.

Another way to avoid deadlocks is through resource allocation. The operating system can use algorithms like Banker's algorithm, which determines if a requested resource allocation will result in a safe state or not. If the allocation will not result in a safe state, the request is denied, and the process must wait for the requested resource.

Overall, understanding these necessary conditions for deadlocks is crucial for designing and implementing operating systems that are robust and reliable. In the following sections, we will explore methods for detecting, preventing, and resolving deadlocks.

3 Deadlock modelling

3.1 Resource allocation graph

Resource Allocation Graph (RAG) is a graphical representation of resources that are being used by a set of processes. In deadlocks, a RAG is used to represent the allocation and request of resources by different processes. The RAG shows the relationships between resources and processes, and helps to determine whether a deadlock exists.

In a RAG, processes are represented by circles and resources by rectangles. The circles are connected to the rectangles by arrows, which represent the allocation of resources from the resource to the process. Additionally, the rectangles can be connected to each other by another set of arrows, which represent the requests of resources from one resource to another.

A RAG can be used to detect whether a deadlock exists in the system. A deadlock is said to occur if and only if there exists a cycle in the graph. This cycle represents a circular wait, which is one of the necessary conditions for a deadlock. If a cycle is detected, it means that there is at least one process that is holding a resource and is waiting for another resource that is being held by a different process.

Moreover, a RAG can also be used to resolve deadlocks. If a cycle is detected in the RAG, the resources involved in the cycle can be examined to determine which resource to preempt, if any. The preempted resource can then be allocated to the process that is waiting for it, and the cycle can be broken.

Example: Here's an example pseudocode for constructing a resource allocation graph:

```
initialize graph G = (V, E)
initialize set of processes P = {P1, P2, ..., Pn}
initialize set of resources R = {R1, R2, ..., Rm}


for each process Pi in P:
   add node Pi to V
for each resource Rj in R:
   add node Rj to V


for each resource allocation edge (Pi, Rj) in E:
   add edge (Pi, Rj) to G


for each request edge (Rj, Pi) in E:
   add edge (Rj, Pi) to G
```

In this pseudocode, we first initialize an empty graph G, as well as the sets of processes P and resources R. We then add nodes to G for each process and resource.

Next, we iterate through the set of edges E and add an edge to G for each resource allocation or request. An edge from a process node Pi to a resource node Rj represents an allocation of Rj to Pi, while an edge from a resource node Rj to a process node Pi represents a request from Pi for Rj. This algorithm can be used to construct a resource allocation graph, which can then be used to detect and prevent deadlocks in a system.

In conclusion, the Resource Allocation Graph is a useful tool for detecting and resolving deadlocks in operating systems. It provides a clear visualization of the relationships between processes and resources, making it easier to understand and identify potential deadlocks.

3.2 Dealing with deadlocks

Deadlocks are a challenging problem for operating systems, as they can bring an entire system to a standstill. Fortunately, there are several strategies that can be used to deal with deadlocks.

The first strategy is simply to ignore the problem and hope it goes away. While this approach might work in some cases, it is not a reliable solution and can lead to long periods of system inactivity.

The second strategy is detection and recovery. This approach involves allowing deadlocks to occur, detecting when they happen, and taking action to recover from them. This can be done by periodically checking the system for deadlocks and releasing resources as necessary to break the deadlock. While this approach can be effective, it can also be costly in terms of system resources and may not be feasible in all situations.

The third strategy is dynamic avoidance through careful resource allocation. This involves monitoring resource usage in real-time and carefully allocating resources to prevent deadlocks from occurring. This approach requires sophisticated algorithms and can be difficult to implement in practice.

The final strategy is prevention by structurally negating one of the four conditions necessary for a deadlock to occur. For example, by ensuring that resources are only ever allocated to a single process at a time, the hold and wait condition can be negated. This approach requires careful design and can be challenging to implement, but is often the most effective way to prevent deadlocks from occurring.

Overall, there is no single best approach for dealing with deadlocks, and the appropriate strategy will depend on the specific circumstances of the system in question. Nevertheless, by carefully monitoring resource usage and adopting appropriate prevention and recovery strategies, it is possible to minimize the impact of deadlocks and ensure that systems remain available and responsive even in the face of these challenging situations.

3.3 The ostrich algorithm

The ostrich algorithm is a simple approach to dealing with deadlocks that involves ignoring the problem and pretending it doesn't exist. This approach is often viewed as unacceptable by mathematicians, who believe that deadlocks must be prevented at all costs. However, engineers tend to take a more practical approach and consider the frequency and severity of the problem.

When deciding whether to use the ostrich algorithm, engineers may ask questions such as: How often do deadlocks occur? How often does the system crash for other reasons? And how serious are the consequences of a deadlock? If deadlocks occur infrequently compared to other system failures, engineers may not be willing to pay a large performance or convenience penalty to prevent them.

While the ostrich algorithm may seem like a tempting solution, it is not a sustainable approach to handling deadlocks in a production system. Ignoring the problem can lead to system instability, and in the worst-case scenario, it can result in catastrophic failures that can bring down the entire system. As such, it is generally recommended to use one of the other strategies for dealing with deadlocks, such as detection and recovery, dynamic avoidance, or prevention.

3.4 Communication deadlocks

Communication deadlocks occur when processes are waiting for each other to communicate and exchange data or messages. This type of deadlock can occur in a distributed system or in a system with multiple processes communicating through message passing. When two processes are waiting for each other to send or receive a message, they can deadlock if neither process releases the resources it is holding until it receives the message it is waiting for.

One example of a communication deadlock is a message buffer deadlock, where two processes are waiting for each other to release a message buffer. For example, Process A may be waiting for Process B to release a buffer so that it can send a message, while Process B may be waiting for Process A to release a buffer so that it can receive a message. This situation can result in a deadlock if neither process releases the buffer it is holding.

Another example of a communication deadlock is a resource deadlock caused by a message. For example, a process may be waiting for a message from another process before it can release a resource, while the other process may be waiting for the same resource before it can send the message. This situation can also result in a deadlock if neither process releases the resource it is holding.

To prevent communication deadlocks, it is important to design protocols that allow processes to release resources in a timely manner. One approach is to use timeouts, which allow processes to release resources if they do not receive a message within a certain period of time. Another approach is to use a centralized scheduler or a distributed algorithm to coordinate the exchange of messages and ensure that resources are released in a way that prevents deadlocks.

Overall, communication deadlocks can be just as problematic as resource deadlocks, and it is important for operating system designers to be aware of the potential for both types of deadlocks in their systems. By designing protocols that allow processes to release resources in a timely manner and by coordinating the exchange of messages, operating system designers can prevent communication deadlocks and ensure the smooth operation of their systems.

4 Deadlock detection

4.1 Deadlock detection with one resource of each type

Deadlocks can be a real challenge for operating systems, but there are various methods to detect and resolve them. In this chapter, we will discuss one method for detecting deadlocks when there is only one resource of each type.

To begin with, we can create a resource graph that shows the relationships between the resources and the processes that use them. The graph will contain nodes representing the processes and the resources, with directed edges connecting a process to the resource it is currently holding and the resource to the process that is waiting for it.

In this system with only one resource of each type, we can assume that a process can hold onto at most one resource at any given time. Therefore, the graph will not contain multiple edges connecting a process to different resources of the same type.

If this resource graph contains one or more cycles, a deadlock exists. A cycle represents a situation where a process is waiting for a resource that is held by another process, which in turn is waiting for a resource held by a third process, and so on, until the cycle completes. Any process that is part of a cycle is deadlocked and cannot make any progress until the deadlock is resolved.

On the other hand, if there are no cycles in the resource graph, the system is not deadlocked. The processes are all able to acquire the resources they need to complete their tasks without waiting indefinitely for other processes to release their resources.

Using this method to detect deadlocks in a system with one resource of each type is relatively simple and straightforward. However, this approach becomes more complicated when there are multiple resources of the same type or when processes can hold multiple resources at the same time. In the next chapter, we will discuss a more complex method for detecting deadlocks in these types of systems.

In order to detect deadlocks in a system where there is only one resource of each type, we can use an algorithm known as the cycle detection algorithm. The algorithm works by constructing a resource graph that shows the relationships between processes and resources. If this graph contains one or more cycles, then a deadlock exists in the system.

The cycle detection algorithm operates by performing a series of steps for each node in the graph. First, we initialize an empty list L and mark all arcs as unmarked. Then, we add the current node to the end of L and check to see if it appears in the list two times. If it does, then the graph contains a cycle, and the algorithm terminates.

If there are any unmarked outgoing arcs from the current node, we pick one at random and mark it. We then follow it to the new current node and go back to step 3. If there are no unmarked outgoing arcs, we have reached a dead end. We remove this node and go back to the previous node, make that one the current node, and go to step 3.

If we reach the initial node again and there are no cycles, then the algorithm terminates. By following these steps, we can detect deadlocks in a system with one resource of each type.

It's worth noting that this algorithm assumes that there is only one resource of each type in the system. If there are multiple resources of a particular type, a different algorithm would be needed to detect deadlocks. However, the cycle detection algorithm is a useful starting point for understanding the principles of deadlock detection in operating systems.

4.2 Deadlock detection with multiple resources of each type

In more complex systems, with multiple copies of some of the resources, a different algorithm is required to detect deadlocks. In this case, a matrix-based approach can be used to detect deadlock among n processes, P_1 through P_n.

Let the number of resource classes be m, with E_1 resources of class 1, E_2 resources of class 2, and so on, with E_i resources of class i ($1 \leq i \leq m$). The existing resource vector, E, gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.

At any instant, some of the resources are assigned and are not available. The available resource vector, A, indicates the number of instances of each resource that are currently available and unassigned. For example, if both tape drives are assigned, A_1 will be 0.

To detect deadlocks, two arrays are used: C, the current allocation matrix, and R, the request matrix. The current allocation matrix, C, has n rows and m columns, with each element C_{ij} indicating the number of resources of type j currently allocated to process P_i. The request matrix, R, has the same dimensions as C and indicates the number of resources of each type that each process is requesting.

The detection algorithm checks whether there is a safe sequence of processes that can complete their execution. A safe sequence is a sequence of processes such that for each process Pi, the resources it needs to complete its execution are available either currently or after completing the execution of some other process Pj. If a safe sequence exists, then there is no deadlock.

The detection algorithm works by initializing the work vector, W, to the available resource vector, A. Then, for each process Pi, the algorithm checks whether Pi can complete its execution by comparing the number of resources it needs, as specified in the request matrix, R, with the number of resources currently available, as specified in the work vector, W. If Pi can complete its execution, then it releases its resources to the system and adds them to the work vector. The algorithm continues this process until all processes can complete their execution or no process can be completed.

If there is no safe sequence of processes, then a deadlock exists. In this case, the algorithm identifies the deadlocked processes by constructing a graph, with one node for each process that is deadlocked, and one edge for each resource that the process is waiting for. The graph is then searched for a cycle, and if one is found, the deadlocked processes are identified.

In summary, the matrix-based algorithm for deadlock detection with multiple resources of each type involves using two arrays, C and R, to represent the current allocation and resource request matrices. The algorithm works by checking for the existence of a safe sequence of processes and identifying deadlocked processes by constructing a graph and searching for a cycle.

In this section, we'll discuss the matrix-based algorithm used to detect deadlocks in a system with n processes and m resource classes. As mentioned earlier, let's assume there are E1 resources of class 1, E2 resources of class 2, and so on, up to Em resources of class m. Additionally, we have an available resource vector A, which specifies the number of instances of each resource that are currently available and unassigned. The current allocation matrix, C, shows the current allocation of resources to processes, while the request matrix, R, shows the additional resources that each process needs to complete its execution.

The deadlock detection algorithm can be described in the following steps:

1. Look for an unmarked process, Pi, for which the ith row of R is less than or equal to A.
2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

In step 1, we're trying to find a process whose resource requests can be satisfied with the available resources. If we find such a process, we add its allocated resources to the available resource vector and mark the process as having been examined. We then start over from step 1 and continue until all processes have been marked, or until there are no more unmarked processes that can be satisfied.

This algorithm is a form of a banker's algorithm that determines whether the system is in a safe state or not. The system is considered to be in a safe state if there exists a sequence of processes such that each process can acquire all the resources it needs before any of the other processes in the sequence request any resources.

In conclusion, this matrix-based algorithm provides an efficient way to detect deadlocks in a system with multiple resources of each type. By following these simple steps, we can identify which processes are deadlocked and take appropriate measures to resolve the issue.

5 Recovery from deadlock

We will delve into the methods of resolving deadlocks, including killing processes, resource preemption, and rollback and recovery. By the end of this chapter, you will have a comprehensive understanding of the causes and prevention of deadlocks and the techniques to resolve them.

5.1 Killing processes

In some cases, it may be necessary to kill one or more processes to break a deadlock. This is often seen as a last resort, as it can result in loss of data or incomplete transactions. In this chapter, we will discuss the process of killing processes as a method of resolving deadlocks.

When a deadlock is detected, the operating system may choose to kill one or more of the processes involved in the deadlock. This is done to free up resources that are being held by those processes and break the deadlock. The operating system must carefully select which process or processes to kill in order to minimize the impact on the system as a whole.

When choosing which process to kill, the operating system should consider several factors, such as the importance of the process, the amount of resources it is currently holding, and the potential impact on other processes. If the process being killed is part of a larger transaction or operation, the operating system should ensure that any necessary rollback or recovery procedures are carried out to minimize data loss.

Once the process or processes have been selected for termination, the operating system will send a signal to those processes to instruct them to terminate. The process being killed should release any resources it is currently holding to ensure that they can be used by other processes in the system.

Killing processes can have a significant impact on system performance, particularly if the process being killed is a critical system process. This can lead to system instability, crashes, or even data loss.

To minimize the impact of killing processes, the operating system should ensure that any necessary recovery procedures are carried out to restore the system to a stable state. In addition, the operating system should monitor system performance after the processes have been killed to ensure that there are no lingering issues that may impact system reliability.

Killing processes is a drastic measure that should only be taken as a last resort when other deadlock resolution methods have failed. The operating system must carefully consider the impact of killing processes on the system as a whole, and take steps to minimize any negative impact on system performance or reliability.

5.2 Resource preemption

In cases where deadlocks cannot be prevented or avoided, the next step is to resolve the deadlock. One method of resolving deadlocks is through resource preemption. Resource preemption is the act of forcibly removing resources from a process in order to free them up and allow other processes to proceed.

Resource preemption can be a useful strategy in resolving deadlocks. However, it can also be a complex and potentially dangerous technique, as forcibly removing resources from a process can result in data loss or corruption if not done carefully. In this chapter, we will explore the concept of resource preemption in deadlocks resolution.

Resource preemption involves the following basic principles:

- Priority: The resources being preempted must have a well-defined priority scheme. This priority scheme is used to determine which process should have its resources preempted in order to break the deadlock.

- Rollback: When a resource is preempted from a process, the process may need to be rolled back to a previous state in order to release the resource. This involves undoing any work that has been done since the resource was acquired.
- Avoidance of Starvation: Resource preemption must be done in a way that does not cause starvation of any process. This means that resources should be preempted in a fair and equitable way, and that all processes should have an equal chance of obtaining the resources they need.

There are several methods of resource preemption, including:

- Victim Selection: The first step in resource preemption is to select a process to be the victim. The victim is the process whose resources will be preempted. The selection process typically involves choosing the process with the lowest priority or the process that has been waiting the longest.
- Rollback: Once a victim has been selected, the process must be rolled back to a previous state in order to release the resources. This involves undoing any work that has been done since the resource was acquired.
- Re-allocation: Once the resources have been preempted, they must be allocated to another process. This may involve choosing a process from a waiting list or using a priority scheme to determine which process should receive the resources.
- Notification: Finally, the process that has had its resources preempted must be notified of the preemption. This allows the process to take any necessary action to recover from the preemption.

Resource preemption is a complex technique that should be used with caution. It is important to have a well-defined priority scheme in place, as well as a plan for rolling back processes and reallocating resources. Additionally, care should be taken to ensure that resource preemption does not cause starvation of any process.

Resource preemption is a powerful technique for resolving deadlocks. By forcibly removing resources from a process, resource preemption can break deadlocks and allow other processes to proceed. However, resource preemption must be used with care, as it can result in data loss or corruption if not done properly. By following the basic principles of resource preemption and using well-defined methods, deadlocks can be resolved safely and efficiently.

5.3 Rollback and recovery

In some cases, killing processes or resource preemption may not be feasible solutions for resolving deadlocks. Another approach is to use rollback and recovery techniques to undo the actions that led to the deadlock and restore the system to a consistent state. In this chapter, we will discuss the use of rollback and recovery techniques for resolving deadlocks in operating systems.

Rollback and recovery techniques involve undoing the actions that led to the deadlock and restoring the system to a consistent state. This can be done by rolling back transactions and re-executing them in a different order. Rollback and recovery techniques can be used to resolve deadlocks in systems where processes communicate with each other through transactions. A transaction is a sequence of operations that must be completed as a whole. In case of a deadlock, the transactions involved in the deadlock can be rolled back and re-executed in a different order to avoid the deadlock.

5.3.1 Two-Phase Commit Protocol

The two-phase commit protocol is a popular rollback and recovery technique used for resolving deadlocks in distributed systems. In this protocol, a coordinator is responsible for managing the transactions and ensuring that they are executed in a consistent manner. The protocol consists of two phases:

- Commit Request Phase: In this phase, the coordinator sends a message to all the participants asking them if they are ready to commit the transaction. If all the participants reply with a yes, the coordinator sends a message to all the participants asking them to commit the transaction.
- Commit Phase: In this phase, the participants commit the transaction and send a message to the coordinator confirming that the transaction has been committed.

If a participant does not reply to the commit request or replies with a no, the coordinator aborts the transaction and rolls it back. This ensures that the system is always in a consistent state and avoids deadlocks.

5.3.2 Checkpointing

Checkpointing is another technique used for rollback and recovery in operating systems. In this technique, the state of the system is saved at regular intervals in a checkpoint file. If a deadlock occurs, the system can be restored to a consistent state by rolling back to the last checkpoint and re-executing the transactions from that point.

Rollback and recovery techniques are often used in distributed systems where processes communicate with each other through transactions. They are effective in resolving deadlocks and ensuring that the system remains in a consistent state. However, they can be costly in terms of time and resources required for rollback and recovery. Compared to other techniques such as killing processes or resource preemption, rollback and recovery techniques are more complex and require more sophisticated algorithms to be implemented.

In conclusion, rollback and recovery techniques are an effective way of resolving deadlocks in operating systems. They involve undoing the actions that led to the deadlock and restoring the system to a consistent state. The two-phase commit protocol and checkpointing are popular rollback and recovery techniques used for resolving deadlocks in distributed systems. However, they can be costly in terms of time and resources required for rollback and recovery.

6 Deadlock Avoidance

Deadlock is a situation that occurs in an operating system when two or more processes are blocked, waiting for each other to release resources. It can cause a significant delay in system performance and result in a loss of data. In this chapter, we will discuss the concept of deadlock avoidance, which is an important aspect of operating system design.

Deadlock avoidance is the approach that an operating system uses to prevent deadlocks from occurring. In this chapter, we will explore the various techniques and algorithms that can be used to prevent deadlocks in a system. We will first discuss the concept of safe and unsafe states and how they relate to deadlock avoidance. Then we will delve into the Banker's algorithm for deadlock avoidance, which is widely used in operating systems.

Deadlock avoidance is a technique used to prevent the occurrence of deadlocks by ensuring that the system remains in a safe state. In this chapter, we will discuss the concept of safe and unsafe states in the context of deadlocks avoidance.

6.1 Safe State

A safe state is a system state in which all processes can complete their execution without leading to a deadlock. In other words, a safe state is a state in which the system can allocate resources to all the processes in some order and still avoid a deadlock. A system can be considered to be in a safe state if there is at least one sequence of resource allocations that can lead to a state where all processes have obtained their required resources and completed their execution.

6.2 Unsafe State

An unsafe state is a system state in which the system may or may not lead to a deadlock. In other words, an unsafe state is a state in which the system cannot allocate resources to all the processes in some order without leading to a deadlock. A system can be considered to be in an unsafe state if there is no sequence of resource allocations that can lead to a state where all processes have obtained their required resources and completed their execution.

6.3 Resource Allocation Graph

The resource allocation graph is a technique used to check if a system is in a safe or unsafe state. The resource allocation graph consists of two types of nodes: process nodes and resource nodes. A process node represents a process, and a resource node represents a resource.

In the resource allocation graph, a directed edge from a process node to a resource node represents a process requesting a resource, and a directed edge from a resource node to a process node represents a resource being held by a process. A cycle in the resource allocation graph indicates the presence of a deadlock.

6.4 Banker's Algorithm

The banker’s algorithm is a scheduling algorithm that can prevent deadlocks by checking if a request can be granted without leading to an unsafe state. It is based on the analogy of a small-town banker who grants lines of credit to customers.

In the banker’s algorithm, each process is assigned a maximum demand for each resource type, which represents the maximum number of resources the process will need at any given time. The system also maintains a current allocation of resources to each process and an available resource vector, which represents the number of resources of each type that are currently available.

When a process requests resources, the banker’s algorithm checks whether granting the request will lead to an unsafe state, i.e., a state where deadlock may occur. To do this, the algorithm simulates the granting of the request by subtracting the requested resources from the available resource vector and adding them to the process’s current allocation. It then checks whether there exists a safe sequence of processes that can complete their work using the available resources. If such a sequence exists, the request is granted; otherwise, it is denied.

The safe sequence is determined using a variation of the deadlock detection algorithm presented earlier. Starting from the current available resource vector, the algorithm searches for a process whose maximum demand can be satisfied using the available resources. If such a process is found, its allocated resources are released, and the process is removed from the system. The resources released by the process are added back to the available resource vector, and the search continues until either all processes have been satisfied or no process can be satisfied.

If the request cannot be granted, the process is forced to wait until the required resources become available. This may cause delays and reduce system efficiency, but it ensures that deadlock cannot occur.

In summary, the banker's algorithm is a scheduling algorithm that can prevent deadlocks by checking if a request can be granted without leading to an unsafe state. It is based on the analogy of a small-town banker who grants lines of credit to customers and ensures that the resources are allocated in a safe and efficient manner.

The banker's algorithm consists of the following steps:

- When a process requests a resource, the system checks if the request can be granted without leading to an unsafe state.
- If the request can be granted, the system allocates the resource to the process.
- If the request cannot be granted, the process is blocked until the resource becomes available.
- When a process releases a resource, the system checks if this release can lead to a safe state.
- If the release leads to a safe state, the system deallocates the resource from the process and grants the resource to the next process in the queue.

The Banker's algorithm uses a set of rules to determine whether a resource request should be granted or not. The rules are as follows:

- If a process requests resources, they are immediately denied if the resources they request are not currently available.
- If a process requests resources, they are immediately denied if granting the request will cause the system to enter an unsafe state.
- If a process requests resources, they are granted the resources if they are immediately available, and granting the resources will not cause the system to enter an unsafe state.
- If a process requests resources that are not immediately available, they are placed in a queue until the resources become available.

When the resources become available, the algorithm checks if granting the request will cause the system to enter an unsafe state. If it does not, the request is granted, and the process can continue.

Example: The concept of safe and unsafe states is typically used in conjunction with the Banker's algorithm for deadlock avoidance. Here is a pseudocode example:

```
/* Assume we have n processes and m resource types */


/* Function to check if a state is safe */

boolean isSafe(int available[], int max[][m], int allocation[][m], int n) {

    int work[m];

    boolean finish[n];

    int i, j;


    /* Initialize work and finish arrays */

    for (i = 0; i < m; i++) {

        work[i] = available[i];

    }

    for (i = 0; i < n; i++) {

        finish[i] = false;

    }


    /* Find an unfinished process with needs less than or equal to work */

    for (i = 0; i < n; i++) {

        if (finish[i] == false) {
            boolean needsMet = true;
            for (j = 0; j < m; j++) {
                if (max[i][j] - allocation[i][j] > work[j]) {
                    needsMet = false;
                    break;
                }
            }
            if (needsMet) {
                /* Release resources from process i */
                for (j = 0; j < m; j++) {
                    work[j] += allocation[i][j];
                }
                /* Mark process i as finished */
                finish[i] = true;
                /* Restart the search for an unfinished process */
                i = -1;
            }
        }
    }

    /* If all processes are finished, the state is safe */
    for (i = 0; i < n; i++) {
        if (finish[i] == false) {
            return false;
        }
    }
    return true;
}
```

This code assumes that the available resources, maximum resource needs, and current resource allocations are stored in arrays available, max, and allocation, respectively. The n parameter is the number of processes in the system, and m is the number of resource types. The function returns true if the state is safe and false otherwise.

In this chapter, we have discussed the concept of safe and unsafe states in the context of deadlock avoidance. We have also discussed the resource allocation graph and the banker's algorithm for deadlock avoidance. By using these techniques, operating systems can ensure that the system remains in a safe state, thereby preventing deadlocks from occurring.

7 Deadlock prevention

7.1 Attacking the mutual-exclusion condition

Mutual exclusion is a fundamental requirement for preventing race conditions and maintaining data consistency. However, as we have seen, it can also lead to deadlocks. In order to prevent deadlocks by attacking the mutual-exclusion condition, we need to find ways to allow multiple processes to access shared resources concurrently without interfering with each other.

One way to achieve this is to make data read-only. This means that multiple processes can access the same data at the same time without causing any conflicts. This approach works well for situations where data is being read, but it is not suitable for situations where data is being modified.

Another way to attack the mutual-exclusion condition is to use spooling. Spooling is a technique where data is temporarily stored in a buffer or queue until it can be processed. For example, when multiple processes want to print output on a shared printer, the output is first spooled to a temporary buffer. Then, a separate process, known as the printer daemon, accesses the printer and prints out the output from the buffer. By using spooling, multiple processes can generate output at the same time without interfering with each other. Since the printer daemon never requests any other resources, we can eliminate deadlock for the printer.

In some cases, it may also be possible to use non-exclusive access control mechanisms to allow multiple processes to access shared resources concurrently. For example, in a database management system, multiple processes can access the same database concurrently by using locking and transaction management mechanisms.

In summary, attacking the mutual-exclusion condition involves finding ways to allow multiple processes to access shared resources concurrently without interfering with each other. This can be achieved by making data read-only, using spooling, or using non-exclusive access control mechanisms. By using these techniques, we can prevent deadlocks caused by mutual exclusion while still maintaining data consistency and integrity.

7.2 Attacking the hold-and-wait condition

While requiring all resources to be requested before execution starts may eliminate deadlocks, it is often impractical. A more flexible approach is to use a technique called resource ordering. With resource ordering, resources are given a fixed order, and a process may only request resources in that order. If a process needs a resource that is later in the order, it must release all resources that come earlier in the order before making the request. This technique ensures that a process never holds resources while waiting for others.

Another approach to attacking the hold-and-wait condition is to use a two-phase locking protocol. In this protocol, a process may request resources one at a time, but once a resource is acquired, it is held until the process releases all resources. This technique ensures that a process will never request a resource while holding another, thereby avoiding the hold-and-wait condition. However, it may lead to resource starvation, as a process that holds a resource cannot request any others until it releases the held resource.

Yet another technique is to require that a process release all its resources whenever it is blocked, and then request all of them again when it is unblocked. This approach is known as restartable atomic actions and is typically used in database systems. While effective, it can be expensive, as it requires redoing all actions that have been completed up to the point of the block.

In summary, there are several ways to attack the hold-and-wait condition, including resource ordering, two-phase locking, and restartable atomic actions. Each technique has its advantages and disadvantages, and the choice of technique depends on the specific requirements of the system being designed.

7.3 Attacking the no-preemption condition

The third condition stated by Coffman et al. is the no-preemption condition. It states that resources cannot be taken away from a process unless that process releases them voluntarily. This condition makes it difficult to prevent deadlocks because it is hard to force a process to release resources that it is holding. However, there are some strategies that can be used to attack this condition and prevent deadlocks.

One approach is to use virtualization to create the illusion of preemption. For example, in the printer and plotter scenario discussed earlier, spooling printer output to disk creates a virtual printer that can be preempted if necessary. The printer daemon has exclusive access to the physical printer, but other processes can write to the spool area and create a queue of print jobs. The daemon reads the spool files and sends them to the printer as resources become available. If a process needs the plotter while waiting for the printer, it can be allocated the plotter resource without affecting the printer resource. When the printer resource becomes available, the process can resume printing without losing any data.

Another approach to attacking the no-preemption condition is to use timeouts to force processes to release resources. A timeout mechanism can be built into the resource allocation algorithm so that if a process holds a resource for too long, it is forcibly released. This approach is effective for some resources, such as network connections, where it is not too disruptive to terminate a connection and start over. However, it may not work for resources that hold state, such as files or database records, because terminating the process could lead to data corruption or inconsistency.

A third approach to attacking the no-preemption condition is to use priority-based scheduling to allocate resources. In a priority-based system, processes are assigned a priority level that determines their access to resources. A process with a higher priority can preempt a process with a lower priority if it needs the same resource. This approach can work well for real-time systems where certain tasks have strict timing requirements. However, it may not work well for general-purpose systems where fairness and equality are important.

In summary, the no-preemption condition makes it difficult to prevent deadlocks, but virtualization, timeouts, and priority-based scheduling are all effective strategies for attacking this condition. Each strategy has its strengths and weaknesses, and the choice of strategy depends on the specific requirements of the system.

7.4 Attacking the circular wait condition

Attacking the circular wait condition is the final step in preventing deadlocks. One way to eliminate circular wait is to enforce a rule that a process can only hold one resource at a time. This approach may work in some scenarios, but it can be impractical for processes that require multiple resources simultaneously.

Another approach is to impose a total ordering of all resources and require processes to request resources in that order. For example, if resources A, B, and C have an order such that $A < B < C$, then a process can only request B after obtaining A and can only request C after obtaining B. This approach can prevent circular wait, but it requires a strict ordering of all resources, which may be difficult to achieve in some systems.

A more flexible approach is to allow processes to request resources in any order but to impose a limit on the number of resources a process can hold at any given time. This limit can be set to the maximum number of resources that any process will need, which can prevent circular wait by limiting the number of processes that can be involved in a circular wait situation.

Another way to prevent circular wait is to use a resource allocation graph (RAG) to track resource requests and allocations. The RAG is a directed graph where nodes represent processes and resources, and edges represent requests and allocations. If the graph contains a cycle, then there is a circular wait situation, and deadlock is possible. To prevent deadlock, the system can use an algorithm to detect cycles in the RAG and break them by releasing resources held by one of the processes involved in the cycle.

In conclusion, the circular wait condition can be eliminated by enforcing a rule that a process can hold only one resource at a time, imposing a strict ordering of all resources, limiting the number of resources a process can hold at any given time, or using a resource allocation graph to detect and break cycles. Each of these approaches has its own strengths and weaknesses, and the choice depends on the specific requirements of the system.

7.5 Prevention through resource ordering and allocation policies

Prevention through resource ordering and allocation policies is an approach used to avoid deadlocks in operating systems. This approach involves imposing a particular order on the acquisition of resources, which helps to avoid the conditions that cause deadlocks.

The idea behind this approach is to define a hierarchy of resources and require that resources be acquired in a specific order. This way, each process will acquire the resources it needs in the right order, preventing circular wait conditions that cause deadlocks.

For instance, if two resources A and B are needed by a process, and the required order is A then B, then the process must first acquire resource A before acquiring resource B. This ensures that resource B is not already held by another process that might cause a deadlock.

This approach can be implemented using various methods, such as using a resource allocation table that defines the order in which resources must be acquired or using a priority-based approach where higher priority processes are given preference in acquiring resources.

One common resource ordering policy is the "first-come, first-served" approach. This approach ensures that resources are allocated to processes in the order in which they request them. However, this approach can lead to inefficient use of resources since a process that is holding a resource might block other processes from using it even when it is not actively using it.

Another approach is the "priority-based" approach, where higher priority processes are given preference in acquiring resources. This approach ensures that critical processes are given access to the resources they need before lower priority processes. However, this approach can also lead to inefficiencies if a higher priority process is waiting for a lower priority process to release a resource it needs.

Overall, prevention through resource ordering and allocation policies is an effective approach to avoid deadlocks in operating systems. However, it is important to choose the right resource allocation policy that balances efficiency and fairness.

7.6 Prevention through timeouts and deadlock detection

Preventing deadlocks is an essential aspect of operating system design. One approach to prevent deadlocks is through the use of timeouts and deadlock detection. In this chapter, we will discuss the concept of timeouts and deadlock detection, and their role in preventing deadlocks.

A timeout is a mechanism that enables a process to give up waiting for a resource after a certain period. Timeouts can be used to avoid deadlocks by enforcing a time limit on how long a process is allowed to wait for a resource. If the resource is not available within the specified time limit, the process is interrupted, and the resource is released, allowing other processes to access it.

Another approach to preventing deadlocks is through deadlock detection. Deadlock detection involves periodically checking the resource allocation graph for cycles, which would indicate the presence of a deadlock. When a cycle is detected, the operating system can take one of two actions: either preempt resources to break the deadlock, or kill one of the processes involved in the cycle. Deadlock detection can be implemented using algorithms such as the Banker's algorithm.

Timeouts and deadlock detection can be used together to prevent deadlocks. In this approach, processes are allowed to wait for a resource for a certain period, after which the operating system checks for deadlocks. If a deadlock is detected, the operating system can take appropriate action, such as releasing resources or killing processes.

Timeouts and deadlock detection are essential techniques for preventing deadlocks in operating systems. Timeouts provide a mechanism for processes to release resources if they are not available within a certain time, while deadlock detection allows the operating system to identify and resolve deadlocks before they cause system-wide issues. By using a combination of these techniques, operating systems can ensure that deadlocks are prevented or resolved quickly, improving system performance and reliability.

7.7 Two-phase locking

Two-phase locking is a technique used in many database systems to prevent deadlocks. It is based on the idea of acquiring all necessary locks before beginning any real work. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

The two-phase locking algorithm can be summarized as follows:

- In the growing phase, a transaction can acquire locks but cannot release any locks.
- In the shrinking phase, a transaction can release locks but cannot acquire any locks.

- Once a transaction enters the shrinking phase, it cannot return to the growing phase.

The two-phase locking algorithm guarantees serializability, meaning that the transactions are executed as if they occurred one at a time in some order, even though they may actually execute concurrently. Serializability ensures that the results of concurrent transactions are equivalent to the results of executing the transactions serially.

One potential drawback of two-phase locking is that it can lead to deadlock if transactions hold locks for an extended period of time. To avoid this, some database systems use a timeout mechanism, where a transaction is forced to release its locks after a certain period of time if it has not completed its updates.

In summary, two-phase locking is a technique used in database systems to prevent deadlocks. It ensures serializability by acquiring all necessary locks before beginning any real work, and releasing them only after all work has been completed. While it can lead to deadlocks if locks are held for an extended period of time, a timeout mechanism can be used to mitigate this risk.

8 Other issues

8.1 Livelock

Livelock is a situation where two or more processes keep changing their state in response to changes in the other process's state, but no progress is made. In other words, the processes are not deadlocked, but they are unable to proceed with their tasks because they are constantly responding to the actions of the other process.

One common cause of livelock is when two or more processes are waiting for a shared resource to become available, but each process releases the resource when it detects that the other process is waiting for it. This results in a situation where the resource is constantly being passed back and forth between the processes, but neither process can actually make progress.

Another cause of livelock is when two or more processes are trying to coordinate their actions, but each process is waiting for the other process to take the first step. For example, consider a situation where two robots are trying to navigate through a narrow corridor. If each robot keeps moving aside to let the other robot pass, they may end up moving back and forth without ever making any progress.

To avoid livelock, it is important to design algorithms that are resilient to unexpected events and that can handle situations where processes need to coordinate their actions. One approach is to use timeouts to ensure that processes do not wait indefinitely for a resource or a response from another process. Another approach is to use randomized algorithms that introduce some degree of randomness into the decision-making process, which can help to break deadlocks and prevent livelocks.

Overall, livelock is a complex issue that requires careful consideration when designing distributed systems and algorithms. By understanding the causes and implications of livelock, it is possible to design systems that are more resilient and that can handle unexpected events in a graceful manner.

8.2 Livelock vs Deadlock

Livelock and deadlock are not always straightforward to identify and can occur in unexpected ways. For example, in some operating systems, the number of processes allowed is limited by the number of entries in the process table. When a program attempts to fork a new process but fails due to a full process table, a reasonable strategy might be to wait for a random time and try again. However, this can lead to livelock if multiple processes are attempting to fork at the same time and repeatedly fail due to the same resource constraint.

In this scenario, the processes are all attempting to acquire the same finite resource (i.e., an entry in the process table) and are repeatedly failing and retrying at the same time. This can lead to a situation where none of the processes are able to make progress, even though they are all technically executing.

To avoid this type of livelock, a better strategy might be for the processes to wait for a random time and then retry the fork operation at different times, rather than all attempting to retry at the same time. This way, there is a higher likelihood that at least one process will be able to acquire the necessary resource and make progress, rather than all of them continuously failing and retrying.

8.3 Starvation

Starvation is a phenomenon that occurs when a process is perpetually denied access to a resource it requires to execute, even though the resource is available. This problem is closely related to deadlock and livelock, as all three can occur due to poor resource allocation policies.

When a system is dynamic, requests for resources occur frequently, and there is a need to allocate resources fairly. However, the resource allocation policy may not be optimal and can lead to some processes never getting the resources they need, even though they are not deadlocked.

A common example of starvation is the allocation of a printer. In a scenario where multiple processes want to use the printer simultaneously, a decision must be made about who gets to use it first. However, if the system's policy is to always give the printer to the same process, other processes may be starved of access to the printer indefinitely.

One way to prevent starvation is to implement a fairness policy that ensures every process gets a chance to use the resource. This policy could be based on a round-robin algorithm, where each process gets a turn to use the resource in a predetermined order.

Another solution is to implement a priority-based resource allocation policy. Processes with higher priority levels are given priority access to the resource, ensuring that they are not starved of the resources they need.

It is important to note that, although a fair allocation policy may prevent starvation, it may also lead to some resources being underutilized. Therefore, a balance must be struck between preventing starvation and maximizing resource utilization.

In conclusion, starvation is a problem that occurs when a process is perpetually denied access to a resource it needs, even though the resource is available. To prevent starvation, a fair resource allocation policy must be implemented that ensures every process gets a chance to use the resource.

9 Case Study: Deadlocks in Linux

Deadlocks are one of the most challenging problems in operating system design and implementation. A deadlock occurs when two or more processes are waiting for resources held by each other, leading to a state of impasse where none of the processes can proceed. This can have severe consequences, such as system crashes, loss of data, and reduced system performance.

In this chapter, we will discuss the necessary conditions for deadlocks, including resource types, allocation policies, hold and wait, no preemption, and circular wait. We will also explore various methods of detection and prevention, including the resource allocation graph, Banker's algorithm, prevention through resource ordering and allocation policies, and prevention through timeouts and deadlock detection.

Furthermore, we will discuss methods of deadlock resolution, including killing processes, resource preemption, and rollback and recovery. We will also delve into the concept of deadlock avoidance, including safe and unsafe states, the Banker's algorithm for deadlock avoidance, and a comparison with other resource allocation algorithms.

Finally, we will take a closer look at the case study of deadlocks in Linux. We will provide an overview of Linux's approach to handling deadlocks and compare it with other operating systems. Additionally, we will examine the impact of deadlocks on Linux's performance and reliability.

9.1 Overview of Linux's approach to handling deadlocks

Linux is an open-source operating system that is widely used in various applications. Linux has a sophisticated approach to handle deadlocks, which is an essential feature of an operating system. This chapter will provide an overview of Linux's approach to handling deadlocks.

The Linux operating system employs a combination of prevention, detection, and resolution techniques to deal with deadlocks. The Linux kernel has a deadlock detection and resolution mechanism that can identify and resolve deadlocks. The deadlock resolution mechanism in Linux is based on resource preemption and rollback techniques.

The Linux kernel's deadlock detection mechanism is based on a resource allocation graph (RAG), which is similar to the one discussed in the previous chapter. The Linux kernel maintains a RAG that represents the current state of the system's resources and their allocation. Whenever a new process requests a resource, the kernel checks whether the request creates a cycle in the RAG. If a cycle exists, the kernel identifies the processes involved in the cycle and takes appropriate actions to resolve the deadlock.

Example: Sure, here's a simple pseudocode for detecting a cycle in a resource allocation graph:

```
1. Mark all nodes as unvisited.

2. For each node in the graph:

   a. If the node is unvisited, perform depth-first search (DFS) traversal.

   b. While traversing, mark the current node as visited.

   c. If we encounter a node that is already marked as visited, then there is a cycle in the graph.

   d. After DFS traversal is complete, clear the visited marks for all nodes.

3. If no cycle is found after DFS traversal of all nodes, the graph does not have any deadlock.
```

Note that this is a simplified pseudocode and there are more efficient algorithms for cycle detection in graphs, such as Tarjan's algorithm or Kosaraju's algorithm.

In Linux, the kernel employs the Ostrich algorithm for deadlock detection. The Ostrich algorithm is a heuristic-based algorithm that uses a combination of cycle detection and process suspension to detect and resolve deadlocks. Whenever a deadlock is detected, the kernel suspends one or more processes involved in the deadlock to break the cycle and resolve the deadlock.

Example: Here's an example pseudocode for the Ostrich algorithm for deadlock detection:

```
// Initialize the data structures

let work = available
let finish = array of size n, filled with false

let deadlock_detected = false

let deadlock_processes = empty list


// Repeat until all processes have finished or a deadlock is
detected

while there are unfinished processes and not deadlock_detected:

    let found = false


    // Check each unfinished process

    for each process in processes:

        if finish[process] == false and need[process] <= work:

            // Found a process that can complete

            found = true

            work += allocation[process]

            finish[process] = true


    // If no process can complete, a deadlock has occurred

    if found == false:

        deadlock_detected = true


        // Find all processes involved in the deadlock

        for each process in processes:

            if finish[process] == false:

                deadlock_processes.add(process)
```

```
// If a deadlock was detected, print the list of processes involved
if deadlock_detected:
    print("Deadlock detected. Processes involved:", deadlock_processes)
```

Note that this is a simplified example and may not be suitable for all situations. The actual implementation may vary depending on the specific requirements and constraints of the system.

Apart from deadlock detection and resolution, Linux also employs several prevention techniques to avoid deadlocks altogether. One of the primary prevention techniques used in Linux is resource ordering. In resource ordering, resources are allocated to processes in a predefined order, thereby preventing the possibility of a circular wait. Linux also uses timeout mechanisms to prevent deadlocks, where a process is forced to release a resource after a specified period to avoid resource starvation.

In conclusion, Linux's approach to handling deadlocks is a combination of prevention, detection, and resolution techniques. The kernel employs the Ostrich algorithm for deadlock detection, and resource preemption and rollback techniques for deadlock resolution. Linux also uses prevention techniques such as resource ordering and timeout mechanisms to avoid deadlocks altogether. Overall, Linux's approach to handling deadlocks is an essential feature of the operating system that ensures the system's reliability and performance.

9.2 Comparison with other operating systems

In this chapter, we will compare the approaches taken by different operating systems in handling deadlocks. Deadlocks are a common problem faced by most operating systems, and different operating systems have different ways of handling them.

Windows and Linux are two popular operating systems that take different approaches to handle deadlocks. Windows uses a combination of prevention, detection, and resolution techniques to handle deadlocks. On the other hand, Linux uses prevention and detection techniques.

Windows uses a resource allocation graph to detect deadlocks. If a cycle is found in the graph, it indicates a deadlock. Windows also uses timeouts to detect deadlocks. If a process is waiting for a resource for too long, it is considered to be deadlocked, and Windows takes appropriate action to resolve the deadlock.

Windows also uses a combination of prevention and resolution techniques to handle deadlocks. Windows prevents deadlocks by ensuring that processes request all the resources they need at once. This eliminates the hold and wait condition. Windows also uses resource preemption to resolve deadlocks. If a process is holding a resource that another process needs, Windows preempts the resource from the holding process to resolve the deadlock.

Linux takes a different approach to handle deadlocks. Linux primarily uses prevention techniques to prevent deadlocks from occurring in the first place. Linux ensures that a process requests all the resources it needs before it begins executing. This eliminates the hold and wait condition.

Linux also uses a timeout mechanism to detect deadlocks. If a process is waiting for a resource for too long, it is considered to be deadlocked, and Linux takes appropriate action to resolve the deadlock.

In terms of handling deadlocks, both Windows and Linux have their advantages and disadvantages. Windows is better at handling complex deadlocks that involve multiple resources and processes, while Linux is better at preventing deadlocks from occurring in the first place.

Overall, it is important for operating systems to have effective deadlock handling mechanisms to ensure the reliability and stability of the system. The choice of approach depends on the specific requirements and constraints of the system.

10 Conclusion

In conclusion, deadlocks are a complex issue that can have serious consequences for the reliability and performance of an operating system. It is essential for operating system designers and developers to have a deep understanding of the necessary conditions for deadlocks, as well as the methods of detection, prevention, and resolution.

In this chapter, we have explored the various aspects of deadlocks, including their definition, necessary conditions, detection and prevention methods, resolution techniques, and avoidance strategies. We also discussed a case study on deadlocks in Linux, which highlights the importance of proper handling of deadlocks for the smooth functioning of a complex operating system.

By implementing effective mechanisms for dealing with deadlocks, operating system designers and developers can ensure that their systems are more reliable and robust. It is important to continuously evaluate and update these mechanisms to adapt to changing technology and system requirements.

Overall, understanding and preventing deadlocks is an essential aspect of operating system design and maintenance. With the right approach, we can minimize the risk of deadlocks and ensure that our systems continue to operate efficiently and reliably.