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 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.

1.2.1 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.

1.2.2 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.

1.3 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 Hold and wait

The hold and wait condition is one of the necessary conditions for a deadlock to occur. This condition states that a process is holding onto a resource while also waiting for another resource that is currently being held by another process. This creates a situation where both processes are waiting for the other to release a resource, resulting in a deadlock.

2.3 No preemption

The no preemption condition is another necessary condition for a deadlock. This 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.

2.4 Circular wait

The circular wait condition is the final necessary condition for a deadlock to occur. This condition states that there is a circular chain of processes, each holding onto a resource that is needed by the next process in the chain. This creates a situation where none of the processes can proceed, leading to a deadlock.

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 Detection and Prevention

Deadlocks are situations that occur when two or more processes are blocked and waiting for each other to release resources, resulting in a standstill in the system. It is essential to understand the necessary conditions that lead to the occurrence of deadlocks and the ways to prevent them.

In the first section, we will discuss the necessary conditions that lead to deadlocks, including resource types and allocation policies, hold and wait, no preemption, and circular wait.

In the second section, we will delve into the detection and prevention techniques of deadlocks. We will cover the resource allocation graph and Banker's algorithm, which are commonly used for detecting deadlocks. Furthermore, we will explore ways to prevent deadlocks through resource ordering and allocation policies and timeouts and deadlock detection.

By the end of this chapter, you will have a comprehensive understanding of the techniques and algorithms for detecting and preventing deadlocks, which will aid you in developing robust and reliable operating systems.

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 Banker's algorithm

Banker's algorithm is one of the most popular algorithms used for deadlock prevention in operating systems. It is a resource allocation and process scheduling algorithm that helps to ensure that a system avoids deadlock by checking the available resources before granting access to a process. In this chapter, we will discuss the Banker's algorithm in detail, its implementation, and its advantages.

The Banker's algorithm is based on the idea of ensuring that a system does not enter an unsafe state. An unsafe state is one in which a deadlock can occur if all processes that have requested resources are allowed to execute. To prevent this, the algorithm first checks if granting

a request will lead to an unsafe state. If it does, the request is denied, otherwise, the request is granted.

The algorithm works by maintaining a data structure called the 'allocation matrix,' which represents the number of resources of each type currently allocated to each process. Additionally, there is a 'request matrix' that represents the number of resources of each type that each process still needs to complete its execution. There is also a 'available matrix' that represents the number of resources of each type that are currently available.

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.

The Banker's algorithm is an effective way to prevent deadlocks, as it ensures that a system never enters an unsafe state. However, it has some limitations. One limitation is that it assumes that the maximum resource requirements of a process are known in advance. In reality, this information may not be available, which can make it difficult to implement the algorithm. Additionally, the algorithm can be slow, as it involves a lot of checking and processing.

Example: Here's an example pseudocode for the Banker's Algorithm:

```
// Define number of processes and resources

let n = number of processes

let m = number of resources


// Define the maximum amount of each resource type each process can request

let max[n][m] = maximum resource needs of each process


// Define the amount of each resource type currently allocated to each process

let allocation[n][m] = current resource allocation for each process


// Define the total amount of each resource type available in the system

let available[m] = total resources available


// Define a Boolean array indicating whether a process has been terminated or not

let terminated[n] = {false, false, ..., false}


// Define a two-dimensional Boolean array indicating whether a process can finish with the resources it has

let can_finish[n][m]


// Initialize can_finish array to false for all processes

for i = 1 to n

    for j = 1 to m
```

```
        can_finish[i][j] = false


// Calculate the need matrix
let need[n][m]
for i = 1 to n
    for j = 1 to m
        need[i][j] = max[i][j] - allocation[i][j]


// Find processes that can finish with the resources they have
for i = 1 to n
    if terminated[i] = false and need[i] ≤ available
        for j = 1 to m
            can_finish[i][j] = true


// Define a sequence to store the safe sequence of processes
let safe_sequence = empty sequence


// While there exists an unfinished process that can finish with the resources it has
while there exists i such that terminated[i] = false and can_finish[i] = true
    // Mark the process as terminated
    terminated[i] = true


    // Release the resources allocated to the process
    for j = 1 to m
```

```
        available[j] = available[j] + allocation[i][j]


    // Add the process to the safe sequence
    append i to safe_sequence


    // Update can_finish array for remaining processes
    for j = 1 to n
        if terminated[j] = false and need[j] ≤ available
            for k = 1 to m
                can_finish[j][k] = true


// If all processes are terminated and the safe sequence has been
found
if length(safe_sequence) = n
    // System is in a safe state
    return safe_sequence
else
    // System is in an unsafe state
    return "unsafe state"
```

Note that this is just one possible implementation, and the specifics may vary depending on the programming language and context.

In conclusion, the Banker's algorithm is an essential tool for preventing deadlocks in operating systems. It provides a way to ensure that a system never enters an unsafe state, which helps to guarantee the reliability and stability of the system. While the algorithm has some limitations, it is still widely used and considered one of the most effective methods for preventing deadlocks.

3.3 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.

Example: Here's an example pseudocode for implementing the "first-come, first-served" approach:

```
while (true) {

   request = getNextRequest(); // get the next resource request

   resource = request.resource; // get the requested resource

   process = request.process; // get the process requesting the resource


   if (resource.isAvailable()) { // if the resource is available

      resource.allocate(process); // allocate the resource to the process

      process.continue(); // continue executing the process

   } else { // if the resource is not available

      resource.addToQueue(process); // add the process to the resource's queue

   }

}
```

This pseudocode checks if a resource is available and, if not, adds the requesting process to a queue. The process is only allocated the resource when it becomes available, ensuring that resources are allocated in the order they are requested.

3.4 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.

3.4.1 Timeouts:

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.

3.4.2 Deadlock Detection:

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.

3.4.3 Combining Timeouts and Deadlock Detection:

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.

4 Deadlock Resolution

Deadlocks are one of the most significant problems in operating systems, and they occur when two or more processes are unable to proceed because each is waiting for a resource held by the other. Deadlocks can cause a system to freeze or crash, resulting in data loss or corruption, and system downtime. Therefore, understanding and preventing deadlocks is essential for the efficient and reliable operation of an operating system.

In this chapter, we will discuss the necessary conditions for deadlocks, which include resource types and allocation policies, hold and wait, no preemption, and circular wait. We will also explore the methods for detecting and preventing deadlocks, such as resource allocation graphs, the Banker's algorithm, prevention through resource ordering and allocation policies, and prevention through timeouts and deadlock detection.

Furthermore, 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.

4.1 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.

4.2 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.

4.3 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.

4.3.1 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.

4.3.2 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.

5 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.

5.1 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.

5.2 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.

5.3 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.

5.4 Banker's Algorithm for Deadlock Avoidance

The banker's algorithm is a deadlock avoidance algorithm that works on the principle of avoiding unsafe states. The algorithm ensures that the system remains in a safe state by checking if a request for a resource can

be granted without leading to an unsafe state. The banker's algorithm uses the resource allocation graph to determine if a request can be granted or not.

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.

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.

6 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.

6.1 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.

6.2 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.

7 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.