Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology

Advanced Operating Systems

Assignment - II

Submitted by:
Priya Shrestha
Roll No: 24

Submitted to:
Dr. Binod Kumar Adhikari
TU, CDCSIT

Date: May 6, 2025

Abstract

Deadlock represents a critical challenge in computing where multiple processes become permanently blocked, each holding resources while waiting indefinitely for others. Four necessary conditions for deadlock (mutual exclusion, hold and wait, no pre-emption and circular wait) are examined alongside key handling approaches (prevention, avoidance, detection and recovery, and the “ostrich” approach) with analysis focused on their safety/performance trade-offs. The analysis of deadlock extends to distributed systems and real-time environments, where traditional centralized deadlock resolution proves inadequate due to partial observability and strict timing constraints. Ultimately, advancing deadlock management in modern systems demands adaptive strategies that balance correctness, efficiency, and scalability across diverse computing paradigms.

Keywords: Deadlock, Resource Allocation graph (RAG) , Banker’s algorithm, distributed systems, deadlock detection.

INTRODUCTION

In modern computing systems, processes often compete for limited resources such as CPU time, memory space, files, and I/O devices. time, memory space, files, and I/O devices. This competition can lead to a problematic phenomenon known as **deadlock** — a critical condition where a set of processes is blocked because each process is waiting for a resource held by another. The concept of deadlock was formally characterized by Coffman, Elphick, and Shoshani in 1971. A process in a multiprogramming environment is said to be in deadlock if it is waiting for a particular event (such as resource availability) that will not occur due to circular dependencies among processes. [1]

Deadlock is particularly problematic in multi-processing and multi-threaded and distributed systems, where concurrent access to shared resources is a necessity. Deadlocks can cause system freezes, resource starvation, and performance degradation, making them a significant concern in operating systems, databases, and distributed environments. For example, crossroad/T-junction deadlock, resource deadlock etc.

The importance of deadlock management lies in its ability to prevent system stagnation and performance degradation, as inadequate handling can lead to indefinite process suspension, resource wastage, and even system failures. With the advancement of computing from single-core architectures to complex distributed networks and cloud environments, effective deadlock prevention, detection, and recovery have become essential areas of research to ensure system reliability and efficiency. [2]

How Does Deadlock occur in the Operating System?

Before going into detail about how deadlock occurs in the Operating System, let's first discuss how the Operating System uses the resources present. A process uses resources by first **requesting** them from the OS (e.g., CPU time, memory, I/O devices); if available, the OS allocates the resource, allowing the process to **use** it for execution (e.g., computations, file operations, or printing). Once the process completes its task, it **releases** the resource back to the OS, making it available for other processes.

A process in an operating system uses resources in the following way:

- Requests a resource
- Use the resource
- Releases the resource

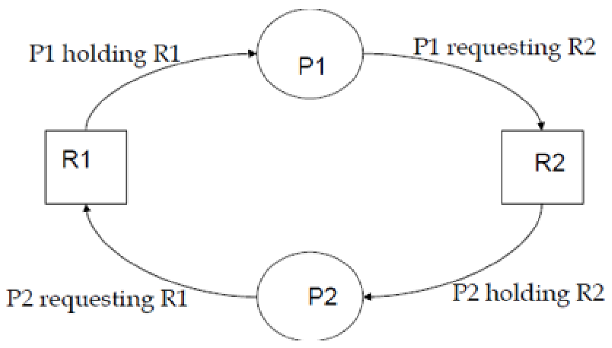

Figure: Circular-wait Resource Deadlock

A situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by others. For example, in the above diagram, Process P1 holds resource R1 and waiting for resource R2. Similarly, Process P2 holds resource R2 and waiting for resource R1.

Summary

Deadlock refers to a specific situation in a system where a group of processes are each waiting for an event that only another process in the group can cause. None of the processes can proceed, leading to a standstill. [3]

Necessary Conditions for Deadlock

According to Coffman et al. (1971), four conditions must simultaneously hold for a deadlock to occur:

1. **Mutual Exclusion:** At least one resource must be held in a non-shareable mode.

2. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources held by other processes.

3. **No Preemption:** Resources cannot be forcibly taken from processes; they must be released voluntarily.

4. **Circular Wait:** A set of processes must exist such that each process is waiting for a resource held by the next process in the set.

If any one of these conditions is absent, deadlock cannot occur.

Strategies for Handling Deadlock

There are four main strategies to address deadlock:

Deadlock prevention

This can be achieved by negating any one of the four deadlock conditions so that deadlock cannot occur.

- **Denying Mutual Exclusion**
  If all the resources were shared among multiple processes, deadlock would never occur.
- **Denying No pre-emption**
  Resources can be pre-empted from processes. If the process holding resources is denied, a request for additional resources, that process must release its held resources and if necessary, request them again together with additional resources.
- **Denying Hold and wait**
  A process must acquire all the necessary resources before execution starts. If complete set of resources (free/unfree) is not available, the process must wait until available. But while waiting, process should not hold any resources.
- **Denying Circular wait**
  One way to deny circular wait is to uniquely numbered all resources and to acquire that processes must request resources in linear ascending/descending order.

Deadlock Avoidance

This can be achieved by careful resource allocation by deciding whether a granted resource is safe or unsafe and make the allocation when it is safe.

Safe state

A system is said to be safe if the system can allocate all resources requested by all processes without entering a deadlock state. It consists of a safe sequence with no deadlocks.

Unsafe state

A system is said to be unsafe if system is not able to prevent processes from requesting resources which can lead to deadlock.

To avoid the deadlock, we also use Banker's Algorithm.

Banker's Algorithm

It is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. It is used to ensure same allocation of resources to processes while avoiding deadlocks.

Deadlock Detection and Recovery

- If all resource has only a single instance, then deadlock detection uses variant of Resource Allocation Graph called wait-for-graph.
- Wait-for-graph is obtained by removing resources nodes and collapsing appropriate edges.
- A deadlock exists in a system if and only if wait-for-graph contains a cycle.
- To detect deadlock, system needs to periodically invoke an algorithm that searches for a cycle in the graph.

  - **Recovery by Resource Preemption**
    Preempt some resources temporarily from a process and give these resources to other processes until the deadlock cycle is broken.
  - **Recovery by Process Termination**
    Select the victim process first and kill them so that deadlock can be eliminated.

Just Ignore the problem

There is no good way of dealing with deadlock so ignore the problem altogether. For most Operating system, deadlock is a rare occurrence. So, problem of deadlock ignores like an ostrich sticking its head in the sand and hoping the problem will go away. This algorithm is used by UNIX and Windows OS.

Literature Review

The foundational work by Coffman, Elphick, and Shoshani (1971) established the conditions for deadlock and initiated systematic research in this domain. Around the same time, Dijkstra (1965) introduced synchronization primitives like **semaphores**, highlighting resource management complexities and laying groundwork for later deadlock studies. [1]

Dijkstra's **Banker's Algorithm** is notable for dynamically examining resource allocation and ensuring that the system never enters an unsafe state, thereby avoiding deadlocks.

Holt (1972) expanded on the concept of safe and unsafe states in systems, providing additional theoretical tools for designing deadlock-avoidant systems. [4]

Detection in Distributed Systems

Chandy, Misra, and Haas (1983) extended deadlock detection into **distributed environments**, where no central control exists. They proposed probe-based algorithms that detect cycles in a distributed resource allocation graph, overcoming the difficulties of partial knowledge and communication delays.

Distributed deadlock detection continues to be challenging, requiring algorithms that minimize communication overhead while accurately detecting cycles in dynamic and large-scale systems (Kshemkalyani & Singhal, 2011).

Deadlocks in Databases

In database systems, transactions often contend for locks on data items, leading to potential deadlocks. The **Wait-For Graph (WFG)** model is commonly used for deadlock detection in database systems. Cycles in the WFG indicate deadlocks (Gray & Reuter, 1993). [2]

The **wait-die** and wound-wait protocols are popular deadlock prevention techniques:

- ➢ In wait-die, older transactions wait for younger ones, but younger transactions abort if they must wait for an older transaction.
- ➢ In wound-wait, older transactions preempt younger ones, forcing younger transactions to roll back if conflict occurs (Eswaran et al., 1976).

Recent Research Trends

With the growth of cloud computing and microservices architectures, deadlock management faces new complexities:

- ➢ **Virtualization and dynamic resource scaling** make resource state prediction more difficult.
- ➢ **Service-Oriented Architectures (SOA)** and **containerized applications** introduce dynamic and complex dependencies between components.
- ➢ **Machine Learning (ML)** approaches have been proposed to predict deadlocks based on system behavior and resource usage patterns (Wang, Li, & Chen, 2019).

Modern research is increasingly focused on **dynamic deadlock detection**, **fault-tolerant deadlock recovery**, and **adaptive resource allocation** in large-scale, heterogeneous systems.

Conclusion

Deadlock remains a critical concern in the field of computing, particularly as systems become more complex and interconnected. Early theories provided a strong conceptual foundation for understanding and managing deadlock, while classical algorithms like the Banker’s Algorithm and distributed detection methods addressed practical concerns in single-machine and distributed environments. [4]

As computing environments evolve, static prevention and detection methods are no longer sufficient. Emerging technologies such as machine learning offer promising avenues for dynamic and predictive deadlock management. Future research must continue to focus on scalable, intelligent deadlock handling mechanisms that can adapt to the ever-increasing complexity of modern computing systems. [1]

A thorough understanding of deadlock, its causes, and management strategies is essential for developing robust, efficient, and reliable computer systems.

References

[1] K. P. G. J. N. L. R. A. &. T. I. L. Eswaran, The notions of consistency and predicate locks in a database system. Communications of the ACM, 19(11), 624–633., 1976.

[2] R. C. Holt, “ Some deadlock properties of computer systems. ACM Computing Surveys (CSUR), 4(3), 179–196,” 1972.

[3] K. M. M. J. &. H. L. M. Chandy, Distributed deadlock detection. ACM Transactions on Computer Systems (TOCS),, 1(2), 144–156, 1983.

[4] E. G. E. M. J. &. S. A. Coffman, System deadlocks. ACM Computing Surveys (CSUR), 3(2), 67–78., 1971.

[5] E. W. Dijkstra, Solution of a problem in concurrent programming control. Communications of the ACM, 8(9), 569., 1965.