Justin Ciocoi

Dec. 7, 2023

CSCI 375 Operating Systems

Chapter 8: Deadlocks

- In a multiprogramming environment, several threads might be competing for a finite amount of resources
- If a thread is waiting for a resource, but because of some activity, is never permitted to leave the waiting state, we say the thread has entered a *deadlock*

8.1: System Model

- Any system consists of a finite number of resources, and in most modern systems these resources must be distributed across a number of competing threads
- Each resource may accomplish a different task and have a different number of instances
- Any instance of a resource class should satisfy a call from a thread for that type of resources
    - If this is not the case, then the resource classes have been designed improperly by the operating system designer
- Mutex locks and semaphores can also be considered system resources, and on modern systems are the most likely source of a deadlock
- However, "lock" is not defined as a resource class since different locks will control access to different shared data, so each lock will be granted its own resource class
- This chapter will discuss deadlocks occurring with kernel resources, but it is also possible for a deadlock to occur between process such as in the case of ongoing interprocess communication
- Under the normal mode of operation, a thread may utilize a resource in only the following sequence
    1. **Request**: The thread requests the resource, which is either granted, or not, which places the thread into a waiting state
    2. **Use**: The thread can operate on the resource

3. **Release**: Once the thread has completed using the resource, it can release the resource such that other threads may now access it

8.3: Deadlock Characterization

- **8.3.1: Necessary Conditions**
    - A deadlock situation can arise in a system if and only if the following four properties hold simultaneously
        1. *Mutual Exclusion*
        2. *Hold and Wait*
        3. *No Preemption*
        4. *Circular Wait*
- **8.3.2: Resource Allocation Graph**
    - A system's *resource allocation graph* is a visual representation of a systems resources and the various different requesting threads in a system
    - Each graph has a set of vertices, V, and a set of edges, E
    - The set V is partitioned into two different types of nodes
        - $T = \{T_1, T_2, ..., T_n\}$, which is the set consisting of all active threads in the system
        - $R = \{R_1, R_2, ..., R_n\}$, which is the set consisting of all resource types in a system
    - An edge directed from a thread to a resource type denotes a request for that particular resource type
    - An edge directed from a resource type to a thread indicates that that thread currently is holding said resource type
    - If a resource allocation graph does not have a cycle, then the system is *not* in a deadlocked state
    - If there is a cycle, then the system *may or may not* be in a deadlocked state

8.4: Methods for Handling Deadlocks

- Generally speaking, there are three ways by which the problem of deadlocks can be dealt with
  - They can be ignored altogether and we can pretend that deadlocks will never occur in a system
  - We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
  - We can allow the system to enter the deadlocked state, detect it, and then recover
- The first solution is the one that is most commonly used by modern operating systems including Windows and Linux
- Thus, it is usually up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution
- In order to ensure deadlocks never occur, a system could use either a *deadlock prevention*, or *deadlock avoidance* scheme
- Deadlock prevention provides a set of methods by which a system will endure that at least one of the necessary conditions for a deadlock cannot hold
- Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime
  - With this knowledge, the operating system is capable of making threads wait or not in a manner that will altogether avoid deadlocks arising in the system

8.5: Deadlock Prevention

- Here we will examine different deadlock prevention schemes which aim to eliminate one of the four necessary conditions for deadlocks
- **8.5.1: Mutual Exclusion**
  - The mutual exclusion condition must hold, since there are certain system resources which are inherently non-shareable, such as a mutex lock
- **8.5.2: Hold and Wait**
  - In order to ensure that the hold and wait condition never occurs, we must guarantee that when a thread requests a resource, it does not already hold any other resources

- We could allow only threads holding no resources to request resources, or make each thread request all resources at once, but either solution will cause low resource utilization and task starvation

- **8.5.3: No Preemption**
  - In order to make sure no preemption does not occur, we can simply create a protocol such that if a thread requests a resource and must wait, it will then release all resources\
  - Alternatively, we can first check whether the resources are available and then take action based upon that, granting the requested resources if they are available

- **8.5.4: Circular Wait**
  - The prior deadlock prevention schemes are largely impractical for a variety of reasons
  - However, the circular wait condition presents an opportunity for a much more practical solution
  - To illustrate this, we will let $R = \{R_1, R_2, ..., R_n\}$ be the set of resource types where each type has a unique integer
  - Formally, we will define a one-to-one function $F : R \to N$ where N is the set of natural numbers

$$F(\text{first\_mutex}) = 1$$
$$F(\text{second\_mutex}) = 5$$

  - Now imagine each thread can request resources only in an increasing order of enumeration
    - This means that a thread can request R_i, and then R_j only if $F(R_j) > F(R_i)$
  - For example, a thread that wants to use both first_mutex and second_mutex, must first request first_mutex, and then second_mutex
  - If these two protocols exist, then the circular wait cannot hold

8.6: Deadlock Avoidance

- An alternative method for ensuring that deadlocks to not occur in a system is deadlock avoidance

- In this method, the system will obtain additional information about resources, threads, and resource usage, and then be able to make scheduling decisions based on this information in a manner in which deadlocks will not occur
- **8.6.1: Safe State**
    - A state is *safe* if the system can allocate resources up to each thread's maximum in some order and still avoid a deadlock
    - More formally, we can say that a system is in a safe state if there exists a safe sequence
    - It is important to note that not all unsafe states are deadlocks, and instead the presence of an unsafe state merely implies the *possibility* of a deadlock