**Deadlocks**

**Deadlock** – It is a state where multiple processes/threads are unable to proceed because each is waiting for other to release a resource, or take action resulting in stalemate.

**System Model**

In a system with multiple competing threads and finite resources, these resources can be categorized into different types or classes, each comprising a certain number of identical instances. For instance, resources like CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) fall into various resource types.

For example, if a system has four CPUs, the resource type "CPU" has four instances. Similarly, the resource type "network" might have two instances. When a thread requests an instance of a resource type, the allocation of any available instance of that type should fulfill the request. If this isn't the case, it suggests that the instances within the resource type are not identical, indicating a problem with the definition of resource type classes.

In Chapter 6, various synchronization tools like mutex locks and semaphores are discussed. These tools are essential for managing concurrent access to shared resources in a system (cause of deadlocks). However, unlike other system resources like CPUs or network interfaces, defining these synchronization tools isn't typically a problem.

Each synchronization tool, like a lock, is usually associated with a specific data structure. For example, one lock might protect access to a queue, while another locks access to a linked list. Because of this association, each instance of a lock is typically assigned its own resource class. This classification helps ensure that each resource type is adequately defined; making it easier to manage concurrent access and prevent issues like deadlock (IPCs can also result in deadlocks).

In a system, a thread needs to formally request a resource before utilizing it and should release the resource once it's done using it. When executing its task, a thread can request as many resources as necessary to fulfill its designated role. However, it's crucial to ensure that the number of resources requested doesn't surpass the total resources available in the system. For instance, a thread can't ask for two network interfaces if the system only has one available.

In the standard mode of operation, a thread utilizes a resource following these steps:

1. **Request**: The thread initiates a request to acquire the resource it needs. If the resource is available, the request is granted instantly. However, if the resource is currently in use (e.g., a mutex lock held by another thread), the requesting thread must wait until it becomes available.

2. **Use**: Once the resource is acquired, the thread can perform operations on it. For example, if the resource is a mutex lock, the thread can access the critical section of code protected by the lock.

3. **Release**: After completing its operations, the thread releases the resource. This step is essential to make the resource available for other threads that might need it.

In system design, the request and release of resources often involve system calls, which are functions provided by the operating system for interacting with hardware or managing system resources. These calls are familiar from Chapter 2, such as requesting and releasing a device, opening and closing a file, or allocating and freeing memory.

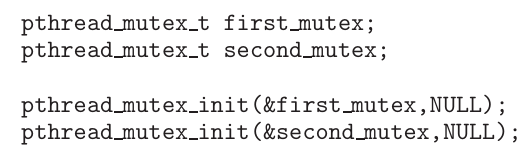
Similarly, in synchronization mechanisms like mutex locks and semaphores discussed in Chapter 6, request and release operations are achieved through specific operations like wait() and signal() for semaphores, or acquire() and release() for mutex locks.

Whenever a thread interacts with a resource managed by the kernel, the operating system ensures that the thread has properly requested and been allocated the resource. A system table maintains records indicating whether each resource is currently free or allocated. For allocated resources, the table also keeps track of the thread to which it is assigned. If a thread attempts to request a resource currently in use by another thread, it may be added to a queue of threads waiting for access to that resource.

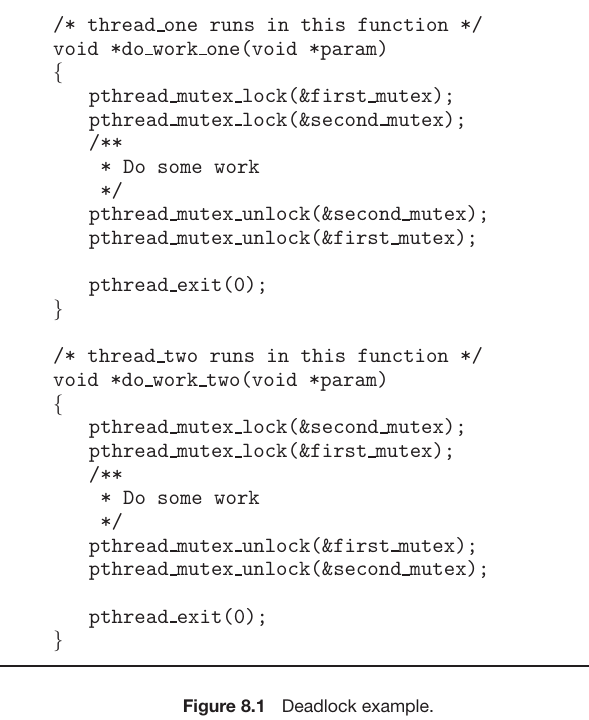
A set of threads enters a deadlocked state when each thread in the set is waiting for an event that can only be triggered by another thread within the same set. These events primarily involve resource acquisition and release. The resources involved are usually logical, such as mutex locks, semaphores, and files. However, deadlocks can also occur due to other types of events, such as reading from a network interface or utilizing inter-process communication (IPC) facilities.

**Deadlock in Multithreaded Application**

The `**pthread\_mutex\_init** **()** ` function initializes an unlocked mutex, and mutex locks are acquired and released using `**pthread\_mutex\_lock ()** ` and `**pthread\_mutex\_unlock ()** `, respectively. If a thread attempts to acquire a locked mutex, it gets blocked until the owner of the mutex lock releases it with `**pthread\_mutex\_unlock ()** `.



In the given code example, two mutex locks, `*first\_mutex*` and `*second\_mutex*`, are created and initialized. Then, two threads, named `*thread\_one*` and `*thread\_two*`, are created. Both threads have access to both mutex locks. `*thread\_one*` executes `**do\_work\_one ()** `, while `*thread\_two*` executes `**do\_work\_two ()** `.



In the code, `thread\_one` tries to acquire the mutex locks in the order `first\_mutex`, followed by `second\_mutex`, while `thread\_two` attempts to acquire them in the reverse order. This situation can lead to deadlock if `thread\_one` **locks** `first\_mutex` while `thread\_two` **locks** `second\_mutex`.

However, whether deadlock occurs depends on the timing and scheduling of threads by the CPU scheduler. If `thread\_one` acquires and releases both mutex locks before `thread\_two` attempts to acquire them, deadlock won't occur.

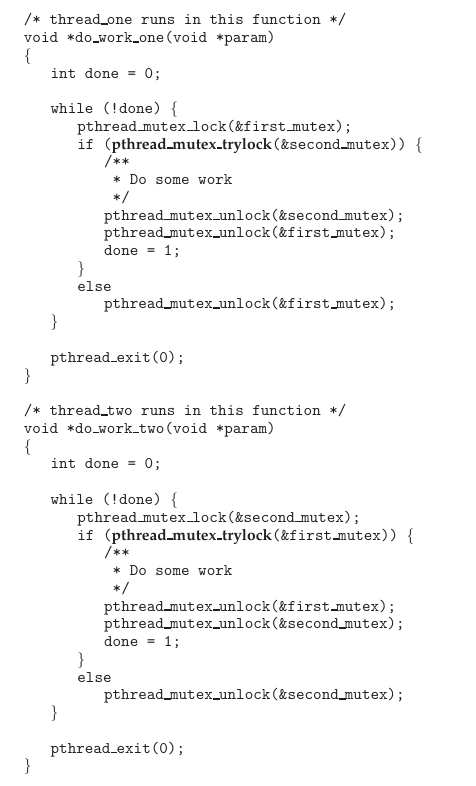
Identifying and testing for deadlocks that arise under specific scheduling circumstances can be challenging.

**Livelock**

Livelock, like deadlock, is a problem that affects the progress of threads in a system. However, in livelock, threads are continuously active but unable to make progress due to the actions of other threads. Unlike deadlock, where threads are blocked waiting for an event that can only be caused by another thread in the set, livelock happens when a thread repeatedly tries an action that keeps failing, preventing progress.

Livelock can be demonstrated using the Pthreads `pthread\_mutex\_trylock () ` function, which tries to acquire a mutex lock without waiting. In the provided code example, both threads attempt to acquire mutex locks in a specific order. However, if one thread acquires a lock before the other thread, and then both threads repeatedly attempt to acquire locks using `pthread\_mutex\_trylock () `, they may fail continuously, releasing and trying to acquire locks indefinitely, causing livelock.

Livelock often arises when multiple threads simultaneously retry failing operations, leading to a continuous cycle of retries. This situation can be mitigated by introducing randomness into the retry timings for each thread, thereby reducing the likelihood of simultaneous retries. While less common than deadlock, livelock remains a challenging issue in concurrent application design, as it may only manifest under certain scheduling conditions.



Above code illustrates livelock. If thread one acquires first mutex and thread two acquires second mutex, both will try to acquire the other mutex without blocking which fails and they release their locks. This cycle will go on indefinitely causing livelock.

**Deadlock Characterization**

**Necessary Conditions**

If the following four conditions hold simultaneously in our system, deadlock will arise:

1. **Mutual** **Exclusion**: At least one resource must be held in a non-sharable mode, meaning only one thread can use it at a time. If another thread wants to use the resource, it must wait until it's released.

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

3. **No** **Preemption**: Resources cannot be forcibly taken away from the thread holding them; they can only be released voluntarily after the thread has finished using them.

4. **Circular Wait**: There must be a circular chain of threads waiting for resources, where each thread in the chain is waiting for a resource held by the next thread in the chain, ultimately leading back to the first thread.

**Resource-Allocation Graph**

Deadlocks can be precisely described using a directed graph called a system resource-allocation graph. This graph consists of vertices V and edges E, where V is divided into two types:

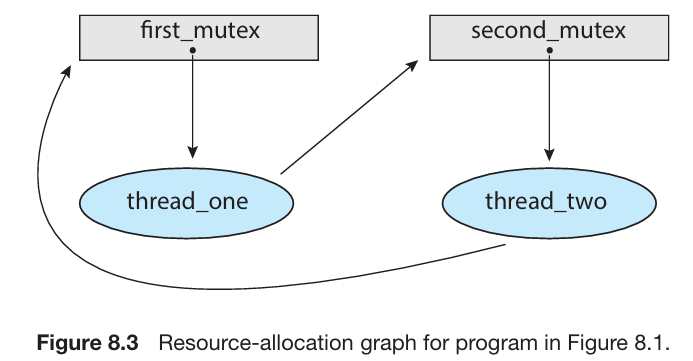
1. (T = {T1, T2... Tn}: representing all active threads in the system.

2. R = {R1, R2... Rm}: representing all resource types in the system.

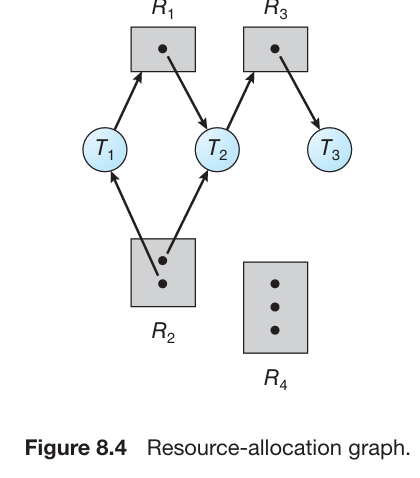
A directed edge from thread Ti to resource type Rj is denoted by Ti -> Rj, indicating that thread Ti has requested an instance of resource type Rj and is waiting for it. Conversely, a directed edge from resource type Rj to thread Ti is denoted by Rj -> Ti, indicating that an instance of resource type Rj has been allocated to thread Ti.

In graphical representation, each thread Ti is depicted as a circle, and each resource type Rj is depicted as a rectangle. If a resource type Rj has multiple instances, each instance is represented by a dot within the rectangle. Request edges point only to the rectangle Rj, while assignment edges also designate one of the dots within the rectangle.

As an example, consider the deadlock situation depicted in Figure 8.3, illustrating the resource allocation graph corresponding to the program scenario described in Figure 8.1.



When thread Ti requests an instance of resource type Rj, a request edge is added to the resource-allocation graph. If the request can be satisfied, the request edge is immediately converted into an assignment edge, indicating that the resource has been allocated to the thread. When the thread finishes using the resource and releases it, the assignment edge is removed from the graph.



The graph above depicts the following situation:

* T = {T1, T2, T3}
* R = {R1,R2, R3, R4}
* E = {T1 → R1,T2 → R3,R1 → T2,R2 → T2,R2 → T1,R3 → T3}

Resource instances:

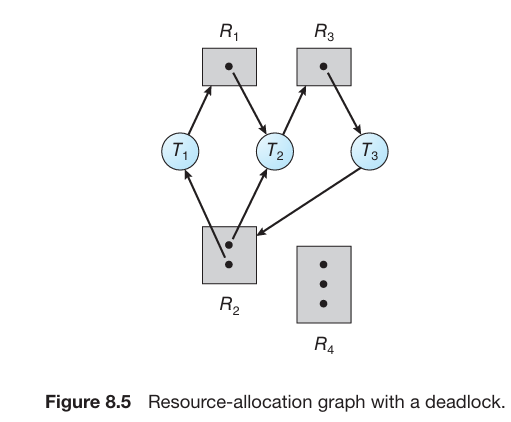
* One instance of resource type R1
* Two instances of resource type R2
* One instance of resource type R3
* Three instances of resource type R

Thread States:

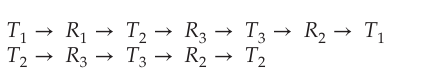
* Thread T1: Holding an instance of resource type R2 and waiting for an instance of resource type R1.
* Thread T2: Holding an instance of R1 and an instance of R2 and waiting for an instance of R3.
* Thread T3: Holding an instance of R3.

In a resource-allocation graph:

* If there are no cycles, then no deadlock exists among the threads.
* If a cycle exists, deadlock might occur.
* If each resource type has only one instance, a cycle guarantees deadlock.
* If each resource type has multiple instances, a cycle might not lead to deadlock, but it indicates a potential for deadlock.



Consider the above graph. It shows a deadlock in the system. At this point there are two cycles:



Threads T1, T2, and T3 are deadlocked. Thread T2 is waiting for the resource R3, which is held by thread T3.Thread T3 is waiting for either thread T1 or thread T2 to release resource R2. In addition, thread T1 is waiting for thread T2 to release resource R1.

**Methods for Handling Deadlocks**

To address the deadlock problem:

1. **Ignoring Deadlocks**: Act as if deadlocks never occur, which may lead to system hang-ups.

2. **Prevention/Avoidance**: Implement protocols to prevent or avoid deadlocks from occurring in the first place, ensuring the system never reaches a deadlocked state.

3. **Detection and Recovery**: Allow the system to potentially enter a deadlock, but implement mechanisms to detect it and recover from it once it occurs.

In most operating systems, including Linux and Windows, the prevailing approach to dealing with deadlocks is to simply ignore the problem. It falls upon kernel and application developers to address deadlocks within their software. They typically employ strategies outlined in the second solution, focusing on prevention or avoidance of deadlocks. However, some specialized systems, such as databases, opt for a different approach. They allow deadlocks to occur but implement mechanisms for detection and recovery, constituting the third solution.

To prevent deadlocks, the system can employ either **deadlock** **prevention** or a deadlock avoidance scheme. Deadlock prevention involves implementing methods that ensure at least one of the necessary conditions for deadlocks cannot occur. These methods effectively constrain how requests for resources are made, thus preventing deadlocks from happening.

**Deadlock avoidance** necessitates providing the operating system with additional information in advance regarding which resources a thread will request and use throughout its lifetime. Armed with this foresight, the operating system can evaluate each request to determine whether the thread should wait or proceed. This decision-making process involves considering the currently available resources, the resources currently allocated to each thread, and the anticipated future requests and releases of each thread.

In a system without deadlock prevention or avoidance mechanisms, deadlocks may occur. In such cases, the system can implement algorithms to assess the system's state to detect deadlock occurrences and subsequently employ recovery algorithms to resolve them.

Without mechanisms to detect and recover from deadlocks, the system may become deadlocked without any means of recognizing the situation. This undetected deadlock leads to deteriorating system performance as resources are held by threads unable to progress, and more threads become deadlocked upon requesting resources. Ultimately, the system will cease functioning, requiring manual intervention to restart.

Despite its apparent drawbacks, the approach of ignoring the deadlock problem is commonly adopted in most operating systems due to cost considerations. It is less expensive to overlook the possibility of deadlocks compared to implementing prevention, avoidance, or recovery mechanisms. Additionally, deadlocks occur relatively infrequently in many systems, perhaps only once per month, making the investment in other methods seem less justified.

Furthermore, recovery methods designed for addressing other liveness issues, like livelock, can sometimes be repurposed for deadlock recovery. There are situations where a system experiences a liveness failure but isn't necessarily deadlocked. For instance, in a scenario where a real-time thread runs at the highest priority or under a non-preemptive scheduler and never relinquishes control to the operating system. In such cases, manual recovery methods are necessary, and these techniques may also be applied for recovering from deadlocks.