# Synchronization

OPERATING SYSTEMS

Sercan Külcü | Operating Systems | 16.04.2023

# Contents

# Chapter 6: Synchronization

## 1 Introduction

Welcome to the chapter on synchronization in multi-threaded and multi-process environments. In this chapter, we will explore the importance of synchronization, which is a critical aspect of operating systems that ensure the proper execution of concurrent programs.

Synchronization refers to the coordination of activities between two or more processes or threads to ensure that they execute in a mutually exclusive and orderly manner. It plays a vital role in ensuring that concurrent programs execute correctly and efficiently, without interfering with one another.

In a multi-threaded or multi-process environment, synchronization is essential to prevent race conditions, deadlocks, and other issues that can arise when multiple threads or processes access the same shared resources concurrently. Without proper synchronization, these issues can lead to unpredictable behavior, data corruption, and other serious problems that can affect the stability and reliability of the system.

The primary goal of synchronization is to ensure the correct and efficient execution of concurrent programs by preventing conflicts and ensuring that threads or processes access shared resources in a mutually exclusive and orderly fashion. This chapter will provide an overview of the different synchronization mechanisms available in operating systems, their strengths and weaknesses, and how they can be used to achieve the synchronization goals.

## 1.1 Definition of synchronization

Synchronization is an essential concept in operating systems that ensures the proper execution and coordination of multiple concurrent processes and threads. In computer science, synchronization is the process of coordinating the execution of multiple threads or processes so that they access shared resources in a mutually exclusive manner.

In simple terms, synchronization refers to the coordination of events to ensure that they occur in a specific order. It ensures that threads or processes do not interfere with each other when accessing shared resources like variables, files, and databases, among others. Without synchronization, concurrent processes or threads may access shared resources simultaneously, leading to unpredictable and undesirable outcomes.

In modern operating systems, synchronization is a fundamental concept in multi-threaded and multi-process environments. Synchronization enables the efficient sharing of resources among concurrent threads and processes while ensuring that each thread or process accesses the resources in a mutually exclusive manner.

There are different synchronization mechanisms that operating systems use to coordinate concurrent threads or processes. These mechanisms include locks, semaphores, monitors, and barriers, among others. These mechanisms ensure that only one thread or process accesses a shared resource at a time, preventing race conditions and other concurrency-related problems.

In conclusion, synchronization is a fundamental concept in operating systems that ensures the proper execution and coordination of concurrent processes and threads. It enables the efficient sharing of resources among concurrent threads and processes while ensuring that each thread or process accesses the resources in a mutually exclusive manner.

## 1.2 Importance of synchronization in multi-threaded and multi-process environments

In today's world, multi-threaded and multi-process environments are ubiquitous, with many software applications taking advantage of the processing power of modern hardware by breaking tasks into smaller, parallelizable subtasks that can be executed simultaneously. However, managing such concurrent executions can be a challenging task. This is where synchronization comes into play.

Synchronization refers to the coordination of multiple concurrent executions to ensure that they proceed correctly without interfering with each other. In multi-threaded or multi-process environments, synchronization is critical to ensure that threads or processes do not interfere with each other's shared resources, leading to race conditions, deadlocks, or livelocks.

Consider, for example, a banking application that processes deposit and withdrawal requests concurrently. Without proper synchronization, it is possible for two threads to access the same account simultaneously, leading to incorrect balances or lost transactions. Another example is a web server that handles multiple requests concurrently. Without synchronization, it is possible for two or more threads to write to the same file simultaneously, leading to data corruption or inconsistency.

**Example:** Here's an example of pseudocode for a banking application that processes deposit and withdrawal requests concurrently:

```
class BankAccount:

    def __init__(self, account_number, balance):

        self.account_number = account_number

        self.balance = balance
```

```
    def deposit(self, amount):
        # Lock the account before making any changes to the balance
        lock.acquire()
        self.balance += amount
        # Release the lock after the changes have been made
        lock.release()

    def withdraw(self, amount):
        # Lock the account before making any changes to the balance
        lock.acquire()
        if self.balance >= amount:
            self.balance -= amount
        # Release the lock after the changes have been made
        lock.release()

# Create an instance of BankAccount with an initial balance of 0
account = BankAccount("123456789", 0)

# Create a lock to prevent multiple threads from accessing the account at the same time
lock = threading.Lock()

# Define a function for making deposits
def make_deposit(amount):
    account.deposit(amount)
```

```
    print(f"Deposit of {amount} was successful. New balance is {account.balance}")


# Define a function for making withdrawals

def make_withdrawal(amount):

    account.withdraw(amount)

    print(f"Withdrawal of {amount} was successful. New balance is {account.balance}")


# Create two threads for making deposits and withdrawals concurrently

deposit_thread = threading.Thread(target=make_deposit, args=(500,))

withdrawal_thread = threading.Thread(target=make_withdrawal, args=(200,))


# Start the threads

deposit_thread.start()

withdrawal_thread.start()


# Wait for the threads to finish before exiting the program

deposit_thread.join()

withdrawal_thread.join()
```

In this pseudocode, we define a BankAccount class that represents a bank account with an account number (account_number) and a current balance (balance). The class has two methods, deposit and withdraw, for making deposits and withdrawals to the account. Both methods use a lock object to ensure that only one thread can access the account at a time, preventing any conflicts that could arise from concurrent access.

We also define two functions, make_deposit and make_withdrawal, that create deposit and withdrawal requests, respectively. These functions create new threads for processing the requests concurrently. Finally, we start the threads and wait for them to finish before exiting the program.

In practice, a real-world banking application would need to be much more complex than this example, with additional functionality for managing customer accounts, handling transactions, and providing security measures to prevent fraud and unauthorized access.

In short, the importance of synchronization in multi-threaded and multi-process environments cannot be overstated. It is essential to ensure correct, consistent, and predictable behavior of concurrent executions.

In the next sections, we will discuss the various synchronization mechanisms and techniques that are commonly used in operating systems and programming languages to achieve proper synchronization in multi-threaded and multi-process environments.

## 1.3 Overview of the goals of synchronization

Synchronization is a fundamental concept in computer science, especially in operating systems, where it plays a crucial role in ensuring that multiple processes or threads access shared resources in a safe and orderly manner. In this chapter, we will provide an overview of the goals of synchronization and how they are achieved in various synchronization mechanisms.

The primary goal of synchronization is to prevent concurrent access to shared resources from causing unexpected or inconsistent behavior. This can occur when multiple processes or threads attempt to modify a shared resource simultaneously. The result can be unpredictable, and in

the worst case, it can lead to data corruption or program crashes. To prevent such situations, synchronization mechanisms are used to ensure that only one process or thread accesses the shared resource at a time.

The second goal of synchronization is to ensure fairness in the allocation of resources. In a multi-threaded or multi-process environment, it is possible for one process or thread to monopolize a shared resource, leading to starvation of other processes or threads that require access to the same resource. To prevent this, synchronization mechanisms are used to ensure that each process or thread gets a fair share of the resource.

The third goal of synchronization is to avoid deadlocks, livelocks, and other concurrency-related problems. Deadlocks occur when two or more processes or threads are blocked, waiting for each other to release a resource that they are holding. Livelocks occur when processes or threads repeatedly change their state without making progress towards completing their task. To avoid these problems, synchronization mechanisms are designed to ensure that processes or threads can access shared resources without getting blocked indefinitely.

Finally, the fourth goal of synchronization is to maximize concurrency and performance. In a multi-threaded or multi-process environment, synchronization mechanisms can impose overhead and reduce the degree of parallelism, resulting in reduced performance. Therefore, synchronization mechanisms should be designed to minimize overhead and maximize concurrency wherever possible.

In summary, synchronization is a crucial concept in multi-threaded and multi-process environments, and it helps prevent data inconsistencies, ensure fairness in resource allocation, avoid deadlocks and livelocks, and maximize performance. The next chapter will discuss the concept of critical sections and race conditions, which are essential concepts in synchronization.

## 1.4 Classical (IPC) problems

Classical Inter-Process Communication (IPC) problems are well-known synchronization problems that arise in concurrent computing when multiple processes or threads try to access shared resources simultaneously. The classical IPC problems include:

- The Dining Philosophers Problem: This problem involves a group of philosophers sitting at a table with a bowl of rice and chopsticks in front of each of them. To eat, a philosopher needs two chopsticks, but only one philosopher can use a chopstick at a time. This creates a deadlock situation where all philosophers are waiting for the chopstick held by their neighbor.
- The Producer-Consumer Problem: This problem involves two processes, the producer, and the consumer, who share a common buffer. The producer produces data and puts it into the buffer, while the consumer consumes data from the buffer. The problem arises when the producer tries to put data into a full buffer or when the consumer tries to consume data from an empty buffer.
- The Readers-Writers Problem: This problem involves multiple readers and writers who need to access a shared resource, such as a file or a database. The problem is to ensure that multiple readers can access the resource simultaneously, but only one writer can access it at a time.
- The Sleeping Barber Problem: This problem involves a barber who serves customers in his shop. There is only one barber chair, and the barber must cut the hair of customers who are waiting in a queue. The problem is to ensure that the barber does not cut the hair of a customer who is not in the chair, and that new customers are not turned away when the waiting room is full.
- The Cigarette Smokers Problem: This problem involves three smokers who each have an infinite supply of one of three ingredients needed to make a cigarette: tobacco, paper, and matches. A non-smoking agent places two of the three ingredients

on a table, and the smoker who has the missing ingredient must pick up the ingredients, make a cigarette, and smoke it. The problem is to ensure that the smokers do not waste ingredients and that they do not smoke without all three ingredients being present.

Solving these problems requires the use of synchronization techniques such as semaphores, monitors, and mutexes to ensure that processes or threads can access shared resources in a safe and orderly manner.

# 2 Critical Sections and Race Conditions

A critical section is a section of code that accesses shared resources that must be executed atomically. It is essential to ensure that only one process or thread can access these shared resources at a time to avoid any inconsistencies in the shared data.

Race conditions occur when multiple processes or threads try to access and modify shared resources simultaneously. These conditions can lead to unexpected results, such as data corruption or program crashes.

Therefore, it is crucial to understand and properly manage critical sections and race conditions in multi-threaded and multi-process environments to ensure the correctness and consistency of the program.

In this chapter, we will define critical sections and race conditions, discuss their importance in synchronization, and explore the consequences of race conditions. Additionally, we will cover some techniques used to prevent race conditions and ensure proper synchronization.

## 2.1 Definition of critical sections

In multi-threaded and multi-process environments, critical sections play a vital role in ensuring the correctness and consistency of shared resources. A critical section is a section of code in which a thread or a process accesses a shared resource, such as a variable, a file, or a network connection, that is also being accessed by other threads or processes.

In such situations, it is essential to ensure that only one thread or process can access the shared resource at a time. This is achieved through synchronization mechanisms, such as locks, semaphores, and monitors, which provide mutual exclusion and prevent multiple threads or processes from accessing the shared resource simultaneously.

The importance of critical sections lies in the fact that without proper synchronization, race conditions may occur, which can lead to unpredictable and erroneous behavior of the program. Therefore, it is crucial to identify the critical sections in the code and protect them with appropriate synchronization mechanisms to ensure the correct and consistent behavior of the program.

In general, a critical section consists of three parts: entry section, critical section, and exit section. The entry section is the code that performs the synchronization mechanism, such as acquiring a lock, before entering the critical section. The critical section is the code that accesses the shared resource and must be protected from concurrent access by other threads or processes. The exit section is the code that releases the synchronization mechanism, such as releasing the lock, after exiting the critical section.

In summary, critical sections are an essential concept in multi-threaded and multi-process programming, and they require proper synchronization mechanisms to ensure the correctness and consistency of shared resources. In the next chapter, we will discuss the importance of critical sections in synchronization.

**Example:** Here's a simple pseudocode example that demonstrates the concept of a critical section:

```
shared_variable = 0


# Function to increment the shared variable in a critical section

def increment_shared_variable():

    global shared_variable

    lock.acquire() # Acquire lock to enter critical section

    temp = shared_variable

    temp = temp + 1

    shared_variable = temp

    lock.release() # Release lock to exit critical section


# Create two threads that will both try to increment the shared
variable

Thread1 = create_thread(increment_shared_variable)

Thread2 = create_thread(increment_shared_variable)


# Create a lock to ensure only one thread can enter the critical
section at a time

lock = create_lock()


# Start both threads

Thread1.start()

Thread2.start()
```

```
# Wait for both threads to finish

Thread1.join()

Thread2.join()


# The final value of the shared variable will be correct because
of the use of a critical section

print("Final value of shared variable: ", shared_variable)
```

In this example, we have a shared variable that is initially set to 0. We then define a function that will increment this shared variable by one, but we include a lock to ensure that only one thread can enter the critical section at a time.

We create two threads that will both call this function to increment the shared variable. The threads are started and allowed to run concurrently. When a thread enters the increment_shared_variable function, it will acquire the lock, enter the critical section, perform the increment, and then release the lock to exit the critical section.

Because only one thread can enter the critical section at a time, we can ensure that the shared variable is incremented correctly each time. This is an example of a critical section, where a section of code that accesses shared resources is protected by a synchronization mechanism to ensure that only one thread can execute it at a time.

It's important to note that critical sections are just one way of ensuring thread safety in concurrent programs. Other synchronization mechanisms such as semaphores and monitors can also be used to ensure that threads can access shared resources without causing race conditions or other concurrency issues.

## 2.2 Importance of critical sections in synchronization

Synchronization is a crucial concept in operating systems that aims to coordinate the activities of multiple threads or processes to ensure that they do not interfere with each other's execution. One of the key goals of synchronization is to protect critical sections of code, which are sections that access shared resources such as variables, files, or devices. In this chapter, we will discuss the importance of critical sections in synchronization.

### 2.2.1 Protecting Shared Resources:

In a multi-threaded or multi-process environment, critical sections of code need to be protected to ensure that only one thread or process accesses them at a time. This is necessary to prevent race conditions, where two or more threads or processes try to access or modify a shared resource simultaneously, leading to unpredictable behavior or incorrect results.

For example, consider two threads that access the same shared variable. If both threads try to modify the variable simultaneously, the final value of the variable will depend on the order in which the threads execute, leading to inconsistent results. To prevent such issues, we need to synchronize access to the shared variable by protecting the critical section of code that accesses it.

### 2.2.2 Ensuring Mutual Exclusion:

One of the primary goals of protecting critical sections is to ensure mutual exclusion, which means that only one thread or process can execute the critical section at a time. This is usually achieved using synchronization mechanisms such as locks, semaphores, or monitors. When a thread or process acquires a lock or semaphore, it prevents

other threads or processes from acquiring it, ensuring that only the thread or process holding the lock can execute the critical section.

### 2.2.3 Preventing Deadlocks:

Another important goal of protecting critical sections is to prevent deadlocks, which occur when two or more threads or processes are blocked, waiting for resources held by each other. Deadlocks can lead to a system freeze or crash, and it is essential to prevent them by carefully designing synchronization mechanisms.

### 2.2.4 Improving System Performance:

Protecting critical sections can also help improve system performance by reducing the number of context switches between threads or processes. Context switching is the process of saving the state of one thread or process and restoring the state of another thread or process to allow it to execute. Context switching is an expensive operation, and reducing the number of context switches can improve system performance.

In conclusion, protecting critical sections is a crucial aspect of synchronization in multi-threaded and multi-process environments. It ensures mutual exclusion, prevents deadlocks, and improves system performance. Synchronization mechanisms such as locks, semaphores, and monitors are used to protect critical sections and coordinate the activities of multiple threads or processes. It is essential to carefully design synchronization mechanisms to ensure that they are deadlock-free and do not cause unnecessary context switches.

## 2.3 Definition of race conditions

Race conditions are one of the most common issues encountered in multi-threaded and multi-process environments. In simple terms, a race

condition occurs when two or more threads or processes access a shared resource in an unexpected order, which leads to unpredictable behavior and incorrect results. This can cause a wide range of problems, including data corruption, deadlock, and program crashes. Therefore, it is crucial to understand the definition of race conditions to prevent these issues from occurring.

A race condition occurs when two or more threads or processes access a shared resource simultaneously and modify its value. The result of this modification depends on the order in which the threads or processes execute. In other words, the outcome of the program is dependent on the race to access the shared resource. The term "race" comes from the idea that the threads or processes are competing to access the resource first, just like in a race.

For example, suppose two threads access a shared variable named x and increment its value. If thread 1 increments the value of x and then thread 2 increments the same variable, the final value of x will be incremented by 2. However, if the threads execute in reverse order, the final value of x will only be incremented by 1. This results in a data inconsistency, where the value of x depends on the order in which the threads execute.

**Example:** Here's an example of pseudocode for two threads that access a shared variable named x and increment its value:

```
# Define a shared variable

x = 0


# Define a function that increments x

def increment_x():

    global x

    # Increment x

    x += 1
```

```
# Create two threads that will access x
thread1 = threading.Thread(target=increment_x)
thread2 = threading.Thread(target=increment_x)

# Start the threads
thread1.start()
thread2.start()

# Wait for the threads to finish before continuing
thread1.join()
thread2.join()

# Print the final value of x
print("Final value of x:", x)
```

In this pseudocode, we define a shared variable x and two threads (thread1 and thread2) that both call a function named increment_x, which simply increments the value of x by 1.

If both threads execute sequentially, the final value of x will be incremented by 2 since each thread increments it by 1. However, if the threads execute in reverse order, the final value of x will only be incremented by 1 since the second thread will overwrite the increment made by the first thread.

To ensure that the two threads execute concurrently, we start them using the start method and wait for them to finish before printing the final value of x.

It's important to note that race conditions can occur in any situation where multiple threads or processes access shared resources without proper synchronization mechanisms in place.

Race conditions can also occur when threads or processes access shared resources that are not designed to handle concurrent access. For example, if two threads attempt to write to the same file simultaneously, it can result in data corruption or lost data. Similarly, if two threads attempt to access the same database record simultaneously, it can cause incorrect data retrieval or update.

In summary, a race condition occurs when multiple threads or processes access a shared resource simultaneously and modify its value, resulting in an unpredictable outcome. It is important to understand the concept of race conditions to prevent issues such as data corruption, program crashes, and deadlock. The next chapter will discuss the consequences of race conditions in more detail.

## 2.4 Consequences of race conditions

Race conditions are a common problem that can occur in multi-threaded and multi-process environments when two or more threads or processes access the same shared resource or variable simultaneously without proper synchronization. In this chapter, we will discuss the consequences of race conditions and how they can lead to serious issues in a program.

### 2.4.1 Inconsistency

Race conditions can cause inconsistency in data. When two or more threads or processes access the same data simultaneously, the data may be updated in an unpredictable way. This can lead to incorrect values or unexpected behavior in a program.

### 2.4.2 Deadlock

Race conditions can also lead to deadlock, which occurs when two or more threads or processes are waiting for each other to release a resource or lock. In this situation, none of the threads or processes can proceed, and the program may become unresponsive.

### 2.4.3 Performance Issues

Race conditions can also cause performance issues in a program. When multiple threads or processes access the same resource simultaneously, they may need to wait for each other to complete, which can lead to unnecessary delays and reduced performance.

### 2.4.4 Security Issues

Race conditions can also pose a security risk in a program. An attacker can exploit race conditions to gain access to sensitive data or to execute malicious code.

### 2.4.5 Unexpected Behavior

Race conditions can cause unexpected behavior in a program. For example, a race condition may cause a program to crash or produce incorrect results.

### 2.4.6 Debugging Challenges

Race conditions can be difficult to detect and debug. Since the behavior of a program with a race condition is unpredictable, it may be difficult to reproduce the issue and identify the root cause.

In conclusion, race conditions can lead to serious issues in a program, including inconsistency, deadlock, performance issues, security risks, unexpected behavior, and debugging challenges. Therefore, it is

important to properly synchronize access to shared resources and variables in multi-threaded and multi-process environments to prevent race conditions from occurring.

# 3 Synchronization Mechanisms

In a multi-threaded or multi-process environment, ensuring proper synchronization is critical to prevent race conditions and other issues that can arise from concurrent access to shared resources. In this chapter, we will explore various mechanisms that can be used to achieve synchronization, including locks (such as mutual exclusion locks, recursive locks, and read-write locks), semaphores, monitors, and barriers. We will also compare the strengths and weaknesses of each mechanism and explore situations where one might be more appropriate than another. By the end of this chapter, you should have a clear understanding of the different synchronization mechanisms available and their respective applications.

## 3.1 Locks

Locks are synchronization primitives that are used to protect shared resources in a multi-threaded or multi-process environment. A lock allows only one thread or process to access a shared resource at any given time. In this chapter, we will discuss different types of locks such as mutual exclusion locks, recursive locks, and read-write locks.

### 3.1.1 Mutual Exclusion Locks:

Mutual exclusion locks, commonly known as mutexes, are a widely used synchronization mechanism in computer systems. Mutexes are used to provide mutual exclusion to shared resources to ensure that only one

thread or process can access the resource at any given time. In this chapter, we will discuss mutexes in detail.

A mutex lock has two states: locked and unlocked. A thread or process that wants to access a shared resource acquires a mutex lock by calling a lock() function. If the lock is already acquired by another thread or process, the requesting thread or process will be blocked until the lock is released. Once the lock is acquired, the thread or process has exclusive access to the shared resource. It can read, write or modify the shared resource without interference from other threads or processes.

When the thread or process has finished using the shared resource, it releases the lock by calling the unlock() function. This allows other threads or processes to acquire the lock and access the shared resource.

Mutexes are often used in multi-threaded and multi-process environments to protect shared resources such as critical sections, data structures, and shared files. For example, in a banking application, a mutex can be used to ensure that only one thread or process can access a customer's bank account information at a time. This prevents race conditions and ensures that the account information is accurate and up-to-date.

One issue with mutexes is that they can lead to deadlocks if not used carefully. A deadlock occurs when two or more threads or processes are waiting for each other to release the mutex lock, resulting in a situation where none of the threads or processes can proceed. To avoid deadlocks, it is important to follow a strict protocol for acquiring and releasing mutex locks.

Mutex locks are implemented using various algorithms such as test-and-set, compare-and-swap, and fetch-and-add, among others. These algorithms ensure that only one thread or process can acquire the lock at any given time, thus preventing concurrent access to the shared resource.

However, using mutex locks comes with its own set of challenges. If a thread or process forgets to release the lock after accessing the shared resource, it can lead to a deadlock or livelock. A deadlock occurs when two or more threads or processes are waiting for each other to release a lock, and none of them can proceed. A livelock occurs when two or more threads or processes keep changing their state or releasing and acquiring locks without making any progress towards completing their tasks.

To prevent deadlocks and livelocks, it is essential to use mutex locks correctly. A thread or process that acquires a lock must release it after accessing the shared resource. Furthermore, the lock should be held for the shortest possible time to reduce the chances of contention and increase the overall throughput of the system.

In addition to mutex locks, other synchronization mechanisms such as semaphores, monitors, and barriers are also used to coordinate the access of shared resources among multiple threads or processes. Each of these mechanisms has its own strengths and weaknesses and is suitable for different types of applications. Therefore, it is crucial to choose the right synchronization mechanism based on the application's requirements.

In conclusion, mutex locks are a simple yet powerful synchronization mechanism that provides mutual exclusion to a shared resource. They prevent concurrent access to the resource and ensure thread safety. However, they should be used carefully to avoid deadlocks and livelocks. It is also essential to choose the right synchronization mechanism based on the application's requirements to ensure optimal performance and reliability.

**Example:** Here's an example Java code that demonstrates the use of mutex to solve the classical producer-consumer problem in inter-process communication:

```
import java.util.concurrent.locks.Condition;
```

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ProducerConsumer {
    private static final int BUFFER_SIZE = 5;
    private final int[] buffer = new int[BUFFER_SIZE];
    private int count = 0;
    private int in = 0;
    private int out = 0;
    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    public void produce(int value) throws InterruptedException {
        lock.lock();
        try {
            while (count == BUFFER_SIZE) {
                notFull.await();
            }
            buffer[in] = value;
            in = (in + 1) % BUFFER_SIZE;
            count++;
            notEmpty.signal();
        } finally {
            lock.unlock();
```

```
        }
    }

    public int consume() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0) {
                notEmpty.await();
            }
            int value = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count--;
            notFull.signal();
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

In this code, we use a mutex lock (ReentrantLock) to synchronize access to the shared buffer between the producer and consumer threads. The Condition objects (notFull and notEmpty) are used to signal when the buffer is full or empty, respectively, and the threads need to wait for the opposite condition to occur before they can proceed.

The produce method waits for the notFull condition to be signaled, indicating that there is space in the buffer for a new item to be produced. Once it acquires the lock, it adds the item to the buffer, updates the

index and count variables, and signals the notEmpty condition, indicating that there is now at least one item in the buffer that can be consumed.

The consume method waits for the notEmpty condition to be signaled, indicating that there is at least one item in the buffer that can be consumed. Once it acquires the lock, it removes the item from the buffer, updates the index and count variables, and signals the notFull condition, indicating that there is now at least one free slot in the buffer that can be filled with a new item.

Using this mutex-based solution ensures that only one thread can access the buffer at any given time, preventing race conditions and ensuring that the producer and consumer threads can safely and correctly access the shared resource.

### 3.1.2 Recursive Locks:

In operating systems, recursive locks, also known as re-entrant locks, are a type of lock that allows a thread or process to acquire the same lock multiple times without causing a deadlock. Recursive locks keep track of the number of times a lock has been acquired by a thread or process. When a thread or process acquires a recursive lock for the first time, it works like a normal mutex lock. However, if the same thread or process tries to acquire the same lock again, it does not block itself but increments the lock count.

Recursive locks are useful in situations where a function or method requires access to a shared resource multiple times. For example, a recursive function that traverses a tree structure may require multiple accesses to a shared resource, such as a node in the tree. Recursive locks ensure that the function can acquire the lock multiple times without causing a deadlock.

The implementation of recursive locks is similar to that of mutex locks. A recursive lock can be implemented using test-and-set, compare-and-

swap, or fetch-and-add algorithms. However, recursive locks require additional bookkeeping to keep track of the number of times the lock has been acquired. When a thread or process acquires the lock for the first time, the lock count is set to one. If the same thread or process acquires the lock again, the lock count is incremented by one. When the thread or process releases the lock, the lock count is decremented. The lock is fully released only when the lock count reaches zero.

Recursive locks are not without their drawbacks. The main disadvantage of recursive locks is that they are slower than normal mutex locks. The additional bookkeeping required to keep track of the lock count can add overhead to the lock acquisition and release operations. Furthermore, recursive locks can make code more complex and harder to debug, especially when dealing with recursive functions.

**Example:** Here's an example Java code that demonstrates Recursive Locks by solving the classical IPC problem of the Dining Philosophers:

```
import java.util.concurrent.locks.ReentrantLock;


public class DiningPhilosophers {

    private static final int NUM_PHILOSOPHERS = 5;


    public static void main(String[] args) throws
InterruptedException {

        ReentrantLock[] forks = new
ReentrantLock[NUM_PHILOSOPHERS];


        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {

            forks[i] = new ReentrantLock();

        }
```

```
        Thread[] philosophers = new Thread[NUM_PHILOSOPHERS];

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            philosophers[i] = new Thread(new Philosopher(i, forks[i], forks[(i + 1) % NUM_PHILOSOPHERS]));
            philosophers[i].start();
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            philosophers[i].join();
        }
    }

    private static class Philosopher implements Runnable {
        private final int id;
        private final ReentrantLock leftFork;
        private final ReentrantLock rightFork;

        public Philosopher(int id, ReentrantLock leftFork, ReentrantLock rightFork) {
            this.id = id;
            this.leftFork = leftFork;
            this.rightFork = rightFork;
        }

        @Override
```

```
        public void run() {
            for (int i = 0; i < 5; i++) {
                leftFork.lock();
                System.out.println("Philosopher " + id + " picked
up left fork.");
                rightFork.lock();
                System.out.println("Philosopher " + id + " picked
up right fork and is eating.");
                rightFork.unlock();
                System.out.println("Philosopher " + id + " put down
right fork.");
                leftFork.unlock();
                System.out.println("Philosopher " + id + " put down
left fork and is thinking.");
            }
        }
    }
}
```

In this code, each philosopher is represented by a thread, and each fork is represented by a ReentrantLock object. The Philosopher class has a run method that implements the logic for each philosopher. In this implementation, each philosopher tries to acquire the left fork first, and then the right fork. If a fork is already held by another philosopher, the current philosopher will wait until it becomes available.

By using ReentrantLock objects for the forks, we can implement recursive locking. If a philosopher needs to pick up the same fork multiple times, the ReentrantLock object will allow it to do so without causing a deadlock. This allows us to solve the Dining Philosophers problem without the risk of deadlocks caused by the use of mutex locks.

### 3.1.3 Read-Write Locks:

Read-write locks are an important synchronization mechanism used in multi-threaded or multi-process programs to improve performance. In this chapter, we will discuss read-write locks and how they work.

A read-write lock provides two types of locks - read lock and write lock. A read lock allows multiple threads or processes to read a shared resource simultaneously, while a write lock allows only one thread or process to write to the shared resource at a time. When a thread or process wants to access a shared resource, it first acquires the appropriate lock.

Multiple threads or processes can acquire a read lock simultaneously. This is because reading from a shared resource does not modify its state, and hence does not affect other readers. However, if a thread or process wants to write to a shared resource, it has to acquire a write lock. When a thread or process acquires a write lock, it ensures that no other thread or process can acquire a read or write lock until the lock is released.

Read-write locks are useful in situations where a shared resource is mostly read and rarely written. By allowing multiple threads or processes to read the shared resource simultaneously, read-write locks can improve the performance of the program. However, if a shared resource is mostly written, read-write locks can cause contention and degrade the performance of the program.

Read-write locks can be implemented using various algorithms, such as the readers-writers problem, which provides a solution to the problem of multiple readers and writers accessing a shared resource. The readers-writers problem ensures that multiple readers can access a shared resource simultaneously, while a writer has exclusive access to the resource.

In summary, read-write locks are an important synchronization mechanism that allows multiple threads or processes to read a shared

resource simultaneously, while ensuring that only one thread or process can write to the shared resource at a time. Read-write locks can improve the performance of multi-threaded or multi-process programs, but their effectiveness depends on the nature of the shared resource.

**Example:** Here's an example Java code that demonstrates the use of read-write locks to solve the classical IPC problem of readers and writers:

```
import java.util.concurrent.locks.*;


public class ReaderWriterProblem {

    private static final int NUM_READERS = 5;

    private static final int NUM_WRITERS = 2;


    private     static     ReadWriteLock     lock     =     new
ReentrantReadWriteLock();

    private static String sharedResource = "";


    private static class Reader implements Runnable {

        private int id;


        public Reader(int id) {

            this.id = id;

        }


        public void run() {

            while (true) {

                lock.readLock().lock();
```

```
                System.out.println("Reader " + id + " read: " +
sharedResource);
                lock.readLock().unlock();

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    private static class Writer implements Runnable {
        private int id;

        public Writer(int id) {
            this.id = id;
        }

        public void run() {
            while (true) {
                lock.writeLock().lock();
                sharedResource = "Written by writer " + id;
                System.out.println("Writer " + id + " wrote: " +
sharedResource);
```

```
                lock.writeLock().unlock();

                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < NUM_READERS; i++) {
            new Thread(new Reader(i)).start();
        }

        for (int i = 0; i < NUM_WRITERS; i++) {
            new Thread(new Writer(i)).start();
        }
    }
}
```

In this example, we have a shared resource (a string) that can be read by multiple readers simultaneously, but can only be written to by one writer at a time. We use a ReentrantReadWriteLock to implement the read-write lock.

The Reader class acquires a read lock before reading the shared resource, and releases the lock afterwards. The Writer class acquires a write lock before writing to the shared resource, and releases the lock afterwards.

When we run the program, we create multiple reader and writer threads, and they access the shared resource concurrently. The read-write lock ensures that multiple readers can access the shared resource simultaneously, but only one writer can access the shared resource at a time.

### 3.1.4 Spin Locks:

Spin locks are a type of synchronization mechanism that uses busy-waiting to achieve synchronization. They are often used in environments where locking time is expected to be short. In a spin lock, a thread repeatedly checks if a lock is available until it can acquire the lock. This is known as busy-waiting because the thread is constantly using the CPU to check if the lock is available.

The advantage of spin locks is that they are very fast when the lock is acquired quickly. In comparison to other locking mechanisms, spin locks require very little overhead and do not require context switching or system calls to acquire a lock. This makes them an efficient solution for low-level synchronization.

However, spin locks are not a good choice for longer locking times. When the lock is held for a long time, the thread using the spin lock will continue to use the CPU, which can lead to inefficient use of resources. Additionally, if many threads are competing for the lock, it can cause contention and waste system resources.

In summary, spin locks are a fast and efficient way of achieving synchronization when locking time is expected to be short. They are most suitable for low-level synchronization, such as within a single process or thread. However, they should be used with caution and other

locking mechanisms should be considered for longer locking times or when there are many threads competing for the lock.

**Example:** The following is an example of a Java code that demonstrates the use of a spin lock:

```
import java.util.concurrent.atomic.AtomicBoolean;


public class SpinLock {
   private AtomicBoolean lock = new AtomicBoolean(false);


   public void acquire() {
      while (!lock.compareAndSet(false, true)) {
         // busy-wait until the lock is available
      }
   }


   public void release() {
      lock.set(false);
   }
}
```

In this example, the AtomicBoolean class is used to implement a spin lock. The acquire() method uses a compareAndSet() method call to repeatedly check if the lock is available. If the lock is not available, the thread will enter a busy-wait loop until it is available. Once the lock is acquired, the release() method is called to release the lock by setting the AtomicBoolean to false.

## 3.2 Semaphores

Semaphores are a synchronization tool that can be used to control access to shared resources in a multi-threaded or multi-process environment. Semaphores are named after the semaphore flags used in maritime signaling, where a semaphore indicates the status of a channel. Similarly, in computer science, semaphores are used to indicate the availability of a shared resource.

A semaphore is essentially a non-negative integer counter that can be accessed atomically by multiple threads or processes. A semaphore can be initialized to a positive integer value, which represents the maximum number of threads or processes that can access the shared resource simultaneously.

A semaphore provides two fundamental operations: wait() and signal(). The wait() operation decrements the value of the semaphore by one, blocking the thread or process if the semaphore value is zero. The signal() operation increments the value of the semaphore by one, allowing other threads or processes to access the shared resource.

One of the main benefits of using semaphores for synchronization is that they can be used to implement other synchronization mechanisms, such as locks and barriers. In fact, a binary semaphore with an initial value of 1 is equivalent to a mutex lock.

There are two types of semaphores: binary and counting. A binary semaphore can only take on two values, 0 and 1, and is typically used to protect a single resource. A counting semaphore can take on any non-negative integer value and is typically used to protect multiple instances of a resource.

Semaphores are a powerful synchronization tool, but they can be difficult to use correctly. One common issue with semaphores is deadlocks, where two or more threads or processes are blocked waiting for each other to release a resource. To prevent deadlocks, it is

important to carefully design the use of semaphores and ensure that threads or processes do not acquire resources in a circular fashion.

**Example:** Here's an example pseudocode for semaphores:

```
// Create a semaphore with an initial value of 1

semaphore s = 1;


// Process A

wait(s); // Decrement semaphore value, blocking if value is 0

// critical section

signal(s); // Increment semaphore value


// Process B

wait(s);

// critical section

signal(s);
```

In this example, s is a semaphore initialized with a value of 1. The wait(s) operation decrements the value of s by 1 and blocks the process if the value becomes 0. The signal(s) operation increments the value of s by 1 and wakes up any blocked processes that were waiting on s.

In the context of synchronization, semaphores are often used to control access to a shared resource. In the example above, the critical section represents code that accesses a shared resource, and the semaphore ensures that only one process can access the critical section at a time.

In summary, semaphores are a powerful synchronization tool that can be used to control access to shared resources in a multi-threaded or multi-process environment. They provide a simple and flexible mechanism for coordinating concurrent access to resources, but they

require careful use to prevent deadlocks and other synchronization issues.

## 3.3 Monitors

Monitors are a synchronization mechanism used in programming languages like Java and C# to ensure mutual exclusion in a thread-safe manner. Monitors provide a high-level abstraction for synchronization by encapsulating shared data and the associated synchronization primitives in a single object. In this chapter, we will discuss monitors in detail and how they help in achieving synchronization.

A monitor is a programming construct that allows threads to access shared data in a mutually exclusive and synchronized manner. It consists of a data structure that holds the shared data and the procedures that operate on that data. The procedures that operate on the shared data are called monitor procedures. The monitor provides a mutual exclusion mechanism that ensures that only one thread can execute a monitor procedure at any given time.

A monitor consists of the following elements:

- The shared data is the data that is accessed by multiple threads in a concurrent program. This data is encapsulated within the monitor, and access to this data is regulated by the monitor procedures.
- Monitor procedures are the methods or functions that operate on the shared data. These procedures can be accessed by multiple threads, but only one thread can execute a monitor procedure at any given time. When a thread executes a monitor procedure, it acquires the monitor's lock, ensuring that no other thread can execute the monitor procedure until the first thread releases the lock.

- Condition variables are synchronization primitives that are used to manage the order of execution of threads waiting for a particular condition. Condition variables allow a thread to wait until a specific condition becomes true. When a thread waits on a condition variable, it releases the monitor's lock, allowing other threads to access the monitor. When the condition becomes true, the waiting thread is signaled, and it reacquires the monitor's lock and resumes execution.

Monitors have several advantages over other synchronization mechanisms, including:

- Simplicity: Monitors provide a simple and intuitive mechanism for synchronization by encapsulating the shared data and the associated synchronization primitives in a single object. This makes it easy to reason about the correctness of a concurrent program.
- Safety: Monitors provide a safe mechanism for synchronization by ensuring that only one thread can execute a monitor procedure at any given time. This prevents race conditions and other synchronization-related bugs.
- Flexibility: Monitors provide flexibility by allowing the programmer to define the synchronization policy for the shared data. This allows the programmer to optimize the synchronization mechanism for the specific requirements of the program.

Monitors also have some disadvantages, including:

- Limited Expressiveness: Monitors are limited in their expressiveness, as they can only be used to synchronize access to shared data within a single process. They cannot be used to synchronize access to shared data across multiple processes.

- Potential Deadlock: Monitors can potentially lead to deadlock if the programmer is not careful when using condition variables. Deadlock occurs when two or more threads are waiting for each other to release the monitor's lock, resulting in a program that is stuck and cannot make progress.

**Example:** Here is an example pseudocode for a monitor that provides mutual exclusion for a shared integer variable count:

```
monitor Counter {
   int count = 0;

   procedure increment() {
      count = count + 1;
   }

   procedure decrement() {
      count = count - 1;
   }
}
```

In this example, the increment() and decrement() procedures operate on the shared count variable. The monitor keyword defines a new monitor called Counter, which encapsulates the shared count variable and the associated monitor procedures.

## 3.4 Barriers

In multi-threaded and multi-process environments, synchronization is crucial for ensuring that concurrent threads and processes coordinate their activities effectively. One of the challenges in synchronization is to ensure that multiple threads or processes reach a particular point in their execution before continuing. This is where barriers come into play. Barriers are synchronization mechanisms that ensure that a group of threads or processes wait until all of them have reached a particular point in their execution before continuing.

This chapter will provide a detailed overview of barriers in the synchronization context. We will start by defining what barriers are and why they are important in synchronization. We will then discuss the different types of barriers and how they work. Finally, we will compare barriers with other synchronization mechanisms and highlight the advantages and disadvantages of using them.

A barrier is a synchronization mechanism that blocks the progress of a group of threads or processes until they all reach a particular point in their execution. Barriers are typically used when a group of threads or processes need to coordinate their activities and must wait until all of them have completed a particular stage of their execution before continuing.

Barriers are essential in synchronization for several reasons. Firstly, they help ensure that all threads or processes complete a particular stage of their execution before continuing. This can be important in situations where one thread or process depends on the results produced by another thread or process. Secondly, barriers can help improve the performance of parallel programs by reducing the amount of idle time spent waiting for threads or processes to synchronize. Finally, barriers can help avoid race conditions and deadlocks that can occur in multi-threaded and multi-process environments.

There are two main types of barriers: centralized barriers and decentralized barriers.

- Centralized barriers rely on a central thread or process to coordinate the synchronization of other threads or processes. This central thread or process is responsible for keeping track of the progress of all the threads or processes and signaling when all of them have reached the synchronization point. The main disadvantage of centralized barriers is that they can become a bottleneck, especially when the number of threads or processes is large.
- Decentralized barriers, on the other hand, do not rely on a central thread or process to coordinate synchronization. Instead, each thread or process is responsible for notifying other threads or processes when it has reached the synchronization point. This approach is more scalable than centralized barriers since there is no single point of failure or bottleneck.

One popular implementation of barriers is the Pthreads barrier. Pthreads barriers are available in most modern operating systems and programming languages, including C and C++. The Pthreads barrier consists of a count variable that is initialized to the number of threads or processes that need to synchronize. When a thread or process reaches the synchronization point, it decrements the count variable. Once the count variable reaches zero, all threads or processes are released, and execution continues.

Another implementation of barriers is the Cyclic Barrier, which is available in Java. The Cyclic Barrier allows a group of threads to wait for each other to reach a synchronization point, and it can be reused after all threads have been released.

Barriers are just one of the many synchronization mechanisms available to developers. Other synchronization mechanisms, such as locks and semaphores, can also be used to coordinate the activities of multiple threads or processes. The advantage of barriers over other

synchronization mechanisms is that they allow multiple threads or processes to synchronize with each other simultaneously. This can result in better performance, especially in situations where there are many threads or processes involved. However, barriers can be less flexible than other synchronization mechanisms and may not be suitable for all situations.

**Example:** Here's some pseudocode for a barrier implementation in a synchronization context:

```
1. Initialize barrier with a count of threads to wait for

    initialize_barrier(int count)

        barrier_count = count

        barrier_current_count = 0

        barrier_mutex = initialize_mutex()

        barrier_condvar = initialize_conditional_variable()


2. Wait for all threads to reach the barrier

    wait_barrier()

        acquire_mutex(barrier_mutex)

        barrier_current_count++

        if barrier_current_count == barrier_count

            signal_all(barrier_condvar)

        else

            wait(barrier_condvar, barrier_mutex)

        release_mutex(barrier_mutex)
```

In this pseudocode, the initialize_barrier function initializes a barrier with a count of threads to wait for. The wait_barrier function is called by each thread to wait for all other threads to reach the barrier. The

acquire_mutex and release_mutex functions are used to acquire and release a mutex to protect shared state, and the wait and signal_all functions are used to wait on and signal a conditional variable, respectively.

When a thread calls wait_barrier, it first acquires the barrier_mutex to protect against race conditions. It then increments the barrier_current_count variable to indicate that it has reached the barrier. If this is the last thread to reach the barrier (i.e., if barrier_current_count == barrier_count), it signals all waiting threads using signal_all(barrier_condvar). Otherwise, the thread waits on the barrier_condvar until signaled by the last thread to reach the barrier. Finally, the thread releases the barrier_mutex.

## 3.5 Comparison of synchronization mechanisms

In the previous chapters, we discussed several synchronization mechanisms, including locks, semaphores, monitors, and barriers. Each mechanism has its strengths and weaknesses, and choosing the appropriate mechanism for a specific application can be challenging. In this chapter, we will compare these synchronization mechanisms based on various criteria to help you choose the right mechanism for your application.

Complexity: One of the most important criteria for selecting a synchronization mechanism is its complexity. Some mechanisms, such as mutual exclusion locks, are relatively simple to use, while others, such as barriers, may be more complex. Semaphores and monitors fall somewhere in between.

Granularity: The granularity of a synchronization mechanism refers to how finely it can control access to shared resources. For example, mutual exclusion locks are typically used to protect a single shared resource, while semaphores can be used to protect multiple resources

simultaneously. Monitors are typically used to protect more complex data structures, such as linked lists or trees.

Performance: Another important criterion for selecting a synchronization mechanism is its performance. Some mechanisms, such as mutual exclusion locks, can be very efficient, while others, such as barriers, can be more expensive in terms of time and resources. In general, simpler mechanisms tend to perform better than more complex ones.

Deadlock and livelock prevention: Deadlocks and livelocks are two common problems that can occur when using synchronization mechanisms. Deadlocks occur when two or more processes are blocked waiting for each other to release resources they are holding. Livelocks occur when two or more processes keep modifying their state without making progress. Some synchronization mechanisms, such as semaphores and monitors, provide built-in support for deadlock prevention, while others, such as mutual exclusion locks, require careful design to avoid deadlocks and livelocks.

Ease of use: Finally, the ease of use of a synchronization mechanism is an important consideration. Some mechanisms, such as mutual exclusion locks, are straightforward to use, while others, such as monitors, can be more complex. Semaphores are somewhere in between, depending on the level of complexity required for the specific application.

In conclusion, the choice of synchronization mechanism depends on several factors, including the complexity of the application, the granularity of the resources being shared, the desired performance, and the need for deadlock and livelock prevention. By considering these factors and comparing the different mechanisms based on them, you can select the synchronization mechanism that is best suited for your application.

# 4 Deadlocks and Livelocks

In multi-threaded and multi-process environments, synchronization is a crucial concept that ensures the proper execution of programs. Synchronization mechanisms help coordinate the execution of multiple threads and processes to prevent conflicts and ensure data consistency.

However, synchronization can lead to issues such as deadlocks and livelocks, which can cause programs to stop functioning properly. It is important for programmers and system designers to understand the causes and prevention of these issues to ensure the reliability of their systems.

This chapter will discuss the concepts of deadlocks and livelocks, including their definitions, causes, and prevention techniques. We will also explore various synchronization mechanisms, such as locks, semaphores, monitors, and barriers, and compare their effectiveness in preventing these issues.

## 4.1 Definition of deadlocks

In a multi-process or multi-threaded environment, deadlocks can occur when two or more processes or threads are waiting for each other to release a resource, resulting in a situation where none of the processes or threads can proceed.

A deadlock is a situation where two or more processes are unable to proceed because each process is waiting for one or more of the others to release resources. The resources may be held by the processes themselves, or they may be external resources such as files, databases, or network connections. Deadlocks can occur when there is a circular chain of resource dependencies between two or more processes, such that each process is waiting for a resource that is held by another process in the chain.

A classic example of a deadlock involves two trains that are traveling towards each other on a single-track railway. If there is only one passing point on the track, and each train must use this passing point to allow the other to pass, then a deadlock can occur if both trains arrive at the passing point simultaneously. If neither train is willing to back up and allow the other to pass first, then both trains will become deadlocked and unable to proceed.

In computer systems, deadlocks can occur when two or more processes or threads are waiting for each other to release a resource, such as a lock on a shared data structure, or access to a shared resource such as a database or network connection. If each process or thread is holding a resource that is required by one or more of the other processes or threads, then a deadlock can occur where none of the processes or threads can proceed.

Deadlocks can be a serious problem in computer systems, particularly in mission-critical applications where a system failure can have catastrophic consequences. Therefore, it is important to design systems in a way that minimizes the risk of deadlocks occurring, and to provide mechanisms for detecting and resolving deadlocks if they do occur.

In the next chapter, we will discuss the causes and prevention of deadlocks in more detail, and explore the various techniques that can be used to detect and resolve deadlocks in computer systems.

**Example:** Here's an example of pseudocode for a possible deadlock situation:

```
Thread A:

    lock Resource 1

    // Do some work with Resource 1

    lock Resource 2

    // Do some work with both resources

    unlock Resource 2
```

```
    unlock Resource 1


Thread B:
    lock Resource 2
    // Do some work with Resource 2
    lock Resource 1
    // Do some work with both resources
    unlock Resource 1
    unlock Resource 2
```

In this example, Thread A acquires a lock on Resource 1 and then tries to acquire a lock on Resource 2. At the same time, Thread B acquires a lock on Resource 2 and then tries to acquire a lock on Resource 1. This creates a potential deadlock situation where both threads are waiting for each other to release the resources they need to proceed. If this happens, the threads will be stuck indefinitely and the program will not make any progress.

### 4.1.1 Causes and prevention of deadlocks

Deadlocks are a major issue in multi-process and multi-threaded environments that can lead to system crashes and decreased performance. In this chapter, we will discuss the causes and prevention of deadlocks.

Deadlocks occur when two or more processes are waiting indefinitely for each other to release resources. The following conditions must hold for a deadlock to occur:

- Mutual Exclusion: At least one resource must be held in a non-sharable mode. That is, only one process can use the resource at a time.

- Hold and Wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.
- No Preemption: A resource cannot be forcibly removed from a process that is holding it.
- Circular Wait: A circular chain of two or more processes exists, where each process is waiting for a resource held by the next process in the chain.

There are several methods to prevent deadlocks. These methods can be classified into two categories: prevention and avoidance.

**Prevention**: The prevention method involves removing one of the four conditions necessary for a deadlock to occur. This can be done by:

- Mutual Exclusion: If resources are shareable, then there is no need for mutual exclusion.
- Hold and Wait: One solution to the hold and wait condition is to require processes to request all the resources they need before starting execution.
- No Preemption: Preemption can be used to remove the hold and wait and circular wait conditions. However, preemption can be difficult to implement and may have a negative impact on system performance.
- Circular Wait: One solution to the circular wait condition is to impose a total ordering of all resource types and require that each process request resources in an increasing order of enumeration.

**Avoidance**: The avoidance method involves a more sophisticated approach to resource allocation. A resource allocation graph can be used to determine whether granting a request will result in a deadlock. The graph consists of nodes representing processes and resources and edges representing requests and allocations. The algorithm can check whether

the graph contains a cycle. If a cycle exists, the allocation request is denied.

Deadlocks are a serious issue in multi-process and multi-threaded environments. The causes of deadlocks can be identified and prevented by removing one of the four necessary conditions or using a resource allocation algorithm to avoid deadlocks. It is essential to use prevention or avoidance methods to ensure that deadlocks do not occur in a system, as they can lead to decreased performance or even system crashes.

## 4.2 Definition of livelocks

In a distributed computing environment, multiple processes or threads communicate and coordinate with each other to achieve a common goal. In this process, sometimes, a situation arises where a process/thread keeps on changing its state without making any progress towards the completion of the task. This condition is known as livelock. Livelocks are equally dangerous as deadlocks and can cause the system to halt indefinitely. In this chapter, we will define livelocks and discuss their characteristics, causes, and prevention techniques.

A livelock is a condition that occurs in a distributed computing environment when two or more processes/threads continuously change their state to avoid a deadlock-like situation. In a livelock, a process/thread keeps on performing some action in response to the action of other processes/threads, but the overall system makes no progress towards the completion of the task. The processes/threads involved in a livelock do not block each other, but they are unable to complete their task.

The following are some of the characteristics of livelocks:

- Two or more processes/threads continuously change their state to avoid deadlock-like situations.

- The processes/threads do not block each other but are unable to make any progress.
- The processes/threads keep on performing some action in response to the action of other processes/threads.
- The system makes no progress towards the completion of the task.

The following are some of the common causes of livelocks:

- Similar to deadlocks, livelocks can occur when multiple processes/threads contend for shared resources.
- When two or more processes/threads have a different perception of the order in which the resources should be acquired, a livelock can occur.
- In some cases, the design of the algorithm or the system itself can cause livelocks.

The following are some of the prevention techniques for livelocks:

- One way to prevent livelocks is to avoid circular dependencies between the processes/threads.
- Designing a proper algorithm can prevent livelocks. The algorithm should ensure that the processes/threads involved do not get stuck in an infinite loop.
- Using timeouts can help in detecting livelocks. If a process/thread does not make progress for a certain amount of time, it can be assumed that it is stuck in a livelock. In such cases, the process/thread can be terminated to break the livelock.

Livelocks are a serious concern in distributed computing environments as they can lead to a halt in the system's progress. Understanding the causes of livelocks and using preventive measures can help in avoiding them. Properly designed algorithms, avoidance of circular dependencies, and the use of timeouts can go a long way in preventing livelocks.

### 4.2.1 Causes and prevention of livelocks

In multi-threaded and multi-process systems, livelocks are one of the major problems that can occur along with deadlocks. Livelocks can occur when two or more processes or threads are blocked and are waiting for each other to complete their work, but none of them can progress because they keep on releasing and acquiring the same resources repeatedly. This chapter will define livelocks, discuss their causes, and provide some preventive measures to avoid them.

Livelocks are similar to deadlocks, but unlike deadlocks, processes or threads in livelocks do not block or get stuck, but keep on performing actions without making any progress towards their goal. In other words, a livelock occurs when two or more processes or threads keep on releasing and acquiring the same resources repeatedly and are unable to progress.

Livelocks are usually caused by the same situations as deadlocks, such as resource contention, improper synchronization, and circular dependencies. For example, in a dining philosophers problem, if each philosopher picks up the fork on the left and waits for the fork on the right to be free, they can end up in a livelock situation where each philosopher keeps on releasing and acquiring the same resources, but none of them can make any progress.

Preventing livelocks is similar to preventing deadlocks. The following are some preventive measures that can be taken to avoid livelocks:

- Avoid Resource Contention: One of the main causes of livelocks is resource contention. To avoid this, resources should be acquired in a consistent and orderly manner. For example, in the dining philosophers problem, if each philosopher picks up the fork on the left and waits for the fork on the right to be free, they can end up in a livelock situation where each philosopher keeps on releasing and acquiring the same resources, but none of them can make any progress. To avoid this, we can assign each

philosopher a unique number, and they can pick up the fork with the lower number first and then the higher number fork.

- Use Timeout Mechanisms: In a livelock situation, processes or threads keep on performing actions without making any progress towards their goal. In such situations, a timeout mechanism can be used, which allows the processes or threads to stop waiting after a certain amount of time and perform some other actions.
- Implement Proper Synchronization Mechanisms: Proper synchronization mechanisms, such as semaphores, mutexes, and monitors, can be used to avoid livelocks. These mechanisms can be used to ensure that processes or threads are not blocked and are allowed to proceed with their work.
- Implement Proper Error Handling Mechanisms: Proper error handling mechanisms can be used to handle unexpected situations, such as livelocks. For example, if a livelock is detected, processes or threads can be killed or restarted.

Livelocks are one of the major problems that can occur in multi-threaded and multi-process systems. They are similar to deadlocks, but processes or threads in livelocks do not block or get stuck, but keep on performing actions without making any progress towards their goal. Livelocks are caused by resource contention, improper synchronization, and circular dependencies. To prevent livelocks, resources should be acquired in a consistent and orderly manner, timeout mechanisms should be used, proper synchronization mechanisms should be implemented, and proper error handling mechanisms should be used to handle unexpected situations.

# 5 Synchronization in Distributed Systems

In today's interconnected world, distributed systems are becoming increasingly common. Distributed systems consist of multiple interconnected nodes that work together to achieve a common goal.

Such systems are prevalent in various domains, including cloud computing, peer-to-peer networks, and the Internet of Things. In such systems, synchronization is critical to ensure the correct execution of processes and the consistency of data.

In this chapter, we will explore the importance of synchronization in distributed systems and the challenges that arise due to the distributed nature of such systems. We will also discuss various methods of synchronization that are commonly used in distributed systems, such as clock synchronization and consensus algorithms.

## 5.1 Definition of distributed systems

In the modern era of computing, distributed systems are becoming increasingly prevalent. A distributed system can be defined as a collection of autonomous computers, connected via a network, that work together to achieve a common goal. These computers may be geographically dispersed, but appear to the user as a single, unified system. The purpose of a distributed system is to provide the user with the illusion of a single, centralized computing resource that can be accessed from anywhere, at any time, from any device.

Distributed systems can be classified based on their communication structure. A distributed system can be centralized, decentralized, or hybrid. A centralized system has a central entity that is responsible for managing the system. In contrast, a decentralized system does not have a central entity, and each computer in the network is responsible for managing its own tasks. A hybrid system has a mixture of centralized and decentralized elements.

Distributed systems are characterized by several key attributes. First, they are highly concurrent, with multiple computers executing tasks simultaneously. Second, they are inherently fault-tolerant, with redundant resources and failover mechanisms to ensure continuous

operation in the event of a failure. Third, they are scalable, allowing resources to be added or removed as needed to meet changing demands.

Examples of distributed systems include cloud computing platforms, peer-to-peer file sharing networks, and grid computing environments. Cloud computing platforms, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform, provide on-demand access to a wide range of computing resources, including servers, storage, and databases. Peer-to-peer file sharing networks, such as BitTorrent, allow users to share files directly with one another, without the need for a centralized server. Grid computing environments, such as the European Grid Infrastructure, provide access to a large number of geographically dispersed computing resources for scientific research.

In summary, a distributed system is a collection of autonomous computers that work together to achieve a common goal. Distributed systems are characterized by their concurrency, fault tolerance, and scalability. They are used in a wide range of applications, from cloud computing to scientific research.

## 5.2 Importance of synchronization in distributed systems

In distributed systems, synchronization is essential to ensure the correct behavior of concurrent operations on different nodes. These operations can access and modify shared resources, and without proper synchronization, there is a risk of data corruption and inconsistency. In this chapter, we will discuss the importance of synchronization in distributed systems, and how it can be achieved.

Distributed systems consist of multiple nodes that work together to provide a service. These nodes may be physically separate, connected by a network, and may have different hardware and software

configurations. In such an environment, ensuring synchronization is vital to guarantee the correctness of the system.

The first reason synchronization is crucial in distributed systems is to prevent conflicts between concurrent operations that access the same shared resource. For example, consider a distributed database system, where multiple nodes can simultaneously read and write to the same data. Without proper synchronization, the nodes may overwrite each other's changes, leading to data corruption and inconsistency.

Another reason synchronization is critical in distributed systems is to ensure consistency across different nodes. In distributed systems, nodes may store replicated copies of data to improve availability and fault-tolerance. However, maintaining consistency between replicas is a challenging task that requires proper synchronization mechanisms.

Finally, synchronization is also essential in distributed systems to achieve coordination and communication between different nodes. For example, a distributed messaging system may use synchronization to ensure that messages are delivered in the correct order, and that all nodes receive the same messages.

There are different methods of synchronization in distributed systems, depending on the specific requirements and constraints of the system. Some of the commonly used methods include:

- Clock Synchronization: In a distributed system, nodes may have different clocks, which can lead to inconsistencies in timestamp-based ordering of events. Clock synchronization techniques, such as the Network Time Protocol (NTP), can be used to synchronize the clocks and ensure consistent ordering of events.
- Consensus Algorithms: Consensus algorithms, such as the Paxos algorithm or the Raft algorithm, are used to achieve agreement among nodes in a distributed system. These algorithms can be used to coordinate the access to shared resources, ensure consistency between replicas, and achieve fault-tolerance.

Synchronization is a critical aspect of distributed systems, and it is essential to ensure the correctness and consistency of the system. Proper synchronization mechanisms, such as clock synchronization and consensus algorithms, can be used to prevent conflicts between concurrent operations, maintain consistency between replicas, and achieve coordination and communication between different nodes.

### 5.2.1 Methods of synchronization in distributed systems: clock synchronization, consensus algorithms

Distributed systems consist of multiple computers connected through a network, working together to achieve a common goal. Synchronization in distributed systems is vital to ensure that these computers operate efficiently and effectively without interfering with each other. In this chapter, we will discuss the methods of synchronization in distributed systems, namely clock synchronization and consensus algorithms.

**Clock synchronization:**

In distributed systems, each computer typically has its own clock. However, these clocks are not perfectly synchronized, and even minor differences can cause significant problems. For example, a transaction initiated on one computer may appear to have occurred before a transaction initiated on another computer, resulting in inconsistencies.

To address this issue, clock synchronization techniques are used to ensure that all clocks in the distributed system are synchronized. The two most common clock synchronization techniques are the Network Time Protocol (NTP) and the Precision Time Protocol (PTP).

NTP is an internet protocol designed to synchronize clocks of networked computers. It works by exchanging time-stamped packets between computers to calculate and adjust clock differences. NTP can achieve synchronization accuracy to within a few milliseconds.

PTP is a newer protocol designed to provide more accurate time synchronization than NTP. PTP works by sending precise timing packets between computers using hardware timestamps. It can achieve synchronization accuracy to within a few microseconds.

**Consensus algorithms:**

Consensus algorithms are used in distributed systems to ensure that all computers agree on a single value or decision. These algorithms are designed to handle failures and ensure that the system can operate correctly even if some computers fail or behave incorrectly.

The most widely used consensus algorithm is the Paxos algorithm. It works by ensuring that a proposal is only accepted if a majority of computers in the system agree to it. If a proposal fails to get a majority, a new proposal is made, and the process is repeated until a consensus is reached.

The basic idea of the Paxos algorithm is to have a group of nodes agree on a value, even if some nodes fail or are delayed. This is accomplished through a series of rounds of voting and proposal exchanges. The algorithm has three roles: proposer, acceptor, and learner.

**Example:** Here is the pseudocode for the Paxos algorithm:

```
Algorithm Paxos

    Upon receiving a proposal, a proposer selects a proposal number
n and sends a prepare message with n to all acceptors.

    Each acceptor, upon receiving a prepare message, responds with
a promise not to accept any proposal numbered less than n.

    If a majority of acceptors respond with promises, the proposer
sends an accept request to all acceptors with its proposal value
and number.

    If an acceptor receives an accept request with proposal number
n greater than any it has seen, it accepts the proposal and informs
all learners.
```

```
    If a learner receives messages from a majority of acceptors
accepting the same proposal number n, it knows that proposal has
been chosen.
```

The Paxos algorithm ensures that only one value is chosen as the final agreement, even in the presence of failures or delays. It provides a fault-tolerant mechanism for reaching consensus in a distributed system.

Another consensus algorithm is the Raft algorithm, which is simpler to understand and implement than the Paxos algorithm. It also uses majority voting to ensure that a value or decision is agreed upon by all computers in the system.

**Example:** Here is a high-level pseudocode of the Raft algorithm, which is a consensus algorithm used in distributed systems:

**Initialization**: Each node initializes its own state, including its current term, a votedFor variable that indicates which candidate the node voted for in the current term (or null if it hasn't voted), and a log that stores all the commands that have been agreed upon.

The node also maintains a list of all other nodes in the system and their current states.

**Leader election:** Nodes start out in the follower state, listening for messages from other nodes.

If a follower doesn't hear from a leader within a certain time period (called the election timeout), it becomes a candidate.

The candidate increments its current term and requests votes from all other nodes.

A node votes for the candidate if it hasn't voted in this term already and if the candidate's log is at least as up-to-date as the voter's log.

If the candidate receives a majority of votes, it becomes the leader and sends out heartbeats to all other nodes to establish its authority.

**Log replication:** When a client sends a command to the leader, the leader appends it to its own log and sends it out to all other nodes as an "append entries" message.

If a node receives an "append entries" message from the leader with a log entry that conflicts with its own log, it rejects the message.

If a node receives an "append entries" message from the leader with a log entry that is not in its own log, it appends the entry and sends back an acknowledgement.

Once the leader has received acknowledgements from a majority of nodes for a given log entry, it considers the entry committed and applies it to its state machine.

There are additional details and optimizations in the Raft algorithm, but this gives a basic idea of how it works.

In summary, synchronization is essential in distributed systems to ensure that all computers operate efficiently and effectively without interfering with each other. Clock synchronization techniques such as NTP and PTP are used to synchronize clocks, while consensus algorithms such as Paxos and Raft are used to ensure that all computers agree on a single value or decision.

# 6 Case Study: Synchronization in Java Concurrency Utilities

In this chapter, we will explore the various synchronization mechanisms that are available for multi-threaded and multi-process environments. We will start by discussing critical sections and race conditions and their importance in synchronization. Then we will delve into different synchronization mechanisms such as locks, semaphores, monitors, and

barriers. We will also compare and contrast these mechanisms based on their performance, complexity, and suitability for different scenarios.

Next, we will discuss deadlocks and livelocks, their definitions, causes, and prevention techniques. We will also examine how to handle these issues in distributed systems, where synchronization across multiple machines is required.

Finally, we will explore a case study on synchronization in Java Concurrency Utilities. We will examine the Java Concurrency Utilities, compare them with synchronization mechanisms in other programming languages, and explore their impact on Java programs' performance, consistency, and correctness.

Overall, this chapter will provide a comprehensive overview of synchronization and its importance in modern computing. We will examine the different synchronization mechanisms available, their strengths and weaknesses, and how they can be used to ensure consistency, prevent race conditions and deadlocks, and improve program performance.

## 6.1 Overview of Java Concurrency Utilities

Java Concurrency Utilities, also known as Java Concurrency API, is a set of tools and features in the Java programming language that helps developers write multithreaded programs with ease. With the increasing demand for concurrent applications, the Java Concurrency Utilities play a crucial role in simplifying the process of creating efficient, thread-safe, and scalable applications.

The Java Concurrency Utilities consist of several components, including:

- Executors: Executors are the core components of the Java Concurrency Utilities. They provide an abstraction layer for managing threads, scheduling tasks, and executing them

asynchronously. Executors are used to create and manage pools of threads, which can be used to execute multiple tasks concurrently.

- Futures: Futures are used to represent the results of an asynchronous computation. They allow developers to obtain the result of a computation that may not have completed yet. Futures provide a way for developers to write asynchronous code that can be executed in parallel.
- Locks: Locks are used to provide mutual exclusion to critical sections of code. They ensure that only one thread can access the critical section at a time, preventing race conditions and other concurrency issues.
- Atomic Variables: Atomic variables are used to provide thread-safe access to shared variables. They ensure that reads and writes to the variable are atomic and do not interfere with other threads.
- Concurrent Collections: Concurrent collections are data structures that are designed to be used in a concurrent environment. They provide thread-safe access to shared data, allowing multiple threads to access the data simultaneously without the risk of data corruption.

The Java Concurrency Utilities have become an essential part of the Java programming language. They provide developers with the tools and features they need to write efficient and scalable multithreaded applications. With the increasing demand for concurrent applications, the Java Concurrency Utilities will continue to play a vital role in simplifying the process of creating thread-safe and scalable applications.

## 6.2 Comparison with synchronization mechanisms in other programming languages

Java Concurrency Utilities provide a high-level and platform-independent framework for managing concurrency in Java programs. The framework provides several synchronization mechanisms, such as locks, semaphores, and barriers, that allow developers to control access to shared resources and coordinate the execution of multiple threads.

Compared to synchronization mechanisms in other programming languages, Java Concurrency Utilities offer several advantages. For example, the framework provides built-in support for thread pools, which can significantly improve the performance of applications that require the execution of multiple tasks concurrently. Additionally, the framework provides several classes that facilitate thread-safe communication between threads, such as BlockingQueue and ConcurrentHashMap.

Another advantage of Java Concurrency Utilities is that they provide a high level of abstraction that allows developers to focus on the functionality of their applications rather than on the low-level details of thread synchronization. This can improve the readability, maintainability, and reusability of code, as well as reduce the likelihood of introducing synchronization bugs.

However, there are also some limitations to Java Concurrency Utilities. For example, the framework does not provide support for distributed synchronization or real-time synchronization, which are important in some applications. Additionally, the performance of some of the synchronization mechanisms provided by the framework can be negatively impacted by contention, which occurs when multiple threads try to access the same resource simultaneously.

Overall, Java Concurrency Utilities offer a powerful and flexible framework for managing concurrency in Java programs, and they are

well-suited for many types of applications. However, developers should carefully consider the specific requirements of their applications when choosing a synchronization mechanism, and they should be aware of the potential limitations and performance trade-offs of the mechanisms provided by the framework.

# 7 Conclusion

In conclusion, synchronization is a fundamental concept in operating systems that ensures the correct and consistent execution of concurrent processes or threads. The importance of synchronization is evident in multi-threaded and multi-process environments, where race conditions and deadlocks can lead to incorrect results, inconsistent states, and system failures.

Various synchronization mechanisms such as locks, semaphores, monitors, and barriers provide a means to enforce synchronization and ensure mutual exclusion, coordination, and communication among concurrent processes or threads. However, each mechanism has its strengths and weaknesses, and the choice of the most appropriate mechanism depends on the specific requirements of the application.

In addition to synchronization within a single machine, synchronization is also critical in distributed systems, where multiple machines need to coordinate and communicate to perform a task. Techniques such as clock synchronization and consensus algorithms enable synchronization in distributed systems.

Finally, modern programming languages such as Java provide built-in concurrency utilities that simplify the use of synchronization mechanisms and enable developers to write correct and efficient concurrent programs. However, care must still be taken to avoid common pitfalls such as race conditions, deadlocks, and livelocks.

Overall, synchronization is a complex and crucial topic in operating systems, and a thorough understanding of its principles and mechanisms is necessary for developing reliable and efficient concurrent applications.