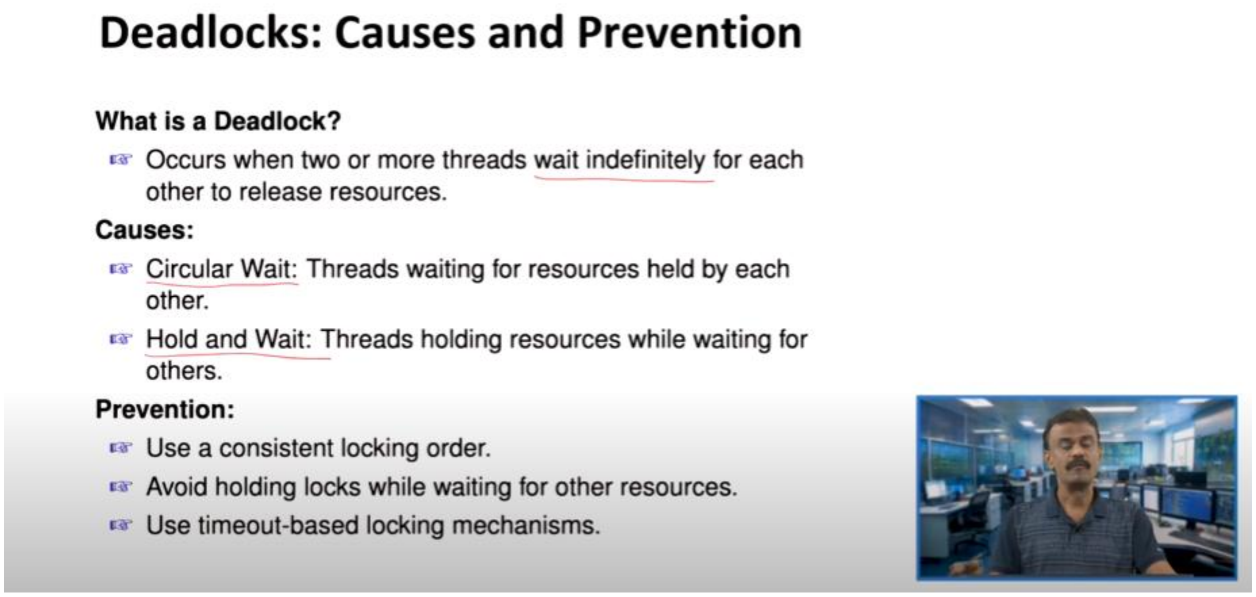
FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture52

Lecture 52: Deadlocks - Causes and Prevention



Deadlocks: Causes and Prevention

What is a Deadlock?
- Occurs when two or more threads wait indefinitely for each other to release resources.

Causes:
- Circular Wait: Threads waiting for resources held by each other.
- Hold and Wait: Threads holding resources while waiting for others.

Prevention:
- Use a consistent locking order.
- Avoid holding locks while waiting for other resources.
- Use timeout-based locking mechanisms.

Thank you. Welcome to lecture number 52, Advanced Topics. So, in the last class, I talked about deadlocks, right. So, here I will start with deadlocks, causes and prevention. So, what do you want by deadlock? So, I gave a physical example.

Suppose, I am in a track, the two trains, right, they are facing each other, right. Assume that no accident is happening, but they are facing each other by somehow, let us say due to signal failure or so, right. So, in that case, how do you, right, overcome this? Or even before, how do you prevent this? So here, right?

So in the programming point of view, right? So deadlock. So this occurs when two or more threats. So here I'll talk in terms of threat, right? So they wait indefinitely, right?

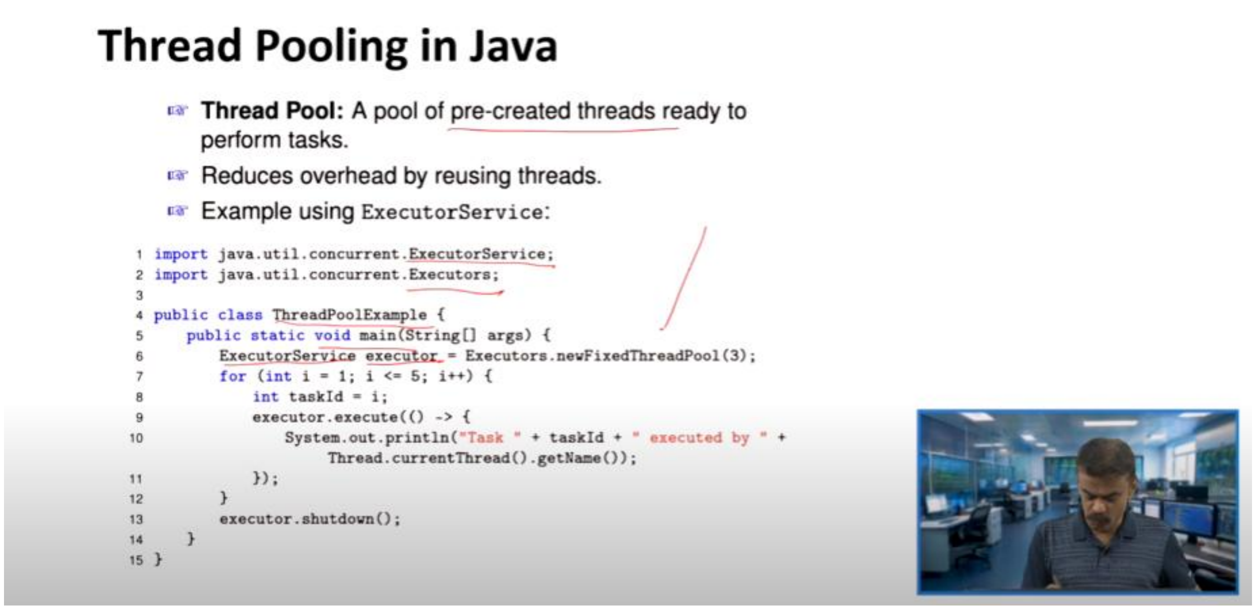
So they wait indefinitely for each other to release the resources, right? Almost the same meaning, right? So this causes a circular wait. So, this causes a circular wait. What do you mean by circular wait?

So, the threads are waiting for resources held by each other. So, this is called a circular wait. Another one is hold and wait. So, threads are holding resources while waiting for others. So, these are all the causes of deadlock.

One is a circular wait. Another one is hold and wait. So, how do you prevent this? So, we have to use a consistent locking order, right? So, when you use a consistent locking order, we can prevent it.

So, the deadlock can be avoided by holding locks while waiting for other resources. Or, we can use timeout-based locking mechanisms, right? So, this is about your deadlock. We will talk about thread pooling in Java, right? So, what do you mean by thread pooling?

A pool of pre-created threads, right? A pool of pre-created threads ready to perform the task. Right. So, this thread pooling reduces overhead by reusing the threads, and we are going to use this executor service, right. In a Java utility, let us use the executor service.



Thread Pooling in Java

- **Thread Pool:** A pool of pre-created threads ready to perform tasks.
- Reduces overhead by reusing threads.
- Example using ExecutorService:

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.execute(() -> {
                System.out.println("Task " + taskId + " executed by " +
                    Thread.currentThread().getName());
            });
        }
        executor.shutdown();
    }
}
```

So, we will take an example with the executor service, and then we will use executors as well, right. So, here you go: public class ThreadPoolExample, right. So, let us consider the class; it is a driver class, ThreadPoolExample. So, here you have the main program, correct. So, here you have the main program, and executor is an object under executor service, right.

So your executors is invoking new fixed thread pool. So here it can hold up to three threaded pools, right. So three pre-created threaded pools, that is what the meaning, right. So here you have three threaded pools. So, now for i is equal to 1 i less than or equal to 5 i plus plus right.

So, three threaded pools for loop and task id is i. So, now you have a lambda expression here it is calling the execute and it is a lambda expression. So, now you have system dot out dot println lambda task right. So, three threaded pools are there that is what line number 6 is saying. Right. So task ID you are printing and the thread, which is calling the getName under the current thread.

OK, so this is going on. This lambda expression is going on. Right. So then once it is printing, executor is invoking shutdown. Right.

Executor is invoking shutdown. So if I run the code. Right. So I'll get. So I said, how many threads, how many threaded pools?

Three. Right. How many threaded pools? Three. So, you can see thread 3, 2, 1 and again 3, 2 right something like that.

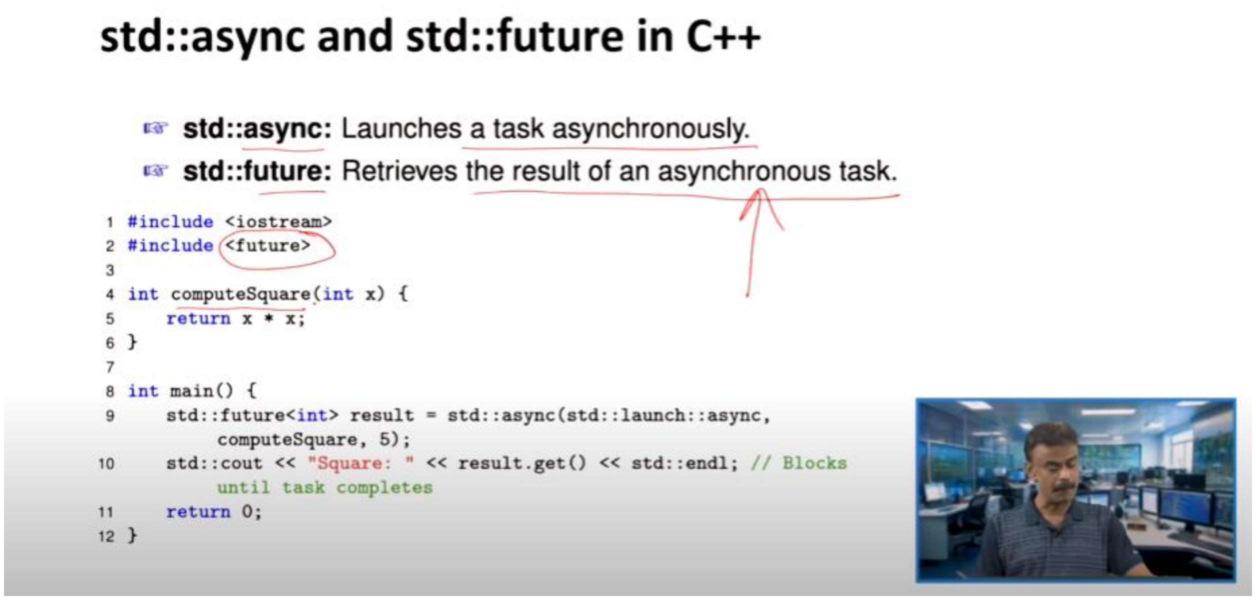
So, here you can see the task id. So, need not be 1, 2, 3, 4, 5 this we had seen previously also right. So, it is starting with 3 and 2, 1 randomly 4, 5 right. But totally you have i is equal to 1, i less than or equal to 5, i plus plus right. So, what we can do we will run this code in java.

So, let us go to the java compiler you can see the output ok. So, the I said 3 right. So, the 3 threaded pools right. So, if you look at the output the 3 2 1 right. So, again 2 3.

So, it is like a slightly a different output. We are not getting 1 like what we used to get here right. So, yeah you have to get like this right. So, here you can see the thread you can see 3 2 1 right. So, here you have 3 2 2 1 3

Whereas I projected here 3, 2, 1, 3, 2. Yes, it is possible. Right. And the ID is also 1, 2, 3, 4, 5. All are covering in a different way.

Right. So you have 3 tasks first: task 3 first, then 2, 5, 1, 4. Right. So this is how the thread pool concept works in Java. So now let us consider two classes.



std::async and std::future in C++

- **std::async:** Launches a task asynchronously.
- **std::future:** Retrieves the result of an asynchronous task.

```cpp
#include <iostream>
#include <future>

int computeSquare(int x) {
    return x * x;
}

int main() {
    std::future<int> result = std::async(std::launch::async,
        computeSquare, 5);
    std::cout << "Square: " << result.get() << std::endl; // Blocks
        until task completes
    return 0;
}
```

Right. Async and future in C++. Right. Asynchronization. So asynchronization is launching a task asynchronously.

All right. And the future class retrieves the result of an asynchronous task. Let us see an example. So let us include future. All right.

Let us include the header file 'future' along with 'iostream'. All right. So now you have 'compute_square'. All right. The function.

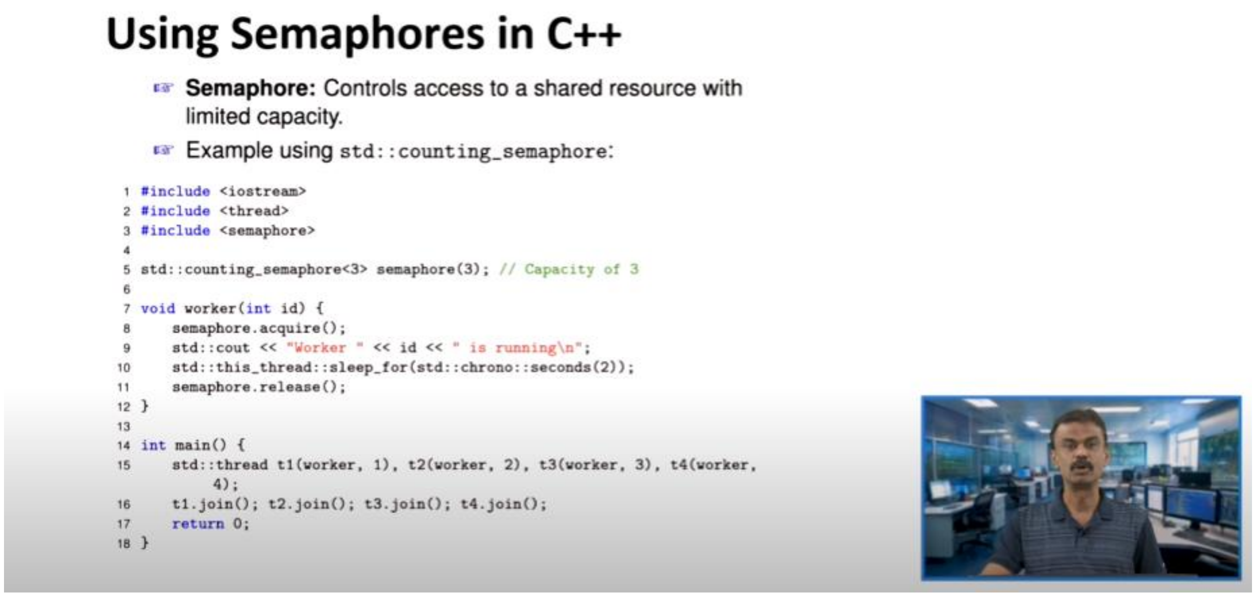
'compute_square', all right. So, which is returning x squared, which is not a problem. Let us go to the main program. So, in the main program, you have an object 'result' under the class 'future', a future of integers, right. So here, 'async', which is a constructor, you are passing 'async', 'compute_square', and 5, right.

So, in fact, we have seen a similar meaning; it is nothing but when 'result' is invoking, right? When 'result' is invoking 'get' under 'future', it will call 'compute_square' by passing x as 5. So now, 'cout' square you put, 'result' is invoking 'get', right? 'result' is invoking 'get', that means you are passing 'compute_square' and 5 over here, in the constructor of 'async'. So what is happening over here? x is what now? x is 5, so that means return x squared So, 5 into 5, 25 will be returned.

So we should get the output square is 25, right. So when I run the code, I have to get the output square, right, colon. I am expecting this. That will be printed. And once the function result.get is invoking.

So async, you are passing compute square and 5, correct. So compute square is nothing but a function in line number 4. That means 5 you are passing. X will take the value 5. and x into x is x square.

So, square is 25. So, this will be printed. So, can you just work it out and yes what I will do I think first time we are running C++ right. So, we will run I will run C++. I thought of you will work what we can do we will work it out Dave C++



Using Semaphores in C++

- **Semaphore:** Controls access to a shared resource with limited capacity.
- Example using `std::counting_semaphore`:

```cpp
1  #include <iostream>
2  #include <thread>
3  #include <semaphore>
4
5  std::counting_semaphore<3> semaphore(3); // Capacity of 3
6
7  void worker(int id) {
8      semaphore.acquire();
9      std::cout << "Worker " << id << " is running\n";
10     std::this_thread::sleep_for(std::chrono::seconds(2));
11     semaphore.release();
12 }
13
14 int main() {
15     std::thread t1(worker, 1), t2(worker, 2), t3(worker, 3), t4(worker,
           4);
16     t1.join(); t2.join(); t3.join(); t4.join();
17     return 0;
18 }
```

or here it is command parameter we can G C G++ I will do yes you can cout the output square is 25, so compilation and run the code in the command prompt right, so you are getting square 25, okay i hope it is clear so the next concept semaphores right using semaphores in c plus plus so semaphore which is controlling access to a shared resource with limited capacity right so here what we can do so we can use counting semaphore right

so we can include semaphore like what we have done in line number three right and here you have the initialization right line number five you have the initialization which is nothing but counting a semaphore right so which is initialized with the capacity of three Right. So the meaning is it can allow up to three threads to proceed at a time.

Right. So let us see how the worker function is doing. Right. So here you have the worker function. Right.

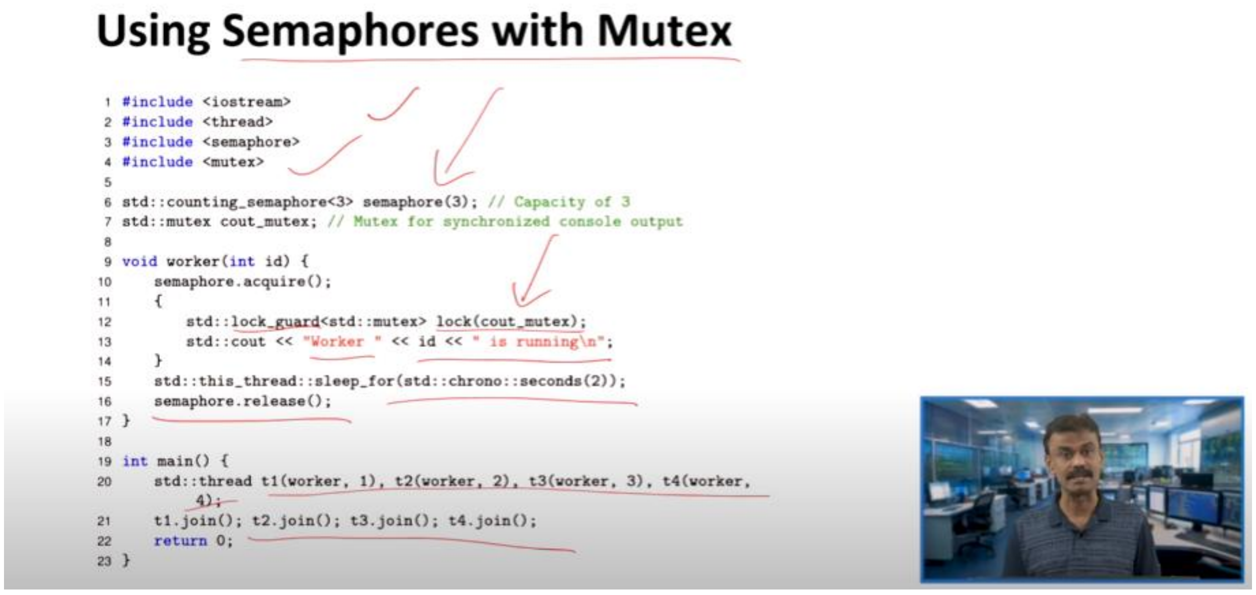
So the worker function, which is acquiring the semaphore. And here you have printing the message worker. Right. With worker ID. Right.

Is running. Right. So here you have this underscore thread. So under this, you have sleep underscore far. So that means this will sleep for 2 seconds.

So this will sleep for 2 seconds and then releases the semaphore. This is what exactly the work function goes. So let us go to the main program. In the main program you have 4 threads. So the total capacity is 3.

You have 4 threads. The threads are nothing but t1, t2, t3, and t4. So each is calling the worker function with a different ID. So everyone is calling the worker function with a different ID. The main thread waits for all worker threads to finish using join.

So t1, t2. In fact, we have seen this example. Only one time we use, two times we use if you recall. So here the main thread waits for all worker threads to finish using join, right.



Using Semaphores with Mutex

```cpp
#include <iostream>
#include <thread>
#include <semaphore>
#include <mutex>

std::counting_semaphore<3> semaphore(3); // Capacity of 3
std::mutex cout_mutex; // Mutex for synchronized console output

void worker(int id) {
    semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock(cout_mutex);
        std::cout << "Worker " << id << " is running\n";
    }
    std::this_thread::sleep_for(std::chrono::seconds(2));
    semaphore.release();
}

int main() {
    std::thread t1(worker, 1), t2(worker, 2), t3(worker, 3), t4(worker,
        4);
    t1.join(); t2.join(); t3.join(); t4.join();
    return 0;
}
```

So, here the semaphore ensures that only three worker threads, right. So, that can execute the critical section, right. So, which can execute the critical section, right. So, printing the message and sleeping at a time. Once a thread finishes its critical section, it releases a semaphore allowing another waiting thread to proceed.

So that is what exactly it is happening. So this regulates the number of threads accessing the shared resource and prevents race conditions. So in the console output.

So in this case, so the program that you had seen, the mutual exclusion is not guaranteed. So here the mutual exclusion is not possible.

So the multiple threads that you are seeing over here may try to write in one go. as you can see in the output right. So, suppose if I run the code I mean here you go here you are see. So, this is output right. So, multiple threads you are seeing, may try to write in one go as you can see in this output right.

So, but this is depending on the underlying resources also right. So, how we can fix this? So, we can fix this using the mutex. So, that we are going to see. So, before that what we can do?

We can run this code in C++. So, we can run this code in C++. So, here you go. So, here you are. That is what I said.

The mutual exclusion is not guaranteed. But here it comes. So, it depends. So, sometimes you may get the output I mentioned in the slide. Right.

So, sometimes you may get this also. So, it is possible, but suppose you are getting this, as we saw here, right? Suppose you are getting this. So, in that case, what can we do, right? So, we are going to use, suppose, right? I am running the code; there is a possibility, right? So, it depends on the resources, right? Suppose you are getting the output like this so this problem can be solved right we are going to see in the next slide using mutex. So now we will use semaphores with mutex so whatever problem that we had seen in the previous slide so when I include mutex right or semaphores with mutex right almost the same program so what we can do we can use the same code semaphore of 3 right and worker function so which is acquiring So, here you can have this lock right.

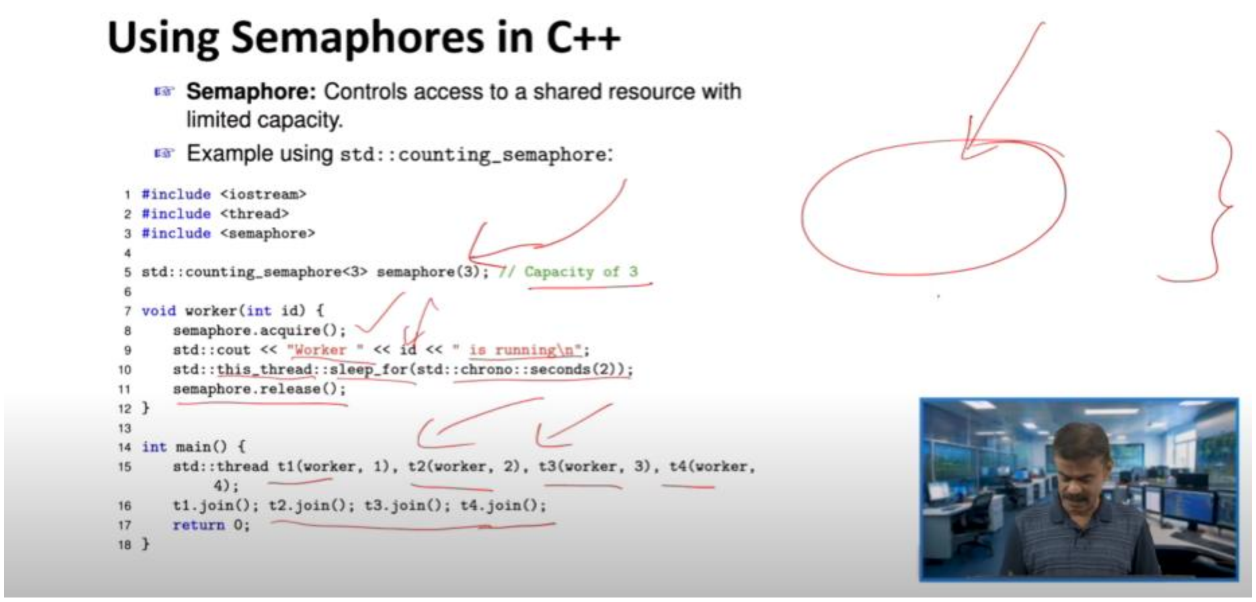
So, under lock right. So, you are pausing lock underscore mutex. In fact, mutex we have studied right under the class lock guard of mutex right. So, now you try to print this worker id and is running right. So, rest are all same.

So, sleep for 3 seconds and then it will release. right. So, you are having 4 threads and then the join you are calling. Rest are all same only line number 12 we are including that means lock function you are passing cout underscore mutex under lock underscore got right. So, now when I run the code right.

So, I have to get the output like this right the id can be changed right it need not be 1 2 3 4. So, you can get like this also right. So, in the previous example previous right previous only semaphores right. So, there is a problem it is all depending on your CPU all right.

So, the one we have run right now I mean the previously I mean you have got right. So, this is all I mean depending on your CPU. So, in this case, so semaphore with mutex, so you will always get like this, ok. So, in the previous example, there is a possibility that you may get like this.

You are seeing worker, worker, worker, 1 is running, yeah you may get, right. So, you can keep on running this. right because in our case. So, sometimes it will work fine also right, but to avoid this problem we are using the mutex and when we are running the code we will get the output like this. So, what we can do we will run the code in C++ right.



Using Semaphores in C++

- **Semaphore:** Controls access to a shared resource with limited capacity.
- Example using `std::counting_semaphore`:

```
#include <iostream>
#include <thread>
#include <semaphore>

std::counting_semaphore<3> semaphore(3); // Capacity of 3

void worker(int id) {
    semaphore.acquire();
    std::cout << "Worker " << id << " is running\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));
    semaphore.release();
}

int main() {
    std::thread t1(worker, 1), t2(worker, 2), t3(worker, 3), t4(worker,
        4);
    t1.join(); t2.join(); t3.join(); t4.join();
    return 0;
}
```

So, you can see over here right we are getting the output. In fact, here it is giving 1, 2, 3, 4 right C++ with mutex right the previous one 1, 2, So, now what I will do same of over with mutex. So, that is you are getting perfectly, right. So, there is a possibility that both will work and give the same output possible, right.

It is all I mean how your CPU is using the resources, correct. But sometimes what happens? So, whatever the output that we got in the previous slide. So, there is a possibility, right. So, you may get like this.

So, you can try only thing you have to try, right. So, you may get like this. So, this problem will be solved. In fact, we had seen also. So, we are getting the output like this.

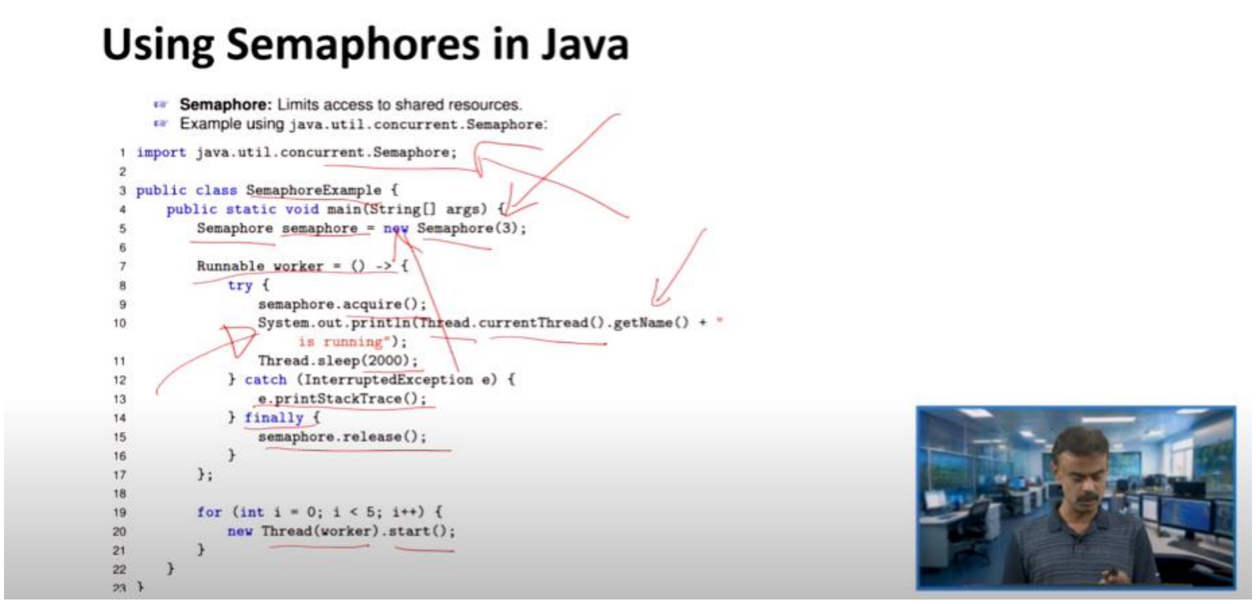
So, this is semaphores with mutex. So, let us consider with the java right. So, semaphores in C++ we had seen. So, now we consider semaphores with I mean in java right. So, here we are including the utility semaphore right concurrent dot semaphore.

So, let us take the class semaphore example right. So, here I have an object semaphore small s. under the class Semaphore right. So, dynamically allocated the memory with the constructor semaphore and as usual right. So, you have three right.

So, three threads are running right. So, three resources right. So, now let us have the lambda expression runnable worker equal to. So, similar we have used in C++ also right. So, you are trying to acquire the semaphore is trying to acquiring

So, almost the same. So, here you have getName under thread right. So, which is available in current thread right is running will be printed. So, which thread will is running let us say thread 1 identifier and thread dot sleep for 2000 milliseconds. So, that means 2 seconds it will sleep same right.

So, here you are putting 2000 because of the milliseconds and try So, if there is any exception, it will be caught over here. It will be calling the print stack trace. And finally, right, so it will be releasing that remover will call the release. Exactly what we have done in C++.



Using Semaphores in Java

- **Semaphore:** Limits access to shared resources.
- Example using `java.util.concurrent.Semaphore`:

```java
1  import java.util.concurrent.Semaphore;
2
3  public class SemaphoreExample {
4      public static void main(String[] args) {
5          Semaphore semaphore = new Semaphore(3);
6
7          Runnable worker = () -> {
8              try {
9                  semaphore.acquire();
10                 System.out.println(Thread.currentThread().getName() + "
                       is running");
11                 Thread.sleep(2000);
12             } catch (InterruptedException e) {
13                 e.printStackTrace();
14             } finally {
15                 semaphore.release();
16             }
17         };
18
19         for (int i = 0; i < 5; i++) {
20             new Thread(worker).start();
21         }
22     }
23 }
```

So, here it is equivalent in Java. So, now for loop i is equal to 0, i less than 5, i plus plus. The thread, right, we are pausing worker. The worker will invoke startup. right the worker will invoke start exactly same what we have done right

So, that means it will acquire, then it will try to print, and then it will sleep for two seconds, right? Finally, it will be released, right? So here, you will get the output like this, right? So, you may get different IDs: thread 0, 3, 1, 4, 2, and you can see they are running, right? So, at a time, you have Three resources—that is what line number 5 is suggesting. So, what can we do?

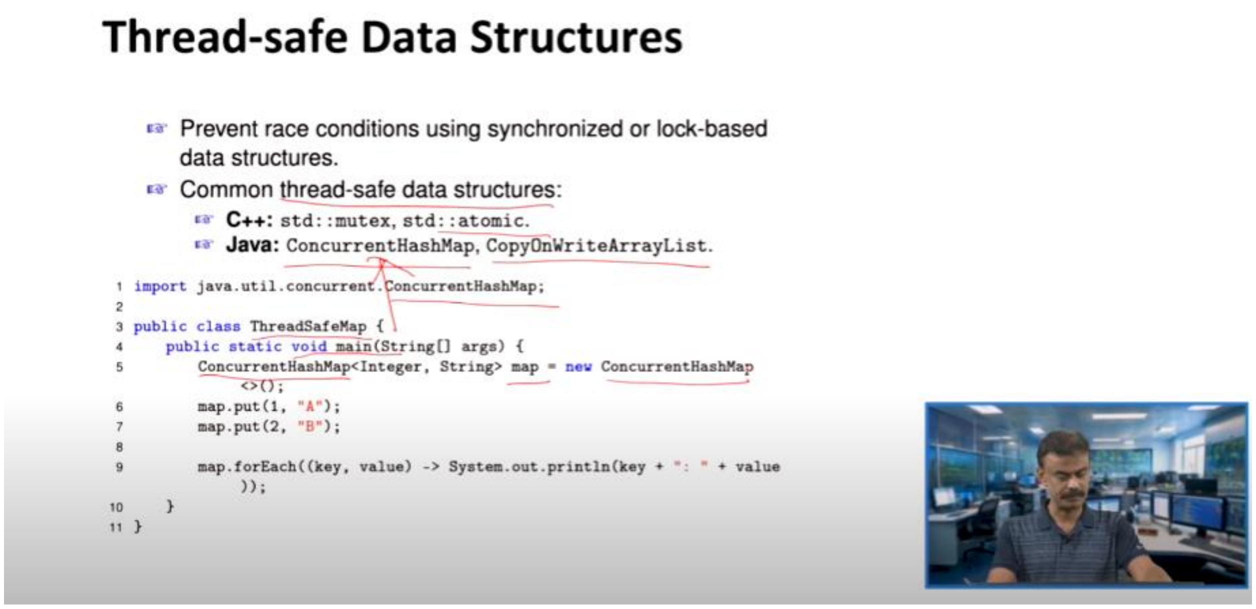
We will run the code in Java. So, use the Java compiler. Yes, you can see. Alright. So, 21034.

Because we have run the loop from 0 to 4, if you recall. Right. So, that is why we are getting output like this. Right. So, here you have 0, 3, 1, 4, 2.

And yeah, you may get any other order also. So, here when we are running 21034. Okay. So, this is how the semaphore works in Java. So, both C++ and Java.

So, now. So, my advice is to practice these codes. So, in fact, in C++, we have seen semaphore with mutex also. Correct. So, these are all the programs you can practice.

So, let us see the next concept. Thread-safe data structures. Right. So, to prevent race conditions, we have already seen mutex. Right, so now using synchronized or lock-based data structures, right? So, we are going to see how this is working, right?



Thread-safe Data Structures

- Prevent race conditions using synchronized or lock-based data structures.
- Common thread-safe data structures:
  - **C++:** `std::mutex`, `std::atomic`.
  - **Java:** `ConcurrentHashMap`, `CopyOnWriteArrayList`.

```java
import java.util.concurrent.ConcurrentHashMap;

public class ThreadSafeMap {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap
            <>();
        map.put(1, "A");
        map.put(2, "B");

        map.forEach((key, value) -> System.out.println(key + ": " + value
            ));
    }
}
```

So that you call it a thread-safe data structure. So in C++, the common thread-safe data structures We have already seen mutex, all right. So apart from mutex, we can also see atomic, right? And in Java, ConcurrentHashMap and CopyOnWriteArrayList, right? So these are all the synchronized or The log-based data structures. So these are all.

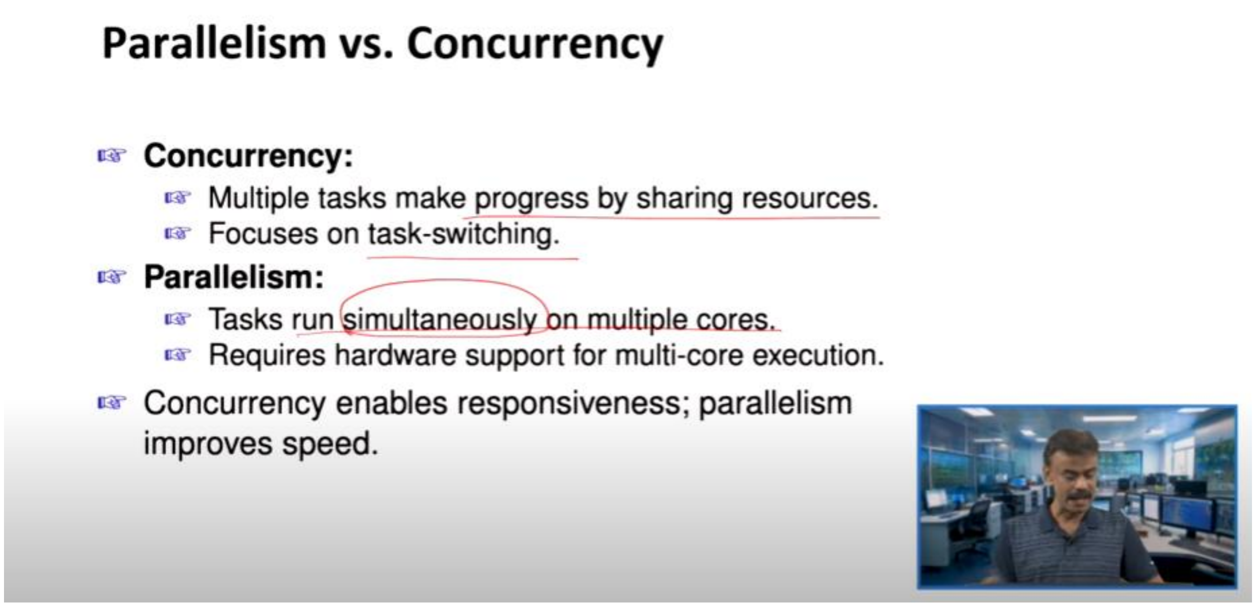
The common thread-safe data structures. So let us consider. In Java. So you include the utility. ConcurrentHashMap.

When you include. The ConcurrentHashMap. So here, public class. You have the name of the class as ThreadSafeMap. So here you have main.

Therefore it is the driver class. So here you have an object map. under ConcurrentHashMap. So, that is what I said previously right ConcurrentHashMap. So, which has integer and string and in the right hand side you have dynamically allocated a memory with a ConcurrentHashMap constructor ok.

So, let us have the map the object map which is invoking put. So, one is a key another one is a value your key is one map right we have seen map data structure. right. So, put, map is invoking put, right, 1 comma a, 1 is the key and a is the value. Similarly, 2 and b, right.

So, now the map is invoking for each, for every, right. So, key and value, right, for each key and value with the help of arrow operator system dot out dot println. So, it is going to print key and value. So, what are all the key and value are there 1 and 2. So, 1 and 2 are nothing but the keys.



Parallelism vs. Concurrency

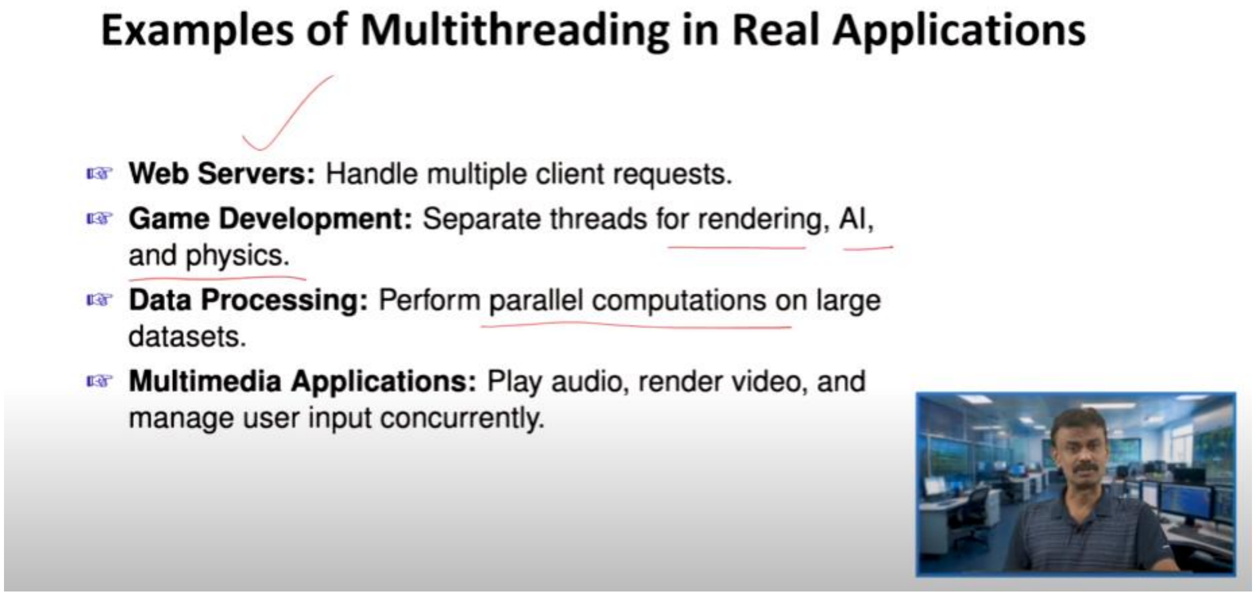
- **Concurrency:**
  - Multiple tasks make progress by sharing resources.
  - Focuses on task-switching.
- **Parallelism:**
  - Tasks run simultaneously on multiple cores.
  - Requires hardware support for multi-core execution.
- Concurrency enables responsiveness; parallelism improves speed.

The corresponding value should also be printed when you are calling this. So, these both will be printed. So, when I run the code, I have to get the output 1 colon. So, that is what key 1 colon value a for first and then again. So, 2 b which will also be printed.

So, this we will see in Java right. So, let us go to the Java compiler we will run the code yes here when you run the code you can see the output right 1 colon a 2 colon b. So, thread safe data structures this is an example. So, in terms of Java I explained this. So, the common thread safe data structures in C++ mutex already we had seen you can also work it out with atomic. And in the case of Java, you can see this example using concurrent hash map, right?

So, you can also think of using, you can try this copy on write array list. So, let us consider this case, right? What is the difference between parallelism and concurrency, right? So, in the concurrency, the multiple tasks make progress by sharing resources. This is what we study, right?

The concurrent and several examples we have seen. Also, concurrency focuses on task switching. So, you are getting 0, 1, 2, 3, or sometimes 1, 2, 0. So, this is concurrency. So, then what do you mean by parallelism?



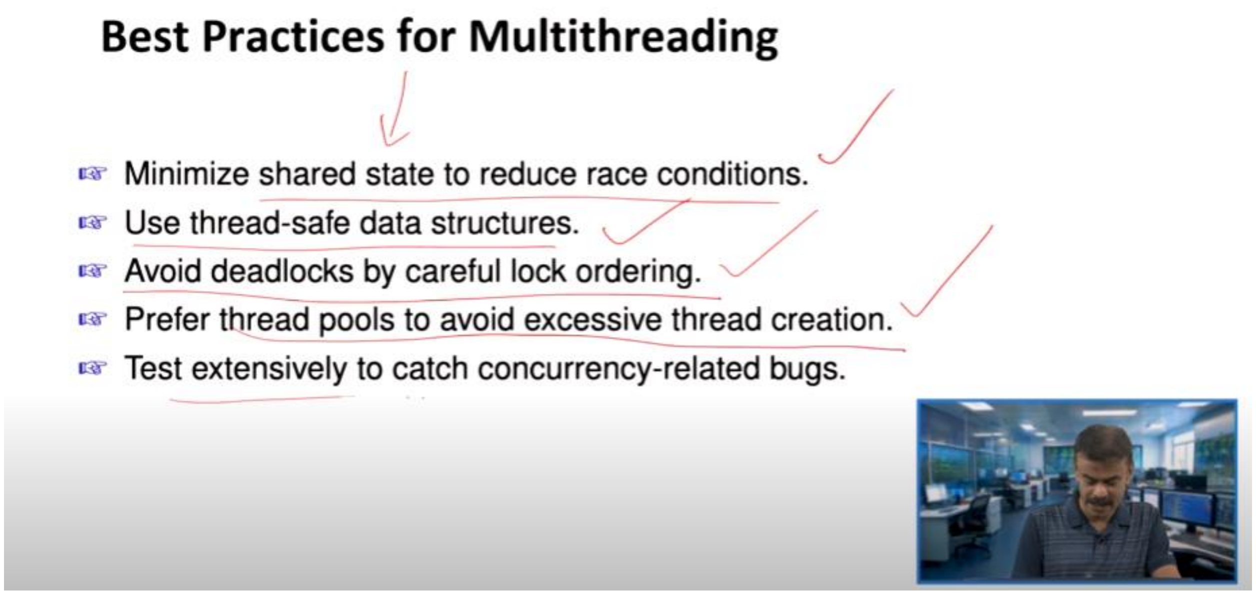
Examples of Multithreading in Real Applications

- **Web Servers:** Handle multiple client requests.
- **Game Development:** Separate threads for rendering, AI, and physics.
- **Data Processing:** Perform parallel computations on large datasets.
- **Multimedia Applications:** Play audio, render video, and manage user input concurrently.

So, parallel—I am calling it a parallel program. So, the task runs simultaneously on multiple cores. So, nowadays you are getting a multiple-core machine. So, the task which runs simultaneously—that is a keyword—on multiple cores. So, parallelism requires hardware support for multiple-core execution, right?

So, whereas concurrency enables responsiveness, parallelism improves the speed. Suppose you have a parallel machine and then assume that you are running the program, right? If you assume that you are using a single machine, right? So, your speed will improve, right? Right, whereas in the case of concurrency, this enables the responsiveness.

So, this is a major difference between parallelism and concurrency. When I am talking about multi-threading, right, in real-time examples, you can have web servers, right? So, web servers can handle multiple client requests. Game development, right? When you have rendering, AI, or physics, right? When you want to develop some games, you can have separate threads, right. So, separate threads for AI, separate threads for physics, rendering, all right.



Best Practices for Multithreading

- Minimize shared state to reduce race conditions.
- Use thread-safe data structures.
- Avoid deadlocks by careful lock ordering.
- Prefer thread pools to avoid excessive thread creation.
- Test extensively to catch concurrency-related bugs.

Another application is data processing, all right. This performs parallel computations on large data sets. Suppose I want to, right, read a satellite image file, all right. So, which is a very hot task, right? It is a hyperspectral or multispectral image, and I require parallel computations or even want to do some, let us say, image processing, right?

Image processing algorithms on satellite images. So, these kinds of applications require parallel computations, all right? So, the next one is multimedia applications. Suppose you want to play audio or render video, all right? Or you want to select some frames, keyframes, right?

Or you want to do some action recognition in the video. Right. And manage user input. Right. So these are all happening concurrently.

OK. So play audio, render video, and manage user input concurrently. So these are all under the multimedia applications. So these are all the real-time applications. So now the next one is best practices for multithreading.

Right. So, you minimize shared state to reduce race conditions, right. So, you have to minimize shared state. Therefore, we can reduce the race conditions, and we use thread-safe data structures. So, this we had seen, this also we had seen, all right.

And for multi-threading, right. So, we have to avoid deadlocks by careful lock ordering. this also right we had seen with an example and i also told right so you have to prefer thread pools to avoid excessive thread creation and we have to test extensively to catch concurrency related bugs right

so these are all the best practices for multi threading so i will summarize the multi-threading concepts right So the multithreading enables concurrent execution in programs. So we have covered these topics, right? So we have talked about thread pooling, synchronization, asynchronization, semaphores, semaphores with mutex, right? So we also talked about thread safe data structures and then we had seen the difference between parallelism and concurrency, right?

So the advice is you have to choose the appropriate strategies to balance the performance and the correctness right. So, with this the multithreading concepts is over. So, in the next class I will talk about the network programming. Thank you very much.