Recitation - 12

Jahnavi - jp5867@nyu.edu

Concurrency Vs Parallelism:

1. Concurrency refers to the ability of a program to perform multiple tasks simultaneously, by dividing the tasks into smaller units called threads or processes. Each thread or process can run independently, and they can communicate with each other as needed.
2. Parallelism, on the other hand, refers to the ability of a program to execute multiple instructions simultaneously, by dividing a task into smaller sub-tasks that can be executed in parallel on multiple processors or cores.
3. In other words, concurrency is about managing multiple tasks at the same time, while parallelism is about speeding up a single task by breaking it down into smaller parts and executing them in parallel.

Thread:

1. A thread is a lightweight, independent unit of execution within a process.
2. Threads share the same memory space and resources of the process that created them, but each thread has its own stack for storing its own local variables and function calls.
3. Threads can run concurrently and independently of each other, allowing a program to perform multiple tasks simultaneously.
4. Threads can communicate with each other by sharing data in the same memory space, but they need to synchronize their access to shared resources to avoid conflicts and race conditions.
5. The use of threads can improve the responsiveness and efficiency of a program, but it can also introduce complexity and potential issues, such as race conditions, deadlocks, and synchronization problems, that need to be carefully managed and addressed.

Tasks in Ada:

1. In Ada, a task is a unit of concurrent execution that can run independently of other tasks. Tasks are used to implement concurrency and parallelism in Ada programs, allowing different parts of the program to execute simultaneously and interact with each other in a controlled manner.
2. Ada allows concurrency to be specified by a language construct known as a task. A non-concurrent Ada program contains a single main procedure that is started when the program is started; and when the procedure terminates the program terminates.
3. A concurrent Ada program can include any number of tasks - declared by

   ```
   task <name> is
   ```

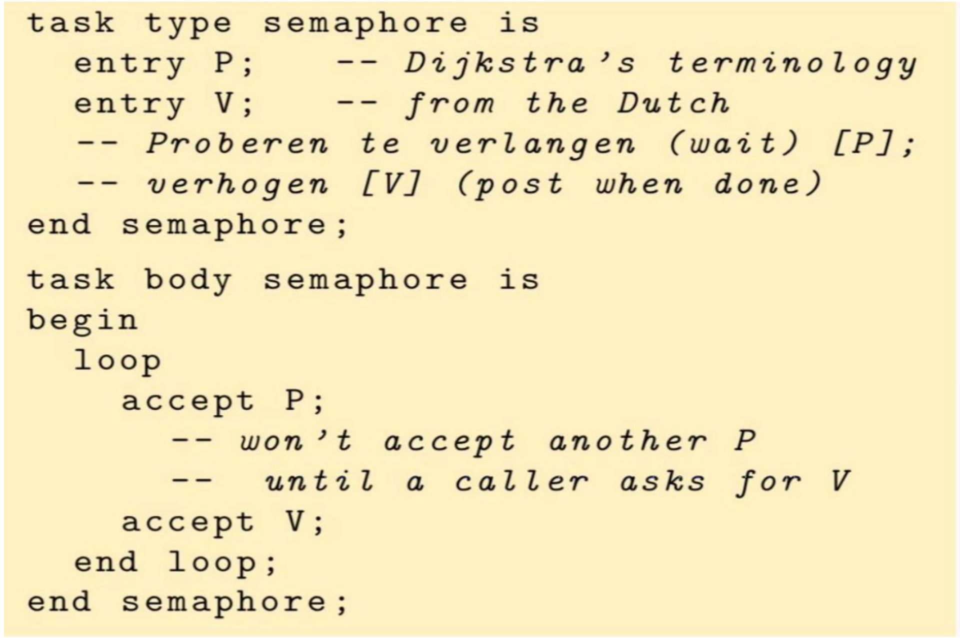
   instead of

   ```
   procedure <name> is
   ```

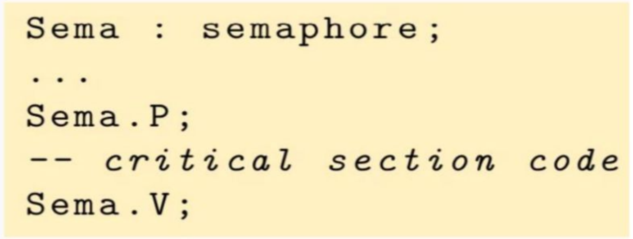
When the program is started, each task is started, and all tasks run in parallel with one another and with the main program. Of course, some tasks may terminate while others are still running. When all tasks terminate (including the main program), the overall program terminates.

Tasks in Ada:

Tasks can communicate with each other using entry and accept statements, which allow them to synchronize their activities and share data.



```
task type semaphore is
   entry P;     -- Dijkstra's terminology
   entry V;     -- from the Dutch
   -- Proberen te verlangen (wait) [P];
   -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
   loop
      accept P;
         -- won't accept another P
         --  until a caller asks for V
      accept V;
   end loop;
end semaphore;
```



```
Sema : semaphore;
...
Sema.P;
-- critical section code
Sema.V;
```

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
  task type Semaphore is
    entry P;
    entry V;
  end Semaphore;

  task body Semaphore is
    Locked : Boolean := False;
  begin
    loop
      accept P do
        if Locked then
          Put_Line("Semaphore: Task blocked");
          delay 0.1;
        else
          Locked := True;
          Put_Line("Semaphore: Task acquired");
        end if;
      end P;
      accept V do
        Locked := False;
        Put_Line("Semaphore: Task released");
      end V;
    end loop;
  end Semaphore;

  task type T1;
  task body T1 is
  begin
    Semaphore.P;
    Put_Line("T1: Critical section");
    delay 1.0;
    Semaphore.V;
  end T1;

  task type T2;
  task body T2 is
  begin
    Semaphore.P;
    Put_Line("T2: Critical section");
    delay 1.0;
    Semaphore.V;
  end T2;

begin
  declare
    S : Semaphore;
    Task1 : T1;
    Task2 : T2;
  begin
    null;
  end;
end Main;
```

In this example, we define a Semaphore task type that has two entry points: P and V. We also define two tasks T1 and T2, each with their own critical section of code.

In the Main procedure, we create an instance of Semaphore and two instances of T1 and T2. When each task executes its critical section of code, it calls the P entry of the Semaphore task to acquire the lock, and then calls the V entry to release the lock when it's finished.

Concurrency in Java:

In Java, a thread is a separate path of execution within a program. When you start a Java program, a thread is created to execute the main method. You can create additional threads to perform tasks concurrently with the main thread.

One way to create a new thread is to extend the Thread class and override the run method. Here's an example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getId());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

Another way to create a new thread is to implement the Runnable interface and pass it to a Thread constructor.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getId());
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

call to start activates thread, which executes run method. Do not call run directly: it will execute like an ordinary method with no new thread.

Volatile keyword in Java:

In Java, the volatile keyword can be used to declare a variable that should be accessed by multiple threads. When a variable is declared volatile, any changes to its value made by one thread are immediately visible to all other threads that access the variable. This ensures that all threads see the most up-to-date value of the variable.

In this example, we define a MyThread class that extends the Thread class and has a volatile boolean variable called stop. The run method of the thread checks the value of stop in a loop and exits the loop if the value is true. We also define a stopThread method that sets the value of stop to true to stop the thread.

```
class MyThread extends Thread {
   private volatile boolean stop = false;

   public void run() {
      while (!stop) {
         // Do some work here
      }
   }

   public void stopThread() {
      stop = true;
   }
}

public class Main {
   public static void main(String[] args) {
      MyThread thread = new MyThread();
      thread.start();

      // Wait for 5 seconds
      try {
         Thread.sleep(5000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      // Stop the thread
      thread.stopThread();
   }
}
```

Synchronised keyword in Java:

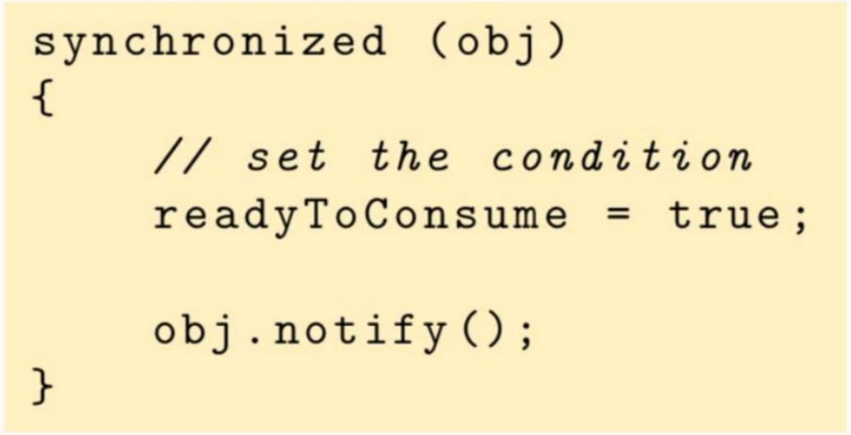
In Java, the synchronized keyword can be used to create synchronized blocks of code and methods that allow multiple threads to access them safely. When a method or block of code is declared as synchronized, only one thread can access it at a time. Other threads that try to access the same method or block of code must wait until the first thread has finished executing it.

We can also create synchronised blocks using synchronized (obj)

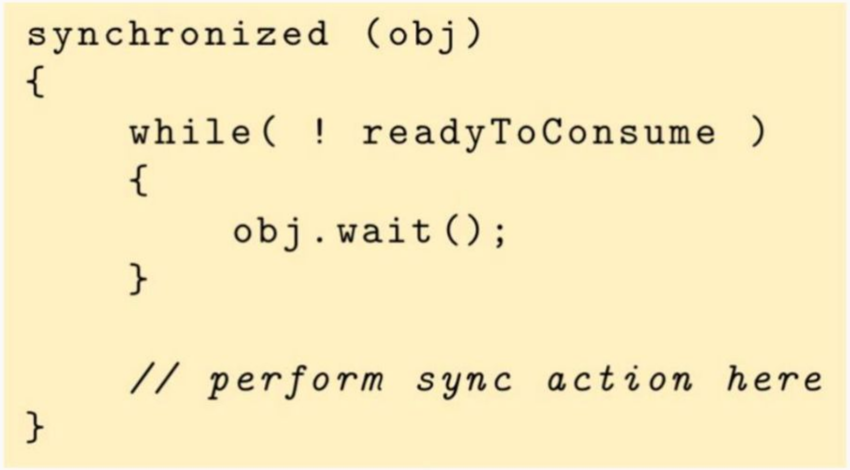
Without the synchronized keyword, the count variable could be accessed by multiple threads at the same time, leading to race conditions and unpredictable results. By using the synchronized keyword, we ensure that the count variable is accessed safely and consistently by all threads.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
    public synchronized void decrement() {
        count--;
    }
    public synchronized int getCount() {
        return count;
    }
}
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter.decrement();
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Count: " + counter.getCount());
    }
}
```

Java notify/wait pattern



```
synchronized (obj)
{
    // set the condition
    readyToConsume = true;

    obj.notify();
}
```



```
synchronized (obj)
{
    while( ! readyToConsume )
    {
        obj.wait();
    }

    // perform sync action here
}
```

In Java, the wait() method is used in synchronization to pause the current thread until another thread invokes the notify() or notifyAll() method for the same object. The wait() method must always be called inside a synchronized block or method, otherwise, an IllegalMonitorStateException will be thrown.

Asynchronous calls:

1. An asynchronous call refers to a programming pattern in which a function or method call does not block the calling thread of execution. Instead, the call returns immediately, allowing the caller to continue executing while the requested operation is performed in the background

2. This is in contrast to synchronous calls, which block the calling thread until the operation completes and the result is returned.

3. Asynchronous calls are often used in situations where the operation being performed may take a significant amount of time to complete, such as accessing a remote resource over a network or performing a complex computation. By performing the operation asynchronously, the calling thread can continue executing other tasks while waiting for the operation to complete.

Futures and Promises in C++:

In C++, futures and promises are used for asynchronous programming and concurrency.

A future is an object that represents a value that will be available at some point in the future.

A promise is an object that can be used to set a value to be returned by a future.

A promise and a future are related, with the promise setting the value of the future when it is ready.

In this example, a promise object is created for an integer value, and its corresponding future is retrieved using the get_future() method. A separate thread is started to set the value of the promise to 42, and the main thread waits for the future to become ready using the get() method. Finally, the result is output and the thread is joined.

```
#include <iostream>
#include <future>
int main() {
    // Create a promise for an int value
    std::promise<int> promise;

  // Get the future from the promise
    std::future<int> future = promise.get_future();

    // Create a lambda function to set the value of the future
    std::thread thread([&promise]() {
        // Set the value of the promise to 42
        promise.set_value(42);
    });

    // Wait for the future to become ready and get its value
    int result = future.get();

    // Output the result
    std::cout << "Result: " << result << std::endl;

    // Join the thread
    thread.join();

    return 0;
}
```

Async/Await

1. async/await is a programming language construct that enables asynchronous programming in a synchronous style. It is used to write non-blocking, asynchronous code that is easier to read and understand than traditional callback-based code.

2. In an async function, operations that take a long time to complete, such as network or disk I/O, are initiated and then immediately returned as a promise. The function then continues executing synchronously, without blocking the calling thread. When the long-running operation is complete, the promise is resolved, and the function can continue executing with the result of the operation.

3. The await keyword is used to pause the execution of the async function until the promise is resolved. When the promise is resolved, the result is returned, and the function resumes execution.

1. When an async function is called, it returns a promise immediately, before the function has finished executing. The function continues executing asynchronously in the background, while the main thread continues with other tasks.

2. When the async function completes its work, it resolves the promise with the result of the function, and the promise's then method is called with the result. This allows other functions in the program to continue executing while the async function is running in the background.

3. In this example, asyncFunc() is an asynchronous function that simulates a delay of 1 second using a Promise. When called, asyncFunc() logs a message to the console, waits for the Promise to resolve, logs another message to the console, and then returns a string.

4. When the program runs, the message "Before asyncFunc call" is logged to the console first, followed by the message "After asyncFunc call". The asyncFunc() function is called asynchronously in the background, and its messages are logged to the console during that time. Finally, when the Promise resolves, the result is logged to the console.

```
async function asyncFunc() {
  console.log('asyncFunc started');
  await new Promise(resolve => setTimeout(resolve, 1000));
  console.log('asyncFunc finished');
  return 'Async result';
}
console.log('Before asyncFunc call');
asyncFunc().then(result => console.log(result));
console.log('After asyncFunc call');
```

```
Before asyncFunc call
asyncFunc started
After asyncFunc call
asyncFunc finished
Async result
```