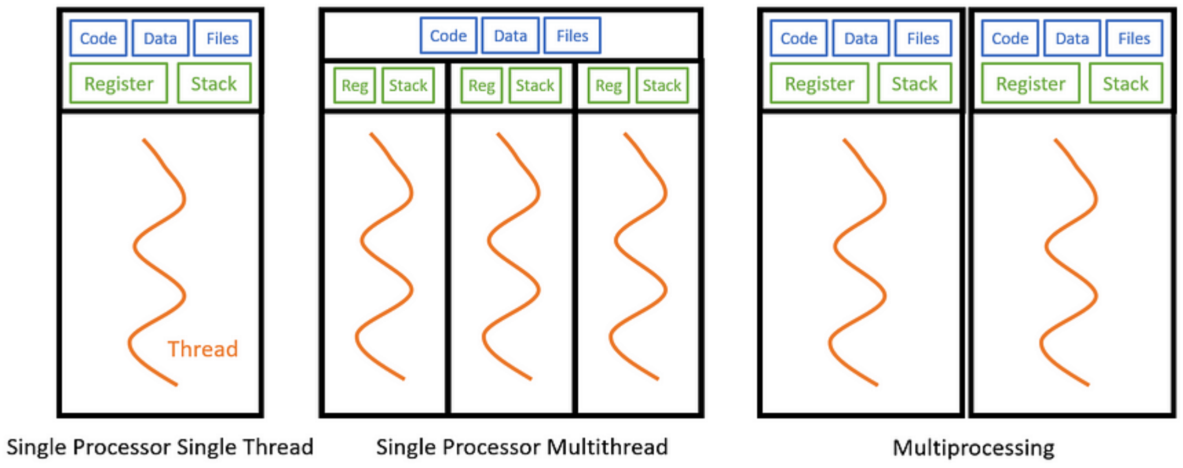
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Operating System Laboratory
Course Code:	DJ19CEL403
Experiment No.:	03

AIM: **BUILDING MULTI-THREADED AND MULTI-PROCESS APPLICATIONS**

THEORY:

MULTI-THREADING:
A Thread is an independent unit of execution created within the context of a process (or application that is being executed). When multiple threads are executing in a process at the same time, we get the term "multithreading."

MULTI-THREADED APPLICATIONS:
A multi-threaded application is an application whose architecture takes advantage of the multi-threading provided by the operating system. Usually, these applications assign specific jobs to individual threads within the process and the threads communicate, through various means, to synchronize their actions.



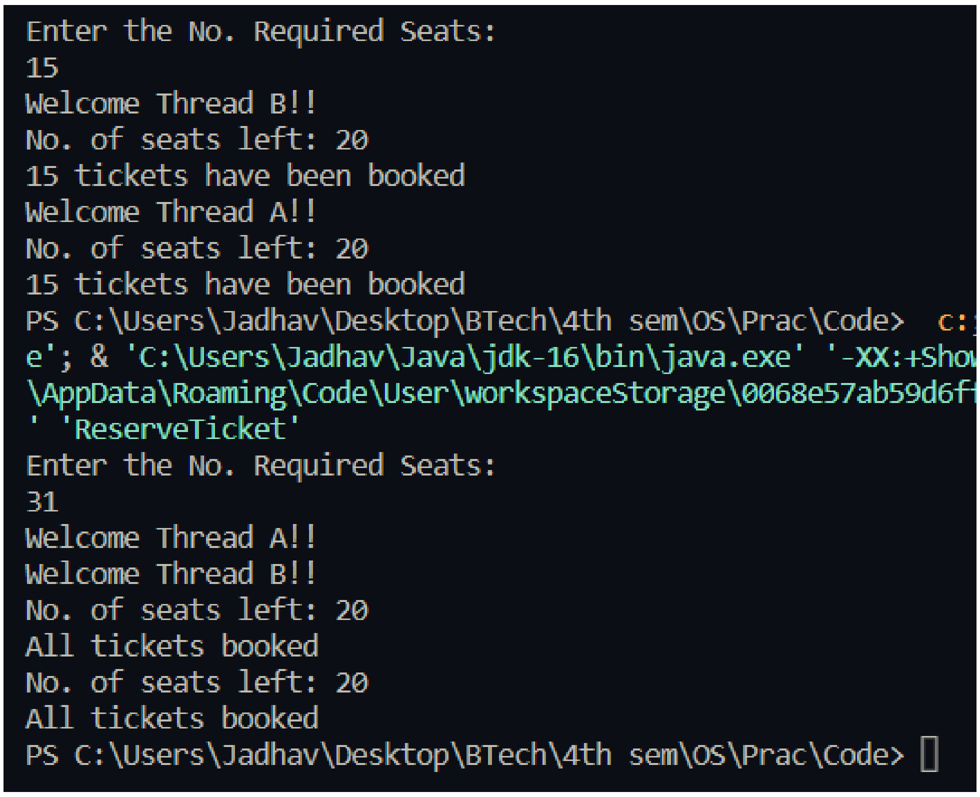
From the diagram above, we can see that in multithreading (middle diagram), multiple threads share the same code, data, and files but run on a different register and stack. Multiprocessing (right diagram) multiplies a single processor — replicating the code, data, and files, which incurs more overhead.

MULTI-PROCESS APPLICATIONS:
Multiprocessing refers to the ability of a system to run multiple processors concurrently, where each processor can run one or more threads. Multiprocessing is useful for CPU-bound processes, such as computationally heavy tasks since it will benefit from having multiple processors; similar to how multicore computers work faster than computers with a single core.

CODE 1:

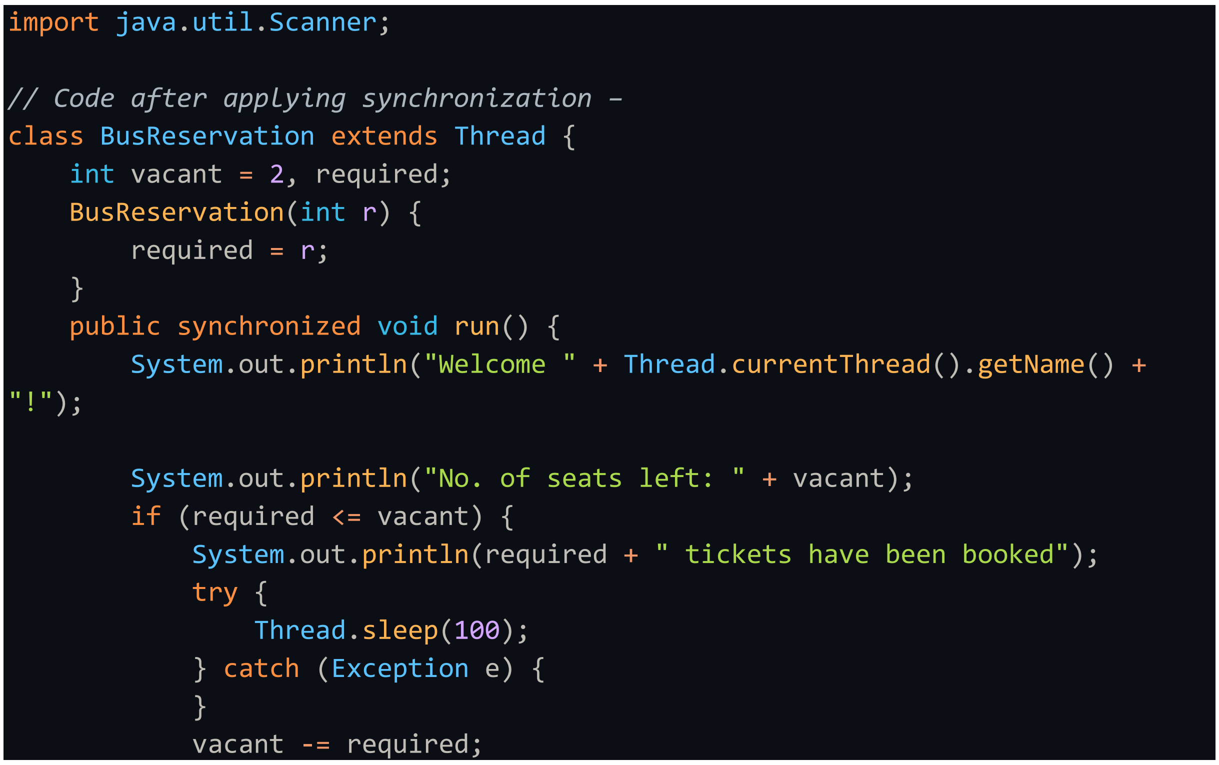
```
import java.util.Scanner;

// Code leading to inconsistency -
class BusReservation extends Thread {
    int vacant = 20, required;

    BusReservation(int r) {
        required = r;
    }

    public void run() {
        System.out.println("Welcome " + Thread.currentThread().getName() +
"!!");

        System.out.println("No. of seats left: " + vacant);
        if (required <= vacant) {
            System.out.println(required + " tickets have been booked");
            try {
                Thread.sleep(100);
            } catch (Exception e) {
            }
            vacant -= required;
        } else {
            System.out.println("All tickets booked");
        }
    }
}

class ReserveTicket {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the No. Required Seats: ");
        int req = sc.nextInt();
        BusReservation br = new BusReservation(req);
        Thread t1 = new Thread(br);
        Thread t2 = new Thread(br);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
        sc.close();
    }
}
```

OUTPUT 1:



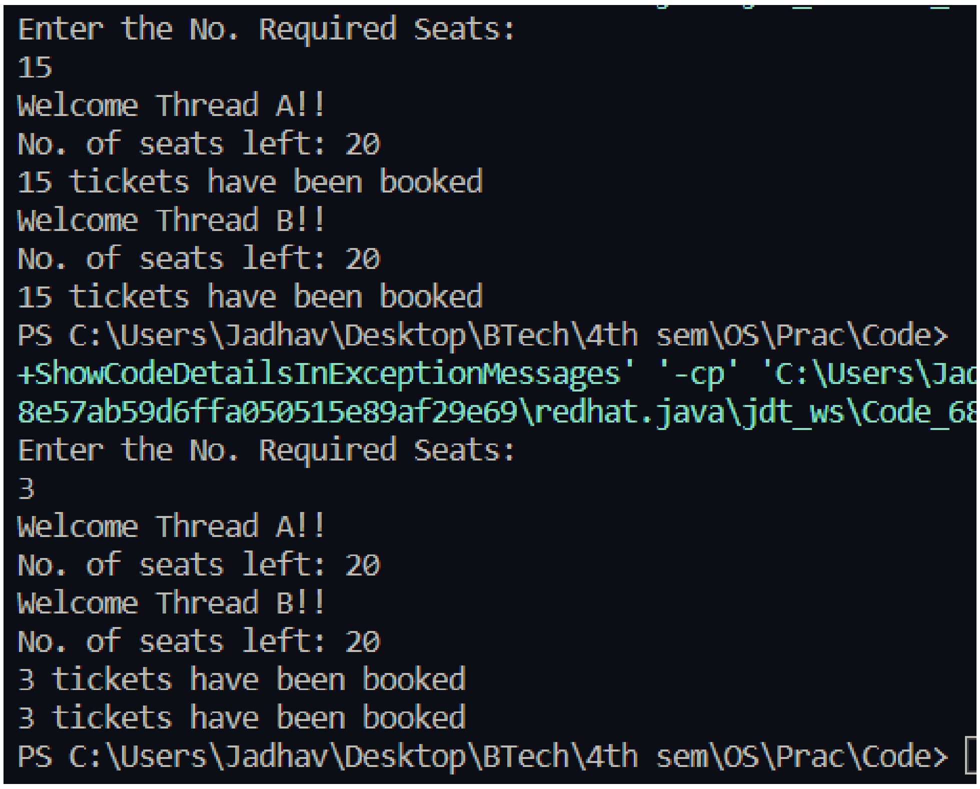
```
Enter the No. Required Seats:
15
Welcome Thread B!!
No. of seats left: 20
15 tickets have been booked
Welcome Thread A!!
No. of seats left: 20
15 tickets have been booked
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>  c:
e'; & 'C:\Users\Jadhav\Java\jdk-16\bin\java.exe' '-XX:+Show
\AppData\Roaming\Code\User\workspaceStorage\0068e57ab59d6f
' 'ReserveTicket'
Enter the No. Required Seats:
31
Welcome Thread A!!
Welcome Thread B!!
No. of seats left: 20
All tickets booked
No. of seats left: 20
All tickets booked
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code> 
```

CODE 2:



```java
import java.util.Scanner;

// Code after applying synchronization -
class BusReservation extends Thread {
    int vacant = 2, required;
    BusReservation(int r) {
        required = r;
    }
    public synchronized void run() {
        System.out.println("Welcome " + Thread.currentThread().getName() +
"!");

        System.out.println("No. of seats left: " + vacant);
        if (required <= vacant) {
            System.out.println(required + " tickets have been booked");
            try {
                Thread.sleep(100);
            } catch (Exception e) {
            }
            vacant -= required;
        } else {
            System.out.println("All tickets booked");
        }
    }
}

class ReserveTicketSync {
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the No. Required Seats: ");
        int req = sc.nextInt();
        BusReservation br = new BusReservation(req);
        Thread t1 = new Thread(br);
        Thread t2 = new Thread(br);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
        sc.close();
    }
}
```

OUTPUT 2:



```
Enter the No. Required Seats:
15
Welcome Thread A!!
No. of seats left: 20
15 tickets have been booked
Welcome Thread B!!
No. of seats left: 20
15 tickets have been booked
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Ja
8e57ab59d6ffa050515e89af29e69\redhat.java\jdt_ws\Code_6
Enter the No. Required Seats:
3
Welcome Thread A!!
No. of seats left: 20
Welcome Thread B!!
No. of seats left: 20
3 tickets have been booked
3 tickets have been booked
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

CONCLUSION:
When unsynchronized we don’t get the desired/expected output. But when we add the synchronized keyword in the run method we are able to achieve the desired output as synchronized keyword locks a single thread with the shared data so that no other thread can access it. The synchronized keyword can only synchronize a block or a method. A thread getting inside a synchronized block acquires the lock and releases it while leaving.