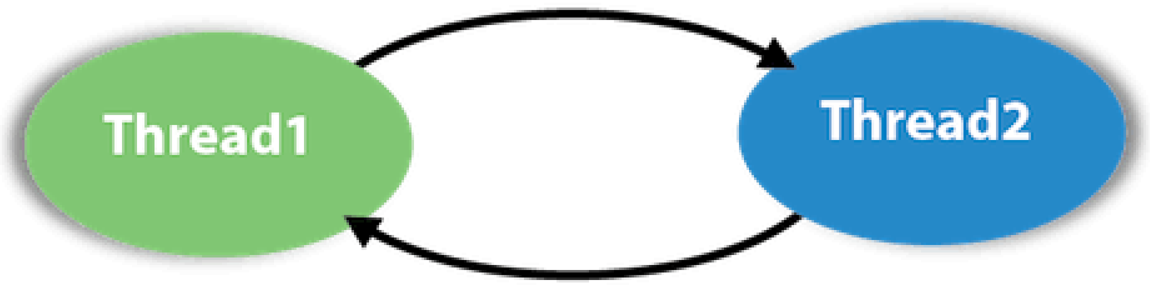
Deadlock

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called Deadlock(A Deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process)



Example of deadlock

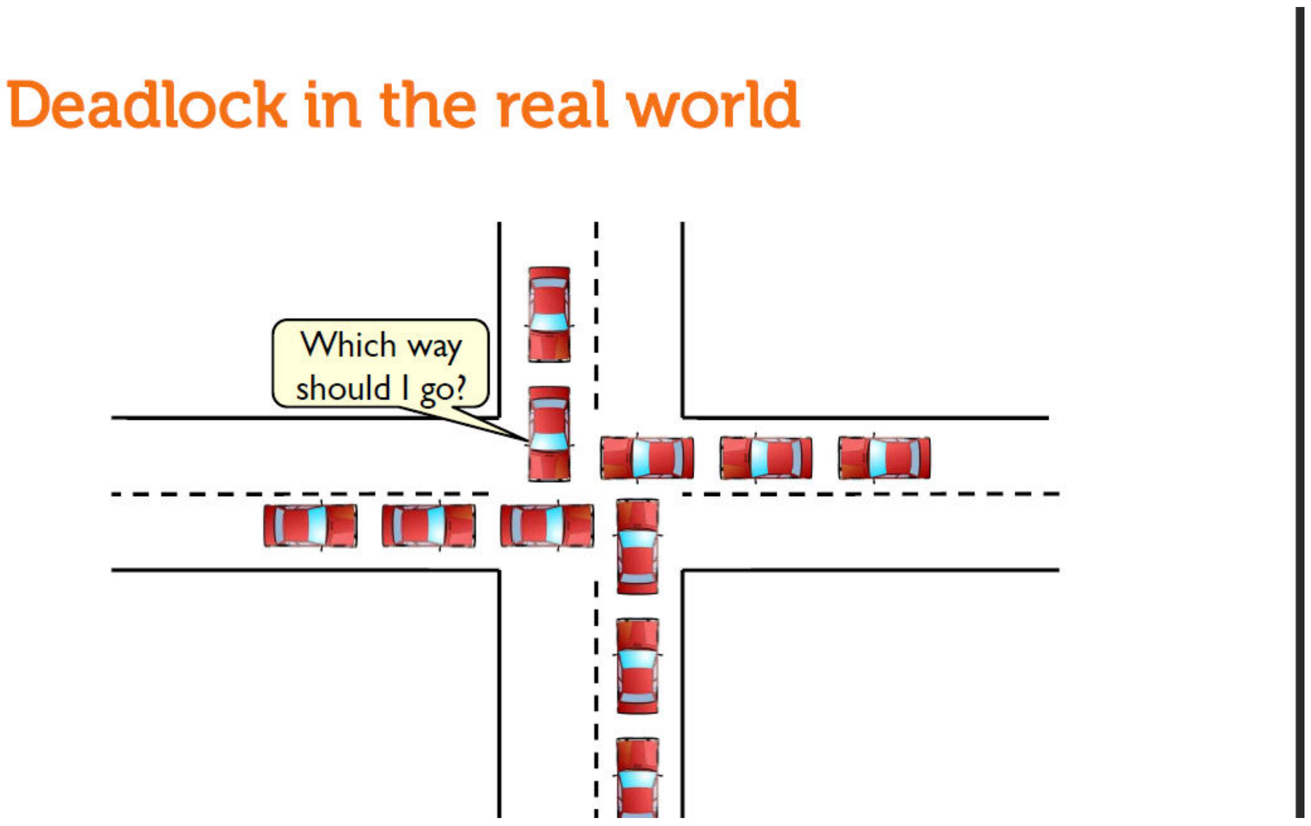
Perhaps the most easily recognizable form of deadlock is traffic gridlock. Multiple lines of cars are all vying for space on the road and the chance to get through an intersection, but it has become so back up there this no free space for potentially blocks around. This causes entire intersections or even multiple intersections to go into a complete stand still. Traffic can only flow in a single direction, meaning that there is nowhere for traffic to go once traffic has stopped. However, if the car at the very end of each line of traffic decides to back up, this frees up room for other cars to do the same and therefore the gridlock is solved. Another real-world example of deadlock is the use of a single track by multiple trains. Say multiple tracks converge onto one; there is a train on each individual track, leading to the one track. All trains are stopped, waiting for another to go, though none of them move. This is an example of deadlock because the resource, the train

Code:

```
        // Java program to illustrate Deadlock

// in multithreading.

class Util

{

        // Util class to sleep a thread
        static void sleep(long millis)
        {
                try
                {
                        Thread.sleep(millis);
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
        }
}

// This class is shared by both threads
class Shared
{
        // first synchronized method
        synchronized void test1(Shared s2)
        {
                System.out.println("test1-begin");
                Util.sleep(1000);

                // taking object lock of s2 enters
                // into test2 method
                s2.test2();
                System.out.println("test1-end");
        }

        // second synchronized method
        synchronized void test2()
        {
                System.out.println("test2-begin");
}
		Util.sleep(1000);
		// taking object lock of s1 enters
		// into test1 method
		System.out.println("test2-end");
	}
}


class Thread1 extends Thread
{
	private Shared s1;
	private Shared s2;

	// constructor to initialize fields
	public Thread1(Shared s1, Shared s2)
	{
		this.s1 = s1;
		this.s2 = s2;
	}

	// run method to start a thread
	@Override
	public void run()
	{
		// taking object lock of s1 enters
		// into test1 method
		s1.test1(s2);
	}
}


class Thread2 extends Thread
```

```
{
        private Shared s1;
        private Shared s2;

        // constructor to initialize fields
        public Thread2(Shared s1, Shared s2)
        {
                this.s1 = s1;
                this.s2 = s2;
        }

        // run method to start a thread
        @Override
        public void run()
        {
                // taking object lock of s2
                // enters into test2 method
                s2.test1(s1);
        }
}


public class Deadlock
{
        public static void main(String[] args)
        {
                // creating one object
                Shared s1 = new Shared();

                // creating second object
                Shared s2 = new Shared();
                // creating first thread and starting it

                Thread1 t1 = new Thread1(s1, s2);

                t1.start();


                // creating second thread and starting it

                Thread2 t2 = new Thread2(s1, s2);

                t2.start();


                // sleeping main thread

                Util.sleep(2000);

        }

}
```

- Thread t1 starts and calls test1 method by taking the object lock of s1.
- Thread t2 starts and calls test1 method by taking the object lock of s2.
- t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
- t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
- Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
- Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
- As none of them is ready to release lock, so this is the Dead Lock condition.
- When you will run this program, it will be look like execution is paused



Solution of deadlock

in sleeping barber deadlock solution

we have customers and barber and many customer need to have a cut hair and wait each other to cut

to solution it we add 3 semaphore and mutex(binary semaphore) so customer can be wait in queue(seats) and barber can be sleep if no customer there

Pseudocode

```
                Semaphore Customers  =  0;
Semaphore Barber = 0;
Mutex accessSeats = 1;
int NumberOfFreeSeats = N;

Barber {
   while(1) {
        sem_wait(Customers); // waits for a customer (sleeps)
        sem_wait(accessSeats); // mutex to protect the number of available seats
        NumberOfFreeSeats++; // a chair gets free
        sem_signal (Barber); // bring customer for haircut
        sem_signal (accessSeats); // release the mutex on the chair
          cuthair()
    }
}

Customer {
    while(1) {
        sem_wait(accessSeats); // protects seats so only 1 thread tries to sit in a chair if that's the case
        if(NumberOfFreeSeats > 0) {
            NumberOfFreeSeats--; // sitting down
            sem_signal (Customers); // notify the barber
            sem_signal (accessSeats); // release the lock
            cuthair( );
            // customer is having hair cut
        } else {
            sem_signal (accessSeats); // release the lock
             leave( );   // customer leaves
        }
    }
}
```

Starvation

is a problem of resource management where in the OS, the process does not have resources because it is being used by other processes. This problem occurs mainly in a priority-based scheduling algorithm where the requests with high priority get processed first and the least priority process takes time.

Example of starvation in the operating system :

High-priority low-priority

-

100

process	Arrival time	Bust time	Priority
P1	0ms	4ms	100ms
P2	0ms	7ms	1
P3	0ms	10ms	2
.....	.....	.....	High priority

In the given example, the P2 process has the highest priority and process P1 has the lowest priority. In the diagram, it can be seen that there is n number of processes ready for their execution. So in the CPU, the process with the highest priority will come in which is P2 and the process P1 will keep waiting for its turn because all other processes are at a high priority concerning that P1. This situation in which the process is waiting is called starvation.

Example of Starvation:

```
Class MyThread extends Thread {
 Public void run() {
 String threadName = Thread.currentThread().getName();
 System.out.println(threadName + " Started");
 Synchronized(MyThread.class) { // lock
 // doing some useful work
 Try {
 Thread.sleep(2000); // 2 sec
 } catch (InterruptedException ie){}
 }
 System.out.println(threadName + " End");
 }
}
Public class Test {
 Public static void main(String[] args) {
 System.out.println("Start of Main thread");
 MyThread mt[] = new MyThread[10];
 For (int i=0; i<mt.length; i++) {
 Mt[i] = new MyThread(); // create thread
 Mt[i].start();
 }
 System.out.println("End of Main thread");
```

In this example, the main thread created 10 child threads. To execute some portion of the run() method (synchronized block) each child thread needs the lock of the current class. At a time only one thread can get the lock of one object. And to complete execution, each thread required more than 2 seconds time.

Among these 10 threads, there will be some threads executing at last. They were waiting for a long period of time because the thread was unable to gain regular access to shared resources (lock of current class) and was unable to make progress.

In the above output, thread-8 waited for a long period of time

Conclusion :

-Starvation is a problem of resource management where in the OS, the process does not has resources because it is being used by other processes.

- It is a problem when the low-priority process gets jammed for a long duration of time because of high-priority requests being executed. A stream of high-priority requests stops the low-priority process from obtaining the processor or resources. Starvation happens usually when the method is delayed for an infinite period of duration.

Causes starvation in OS :

-In starvation, a process with low priority might wait forever if the process with higher priority uses a processor constantly.

-There are not enough resources to provide for every resource.

- If a process selection is random then there can be a possibility that a process might have to wait for a long duration.

- Resource is never provided to a process for execution due to faulty allocation of resources.

Solution of starvation :

-The allocation of resources by CPU should be taken care of by a freelance manager to ensure that there is an even distribution of resources.

-Random choice of process method should be avoided.

-The aging criteria of processes should be taken into consideration.Where Aging is a technique of gradually increasing the priority of processes that wait in the

system for a long time. For example, if priority range from 127(low) to 0(high), we

could increase the priority of a waiting process by 1 Every 15 minutes. Eventually,

even a process with an initial priority of 127 would take no more than 32 hours for

the priority 127 process to age to a priority-0 process

-If a process selection is random then there can be a possibility that a process might have to wait for a long duration.

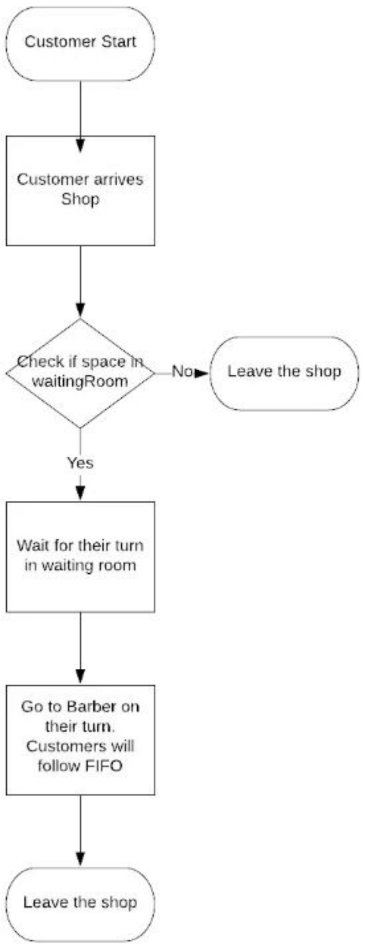
-Starvation can also occur when a resource is never provided to a process for execution due to faulty allocation of resources.

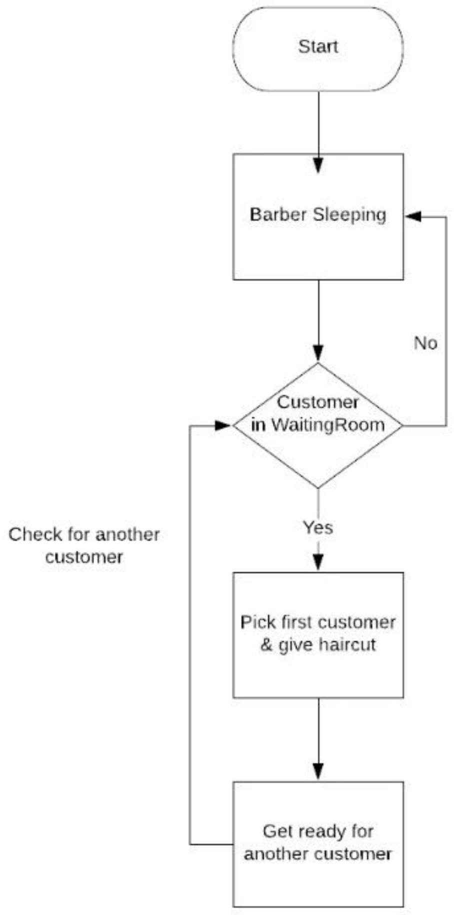
sleeping-Barber :

-Starvation: Then there might also be a case of starvation when the customers don't follow an order to cut their hair, as some people won't get a haircut even though they had been waiting a long.

-The solution to this problem involves the use of three semaphores out of which one is a mutex (binary semaphore). They are Customers: Helps count the waiting customers. Barber: To check the status of the barber, if he is idle or not. accessSeats: A mutex that allows the customers to get exclusive access to the number of free seats and allows them to increase or decrease the number. NumberOfFreeSeats: To keep the count of the available seats, so that the customer can either decide to wait if there is a seat free or leave if there is none.

FlowChart:


Customer Start
Customer arrives Shop
Check if space in waitingRoom
No
Leave the shop
Yes
Wait for their turn in waiting room
Go to Barber on their turn. Customers will follow FIFO
Leave the shop


Start
Barber Sleeping
No
Customer in WaitingRoom
Yes
Check for another customer
Pick first customer & give haircut
Get ready for another customer

Real World Example:

This design is the best analogy for a customer care call centre. Initially when there is no customer on-call all call-executives just relax and wait for the call. The moment the first customer dials the number he/she is connected to any call-executive and in a scenario when all call-executives are busy the customer will have to wait in a queue till they are assigned to a call-executive. If all executives are busy and the waiting line is full, the customers are disconnected with a message that executives are busy and customers will be contacted later by the company. This best relates to this design as the customers are picked from the queue in a first come first serve basis and call-executives are utilized in such a way that everyone executive gets at least one call.

In this scenario we can have the following design similarities:

1. The critical section will be the call between executive and customer

2. The waiting room will be the waiting queue over a call, where customers will be held in a FIFO manner.

3. Locks can be acquired on the waiting queue so that no two executives pick the same customer.