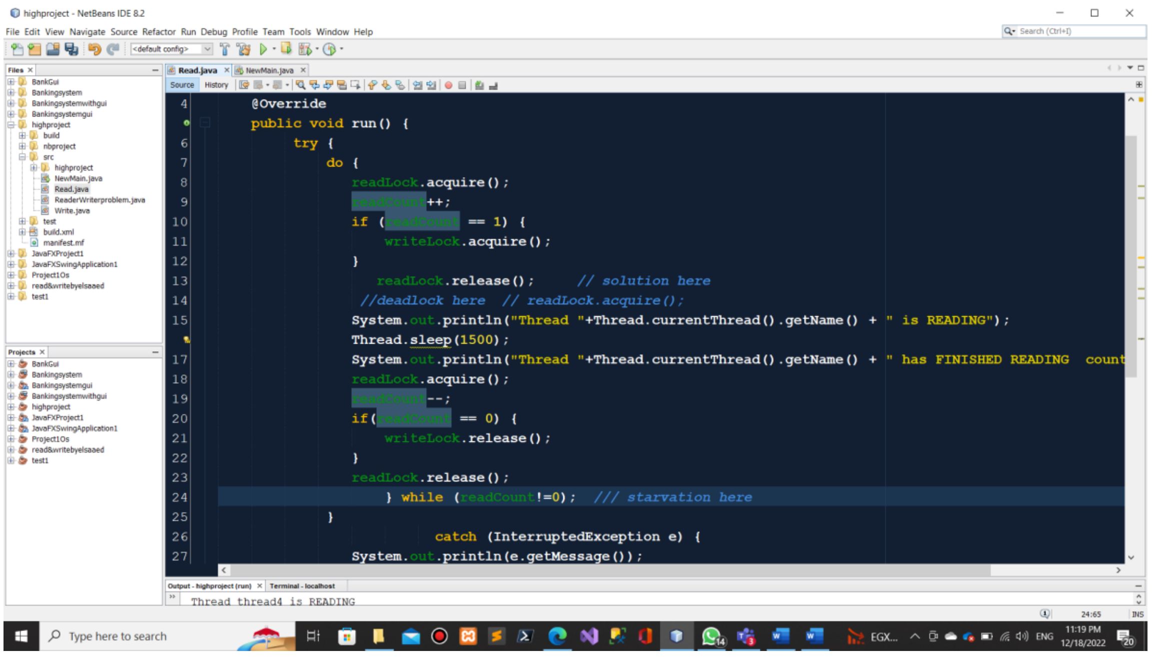
Project#4

Readers And writers Problem

Starvation



```
        @Override
        public void run() {
            try {
                do {
                    readLock.acquire();
                    readCount++;
                    if (readCount == 1) {
                        writeLock.acquire();
                    }
                       readLock.release();      // solution here
                     //deadlock here  // readLock.acquire();
                    System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                    Thread.sleep(1500);
                    System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING  count
                    readLock.acquire();
                    readCount--;
                    if(readCount == 0) {
                        writeLock.release();
                    }
                    readLock.release();
                        } while (readCount!=0);  /// starvation here
                }
                            catch (InterruptedException e) {
                    System.out.println(e.getMessage());
```

```
Thread thread4 is READING
```

Example about Starvation

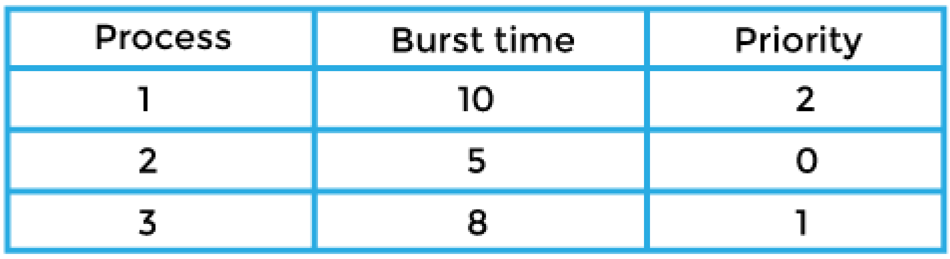
Starvation and Aging in Operating Systems

Generally, Starvation occurs in ***Priority Scheduling*** or ***Shortest Job First Scheduling***. In the Priority scheduling technique, we assign some priority to every process we have, and based on that priority, the CPU will be allocated, and the process will be executed. Here, the CPU will be allocated to the process that has the highest priority.

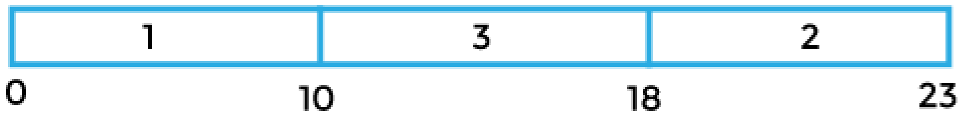
Even if the burst time is low, the CPU will be allocated to the highest priority process.

Starvation is very bad for a process in an operating system, but we can overcome this starvation problem with the help of Aging. What is Starvation?

Starvation or indefinite blocking is a phenomenon associated with the Priority scheduling algorithms. A process that is present in the ready state and has low priority keeps waiting for the CPU allocation because some other process with higher priority comes with due respect time. Higher-priority processes can prevent a low-priority process from getting the CPU.



Process	Burst time	Priority
1	10	2
2	5	0
3	8	1



For example, the above image process has higher priority than other processes getting CPU earlier. We can think of a scenario in which only one process has very low priority (for example, 127), and we are giving other processes high priority. This can lead to indefinitely waiting for the process for CPU, which is having low-priority, which leads to **Starvation**.

Causes of Starvation

[redacted]

Some of the common causes of Starvation in the operating system are as follows: ○ When Starvation occurs, there are not enough resources to go around, and the priority of the processes starts becoming low.

- ○ A lower priority process may wait forever if higher priority processes constantly monopolize the processor.Since the low priority programs are not interacting with anything, it becomes impossible for Starvation to cause a deadlock.
- ○ If a random selection of processes is used, then a process may wait for a long time because of nonselection.
- ○ Starvation is a fail-safe method to get out of a deadlock, making it much more important how it affects the system as a whole.
- ○ If a process is never provided the resources, it is required for execution because of faulty resource allocation decisions, and Starvation can occur. ○ Starvation may occur if there are not enough resources to provide to every process as required. Solutions to Handle Starvation

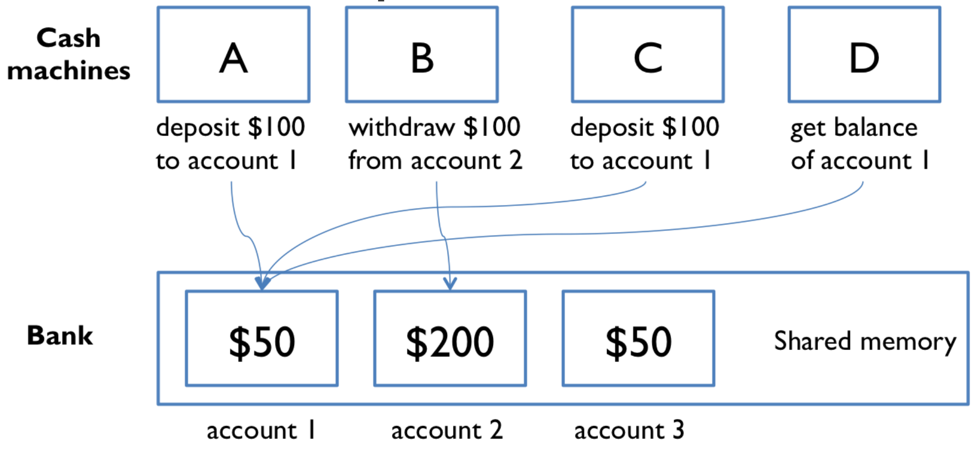
Some solutions that can be implemented in a system to handle Starvation are as follows:

- ○ An independent manager can be used for the allocation of resources. This resource manager distributes resources fairly and tries to avoid Starvation.
- ○ Random selection of processes for resource allocation or processor allocation should be avoided as they encourage Starvation. ○ The priority scheme of

resource allocation should include concepts such as Aging, where the priority of a process is increased the longer it waits, which avoids Starvation.

Real World ex(Reader_Writer)

Bank account example



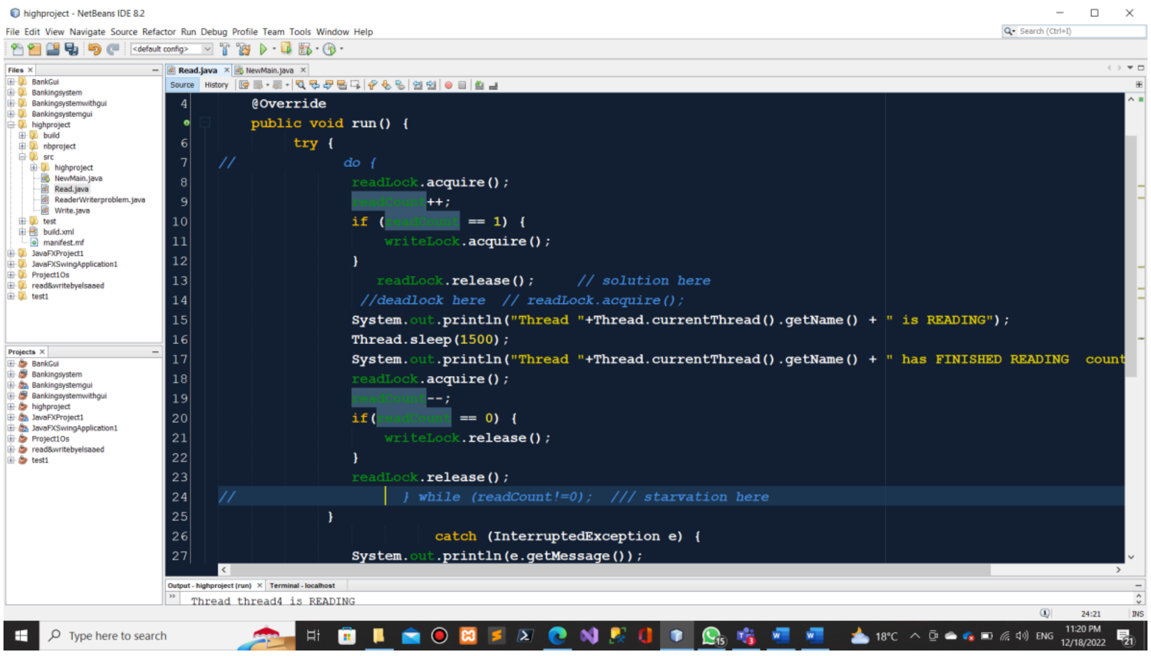
Our example of shared memory concurrency was a bank with cash machines. The diagram from that example is on the right.

The bank has several cash machines, all of which can read and write the same account objects in memory.

Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong (Race condition).

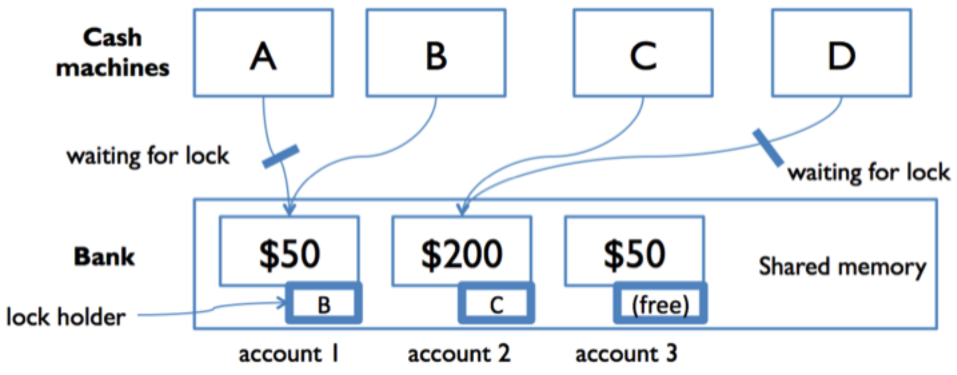
The solution:

To solve this problem with locks, we can add a lock that protects each bank account. Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.



```
        @Override
        public void run() {
            try {
//              do {
                readLock.acquire();
                readCount++;
                if (readCount == 1) {
                    writeLock.acquire();
                }
                   readLock.release();      // solution here
                 //deadlock here  // readLock.acquire();
                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING  count
                readLock.acquire();
                readCount--;
                if(readCount == 0) {
                    writeLock.release();
                }
                readLock.release();
//                  } while (readCount!=0);  /// starvation here
            }
                        catch (InterruptedException e) {
                System.out.println(e.getMessage());
```

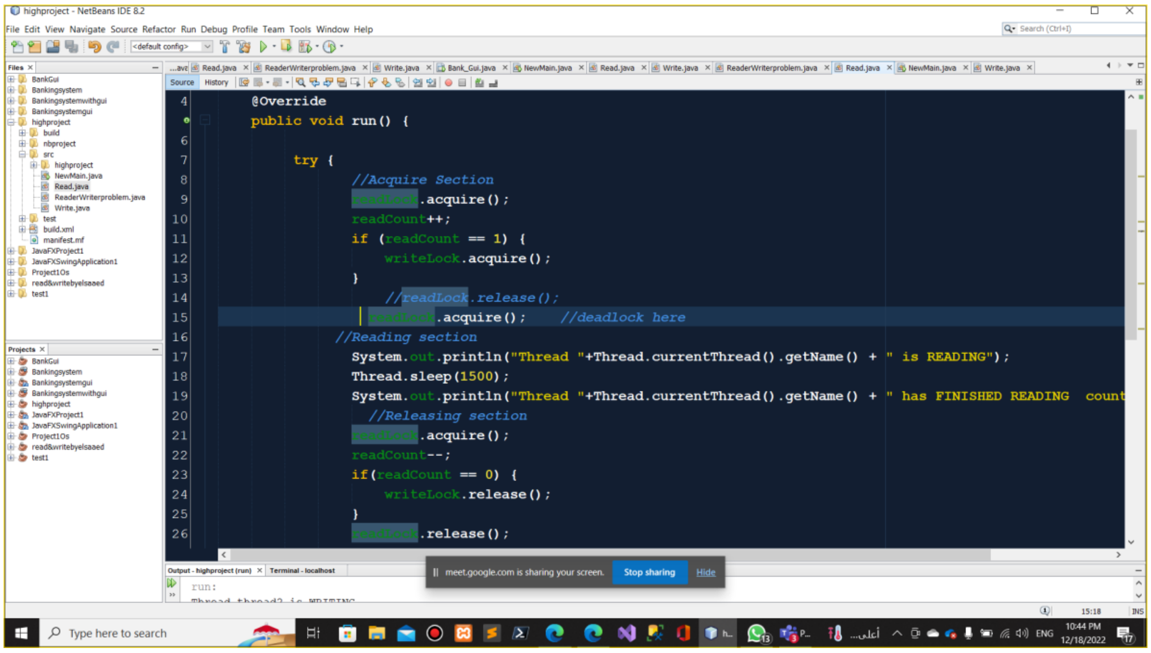
```
Thread thread4 is READING
```



In the diagram to the right, both A and B are trying to access account 1. Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

Deadlock

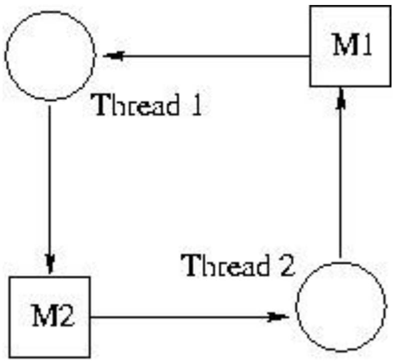
Examples of Deadlock


highproject - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Source History
Output - highproject (run)
Terminal - localhost
meet.google.com is sharing your screen.
Stop sharing
Hide
Type here to search

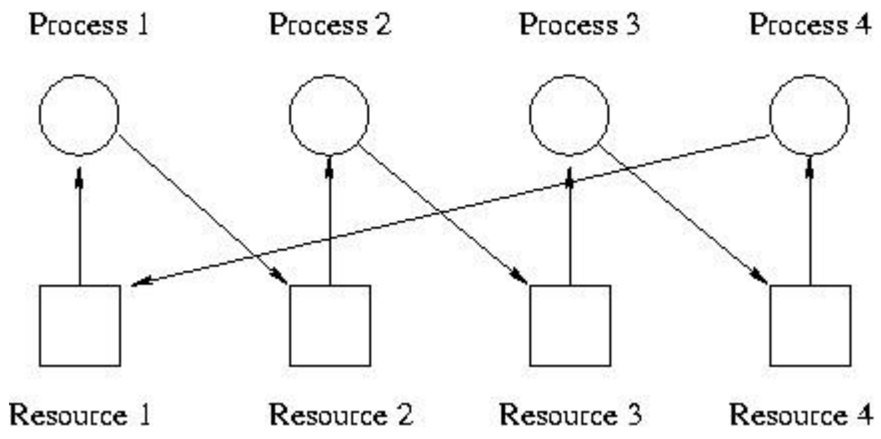
```
    @Override
    public void run() {

        try {
            //Acquire Section
            readLock.acquire();
            readCount++;
            if (readCount == 1) {
                writeLock.acquire();
            }
              //readLock.release();
             readLock.acquire();    //deadlock here
          //Reading section
            System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
            Thread.sleep(1500);
            System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING  count
              //Releasing section
            readLock.acquire();
            readCount--;
            if(readCount == 0) {
                writeLock.release();
            }
            readLock.release();
```

```
run:
Thread thread2 is WRITING
```

The deadlock situation can be modeled like this:



This graph shows an extremely simple deadlock situation, but it is also possible for a more complex situation to create deadlock. Here is an example of deadlock with four processes and four resources.



There are a number of ways that deadlock can occur in an operating situation. We have seen some examples, here are two more.

- Two processes need to lock two files, the first process locks one file the second process locks the other, and each waits for the other to free up the locked file.
- Two processes want to write a file to a print spool area at the same time and both start writing. However, the print spool area is of fixed

  Two process are looping forever and prevent each other from entering the critical section.

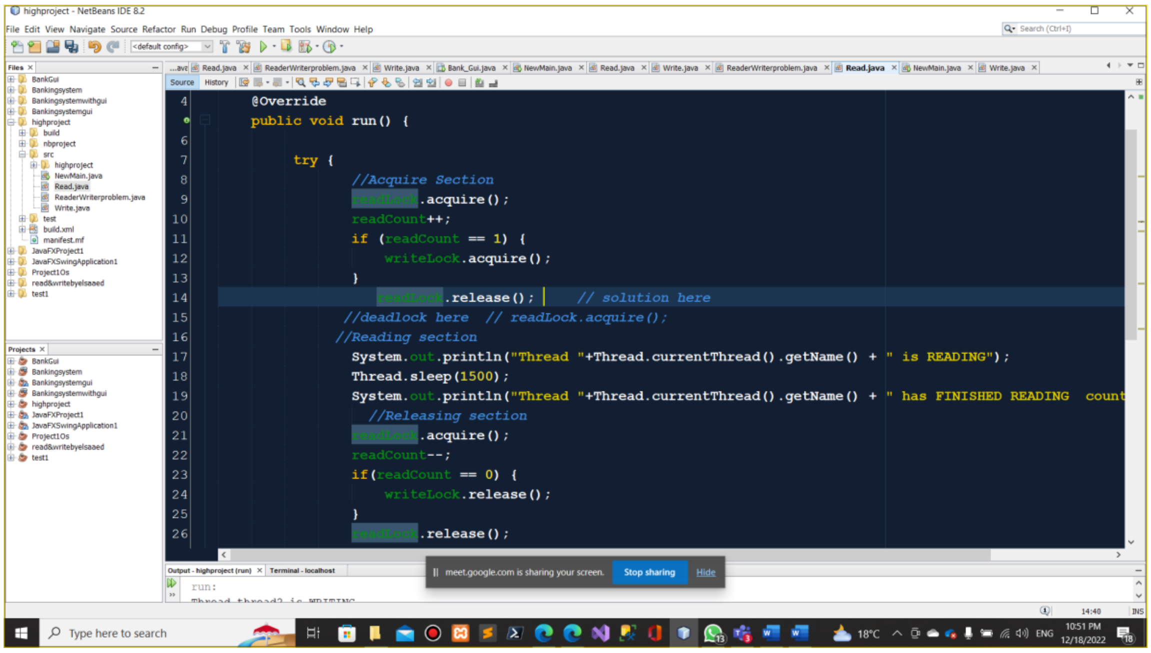
  size, and it fills up to become available

□

____

____

The solutions



```java
    @Override
    public void run() {

        try {
            //Acquire Section
            readLock.acquire();
            readCount++;
            if (readCount == 1) {
                writeLock.acquire();
            }
               readLock.release();       // solution here
           //deadlock here  // readLock.acquire();
          //Reading section
            System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
            Thread.sleep(1500);
            System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING  count
              //Releasing section
            readLock.acquire();
            readCount--;
            if(readCount == 0) {
                writeLock.release();
            }
            readLock.release();
```

```
run:
Thread thread2 is WRITING
```

The following are the four conditions that must hold simultaneously for a deadlock to occur:

1. **Mutual Exclusion –** A resource can be used by only one process at a time. If another process requests for that resource then the requesting process must be delayed until the resource has been released.
2. **Hold and wait –** Some processes must be holding some resources in non shareable mode and at the same time must be waiting to acquire some more resources, which are currently held by other processes in non-shareable mode.
3. **No pre-emption –** Resources granted to a process can be released back to the system only as a result of voluntary action of that process, after the process has completed its task.
4. **Circular wait –** Deadlocked processes are involved in a circular chain such that each process holds one or more resources being requested by the next process in the chain.

There are several ways to address the problem of deadlock in an operating system.

- Just ignore it and hope it doesn't happen Detection and recovery - if it happens, take action

- Dynamic avoidance by careful resource allocation. Check to see if a resource can be granted, and if granting it will cause deadlock, don't grant it.
- Prevention - change the rules

The solution mathods

Methods of handling deadlocks : There are three approaches to deal with deadlocks.

1. Deadlock Prevention
2. Deadlock avoidance
3. Deadlock detection

These are explained as following below.

1. <u>Deadlock Prevention</u> : The strategy of deadlock prevention is to design the system in such a way that the possibility of deadlock is excluded. Indirect method prevent the occurrence of one of three necessary condition of deadlock i.e., mutual exclusion, no pre-emption and hold and wait. Direct method prevent the occurrence of circular wait. **Prevention techniques – Mutual exclusion –** is supported by the OS. **Hold and Wait –** condition can be prevented by requiring that a process requests all its required resources at one time and blocking the process until all of its requests can be granted at a same time simultaneously. But this prevention does not yield good result because :

- long waiting time required
- in efficient use of allocated resource
- A process may not know all the required resources in advance

No pre-emption – techniques for ‘no pre-emption are’

- If a process that is holding some resource, requests another resource that can not be immediately allocated to it, the all resource currently being held are released and if necessary, request them again together with the additional resource.
- If a process requests a resource that is currently held by another process, the OS may pre-empt the second process and require it to release its resources. This works only if both the processes do not have same priority.

Circular wait One way to ensure that this condition never hold is to impose a total ordering of all resource types and to require that each process requests resource in an increasing order of enumeration, i.e., if a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in ordering.

2. **Deadlock Avoidance :** This approach allows the three necessary conditions of deadlock but makes judicious choices to assure that deadlock point is never reached. It allows more concurrency than avoidance detection A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to deadlock. It requires the knowledge of future process requests. Two techniques to avoid deadlock :

1. Process initiation denial
2. Resource allocation denial

---

3. **Deadlock Detection :** Deadlock detection is used by employing an algorithm that tracks the circular waiting and killing one or more processes so that deadlock is removed. The system state is examined periodically to determine if a set of processes is deadlocked. A deadlock is resolved by aborting and restarting a process, relinquishing all the resources that the process held.

- This technique does not limit resources access or restrict process action.
- Requested resources are granted to processes whenever possible.
- It never delays the process initiation and facilitates online handling.
- The disadvantage is the inherent preemption losses.

Solution pseudocode

```
in the critical section,    if
(readcnt == 0)
signal(wrt);         //
writers can enter

   signal(mutex); // reader leaves

} while(true); Writer
process:

 do

{

    // writer requests for critical
section     wait(wrt);

    // performs the write

    // leaves the critical section
signal(wrt);

} while(true);
```

Reader process:

```
do {
//
Reade
r
wants
to
enter
the
criti
cal
secti
on
wait(
mutex
);

   // The number of readers has now
increased by 1
   readcnt++;
   // there is atleast one reader in
the critical section
```

```
   // this ensure no writer can enter
if there is even one reader    //
thus we give preference to readers
here    if (readcnt==1)
      wait(wrt);

   // other readers can enter while
this current reader is inside
// the critical section
signal(mutex);

   // current reader performs reading
here
   wait(mutex);   // a reader wants
to leave

   readcnt--;

   // that is, no reader is left
```