Readers-Writers Problem

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e., they only want to read the data from the object and some of the processes are writers i.e., they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However, if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e., when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores.

# Starvation:

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **resource starvation** is a problem encountered in [concurrent computing](https://en.wikipedia.org/wiki/Concurrent_computing) where a [process](https://en.wikipedia.org/wiki/Computer_process) is perpetually denied necessary [resources](https://en.wikipedia.org/wiki/Resource_(computer_science)) to process its work. Starvation may be caused by errors in a scheduling or [mutual exclusion](https://en.wikipedia.org/wiki/Mutual_exclusion) algorithm, but can also be caused by [resource leaks](https://en.wikipedia.org/wiki/Resource_leak), and can be intentionally caused via a [denial-of-service attack](https://en.wikipedia.org/wiki/Denial-of-service_attack) such as a [fork bomb](https://en.wikipedia.org/wiki/Fork_bomb).

When starvation is impossible in a [concurrent algorithm](https://en.wikipedia.org/wiki/Concurrent_algorithm), the algorithm is called **starvation-free**, **lockout-freed** or said to have **finite bypass**. This property is an instance of [liveness](https://en.wikipedia.org/wiki/Liveness), and is one of the two requirements for any mutual exclusion algorithm; the other being [correctness](https://en.wikipedia.org/wiki/Correctness_(computer_science)). The name "finite bypass" means that any process (concurrent part) of the algorithm is bypassed at most a finite number times before being allowed access to the [shared resource](https://en.wikipedia.org/wiki/Shared_resource) .

Starvation is usually caused by an overly simplistic [scheduling algorithm](https://en.wikipedia.org/wiki/Scheduling_algorithm). For example, if a (poorly designed) [multi-tasking system](https://en.wikipedia.org/wiki/Computer_multitasking) always switches between the first two tasks while a third never gets to run, then the third task is being starved of [CPU time](https://en.wikipedia.org/wiki/CPU_time). The scheduling algorithm, which is part of the [kernel](https://en.wikipedia.org/wiki/Kernel_(computer_science)), is supposed to allocate resources equitably; that is, the algorithm should allocate resources so that no process perpetually lacks necessary resources.

Many operating system schedulers employ the concept of process priority. A high priority process A will run before a low priority process B. If the high priority process (process A) blocks and never yields, the low priority process (B) will (in some systems) never be scheduled—it will experience starvation. If there is an even higher priority process X, which is dependent on a result from process B, then process X might never finish, even though it is the most important process in the system. This condition is called a [priority inversion](https://en.wikipedia.org/wiki/Priority_inversion). Modern scheduling algorithms normally contain code to guarantee that all processes will receive a minimum amount of each important resource (most often CPU time) in order to prevent any process from being subjected to starvation.

In computer networks, especially wireless networks, [scheduling algorithms](https://en.wikipedia.org/wiki/Scheduling_algorithm) may suffer from scheduling starvation. An example is [maximum throughput scheduling](https://en.wikipedia.org/wiki/Maximum_throughput_scheduling).

Starvation is normally caused by [deadlock](https://en.wikipedia.org/wiki/Deadlock) in that it causes a process to freeze. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set. On the other hand, a process is in starvation when it is waiting for a resource that is continuously given to other processes. Starvation-freedom is a stronger guarantee than the absence of deadlock: a mutual exclusion algorithm that must choose to allow one of two processes into a [critical section](https://en.wikipedia.org/wiki/Critical_section) and picks one arbitrarily is deadlock-free, but not starvation-free.

A possible solution to starvation is to use a scheduling algorithm with priority queue that also uses the [aging](https://en.wikipedia.org/wiki/Aging_(scheduling)) technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

to pervent starvation we use semaphore , semaphore is lock but we can make some number of process to do it by put argument in the semaphore and that mean the number of process that will work in parallel

# Deadlock:

A deadlock is a situation in which two computer [programs](https://www.techtarget.com/searchsoftwarequality/definition/program) sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

The earliest computer [operating systems](https://www.techtarget.com/whatis/definition/operating-system-OS) ran only one program at a time. All the resources of the system were available to this one program. Later, operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running. This led to the problem of the deadlock.

We aren’t making synchronization between two object like this

|  |  |
| --- | --- |
| Thread1 | Thread2 |
| synchronized (object1) { // do something here synchronized (object2) { // do something here } } | synchronized (object1) { // do something here synchronized (object2) { // do something here } } |

Pseudocode

Write process

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

Read process

do {

// Reader wants to enter the critical section

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 in the critical section, if (readcnt == 0) signal(wrt);

// writers can enter signal(mutex);

// reader leaves }

while(true);