# **READERS WRITERS PROBLEM**: The readers-writers problem is a classical problem of process synchronization; it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers- can both read and write in the data sets.

# The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

# Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

Let us understand the possibility of reading and writing with the table given below

**TABLE 1**

**Case Process 1 Process 2 Allowed / Not Allowed**

**Case 1 Writing Writing Not Allowed**

**Case 2 Reading Writing Not Allowed**

**Case 3 Writing Reading Not Allowed**

**Case 4 Reading Reading Allowed**

# Solution pseudocode

**Reader**

wait(enter);

s\_reader <-- s\_reader+1;

signal(enter);

wait(enter2);

c\_reader <-- c\_reader+1;

if (w\_writer && s\_reader==c\_reader ) then

signal(writer);

end if

signal(enter2);

**writer**

wait(enter);

wait(enter2);

if(s\_reader==c\_reader) then

signal (enter2);

else {

w\_writer :=true;

signal(enter2);

wait(writer);

w\_writer :=false; }

end if

signal(enter);

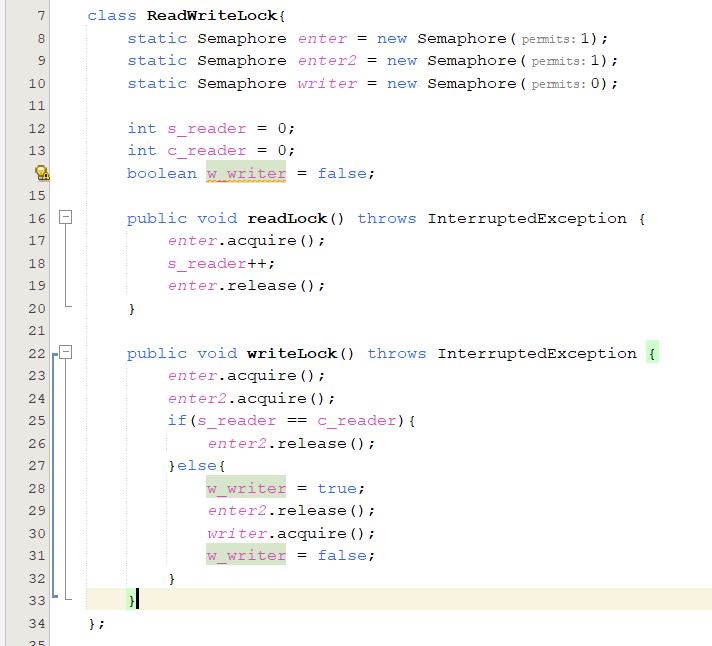
**Deadlock**

Deadlock occurs when each process holds a resource and wait for other resource held by any other process. Necessary conditions for deadlock to occur are Mutual Exclusion, Hold and Wait, No Preemption and Circular Wait. In this no process holding one resource and waiting for another get executed. For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1. Hence both process 1 and process 2 are in deadlock.

Diagram

Description automatically generated

# Example of deadlock



If we suppose that there is no read unlock or write unlock functions in this situation, a deadlock will result if the reader takes the lock and then finishes reading (critical section) and the lock is not released to allow the writer to enter the critical section.

**The output will be as the following:**

Graphical user interface, text, application, email

Description automatically generated

# How did we solve deadlock

Graphical user interface, text, application, email

Description automatically generated

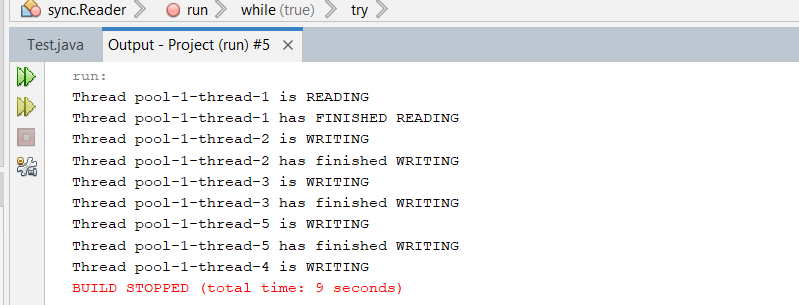
The solution is to use the function read unlock to release the process once the reader has finished reading so that the writer can access the critical section.

Graphical user interface, text, application, email

Description automatically generated

The writer will do the same thing after complete writing by calling the function writer unlock to enable readers to go on to the essential part.

**Thus, the program will run correctly:**

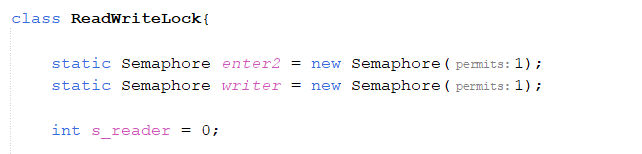


**Starvation**

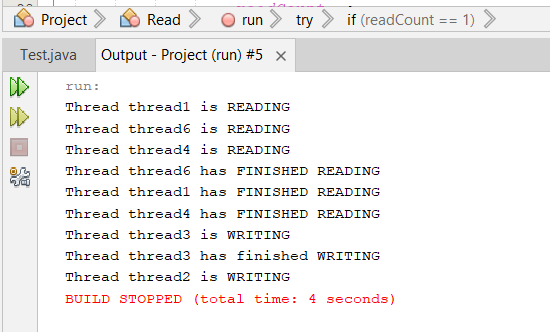
Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Problem of starvation can be resolved using Aging. In Aging priority of long waiting processes is gradually increased.

# Example of Starvation

**Using 2 semaphore and 1 integer variable**



if a reader is waiting for a writer process to signal the **writer**, all the other readers are waiting on **enter2**. After the writer process signals the **enter2**, all the readers can simultaneously perform the read operations. Till all the readers are done, all the writers are paused on **writer**, thus, causing starvation.

**The output will be as the following:**

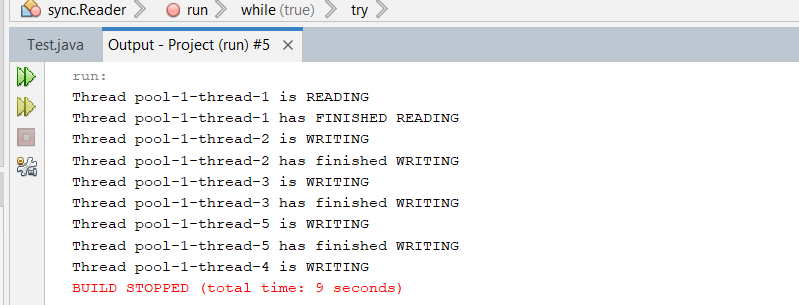
# How did we solve starvation

**Graphical user interface, text, application

Description automatically generatedUsing 3 semaphore and 2 integer variable and a Boolean variable**

**Implementation writer:**  the queue of **enter** serves as a common waiting queue for the readers and the writers. And an if condition that compares the variables **s\_reader** and **c\_reader**, If they are equal it means no reader is executing in its critical section currently. the writer simply continues with its critical section. If the variables **s\_reader** and **c\_reader** are not equal there is are reader processes executing their critical sections, then, the writer changes the variable **W\_writer** to true to state its presence

**Implementation reader:** we have the **enter** Again, all the readers and writers must queue in this mutex to ensure equal priority. then it goes to check if any writer is waiting by checking the variable **W\_writer** If yes, it checks if any fellow readers are executing in their critical sections. If not, it signals **writer** semaphore it signals the **enter2** to allow writer to enter the critical section.

**Thus, the program will run correctly:**

* Explanation for real world application and how did apply the problem

Consider **cinema booking system** in which many clients are attempting to book seats in cinema hall. All the information about seats is stored in a common database in the memory. The database consists of many entries, each representing a seat in cinema hall. In a typical Cinema booking system scenario, the client will probe around in the database looking for the "optimal" seat to meet that client's needs. So, a client may examine the database many times before deciding to try and book a particular seat. A seat that was available during this searching phase could easily be booked by someone else before the client has a chance to book it after deciding on it. In that case, when the client attempts to make the reservation, the client will discover that the data has changed, and the seat is no longer available. The client probing around the database is **called a reader**. The client attempting to book the seat is **called a writer.** Clearly, any number of readers can be probing shared data at once, but each writer needs exclusive access to the shared data to prevent the data from being corrupted.

We applied to the problem on this real-life application by Writing a multithreaded Java program that launches multiple reader threads and multiple writer threads, each attempting to access a single reservation record. A writer thread has two possible transactions, make Reservation and cancel Reservation. A reader has one possible transaction, query Reservation. First implement a version of program that allows unsynchronized access to the reservation record. Show how the integrity of the database can be corrupted. Next implement a version of program that uses Java thread synchronization with Locks to enforce a disciplined protocol for readers and writers accessing the shared reservation data.