READERS-WRITERS PROBLEM

**Introduction**

The readers/writers problem is one of the classic problems to test synchronization and concurrency mechanism. This problem occurs when a data area is to be shared among several concurrent processes. The data area could be a file, a block of main memory, or even a bank of processor register. Some of the processes may want to only to read the content of the shared area, whereas others may want to update the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as **readers**, and to the rest as **writers**. A common synchronization problem occurs when a resource can be used in two modes, sharable by any number of readers or exclusively by one write at a time. The problems deal with situations in which many processes must access the same [shared memory](http://en.wikipedia.org/wiki/Shared_memory) at one time,  some reading and some writing, with the natural constraint that no process may access the share for reading or writing while another process is in the act of writing to it. Following are the two readers-writers problems:

First Reader Writer Problem:

*“No reader shall be kept waiting unless a writer has already obtained permission to use the shared object*.” This is also called **readers-preference**.

Second Reader Writer Problem:

The second readers-writers problem requires that “*once a writer is ready, that writer performs its write as soon as possible*.” This is also called **writers- preference.**

The solution to first problem may result in starvation. In first case, the writers may starve.

**Critical Section:**

Consider a system consisting of *n* processes {Po, P1, ........., *P11 \_* I}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the *entry section.* The critical section may be followed by an *exit section*. The remaining code is the *remainder section*. The general structure of a typical process *Pi* is shown in

do {

*entry section*

*critical section*

*exit section*

*remainder section*

} while (TRUE);

**Figure 6.1** General structure of a typical process *A.*

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded wait:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

**Semaphore:**

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal (). The wait () operation was originally termed P (from the Dutch *proberen,* "to test"); signal () was originally called V (from *verhogen,* "to increment").

The definition of wait () is as follows:

wait (S) {

while S <= 0

; *II* no-op

s-- ;

}

The definition of signal () is as follows:

signal(S) {

S++;

}

All modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait (S), the testing of the integer value of S (S<= 0), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented in Section 6.5.2; first, let us see how semaphores can be used.

**Understanding the problems:**

**First Reader Writer Problem:**

The first reader writer problem requires that *“No reader shall be kept waiting unless a writer has already obtained permission to use the shared object*.”

Suppose we have a shared memory area. It is possible to protect the shared data behind a mutual exclusion **[mutex](http://en.wikipedia.org/wiki/Mutex" \o "Mutex)**, in which no two processes can access the data at the same time. However, this solution is suboptimal, because it is possible that a reader *R1* might have the lock, and then another reader *R2*requests access. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. It would be foolish to make *R2* wait until *R1* was done before starting its own read operation; instead, *R2* should start right away. This is the motivation for the **first readers-writers problem**.

semaphore wrt=1;

semaphore mutex=1;

readcount=0;

writer() {

<ENTRY Section>

wait(wrt); //Lock the shared file for a writer

<CRITICAL Section> // Writing is done

<EXIT Section>

signal (wrt); //Release the shared file for use by other readers. Writers are allowed if there are no readers requesting it.

}

reader() {

<ENTRY Section>

wait (mutex ); //Ensure that no other reader can execute the <Entry> section while you are in it

readcount++; //Indicate that you are a reader trying to enter the Critical Section

if (readcount == 1) //Checks if you are the first reader trying to enter CS

wait (wrt); //If you are the first reader, lock the resource from writers. Resource stays reserved for subsequent readers

signal (mutex); //Release <Entry> Section. Let other readers enter the <Entry> section, now that you are done with it.

<CRITICAL Section>

// (Critical Section Area) // Do the Reading

<EXIT Section>

wait (mutex); //Ensure that no other reader can execute the <Exit> section while you are in it

readcount--; //Indicate that you are no longer needing the shared resource. One less readers

if (readcount == 0) //Checks if you are the last (only) reader who is reading the shared file

signal(wrt); //If you are last reader, then you can unlock the resource. This makes it available to writers.

signal (mutex) ; //Let other readers enter the <Exit> section, now that you are done with it.

}

In this solution of the readers/writers problem, the first reader must lock the resource (shared file) if such is available. Once the file is locked from writers, it may be used by many subsequent readers without having them to re-lock it again.

Before entering the CS, every new reader must go through the entry section. However, there may only be a single reader in the entry section at a time. This is done to avoid race conditions on the readers. To accomplish this, every reader which enters the <ENTRY Section> will lock the <ENTRY Section> for themselves until they are done with it. Please note that at this point the readers are not locking the resource. They are only locking the entry section so no other reader can enter it while they are in it. Once the reader is done executing the entry section, it will unlock it by signalling the mutex semaphore. Same is valid for the <EXIT Section>. There can be no more than a single reader in the exit section at a time, therefore, every reader must claim and lock the Exit section for themselves before using it.

Once the first reader is in the entry section, it will lock the resource. Doing this will prevent any writers from accessing it. Subsequent readers can just utilize the locked (from writers) resource. The very last reader (indicated by the *readcount* variable) must unlock the resource, thus making it available to writers.

In this solution, every writer must claim the resource individually. This means that a stream of readers can subsequently lock all potential writers out and starve them. This is so, because after the first reader locks the resource, no writer can lock it, before it gets released. And it will only be released by the very last reader. Hence, this solution does not satisfy fairness.

**Second Readers Writer Problem:**

The second readers-writers problem requires that “*once a writer is ready, that writer performs its write as soon as possible*.” This is also called **writers- preference.**

In the previous solution, readers have priority. Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. Therefore, writers are subjected to starvation.

So, we must guarantee that no new readers are allowed to access to the data area once the writer declare the desire to write.

For readers, one additional semaphore is needed in addition to the ones already defined.

int readCount=0, writeCount=0;

semaphore mutex1=1, mutex2=1;

semaphore RD=1,WRT=1

**writer**() {

while(TRUE) {

<other computing>;

wait(mutex2);

writeCount++;

if(writeCount == 1)

wait(RD);

signal(mutex2);

wait(WRT);

**access(resource);**

signal(WRT);

wait(mutex2)

writeCount--;

if(writeCount == 0)

signal(RD);

signal(mutex2);

}

}

**reader**() {

while(TRUE) {

<other computing>;

wait(RD);

wait(mutex1);

readCount++;

if(readCount == 1)

wait(WRT);

signal(mutex1);

signal(RD);

**access(resource);**

wait(mutex1);

readCount--;

if(readCount == 0)

signal(WRT);

signal(mutex1);

}

}

In this solution preference is given to the writers. This is accomplished by forcing every reader to lock and release the read semaphore individually. The very writer must release the read semaphore, thus opening the gate for readers to try reading.

No reader can engage in the entry section if the read seamphore has been set by a writer previously. The reader must wait for the last writer to unlock the resource and read semaphores. On the other hand, if a particular reader has locked the read seamphore, this will indicate to any potential concurrent writer that there is a reader in the entry section. So the writer will wait for the reader to release the read and then the writer will immediately lock it for itself and all subsequent writers. However, the writer will not be able to access the resource until the current reader has released the resource, which only occurs after the reader is finished with the resource in the CS.

If there are no writers wishing to get to the resource, indicated to the reader by the status of the read semaphore, then the readers will not lock the resource. This is done to allow a writer to immediately take control over the resource as soon as the current reader is finished reading. Otherwise, the writer would need to wait for a queue of readers to be done before the last one can unlock the read semaphore. As soon as a writer shows up, it will try to set the read and hang up there waiting for the current reader to release the read. It will then take control over the resource as soon as the current reader is done reading and lock all future readers out.