Paper name – Operating System

Process Synchronization

Monitors -

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing error that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur. Unfortunately, such timing errors can still occur when semaphores are used. Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);

………

critical section

………

wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);

……

critical section

…….

wait(mutex);
```

In this case, a deadlock will occur.

Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct-the monitor type.

A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

The syntax of a monitor is as follows

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }


         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

The monitor construct ensures that only one process at a time is active within the monitor.

However the monitor construct, as defined so far is not sufficiently powerful for modeling some

synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct.
A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:
condition x, y

The only operations that can be invoked on a condition variable are wait () and signal().
The operation x. wait();- means that the process invoking this operation is suspended until another process invokes x. signal()

The x. signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect.

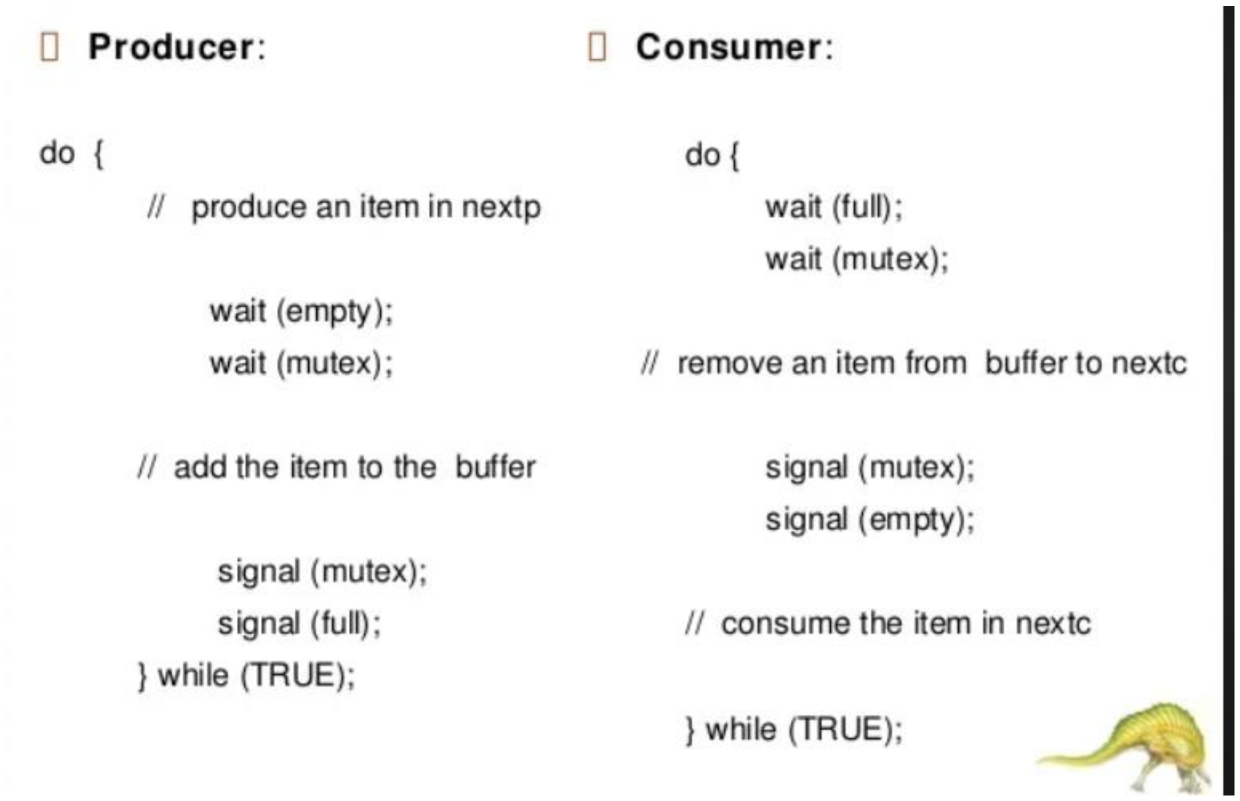
The Producer/Consumer Problem
We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem.
The general statement is this: There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer operations.
That is, only one agent (producer or consumer) may access the buffer at any one time.
The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows

```
producer:
while (true) {
    /* produce item v */;
    b[in] = v;
    in++;
}
```

```
consumer:
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */;
}
```

The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the consumer makes sure that the producer has advanced beyond it (in > out) before proceeding.

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```



- **Producer**:

```
do {
        //   produce an item in nextp

             wait (empty);
             wait (mutex);

       // add the item to the  buffer

             signal (mutex);
             signal (full);
       } while (TRUE);
```

- **Consumer**:

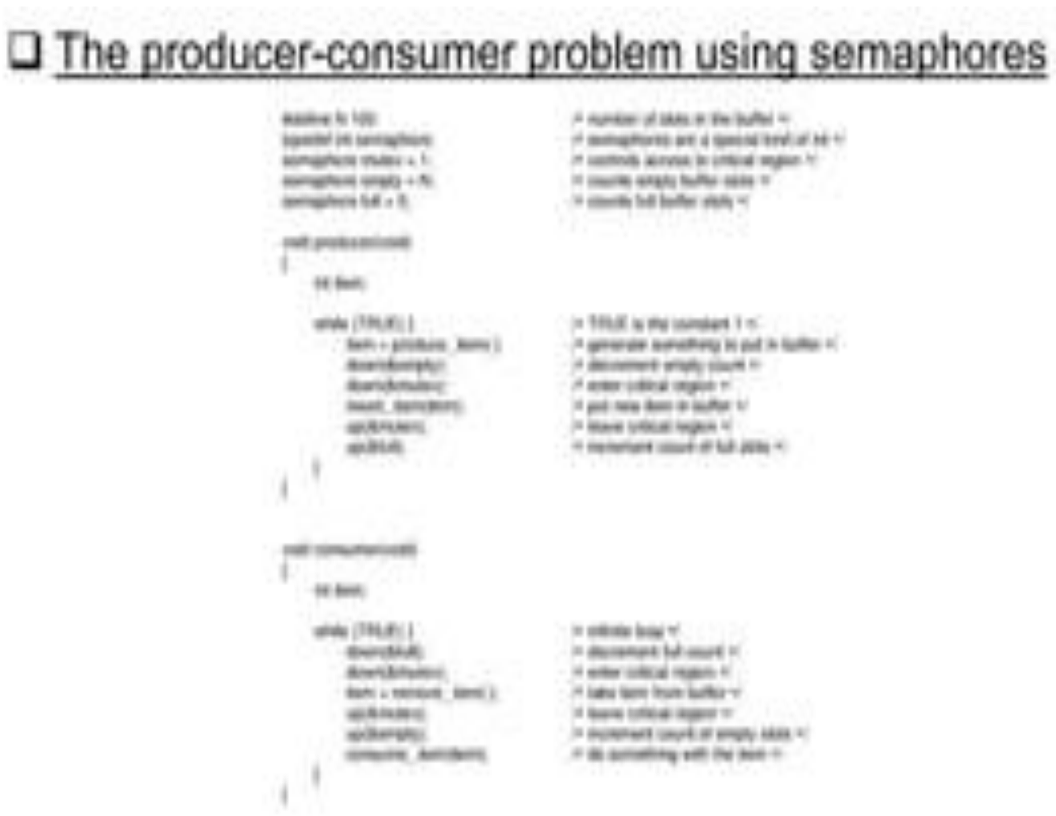
```
do {
       wait (full);
       wait (mutex);

  // remove an item from  buffer to nextc

       signal (mutex);
       signal (empty);

  // consume the item in nextc

} while (TRUE);
```



❑ The producer-consumer problem using semaphores

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                 /* TRUE is the constant 1 */
          item = produce_item();    /* generate something to put in buffer */
          down(&empty);             /* decrement empty count */
          down(&mutex);             /* enter critical region */
          insert_item(item);        /* put new item in buffer */
          up(&mutex);               /* leave critical region */
          up(&full);                /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                 /* infinite loop */
          down(&full);              /* decrement full count */
          down(&mutex);             /* enter critical region */
          item = remove_item();     /* take item from buffer */
          up(&mutex);               /* leave critical region */
          up(&empty);               /* increment count of empty slots */
          consume_item(item);       /* do something with the item */
     }
}
```

Here the producer will wait till empty>0 and then decrement "empty" semaphore. Then it will acquire lock through wait (mutex) and then add item to the buffer. After adding it will release the lock through signal (mutex) and then increment the value of "full" semaphore.

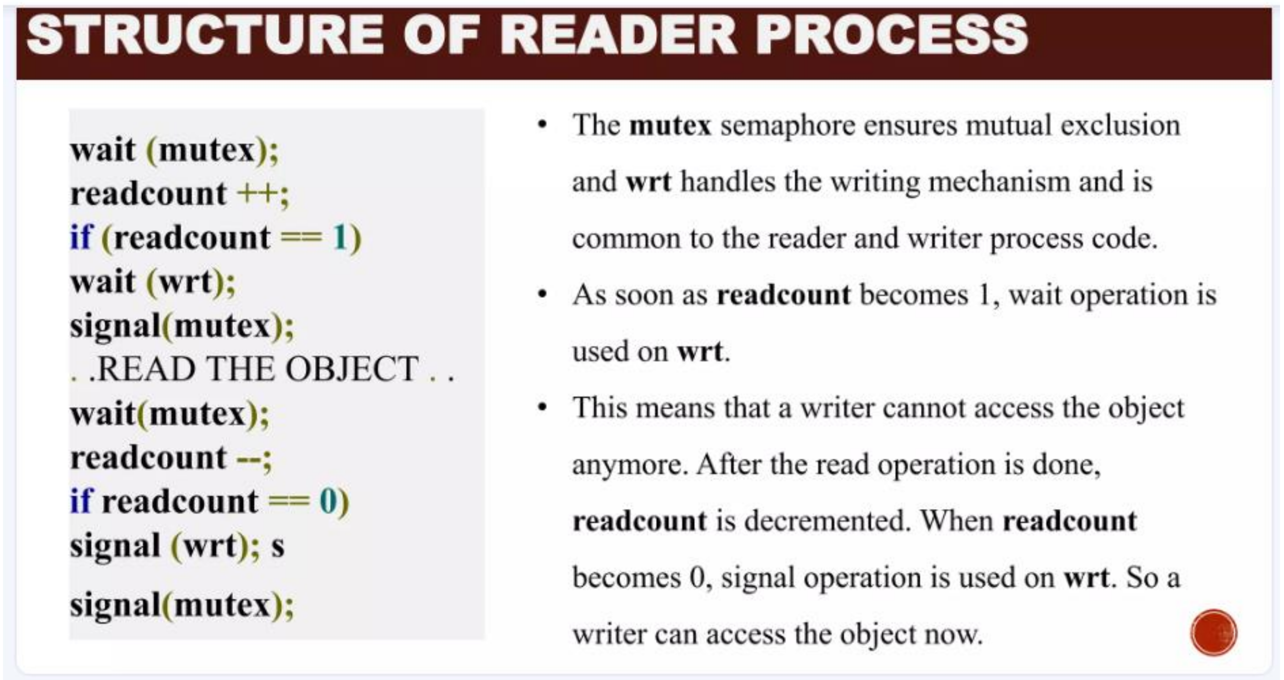
Here the consumer will check if full > 0 and then decrement full. It will acquire lock and remove data from buffer . It will then release lock through signal (mutex) and then increment the value of "empty " semaphore.

The Readers/Writers Problem

In this section, we look at another classic problem: the readers/writers problem.
The readers/writers problem is defined as follows:
There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).
The conditions that must be satisfied are as follows:
1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike.
At the same time, it is important to prevent writers from interfering with each other and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

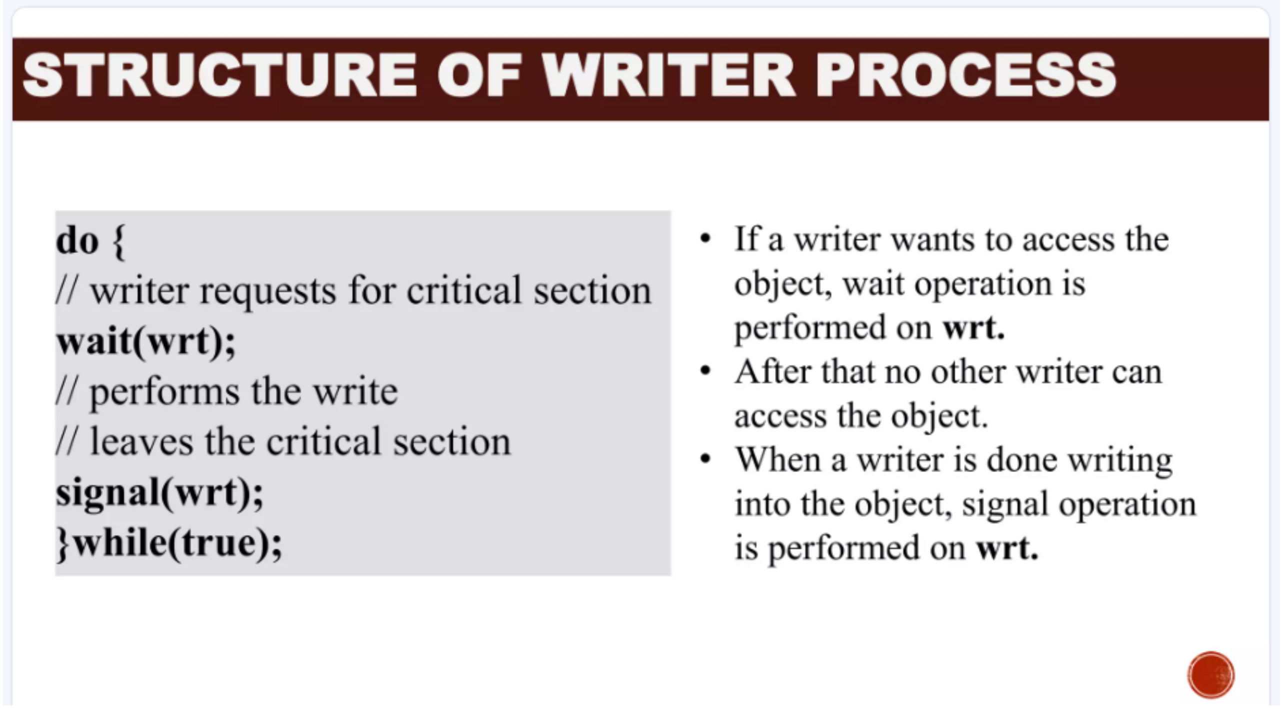
.



STRUCTURE OF READER PROCESS

```
wait (mutex);
readcount ++;
if (readcount == 1)
wait (wrt);
signal(mutex);
. .READ THE OBJECT . .
wait(mutex);
readcount --;
if readcount == 0)
signal (wrt); s
signal(mutex);
```

- The **mutex** semaphore ensures mutual exclusion and **wrt** handles the writing mechanism and is common to the reader and writer process code.
- As soon as **readcount** becomes 1, wait operation is used on **wrt**.
- This means that a writer cannot access the object anymore. After the read operation is done, **readcount** is decremented. When **readcount** becomes 0, signal operation is used on **wrt**. So a writer can access the object now.



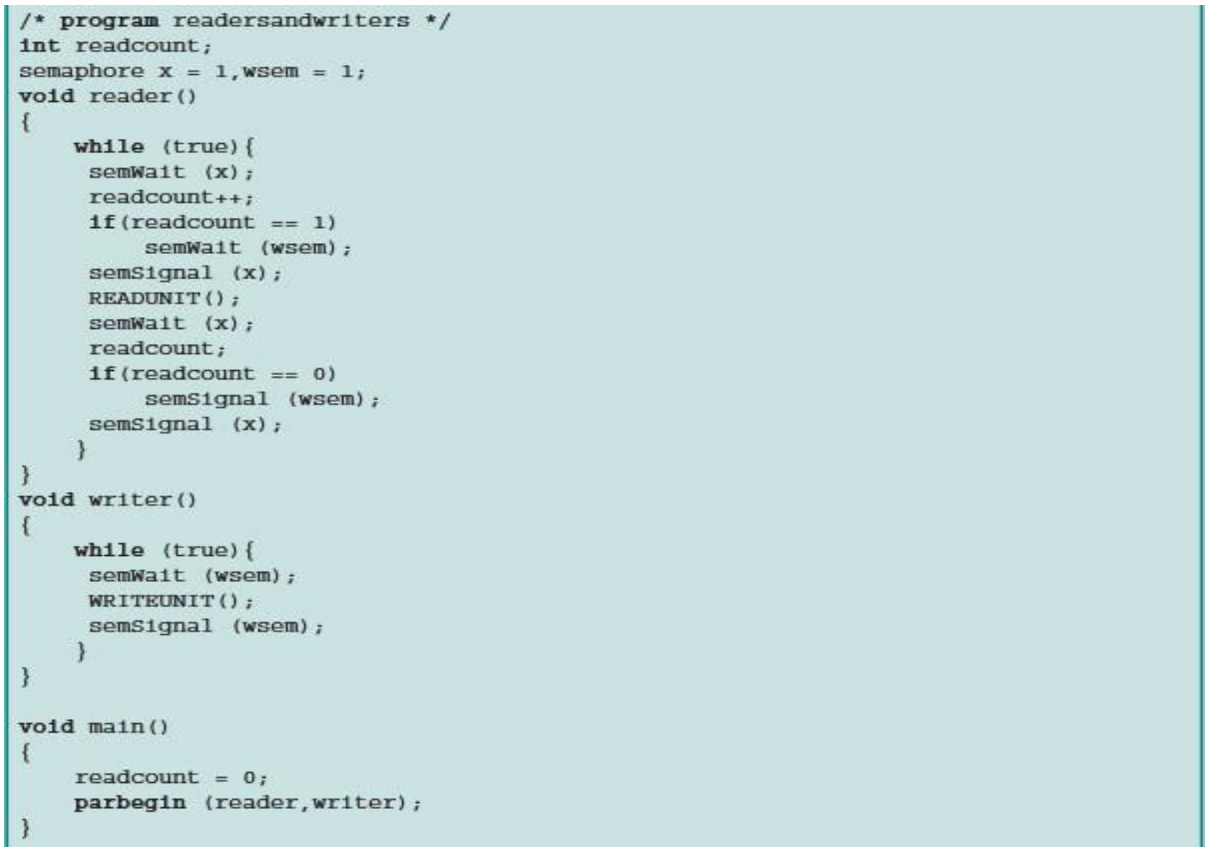
STRUCTURE OF WRITER PROCESS

```
do {
// writer requests for critical section
wait(wrt);
// performs the write
// leaves the critical section
signal(wrt);
}while(true);
```

- If a writer wants to access the object, wait operation is performed on **wrt.**
- After that no other writer can access the object.
- When a writer is done writing into the object, signal operation is performed on **wrt.**

Readers have priority

The semaphore wsem is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that,when there are no readers reading, the first reader that attempts to read should wait on wsem.When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable readcount is used to keep track of the number of readers, and thesemaphore x is used to assure that readcount is updated properly



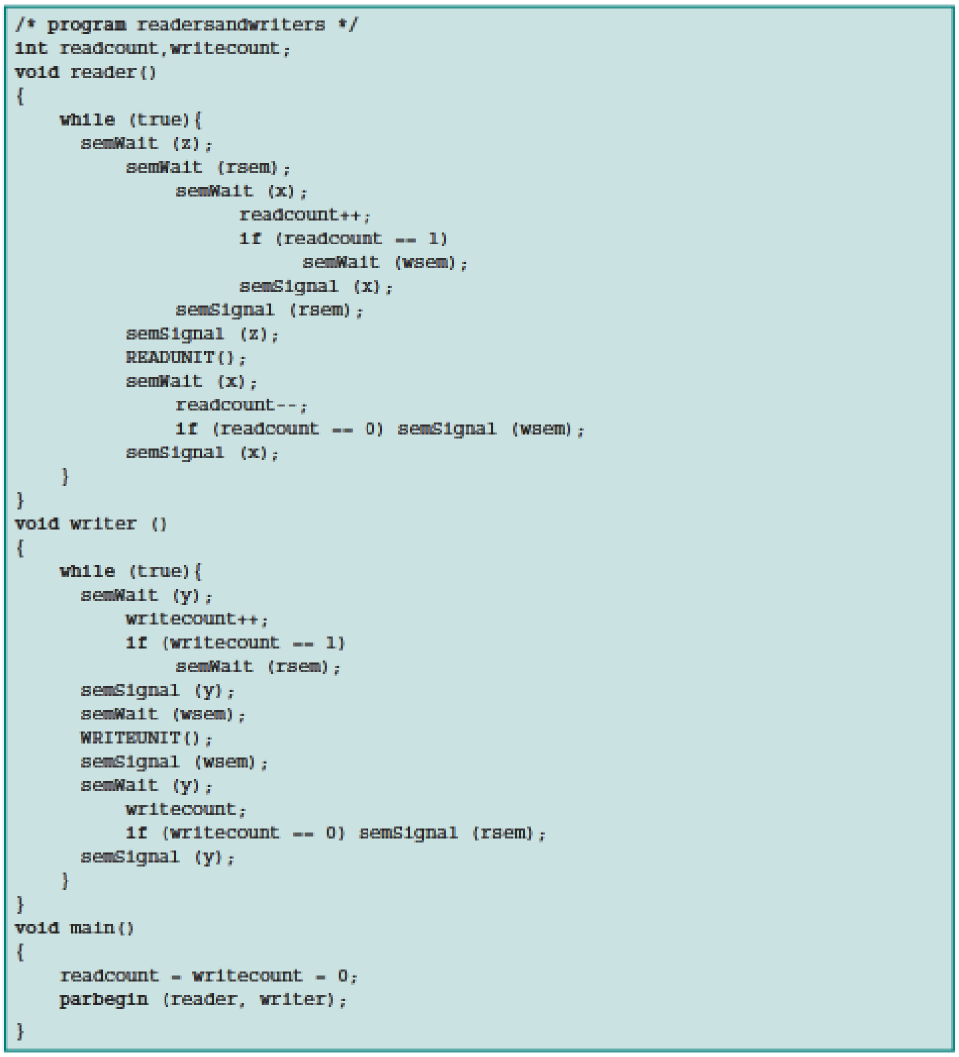
```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

Writers 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 subject to starvation.

- A semaphore rsem that inhibits all readers while there is at least one writer desiring access to the data area
- A variable writecount that controls the setting of rsem
- A semaphore y that controls the updating of writecount

For readers, one additional semaphore is needed. A long queue must not be allowed to build up on rsem; otherwise writers will not be able to jump the queue. Therefore, only one reader is allowed to queue on rsem , with any additional readers queueing on semaphore z , immediately before waiting on rsem.



```
/* program readersandwriters */
int readcount,writecount;
void reader()
{
    while (true){
      semWait (z);
          semWait (rsem);
              semWait (x);
                   readcount++;
                   if (readcount == 1)
                        semWait (wsem);
                   semSignal (x);
              semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
              readcount--;
              if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
void writer ()
{
    while (true){
      semWait (y);
          writecount++;
          if (writecount == 1)
              semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
          writecount;
          if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```