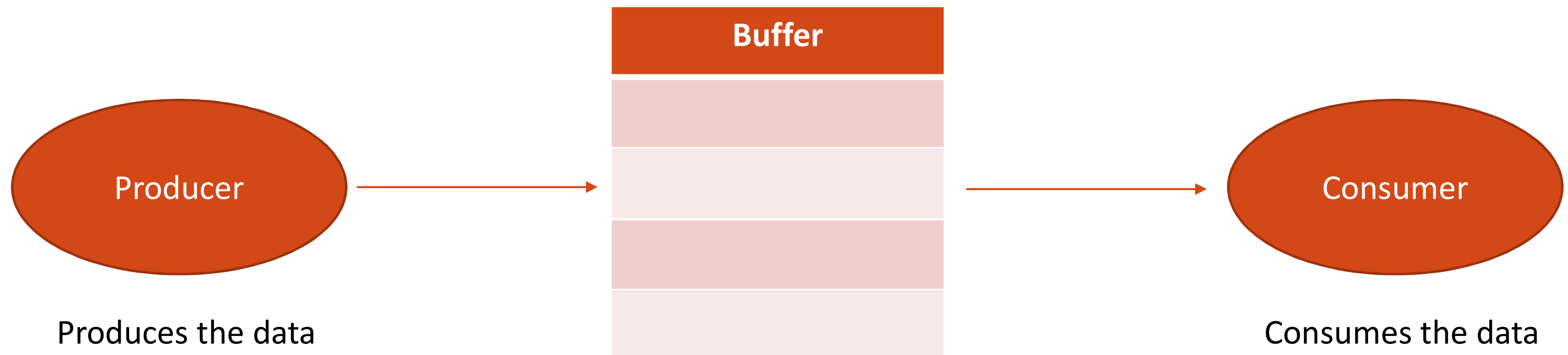
Process Synchronization

Producer Consumer Problem



The producers and consumers must operate concurrently and without interfering with each other.

- **The producers must wait if the buffer is full, and the consumers must wait if the buffer is empty.**
- To solve the producer-consumer problem, various synchronization techniques can be used.

Producer Consumer Problem

- If the Producer does not know the size of the buffer, it will keep on producing the data that would result in a **buffer overflow**
- Similarly, if the Consumer does not know the size of the buffer, it will try to get the data from the buffer and would result in no **data found exception**.
- This is also called a **Bounded Buffer Problem**.
- To solve this problem, both producer and consumer are told the size of the buffer.
- If the Producer knows that the buffer is full, it will not produce data and wait
- If the Consumer knows that the buffer is empty, it will not get the data and wait

Process Synchronization

- Processes can execute concurrently
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
Suppose that we wanted to provide a solution to the consumer-producer problem. We can do so by having an integer counter that keeps track of the number of items in the buffer. Initially, the counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Process Synchronization

Producer

```
int counter =0;
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter            {register1 = 5}
S1: producer execute register1 = register1 + 1      {register1 = 6}

S2: consumer execute register2 = counter            {register2 = 5}
S3: consumer execute register2 = register2 - 1      {register2 = 4}
S4: producer execute counter = register1            {counter = 6 }
S5: consumer execute counter = register2            {counter = 4 }
```

Race Condition

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter            {register1 = 5}
S1: producer execute register1 = register1 + 1      {register1 = 6}
S2: consumer execute register2 = counter            {register2 = 5}
S3: consumer execute register2 = register2 – 1      {register2 = 4}
S4: producer execute counter = register1            {counter = 6 }
S5: consumer execute counter = register2            {counter = 4 }
```

Counter should be 5, since we added 1 to it and subtracted 1 from it

Race Condition

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 – 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4 }
```

Counter should be 5, since we added 1 to it and subtracted 1 from it

This happened because processes are not synchronized properly and hence Race Condition

it's important to use appropriate synchronization mechanisms, to ensure that concurrent access to shared variables is properly coordinated

Critical Section Problem

```
Process 1
{
        Non - Critical Section
        {
                int x, y;
        //working with x and way
        }

        Critical Section
        {
                count = 0;
        //working with count variable
        }
}
```

```
Process 2
{
        Non - Critical Section
        {
                int i, j;
        //working with x and way
        }

        Critical Section
        {
                count = 0;
        //working with count variable
        }
}
```

Critical Section Problem

- Consider system of n processes `{p0, p1, … pn-1}`
- Each process has a **critical section segment** of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process is in the critical section, no other may be in its critical section
- **Critical section problem** is to design a protocol to solve this
- Each process must ask permission to enter a critical section in its **entry section**, perform its critical section tasks, release the critical section in its **exit section**, and then proceed to the remaining section.

Critical Section Problem

General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical Section Problem

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. Processes must not stop each other indefinitely.
- **Bounded Waiting** - A bound must exist 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

Critical Section handling in OS

- Two approaches depending on if the kernel is preemptive or non-preemptive
  - **Preemptive Scheduling**– allows preemption of the process when running in kernel mode. It means the OS can take a CPU from one process and give it to another.
  - **Non-preemptive Scheduling**– process runs until exits kernel mode, blocks, or voluntarily yields CPU. The OS can not take from it the CPU until the process finishes.
    - Essentially free of race conditions in kernel mode

Peterson's Solution

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

Process i code

```
do {
flag[i] := TRUE;
turn := j;
while (flag[j] and turn=j) do no-op;
CRITICAL SECTION
flag[i] := FALSE;
REMAINDER SECTION }
while(1);
```

Process j code

```
do {
flag[j] := TRUE;
turn := i;
while (flag[i] and turn=i) do no-op;
CRITICAL SECTION
flag[j] := FALSE;
REMAINDER SECTION }
while(1);
```

Hardware Synchronization

- Many systems provide hardware support for implementing the critical section code.
- Protecting critical regions via locks
- Currently running code would execute without preemption
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Synchronization

Test and Set Instructions

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv:
  }
```

Hardware Synchronization

There is a global variable called **lock = False handled by the OS**

Test and Set Instructions

```
do {
      while (test_and_set(&lock))
      {
        /* do nothing */
      }
      /* critical section */
      lock = false;
      /* remainder section */
} while (true);
```

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target; // these two are
    *target = TRUE; // atomic
    return rv;
}
// 1st time lock is False→ target is True→ rv is False
// 2nd time target is True→ rv is true
```

Hardware Synchronization

Compare and Swap Instructions

```
int compare_and_swap(int *M, int expected, int new_value)
{
        int temp = *M;
        if (*M == expected)
           *M = new_value;
      return temp;
}
```

Hardware Synchronization

Compare and Swap Instructions

```
int compare_and_swap(int *M, int expected, int new_value)
{
      int temp = *M;
      if (*M == expected)
          *M = new_value;
      return temp;
}

do {// So, 1st time 0!=0 No→ enter critical section
    while (compare_and_swap(&lock, 0, 1) != 0)
      {
             /* do nothing */
      }
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

```
1st Time
//so temp = 0 or F
// 0==0 yes
// *M = 1
// return temp ==0

2nd Time
//so temp = 1 or T
// return temp ==1
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting: This lock therefore called a spinlock
- Spinlock keeps checking the lock (busy waiting), while mutex puts threads waiting for the lock into sleep (blocked).
- A busy-waiting thread wastes CPU cycles, while a blocked thread does not.

Mutex Locks

```
acquire() {
       while (!available)
          ; /* busy wait */
       available = false;; //once in the critical section
}
release() {
       available = true;
}
```

```
do {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
- wait() and signal() also called P() and V() respectively [the letters come from the Dutch words Probeer (try) and Verhoog (increment)].
- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a mutex lock

Semaphores

In 1965, Dijkstra proposed a new technique for managing concurrent processes.

He used an integer variable to synchronize the progress of interacting processes.

This variable is called a semaphore. It can have the value of 1 [1 printer] or more [many printers].

So it is a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively [From the Dutch words Probeer (try) and Verhoog (increment)].

In simple words, the semaphore is a variable that can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

```
P(S): if S >= 1 then S := S - 1 else <block and enqueue the process>;
V(S): if <some process is blocked on the queue> then <unblock a process> else S
:= S + 1;
```

Wait and Signal Operations

Wait: This operation decrements the value of its argument S, as soon as it would become >=1.

This Operation helps you to control the entry of a task into the critical section.

In the case of the negative or zero value, no operation is executed.

```
wait(S)
{
   while (S<=0);//Atomic
   S--;          //Operations
}
```

Note: When one process modifies the value of a semaphore, no other process can simultaneously modify that same semaphore's value.

Signal: Increments the value of its argument S, as there is no more process blocked on the queue.

This Operation is used to control the exit of a task from the critical section.

```
signal(S)
{
S++;
}
```

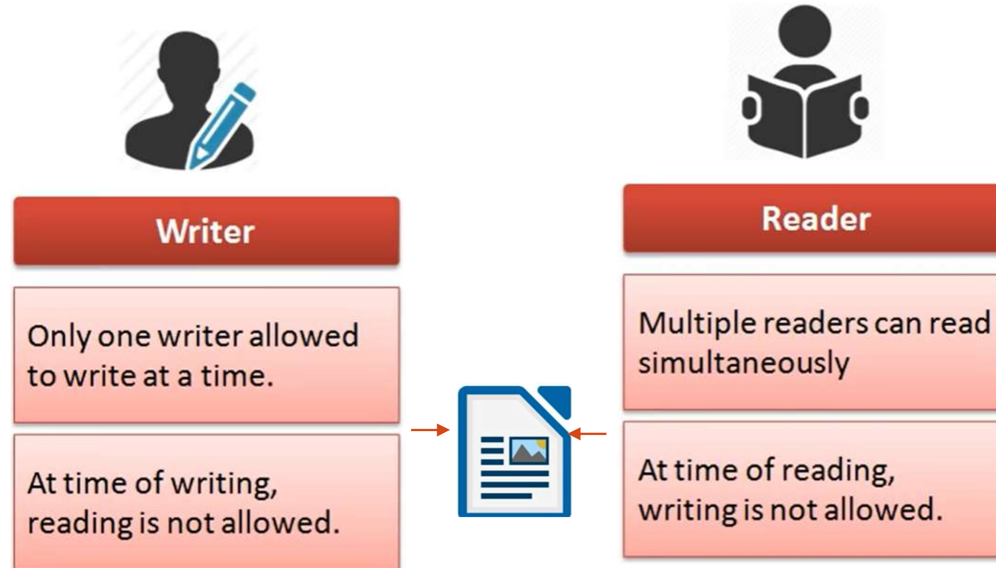
All modifications to the integer value of semaphore in the wait() and signal() operations must be executed indivisibly.

wait and signal controls the number of processes entering and existing a critical section.

Reader Writer Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered
- Shared Data
  - Data set

Reader Writer Problem



Writer

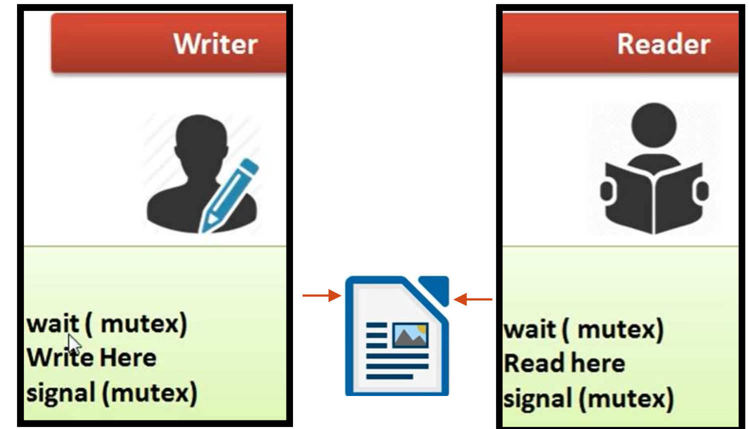
Only one writer allowed to write at a time.

At time of writing, reading is not allowed.

Reader

Multiple readers can read simultaneously

At time of reading, writing is not allowed.



First Solution: mutex = 1.

This solution allows only one reader or one writer to access shared resources.

Writer

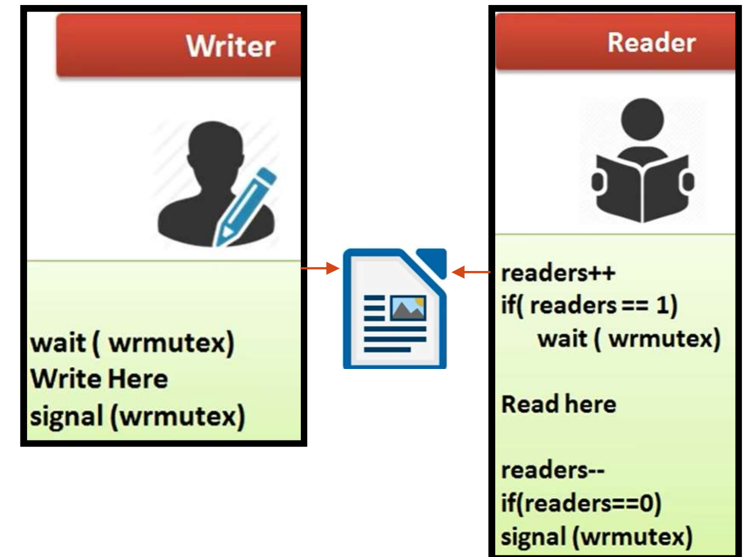
Only one writer allowed to write at a time.

At time of writing, reading is not allowed.

Reader

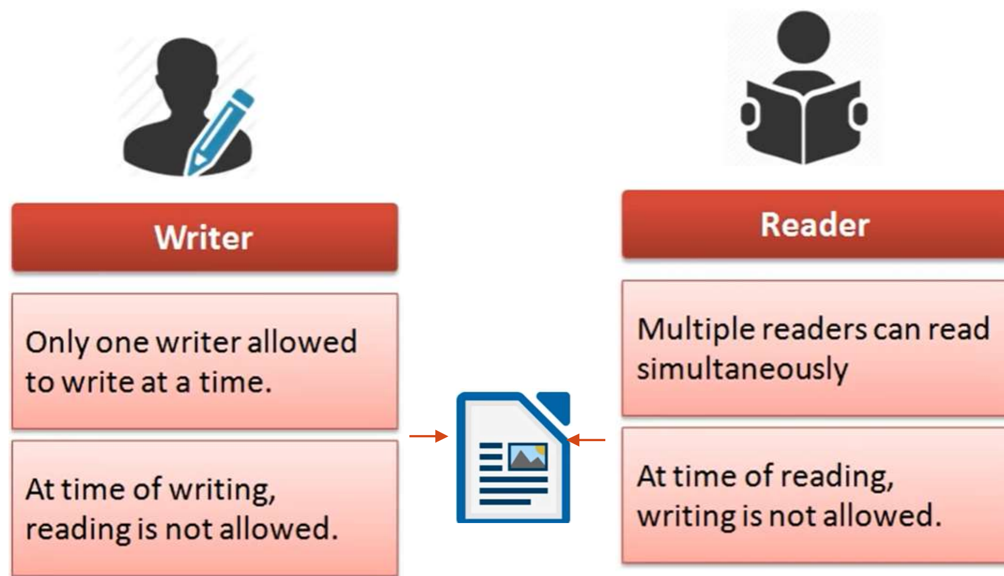
Multiple readers can read simultaneously

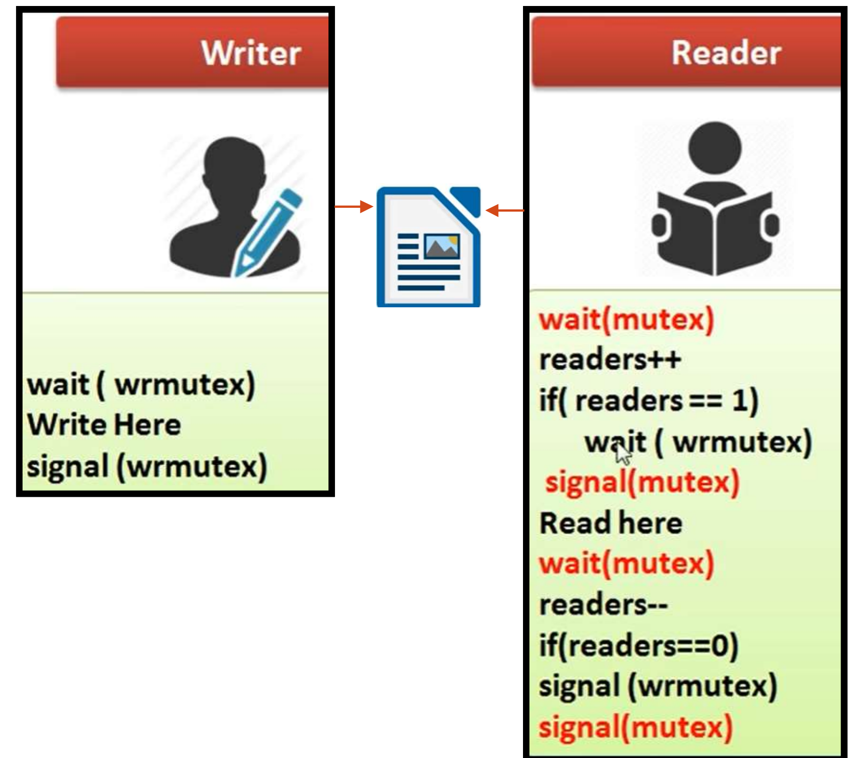
At time of reading, writing is not allowed.



Second Solution:

This solution allows many readers to access shared resources (race condition).





- Semaphore rw_mutex and mutex initialized to 1
- Integer counter readers initialized to 0

Third Solution: mutex != 1 and no race condition. This solution allows many readers to access shared resources.

Reader Writer Problem

Writer

```
do {
        wait(rw_mutex);

             ...
        /* writing is performed
             ...

        signal(rw_mutex);

    } while (true);
```

Reader

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)

           wait(rw_mutex);

        signal(mutex);

             ...
        /* reading is performed */

             ...

        wait(mutex);
        read count--;
        if (read_count == 0)

             signal(rw_mutex);

        signal(mutex);

    } while (true);
```