**Multiple-Reader-writer**

The multiple user reader writer problem wherein multi- ple “writers” can write to a shared queue and multiple “readers” can read data off the shared queue with no race conditions. While a reader “writes” a certain element to the queue, no “reader” can read that element. The readers may read other elements but not the one current written to. Similarly, only one “writer” can update the queue at a time. Multiple “readers” may read elements off the queue but dequeuing from the queue by the “readers” must be atomic.

**Description Of Code**

1. Main function :

* Creates a new Queue.
* Initialises 2 semaphores, namely mutex and wrt. "mutex" is used for keeping track of the no, of readers, whereas "wrt" can be regarded as lock on the queue.
* Pthreads are then created according to the user input.

1. Reader function : Serves the reading purpose of multiple readers via menu.

* The reader first tries to get "mutex" via "sem\_wait", in order to tell that first reader has arrived.
* If it gets it, the readercount is incremented by that process only.
* If the process is the first process to read from the shared queue, then "wrt" semaphore is also acquired so as to prevent the writers from updating the contents of the queue at that time.
* Reader menu provides (based upon the choice of the user): i) Printing entire queue elements, except the current written one. (Call to the PrintPrevQ() function). ii) Printing an element based upon the index entered by the user, except the current written one. (1-based indexing is followed, Call to the PrintQElement() functiom). iii) Reader can quit anytime.
* "sem\_post" on "mutex" allows other reader process to read from the queue.
* Again the "mutex" is acquired so as to now decrement the readercount.
* If no readers are left, "wrt" lock on the shared queue is lifted to allow writers to write.
* Lock on "mutex" is also lifted as no readers are left now.
* Above procedure also maitains atomicity among the readers.

1. Writer function : Serves the writing purpose of multiple writers via menu.

* First the lock on "wrt" is acquired, signalling that a writer process has arrived and only 1 writer at a time can do the updation.
* Writer menu provides (based upon the choice of the user): i) Proper message is received from the user and enQueue() function is called. ii) Writer can quit anytime.
* Lock on "wrt" is lifted, signalling that now the lock on shared queue can be acquired by other writer processes.

1. PrintPrevQ function : Prints the contents of the shared queue, except the last written one. (Performs reading)
2. enQueue function : Enqueues a new message by a writer into the shared queue.
3. PrintQ function : Prints all the contents of the shared queue.
4. PrintQElement function : Prints an element from the queue based upon its index.

**Compiling And Testing**

* Ensure that the Makefile and the source code are in the same directory.
* In the terminal, type: 1. make 2. ./rw
* To terminate the program, quit from all the reader-writer threads. (Option is provided in the menu).

**Inputs To Give**

The code is user-friendly (No need to give arguments at command line).

* Enter the no. of reader and writer threads (Must be between 1 and 100 [inclusive]).
* Provide proper messages when asked.
* Provide proper choice for menus as is asked.
* Provide proper index while printing a specific element. 1-based indexing is followed. Since the reader cannot access the current written element, do not provide the index equal to queue size (will give Element Access Error).

**Expected Output**

* Only 1 writer should write to the shared queue at a time.
* Multiple readers should read from the shared queue.
* Messages read by the readers should not be the current written message, irrespective of the writer.
* Specific messages that readers want to read from the queue.
* Menu-driven functionality is provided.
* Pthreads may generate random outputs (Outputs need not be same after each run), but above rules should follow.

**Error Value And Interpretation**

* Scanf Error:- If argument provided by the user is 1. Not a number 2. A number < 1 or > 100 (when providing No. of reader/writers)
* Message Error:- If the message provided by the user is newline.
* Index Error:- If the index entered while seeking a message at a specific index in the shared queue is < 1 or > the queue size.
* Element Access Error:- If the index entered while seeking a message at a specific index in the shared queue is equal to the queue size since seeking current written elements is not allowed.
* Input Error:- If the choice entered is not what is expected.

**Code**

## #include <stdio.h>

## #include <pthread.h>

## #include <semaphore.h>

## #include <stdlib.h>

## #include <string.h>

## #include <sys/types.h>

## #include <sys/ipc.h>

## #include <sys/shm.h>

## sem\_t mutex, wrt;

## int readcount;

## void \*Reader(void \*arguments);

## void \*Writer(void \*arguments);

## char prevdata[500];

## struct SharedQNode

## {

## char msg[500];

## struct SharedQ \*link;

## };

## struct SharedQ

## {

## struct SharedQNode \*front;

## struct SharedQNode \*rear;

## int size;

## };

## struct SharedQ \*createNewQ()

## {

## struct SharedQ \*q = (struct SharedQ\*)malloc(sizeof(struct SharedQ));

## q->front = q->rear = NULL;

## q->size = 0;

## return q;

## }

## void enQueue(struct SharedQ \*q, char data[500])

## {

## struct SharedQNode \*temp = (struct SharedQNode\*)malloc(sizeof(struct SharedQNode));

## strcpy(temp->msg, data);

## temp->link = NULL;

## if (q->rear == NULL)

## {

## q->front = q->rear = temp;

## q->size = q->size + 1;

## PrintQ(q);

## return;

## }

## q->rear->link = temp;

## q->rear = temp;

## q->size = q->size + 1;

## PrintQ(q);

## }

## void PrintPrevQ(struct SharedQ \*q)

## {

## struct SharedQNode \*temp = q->front;

## printf("\nQueue contains :\n");

## while (temp != NULL && strcmp(temp->msg, prevdata)!=0)

## {

## printf("%s", temp->msg);

## temp = temp->link;

## }

## }

## void PrintQ(struct SharedQ \*q)

## {

## struct SharedQNode \*temp = q->front;

## printf("\nWriting to queue successfull. Queue contains :\n");

## while (temp != NULL)

## {

## printf("%s", temp->msg);

## temp = temp->link;

## }

## }

## void PrintQElement(struct SharedQ \*q, int index)

## {

## struct SharedQNode \*temp = q->front;

## printf("\nElement %d is : ", index);

## int ctr = 1;

## while (temp != NULL && ctr < index)

## {

## temp = temp->link;

## ctr = ctr + 1;

## }

## printf("%s\n", temp->msg);

## }

## struct SharedQ \*q;

## int main()

## {

## q = createNewQ();

## int i = 0, ReaderCount = 0, WriterCount = 0;

## sem\_init(&mutex, 0, 1);

## sem\_init(&wrt, 0, 1);

## pthread\_t Readers[100], Writers[100];

## printf("\nEnter number of Readers thread (Max 100) : ");

## if (scanf("%d", &ReaderCount) <= 0 || ReaderCount < 1 || ReaderCount > 100)

## {

## printf("Scanf error: Only positive numbers expected\n");

## return 0;

## }

## printf("\nEnter number of Writers thread (Max 100) : ");

## if (scanf("%d", &WriterCount) <= 0 || WriterCount < 1 || WriterCount > 100)

## {

## printf("Scanf error: Only positive numbers expected\n");

## return 0;

## }

## getchar();

## for(i = 0; i < ReaderCount; i++)

## pthread\_create(&Readers[i], NULL, Reader, (void \*)i);

## for(i = 0; i < WriterCount; i++)

## pthread\_create(&Writers[i], NULL, Writer, (void \*)i);

## for(i = 0; i < WriterCount; i++)

## pthread\_join(Writers[i], NULL);

## for(i = 0; i < ReaderCount; i++)

## pthread\_join(Readers[i], NULL);

## sem\_destroy(&wrt);

## sem\_destroy(&mutex);

## printf("\n");

## return 0;

## }

## void \*Reader(void \*arguments)

## {

## do

## {

## sleep(1);

## int readerid = (int)arguments;

## sem\_wait(&mutex);

## readcount++;

## if(readcount == 1)

## {

## sem\_wait(&wrt);

## int choice;

## printf("\nReader %d Menu: ", readerid);

## printf("\n1. Print Queue contents.");

## printf("\n2. Print a Queue entry.");

## printf("\n3. Quit");

## printf("\nEnter your choice (1-3) : ");

## if (scanf("%d", &choice) <= 0)

## {

## printf("Scanf error: Only numbers expected\n");

## exit(0);

## }

## if (choice == 1)

## PrintPrevQ(q);

## else if (choice == 2)

## {

## int index;

## printf("\nQueue size is = %d", q->size);

## if (q->size == 0)

## {

## printf("\nQueue does not contain any element yet.\n");

## }

## else

## {

## printf("\nEnter Index (1-%d) : ", q->size);

## scanf("%d", &index);

## if (index < 1 || index > q->size)

## {

## printf("\nIndex Error: Index out of bounds.\n");

## exit(0);

## }

## else if (index == q->size)

## {

## printf("\nElement Access Error: Cannot access this element.\n");

## exit(0);

## }

## PrintQElement(q, index);

## }

## }

## else if (choice == 3)

## {

## printf("\nReader %d quitted\n", readerid);

## pthread\_cancel(pthread\_self());

## }

## else

## {

## printf("\nInput Error: Invalid choice input.\n");

## exit(0);

## }

## }

## sem\_post(&mutex);

## sem\_wait(&mutex);

## readcount--;

## if(readcount == 0)

## sem\_post(&wrt);

## sem\_post(&mutex);

## } while(1);

## }

## void \*Writer(void \*arguments)

## {

## do

## {

## sleep(1);

## int writerid = (int)arguments;

## sem\_wait(&wrt);

## char msg[100];

## int choice;

## printf("\n\nWriter %d Menu : ", writerid);

## printf("\n1. Write to Queue.");

## printf("\n2. Quit");

## printf("\nEnter your choice (1 or 2) : ");

## if (scanf("%d", &choice) <= 0)

## {

## printf("Scanf error: Only numbers expected\n");

## exit(0);

## }

## getchar();

## if (choice == 1)

## {

## printf("\nEnter message : ");

## fgets(msg, sizeof(msg), stdin);

## if (strcmp(msg, "\n")==0)

## {

## printf("\nMessgae error: Proper text message expected\n");

## exit(0);

## }

## enQueue(q, msg);

## strcpy(prevdata, msg);

## }

## else if (choice == 2)

## {

## printf("\nWriter %d quitted\n", writerid);

## pthread\_cancel(pthread\_self());

## }

## else

## {

## printf("\nInput Error: Invalid choice input.\n");

## exit(0);

## }

## sem\_post(&wrt);

## } while(1);

## }Classical Problems of Synchronization :-

1. Using Semaphore :-

Semaphore can be used in other synchronization problems besides Mutual Exclusion.  
Following are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

In this tutorial we will just focus on **Readers Writers Problem**:

### Problem :-

* There is a shared resource which should be accessed by multiple processes.
* There are two types of processes in this context.
* They are **reader** and **writer**.
* Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource.
* When a writer is writing data to the resource, no other process can access the resource.
* A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

### **Solution 1 Using Semaphores :-**

Here, readers have **higher priority** than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.  
Here, we use :-

* one mutex **m** and a semaphore **w**.
* An integer variable **read\_count** :- used to maintain the number of readers currently accessing the resource. The variable read\_count is initialized to 0.
* A value of **1** is given initially to **m** and **w**.  
  Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the **read\_count** variable.  
  a. **Writer Process** :

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

while(TRUE)

{

wait(w);

/\* perform the write operation \*/

signal(w);

}

C++

Copy

b. **Reader Process** :

1. Reader requests the entry to critical section.
2. If allowed:  
   i. it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **w** semaphore to restrict the entry of writers if any reader is inside.  
   ii.It then, signals mutex as any other reader is allowed to enter while others are already reading.  
   iii. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore **w** as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

while(TRUE)

{

//acquire lock

wait(m);

read\_count++;

if(read\_count == 1)

wait(w);

//release lock

signal(m);

/\* perform the reading operation \*/

// acquire lock

wait(m);

read\_count--;

if(read\_count == 0)

signal(w);

// release lock

signal(m);

}

Thus, the semaphore **w** is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

## Limitations of Semaphores

* Priority Inversion is a big limitation of semaphores.
* Their use is not enforced, but is by convention only.
* With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in details in coming lessons.