***Project:***

***Reader Writer Problem System Call***

***Instructor:***

***Madam Nausheen***

***Members:***

***Salman Ahmed Khan (19k-1043)***

***Sadeem Sattar (19k-1102)***

***Abdullah Tilal Khan (19k-1103)***

***Date: 21-May-2021***

**Project Objective:**

The reader-writer problem is a famous problem in which there is a content, a file or any shared data between processes. The readers will read the data from the content and the writer will write the content.

The problems occurs when readers and writers both occur at a same time such that the readers won't be able to read all the data/message and the writers will update/manipulate the data/message simultaneously.

Therefore, the objective of the problem is to avoid the race condition and simultaneously occurrences of the readers and writers.

**Project Description:**

In the reader writer, the problem occurs when the reader starts to read the content of and at the same time the writer starts to write some content, so it will create problems in the

content and thus problems will occur.

This problem is used to manage the synchronization of the readers and writer’s process

so that the same phenomenon does not happen at the same time. When the writer will

get the access of writing then no reader and no other writer can write. Similarly, when a

reader starts to read the content so no writer would be able to write the

content, however the other reader could read the content.

Therefore, the cases become:

W - W: False

W - R: False

R - W: False

R - R: True

The data structure used is a AVL (binary tree) in which messages is stored and accessed by reader and writer for their purposes.

The problem is solved using semaphores. The process thread is created for readers and writers as per the user input.

Two semaphores used are naming as mutex and write\_msg.

The mutex semaphore is used to create a lock in a reader function so that the count of readers could be known.

The write\_msg semaphore is used to have a lock in writers’ function so that only one writer is allowed in critical section at a time.

Another semaphore naming msg\_lock is used to maintain the reader's input choice of reading the content or exit.

A system call is also created naming sys\_PrintMessage which take a string parameter and print the data/message/content on the kernel.

**Code:**

#include <linux/kernel.h>  
#include <sys/syscall.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <malloc.h>  
#include <stdlib.h>  
#include <string.h>  
#include <sys/unistd.h>  
#include <sys/types.h>  
#include <pthread.h>  
#include <stdbool.h>  
#include <semaphore.h>  
struct BNode  
{  
    int key, heightFactor;  
    struct BNode \*left, \*right;  
    char message[100];  
};  
  
struct BNode \*newNode(int key, char msg[])  
{  
    struct BNode \*temp = (struct BNode \*)malloc(sizeof(struct BNode));  
    temp->key = key;  
    strcpy(temp->message, msg);  
    temp->heightFactor = 1;  
    temp->left = NULL;  
    temp->right = NULL;  
    return temp;  
}  
  
int get\_height\_factor(struct BNode \*node)  
{  
    if (node == 0)  
        return 0;  
    return node->heightFactor;  
}  
//Get the height balance factor  
//Leftsubtree subtract rightsubtree  
int get\_Balancefactor(struct BNode \*node)  
{  
    if (node == 0)  
        return 0;  
    return get\_height\_factor(node->left) - get\_height\_factor(node->right);  
}  
  
int max(int a, int b)  
{  
    return (a > b) ? a : b;  
}  
  
//Right Rotate the subtree  
struct BNode \*rightRotation(struct BNode \*y)  
{  
    struct BNode \*x = y->left;  
    struct BNode \*T2 = x->right;  
  
    //Perform Rotation  
    x->right = y;  
    y->left = T2;  
  
    //Adjust the new heights factor  
    y->heightFactor = 1 + max(get\_height\_factor(y->left), get\_height\_factor(y->right));  
    x->heightFactor = 1 + max(get\_height\_factor(x->left), get\_height\_factor(x->right));  
  
    return x;  
}  
  
//Left Rotate the subtree  
struct BNode \*leftRotation(struct BNode \*x)  
{  
    struct BNode \*y = x->right;  
    struct BNode \*T2 = y->left;  
  
    //Perform Rotation  
    x->left = y;  
    y->right = T2;  
  
    //Adjust the new heights factor  
    x->heightFactor = 1 + max(get\_height\_factor(x->left), get\_height\_factor(x->right));  
    y->heightFactor = 1 + max(get\_height\_factor(y->left), get\_height\_factor(y->right));  
  
    return y;  
}  
  
struct BNode \*Insert(struct BNode \*node, int key, char msg[])  
{  
    struct BNode \*temp = newNode(key, msg);  
  
    if (node == NULL)  
    {  
        node = temp;  
        temp = 0;  
    }  
    else  
    {  
        struct BNode \*previous = NULL;  
        struct BNode \*current = node;  
        while (current != NULL)  
        {  
            if (current->key == temp->key)  
            {  
                printf("Data Already Exist\n");  
                return node;  
            }  
            previous = current;  
            if (current->key > temp->key)  
                current = current->left;  
            else if (current->key < temp->key)  
                current = current->right;  
        }  
        if (previous->key > temp->key)  
            previous->left = temp;  
        else  
            previous->right = temp;  
        temp = 0;  
        current = 0;  
        previous = 0;  
        free(temp);  
        free(current);  
        free(previous);  
    }  
    int balance = get\_Balancefactor(node);  
  
    //left left case  
    if (balance > 1 && key < node->left->key)  
        return rightRotation(node);  
  
    //right right case  
    if (balance < -1 && key > node->right->key)  
        return leftRotation(node);  
  
    //left right case  
    if (balance > 1 && key > node->left->key)  
    {  
        node->left = leftRotation(node->left);  
        return rightRotation(node);  
    }  
  
    //right left case  
    if (balance < -1 && key < node->right->key)  
    {  
        node->right = rightRotation(node->right);  
        return leftRotation(node);  
    }  
  
    /\* return the (unchanged) node pointer \*/  
    return node;  
}  
  
void InOrder\_Traversal(struct BNode \*root)  
{  
    struct BNode \*current = root;  
    Inorder\_Print(current);  
    printf("\n");  
}  
  
void Inorder\_Print(struct BNode \*node)  
{  
    void \*p1;  
    if (node)  
    {  
        Inorder\_Print(node->left);  
        p1=node->message;  
        syscall(333,p1);  
        Inorder\_Print(node->right);  
    }  
       
}  
  
void DeleteAll(struct BNode \*node)  
{  
    if (node)  
    {  
        DeleteAll(node->left);  
        DeleteAll(node->right);  
        node->left = 0;  
        node->right = 0;  
        free(node);  
    }  
}  
  
bool IsKey(int key, struct BNode \*root)  
{  
    if (!root)  
        return false;  
    struct BNode \*current = root;  
    while (current != 0)  
    {  
        if (current->key == key)  
            return true;  
        if (current->key > key)  
            current = current->left;  
        else if (current->key < key)  
            current = current->right;  
    }  
    current = 0;  
    free(current);  
    return false;  
}  
  
struct BNode \*root = NULL;  
sem\_t mutex, write\_msg, msg\_lock;  
int readers\_count = 0;  
  
//Readers - Writer Problem Implementation  
  
//Writer Execution  
void \*Writer(void \*ID)  
{  
     
        sleep(1);  
        int writer\_id = (int \*)ID;  
        sem\_wait(&write\_msg);  
        int key = 0;  
        char msg[100];  
        int choice;  
        do  
        {  
            printf("\n\nWriter Number %d Writing... ", writer\_id);  
            printf("\n\n1.Write A Message In A Tree.");  
            printf("\n\n2.Exit.");  
  
            printf("\n\nEnter Your Choice: ");  
            fflush(stdin);  
            scanf("%d", &choice);  
            if (choice <= 0 || choice > 2)  
            {  
                printf("\n\nRedo...Enter Any Key");  
                 
            }  
        } while (choice <= 0 || choice > 2);  
  
        if (choice == 1)  
        {  
            do  
            {  
                printf("\nEnter A Key: ");  
                scanf("%d", &key);  
                fflush(stdin);  
                if (IsKey(key, root))  
                {  
                    printf("\nKey Already Exist...Press Any Key To Redo\n");  
                   
                }  
            } while (IsKey(key, root));  
            printf("\nEnter A Message: ");  
            scanf("%s", msg);  
            if (strcmp(msg, "\n") == 0)  
            {  
                printf("\nMessage Error: Proper Message Require\n");  
                exit(0);  
            }  
            root = Insert(root, key, msg);  
        }  
        else if (choice == 2)  
        {  
            printf("\n\nWriter Number %d Exit.", writer\_id);  
            pthread\_cancel(pthread\_self());  
        }  
        sem\_post(&write\_msg);  
}  
  
//Reader Execution  
void \*Reader(void \*ID)  
{  
     
        sleep(1);  
        int reader\_id = (int \*)ID;  
        if (root == NULL)  
            sleep(1);  
           
        sem\_wait(&mutex);  
        readers\_count += 1;  
        if (readers\_count == 1)  
            sem\_wait(&write\_msg);  
        sem\_post(&mutex);  
  
        //Critical Section msg lock  
        int choice;  
        sem\_wait(&msg\_lock);  
        do  
        {  
           
            printf("\n\nReader Number %d Reading.. ", reader\_id);  
            printf("\n\n1.Read A Message From A Tree.");  
            printf("\n\n2.Exit.");  
  
            printf("\n\nEnter Your Choice: ");  
            fflush(stdin);  
            scanf("%d", &choice);  
            if (choice <= 0 || choice > 2)  
            {  
                printf("\n\nRedo...Enter Any Key");  
                 
            }  
        } while (choice <= 0 || choice > 2);  
          sem\_post(&msg\_lock);  
  
      //End of Critical Section msg lock  
  
        if (choice == 1)  
        {  
            InOrder\_Traversal(root);  
        }  
        else if (choice == 2)  
        {  
            printf("\n\nReader Number %d Exit.", reader\_id);  
            pthread\_cancel(pthread\_self());  
        }  
        //End of Critical Section  
  
        sem\_wait(&mutex);  
        readers\_count -= 1;  
        if (readers\_count == 0)  
            sem\_post(&write\_msg);  
        sem\_post(&mutex);  
     
}  
  
int main()  
{  
    int i = 0, readers = 0, writers = 0;  
    sem\_init(&mutex, 0, 1);  
    sem\_init(&msg\_lock, 0, 1);  
    sem\_init(&write\_msg, 0, 1);  
    pthread\_t Readers[100], Writers[100];  
    printf("\n\nEnter Number Of Readers - MAX(100): ");  
    if (scanf("%d", &readers) < 1 || readers > 100)  
    {  
        printf("\nOut Of Bound");  
        exit(0);  
    }  
    printf("\n\nEnter Number Of Writers - MAX(100): ");  
    if (scanf("%d", &writers) < 1 || writers > 100)  
    {  
        printf("\nOut of Bound");  
        exit(0);  
    }  
  
    for (i = 0; i < writers; i++)  
        pthread\_create(&Writers[i], NULL, Writer, (void \*)i);  
    for (i = 0; i < readers; i++)  
        pthread\_create(&Readers[i], NULL, Reader, (void \*)i);  
    for (i = 0; i < writers; i++)  
        pthread\_join(Writers[i], NULL);  
    for (i = 0; i < readers; i++)  
        pthread\_join(Readers[i], NULL);  
    sem\_destroy(&mutex);  
    sem\_destroy(&write\_msg);  
    sem\_destroy(&msg\_lock);  
    printf("\n\n");  
    return 0;  
}

**Conclusion:**

Hence, by using the semaphore we have got the synchronization between the reader and writer process and with the help of it the readers and writers don't process simultaneously and the shared data is now free from race condition.