Report

Jatin Puri s3832666

https://github.com/S3832666/Programming_Ass1_OSP

B: The Readers and Writers Problem

The Readers and Writers problem is when an object such as a file is shared between many processes. The processes include readers – people that only want to read the file and writers- people that want to edit the file. If one of the writers edits the file, no one is allowed to read or write at the same time or else the changes will not be seen by them. If someone is just reading the file, others can read it at the same time too. This problem manages synchronization to ensure there are no problems with the object data.

Solution Concept:

Create a mutex for the resource to lock the resource for the using thread. The reading threads will try to lock the resource to prevent to lock out the writer threads but the reader threads will not wait for the resource to be unlocked to reader. The writer thread shall wait for the resource to be unlocked before writing a random value in it.

We use the function pthread_mutex_trylock to check if the resource has already been used by a writer. This way, only one writer can change the value of the resource at any time. The same thing is also applied for the reader.

Limitations and Real-Life Application:

The problem with mutex is that the process is structured in a polling type system where the thread continuously monitors the status of the mutex. This process takes a lot of computing power and may waste resources. With more threads running, the performance of the overall system may reduce. It may be desirable depending on the situation to use an interrupt style system where the system sleeps when there is an on-going use of the system.

In systems like Twitter, the database is the central resource where all the users read and write to. This system uses a more sophisticated system than mutex (mutex + token) to maintain the integrity of the data in the database. The problem of race condition and deadlocking is prevalent in systems when there are multiple readers and writers operating on a single resource. Twitter is one example and there are a lot more.

Another example is the airline reservation system which contains a large database with processes that read and write data. There are no problems when you are reading information from the database as nothing is being changed. However there are problems when you are writing information to the database. There needs to be restrictions on access to the data base otherwise data can change anytime. By the time a reading process displays the result of a request for information to the user, the data in the data base could have already been changed. For example , there is a process that reads the number of available seats on a flight, finds one seat and informs the customer. Before the customer can book the seat they have been informed about, another process does the same and reserves the same seat for

another customer, making the available seats equal zero. These problems are solves using semaphores, monitors and message passing.

Code Screenshot

```
// B: The Readers and Writers Problem
// Solution:
// Create a mutex for the resource to lock the resource for the using thread.
// The reading threads will try to lock the resoure to prevent to lock out
// the writer threads but the reader threads will not wait for the resource
// to be unlocked to reader. The writer thread shall wait for the resource
// to be unlocked before writing a random value in it.
#include <iostream>
#include <sstream>
#include <string>
#include <unistd.h>
#include <pthread.h>

using namespace std;

#define COUNT 5

// Global variables
pthread_mutex_t mutex;
int resource = 0;
bool running = true;

// Function prototypes for the threads
void *reader_thread(void *ptr);
void *writer_thread(void *ptr);

// Main entry point of the program
int main()
{
    // Initial the variables
    pthread_mutex_init(&mutex, NULL);
    pthread_t readers[COUNT], writers[COUNT];

    // Make the threads
    for (int i = 0; i < COUNT; i++)
    {
        pthread_create(&writers[i], NULL, writer_thread, (void *)(long)i);
        pthread_create(&readers[i], NULL, reader_thread, (void *)(long)i);
    }
```

```
    // Run for 10 seconds
    sleep(10);

    // End the simulation
    running = false;

    // Wait for the child threads to exit before cleaning up
    for (int i = 0; i < COUNT; i++)
    {
        pthread_join(writers[i], NULL);
        pthread_join(readers[i], NULL);
    }

    // Destroy mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}

// Definition of the threads
void *reader_thread(void *ptr)
{
    int index = (int)(long)ptr;
    stringstream ss;

    while (running)
    {
        // Try to lock the mutex to prevent writing
        int res = pthread_mutex_trylock(&mutex);

        ss.str("");
        ss << "Reader " << index << " reads the value as " << resource << "." <<
endl;
        cout << ss.str();

        if (res == 0)
            pthread_mutex_unlock(&mutex);

        // Allow other threads to run
        pthread_yield();
    }

    return 0;
}

void *writer_thread(void *ptr)
{
    int index = (int)(long)ptr;
```

```
    stringstream ss;

    while (running)
    {
        // Try to lock the mutex to prevent writing
        if (pthread_mutex_trylock(&mutex) == 0)
        {
            resource = rand();
            ss.str("");
            ss << "Writer " << index << " writes the value of " << resource << ".
" << endl;
            cout << ss.str();
            pthread_mutex_unlock(&mutex);
        }

        // Allow other threads to run
        pthread_yield();
    }

    return 0;
}
```

D: The Sleeping Barbers Problem

The sleeper barber problem is explained by imagining a barber that cuts people's hair. There is one barber, one barber chair and N chairs for customers waiting for a cut. The barber can only cut one person's hair at a time. If the barber starts cutting someone's hair , it will cut the hair till the job is finished. If there is no one customer for a haircut, the barber will go to sleep in the barber chair. If a customer arrives, the barber will wake up to cut hair. If there are more customers , they either sit in any of the N chairs if they are empty or leave if the chairs are full. The goals is to program the barber and customer without getting into race problems.

Solution Concept:

We will create a thread for each barber and one thread for arriving customers. For each barber, we will create a mutex that will lock if it is currently serving a customer. If it's unlock, its ready to accept a new customer. A condition variable is used to signal the barbers that a customer has arrived. The barber will wait for the condition before it awakes to serve the customer.

We use the pthread_cond_signal function to inform the barbers that a customer has arrived in the salon. All the barbers are using the pthread_cond_wait function to wait for the arrival of a customer. With this

system, the first barber to receive the signal will be the one to serve the barber. This is some sort of a queuing problem.

Limitations and Real-life Application:

The problem with mutex is that the process is structured in a polling type system where the thread continuously monitors the status of the mutex. This process takes a lot of computing power and may waste resources. With more threads running, the performance of the overall system may reduce. It may be desirable depending on the situation to use an interrupt style system where the system sleeps when there is an on-going use of the system.

Autonomous driving systems such as those that in Tesla rely on sensor signals to perform action to correct the path of the vehicle. With all the sensor threads sending signal to the computer, some signals may be ignored by the CPU due to bombardment. Hence, this system implements queueing style system to schedule the processing of the signals. Just like in our problem, we use mutex and condition signals to react to each signal immediately and in correct sequence.

Another real world application for the sleeping barber problem is a customer care call Centre. When no customers are on the call, all call centre agents take it easy and wait for the call. When one person calls , he/she is connected to any of the agents and when all the agents are busy , the customer will have to wait in line until they are connected to an agent. If all the agents are busy and all the waiting lines are full , the customer is disconnected with a message that all lines are busy and to contact later. The customers are picked in a FIFO manner and it is designed in a manner that every agent will get at least one call .

Code Screenshot:

```
// D: The Sleeping Barbers Problem

// Solution:
// We will create a thread for each barber and one thread for arriving customers.
// For each barber, we will create a mutex that will lock if it is currently serv
ing
// a customer. If its unlock, its ready to accept a new customer. A condition var
iable
// is used to signal the barbers that a customer has arrived. The barber will wai
t for
// the condition before it awakes to serve the customer.
#include <iostream>
#include <string>
#include <sstream>
#include <unistd.h>
#include <time.h>
#include <pthread.h>

using namespace std;

#define BARBERS 5

bool running = true;
```

```
int customer_index = 0;
pthread_mutex_t barbers[BARBERS];
pthread_cond_t customer_arrived;

// Function prototype for the threads.
void *customer_thread(void *ptr);
void *barber_thread(void *ptr);

// Function to sleep for specified milliseconds.
void msleep(int msec);

// Main entry point of the program
int main()
{
    pthread_t barber_threads[BARBERS], customer_threadt;

    // Initialize cond and mutex variables
    for (int i = 0; i < BARBERS; i++)
        pthread_mutex_init(&barbers[i], NULL);

    pthread_cond_init(&customer_arrived, NULL);

    // Create the threads
    pthread_create(&customer_threadt, NULL, customer_thread, NULL);
    for (int i = 0; i < BARBERS; i++)
    {
        pthread_create(&barber_threads[i], NULL, barber_thread, (void *)(long)i);
    }

    // Run for 10 seconds
    sleep(10);

    // End the simulation
    running = false;

    // Send signals to all threads to stop
    for (int i = 0; i < BARBERS; i++)
        pthread_cond_signal(&customer_arrived);

    // Wait for the child threads to exit before cleaning up
    pthread_join(customer_threadt, NULL);
    for (int i = 0; i < BARBERS; i++)
    {
        pthread_join(barber_threads[i], NULL);
    }

    // Destroy cond and mutex variables
    for (int i = 0; i < BARBERS; i++)
```

```
        pthread_mutex_destroy(&barbers[i]);

    pthread_cond_destroy(&customer_arrived);

    return 0;
}

// Definition of the threads
void *customer_thread(void *ptr)
{
    while (running)
    {
        // Wait for a customer to arrive
        msleep(rand() % 1000);

        // Pick an empty
        for (int i = 0; i < BARBERS; i++)
        {
            // This slot is available
            if (pthread_mutex_trylock(&barbers[i]) == 0)
            {
                pthread_mutex_unlock(&barbers[i]);
                customer_index++;
                cout << "Customer " << customer_index << " arrives." << endl;
                pthread_cond_signal(&customer_arrived);
                break;
            }
        }

        // Yield to other process
        pthread_yield();
    }

    return 0;
}

void * barber_thread(void * ptr)
{
    int index = (int)(long)ptr;

    while (running)
    {
        // Wait for a customer to arrive
        pthread_cond_wait(&customer_arrived, &barbers[index]);

        if (!running)
            break;
```

```
        // Wake up the barber and serve customer
        cout << "Barber " << index << " is awake and serving customer ";
        cout << customer_index << "." << endl;

        // Yield while serving this customer
        msleep(rand() % 3000);

        // The barber sleeps
        cout << "Barber " << index << " is asleep." << endl;
        pthread_mutex_unlock(&barbers[index]);
        pthread_yield();
    }

    return 0;
}

// Definition of msleep function
void msleep(int msec)
{
    struct timespec ts;
    ts.tv_sec = msec / 1000;
    ts.tv_nsec = (msec % 1000) * 1000000;

    nanosleep(&ts, &ts);
}
```