600086 Lab Book

Week 6 – Lab H

Date: 24th Mar 2022

Exercise 1. Condition variables

Questions:

1. create a rust program using a producer consumer architecture with condition variables to prevent race conditions
2. modify the code to allow the code to work with multiple consumers

Solution:

1.

```
use std::sync::{Arc, Mutex, Condvar};
pub fn main()
{
    println!("Begin");
    let D = Arc::new(Data::new());

    let D_clone = D.clone();

    let loops = 5;

    let producers = std::thread::spawn(move || producer_main( &D,loops));
    let consumers = std::thread::spawn(move || consumer_main(&D_clone,
loops));

    producers.join();
    consumers.join();

    println!("Cease");

}

pub fn producer_main(data : & Data, loop_limit : u32)
{
    let data = data;
    for _i in 0..loop_limit
    {
        let mut full = data.value.lock().unwrap();
        while *full
        {
            full = data.condition.wait(full).unwrap();
        }
        //"produce" by setting the mutex value to true
        *full = true;
```

```
        println!("Producer_ID: {} has produced",_i);
        data.condition.notify_all(); // notify any waiting threads
    }
}

pub fn consumer_main(data : & Data,loop_limit : u32)
{
    for _i in 0..loop_limit
    {
        let mut full = data.value.lock().unwrap();
        while !*full // de reference the guard variable
        {
            full = data.condition.wait(full).unwrap();
        }
        *full = false;
        println!("Consumer_ID: {} has consumed",_i);
        data.condition.notify_all();
    }
}

pub struct Data
{
    condition : Condvar,
    value : Mutex<bool>
}
impl Data
{
    pub fn new() -> Data
    {
        Data
        {
            condition : Condvar::new(),
            value : Mutex::new(false)
        }
    }
}
```

The above code implements a struct called Data which has a CondVal and a mutex Boolean value the Mutex bool is the value of the data and the CondVal is there to organise the access to the variable

When run it produces the output shown in the sample output section ref 1

2. To run the code with multiple consumers and producers I created a thread_pool variable and pushed multiple threads of each kind to create multiple consumer and producer threads.

```
pub fn main()
{
    println!("Begin");
    let D = Arc::new(Data::new());
    let mut thread_pool = Vec::new();

    let loops = 5;
    let num_producers = 2;
    let num_consumers = 2;

    for _i in 0..num_producers
    {
        let d_clone = D.clone();
        thread_pool.push(std::thread::spawn(move || producer_main(&d_clone,
loops)));
    }
    for _i in 0..num_consumers
    {
        let d_clone = D.clone();
        thread_pool.push(std::thread::spawn(move || consumer_main(&d_clone,
loops)));
    }
    for t in thread_pool
    {
        t.join();
    }

    println!("Cease");

}
```

Test data:

N/A

Sample output:

ref	output
1	Begin Producer_ID: 0 has produced Consumer_ID: 0 has consumed Producer_ID: 1 has produced Consumer_ID: 1 has consumed Producer_ID: 2 has produced Consumer_ID: 2 has consumed Producer_ID: 3 has produced Consumer_ID: 3 has consumed Producer_ID: 4 has produced Consumer_ID: 4 has consumed Cease

2	Begin Producer_ID: 0 has produced Consumer_ID: 0 has consumed Producer_ID: 1 has produced Consumer_ID: 1 has consumed Producer_ID: 0 has produced Consumer_ID: 0 has consumed Producer_ID: 2 has produced Consumer_ID: 2 has consumed Producer_ID: 3 has produced Consumer_ID: 3 has consumed Producer_ID: 4 has produced Consumer_ID: 4 has consumed Producer_ID: 1 has produced Consumer_ID: 1 has consumed Producer_ID: 2 has produced Consumer_ID: 2 has consumed Producer_ID: 3 has produced Consumer_ID: 3 has consumed Producer_ID: 4 has produced Consumer_ID: 4 has consumed Cease

Reflection:

Part one I produced a producer consumer structured program, part 2 I scaled this program to work with multiple consumers and producers however this is not a massively scalable solution as there is still only one piece of data being accessed meaning it bottlenecks the parallel program and makes it very concurrent slightly defeating the purpose.

Metadata:

Produce, consumer, architecture, Condvar

Further information:

N/A

Q2. Striped Arrays

Question:

1. Add timing code to the provided program to measure the data access time, vary the number of threads to check how this changes the value
2. Modify the code to simulate random access

Solution:

1. Added a start_time and end_time variable to the code that wraps the thread creation and joining to time the data access portion of the programme I also modified the strip sizes so that it is the total divided by the number of threads so that the same number of accesses are made.

```
fn main() {

    println!("Begin");

    let num_of_threads: usize = 1;
    let mut list_of_threads: Vec<JoinHandle<()>> = vec!();
    let shared_data: Arc<Data> = Arc::new( data: Data::new(num_of_threads, len: 32767/num_of_threads));
    let start_time: SystemTime = SystemTime::now();
    for id: usize in 0..num_of_threads {
        let data_clone: Arc<Data> = shared_data.clone();
        list_of_threads.push( std::thread::spawn( move || thread_main(id, data_clone) ) );
    }

    for t: JoinHandle<()> in list_of_threads {
        t.join().unwrap();
    }
    let end_time: u128 =start_time.elapsed().unwrap().as_micros();
    for i: usize in 0..shared_data.length_of_strip*shared_data.num_of_strips {
        println! ("{} : {}", i, shared_data._read(i));
    }
    println!("Total data access time: {}us",end_time);
    println!("End");
}
```

2. To simulate random access I implemented the following code as per the lab notes in the thread_main

```
fn thread_main(id: usize, data: Arc<Data>) {
    for _i: i32 in 0..10 {
        for _j: usize in 0..data.length_of_strip*data.num_of_strips {
            let index: usize = rand::random::<usize>() % data.length_of_strip*data.num_of_strips;
            data.write(index, value: id);
        }
    }
}
```

Test data:

N/A

Sample output:

No. of threads	Output - seq	Output - rand
1	32766 : 0 Total data access time: 3443us End	32766 : 0 Total data access time: 6557us End
2	32765 : 1 Total data access time: 8801us End	32765 : 1 Total data access time: 20398us End

4	32763 : 1 Total data access time: 42904us End	32763 : 0 Total data access time: 60417us End
8	32759 : 3 Total data access time: 186250us End	32751 : 14 Total data access time: 871157us End
16	32751 : 14 Total data access time: 471790us End	32751 : 11 Total data access time: 990024us End
32	32735 : 20 Total data access time: 969471us End	32735 : 14 Total data access time: 1955849us End

Reflection:

In the sequential implementation the run time gets greater every time as the amount of work being done is very small compared to the overhead of spinning up and joining threads.

When implementing the random-access code. Expected that the code would have roughly the same access time however on average the access time was roughly double that of the sequential implementation

Metadata:

Stripped Array

Further information:

N/A