600086 Lab Book

Week 5 – Lab E

Date: 9th Mar 2022

Q1. Thread safe printing

Question:

Build and run the unsafe_print folder

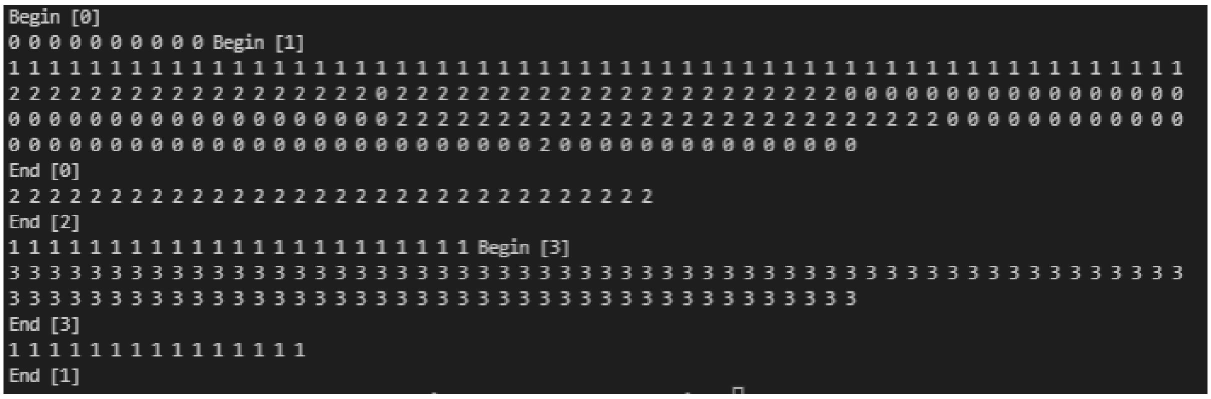
Solution:

```rust
fn main() {
    let num_of_threads = 4;
    let mut array_of_threads = vec!();

    for id in 0..num_of_threads {
        array_of_threads.push(std::thread::spawn(move || print_lots(id)) );
    }

    for t in array_of_threads {
        t.join().expect("Thread join failure");
    }
}

fn print_lots(id: u32) {
    println!("Begin [{}]", id);
    for _i in 0..100 {
        print!("{} ", id);
    }
    println!("\nEnd [{}]", id);
}
```

Test data:

N/A

Sample output:



```
Begin [0]
0 0 0 0 0 0 0 0 0 0 Begin [1]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
End [0]
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
End [2]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 Begin [3]
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
End [3]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
End [1]
```

Reflection:
The prints appear out of order and seemingly random as each thread is executing entirely independently this means that there is nothing to control the order and ensure that they are printed in sync. This is known as a race condition as each thread is effectively racing to print first.

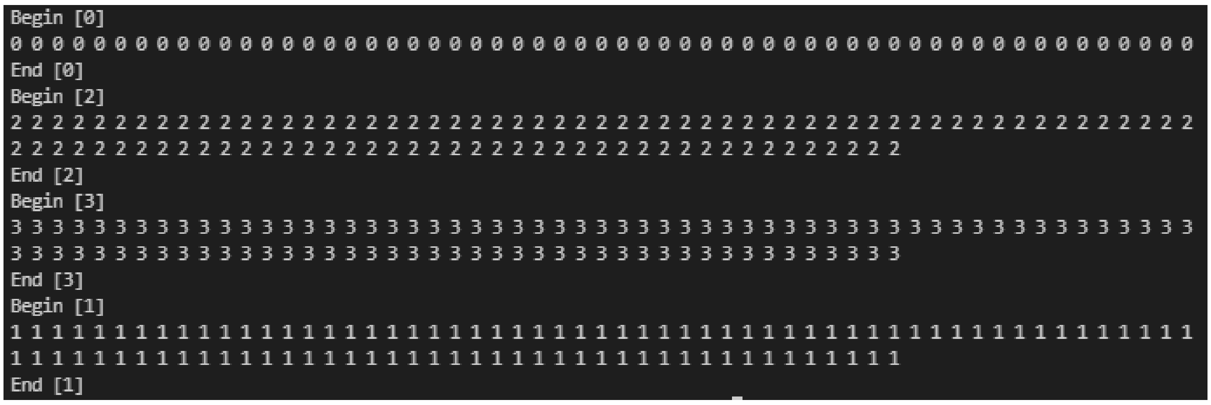
Metadata:
Race Conditions Unsafe Print

Further information:
none

Question:
Reproduce the unsafe_print code accounting for the race condition

Solution:
```rust
use std::sync::{Arc, Mutex};

fn main() {
    let num_of_threads = 4;
    let mut array_of_threads = vec!();
    let counter = Arc::new(Mutex::new(0));

    for id in 0..num_of_threads {
        let counter_clone = counter.clone();
        array_of_threads.push(std::thread::spawn(move ||
print_lots(id,counter_clone)) );
    }

    for t in array_of_threads {
        t.join().expect("Thread join failure");
    }
}

fn print_lots(id: u32, c:Arc<Mutex<u32>>) {

    let _guard = c.lock().unwrap();
    println!("Begin [{}]", id);
    for _i in 0..100 {
        print!("{} ", id);
    }
    println!("\nEnd [{}]", id);
}
```

Test data:
N/A

Sample output:

```
Begin [0]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
End [0]
Begin [2]
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
End [2]
Begin [3]
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
End [3]
Begin [1]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
End [1]
```

Reflection:
The output is still not in order but each section is uninterrupted. To achieve this I created a counter Arc<Mutex<u32>> and passed this into the thread when a thread entered the function it locked the. Arc<Mutex<u32>> which was a shared asynchronous reference this meant that the other threads

could not continue to the print function until the current thread released the lock when it fell out of scope.

What happens to your code if you fail to release the mutex?

Should the mutex fail to be released then the code will not complete as the remaining threads would be locked out of completing indefinitely.

What happens if you raise an exception within the critical section?

Should the print statement return an error or throw an exception for any reason then the function would be left without leaving scope keeping the lock in place.

There is also still a race condition albeit a slightly less crucial one as the threads are now effectively racing to be the thread to lock the Arc<Mutex<u32>> and as such the order of prints is not guaranteed.

Metadata:

Safe print ARC Mutex Lock

Further information:

How do you raise an exception that can cause the thread to end silently so as to prove the principles stated above?

Q2. Triangles using OpenGL

Question:

Adapt the provided code to produce a more chaotic movement of triangles.

Solution:

```
    let mut delta_x: f32 = -0.5;
    let mut delta_y: f32 = -0.5;
    let mut randomizer= rand::thread_rng();
//
…
// Begin render loop

        // Animation counter
        let x:f32 = randomizer.gen_range(-0.02..0.02);
        let y:f32 = randomizer.gen_range(-0.02..0.02);
        delta_x += x;
        if delta_x > 0.7 {
            delta_x = -1.4;
        }
        if delta_x < -1.4{
            delta_x = 0.7
        }
        delta_y += y;
        if delta_y > 0.7 {
            delta_y = -1.4;
        }
        if delta_y < -1.4{
            delta_y = 0.7
        }

        // Create a drawing target
        let mut target = display.draw();

        // Clear the screen to black
        target.clear_color(0.0, 0.0, 0.0, 1.0);

        // Iterate over the 10 triangles
        for i in 0 .. 10 {

            // Calculate the position of the triangle
            let pos_x : f32 = delta_x + ((i as f32) * 0.1);
            let pos_y : f32 = delta_y + ((i as f32) * 0.1);
            let pos_z : f32 = 0.0;

            // Create a 4x4 matrix to store the position and orientation of
the triangle
            let uniforms = uniform! {
                matrix: [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [pos_x, pos_y, pos_z, 1.0],
                ]
            };

            // Draw the triangle
            target.draw(&vertex_buffer, &indices, &program, &uniforms,
&Default::default()).unwrap();
        }

        // Display the completed drawing
        target.finish().unwrap();

        // End render loop
```

Test data:

N/A

Sample output:

Can’t be shown as it is video but will present in Lab.

Reflection:

Modified delta to be replaced by two variables one governing the X dimension increment and the other governing the y dimension increment:

delta_x

delta_y

I also introduced a random variable that can be used to generate a random number in a range as done at the start of the //Animation Counter section. I also allowed the triangles to move in all directions by introducing negative increment values into the range. Because I had now implemented negative movement, I had to expand the value checks to allow for the reverse case otherwise the triangles can travel backwards indefinitely.

Metadata:

OpenGL

Further information:

N/A