Atomics and visibility problem

Safe concurrent operations Integer operations

Stefan Schindler @dns2utf8

Wednesday, 24. October 2018

Rust Zürichsee Meetup hosted by coredump.ch

Table of contents

Introduction

Who I am?

My name is Stefan and I ...

- studied Computer Science at HSR
- started with Rust in 2016
- maintain crates: threadpool, wipe_buddy, son_grid_engine, ... some more
- organize RustFest.eu next up in Rome November 24th to 27th
- talk about rust
- am looking for a new rusty job

What will we learn tonight?

- What is a Symbol?
- What is a Register?
- What is an Atomic Operation?
- What is the visibility problem?
- How to solve it?

Integer Data

Where is index?

```
let data = vec![42, 42, 42, 42];

let mut index = 0; // <-- what is the kind of index?
let length = data.len();
while index < length {
    println!("{}: {}", index, data[index]);
    index += 1
}
```

and where will it be stored at runtime?

Watching a value from another thread

Scenario

- Control thread allocates global memory for `threshold`
- $Thread_W$ will wait for `threshold` to pass a certain value and then alert the user
- $Thread_S$ is waiting over input from the world and updates `threshold`

Monitor data thread - 0

```
use std::thread::{sleep, spawn};
use std::time::Duration;
#[allow(non_upper_case_globals)]
static mut threshold: isize = 0;
const MAX_TEST: usize = 100000;
fn main() {
  let _counter = spawn(|| {
    loop {
// note: mutable statics can be mutated by multiple
// threads: aliasing violations or data races will
// cause undefined behavior
      unsafe {
          threshold = (threshold + 1) % 100;
          //println!("counter: {}", threshold);
      }
    }
```

Monitor data thread - 1

```
let watcher = spawn(|| {
    sleep(Duration::from_millis(500));
    let mut history = Vec::with_capacity(MAX_TEST);
    let mut last = unsafe { threshold };
    let mut count = 0;
    for _ in 0..MAX_TEST {
        let threshold_local = unsafe { threshold };
        if last == threshold_local {
            count += 1;
        } else {
            history.push((last, count));
            last = threshold_local;
            count = 0;
        }
    }
    history
```

Monitor data thread - 2

```
// back in fn main() { ...
  let history = watcher.join().expect("watcher failed");
  println!("{:?}\nn transitions recorded: {}"
           , history, history.len());
  //_counter.join();
}
```

What kind of out put would you expect?

Monitor data thread - 3

Debug mode

```
[ ...
    (88, 0), (93, 0), (98, 0), (4, 0), (10, 0),
    (15, 0), (20, 0), (26, 0), (31, 0), (36, 0),
    (41, 0), (46, 0), (53, 0), (58, 0), (63, 0)
]
n transitions recorded: 99769
```

Now we want more speed. What to do?

Monitor data thread - 4

Debug mode

```
[ ...
    (88, 0), (93, 0), (98, 0), (4, 0), (10, 0),
    (15, 0), (20, 0), (26, 0), (31, 0), (36, 0),
    (41, 0), (46, 0), (53, 0), (58, 0), (63, 0)
]
n transitions recorded: 99769
```

Release mode

```
[]
n transitions recorded: 0
```

What happend? Why did it stop working? Feel free to guess

Monitor data thread - 5

A new counter function:

```
let _counter = spawn(|| {
  let threshold_ptr = unsafe {
                        &mut threshold as *mut isize };
  loop {
      unsafe {
          write_volatile(
            threshold_ptr,
            (read_volatile(threshold_ptr) + 1) % 100);
      }
  }
});
```

Atomic Operations

Atomic Access

From page 117 section 7.3.2 [2]

Cacheable, naturally-aligned single loads or stores of up to a quadword are atomic on any processor model, as are misaligned loads or stores of less than a quadword that are contained entirely within a naturally-aligned quadword. Misaligned load or store accesses typically incur a small latency penalty. Model-specific relaxations of this quadword atomicity boundary, with respect to this latency penalty, may be found in a given processor's Software Optimization Guide. Misaligned accesses can be subject to interleaved accesses from other processors or cache-coherent devices which can result in unintended behavior.

Atomicity for misaligned accesses can be achieved where necessary by using the **XCHG** instruction or any suitable **LOCK**-prefixed instruction. Note that misaligned locked accesses may incur a significant performance penalty on various processor models.

The LOCK prefix F0

From page 112 section 3.5.1.3 [1]

The LOCK prefix causes certain read-modify-write instructions that access memory to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve locking of data-cache lines that contain copies of the referenced memory operands, and/or bus signaling or packet-messaging on the bus). The prefix is intended to give the processor exclusive use of shared memory operands in a multiprocessor system.

The prefix can only be used with forms of the following instructions that write a memory operand: **ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG , NOT, OR, SBB, SUB, XADD, XCHG, and XOR.** An invalid-opcode exception occurs if **LOCK** is used with any other instruction.

For further details on these prefixes, see “Lock Prefix” in Volume 3 [3].

Performance differences

Old Intel performance for Atomic Interger Operation: 20 - 120 cycles

Old AMD performance for Atomic Integer Operation: 40 cycles

Most recent AMD architecture[1] online TODO...

The counter race - 0

Let s be clever and fast!

```
const N_PARTIES: usize = 4;
const N_INCREMENTS: usize = 100000;

static GLOBAL_COUNTER: usize = 0;
```

The counter race - 1

```
pub fn counter_race() {
    (0..N_PARTIES).map(|_i| {
        spawn(move || {
  let counter_ptr = unsafe { &mut GLOBAL_COUNTER as *mut usize };
            for _ in 0..N_INCREMENTS {
                unsafe {
  write_volatile(counter_ptr, read_volatile(counter_ptr) + 1);
                }
            }
        })
    })
    .collect::<Vec<_>>()    .into_iter()
    .for_each(|t| t.join().expect("counter thread failed"));
let counter_ptr = unsafe { &mut GLOBAL_COUNTER as *mut usize };
    println!("expected: {}, got: {}", N_PARTIES * N_INCREMENTS,
            unsafe { read_volatile(counter_ptr) });
```

The counter race - 2

read_volatile and write_volatile

```
expected: 400000, got: 129861
```

What to do?

What do we know about the result? Do we have a lower band of what we can expect?

The counter race - 3

```
static GLOBAL_ATOMIC_COUNTER: AtomicUsize = ATOMIC_USIZE_INIT;
pub fn counter_race_atomic() {
    (0..N_PARTIES).map(|_| {
        spawn(|| {
            for _ in 0..N_INCREMENTS {
                GLOBAL_ATOMIC_COUNTER.fetch_add(1, Ordering::Relaxed);
            }
        })
    })
    .collect::<Vec<_>>()
    .into_iter()
    .for_each(|t| t.join().expect("counter thread failed"));

println!("expected: {}, got: {}", N_PARTIES * N_INCREMENTS,
        GLOBAL_ATOMIC_COUNTER.load(Ordering::SeqCst));
}
```

The counter race - 4

read_volatile and write_volatile

```
expected: 400000, got: 129861
```

Atomic .fetch_add and .load

```
expected: 400000, got: 400000
```

Hurray!

Conclusion

Summary

Multi-Thread-Programms require atomic operations. Using them with an abstraction allows us to keep the development speed up.

Slides: dns2utf8/atomics_and_visibility_problem

Questions?

Manipulate data in RAM - 0

What if we have a list of objects and we need the value furthest away from Zero?

```
fn absolute_max(result: &mut i64, list: &Vec<i64>) {
    for i in list {
        let abs = if i < 0 { -i } else { i };
        if result < abs {
            result = i;
        }
    }
}
```

Quick question: *Where is the data?* and why does it not compile?

Manipulate data in RAM - 1

Making it compile with rust reveals the problem:

```
fn absolute_max(result: &mut i64, list: &Vec<i64>) {
    for i in list {
        let i = *i;
        let result_local = *result;

        let abs_i = if i < 0 { -i } else { i };
        let abs_r = if result_local < 0 { -result_local } else { result
        if abs_r < abs_i {
            *result = i;
        }
    }
}
```

Full source:

https://play.rust-lang.org/?gist=11b541f0b4165f1cc39472c15f494a00&

References i


AMD, Reading, MA.
AMD64 Architecture Programmes's Manual, Volume 1: Application Programming, December 2017.
Revision 3.22: amd.com/system/files/TechDocs/24592.pdf.


AMD, Reading, MA.
AMD64 Architecture Programmes's Manual, Volume 2: System Programming, September 2018.
Revision 3.30: amd.com/system/files/TechDocs/24594.pdf.


AMD, Reading, MA.
AMD64 Architecture Programmes's Manual, Volume 3: General-Purpose and System Instructions, May 2018.
Revision 3.26: amd.com/system/files/TechDocs/24594.pdf.