

CS 554: Advanced Topics in Concurrent Computing Systems

Project Check-In

Correctness Analysis of Classic Mutual Exclusion Algorithms Under different Memory Models

Team Members

Amit Maheshwar Varanasi
Dhanush Bommavaram

1. Project Overview

As proposed, we are analyzing the correctness of Peterson's and Dekker's algorithms under various memory models (SC+NA, RA+NA, RA+NA+RLX, and NA). Our goal is to bridge the theoretical understanding of these models with practical systems programming on x86 and ARM architectures.

2. Progress to Date

We have successfully moved from the proposal phase into implementation and initial empirical testing. Github repository: <https://github.com/BDhanush/Concurrent-Algorithms-Analysis>. Our key achievements include:

- **Implementation of Algorithms:** We have written C++ implementations using `thread` class and atomics for both Peterson's and Dekker's algorithms. We have covered all four targeted memory model configurations:
 - `memory_order_seq_cst` (SC+NA)
 - `memory_order_acq_rel` (RA+NA)
 - `memory_order_relaxed` (RA+NA+RLX)
 - No atomic operations (NA)
- **Empirical Testing & Results:** We executed these implementations on both x86 and ARM processors and checked for data race using `Clang's ThreadSanitizer` tool. Our initial findings confirm that:
 - **Success:** The algorithms function correctly without data races **only** under `memory_order_seq_cst` (SC+NA).
 - **Failures:** We observed data races in all other configurations (NA, RA+NA, RA+NA+RLX) on both processor architectures.
- **Bonus:** We implemented spin locks as well to test as well with and without CAS. As expected we find a data race in the implementation without CAS and no data race in the sequentially consistent implementation. Surprisingly enough we found a data race when using `memory_order_acq_rel` memory order, but replacing it with `memory_order_acquire` for `test_and_set()` and `memory_order_release` gets rid of the data race.

With atomic int:

```
void process(int id) {  
    while(true){  
        int expected = 0;  
        while (!lock.compare_exchange_strong(expected, 1,  
std::memory_order_acq_rel)) {  
            expected = 0;  
        }  
        a=!a;  
        lock.store(0, std::memory_order_acq_rel);  
    }  
}
```

Data race

```
void process(int id) {  
    while(true){  
        int expected = 0;  
        while (!lock.compare_exchange_strong(expected, 1,  
std::memory_order_acquire, std::memory_order_relaxed)) {  
            expected = 0;  
        }  
        a=!a;  
        lock.store(0, std::memory_order_release);  
    }  
}
```

No data race

With atomic flag:

```
void process(int id) {  
    while(true){  
        while(lock.test_and_set(std::memory_order_acq_rel));  
        a=!a;  
        lock.clear(std::memory_order_acq_rel);  
    }  
}
```

Data race

```
void process(int id) {  
    while(true){  
        while(lock.test_and_set(std::memory_order_acquire));  
        a=!a;  
        lock.clear(std::memory_order_release);  
    }  
}
```

No data race

3. Artifacts Created

- Source code for Peterson's algorithm, Dekker's algorithm and spin lock across all listed memory models.
- Binary generation script for x86 and ARM environments (for easier and more organized building).
- Structure

```
/  
tsan_build.sh  
  
Implementation_Type/  
  x86/  
    binaries  
  Arm/  
    binaries  
  source_code  
  .  
  .  
  .  
  .
```

4. Current Challenges and Next Steps

While we have strong empirical data confirming *where* the algorithms fail, our current challenge is constructing formal theoretical proofs to explain *why* they fail in the specific manner observed.

- **Theoretical Mapping:** We are currently working to map our assembly-level findings back to formal event structures to provide the theoretical analysis promised in the proposal.
- **Unexpected behaviour:** We have to also understand why the spin lock implementation with `memory_order_acq_rel` produces data races.

We are on track to complete the theoretical analysis within the remaining timeline.