Transactions: Making CMRX kernel internals lock-free

Eduard Drusa <nopisonnope@gmail.com>

Agenda:

1 The problem

2 Land of lock-free

3 The solution

4 What's new?

CMRX
realtime
microkernel

The Problem

CMRX interrupt handlers can use kernel services

This causes race conditions

Disabling interrupts in critical sections increases latency

Mutexes cause instant deadlock

CMRX
realtime
microkernel

Investigation

Ferdinand: "This looks a lot like problem for lock-free approach"

Research papers, also using LLMs

Suggested code reminds me of transactions

It has a lot of overhead


```
typedef struct {
    /* internal structure */
} complex_data_t;


complex_data_t shared_data;
void update_data(int input) {
    complex_data_t local;
    /* fill in local copy */

    mutex_lock(&data_lock);
    shared_data = local;
    mutex_unlock(&data_lock);
}

complex_data_t read_data(void) {
    complex_data_t local;
    mutex_lock(&data_lock);
    local = shared_data;
    mutex_unlock(&data_lock);
    return local;
}
```

Land of Lock-Free

Lock-free programming:

At least one context runs to completion without blocking

Wait-free programming:

All contexts run to completion without blocking

Blocking is traded for retrying (hello LOAD/STORE conditional)

Critical Section

Read+Write case:

```
mutex_lock(&data_lock);

complex_data_t * entry;
entry = /* find entry */;

/* modify data */
mutex_unlock(&data_lock);
```

Transaction

```
txn_id = txn_start();

complex_data_t * entry;
entry = /* find entry */;

if (txn_commit(txn_id, TXN_RW)) {
  /* modify data */
  txn_done();
}
```

CMRX
realtime
microkernel

Critical Section

Read-Only case:

```
mutex_lock(&data_lock);

complex_data_t * entry;
entry = /* find entry */;

/* examine data */
mutex_unlock(&data_lock);
```

Transaction

```
txn_id = txn_start();

complex_data_t * entry;
entry = /* find entry */;
/* examine data */

if (!txn_commit(txn_id, TXN_RO)) {
  /* data inconsistent */
}
```

CMRX
realtime
microkernel

Transactions

Transactions implement `read committed` level of isolation

Read+Write transaction invalidates all transactions started later on commit

Read-Only transaction invalidates nothing on commit

Any invalidated transaction will fail to commit

CMRX realtime microkernel

// here be dragons

The spirit of lock-free: Blocking is traded for iterating

Data structures develop some interesting properties

Uniprocessor: Entries are consistent, data structure may temporarilly not be

Multiprocessor: Both entries and whole data structures may be inconsistent

Why bother + Final thoughts

Readers don't block writers

Transaction implementation is really trivial

Opportunity for customized reaction to contention

Need for defensive programming

CMRX kernel case: Long lookup, short modifications, YMMV

It becomes really funny if you go multicore

What's new?

Support for FPU

ARM Cortex-M4 and M7, possibly other architectures

User-space mutexes

On platforms with atomics support

Porting efforts

POSIX hosted port (you are watching it right now)

Basic ARMv8M support, RISC-V and MIPS ports in progress

Community

Increased interest, contributors

Memory protection is still a curse word around here

CMRX
realtime
microkernel

Thank you!

Q&A?

GitHub Repository:

https://github.com/ventZl/cmrx

Project Website:

https://cmrxrtos.org/

Eduard Drusa <nopisonnope@gmail.com>