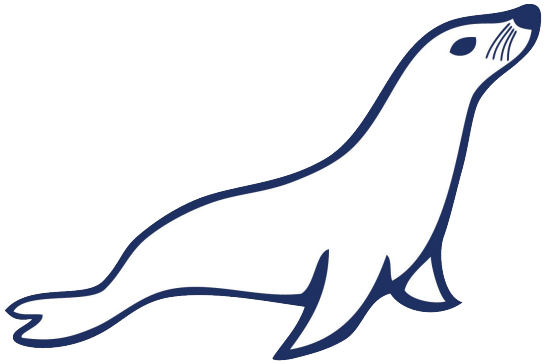

MariaDB®
FOUNDATION

Atomic operations in C

Sergey Vojtovich
Software Engineer @ MariaDB Foundation

Agenda

- API overview
- details, examples, common mistakes
- a few words about **volatile**

Rationale

Atomic operations are intended to allow access to shared data without extra protection (mutex, rwlock, …).

This may improve:

- single thread performance
- scalability
- overall system performance.

Atomic API

- NoAPI (asm, volatile, read/write/full memory barrier)

- GCC sync builtins
- Windows Interlocked Variable Access
- Solaris atomic operations

- GCC atomic builtins
- **C11 atomic operations**

- MySQL (removed in 8.0)
- InnoDB (disaster, removed in MariaDB 10.2)
- MariaDB (compatible with MySQL, but closer to C11)

MariaDB atomic API

Simple operations (cheaper):

- load
- store

Read-modify-write (more expensive):

- fetch-and-store
- add
- compare-and-swap

Additional C11 atomic operations

- test-and-set (similar to fetch-and-store)
- clear (similar to store)
- sub (same as add(-1))
- or
- xor
- and

Atomic load

```
my_atomic_load32(int32_t *var)
my_atomic_load64(int64_t *var)
my_atomic_loadptr(void **var)

my_atomic_load32_explicit(int32_t *var, memory_order)
my_atomic_load64_explicit(int64_t *var, memory_order)
my_atomic_loadptr_explicit(void **var, memory_order)

  return *var;

  Order must be one of MY_MEMORY_ORDER_RELAXED, MY_MEMORY_ORDER_CONSUME,
  MY_MEMORY_ORDER_ACQUIRE, MY_MEMORY_ORDER_SEQ_CST.
```

Atomic store

```
my_atomic_store#(&var, what)
my_atomic_store#_explicit(&var, what, memory_order)

  *var= what;

  Order must be one of MY_MEMORY_ORDER_RELAXED, MY_MEMORY_ORDER_RELEASE,
  MY_MEMORY_ORDER_SEQ_CST.
```

Atomic fetch-and-store

```
my_atomic_fas#(&var, what)
my_atomic_fas#_explicit(&var, what, memory_order)

  old= *var; var= *what; return old;

  All memory orders are valid.
```

Atomic add

```
my_atomic_add#(&var, what)
my_atomic_add#_explicit(&var, what, memory_order)

  old= *var; *var+= what; return old;

  All memory orders are valid.
```

Atomic compare-and-swap

```
my_atomic_cas#(&var, &old, new)
my_atomic_cas#_weak_explicit(&var, &old, new, succ, fail)
my_atomic_cas#_strong_explicit(&var, &old, new, succ, fail)

  if (*var == *old) { *var= new; return TRUE; }
  else { *old= *var; return FALSE; }

  succ - the memory synchronization ordering for the read-modify-write
  operation if the comparison succeeds. All memory orders are valid.
  fail - the memory synchronization ordering for the load operation if the
  comparison fails. Cannot be MY_MEMORY_ORDER_RELEASE or
  MY_MEMORY_ORDER_ACQ_REL and cannot specify stronger ordering than succ.

  The weak form is allowed to fail spuriously, that is, act as if
  *var != *old even if they are equal. When a compare-and-exchange is in
  a loop, the weak version will yield better performance on some platforms.
  When a weak compare-and-exchange would require a loop and a strong one
  would not, the strong one is preferable.
```

References

https://github.com/MariaDB/server/blob/10.3/include/my_atomic.h

http://en.cppreference.com/w/c/atomic

Example

mutex based solution	atomic based solution

```
pthread_mutex_lock(&LOCK_thread_count);
thd->query_id= global_query_id++;
pthread_mutex_unlock(&LOCK_thread_count);
```

Example

mutex based solution

```
pthread_mutex_lock(&LOCK_thread_count);
thd->query_id= global_query_id++;
pthread_mutex_unlock(&LOCK_thread_count);
```

atomic based solution

```
thd->query_id=
  my_atomic_add64_explicit(&global_query_id, 1,
                           MY_MEMORY_ORDER_RELAXED);
```

Example

mutex based solution

```
pthread_mutex_lock(&LOCK_thread_count);
thd->query_id= global_query_id++;
pthread_mutex_unlock(&LOCK_thread_count);
```

ACQUIRE + atomic read-modify-write
non-atomic load-increment-store
RELEASE + atomic store

atomic based solution

```
thd->query_id=
  my_atomic_add64_explicit(&global_query_id, 1,
                           MY_MEMORY_ORDER_RELAXED);
```

no memory barriers
atomic read-modify-write

The rule

Any time two (or more) threads operate on a shared variable concurrently, and one of those operations performs a **write**, **both** threads **must** use atomic operations.

Example

thread 1	thread 2
`my_atomic_store64_explicit(&var, 1, MY_MEMORY_ORDER_RELAXED);`	`if (var) { … }`

Not guaranteed to observe consistent value, subject of compiler optimizations.

Example

thread 1	thread 2
`my_atomic_store64_explicit(&var, 1, MY_MEMORY_ORDER_RELAXED);`	`if (var) { … }`

Not guaranteed to observe consistent value, subject of compiler optimizations.

BAD

Example

thread 1	thread 2
`var= 1;`	`if (my_atomic_load64_explicit(&var, MY_MEMORY_ORDER_RELAXED)) { … }`
Subject for compiler optimizations.	Not guaranteed to observe consistent value.

Example

thread 1	thread 2
`var= 1;`	`if (my_atomic_load64_explicit(&var, MY_MEMORY_ORDER_RELAXED)) { … }`
Subject of compiler optimizations.	Not guaranteed to observe consistent value

BAD

Example

thread 1	thread 2
`my_atomic_store64_explicit(&var, 1, MY_MEMORY_ORDER_RELAXED);`	`if (my_atomic_load64_explicit(&var, MY_MEMORY_ORDER_RELAXED)) { … }`

Example

thread 1	thread 2
`my_atomic_store64_explicit(&var, 1, MY_MEMORY_ORDER_RELAXED);`	`if (my_atomic_load64_explicit(&var, MY_MEMORY_ORDER_RELAXED)) { … }`

OK

The difference

By all means atomic operations do make foreign threads happy:

- other threads guaranteed to see consistent values
- disable some compiler optimizations, like: dead store elimination, constant folding
- allow memory barrier.

Atomicity

```
uint32_t foo= 0;

void store()
{
    foo= 0x80286;
}
```

Atomicity

```
uint32_t foo= 0;

void store()
{
    foo= 0x80286;
}
```

Atomicity

Non-atomic 64 bit store:

```
#include <stdint.h>

uint64_t var= 0;

void store()
{
  var= 0x100000002;
}
```

Now compile this for IA-64 with GCC:

```
$ gcc -O3 -m64 -S atomic.c && cat atomic.s
…
      movabsq    $4294967298, %rax
      movq  %rax, var(%rip)
      ret
…
```

Atomicity

Atomic 64 bit store:

```
#include <stdint.h>

uint64_t var= 0;

void store()
{
  __atomic_store_n(&var, 0x100000002, __ATOMIC_RELAXED);
}
```

Now compile this for IA-64 with GCC:

```
$ gcc -O3 -m64 -S atomic.c && cat atomic.s
…
      movabsq    $4294967298, %rax
      movq  %rax, var(%rip)
      ret
…
```

Atomicity

Non-atomic 64 bit store:

```
#include <stdint.h>

uint64_t var= 0;

void store()
{
  var= 0x100000002;
}
```

Now compile this for IA-32 with GCC:

```
$ gcc -O3 -m32 -S atomic.c && cat atomic.s
…
      movl  $2, var
      movl  $1, var+4
      ret
…
```

Atomicity

thread 1	thread 2
`var= 0x100000002;`	`var= 0x200000001;`

Concurrent thread loading var may observe:

0x000000000
0x000000001
0x000000002
0x100000000
0x100000001
0x100000002
0x200000000
0x200000001
0x200000002

Atomicity

Atomic 64 bit store:

```
#include <stdint.h>

uint64_t var= 0;

void store()
{
  __atomic_store_n(&var, 0x100000002, __ATOMIC_RELAXED);
}
```

Now compile this for IA-32 with GCC:

```
$ gcc -O3 -m32 -S atomic.c && cat atomic.s
…
      subl  $12, %esp
      .cfi_def_cfa_offset 16
      movl  $2, %eax
      movl  $1, %edx
      movl  %eax, (%esp)
      movl  %edx, 4(%esp)
      fildq (%esp)
      fistpq      var
      addl  $12, %esp
      .cfi_def_cfa_offset 4
      ret
```

Compiler optimizations

```
#include <stdint.h>

uint32_t stage= 0;

void thread1()
{
  stage= 1;
  while (stage != 2);
  stage= 3;
}
```

Compiler optimizations

```
#include <stdint.h>

uint32_t stage= 0;

void thread1()
{
  stage= 1;
  while (stage != 2);
  stage= 3;
}
```

Now compile this with GCC:

```
$ gcc -O3 -S atomic.c && cat atomic.s
…
      movl  $1, stage(%rip)
.L2:
      jmp   .L2
…
```

Compiler optimizations

```
#include <stdint.h>

uint32_t stage= 0;

void thread1()
{
  __atomic_store_n(&stage, 1, __ATOMIC_RELAXED);
  while (__atomic_load_n(&stage, __ATOMIC_RELAXED) != 2);
  __atomic_store_n(&stage, 3, __ATOMIC_RELAXED);
}
```

Compiler optimizations

```
#include <stdint.h>

uint32_t stage= 0;

void thread1()
{
  __atomic_store_n(&stage, 1, __ATOMIC_RELAXED);
  while (__atomic_load_n(&stage, __ATOMIC_RELAXED) != 2);
  __atomic_store_n(&stage, 3, __ATOMIC_RELAXED);
}
```

Now compile this with GCC:

```
$ gcc -O3 -S atomic.c && cat atomic.s
…
      movl  $1, stage(%rip)
.L2:
      movl  stage(%rip), %eax
      cmpl  $2, %eax
      jne   .L2
      movl  $3, stage(%rip)
      ret
…
```

Ordering

Atomic operations on any particular variable are not allowed to be reordered against each other independently of memory barrier.

```
my_atomic_store32_explicit(&a, 1, MY_MEMORY_ORDER_RELAXED);
d= my_atomic_load32_explicit(&a, MY_MEMORY_ORDER_RELAXED);
my_atomic_store32_explicit(&a, 2, MY_MEMORY_ORDER_RELAXED);
b= my_atomic_store32_explicit(&a, MY_MEMORY_ORDER_RELAXED);
```

Disadvantages

- atomic operations may drive you crazy
- absence of easy understandable manual
- atomic operations are more expensive than their non-atomic counterparts

References

http://preshing.com/20130618/atomic-vs-non-atomic-operations/

https://gcc.gnu.org/wiki/Atomic/GCCMM/Optimizations

volatile type qualifier

In C/C++ the volatile keyword was intended to:

- allow access to memory mapped devices
- allow uses of variables between setjmp and longjmp
- allow uses of sig_atomic_t variables in signal handlers.

volatile type qualifier

Useful side effects:

- compiler can’t optimize out volatile access
- compiler can’t reorder volatile access relative to some other special access (e.g. another volatile access).

volatile type qualifier

Disadvantages:

- operations on volatile variables are not atomic
- compiler may reorder volatile access relative to regular variable access
- CPU may reorder volatile access relative to any other access (e.g. another volatile access).

volatile type qualifier

NO

According to the C++11 ISO Standard, the volatile keyword is only meant for use for hardware access; do not use it for inter-thread communication.

References

https://en.wikipedia.org/wiki/Volatile_(computer_programming)

http://en.cppreference.com/w/c/language/volatile

https://msdn.microsoft.com/en-us/library/12a04hfd.aspx