Concurrent Programming

with the shared memory paradigm

Tazio Ceri

Cynny

July 20, 2016

Presentation plan

Introduction

What is concurrent programming?

- Contemporary execution of multiple instructions in an algorithm.
- No implicit or compile time ordering between instructions!

Two models

- Message Passing
- Shared Memory

Message Passing

Everybody uses this model as form of Inter Process Communication ...

... in a network or ...

- write/writev/send/sendto/sendmsg
- read/readv/recv/recvfrom/recvmsg

... inside a single host.

- fork/waitpid
- unix socket, signals, System V IPC

This is not what we want to talk about today!

Shared Memory

One process... many threads of execution!
Every thread can access the same memory address space.

- faster than message passing model,
- but also harder to get it right.
- Quite impossible to debug and test!

We need something better than a debugger
and some test to reliably build multi-threaded software! We'll see that later!

Atomic problem

```
std::vector<int> limits = { 0, 50000, 100000 };
int result = 0;
std::vector<int> v1(100000,1), v2(100000,2);
void* dot_product( void* x ){
    int* y = (int*) x;
    for(int i = limits[*y]; i < limits[1+*y]; ++i)
        result += v1[i] * v2[i];

    return NULL;
}
```

- We have *v1* full of 100000 elements, all 1, and *v2* with 100000 elements, all 2.
- Two threads - limits is a trick to slice the vectors and let the threads run.

Surprising results

107622
99808
171422
104284
156516
129352

- The correct result is 200000!
- Where is the mistake?

Surprising results

107622
99808
171422
104284
156516
129352

- The correct result is 200000!
- Where is the mistake?
- Operations on result are not atomic!
- Now the fix looks obvious!

Atomic solution: __sync_fetch_and_add

```
int result = 0;
void* dot_product( void* x ){
  int* y = (int*) x;
  for(int i = limits[*y]; i < limits[1+*y]; ++i){
    int tres = v1[i] * v2[i];
    __sync_fetch_and_add( &result, tres );
  }
  return NULL;
}
```

- Now we get the correct result, that is 200000!
- it makes impossible to interleave fetch and add between different threads, so we do not loose numbers around.
- Atomicity is a property of operations on a variable, *result* is still the same int, no *volatile*!

What does 'atomic' mean anyway?

1. An atomic operation must run completely before someone else, on another thread, can read the update;
2. on POD there is always a mapping to hardware instruction, that must execute operations in one cycle;
3. many atomic operations are a composition of elementary operations, but property 1 must hold;
4. typical atomic operations: compare and swap, read-write-modify.

What happens when data size is too large for an atomic operation?

The case against *volatile*

- *volatile* is totally wrong when used for concurrent programming;
- it basically only disables cache and some compiler optimizations;
- no protection against OOO operations.
- Don't use it to make a variable "atomic"!

volatile quotes (1)

David Butenhof

"...the use of volatile accomplishes nothing but to prevent the compiler from making useful and desirable optimizations, providing no help whatsoever in making code "thread safe". "

volatile quotes (2)

Linus Torvalds

If the memory barriers are right, then the "volatile" doesn't matter. And if the memory barriers aren't right, then "volatile" doesn't help. Using "volatile" in data structures is basically *always* a bug.

Producer - Consumer problem

- Single producer of data, let's say integers;
- single consumer of data;
- typical data structure: a circular buffer that can hold N integers.

```
class Buffer
{
    int buf[N];
    int first{0};
    int last{0};
public:
    void push( int elem );
    int pop();
};
```

Producer - Consumer problem code with single thread

```
void Buffer::push( int elem )
{
    if ( last != first )
    {
        buf[last] = elem;
        last = ( last + 1 ) % N;
    }
    else throw buffer_full_error();
}
```

What happens when multiple threads call push at the same time?

Producer - Consumer problem code with single thread

```
int Buffer::pop()
{
    if ( last != first )
    {
        int r = buf[first];
        first = ( first + 1) % N;
    }
    else throw buffer_empty_error();
}
```

What happens when multiple threads call pull at the same time?

Producer - Consumer problem

- In both cases we need atomic access to the whole structure;
- but hardware can't help us with arbitrarily sized data.
- We should need transactional memory, but that's not available.
- We need the simplest tool of synchronization: a *mutex*.

Producer - Consumer problem

- Many producers of data;
- many consumers of data;
- the same data structure, but with synchronization.

```
class Buffer
{
    int buf[N];
    int first{0};
    int last{0};
    pthead_mutex_t m{PTHREAD_MUTEX_INITIALIZER};
public:
    void push( int elem );
    int pop();
};
```

Producer - Consumer problem, multiple writers

```
void Buffer::push( int elem )
{
    pthread_mutex_lock( &m );
    if ( last != first )
    {
        buf[last] = elem;
        last = ( last + 1 ) % N;
    }
    else
        unlock_and_throw( &m,  buffer_full_error() );
    pthread_mutex_unlock( &m );
}
```

unlock_and_throw is an obvious utility function.

Producer - Consumer problem, multiple readers

```
int Buffer::pop()
{
    pthread_mutex_lock( &m );
    if ( last != first )
    {
         int r = buf[first];
        first = ( first + 1) % N;
    }
    else
        unlock_and_throw( &m,  buffer_empty_error() );
    pthread_mutex_lock( &m );
}
```

Would it be better if we could get rid of the exception?

Producer - Consumer problem, empty/full conditions

Throwing exceptions when we don't have data, or we have no space left, does not look the best option:

- these are common cases, not real error cases;
- a thread cannot know in advance whether the operation will fail without an API that exposes internal rapresentation, ie: lock(), unlock();
- returning a boolean pollutes user's code with *if*.

Ideally we would avoid the problem altogether.

- We would ensure a kind of ordering between producers and consumers;
- if $0 \leq nP - nC$ then consumers should stop when popping;
- if $nP - nC \geq N$ then producers should stop when pushing.

Producer - Consumer problem, condition variables

We should use condition variables to implement this kind of blocking.

```
class Buffer
{
    int buf[N];
    int first{0};
    int last{0};
    pthead_mutex_t m{PTHREAD_MUTEX_INITIALIZER};
    pthread_cond_t c{PTHREAD_COND_INITIALIZER};
public:
    void push( int elem );
    int pop();
};
```

Producer - Consumer problem, condition variables

```
void Buffer::push( int elem )
{
    pthread_mutex_lock( &m );
    while ( last == first )
        pthread_cond_wait( &c, &m );
    buf[last] = elem;
    last = ( last + 1 ) % N;
    pthread_cond_broadcast( &c );
    pthread_mutex_unlock( &m );
}
```

Producer - Consumer problem, condition variables

```
int Buffer::pop()
{
    pthread_mutex_lock( &m );
    while ( last == first )
        pthread_cond_wait( &c, &m );

    int r = buf[first];
    first = ( first + 1) % N;
    // not pthread_cond_signal!
    pthread_cond_broadcast( &c );
    pthread_mutex_lock( &m );
}
```

Mutexes and condition variables

- A condition variable needs always a mutex, because the wait and the condition are not checked atomically;
- condition_variables are allowed to unblock randomly, so *while* is mandatory;
- they implement Hoare's monitor;
- pthread_cond_broadcast awakes all waiting threads;
- pthread_cond_signal awakes only one thread;
- using signal in this case is wrong because it could also awake a producer, and that would led to a deadlock!

Deadlock

- A deadlock happens when all execution stream block each other, and none can make progress and call an *unlock*;
- a thread could also deadlock itself, but calling a *lock* recursively, Posix Thread have an option to allow that with no problem;
- there are deadlock avoidance algorithms to avoid deadlocking like Banker's algorithm, but they have usually exponential complexity so they are not implemented;
- remember, testing can not prove that your code is deadlock free;
- so there are many formal methods that have developed, for example $\pi - calculus$.

Design by contract

- A general methodology to avoid bugs in OOP software;
- built inside the language *Eiffel*, by Bertrand Meyer;
- and also in Perl6;
- *preconditions*, *postconditions* and *consistency rules* become defined and checked with assertions;
- it's possible to use them as a basis to *show correctness mathematically*.

pthread_cond_broadcast and a *thundering herd*

- pthread_cond_broadcast is easier and can always be used;
- it awakes all waiting thread at the same time;
- but only one will take the lock and will find the condition so it can perform work;
- if the situation happens too often, we could have a performance problem;
- where all threads awake only to hit the lock and waste context switches;
- its name is *thundering herd*.

Avoiding *thundering herd* problem

Let's reimplement the data structure to avoid the broadcast.

```
class Buffer
{
    int buf[N];
    int first{0};
    int last{0};
    pthead_mutex_t m{PTHREAD_MUTEX_INITIALIZER};
    pthread_cond_t readers{PTHREAD_COND_INITIALIZER};
    pthread_cond_t writers{PTHREAD_COND_INITIALIZER};
public:
    void push( int elem );
    int pop();
};
```

Avoiding *thundering herd* problem

```
void Buffer::push( int elem )
{
    pthread_mutex_lock( &m );
    while ( last == first )
        pthread_cond_wait( &readers, &m );
    buf[last] = elem;
    last = ( last + 1 ) % N;
    pthread_cond_signal( &readers );
    pthread_mutex_unlock( &m );
}
```

Avoiding *thundering herd* problem

```
int Buffer::pop()
{
    pthread_mutex_lock( &m );
    while ( last == first )
        pthread_cond_wait( &writers, &m );

    int r = buf[first];
    first = ( first + 1) % N;
    pthread_cond_signal( &writers );
    pthread_mutex_lock( &m );
}
```

Homework: showing correctness

- What are the invariants?
- How can you be sure that many readers will not be wrongly blocked, if wake only one?
- The situation is the same for writers.
- When a reader is awakened, how many meaningful data are there?

Beware of your memory model!

Here it is the Double-Checked Locking pattern.

```
class Singleton { ... ; static Singleton* instance; ... };
Singleton& Singleton::getInstance()
{
    if ( ! instance ) {
        pthread_mutex_lock( &m );
        if ( ! instance ) {
            Singleton* t = new Singleton;
            instance = t;
        }
        pthread_mutex_unlock( &m );
    }
    return *instance;
}
```

Double Checked Locking Pattern explained

- All looks beautiful, except it does not work in C99 or C++98;
- because there are no fence, and instance is not atomic;
- fixed in C++11 and C11.

```
//C++11 API
#include <atomic>
std::atomic_thread_fence( std::memory_order order );

/* C11 API */
#include <stdatomic.h>
atomic_thread_fence(memory_order order);
```

Atomic fences

- These fences are supposed to be very low level, faster than atomic types in some cases;
- a *memory_order_acquire* prevents the memory reordering of any read which precedes it in program order with any read or write which follows it in program order;
- a *memory_order_release* prevents the memory reordering of any read or write which precedes it in program order with any write which follows it in program order;
- they can establish a *synchronizes-with* relationship, which means that they prohibit memory reordering in a way that allows you to pass information reliably between threads;
- there are more *memory_order* in both C++11 and C11 standards.

Double Checked Locking Pattern fixed

```
#include <stdatomic.h>
Atomic_ S* S::instance;
pthread_mutex_t S::m_mutex{PTREAD_MUTEX_INITIALIZER};

S* S::getInstance() {
    S* tmp = atomic_load_explicit(&instance, memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
    if (tmp == nullptr) {
       pthread_mutex_lock( &m_mutex );
       tmp = atomic_load_explicit(&instance, memory_order_relaxed);
       if (tmp == nullptr) {
          tmp = new S;
          atomic_thread_fence(memory_order_release);
          atomic_store_explicit(&instance, &tmp, memory_order_relaxed);
       }
       pthread_mutex_unlock( &m_mutex );
    }
    return tmp; // And thanks to Preshing!
}
```

Readers and writers problem

- many readers;
- readers can all read at the same time in parallel;
- writers need exclusive access;
- writes are normally rarer than reads;
- a write can safely erase the older information;
- none should risk *starvation*.

Starvation: indefinitely long wait of a thread

Readers and writers first implementation

```
struct Data { ... };
class Buffer
{
    Data data;
    pthread_rwlock_t rwlock{PTHREAD_RWLOCK_INITIALIZER};
public:
    Data read();
    void write(Data&);
    Buffer();
};
```

Readers and writers first implementation methods

```
Data Buffer::read()
{
    pthread_rwlock_rdlock( & rwlock );
    Data tmp = data;
    pthread_rwlock_unlock( & rwlock );
    return tmp;
}

void Buffer::write(Data& x)
{
    pthread_rwlock_wrlock( & rwlock );
    data = x;
    pthread_rwlock_unlock( & rwlock );
}
```

- As said, all readers can access data at the same time;
- if there are readers that hold rwlock, writers will starve;
- that's the default.

Readers and writers first implementation constructor

```
Buffer::Buffer()
{
   pthread_rwlockattr_setkind_np( &rwlock,
     PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP );
}
```

- Now readers, not writers, will starve;
- *PTHREAD_RWLOCK_PREFER_WRITER_NP* exists but has a bug: writers will keep starving;
- however, if writes are rare, this is not supposedly a problem;
- NP means Not Portable. Use them on Linux.
- Is there a way to not anyone starve?

Readers and writers without starvation

```
class Buffer
{
Data data;
    pthead_mutex_t m{PTHREAD_MUTEX_INITIALIZER};
    int readers{0};
    pthread_cond_t rc{PTHREAD_COND_INITIALIZER};
    int writers{0};
    pthread_cond_t wc{PTHREAD_COND_INITIALIZER};
public:
    Data read();
    void write(Data&);
}
```

Readers and writers without starvation - read

```
Data Buffer::read()
{
    pthread_mutex_lock( &m );
    ++readers;
    while ( writers > 0 )
        pthread_cond_wait( &rc, &m );
    pthread_mutex_unlock( &m );
    Data tmp = data;

    pthread_mutex_lock( &m );
    --readers;
    pthread_cond_broadcast( &wc );
    pthread_mutex_unlock( &m );
    return tmp;
}
```

Readers and writers without starvation - write

```
void Buffer::write(Data& d)
{
    pthread_mutex_lock( &m );
    ++writers;
    while ( readers > 0 )
       pthread_cond_wait( &wc, &m );
    data = d;
    --writers;
    pthread_cond_broadcast( &rc );
    pthread_mutex_unlock( &m );
}
```

Definitions (1)

- *Spinlock*: It works like a *Mutex*, but it does not let the thread go into "sleeping state", it actively checks until the spinlock is not any more locked. Useful when a context switch costs more than the whole critical region.
- *Race Condition*: consequence of concurrent read/write access to shared data without proper synchronization.
- *Critical Region*: part of the program that requires synchronization. Rember that synchronization primitives that are around the critical region are to protect data by concurrent r/w access, not code.

Definitions (2)

- *Semaphore* A generalized *Mutex*, initialized with an integer N, it lets N thread access to the Critical Region.
- *Starvation*: problem where a process is perpetually denied necessary resources to process its work.
- *False Sharing*: performance problem that happens when two thread access to indipendently to different data in the same cache line; and every write invalidates the other's thread cache.

Mad, but legal, sharing in C

```
struct btrfs_block_rsv {
    u64 size;
    u64 reserved;
    struct btrfs_space_info *space_info;
    spinlock_t lock;
    unsigned int full:1;
};
```

spinlock_t in kernel space, just near a bitfield! On Itanium architecture, gcc *legally* updates *full* and *lock* at the same time! It's legal with the C memory model. BTRFS developers have found out painfully why their *lock* was currupted.

Famous concurrency problems(1)

Philosophers' problem

N philosophers are eating Spaghetti. A rules states that each philosopher must take the fork at his left and the one at his right to eat. The order in which that happens is unspecified. Teach the philosophers how to do it avoiding deadlocks, starvation, and letting them eat as much as possible in parallel!

Famous concurrency problems(2)

Smokers' problem(1)

Three smokers and an agent. Three ingredients for a cigarette: tobacco, paper and matches. Each smoker has an infinite amount of an ingredient, but he waits for the agent to give him the other two. The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed. For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent. To explain the premise, the agent represents an operating system that allocates resources, and the smokers represent applications that need resources.

Famous concurrency problems(3)

Smokers' Problem(2)

The problem is to make sure that if resources are available that would allow one more applications to proceed, those applications should be woken up. Conversely, we want to avoid waking an application if it cannot proceed.

There are many other interesting problems to solve!

I want to implement my synchronization primitives!

```
#include <linux/futex.h>
#include <sys/time.h>
int futex(int *uaddr, int futex_op, int val,
  const struct timespec *timeout, /* or: uint32_t val2 */
  int *uaddr2, int val3);
```

- There are faster ways to suicide, however...
- on Linux, use the *futex* system call;
- but remember: the original author wrote examples about how to use it;
- and they had concurrency issues!
- You have been warned!

Pointers to knowledge

- The C11 Standard
- The C++11 Standard
- Programming Posix Threads by David Butenhof
- C++ and the Perils of Double-Checked Locking
- The little book of semaphores
- Preshing on Programming
- Unix Network Programming Vol.2 IPC by Richard Stevens
- The Art of Multiprocessor Programming
- C++ Concurrency in Action: Practical Multithreading
- Futexes are tricky
- Object-Oriented Software Construction