Títol: Implementation of a Lock-Free Queue

Volum: Volum 1
Alumne: Nicolas MOTTE

Director/Ponent: Josep FERNANDEZ
Departament: Enginyeria de Sistemes, Automàtica i Informàtica Industrial
Data: 13 Abril 2010
**DADES DEL PROJECTE**

Títol del Projecte: Implementation of a Lock-Free Queue  
Nom de l'estudiant: Nicolas MOTTE  
Titulació: Doble titulación  
Crèdits: 37,5  
Director/Ponent: Thomas PARE / Josep FERNANDEZ  
Departament: Enginyeria de Sistemes, Automàtica i Informàtica Industrial

**MEMBRES DEL TRIBUNAL (nom i signatura)**  
President: Antoni Grau Saldes  
Vocal: Alejandro Ramirez Bellido  
Secretari: Josep Fernandez Ruzafa

**QUALIFICACIÓ**

Qualificació numèrica:  
Qualificació descriptiva:  
Data:
8.4 Distributed Database ................................................................. 40
  8.4.1 Berkeley DB ...................................................................... 40
  8.4.2 Tokyo Cabinet .................................................................. 40
  8.4.3 Crescando ......................................................................... 41
  8.4.4 Prototype ......................................................................... 41

9. Conclusion .................................................................................. 42

10. References .................................................................................. 42

11. Bibliography ............................................................................... 43

12. Annexes ...................................................................................... 43
1. Company Presentation

In this part, I'll briefly present Amadeus and the context in which I am completing my 3rd year internship.

1.1 Amadeus

Amadeus is the world leader in computer science applied to distribution and selling of products related to travel industry. Amadeus data base allows users to book flights on the Internet but although offers other services such as hotel or rental car. More than 400 millions reservations are yearly treated by Amadeus.

Amadeus clients are airlines, hotels, tour operators, travel agencies and transport companies.

Amadeus was founded in 1987 by Air France, Lufthansa, Iberia and SAS. The company is implanted in several countries:
- Madrid, Register Office
- Sophia-Antipolis, Development and Marketing
- Erding, Data Center
- many other places as London, Sydney, Miami, Buenos Aires, Bangkok ...

1.2 Team Presentation

Initially, Core Middleware and Development Support (CMD) was a team of about 10 persons dealing with the reservation: communication between airlines and travel agencies. But in a decade, other functionalities have been integrated and the team has grown up to become a department and now a division of 3 departments:
- AMD, Application Middleware Department,
- DSP, Development Support,
- CSM, Communications and Security Middleware.

The team I integrated is Middleware Component (COM) part of CSM department. This team delivers components used by Amadeus applications running on Linux/OTF (Open Transaction Framework) platforms.

There are about 10 persons sharing the 52 components. In theory, there is a main support and 2 people who have the acknowledgment.

The work is shared between the support and the evolution of the existing components and the development of new one or bigger evolutions.
2. Introduction

To complete my engineer formation, a 5-month internship is planned. I'm doing mine in Middleware Component team, at Amadeus on Sophia-Antipolis site during 6 months.

The transactional framework used to develop Amadeus C++ applications is based on a mechanism of queues to manage the message exchanges between components. These structures are protected from concurrent access thanks to synchronization services provided by Linux. But these services have a cost in term of performance and they bound the volume of messages transmitted by these queues.

2.1 Goal of the PFC

In a first step, I have to investigate on the state-of-the-art in term of management algorithms of synchronization-free queue (Lock-Free Queue). Several algorithms will be studied for different cases: 1 Producer – 1 Consumer, 1 Producer – N Consumer, N Producer – 1 Consumer, N Producer – N Consumer.

In a second one I will participate to the implementation of these algorithms and to the validation of the performance improvements obtained regarding current solutions deployed in Amadeus.

2.2 Additional Goals

I also had the chance to work on Inter-Process Communication (IPC) queues. A component (the LogServer) creates threads (plugins) to treat log messages. If one of them happens to core, the LogServer and all other plugins will core. Thus we want to be able to create plugins in a new process. With this solution, plugins and LogServer won’t share memory, and if a plugin has a problem, it won’t affect the LogServer. As the new plugin process and the LogServer exchange messages. I had to serialize data and send it through an IPC system from a process to another.

In a first part, I tried several solutions for IPC and data serialization. Then I have implemented the best solution, using an IPC system created by another person from Amadeus, and serialization created by Google.

Finally I worked on the indexing of a huge distributed database (several TB). I had to find the best solution to minimize disk space used by the database and answer to queries in an appropriate time.
3. Acknowledgments

I want to thank the whole team for the welcome they gave to me. During my internship, I felt like a real member of the team.

I also want to thank Thomas PARE, my team leader and training mentor, who proposed me this internship, making it more interesting at each step, always thinking about what I can learn.

I want to particularly thank Matthieu PASCAL and Bruno MARTINEZ, in charge of the LogServer and the Toolbox for their patience, kindness, encouragements and trust. They were always here when I needed help, support or explanations.

I also thank Herb SUTTER and Michael L. SCOTT who took some of their precious time to answer to all my emails when I needed help.

Personally, it was a great experience. During 6 months, I have felt completely independent and integrated in the team. I have played twice a month in the Amadeus championship team. I trained with my development team twice a week and on weekends we were hanging out together. I enjoyed the place so much: sea, skiing, sun, hiking, cycling, everything was here.

Finally, I warmly thank Marianne EMILE, Florent EXBRAYAT, Jeremy BARLET and Thomas PARE from COM team for sharing their office and giving me some advice.
4. Specifications

As I said, I work on the LogServer. But all the core components of Amadeus C++ are using these memory queues. In addition to LogServer, my optimization will be used by the Enterprise Service Bus (the SI) and the Transactional Framework (OTF). This is critical piece of software. The LogServer receives logs from different airlines companies. The LogServer uses several “1 Producer – 1 Consumer” queues. The implementation uses monitors (mutex + condition variables) to handle synchronization. If a queue is full, logs are discarded. At first, the logs were not critical, so it was not a problem to lose some of them. But now, airlines also use these logs for billing. Such as, losing logs means losing money!

The aim of my internship was to improve the performance of the LogServer component, by implementing Lock-Free Queues. These does not uses mutexes. Consumer and Producer can consume and produce logs at the same time, without locking shared data. To do so, we need to use atomic variables.

Atomic variables will be available in the next C++ standard (C++0x). Meanwhile, I will have to implement my own functions to read/write shared variables atomically. Then I will need to validate the improvement of performance by running some tests and finally the project will be loaded in DEV (it’s a test environment owned by Amadeus on which we can inject the traffic we want). It is one of the 13 test phases. In this environment my software will be connected with other applications.

This is the end of my official internship subject. But I had the chance to work on another subject. Here is how a LogServer works.

In a LogServer, there are plugins running in different threads. Each plugin has a work to do on received logs. For example it can write the content of a log in a local file, print it on the screen, compute statistics, index logs in a database, etc... But if a plugin cores or leak, the LogServer and the other threads are affected. That’s why we need to allow the user to choose between executing a plugin in a new thread or in a new process. Thereby, if a plugin executed in a new process cores, it doesn’t affect the LogServer and other plugins, because they don’t share the same memory space. But as they don’t share memory, we need an IPC (Inter-Process Communication) between two processes if they need to communicate. I tried different solutions (local file, pipe ...) and finally I used a shared memory queue, created by Lenaic HUARD, member of the OTF Team. Moreover, if a complex object (a class with an attribute of pointer type) is sent from a process to another one, we have to serialize it before, otherwise, the process will receive an object with pointers pointing to nothing in memory (or meaningless data). I used Protocol Buffer (a Google project) to handle the serialization.

Finally I had to check the behavior when either a plugin either the LogServer cores. In the first case, the LogServer must stop sending messages to the plugin. In the second case, the plugin must be killed. Of course in both cases shared memory should be cleaned.

Then a third subject arrived, even more interesting than the two others. The challenge was to allow a search through one week of log using an index on a
distributed database. Using Oracle or MySQL was not a solution. Indeed, in a production environment we can reach up to 270kTPS at peak time. For 8 days we have 100TB of data. We needed a database fast enough to handle that kind of traffic. Thus we took the time to discover two efficient key/value databases: Berkeley DB and Tokyo Cabinet. These in-memory databases are the only realistic alternatives to handle this throughput.
In the chapters 5, 6 and 7 of this report, I provide an overview of what I found in the documentation from the Amadeus library. Then in the 9th chapter, I will present you my own work and the results.

5. Version 1 of a Lock-Free Queue – Petru Marginean

Petru Marginean is an Expert C++ developer. He wrote several articles for Dr Dobb’s and C/C++ Users Journal (http://www.petrum.net/). In Dr Dobb’s, he wrote an article on lock-free queues. In this part, we will see the hypothesis under which his implementation is available, the structure of its queue, the implementation of the function "Consume" and "Produce", and finally we will compare several ways to poll a queue.

5.1 Requirements – Hypothesis

- We assume we have only one Producer (or Writer) and one Consumer (or Reader) accessing the queue at the same time.
- When inserting/erasing to/from a std::list<T>, the iterators for the existing element must remain valid (atomicity)
- Only one thread modifies the queue, the producer thread both adds/erases elements in the queue
- Beside the std::list<T> used as the container, the lock-free queue class holds two iterators pointing to the not-yet consumed range of elements, each is modified by one thread and read by the other. They are called _head and _tail. The Producer modifies _tail and the Consumer modifies _head.
- Reading/Writing list<T>::iterator is atomic on the machine upon which you run the application.

5.2 Data Structure of the Lock-Free queue

```cpp
template <typename T>
struct LockFreeQueue {
    LockFreeQueue();
    void Produce(const T& t);
    bool Consume(T& t);
private:
    typedef std::list<T> TList;
    TList list;
    typename TList::iterator _head, _tail;
};
```

The most important to understand is that the Producer is the only one that changes the list. The _tail iterator changes only after a new element is added. That’s why when the Consumer tries to read the new added element, it is ready to be used.
The std::list<T> maintains an element that is considered already read. As we want an algorithm that works even with an empty queue, the constructor inserts an empty T() element in the queue:

```cpp
LockFreeQueue::LockFreeQueue() {
    list.push_back(T());
    _head = list.begin();
    _tail = list.end();
}
```

### 5.3 Producer

```cpp
void LockFreeQueue::Produce(const T& t) {
    list.push_back(t);
    _tail = list.end();
    list.erase(list.begin(), _head);
}
```

It handles the modification of the _tail iterator and the cleaning of the queue.

### 5.4 Consumer

The Consumer only reads the elements, it does not erase them, and it accesses the list through iterators.

```cpp
bool LockFreeQueue::Consume(T& t) {
    typename TList::iterator _next = _head;
    _next++;
    if (_next != _tail) {
        _head = _next;
        t = *_head;
        return true;
    }
    return false;
}
```

But if the queue is empty, we need to wait. To do this, we have several solutions.

#### 5.4.1 Naïve Polling

```cpp
T LockFreeQueue::Consume() {
    T tmp;
    while (!Consume(tmp)) ;
    return tmp;
}
```

Advantage: Good performance if the queue is not empty.
Inconvenient: If the queue is empty, explosion of the CPU. And if the queue is almost never empty, it demonstrates that the Consumer is unable to keep pace with Producer.

5.4.2 Sleep

T LockFreeQueue::Consume(int wait_time=1/*milliseconds*/) {  
    T tmp;  
    while (!Consume(tmp)) {  
        Sleep(wait_time/*milliseconds*/);  
    }  
    return tmp;  
}

Sleep() can be the boost::thread::sleep() function.

5.4.3 Time Wait

template <typename T>  
struct WaitFreeQueue {  
    void Produce(const T& t) {  
        _queue.Produce(t);  
        _cond.notify_one();  
    }  
    bool Consume(T& t) {  
        return _queue.Consume(t);  
    }  
    T Consume(int wait_time = 1 /*milliseconds*/) {  
        T tmp;  
        // Queue not empty  
        if (Consume(tmp))  
            return tmp;  
        // Queue empty  
        boost::mutex::scoped_lock lock(_mtx);  
        while (!Consume(tmp)) { //A  
            boost::xtime t;  
            boost::xtime_get(&t, boost::TIME_UTC);  
            AddMilliseconds(t, wait_time);  
            cond.timed_wait(lock, t); //B  
        }  
        return tmp;  
    }  
private:  
    LockFreeQueue<T> _queue;  
    boost::condition _cond;  
    boost::mutex _mtx;  
};
Time_wait() is used instead of wait() to avoid a possible deadlock (if Produce() is called between A and B). As we explained before, the Consumer consumes data as fast possible, even when the Producer is writing. But in that case, the Consumer waits when the queue is empty.

You can find all the details of the boost functions in *Annexe 4 – Boost details.doc*.

### 5.5 Results

To compare the three approaches (NAIVE_POLLING, SLEEP, and TIME_WAIT), Petru implemented a test called "Ping-Pong" that is similar to the game of table tennis. There are two identical queues between two threads. You first load one of the queues with a number of "balls," then ask each thread to read from one queue and write to the other. The result is a controlled infinite loop. By limiting the game to a fixed number of reads/writes ("shots"), you get an understanding of how the queue. The faster, the better. We should also check CPU usage to see how much of it is used for real work.

![Execution Time Comparison](image1)

I didn't display the SLEEP solution in this graph because the performance are really too bad and it doesn't fit this scale. The NO_WAIT strategy is fast but behaves worst when there is no ball regarding CPU usage:

![CPU Usage Comparison](image2)
The best combination is the TIME_WAIT with a wait around 1ms. It has a very fast response time and a CPU usage around 0% when the queue is empty. The worst solution is the SLEEP, even with a time sleep of 0.

5.6. Version 1 – Problems

5.6.1. Limitations

The fundamental reason that the code is broken is that it has race conditions on both would-be lock-free variables: _head and _tail. To avoid a race, a lock-free variable must have two key properties that we need to watch for and guarantee: atomicity and ordering. These variables are neither.

5.6.2 Atomicity

Reads and writes of a lock-free variable must be atomic. That’s why lock-free variables can’t be larger than the machine’s native word size. They can be pointers in C++, object reference in Java or .NET or integers.

The use of a list<T>::iterator doesn’t always meet this requirement so it can’t be read/write with atomic loads and stores. Moreover, even if they were with appropriate size, you would need to add other decorations to the variable to ensure for example that the variable is aligned in memory.

Illustration of the problem:

```cpp
void LockFreeQueue::Produce(const T& t) {
    ...
    _tail = list.end();
    list.erase(list.begin(), _head);
}

bool LockFreeQueue::Consume(T& t) {
    ...
    if (_next != _tail) {
        _head = _next;
        ...
    }
}
```

If the reads/writes of _head and _tail are not atomic, the Producer could read a partially updated _head by the Consumer and try to dereference it, or the Consumer could read a partially updated _tail and fall off the end of the queue.

5.6.3 Ordering Problems

**Producer**

Reads and writes must occur in an expected order, and even if it’s almost always the case, compilers, processors and caches like to optimize reads and writes and can reorder, invent or remove them. We need to prevent it. There are different solutions to
handle this, using mutex and ordered atomic variables (std::atomic in C++, volatile in Java/.NET), or doing it explicitly using API calls or memory fences/barriers (mb() on Linux). If we want a lock-free code, we need to use one of these tools.

Example:
```cpp
list.push_back(t);
_tail = list.end();
```

If these two lines are reordered, and if some of the writes are delayed after the assignation of _tail, the Consumer would think that a new object is available (because _tail has been updated), and it could access a partly assigned T object.

Solution 1: add a memory barrier (mb) between the two lines. Problem: list.end() is library function, so we don’t really know if it performs writes or not. As a solution, we can store its result in a temporary variable, unshared, and then assign it to _tail:

```cpp
list.push_back(t);
tmp = list.end();
mb();
_tail = tmp;
```

However, the assignation of _tail is still not atomic, and in fact, compilers and processors can also create writes to _tail that break the code, in case of branch prediction. More on that in the next section.

**Consumer**

Still with in the same idea:

```cpp
if (_next!= _tail) {
    _head = _next;
    t = *_head;
}
```

As seen before, the two lines can be reordered by the compiler.

First case (lucky case): the _head is updated to _next, and before that t be assigned, the Producer writes a data and erase everything between list.begin() and _head.

=> No problem because there is always an empty data between the list to erase and the list to consume.

Second Case: The Consumer thread performs two consecutive calls.

=> The Producer could erase a data which is still used by a Consumer. And as the compiler reorders the lines, exchanging them is not a solution.
Another problem is that compilers and processors can invent writes. So that kind of thing can happen. In case of a predictive compiler (_next != _tail), we could have:

```c
__temp = _head;
_head = _next;
if (_next == _tail) {
    _head = __temp;   // Wrong prediction
} else {
    t = *__head;
}
```

We could solve a lot of problems by inverting the two lines and inserting a full fence:

```c
if (_next!= _tail) {
    t = *__head;
    mb();
    _head = _next;
}
```

But it won’t solve the problem of invented writes.


Herb Sutter is a prominent C++ expert. He is also a book author and a columnist for Dr. Dobb’s Journal. Through 6 articles in Dr Dobb’s (see References) he described how to implement a lock-free queue getting around problems seen in the last section. In this part we will see the fundamental notions in this field, and then the implementation he proposed for the “1-1” and the “N-N” problem, taking a look at the data structure and the “Consume” and “Produce” functions.

6.1 Fundamentals

Atomicity: Each individual read and write is guaranteed to be atomic with respect to all other reads and writes of that variable. The variable typically fit into the machine’s native word size, and so are usually pointers in C++, object references in Java and .NET or integers.

Order: Each read and write is guaranteed to be executed in source code order. Compilers, CPU and caches will respect it and not try to optimize these operations the way they routinely distort reads and writes of ordinary variables.

Compare-and-Swap: There is a special operation you can call using syntax like variable.compare_exchange(expectedValue, newValue) that does the following as an atomic operation: If variable currently has the value expected value, it sets the value to newValue and returns true, else returns false. A common use is
if(variable.compare_exchange(x,y)), which you should get in the habit of reading as, “if I’m the one who gets to change variable from x to y”.

Atomic variable can be:
volatile in C++ (volatile int)
volatile or *Atomic* in Java (volatile int or AtomicInteger)
atomic<T> in C++0x (atomic<int>)

6.2 Solution for a 1 Producer/1 Consumer Lock-Free queue

6.2.1 Data Structure

template<typename T>
class LockFreeQueue {
private :
    struct Node {
        Node ( T val ) : _value(val), _next(nullptr) { }
        T _value;
        Node* _next;
    };
    Node* _first;
    atomic<Node*> _head, _tail; // Atomic Variable
public :
    LockFreeQueue() {
        _first = _head = _tail = new Node( T() );
    }
    ~LockFreeQueue() {
        while ( _first!= nullptr ) {
            Node* tmp = _first;
            _first = tmp->_next;
            delete tmp;
        }
    }

6.2.2 Producer

void LockFreeQueue::Produce (const T* t) {
    _tail->_next = new Node(t);
    _tail = _tail->_next;
    while ( _first != _head ) {
        Node* tmp = _first;
        _first = tmp->_next;
        delete tmp;
    }
}

The code is ordered and the modifications atomic. The atomicity is guaranteed by the data structure. The only two shared variables are _head and _tail. They are just pointers of 64 bits. In the up-coming C++ standard (C++1x), the assignation of an
atomic variable is internally handled, the code would remain exactly the same. But for my internship I had to write my own assembly functions to respect atomicity:

```c
static inline void _AtomicAssign( volatile unsigned long *obj1, volatile unsigned long *obj2 ) {
    __asm__ __volatile__ (
        "movq %[obj2], %%rax
        "lock; xchg %%rax,%[obj1]\n"
        :"=m" (*obj1)
        :
        "a" (*obj2)
    );
}
```

The "lock" instruction reserves the data bus for the executing thread. No other thread will use it.

Concerning the ordering problem the compiler options and the "volatile" variable are sufficient. Optimizations are given by the -o flags. In our case, we can use -01 and -02 flags.

6.2.3 Consumer

```c
bool LockFreeQueue::Consume(T& t) {
    if (_head != _tail) {
        t = _head->_next->_value;
        _head = _head->_next;
        return true;
    }
    return false
}
```

Note: First we check atomically _head and _tail. Even if _tail is modified by the Producer, if the check is true once, it will stay true.

Important: _tail could be a non-shared variable. The check

```
_head != _tail
```

would become

```
_head->_next != null
```

But in that case, every "next" become a shared variable and we need to change its type from Node* to atomic<Node*>
7. Generalization (Two-Lock N Producer/ N Consumer queue)

We have seen a solution for the "1-1" problem. Herb Sutter goes further and tries to implement a solution suited for the "N-N" problem. He thinks that a fully lock-free implementation is impossible and provides a 2-lock implementation. First we will see some improvements in the implementation and then the data structure and the usual functions.

7.1 Four techniques

7.1.1 Two Locks

We will use two locks, one for the head of the queue, to regulate concurrent consumers, and one for the tail of the queue, to regulate concurrent producers. Also we will use ordered atomic variables, as seen before instead of prefabricated mutexes. It’s not anymore a complete lock-free algorithm, but if we do as much work as possible outside the critical area (when _head and _tail are updated), it’s still concurrent.

7.1.2 Allocation of the “T” objects on the heap

The T object will be allocated on the heap and kept in the queue by pointer.
Inconvenient: heap allocations are bad for scalability
Advantages: keeping the object by pointer gives us greater concurrency and scalability among the consumer threads because we can copy the T value out of the critical section.
As the advantages apparently outweigh the inconvenient, we will keep this solution.

7.1.3 Consumer deletes the data he consumed

Until now, the Producer is responsible for cleaning the queue. Sutter thinks it’s bad for performance because it adds contention on the queue’s head and delays the cleaning. From now on, we will let each consumer delete the data consumed, it gives better locality.

7.1.4 Padding

Finally we want to respect this: if two variables are not protected by the same mutex, and if these two variables can be used by two different threads, we need to keep them in two separated cache lines. It avoids false sharing and ping-ponging which limits scalability. That’s why we will add padding: the nodes, the _head and _tail pointers, and the two locks will be stored in different cache lines.
7.2 Data Structure

The queue data structure is a singly linked list. An empty queue contains an empty node, so the first real element is in the node after the first one.

```cpp
template<typename T>
struct LowLockQueue {
private:
    struct Node {
        Node (T* val): _value(val), _next(nullptr) { }
        T* _value; // Pointer stored instead of object
        Atomic<Node*> _next; // Next pointer is now shared
        // Padding added
        char pad[CACHE_LINE_SIZE - sizeof(T*) - sizeof(atomic<Node*>)];
    };

    char pad0[CACHE_LINE_SIZE];

    Node* _head; // _head is not shared anymore
    char pad1[CACHE_LINE_SIZE - sizeof(Node*)];

    Atomic<bool> _consumerLock; // Shared between consumers
    char pad2[CACHE_LINE_SIZE - sizeof(atomic<bool>)];

    Node* _tail; // _tail is not shared anymore
    char pad3[CACHE_LINE_SIZE - sizeof(Node*)];

    Atomic<bool> _producerLock; // Shared between producers
    char pad4[CACHE_LINE_SIZE - sizeof(atomic<bool>)];
};
```

1. The "_next" pointer is shared so it’s declared with an ordered atomic type. The padding is used to ensure that two different nodes don’t occupy the same cache line. In particular, in a non empty queue, if we had the first and the last nodes in the same cache line, it would cause invisible contention between producer and consumer.

   Note: the nodes are allocated on the heap, and heap allocator may add its own overhead (for alignment and housekeeping information). If it happens, and if we know how much it will be, we should modify our internal padding in order to have 1 Node = 1 cache line.

2. Thanks to padding, the data used by different threads are physically separated in memory and cache. It avoids false sharing.

   We solve several problems here. Obviously, _head and _tail are stored on different cache lines. But we also need to keep the locking variable in separate cache lines, because while consumers are spinning on the _consumerLock, the active consumer is updating the _head variable. It avoids another contention. And on the side of the Producer, while producers are spinning on the _producerLock, they won’t slow down the active producer which is updating _tail.
inline LowLockQueue::LowLockQueue() {
    _head = _tail = new Node(nullptr);
    _producerLock = _consumerLock = false;
}
inline ~LowLockQueue::LowLockQueue() {
    while (_head != nullptr) {
        Node* tmp = _head;
        _head = tmp->_next;
        delete tmp->_value;
        delete tmp;
    }
}

7.3 Producer

void LockFreeQueue::Produce(const T& t) {
    Node* tmp = new Node( new T(t));
    while (_producerLock.exchange(true)) { } // Mutex
    _tail->_next = tmp;
    _tail = tmp;
    _producerLock = false;
}

Now _tail is shared between the producers, so it needs to be updated in the critical area.

7.4 Consumer

bool LockFreeQueue::Consume(T& t) {
    while (!_consumerLock.exchange(true)) { } // Mutex
    Node* _head = _head;
    Node* _next = _head->_next;
    if (_next != nullptr) {
        T* val = _next->_value;
        _next->_value = nullptr;
        _head = _next;
        _consumerLock = false;
        t = *val;
        delete _val;
        delete _head;
        return true;
    }
    _consumerLock = false;
    return false;
}

One more time, we do everything we can out of the non-critical area. It improves concurrency as much as possible. The problem is that the code still uses two blocks.
Two documents describe an improvement. But these two solutions require a double-width compare-and-swap operation (DCAS) to treat a pointer and an integer in a single atomic unit. And not all platforms can handle this...

References:
M. Michael and M. Scott “Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” 1996

M. Herlihy, V. Luchango and M. Moir “Obstruction-Free Synchronization : Double-Ended Queues as an Example” 2003

**Key properties**

In this section I explain some of the metrics used to evaluate a multithreaded software.

**Throughput**
The throughput is the total amount of work the system is able to accomplish.

**Scalability**
Scalability is the ability to use more cores to get more work done.

**Contention**
Contention represents how much different threads interfere with each other by fighting for resources

**Oversubscription**
A state of oversubscription is having more CPU-bound work ready to execute than available hardware to execute it

**8. Internship**

In this chapter I will present the technical environment in which I worked for 6 months, then the performance analysis of the current queues, followed by the implementation of my lock-free queue and the analysis of the improvements.

**8.1 Technical Environment**

I used four components during my internship: Toolbox, Tracer, LogServer and LogInjector. You can find the descriptions in Annexe 5 - Technical Environment.doc.

**8.2 Lock-Free Queues**
I coded in C++ an implementation of the lock-free queue in the Toolbox component. This queue is used by the LogServer component to handle the communication between threads:

8.2.1 Performance Analysis

The performance analysis is based on a simple test. Each time a log is injected by the LogInjector, it writes the time in a file. Then the log is received by the LogServer, a plugin get it and run the “exec” function. In our test, this function writes the time in another file. At the end of the test, we have the elapsed time needed to process each individual log. In Amadeus environments several log servers are running on the same host, especially for test system log servers which are receiving a low traffic.

A script (average.sh) calculates the average begin hour and the average finish hour. It provides us the average time it takes to a log to be handled. As a log goes through two queues, any change on the queue algorithm should have an impact on the average time measured.

Before any code modification, I ran benchmarks tests to have a reference before future improvements:
<table>
<thead>
<tr>
<th>LGS</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2ms</td>
<td>2ms</td>
<td>2ms</td>
<td>1ms</td>
</tr>
<tr>
<td>10</td>
<td>2ms</td>
<td>6ms</td>
<td>4ms</td>
<td>4ms</td>
</tr>
<tr>
<td>30</td>
<td>2ms</td>
<td>6ms</td>
<td>5ms</td>
<td>5ms</td>
</tr>
<tr>
<td>50</td>
<td>4ms</td>
<td>6ms</td>
<td>8ms</td>
<td>6ms</td>
</tr>
<tr>
<td>75</td>
<td>7ms</td>
<td>9ms</td>
<td>8ms</td>
<td>6ms</td>
</tr>
<tr>
<td>100</td>
<td>12ms</td>
<td>12ms</td>
<td>9ms</td>
<td>12ms</td>
</tr>
<tr>
<td>200</td>
<td>20ms</td>
<td>9ms</td>
<td>6ms</td>
<td>6ms</td>
</tr>
<tr>
<td>500</td>
<td>315ms</td>
<td>315ms</td>
<td>315ms</td>
<td>315ms</td>
</tr>
</tbody>
</table>

Then I needed a lower bound of the possible improvement. The best possible result is to have a queue in which nobody ever waits. I deleted all the mutexes and condition variables and ran a new test. Of course we know that lots of logs will be lost, and some LogServers will core, because some logs will be written in the queue at the same time, or parameters of the queue will be read and written altogether. But we only need some data to get our bound:

<table>
<thead>
<tr>
<th>LGS</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2ms</td>
<td>2ms</td>
<td>2ms</td>
<td>1ms</td>
</tr>
<tr>
<td>10</td>
<td>2ms</td>
<td>6ms</td>
<td>4ms</td>
<td>4ms</td>
</tr>
<tr>
<td>30</td>
<td>2ms</td>
<td>6ms</td>
<td>5ms</td>
<td>5ms</td>
</tr>
<tr>
<td>50</td>
<td>4ms</td>
<td>6ms</td>
<td>8ms</td>
<td>6ms</td>
</tr>
<tr>
<td>75</td>
<td>7ms</td>
<td>9ms</td>
<td>8ms</td>
<td>6ms</td>
</tr>
<tr>
<td>100</td>
<td>12ms</td>
<td>12ms</td>
<td>9ms</td>
<td>12ms</td>
</tr>
<tr>
<td>200</td>
<td>20ms</td>
<td>9ms</td>
<td>6ms</td>
<td>6ms</td>
</tr>
<tr>
<td>500</td>
<td>315ms</td>
<td>315ms</td>
<td>315ms</td>
<td>315ms</td>
</tr>
</tbody>
</table>

Now we know the minimum time a log can spend in the queue. We can try our new implementation and compare it.

### 8.2.2 Implementation of the 1 Producer – 1 Consumer Algorithm

In the toolbox component I implemented exactly the code provided by Herb Sutter, except that each assignment of a shared data is handled by the AtomicAssign function seen in 7.2.2.

### 8.2.3 Improvement Analyze

In order to test my code, I created instances of LogServers on a machine, and one LogInjector per LogServer on another machine. In order to have meaningful results I needed two machines for my benchmarks on which I would be the only user, because if there was another person working on it, the performance of the machine would be a new variable to take in account. Thus I reserved two machines. We have seen that each log is treated by one or several plugins in a LogServer. The work of plugins wasn’t an interesting data so in my tests plugins don’t do anything (especially no
writes on disk to avoid I/O penalty). As results depend on the hardware performance, here is some information:

<table>
<thead>
<tr>
<th>Server:</th>
<th>Injector:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 CPU (AMD Opteron Processor 285)</td>
<td>4 CPU (AMD Opteron Processor 885)</td>
</tr>
<tr>
<td>1.8 GHz</td>
<td>1.8 GHz</td>
</tr>
<tr>
<td>RAM: 16GB</td>
<td>RAM: 16GB</td>
</tr>
</tbody>
</table>

**Step 1: Algorithm Modified, Lib Boost not used for mutex**

In this first step, I just put the Consume and Produce functions out of the scope of the mutex. But a mutex is still locked/unlocked for each message in order to set a bench of value (ex: number of message in queue).

Note: The performance depends on the network state and so on the time when it is run. As servers and injectors are on two different machines, all the logs go through the network. The traffic on the network being variable, results of my tests are also variable in time. That’s why I had to rerun the test seen in 6.2. Each time, I run the test on the LockFreeQueue, and immediately after, I run the same test on the initial Queue.

<table>
<thead>
<tr>
<th>Average Time injection-&gt;plugin exec</th>
<th>(plugins do nothing and server/injector are on different servers, Amadeus mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Size : 512</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LGS</th>
<th>LockFreeQueue</th>
<th>Queue</th>
<th>Plugin per LGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>50ms 609us 016ns</td>
<td>54ms 093us 834ns</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>51ms 298us 746ns</td>
<td>61ms 093us 515ns</td>
<td>Frequency</td>
</tr>
<tr>
<td>50</td>
<td>116ms 825us 735ns</td>
<td>126ms 934us 329ns</td>
<td>100 log/sec</td>
</tr>
<tr>
<td>50</td>
<td>217ms 439us 396ns</td>
<td>219ms 085us 926ns</td>
<td>Messages / Plugin</td>
</tr>
</tbody>
</table>

**Average Improvement : ** 1.09
Step 2: Algorithm Modified and improved, Lib boost used for mutex

In this version, if the queue is not empty, the log is directly dequeued by Consumer, and Producer can enqueue without any wait. We now use the boost library for mutexes and conditional variables.

<table>
<thead>
<tr>
<th>LGS</th>
<th>LockFreeQueue</th>
<th>Queue</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>40ms 767us 497ns</td>
<td>48ms 767us 749ns</td>
<td>1.20</td>
</tr>
<tr>
<td>50</td>
<td>61ms 733us 341ns</td>
<td>85ms 719us 172ns</td>
<td>1.39</td>
</tr>
<tr>
<td>50</td>
<td>113ms 500us 201ns</td>
<td>168ms 263us 413ns</td>
<td>1.49</td>
</tr>
<tr>
<td>50</td>
<td>184ms 549us 416ns</td>
<td>302ms 813us 025ns</td>
<td>1.64</td>
</tr>
<tr>
<td>75</td>
<td>102ms 298us 324ns</td>
<td>192ms 101us 683ns</td>
<td>1.88</td>
</tr>
<tr>
<td>100</td>
<td>62ms 035us 895ns</td>
<td>132ms 476us 685ns</td>
<td>2.13</td>
</tr>
<tr>
<td>100</td>
<td>49ms 561us 021ns</td>
<td>144ms 422us 426ns</td>
<td>2.94</td>
</tr>
<tr>
<td>100</td>
<td>33ms 070us 946ns</td>
<td>99ms 930us 848ns</td>
<td>3.00</td>
</tr>
<tr>
<td>200</td>
<td>31ms 215us 046ns</td>
<td>210ms 339us 669ns</td>
<td>6.77</td>
</tr>
<tr>
<td>500</td>
<td>33ms 915us 901ns</td>
<td>249ms 599us 321ns</td>
<td>7.45</td>
</tr>
</tbody>
</table>

The most important thing to see is that the improvement increases with the number of LogServers. And at this step, no logs are lost. The time spent in the LogServer by the logs is not sufficient to analyze performance, because it depends on the network state. Whereas the improvement is network state-free.

In conclusion the implementation is a real success. In benchmarks, with 500 LogServers, the time spend to treat a message can be divided up to 7.45. The code is stable and no messages are lost. In order to validate these improvements, the lock-free queue has been loaded in an environment of test (DEV). Before loading a new release in production, it needs to validate several steps. DEV is one of the test environments. It is connected to all applications.

You can find a detailed analysis of the results in Annexe 1 - Results DEV Lock Free Queue. I compare traffic between two machines, one with a classic queue
(obead201) and one with a lock-free queue (obead401). Here is the most relevant information:

Between January 1st and January 4th, the traffic on the two machines was very similar. obead401 (Lock-Free) was handling 60,000 TPS:

Meanwhile, obead201 was receiving more or less 55,000 TPS:
The comparison of their CPU usage illustrates the improvement (maybe smaller than expected) brought by the Lock-Free queue implementation:

In red, obedient201 (Classic), in green, obedient401 (Lock-Free). We can see explicitly that even if obedient401 receives 5,000 TPS more than obedient201, its CPU usage is smaller.

I was expecting better result than that. Of course there is still an improvement, but regarding the benchmarks, I thought that the CPU usage would be more reduced. In DEV, there are around 2000 LogServers, logging thousands of messages. It leads to an enormous number of kernel calls to lock and unlock mutexes. We know that lock-free queues are more efficient with a non-empty queue and in the DEV environment it is not the case. Maybe the improvements will be more outstanding with the SI (Services Integrator) queues which are more requested.
8.3 IPC Queues

As described in the introduction, at the end of my official internship, I had the occasion to work on an IPC problem. We wanted to add an option in the LogServer which allows the user to execute a plugin in a new process instead of a new thread. I tried several solutions to handle the IPC (Inter-Process Communication), to finally use what Lenaic Huard from the OTF team (Open Transactional Framework) has coded. In a first part I will explain for what purpose he has implemented this IPC, and then how we can use it for our problem.

8.3.1 OTF Today:

Clients target a SAP (Service Access Point) and a service

Traffic is load balanced to OTF Application Servers

Each application server hosts a set of OTF Application Routines

OTF Applicative Routine
Each Application Server is composed by a Front-End, which receives messages and distributes them among Back-Ends. It has one queue per Back-End to send them messages, and one queue for all Back-Ends to receive their answers. These queues are exactly the ones I improved in the first part of my internship.

IPC communication is used between the front-end and the several back-ends. Today, TCP/IP is used.
8.3.2 OTF Tomorrow:

With that model, if we need to face an increase of the input throughput, we just have to create more Application Servers. But there is a problem with another business branch of Amadeus (Fair Quote). In their case, an Application Server can have several thousands of Back-Ends for one Front-End. Of course the queues are likely to be completely overloaded. That’s why the OTF team tried to improve the communication between Front-End and Back-End. Instead of TCP/IP, they will use queues in shared memory:

Details:

Data is put in a segment in shared memory by the server. The Server signals that a data is available in shared memory by putting a message in the message queue. The Client gets this message and knows that a data is available. He gets the data from the shared memory and signals to the Server that the data can be deleted by using another message queue. The Server gets the message and finally deletes the data.

More details:

The message queue is a POSIX queue ([http://linux.die.net/man/7/mq_overview](http://linux.die.net/man/7/mq_overview)). The Client uses a “blocking read” provided by the POSIX queue. If he tries to read in the queue when it’s empty, he’s sent to sleep by the kernel. Then when the Server put a message in the queue (a data is available), the Client is awoken by the kernel.

This first solution can raise several questions:

Why are we using a blocking read?

First, we want to avoid polling. In that case, the Client checks periodically the message queue. In the 1 Producer – 1 Consumer case, it could be ok. But in our case,
we have hundreds of couples Client-Server on the same machine. If all the queues are empty, each Client polls the queue periodically. And when a queue receives a message, the corresponding Client needs to wait that all the other Client poll the queue before getting the message. With a blocking read, every Client sleeps and they’re awoken only when they need to be.

**Why are we using a message queue?**

Now you could think: “Ok, the blocking read is a good solution, but why can’t we directly read the data in the shared memory using a blocking read?”

The first thing is that POSIX doesn’t provide a blocking read on shared memory. But boost.interprocess does. Boost allows us to use a semaphore in shared memory, and so to implement a blocking read in shared memory. **BUT:**

Using shared memory is really dangerous!

A process can map a segment in shared memory in read only or read/write. If a process tries to write in a read only segment, it dumps a core. If we have a semaphore in shared memory, the client and the server needs to access it and override it. It implies shared memory being accessed in read/write by both Client and Servers. You could think, “Why not? It could work!”.

Let’s take an example. If we have a bugged program (which happens all the time), and this program writes in an address where he shouldn’t (it happens every time you get a core in one of your program). We have several possibilities:

- the address is invalid => segmentation fault
- the address points to a segment in shared memory:
  - this segment is in “read only” mode => segmentation fault
  - this segment is in read/write” mode (which is the case with the boost solution)

=> the bugged program corrupts a shared data structure and finally the processes using this shared memory will core whereas they were not bugged.

In order to avoid this problem, we needed to have read/write process as less as possible, which means only the Server. That’s why the Server provides and deletes the data. As the Client needs to know when the data is available, we used the message queues. The POSIX queues are stored in a private area of the kernel and can’t be corrupted.

The next question is now:

**“These message queues seem really great, why can’t we just use them to handle the IPC?”**

It would be too easy! The messages of a message queue are limited in size, so the data can’t go through it. The Server maps an area in the shared memory, puts the data in it, and then gives the offset where the data is available to the Client via the message queue. When the Client has read it, as he can’t delete it himself, he sends a message to the Server it via another POSIX queue in order to delete. The clean up is handled by a dedicated thread of the Server.
8.3.3. LogServer Today

Each log received by the Logserver is populated to all the plugins. All plugins are executed in different threads.

Plugins in LogServer are created as threads. When a new plugin is added to a LogServer, we need to be sure that it is not going to core and affect other plugins. That’s why we want to create it in a new process.

After two months spent on the lock-free queues, I spent another two months on this subject. My job was to:

- add an option in the configuration file (outOfProcess)
- depending on that option, create the plugin in a new process or not
- create a PluginWrapper that will take care of the communication between the LogServer and the plugin
- use the shared memory queue described before as IPC
- add an option to choose between two modes: Best Effort and Blocking
- add an option to set the low and high watermarks

Warning: I first used fork() instead of fork() + exec(). Fork copy the whole process memory. Then for example, if in our process we have two threads, T1 and T2, and a mutex M1. T1 locks M1, and while M1 is locked, T2 forks. We have a child process with only one thread, M1 locked, and no thread to unlock it!

Exec() clean the memory space and avoids this problem.
8.3.4 LogServer Tomorrow:

An option in the configuration file (etc/Options.xml) allows the user to decide if a given plugin will be executed in another process or not, but also if he wants Best Effort or Blocking, and what watermark does he want.

Example of the configuration file:
<Plugin name="/defaultPlugin" libraryPath="/aPath " status="Activated" outOfProcess="true" mode="Best Effort" lowWaterMark="10" highWaterMark="90" />

The outOfProcess attribute is optional. By default the plugin will be executed in a thread. Each log received by the LogServer is populated to all plugins and pluginWrappers. When a log is sent to a pluginWrapper, the log is transmitted by the shared memory queue seen before. The pluginWrapper reads it and gives it to the plugin.

If a Logserver is killed, all the pluginWrappers and their plugin are killed too. In that case we clean all the shared memory and message queues. If the plugin of a pluginWrapper cores, its father stops sending logs to its child (detected via the SIGCHLD signal).

An outOfProcess plugin has two modes: Best Effort and Blocking. In Best Effort, we discard logs when the queue is full, in blocking mode, when the queue is full we wait till we can send the next log.
### 8.3.5 Plugin Wrapper Interface

We must provide a way to administrate pluginWrappers.

Services available on a pluginWrapper:

- close()
- putlog(…)
- isActive()
- setUnactive()
- setActive()
- isTracing()
- getName()

### 8.3.6 Use cases

Signal Handling is one of the most important parts of this section. When the new process dies, it sends a signal to its father (signal 17). The father has to catch it and work on it. I tested the following use cases to observe the behavior of the system.

<table>
<thead>
<tr>
<th>Event</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kill logsrv</td>
<td>Clean shared memory and message queues and kill all plugin wrappers</td>
</tr>
<tr>
<td>Kill plugin</td>
<td>The LogServer clear all plugin wrappers and keep working</td>
</tr>
<tr>
<td>Shared memory queue is Full</td>
<td>Logs discarded or blocked</td>
</tr>
<tr>
<td>Shared memory queue is Empty</td>
<td>Blocking read, thread put to sleep</td>
</tr>
<tr>
<td>Core plugin</td>
<td>Kill all plugin wrappers, shared memory is lost</td>
</tr>
</tbody>
</table>

I finished this part in 2 months. Everything works fine. We can now create a plugin in a new thread, the user can choose if it works in blocking mode or best effort. He can set the high and low watermarks. I tried the use cases mentioned above and there is no problem. In the future, we could allow the user to execute several plugins in the same pluginWrapper and improve the global behavior when one of them cores.
8.4 Distributed Database

A machine dedicated to logs can receive up to 200000 logs per second. Developers often need to look for logs. At first we wanted to perform a full-text search. But regarding the amount of data, it was impossible. So we decided to provide a highly discriminating search on a single field. CorrID is a field which identifies a conversation composed by one or more logs. The challenge is to allow them to look for logs in a period of one week (today, we are able to search in a period of one minute…). Then we had the idea to use Crescando to perform full-scan search (see 9.4.3).

To create a prototype, we captured one hour of log received by one farm (sialt1). There are three farms like that. One hour of log represents 365 zip file of more or less 8Mo each. An index on one zip file is a file of 2Mo.

Thus, if we want to keep track on one week:

\[
365 \text{ zip file} \times 8 \text{ Mo} \times 24 \text{ h} \times 7 \text{ days} \times 6 \text{ LogServer} \times 3 \text{ farms} = 8.82 \text{ Tb of compressed data (around 100 Tb uncompressed) and an index of 2 Tb (see Annex} \text{ e} 2 \text{ for exact figures)}.
\]

We had to look for the best possible solution in term of performance to be sure that the project was workable. The most important was the database used. I took a look at three cutting-edge DB projects. I worked on the design of a possible solution to this problem. But the project will not truly begin before March. My job was all about answering to technical matters.

8.4.1 Berkeley DB

Oracle Berkeley DB is the industry-leading open source, embeddable database engine that provides developers with fast, reliable, local persistence with zero administration. Oracle Berkeley DB is a library that links directly into your application. Your application makes simple function calls, rather than sending messages to a remote server, eliminating the performance penalty of client-server architectures. Oracle Berkeley DB eliminates the overhead of SQL query processing, enabling applications with predictable access patterns to run faster.


8.4.2 Tokyo Cabinet

Tokyo Cabinet is a library of routines for managing a database. The database is a simple data file containing records; each is a pair of a key and a value. Every key and value is serial bytes with variable length. Both binary data and character string can be used as a key and a value. There is neither concept of data tables nor data types. Records are organized in hash table, B+ tree, or fixed-length array. Tokyo Cabinet is developed as the successor of GDBM and QDBM on the following purposes. They are achieved and Tokyo Cabinet replaces conventional DBM products.

8.4.3 Crescando

Crescando is a scalable, distributed relational table implementation designed to perform large numbers of queries and updates with guaranteed access latency and data freshness. To this end, Crescando leverages a number of modern query processing techniques and hardware trends. Specifically, Crescando is based on parallel, collaborative scans in main memory and so-called "query-data" joins known from data-stream processing. While the approach is not always optimal for a given workload, it provides latency and freshness guarantees for all workloads. Thus, Crescando is particularly attractive if the workload is unknown, changing, or involves many different queries.

Crescando is a joint effort of the ETH Zurich Systems Group and Amadeus IT Sophia-Antipolis.

We would like to use it that way:

![Diagram of Crescando](image)

The model can be viewed as a big table where each row will match one DCXID, and columns would be functional qualifiers that have been stored by Open Backends. Each row will contain:

- => DCXID (primary key)
- => N qualifiers made of (key/value) (example RLOC/ABCDEF, or FlightDate/23JANAF1234)

Such model is similar to Google Big Table. It is not natively supported by Crescando. According to the Crescando development team, it should be possible to implement it with limited impact on the engine.

References: [http://www.ecc.ethz.ch/research/largescalemainmemorydb](http://www.ecc.ethz.ch/research/largescalemainmemorydb)

8.4.4 Prototype

Crescando is more useful if we can’t predict the kind of query. Thus it has been decided to use it to perform full-scan search on logs. Then I had to decide what DB we would use to index CorrIDs. You can take a look at Annexe 2 - LGS Searching Logs - Berkeley against Tokyo Cabinet to see the detailed comparison, and also some researches on available hardware.
I worked on that project for 2 months. It is a really important project, and it has not begun yet, so I hope I will have the chance to work on it later as it seems really interesting. I have done my part of the job, Berkeley has been designated as the database we will use. Tokyo Cabinet is really impressive but it doesn’t fit our multi-threaded problem. Anyway it would be very interesting to know how it compresses data because the difference with Berkeley DB is remarkable.

9. Conclusion

What could I say about this internship? I learnt SO many things. I have finally put into practice everything that I have studied for three years. Linked queues, forks, signal handling, assembly language, database, I feel completely full. I mean, now everything makes sense, I’m finally ready to work, and I think it was the aim of this internship. I faced a lot of problems, but at the end, I achieved everything. I have an implementation of a lock-free queue which works, I have implemented an IPC protocol, using shared memory, POSIX queues and Google Protocol Buffer, and I studied three cutting-edge databases from every angle. I even had the chance to exchange mails with Herb Sutter (http://fr.wikipedia.org/wiki/Herb_Sutter).

Thanks to this internship, I really improved my technical skills, particularly in C++. It's quite impressive to see what I learnt in few months of practice.

Fortunately, I have been helped by the person in charge of the component in the team. I consider myself lucky because the work I have done has been integrated into the LogServer. It’s rewarding!

10. References

Dr.Dobb’s:
The Many Faces of Deadlock (Aug 2008)
http://www.ddj.com/hpc-high-performance-computing/209900973
Lock-Free Code: A False Sense of Security (Sep 2008)
http://www.ddj.com/cpp/210600279
Writing Lock-Free Code: A Corrected Queue (Oct 2008)
http://www.ddj.com/hpc-high-performance-computing/210604448
Writing a Generalized Concurrent Queue (Nov 2008)
http://www.ddj.com/cpp/211601363
Understanding Parallel Performance (Dec 2008)
http://www.ddj.com/cpp/211800538
Measuring Parallel Performance: Optimizing a Concurrent Queue (Jan 2009)
http://www.ddj.com/hpc-high-performance-computing/212201163

http://www.boost.org/
http://www.justsoftwaresolutions.co.uk/
11. Bibliography

“Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” by Maged M. Michael and Michael L. Scott

“Nonblocking Concurrent Data Structures with Condition Synchronization” by William N. Scherer III and Michael L. Scott

“Exceptional C++, 47 Engineering Puzzles, Programming Problems, and Solutions”, by Herb Sutter, Addison-Wesley, 2000


12. Annexes

Annexe 1 - Results DEV Lock Free Queue
Annexe 2 - LGS Searching Logs - Berkeley against Tokyo Cabinet
Annexe 3 - COM Team Meeting Presentation: Multithread, Atomics and Lock Free Queues
Annexe 4 - Boost Details
Annexe 5 – Technical Environment
Results of the Lock-Free queue LGS in DEV

obead401: Lock-Free queue LGS
obead201: Classic LGS

The aim of this document is to compare the CPU usage between bead401 and bead201 while they receive more or less the same TPS.

Between January 1\textsuperscript{st} and January 4\textsuperscript{th}, the traffic on the two machines was very similar. bead401 (Lock-Free) was handling 60 000 TPS:
Meanwhile, obead201 was receiving more or less 55 000 TPS:
The comparison of their CPU usage illustrates the improvement (maybe smaller than expected) brought by the Lock-Free queue implementation:

In red, obead201 (Classic), in green, obead401 (Lock-Free). We can see explicitly that even if obead401 receives 5 000 TPS more than obead201, its CPU usage is smaller.
Moreover, obead401 reached 140,000 TPS around December 31st and no message seemed to have been discarded (to confirm):
BERKELEY DB / TOKYO CABINET
COMPARISON
# TABLE OF CONTENTS

1. Environment..................................................................................................................3
   1.1 Measures for one log file .........................................................................................3
   1.2 Forecast for one hour of logs (6 DU, 1 Node) ......................................................3
   1.3 Forecast for one day of logs ....................................................................................3
   1.4 Forecast for one week of logs ..................................................................................3
   1.5 Conclusion ..............................................................................................................4

2. Improvement ..................................................................................................................4
   2.1 Measures for one log file .........................................................................................4
   2.2 Forecast for one day of logs (6DU, 1 Node) ..........................................................4
   2.3 Forecast for one week of logs ..................................................................................4
   2.4 Tests get/put (6 hours of log, 1 Node, 1 DU) .........................................................4
      2.4.1 Access time distribution ....................................................................................5
      2.4.2 Contention .........................................................................................................6

3. Measures for Crescando DB (6DU, 1 Node) ................................................................7
   3.1 Sizing ......................................................................................................................7

4. Available Hardware ........................................................................................................8

5. Oracle Berkeley DB Licensing Information ................................................................8
1. Environment

The executed code is exactly the same for the two databases. We only switch the open, put, get and close functions. In each test, the database is opened one time and closed at the end. For each log received there are two calls to put (one for logid, one for corrid) and two calls to get (in order to check the data in the database). We chose a BTREE to store the data in the two databases.

IMPORTANT: We only try to compare two solutions with the same environment. The performance can be easily improved (suppression of the two “get” to increase TPS and store data in binary instead of string for disk space).

<table>
<thead>
<tr>
<th>TPS (on lnx148)</th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2902.68</td>
<td>2215.38</td>
</tr>
</tbody>
</table>

1.1 Measures for one log file

Initial Zipped Log File: 7.65Mb
Unzipped File: 50Mb
We inject and index all the logs contained in this file (36 593 logs)

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ logid</td>
<td>logname ] db</td>
<td>32.77 Kb</td>
</tr>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>1.458 Mb</td>
</tr>
<tr>
<td>105 Zipped Log Files</td>
<td>7.5 Mb</td>
<td>7.5 Mb</td>
</tr>
</tbody>
</table>

1.2 Forecast for one hour of logs (6 DU, 1 Node)

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ logid</td>
<td>logname ] db</td>
<td>71.97 Mb</td>
</tr>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>3.21Gb</td>
</tr>
<tr>
<td>230 580 Zipped Log Files</td>
<td>16.47 Gb</td>
<td>16.47 Gb</td>
</tr>
</tbody>
</table>

1.3 Forecast for one day of logs

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ logid</td>
<td>logname ] db</td>
<td>1.73 Gb</td>
</tr>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>76.85 Gb</td>
</tr>
<tr>
<td>5 533 920 Zipped Log Files</td>
<td>395.28 Gb</td>
<td>395.28 Gb</td>
</tr>
</tbody>
</table>

1.4 Forecast for one week of logs

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ logid</td>
<td>logname ] db</td>
<td>12.11 Gb</td>
</tr>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>537.95 Gb</td>
</tr>
<tr>
<td>38 737 440 Zipped Log Files</td>
<td>2.38 Tb</td>
<td>2.38 Tb</td>
</tr>
</tbody>
</table>
1.5 Conclusion
Berkeley DB and Tokyo Cabinet are equivalent in TPS handled. But Tokyo Cabinet saves a lot of disk space. The “recno” db of Berkley DB is really efficient (60% smaller than TC for logid DB) but it’s nothing compared to disk space saved by Tokyo Cabinet on corrid DB (31 Gb).

2. Improvement

logid+offset can be stored on 5 bytes (Offset on 24 bits and LogID on 16 bits)
=>65 536 files of 16.8Mb (16 777 216 bytes) = 1.1Tb (1 099 511 627 776 bytes) / day

2.1 Measures for one log file
Initial Zipped Log File: 7.65Mb
Unzipped File: 50Mb
We inject and index all the logs contained in this file (36 593 logs)

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>1.212 Mb</td>
</tr>
</tbody>
</table>

2.2 Forecast for one day of logs (6DU, 1 Node)

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>63.9 Gb</td>
</tr>
</tbody>
</table>

2.3 Forecast for one week of logs

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ corrid</td>
<td>logid+offset ] db</td>
<td>447.3 Gb</td>
</tr>
</tbody>
</table>

2.4 Tests get/put (6 hours of log, 1 Node, 1 DU)

<table>
<thead>
<tr>
<th></th>
<th>Berkeley DB</th>
<th>Tokyo Cabinet</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>2.56 Gb</td>
<td>1 Gb</td>
</tr>
<tr>
<td>Put (min-max)</td>
<td>3 – 102 us</td>
<td>2-208 us</td>
</tr>
<tr>
<td>Get (min-max)</td>
<td>3 - 26 us</td>
<td>2 - 21 us</td>
</tr>
</tbody>
</table>
2.4.1 Access time distribution

**Distribution of "put" time**

<table>
<thead>
<tr>
<th>Time (us)</th>
<th>BDB</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11-20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20-75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75-150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Distribution of "get" time**

<table>
<thead>
<tr>
<th>Time (us)</th>
<th>BDB</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4.2 Contention

**DS:** Data Store

**CDS:** Berkeley DB Concurrent Data Store (CDS) layers on top of Data Store (DS) adding a locking subsystem which manages concurrent access to databases.

**TC:** Tokyo Cabinet (WARNING: This database can be opened by only one writer at a time; the lock in multithread is done when we open the database, and not when we store or retrieve a record. That’s why the access time is stable, but TPS falls down.)

Logs received at 100 TPS
3. Measures for Crescando DB (6DU, 1 Node)

<table>
<thead>
<tr>
<th>DCX IDs</th>
<th>per hour:</th>
<th>8 519 408</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast per day:</td>
<td>204 465 792</td>
<td></td>
</tr>
<tr>
<td>forecast per week:</td>
<td>1 431 260 544</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unique DCX IDs</th>
<th>per hour:</th>
<th>2 164 989</th>
</tr>
</thead>
<tbody>
<tr>
<td>forecast per day:</td>
<td>51 959 748</td>
<td></td>
</tr>
<tr>
<td>forecast per week:</td>
<td>363 718 236</td>
<td></td>
</tr>
</tbody>
</table>

Logs per DCX IDs 3.93

3.1 Sizing:

For all Nodes, we have about 1 billion of unique DCX IDs => 1 billion of rows in DB
If we dedicate 1 Kb per row => 1 Tb of data in Crescando DB
We need 20 Machines of 50 Gb, with 10 processors per machine, each one scanning 5GB of data in Crescando DB.

----------------------------------------------------------------------------------------------------------------------------------
4. Available Hardware

<table>
<thead>
<tr>
<th>Server</th>
<th>CPUs</th>
<th>Model Name</th>
<th>MHz</th>
<th>cache size</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>lgsap101</td>
<td>4</td>
<td>16GB AMD Opteron(tm) Processor 280</td>
<td>2405.488</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap201</td>
<td>4</td>
<td>16GB Dual-Core AMD Opteron(tm) Processor 2218</td>
<td>2600.119</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap301</td>
<td>8</td>
<td>16GB AMD Opteron(tm) Processor 275</td>
<td>2205.016</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap401</td>
<td>8</td>
<td>16GB Dual-Core AMD Opteron(tm) Processor 2218</td>
<td>2600.087</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap501</td>
<td>4</td>
<td>12GB AMD Opteron(tm) Processor 275</td>
<td>2205.074</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap601</td>
<td>4</td>
<td>16GB AMD Opteron(tm) Processor 275</td>
<td>2205.075</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap102</td>
<td>8</td>
<td>32GB Dual-Core AMD Opteron(tm) Processor 8220</td>
<td>2813.231</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap202</td>
<td>8</td>
<td>64GB Dual-Core AMD Opteron(tm) Processor 8222 SE</td>
<td>3014.198</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lgsap302</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lgsap402</td>
<td>8</td>
<td>Dual-Core AMD Opteron(tm) Processor 8220</td>
<td>2813.232</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap502</td>
<td>8</td>
<td>64GB Dual-Core AMD Opteron(tm) Processor 8220</td>
<td>2813.219</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap602</td>
<td>8</td>
<td>Dual-Core AMD Opteron(tm) Processor 8220</td>
<td>2813.236</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap103</td>
<td>8</td>
<td>32GB Dual-Core AMD Opteron(tm) Processor 8222 SE</td>
<td>3014.198</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap403</td>
<td>16</td>
<td>Quad-Core AMD Opteron(tm) Processor 8389</td>
<td>2913.427</td>
<td>512 KB</td>
<td>4</td>
</tr>
<tr>
<td>lgsap503</td>
<td>8</td>
<td>64GB Dual-Core AMD Opteron(tm) Processor 8222 SE</td>
<td>3014.200</td>
<td>1024 KB</td>
<td>2</td>
</tr>
<tr>
<td>lgsap603</td>
<td>8</td>
<td>64GB Dual-Core AMD Opteron(tm) Processor 8222 SE</td>
<td>3014.200</td>
<td>1024 KB</td>
<td>2</td>
</tr>
</tbody>
</table>

5. Oracle Berkeley DB Licensing Information

Oracle employs a dual licensing model that offers customers a choice of either our open source license or a commercial license. Our open source license is OSI-certified and permits use of Berkeley DB in open source projects or in applications that are not distributed to third parties. Our commercial license permits closed-source distribution of an application to third parties and provides business assurance.

This model gives customers significant benefits:

**Open Source License**
- Huge user community
- Very high code quality
- Easier debugging and integration
- Easy download and trial
- No escrow issues
- Freedom from vendor lock-in

**Commercial License**
- Application source code stays private
- Legal assurances, warranties and indemnification
- Full-time, dedicated development team provides ongoing maintenance and development, documentation, testing
- Single vendor to hold accountable
Multithread, Atomics and Lock Free Queues

Nicolas Motte, DEV-AIR-CMD-CSM
Nice, 06-Nov-09
## Document control

<table>
<thead>
<tr>
<th>Security level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company</td>
</tr>
<tr>
<td>Department</td>
</tr>
<tr>
<td>Author</td>
</tr>
<tr>
<td>Reviewed by</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Approved by</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Version</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
Purpose

- Comfort for users
- Simultaneous execution
Purpose

\[
\begin{pmatrix}
    a_{11} & a_{12} & a_{13} \\
    a_{21} & a_{22} & a_{23} \\
    a_{31} & a_{32} & a_{33}
\end{pmatrix}
\times
\begin{pmatrix}
    b_{11} & b_{12} & b_{13} \\
    b_{21} & b_{22} & b_{23} \\
    b_{31} & b_{32} & b_{33}
\end{pmatrix}
= \\
\begin{pmatrix}
    a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\
    a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\
    a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}
\end{pmatrix}
\]

Singlethread : 9 operations serialized
Multithread : 9 operations parallelized => speed up
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
- Warnings
- Terminology
- Test Bench
Downsides

- More context switches
- Complexity (Debug)
- Shared Resources

Diagram:

- Thread1
- Global variable
- Thread2
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use : Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
Tools

- Mutex (MUTual EXclusion)
  - It delimits a critical area:

```c
lock(mutex)
//access to shared variables
unlock(mutex)
```
Tools

Semaphore

<table>
<thead>
<tr>
<th>Sem</th>
</tr>
</thead>
<tbody>
<tr>
<td>int count</td>
</tr>
<tr>
<td>list waiting</td>
</tr>
</tbody>
</table>

(P(sem) { 
  If (sem.count<=0) 
  { 
    goToSleep() 
  } 
  sem.count--
})

(V(sem) { 
  sem.count++ 
  If (sem.count<=0) 
  { 
    wakeUp() 
  }
})

Mutex : Semaphore with sem.val=0
Tools

- Monitor (Concept)
  - Object intended to be used safely by more than one thread (Implementation: mutex)
  - Provides a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met (Implementation: condition variable)
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
  - Atomic Variable
  - Lock-Free Queue
    - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
Practical use: Queues

- Producer
- Consumer

Queue:

```
  iHead  x  iTail
```

Producers and consumers use queues to manage the flow of data. Queues are data structures used for organizing data in a specific order, typically first in, first out (FIFO). In this context, producers add data to the queue, while consumers remove data from the queue, ensuring a coordinated and organized data exchange.
Practical use: Queues

Producer
While (true) {
    put(data)
    iTail->next=data
    iTail=iTail->next
}

Consumer
While (true) {
    get(data)
    data=iHead->next->val
    iHead=iHead->next
}
Practical use: Queues

- Producer
  While (true) {
  put(data)
  iTail->next=data
  iTail=iTail->next
  }

- Consumer
  While (true) {
  get(data)
  data=iHead->next->val
  iHead=iHead->next
  }

- PB: get() before put() → core dumped
Practical use: Queues

Producer
While (true) {
    put(data)
    V(semEmpty)
}

Consumer
While (true) {
    P(semEmpty)
    get(data)
}

Queue:

iHead

iTail

semEmpty.count=0
Practical use: Queues

- **Producer**
  ```
  While (true) {
    P(semFull)
    put(data)
    V(semEmpty)
  }
  ```

- **Consumer**
  ```
  While (true) {
    P(semEmpty)
    get(data)
    V(semFull)
  }
  ```

**Queue**:  
- `iHead`
- `iTail`
- `semEmpty.count=0`
- `semFull.count=MAX_SIZE`
Practical use: Queues

- Producer
  While (true) {
    P(semFull)
    put(data)
    V(semEmpty)
  }

- Consumer
  While (true) {
    P(semEmpty)
    get(data)
    V(semFull)
  }

- Pb: Multiple producers  ->  data lost

Queue:

- iHead
- iTail
- semEmpty.count=0
- semFull.count=MAX_SIZE
**Practical use: Queues**

- **Producer**
  - `While (true) {`
  - `P(semFull)`
  - `lock(mutex)`
  - `put(data)`
  - `unlock(mutex)`
  - `V(semEmpty)`
  - `}`

- **Consumer**
  - `While (true) {`
  - `P(semEmpty)`
  - `lock(mutex)`
  - `get(data)`
  - `unlock(mutex)`
  - `V(semFull)`
  - `}`

Queue:

- `iHead`
- `iTail`
- `semEmpty.count=0`
- `semFull.count=MAX_SIZE`
Practical use: Queues

Producer
While (true) {
  lock(mutex)
  CFull.wait()
  put(data)
  CEmpty.signal()
  unlock(mutex)
}

Consumer
While (true) {
  lock(mutex)
  CEmpty.wait()
  get(data)
  CFull.signal()
  unlock(mutex)
}

Queue:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

iHead   iTail

CEmpty -> mutex
CFull -> mutex
Practical use: Queues

- **Producer**
  ```java
  While (true) {
    lock(mutex)
    CFull.wait()
    put(data)
    CEmpty.signal()
    unlock(mutex)
  }
  ```

- **Consumer**
  ```java
  While (true) {
    lock(mutex)
    CEmpty.wait()
    get(data)
    CFull.signal()
    unlock(mutex)
  }
  ```

**Queue:**
- **iHead**
- **iTail**

**Pb:** With lots of processes, everyone is locked on the mutex
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
Atomic Variable

- Each access to a variable (read or write) is atomic
- Included in the next C++ standard (C++0x)
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
  - Terminology
- Test Bench
1 Producer - 1 Consumer Case

- Data structure

```c
LockFreeQueue {
    struct Node {...}
    Node *first
    Atomic<Node*> iHead
    Atomic<Node*> iTail
}
```

Lock-Free Queue:

```
  first  iHead  iTail
```
1 Producer - 1 Consumer Case

- Lock Version (Simplified):
  - Producer
    - While (true) {
      - lock(mutex)
      - put(data)
      - CEmpty.signal()
    }
  - Consumer
    - While (true) {
      - lock(mutex)
      - CEmpty.wait()
      - get(data)
      - unlock(mutex)
    }

Queue:
- Queue:
  - Queue:
    - CEmpty -> mutex
    - iHead
    - iTail
1 Producer - 1 Consumer Case

- **Lock-Free Version:**
  ```
  Producer
  While (true) {
    put(data)
    CEmpty.signal()
  }

  Consumer
  While(true) {
    if (!get(data) {
      lock(mutex)
      CEmpty.wait()
      unlock(mutex)
    }
  }
  ```

- **A thread waits only if the queue is empty.**
1 Producer - 1 Consumer Case

- Implementation of put and get:

  ```c
  void put(data) {
    iTail->next = new Node(data)
    iTail = iTail->next
    while (first != iHead) {
      Node *tmp = first
      first = first->next
      delete tmp
    }
  }
  ```

  ```c
  void get(data) {
    if (iHead != iTail) {
      data = iHead->next
      iHead = iHead->next
      return true;
    }
    return false
  }
  ```

- Each shared variable must be atomic
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
- Warnings
- Terminology
- Test Bench
Warnings

- Code mustn’t be reordered by compiler
  - Be careful with code optimization
  - Avoid branch predictions
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
  - Warnings
- Terminology
- Test Bench
Terminology

- **Compare And Swap (CAS)**

- **Example:**

  ```java
  If (producerLock.compare_and_swap(false,true))
  =
  "If I’m the one who gets the lock"
  ```

- **Pb:** “ABA Problem”
Terminology

- Double-Compare And Swap (DCAS)
  - CAS with an integer increased at each swap
Agenda

- Multithreading
  - Purpose
  - Downsides
  - Tools
- Practical use: Queues
- Atomic Variable
- Lock-Free Queue
  - 1 Producer - 1 Consumer Case
- Warnings
- Terminology
- Test Bench
Test Bench

**Metric**

Each log goes through 2 queues
Test Bench

<table>
<thead>
<tr>
<th>Number of LogServers</th>
<th>Log Size : 512</th>
<th>UDP</th>
<th>LockFreeQueue</th>
<th>Queue</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td>40ms 767us 497ns</td>
<td>48ms 767us 749ns</td>
<td>1.20</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>61ms 733us 341ns</td>
<td>85ms 719us 172ns</td>
<td>1.39</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>113ms 500us 201ns</td>
<td>168ms 263us 413ns</td>
<td>1.49</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>184ms 549us 416ns</td>
<td>302ms 813us 025ns</td>
<td>1.64</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td>102ms 298us 324ns</td>
<td>192ms 101us 683ns</td>
<td>1.88</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>62ms 035us 895ns</td>
<td>132ms 476us 685ns</td>
<td>2.13</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>49ms 561us 021ns</td>
<td>144ms 422us 426ns</td>
<td>2.94</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>33ms 070us 946ns</td>
<td>99ms 930us 848ns</td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>31ms 215us 046ns</td>
<td>210ms 339us 669ns</td>
<td>6.77</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>30ms 376us 808ns</td>
<td>209ms 826us 679ns</td>
<td>6.97</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>33ms 915us 901ns</td>
<td>249ms 599us 321ns</td>
<td>7.45</td>
<td></td>
</tr>
</tbody>
</table>

Note: the time measured depends on the network state, only the improvement has a meaning
Test Bench

Results

![Graph showing the improvement in performance with increasing number of LogServers.](image-url)
Questions
BOOST DETAILS

References: www.boost.org

void notify_one();

Effects: If there is a thread waiting on *this, change that thread's state to ready. Otherwise there is no effect.

Notes: If more than one thread is waiting on *this, it is unspecified which is made ready. After returning to a ready state the notified thread must still acquire the mutex again (which occurs within the call to one of the condition object's wait functions.)

Mutex Concept
A Mutex object has two states: locked and unlocked. Mutex object state can only be determined by a lock object meeting the appropriate lock concept requirements and constructed for the Mutex object.
A Mutex is NonCopyable.
For a Mutex type M and an object m of that type, the following expressions must be well-formed and have the indicated effects.

Mutex Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>M m;</td>
<td>Constructs a mutex object m. Postcondition: m is unlocked.</td>
</tr>
<tr>
<td>(&amp;m)-&gt;~M();</td>
<td>Precondition: m is unlocked. Destroys a mutex object m.</td>
</tr>
</tbody>
</table>

M::scoped_lock A model of ScopedLock

ScopedLock Concept
A ScopedLock is a refinement of Lock. For a ScopedLock type L and an object lk of that type, and an object m of a type meeting the Mutex requirements, and an object b of type bool, the following expressions must be well-formed and have the indicated effects.

ScopedLock Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>L lk(m);</td>
<td>Constructs an object lk, and associates mutex object m with it, then calls lock()</td>
</tr>
<tr>
<td>L lk(m,b);</td>
<td>Constructs an object lk, and associates mutex object m with it, then if b, calls lock()</td>
</tr>
</tbody>
</table>
template<typename ScopedLock>
bool timed_wait(ScopedLock& lock, const boost::xtime& xt);

Requires: ScopedLock meets the ScopedLock requirements.

Effects: Releases the lock on the mutex object associated with lock, blocks the current thread of execution until readied by a call to this->notify_one() or this->notify_all(), or until time xt is reached, and then reacquires the lock.

Returns: false if time xt is reached, otherwise true.

Throws: lock_error if !lock.locked()
TECHNICAL ENVIRONMENT

1. Toolbox

Miscellaneous wrappers and tools for cross-platform system programming

Abstract:
A component offering C++ wrappers for system programming and cross-platform utilities, for data serialization, buffer management, etc.

Description:
List of the currently available tools:

Generic strings

- toolbox::BoundedString : a fixed-size string class (template)
- toolbox::OString : an optimized string class

Buffer, object and data management

- toolbox::Marshaller : tools to serialize / unserialize objects
- toolbox::BufferMgr : buffer pooling
- toolbox::ObjectMgr : object pooling (template)

Multithreading

- toolbox::Thread : static methods to deal with threads (creation, joining, suspending...)
- toolbox::ThreadMutex : basic thread mutex
- toolbox::RecursiveThreadMutex : recursive thread mutex
- toolbox::RWThreadMutex : thread mutex for read / write operations
- toolbox::CondThreadMutex : synchronized condition variable
- toolbox::Guard : simple synchronization guard
- toolbox::ReadGuard / toolbox::WriteGuard : read / write synchronization guard

Other

- toolbox::OS : cross-platform wrappers at OS level
- toolbox::TimeValue : tools to deal with time and date
- toolbox::Exception : base exception class
- toolbox::HashTable : lightweight hash table
- toolbox::VolatileUID : reusable IDs
- toolbox::Singleton : Singleton pattern
- toolbox::Tools : basic low-level tools
- toolbox::BitSet : bit set management class
- toolbox::OptionParser : command line option parser
- toolbox::FSM : pattern for a finite state machine
- toolbox::Queue : queues to improve
2. Tracer

Standard OBE logging library

Abstract:
Tracer is a component that allows C++ developers to delegate all the trace and exception management to a separate library.

The underlying trace mechanism is based on three features:
Each trace message is assigned a functional group and a level (or severity). Trace output is delegated to trace handlers (console, file, ...). Each handler has a filter on the eligible trace groups and levels. A trace manager is in charge of all the trace resources. Tracer also defines a hierarchy of exception classes mapped to the error levels. Tracer can be configured dynamically from an API, or statically from a configuration file.

Tracer is designed to work in a single-thread or multi-thread environment. In a multi-thread environment, traces are queued and processed by a dedicated thread. Tracer provides macros for generating traces and for throwing exceptions. Generating a trace in your application code is as simple as this:

TRC_TRACE(MDW, INFO, "My first trace")

which will generate the following output:

2003/09/22 11:19:41.083482 ncegc202 MDW INFO <Test.cpp#999> My first trace

The current version supports the following trace output handlers:

Console (standard input and output)
- File
- Shared File
- Rolling File (with a maximum size)
- Sized file (with a minimum and a maximum size)
- Remote log server (UDP/TCP)
- BMC Patrol agent
- Event Log (Windows NT only)

3. LogServer

UDP or TCP server to receive logs from various nodes and processes

Abstract:
The Log Server is a UDP or TCP server that acts as a central point for collecting logs from various applications. The collected logs can be displayed at the console and/or written to log files. Logs can be written to separate files according to a discriminant value provided with each log.
4. LogInjector

Abstract:
Tool able to inject logs in a LogServer via UDP or TCP. Different options allow you to choose the size and the number of the logs and the sending frequency. Logs sent can be generated randomly or from a file. It is used only for test purpose.