Best Practices

- General
- Multithreading
- Parallel
- Memory Model
Best Practices

General

Multithreading

Memory Model
Code Reviews

```cpp
map<string, int> teleBook = {"Dijkstra", 1972},

shared_timed_mutex teleBookMutex;

void addToTeleBook(const string& na, int tele){
    lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
    cout << "STARTING UPDATE " << na << endl;
    teleBook[na] = tele;
    cout << "... ENDING UPDATE " << na << endl;
}

void printNumber(const string& na){
    shared_lock<shared_timed_mutex> readerLock(teleBookMutex);
    cout << na << ": " << teleBook[na] << endl;
}

thread reader1([]{ printNumber("Scott"); });
thread reader2([]{ printNumber("Ritchie"); });
thread w1([]{ addToTeleBook("Scott", 1968); });
thread reader3([]{ printNumber("Dijkstra"); });
thread reader4([]{ printNumber("Scott"); });
thread w2([]{ addToTeleBook("Bjarne", 1965); });
thread reader5([]{ printNumber("Scott"); });
thread reader6([]{ printNumber("Ritchie"); });
thread reader7([]{ printNumber("Scott"); });
thread reader8([]{ printNumber("Bjarne"); });

reader1.join(), reader2.join();
reader3.join(), reader4.join();
reader5.join(), reader6.join();
reader7.join(), reader8.join();
w1.join(), w2.join();

cout << "The new telephone book" << endl;
for (auto teleIt: teleBook){
    cout << teleIt.first << ": " << teleIt.second << endl;
}
```

![Code execution result](image)
Code Reviews

map<string,int> teleBook{"Dijkstra", 1972},

shared_timed_mutex teleBookMutex;

void addToTeleBook(const string& na, int tele){
  lock_guard<shared_timed_mutex> writerLock(teleBookMutex);
  cout << "\nSTARTING UPDATE " << na;
  this_thread::sleep_for(chrono::milliseconds(500));
  teleBook[na]= tele;
  cout << " ... ENDING UPDATE " << na << endl;
}

void printNumber(const string& na){
  shared_lock<shared_timed_mutex> readerLock(teleBookMutex);
  cout << na << ": " << teleBook[na] << endl;
}

thread reader1{}{ printNumber("Scott"); }
thread reader2{}{ printNumber("Ritchie"); }
thread w1{}{ addToTeleBook("Scott",1968); }
thread reader3{}{ printNumber("Dijkstra"); }
thread reader4{}{ printNumber("Scott"); }
thread w2{}{ addToTeleBook("Bjarne",1965); }
thread reader5{}{ printNumber("Scott"); }
thread reader6{}{ printNumber("Ritchie"); }
thread reader7{}{ printNumber("Scott"); }
thread reader8{}{ printNumber("Bjarne"); }

reader1.join(), reader2.join();
reader3.join(), reader4.join();
reader5.join(), reader6.join();
reader7.join(), reader8.join();
w1.join(), w2.join();

cout << \nThe new telephone book" << endl;
for (auto teleIt: teleBook){
  cout << teleIt.first << ": " << teleIt.second << endl;
}

rainer@seminar:~> readerWriterLocks

Bjarne: 0
Ritchie: 1983
STARTING UPDATE Scott ... ENDING UPDATE Scott

STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@bash
Minimize Sharing

- Summation of a vector with 100 000 000 elements

```cpp
constexpr long long size = 100000000;
...

// random values
std::vector<int> randValues;
randValues.reserve(size);

std::random_device seed;
std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);
for (long long i = 0; i < size; ++i)
    randValues.push_back(uniformDist(engine));
...
// calculate sum
...
Minimize Sharing

- Single-threaded in two variations

```cpp
unsigned long long sum {};  
for (auto n: randValues) sum += n;  
const unsigned long long sum = accumulate(randValues.begin(), randValues.end(), 0ll);
```
Minimize Sharing

- Four threads with a shared summation variable

```c++
#include <vector>
#include <mutex>

void sumUp(unsigned long long& sum, const vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        lock_guard<mutex> myLock(myMutex);
        sum += val[it];
    }
}

...  
unsigned long long sum[];

thread t1(sumUp, ref(sum), ref(randValues), 0, fir);
thread t2(sumUp, ref(sum), ref(randValues), fir, sec);
thread t3(sumUp, ref(sum), ref(randValues), sec, thi);
thread t4(sumUp, ref(sum), ref(randValues), thi, fou);
```
Minimize Sharing

- Four threads with a shared, atomic summation variable

```c
void sumUp(atomic<unsigned long long>& sum, const vector<int>& val, unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum += val[it];
    }
}
void sumUp(atomic<unsigned long long>& sum, const vector<int>& val, unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it], memory_order_relaxed);
    }
}
```

```bash
rainer@suse:~$ calculateWithAtomic
sum.is_lock_free(): true
Time for addition 1.33837 seconds
Result: 549992025
Time for addition 1.34625 seconds
Result: 549992025
rainer@suse:~$ 
```
Minimize Sharing

- Four threads with a local summation variable

```c
void sumUp(unsigned long long & sum, const vector<int> & val,
           unsigned long long beg, unsigned long long end)
{
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i)
    {
        tmpSum += val[i];
    }
    lock_guard<mutex> lockGuard(myMutex);
    sum += tmpSum;
}
```
Minimize Sharing

- The results

<table>
<thead>
<tr>
<th></th>
<th>Single-threaded</th>
<th>4 threads with a lock</th>
<th>4 threads with an atomic</th>
<th>4 threads with an local variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in seconds</td>
<td>0.07 sec</td>
<td>3.34 sec</td>
<td>1.34 sec</td>
<td>0.03 sec</td>
</tr>
</tbody>
</table>

All Fine?
Minimize Sharing

- The CPU utilisation
Minimize Sharing

- The memory wall

The RAM is the bottleneck.
Minimize Waiting (Amdahl’s Law)

\[
\frac{1}{1 - p}
\]

\( p: Parallel \ Code \)

\( p = 0.5 \rightarrow 2 \)
Minimize Waiting (Amdahl‘s Law)

Amdahl's Law

Use Code Analysis Tools (ThreadSanitizer)

- Detecks Data Races
- Memory- and Performance overhead: 10x

```cpp
#include <thread>

int main(){

  int globalVar{};
  std::thread t1([&globalVar]{ ++globalVar; });
  std::thread t2([&globalVar]{ ++globalVar; });
  t1.join(), t2.join();
}
```
Use Code Analysis Tools (ThreadSanitizer)

g++ dataRace.cpp -fsanitize=thread -pthread -g -o dataRace
Use Code Analysis Tools (ThreadSanitizer)

```cpp
bool dataReady = false;
std::mutex mutex_;  
std::condition_variable condVar1; 
std::condition_variable condVar2;

int counter=0;  
int COUNTLIMIT=50; 

void setTrue(){
    while(counter <= COUNTLIMIT){  
        std::unique_lock<std::mutex> lck(mutex_);  
        condVar1.wait(lck,[]){return dataReady == false;});  
        dataReady= true;  
        ++counter;
        std::cout << dataReady << std::endl;  
        condVar2.notify_one();
    }
}

void setFalse(){
    while(counter < COUNTLIMIT){
        std::unique_lock<std::mutex> lck(mutex_);  
        condVar2.wait(lck,[]){return dataReady == true;});  
        dataReady= false; 
        std::cout << dataReady << std::endl;  
        condVar1.notify_one();
    }
}

int main(){
    std::cout << std::boolalpha << std::endl; 
    std::cout << "Begin: " << dataReady << std::endl; 
    std::thread t1(setTrue); 
    std::thread t2(setFalse); 
    t1.join();  
    t2.join(); 
    dataReady= false; 
    std::cout << "End: " << dataReady << std::endl; 
    std::cout << std::endl;
}
```
Use Code Analysis Tools (ThreadSanitizer)

```cpp
begin: false
true

WARNING: ThreadSanitizer: data race (pid=18472)
Read of size 4 at 0x0000000000004350 by thread T2:
  #0 setFalse() /home/rainer/conditionVariablePingPong.cpp:30
  #1 void std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1408
  #2 std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1309
  #3 std::thread::State_impl<std::Bindシンプル veil<void (*)(())> ()> ::M_run() /user/include/c++/6/thread:196
  #4 <null> <null> (libstdc++.so.6:0x00000000c22d)

Previous write of size 4 at 0x0000000000004350 by thread T1 (mutexes: write M11):
  #0 setTrue() /home/rainer/conditionVariablePingPong.cpp:21
  #1 void std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1408
  #2 std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1309
  #3 std::thread::State_impl<std::Bindシンプル veil<void (*)(())> ()> ::M_run() /user/include/c++/6/thread:196
  #4 <null> <null> (libstdc++.so.6:0x00000000c22d)

Location is global `counter` of size 4 at 0x0000000000004350 (conditionVariablePingPong=0x000000004350)

Mutex M11 (0x00000000642a0) created at:
  #0 pthread_mutex_lock <null> (libstdc++.so.6:0x00000000b3c80f)
  #1 _pthread_mutex_lock /user/include/c++/6/x86_64-suse-linux/bits/gthr-default.h:48 (conditionVariablePingPong=0x000000004350)
  #2 std::mutex::lock<std::mutex::*lock>() /user/include/c++/6/lock/std_mutex.h:267 (conditionVariablePingPong=0x000000004350)
  #3 std::unique_lock<std::mutex>::lock() /user/include/c++/6/lock/std_mutex.h:103 (conditionVariablePingPong=0x000000004350)
  #4 std::unique_lock<std::mutex>::unique_lock() /user/include/c++/6/lock/std_mutex.h:197 (conditionVariablePingPong=0x000000004350)
  #5 setTrue() /home/rainer/conditionVariablePingPong.cpp:21
  #6 void std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1408
  #7 std::Bindシンプル veil<void (*)(())> () <user/include/c++/6/functional:1309
  #8 std::thread::State_impl<std::Bindシンプル veil<void (*)(())> ()> ::M_run() /user/include/c++/6/thread:196
  #9 <null> <null> (libstdc++.so.6:0x00000000c22d)

Thread T2 (tid=18472, running) created by main thread at:
  #0 pthread_create <null> (libstdc++.so.6:0x00000000b740)
  #1 std::thread::M_start_thread(std::unique_ptr<std::thread::State, std::default deleter</std::thread::State>
      5d4)
  #2 main /home/rainer/conditionVariablePingPong.cpp:49 (conditionVariablePingPong=0x000000004197c)

Thread T1 (tid=18139, running) created by main thread at:
  #0 pthread_create <null> (libstdc++.so.6:0x00000000b740)
  #1 std::thread::M_start_thread(std::unique_ptr<std::thread::State, std::default deleter</std::thread::State>
      5d4)
  #2 main /home/rainer/conditionVariablePingPong.cpp:48 (conditionVariablePingPong=0x0000000040196b)

SUMMARY: ThreadSanitizer: data race /home/rainer/conditionVariablePingPong.cpp:30 in setFalse()
```
Use Code Analysis Tools (CppMem)
int x = 0, std::atomic<int> y{0};

void writing() {
    x = 2000;
    y.store(11, std::memory_order_release);
}

void reading() {
    std::cout << y.load(std::memory_order_acquire) << " ";
    std::cout << x << std::endl;
}
...
std::thread thread1(writing);
std::thread thread2(reading);
thread1.join(), thread2.join();
Use Code Analysis Tools (CppMem)

```cpp
int x = 0, atomic_int y = 0;

{{
    x = 2000;
    y.store(11, memory_order_release);
}

|||

y.load(memory_order_acquire);
x;
}
}}
```
Use Immutable Data

**Data Race**: At least two threads access a shared variable at the same time. At least one thread tries to modify the it.

<table>
<thead>
<tr>
<th>Shared?</th>
<th>Mutable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>no</td>
<td>OK</td>
</tr>
<tr>
<td>yes</td>
<td>OK</td>
</tr>
<tr>
<td>yes</td>
<td>OK</td>
</tr>
</tbody>
</table>
Use Immutable Data

The remaining Problem: initialise the data thread-safe

1. Early Initialisation
   
   ```cpp
   const int val = 2011;
   thread t1([&val] { .... });
   thread t2([&val] { .... });
   ```

2. Constant Expressions
   
   ```cpp
   constexpr auto doub = 5.1;
   ```

3. `call_once` and `once_flag`
   
   ```cpp
   void onlyOnceFunc() {
     ....
   }
   call_once(onceFlag, onlyOnceFunc);
   thread t3{ onlyOnceFunc() };
   thread t4{ onlyOnceFunc() };
   ```

4. Static variables in a scope
   
   ```cpp
   void func()
   {
     .... static int val 2011; .... 
   }
   thread t5{ func() };
   thread t6{ func() };
   ```
Use Pure Functions

<table>
<thead>
<tr>
<th>Pure Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce always the same result when given the same arguments</td>
</tr>
<tr>
<td>Has no side effect</td>
</tr>
<tr>
<td>Don't change the global state of the program</td>
</tr>
</tbody>
</table>

Added Value

- Easier to make correctness proofs
- Refactoring and testing is easier
- Results from previous function calls can be memorised
- The sequence of function invocations can automatically be changed or the function can automatically be parallelised
Use Pure Functions

- **Function**

  ```c
  int powFunc(int m, int n){
    if (n == 0) return 1;
    return m * powFunc(m, n-1);
  }
  ```

- **Metafunction**

  ```c
  template<int m, int n>
  struct PowMeta{
    static int const value = m * PowMeta<m, n-1>::value;
  };

  template<int m>
  struct PowMeta<m, 0>{
    static int const value = 1;
  };
  ```
Use Pure Functions

- constexpr Function
  - almost pure functions

```cpp
constexpr int powConst(int m, int n) {
    int r = 1;
    for(int k = 1; k <= n; ++k) r *= m;
    return r;
}
```

```cpp
auto res = powConst(2, 10);
auto constexpr res2 = powConst(2, 10);
```
Best Practices

- General
- Multithreading
- Memory Model
Use Tasks instead of Threads

### Thread

```cpp
int res;
thread t([&]{ res = 3 + 4; });
t.join();
cout << res << endl;
```

### Task

```cpp
auto fut = async([]{ return 3 + 4; });
cout << fut.get() << endl;
```

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Thread</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties Involved</td>
<td>creator thread and child thread</td>
<td>promise and future</td>
</tr>
<tr>
<td>Communication</td>
<td>shared variable</td>
<td>communication channel</td>
</tr>
<tr>
<td>Thread Creation</td>
<td>obligatory</td>
<td>optional</td>
</tr>
<tr>
<td>Synchronisation</td>
<td><code>join</code> call blocks</td>
<td>get call blocks</td>
</tr>
<tr>
<td>Exception in Child</td>
<td>creator thread and child thread die</td>
<td>return value of the promise</td>
</tr>
<tr>
<td>Forms of Communication</td>
<td>values</td>
<td>values, notifications, and threads</td>
</tr>
</tbody>
</table>
Use Tasks instead of Threads

C++20: Extend futures will support composition

- **then**: Execute the future if the previous future is done
- **when_any**: Execute the future if any of the previous future is done
- **when_all**: Execute the future if all of the previous futures are done
Use Tasks instead of Condition Variables

**Thread 1**

```cpp
{
    lock_guard<mutex> lck(mut);
    ready = true;
}
condVar.notify_one();
prom.set_value();
```

**Thread 2**

```cpp
{
    unique_lock<mutex> lck(mut);
    condVar.wait(lck, []{ return ready; });
}
fut.wait();
```

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Condition Variable</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Region</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Spurious Wakeup</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Lost Wakeup</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Repeated Synchronisation</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Pack Mutexes in Locks

No Release of the lock

- Mutex

```cpp
mutex m;
{
    m.lock();
    shrVar = getVar();
    m.unlock();
}
```

- `lock_guard`

```cpp
mutex m;
{
    lock_guard<mutex> myLock(m);
    shaVar = getVar();
}
```
Pack Mutexes in Locks

Locking of the mutexes is different order

Thread 1

Thread 2

Lock 1

Lock 2
Pack Mutexes in Locks

Atomic lock of the mutex

- `unique_lock`

```cpp
unique_lock<mutex> guard1(mut1, defer_lock);
unique_lock<mutex> guard1(mut2, defer_lock);
lock(guard1, guard2);
```

- `scoped_lock (C++17)`

```cpp
std::scoped_lock(mut1, mut2);
```
Best Practices

General

Multithreading

Memory Model
Don't Program lock-free

Guide to Threaded Coding

1. Forget what you learned in Kindergarten (ie stop Sharing)
2. Use Locks
3. Measure
4. Measure
5. Change your Algorithm
6. GOTO 1

∞. Lock-free

Lock-free coding is the last thing you want to do.

- Writing lock-free programs is hard
- Writing correct lock-free programs is even harder

The ugly side of weakly ordered atomics

Extreme complexity.
- The rules are not obvious.
- They're often downright surprising.
- And not even well understood.
- The committee still hasn't figured out how to define memory_order_relaxed.
... and I'm not even going to talk about memory_order_consume.

The specification of release-consume ordering is being revised, and the use of memory_order_consume is temporarily discouraged.
Don't Program lock-free: ABA
Don't Program lock-free: ABA

A lock-free, singly linked list (stack)

- Thread 1
  - wants to remove A
  - stores
    - head = A
    - next = B
  - checks if A == head
  - make B to the new head
  - B is already deleted by Thread 2

- Thread 2
  - removes A
  - removes B and deletes B
  - pushes A back
Use Proven Patterns

Wait with sequentiel consistency

```cpp
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    while (!dataReady.load()){
        std::this_thread::sleep_for(5ms);
    }
    mySharedWork[1] = 2;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady.store(true);
}

int main(){
    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);
    t1.join();
    t2.join();
    for (auto v: mySharedWork){
        std::cout << v << " ";
    }
};
```
Use Proven Patterns

Wait with acquire-release semantic

```cpp
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    while (!dataReady.load(std::memory_order_acquire)){
        std::this_thread::sleep_for(5ms);
    }
    mySharedWork[1] = 2;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady.store(true, std::memory_order_release);
}

int main(){
    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);
    t1.join();
    t2.join();
    for (auto v: mySharedWork){
        std::cout << v << " "; // 1 2 3
    }
}
```
Use Proven Patterns

Atomic counter

```cpp
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> count{0};
void add(){
    for (int n = 0; n < 1000; ++n){
        count.fetch_add(1, std::memory_order_relaxed);
    }
}

int main(){
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n){
        v.emplace_back(add);
    }
    for (auto& t : v) { t.join(); }
    std::cout << count; // 10000
}
```
Don't Reinvent the Wheel

Boost.Lockfree
CDS (Concurrent Data Structures)
Don't Reinvent the Wheel

- Boost.Lockfree
  - Queue
    - A lock-free multi-producer/multi-consumer queue
  - Stack
    - A lock-free multi-producer/multi-consumer stack
- spsc_queue
  - A wait-free single-producer/single-consumer queue (commonly known as ringbuffer)
Don't Reinvent the Wheel

- Concurrent Data Structures (CDS)
  - Contains a lot of intrusive and non-intrusive containers
    - Stacks (lock-free)
    - Queues and Priority-Queues (lock-free)
    - Ordered lists
    - Ordered sets and maps (lock-free and lock-based)
    - Unordered sets and maps (lock-free and lock-based)
Blogs

www.grimm-jaud.de [De]
www.ModernesCpp.com [En]