Basic C++

Concurrency

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Concurrency levels

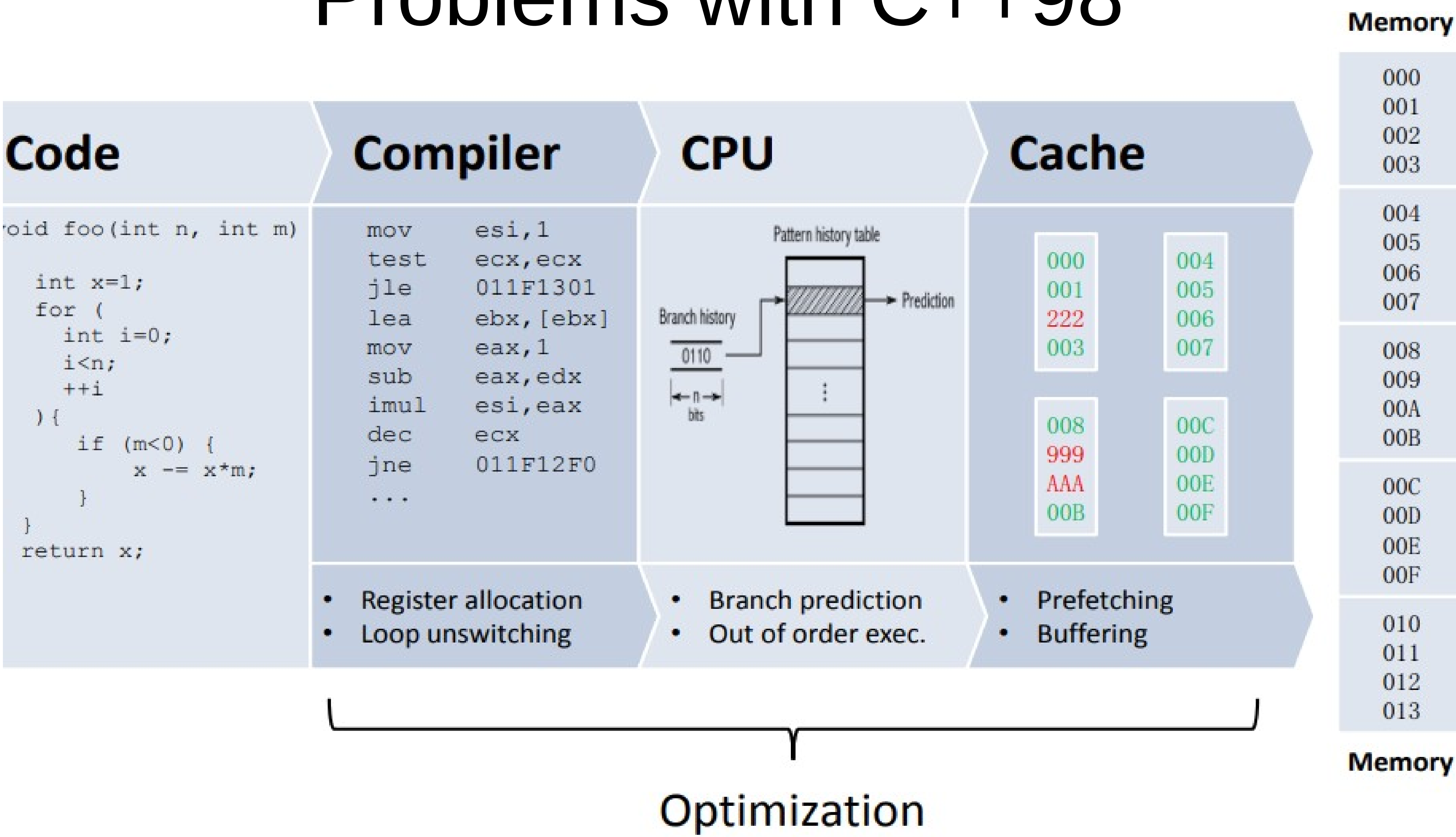
- Process level
- Thread level
  - Library level
  - Kernel level
- Coroutines

Concurrent programming in C++11

Multithreading is just one damn thing after, before, or simultaneous with another. --Andrei Alexandrescu

- Problems with C++98 memory model
- Double-checked locking pattern
- C++11 memory model
- Atomics
- Std::thread
- Mutex/Lock
- Conditional variable
- Future/Promise/Async

Problems with C++98



Valentin Ziegler, Fabio Fracassi C++ Memory Model (Meeting C++ Berlin, 2014)
https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf

Problems with C++98

```
int X = 0;
int Y = 0;

// thread 1
int r1 = X;
if ( 1 == r1 )
    Y = 1;

// thread 2
int r2 = Y;
if ( 1 == r2 )
    X = 1;

// can it be at the end of execution  r1 == r2 == 1 ?
```

Problems with C++98

```
struct S { char a; char b; };
struct S s = { 'a', 'b' };
```

```
// thread 1
s.a = 'x';
```

```
// thread 2
s.b = 'y';
```

```
// thread 1 may compiled:
struct S tmp = x;
tmp.a = 'x';
s = tmp;
```

```
// thread 2 may be compiled:
struct S tmp = x;
tmp.b = 'y';
s = tmp;
```

```
// can the result be { 'a', 'y' } or { 'x', 'b' }?
```

Problems with C++98

Hans Böhm: Threads cannot be implemented as a library
https://dl.acm.org/doi/10.1145/1065010.1065042

Francesco Zappa Nardelli EuroLLVM 2015
https://llvm.org/devmtg/2015-04/slides/CConcurrency_EuroLLVM2015.pdf

Singleton pattern

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
}

// Usage:

   Singleton::istance()-> other_method();
```

Thread safe singleton construction

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
    static Mutex      lock_;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    Guard<Mutex> guard(lock_);  // constructor acquires lock_: not efficient
    // this is now the critical section
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
} // destructor releases lock_
```

Thread safe singleton construction?

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
    static Mutex      lock_;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    // this is now the critical section
    if ( 0 == pinstance )
    {
        Guard<Mutex> guard(lock_);  // constructor acquires lock_: too late!
        // this is now the critical section
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
} // destructor releases lock_
```

Double checked locking pattern

```
Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        Guard<Mutex> guard(lock_);  // constructor acquires lock_
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            pinstance = new Singleton;  // lazy initialization
        }
    }                               // destructor releases lock_
    return pinstance;
}



Singleton::istance()-> other_method();  // does not lock when not necessary
```

Double checked locking pattern

```
Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        Guard<Mutex> guard(lock_);  // constructor acquires lock_
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            pinstance = new Singleton;  // lazy initialization
        }
    }                                   // destructor releases lock_
    return pinstance;
}
```

```
Singleton::istance()-> other_method();  // does not lock when not necessary
```

Meyers and Alexandrescu: C++ and the Perils of Double-Checked Locking:
https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

Problems with DCLP

```
if ( 0 == pinstance )
{
      // ...
      pinstance = new Singleton;  // atomic?
      // ...
}
return pinstance;

// might use half-initialized pointer value
Singleton::istance()-> other_method();
```

- Pointer assignment may not be atomic
  - We can observe a not null but still invalid pointer value

New expression

```
pinstance = new Singleton;  // how this is compiled?
```

- New expression include many steps
  - (1) Allocation space with ::operator new()
  - (2) Run of constructor
  - (3) Returning the pointer

- If the compiler does (1) + (3) and leaves (2) as the last step the pointer points to uninitialized memory area

Observable behavior in C++98

```
void foo()
{
    int x = 0, y = 0;        // (1)
    x = 5;                   // (2)
    y = 10;                  // (3)
    printf( "%d,%d", x, y); // (4)
}
```

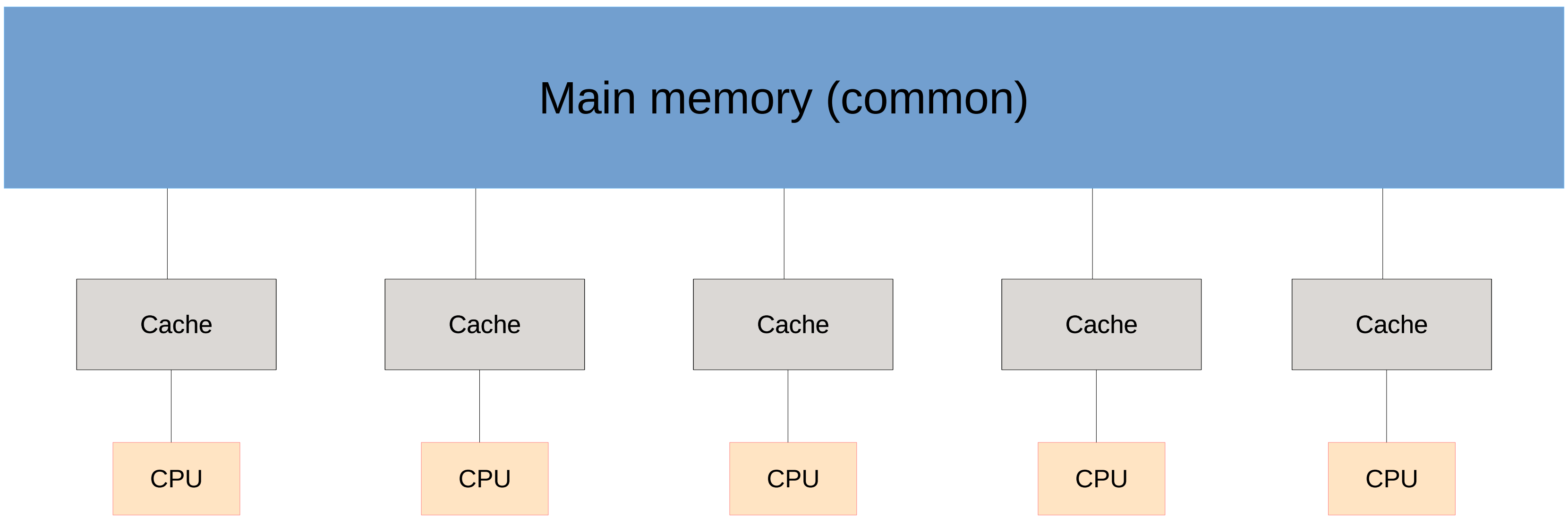
- What is visible for the outer word
  - I/O operations
  - Read/write volatile objects
- Defined by a singled-threaded mind

Sequence point

```
if ( 0 == pinstance )   // re-check pinstance
{
    // pinstance = new Singleton;
    Singleton *temp = operator new( sizeof(Singleton) );
    new (temp) Singleton;  // run the constructor
    pinstance = temp;
}
```

- The compiler can completely optimize out temp
- Even if we are using volatile temp we have issues

Modern hardware architecture



Singleton pattern

```
Singleton *Singleton::instance()
{
    Singleton *temp = pInstance;    // read pInstance

    Acquire();    // prevent visibility of later memory operations
                  // from moving up from this point

    if ( 0 == temp )
    {
        Guard<Mutex> guard(lock_);
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            temp = new Singleton;

            Release();  // prevent visibility of earlier memory operations
                        // from moving down from this point

            pinstance = temp;   // write pInstance
        }
    }
    return pinstance;
}
```

C++11 memory model

- Describes the interactions of threads through memory
- Describes well defined behavior
- Constraints compiler for code generation

- C++ memory model contract
  - Programmer ensures that the program has no data race
  - System guarantees sequentially consistent execution

Terminology

- Only minimal progress guaranties are given on threads:
  - unblocked threads will make progress
  - implementation should ensure that writes in a thread should be visible in other threads "in a finite amount of time".
- The A happens before B relationship:
  - A is sequenced before B or
  - A inter-thread **happens before** B

    == there is a **synchronization point** between A and B
- Synchronization point:
  - thread creation sync with start of thread execution
  - thread completion sync with the return of join()
  - unlocking a mutex sync with the next locking of that mutex

Terminology

- Memory location
  - an object of scalar type
  - a maximal sequence of adjacent bit-fields all having non-zero width
- Data race

  A program contains **data race** if contains two actions in different threads, at least one is not "atomic" **and** neither happens before the other.

- Two threads of execution can update and access separate memory locations without interfering each others

Terminology

- Memory location
  - an object of scalar type
  - a maximal sequence of adjacent bit-fields all having non-zero width
- Data race == undefined behavior

  A program contains **data race** if contains two actions in different threads, at least one is not "atomic" **and** neither happens before the other.
- Two threads of execution can update and access separate memory locations without interfering each others

Sequential consistency

- Sequential consistent (default behavior)
  - Leslie Lamport, 1979
  - Each threads are executed in sequential order
  - The operations of each thread appear in this sequence for the other threads in that order

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                   |  // thread 2
void Produce()                |  void Consume()
{                             |  {
                              |
  d = result;                 |
  flag = true;                |
                              |
                              |
                              |    bool ready = flag;
                              |
                              |
                              |    if ( ready )  use(d);
}                             |  }
```

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                  |  // thread 2
void Produce()               |  void Consume()
{                            |  {
                             |
  d = result;                |
  flag = true;               |
                             |
                             |
                             |    bool ready = flag;
                             |
                             |
                             |    if ( ready )  use(d);
}                            |  }
```

Data race!

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                  |  // thread 2
void Produce()               |  void Consume()
{                            |  {
  m.lock();                  |
  d = result;                |
  flag = true;               |
  m.unlock();                |
                             |    m.lock();
                             |    bool ready = flag;
                             |    m.unlock();
                             |
                             |    if ( ready )  use(d);
}                            |  }
```

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                  |  // thread 2
void Produce()               |  void Consume()
{                            |  {
  m.lock();                  |    bool ready;
  d = result;                |
  flag = true;               |
  m.unlock();                |
                             |    m.lock();
                             |    bool ready = flag;
                             |    m.unlock();
                             |
                             |    if ( ready )  use(d);
}                            |  }
```

Synchronized with

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                    |  // thread 2
void Produce()                 |  void Consume()
{                              |  {
  m.lock();                    |    bool ready;
  d = result;                  |
  flag = true;                 |
  m.unlock();                  |
                               |    m.lock();
                               |    bool ready = flag;
                               |    m.unlock();
                               |
                               |    if ( ready )  use(d);
}                              |  }
```

Happens before

Synchronized with

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                  |  // thread 2
void Produce()               |  void Consume()
{                            |  {
  m.lock();                  |    bool ready;
  d = result;                |
  flag = true;               |
  m.unlock();                |
                             |    m.lock();
                             |    bool ready = flag;
                             |    m.unlock();
                             |
                             |    if ( ready )  use(d);
}                            |  }
```

Happens before

Synchronized with

Sequential consistency

```
std::mutex m;
Data d;
bool flag = false;


// thread 1                        | // start of thread 2
void Produce()                     | void Consume()
{                                  | {
  m.lock();                        |   bool ready;
  d = result;                      |
  flag = true;                     |
  m.unlock();                      |
                                   |   m.lock();
                                   |   bool ready = flag;
                                   |   m.unlock();
                                   |
                                   |   if ( ready )  use(d);
}                                  | }
```

Happens before

Synchronized with

Sequential consistency

```
std::mutex m;
Data d;
std::atomic<bool> flag = false;


// thread 1                 |      // thread 2
void Produce()              |  void Consume()
{                           |  {
  m.lock();                 |    bool ready;
  d = result;               |
  flag.store(true);         |
  m.unlock();               |
                            |    m.lock();
                            |    bool ready = flag.load();
                            |    m.unlock();
                            |
                            |    if ( ready )  use(d);
}                           |  }
```

Sequential consistency

```
std::mutex m;
Data d;
std::atomic<bool> flag = false;


// thread 1                     |        // thread 2
void Produce()                  |   void Consume()
{                               |   {
  m.lock();                     |     bool ready;
  d = result;                   |
  flag.store(true);             |
  m.unlock();                   |
                                |     m.lock();
                                |     bool ready = flag.load();
                                |     m.unlock();
                                |
                                |     if ( ready )  use(d);
}                               |   }
```

Atomic operations

- Supports lock-less concurrent programming
- Non-interleaving read and write operations (no data race)
- May define inter-thread synchronization (based on memory model)
- The encapsulated type must be trivially constructible, copy- and movable.

```
#include <atomic>
template<typename T> struct atomic;       // atomic class template
template<typename P> struct atomic<P*>;   // pointer specialization

#include <memory>
template<typename T> struct atomic<std::shared_ptr<T>>; // C++20 shared_ptr
template<typename T> struct atomic<std::weak_ptr<T>>;   // C++20 weak_ptr

std::atomic<int*>  ptr;
ptr.fetch_add(3);  ptr.fetch_sub(2); // provides pointer arithmetics

std::atomic<std::shared_ptr<T>>  asp(new int);  // atomic shared_ptr
*sp = 42;      // provides shared_ptr (weak_ptr) functionality
```

C++11 memory model

```
int x, y;

// thread 1        |       // thread 2
x = 1;             |       cout << y << ", ";
y = 2;             |       cout << x << endl;
```

```
  In C++03  not even Undefined Behavior

  In C++11  Undefined Behavior
```

C++11 memory model

```
std::atomic<int> x, y;

// thread 1       |       // thread 2
x.store(1);       |       cout << y.load() << ", ";
y.store(2);       |       cout << x.load() << endl;


// Equivalent to:

int x, y;
mutex x_mutex, y_mutex;

// thread 1       |       // thread 2
x_mutex.lock()    |       y_mutex.lock();
x = 1;            |       cout << y << ", ";
x_mutex.unlock()  |       y_mutex.unlock();
y_mutex.lock()    |       x_mutex.lock();
y = 2;            |       cout << x << endl;
y_mutex.unlock()  |       x_mutex.unlock();
```

C++11 memory model (default)

```
std::atomic<int> x, y;
x.store(0); y.store(0);

// thread 1       |       // thread 2
x.store(1);       |       cout << y.load() << ", ";
y.store(2);       |       cout << x.load() << endl;


Result can be:


0 0
2 1
0 1
// never prints: 2 0


Sequential consistency:   atomics == atomic load/store + ordering
```

Memory ordering

- memory_order_seq_cst (default)
- memory_order_consume
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_relaxed

X86/x86_64 does not require additional instructions to implement acquire-release ordering

Relaxed memory order

- Each memory location has a total modification order
  - But this may be not observable directly
- Memory operations performed by
  - The same thread and
  - On the same memory location

  are not reordered with respect of modification order

Relaxed memory order

```
std::atomic<int> x, y;

// relaxed
// thread 1                        |  // thread 2
x.store(1, memory_order_relaxed); |  cout << y.load(memory_order_relaxed) << ", ";
y.store(2, memory_order_relaxed); |  cout << x.load(memory_order_relaxed) << endl;

// Defined, atomic, but not ordered, result may be:
0 0
2 1
0 1
2 0
```

Acquire – release memory order

- A store-release operation **synchronizes** with all load-acquire operations reading a stored value
- Operations preceding the store-release in the releasing thread **happens before** operations following the load-acquire
- On some platforms acquire-release is cheaper than sequention consistency

```
std::mutex m; Data d;
std::atomic<bool> flag = false;

// thread 1                       |  // thread 2
void Produce()                    |  void Consume()
{                                 |  {
  d = result;                     |
  flag.store(true,                |    bool ready =
   memory_order_release);         |         flag.load(memory_order_acquire);
                                  |    if ( ready )  use(d);
}                                 |  }
```

Acquire – release memory order

- A store-release operation **synchronizes** with all load-acquire operations reading a stored value
- Operations preceding the store-release in the releasing thread **happens before** operations following the load-acquire
- On some platforms acquire-release is cheaper than sequention consistency

```
std::mutex m; Data d;
std::atomic<bool> flag = false;

// thread 1                         |  // thread 2
void Produce()                      |  void Consume()
{                                   |  {
  d = result;                       |
  flag.store(true,                  |    bool ready =
   memory_order_release);           |      flag.load(memory_order_acquire);
                                    |    if ( ready )  use(d);
}                                   |  }
```

Acquire – release memory order

- A store-release operation **synchronizes** with all load-acquire operations reading a stored value
- Operations preceding the store-release in the releasing thread **happens before** operations following the load-acquire
- On some platforms acquire-release is cheaper than sequention consistency

```
std::mutex m; Data d;
std::atomic<bool> flag = false;

// thread 1                          |  // thread 2
void Produce()                       |  void Consume()
{                                    |  {
  d = result;                        |
  flag.store(true,                   |    bool ready =
   memory_order_release);            |   flag.load(memory_order_acquire);
                                     |    if ( ready )  use(d);
}                                    |  }
```

Acquire – release memory order

- A store-release operation **synchronizes** with all load-acquire operations reading a stored value
- Operations preceding the store-release in the releasing thread **happens before** operations following the load-acquire
- On some platforms acquire-release is cheaper than sequention consistency

```
std::mutex m; Data d;
std::atomic<bool> flag = false;

// thread 1                          |  // thread 2
void Produce()                       |  void Consume()
{                                    |  {
  d = result;                        |
  flag.store(true,                   |    bool ready =
   memory_order_release);            |    flag.load(memory_order_acquire);
                                     |    if ( ready )  use(d);
}                                    |  }
```

Acquire – release memory order

```
// acquire-release
// thread 1                      |  // thread 2
x.store(1, memory_order_release); |  cout << y.load(memory_order_acquire) << ", ";
y.store(2, memory_order_release); |  cout << x.load(memory_order_acquire) << endl;

// In C++11 Defined and the result can be:
0 0
2 1
0 1
// never prints: 2 0, but can be faster than strict ordering.
// results may be different in more complex programs
```

Consume – release memory order

- Operations preceding the store-release in the releasing thread **happens before** an operation X in the consuming thread where X has a **data dependency** on the loaded value

```
int d = 0;
std::atomic<int *> ptr = std::nullptr;

// thread 1                        |  // thread 2
void Produce()                     |  void Consume()
{                                  |  {
  d = 42                           |    int *p;
  ptr.store(&d,                    |
   memory_order_release);          |    if (p= ptr.load(memory_order_consume))
                                   |    {
                                   |      assert ( 42 == *p );
                                   |
                                   |      assert ( 42 ==  d );
                                   |    }
}                                  |  }
```

Consume – release memory order

- Operations preceding the store-release in the releasing thread **happens before** an operation X in the consuming thread where X has a **data dependency** on the loaded value

```
int d = 0;
std::atomic<int *> ptr = std::nullptr;

// thread 1                    |  // thread 2
void Produce()                 |  void Consume()
{                              |  {
  d = 42                       |    int *p;
  ptr.store(&d,                |
   memory_order_release);      |    if (p= ptr.load(memory_order_consume))
                               |    {
                               |      assert ( 42 == *p );
                               |
                               |      assert ( 42 ==  d );
                               |    }
}                              |  }
```

Consume – release memory order

- Operations preceding the store-release in the releasing thread **happens before** an operation X in the consuming thread where X has a **data dependency** on the loaded value

```
int d = 0;
std::atomic<int *> ptr = std::nullptr;

// thread 1                          |  // thread 2
void Produce()                       |  void Consume()
{                                    |  {
  d = 42                             |    int *p;
  ptr.store(&d,                      |
   memory_order_release);            |    if (p= ptr.load(memory_order_consume))
                                     |    {
                                     |      assert ( 42 == *p );  // ok
                                     |
                                     |      assert ( 42 ==  d );
                                     |    }
}                                    |  }
```

Consume – release memory order

- Operations preceding the store-release in the releasing thread **happens before** an operation X in the consuming thread where X has a **data dependency** on the loaded value

```
int d = 0;
std::atomic<int *> ptr = std::nullptr;

// thread 1                          |  // thread 2
void Produce()                       |  void Consume()
{                                    |  {
  d = 42;                            |    int *p;
  ptr.store(&d,                      |
   memory_order_release);            |    if (p= ptr.load(memory_order_consume))
                                     |    {
                                     |      assert ( 42 == *p );  // ok
                                     |
                                     |      assert ( 42 ==  d );  // DATA RACE
                                     |    }
}                                    |  }
```

std::thread

```
class thread
{
public:
  typedef native_handle ...;
  typedef id ...;

  thread() noexept;                 // does not represent a thread
  thread( thread&& other) noexept; // move constructor
  ~thread();                       // if joinable() calls std::terminate()

  template <typename Function, typename... Args> // copies args to thread local
  explicit thread( Function&& f, Arg&&... args); // then execute f with args

  thread(const thread&) = delete;  // no copy
  thread& operator=(thread&& other) noexept;  // move
  void swap( thread& other);  // swap

  bool joinable() const; // thread object owns a physical thread
  void join();           // blocks current thread until *this finish
  void detach();         // separates physical thread from the thread object

  std::thread::id get_id() const;             // std::this_thread
  static unsigned int hardware_concurrency();  // supported concurrent threads
  native_handle_type native_handle();          // e.g. thread id
};
```

Usage of std::thread

```
void f( int i, const std::string&);
{
    std::cout << "Hello concurrent world" << std::endl;
}

int main()
{
    int i = 3;
    std::string s("Hello");

    // Will copy both i and s
    // We can prevent the copy by using reference wrapper
    // std::thread t( f, std::ref(i), std::ref(s));
    std::thread t( f, i, s);

    // if the thread destructor runs and the thread is joinable, than
    // std::system_error will be thrown.
    // Use join() or detach() to avoid that.
    t.join();

    return 0;
}
```

Issue with join()

- If the thread destructor called when the thread is still *joinable* std::system_error will be thrown
- Alternatives are not really feasible:
- Implicit join:
  - The destructor waits until the thread execution is completed
  - Hard-to detect performance issues
- Implicit detach
  - The destructor may run, but the underlying thread is still under execution
  - We may destroy resources still used by the thread
- Scoped_thread or thread_strategy parameters

Usage of std::thread

```
struct func
{
    int& i;
    func(int& i_) : i (i_) { }

    void operator()()
    {
        for(unsigned int j=0; j < 1000000; ++j)
        {
            do_something(i);  // i refers to a destroyed variable
        }
    }
};

void oops()
{
    int some_local_state=0;

    func my_func(some_local_state);

    std::thread my_thread(my_func);

    my_thread.detach();  // don't wait the thread to finish
}  // i is destroyed, but the thread is likely still running..
```

std::thread works with containers

```
void do_work(unsigned id);

void f()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.push_back(std::thread(do_work,i));
    }
    std::for_each(threads.begin(),threads.end(),
                  std::mem_fn(&std::thread::join));
}
```

std::jthread (C++20)

```
class jthread
{
public:
  typedef native_handle ...;
  typedef id ...;

  jthread() noexept;                // does not represent a thread
  jthread( thread&& other) noexept; // move constructor
  ~jthread();                // if joinable() a stop requested and the thread joins

  template <typename Function, typename... Args>    // copies args to thread local
  explicit jthread( Function&& f, Arg&&... args);   // then execute f with args

  jthread(const jthread&) = delete;             // no copy
  jthread& operator=(jthread&& other) noexept; // move
  void swap( jthread& other);  // swap

  bool joinable() const; // thread object owns a physical thread
  void join();           // blocks current thread until *this finish
  void detach();         // separates physical thread from the thread object

  std::stop_source get_stop_source() noexcept;
  std::stop_token  get_stop_token() noexcept;
  bool request_stop() nexcept;

  std::jthread::id get_id() const;             // std::this_jthread::id
  static unsigned int hardware_concurrency();  // supported concurrent threads
  native_handle_type native_handle();          // e.g. thread id
};
```

Usage of std::jthread

```
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);


    std::cout << "main() exiting ..." << '\n';
```

Usage of std::jthread

```
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);


    std::cout << "main() exiting ..." << '\n';
} // destructor of jthread t calls request_stop() and join().
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
main() exiting ...
Stop requested
```

Usage of std::jthread

```
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);

    t.join();    // waiting the end of jthread t
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);

    t.join();    // waiting the end of jthread t
    std::cout << "main() exiting ..." << '\n';
}
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
15. working
16. working
17. working
18. working
19. working
20. working
21. working
22. working
23. working
24. working
...
```

Usage of std::jthread

```
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    t.join();     // waiting the end of jthread t
    std::cout << "main() exiting ..." << '\n';
}
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    t.join();     // waiting the end of jthread t
    std::cout << "main() exiting ..." << '\n';
}
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
Stop requested
main() exiting ...
```

Usage of std::jthread

```
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    // t.join();
    std::cout << "main() exiting ..." << '\n';
} // destructor of jthread join()
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    // t.join();
    std::cout << "main() exiting ..." << '\n';
} // destructor of jthread join()
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
main() exiting ...
Stop requested
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    std::this_thread::sleep_for(500ms); // wait before print
    std::cout << "main() exiting ..." << '\n';
} // destructor of jthread join()
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);
    std::this_thread::sleep_for(3s);
    t.request_stop();   // send stop request to t
    std::this_thread::sleep_for(500ms); // wait before print
    std::cout << "main() exiting ..." << '\n';
} // destructor of jthread join()
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
Stop requested
main() exiting ...
```

Usage of std::jthread

```cpp
#include <thread>
#include <chrono>
#include <string>
#include <iostream>

using namespace std::literals::chrono_literals;

void f(std::stop_token st, int i, const std::string& s)
{
    while (!st.stop_requested())
    {
        std::cout << i++ << ". " << s << '\n' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    // std::cout << "Stop requested" << std::endl;
}

int main()
{
    int i = 0;
    std::string s{"working"};
    std::jthread t(f, i, s);

    std::stop_callback cb{t.get_stop_token(), []() {
        std::cout << "Stop requested" << std::endl; }};

    std::this_thread::sleep_for(3s);
    t.request_stop();
    std::this_thread::sleep_for(500ms);

    std::cout << "main() exiting ..." << '\n';
}
```

```
0. working
1. working
2. working
3. working
4. working
5. working
6. working
7. working
8. working
9. working
10. working
11. working
12. working
13. working
14. working
Stop requested
main() exiting ...
```

std::(j)thread works with containers

```
// std::thread::id identifiers returned by std::this_thread::get_id()
// it returns std::thread::id() if there is no associated thread.
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
         do_master_thread_work();
    }
    do_common_work();
}


// gives a hint about the available cores. Be aware of
// “oversubscription", i.e. using more threads than cores we have.
std::thread::hardware_concurency()
```

Syncronization objects: mutex

```
#include <mutex>

void f()
{
    std::mutex m;
    int sh;  // shared data
    // ...
    m.lock();
    // manipulate shared data:
    sh+=1;
    m.unlock();
}


void g()
{
    std::mutex m;
    int sh;  // shared data
    // ...
    if ( m.try_lock() )
    {
      // manipulate shared data:
      sh+=1;
      m.unlock();
    }
}
```

```
// Recursive mutex
std::recursive_mutex m;
int sh;  // shared data

void h(int i)
{
    // ...
    m.lock();
    // manipulate shared data:
    sh+=1;
    if (--i>0) f(i);
    m.unlock();
    // ...
}
```

Syncronization objects: timed mutex

```
void f1()
{
    std::timed_mutex m;
    int sh;  // shared data
    // ...
    if (m.try_lock_for(std::chrono::seconds(10)))
    {
      // manipulate shared data:
      sh+=1;
      m.unlock();
    }
    else
      // we didn't get the mutex; do something else
}
void f2()
{
    std::timed_mutex m;
    int sh;  // shared data
    // ...
    if (m.try_lock_until(midnight))
    {
      // manipulate shared data:
      sh+=1;
      m.unlock();
    }
    else
      // we didn't get the mutex; do something else
}
```

RAII support

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> l;
std::mutex     m;

void add_to_list(int value);
{
    // lock acquired - with RAII style lock management
    std::lock_guard< std::mutex > guard(m);
    l.push_back(value);
}   // lock released
```

Pointers or references pointing out from the guarded area may be an issue!

Can this go dead-locked?

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
      return false;

    std::lock_guard< std::mutex > guard(lhs.m)
    std::lock_guard< std::mutex > guard(rhs.m)

    return lhs.data < rhs.data;
}
```

Can this go dead-locked?

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
      return false;

    std::lock_guard< std::mutex > guard(lhs.m)
    std::lock_guard< std::mutex > guard(rhs.m)

    return lhs.data < rhs.data;
}



// thread1             |           thread2

    a < b              |            b < a
```

Correct solution

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
        return false;

    // std::lock - lock two or more mutexes
    std::lock( lhs.m, rhs.m);
    std::lock_guard< std::mutex >  lock_lhs( lhs.m, std::adopt_lock);
    std::lock_guard< std::mutex >  lock_rhs( rhs.m, std::adopt_lock);

    return lhs.data < rhs.data;
}

// attempts to lock in unspecified order
template <class Lockable1, class Lockable2, class Lockable3, ...>
void std::lock( Lockable1 m1, Lockable2 m2, Lockable3 m3, ...);

// attempts to lock in left-to-right order
// returns -1 on success, otherwise the index of first failed
template <class Lockable1, class Lockable2, class Lockable3, ...>
int std::try_lock( Lockable1 m1, Lockable2 m2, Lockable3 m3, ...);
```

Unique_lock with defer_lock

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
        return false;

    // std::unique_locks constructed with defer_lock can be locked
    // manually, by using lock() on the lock object ...
    std::unique_lock< std::mutex >  lock_lhs( lhs.m, std::defer_lock);
    std::unique_lock< std::mutex >  lock_rhs( rhs.m, std::defer_lock);
    // lock_lhs.owns_lock() now false

    // ... or passing to std::lock
    std::lock( lock_lhs, lock_rhs);  // designed to avoid dead-lock
    // also there is an unlock() memberfunction

    // lock_lhs.owns_lock() now true
    return lhs.data < rhs.data;
}
```

Unique_lock only moveable

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk; // same as std::move(lk),
               // return does not require std::move
}

void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

Shared_lock in C++14

```
std::shared_timed_mutex m;
my_data d;

void reader()
{
    std::shared_lock<std::shared_timed_mutex>  rl(m);
    read_only(d);
}

void writer()
{
    std::lock_guard<std::shared_timed_mutex>   wl(m);
    write(d);
}
```

```
Use of shared_timed_mutex may have worse performance
```

Mutex management

```
lock_guard
   C++11: Simple scoped wrapper around a mutex
          Non-copyable, non-movable

unique_lock
   C++11: Simple scoped wrapper around a mutex
          Non-copyable,
          Movable: unique_lock(unique_lock&&) operator=(unique_lock&&)
          unlock()

shared_lock
   C++14: lock the mutex in shared mode e.g shared_timed_mutex (c++14)
          Non-copyable, movable

scoped_lock
   C++17: variadic template class RAII to own one or more mutexes
          Non-copyable, owning multiple mutexes with std::lock()
```

Concurrent singleton

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        std::call_once( resource_init_flag, init_resource);
        return resource_ptr;
    }
private:
    void init_resource()
    {
        resource_ptr.reset( new T(...) );
    }
    std::shared_ptr<T> resource_ptr;
    std::once_flag     resource_init_flag; // can't be moved or copied
};
```

Meyers singleton

```
// Meyers singleton:
// C++11 guaranties: local static is initialized in a thread safe way
//
class MySingleton;
MySingleton& MySingletonInstance()
{
    static MySingleton _instance;
    return _instance;
}
```

Spin lock

```
bool flag;   // waiting for this flag
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

Condition variable

```
std::mutex                  my_mutex;
std::queue< data_t >        my_queue;
std::conditional_variable data_cond;  // conditional variable

void producer()
{
    while ( more_data_to_produce() )
    {
        const data_t data = produce_data();
        std::lock_guard< std::mutex > prod_lock(my_mutex);  // guard the push
        my_queue.push(data);
        data_cond.notify_one(); // notify the waiting thread to evaluate cond.
    }
}

void consumer()
{
    while ( true )
    {
        std::unique_lock< std::mutex > cons_lock(my_mutex); // not lock_guard
        data_cond.wait(cons_lock,                         // returns if lamdba returns true
                [&my_queue]{return !my_queue.empty();}); // else unlocks and waits
        data_t data = my_queue.front();  // lock is hold here to protect pop...
        my_queue.pop();
        cons_lock.unlock();              // ... until here
        consume_data(data);
    }
}
```

Condition variable

- During the wait the condition variable may check the condition any time
- But under the protection of the mutex and returns immediately if condition is true.

- Spurious wake: wake up without notification from other thread. Undefined times and frequency -> better to avoid functions with side effect (e.g. using a counter in lambda to check how many notifications were is bad)

Condition variable

```
std::mutex                my_mutex;
std::queue< data_t >      my_queue;
std::conditional_variable data_cond;  // conditional variable

void producer()
{
    while ( more_data_to_produce() )
    {
        const data_t data = produce_data();
        { // more optimal: release lock before notify
          std::lock_guard< std::mutex > prod_lock(my_mutex);  // guard the push
          my_queue.push(data);
        }
        data_cond.notify_one(); // notify the waiting thread to evaluate cond.
    }
}

void consumer()
{
    while ( true )
    {
        std::unique_lock< std::mutex > cons_lock(my_mutex); // not lock_guard
        data_cond.wait(cons_lock,                          // returns if lamdba returns true
                   [&my_queue]{return !my_queue.empty();}); // else unlocks and waits
        data_t data = my_queue.front();  // lock is hold here to protect pop...
        my_queue.pop();
        cons_lock.unlock();                  // ... until here
        consume_data(data);
    }
}
```

Latch

- A single-use semaphore-like construct (atomic modification)
- Set a downward counter (no reset, no increment)
- May block until counter goes to zero

```
Need example

}
```

Barrier

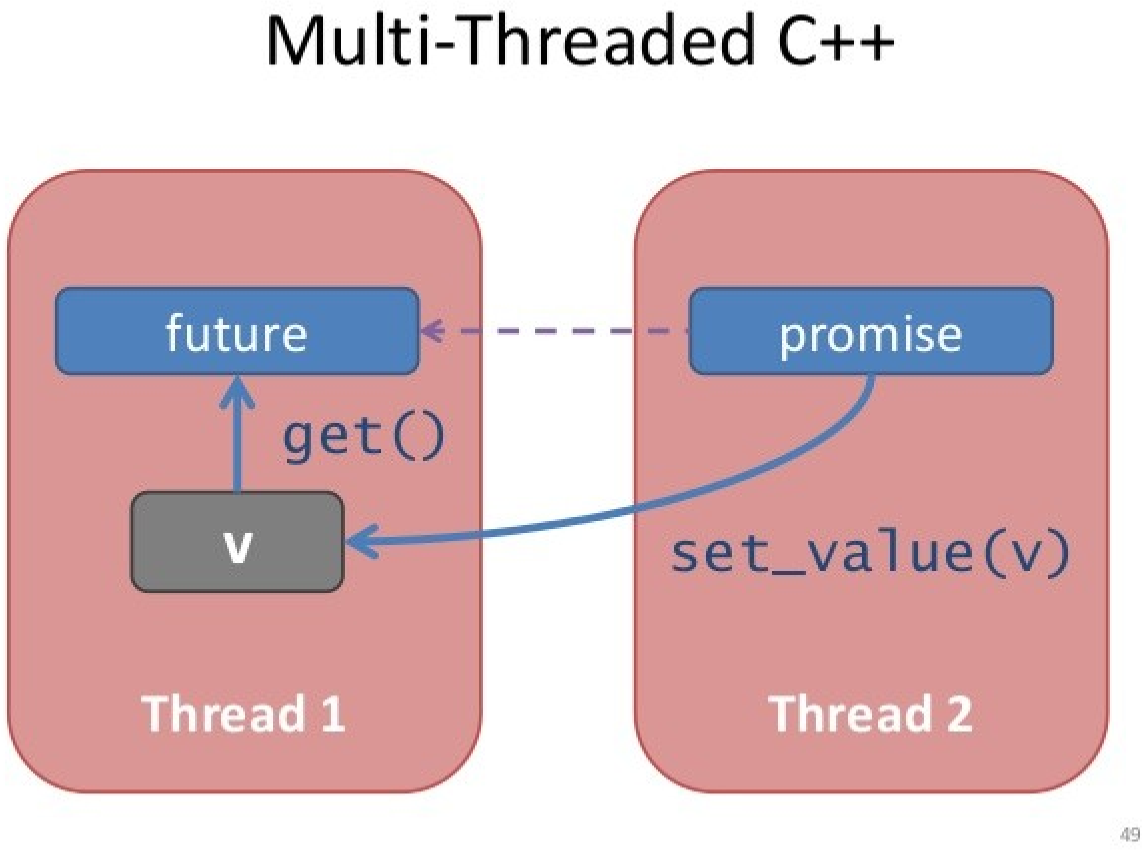
- A reusable semaphore-like construct (atomic modification)
- Set a downward counter (no reset, no increment)
- May block until counter goes to zero
- When all threads reached the barrier, the barrier can be re-used

```
Snedd example
```

Future

- 1976 Daniel P. Friedman and David Wise: promise
- 1977 Henry Baker and Carl Hewitt: future
- Future: a read-only placeholder view of a variable or exception
- Promise: a writeable, single assignment container (to set the future)
- Communication channel: promise → future
- std::future the
  - Only instance to refer the async event
  - Move-only
- std::shared_future
  - Multiple instances referring to the same event
  - Copiable
  - All instances will be ready on the same time

Future-Promise



std::async

```
#include <future>
#include <iostream>

int f(int);
void do_other_stuff();

int main()
{
    std::future<int> the_answer = std::async(f,1);
    do_other_stuff();
    std::cout<< "The answer is " << the_answer.get() << std::endl;
}

// The std::async() executes the task either in a new thread or on get()

// starts in a new thread
auto fut1 = std::async(std::launch::async, f, 1);
// run in the same thread on wait() or get()
auto fut2 = std::async(std::launch::deferred, f, 2);
// default: implementation chooses
auto fut3 = std::async(std::launch::deferred | std::launch::async, f, 3);
// default: implementation chooses
auto fut4 = std::async(f, 4);

// If no wait() or get() is called, then the task may not be executed at all.
```

std::async

```
// from cppreference.com
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int main()
{
    std::future<int> future = std::async(std::launch::async, [](){
        std::this_thread::sleep_for(std::chrono::seconds(3));
        return 8;
    });

    std::cout << "waiting...\n";
    std::future_status status;
    do {
        status = future.wait_for(std::chrono::seconds(1));
        if (status == std::future_status::deferred) {
            std::cout << "deferred\n";
        } else if (status == std::future_status::timeout) {
            std::cout << "timeout\n";
        } else if (status == std::future_status::ready) {
            std::cout << "ready!\n";
        }
    } while (status != std::future_status::ready);

    std::cout << "result is " << future.get() << '\n';
}
```

Exceptions

```
double square_root(double x)
{
    if ( x < 0 )
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}

int main()
{
    std::future<double> fut = std::async( square_root, -1);
    // do something else...
    double res = fut.get(); // f becomes ready on exception and rethrows
}                           // exception object could be a copy of original
```

Exceptions

```
void asyncFun( std::promise<int> myPromise)
{
    int result;
    try
    {
        // calculate the result
        myPromise.set_value(result);
    }
    catch ( ... )
    {
        myPromise.set_exception(std::current_exception());
    }
}

// In the calling thread:
int main()
{
    std::promise<int> intPromise;
    std::future<int> intFuture = intPromise.getFuture();
    std::thread t(asyncFun, std::move(intPromise));

    // do other stuff here, while asyncFun is working

    int result = intFuture.get();  // may throw MyException
    return 0;
}
```

par_algorithms (C++17)

- Based on Intel's Threading Building Blocks (TBB)
- Extends STL algorithms with execution policy
  - std::execution::seq Sequential execution
  - std::execution::par Parallel execution
  - std::execution::par_unseq Parallel SIMD execution
  - std::execution::unseq Sequential SIMD execution

- These policies are permissions not obligations. Implementation may choose what can be parallelized
- Minimal requirement: forward iterator
- The programmer's task to ensure that element access functions will not cause dead lock or data race
- In case of paralellization and vectorization access must not use any blocking synchronization

Parallel STL

```
// Example from Stroustrup

template<class T, class V>
struct Accum     // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)   // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Parallel STL

```
// Example from cppreference

template<class T, class V>
struct Accum     // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
{
    // non-deterministic if binary_op is not associative or not commutative
    double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
    return res;
}
```

Vectorization

```
std::vector<int> v {1,2, ... };

int sum { std::accumulate(v.begin(), v.end(), 0) };


int sum = 0;
for ( size_t i = 0; i < v.size(); ++i)
{
    sum += v[i];
}



int sum = 0;
for ( size_t i = 0; i < v.size() / 4; i+=4)
{
    sum += v[i] + v[i+1] + v[i+2] + v[i+3];   // most CPU supports this
}
// handle if (v.size()/4) is not 0
```

accumulate() vs reduce()

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  for ( int i = 0; i < 10; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
  }

  long long sum = 0;
  for ( std::size_t i = 0; i < v1.size(); ++i) // summa x^2  x in [0..49]
  {
    sum += v1[i]*v1[i];
  }

  std::cout << sum << '\n';

  return 0;
}


$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>


std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 10; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});  // creates 50 elements
  }

  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);  // classical STL



  std::cout << sum1 << '\n';
  return 0;
}
```

```
$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>


std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 10; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});  // creates 50 elements
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);  // classical STL



  std::cout << sum1 << '\n';
  return 0;
}
```

```
$ ./a.out
300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 10; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  // reduce can work parallel
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

  std::cout << sum1 << ", " << sum2 << '\n';
  return 0;
}
```

```
$ ./a.out
300, 300
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 1000; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  // reduce can work parallel
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

  std::cout << sum1 << ", " << sum2 << '\n';
  return 0;
}
```

```
$ ./a.out
30000, 30000
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 1000000; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  // reduce can work parallel
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

  std::cout << sum1 << ", " << sum2 << '\n';
  return 0;
}
```

```
$ ./a.out
30000000, 59820950156796
```

accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };  // not commutative

int main()
{
  for ( int i = 0; i < 1000000; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  // reduce can work parallel
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

  std::cout << sum1 << ", " << sum2 << '\n';
  return 0;
}
```

```
$ ./a.out
30000000, 59820950156796
```

accumulate() vs reduce()

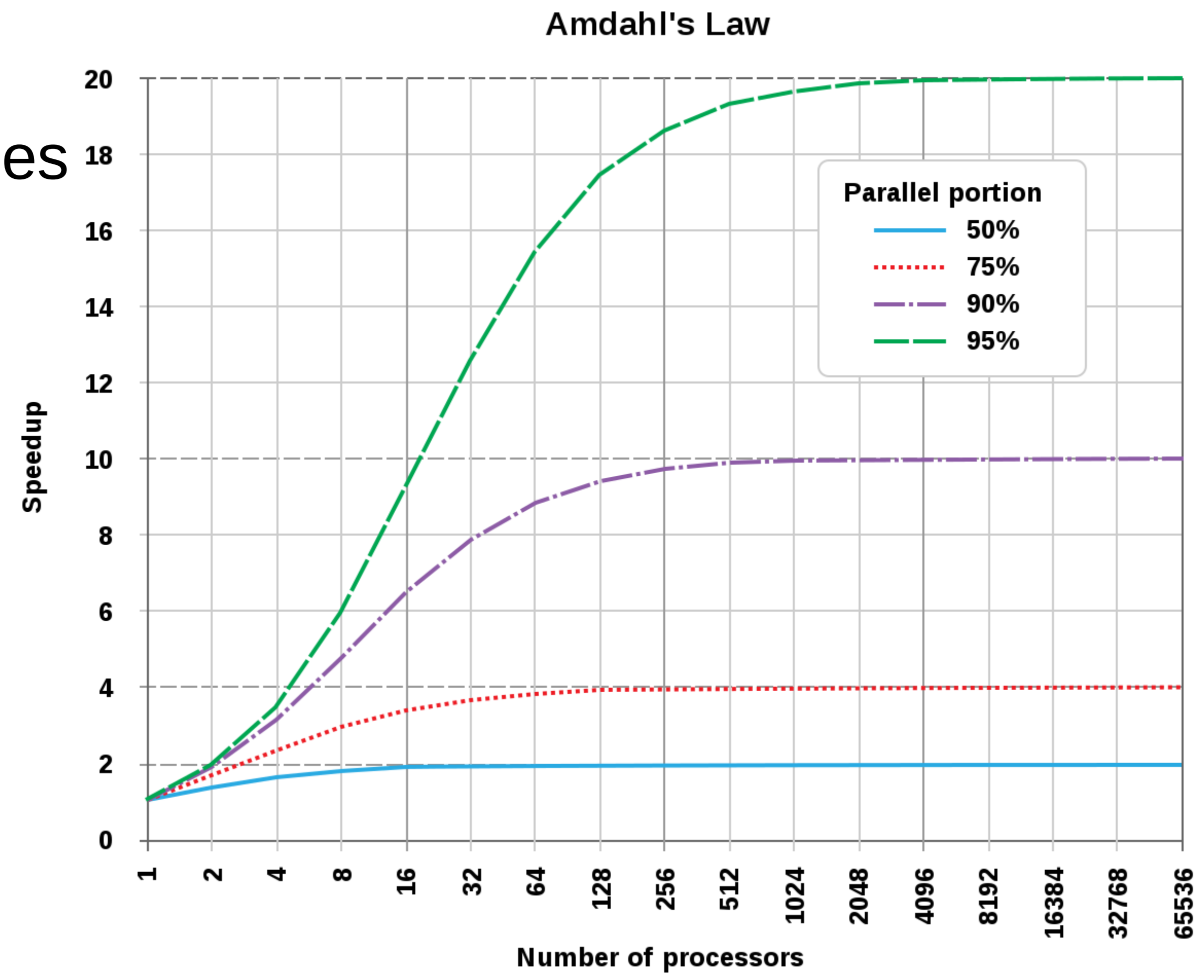
```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>
#include <functional>
std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
  for ( int i = 0; i < 1000000; ++i)
  {
    v1.insert( v1.end(), {0,1,2,3,4});
  }
  // accumulate is guaranteed left associative
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::transform_reduce(std::execution::par,  // map-reduce
             v1.begin(), v1.end(), 0LL,
             std::plus<>(),
             [](auto v) { return v*v; });
  std::cout << sum1 << ", " << sum2 << '\n';
  return 0;
}
```

```
$ ./a.out
30000000, 30000000
```

Lock free programming

- Mutexes are implemented by OS features
- Improper use of locks can cause deadlock
- Locking/unlocking can cause context switch
  - Clear the cache
  - Further cache-misses
- Amdahl law
- Blocking vs non-blocking



Lock free programming

- Non-Blocking
- Lock-free
  - Non-blocking, some thread(s) may progress
  - Starvation can happen
- Wait-free
  - Lock-free, all threads are progressing
  - Every operation can be performed in a single pass
  - Steps does not cause any operations to fail

Atomic operations

- Supports lock-less concurrent programming
- Non-interleaving read and write operations (no data race)
- May define inter-thread synchronization (based on memory model)
- The encapsulated type must be trivially constructible, copy- and movable.

```
#include <atomic>
template<typename T> struct atomic; // atomic class template since C++11
template<> struct atomic<Integral>; // specialization for integral types C++11
template<> struct atomic<bool>;    // specialization for bool C++11
template<typename T> struct atomic<T*>;  // specialization for pointer C++11

template<> struct atomic<FloatingPoint>; // from C++20

#include <memory>
template<typename T> struct atomic<std::shared_ptr<T>>; // C++20 shared_ptr
template<typename T> struct atomic<std::weak_ptr<T>>;   // C++20 weak_ptr
```

Atomic operations

- Default constructor does not initialize
- Atomic are not copyable, not movable.

```
#include <atomic>

std::atomic<long>  al1;       // default constructor, do not initialize
std::atomic<long>  al2{0L};   // likely better

al1.store(42L); // atomic store
al2 = 43L;      // overloaded operator=(), same as store

long l1 = al1.load();  // atomic load
long l2 = al2;         // overloaded operator long(), same as load

long i3 = al1.exchange(l2);   // write l2 to al1, return the previos value

long expected = al1.load();
long desired  = expected+1;
if ( al1.compare_exchange_strong(expected, desired) )
    return; // if al1 == expected then al1 := desired; return true, else false

while (!al1.compare_exchange_weak(expected, desired)) ; // may fail spuriously
```

Atomic operations

- Specializations for Integral types, e.g. atomic<int>

```
#include <atomic>

std::atomic<int>   ai;

                 ++ai;
                 --ai;
                   ai++;
                   ai--;

                   ai += 42;
                   ai -= 42;
                   ai &= 42;
                   ai |= 42;
                   ai ^= 42;

                   ai.fetch_add(42);  // returns prevoius value
                   ai.fetch_sub(42);  // returns prevoius value
                   ai.fetch_and(42);  // returns prevoius value
                   ai.fetch_or (42);  // returns prevoius value
                   ai.fetch_xor(42);  // returns prevoius value
```

Atomic operations

- Specializations for Floating point types, e.g. atomic<double>
  - Since C++20
  - T operator+=, etc…
  - Compare-exchange exists, but be careful, it is bitwise !
  - T fetch_add(), etc.
- Specializations for pointers
  - T* operator+=(std::ptrdiff_t), etc.
  - T* fetch_add(), etc.

Atomic operations

- is_lock_free is per instance()
- is_always_lock_free() since C++17

```
#include <atomic>

class X { ... };

std::atomic<int>  a1;
std::atomic<int>  a2;

if (a1.is_lock_free() && !a2.is_lock_free()) // may be true on some platforms

if ( std::atomic<int>::is_always_lock_free() ) // likely true

if ( std::atomic<X>::is_always_lock_free() ) // likely false, static,constexpr


a1.notify_one()    // unblocks at least one thread waiting for, C++20
a1.notify_all()    // unblocks all threads to waiting for, C++20
a1.wait(42);       // waits until the value changes, and notified, C++20
```

Atomic operations

```
// read-modify-write
std::atomic<int> x;

int i = x.load();                                  // current value of x
while( !x.compare_exchange_weak( i, // expected value of x, i updated on fail
                                 i+1, // desired value of x
                                 memory_order_relaxed
                               )
     ) ; // try it again on failure

x.fetch_add( 1, memory_order_relaxed); // equivalent solution



std::.atomic<double>   ad;

// thread1
ad.store(3.14);        // compute value and store in ad

// thread2

for ( auto &v : vec )  // increase all elements by ad
{
    v *= ad.load();    // increase all elements
}
```

Atomic operations

```
// read-modify-write
std::atomic<int> x;

int i = x.load();                                  // current value of x
while( !x.compare_exchange_weak( i, // expected value of x, i updated on fail
                                 i+1, // desired value of x
                                 memory_order_relaxed
                               )
     ) ; // try it again on failure

x.fetch_add( 1, memory_order_relaxed); // equivalent solution



std:.atomic<double>   ad;

// thread1
ad.store(3.14);        // compute value and store in ad

// thread2

for ( auto &v : vec )  // increase all elements by ad
{
    v *= ad.load();    // inefficient and ad may change inside the loop
}
```

Atomic operations

```
// read-modify-write
std::atomic<int> x;

int i = x.load();                                  // current value of x
while( !x.compare_exchange_weak( i, // expected value of x, i updated on fail
                                 i+1, // desired value of x
                                 memory_order_relaxed
                               )
     ) ; // try it again on failure

x.fetch_add( 1, memory_order_relaxed); // equivalent solution



std:.atomic<double>   ad;

// thread1
ad.store(3.14);       // compute value and store in ad

// thread2
double d = ad.load();
for ( auto &v : vec )  // increase all elements by ad
{
    v *= d;    // instead of ad.load()
}
```

Atomic operations

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

using namespace std::literals;
int main()
{
    const int ntasks = 10;
    std::atomic<bool> all_completed{false};
    std::atomic<int> count{0};
    std::vector<std::future<void>> tasks;

    for (int i = 0; i < ntasks; ++i)   // spawn tasks
    {
        tasks.push_back(std::async([&]
        {
            std::this_thread::sleep_for(50ms); // Sleep represents doing real work...
            ++count;

            if (count.load() == ntasks)  // check all tasks finished
            {
                all_completed = true;
                all_completed.notify_one();    // notify main thread
            }
        }));
    }
    all_completed.wait(false);  // wait until value changed
    std::cout << "Tasks completed = " << count.load() << '\n';
}
```

Atomic operations

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

struct X
{
    // big enought not to be atomic
};

std::atomic<X>    data{ ... };      // works, but likely no lock-free
std::atomic<X*>   ptr{nullptr};     // pointers are usually lock-free

void update()     // continously sending data to use()
{
    X *newX = new X { ... };  // heap operation may block
    // update *newX

    X *oldX = ptr.exchange(newX);

    delete oldX;
}

void use()        // use the data sent by update()
{
    X *currX = ptr.exchange(nullptr); // only one owner should own the object
    // use currX

    delete currX;
}
```

Atomic_ref

- Light_weight wrapper around buildin globals
  - Cheap to create on demand
  - Be sure the right alignment

```
#include <iostream>
#include <atomic>

struct foo {
    double a;
    double b;
};
std::atomic<foo> var;

int main()
{
    std::cout << var.is_lock_free() << std::endl;
    std::cout << sizeof(foo) << std::endl;
    std::cout << sizeof(var) << std::endl;
}
$ ./a.out
```

```
$ g++ lock.cpp -std=c++20 -lpthread
/usr/bin/ld: /tmp/cc4ETaLh.o: in function
`std::atomic<foo>::is_lock_free() const':
lock.cpp:
(.text._ZNKSt6atomicI3fooE12is_lock_freeEv[_ZNKSt
6atomicI3fooE12is_lock_freeEv]+0x1d): undefined
reference to `__atomic_is_lock_free'
collect2: error: ld returned 1 exit status
```

Atomic_ref

- Light_weight wrapper around buildin globals
  - Cheap to create on demand
  - Be sure the right alignment

```
#include <iostream>
#include <atomic>

struct foo {
    double a;
    double b;
};
std::atomic<foo> var;

int main()
{
    std::cout << var.is_lock_free() << std::endl;
    std::cout << sizeof(foo) << std::endl;
    std::cout << sizeof(var) << std::endl;
}
$ ./a.out
0
16
16
```

```
$ g++ --version
g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
...
$ g++ lock.cpp -std=c++20 -lpthread -latomic
$
```

Atomic_ref

- Light_weight wrapper around buildin globals
  - Cheap to create on demand
  - Be sure the right alignment

```
#include <iostream>
#include <atomic>

struct foo {
    double a;
    double b;
};
std::atomic<foo> var;

int main()
{
    std::cout << var.is_lock_free() << std::endl;
    std::cout << sizeof(foo) << std::endl;
    std::cout << sizeof(var) << std::endl;
}
$ ./a.out
0
16
16
```

Atomic_ref

- Light_weight wrapper around buildin globals
  - Cheap to create on demand
  - Be sure the right alignment

```
#include <iostream>
#include <atomic>

struct foo {
    double a;
    double b;
};
std::atomic<foo> var;

int main()
{
    std::cout << var.is_lock_free() << std::endl;
    std::cout << sizeof(foo) << std::endl;
    std::cout << sizeof(var) << std::endl;
}
$ ./a.out
0
16
16
```

WHERE IS THE LOCK?

Lock-free queue

```
// Timur Doumler Accu 2017
// https://www.youtube.com/watch?v=qdrp6k4rcP4

template <typename T, size_t size>
class LockFreeQueue   // Single reader, single writer, lock-free + wait-free
{
public:
  bool pop(T& ret)
  {
    auto oldWritePos = writePos.load();
    auto oldReadPos  = readPos.load();

    if ( oldWritePos == oldReadPos )
      return false;

    retElem = std::move(ringBuffer[oldReadPos]);

    readPos.store(nextPos(oldReadPos));
    return true;
  }
private:
  static constexpr size_t nextPos(size_t pos) noexcept { return ++pos % ringBufferSize; }
  static constexpr size_t ringBufferSize = size + 1;  // keep one element as separator
  std::array<T,ringBufferSize> ringBuffer; // fix sized to avoid heap, it may blocking
  std::atomic<size_t> readPos{0};
  std::atomic<size_t> writePos{0};
};
```

Timur Doumler, ACCU 2017 https://www.youtube.com/watch?v=qdrp6k4rcP4

Lock-free queue

```
// Timur Doumler Accu 2017
// https://www.youtube.com/watch?v=qdrp6k4rcP4

template <typename T, size_t size>
class LockFreeQueue   // Single reader, single writer, lock-free + wait-free
{
public:
  bool pop(T& ret)                                bool push(T& t)
  {                                               {
    auto oldWritePos = writePos.load();             auto oldWritePos = writePos.load();
    auto oldReadPos  = readPos.load();              auto newWritePos = nextPos(oldwritePos);

    if ( oldWritePos == oldReadPos )                if ( newWritePos == readPos.load() )
      return false;                                   return false;

    retElem = std::move(ringBuffer[oldReadPos]);    ringBuffer[oldWritePos] = t;

    readPos.store(nextPos(oldReadPos));             writePos.store(newWritePos);
    return true;                                    return true;
  }                                               }
private:
  static constexpr size_t nextPos(size_t pos) noexcept { return ++pos % ringBufferSize; }
  static constexpr size_t ringBufferSize = size + 1;  // keep one element as separator
  std::array<T,ringBufferSize> ringBuffer; // fix sized to avoid heap, it may blocking
  std::atomic<size_t> readPos{0};
  std::atomic<size_t> writePos{0};
};
```

Timur Doumler, ACCU 2017 https://www.youtube.com/watch?v=qdrp6k4rcP4

Lock-free queue

```
// Timur Doumler Accu 2017
// https://www.youtube.com/watch?v=qdrp6k4rcP4

template <typename T, size_t size>
class LockFreeQueue   // Multiple reader, lock-free + not wait-free (looping)
{
public:
  bool pop(T& ret)
  {
    auto oldWritePos = writePos.load();
    auto oldReadPos  = readPos.load();

    if ( oldWritePos == oldReadPos )
      return false;

    while (true)
    {
      retElem = ringBuffer[oldReadPos].load();

      if (readPos.compare_exchange_strong(
         oldReadPos, nextPos(oldReadPos)) )
         return true;

      oldReadPos = readPos.load();
    }
  }
  bool push(T& t)
  {
    auto oldWritePos = writePos.load();
    auto newWritePos = nextPos(oldwritePos);

    if ( newWritePos == readPos.load() )
      return false;

    ringBuffer[oldWritePos].store(t);

    writePos.store(newWritePos);
    return true;
  }
private:
  std::array<atomic<T>,ringBufferSize> ringBuffer; // use T* for large objects
};
```

Lock-free queue

```
// Timur Doumler Accu 2017
// https://www.youtube.com/watch?v=qdrp6k4rcP4

template <typename T, size_t size>
class LockFreeQueue   // Multiple reader, lock-free + not wait-free (looping)
{
public:
  bool pop(T& ret)
  {
    while (true)  // detect empty queue
    {
      auto oldWritePos = writePos.load();
      auto oldReadPos  = readPos.load();

      if ( oldWritePos == oldReadPos )
        return false;

      retElem = ringBuffer[oldReadPos].load();

      if (readPos.compare_exchange_strong(
        oldReadPos, nextPos(oldReadPos)) )
        return true;
    }
  }
private:
  std::array<atomic<T>,ringBufferSize> ringBuffer;
};
```

```
bool push(T& t)
{
  auto oldWritePos = writePos.load();
  auto newWritePos = nextPos(oldwritePos);

  if ( newWritePos == readPos.load() )
    return false;

  ringBuffer[oldWritePos].store(t);

  writePos.store(newWritePos);
  return true;
}
```

ABA problem

```
// https://en.wikipedia.org/wiki/ABA_problem
class Stack {
  std::atomic<Obj*> top_ptr;

  Obj* pop() {
    while (1) {
      Obj* ret_ptr = top_ptr;
      if (ret_ptr == nullptr) return nullptr;
      // For simplicity, suppose that we can ensure that this dereference is safe
      // (i.e., that no other thread has popped the stack in the meantime).
      Obj* next_ptr = ret_ptr->next;
      // If the top node is still ret, then assume no one has changed the stack.
      if (top_ptr.compare_exchange_weak(ret_ptr, next_ptr)) {
        return ret_ptr;
      }
      // The stack has changed, start over.
    }
  }
  void push(Obj* obj_ptr) {
    while (1) {
      Obj* next_ptr = top_ptr;
      obj_ptr->next = next_ptr;
      // If the top node is still next, then assume no one has changed the stack.
      // Atomically replace top with obj.
      if (top_ptr.compare_exchange_weak(next_ptr, obj_ptr)) {
        return;
      }
      // The stack has changed, start over.
    }
  }
};
```

ABA problem

```
// https://en.wikipedia.org/wiki/ABA_problem
class Stack {
  std::atomic<Obj*> top_ptr;

  Obj* pop() {
    while (1) {
      Obj* ret_ptr = top_ptr;
      if (ret_ptr == nullptr) return nullptr;
      // For simplicity, suppose that we can ensure that this dereference is safe
      // (i.e., that no other thread has popped the stack in the meantime).
      Obj* next_ptr = ret_ptr->next;
      // If the top node is still ret, then assume no one has changed the stack.
      if (top_ptr.compare_exchange_weak(ret_ptr, next_ptr)) {
        return ret_ptr;
      }
      // The stack has changed, start over.
    }
  }
  void push(Obj* obj_ptr) {
    while (1) {
      Obj* next_ptr = top_ptr;
      obj_ptr->next = next_ptr;
      // If the top node is still next, then assume no one has changed the stack.
      // Atomically replace top with obj.
      if (top_ptr.compare_exchange_weak(next_ptr, obj_ptr)) {
        return;
      }
      // The stack has changed, start over.
    }
  }
};
```

C++20

- resumable functions
  - async ... wait
- continuation
  - then()
  - when_any()
  - when_all()
- transactional memory – ???

- Critics on C++ concurrency:

  Bartosz Milewski's blog: Broken promises - C++0x futures
  http://bartoszmilewski.com/2009/03/03/broken-promises-c0x-futures/

  MeetingC++ - Hartmut Kaiser: Plain Threads are the GOTO of todays computing https://www.youtube.com/watch?v=4OCUEgSNIAY