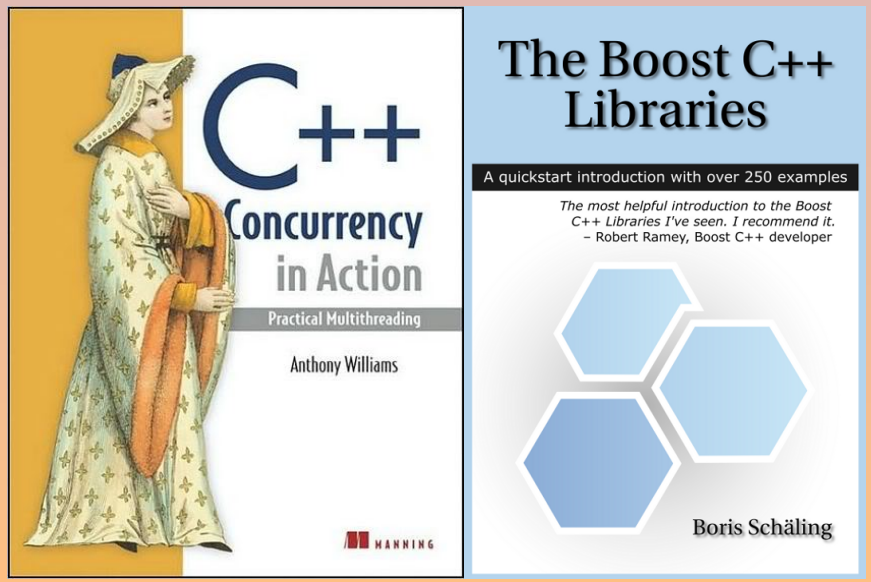
Concurrency in C++
Part I

Akim Demaille akim@lrde.epita.fr

May, 14th 2013

(2016-11-16 09:53:54 +0100 121bea3)



Concurrency in C++
Part I

Concurrency

Concepts

Concurrency, Parallelism [Pike, 2012]

- Concurrency
  - A programming model
  - Aims at clarity, not efficiency (nice by-product)
  - Go, urbiscript, etc.
- Parallelism
  - Simultaneous execution of computations
  - Hyperthreads, Multi-CPU, Multi-core

Concurrency, Parallelism [Pike, 2012]

- Concurrency
  - A programming model
  - Aims at clarity, not efficiency (nice by-product)
  - Go, urbiscript, etc.
- Parallelism
  - Simultaneous execution of computations
  - Hyperthreads, Multi-CPU, Multi-core

Memory Sharing [Milewski, 2011b]

- Processes
  - Each has a separate address space
  - Communication only through special channels
    Messages, sockets, (memory-mapped) files
- Threads
  - Share the same address space within process
  - Can read and write to the same memory address
  - Usually local (stack) variables are considered thread-private
    Beware of closures
  - Concurrent memory access leads to collisions
- Coroutines
  - Aka Green Threads or light-weight threads
  - Dealt with by the language

Memory Sharing [Milewski, 2011b]

- Processes
  - Each has a separate address space
  - Communication only through special channels
    Messages, sockets, (memory-mapped) files
- Threads
  - Share the same address space within process
  - Can read and write to the same memory address
  - Usually local (stack) variables are considered thread-private
    Beware of closures
  - Concurrent memory access leads to collisions
- Coroutines
  - Aka Green Threads or light-weight threads
  - Dealt with by the language

Memory Sharing [Milewski, 2011b]

- Processes
  - Each has a separate address space
  - Communication only through special channels
    Messages, sockets, (memory-mapped) files
- Threads
  - Share the same address space within process
  - Can read and write to the same memory address
  - Usually local (stack) variables are considered thread-private
    Beware of closures
  - Concurrent memory access leads to collisions
- Coroutines
  - Aka Green Threads or light-weight threads
  - Dealt with by the language

Threads vs. Tasks [Milewski, 2011b]

- Multithreading
  - Explicit thread creation and work assignment
  - Improves latency
  - Doesn't scale well
- Task-based Concurrency
  - Partitioning of work for parallel execution: tasks
  - The system, runtime, or library, assigns tasks to threads
  - Improves performance, if done correctly
  - Scales better with the number of cores

Threads vs. Tasks [Milewski, 2011b]

- Multithreading
  - Explicit thread creation and work assignment
  - Improves latency
  - Doesn't scale well
- Task-based Concurrency
  - Partitioning of work for parallel execution: tasks
  - The system, runtime, or library, assigns tasks to threads
  - Improves performance, if done correctly
  - Scales better with the number of cores

Communication Between Threads [Milewski, 2011b]

- Shared memory
  - Threads accessing concurrently shared variables/objects
  - If not synchronized, leads to data races, atomicity violations
  - Synchronization through locks/atomic variables
- Message passing
  - No sharing of memory (hard to enforce between threads)
  - Message Queues, Channels, Mailboxes, Actors
  - Scales up to inter-process and distributed communications (marshaling)
    But usually slower that sharing
  - Within a process, possibility of passing references through messages
    Unintended sharing, races

Communication Between Threads [Milewski, 2011b]

- Shared memory
  - Threads accessing concurrently shared variables/objects
  - If not synchronized, leads to data races, atomicity violations
  - Synchronization through locks/atomic variables
- Message passing
  - No sharing of memory (hard to enforce between threads)
  - Message Queues, Channels, Mailboxes, Actors
  - Scales up to inter-process and distributed communications (marshaling)
    But usually slower that sharing
  - Within a process, possibility of passing references through messages
    Unintended sharing, races

Pitfalls

Races [Milewski, 2011b]

- Conflict
  - two or more threads accessing the same memory location
  - at least one of them writing
  - (others may be reading or writing)
- Data Race
  - A conflict with no intervening synchronization
  - In most cases synchronization by locking (Java synchronized)
    Lock both reads and writes with the same lock
  - Atomic variables (Java volatile) used in lock-free programming
    To be left to gurus

Races [Milewski, 2011b]

- Conflict
  - two or more threads accessing the same memory location
  - at least one of them writing
  - (others may be reading or writing)
- Data Race
  - A conflict with no intervening synchronization
  - In most cases synchronization by locking (Java synchronized)
    Lock both reads and writes with the same lock
  - Atomic variables (Java volatile) used in lock-free programming
    To be left to gurus

Relaxed Memory Models [Milewski, 2011b]

- Processors don't have a consistent view of memory
  - Each processor has a local cache
  - Relaxed guarantees about write propagation between caches
  - Special instructions (lock, memory fences) to control consistency
  - Reflected in modern languages
- Atomic variables (Java volatile)
  - Atomic operations
  - Fences
  - Higher synchronization primitives (locks, etc.) built on top

Relaxed Memory Models [Milewski, 2011b]

- Processors don't have a consistent view of memory
  - Each processor has a local cache
  - Relaxed guarantees about write propagation between caches
  - Special instructions (lock, memory fences) to control consistency
  - Reflected in modern languages
- Atomic variables (Java volatile)
  - Atomic operations
  - Fences
  - Higher synchronization primitives (locks, etc.) built on top

DRF Guarantee [Milewski, 2011b]

- Modern multicores/languages break sequential consistency
- The usual reasoning about threads fails

```
// Initially: x = 0, y = 0;
```

```
 x = 1
r1 = y
```

```
 y = 1
r2 = x
```

- Possible outcome: r1 = 0, r2 = 0 (on x86!)
- The DRF guarantee
  - A data race free program is sequentially consistent
  - True in C++, as long as no "weak atomics"

DRF Guarantee [Milewski, 2011b]

- Modern multicores/languages break sequential consistency
- The usual reasoning about threads fails

```
// Initially: x = 0, y = 0;
```

```
 x = 1
r1 = y
```

```
 y = 1
r2 = x
```

- Possible outcome: r1 = 0, r2 = 0 (on x86!)
- The DRF guarantee
  - A data race free program is sequentially consistent
  - True in C++, as long as no "weak atomics"

DRF Guarantee [Milewski, 2011b]

- Modern multicores/languages break sequential consistency
- The usual reasoning about threads fails

```
// Initially: x = 0, y = 0;
```

```
 x = 1
r1 = y
```

```
 y = 1
r2 = x
```

- Possible outcome: r1 = 0, r2 = 0 (on x86!)
- The DRF guarantee
  - A data race free program is sequentially consistent
  - True in C++, as long as no "weak atomics"

DRF Guarantee [Milewski, 2011b]

- Modern multicores/languages break sequential consistency
- The usual reasoning about threads fails

```
// Initially: x = 0, y = 0;
```

```
 x = 1
r1 = y
```

```
 y = 1
r2 = x
```

- Possible outcome: r1 = 0, r2 = 0 (on x86!)
- The DRF guarantee
  - A data race free program is sequentially consistent
  - True in C++, as long as no "weak atomics"

Risks of Concurrency [Milewski, 2011b]

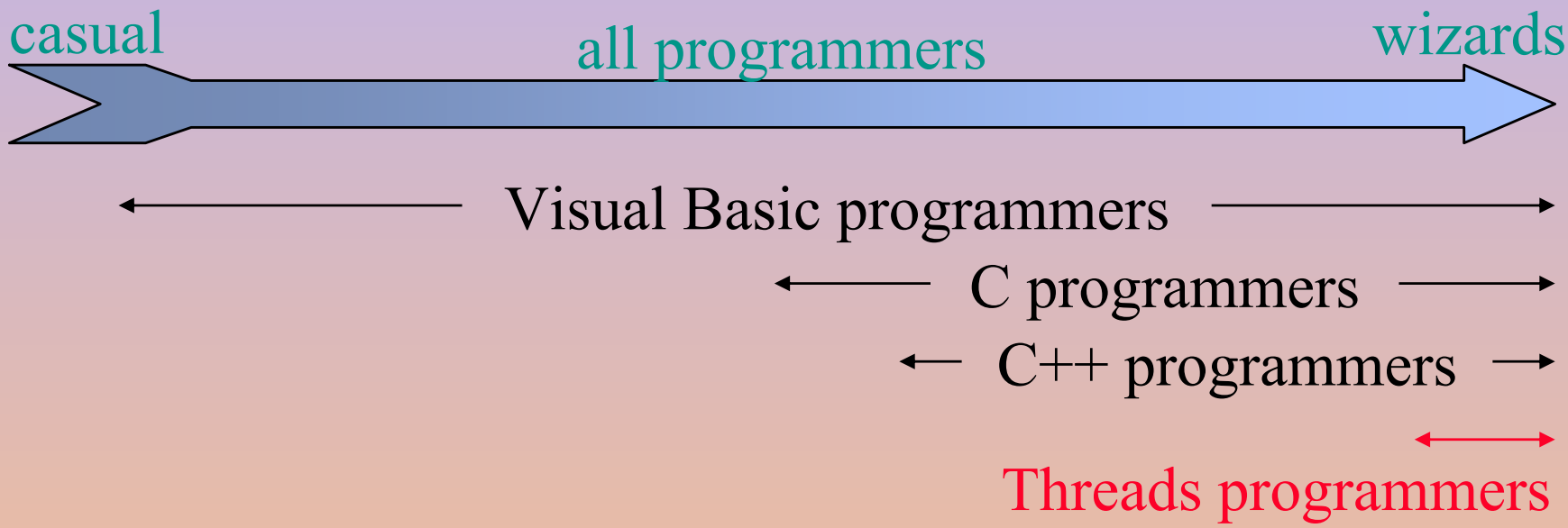
- Exponential number of interleavings
  - Hard to reason about
  - Hard to test exhaustively
- Concurrency bugs (races, atomicity violations, deadlocks)
  - Hard to find
  - Hard to reproduce
  - Hard to find the cause

Risks of Concurrency [Milewski, 2011b]

- Exponential number of interleavings
  - Hard to reason about
  - Hard to test exhaustively
- Concurrency bugs (races, atomicity violations, deadlocks)
  - Hard to find
  - Hard to reproduce
  - Hard to find the cause

Programming with Threads

What's Wrong With Threads?



- **Too hard for most programmers to use.**
- **Even for experts, development is painful.**

Threads

Threads

Threads

Threads

Using Threads

HelHellolo W Woorrlldd!!

Hello World [ccia1.1]

[Williams, 2012, Listing 1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
}
```

Hello World [ccia1.1]

[Williams, 2012, Listing 1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
}
```

```
libc++abi.dylib: terminate called without an active exceptionHello, World!

zsh: abort      ./5-concurrency/hello-world-fail
```

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

```
Hello, World!
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

```
Hello, World!
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

```
Hello, World!
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

```
Hello, World!
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Hello World [ccia1.1]

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

void hello()
{
  std::cout << "Hello, World!\n";
}

int main()
{
  std::thread t{hello};
  t.join();
}
```

```
Hello, World!
```

- `std::thread` is *not* a thread
- it *manages* a thread
- sadly enough, no *RAII* off-the-shelf
- sadlier enougher, no pool off-the-shelf

Passing Arguments to Threads

```
void display(const std::string& s)
{
  std::cout << s << "!\n";
}

void work(int v)
{
  char buf[1024];
  sprintf(buf, "%d + %d = %d",
          v, v, v+v);
  std::thread{display, buf}.detach();
}

int main()
{
  work(21);
  std::chrono::seconds s1{1};
  std::this_thread::sleep_for(s1);
}
```

- WTF???
- (What The Fread???)
- works as expected with `join` instead of `detach`

Passing Arguments to Threads

```
void display(const std::string& s)
{
  std::cout << s << "!\n";
}

void work(int v)
{
  char buf[1024];
  sprintf(buf, "%d + %d = %d",
          v, v, v+v);
  std::thread{display, buf}.detach();
}

int main()
{
  work(21);
  std::chrono::seconds s1{1};
  std::this_thread::sleep_for(s1);
}
```

```
21 + 21 = 42!
```

- WTF???
- (What The Fread???)
- works as expected with `join` instead of `detach`

Passing Arguments to Threads

```
void display(const std::string& s)
{
  std::cout << s << "!\n";
}

void work(int v)
{
  char buf[1024];
  sprintf(buf, "%d + %d = %d",
          v, v, v+v);
  std::thread{display, buf}.detach();
}

int main()
{
  work(21);
  std::chrono::seconds s1{1};
  std::this_thread::sleep_for(s1);
}
```

```
21 + 21 = 42!
```

- WTF???
- (What The Fread???)
- works as expected with `join` instead of `detach`

Passing Arguments to Threads

```
void display(const std::string& s)
{
  std::cout << s << "!\n";
}

void work(int v)
{
  char buf[1024];
  sprintf(buf, "%d + %d = %d",
          v, v, v+v);
  std::thread{display, buf}.detach();
}

int main()
{
  work(21);
  std::chrono::seconds s1{1};
  std::this_thread::sleep_for(s1);
}
```

```
21 + 21 = 42!
```

- WTF???
- (What The Fread???)
- works as expected with `join` instead of `detach`

Passing Arguments to Threads

```
void display(const std::string& s)
{
  std::cout << s << "!\n";
}

void work(int v)
{
  char buf[1024];
  sprintf(buf, "%d + %d = %d",
          v, v, v+v);
  std::thread{display, buf}.detach();
}

int main()
{
  work(21);
  std::chrono::seconds s1{1};
  std::this_thread::sleep_for(s1);
}
```

```
21 + 21 = 42!
```

- WTF???
- (What The Fread???)
- works as expected with `join` instead of `detach`

Threads: Argument Passing

- Arguments are always passed *by value*
- Conversions are performed *in the new thread*
- Use `std::ref` to pass by references
- But then beware of dangling references

APIs

std::thread

```
// Flexible invocation.
template <typename Function, typename... Args>
explicit thread(Function&& f, Args&&... args);

thread();

// Move semantics only.
thread(thread&& other);
thread& operator=(thread&& other);
thread(const thread&) = delete;

// Management.
void join();
void detach();
```

Scoped Threads (<boost/thread/scoped_thread.hpp>)

```
namespace boost
{
  struct detach;
  struct join_if_joinable;
  struct interrupt_and_join_if_joinable;

  template <typename CallableThread = join_if_joinable>
  class strict_scoped_thread;

  template <typename CallableThread = join_if_joinable>
  class scoped_thread;

  void swap(scoped_thread& lhs, scoped_thread& rhs) noexcept;
}
```

- `strict_scoped_thread` hides completely the thread only control: the specific action given as argument
- `scoped_thread` provides the same interface than `thread`

Threads in C [Burelle, 2013]

PThreads

```
int pthread_create(pthread_t *restrict thr,
                   const pthread_attr_t *restrict attr,
                   void *(*func) (void *), void *restrict arg);
int pthread_join(pthread_t thr, void **res_ptr);
int pthread_cancel(pthread_t thr);
```

Threads in C11

```
typedef int (*thrd_start_t)(void *);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
int thrd_join(thrd_t thr, int *res);
int thrd_detach(thrd_t thr);
```

Threads in C [Burelle, 2013]

PThreads

```
int pthread_create(pthread_t *restrict thr,
                   const pthread_attr_t *restrict attr,
                   void *(*func) (void *), void *restrict arg);
int pthread_join(pthread_t thr, void **res_ptr);
int pthread_cancel(pthread_t thr);
```

Threads in C11

```
typedef int (*thrd_start_t)(void *);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
int thrd_join(thrd_t thr, int *res);
int thrd_detach(thrd_t thr);
```

C11 Memory Model

- C99 (and earlier) defines only sequence points
  - operations between two sequence points can happen in any order
  - compilers and processors may reorder the code
- C11/C++11 memory models extend this to concurrency:
  it specifies sequenciality and memory access constraints
  - **Atomicity** must act as one single step
  - **Release Semantic** all past writes have completed and become visible by the time that the release happens
  - **Acquire Semantic** no future reads have started yet so that it will see any writes released by other processors
  - **Memory Fence** combine release and acquire.
  - ...

Using Mutexes

Thread-Safe List

Protecting a List [ccia3.1]

```
std::list<int> list;
std::mutex mutex;

void list_add(int new_value)
{
  std::lock_guard<std::mutex> guard(mutex);
  list.push_back(new_value);
}

bool list_contains(int v)
{
  std::lock_guard<std::mutex> guard(mutex);
  return std::find(begin(list), end(list), v) != list.end();
}
```

- Let's encapsulate this

Protecting a List [ccia3.1]

```
std::list<int> list;
std::mutex mutex;

void list_add(int new_value)
{
  std::lock_guard<std::mutex> guard(mutex);
  list.push_back(new_value);
}

bool list_contains(int v)
{
  std::lock_guard<std::mutex> guard(mutex);
  return std::find(begin(list), end(list), v) != list.end();
}
```

- Let's encapsulate this

Thread-Safe List: Type

```
template <typename T>
class ts_list
{
public:
  using const_iterator = typename std::list<T>::const_iterator;

  auto add(const T& v) -> void;
  auto find(const T& v) const -> const_iterator;
  auto has(const T& v) const -> bool;

  template <typename Func>
  auto for_each(Func f) const -> void;

private:
  std::list<T> list_;
  mutable std::mutex mutex_;
};
```

Thread-Safe List: Functions

```
template <typename T>
auto ts_list<T>::add(const T& v) -> void
{
  std::lock_guard<std::mutex> guard(mutex_);
  list_.push_back(v);
}

template <typename T>
auto ts_list<T>::find(const T& v) const -> const_iterator
{
  std::lock_guard<std::mutex> guard(mutex_);
  return std::find(begin(list_), end(list_), v);
}

template <typename T>
auto ts_list<T>::has(const T& v) const -> bool
{
  return find(v) != list_.end();
}
```

Thread-Safe List: for_each

```
template <typename T>
template <typename Func>
auto ts_list<T>::for_each(Func f) const -> void
{
  std::lock_guard<std::mutex> guard(mutex_);
  for (auto& v: list_)
    f(v);
}

template <typename T>
std::ostream& operator<<(std::ostream& o, const ts_list<T>& l)
{
  l.for_each([&o](const T& t) { o << t; });
  return o;
}
```

Thread-Safe List: Several Errors

- The `has` function is not thread-safe
  - It might return non-sensical results
  - Lock the mutex here too!
- The API is bad
  - Returning an iterator (or anything private) breaks the safety!
  - What use is there for `has`? (Even fixed)
  - Running user code is dangerous

Thread-Safe List: Several Errors

- The `has` function is not thread-safe
  - It might return non-sensical results
  - Lock the mutex here too!
- The API is bad
  - Returning an iterator (or anything private) breaks the safety!
  - What use is there for `has`? (Even fixed)
  - Running user code is dangerous

Thread-Safe List: Running User Code is Dangerous

```
template <typename T>
void opposites(ts_list<T>& l)
{
  l.for_each([&l](int i)
             {
               if (!l.has(-i))
                 l.add(-i);
             });
}
```

- Hangs for ever
- Auto-dead-locked
- Wrong anyway ([W TOCTOU])

Thread-Safe List: Running User Code is Dangerous

```
template <typename T>
void opposites(ts_list<T>& l)
{
  l.for_each([&l](int i)
             {
               if (!l.has(-i))
                 l.add(-i);
             });
}
```

- Hangs for ever
- Auto-dead-locked
- Wrong anyway ([W TOCTOU])

Thread-Safe List: Running User Code is Dangerous

```
template <typename T>
void opposites(ts_list<T>& l)
{
  l.for_each([&l](int i)
             {
               if (!l.has(-i))
                 l.add(-i);
             });
}
```

- Hangs for ever
- Auto-dead-locked
- Wrong anyway ([W TOCTOU])

Thread-Safe List: Running User Code is Dangerous

```
template <typename T>
void opposites(ts_list<T>& l)
{
  l.for_each([&l](int i)
             {
               if (!l.has(-i))
                 l.add(-i);
             });
}
```

- Hangs for ever
- Auto-dead-locked
- Wrong anyway ([WTOCTOU])

Thread-Safe Queue: API

Thread-Safe APIs: std::queue: Special Functions

```
template <class T, class Container = std::deque<T>>
struct queue
{
                          explicit queue(const queue& q);
                          explicit queue(      queue&& q);
                          explicit queue(const Container&);
                          explicit queue(      Container&& = Container{});
  template <class Alloc>          queue(const queue&,  const Alloc&);
  template <class Alloc>          queue(      queue&&, const Alloc&);
  template <class Alloc>          queue(const Container&,  const Alloc&);
  template <class Alloc>          queue(      Container&&, const Alloc&);
  template <class Alloc> explicit queue(const Alloc&);

  queue& operator=(queue&& q);
  void swap(queue&& q);
  //...
};
```

Thread-Safe APIs: std::queue: Core Functions

```
template <class T, class Container = std::deque<T>>
struct queue
{
  //...
  bool empty() const;
  size_t size() const;

  auto front()       -> T&;
  auto front() const -> const T&;
  auto back ()       -> T&;
  auto back () const -> const T&;

  void push(const T& x);
  void push(T&& x);
  void pop();
};
```

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front(/top)` and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue
  `const auto v = std::move(s.front())`

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front(/top)` and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue
  `const auto v = std::move(s.front())`

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front(/top)` and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue
  `const auto v = std::move(s.front())`

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front`(/`top`) and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue

  `const auto v = std::move(s.front())`

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front`(/`top`) and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue

  `const auto v = std::move(s.front())`

std::queue: TOCTOU

```
std::queue<T> q;
if (!q.empty())
{
  const auto v = q.front();
  q.pop();
  work(v);
}
```

```
std::queue<T> q;
try
{
  const auto v = q.pop();
  work(v);
}
catch (empty_queue& e)
  ...
```

- Why on Earth did C++ separate `front(/top)` and `pop`???
- For sake of exception safety!
- Think about a container of large objects
- C++11 lessens the issue
  `const auto v = std::move(s.front())`

std::queue: Merry Popping

- Pass a reference

```
T top;
queue.pop(top);
```

  - Requires default constructible
  - Requires assignment

- Be elitist
  - Require T to support no-throw copy or move ctors
    std::is_nothrow_copy_constructible and
    std::is_nothrow_move_constructible
- Be spendthrift (wasteful, improvident or profligate)
  - Return pointers
  - How do you manage?
- Blend

std::queue: Merry Popping

- Pass a reference

```
T top;
queue.pop(top);
```

  - Requires default constructible
  - Requires assignment
- Be elitist
  - Require `T` to support no-throw copy or move ctors `std::is_nothrow_copy_constructible` and `std::is_nothrow_move_constructible`
- Be spendthrift (wasteful, improvident or profligate)
  - Return pointers
  - How do you manage?
- Blend

std::queue: Merry Popping

- Pass a reference

```
T top;
queue.pop(top);
```

  - Requires default constructible
  - Requires assignment
- Be elitist
  - Require `T` to support no-throw copy or move ctors `std::is_nothrow_copy_constructible` and `std::is_nothrow_move_constructible`
- Be spendthrift (wasteful, improvident or profligate)
  - Return pointers
  - How do you manage?
- Blend

std::queue: Merry Popping

- Pass a reference

```
T top;
queue.pop(top);
```

  - Requires default constructible
  - Requires assignment
- Be elitist
  - Require `T` to support no-throw copy or move ctors `std::is_nothrow_copy_constructible` and `std::is_nothrow_move_constructible`
- Be spendthrift (wasteful, improvident or profligate)
  - Return pointers
  - How do you manage?
- Blend

Condition Variables

How Can a Worker Wait for Some Work?

```
while (true) // for (;;)
{
  try
  {
    auto v = queue.pop();
    work(v);
  }
  catch (queue_empty&)
  {
    std::this_thread::sleep_for(std::chrono::milliseconds{100});
  }
}
```

[WSpinlock], [WBusy_waiting]

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <queue>
#include <thread>

int main()
{
  std::queue<int> queue;
  std::mutex m;
  std::condition_variable cond_var;
  bool done = false;
  bool notified = false;
  // std::thread producer([&] {...})
  // std::thread consumer([&] {...})
  producer.join();
  consumer.join();
}
```

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
std::thread consumer{[&]{
    std::unique_lock<std::mutex> lock{m};
    while (!done)
      {
        while (!notified)
          cond_var.wait(lock);
        while (!queue.empty())
          {
            std::cout << "consuming " << queue.front() << '\n';
            queue.pop();
          }
        notified = false;
      }
  }};
```

- wait releases the mutex while waiting, then locks it
- loop to avoid spurious wakeups [WSpurious_wakeup]
- scoped_lock can only be used for RAII (cannot unlock/relock)

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
std::thread consumer{[&]{
    std::unique_lock<std::mutex> lock{m};
    while (!done)
      {
        while (!notified)
          cond_var.wait(lock);
        while (!queue.empty())
          {
            std::cout << "consuming " << queue.front() << '\n';
            queue.pop();
          }
        notified = false;
      }
  }};
```

- wait releases the mutex while waiting, then locks it
- loop to avoid spurious wakeups [WSpurious_wakeup]
- scoped_lock can only be used for RAII (cannot unlock/relock)

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
std::thread consumer{[&]{
    std::unique_lock<std::mutex> lock{m};
    while (!done)
      {
        while (!notified)
          cond_var.wait(lock);
        while (!queue.empty())
          {
            std::cout << "consuming " << queue.front() << '\n';
            queue.pop();
          }
        notified = false;
      }
  }};
```

- wait releases the mutex while waiting, then locks it
- loop to avoid spurious wakeups [WSpurious_wakeup]
- scoped_lock can only be used for RAII (cannot unlock/relock)

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
std::thread consumer{[&]{
    std::unique_lock<std::mutex> lock{m};
    while (!done)
      {
        while (!notified)
          cond_var.wait(lock);
        while (!queue.empty())
          {
            std::cout << "consuming " << queue.front() << '\n';
            queue.pop();
          }
        notified = false;
      }
  }};
```

- wait releases the mutex while waiting, then locks it
- loop to avoid spurious wakeups [WSpurious_wakeup]
- scoped_lock can only be used for RAII (cannot unlock/relock)

Condition Variables

http://en.cppreference.com/w/cpp/thread/condition_variable

```
std::thread producer{[&]{
    for (int i = 0; i < 5; ++i)
      {
        std::lock_guard<std::mutex> lock{m};
        std::cout << "producing " << i << '\n';
        queue.push(i);
        notified = true;
        cond_var.notify_one();
      }

    std::lock_guard<std::mutex> lock{m};
    notified = true;
    done = true;
    cond_var.notify_one();
  }};
```

Condition Variables Runs

http://en.cppreference.com/w/cpp/thread/condition_variable

```
producing 0
consuming 0
producing 1
producing 2
consuming 1
consuming 2
producing 3
producing 4
consuming 3
consuming 4
```

```
producing 0
consuming 0
producing 1
producing 2
producing 3
consuming 1
consuming 2
consuming 3
producing 4
consuming 4
```

```
producing 0
consuming 0
producing 1
producing 2
producing 3
producing 4
consuming 1
consuming 2
consuming 3
consuming 4
```

Thread-Safe Queue API [ccia4.3]

```
template<typename T>
class threadsafe_queue
{
public:
  threadsafe_queue();
  threadsafe_queue(const threadsafe_queue&);
  threadsafe_queue& operator=(const threadsafe_queue&) = delete;

  void push(T new_value);

  bool try_pop(T& value);
  std::shared_ptr<T> try_pop();

  void wait_and_pop(T& value);
  std::shared_ptr<T> wait_and_pop();

  bool empty() const;
};
```

Thread-Safe Queue [ccia4.5]

```
template <typename T> class threadsafe_queue
{
public:
  threadsafe_queue() {}
  threadsafe_queue(threadsafe_queue const& other)
  {
    std::lock_guard<std::mutex> lock{other.mutex_};
    queue_ = other.queue_;
  }
  // ...
  bool empty() const
  {
    std::lock_guard<std::mutex> lock{mutex_};
    return queue_.empty();
  }
private:
  mutable std::mutex mutex_;
  std::queue<T> queue_;
  std::condition_variable cond_;
};
```

Thread-Safe Queue [ccia4.5]

```
bool try_pop(T& value)
{
  std::lock_guard<std::mutex> lock{mutex_};
  if (queue_.empty())
    return false;
  value = std::move(queue_.front());
  queue_.pop();
  return true;
}

std::shared_ptr<T> try_pop()
{
  std::lock_guard<std::mutex> lock{mutex_};
  if (queue_.empty())
    return {};
  auto res = std::make_shared<T>(std::move(queue_.front()));
  queue_.pop();
  return res;
}
```

Thread-Safe Queue [ccia4.5]

```
void push(T new_value)
{
  std::lock_guard<std::mutex> lock{mutex_};
  queue_.push(std::move(new_value));
  cond_.notify_one();
}

void wait_and_pop(T& value)
{
  std::unique_lock<std::mutex> lock{mutex_};
  // Spurious-wakeup protected!
  cond_.wait(lock, [this]{ return !queue_.empty(); });
  value = std::move(queue_.front());
  queue_.pop();
}
```

Thread-Safe Queue [ccia4.5]

```
std::shared_ptr<T> wait_and_pop()
{
  std::unique_lock<std::mutex> lock{mutex_};
  cond_.wait(lock, [this]{ return !queue_.empty(); });
  auto res = std::make_shared<T>(std::move(queue_.front()));
  queue_.pop();
  return res;
}
```

Thread-Safe Queue [ccia4.5]

```
std::shared_ptr<T> wait_and_pop()
{
  std::unique_lock<std::mutex> lock{mutex_};
  cond_.wait(lock, [this]{ return !queue_.empty(); });
  auto res = std::make_shared<T>(std::move(queue_.front()));
  queue_.pop();
  return res;
}
```

Not exception safe!

- If `make_shared` fails, the `notify_code` is lost
- The value is not lost though
- Fixes?

Thread-Safe Queue [ccia6.3]

```
template<typename T>
class threadsafe_queue
{
public:
  threadsafe_queue() {}
  threadsafe_queue(threadsafe_queue const& other)
  {
    std::lock_guard<std::mutex> lock{other.mutex_};
    queue_ = other.queue_;
  }
  // ...
private:
  mutable std::mutex mutex_;
  std::queue<std::shared_ptr<T>> queue_;
  std::condition_variable cond_;
};
```

Thread-Safe Queue [ccia6.3]

```
void push(T new_value)
{
  auto data = std::make_shared<T>(std::move(new_value));
  std::lock_guard<std::mutex> lock{mutex_};
  queue_.push(std::move(data));
  cond_.notify_one();
}
```

Thread-Safe Queue [ccia6.3]

```
void wait_and_pop(T& value)
{
  std::unique_lock<std::mutex> lock{mutex_};
  cond_.wait(lock, [this]{ return !queue_.empty(); });
  value = std::move(*queue_.front());
  queue_.pop();
}

bool try_pop(T& value)
{
  std::lock_guard<std::mutex> lock(mutex_);
  if (queue_.empty())
    return false;
  value = std::move(*queue_.front());
  queue_.pop();
  return true;
}
```

Thread-Safe Queue [ccia6.3]

```
std::shared_ptr<T> wait_and_pop()
{
  std::unique_lock<std::mutex> lock(mutex_);
  cond_.wait(lock, [this]{return !queue_.empty();});
  std::shared_ptr<T> res=queue_.front();
  queue_.pop();
  return res;
}

std::shared_ptr<T> try_pop()
{
  std::lock_guard<std::mutex> lock(mutex_);
  if (queue_.empty())
    return std::shared_ptr<T>();
  auto res = queue_.front();
  queue_.pop();
  return res;
}
```

Thread-Safe Queue: Finally

Thread-Safe Queue: Finally?

Thread-Safe Queue: Not Yet

Thread-Safe Queue: Limitations

- There is a single protected data: the whole queue
- Hence a single mutex for the whole
- So consumers at one end are blocked by producers at the other end
- To break this limitation one needs. . .
- one mutex per cell

Thread-Safe Queue: Limitations

- There is a single protected data: the whole queue
- Hence a single mutex for the whole
- So consumers at one end are blocked by producers at the other end
- To break this limitation one needs. . .
- one mutex per cell

Thread-Safe Queue: Start from Scratch!

See session #6.

APIs

Mutex (<mutex>)

```
namespace std
{
  class mutex // implements Lockable.
  {
  public:
    mutex(mutex const&) = delete;
    mutex& operator=(mutex const&) = delete;
    constexpr mutex() noexcept;
    ~mutex();
    void lock();
    void unlock();
    bool try_lock();
  };
  class recursive_mutex;       // implements Lockable.
```

Timed Mutex (<mutex>)

```
namespace std
{
  class timed_mutex                // implements TimedLockable.
  { ...
    template <typename Rep, typename Period>
    bool try_lock_for(std::chrono::duration<Rep, Period> const& duration);
    template <typename Clock, typename Duration>
    bool try_lock_until(std::chrono::time_point<Clock,Duration>const& tp);
  };
  class recursive_timed_mutex; // implements TimedLockable.
}
```

Shared Mutex (<boost/thread/shared_mutex.hpp>)

- same interface as `std::thread`
- multiple-reader / single-writer
- no reader-writer priority policies
- the OS decide which thread is the next to get the lock
- lack of reader or writer starvation: guaranteed fairness

Upgrade Mutex (<boost/thread/shared_mutex.hpp>)

- UpgradeLockable concept refines SharedLockable concept
- extension to the multiple-reader / single-write model
- a single thread may have upgradable ownership at the same time as others have shared ownership
- the thread with upgradable ownership may at any time attempt to upgrade that ownership to exclusive ownership
- if no other threads have shared ownership, the upgrade is completed immediately, and the thread now has exclusive ownership
- if a thread with upgradable ownership tries to upgrade whilst other threads have shared ownership, the attempt will fail and the thread will block until exclusive ownership can be acquired.
- ownership can also be downgraded

Locks (<mutex>)

```
// Not MoveConstructible or MoveAssignable.
// Not CopyConstructible or CopyAssignable.
template <typename LockableType>
class lock_guard;

// MoveConstructible and MoveAssignable.
// Not CopyConstructible or CopyAssignable.
template <typename LockableType>
class unique_lock;

template <typename LockableType1, typename... LockableType2>
void lock(LockableType1& m1, LockableType2& m2...);

template <typename LockableType1, typename... LockableType2>
int try_lock(LockableType1& m1, LockableType2& m2...);
```

Lock Extentions

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

namespace boost
{
  template <typename Lockable>
  class shared_lock;

  template <typename Lockable>
  class upgrade_lock;

  template <class Mutex>
  class upgrade_to_unique_lock;

  template <typename Lockable>
  class strict_lock;

  template <typename Lock>
  class nested_strict_lock;
}
```

Lock Functions

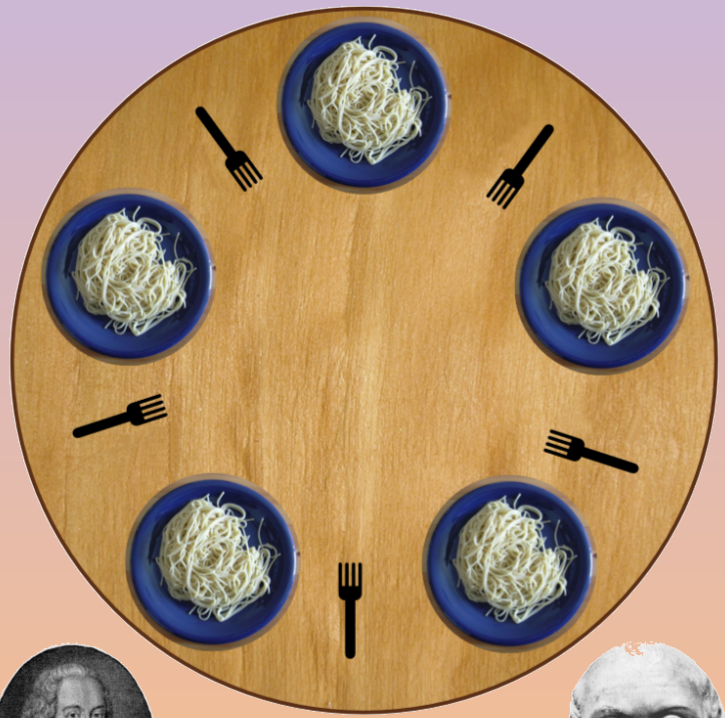
```
namespace boost
{
  template <typename ForwardIterator>
  void lock(ForwardIterator begin, ForwardIterator end);

  template <typename ForwardIterator>
  ForwardIterator try_lock(ForwardIterator begin, ForwardIterator end);
}
```

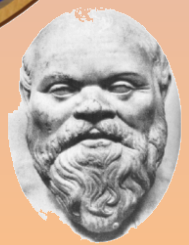
Lock Factories

```
namespace boost
{
  template <typename Lockable>
  lock_guard<Lockable> make_lock_guard(Lockable& mtx);
  template <typename Lockable>
  lock_guard<Lockable> make_lock_guard(Lockable& mtx, adopt_lock_t);

  template <typename Lockable>
  unique_lock<Lockable> make_unique_lock(Lockable& mtx);

  template <typename Lockable>
  unique_lock<Lockable> make_unique_lock(Lockable& mtx, adopt_lock_t);
  template <typename Lockable>
  unique_lock<Lockable> make_unique_lock(Lockable& mtx, defer_lock_t);
  template <typename Lockable>
  unique_lock<Lockable> make_unique_lock(Lockable& mtx, try_to_lock_t);

  template <typename ...Lockable>
  std::tuple<unique_lock<Lockable> ...> make_unique_locks(Lockable& ...mtx)
}
```

Locks





12h Plato

2h24 Konfuzius

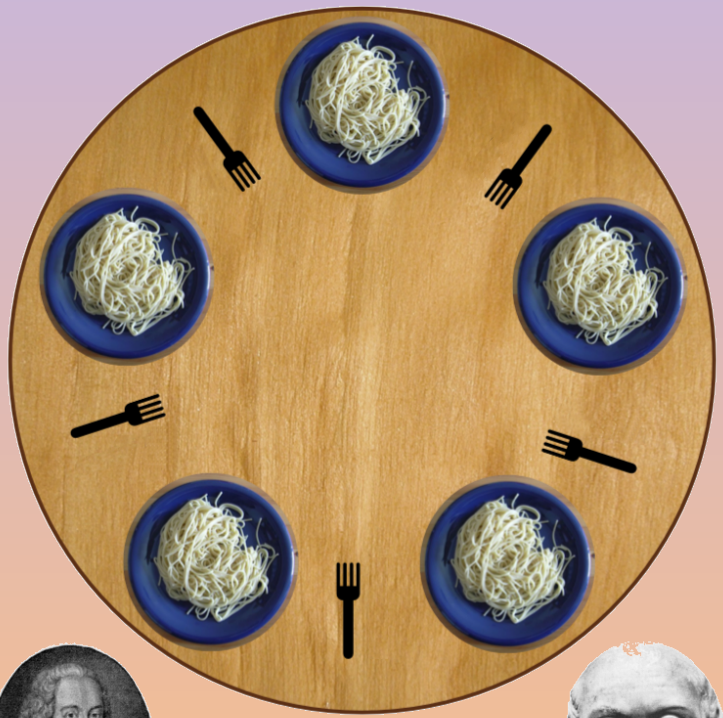
4h48 Socrates

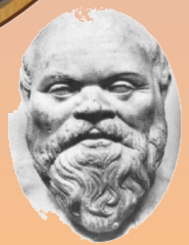
7h12 Voltaire

9h36 Descartes

Dining Philoshophers [WDining_philosophers_problem]





12h Plato

2h24 Konfuzius

4h48 Socrates

7h12 Voltaire

9h36 Descartes

Dining Philoshophers [W Dining_philosophers_problem]

12h Plato

2h24 Konfuzius

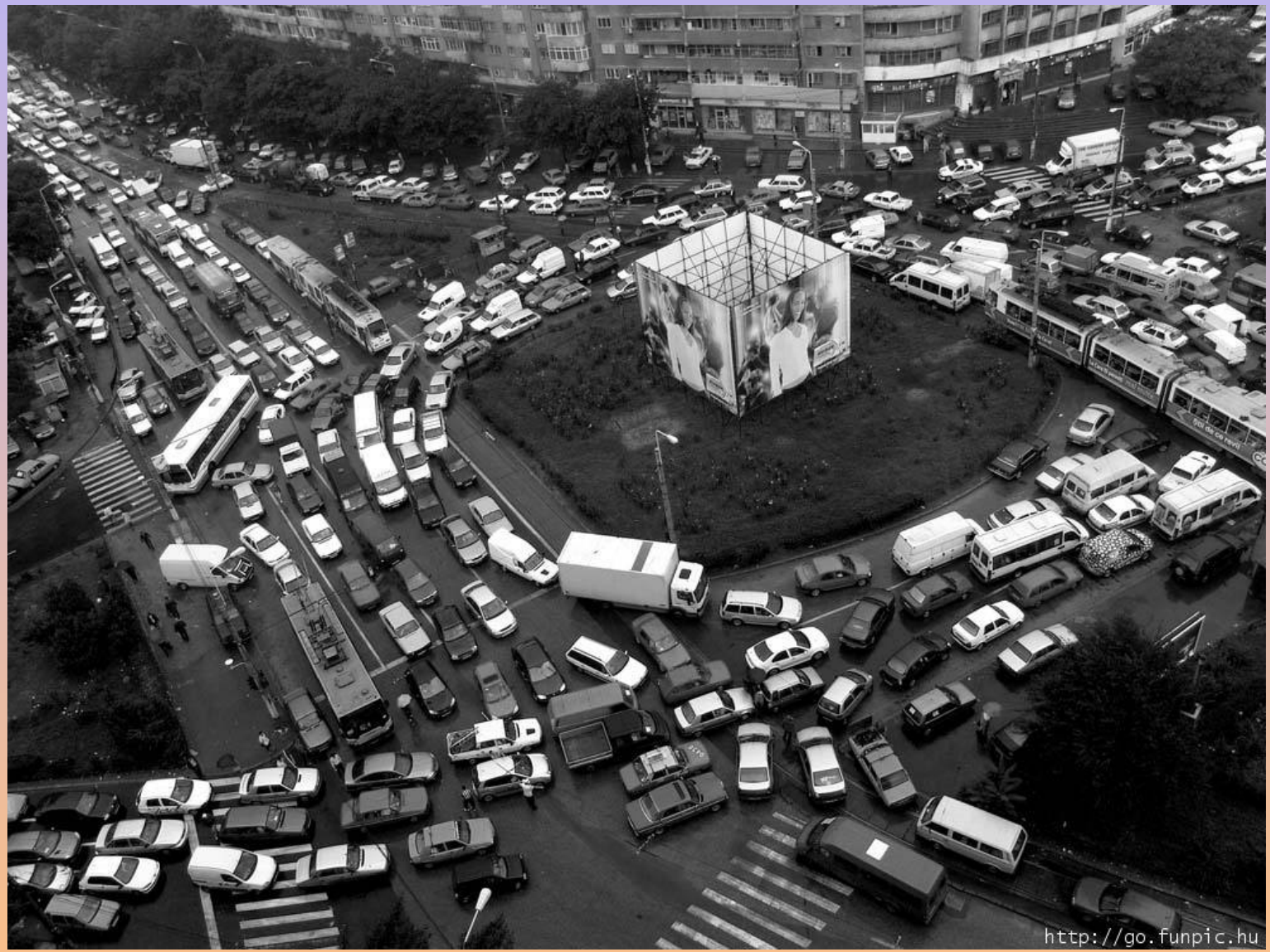
4h48 Socrates

7h12 Voltaire

9h36 Descartes

Deadlocks

Dead Locks



Avoid Dead Locks

- Use `std::lock(begin, end)`
- Don't use locks at all

Avoid Dead Locks

- Use `std::lock(begin, end)`
- Don't use locks at all

Avoid Dead Locks

- Use `std::lock(begin, end)`
- Don't use locks at all

Atomic Types

Race Conditions

Race Conditions

Race Conditions

Undefined Behavior

Race Condition and Undefined Behavior

- Data race & Undefined Behavior
  - no enforced ordering bw two accesses
  - to a single memory location
  - from separate threads
  - one or both is not atomic
  - one or both is a write
- Data race
  - both are atomic

Race Condition and Undefined Behavior

- Data race & Undefined Behavior
  - no enforced ordering bw two accesses
  - to a single memory location
  - from separate threads
  - one or both is not atomic
  - one or both is a write
- Data race
  - both are atomic

Atomic Operations and Types

- operations on atomic types are atomic
- only those types have atomic operations (per language definition)
- some are "truly" atomic (lock-free)
- others are implemented using locks
- see `is_lock_free`
- `std::atomic_flag` *is* lock-free
- `std::atomic<>` specializations

Unprotected vs. Atomic vs. Locked

```
template <typename T> struct scoped_clock
{
  using clock = std::chrono::steady_clock;
  scoped_clock(const std::string& n, T& v) : name_(n), value_(v){}
  ~scoped_clock() {
    std::chrono::duration<double, std::milli> d = clock::now() - start_;
    std::cout << "// begin:" << name_ << std::endl
              << boost::format("%|16|: %|8d| %|8.2f|ms")
                 % name_ % value_ % d.count() << std::endl
              << "// end:" << name_ << std::endl << std::endl;
  }
  std::string name_;
  T& value_;
  clock::time_point start_ = clock::now();
};

template <typename T>
scoped_clock<T> make_clock(const std::string& name, T& value) {
  return scoped_clock<T>{name, value};
}
```

Unprotected vs. Atomic vs. Locked

```
void plain()
{
  int shared{0};
  std::vector<std::thread> threads;
  auto chrono = make_clock("plain", shared);
  for (int i = 0; i < nthreads; ++i)
    threads.emplace_back([&]{
        for (int j = 0; j < niters; ++j)
          ++shared;
      });
  for (auto& t: threads)
    t.join();
}
```

Unprotected vs. Atomic vs. Locked

```
void atomic()
{
  std::atomic<int> shared{0};
  std::vector<std::thread> threads;
  auto chrono = make_clock("atomic", shared);
  for (int i = 0; i < nthreads; ++i)
    threads.emplace_back([&]{
        for (int j = 0; j < niters; ++j)
          ++shared;
      });
  for (auto& t: threads)
    t.join();
}
```

Unprotected vs. Atomic vs. Locked

```
void lock()
{
  std::mutex mutex;
  int shared{0};
  std::vector<std::thread> threads;
  auto chrono = make_clock("lock", shared);
  for (int i = 0; i < nthreads; ++i)
    threads.emplace_back([&]{
        for (int j = 0; j < niters; ++j)
          {
            std::lock_guard<std::mutex> lock{mutex};
            ++shared;
          }
      });
  for (auto& t: threads)
    t.join();
}
```

Unprotected vs. Atomic vs. Locked

```
const int nthreads = std::thread::hardware_concurrency();
constexpr int niters = 1000000;
int main()
{
  std::cout << "nthreads: " << nthreads
            << ", niters: " << niters << '\n';
  plain();
  atomic();   boost_atomic();  tbb_atomic();
  lock();     boost_lock();    tbb_lock();
}
```

Unprotected vs. Atomic vs. Locked

```
const int nthreads = std::thread::hardware_concurrency();
constexpr int niters = 1000000;
int main()
{
  std::cout << "nthreads: " << nthreads
            << ", niters: " << niters << '\n';
  plain();
  atomic();   boost_atomic();  tbb_atomic();
  lock();     boost_lock();    tbb_lock();
}
```

```
nthreads: 4, niters: 1000000
          plain:  1364710    10.49ms
         atomic:  4000000    81.48ms
   boost atomic:  4000000    80.36ms
     tbb atomic:  4000000    78.18ms
           lock:  4000000 19294.96ms
     boost lock:  4000000 15911.16ms
       tbb lock:  4000000 16862.45ms
```

Tasks

Returning Values from Threads

Returning Values from Threads

- Need for a channel to communicate between threads
- Two ends

  promise the input end

  future the output end
- The future blocks until the promise is fulfilled
- Be cautious about resource management

Promises & Futures [Milewski, 2011a]

```
void hello(std::promise<std::string>&& prm)
{
  std::string res{"Hello from future"};
  prm.set_value(res);
}

int main()
{
  std::promise<std::string> prm;
  std::future<std::string> fut{prm.get_future()};
  std::thread thread{hello, std::move(prm)};
  std::cout << "Hello from main\n";
  std::cout << fut.get() << '\n';
  thread.join();
}
```

Promises & Futures [Milewski, 2011a]

```
void hello(std::promise<std::string>&& prm)
{
  std::string res{"Hello from future"};
  prm.set_value(res);
}

int main()
{
  std::promise<std::string> prm;
  std::future<std::string> fut{prm.get_future()};
  std::thread thread{hello, std::move(prm)};
  std::cout << "Hello from main\n";
  std::cout << fut.get() << '\n';
  thread.join();
}
```

```
Hello from main
Hello from future
```

Promises & Futures

```
void hello(std::promise<std::string>& prm)
{
  std::string res{"Hello from future"};
  prm.set_value(res);
}

int main()
{
  std::promise<std::string> prm;
  std::future<std::string> fut{prm.get_future()};
  std::thread thread{hello, std::ref(prm)};
  std::cout << "Hello from main\n";
  std::cout << fut.get() << '\n';
  thread.join();
}
```

```
Hello from main
Hello from future
```

Handling Exceptions

```
void hello(std::promise<std::string>&&)
{
  throw std::runtime_error("No future! (And Punk not dead)");
}

int main()
{
  std::promise<std::string> prm;
  std::future<std::string> fut{prm.get_future()};
  std::thread thread{hello, std::move(prm)};
  std::cout << "Hello from main\n";
  std::cout << fut.get() << '\n';
  thread.join();
}
```

Handling Exceptions

```
void hello(std::promise<std::string>&&)
{
  throw std::runtime_error("No future! (And Punk not dead)");
}

int main()
{
  std::promise<std::string> prm;
  std::future<std::string> fut{prm.get_future()};
  std::thread thread{hello, std::move(prm)};
  std::cout << "Hello from main\n";
  std::cout << fut.get() << '\n';
  thread.join();
}
```

```
Hello from main
libc++abi.dylib: terminate called throwing an exception
zsh: abort      ./5-concurrency/promise-throw
```

Handling Exceptions

```
void hello(std::promise<std::string>&& prm)
{
  try
    {
      throw std::runtime_error("No future! (And Punk not dead)");
    }
  catch (...)
    {
      prm.set_exception(std::current_exception());
    }
}
```

Handling Exceptions

```
std::promise<std::string> prm;
std::future<std::string> fut{prm.get_future()};
std::thread thread{hello, std::move(prm)};
std::cout << "Hello from main\n";
try
  {
    std::cout << fut.get() << '\n';
  }
catch (const std::exception& e)
  {
    std::cout << "Caught: " << e.what() << '\n';
  }
thread.join();
```

Handling Exceptions

```
std::promise<std::string> prm;
std::future<std::string> fut{prm.get_future()};
std::thread thread{hello, std::move(prm)};
std::cout << "Hello from main\n";
try
  {
    std::cout << fut.get() << '\n';
  }
catch (const std::exception& e)
  {
    std::cout << "Caught: " << e.what() << '\n';
  }
thread.join();
```

```
Hello from main
Caught: No future! (And Punk not dead)
```

Async

Async Builds a Future

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

std::string hello()
{
  return "Hello from future";
}

int main()
{
  auto future = std::async(hello);
  std::cout << "Hello from main\n";
  std::cout << future.get() << '\n';
}
```

Async Builds a Future

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

std::string hello()
{
  return "Hello from future";
}

int main()
{
  auto future = std::async(hello);
  std::cout << "Hello from main\n";
  std::cout << future.get() << '\n';
}
```

```
Hello from main
Hello from future
```

Async: Exceptions

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

int main()
{
  auto future
    = std::async([] { throw std::logic_error("Free Private School"); });
  try
    {
      future.get();
    }
  catch (const std::exception& e)
    {
      std::cout << "Caught: " << e.what() << '\n';
    }
}
```

Async: Exceptions

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

int main()
{
  auto future
    = std::async([] { throw std::logic_error("Free Private School"); });
  try
    {
      future.get();
    }
  catch (const std::exception& e)
    {
      std::cout << "Caught: " << e.what() << '\n';
    }
}
```

```
Caught: Free Private School
```

Sequential Quick Sort [ccia4.12]

```
template <typename T>
std::list<T> quick_sort(std::list<T> input)
{
  if (input.empty())
    return input;
  auto res = std::list<T>{};
  res.splice(res.begin(), input, input.begin()); // Steal input[0].
  T const& pivot = res.front();
  auto divide_point = std::partition(input.begin(), input.end(),
                                     [&](T const& t){ return t<pivot; });
  auto lower_part = std::list<T>{};
  lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
  auto new_lower = quick_sort(std::move(lower_part));
  auto new_higher = quick_sort(std::move(input));
  res.splice(res.end(), new_higher);
  res.splice(res.begin(), new_lower);
  return res;
}
```

Parallel Quick Sort [ccia4.13]

```
template <typename T>
std::list<T> quick_sort(std::list<T> input)
{
  if (input.empty())
    return input;
  auto res = std::list<T>{};
  res.splice(res.begin(), input, input.begin()); // Steal input[0].
  T const& pivot = res.front();
  auto divide_point = std::partition(input.begin(), input.end(),
                                     [&](T const& t){ return t<pivot; });
  auto lower_part = std::list<T>{};
  lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
  auto new_lower = std::async(&quick_sort<T>, std::move(lower_part));
  auto new_higher = quick_sort(std::move(input));
  res.splice(res.end(), new_higher);
  res.splice(res.begin(), new_lower.get());
  return res;
}
```

Tasks

Launch Policies

- Run in new thread
  `std::async(std::launch::async, ...)`
- Run in `wait/get`
  `std::async(std::launch::deferred, ...)`
- Let implementation chose
  `std::async(...)`
  `std::async(std::launch::async | std::launch::deferred, ...)`

Launch Policies

```
int working_thread_id()
{
  static std::unordered_map<std::thread::id, int> map;
  static std::mutex mapmut;

  std::this_thread::sleep_for(std::chrono::milliseconds{1});
  std::lock_guard<std::mutex> lock{mapmut};
  return
    map
    .emplace(std::this_thread::get_id(), map.size())
    .first
    ->second;
}
```

Launch Policies: async | deferred

```
std::vector<std::future<int>> futures;
for (size_t i = 0; i < 198; ++i)
  futures.emplace_back(std::async(l, working_thread_id));
for (size_t i = 0; i < futures.size(); ++i)
  std::cout << std::setw(3) << futures[i].get()
            << ((i+1) % 18 ? ' ' : '\n');
```

Launch Policies: async | deferred

```
std::vector<std::future<int>> futures;
for (size_t i = 0; i < 198; ++i)
  futures.emplace_back(std::async(l, working_thread_id));
for (size_t i = 0; i < futures.size(); ++i)
  std::cout << std::setw(3) << futures[i].get()
            << ((i+1) % 18 ? ' ' : '\n');
```

```
  1   2   3   4   6   1   5   7  10   8  12  13  16  11  14   9  15  17
 18  19  20  21  22   2  23  27  24  30  25  29   3   4  26  31  28  32
  6   1   5   7   2  10   3   1   4   6   5   2   7  10   1   3   4   8
 12  13  16  11  14   6   9  15  17  18  19  20  21   5   2  22  23  27
 24  30  25  29   7  26  31  28  10   1   3   4   8  12   6  13  16   2
  5  11   1   3   4   7   6  10   8  12   2   5  13  16  14   9  11  15
 17  18  19  20  21  22  23  27  24  30  25   1   3  29   4   6   7  10
  8   2   5  12  13   1   3   4   2   1   1   1   1   1   1   2   1   1
  1   1   1   1   1   1   1   1   1   1   1   2   1   2   3   4   1   2
  6   5   7  10   8  12  13  16  11  14   9  15  17  18  19   3  20  21
 22  23   4   1  27   2   6   5   7  12  10   8  13  16   3  11  14   9
```

Launch Policies: `deferred`

With `std::async(std::launch::deferred, working_thread_id)`

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Launch Policies: async

With `std::async(std::launch::async, working_thread_id)`

1	2	1	3	1	2	3	4	6	5	7	10	8	12	13	16	11	14
9	15	17	18	19	20	21	22	23	27	24	30	25	29	26	31	28	32
33	1	2	3	4	6	5	1	3	7	2	10	8	12	13	16	11	14
9	15	17	18	4	19	6	20	21	22	1	3	2	4	6	5	7	10
8	12	13	16	11	14	9	15	17	18	19	20	1	21	22	2	3	4
23	27	6	5	7	10	8	1	12	13	2	3	4	6	16	11	14	9
15	17	5	7	10	8	18	19	20	21	22	1	23	27	24	30	25	29
26	12	31	2	3	1	4	6	5	2	7	10	3	8	1	4	2	6
3	5	1	7	10	2	4	8	12	13	16	11	14	3	9	15	17	6
5	1	18	19	20	21	22	23	27	7	24	2	3	1	4	6	5	7
10	2	3	8	12	13	16	11	14	1	9	4	6	1	2	3	4	1

And also

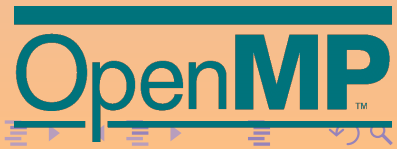
Tools

Boost.Interprocess

- Shared memory
- Mapped memory
- Mutexes, condition variables, semaphores, etc.
- Message queues

Open-MP [WOpenMP]

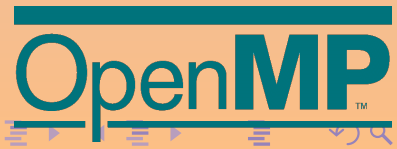
```
// CXX: g++-mp-5
// CXXFLAGS: -fopenmp

#include <iostream>

int main(void)
{
# pragma omp parallel
  std::cout << "Hello, " << "world" << '!' << std::endl;
  std::cout << "STOP." << std::endl;
}
```



Open-MP [WOpenMP]

```
// CXX: g++-mp-5
// CXXFLAGS: -fopenmp

#include <iostream>

int main(void)
{
# pragma omp parallel
  std::cout << "Hello, " << "world" << '!' << std::endl;
  std::cout << "STOP." << std::endl;
}
```

```
Hello, Hello, Hello, Hello, worldworldworldworld!!!!



STOP.
```



Open-MP [WOpenMP]

```
  int th_id, nthreads;
# pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
#   pragma omp critical
    std::cout << "Hello World from thread " << th_id << '\n';
#   pragma omp barrier
#   pragma omp master
    {
      nthreads = omp_get_num_threads();
      std::cout << "There are " << nthreads << " threads\n";
    }
  }
```

Open-MP [WOpenMP]

```
  int th_id, nthreads;
# pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
#   pragma omp critical
    std::cout << "Hello World from thread " << th_id << '\n';
#   pragma omp barrier
#   pragma omp master
    {
      nthreads = omp_get_num_threads();
      std::cout << "There are " << nthreads << " threads\n";
    }
  }
```

```
Hello World from thread 0
Hello World from thread 2
Hello World from thread 1
Hello World from thread 3
There are 4 threads
```

Helgrind

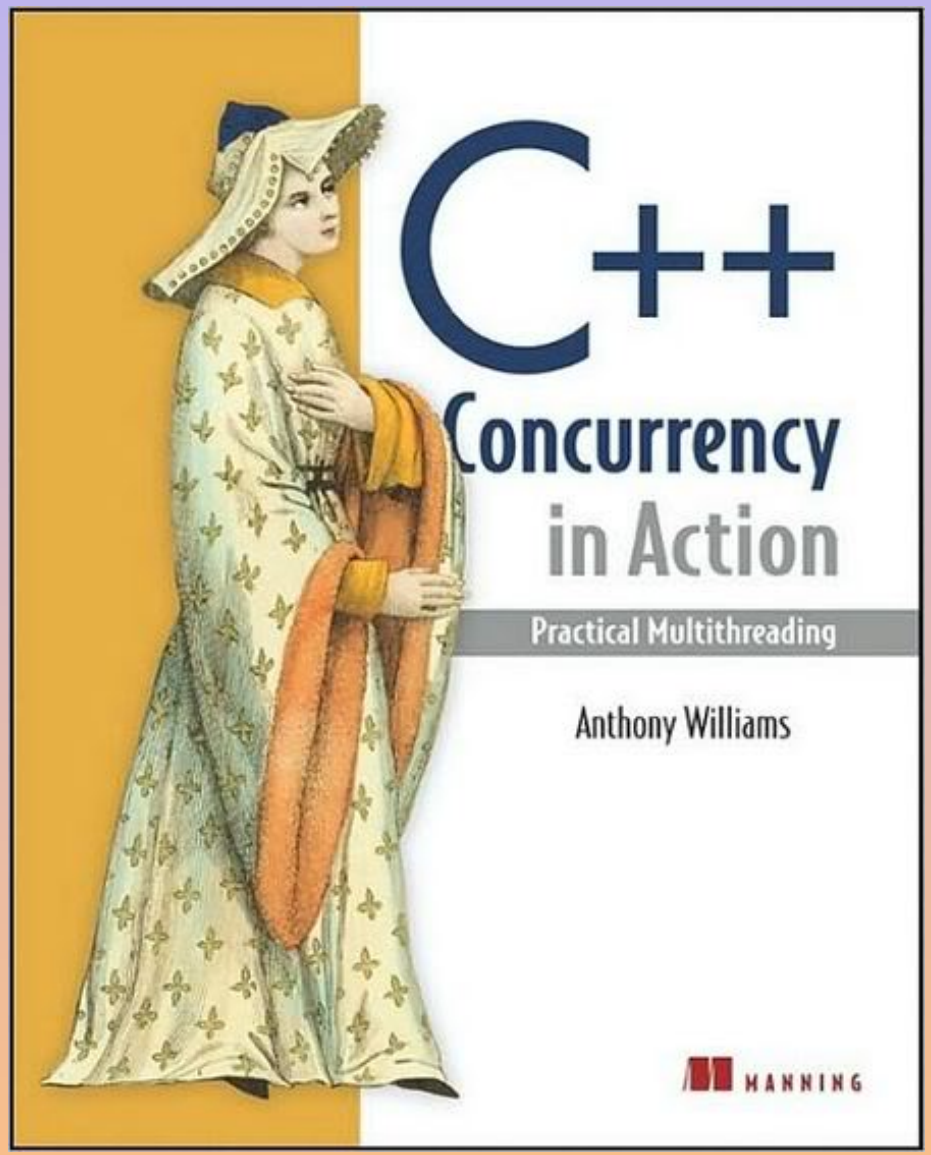
```
#include <iostream>
#include <thread>

int main()
{
  int var = 0;
  std::thread child{[&]{ ++var; }};
  ++var;
  child.join();
  std::cout << var << '\n';
}
```

Helgrind

```
Possible data race during read of size 4 at 0x1048048DC by thread #2
Locks held: none
   at 0x100001A7F: main::$_0::operator()() const (helgrind.cc:7)
   by 0x100001953: _ZNSt3__114__thread_proxyINS_5tupleIJZ4mainE3__0EEEEEPvS4_ (__fu
   by 0x1F57A1: _pthread_start (in /usr/lib/system/libsystem_c.dylib)
   by 0x1E21E0: thread_start (in /usr/lib/system/libsystem_c.dylib)

This conflicts with signed char previous write of size 4 by thread #1
Locks held: none
   at 0x100001210: main (helgrind.cc:8)
------------------------------------------------------------
Possible data race during write of size 4 at 0x1048048DC by thread #2
Locks held: none
   at 0x100001A86: main::$_0::operator()() const (helgrind.cc:7)
   by 0x100001953: _ZNSt3__114__thread_proxyINS_5tupleIJZ4mainE3$_0EEEEEPvS4_ (__fu
   by 0x1F57A1: _pthread_start (in /usr/lib/system/libsystem_c.dylib)
   by 0x1E21E0: thread_start (in /usr/lib/system/libsystem_c.dylib)

This conflicts with signed char previous write of size 4 by thread #1
Locks held: none
   at 0x100001210: main (helgrind.cc:8)
```

References

[Williams, 2012]



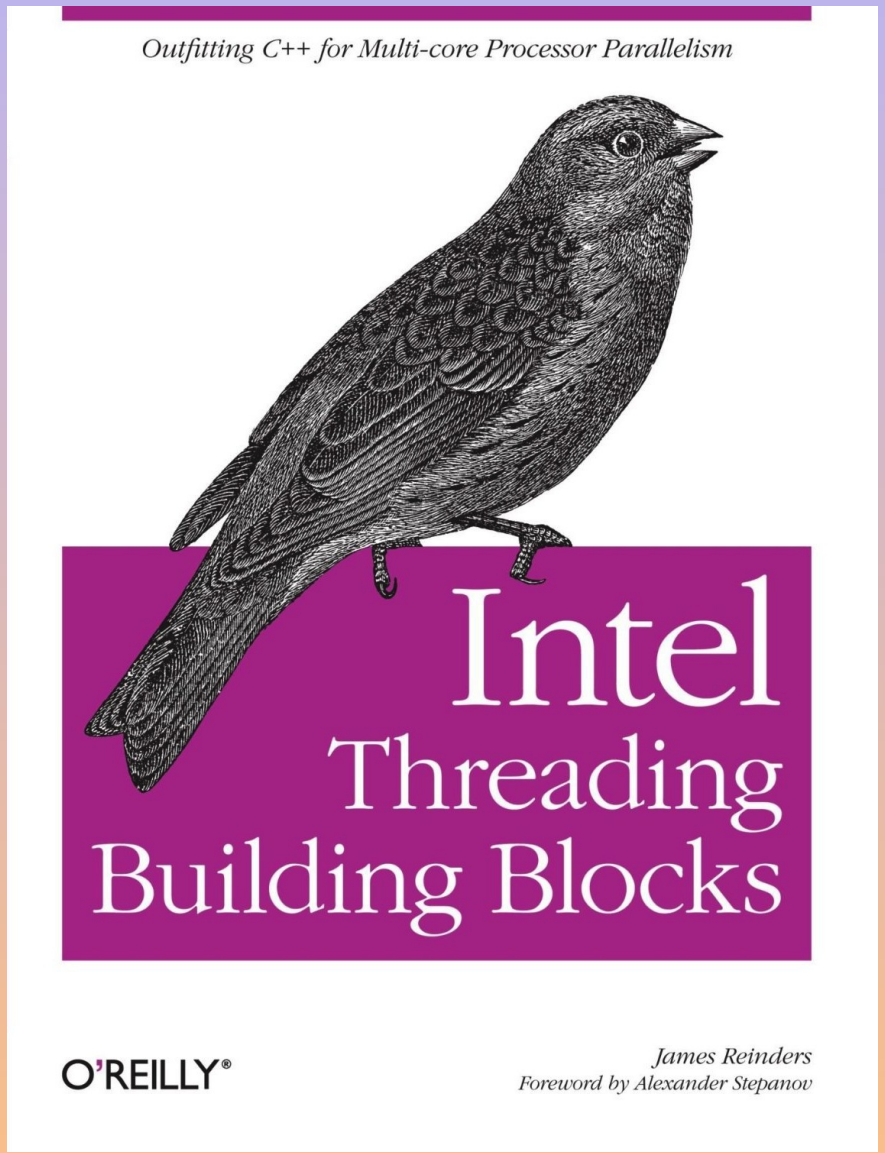
C++11 Concurrency Series [Milewski, 2011a]

1. Fork/Join
2. Passing Arguments to Threads
3. Sharing Data Between Threads
4. Futures and Promises
5. Tasks
6. MapReduce
7. Mutexes, Locks, and Monitors
8. Data Races
9. Condition Variables

More Videos

- [Sutter, 2013]
- ...

[Reinders, 2007]



Bibliography I


Burelle, M. (2013).
Parallel and concurrent programming introduction and foundation.
http://wiki-prog.infoprepa.epita.fr/.


Milewski, B. (2011a).
C++11 concurrency tutorial.
http://bartoszmilewski.com/2011/08/29/c11-concurrency-tutorial/.


Milewski, B. (2011b).
The language of concurrency video.
http://bartoszmilewski.com/2011/06/27/the-language-of-concurrency-video/.

Bibliography II


Ousterhout, J. K. (1996).
Why threads are a bad idea (for most purposes).
http://www.cc.gatech.edu/classes/AY2009/cs4210_fall/papers/ousterhout-threads.pdf.


Pike, R. (2012).
Concurrency is not parallelism (it's better).
http://blog.golang.org/2013/01/concurrency-is-not-parallelism.html.


Reinders, J. (2007).
Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.
O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

Bibliography III


Sutter, H. (2013).
C++ concurrency.
http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism.


Williams, A. (2012).
C++ Concurrency in Action: Practical Multithreading.
Manning Pubs Co Series. Manning Publications Company.

Concurrency in C++
Part I: `audience.unlock()`

Bartosz Milewski

Anthony Williams