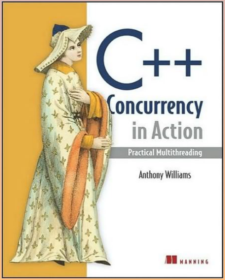
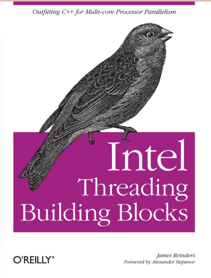
Concurrency in C++
Part II

Akim Demaille akim@lrde.epita.fr

May, 14th 2013

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





Concurrency in C++ Part II

Tasks

1. Tasks
   - Async
   - Tasks
   - A Glance at C++ 1y
   - Threading Building Blocks
2. Asynchronous Input/Output
3. Data Protection
4. References

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;
}
```

Quick Sorts [ccia4.12]

```
int main(int argc, const char* argv[])
{
  size_t n = 1 < argc ? boost::lexical_cast<size_t>(argv[1]) : 4000;

  using ints = std::list<int>;
  auto l = ints(n); // *Not* ints{n}!
  std::generate(std::begin(l), std::end(l), std::rand);

  {
    auto res = ints{};
    chrono("sequential", n, [&]{ res = seq::quick_sort(l); });
    assert(std::is_sorted(std::begin(res), std::end(res)));
  }

  {
    auto res = ints{};
    chrono("parallel", n, [&]{ res = par::quick_sort(l); });
    assert(std::is_sorted(std::begin(res), std::end(res)));
  }
}
```

Quick Sorts [ccia4.12]

```
    parallel:     4000   456.60ms
```

Quick Sorts [ccia4.12]

```
parallel:     4000    456.60ms
```

```
sequential:     4000      3.04ms
```

Scalability Issues

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

```cpp
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');
```

```
  2   1   3   4   5   6   7   8   9  11  12  10  13  14  15  16  17  18
 19  20  21  22  23  25  24  26  27  28  29  30  31  32  33  34  35  36
 37  38  40  39  41  42  43  44  45  46  47   2   1   3   4   5   6   7
  8   9  11  12  10  13  14  15  16  17  18  19  20  21  22  23  25  24
 26  27  28  29  30  31  32  33  34  35  36   2  37   3   1   4   5   6
  7  38  40  39   8   9  11  12  10  13  14  15  16  17  18  19  20  21
 22  23  25  24  26  27  28  41  29  30  31  32  33  42  34  35   2  36
  3   1   4  37   5   6   7   8   9  11  12  10  13  14  15  16  17  18
 20  19  21  22  23  25  24  26  27  28  29  30  31  32  33  34  35  38
  2  36   3   1   4  37   5  40   6   8  39   9  41  11   7  12  10  13
 14  42  43  44  18  20  19  45  46  15  16  17  21  22  23  25  24  26
```

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)`

2	1	3	4	5	6	7	8	9	11	12	10	13	14	15	16	17	18
19	20	21	22	23	25	24	26	27	28	29	30	31	32	33	34	35	36
37	38	40	39	41	42	2	1	3	4	5	6	7	8	9	11	12	10
13	14	15	16	17	18	19	20	21	22	23	25	24	26	27	28	29	30
31	32	33	2	1	3	4	5	6	7	8	9	11	12	10	13	14	15
34	35	36	16	17	18	19	20	37	38	21	22	23	25	40	24	26	27
28	29	30	31	39	32	2	1	3	4	5	6	33	41	7	9	8	11
42	12	10	13	14	15	34	16	17	18	19	20	21	22	23	25	24	26
27	28	29	35	30	31	32	36	37	2	1	38	3	4	5	6	33	7
8	9	11	40	12	10	13	39	14	15	16	17	18	34	19	41	20	42
21	22	23	25	24	26	27	28	29	35	30	31	32	36	2	1	37	3

A Glance at C++ 1y

Executors and Schedulers

executor [Austern et al., 2013]

```
class executor
{
public:
  virtual ~executor();
  virtual void add(function<void()> closure) = 0;
  virtual size_t num_pending_closures() const = 0;
};

// An executor that immediately executes any 'add'ed closure.
executor* singleton_inline_executor();
```

Executors and Schedulers

scheduled_executor [Austern et al., 2013]

```
class scheduled_executor
  : public executor
{
public:
  virtual void add_at(chrono::system_clock::time_point abs_time,
                      function<void()> closure) = 0;
  virtual void add_after(chrono::system_clock::duration rel_time,
                         function<void()> closure) = 0;
};

void set_default_executor(scheduled_executor* executor);

// Implementations are encouraged to ensure that separate tasks added
// to the initial default executor can wait on each other without
// deadlocks.
scheduled_executor* default_executor();
```

```
// Closures are executed in FIFO order in std::this_thread.
class loop_executor : public executor
{
public:
  // Does not spawn any threads.
  loop_executor();
  virtual ~loop_executor();

  // Execute the next closure (if there is one) and return.
  bool try_run_one_closure();

  // Run closures on this_thread until make_loop_exit() is called.
  void loop();

  // Run already queued closures only (or until make_loop_exit).
  void run_queued_closures();

  void make_loop_exit();
};
```

Executors and Schedulers

serial_executor [Austern et al., 2013]

```
class serial_executor
  : public executor
{
public:
  explicit serial_executor(executor* underlying_executor);
  virtual ~serial_executor();
  executor* underlying_executor();
};
```

- FIFO adapter
- runs its closures on a particular thread
- by scheduling its closures on another executor
- creates batches

Executors and Schedulers

thread_pool [Austern et al., 2013]

```
class thread_pool
  : public scheduled_executor
{
public:
  explicit thread_pool(int num_threads);
  ~thread_pool();
};
```

- a simple thread pool class
- *creates* a fixed number of threads
- multiplexes closures onto them

Executors and Schedulers

thread_pool [Gustafsson et al., 2013b]

```
template<class F, class... Args>
  future<typename result_of<F(Args...)>::type>
  async(executor& ex, F&& f, Args&&... args);
```

Threading Building Blocks

TBB: tbb::parallel_sort

```
template <typename RandomAccessIterator>
inline void parallel_sort(RandomAccessIterator begin,
                          RandomAccessIterator end)
{
  using value_type
    = typename std::iterator_traits<RandomAccessIterator>::value_type;
  parallel_sort(begin, end, std::less<value_type>());
}
```

TBB: tbb::parallel_sort

```
template <typename RandomAccessIterator, typename Compare>
void parallel_sort(RandomAccessIterator begin, RandomAccessIterator end,
                   const Compare& comp)
{
  if (begin < end)
    {
      const int min_parallel_size = 500;
      if (end - begin < min_parallel_size)
        std::sort(begin, end, comp);
      else
        internal::parallel_quick_sort(begin, end, comp);
    }
}
```

TBB: tbb::parallel_sort

```
namespace internal
{
  // Wrapper method to initiate the sort by calling parallel_for.
  template <typename RandomAccessIter, typename Comp>
  void
  parallel_quick_sort(RandomAccessIter begin, RandomAccessIter end,
                      const Comp& comp)
  {
    tbb::parallel_for
      (quick_sort_range<RandomAccessIter, Comp>(begin, end-begin, comp),
       quick_sort_body<RandomAccessIter, Comp>(),
       auto_partitioner());
  }
}
```

TBB: tbb::parallel_sort

```
namespace internal
{
  // Sort elements in a range that is smaller than the grainsize.
  template <typename RandomAccessIterator, typename Compare>
  struct quick_sort_body
  {
    using range_t = quick_sort_range<RandomAccessIterator, Compare>;
    void operator()(const range_t& range) const
    {
      std::sort(range.begin, range.begin + range.size, range.comp);
    }
  };
}
```

TBB: tbb::parallel_sort

```
template <typename RandomAccessIterator, typename Compare>
class quick_sort_range: private no_assign
{
public:
  const Compare &comp;
  RandomAccessIterator begin;
  size_t size;

  quick_sort_range(RandomAccessIterator b, size_t s, const Compare &c)
    : comp(c), begin(b), size(s)
  {}

  bool empty() const { return size == 0; }
  bool is_divisible() const
  {
    static const size_t grainsize = 500;
    return grainsize <= size;
  }
};
```

TBB: tbb::parallel_sort

```
quick_sort_range(quick_sort_range& range, split) : comp(range.comp)
{
  RandomAccessIterator array = range.begin;
  if (size_t m = pseudo_median_of_9(array, range))
    std::swap(array[0], array[m]);
  RandomAccessIterator key0 = range.begin;

  // Really partition...
  // array[0..j) is less or equal to key.
  // array(j..s) is greater or equal to key.
  // array[j] is equal to key.

  begin = array + (j + 1);
  size = range.size - (j + 1);
  range.size = j;
}
```

TBB: tbb::parallel_sort

```
using ints_t = std::vector<int>;
ints_t ints(n); // *Not* ints{n}!
std::generate(begin(ints), end(ints), std::rand);
{
  ints_t l{ints};
  chrono("std", n, [&]{ std::sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
{
  ints_t l{ints};
  chrono("tbb", n, [&]{ mytbb::parallel_sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
```

TBB: tbb::parallel_sort

```
using ints_t = std::vector<int>;
ints_t ints(n); // *Not* ints{n}!
std::generate(begin(ints), end(ints), std::rand);
{
  ints_t l{ints};
  chrono("std", n, [&]{ std::sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
{
  ints_t l{ints};
  chrono("tbb", n, [&]{ mytbb::parallel_sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
```

```
            std:  10000000    802.49ms
```

TBB: tbb::parallel_sort

```
using ints_t = std::vector<int>;
ints_t ints(n); // *Not* ints{n}!
std::generate(begin(ints), end(ints), std::rand);
{
  ints_t l{ints};
  chrono("std", n, [&]{ std::sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
{
  ints_t l{ints};
  chrono("tbb", n, [&]{ mytbb::parallel_sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
```

```
         std:  10000000   802.49ms
```

```
         tbb:  10000000   450.36ms
```

TBB: Many More Tools

Basic algorithms `parallel_for`, `parallel_reduce`, `parallel_scan`

Advanced algorithms `parallel_while`, `parallel_do`, `parallel_pipeline`, `parallel_sort`

Containers `concurrent_queue`, `concurrent_priority_queue`, `concurrent_vector`, `concurrent_hash_map`

Scalable memory allocation `scalable_malloc`, `scalable_free`, `scalable_realloc`, `scalable_calloc`, `scalable_allocator`, `cache_aligned_allocator`

Mutual exclusion `mutex`, `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, `queuing_rw_mutex`, `recursive_mutex`

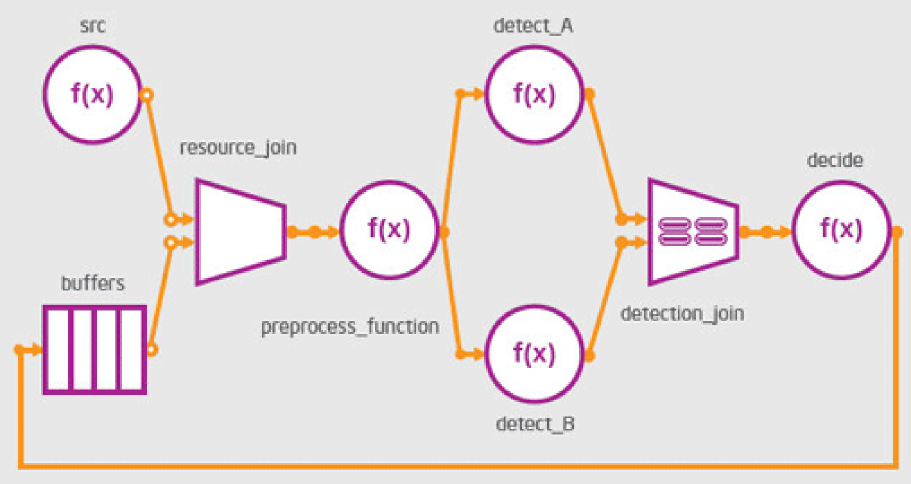
Atomic operations `fetch_and_add`, `fetch_and_increment`, `fetch_and_decrement`, `compare_and_swap`, `fetch_and_store`

Timing portable fine grained global time stamp

Task Scheduler direct access to control the creation and activation of tasks

TBB: Flow Graph

A simple feature detection application



http://www.drdobbs.com/231900177

Boost.Asio

Boost.Asio [Schäling, 2011]

```
boost::asio::io_service io_service;

boost::asio::deadline_timer
  timer1{io_service, boost::posix_time::seconds{3}},
  timer2{io_service, boost::posix_time::seconds{2}};

timer1.async_wait([](const boost::system::error_code&)
                  { std::cout << "World" << std::endl; });
timer2.async_wait([](const boost::system::error_code&)
                  { std::cout << "Hello, "; });



                            io_service.run();

;
```

```
Hello, World
```

Boost.Asio Multithreaded

```
boost::asio::io_service io_service;

boost::asio::deadline_timer
  timer1{io_service, boost::posix_time::seconds{3}},
  timer2{io_service, boost::posix_time::seconds{2}};

timer1.async_wait([](const boost::system::error_code&)
                  { std::cout << "World" << std::endl; });
timer2.async_wait([](const boost::system::error_code&)
                  { std::cout << "Hello, "; });

auto threads = std::vector<std::thread>{};
for (size_t i = 0; i < 2; ++i)
  threads.emplace_back([&]{ io_service.run(); });
for (auto& t: threads)
  t.join();
```

```
Hello, World
```

Boost.Asio Networking 1

```
#include <boost/asio.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <string>

boost::asio::io_service io_service;

int main()
{
  boost::asio::ip::tcp::resolver::query query{"localhost", "631"};
  boost::asio::ip::tcp::resolver resolver{io_service};
  resolver.async_resolve(query, resolve_handler);
  io_service.run();
}
```

Boost.Asio Networking 2

```
boost::asio::ip::tcp::socket sock{io_service};

void resolve_handler(const boost::system::error_code& ec,
                     boost::asio::ip::tcp::resolver::iterator it)
{
  if (!ec)
    sock.async_connect(*it, connect_handler);
}
```

Boost.Asio Networking 3

```
boost::array<char, 4096> buffer;

void connect_handler(const boost::system::error_code& ec)
{
  if (!ec)
    {
      const char buf[] = "GET / HTTP 1.1\r\nHost: localhost\r\n\r\n";
      boost::asio::write(sock, boost::asio::buffer(buf));
      sock.async_read_some(boost::asio::buffer(buffer), read_handler);
    }
}

void read_handler(const boost::system::error_code& ec, size_t size)
{
  if (!ec)
    {
      std::cout << std::string(buffer.data(), size) << std::endl;
      sock.async_read_some(boost::asio::buffer(buffer), read_handler);
    }
}
```

Boost.Asio Networking: It Works!

```
HTTP/1.0 400 Requete incorrecte
Date: Thu, 30 May 2013 14:40:54 GMT
Server: CUPS/1.6
Upgrade: TLS/1.0,HTTP/1.1
Content-Type: text/html; charset=utf-8
Content-Length: 362

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.
     org/TR/html4/loose.dtd">
<HTML>
<HEAD>
        <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
        <TITLE>Requete incorrecte - CUPS v1.6.2</TITLE>
        <LINK REL="STYLESHEET" TYPE="text/css" HREF="/cups.css">
</HEAD>
<BODY>
<H1>Requete incorrecte</H1>
<P></P>
</BODY>
</HTML>
```

Asynchronous I/O

The key is passing a function to call in the end.

Asynchronous I/O

The Future of Futures

And Then [Gustafsson et al., 2013b]

```
#include <future>

int main()
{
  std::future<int> f1 = std::async([]() { return 123; });

  std::future<string> f2
    = f1.then([](future<int> f)
      {
        return f.get().to_string(); // here .get() won't block
      });
}
```

Ready... [Gustafsson et al., 2013b]

```
#include <future>
int main()
{
  std::future<int> f1 = std::async([]{ return possibly_long(); });
  if (f1.ready())
    // No need to add continuation, process value right away.
    process_value(f1.get());
  else
    // Attach a continuation and avoid a blocking wait.
    f1.then([] (std::future<int> f2)
            {
              process_value(f2.get());
            });
}
```

Proposed Specifications

```
namespace std
{
  template <typename F>
  class future
  {
  public:
    // ...
    auto then(                  F&& func) -> future<decltype(func(*this))>;
    auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
    auto then(executor& ex,   F&& func) -> future<decltype(func(*this))>;
  };
}
```

[Gustafsson et al., 2013b, Austern et al., 2013]

Unwrap. . . [Gustafsson et al., 2013b]

```
#include <future>

int main()
{
  std::future<std::future<int>> outer
    = std::async([]
      {
        return std::async([] { return 1; });
      });

  std::future<int> inner = outer.unwrap();

  inner.then([](future f)
  {
    do_work(f);
  });
}
```

When any... [Gustafsson et al., 2013b]

```
using namespace std;

future<int> futures[] = { async([]{ return 125; }),
                          async([]{ return 456; }) };

using futures = future<vector<future<int>>>;
futures any = std::when_any(std::begin(futures), std::end(futures));

future<int> result
  = any.then([](futures f)
    {
      for (future<int> i : f.get())
        if (i.ready())
          return i.get();
    });
```

When all. . . [Gustafsson et al., 2013b]

```
using namespace std;

shared_future<int> fut1 = async([] { return 125; });
future<string>     fut2 = async([] { return "hi"; });

using futures = future<tuple<shared_future<int>, future<string>>>;
futures all = std::when_all(fut1, fut2);

future<int> result
  = all.then([](futures f)
    {
      return doWork(f.get());
    });
```

More possibilities

How about:

- for-loops
- while-loops
- etc.

Continuations for I/O [Gustafsson et al., 2013a]

```
future<int> f(shared_ptr<stream> str)
{
  shared_ptr<vector<char>> buf = ...;
  return str->read(512, buf)
         .then([](future<int> op)
          {
            return op.get() + 11;
          });
}

future<void> g()
{
  shared_ptr<stream> s = ...;
  return f(s).then([s](future<int> op)
              {
                s->close();
              });
}
```

- we might be building useless futures

Continuations for I/O [Gustafsson et al., 2013a]

```
future<int> f(shared_ptr<stream> str)
{
  shared_ptr<vector<char>> buf = ...;
  return str->read(512, buf)
         .then([](future<int> op)
          {
            return op.get() + 11;
          });
}

future<void> g()
{
  shared_ptr<stream> s = ...;
  return f(s).then([s](future<int> op)
              {
                s->close();
              });
}
```

- we might be building useless futures

await and resumable [Gustafsson et al., 2013a]

```
future<int> f(stream str) resumable
{
  shared_ptr<vector<char>> buf = ...;
  int count = await str.read(512, buf);
  return count + 11;
}

future<void> g() resumable
{
  stream s = ...;
  int pls11 = await f(s);
  s.close();
}
```

Asynchronous Copy [Gustafsson et al., 2013a]

```
auto write =
  [&buf](future<int> size) -> future<bool>
  {
    return streamW.write(size.get(), buf)
           .then([](future<int> op){ return 0 < op.get(); });
  };

auto future_false =
  [](future<int> op){ return future::make_ready_future(false); };

auto copy =
  do_while([&buf]() -> future<bool>
  {
    return streamR.read(512, buf)
           .choice([](future<int> op){ return 0 < op.get(); },
                   write, future_false);
  });
```

Asynchronous Copy [Gustafsson et al., 2013a]

```
int cnt = 0;
do
  {
    cnt = await streamR.read(512, buf);
    if (cnt == 0)
      break;
    cnt = await streamW.write(cnt, buf);
  }
while (0 < cnt);
```

Boost.Coroutine

```cpp
#include <iostream>
#include <boost/coroutine/coroutine.hpp>
using coro_t = boost::coroutines::asymmetric_coroutine<unsigned>;
```

```cpp
void fibo(coro_t::push_type& sink)
{
  unsigned fst = 1, snd = 1;
  sink(fst);
  sink(snd);
  for (;;)
    {
      unsigned res = fst + snd;
      fst = snd;
      snd = res;
      sink(res);
    }
};
```

```cpp
int main()
{
  auto source
    = coro_t::pull_type{fibo};

  for (unsigned i = 0; i < 15; ++i)
    {
      std::cout << source.get()
                << ' ';
      source();
    }
  std::cout << '\n';
}
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Boost.Coroutine

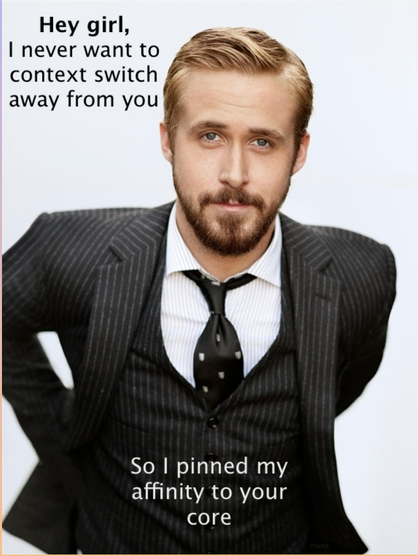
```
#include <iostream>
#include <boost/coroutine/coroutine.hpp>
using coro_t = boost::coroutines::asymmetric_coroutine<unsigned>;
```

```
auto fibo = coro_t::pull_type{
  [](coro_t::push_type& sink) {
    unsigned fst = 1, snd = 1;
    sink(fst);
    sink(snd);
    for (int i = 2 ; i < 15; ++i) {
        unsigned res = fst + snd;
        fst = snd;
        snd = res;
        sink(res);
      }
  }
};
```

```
int main()
{
  for (auto i : fibo)
    std::cout << i << ' ';
  std::cout << '\n';
}
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Coroutines


Hey girl,
I never want to
context switch
away from you
So I pinned my
affinity to your
core

ASIO Without Coroutines

```
namespace asio = boost::asio;

class session
{
public:
  session(asio::io_service& io_service)
    : socket_(io_service) // construct a TCP-socket from io_service
  {}

  tcp::socket& socket(){
    return socket_;
  }

  void start(){
    // initiate asynchronous read; handle_read() is callback-function
    socket_.async_read_some
      (asio::buffer(data_, max_length),
       boost::bind(&session::handle_read,this,
                   asio::placeholders::error,
                   asio::placeholders::bytes_transferred));
  }
```

ASIO With Coroutines

```
namespace asio = boost::asio;

void session(asio::io_service& io_service)
{
  // construct TCP-socket from io_service
  asio::ip::tcp::socket socket(io_service);

  try {
    for(;;) {
      // local data-buffer
      char data[max_length];

      boost::system::error_code ec;

      // read asynchronous data from socket
      // execution context will be suspended until
      // some bytes are read from socket
      auto length = socket.async_read_some(asio::buffer(data),
                                           asio::yield[ec]);
      if (ec == asio::error::eof)
        break; //connection closed cleanly by peer
```

Locks

Locks [Drepper, 2008]

```
long counter1, counter2;
time_t timestamp1, timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}
```

```
void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

- make it thread-safe
- use of a single lock is simple
- yet inefficient
- two locks won't do

Locks [Drepper, 2008]

```
long counter1, counter2;
time_t timestamp1, timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}
```

```
void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

- make it thread-safe
- use of a single lock is simple
- yet inefficient
- two locks won't do

Locks [Drepper, 2008]

```
long counter1, counter2;
time_t timestamp1, timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}
```

```
void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

- make it thread-safe
- use of a single lock is simple
- yet inefficient
- two locks won't do

Locks [Drepper, 2008]

```
long counter1, counter2;
time_t timestamp1, timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}
```

```
void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

- make it thread-safe
- use of a single lock is simple
- yet inefficient
- two locks won't do

Locks [Drepper, 2008]

```
long counter1, counter2;
time_t timestamp1, timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}
```

```
void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

- make it thread-safe
- use of a single lock is simple
- yet inefficient
- two locks won't do

Locks

```
std::mutex c1_mut, c2_mut, t1_mut, t2_mut;
using guard = std::lock_guard<std::mutex>;

void f1_1(long *r, time_t *t) {
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t) {
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    {
      guard gt2(t2_mut);
      *t = timestamp2;
    }
}
```

- this is wrong
- inconsistent results (bw `counter1` and `timestamp2`)

Locks

```
std::mutex c1_mut, c2_mut, t1_mut, t2_mut;
using guard = std::lock_guard<std::mutex>;

void f1_1(long *r, time_t *t) {
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t) {
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    {
      guard gt2(t2_mut);
      *t = timestamp2;
    }
}
```

- this is wrong
- inconsistent results (bw `counter1` and `timestamp2`)

Locks

```
std::mutex c1_mut, c2_mut, t1_mut, t2_mut;
using guard = std::lock_guard<std::mutex>;

void f1_1(long *r, time_t *t) {
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t) {
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    {
      guard gt2(t2_mut);
      *t = timestamp2;
    }
}
```

- this is wrong
- inconsistent results (bw counter1 and `timestamp2`)

Locks

```
void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}
```

```
void w1_2(long *r, time_t *t)
{
  guard gc1(c1_mut);
  guard gt2(t2_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

- this is wrong
- deadlocked!

Locks

```
void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}
```

```
void w1_2(long *r, time_t *t)
{
  guard gc1(c1_mut);
  guard gt2(t2_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

- this is wrong
- deadlocked!

Locks

```
void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}
```

```
void w1_2(long *r, time_t *t)
{
  guard gc1(c1_mut);
  guard gt2(t2_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

- this is wrong
- deadlocked!

Locks

```
void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}
```

```
void w1_2(long *r, time_t *t)
{
  guard gt2(t2_mut);
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

Mutexes: A Thread-Safe Queue

A Thread-Safe Queue

- `std::queue` forces a single lock for the whole queue
- make `push` and `pop` independent
- we *must* implement the queue ourselves

A Simple Queue [ccia6.4]

```
template <typename T> class queue
{
private:
  struct node
  {
    node(T d) : data(std::move(d)) {}
    T data;
    std::unique_ptr<node> next;
  };

  std::unique_ptr<node> head_; // pop point.
  node* tail_;                 // push point.

public:
  queue() : tail_{nullptr} {}
  queue(const queue& other) = delete;
  queue& operator=(const queue& other) = delete;
  // ...
};
```

A Simple Queue [ccia6.4]

```
void push(T new_value)
{
  std::unique_ptr<node> last{new node{std::move(new_value)}};
  node* const new_tail = last.get();
  if (tail_)
    tail_->next = std::move(last);
  else
    head_ = std::move(last);
  tail_ = new_tail;
}
```

A Simple Queue [ccia6.4]

```
std::shared_ptr<T> try_pop()
{
  if (head_)
    {
      const auto res = std::make_shared<T>(std::move(head_->data));
      const auto old_head = std::move(head_);
      head_ = std::move(old_head->next);
      if (old_head.get() == tail_)
        tail_ = nullptr;
      return res;
    }
  else
    return nullptr;
}
```

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

```
std::shared_ptr<T> try_pop() {
  // ...
  head_ = std::move(old_head->next);
  // ...
}

void push(T new_value) {
  // ...
  tail_->next = std::move(p);
  // ...
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

```
std::shared_ptr<T> try_pop() {
  // ...
  head_ = std::move(old_head->next);
  // ...
}

void push(T new_value) {
  // ...
  tail_->next = std::move(p);
  // ...
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

```
std::shared_ptr<T> try_pop() {
  // ...
  head_ = std::move(old_head->next);
  // ...
}

void push(T new_value) {
  // ...
  tail_->next = std::move(p);
  // ...
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

```
std::shared_ptr<T> try_pop() {
  // ...
  head_ = std::move(old_head->next);
  // ...
}

void push(T new_value) {
  // ...
  tail_->next = std::move(p);
  // ...
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Simple Queue: Mutex [ccia6.4]

```
template <typename T> class queue
{
  // ...
  std::unique_ptr<node> head_;
  node* tail_;
  // ...
};
```

```
void push(T new_value)
{
  // ...
  if (tail_)
    tail_->next = std::move(p);
  else
    head_ = std::move(p); // lock
  tail_ = new_tail;       // lock
}
```

```
std::shared_ptr<T> try_pop() {
  // ...
  head_ = std::move(old_head->next);
  // ...
}

void push(T new_value) {
  // ...
  tail_->next = std::move(p);
  // ...
}
```

- They might be the same object
- In which case it needs protection
- How do you check that?
- So you'd lock in every case

A Thread-Safe Queue

- much simpler to introduce a terminator, a sentinel, a dummy node
- pop from head, push at tail, enforce sentinel
- two locks, two mutexes

Queue with a Dummy Node [ccia6.5]

```
template <typename T> class queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };

  std::unique_ptr<node> head_;
  node* tail_;

public:
  queue()
    : head_(new node), tail_(head_.get())
  {}
  queue(const queue& other) = delete;
  queue& operator=(const queue& other) = delete;
  // ...
};
```

Queue with a Dummy Node [ccia6.5]

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> last{new node};
  node* const new_tail = last.get();
  // tail_ always exists.
  tail_->data = std::move(new_data);
  tail_->next = std::move(last);
  tail_ = new_tail;
}
```

Queue with a Dummy Node [ccia6.5]

```
void push(T new_value)
{
  std::unique_ptr<node> last{new node{std::move(new_value)}};
  node* const new_tail = last.get();
  if (tail_)
    tail_->next = std::move(last);
  else
    head_ = std::move(last);
  tail_ = new_tail;
}
```

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> last{new node};
  node* const new_tail = last.get();
  // tail_ always exists.
  tail_->data = std::move(new_data);
  tail_->next = std::move(last);
  tail_ = new_tail;
}
```

```cpp
std::shared_ptr<T> try_pop()
{
  if (head_.get() != tail_) // Non-empty list?
    {
      const auto res = std::move(head_->data);
      const auto old_head = std::move(head_);
      head_ = std::move(old_head->next);
      return res;
    }
  else
    return nullptr;
}
```

```cpp
std::shared_ptr<T> try_pop()
{
  if (head_)
    {
      const auto res = std::make_shared<T>(std::move(head_->data));
      const auto old_head = std::move(head_);
      head_ = std::move(old_head->next);
      if (old_head.get() == tail_)
        tail_ = nullptr;
      return res;
    }
  else
    return nullptr;
}
```

```cpp
std::shared_ptr<T> try_pop()
{
  if (head_.get() != tail_) // Non-empty list?
    {
      const auto res = std::move(head_->data);
      const auto old_head = std::move(head_);
      head_ = std::move(old_head->next);
      return res;
    }
  else
    return nullptr;
}
```

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for "last")
- data is stored in the *current* sentinel
- then a new one is appended
- `push` no longer acts upon `head`
- `try_pop` accesses both `head`/`tail` though, but just for a comparison
- neither `push`/`try_pop` perform heavy work on both

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for “last”)
- data is stored in the *current* sentinel
- then a new one is appended
- `push` no longer acts upon `head`
- `try_pop` accesses both `head`/`tail` though, but just for a comparison
- neither `push`/`try_pop` perform heavy work on both

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for "last")
- data is stored in the *current* sentinel
- then a new one is appended
- push no longer acts upon head
- try_pop accesses both head/tail though, but just for a comparison
- neither push/try_pop perform heavy work on both

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for "last")
- data is stored in the *current* sentinel
- then a new one is appended
- `push` no longer acts upon `head`
- `try_pop` accesses both `head`/`tail` though, but just for a comparison
- neither `push`/`try_pop` perform heavy work on both

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for "last")
- data is stored in the *current* sentinel
- then a new one is appended
- push no longer acts upon head
- try_pop accesses both head/tail though, but just for a comparison
- neither push/try_pop perform heavy work on both

Queue with a Dummy Node [ccia6.5]

- an extra level of indirection on data (for "last")
- data is stored in the *current* sentinel
- then a new one is appended
- `push` no longer acts upon `head`
- `try_pop` accesses both `head`/`tail` though, but just for a comparison
- neither `push`/`try_pop` perform heavy work on both

Fine Grained Thread Safe Queue: Structure [ccia6.6]

```
template <typename T> class threadsafe_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };

  std::mutex head_mutex_;
  std::unique_ptr<node> head_;
  std::mutex tail_mutex_;
  node* tail_;
  // ...
};
```

Fine Grained Thread Safe Queue: Push/Pop [ccia6.6]

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> last{new node};
  node* const new_tail = last.get();
  std::lock_guard<std::mutex> tail_lock{tail_mutex_};
  tail_->data = std::move(new_data);
  tail_->next = std::move(last);
  tail_ = new_tail;
}
```

Fine Grained Thread Safe Queue: Push/Pop [ccia6.6]

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> last{new node};
  node* const new_tail = last.get();
  // tail_ always exists.
  tail_->data = std::move(new_data);
  tail_->next = std::move(last);
  tail_ = new_tail;
}
```

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> last{new node};
  node* const new_tail = last.get();
  std::lock_guard<std::mutex> tail_lock{tail_mutex_};
  tail_->data = std::move(new_data);
  tail_->next = std::move(last);
  tail_ = new_tail;
}
```

Fine Grained Thread Safe Queue: Push/Pop [ccia6.6]

```
std::shared_ptr<T> try_pop()
{
  if (auto old_head = pop_head_())
    return old_head->data;
  else
    return nullptr;
}
```

Fine Grained Thread Safe Queue: Push/Pop [ccia6.6]

```
std::shared_ptr<T> try_pop()
{
  if (head_.get() != tail_) // Non-empty list?
    {
      const auto res = std::move(head_->data);
      const auto old_head = std::move(head_);
      head_ = std::move(old_head->next);
      return res;
    }
  else
    return nullptr;
}
```

```
std::shared_ptr<T> try_pop()
{
  if (auto old_head = pop_head_())
    return old_head->data;
  else
    return nullptr;
}
```

Fine Grained Thread Safe Queue: Private Functions [ccia6.6]

```
private:
  node* get_tail_()
  {
    std::lock_guard<std::mutex> tail_lock{tail_mutex_};
    return tail_;
  }

  std::unique_ptr<node> pop_head_()
  {
    std::lock_guard<std::mutex> head_lock{head_mutex_};
    if (head_.get() != get_tail_())
      {
        auto old_head = std::move(head_);
        head_ = std::move(old_head->next);
        return old_head;
      }
    else
      return nullptr;
  }
```

A Queue With Locks

- get_tail_ enforces order between accesses
- but we lack the blocking versions of pop!

A Queue With Locks

- get_tail_ enforces order between accesses
- but we lack the blocking versions of pop!

Condition Variables

Thread Safe Queue with Locking & Waiting

Private Parts [ccia6.7–10]

```
template<typename T>
class threadsafe_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };

  std::mutex head_mutex_;
  std::unique_ptr<node> head_;
  std::mutex tail_mutex_;
  node* tail_;
  std::condition_variable data_cond_;
  //...
};
```

Thread Safe Queue with Locking & Waiting

Private Parts

```
public:
  threadsafe_queue()
    : head_(new node), tail_(head_.get())
  {}
  threadsafe_queue(const threadsafe_queue& other) = delete;
  threadsafe_queue& operator=(const threadsafe_queue& other) = delete;
```

- establish the invariants

Thread Safe Queue with Locking & Waiting

Push

```
void push(T new_value)
{
  auto new_data = std::make_shared<T>(std::move(new_value));
  std::unique_ptr<node> dummy{new node};
  {
    std::lock_guard<std::mutex> tail_lock{tail_mutex_};
    tail_->data = new_data;
    node* const new_tail = dummy.get();
    tail_->next = std::move(dummy);
    tail_ = new_tail;
  }
  data_cond_.notify_one();
}
```

- release the lock asap

Thread Safe Queue with Locking & Waiting

Pop Private 1

```
private:
  node* get_tail_()
  {
    std::lock_guard<std::mutex> tail_lock{tail_mutex_};
    return tail_;
  }

  std::unique_ptr<node> pop_head_()
  {
    std::unique_ptr<node> res = std::move(head_);
    head_ = std::move(res->next);
    return res;
  }
```

- don't call pop_head_ on an empty queue
- lock calls to pop_head_

Thread Safe Queue with Locking & Waiting

Pop Private 2

```
std::unique_lock<std::mutex> wait_for_data_() {
  std::unique_lock<std::mutex> head_lock{head_mutex_};
  data_cond_.wait(head_lock, [&]{ return head_ != get_tail_(); });
  return std::move(head_lock);
}

std::unique_ptr<node> wait_pop_head_() {
  std::unique_lock<std::mutex> head_lock{wait_for_data_()};
  return pop_head_();
}

std::unique_ptr<node> wait_pop_head_(T& value) {
  std::unique_lock<std::mutex> head_lock{wait_for_data_()};
  value = std::move(*head_->data);
  return pop_head_();
}
```

- to factor, return the lock

Thread Safe Queue with Locking & Waiting

Wait and Pop

```
public:
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_ptr<node> const old_head = wait_pop_head_();
    return old_head->data;
  }

  void wait_and_pop(T& value)
  {
    std::unique_ptr<node> const old_head = wait_pop_head_(value);
  }
```

```
private:
  std::unique_ptr<node> try_pop_head_()
  {
    std::lock_guard<std::mutex> head_lock{head_mutex_};
    if (head_.get() == get_tail_())
      return nullptr;
    else
      return pop_head_();
  }

  bool try_pop_head_(T& value)
  {
    std::lock_guard<std::mutex> head_lock{head_mutex_};
    if (head_.get() != get_tail_())
      {
        value = std::move(*head_->data);
        pop_head_();
        return true;
      }
    else
      return false;
  }
```

Thread Safe Queue with Locking & Waiting

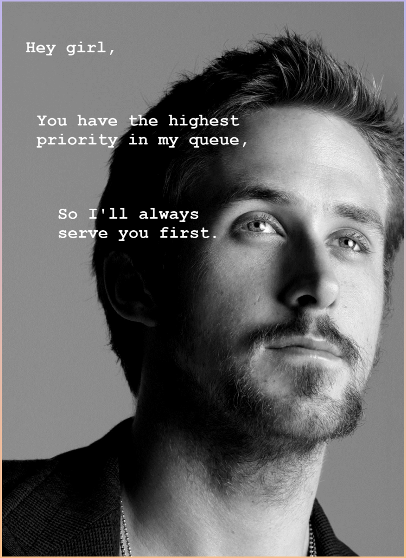
Try Pop

```
public:
  std::shared_ptr<T> try_pop()
  {
    if (std::unique_ptr<node> const old_head = try_pop_head_())
      return old_head->data;
    else
      return nullptr;
  }

  bool try_pop(T& value)
  {
    return try_pop_head_(value);
  }
```

That Was Not Easy. . . How About a Priority Queue?

That Was Not Easy... How About a Priority Queue?



Thread Local Storage

Plain concurrency

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

```
         plain:   1341273      9.36ms
```

A Mutex

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

```
     plain:   1341273     9.36ms
      lock:   3000000   9095.04ms
```

Thread-local

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

```
         plain:  1341273     9.36ms
          lock:  3000000  9095.04ms
  thread_local:  3000000     7.24ms
```

Databases

http://www.postgresql.org/docs/8.3/static/tutorial-transactions.html

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Transactions in Databases

http://www.postgresql.org/docs/8.3/static/tutorial-transactions.html

```
BEGIN;
```

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

```
COMMIT;
```

Database Adm under ACID

Atomicity: all operations are completed successfully *or* previous operations are rolled back to their former state.

Consistency: the database properly changes states upon a successfully committed transaction.

Isolation: transactions operate independently of and transparent to each other.

Durability: the result or effect of a committed transaction persists in case of a system failure.

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Transactions

- *data* oriented (not process oriented)
- easy to compose/nest
- no explicit locks
- fine grained
- no deadlocks
- but *conflicts*
- solve by *retry* or *rollback*
- many different implementations (lazy, eager)
- locks are fundamentally *pessimistic*
- TM is fundamentally *optimistic*: scalable!

Software Transactional Memory

Already supported by

- many managed languages
  - Scala
  - Clojure
- powerful environments
  - Haskell
  - Smalltalk
  - Fortress

Locks vs. STM [Drepper, 2008]

```
std::mutex c1_mut, c2_mut, t1_mut, t2_mut;
using guard = std::lock_guard<std::mutex>;

void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t)
{
  guard gt2(t2_mut);
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

```
void f1_1(long *r, time_t *t)
{
  __transaction_atomic
    {
      *t = timestamp1;
      *r = counter1++;
    }
}

void w1_2(long *r, time_t *t)
{
  __transaction_atomic
    {
      *r = counter1++;
      if (*r & 1)
        *t = timestamp2;
    }
}
```

Locks vs. STM [Drepper, 2008]

```cpp
std::mutex c1_mut, c2_mut, t1_mut, t2_mut;
using guard = std::lock_guard<std::mutex>;

void f1_1(long *r, time_t *t)
{
  guard gt1(t1_mut);
  guard gc1(c1_mut);
  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t)
{
  guard gt2(t2_mut);
  guard gc1(c1_mut);
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}
```

```cpp
void f1_1(long *r, time_t *t)
{
  __transaction_atomic
    {
      *t = timestamp1;
      *r = counter1++;
    }
}

void w1_2(long *r, time_t *t)
{
  __transaction_atomic
    {
      *r = counter1++;
      if (*r & 1)
        *t = timestamp2;
    }
}
```

Think about. . .

- implementing a double-linked list
- implementing a hash-table
- implementing a R&B tree
- buying Valium

Think about...

- implementing a double-linked list
- implementing a hash-table
- implementing a R&B tree
- buying Valium

Think about...

- implementing a double-linked list
- implementing a hash-table
- implementing a R&B tree
- buying Valium

Think about. . .

- implementing a double-linked list
- implementing a hash-table
- implementing a R&B tree
- buying Valium

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Implementation

- optimistic by nature
- perform and record the accesses
- if conflicts are detected, handle them
  - retry
  - beware of livelocks
- possibly provide a means to *wait* for a condition
- might even improve lock-based implementations
  (make it a transaction, and fail to lock-based on conflicts)
- many gory details
  - two versions of functions with transactions
  - penalty on each access
  - memory occupation
  - keep transactions small

Software Transactional Memory

- everybody is working on it
- GCC, clang, Intel, IBM
- hardware is coming (Intel & POWER)!
- Hardware Transaction Memory
- Hybrid Transaction Memory
- a nice talk: `http://www.youtube.com/watch?v=y906i0xtP8E`

STM in GCC 4.7+

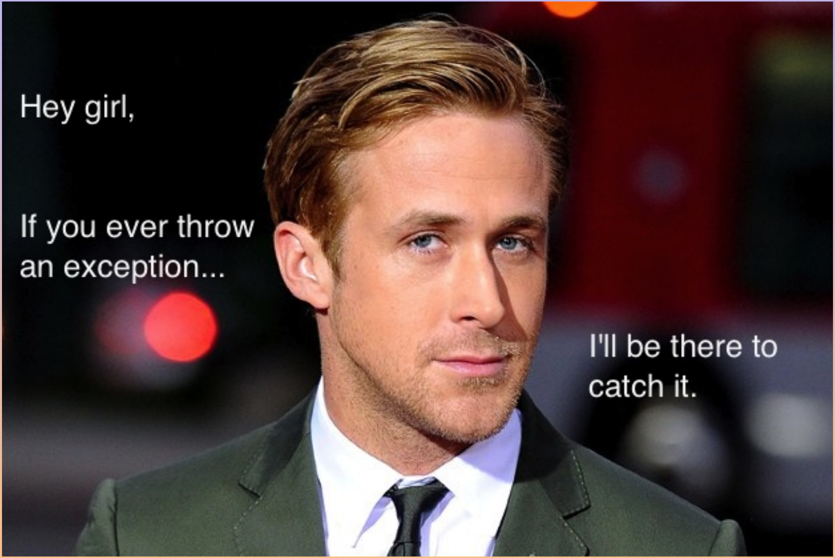
```
int shared{0};
std::vector<std::thread> threads;
auto chrono = make_clock("stm", shared);
for (size_t i = 0; i < nthreads; ++i)
  threads.emplace_back([&]{
      for (size_t j = 0; j < niters; ++j)
        __transaction_atomic {
          ++shared;
        }
    });
for (auto& t: threads)
  t.join();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

STM in GCC 4.7+

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

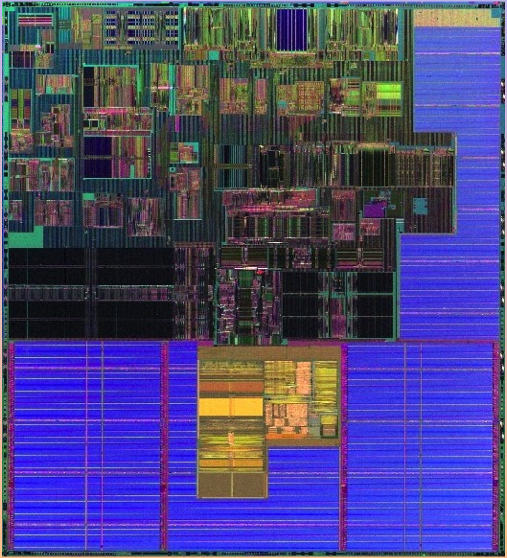
```
       plain:  1341273      9.36ms
      atomic:  3000000     58.34ms
        lock:  3000000   9095.04ms
         stm:  3000000    560.29ms
thread_local:  3000000      7.24ms
```

It Works with Exceptions!

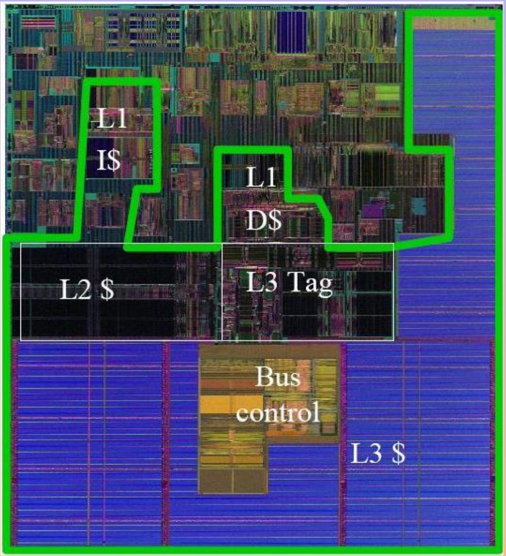


Why Do We Need All This?

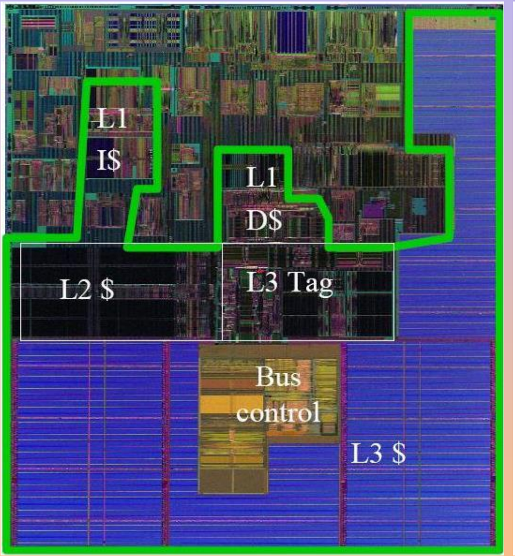
Who The Heck???



Who The Heck??? [Sutter, 2013]



Who The Heck??? [Sutter, 2013]



Sample Modern CPU

Original Itanium 2 had 211Mt, **85%** for cache:

16 KB L1I$ 16 KB L1D$
256 KB L2$ 3 MB L3$

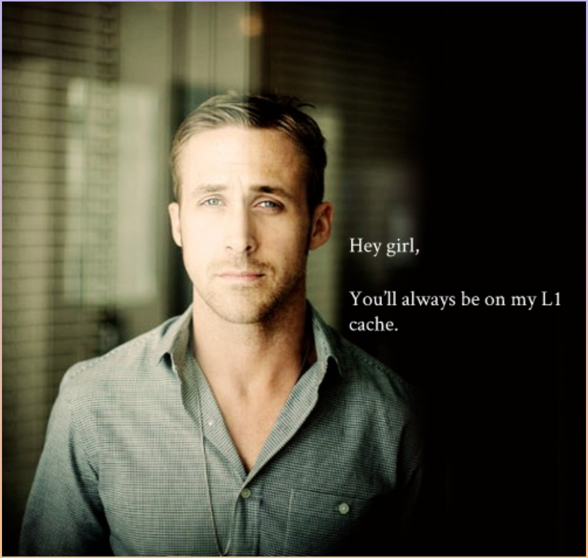
1% of die to compute,
99% to move/store data?

Itanium 2 9050:

Dual-core **24 MB L3$**

Source: David Patterson, UC Berkeley, HPEC keynote, Oct 2004 http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf

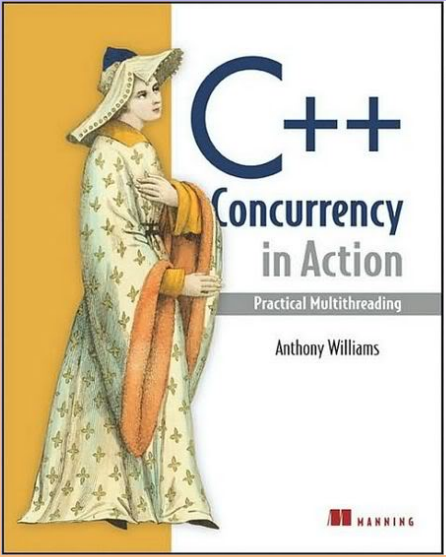
Caches

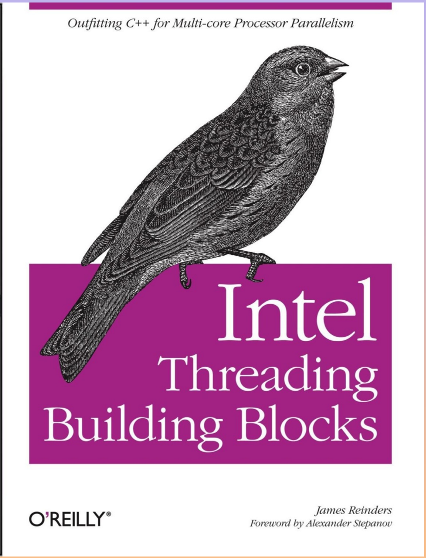


Cache Invalidation

References

Books


C++
Concurrency
in Action
Practical Multithreading
Anthony Williams
MANNING


Outfitting C++ for Multi-core Processor Parallelism
Intel
Threading
Building Blocks
O'REILLY®
James Reinders
Foreword by Alexander Stepanov

Bibliography I

Austern, M., Crowl, L., Carruth, C., Gustafsson, N., Mysen, C., and Yasskin, J. (2013).
N3562: Executors and schedulers, revision 1.
Technical Report ISO/IEC JTC1 SC22 WG21 N3562.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3562.pdf.

Drepper, U. (2008).
Parallel programming with transactional memory.
Queue, 6(5):38–45.

Gustafsson, N., Brewis, D., Sutter, H., and Mithani, S. (2013a).
N3564: Resumable functions.
Technical Report N3564.
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3564.pdf.

Bibliography II

Gustafsson, N., Laksberg, A., Sutter, H., and Mithani, S. (2013b).
N3558: A standardized representation of asynchronous operations.
Technical Report N3558.
http://isocpp.org/files/papers/N3558.pdf.

Schäling, B. (2011).
The Boost C++ Libraries.
XML Press.
http://en.highscore.de/cpp/boost/.

Sutter, H. (2013).
atomic<> weapons: The C++ memory model and modern hardware.
http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/.

Hey Girl. . .

Do you have questions?