HPCSE I

Multithreading

Threads and Processes

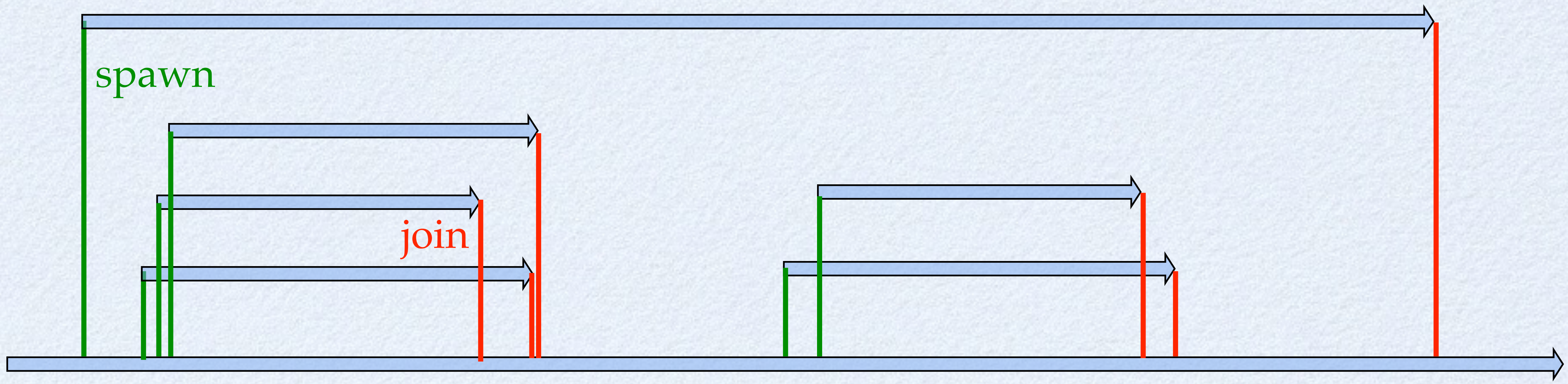
- Process – execution sequence within the OS, i.e. a program
  - Relatively expensive to create
  - Independent resources, state (by default)
  - Immune to many concurrency issues
- Thread – execution sequence within the process
  - main() is the first thread
  - Cheap to create
  - Shared resources, state
  - Difficult to use correctly

Concurrency Tradeoffs

- Pros
  - Compute faster
  - “Unblocking” – get work done while waiting for events to occur outside the CPU
- Cons
  - Synchronization overhead
  - Programming discipline – problems abound!
  - Harder to reason about
  - Harder to debug
- Don’t use threads unless you can’t avoid it!
- However, we can’t avoid it and thus have to learn it.

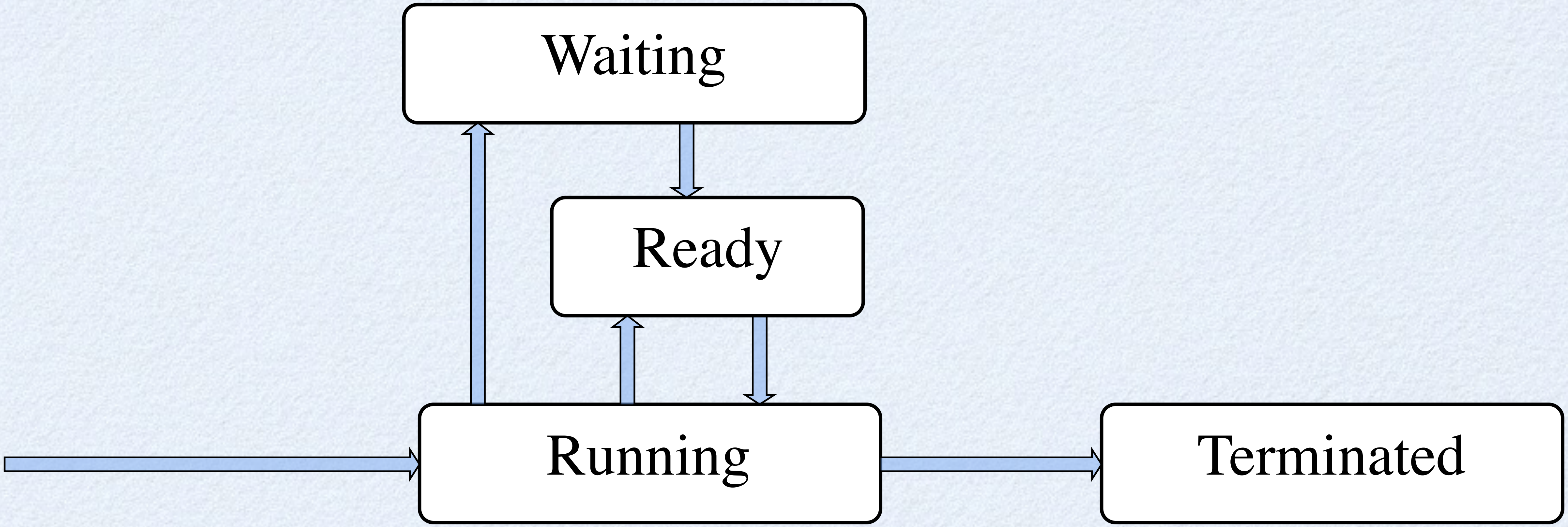
Spawning and joining threads

- During execution of a multithreaded program threads get spawned and joined dynamically:



Thread States

- **Running** – Currently executing. #running threads <= #CPU cores
- **Ready** – Prepared to run whenever processor time can be allocated to it. Not waiting.
- **Blocked** (or waiting) – Paused until some resource (other than processor) is allocated to it.
- **Terminated** – Finished execution but OS resources not yet deallocated.



Threading libraries

- Most operating systems have some kind of native threading library (pthreads on Unix, Win32 threads, ...). To achieve portable codes we want to use a cross-platform standard
- The **C++11** standard contains a cross-platform threading library
  - supported by g++-4.7, clang++-4.0 and MSVC11
- On other compilers
  - Use **Boost.Threads**, the predecessor to C++11 threads on many compilers
  - **Intel Thread Building Blocks (TBB)** with Intel C++, based on draft version of the C++11 standard with its own extensions
- We will use the **C++11** subset supported by most modern compilers

Compiling C++11 codes

- With **g++** use -pthread or you will get runtime errors:
  - **g++ -std=c++11 -pthread ...**
- With **clang** specify that we want to use the clang standard library and not the gnu version:
  - For clang 4.0: **clang++ -std=c++11 -stdlib=libc++ ...**
  - For clang 3.x: **clang++ -std=c++0x -stdlib=libc++ ...**
- Does anyone want to use MSVC compilers on Windows?
- Example sources are available by git:
  - git clone https://gitlab.phys.ethz.ch/hpcse_hs14/lecture.git

Launching and joining threads

- A thread is launched by passing a function (and optionally its arguments) to the thread constructor:

```
std::thread t (foo, arg1);
```

- We can also use C++11 lambda functions:

```
std::thread t ([] () { std::cout << “Hello world!\n”;});
```

- Threads are joined calling the join function:

```
t.join();
```

std::thread (abridged)

```
class thread
{
 public:
    thread() noexcept;
    ~thread();

    void swap(thread& x);
    thread& operator=(thread&&) noexcept;
    // …move support but noncopyable…

    typedef platform-specific-type
    native_handle_type;
    native_handle_type native_handle();

    // launch
    template <class F>
    explicit thread(F f);

    template <class F,class A1,class A2,...>
    thread(F f,A1 a1,A2 a2,...);

    void join();

    bool joinable() const;  // still attached?
    void detach();

    static unsigned hardware_concurrency();

    class id;
    id get_id() const noexcept;

    static void yield();
    static void sleep(const system_time& xt);
};
```

Movable/Noncopyable Types

- Can't copy/assign from lvalues
  - `std::thread x, y;  x = y; // error!`
- Can place in C++11 containers...
  - `std::vector<std::thread> v(10); // ok!`
- Can "copy"/assign from rvalues
  - `pool[3] = std::thread(f);`
- Can pass to/return from functions
  - `std::thread t = make_thread();`
  - `do_something(make_thread());`
- Can swap
  - `x.swap(y);`
  - `swap(x,y);`

Detaching and destroying threads

- The detach() member function let's the thread run on, but the object no longer refers to it.

- Destroying running threads is different in C++11 and Boost:
  - **Boost** silently detaches a joinable (still running) thread
  - **C++11** terminates the program if the thread is joinable
  - The reason is that a detached thread might be a security hole or bug

Example: Integration using Simpson's rule

simpson.hpp

```
#include <cassert>
#include <functional>

inline double
simpson(std::function<double(double)> f,
        double a, double b, unsigned int N)
{
    assert (b>=a);
    assert (N!=0u);

    double h=(b-a)/N;

    // boundary values
    double result = ( f(a) + 4*f(a+h/2) +
                      f(b) ) / 2.0;

    // values between boundaries
    for ( unsigned int i = 1; i <= N-1; ++i )
        result += f(a+i*h) + 2*f(a+(i+0.5)*h);

    return result * h / 3.0;
}
```

simpson_serial.cpp

```
#include "simpson.hpp"
#include <cmath>
#include <iostream>


double func(double x)
{
  return x * std::sin(x);
}


int main()
{
  double a;    // lower bound of integration
  double b;    // upper bound of integration
  unsigned int nsteps; // number of subintervals

  // read the parameters
  std::cin >> a >> b >> nsteps;

  // print the result
  std::cout << simpson(func,a,b,nsteps)
            << std::endl;

  return 0;
}
```

Simpson's rules using two threads

simpson_threaded1.cpp

```
#include "simpson.hpp"
#include <cmath>
#include <iostream>
#include <thread>

double func(double x) { return x * std::sin(x); }

int main()
{
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration

  std::cin >> a >> b >> nsteps;

  double result1;  // the integral of the first half

  // spawn a thread for the first half of the interval
  std::thread t( [&] () { result1 = simpson(func,a,a+(b-a)/2.,nsteps/2);} );

  // locally integrate the second half
  double result2 = simpson(func,a+(b-a)/2.,b,nsteps/2);

  t.join();           // wait for the thread to join
  std::cout << result1 + result2 << std::endl;

  return 0;
}
```

Futures

- This worked but was clumsy:
  - we needed to declare a variable to hold the return type
  - we needed to create a (lambda) function to fill it
- Futures hold future return values of a function called asynchronously in a thread.
  - we use the future to specify the return value
  - we can access its value after the asynchronous call finishes

Simpson's rules using a future

simpson_threaded2.cpp

```
#include "simpson.hpp"
#include <cmath>
#include <iostream>
#include <thread>
#include <future>

double func(double x) { return x * std::sin(x); }

int main()
{
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration
  std::cin >> a >> b >> nsteps;

  // create a packaged task
  std::packaged_task<double()> pt(std::bind(simpson,func,a,a+(b-a)/2.,nsteps/2));
  std::future<double> fi = pt.get_future(); // get the future return value
  std::thread t (std::move(pt));            // launch the thread

  double result2 = simpson(func,a+(b-a)/2.,b,nsteps/2);

  fi.wait(); // wait for the task to finish and the future to be ready

  std::cout << result2 + fi.get() << std::endl;
  t.join();

  return 0;
}
```

Simpson’s rules using a future

simpson_threaded2.cpp

```
#include "simpson.hpp"
#include <cmath>
#include <iostream>
#include <thread>
#include <future>

double func(double x) { return x * std::sin(x); }

int main()
{
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration
  std::cin >> a >> b >> nsteps;

  // create a packaged task
  std::packaged_task<double()> pt(std::bind(simpson,func,a,a+(b-a)/2.,nsteps/2));
  std::future<double> fi = pt.get_future(); // get the future return value
  std::thread t (std::move(pt));            // launch the thread

  double result2 = simpson(func,a+(b-a)/2.,b,nsteps/2);

  std::cout << result2 + fi.get() << std::endl; // fi.get() will wait for the result

  t.join();

  return 0;
}
```

Asynchronous function calls

- even simpler are explicit asynchronous calls

```
std::future<double> fi = std::async(simpson,a,b,n);

std::cout << fi.get() << std::endl; // get blocks automatically
```

- but this might or might not run in a new thread.
- better might be to force an asynchronous call:

```
std::future<double> fi=std::async(std::launch::async, simpson,a,b,n);

std::cout << fi.get() << std::endl;
```

Simpson's rules using asynchronous calls

simpson_threaded4.cpp

```
#include "simpson.hpp"
#include <cmath>
#include <iostream>
#include <thread>
#include <future>

double func(double x) { return x * std::sin(x); }

int main()
{
  double a;           // lower bound of integration
  double b;           // upper bound of integration
  unsigned int nsteps; // number of subintervals for integration
  std::cin >> a >> b >> nsteps;

  // even easier: launch an asynchronous function call
  // force it to be asynchronous and thus in a seperate thread
  std::future<double> fi = std::async(std::launch::async,simpson,func,a,a+(b-a)/2.,nsteps/2);

  // locally integrate the second half
  double result = simpson(func,a+(b-a)/2.,b,nsteps/2);

  std::cout << result + fi.get() << std::endl;

  return 0;

}
```

The running thread in C++11

- Information about the thread is in the namespace std::this_thread:

```
namespace std {
  namespace this_thread {
    thread::id get_id() noexcept;

    void yield() noexcept;

    template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>  t) noexcept;

  template <class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& t) noexcept;
  }
}
```

Calculating π through a series

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


// sum terms [i-j) of the power series for
// pi/4
void sumterms(long double& sum,
              std::size_t i, std::size_t j)
{
  sum = 0.0;

  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}
```

$$\arctan 1 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}$$

```
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  std::vector<long double> results(nthreads);

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(results[i]),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  long double pi = 4 * std::accumulate(
              results.begin(), results.end(), 0.);

  std::cout << "pi=" << std::setprecision(18)
            << pi << std::endl;

  return 0;
}
```

But why keep so many return values?

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


// sum terms [i-j) of the power series for
// pi/4
void sumterms(long double& sum,
              std::size_t i, std::size_t j)
{
  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}
```

```cpp
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  // let us just use a single result
  long double result=0.;

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(result),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  std::cout << "pi=" << std::setprecision(18)
            << 4.*result << std::endl;

  return 0;
}
```

Do you see a problem?

Threading and race conditions

- In threaded programs, we must stop other threads from looking (or touching) data that we need. Requires cooperation!
- A concurrent program that doesn't control visibility of broken invariants has a **race condition**
- From the point of view of threading, even an int has an invariant that is broken during mutation

Thread Safety: Serializing Access

- Basic mechanism: **mutex**
- Associated with some shared mutable data, which may be
  - as small as a char
  - as large as a list<vector<string> > (or larger)
- At any time, a mutex is either **locked by one thread** or **unlocked**.
- When a thread asks to lock a mutex
  - If the mutex is unlocked, it becomes locked and the thread proceeds
  - If the mutex is locked, the thread is blocked until the lock is released and reallocated to the locking thread.
- Protocol – threads agree to:
  - acquire a lock on the mutex before accessing the data
  - release the lock when done accessing the data

Locks

- Movable/noncopyable. Expresses ownership of a thread
- Forgetting to unlock a mutex will cause the next thread that locks it to wait forever
- Use RAII (resource acquisition is initialization) lock objects to eliminate this problem:
    - Constructor locks (acquires) the mutex
    - Destructor unlocks (releases) it
    - Very safe!
- Note: one lock object should never be accessed by multiple threads!

Safety through mutex and lock_guard

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


// sum terms [i-j) of the power series for
// pi/4
void sumterms(
  std::pair<long double, std::mutex>& result,
  std::size_t i, std::size_t j
)
{
  long double sum=0.;

  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);

  std::lock_guard<std::mutex> l (result.second);
  result.first += sum;
}
```

```
int main()
{
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
          std::thread::hardware_concurrency());

  std::vector<std::thread> threads(nthreads);
  // let us just use a single result
  std::pair<long double, std::mutex> result;
  result.first = 0.;

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) / nthreads;

  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] =std::thread(
        sumterms, std::ref(result),
        i * step, (i+1) * step
      );

  for (std::thread& t : threads)
    t.join();

  // no need to lock here
  std::cout << "pi=" << std::setprecision(18)
            << 4.*result << std::endl;

  return 0;
}
```

Now we are safe

Example: garbled I/O

garbledio.cpp

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

void printer( int n )
{
  for ( int i = 0; i < 100; ++i)
    std::cout << "do not garble thread " << n << ": " << i << std::endl;
}


int main()
{
  std::vector<std::thread> threads;

  for (int n = 1; n < 10; ++n)
    threads.push_back(std::thread(printer, n));

  for (std::thread& t : threads)
    t.join();
}
```

Example: synchronized I/O

syncedio.cpp

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


std::mutex io_mutex;  // global

struct sync
{
  sync( std::ostream& os )
  : os(os)
  , lock(io_mutex) {}

  template <class T>
  std::ostream& operator<<(T const& x)
  {
    return os << x;
  }

private:
  std::ostream& os;
  std::lock_guard<std::mutex> lock;
};
```

```
void printer( int n )
{
  for ( int i = 0; i < 100; ++i) {
    sync(std::cout)
    << "do not garble thread "
    << n << ": " << i << std::endl;
  }
}


int main()
{
  std::vector<std::thread> threads;

  for (int n = 1; n < 10; ++n)
    threads.push_back(std::thread(printer, n));

  for (std::thread& t : threads)
    t.join();
}
```

Mutexes and Locks

- We have four basic mutex types
  - **mutex**
  - **recursive_mutex**: allows multiple locking by the same thread
  - **timed_mutex**: allows time-outs in lock attempts
  - **recursive_timed_mutex**: both of the above
  - We need to use a timed mutex for timed locks
- Lock types:
  - **lock_guard**
  - **unique_lock**

unique_lock

- The unique_lock is more flexible and allows deferring the lock
  - unique_lock<mutex> l(m); // locks the lock
  - unique_lock<mutex> l(m,std::adopt_lock); // adopts the lock state
  - unique_lock<mutex> l(m, std:: defer_lock); // does not lock yet
  - unique_lock<mutex> l(m, std:: try_to_lock); // tries to lock
  - unique_lock<mutex> l(m,*abs_time*); // tries to lock, with timeout
- And it has some important functions:
  - l.owns_lock(); // returns whether It is locked
  - if (l) ... // tests whether locked

unique_lock (continued)

- It can be locked later:
  - **l.try_lock**(); // tries to lock and returns whether is succeeded
  - **l.try_lock_for**(*rel_time*); // tries to lock with timeout
  - **l.try_lock_until**(*abs_time*); // tries to lock with timeout
  - **l.lock**(); // locks the lock
  - **l.unlock**();
  - **std::lock**(l1,l2); **std::lock**(l1,l2,l3); ... // lock multiple locks at the same time
- The timed locks and time specification are slightly different in Boost.Thread .

Example: Pairwise Associations

```
struct collab
{
  collab() : partner(0) {}
  ~collab() { decouple(); }

  void couple(collab* new_partner);
  void decouple();

private:
  collab* partner;
  std::mutex gate;
};

typedef std::lock_guard<std::mutex> guard;

struct lock2
{
  lock2(std::mutex& a, std::mutex& b)
  : l0( a ),
  l1( b )
  {}
  guard l0, l1;
};
```

```
void collab::couple(collab* other)
{
  decouple();
  other->decouple();
  lock2 g(gate,other->gate);
  if (partner || other->partner) return;
  partner = other;
  other->partner = this;
}

void collab::decouple()
{
  collab* cur;
  {
    guard g0(gate);
    cur = partner;
    if (!cur) return;
  }

  lock2 g(gate,cur->gate);
  if (partner != cur) return;
  partner = 0;
  cur->partner = 0;
}
```

Deadlock: The Deadly Embrace

- Once you have synchronization, you can also have **deadlock**
- Scenario:
  - Mutexes 1 and 2, unlocked
  - Thread A locks mutex 1 — A still running
  - Thread B locks mutex 2 — B still running
  - Thread A locks mutex 2 — A waits (for B)
  - Thread B locks mutex 1 — B waits (for A)
- Solution:
  - We need to lock both in the same order
  - Introduce an ordering on mutexes, e.g. by address
  - Easier solution: use **std::lock()** function with multiple mutexes

We need to lock both at the same time

```
struct collab
{
  collab() : partner(0) {}
  ~collab() { decouple(); }
  void couple(collab* new_partner);
  void decouple();
private:
  collab* partner;
  std::mutex gate;
};

typedef std::unique_lock<std::mutex> guard;
```

```
void collab::couple(collab* other)
{
  decouple();
  other->decouple();
  guard g1(gate,defer_lock);
  guard g2(other->gate,defer_lock);
  std::lock(g1,g2); // lock both simultaneously
  if (partner || other->partner) return;
  partner = other;
  other->partner = this;
}

void collab::decouple()
{
  collab* cur;
  {
    guard g0(gate);
    cur = partner;
    if (!cur) return;
  }

  guard g1(gate,defer_lock());
  guard g2(cur->gate,defer_lock());
  std::lock(g1,g2); // lock both simultaneously
  if (partner != cur) return;
  partner = 0;
  cur->partner = 0;
}
```

std::call_once

- "Once routines"
  - Executed once, no matter how many invocations
  - No invocation will complete until the one execution finishes
  - Typical use: initialization of static and function-static data
- Protocol:
  - Declare a global (namespace scope) once_flag for each once routine
  - Invoke the once routine indirectly by passing its address and once_flag to call_once.

```
std::once_flag printonce_flag;

void printonce() { std::cout << "This should be printed only once\n"; }

int main()
{
  std::vector<std::thread> threads;
  for (int n = 0; n < 10; ++n)
    threads.push_back(
        std::thread([&](){std::call_once(printonce_flag,printonce);}));

  for (std::thread& t : threads)
    t.join();

  return 0;
```