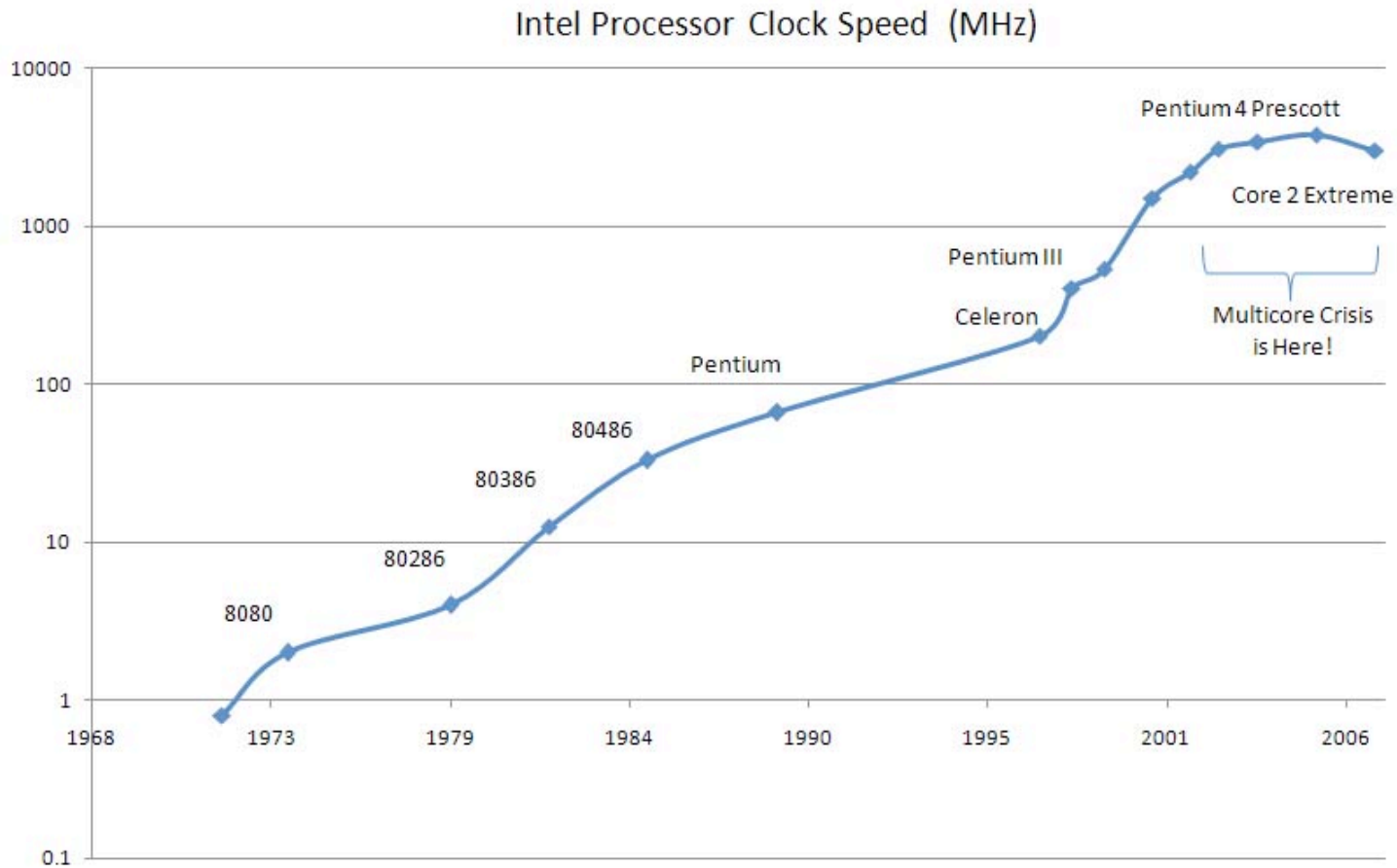
boostpro computing

Concurrency

Using Boost.Threads

Kids, the Fun is Over



boostpro computing

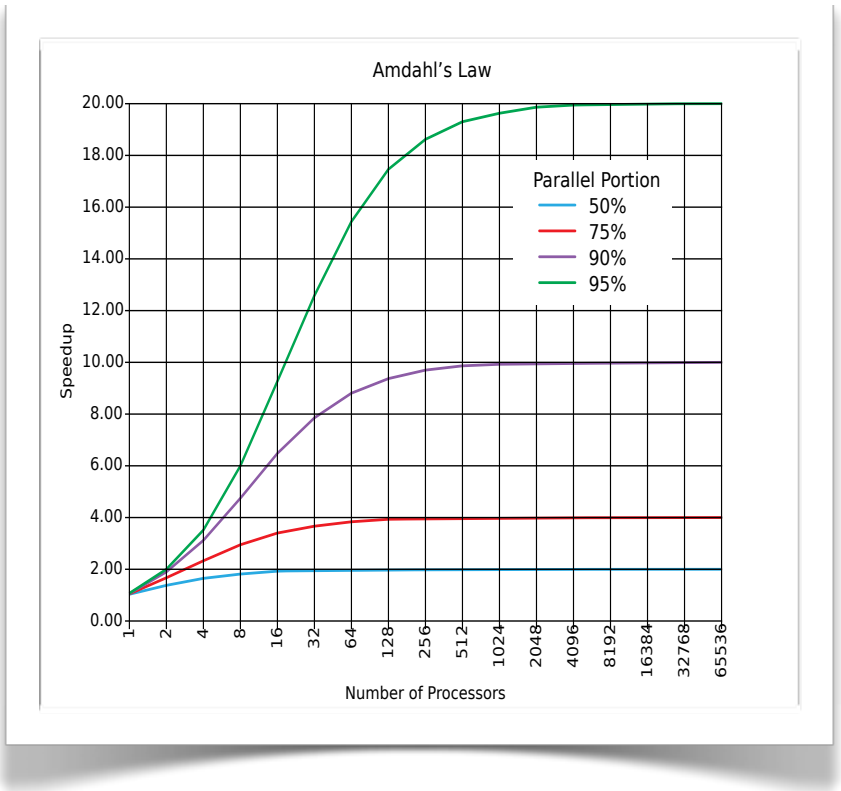
It's a parallel world

- You program while your boss chats on the phone
- Quad core processors today
- Beside which, the speed of light is a problem
- So pretty soon, we'll be running 64 cores.

Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

- n: processors or cores
- p: parallelizability



boostpro computing

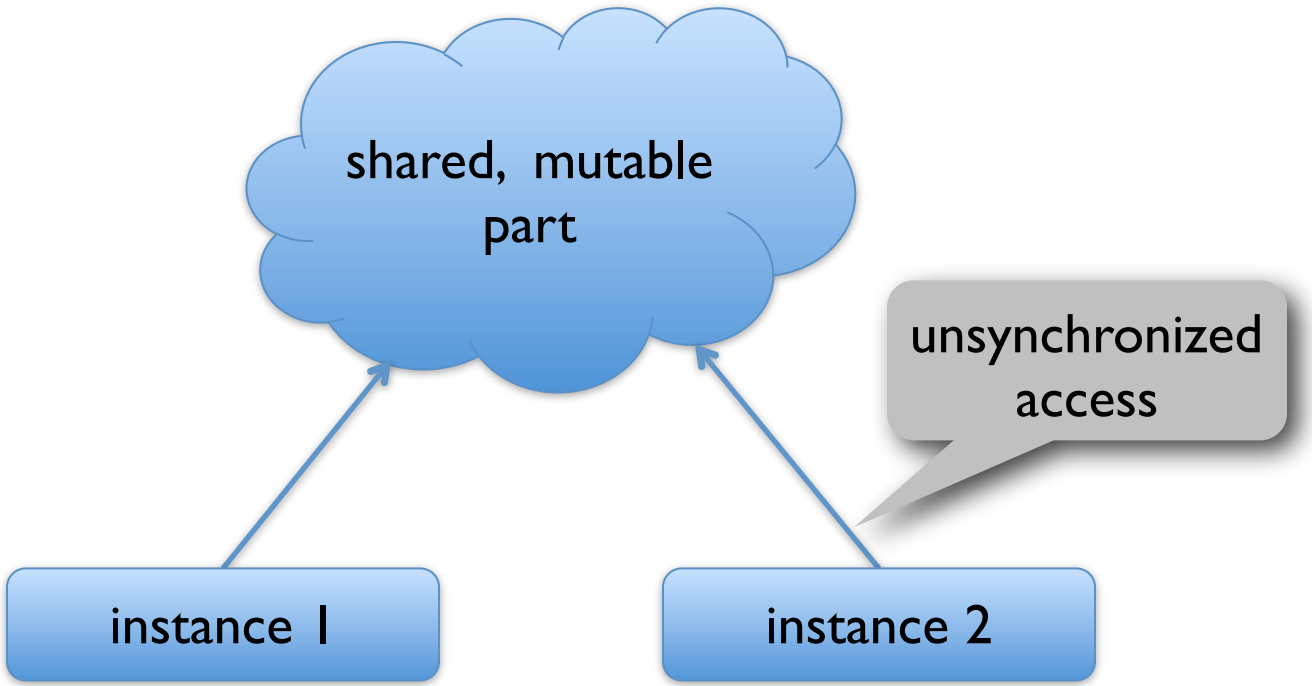
Efficiency Matters

- Waiting for CPUs to get faster no longer works
- Waiting for more cores doesn't work either
- Don't trade away cycles early!
- To maximize parallelizability, minimize synchronization
- All good news for C++

Basic Thread Safety

- Minimal requirement for multithreaded code
  - A single instance can be read concurrently
  - Distinct instances can be read/written concurrently
- `int` is thread-safe
- `std::vector<std::string>` is thread-safe

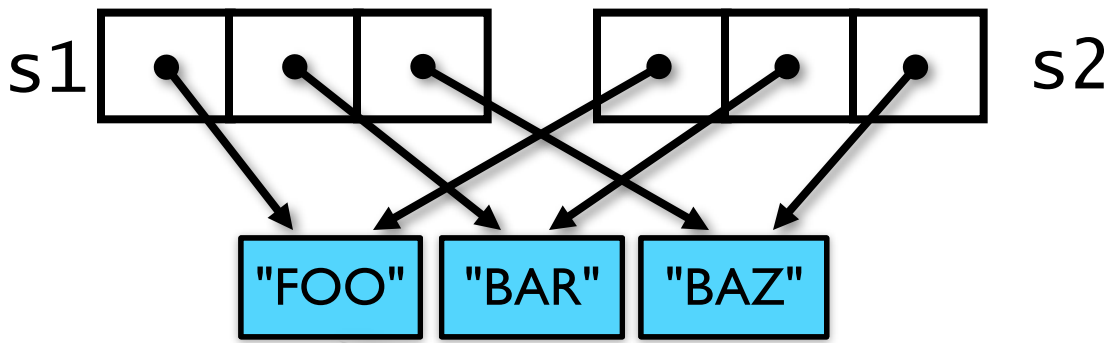
A Non-Thread-Safe Type



boostpro computing

Incidental Data Structures

```
typedef vector<shared_ptr<string> > svec;
...
svec s2 = s1;
std::for_each( s2.begin(), s2.end(), uppercase );
```



Expected whole/part relationship broken!

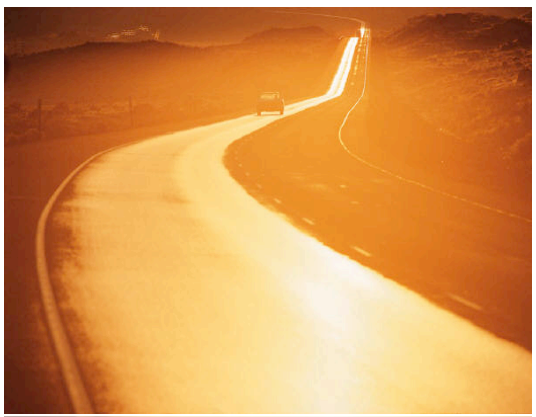
boostpro computing

The Legacy of OOP

- Handle subtypes through a pointer to base
  - Subtypes have different sizes
  - which implies dynamic allocation
  - which implies lifetime management
  - which leads to shared_ptrs and GC
  - which encourage shared ownership (and incidental data structures—and incidental algorithms!)
  - which leads to synchronization (and bugs, and inefficiency) ☹

Roads to Thread-Safety

- Share liberally and synchronize access
  - Limits parallelizability
- Share only immutable data
  - Usually requires garbage collection
- Avoid mutating data
  - Total immutability is a costly abstraction
- Avoid sharing data
  - Let's talk more about this one



boostpro computing

Value Semantics

- Copies are equal and logically disjoint
- Equality means substitutability
- Assignment makes one object into a copy of the other
- Nice properties
  - Referential transparency, equational reasoning
  - Naturally thread-safe

“What, Me Worry?”

```
typedef vector<string> index_t;
index_t get_index();

index_t const index = get_index();
```

$O(N)$?

boostpro computing

“Chickening Out”

```
typedef vector<string> index_t;
void get_index( index_t& i );

index_t index;
get_index( index );
```

distraction

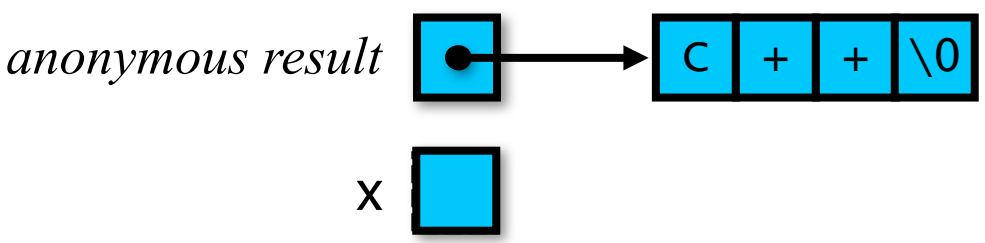
mutation

boostpro computing

Observation

- Function return values are anonymous temporary objects, or "rvalues"
- Such an object can only be used once in an expression
  - how would you refer to it a second time?
- The contents of the return value could be used directly by the caller (without creating an expensive copy)

```
std::string x = f();
```



boostpro computing

Return Value Optimization (RVO)

- Callee can construct result directly in caller's stack frame
- The copy is elided (edited out) for pure speed
- Most compilers perform this optimization today
- Also works for rvalues passed as function arguments!

boostpro computing

Reference Counting?

- Copy construction and assignment using reference counting is very fast.
- Q: What could be faster?
- A: Not counting (especially if you need atomics)
- Also, reference semantics obfuscates code
- Also, bad for generic code
  - Good idea: reference counting vectors
  - Very expensive idea: reference counting ints

Quotation

"Cheap assignment and function return are no longer a sufficient reason for the more complex shared ownership model."

– Howard Hinnant

Making a Type Movable

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p)
    {
        if (this != &p)
        {
            T* tmp = p.ptr_ ? p.ptr_->clone() : 0;
            delete ptr_;
            ptr_ = tmp;
        }
        return *this;
    }
private:
    T* ptr_;
};
```


boostpro computing

Making a Type Movable

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p) {…}

private:
    T* ptr_;
};
```


boostpro computing

Move Constructor

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p) {…}
    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

private:
    T* ptr_;
};
```


boostpro computing

Move Assignment

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p) {…}
    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(clone_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    T* ptr_;
};
```


boostpro computing

Using Boost.Move

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p) {…}
    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(clone_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:

    T* ptr_;
};
```


boostpro computing

Using Boost.Move

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(BOOST_COPY_ASSIGN_REF(clone_ptr) p) {…}
    clone_ptr(BOOST_RV_REF(clone_ptr) p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(BOOST_RV_REF(clone_ptr) rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    BOOST_COPYABLE_AND_MOVABLE(clone_ptr)
    T* ptr_;
};
```

boostpro computing

Back to C++11

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p) {…}
    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(clone_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    T* ptr_;
};
```


boostpro computing

Disabling Copy…

```
template <class T>
struct clone_ptr
{
    clone_ptr(const clone_ptr& p)
      : ptr_(p.ptr_ ? p.ptr_->clone() : 0) {}
    clone_ptr& operator=(const clone_ptr& p) {…}

    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(clone_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    T* ptr_;
};
```

boostpro computing

Disabling Copy…

```
template <class T>
struct clone_ptr
{
 private:
    clone_ptr(const clone_ptr& p);
    clone_ptr& operator=(const clone_ptr& p) {…}
 public:
    clone_ptr(clone_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    clone_ptr& operator=(clone_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    T* ptr_;
};
```


boostpro computing

Move-Only: unique_ptr

```
template <class T>
struct unique_ptr
{
 private:
    unique_ptr(const unique_ptr& p);
    unique_ptr& operator=(const unique_ptr& p);
 public:
    unique_ptr(unique_ptr&& p)
      : ptr_(p.ptr_) { p.ptr_ = 0; }

    unique_ptr& operator=(unique_ptr&& rhs)
    {
        delete ptr_; ptr_ = rhs.ptr_;
        rhs.ptr_ = 0;
        return *this;
    }
private:
    T* ptr_;
};
```


boostpro computing

unique_ptr Examples

```
typedef unique_ptr<int> ptr;
ptr lval( new int(42) );
ptr p2 = lval;                // compile-time error
ptr p3 = std::move( lval ); // OK, moves

ptr rval();
ptr p4 = rval();              // OK, moves

std::vector<ptr> v;
v.push_back( rval() );        // OK, moves
```

- Unlike std::auto_ptr:
  - move-only types don't move *implicitly* from lvalues
  - move-only types can be stored in std containers

boostpro computing

Exercise

- Create a class X whose constructors (including copy constructor), assignment operator, and destructor are instrumented so you can keep track of what's happening
- Create and call the following functions, with both lvalue and rvalue arguments:

```
void f(X) {}

X g() { return X(); }

X h(X a) { return a; }
```

- Explain your observed results
- Bonus: can you eliminate a copy in h? Hint: use swap()

boostpro computing

Streeeetch!

Finally, Boost.Threads

Boost.Thread in Transition

- Heading toward full C++11 Compatibility
- Select version: `-D BOOST_THREAD_VERSION=X`

BOOST_THREAD_VERSION	2	3
Default in	Boost ≤ 1.52	Boost ≥ 1.53
Name of "future" template	unique_future	future
Move Semantics Emulation	Internal	Boost.Move
Joinable ~thread semantics	detach	terminate()

Thread States

- Running – Currently executing. By definition, #running threads <= #processors
- 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.

boost::thread

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

    void swap(thread& x);
    thread(thread &&);
    thread& operator=(thread &&);

    explicit thread(function<void()>);

    void join();

    bool timed_join(const system_time&);

    template<typename Duration>
    bool timed_join(Duration const&);

    bool joinable() const;
    void detach();

    static unsigned
    hardware_concurrency();

    typedef platform-specific-type
        native_handle_type;

    native_handle_type
    native_handle();

    void interrupt();
    bool interruption_requested() const;

    class id { … };
    id get_id() const;
};
```

Interruptible

boostpro computing

boost/std::thread exception safety

boostpro computing

Fast π: boost::thread

$$\pi = 16 \arctan(1/5) - 4 \arctan(1/239)$$

```
#include <boost/thread.hpp>
#include <boost/spirit/home/phoenix.hpp>
int main()
{
    using namespace boost::phoenix;

    InfPrec atani5;  // compute atan(1/5)
    boost::thread t(
        ref(atani5) = bind(atan1over, 5)
    );

    // compute atan(1/239)
    InfPrec atani239 = atan1over(239);

    t.join(); // wait for atani5
    InfPrec pi = (4*atani5-atani239) * 4;

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

```
// Compute atan( 1/x )
InfPrec atan1over(int x)
{
    InfPrec Result = InfPrec(1) / x;
    int XSquared = x * x;

    int Divisor = 1;
    InfPrec Term = Result;
    InfPrec TempTerm;
    while (Term != 0)
    {
        Divisor += 2;  Term /= XSquared;
        Result -= Term / Divisor;

        Divisor += 2;  Term /= XSquared;
        Result += Term / Divisor;
    }
    return Result;
}
```

http://www.cygnus-software.com/misc/pidigits.htm

boostpro computing

More π: std::future

$$\pi = 16 \arctan(1/5) - 4 \arctan(1/239)$$

```
#include <boost/thread.hpp>
#include <boost/spirit/home/phoenix.hpp>
int main()
{
    using namespace boost::phoenix;

    std::future<InfPrec> atani5
      = std::async( bind(atan1over, 5) );



    // compute atan(1/239)
    InfPrec atani239 = atan1over(239);

    t.join(); // wait for atani5
    InfPrec pi = (4*atani5-atani239) * 4;

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

```
// Compute atan( 1/x )
InfPrec atan1over(int x)
{
    InfPrec Result = InfPrec(1) / x;
    int XSquared = x * x;

    int Divisor = 1;
    InfPrec Term = Result;
    InfPrec TempTerm;
    while (Term != 0)
    {
        Divisor += 2;  Term /= XSquared;
        Result -= Term / Divisor;

        Divisor += 2;  Term /= XSquared;
        Result += Term / Divisor;
    }
    return Result;
}
```

http://www.cygnus-software.com/misc/pidigits.htm

boostpro computing

More π: std::future

$$\pi = 16 \operatorname{atan}(1/5) - 4 \operatorname{atan}(1/239)$$

```
#include <boost/thread.hpp>
#include <boost/spirit/home/phoenix.hpp>
int main()
{
    using namespace boost::phoenix;

    std::future<InfPrec> atani5
      = std::async( bind(atan1over, 5) );



    // compute atan(1/239)
    InfPrec atani239 = atan1over(239);

    InfPrec pi
      = (4 * atani5.get() - atani239) * 4;

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

```
// Compute atan( 1/x )
InfPrec atan1over(int x)
{
    InfPrec Result = InfPrec(1) / x;
    int XSquared = x * x;

    int Divisor = 1;
    InfPrec Term = Result;
    InfPrec TempTerm;
    while (Term != 0)
    {
        Divisor += 2;  Term /= XSquared;
        Result -= Term / Divisor;

        Divisor += 2;  Term /= XSquared;
        Result += Term / Divisor;
    }
    return Result;
}
```

http://www.cygnus-software.com/misc/pidigits.htm

boostpro computing

More π: boost::future

$$\pi = 16 \operatorname{atan}(1/5) - 4 \operatorname{atan}(1/239)$$

```
#include <boost/thread.hpp>
#include <boost/spirit/home/phoenix.hpp>
int main()
{
    using namespace boost::phoenix;

    boost::packaged_task<InfPrec> p(
                    bind(atan1over, 5) );
    boost::unique_future<InfPrec> atani5
      = p.get_future();
    boost::thread t( boost::move(p) );

    InfPrec atani239 = atan1over(239);

    InfPrec pi
      = (4 * atani5.get() - atani239) * 4;

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

```
// Compute atan( 1/x )
InfPrec atan1over(int x)
{
    InfPrec Result = InfPrec(1) / x;
    int XSquared = x * x;

    int Divisor = 1;
    InfPrec Term = Result;
    InfPrec TempTerm;
    while (Term != 0)
    {
        Divisor += 2;  Term /= XSquared;
        Result -= Term / Divisor;

        Divisor += 2;  Term /= XSquared;
        Result += Term / Divisor;
    }
    return Result;
}
```

http://www.cygnus-software.com/misc/pidigits.htm

boostpro computing

More π: boost::future

$$\pi = 16 \operatorname{atan}(1/5) - 4 \operatorname{atan}(1/239)$$

```
#include <boost/thread.hpp>
#include <boost/spirit/home/phoenix.hpp>
int main()
{
    using namespace boost::phoenix;

    boost::future<InfPrec> atani5
          = async( bind(atan1over, 5) );



    // compute atan(1/239)
    InfPrec atani239 = atan1over(239);

    InfPrec pi
      = (4 * atani5.get() - atani239) * 4

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

```
#include <boost/thread.hpp>
#include <boost/utility/result_of.hpp>

template <class F>
boost::detail::thread_move_t<
  boost::unique_future<
    typename boost::result_of<F()>::type> >
async(F f)
{
    typedef typename
      boost::result_of<F()>::type R;

    boost::packaged_task<R> pt( f );
    boost::unique_future<R> future
      = pt.get_future();
    boost::thread( boost::move(pt) )
      .detach();
    return boost::move(future);
}
```

http://www.cygnus-software.com/misc/pidigits.htm

boostpro computing

Future Species

- `std::future<T>`, `boost::unique_future<T>`
  - Read-once, move-only, one object holds a given result
  - Convertible (via move) into `shared_`/`atomic_future`
- `shared_future<T>`
  - Read-many, copiable, copies hold the same result
- `atomic_future<T>`
  - Read-many, copiable, accessible from multiple threads

std::packaged_task

```
template<class> class packaged_task;

template <class R, class...ArgTypes>
struct packaged_task<R(ArgTypes...)>
  : noncopyable
{
    typedef R result_type;

    packaged_task();
    template <class F>
    explicit packaged_task(F f);

    template <class F>
    explicit packaged_task(F&&f);

    ~packaged_task();

    packaged_task(
      packaged_task&& other);
    packaged_task&operator=(
      packaged_task&& other);

    void swap(packaged_task& other);

    explicit operator bool()const;

    future<R> get_future();

    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(
      ArgTypes...);

    void reset();
};
```


boostpro computing

boost::packaged_task

```
template <class R>
struct packaged_task
  : noncopyable
{
    typedef R result_type;

    packaged_task();
    template <class F>
    explicit packaged_task(F f);

    template <class F>
    explicit packaged_task(F&&f);

    ~packaged_task();

    packaged_task(
      packaged_task&& other);
    packaged_task&operator=(
      packaged_task&& other);

    void swap(packaged_task& other);

    explicit operator bool()const;

    unique_future<R> get_future();

    void operator()();
    template <typename F>
    void set_wait_callback(F f);

    void reset();
};
```


boostpro computing

Summary

- Use a `future` to read a value (or throw an exception) generated in another thread
- Launch a thread with `async` to immediately start computing a `future` result
- Create a `packaged_task` to represent a delayed function call with a `future` result.
- Use a `promise` to explicitly write a `future` result.

Promise

```
template <class R>
struct promise : noncopyable
{
    promise();
    template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
    ~promise();

    promise(promise&& rhs);
    promise& operator=(promise&& rhs);
    void swap(promise& other);

    future<R> get_future();

    void set_value(const R& r);
    void set_exception(exception_ptr p);

    void set_value_at_thread_exit(const R& r);
    void set_exception_at_thread_exit(exception_ptr p);
};
```

boostpro computing

A Container of Threads

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

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

    return sum;
}

unsigned const max_thread = 16;
boost::array<
  boost::thread, max_thread> threads;
boost::array<long double, max_thread> results;

std::size_t const nthreads = (std::min)(
  (std::max)(
    1u,
    boost::thread::hardware_concurrency()
  ),
  max_thread
);
```

```
int main()
{
    unsigned long const nterms = 100000000;
    long double const step
      = long double(nterms) /  nthreads;

    for (unsigned i = 0; i < nthreads; ++i)
    {
        using namespace boost::phoenix;
        threads[i] = boost::move( boost::thread(
          var(results[i])
            = bind(sumterms,i*step,(i+1)*step)
        ) );
    }
    for (int i = 0; i < nthreads; ++i)
        threads[i].join();

    long double pi = 4 * std::accumulate(
      results.begin(),
      results.begin() + nthreads, 0.0 );

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


boostpro computing

A Container of Threads

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

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

    return sum;
}



boost::thread_group threads;


std::size_t const nthreads =
  (std::max)(
    1u,
    boost::thread::hardware_concurrency()
  );

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

```
int main()
{
    unsigned long const nterms = 100000000;
    long double const step
      = long double(nterms) /  nthreads;

    for (unsigned i = 0; i < nthreads; ++i)
    {
        using namespace boost::phoenix;
        threads.create_thread(
          var(results[i])
            = bind(sumterms,i*step,(i+1)*step)
        );
    }
    threads.join_all();


    long double pi = 4 * std::accumulate(
      results.begin(),
      results.begin() + nthreads, 0.0 );

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


boostpro computing

boost::thread_group

```
class thread_group : noncopyable
{
public:
  thread_group();

  thread* create_thread(
    function<void()> const&);

  void add_thread(thread*);
  void remove_thread(thread*);

  void join_all();
  void interrupt_all();
  int size() const;
};
```

- create_thread() creates a thread and adds it to the group
- add_thread() takes ownership of a thread
- remove_thread() releases ownership of a thread
- join_all()/ interrupt_all() joins or interrupts all of the threads in the group

Threading and Invariants

- In serial programs, controlling which code sees broken invariants is relatively easy
- In threaded programs, we must stop other threads from looking (or touching). Requires cooperation!
- A concurrent program that doesn't control visibility of broken invariants has a race condition
- From the POV of threading, even an `int` has an invariant that is broken during mutation

Serializing Access

- Basic mechanism: mutex
- Associated with some shared mutable data (of any size)
- 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

Boost Locks

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

mutex, lock_guard

```
std::pair<
    long double, boost::mutex
> pi_over_4;

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

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

    boost::lock_guard<boost::mutex>
        l(pi_over_4.second);
    pi_over_4.first += sum;
}

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

boost::thread_group threads;
std::vector<long double> results(nthreads);
```

```
int main()
{
    unsigned long const nterms = 100000000;
    long double const step
      = long double(nterms) /  nthreads;

    for (unsigned i = 0; i < nthreads; ++i)
    {
        using namespace boost::phoenix;
        threads.create_thread(
          bind(sumterms,i*step,(i+1)*step)
        );
    }

    threads.join_all();

    // no need to lock here
    long double pi = 4 * pi_over_4.first;

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


boostpro computing

Synchronized I/O

```
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <iostream>

boost::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;
    }

    boost::lock_guard<boost::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()
{
    boost::thread_group t;

    for (int n = 1; n < 10; ++n)
      t.create_thread(
        boost::bind(printer, n) );

    t.join_all();
}
```


boostpro computing

Pairwise Association
A Bad Example

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

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

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

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

struct lock2
{

    lock2(boost::mutex& a, boost::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;
}
```


boostpro computing

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)

Pairwise Association
A Bad Example

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

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

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

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

struct lock2
{
    lock2(boost::mutex& a, boost::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;
}
```


boostpro computing

Pairwise Association
A Bad Example

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

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

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

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

struct lock2
{
    typedef std::less<mutex*> cmp;
    lock2(mutex& a, mutex& b)
      : l0( cmp()(&a,&b) ? a : b ),
        l1( cmp()(&a,&b) ? b : a )
    {}
    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;
}
```

mutexes locked in address order

boostpro computing

Pairwise Association

Time Out!

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

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

    typedef boost::timed_mutex mutex;
 private:
    collab* partner;
    mutex gate;
};

struct lock2
{
    typedef std::less<boost::mutex*> cmp;
    lock2(mutex& a, mutex& b)
      : l0( cmp()(&a,&b) ? a : b ),
        l1( cmp()(&a,&b) ? b : a )
    {}

    boost::lock_guard<collab::mutex> l0, l1;
};
```

```
void collab::decouple()
{
    typedef boost::unique_lock<mutex> lock;
    while (1)
    {
        using boost::posix_time::millisec;
        lock g0(
          gate, boost::get_system_time()
                + millisec(1));
        if (!g0.owns_lock()) continue;

        if (!partner) return;

        lock g1(
          partner->gate, boost::defer_lock);

        if ( !g1.timed_lock(
             boost::get_system_time()
             + millisec(1)) )
            continue;

        partner->partner = 0;
        partner = 0;
        return;
    }
}
```


boostpro computing

Mutex/Lock Bazaar

Nested Lock Type / Mutex Type	`scoped_lock`	`scoped_try_lock`	`scoped_timed_lock`
`mutex::`	Yes	Yes	No
`timed_mutex::`	Yes	Yes	Yes
`recursive_mutex::`	Yes	Yes	No
`recursive_timed_mutex::`	Yes	Yes	Yes

Waiting for Particular Shared State(s)

- Basic mechanism: "condition variable"
- Blocks a thread until some predicate might be satisfied

- Note: Always used with a mutex to ensure the predicate sees only non-broken invariants

boostpro computing

boost::condition

```
struct condition : noncopyable
{
  // Awakens one waiting thread
  void notify_one();

  // Awakens all waiting threads
  void notify_all();

  // Temporarily releases lk; blocks until notified; reacquires lk
  template<class ScopedLock>
  void wait(ScopedLock& lk);

  // while (!p()) { this->wait(lk); }
  template<class ScopedLock, class Pred>
  void wait(ScopedLock& lk, Pred p);

  // Temporarily releases lk; blocks until notified or t has elapsed; reacquires lk
  template<class ScopedLock>
  bool timed_wait(ScopedLock& lk, const xtime& t);

  //  while (!p()) { this->wait(lk, t); }
  template<class ScopedLock, class Pred>
  bool timed_wait(ScopedLock& lk, const xtime& t, Pred p);
};
```

Condition Example: Message Queue

```
bounded_msg_queue q;

void sender()
{
    for (int n = 0; n < 100; ++n)
        q.send(n);
    q.send(-1); // end sentinel
}

void receiver()
{
    for (int n = 0; n != -1;)
    {
        n = q.receive();
        std::cout << n << std::endl;
    }
}
```

```
int main()
{
  boost::thread t1(sender);
  boost::thread t2(receiver);
  t1.join();
  t2.join();
}
```

- If q is full when sending, must block until no longer full
- If q is empty when receiving, must block until no longer empty

Condition Example: Message Queue

```
template <unsigned size, class T>
struct bounded_msg_queue
{
    bounded_msg_queue()
      : begin(), end(), buffered() {}

    void send(T m)
    {
        boost::unique_lock<boost::mutex>
          lk(broker);
        while (buffered == size)
            not_full.wait(lk);
        buf[end] = m;
        end = (end + 1) % size;
        ++buffered;
        not_empty.notify_one();
    }

    ...

    ...

    T receive();
 private:
    int begin, end, buffered;
    boost::condition not_full, not_empty;
    boost::mutex broker;
    T buf[size];
};
```

- Lock the mutex before checking the predicate
- Keep checking until true, in case of spurious wakeups, shared conditions
- `notify_one` wakes a waiting thread
- `receive()` left as an exercise for the reader

Threadsafe Initialization

- "Once routines"
  - Executed once, no matter how many invocations
  - No invocation will complete until the one execution finishes
- Typical use: protect function-static data
  - Some compilers do this automatically
  - Not necessarily a good idea (expensive)
  - The standard doesn't mandate it

boost::call_once

```
typedef unspecified once_flag;

#define BOOST_ONCE_INIT unspecified

void call_once( void(*)(), once_flag& );
```

- Declare a namespace scope once_flag for each once routine
- Initialize it to BOOST_ONCE_INIT
- Invoke the once routine indirectly by passing its address and once_flag to call_once.

Meyers Singleton

```
class my_singleton
{
 public:
    static my_singleton& instance()
    {
        static my_singleton self;
        return self;
    }

 private:
    my_singleton() {...}
};
```

- Avoids order-of-initialization issues common to namespace-scope variables.
- Warning: does not avoid destruction order issues of namespace-scope variables.

Threadsafe Meyers Singleton

```
class my_singleton
{
 public:
    static my_singleton& instance()
    {
        boost::call_once(
          &my_singleton::init, once);
        return get_instance();
    }

    …
```

```
   …
private:
    my_singleton() {…}

    static void init() { get_instance(); }

    static my_singleton& get_instance()
    {
        static my_singleton self;
        return self;
    }

  static boost::once_flag once;
};

boost::once_flag my_singleton::once =
    BOOST_ONCE_INIT;
```

Exercise 2

- Complete the message queue
- Launch two producer threads that insert messages
- Launch one consumer thread that reads them out and prints them to cout
- Messages should not be garbled!

boostpro computing