Boost Thread Library

Kevin Heifner

www.ociweb.com

Overview

- **`#include <boost/thread.hpp>`**
  - includes entire library
  - requires linking to thread library
  -  **`#include <boost/thread/thread.hpp>`**
  -  **`#include <boost/thread/exceptions.hpp>`**
  -  **`#include <boost/thread/mutex.hpp>`**
  -  **`#include <boost/thread/condition.hpp>`**
- **`Not covered`**
  - **`#include <boost/thread/once.hpp>`**
  - **`#include <boost/thread/tss.hpp>`**
  - **`#include <boost/thread/barrier.hpp>`**

Multiprocessing vs. Multithreading

- Multiprocessing
  - simultaneous execution of applications (ex. Firefox & Thunderbird)
  - simulated on single-processor machines
  - heavyweight processes - high OS overhead
  - can make applications less efficient
    - they're not running when another application takes over
  - no code changes are needed - handled by OS

Multiprocessing vs. Multithreading

- Multithreading
  - simultaneous execution of pieces of one application
  - simulated on single-processor machines
  - lightweight processes - low OS overhead
  - can make applications more efficient
    - execution of one part (e.g. calculations, screen repaint, audio playing) can proceed while another part waits on something (e.g. file I/O or user input)
  - can make applications less efficient
    - e.g. overhead associated with acquiring exclusive access to shared resources
  - must write code to use it 
  - ***greatly*** complicates code 

Threading Definitions 

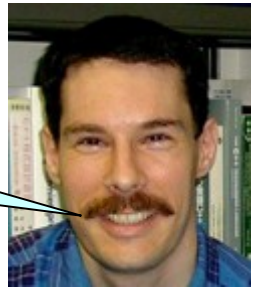
- A ***thread***, ***thread of execution***, is a single series of steps performed by a program
- A program that has several threads of control executing concurrently is ***multithreaded***
- ***Concurrency*** is multiple operations performed simultaneously.
- A function is ***thread-safe*** if it can be called by many threads simultaneously without any other action from the caller
- Each thread executes a function which serves as the ***entry-point*** into the thread
  - `main()` is the entry-point for the main application thread
- The ***scheduler*** is responsible for putting/removing threads on/off CPUs for execution

Reasons To Use Threads

- To achieve asynchronous operation
  - when a client requests a service it may want to continue executing before the service completes
  - models real world activities where many things are acting simultaneously
- To provide high availability of services
  - one thread accepts and manages service requests
  - it starts new threads to provide the service to clients
- To control execution of a piece of code
  - threads can wait while other threads execute, and be notified to continue
- To execute code based on availability of input data
  - thread getting data invokes some method on completion
- To improve performance (not guaranteed)
  - can get parallel thread execution with multiple processors
- To make a graphical user interface more responsive

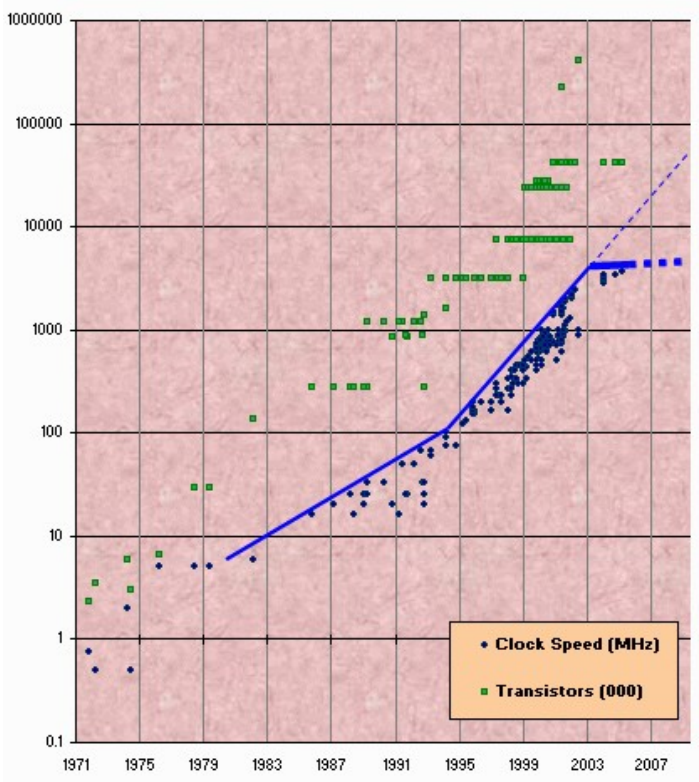
The Free Lunch Is Over

The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.



Herb Sutter

http://www.gotw.ca/publications/concurrency-ddj.htm

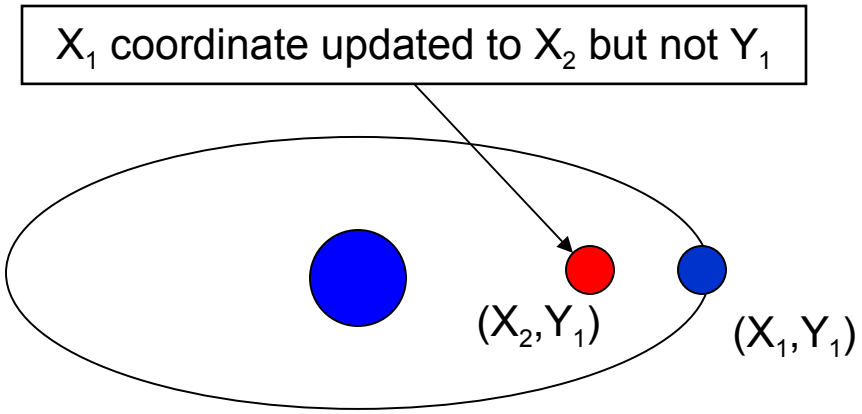


Intel CPU Introductions (sources: Intel, Wikipedia)

- Intel chips
  - 2 Ghz in 2001
  - 10 Ghz in 2005 ?!
- Number of transistors continues to grow
  - dual-core and quad-core processors today
  - many-core processors tomorrow
- Single threaded applications no longer run significantly faster on new hardware

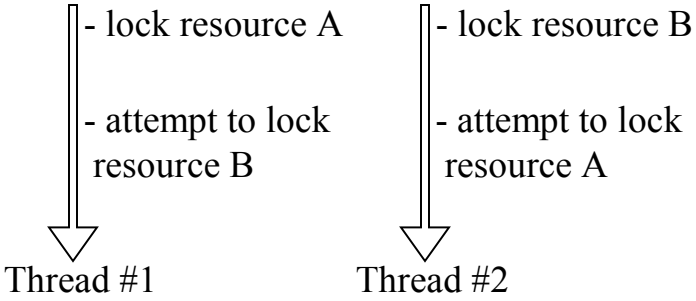
Issues with Using Threads

- Race Condition
  - two threads try to access the same data at the same time
  - one may only update the data partially before the other reads it
    - resulting in read of illegal (unknown or corrupted) state
- Synchronization
  - must be properly addressed when threads share data
  - careful of globals including singletons and statics
  - e.g. a planet would be displayed in an invalid position if the 1st thread used a position that was only partially updated by the 2nd thread



Issues with Using Threads

- Failure to execute
  - threads may fail to execute due to
    - starvation
      - not getting CPU time because other threads get all of it
    - deadlock ("deadly embrace")
      - bad synchronization
      - e.g. thread attempts to acquire a lock held by another thread that is never released



Issues with Using Threads 

- Unpredictability
  - execution order often unpredictable and not repeatable
    - between runs on the same platform
    - on different platforms
      - multi-cpu vs. dual-core vs. hyper-threading, etc
      - Microsoft Windows vs. Linux vs. Solaris, etc.
    - debug vs. non-debug vs. running in debugger
  - synchronization techniques can help
  - test, test, test and repeat
- Overhead
  - creating and starting threads has overhead
    - slower due to scheduler interaction and synchronization
    - synchronization memory barriers hurt cpu pipelining
    - more memory needed for thread objects and lock bookkeeping

Threads And System Resources

- Most operating systems provide low-level mechanisms for maintaining separation of threads in a process
- Many process resources are shared by all threads
  - address space
  - file descriptor table
  - timers
- Some resources are specific to each thread

  (no synchronization required)
  - thread identifier
  - set of registers
  - stack
  - thread-local variables (e.g. **`errno`**)
  - thread-specific storage
  - exceptions

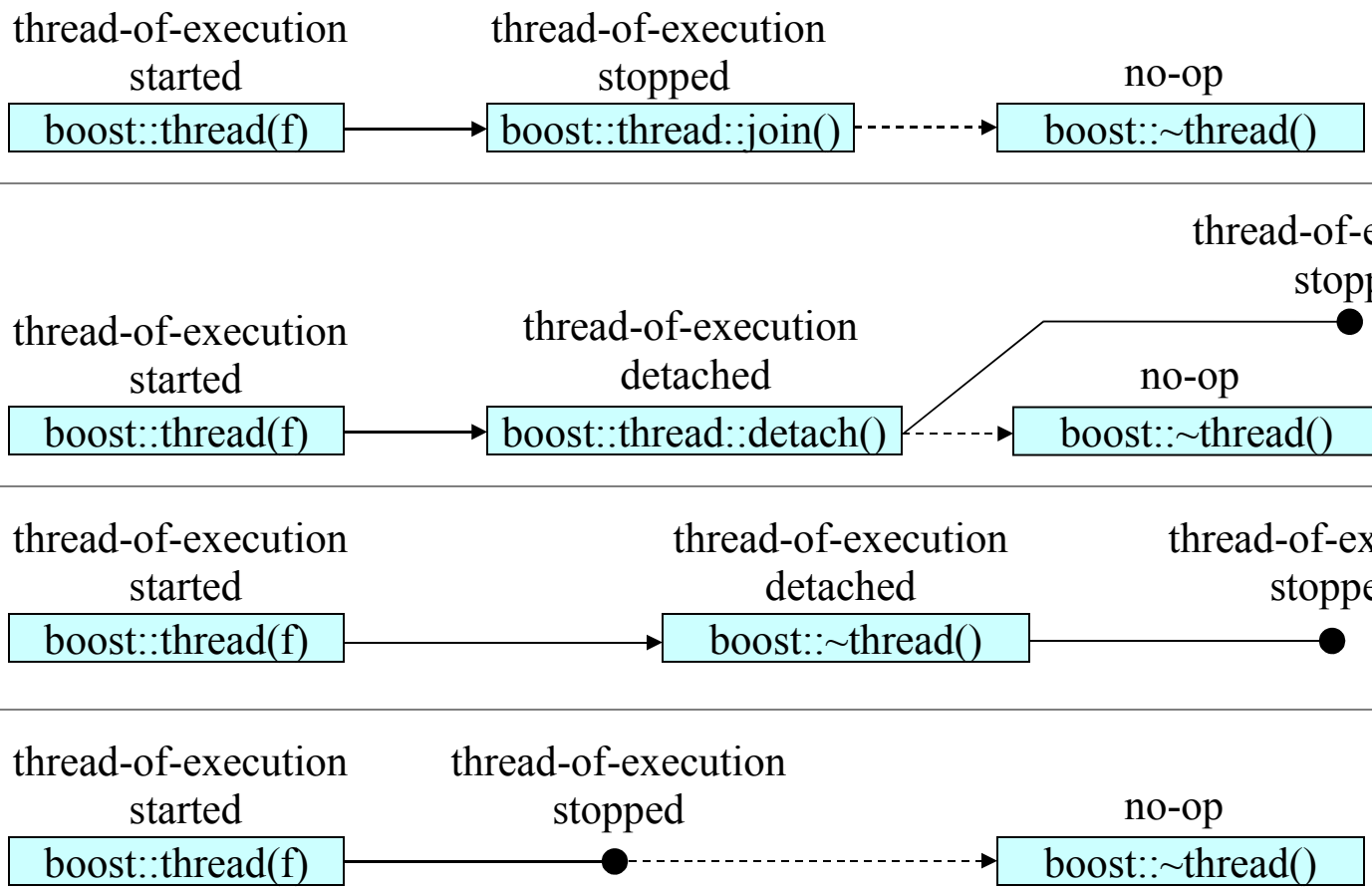
boost::thread

- **`#include <boost/thread/thread.hpp>`**
- Requires linking to library
  - most Windows compilers have auto-linking support
- Lifetime of **`boost::thread`** object is different than lifetime of thread of execution
  - thread-of-execution is started in constructor of **`boost::thread`**
  - after call to **`boost::thread::join()`** thread of execution is complete
  - destruction of **`boost::thread`** object without a call to **`boost::thread::join()`** allows thread of execution to continue
    - destructor of **`boost::thread`** detaches thread of execution
  - **`main()`** does not wait for child threads of execution to exit, all spawned threads are terminated

boost::thread

- Default constructor creates object representing *Not-a-Thread* (<1.35 represented current thread)
- Non-default constructor spawns a new thread of execution and calls a copy of the given function
  - return from constructor does not mean that the first line of the given function has been reached
  - thread of execution could be complete before returning from the constructor
- Conforms to the C++0x Working Draft 2008-03-17 (N2588) Thread library specification
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf
  - thread, mutex, condition_variable

Thread Lifetime



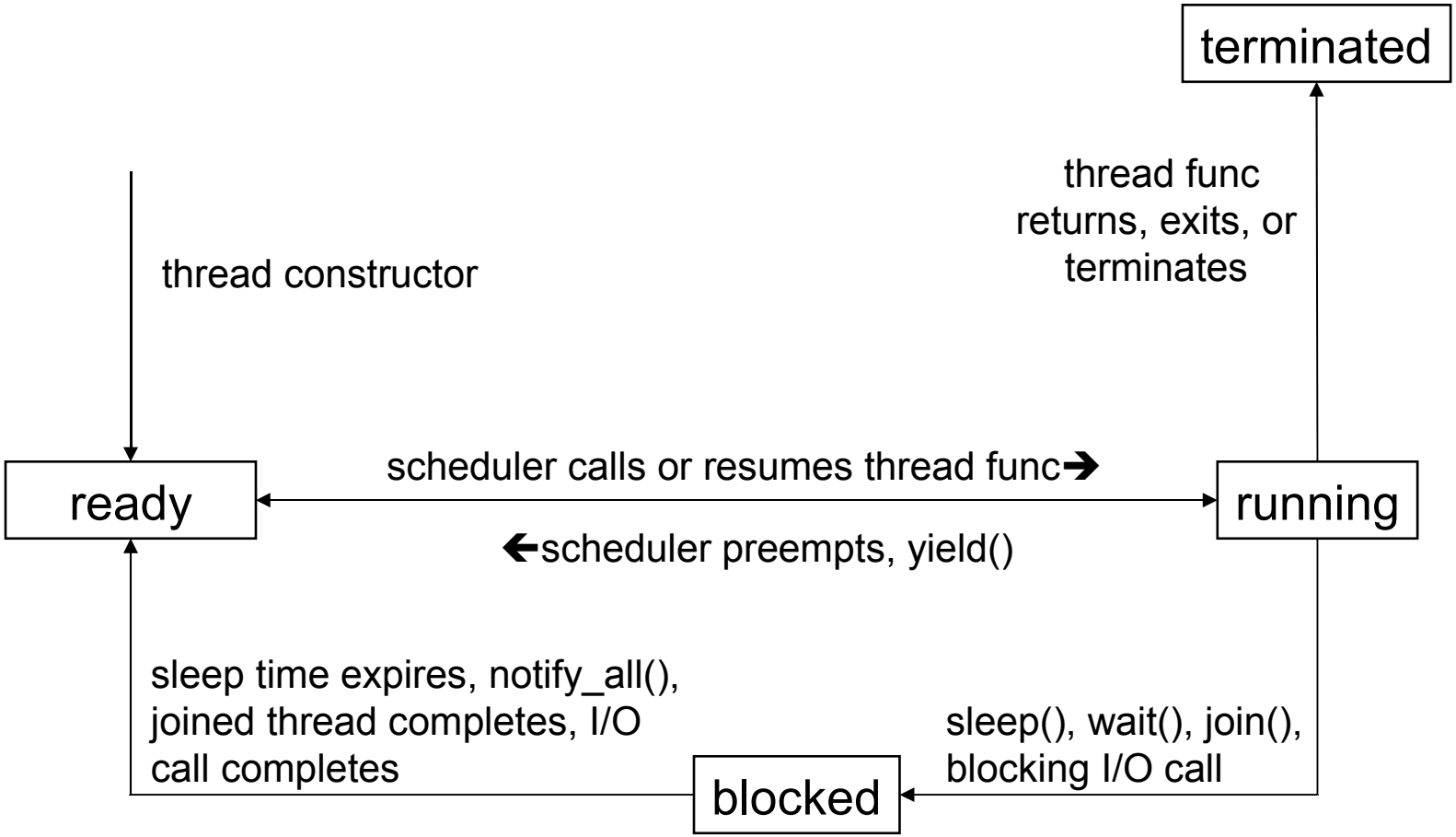
Thread Example

```
// very inefficient prime number finder
class PrimeFinder {
  public:
    PrimeFinder(long begin, long end) : begin_(begin), end_(end)
   {}
    void calcPrimes() {
      for (long i = begin; i <= end; ++i) {
        if (isPrime(i)) primes_.push_back(i);
      }
    }
    const std::vector<long>& getPrimes() const { return primes_; }
  private:
    bool isPrime(long num) const {
      long i = 2;
      for (; !(num % i == 0); ++i);
      return num == i;
    }
  private:
    long begin_, end_;
    std::vector<long> primes_;
};
```

Thread Example

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

PrimeFinder pf(50001, 100000);

// spawn thread
boost::thread t(boost::bind(&PrimeFinder::calcPrimes, &pf));

// perform other work while thread runs
PrimeFinder pf2(2, 50000);
pf2.calcPrimes();
std::copy(pf2.getPrimes().begin(), pf2.getPrimes().end(),
  std::ostream_iterator<long>(std::cout, " "));

t.join(); // wait for thread to finish

std::copy(pf.getPrimes().begin(), pf.getPrimes().end(),
  std::ostream_iterator<long>(std::cout, " "));
```

Thread States



sleep()

- **`boost::this_thread::sleep()`** takes a duration
- **`boost::thread::sleep()`** takes an absolute time
- Times and durations from **`boost::date_time`** lib

```
boost::this_thread::sleep(boost::posix_time::hours(1));
boost::this_thread::sleep(boost::posix_time::minutes(1));
boost::this_thread::sleep(boost::posix_time::seconds(1));
// 1000th of a second
boost::this_thread::sleep(boost::posix_time::milliseconds(1));
// 1,000,000th of a second
boost::this_thread::sleep(boost::posix_time::microseconds(1));
```

```
boost::system_time now = boost::get_system_time();
boost::posix_time::ptime tomorrow(now +
   boost::gregorian::days(1));
boost::thread::sleep(tomorrow);
```

Thread Interruption (new 1.35)

- Not part of C++0x working draft 2008-03-17
- **`boost::thread::interrupt()`**
  - politely ask a **`boost::thread`** to stop
  - sets a flag, checked at interruption points
- **`boost::thread_interrupted`** thrown at
  - **`boost::thread::join()`** and **`time_join()`**
  - **`boost::thread::sleep()`**
  - **`boost::this_thread::sleep()`**
  - **`boost::this_thread::interruption_point()`**
  - **`boost::condition_variable::wait()`** and **`timed_wait()`**
  - **`boost::condition_variable_any::wait()`** and **`timed_wait()`**

Synopsis boost::thread

```
namespace boost {

class thread { // non-copyable, moveable
  public:
    thread(); // not-a-thread

    // stores copy of thr_func, spawns new thread of execution
    template <typename F> explicit thread(F thr_func);
    ~thread(); // if joinable() then detach(), not join()

    // move emulation and swap() not shown

    id get_id() const;
    bool joinable() const;
    void join(); // wait for thread of execution to complete
    void timed_join(const system_time& t); // also duration version
    void detach();

    native_handle_type native_handle();

    void interrupt(); // set interruption flag, not part of C++0x
    bool interruption_requested() const;     // not part of C++0x
```

Synopsis boost::thread

```
  // number of hardware threads
  static unsigned hardware_concurrency();

  // see boost::this_thread::sleep() for time duration sleep
  static void sleep(const system_time& t);
  static void yield();

  class id {
    public:
      id(); // not-a-thread
      // operators ==, !=, <, >, <=, >=
      // operator <<
  };

}; // class boost::thread

class thread_exception : public std::exception {};
class thread_interrupted {};
```

Synopsis boost::this_thread

```
namespace this_thread {
  thread::id get_id();
  void yield() { thread::yield(); }
  template <typename TimeDuratoin>
    void sleep(const TimeDuration& t) {
      thread::sleep(boost::get_system_time() + t);
  } // expects boost::date_time::time_duration type

  void interruption_point(); // check flag, throw thread_interrupted
  bool interruption_requested(); // return flag value
  bool interruption_enabled();
  class disable_interruption { // disable interruption for scope
    public:
      disable_interruption();
      ~disable_interruption();
  };
  class restore_interruption { // restore to previous interruption
    public:                    // state before given 'di'
      explicit restore_interruption(disable_interruption& di);
      ~restore_interruption();
  };
 } // namespace this_thread
} // namespace boost
```

Thread Group Example

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

PrimeFinder pf(2, 50000);
PrimeFinder pf2(50001, 100000);

boost::thread_group g;
// spawn threads
g.create_thread(boost::bind(&PrimeFinder::calcPrimes, &pf));
g.create_thread(boost::bind(&PrimeFinder::calcPrimes, &pf2));

g.join_all(); // wait for all threads to finish

std::copy(pf.getPrimes().begin(), pf.getPrimes().end(),
  std::ostream_iterator<long>(std::cout, " "));
std::copy(pf2.getPrimes().begin(), pf2.getPrimes().end(),
  std::ostream_iterator<long>(std::cout, " "));
```

Synopsis boost::thread_group

```
namespace boost {

class thread_group { // non-copyable
  public:
    thread_group();  // empty group
    ~thread_group(); // does not call join_all()

    // stores copy of thr_func, spawns new thread of execution
    // maintains ownership of returned thread
    thread* create_thread(const boost::function<void ()>& thr_func);

    // takes ownership of given thread (calls delete on destruction)
    void add_thread(thread* t);
    void remove_thread(thread* t); // releases ownership

    void join_all(); // wait for all threads of execution to complete
    void interrupt_all();
    size_t size() const;
};
} // namespace boost
```

boost::mutex and boost::mutex::scoped_lock

- **`#include <boost/thread/mutex.hpp>`**
- Provides ***mut***ually ***ex***clusive access to resource(s)
  - can be locked/unlocked by only one thread of execution at a time
    - attempt to lock when already locked by another thread causes thread to wait (blocked state) until unlocked by other thread
  - has no knowledge of resource it is guarding
  - (<1.35) platform dependent if mutex was recursive
    - recursive mutex allows a thread to acquire a mutex multiple times before releasing it (now provided via **`boost::recursive_mutex`**)
- **`boost::mutex::scoped_lock`** provides RAII exception- safe locking/unlocking of **`boost::mutex`**
- Always acquire **`mutex`**es in the same order when acquiring more than one to avoid deadlock
- Recursive and timed mutexes are also provided

Mutex Example

```
template<class T>
class ThreadSafeStack {
  public:
    typedef typename std::stack<T>::container_type
   container_type;
    typedef typename std::stack<T>::size_type       size_type;
    typedef typename std::stack<T>::value_type      value_type;

    ThreadSafeStack() {}
    ~ThreadSafeStack() {}

  private:
    mutable boost::mutex mutex_;
    std::stack<T> stack_;

  public:
    // continued next slide
```

Mutex Example

```
  void push(const T& t) {
    boost::mutex::scoped_lock lock(mutex_);
    stack_.push(t);
  }

  // returns true if not empty (t contains top of stack)
  bool pop(T& t) {
    boost::mutex::scoped_lock lock(mutex_);
    if (stack_.empty()) return false;
    t = stack_.top();
    stack_.pop();
    return true;
  }

  size_type size() const {
    boost::mutex::scoped_lock lock(mutex_);
    return stack_.size();
  }

}; // class ThreadSafeStack
```

Synopsis boost::mutex

```
namespace boost {
class lock_error : public thread_exception {};

class mutex : boost::noncopyable {
  public:
    mutex();
    ~mutex();

    void lock();  // block until able to lock
    bool try_lock(); // return true if locked
    void unlock();

    native_handle_type native_handle();

    typedef unique_lock<mutex> scoped_lock;
}; // class mutex
} // namespace boost
```

Synopsis boost::mutex

- **`boost::recursive_mutex`** has the same definition
- **`boost::timed_mutex`** and **`recursive_timed_mutex`** add:

```
bool timed_lock(const system_time& t);
template <typename TimeDuration>
  bool timed_lock(const TimeDuration& t);
```

- **`boost::shared_mutex`** (multi-reader/single-writer) not shown
  - **`shared_lock, upgrade_lock, upgrade_to_unique_lock not shown`**
  - **`not part of C++0x`**

Synopsis boost::unique_lock

```
namespace boost {
template <typename Mutex> class unique_lock { // non-copyable, moveable
  public:
    explicit unique_lock(Mutex& m);        // m.lock(), takes ownership
    unique_lock(Mutex& m, adopt_lock_t);  // takes ownership
    unique_lock(Mutex& m, defer_lock_t);  // does not take ownership
    unique_lock(Mutex& m, try_to_lock_t); // owns if m.try_lock()
    unique_lock(Mutex& m,
                const system_time& t);    // owns if m.timed_lock(t)
    ~unique_lock(); // unlocks mutex if owned
    // move emulation and swap() not shown

    void lock();     // m.lock(), throws if already locked
    bool try_lock(); // m.try_lock(), throws if already locked
    bool timed_lock(const system_time& t); // m.timed_lock(t)
    template <typename TimeDur> bool timed_lock(const TimeDur& t);
    void unlock(); // m.unlock(), throws if not locked

    bool owns_lock() const; // operator!, operator bool-type()
    Mutex* mutex() const; // accessor
    Mutex* release(); // releases ownership, user responsible to unlock
}; // class unique_lock
} // namespace boost
```

boost::condition_variable

- **`#include <boost/thread/condition.hpp>`**
- Provides a coordination mechanism between threads
  - a thread can notify one or more other threads through a condition variable
- Used with a **`boost::mutex`**
  - **`condition_variable_any`** templated methods works with any **`Lockable`** concept type (not shown)
  - mutex must be locked prior to waiting and signaling
- Thread(s) must be waiting when notified otherwise the notification is lost, called *lost wakeup*
  - no queuing of notifications
  - mutex must be locked before signaling
- Conditions are subject to *spurious wakeup*
  - it is possible to come out of **`condition.wait()`** without a notify
  - use predicate versions of **`wait()`** to avoid spurious wakeup or follow the pattern of **`while(!pred) condition.wait()`**

Condition Example

```
template<class T>
class ThreadSafeStack {
  public:
    typedef typename std::stack<T>::container_type
   container_type;
    typedef typename std::stack<T>::size_type       size_type;
    typedef typename std::stack<T>::value_type      value_type;

    ThreadSafeStack() {}
    ~ThreadSafeStack() {}

  private:
    mutable boost::mutex mutex_;
    boost::condition_variable cond_;
    std::stack<T> stack_;

  public:
    // continued next slide
```

Condition Example

```
  void push(const T& t) {
    {
      boost::mutex::scoped_lock lock(mutex_);
      stack_.push(t);
    }
    cond_.notify_one(); // notify other thread of addition to stack
  }

  void pop(T& t) {
    boost::mutex::scoped_lock lock(mutex_);
    // wait for condition
    while (stack_.empty()) cond_.wait(lock);
    t = stack_.top();
    stack_.pop();
  }

  size_type size() const {
    boost::mutex::scoped_lock lock(mutex_);
    return stack_.size();
  }
}; // class ThreadSafeStack
```

Synopsis boost::condition_variable

```
namespace boost {

class condition_variable { // non-copyable
  public:
    condition_variable();
    ~condition_variable();

    void notify_one(); // wakes one and only one waiting thread
    void notify_all(); // wakes all waiting threads
                       // (typically they will then compete for mutex)

    native_handle_type native_handle();
```

Synopsis boost::condition_variable

```
    void wait(boost::mutex::scoped_lock& m);
    template <typename Pred> // while (!p) wait(m);
      void wait(boost::unique_lock<mutex>& m, Pred p);

    void timed_wait(boost::unique_lock<mutex>& m,

                    const boost::system_time& t);
    template <typename Pred>
      void timed_wait(boost::unique_lock<mutex>& m,

                      const boost::system_time& t, Pred p);

    template <typename TimeDuration>
      void timed_wait(boost::unique_lock<mutex>& m,
                      const TimeDuration& t);
    template <typename TimeDuration, typename Pred>
      void timed_wait(boost::unique_lock<mutex>& m,
                      const TimeDuration& t, Pred p);
};  // class condition_variable
} // namespace boost
```

- **`condition_variable_any`** same as **`condition_variable`** except wait methods also templated on **`Lock`** instead of **`boost::unique_lock<mutex>`**

ThreadPool Example

- Now lets look at a non-trivial example
  - building a thread pool
- Includes
  - **`boost::thread_group`**
  - **`boost::mutex`**
  - **`boost::mutex::scoped_lock`**
  - **`boost::condition_variable`**
  - **`boost::shared_ptr`**
  - **`boost::lambda`**
  - **`boost::lambda::bind`**
  - **`boost::function`**

ThreadPool Example - ThreadPool.h

```
#ifndef BOOSTX_THREADPOOL_H
#define BOOSTX_THREADPOOL_H

// ThreadPool.h

#include <boost/function.hpp>
#include <boost/thread.hpp>
#include <boost/shared_ptr.hpp>
#include <vector>
#include <queue>
#include <utility>

namespace boostx {

  typedef boost::function<void ()> ThreadFunc;

  namespace detail {
    class Worker;
  }
```

ThreadPool Example - ThreadPool.h

```
class ThreadPool : boost::noncopyable {
 public:
   // size must be >= 1
   explicit ThreadPool(int size = 5);

   // executes all queued tasks before returning
   ~ThreadPool();

   // schedule thr_func to be executed
   void queueTask(const ThreadFunc& thr_func);

 private:
   friend class boostx::detail::Worker;
   void notifyComplete(int id);
```

ThreadPool Example - ThreadPool.h

```
  private:
    boost::thread_group thread_group_;
    boost::mutex mutex_;
    boost::condition_variable cond_;
    typedef std::queue<ThreadFunc> FuncQueue;
    FuncQueue queuedFuncs_;
    typedef boost::shared_ptr<detail::Worker> WorkerPtr;
    // bool is running flag
    typedef std::pair<bool, WorkerPtr> Task;
    typedef std::vector<Task> Pool;
    Pool pool_;
  }; // class ThreadPool

} // namespace boostx

#endif // BOOSTX_THREADPOOL_H
```

ThreadPool Example - ThreadPool.cpp

```
// ThreadPool.cpp

#include "ThreadPool.h"

#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>

namespace boostx {
  namespace detail {

    class Worker {
     public:
      explicit Worker(const boost::function<void ()>& callback)
        : callback_(callback)
        , mutex_()
        , cond_()
        , func_()
        , terminate_(false)
      {}
```

Example - ThreadPool.cpp - Worker

```
void run() {
  while (true) {
    {
      boost::mutex::scoped_lock lock(mutex_);
      namespace bll = boost::lambda;
      cond_.wait(lock, bll::var(terminate_) || bll::var(func_));
      if (terminate_ && func_ == 0) return;
    }
    try {
      func_();
    } catch (std::exception& e) { /* log error */ }
    } catch (...) { /* log error */ }
    {
      boost::mutex::scoped_lock lock(mutex_);
      func_ = 0;
    }
    callback_();
  }
}
```

Example - ThreadPool.cpp - Worker

```
// called from a different thread than run()
void stop() {
  boost::mutex::scoped_lock lock(mutex_);
  terminate_ = true;
  lock.unlock();
  // only one thread per worker
  cond_.notify_one();
}

// called from a different thread than run()
void execute(const ThreadFunc& func) {
  {
    boost::mutex::scoped_lock lock(mutex_);
    func_ = func;
  }
  cond_.notify_one();
}
```

Example - ThreadPool.cpp - Worker

```
  private:
    ThreadFunc callback_;
    mutable boost::mutex mutex_;
    // mutex_ guards the following attributes
    boost::condition_variable cond_;
    ThreadFunc func_;
    bool terminate_;

  }; // class Worker

} // namespace detail
```

Example - ThreadPool.cpp - ThreadPool

```
ThreadPool::ThreadPool(int size) {
  namespace bll = boost::lambda;
  if (size < 1) throw std::runtime_error("ThreadPool size < 1");

  boost::mutex::scoped_lock lock(mutex_);
  // spawn the worker threads
  for (int i = 0; i < size; ++i) {
    WorkerPtr worker(
      new detail::Worker(
        bll::bind(&ThreadPool::notifyComplete, this, i)));

    thread_group_.create_thread(
      bll::bind(&detail::Worker::run, worker.get()));

    pool_.push_back(Task(false, worker));
  }
}
```

Example - ThreadPool.cpp - ThreadPool

```
ThreadPool::~ThreadPool() {
  namespace bll = boost::lambda;
  {
    boost::mutex::scoped_lock lock(mutex_);
    while (!queuedFuncs_.empty()) cond_.wait(lock);
  }
  std::for_each(pool_.begin(), pool_.end(),
    bll::bind(&detail::Worker::stop,
      *bll::bind(&Task::second, bll::_1)));
  thread_group_.join_all();
}
```

Example - ThreadPool.cpp - ThreadPool

```
void ThreadPool::queueTask(const ThreadFunc& thr_func) {
  if (!thr_func) throw std::runtime_error("invalid function");
  namespace bll = boost::lambda;
  boost::mutex::scoped_lock lock(mutex_);

  // find an idle thread
  Pool::iterator i =
    std::find_if(pool_.begin(), pool_.end(),
      !bll::bind(&Task::first, bll::_1));

  if (i != pool_.end()) {
    Task& task = *i;
    task.first = true;
    task.second->execute(thr_func);
  } else {
    queuedFuncs_.push(thr_func);
  }
}
```

Example - ThreadPool.cpp - ThreadPool

```
  // called by Worker when finished with task
  void ThreadPool::notifyComplete(int id) {
    bool empty = false;
    {
      boost::mutex::scoped_lock lock(mutex_);
      Task& task = pool_[id];
      task.first = false;
      if (queuedFuncs_.empty()) {
        empty = true;
      } else {
        task.first = true;
        task.second->execute(queuedFuncs_.front());
        queuedFuncs_.pop();
      }
    }
    // in case we are waiting in the destructor
    if (empty) cond_.notify_one();
  }

} // namespace boostx
```

ThreadPool Example - main.cpp

```
#include "ThreadPool.h"
#include "PrimeFinder.h"

int main() {
  namespace bl = boost::lambda;

  const int step = 50000;
  typedef boost::shared_ptr<PrimeFinder> PrimeFinderPtr;
  std::vector<PrimeFinderPtr> v;

  {
    boostx::ThreadPool pool(5);
    for (int i = 0; i < 1000000; i += step) {
      PrimeFinderPtr ptr(new PrimeFinder(i, i+step));
      v.push_back(ptr);
      pool.queueTask(bl::bind(&PrimeFinder::calcPrimes,
   bl::var(*ptr)));
    }
  } // ThreadPool destructor executes
```

ThreadPool Example - main.cpp

```
  long sum = 0;
  std::for_each(v.begin(), v.end(), bl::var(sum) +=
    bl::bind(&std::vector<long>::size,
      bl::bind(&PrimeFinder::getPrimes, *bl::_1)));

  std::cout << "Number of Primes: " << sum << std::endl;

} // main()
```

Exercises

- Given
  - `std::cout, std::cerr, std::ofstream` are not thread safe
  - I/O provides opportunity for cpu to be doing other tasks
  - logging can be expensive
- Create
  - a delayed logger that uses a single thread for logging

```
class ActiveLogger {
  public:
    // spawns worker thread to process log requests
    explicit ActiveLogger(std::ostream& os);
    // flushes all log messages to ostream
    ~ActiveLogger();
    // add msg to queue to be logged
    // logging happens on background thread
    void log(const std::string& msg);
  private:
    std::ostream& os_;
    // ...
};
```

More Information

- Boost 1.35 thread docs
  - http://www.boost.org/doc/libs/1_35_0/doc/html/thread.html
- C++0x Working Draft 2008-03-17 (N2588)
  - includes thread support library
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf
- PThreads Primer A Guide to Multithreaded Programming
  - http://www.cs.umu.se/kurser/TDBC64/VT03/pthreads/pthread-primer.pdf
  - Bil Lewis, Daniel J. Berg
- Programming with Threads
  - Steve Kleiman, Devang Shah, Bart Smaalders
- Pattern-Oriented Software Architecture Volume 2 (POSA2)
  - Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann
  - Careful, Double-Checked Locking Optimization does not work on some platforms. http://en.wikipedia.org/wiki/Double-checked_locking

- Icons used in presentation courtesy of
  - http://www.famfamfam.com/lab/icons/silk/