HPCSE II

More m

Waiting for Particular Shared State(s)

- How do we wait for a certain state to be reached?
  - Check all the tine in a loop?

    Wastes resources.
  - Sleep for a while before checking again?

    Might waste time by waiting too long
- Solution:

  let the threading library help you by using a condition variable

Condition variable

- Blocks a thread until some condition might be satisfied
- Always used with a mutex to ensure the condition sees only non-broken invariants
- Always enter it with a locked lock
- Always call in a loop that checks the condition at the end, to see whether the notification condition is still valid (not needed for the versions that take a predicate)
- Two types in C++11:
  - **condition_variable**: optimized version, needs to be used with unique_lock<mutex>
  - **condition_variable_any**: can be used with any lock

std::condition_variable_any

```
class condition_variable_any // noncopyable but movable
{
  void notify_one();
  void notify_all();

  template<class Lock>
  void wait(Lock& lk);

  template<class Lock, class Pred>
  void wait(Lock& lk, Pred p);

  template <class Lock, class Clock, class Duration>
  cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);

  template <class Lock, class Clock, class Dur, class Pred>
  cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Dur >& abs_time, Pred pred);

  template <class Lock, class Rep, class Period>
  cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);

  template <class Lock, class Rep, class Period, class Pred>
  cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Pred pred);
};

enum class cv_status { no_timeout, timeout };
```

The non-predicate wait functions must be called in a loop that checks the condition

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()
{
  std::thread t1(sender);
  std::thread t2(receiver);
  t1.join();
  t2.join();
}
```

- If queue is full when sending, must block until no longer full
- If queue 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)
    {
        std::unique_lock<std::mutex> lk(broker);
        not_full.wait(lk,[&] () { return buffered < size;});
        buf[end] = m;
        end = (end + 1) % size;
        ++buffered;
        not_empty.notify_one();
    }

    …
```

```
    …

    T receive();
  private:
    int begin, end, buffered;
    std::condition_variable not_full, not_empty;
    std::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
- look at example codes for receive

A barrier

- Synchronization between threads
  - avoid it whenever possible since it serializes and slows down the code (Amdahl's law)!
  - is sometimes unavoidable: wait for all threads to finish between update steps in a Monte Carlo simulation or integration of a PDE
- No C++ intrinsic, but we can write a barrier class

```
class barrier
{
private:
    mutable std::mutex m_mutex;
    std::condition_variable m_cond;
    unsigned int const m_total;
    unsigned int m_count;
    unsigned int m_generation;

public:
    barrier(unsigned int count)
    : m_total(count)
    , m_count(count)
    , m_generation(0)
    {
      assert(count != 0);
    }
```

```
    void wait()
    {
      std::unique_lock<std::mutex> lock(m_mutex);
      unsigned int gen = m_generation;

      // decrease the count
      if (--m_count==0) {
        // if done reset to new generation of wait
        // and wake up all threads
        m_count = m_total;
        m_generation++;
        m_cond.notify_all();
      }
      else
        while (gen == m_generation)
            m_cond.wait(lock);
    }
};
```

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;
```

Thread-local data

- C++11 has a new keyword **thread_local**, a static variable for each thread

```
int times_called()
{
 thread_local int count=0;
 return ++count;
}
```

- **Unfortunately not yet implemented by any compiler!!!**

- C++03 needs helps from Boost

```
boost::thread_specific_ptr<int>  count;
int foo() // the function running in a thread
{
 count.reset(new int(0)); // will be cleaned up at thread exit
   ...
}

int times_called()
{
  return ++*count;
```