Concurrency in the Real World

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

14th April 2010
Concurrent in the Real World

- Problems with multithreaded code
- Solutions to some of the problems
- Adding concurrency to an application
Problems with multithreaded code
The biggest problem with multithreaded code is incorrect synchronization.
Too much synchronization
Too little synchronization

Anthony Williams
Just Software Solutions Ltd http://www.justsoftwaresolutions.co.uk
Concurrency in the Real World
unsigned i=0;
void func()
{
    for(unsigned c=0;c<2000000;++c)
        ++i;
    for(unsigned c=0;c<2000000;++c)
        --i;
}
std::thread t1(func),t2(func);
Data Races

Final i=0
Final i=4294345393
Final i=169708
Other Race Conditions

A mutex doesn’t save us from race conditions

template<typename Data>
class racy_queue
{
    std::mutex the_mutex;
    std::queue<Data> the_queue;

public:
    void push(Data const& data);
    void pop();
    bool empty();
    Data front();
};
Consider a queue with one element ...

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(q.empty()) return;</td>
<td></td>
</tr>
<tr>
<td>Data local=q.front();</td>
<td></td>
</tr>
<tr>
<td>q.pop();</td>
<td></td>
</tr>
<tr>
<td>if(q.empty()) return;</td>
<td></td>
</tr>
<tr>
<td>Data local=q.front();</td>
<td></td>
</tr>
<tr>
<td>q.pop();</td>
<td></td>
</tr>
</tbody>
</table>
We’re not used to thinking “concurrently”
Things can happen out of order

WISH YOU WERE HERE!
Out of order visibility

Initially $x=\text{false}$, $y=\text{false}$

- $x.\text{store}(\text{true}, \text{relaxed})$
- $y.\text{store}(\text{true}, \text{relaxed})$
- $y.\text{load}(\text{relaxed})$ returns $\text{true}$
- $x.\text{load}(\text{relaxed})$ returns $\text{false}$

write$_x$ then $y$

read$_y$ then $x$
Why we get concurrency wrong

- We work in imperative languages — do this, then do that
- True concurrency is hard to reason about
- Single core systems interleave threads
Problems with multithreaded code

Hidden dependencies are evil
Problems with multithreaded code

Too many notes threads (oversubscription)
Task switching and oversubscription

Core 1

Core 2

Core 1

Core 2
How can we avoid these problem?
Avoiding the Pitfalls

- Share less mutable data
- Use carefully chosen mechanisms for sharing
- Scale the number of threads with the hardware
Sharing less mutable data

No globals
Sharing less mutable data

No singletons either!
Sharing less mutable data

**Copy** data and merge changes later
Communication Mechanisms
template<typename T>
class SynchronizedValue {
    T data;
    std::mutex m;

public:
    struct Updater {
        T* operator->();
        T& operator*();
    };
    Updater operator->();
    Updater get();
    T operator*();
    void set(T const&);
};
void foo(SynchronizedValue<std::string>& s) {
    std::string local=*s;
    s->append("foo");
    auto locked=s.get();
    unsigned pos=locked->find("f");
    if(pos==std::string::npos)
        *locked = "bar";
    else
        (*locked)[pos] = 'b';
}
DataFlowValue

template<typename T>
struct DataFlowValue {
    template<typename F>
    void task(F); // compute value
    void operator=(T value); // set value
    T const& get(); // blocking wait
};
DataFlowValue usage

```c++
void test_sum() {
    DataFlow<int> x, y, z;
    z.task([&]{return x.get()+y.get();});
    y=99;
    x=123;
    assert(z.get()==222);
}
```
Use futures for synchronization

- Futures are a one-way, one-time channel
- Good for passing computation results and “done” flags
- std::async makes it easy to get started.
Use std::async for “divide and conquer” algorithms

template<typename Iter,typename Func>
void parallel_for_each(Iter first,Iter last,Func f) {
    ptrdiff_t const range_length=last-first;
    if(!range_length) return;
    if(range_length==1) {
        f(*first); return;
    }
    Iter const mid=first+(range_length/2);
    std::future<void> bgtask=std::async(
        &parallel_for_each<Iter,Func>,first,mid,f);
    parallel_for_each(mid,last,f);
    bgtask.get();
}
Scaling threads to the hardware

- `std::async()` automatically scales number of threads to the hardware
- Use `std::hardware_concurrency()` when scaling manually
Use message queues for communication between threads
void gui_thread(msg_queue& messages) {
    HANDLE handles[]={messages.event(),...};
    for(;;) {
        MsgWaitForMultipleObjects(arraySize(handles),
            &handles,false,INFINITE,QS_ALLINPUT);
        for(bool do_loop=true;do_loop;) {
            do_loop=false;
            if(message_type local_message=messages.try_pop()) {
                process(local_message); do_loop=true; }
            MSG winMsg={0};
            if(PeekMessage(&winMsg,NULL,0,0,PM_REMOVE)) {
                processWindowsMessage(&winMsg); do_loop=true; }
        }
    }
}
Message types

There are two ways to pass messages:

- As data, like Windows messages,
- As tasks, like the Command pattern
Typed messages

```cpp
struct HandlerBase {
    virtual ~HandlerBase() {}};
template<typename MessageType>
struct MessageHandler: public virtual HandlerBase {
    virtual void handle(MessageType const&)=0;
};
struct MessageBase {
    virtual ~MessageBase() {};
    virtual void dispatch(HandlerBase* handler) const=0;
};
threadsafe_queue<std::shared_ptr<MessageBase>> queue;
```
struct Msg1: MessageBase {
    void dispatch(HandlerBase* handler) const {
        MessageHandler<Msg1>* my_handler =
            dynamic_cast<MessageHandler<Msg1>*>(handler);
        if(my_handler) my_handler->handle(*this);
    }
};

struct Msg2: MessageBase {
    void dispatch(HandlerBase* handler) const;
};

struct MyHandler:
    MessageHandler<Msg1>,
    MessageHandler<Msg2>,
    MessageHandler<Msg3> { /* ... */ };
Task messages

Post tasks rather than messages for

- GUI thread — simpler than multiple custom message types
- Worker threads (e.g. in a thread pool)
threadsafe_queue<std::packaged_task<void()>> queue;

template<typename Func>
std::future<void> post_task_on_queue(Func f) {
    std::packaged_task<void()> task(f);
    std::future<void> res=task.get_future();
    queue.push_back(std::move(task));
    return res;
}
Why add concurrency?
More concurrency ⇒ Faster execution?
How much faster?

Amdahl’s Law:

\[ \gamma = \frac{T_{\text{serial}}}{T_{\text{total}}} \]

\[ S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}} \]
More concurrency $\Rightarrow$ Process more data?
How much more data?

Fixed execution time ⇒

\[ T(\ n_2, \ P) = \frac{T(\ n_2, \ 1)}{S(\ P)} = T(\ n_1, \ 1) \]

How much more can we process in the same time?

\[ O(n) \Rightarrow n_2 \simeq n_1.S(\ P) \]

\[ O(n^2) \Rightarrow n_2 \simeq n_1.\sqrt{S(\ P)} \]

\[ O(\lg n) \Rightarrow n_2 \simeq n_1^{S(\ P)} \]
More concurrency ⇒ Do extra stuff?
What extra stuff?

- Background spell checking
- Incremental search
- Autosuggest
- Background compilation
- Background testing
Background processing must be interruptible
More concurrency $\Rightarrow$ Greater responsiveness?
Improving Responsiveness

Get “work” off the GUI thread
A progress bar (with a cancel button) is better than a “frozen” application.
Don’t move existing GUI elements in the background
The future

- Newer CPUs have more cores rather than faster clock speeds
- New approaches to concurrency will make things easier
just::thread provides a complete implementation of the C++0x thread library for MSVC 2008, MSVC 2010, g++ 4.3 and g++ 4.4.

For a 50% discount go to:

http://www.stdthread.co.uk/accu2010
C++ Concurrency in Action: Practical Multithreading with the new C++ Standard, currently available under the Manning Early Access Program at

http://www.stdthread.co.uk/book/
The images used in this presentation are courtesy of:

Chad Davis
toastmonster
bixentro
Ewan-M
Hljod.Huskona
NASA