

Pocket Hardware Parallelism Today



Processor

- US: Qualcomm Snapdragon 835 Octa core (2.35 GHz Quad + 1.7 GHz Quad)
- International: Samsung Exynos Octa core (2.35 GHz Quad + 1.9 GHz Quad)

THIS IS THE SAMSUNG GALAXY S8, COMING APRIL 21ST

Preorders begin March 30th, three weeks before the release date

by Dieter Bohn | @diboehn | Mar 29, 2017, 11:00am EDT

Photography by Vjeran Pavic and Amelia Holowaty Krales

► Source: The Verge, 29 Mar 2017

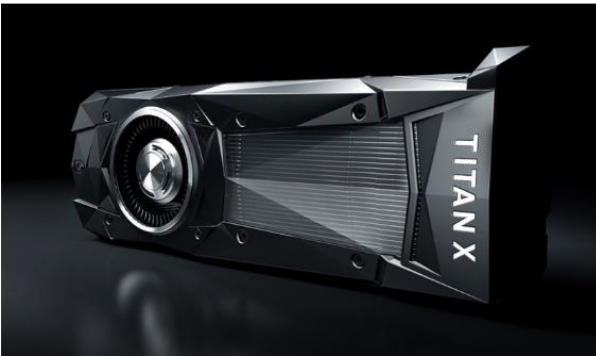
Desktop Hardware Parallelism Today

Ars Technica

Nvidia delivers new and improved Titan Xp—3,840 cores, 550GB/s memory bandwidth

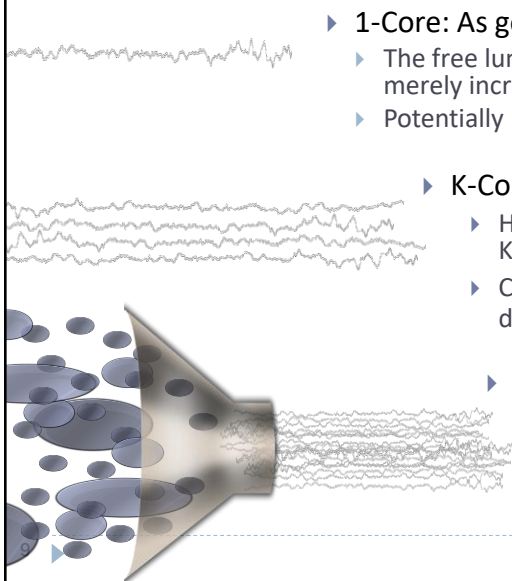
Thu, April 6 8:42 AM • Sebastian Anthony

[Read now](#)



► Source: Ars Technica, 6 Apr 2017

1-Core, K-Core, or N-Core Parallelism?



- ▶ **1-Core: As good as sequential.**
 - ▶ The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
 - ▶ Potentially poor responsiveness.
- ▶ **K-Core: Explicitly fixed throughput.**
 - ▶ Hardwired # of threads that prefer K CPUs (for a given input workload).
 - ▶ Can penalize <K CPUs, doesn't scale >K CPUs.
- ▶ **N-Core: Scalable throughput.**
 - ▶ Workload decomposed into a "sea" of heterogeneous chores.
 - ▶ Lots of latent parallelism we can map down to N cores.

Some Lead Bullets *(useful, but mostly mined)*

- ▶ **Automatic parallelization (e.g., compilers, ILP):**
 - ▶ Amdahl's Law: Sequential programs tend to be... well, sequential.
 - ▶ Requires accurate program analysis: Challenging for simple languages (Fortran), intractable for languages with pointers.
 - ▶ Doesn't actually shield programmers from having to know about concurrency.
- ▶ **Functional languages:**
 - ▶ Contain natural parallelism... except it's too fine-grained.
 - ▶ Use pure immutable data... except those in commercial use.
 - ▶ Not known to be adoptable by mainstream developers.
 - ▶ Borrow some key abstractions/styles from these languages (e.g., closures) and support them in imperative languages.
- ▶ **OpenMP, OpenMPI, et al.:**
 - ▶ "Industrial-strength duct tape," but useful where applicable.

Some New Tools

- ▶ C++17 parallel algorithms:
 - ▶ `std::execution::par`: Parallel chunks in separate threads/cores
 - ▶ Can't perform racy access to the same data, or depend on ordering
 - ▶ `std::execution::par_unseq`: Parallel + within a thread
 - ▶ Can't perform synchronization (e.g., lock a mutex), allocate/deallocate memory, or do other vectorization-unsafe work
 - ▶ `std::execution::seq`: Sequential
 - ▶ Primarily for debugging and comparison
 - ▶ NB: Not the same as "no policy"

- ▶ Examples:

```
for_each( execution::par, begin(a), end(a), [&]{  
    // parallel loop body  
});  
for_each( execution::par_unseq, begin(a), end(a), [&]{  
    // parallel + vectorized loop body  
});
```



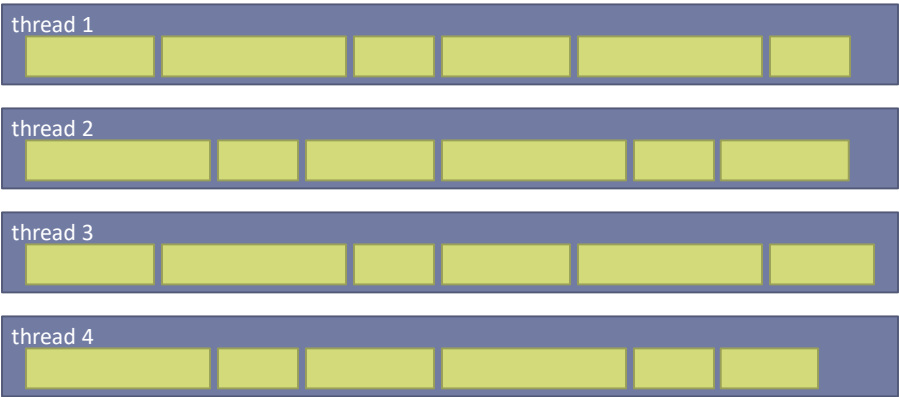
Sequential unsequenced (`std::seq`)

- ▶ Sequential but not sequenced
 - ▶ Not guaranteed to visit elements in the same order as the unparameterized algorithm
 - ▶ But still useful for debugging



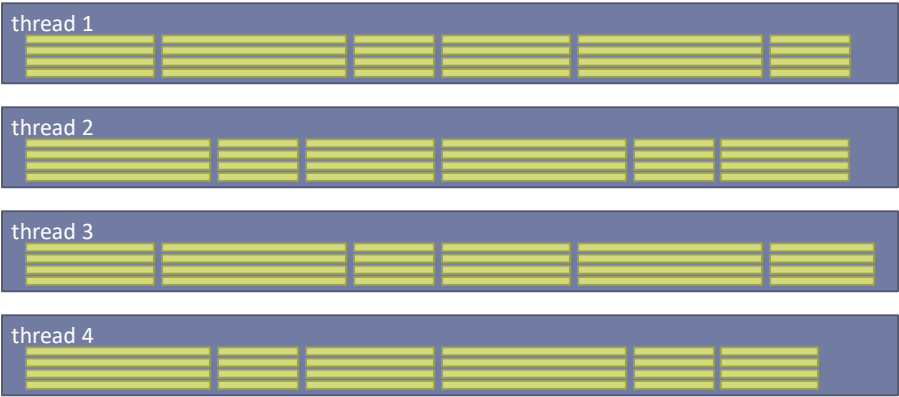
Parallel (std::par)

- ▶ Parallel == take advantage of multiple cores
- ▶ Thread IDs are unique \Rightarrow synchronization is okay (lock, sem)
 - ▶ But watch out for deadlocks



Parallel unsequenced (std::par_unseq)

- ▶ Parallel + vector == use multiple cores + vector (& GPU?) units
- ▶ Thread IDs not unique \Rightarrow no synchronization (e.g., lock, sem)
 - ▶ Includes no allocation (most allocators are synchronized)



Three Pillars of the Dawn			
A Framework for Evaluating Requirements & Tools			
	I. Concurrency for isolation	II. Parallelism for scalability	III. Consistency by synchronization
Tagline	Keep things separate	Re-enable the free lunch	Don't corrupt shared state
Summary	Stay responsive and avoid blocking by running tasks in isolation and communicating via messages	Use more cores to get the answer faster by running operations on groups of things; exploit parallelism in data/algorithm structures	Avoid races by synchronizing access to shared resources, esp. mutable objects in shared memory
Examples	GUIs, web services, bkgd print/compile	Trees, quicksort, compilation	Mutable shared objects in memory, database tables
Key metrics	Responsiveness, latency hiding	Throughput, scalability	Race-free, deadlock-free
Requirements	Isolated, independent	Low overhead, side effect free	Composable, serializable
Old tools & abstractions	Threads, message queues, OpenMPI	Thread pools, OpenMP	Explicit locks, lock-free libraries, transactions
New tools & abstractions	Active objects, futures	Parallel Patterns Library, Threading Building Blocks, Cilk work stealing	Lock-data association, lock ordering (& someday transactional memory?)

Habits of Highly Successful Developers

- ▶ **A Few Good Primitives (preview)**
 - ▶ **Threads, pools, lambdas, futures, locks and atomics**
- ▶ Think In Transactions
 - ▶ Exception safety, public functions, and locked blocks, oh my!
- ▶ Habitually Prefer Structured/Bounded Lifetimes
 - ▶ Locks, tasks, and more
- ▶ Recognize and Avoid Today's Overheads
 - ▶ Switching, contention, and the cost of unrealized parallelism
- ▶ Understand the True Nature of Deadlock
 - ▶ “Not just locks” = locks + messages + I/O + any other blocking

▶

Threads

- ▶ Thread: Today's basic tool for an asynchronous agent.
 - ▶ The abstraction of "identify potentially-long-running independent work to run asynchronously" is sound. We (the industry) will improve the details as we move beyond threads.
- ▶ We won't see these much explicitly, because in Pillar 1 one of the first things we'll do is see how to wrap them inside an Agent-like classes to get better abstractions.



Thread/Work Pools...

- ▶ Thread pool: Today's basic tool for executing chunks of work.
 - ▶ The abstraction of "just identify the chunk of work to be executed asynchronously" is sound. We (the industry) will improve the details as we move beyond thread pools.
- ▶ Under the covers, the implementation "rightsizes" the pool to:
 - ▶ Match available hardware resources.
 - ▶ Avoid deadlock. (Sometimes you need a minimum #threads.)
- ▶ From an example later in the course:

```
void SendPackets( Buffers& bufs ) {  
    auto i = bufs.begin();  
    while( i != bufs.end() ) {  
        auto next = i + min( chunkSize, distance(i, bufs.end()) );  
        pool.run( [=] { SendPackets( i, next ); } );  
        i = next;  
    }  
    pool.join();  
}
```



... Plus Lambda Functions

- ▶ Lambda function: A convenient way to write a functor or “closure” to be executed later/elsewhere.
 - ▶ Maybe “mere syntactic sugar,” but very important sugar.
 - ▶ Available in C#. Trivial example: `(int i) => counter += i`
 - ▶ Available in C++11. Trivial example: `[&] (int i) { counter += i; }`
 - ▶ Coming in Java (8?).
- ▶ I’ll mostly show C++11 syntax:
`[capture-list] (param-list) { statements; }`
 - ▶ `[=]` is shorthand for “capture all locals by value”
 - ▶ `[&]` is shorthand for “capture all locals by reference”
- ▶ From the last slide’s example: `[=] { SendPackets(i, next); }`
 - ▶ Creates a function object that captures local copies of `i` and `next`. All uses of `i` and `next` in the body refer to these copies.
 - ▶ When invoked, executes `SendPackets` with the values of `i` and `next` as they were captured when the closure was created.

Futures

- ▶ Future: An asynchronous value.
 - ▶ Available in Java as `Future<T>`.
 - ▶ Available in C++11 as `future<T>` / `shared_future<T>`.
 - ▶ Available in .NET as `Task<T>`.

- ▶ Contrast a synchronous call:

```
int result = CallSomeFunc(x,y,z);    // synchronous call
// ... code here doesn't run until call completes and result is ready ...
DoSomethingWith( result );           // value available
```

blocks

with an asynchronous call:

```
future<int> result = pool.run( [=] { return CallSomeFunc(x,y,z); } );
// asynchronous call
```

```
// ... code here runs concurrently with CallSomeFunc ...
```

```
DoSomethingWith( result.value() ); // use it when ready
```

may block

Locks

- ▶ **Mutex lock:** Today's best tool for mutual exclusion.
 - ▶ Playing in theaters everywhere.
 - ▶ Fraught with troubles and hard to use correctly, so we'll spend some time on how to use them well.
 - ▶ Learn to love them: There's no replacement on the horizon, unless transactional memory pans out.
- ▶ **Traditional example:**

```
{  
    lock_guard<mutex> lock1( acct1 );  
    lock_guard<mutex> lock2( acct2 );  
    acct1.debit( 100 );  
    acct2.credit( 100 );  
} // release lock1 and lock2
```

Ordered Atomics

- ▶ **Ordered atomic variables** for “lock-free” programming: Today's intricate alternative to for low-lock and lock-free code.
 - ▶ Available in Java as *volatile* and *Atomic**.
 - ▶ Available in .NET as *volatile*.
 - ▶ Available in C++11 as *atomic<T>*, C11 as *atomic_**. (NB: Not *volatile*!)
- ▶ **Semantics and operations:**
 - ▶ Each individual read and write is atomic, no locking required.
 - ▶ Reads/writes are guaranteed not to be reordered.
 - ▶ Compare-and-swap (CAS)... conceptually an atomic execution of:

```
bool atomic<T>::compare_exchange_strong( T& expected, T desired ) {  
    if( this->value == expected ) { this->value = desired; return true; }  
    else { expected = this->value; return false; }  
}
```

 - ▶ **compare_exchange_weak** for use in loops (is allowed to fail spuriously)
 - ▶ **exchange** for when a “blind write that returns the old value” is sufficient

Habits of Highly Successful Developers

- ▶ A Few Good Primitives (preview)
 - ▶ Threads, pools, lambdas, futures, locks and atomics
- ▶ **Think In Transactions**
 - ▶ **Exception safety, public functions, and locked blocks, oh my!**
- ▶ Habitually Prefer Structured/Bounded Lifetimes
 - ▶ Locks, tasks, and more
- ▶ Recognize and Avoid Today's Overheads
 - ▶ Switching, contention, and the cost of unrealized parallelism
- ▶ Understand the True Nature of Deadlock
 - ▶ “Not just locks” = locks + messages + I/O + any other blocking



Errors and Exceptions: In One Slide

- ▶ **Q: What’s an error? A: A function can’t do what it advertised.**
 - ▶ Something happened where it can’t achieve documented results/postconditions.
- ▶ **Q: Error codes vs. exceptions? A: No fundamental difference.**
 - ▶ Both work, just with different syntax/mechanics.
 - ▶ Aside #1: Prefer exceptions.
 - ▶ Error codes are ignored by default. (ouch)
 - ▶ Error codes have to be manually propagated by intermediate code.
 - ▶ Error codes don’t work well for errors in constructors and operators.
 - ▶ Error codes interleave “normal” and “error handling” code.
 - ▶ Aside #2: **Use exceptions only to report errors** (as defined above).
 - ▶ That’s a lot clearer than the circular (and ultimately unsatisfying) advice to “use exceptions for exceptional situations.”



~~Exception~~ Error Safety: In One Slide

- ▶ **What:** Every function must provide one of three guarantees.
 - ▶ **Basic: Consistency.** If the function fails, the system will be in a valid consistent state (but not necessarily unchanged or predictable).
 - ▶ **Strong: All-or-nothing.** If the function fails, the system will be unchanged (e.g., iterators and pointers/references are still valid).
 - ▶ **Nofail:** The function will not fail.
- ▶ **How:** Do a few things consistently.
 - ▶ “Have objects own resources” (aka RAII) to automate cleanup of side effects.
 - ▶ “Do all the work off to the side, then commit (e.g., swap) using nonthrowing operations only.”
 - ▶ “Commit/rollback-critical operations don’t throw,” notably:
 - ▶ using variables of a built-in type (e.g., ints, pointers)
 - ▶ destructors
 - ▶ deallocation functions
 - ▶ swap

Fanfare for the Common Function

- ▶ **The function is the basic unit of work** in every mainstream language.
 - ▶ Only functions do anything!
 - ▶ Sure, we love objects... but that’s why they have member functions.
- ▶ **Every non-private function must be a transaction** on the object(s) it manipulates.
 - ▶ Must take objects from one valid state (precondition) to another, and must preserve any invariants it’s responsible for maintaining.
 - ▶ Any non-private member function of a type T is a transaction on *this, and must maintain the invariants of a T.
 - ▶ Any externally callable member/nonmember function is a transaction on any objects it uses (passed by pointer/reference, or global), and must maintain the invariants of its class/module.
 - ▶ We can distinguish between read-only (\cong const) and read-write accesses and transactions.

Ah, Transactions!

- ▶ A **transaction** must be “**ACID**”:
 - ▶ **Atomic**: All-or-nothing. Other parts of the program should not be able to see part of the work done and part not done.
 - ▶ **Consistent**: Take the parts of the system it operates on from one valid state to another.
 - ▶ **Isolated**: Different transactions should not interfere with each other, even if they are on the same objects.
 - ▶ **Durable**: A committed transaction is never overwritten by second transaction that did not see the results of the first (a “lost update”).
- ▶ Examples of transactions:
 - ▶ Database operations.
 - ▶ **Locked regions**.
 - ▶ Nonprivate methods on objects.
 - ▶ Nonprivate methods on internally synchronized (e.g., “lock-free,” “synchronized”) objects.
 - ▶ Exception-safe functions (NB, *any* guarantee – basic, strong, nofail).

Everybody Loves Transactions (and Keeps Rediscovering Why)

- ▶ Example 1: The lowly database transaction.
 - ▶ T-SQL syntax:

```
BEGIN TRANSACTION T1;           // begin unit of work
UPDATE table1 ...;
SELECT * from table1;
UPDATE table3 ...;
COMMIT TRANSACTION T1;          // publish results atomically
```
 - ▶ java.sql syntax:

```
con.setAutoCommit(false);        // begin longer transaction
Statement stmt = con.createStatement();
stmt.addBatch( "INSERT INTO COFFEES ..." );
stmt.addBatch( "INSERT INTO COFFEES ..." );
stmt.addBatch( "INSERT INTO COFFEES ..." );
stmt.addBatch( "INSERT INTO COFFEES ..." );
int [] updateCounts = stmt.executeBatch();
con.commit();                     // publish results atomically
```
 - ▶ etc.

Everybody Loves Transactions (and Keeps Rediscovering Why)

► Example 2: Locked regions.

► C++11 syntax:

```
{  
    lock_guard<mutex> lock( myMut );           // calls myMut.lock()  
    x = 32;  
    y = y + z * 3.14 * x;  
}                                              // calls myMut.unlock()
```

► C# syntax:

```
lock( obj ) {                               // calls Monitor.Enter(obj)  
    DoSomeWork( x );  
    x += 42;  
}                                              // calls Monitor.Exit(obj):
```

► etc.



Everybody Loves Transactions (and Keeps Rediscovering Why)

► Example 3: **Nonprivate** methods / member functions.

```
void MyClass::MyFunc() {                     // invariants hold here  
    x = 32;  
    y = y + z * 3.14 * x;  
}                                              // invariant reestablished
```



Everybody Loves Transactions (and Keeps Rediscovering Why)

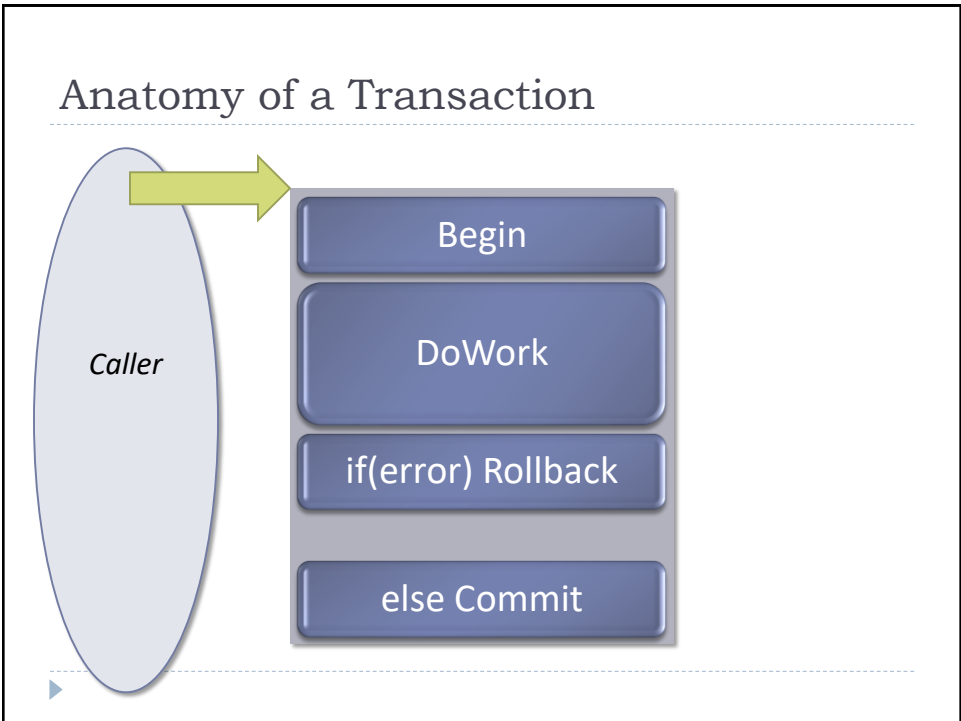
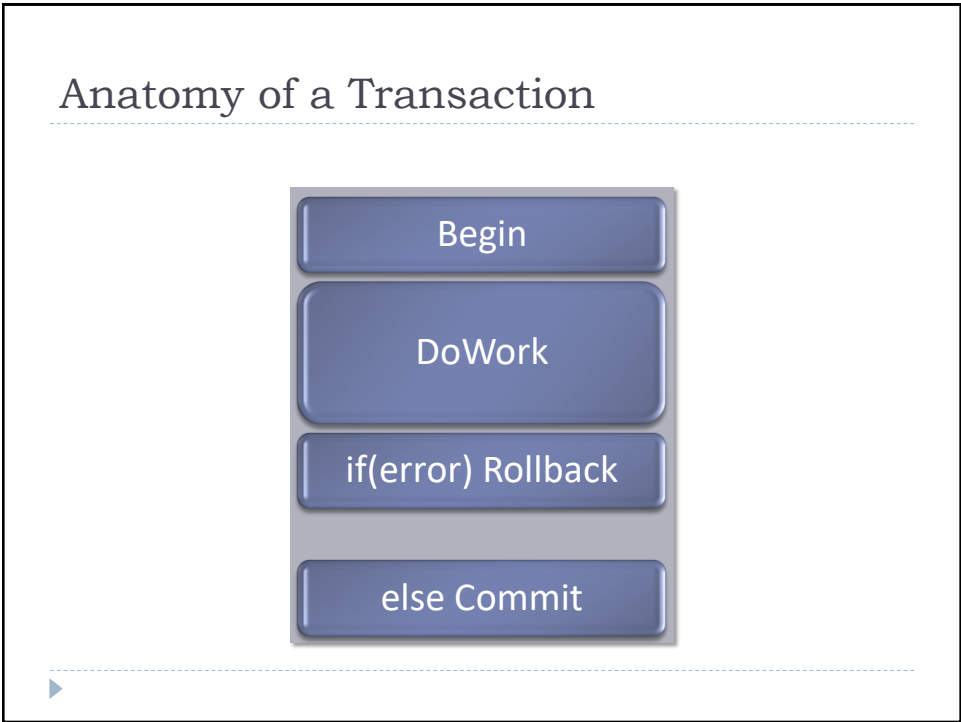
- Example 4: **Exception-safe** methods / functions.

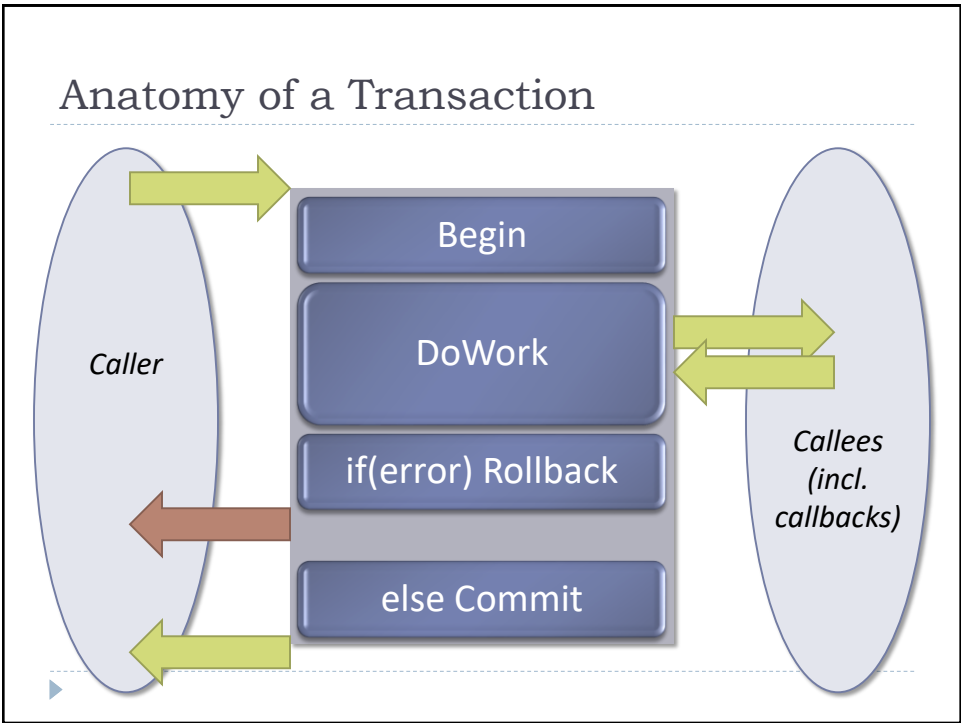
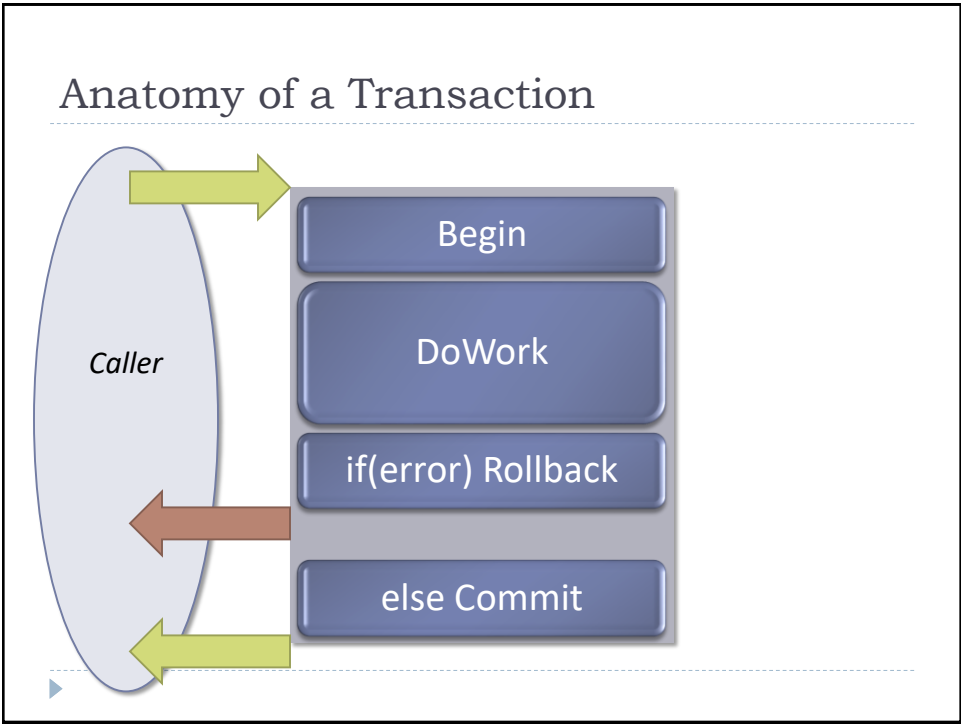
```
void MyClass::TransferFunds(  
    Account acct1,  
    Account acct2,  
    Money amount  
) { // invariants hold here  
    acct1.Debit( amount );  
    try {  
        SomeFunctionThatCouldThrow();  
    } catch(...) {  
        acct1.Credit( amount ); // invariant reestablished  
        throw;  
    }  
    acct2.Credit( amount );  
} // invariant reestablished
```

Everybody Loves Transactions (and Keeps Rediscovering Why)

- Example 5: Handling messages.

```
void MyClass::OnEditMenuClick() { // things are grey/disabled  
    EnableEditMode();  
    MakeSomeOtherButtonsActiveAndClickableToo();  
    Oh_AndMakeOurLittleButterflyMascotIconStartFlappingItsWings();  
} // ok, our GUI is lit up
```



Invariants

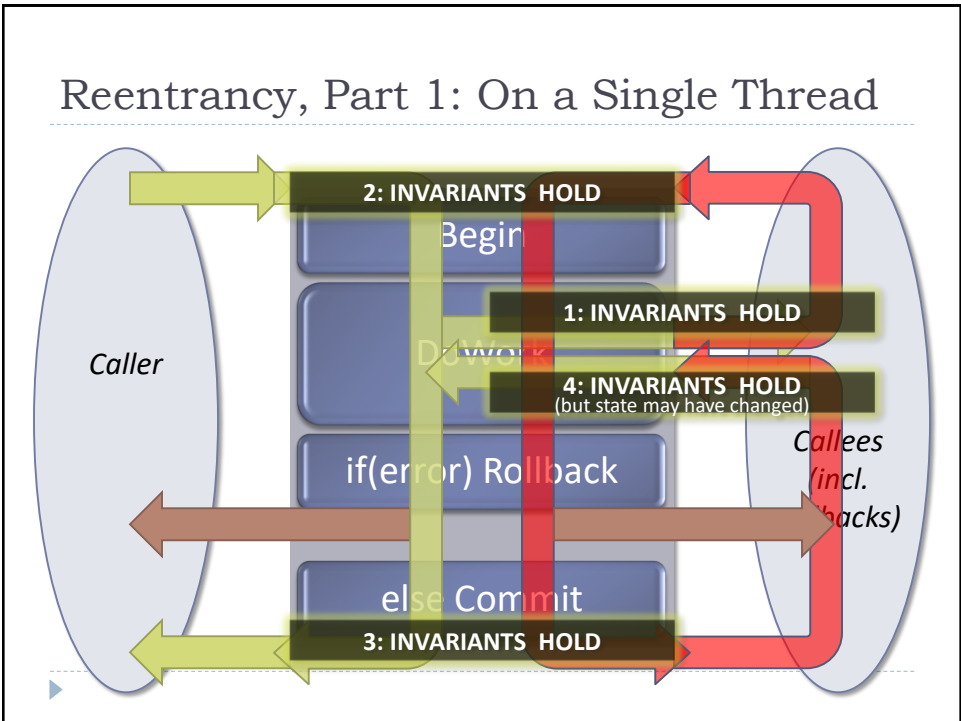
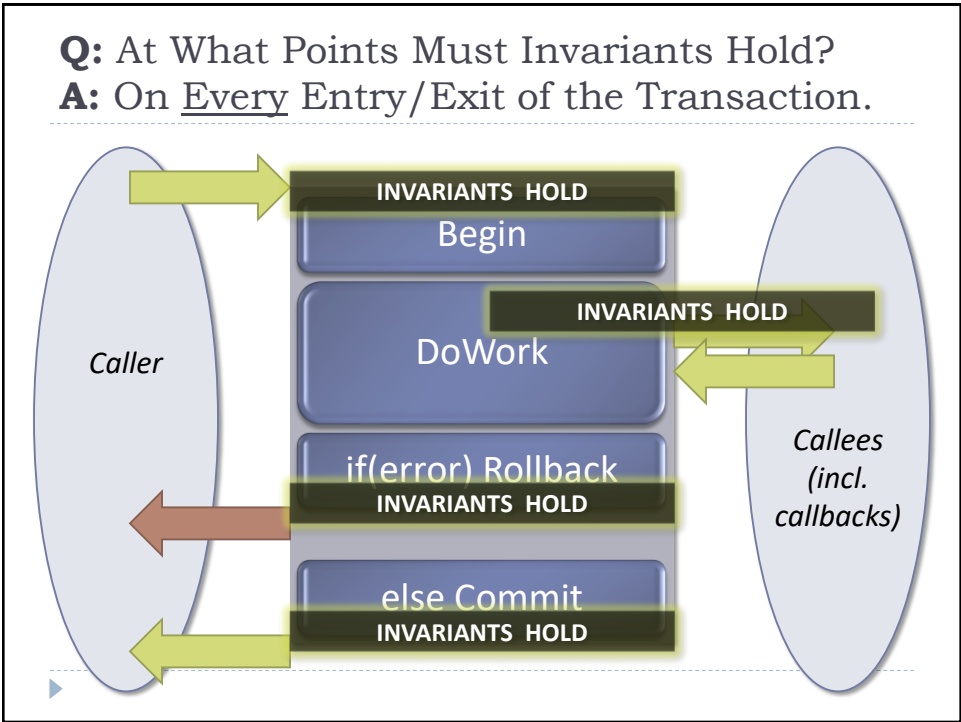
- ▶ **Invariants** are the measure of program consistency or corruption. Examples:
 - ▶ Every object's bits must represent a valid object of its type.
 - ▶ Including transitively to heap objects that form part of the object.
 - ▶ Invariants are set up by the constructor, torn down by the destructor.
 - ▶ Likewise every module, subsystem, etc. has invariants that must be preserved as the system and its parts move from one valid state to another.
- ▶ **Q: In a transaction, at what points must invariants hold?**



Invariants

- ▶ **Invariants** are the measure of program consistency or corruption. Examples:
 - ▶ Every object's bits must represent a valid object of its type.
 - ▶ Including transitively to heap objects that form part of the object.
 - ▶ Invariants are set up by the constructor, torn down by the destructor.
 - ▶ Likewise every module, subsystem, etc. has invariants that must be preserved as the system and its parts move from one valid state to another.
- ▶ **Q: In a transaction, at what points must invariants hold?**
- ▶ **A: Every time we enter or exit the transaction body.**
 - ▶ Invariants are routinely broken in the middle of a transaction. That's okay... as long as they're reestablished before outside code tries to look at the invariant again.





A Small Example (Flawed)

- Imagine you're writing a browser that supports plugins:

```
void CoreBrowser::PrefilterElements() {  
    for( list<PageElement>::iterator i = page.begin(); i != page.end(); ) {  
        if( WeShouldFilterOut( *i ) ) {  
            plugin.OnRemove( *i );  
            i = page.erase( i );  
        }  
        else ++i;  
    }  
    cachedPageSize = page.size(); // update some cached data  
}
```

- Q: What if plugin.OnRemove could reenter this or another function that may traverse or modify this page?



A Small Example (Flawed)

- Imagine you're writing a browser that supports plugins:

```
void CoreBrowser::PrefilterElements() {  
    for( list<PageElement>::iterator i = page.begin(); i != page.end(); ) {  
        if( WeShouldFilterOut( *i ) ) {  
            plugin.OnRemove( *i );  
            i = page.erase( i ); // 2: might not be valid  
        }  
        else ++i;  
    }  
    cachedPageSize = page.size(); // 1: only updated at the end  
}
```

- Q: What if plugin.OnRemove could reenter this or another function that may traverse or modify this page?
 - **1. We'd have to reestablish invariants before calling the plugin.**
Example problem: cachedPageSize is not reestablished until the end.
 - **2. We couldn't assume that our state hasn't changed.**
Example problem: The iterator may be invalidated. The list will be in a consistent state, but not necessarily the same state.



Attempt #2: Better (On a Single Thread)

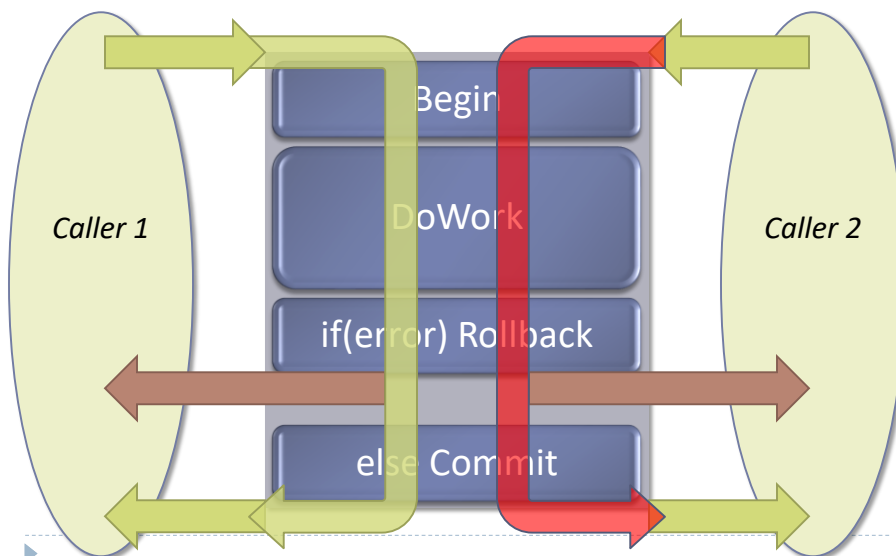
- ▶ Here's a way to address the reentrancy issues:

```
void CoreBrowser::PrefilterElements() {  
    for( list<PageElement>::iterator i = page.begin(); i != page.end(); ) {  
        if( WeShouldFilterOut( *i ) ) {  
            plugin.OnRemove( *i );    // require that plugin must not call  
                                     // anything that could modify page  
            i = page.erase( i );      // immediately update cached data so  
                                     // invariant holds on next plugin call  
            --cachedPageSize;  
        }  
        else ++i;  
    }  
}
```

- ▶ Reentrant nested transaction case: Addressed.
 - ▶ We've reestablished a valid cachedPageSize before the next call to the plugin.
 - ▶ We've required that the plugin not modify this page.



Reentrancy, Part 2: Concurrent Calls



Attempt #3: Better + Concurrency-Safe

- ▶ Here's a way to address both reentrancy and concurrency:

```
void CoreBrowser::PrefilterElements() {  
    lock_guard<mutex> lock( mutPage ); // lock this transaction  
    for( list<PageElement>::iterator i = page.begin(); i != page.end(); ) {  
        if( WeShouldFilterOut( *i ) ) {  
            plugin.OnRemove( *i ); // require that plugin must not call  
            i = page.erase( i );    // anything that could modify page  
            --cachedPageSize;       // immediately update cached data so  
        }                          // invariant holds on next plugin call  
        else ++i;  
    }  
}
```

- ▶ Concurrency race case: Solved. No race, thanks to mutPage.
- ▶ Reentrant nested transaction case: Addressed.
 - ▶ If mutPage isn't reentrant: Plugin will break immediately at test time.
 - ▶ If mutPage is reentrant: We've reestablished a valid cachedPageSize before the next call to the plugin, and required that the plugin not modify this page.



Controlling Overlapping/Conflicting Transactions

- ▶ **Key commonality: Two transactions that can be running at the same time (interleaved or simultaneously) on the same data.**
 - ▶ In ACID terms, we'd be breaking both "Atomic" and "Isolated."
- ▶ For concurrent calls: Use synchronization to prevent races.
 - ▶ Definition: A **race** occurs when two threads can access the same memory concurrently, and at least one is a write.
 - ▶ We don't want the other thread to see broken invariants. Synchronization prevents the two transactions from interfering.
- ▶ For same-thread reentrancy: Reestablish invariants first before, and reload state again after, calling out into possibly-reentrant code.
 - ▶ We're splitting our larger transaction into smaller parts that can interleave with other transactions at well-defined points.
 - ▶ Don't assume state hasn't changed. After the call your invariants must hold, but they might have been temporarily broken and reestablished with new values. Refresh any local copies of previous state.



Anatomy of a Transaction: Examples			
	Non-private function	Locked region	Function on a monitor
Begin	Enter function	Acquire lock	Enter function and acquire lock
Do Work	Reestablish invariants before calling out	Reestablish invariants before calling out	Reestablish invariants (incl. release lock!) before calling out
Rollback	Reestablish invariants and throw (or, return error)	Reestablish invariants (incl. release lock!) and exit block	Reestablish invariants (incl. release lock!) and throw (or, return error)
Commit	Reestablish invariants and return result	Reestablish invariants (incl. release lock!)	Reestablish invariants (incl. release lock!) and return result

- Habits of Highly Successful Developers
- ▶ A Few Good Primitives (preview)
 - ▶ Threads, pools, lambdas, futures, locks and atomics
 - ▶ Think In Transactions
 - ▶ Exception safety, public functions, and locked blocks, oh my!
 - ▶ **Habitually Prefer Structured/Bounded Lifetimes**
 - ▶ **Locks, tasks, and more**
 - ▶ Recognize and Avoid Today's Overheads
 - ▶ Switching, contention, and the cost of unrealized parallelism
 - ▶ Understand the True Nature of Deadlock
 - ▶ “Not just locks” = locks + messages + I/O + any other blocking



Structured Programming

► Consider:

```
void f() {  
  
    // ...  
  
    g();    // ⇒ jump to function g here and then...  
    // ⇐ ...return from function g and continue here!  
  
    // ...  
  
}
```



Structured vs. Unstructured: Examples			
	Object lifetimes	Thread/task lifetimes	Lock lifetimes
Structured Examples			
Unstructured Examples			
Unstructured Costs and Drawbacks			

Structured vs. Unstructured: Examples			
	Object lifetimes	Thread/task lifetimes	Lock lifetimes
Structured Examples	Function local vars: <ul style="list-style-type: none">• C++ stack scope• C# using blocks• Java dispose pattern By-value parameters By-value nested objects		
Unstructured Examples	Global objects Heap/lib allocation		
Unstructured Costs and Drawbacks	Nondeterministic finalization timing Allocation overhead Tracking overhead to not use after delete: <ul style="list-style-type: none">• GC• smart pointers Ownership cycles		

Structured vs. Unstructured: Examples

	Object lifetimes	Thread/task lifetimes	Lock lifetimes
Structured Examples	Function local vars: <ul style="list-style-type: none">• C++ stack scope• C# using blocks• Java dispose pattern By-value parameters By-value nested objects	Recursive/data decomposition: <ul style="list-style-type: none">• Subrange per nested stack call• Partitioned sub-ranges, parallel calls Join-before-return, internal parallelism	
Unstructured Examples	Global objects Heap/lib allocation	Spawn “new thread()” Spawn process	
Unstructured Costs and Drawbacks	Nondeterministic finalization timing Allocation overhead Tracking overhead to not use after delete: <ul style="list-style-type: none">• GC• smart pointers Ownership cycles	Nondeterministic execution/join timing Allocation overhead Tracking overhead to not use task after EOT, join task before EOP Ownership/waiting cycles	

Structured vs. Unstructured: Examples

	Object lifetimes	Thread/task lifetimes	Lock lifetimes
Structured Examples	Function local vars: <ul style="list-style-type: none">• C++ stack scope• C# using blocks• Java dispose pattern By-value parameters By-value nested objects	Recursive/data decomposition: <ul style="list-style-type: none">• Subrange per nested stack call• Partitioned sub-ranges, parallel calls Join-before-return, internal parallelism	Scoped locking: <ul style="list-style-type: none">• C++ lock_guard• C# lock blocks• Java synchronized blocks Release-before-return
Unstructured Examples	Global objects Heap/lib allocation	Spawn “new thread()” Spawn process	Explicit “lock” call, with no automatic unlock or “finally unlock” call
Unstructured Costs and Drawbacks	Nondeterministic finalization timing Allocation overhead Tracking overhead to not use after delete: <ul style="list-style-type: none">• GC• smart pointers Ownership cycles	Nondeterministic execution/join timing Allocation overhead Tracking overhead to not use task after EOT, join task before EOP Ownership/waiting cycles	Nondeterministic lock acq/rel ordering Allocation overhead Tracking overhead to avoid deadlock, priority inversion, ... Waiting/blocking cycles (deadlock)

Prefer Structured/Bounded Lifetimes

- ▶ Of what? Several things:
 - ▶ Objects, esp. ones that own resources.
 - ▶ Tasks (e.g., structured fork/join).
 - ▶ Locked regions.
- ▶ There will be exceptions, but always look to “scoped”/“structured” as the default and make you sure you understand why you opt out to unstructured uses.
- ▶ Follows directly from thinking in transactions:
 - ▶ Transactions are scoped.
 - ▶ Transactions nest.
- ▶ Follows directly from “structured programming”:
 - ▶ More than just a holdover from the 1970s!



Example: Naïve Sequential Quicksort

- ▶ Here’s a sequential implementation of quicksort, simplified to its essentials:

```
template<typename RAlter>
void Quicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    Quicksort( first, pivot );
    Quicksort( pivot, last );
}
```
- ▶ Traditional divide-and-conquer, performed sequentially.



Quicksorting in Parallel, Take 1 (Flawed)

- ▶ Now consider the following parallelized version of the code:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
}
```

- ▶ **Q: What's wrong with this code?**



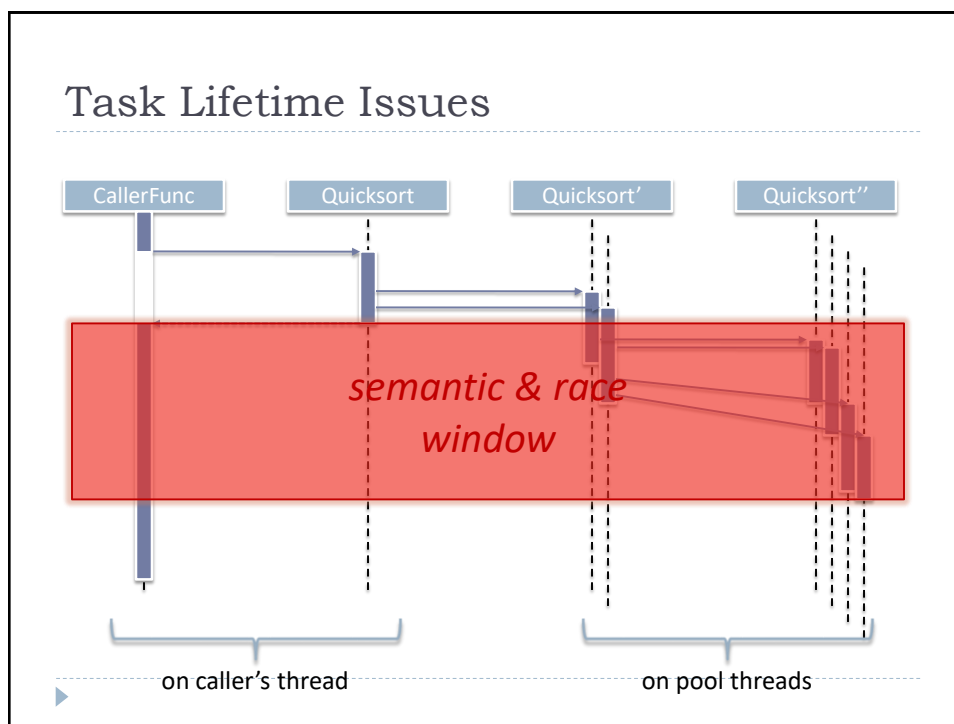
Quicksorting in Parallel, Take 1 (Flawed)

- ▶ Now consider the following parallelized version of the code:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
}
```

- ▶ **Q: What's wrong with this code?** A: This doesn't guarantee that the subrange sorts will be complete before we return to the caller.
- ▶ **Q2: So what?**





Quicksorting in Parallel, Take 1 (Flawed)

- ▶ Now consider the following parallelized version of the code:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
}
```

- ▶ **Q: What's wrong with this code?** A: This doesn't guarantee that the subrange sorts will be complete before we return to the caller.
- ▶ **Q2: So what?** A: Because to the caller this is a synchronous API:
 - ▶ **Completion of side effects:** The caller expects the range is sorted.
 - ▶ **Synchronization:** The caller expects we won't still be accessing the data, which would race with the caller's subsequent uses of the data.

Quicksorting in Parallel, Take 2 (Better)

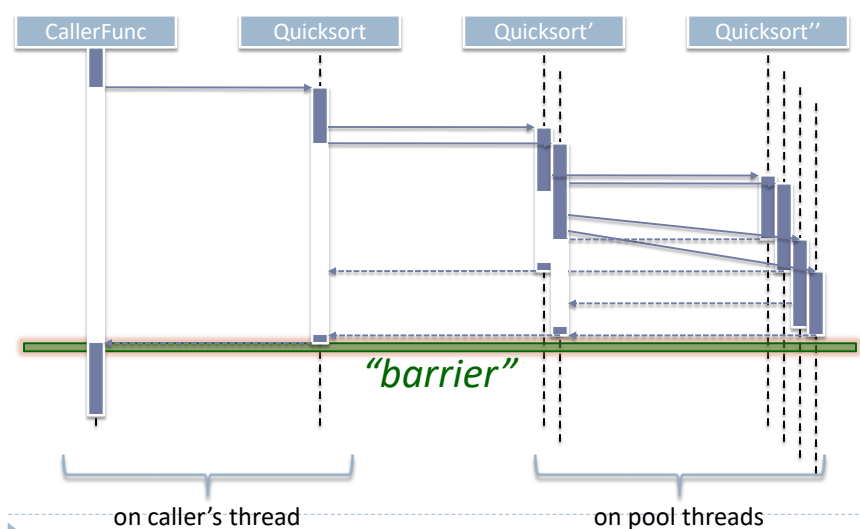
- ▶ Another try:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    f1.wait();
    f2.wait();
}
```

- ▶ Now we've returned the desired amount of synchronicity:
 - ▶ **Completion of side effects:** That the function did its job and the range is sorted.
 - ▶ **Synchronization:** That ParallelQuicksort won't still be accessing the data, and so won't race with the caller's subsequent uses of the data.



Scoped Task Lifetimes



What Have We Learned?

- ▶ Just because some work is done in parallel (e.g., internally behind a synchronous API) doesn't mean there isn't any sequential synchronization with the caller.
 - ▶ Usually there is.
 - ▶ Runtime systems can often exploit this to give you better performance. (Especially future runtimes...)
- ▶ Structured task lifetimes make it easier to reason correctly about our code.
 - ▶ Unstructured lifetimes can be appropriate, but be suspicious of them by default.
- ▶ When a synchronous function returns to the caller, even if it did internal work in parallel it must guarantee:
 - ▶ **Completion of side effects:** That it did its job completely.
 - ▶ **Synchronization:** That it won't still be accessing data provided by the caller, which would race with the caller's subsequent uses of that data.

Structured vs. Unstructured: The Graphic Novel



Habits of Highly Successful Developers

- ▶ A Few Good Primitives (preview)
 - ▶ Threads, pools, lambdas, futures, locks and atomics
- ▶ Think In Transactions
 - ▶ Exception safety, public functions, and locked blocks, oh my!
- ▶ Habitually Prefer Structured/Bounded Lifetimes
 - ▶ Locks, tasks, and more
- ▶ **Recognize and Avoid Today's Overheads**
 - ▶ **Switching, contention, and the cost of unrealized parallelism**
- ▶ Understand the True Nature of Deadlock
 - ▶ “Not just locks” = locks + messages + I/O + any other blocking



Recall: **Naïve** Sequential Quicksort (Flawed)

- ▶ Here's a sequential implementation of quicksort, simplified to its essentials:

```
template<typename RAlter>
void Quicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    Quicksort( first, pivot );
    Quicksort( pivot, last );
}
```

- ▶ **Q: What's wrong with this sequential code?**



Recall: **Naïve** Sequential Quicksort (Flawed)

- ▶ Here's a sequential implementation of quicksort, simplified to its essentials:

```
template<typename RAlter>
void Quicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    Quicksort( first, pivot );
    Quicksort( pivot, last );
}
```

- ▶ **Q: What's wrong with this sequential code?**
- ▶ A: A real quicksort implementation would use a cutoff to sort small subranges using a different algorithm.



Sequential Quicksort, Take 2 (Better)

- ▶ Here's a more typical sequential implementation of quicksort, simplified to its essentials:

```
template<typename RAlter>
void Quicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit )
        { OtherSort(first,last); return; }
    RAlter pivot = Partition( first, last );
    Quicksort( first, pivot );
    Quicksort( pivot, last );
}
```

- ▶ Someone chooses "limit" appropriately.



Recall: Quicksorting in Parallel, Take 2 (Better)

- ▶ Here's the code we came up with before:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );

    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );

    f1.wait();
    f2.wait();
}
```

- ▶ **Q: What's wrong with this code?**



Recall: Quicksorting in Parallel, Take 2 (Better)

- ▶ Here's the code we came up with before:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );

    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );

    f1.wait();
    f2.wait();
}
```

- ▶ **Q: What's wrong with this code?**
- ▶ A (today): Most of the spun-off work will be too fine-grained.
 - ▶ Most work items are at the leaves of the computation and process containing subranges of a few elements or even just one.
 - ▶ Just as sequential code typically to switches to something else when the subrange's size falls below a threshold, similarly in parallel code for small ranges we should switch to a sequential sort.



Recall: Quicksorting in Parallel, Take 2 (Better)

- ▶ Here's the code we came up with before:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= 1 ) return;
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    f1.wait();
    f2.wait();
}
```

- ▶ **Q: What's wrong with this code?**
- ▶ A' (tomorrow): Nothing.
 - ▶ Upcoming runtimes based on work stealing will drive down the cost of unrealized parallelism to nearly nothing.



Quicksorting in Parallel, Take 3 (Better)

- ▶ Another try:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit ) { OtherSort(first,last); return; }
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    f1.wait();
    f2.wait();
}
```

- ▶ **Q: What can we improve further?**



Quicksorting in Parallel, Take 3 (Better)

- ▶ Another try:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit ) { OtherSort(first,last); return; }
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    f1.wait();
    f2.wait();
}
```

- ▶ **Q: What can we improve further?**
- ▶ A: Possible multiple context switches on the wait.
 - ▶ Prefer to use a wait-all primitive where available.

▶

Quicksorting in Parallel, Take 4 (Better)

- ▶ Another try:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit ) { OtherSort( first, last ); return; }
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    wait_all( f1, f2 );
}
```

- ▶ **Q: What can we improve *still* further?**

▶

Quicksorting in Parallel, Take 4 (Better)

- ▶ Another try:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit ) { OtherSort( first, last ); return; }
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    future f2 = global_pool.run( [&]{ ParallelQuicksort( pivot, last ); } );
    wait_all( f1, f2 );
}
```

- ▶ **Q: What can we improve *still* further?**
- ▶ A (today): Avoid one context switch per call by keeping the last piece of work.
 - ▶ Again, this cost will go away on upcoming work-stealing runtimes.



Quicksorting in Parallel, Take 5 (Best)

- ▶ Finally:

```
template<typename RAlter>
void ParallelQuicksort( RAlter first, RAlter last ) {
    if( distance(first,last) <= limit ) { OtherSort( first, last ); return; }
    RAlter pivot = Partition( first, last );
    future f1 = global_pool.run( [&]{ ParallelQuicksort( first, pivot ); } );
    ParallelQuicksort( pivot, last );
    wait( f1 );
}
```



Habits of Highly Successful Developers

- ▶ A Few Good Primitives (preview)
 - ▶ Threads, pools, lambdas, futures, locks and atomics
- ▶ Think In Transactions
 - ▶ Exception safety, public functions, and locked blocks, oh my!
- ▶ Habitually Prefer Structured/Bounded Lifetimes
 - ▶ Locks, tasks, and more
- ▶ Recognize and Avoid Today's Overheads
 - ▶ Switching, contention, and the cost of unrealized parallelism
- ▶ **Understand the True Nature of Deadlock**
 - ▶ **"Not just locks" = locks + messages + I/O + any other blocking**



Basic Concepts: Deadlock

- ▶ **Q: What is deadlock?**



Basic Concepts: Deadlock

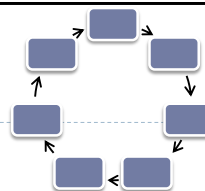
- ▶ **Q: What is deadlock?**
- ▶ A1: My application, or some part thereof, seizes up for no apparent reason.
 - ▶ **Weak:** Yes, that's a symptom of deadlock. (But it could be caused by other things, too.)
- ▶ A2: When two threads each try to take a lock the other already holds.
 - ▶ **Better:** We have a potential deadlock anytime two threads can try to acquire the same two locks in opposite orders.
- ▶ A3: When N threads enter a locking cycle where each tries to take a lock its neighbor already holds.
 - ▶ **Almost there:** We have a potential deadlock anytime there can be a (b)locking cycle.
- ▶ A4: When N threads enter a cycle where each is waiting for its neighbor.
- ▶ **Best:** Deadlock = waiting cycle.

Deadlock Examples

- ▶ Locks:

```
// Thread 1
mut1.lock();
mut2.lock();    // blocks
```

```
// Thread 2
mut2.lock();
mut1.lock();    // blocks
```



Deadlock Examples

► Locks:

```
// Thread 1
mut1.lock();
mut2.lock(); // blocks
```

```
// Thread 2
mut2.lock();
mut1.lock(); // blocks
```

► Messages:

```
// Thread 1
mq1.receive(); // blocks
mq2.send( x );
```

```
// Thread 2
mq2.receive(); // blocks
mq1.send( y );
```

A More Realistic Deadlock

► Consider:

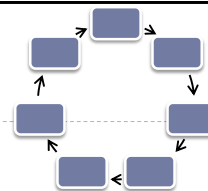
```
// Thread 1
Win::HandleMessage(...) {
  mut.lock(); // blocks

  // Thread 3
  OpenFile( "xyz", "w+" );
  w.SendMessage(...); // blocks
```

```
// Thread 2
OpenFile( "xyz", "w" ); // blocks
mq.send( x );

// Thread 4
mut.lock();
mq.receive(); // blocks
```

You (or Your User) Could Be Part of the Problem



// Thread 1 (running in the CPU hardware)

`mut.lock();`

`char = ReadKeystroke();`

// blocks

// Task 2 (running in your brain's wetware)

Wait for the prompt to appear.

// blocks

OK, now start typing.

// Thread 3 (back in the hardware again)

`mut.lock();`

`Print ("Press 'Any' key to continue...");`

// blocks



Remember This Example?

► Consider:

```
future<int> result = pool.run( [=] { return CallSomeFunc(x,y,z); } );
```

// ... code here runs concurrently with CallSomeFunc ...

```
DoSomethingWith( result.value() );
```

with adding a lock:

```
future<int> result = pool.run( [=] { return CallSomeFunc(x,y,z); } );
```

// locks mut (!)

// ... code here runs concurrently with CallSomeFunc ...

```
lock(mut);
```

```
DoSomethingWith( result.value() );
```



Deadlock Summary

- ▶ Deadlock can arise whenever there is a blocking (or waiting) cycle among concurrent tasks, where each one is waiting for the next to produce some value or release some resource.
- ▶ Eliminate deadlocks as much as possible:
 - ▶ Apply ordering techniques like lock hierarchies and message contracts. These techniques are important, even though they are incomplete because each one deals with only a specific kind of waiting.
 - ▶ Consider adding your own deadlock detection by instrumenting the wait points in your code in a uniform way.
- ▶ Spelling note: A more correct name for deadlock could be “deadblock”...
 - ▶ But the world has already adopted a common spelling that’s one letter shorter, and this isn’t the time to try to change that.
 - ▶ Just remember the ‘b’, even though it’s silent.



Habits of Highly Successful Developers

- ▶ A Few Good Primitives (preview)
 - ▶ Threads, pools, lambdas, futures, locks and atomics
- ▶ Think In Transactions
 - ▶ Exception safety, public functions, and locked blocks, oh my!
- ▶ Habitually Prefer Structured/Bounded Lifetimes
 - ▶ Locks, tasks, and more
- ▶ Recognize and Avoid Today's Overheads
 - ▶ Switching, contention, and the cost of unrealized parallelism
- ▶ Understand the True Nature of Deadlock
 - ▶ “Not just locks” = locks + messages + I/O + any other blocking

