

Concurrency in Software Designs: How to Avoid Nasty Surprises?

Prof. Dr. Luc Bläser

Hochschule für Technik Rapperswil

Concurrency Becomes Inevitable

- Imposed by Designs
 - Making GUIs responsive
 - External libraries
 - Distributed systems
 - Some language features
- Accelerating Performance
 - Clock speed at ceiling
 - Parallelizing on multi-cores
 - Must program this explicitly
 - “The free lunch is over” – H. Sutter

So Much Can Go Wrong

- New Sorts of Potential Errors

- Race Conditions
- Deadlocks & Livelocks
- Starvations

- Non-Deterministic Errors

- Occur Sporadically
- Hard to find
- Hard to test

Should be alarmed

- “No clue why this test failed. We reran it many times and it stays green now.”
- “The software hangs on very rare occasions. Just restart it; nothing harmful.”

Therac-25 RT Linac



Patients died by overradiation
Cause: Race condition

N. G. Leveson, C. S. Turner, *An Investigation of the Therac-25 Accidents*,
IEEE Computer, Volume 26, Issue 7 (Jul 1993), pp. 18-41

Talk Outline

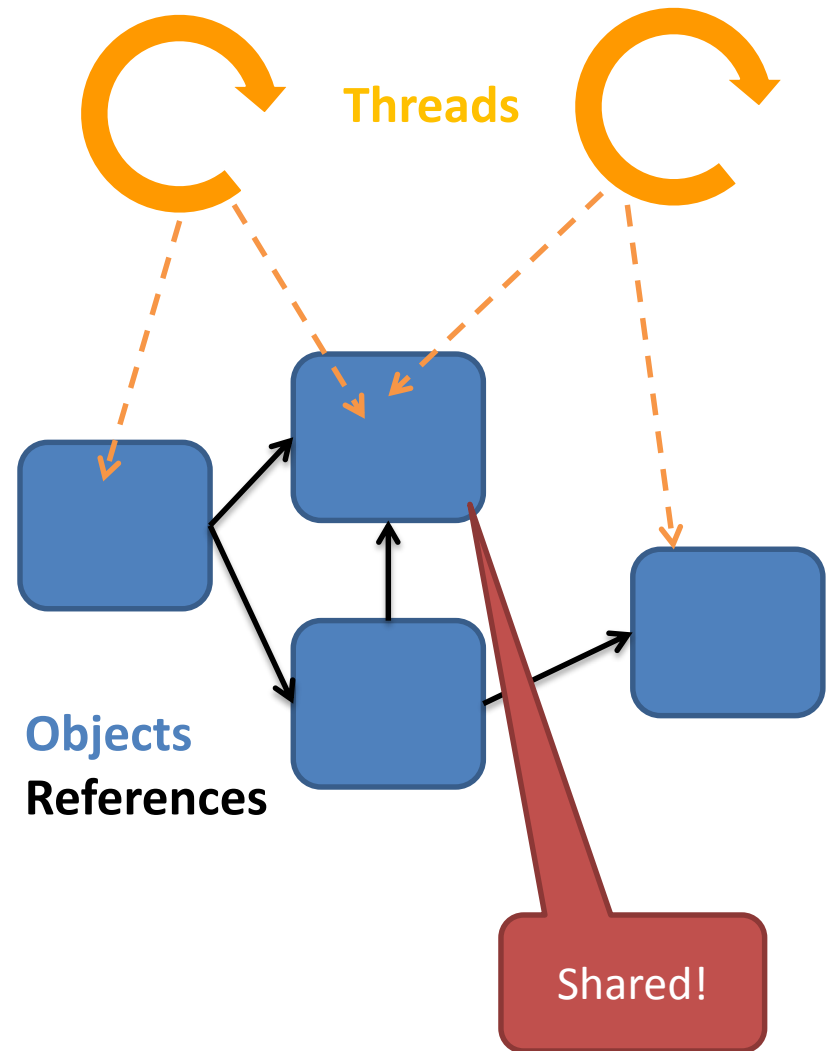
- Concurrency Essentials
 - Multi-Threading in a Nutshell
 - Safety & Liveness Criteria
- Structuring Concurrency
 - Design in Architectures
 - Scenarios, patterns & pitfalls
 - Producer / Consumer
 - Responsive UI / logic
 - Algorithmic parallelization
- Conclusions

Sources of Concurrency in .NET

- Explicit Threads
 - TPL Tasks (Thread Pools)
 - Parallel LINQ
 - Finalizers, “C# Destructors”
 - Web services, sockets
 - Background-Worker
 - Asynchronous calls
 - async/await
 - Timers
 - External calls/callbacks
 - Across Processes (via DB, files, network etc.)
- Multi-threading underneath
-

Concurrent Programming Model

- Threads operating on passive objects
 - Via direct or indirect method calls
 - Parallel or arbitrary interleaving
 - By default, uncontrolled
 - Need to synchronize explicitly
- Better models exist
 - Actors, (STM)



Predestinates of Race Conditions

```
int Next() {  
    return counter++;  
}
```

```
if (!loaded) {  
    loaded = true;  
    Internalize();  
}
```

```
void Swap() {  
    t = y; y = x; x = t;  
}
```

```
foreach (T item in threadSafeCollection) {  
    ...  
}
```

```
while (safeBuffer.Size > 0) {  
    safeBuffer.RemoveFirst();  
}
```

```
if (weakReference.IsAlive) {  
    weakReference.Value.Op();  
}
```

And so on...

Race Condition

- Insufficiently synchronized accesses on shared resources
 - Erroneous or undefined behavior
 - Depends on timing/interleaving of concurrent execution
- Low level: Data races
 - Concurrent accesses without sync
 - On same variable or array element
 - Read-write, write-read, write-write
- High level: Unsynchronized sequences
 - Critical (atomic) sections not ensured

Synchronization Abstractions in .NET

- Monitor (aka C# lock with Wait & Pulse)
- Reader-writer lock
- Concurrent collections
- Primitives: Semaphore, Barrier, Mutex, CountdownEvent, wait handles, ...
- Thread/Task Joins
- Memory level: Interlocked, volatile, barriers/fences

Fixing Races, Causing Deadlocks

```
class Repository { ...  
    void CopyTo(Repository target) {  
        lock(this) {  
            // get  
            target.Add(content);  
        }  
    }  
  
    public void Add(T content) {  
        lock(this) {  
            // add  
        }  
    }  
}
```

Nested lock

Deadlock

T1 locks a
T2 locks b
T1 wants b
T2 wants a

lock a
lock b

Thread T1
a.CopyTo(b);

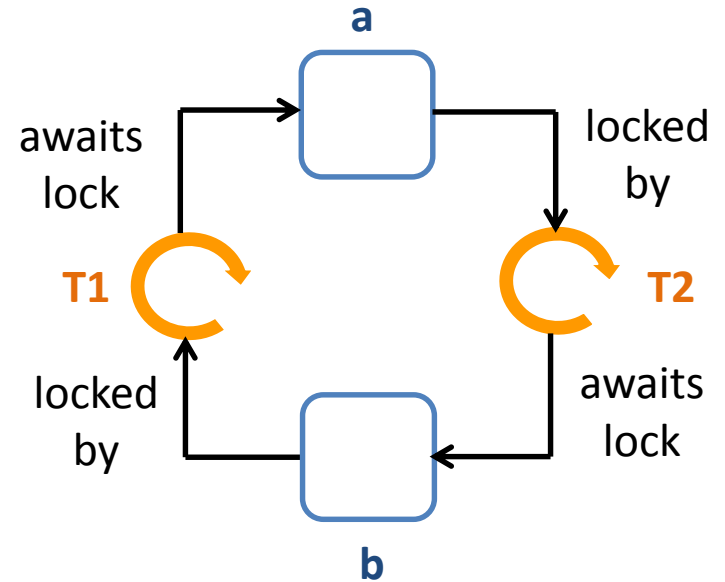
lock b
lock a

Thread T2
b.CopyTo(a);

Deadlock

- Threads wait for each to release a resource such that none can proceed
 - Nested locks
 - Cyclic wait dependencies
 - (Mutual blocking without timeouts)
- Livelock = Deadlock consuming processor while waiting

```
while (!Wait(timeout)) { }
```




Starvation: Not Much Better

- Fairness problem
 - A thread may never get the chance to access a resource
 - Others could continuously overrun the waiting thread
- Frequent candidates
 - Timeout and retry
 - Thread priorities
 - Optimistic concurrency control
 - Self-designed read/write locks
 - .NET sync primitives do not guarantee strict fairness

```
a.Lock();  
while (!b.TryLock()) {  
    a.Unlock();  
    // let others continue  
    a.Lock();  
}
```

Safety & Liveness Conditions

- Mutual exclusion
 - Critical sections on shared resources are properly synchronized
 - No deadlocks
 - Threads cannot lock each other for indefinite time
 - No starvation
 - Thread waiting for a condition should proceed after some time if the condition is sufficiently often fulfilled
- 

Testing Concurrency Bugs?

- Errors are time-dependent / non-deterministic
 - May show up sporadically and extremely seldom
- Multi-thread testing has limitation
 - Would need to test all relevant / possible interleavings
 - Watch sporadic failures!
- Other approaches needed
 - Static checkers?
 - Analytical approach

Concurrency Checkers

- Static analysis is an unfulfillable wish
 - Exponential state explosion
 - Eventually as hard as «halting problems»
 - Good results with some model and dynamic checkers
 - MSR Chess for .NET (no longer maintained)
 - Intel Inspector XE (also for .NET)
 - Java Pathfinder and others
 - Own new research project...
- Most analysis tools only detect primitive style issues
 - VS Code Analysis, FxCop, Resharper
 - Detects empty lock-blocks, locking null etc.

Need for Analytical Approach

- Our problem: Threads can run arbitrarily on objects
 - Which threads may access which objects?
 - Which objects must be thread-safe and which not?
 - Exist potentially cyclic wait dependencies?

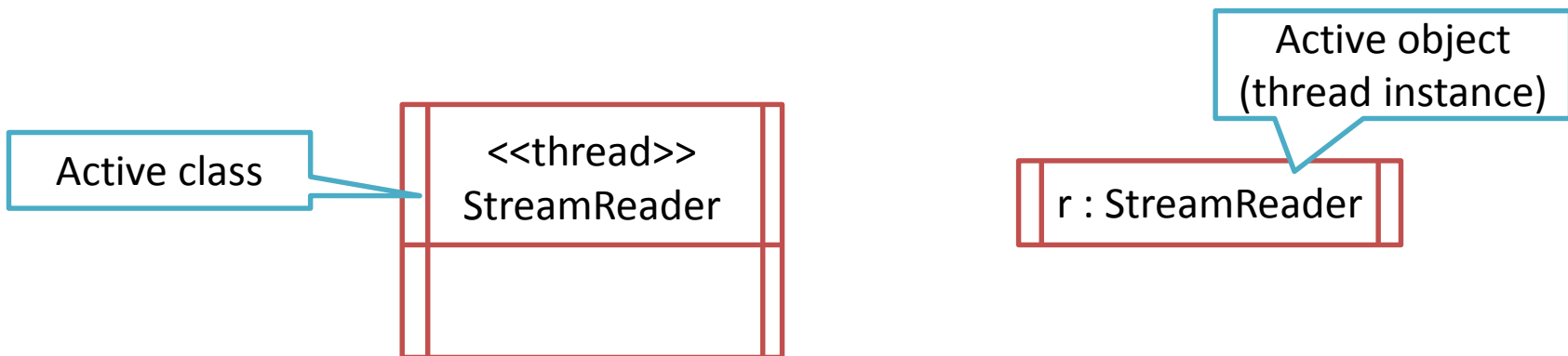
Need Concurrency Design for Architecture!

Concurrency Model in SW Architecture

1. Identify active instances
2. Specify interactions
3. Define synchronization
4. Reason about correctness

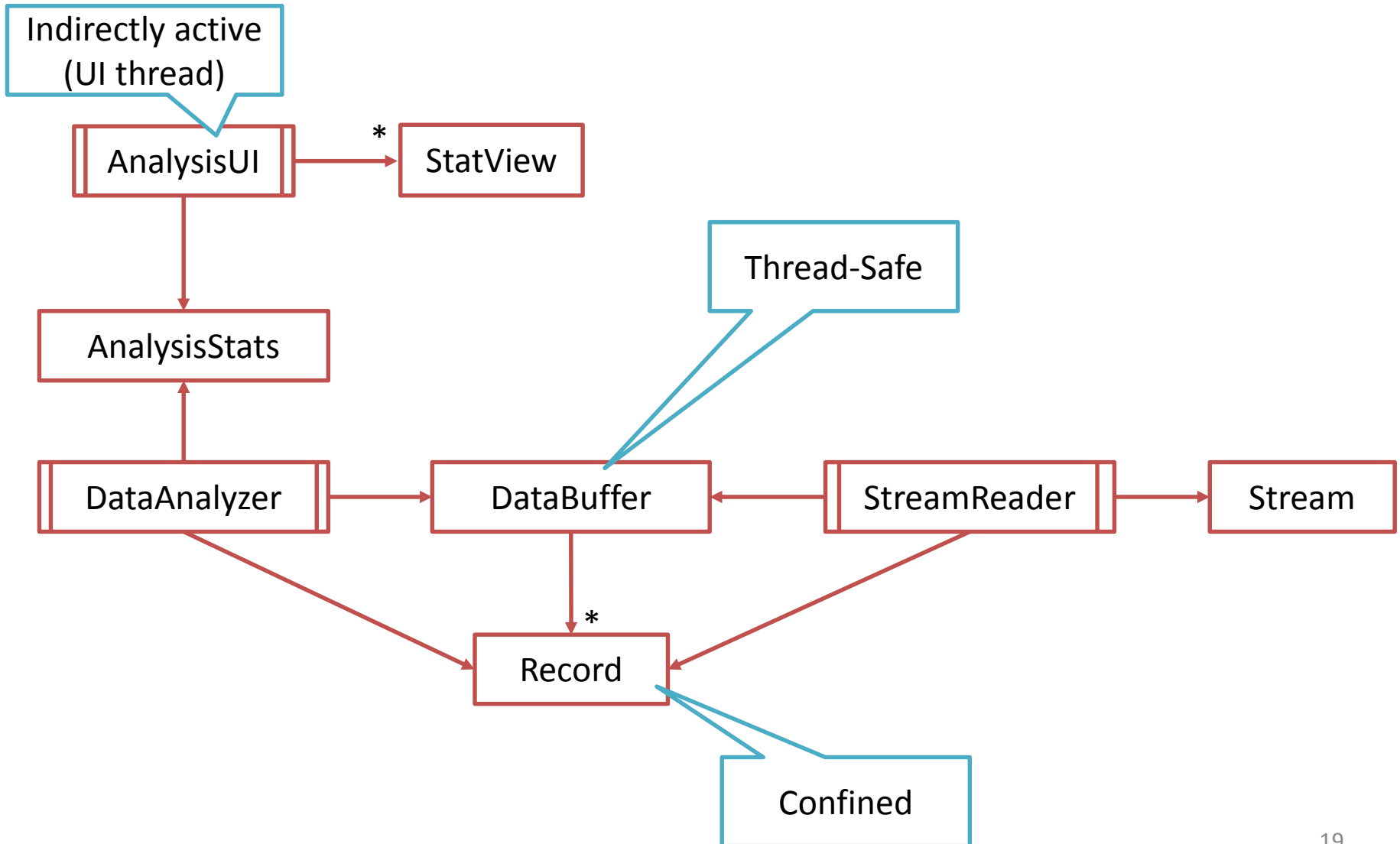
1. Identify Active Instances

- Threads, parallel tasks etc.
 - Self-defined
 - Objects running partially or fully decoupled activities
 - Externally imposed
 - Single UI thread, service worker threads, ...

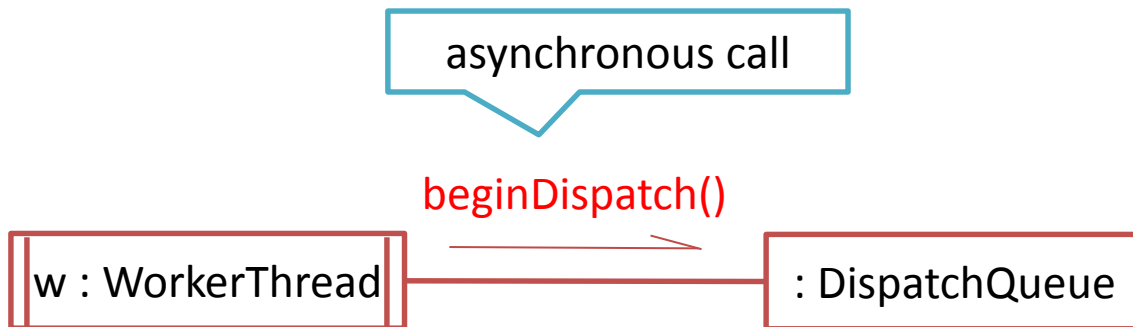
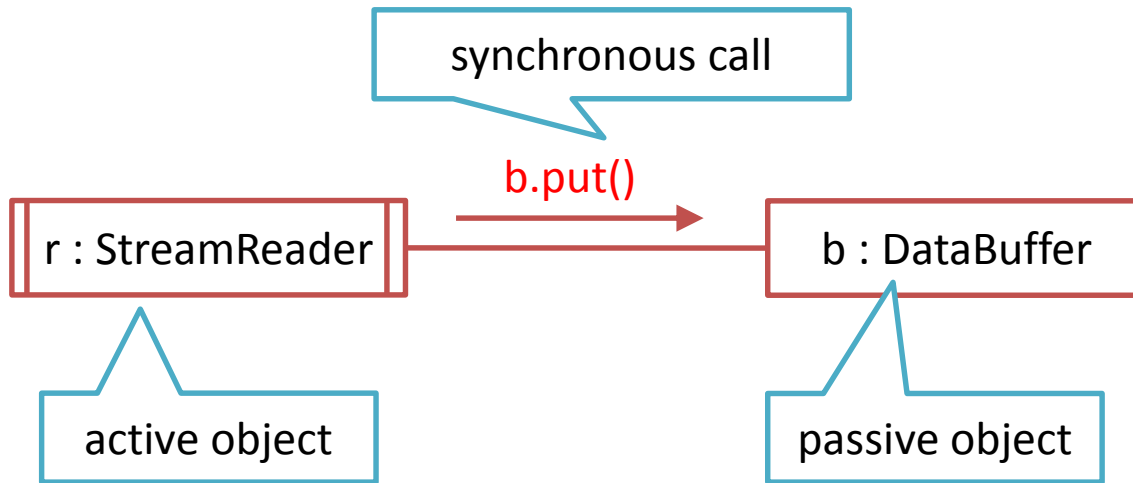


May also model conceptually active instances (running inner threads)

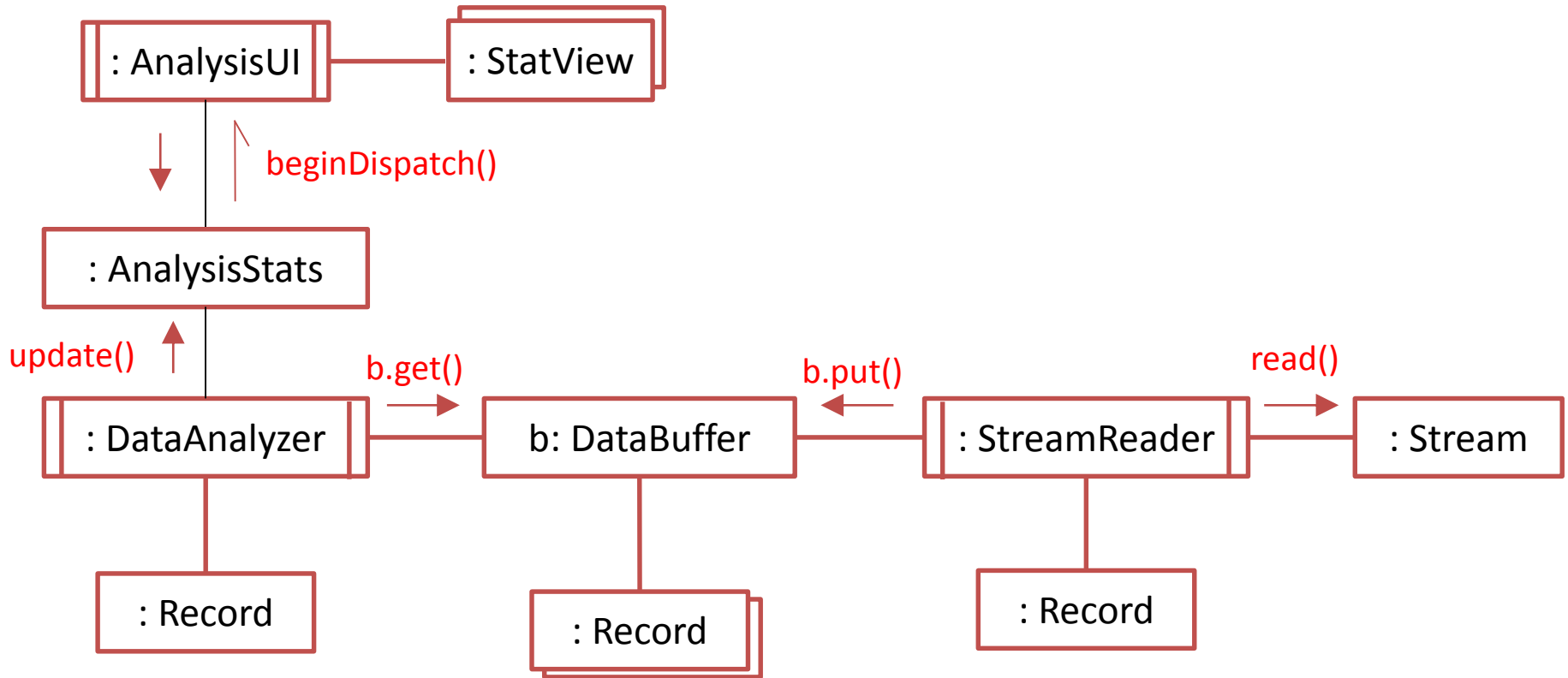
Concurrent Class Diagram



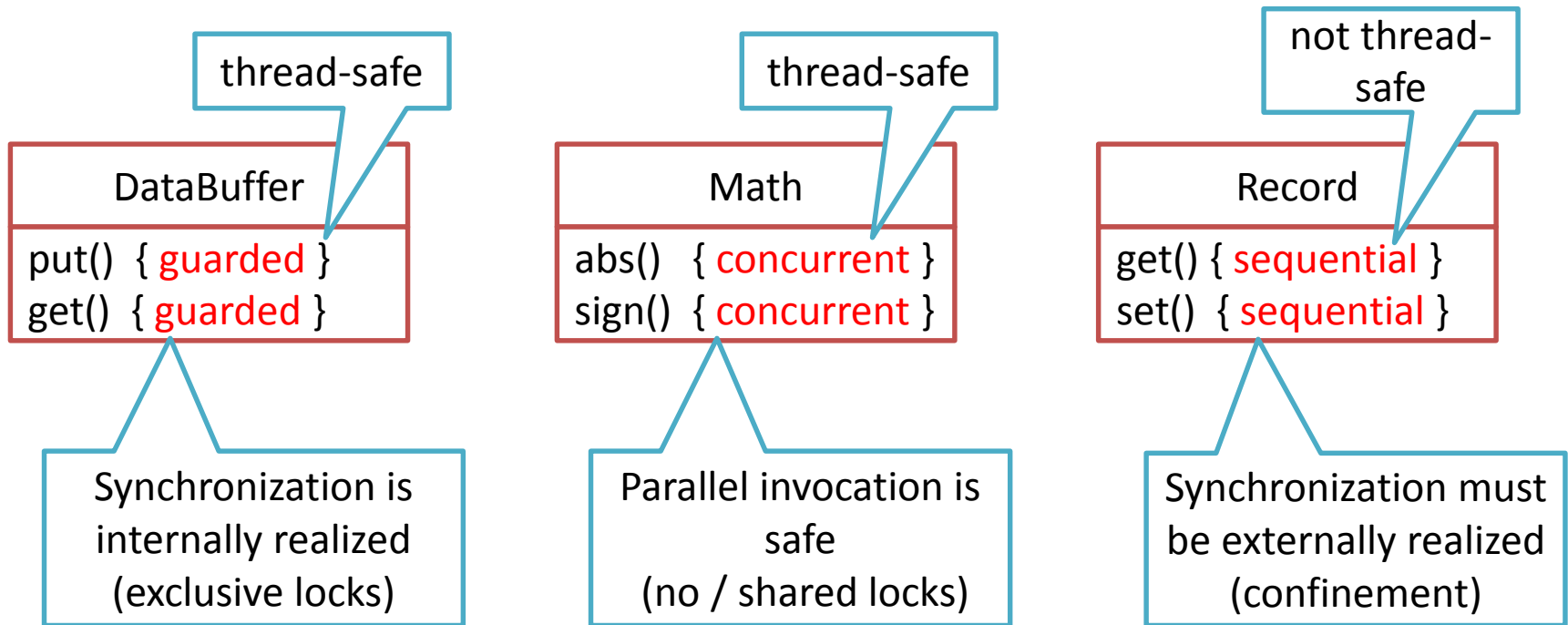
2. Specify Interactions



Concurrent Communication Diagram

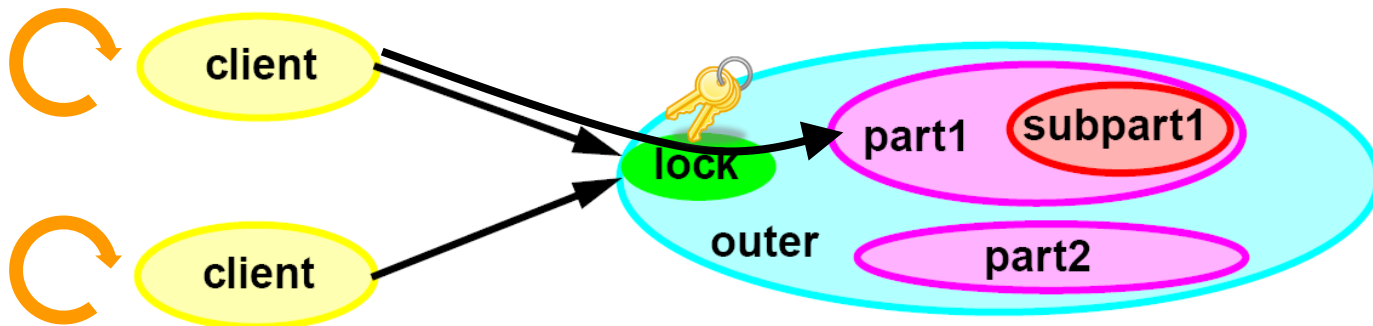


3. Define Synchronization



When No Sync is Needed

- Immutable state/objects
 - Synchronization/fence after construction needed
- Confined objects
 - Only used by 1 thread at a time
 - Local to one thread only
 - Encapsulated inside thread-safe container
 - Moved across threads (sync on moving)
 - Threads operate on disjoint parts

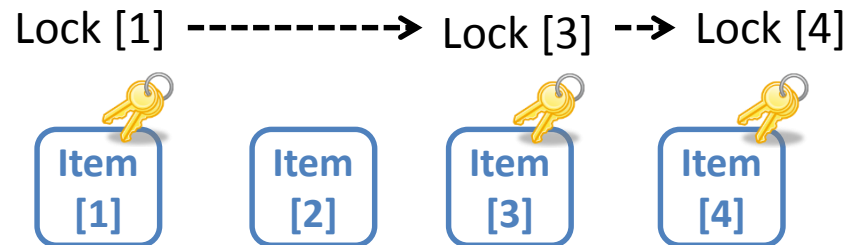


4. Reason About Correctness

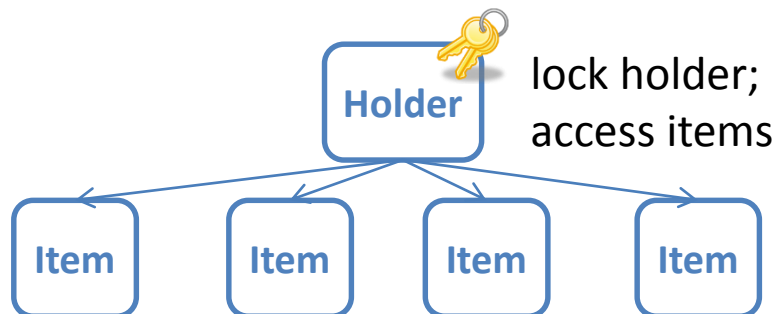
- Race condition
 - Is Synchronization or confinement defined per class?
 - Are critical sections defined and protected?
- Deadlocks
 - For nested locks: Is a linear order of locking defined?
 - No read-write lock upgrade (unless supported)?
- Starvation
 - Fair synchronization primitives (if important)?
 - No priority inversion with multiple thread priorities?

Deadlock Prevention

- Introduce linear order on resources
 - Only acquire locks on resources in ascending order

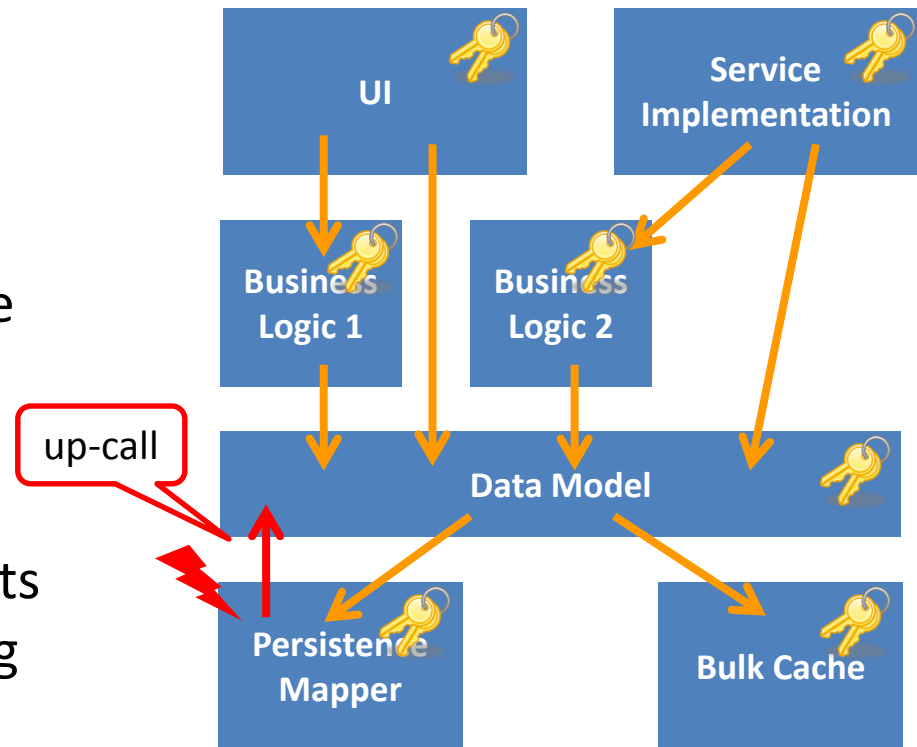


- More coarse grained lock
 - If ordering is not possible (too dynamic object structures)



Architectural Deadlock Prevention

- Hierarchy of components / modules
 - Lock per component
 - Component may comprise multiple objects
 - Inside component, only use the corresponding lock
- Call only from upper to lower modules
 - Partial order on components
 - Nested locks only according in this order
- Up-calls are deadlock-prone
 - Per events, delegates, lambdas

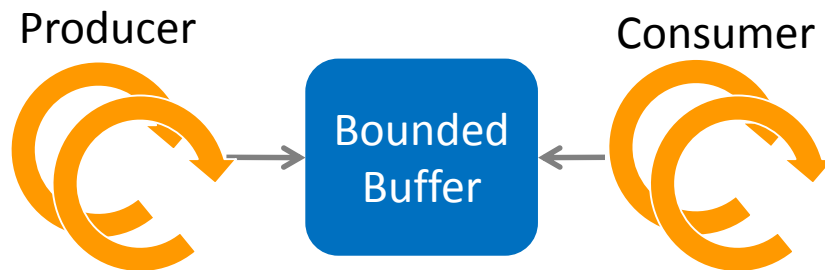


Archetypical Scenarios & Patterns

- Loosely coupled activities
 - Producer-consumer
- Responsive UI/logic
 - Asynchronous execution
- Algorithmic parallelization
 - Divide & parallel conquer

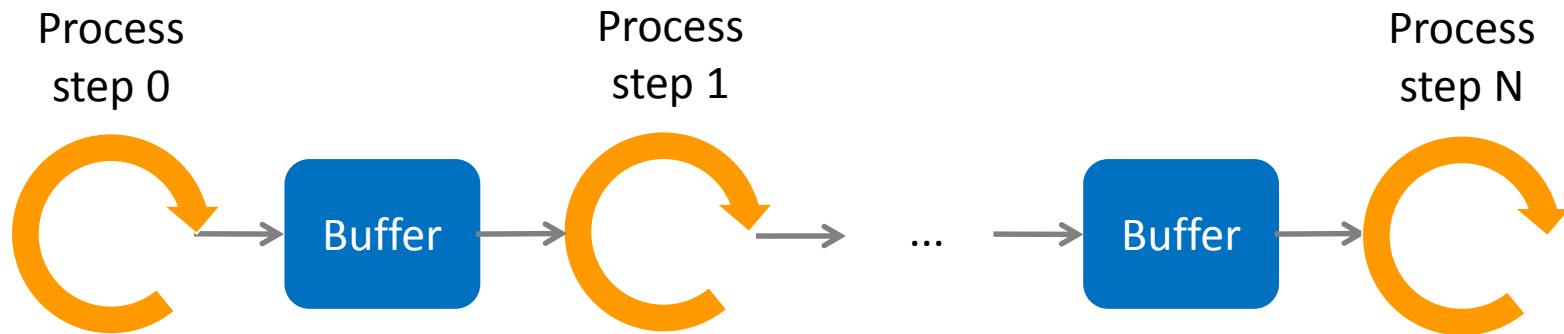
Producer-Consumer

- Faster than sequential processing
 - Partial decoupling of processing steps
 - Wait only when buffer is full or empty
 - Serving IO channels, delayed logging etc.
- Buffer must be thread-safe
 - Concurrent blocking collection, monitor sync etc.



Concurrent Pipelines

- Faster than sequential series processing



- Do not use thread pools for this
 - Unsuitable for mutual task dependencies
 - Deadlocks for fixed-sized thread pools
 - Thread injection (worker threads increase after delay)

Monitor: Know the Pitfalls!

```
class BoundedBuffer<T> {
    ...
    public void Put(T x) {
        lock(this) {
            while(queue.Count == limit) { Monitor.Wait(this); }
            queue.Enqueue(x);
            Monitor.PulseAll(this); // signal non-free
        }
    }
    public T Get() {
        lock(this) {
            while(queue.Count == 0) { Monitor.Wait(this); }
            T x = queue.Dequeue();
            Monitor.PulseAll(this); // signal non-full
            return x;
        }
    }
}
```

Monitor Pitfalls

- Recheck wait conditions
 - Between pulse and monitor reentrance, other threads may enter before and invalidate the condition
 - **while**(!condition) {
 Monitor.Wait(syncNode);
}
- PulseAll() in case of multiple wait conditions
 - E.g. non-full, non-empty
 - Monitor.Pulse**All**(syncNode)

Responsive UI/Logic

- UI is single-threaded
 - Keep responsive by outsourcing blocking/computing-intensive tasks to other threads
 - Only UI thread must access GUI controls
- E.g. C# 5 async/await model

```
public async Task<int> LongOperationAsync() { ... }
```

```
...
```

```
Task<int> task = LongOperationAsync();
```

```
// other work
```


```
int result = await task;
```

```
// continue
```



Caution: Async/Await Execution Model

- Async methods are half-synchronous/half-asynchronous
 - Caller executes methods synchronously until a blocking await occurs
 - Afterwards method runs asynchronously

```
async Task<int> GetSiteLengthAsync(string url) {  
    HttpClient client = new HttpClient();  
    Task<string> task = client.GetStringAsync(url);  
    string site1 = await task;  
    return site1.Length;  
}
```



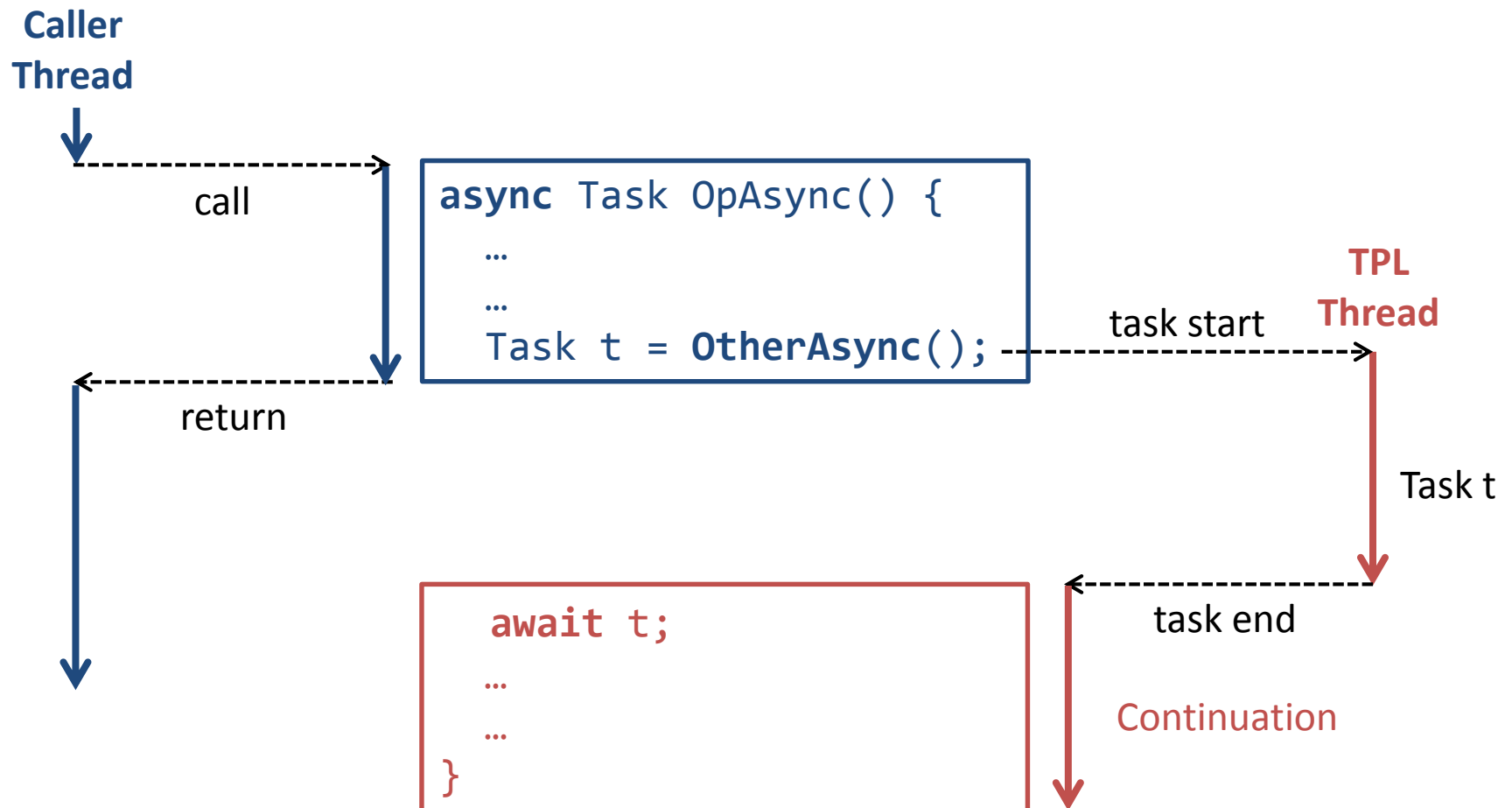
Synchronous
(caller thread)



Asynchronous
(potentially other thread)

Case 1: No Synchronization Context

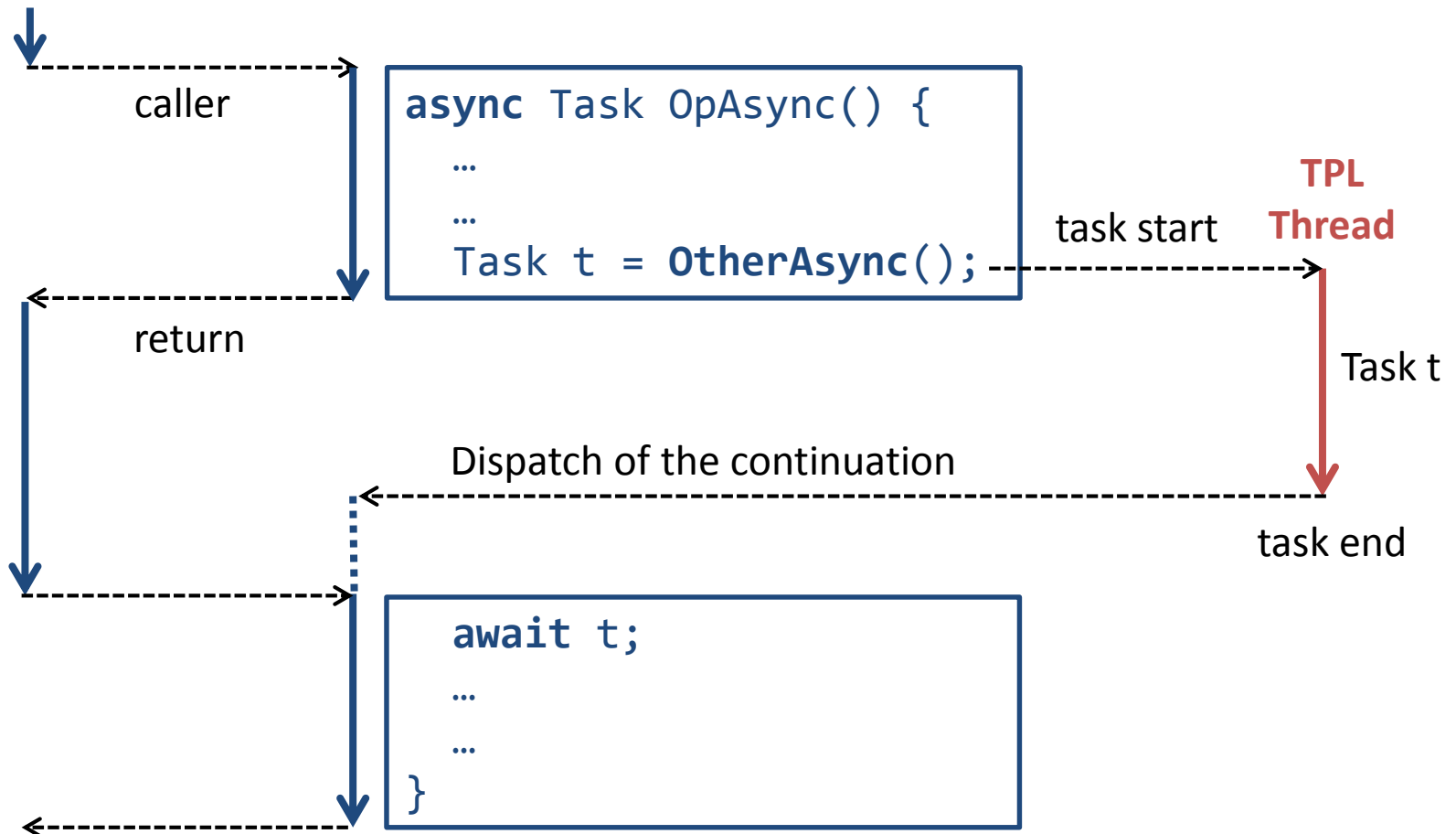
- Task thread executes part after await



Case 2: With Synchronization Context

- E.g. GUI thread as caller: dispatch

GUI Thread



Various Async/Await Pitfalls



1. Async methods are not per se asynchronous
2. Thread switch inside methods
3. Quasi-parallelism in UI event handler
4. Race conditions remain possible
5. UI deadlocks because of wrong task access
6. Exceptions are ignored for «fire-and-forget»
7. Premature termination of «fire and forget»

Divide & Parallel Conquer

- Classical parallelization for acceleration
- Thread pool is the way to go

```
void MergeSort(l, r) {  
    long m = (l + r)/2;  
    MergeSort(l, m);  
    MergeSort(m, r);  
    Merge(l, m, r);  
}
```

```
Parallel.Invoke(  
    () => MergeSort(l, m),  
    () => MergeSort(m, r)  
);
```

```
void Convert(IList<File> files) {  
    foreach (File f in files) {  
        Convert(f);  
    }  
}
```

```
Parallel.Foreach(files,  
    f => Convert(f)  
);
```

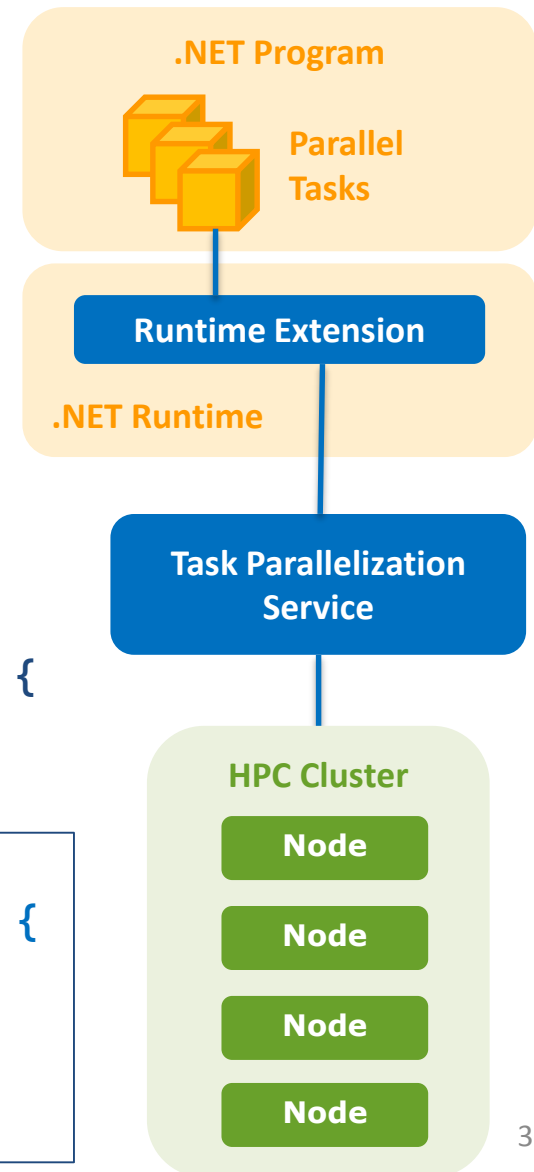
Task Parallelization in the Cloud

- Program parallel tasks in .NET
- Send to cloud for execution
- Cloud side has e.g. MS HPC cluster

<http://concurrency.ch/Projects/TaskParallelism>

```
var distribution =  
    new Distribution("tasks.concurrency.ch", ...);  
...  
distribution.ParallelFor(0, inputs.Length, (i) => {  
    outputs[i] = Factorize(inputs[i]);  
});
```

```
private long Factorize(long number) {  
    for (long k = 2; k * k <= number; k++) {  
        if (number % k == 0) { return k; }  
    }  
    return number;  
}
```



Conclusions

- Concurrency becomes increasingly important
 - Programmers need to strengthen their skills
- Danger of non-deterministic errors
 - Clear concurrency design for SW architecture is vital
- Various pitfalls lurk in maintaining technologies
 - Awareness is required as long as tools do not help here

Thanks for Your Attention!

- .NET Concurrency Courses
 - <http://concurrency.ch/Training>
- Consulting & Reviews
- Engineering & Research Projects
- Contact

Prof. Dr. Luc Bläser
HSR Hochschule für Technik Rapperswil
IFS Institut für Software
lblaeser@hsr.ch
<http://concurrency.ch>, <http://ifs.hsr.ch>

Institut für Software (IFS)

Partners



Academic collaborators

