Course 02158

Concurrency Paradigms

Hans Henrik Løvengreen

DTU Compute

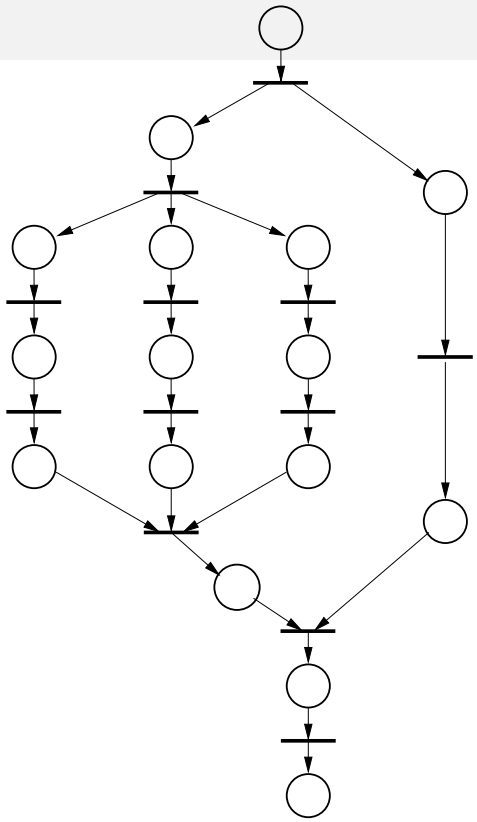
Concurrency Paradigms

- An overall model for application of concurrency
- Goals: Structure, scalability, recognizability

Approaches

- Classic: Fixed, dedicated threads interacting through shared components
- Message based: Actors, tuple-spaces
- Task-based: Asynchronous tasks submitted to thread pool
- Fork/join pattern
- Asynchronous programming (.NET async)
- Reactive programming
- Implicit data-parallel model (Java Streams)

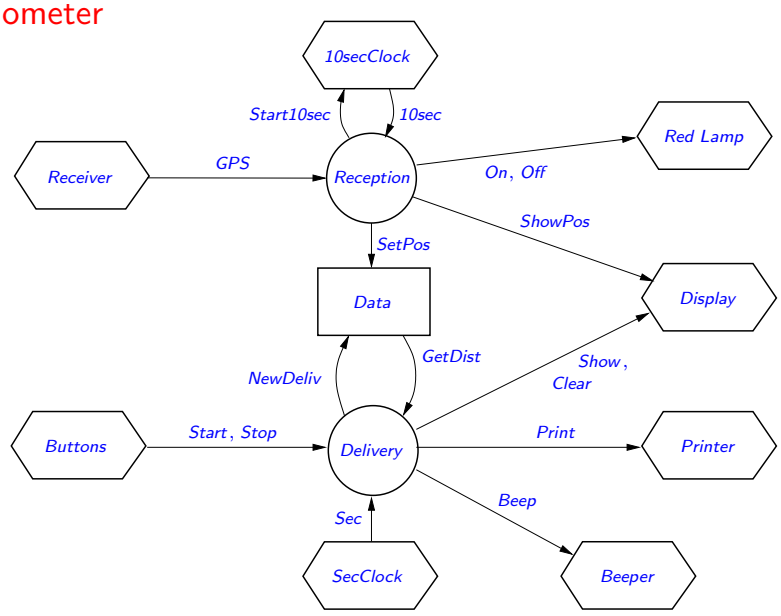
Underlying Petri Net



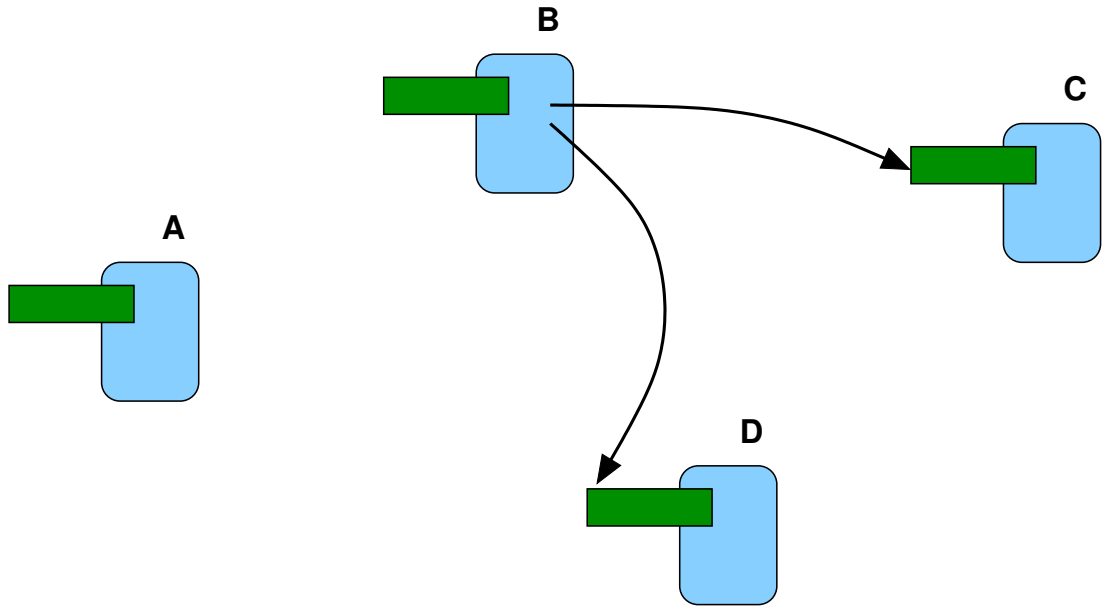
Classic Approach

- Fixed number of threads dedicated to particular tasks

Example: Cyclometer



Actor Model



The Scala Language

- Multi-paradigm langauge developed at EPFL since 2001
- Lead by *Martin Odersky* (Java compiler, Java generics)
- Predecessors: Pizza, Funnel
- Open source, active community
- Famous industrial application: Twitter distribution engine.

Aims

- Expressive, expandable langaguage (contrast to. eg. Java, C#)
- Consise, flexible syntax
- Production quality (static typing, interoperability, JVM, .NET?)
- Simple support for concurrency

Recent adoption

- Basis for the chisel hardware description languge (for FPGA programming)

Scala Example: Resource Allocator

```
class TypedAllocator[T](pool : Seq[T]) extends Actor{
  trait AllocatorReq
    case class Acquire() extends AllocatorReq
    case class Release(r : T) extends AllocatorReq
  val free =new ListBuffer[T]
  def act() = {
    free ++= pool
    loop {
      receive {
        case Acquire() if !free.isEmpty => reply(free.remove(0))
        case Release(r) => free+ = r
      }
    }
  }
}
```

Scala Example: Resource Allocator — usage

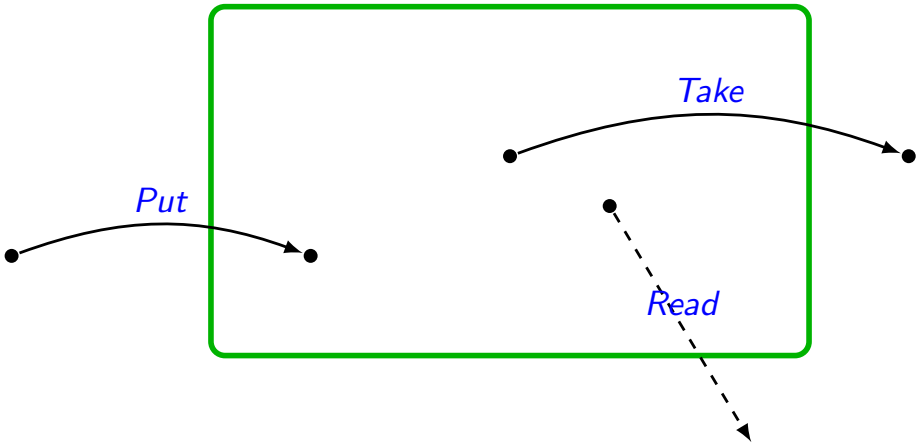
- Interface wrappers:

```
def acquire() : T = (this !? Acquire()).asInstanceOf[T]
def release(r : T) = this ! Release(r)
```

-
```
class Client(manager : TypedAllocator[Res]) extends Actor{
  def act() = {
    valres = manager.acquire()
    res.use()
    manager.release(res)
  }
}
```

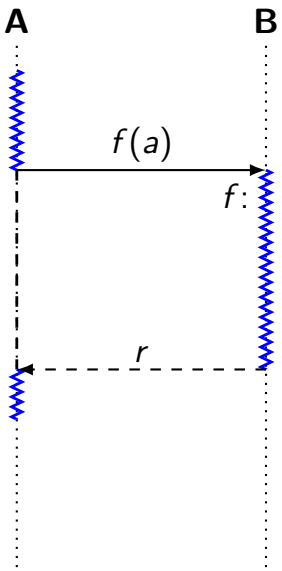
-
```
val manager = newallocator.TypedAllocator[Res](resseq)
manager.start
for (i <− 1 to 5) {new Client(manager).start}
```

The Tuple Space Model

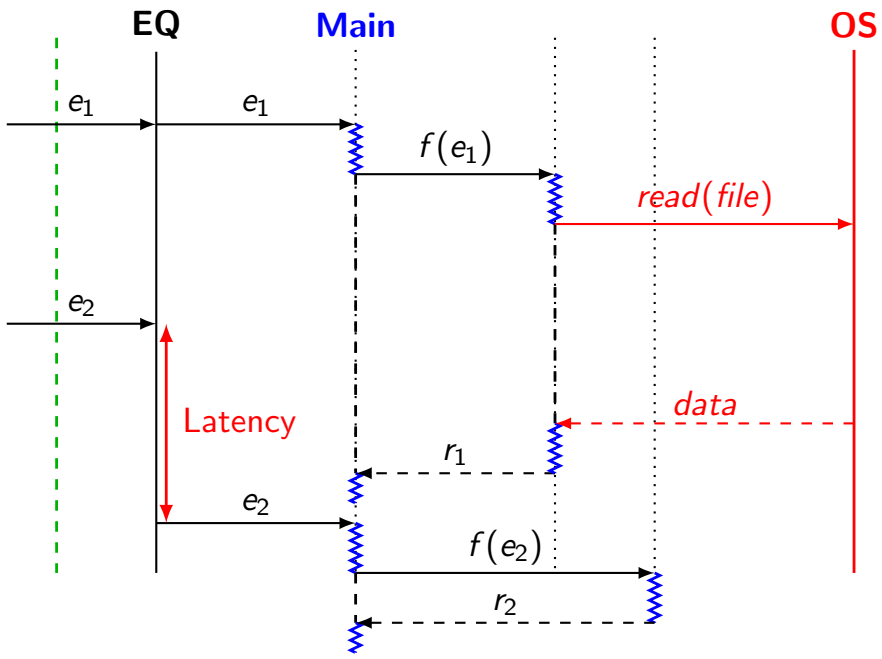


- Most known tuple space implementation: Linda

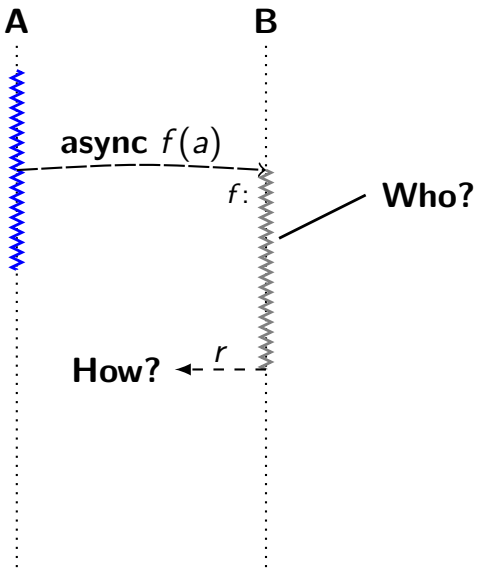
Synchronous execution



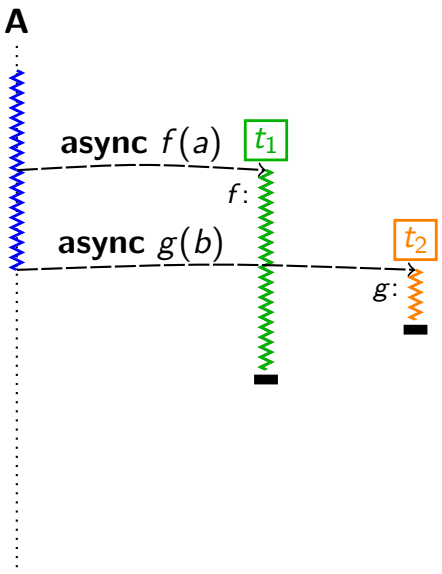
Event-driven Systems



Asynchronous call



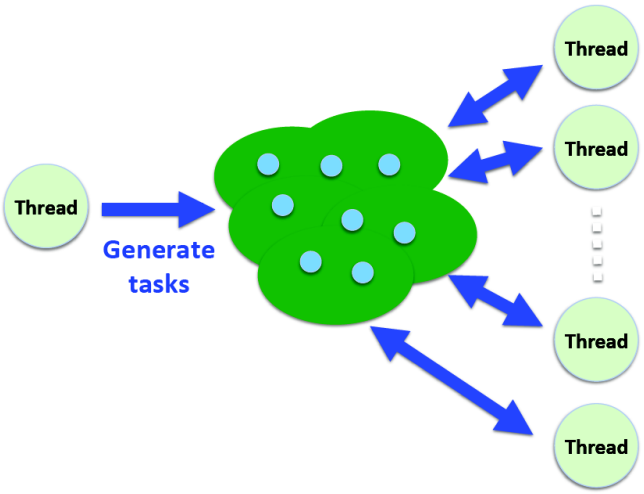
Multi-threading



- **Idea:** Create a new thread per asynchronous call
- Works ok for smaller number of threads, but does not scale well

Task-based Approach

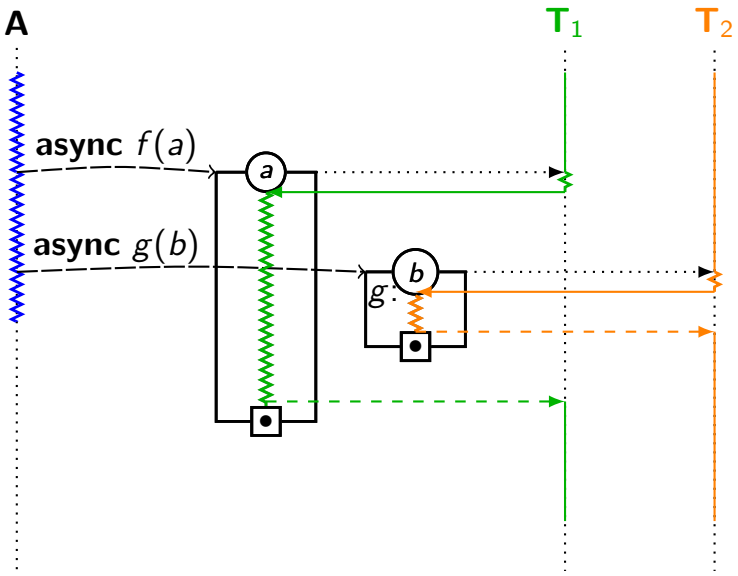
- Task: Well-defined, terminating, (non-trivial) sub-computation
- Executed by a *pool of threads*:



- Many names: *Bag-of-tasks*, *supervisor-worker*, ...

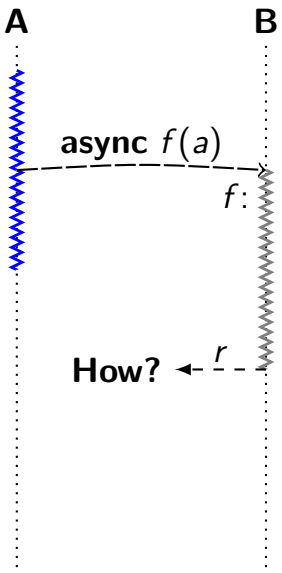
Graphics by Ruud van der Pas

Task Execution

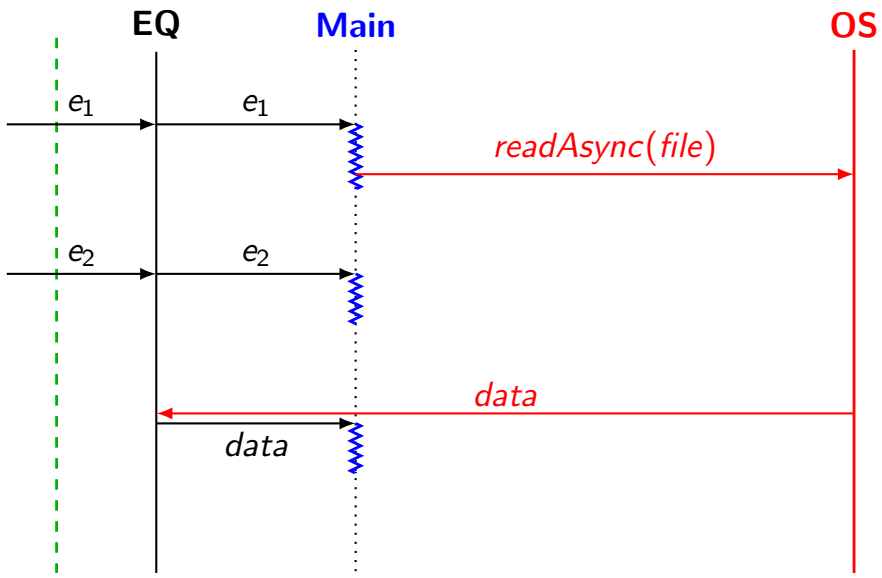


- A task is represented by a *closure* — a function with an environment.

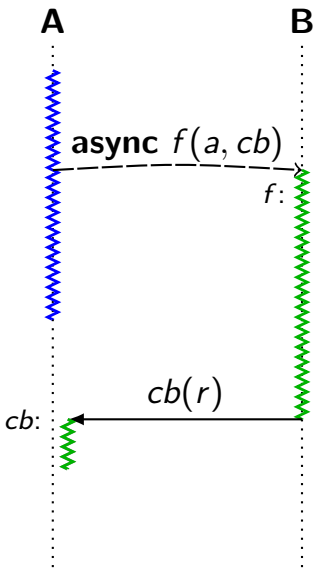
Asynchronous call — result handling



Fixed return path — asynchronous I/O

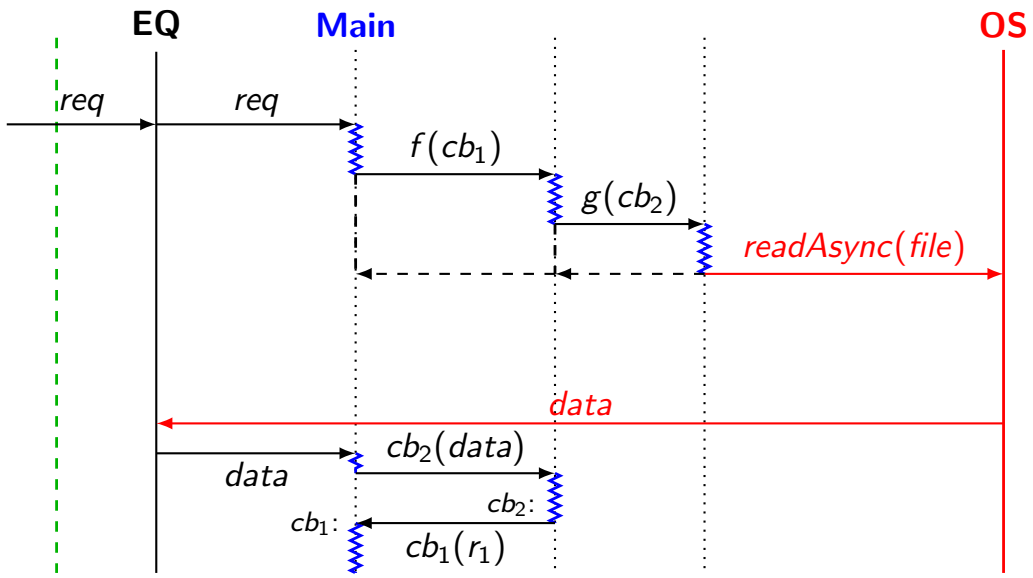


Call-backs

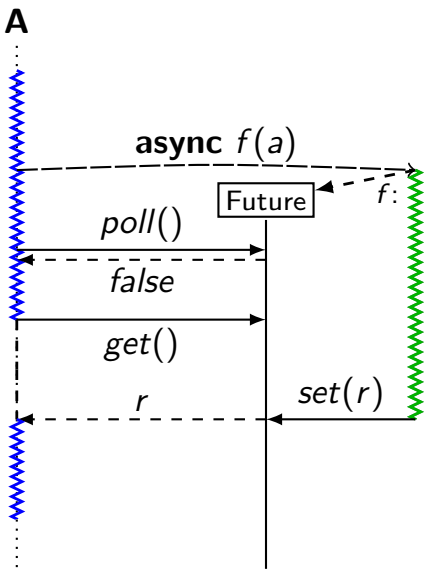


- Concurrency issues remain

Call-backs in Node.js

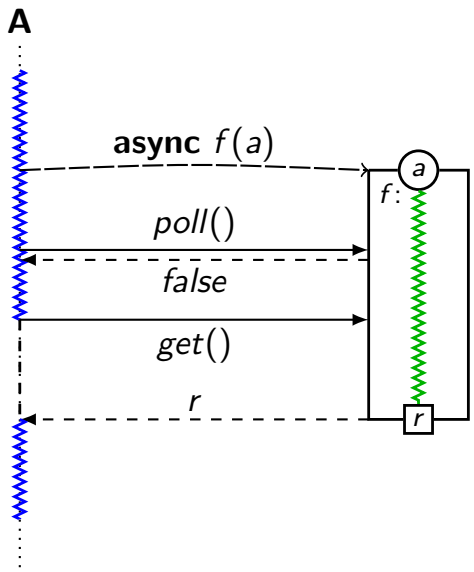


Futures



- A *future* represents a value to be
- May be *polled* or *awaited* (get, join, await, . . .)

Tasks as Futures



- Tasks often provide a *future interface*

Thread Pool Caveats

Sizing

- Number of threads should match number of processors
- Tasks should not bee too large
- Tasks should not be too small
- Rule of thumb: 100-10000 basic operations

Scheduling

- No particular number of threads should be asssumed
- No particular ordering of task should be assumed
- Any two submitted task should be considered concurrent

Synchronization

- Tasks should not use blocking system calls (synchronous I/O)
- Task may use **synchronized** for small critical sections
- Tasks should not do conditional synchronization (**wait**)

General Fork/Join Pattern

- *SolvePar*(x)
  if *simple*(x) **then**
    return *solve*(x)
  else
    $(x1, x2)$ = *split*(x)
    parallel
      $r1$ = *SolvePar*$(x1)$
      $r2$ = *SolvePar*$(x2)$
    return *reduce*$(r1, r2)$

Recursive Fork/Join using ThreadPool

- Let `pool` be a given thread pool
- ```
  V SolvePar(T x) {
    if simple(x) {
      return solve(x);
    } else {
      Future<V> r1 = pool.submit(x.left());
      Future<V> r2 = pool.submit(x.righ());
      v1 = r1.get();
      v2 = r2.get();
      return reduce(v1, v2);
    }
  }
  ```
- Will require a thread for each task! Why?
  ```

Fork/Join Pool

- `ThreadPoolExecutor` cannot handle task *dependencies* well
- Special `ForkJoinPool` allows threads to work while waiting for results
- Queues tasks with *thread affinity* and *work-stealing*

ForkJoinTask

- Specialized tasks with operations `fork()` and `join()`
- `fork()` causes task to be submitted to queue for current thread
- `join()` allows thread to execute other tasks from queue while waiting
- Subclass `RecursiveTask` defines computation in `compute()`

ForkJoinPool

- Default instance: `ForkJoinPool.commonPool()`

Example: Fibonacci

```
class Fibonacci extends RecursiveTask<Integer> {
  final int n;
  Fibonacci(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1)
      return n;
    Fibonacci f1 = new Fibonacci(n - 1);
    f1.fork();
    Fibonacci f2 = new Fibonacci(n - 2);
    return f2.compute() + f1.join();
}
```

```
Fibonacci fibtask = new Fibonacci(50);
int fibOf50 = ForkJoinPool.commonPool().invoke(fibtask);
```

Example: Array summation I

```
public class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;

    int low, high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr; low = lo; high = hi;
    }

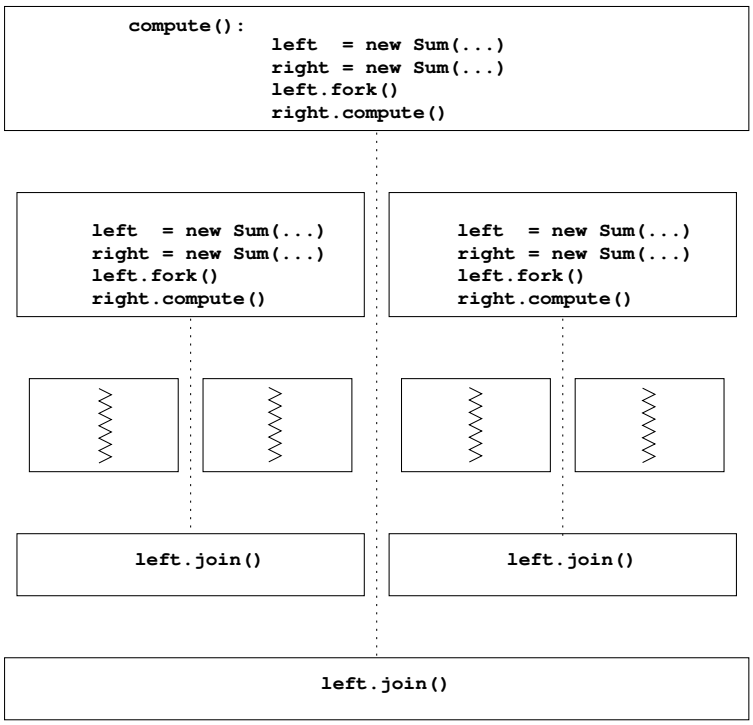
    protected Long compute() {...}

    static long sumArray(int[] array) {
        RecursiveTask task = new Sum(array,0,array.length);
        return ForkJoinPool.commonPool().invoke(task);
    }
}
```

Example: Array summation II

```
    protected Long compute() {
        if (high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns = left.join();
            return leftAns + rightAns;
        }
    }
```

Example: Array summation III



Task Parallel Library in .NET

TPL

- Flexible system-provided thread pool(s)
- Tasks and and futures are built together
- Divided into different *synchronization contexts* (= thread pools)
- The GUI event queue is considered a special synchronization context.
  - Only one thread
  - Tasks are executed in FIFO order
- Tasks may be combined, e.g. by *continuation tasks*
- Used by the **async** syntactic framework in C#

Async Example I

- Explicit post to GUI-framework
- ```
  private void ButtonClick(object sender, RoutedEventArgs e)
  {
      SynchronizationContext ctx = SynchronizationContext.Current;

      Task.Factory.StartNew( () =>
      {
          decimal result = CalculateMeaningOfLife();

          ctx.Post(state => resultLabel.Text=result.ToString(), null);
      });
  }
  ```

Async Example II

- Using a continuation task
- ```
  private void ButtonClick(object sender, RoutedEventArgs e)
  {
      Task<decimal> calc = Task.Factory.StartNew<decimal>(
                                () => CalculateMeaningOfLife() );


      calc.ContinueWith(
            t => resultLabel.Text = t.Result.ToString(),
            TaskScheduler.FromCurrentSynchronizationContext() );
  }
  ```

Async Example III

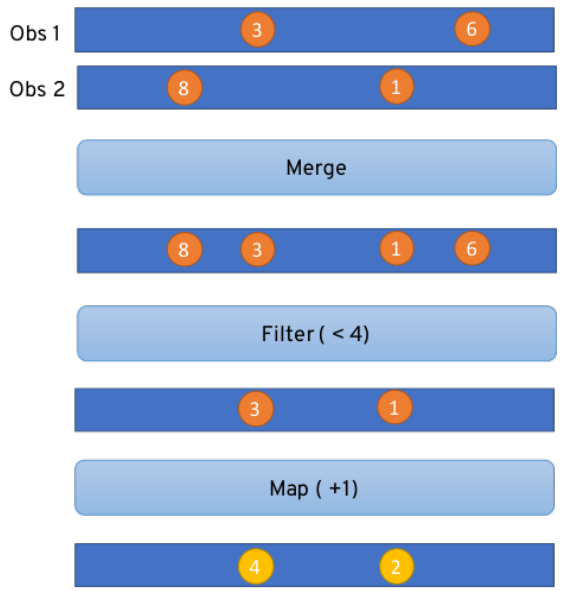
- Using an asynchronous task
-
```
private Task<decimal> CalculateMeaningOfLifeAsync() {
    return Task.Factory.StartNew<decimal>(
                                () => CalculateMeaningOfLife() );
}

private async Task ButtonClick(object sender, RoutedEventArgs e)
{
    decimal result = await CalculateMeaningOfLifeAsync();

    resultLabel.Text = t.Result.ToString();
}
```

Reactive Programming

- Marble diagram




```
Observable<Integer> obs1
 = Observable.from(new Integer[] {3, 6});
Observable<Integer> observable2
 = Observable.from(new Integer[] {8, 1});
Observable.merge(obs1, obs2)
    .filter(i -> i < 4)
    .map( i -> i+1)
    .subscribe(System.out::print);
```

Java Streams

- Introduced in Java 1.8
- Applies ideas from *functional programming* and *reactive programming*
- General way of handling large amounts of *data* or indefinite streams of *events*
- Everything is considered a *stream* of values
- Streams may be *composed* in various ways
- May use *higher order functions*
- May work with *infinite streams*

Parallel Streams

- A stream may be parallelized by applying method `parallel()`
- A parallel stream is automatically computed by tasks on the fork/join pool
- Subtle concurrency issues may persist!

Example: Array summation using streams

- Given an array of integers: `int [] a;`
- Find sum of all even element squares
- ```
  IntStream src = a.stream();
  IntStream even = src.filter((i) -> i % 2 ==0 );
  IntStream squares = even.map( (i) -> i*i );
  int sum = squares.sum();
  ```
  ```

Example: Array summation using streams

- Given an array of integers: `int [] a;`
- Find sum of all even element squares
- ```
  int sum = a.stream()
      .filter((i) -> i % 2 ==0 )
      .map( (i) -> i*i )
      .sum();
  ```

Example: Array summation using parallel streams

- Given an array of integers: `int [] a;`
- Find sum of all even element squares
- ```
  int sum = a.stream().parallel()
      .filter((i) -> i % 2 ==0 )
      .map( (i) -> i*i )
      .sum();
  ```

Task-based programming summary

- Task-based programming provides *fine-grained concurrency*
- Separates logic from execution
- Enables *scalable* programs
- Concurrency issues remain — but may be less identifiable!
- Appears in many languages
- May be disguised as *async/await* syntax
- Thread skills may still be required
- Some new langauges have concealed underlying threads:
  - Go: Only lightweight *goroutines*
  - Kotlin: Only lightweight *coroutines*
  - Java: New lightweight *virtual threads* (Java 19, Aug 2022) [Loom]
- Might eventually be replaced by system-controlled parallelization