Course 02158

Concurrency in Java

Hans Henrik Løvengreen

DTU Compute

Java Concurrency Means

Explicit threads

- Thread creation and termination
- Thread synchronization
- New in Java 19: *Virtual threads*

Task-based

- Thread pools: Tasks and futures
- Fork/join pools

Implicit concurrency

- Java Streams
- Reactive programming

Java Thread Creation — basic form

- ```
  class P implements Runnable {
    void run() {
        ⋮
    }
  }
  ```
- In `main()`:

  ```
  P p = new P();
  Thread t = new Thread(p);
  t.start();
  ```
- `other work`
- `t.join();`

Java Thread Creation — with parameter

- ```
  class P implements Runnable {
    int n;
    P(int n) {this.n = n; }
    void run() {
        ⋮  (using n)
    }
  }
  ```
- In `main()`:

  ```
  P p = new P(117);
  Thread t = new Thread(p);
  t.start();
  ```
- `other work`
- `t.join();`

Java Thread Creation — extending Thread

- ```
  class P extends Thread{
    int n;
    P(int n) {this.n = n; }
    void run() {
       ⋮  (using n)
    }
  }
  ```
- In `main()`:

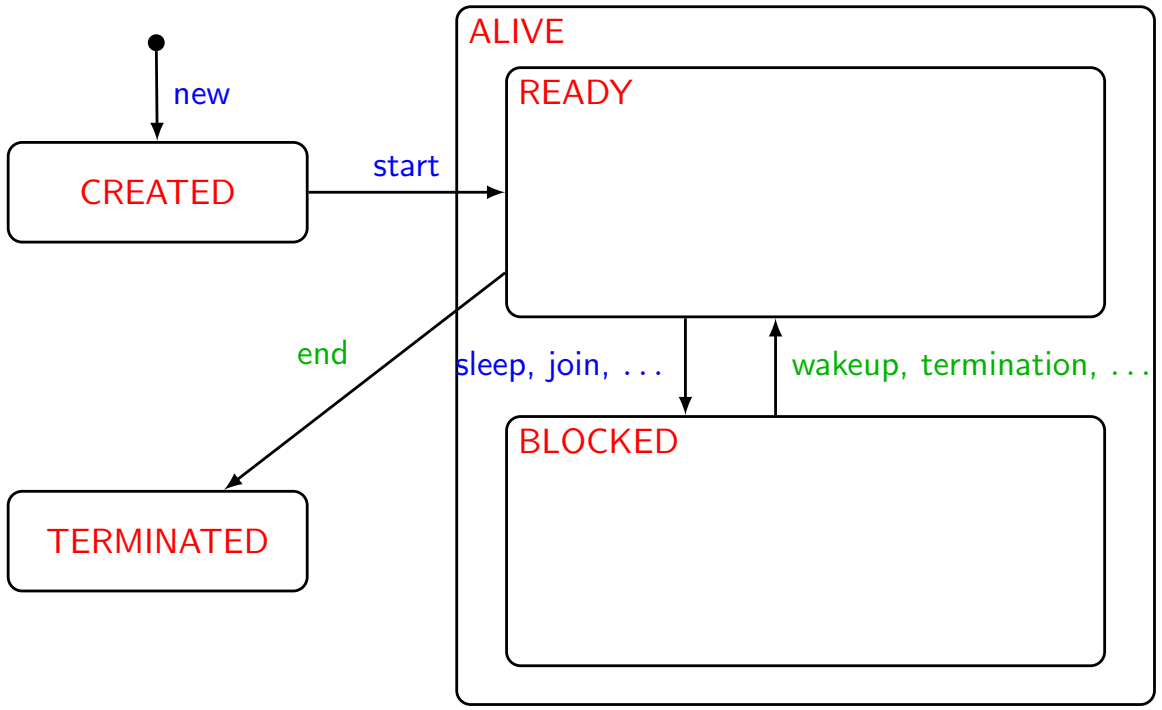
  ```
  Thread t = new P(117);
  t.start();
  ```
- other work
- `t.join();`

Java Thread Creation — with λ

- In `main()`:

  ```
  Thread t = new Thread( () -> { ... });
  t.start();
  ```
- other work
- `t.join();`
  ```

Java Thread States



Java Thread Cancellation

Bad

- The `t.stop()` method throws a `ThreadDeadth` exception immediately
- Unsafe — deprecated

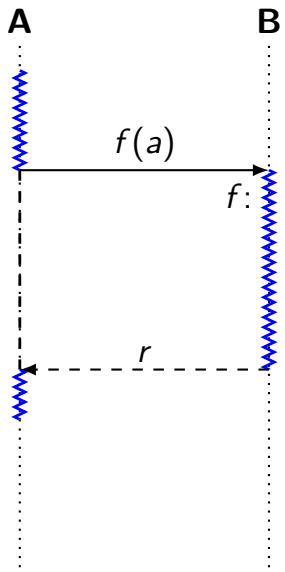
Better

- Let thread terminate itself (including release of resources)
- A flag may be used to indicate desired cancellation
- Not observed during blocking operations (e.g. `sleep()`)

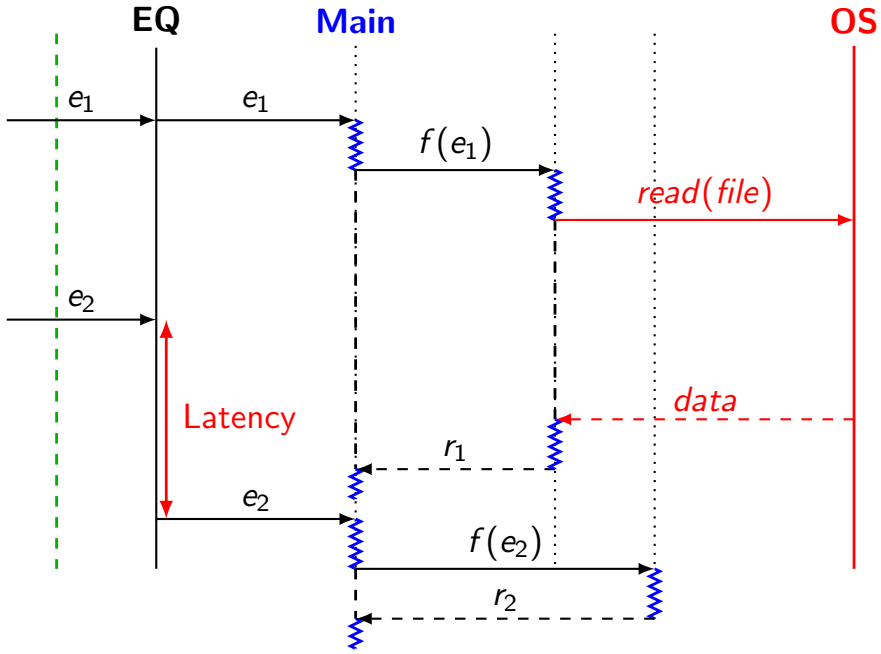
Best

- Java provides a *system supported cancellation flag* per thread (interrupt)
- `t.interrupt()` sets the flag
- Any *blocking operation* will throw an `InterruptedException` when met
- Flag may be interrogated by `t.interrupted()` (clears flag)

Synchronous execution



Event-driven Systems



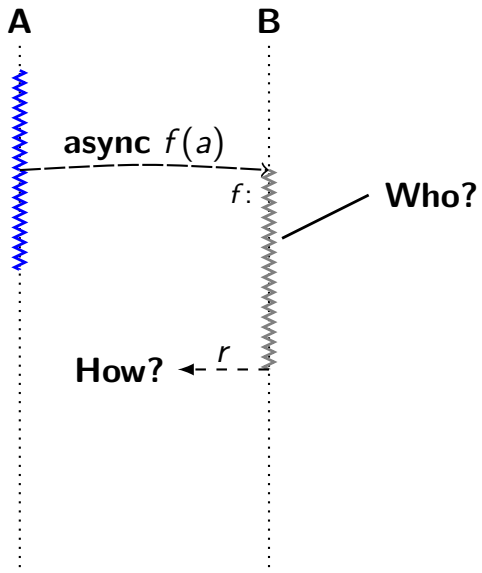
Example: Web Server — Sequential

```
public class SingleThreadWebServer {

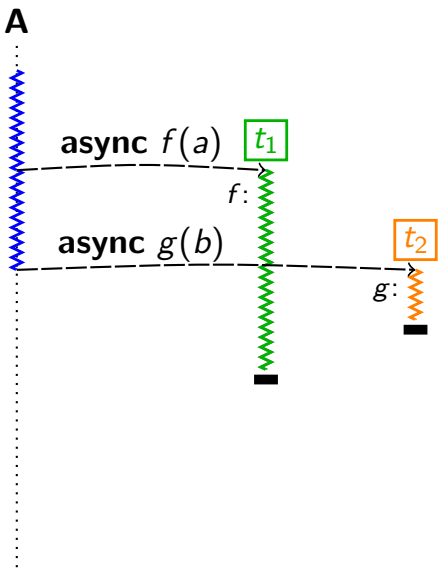
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }

    private static void handleRequest(Socket connection) {
        // request-handling here
    }
}
```

Asynchronous call



Multi-threading



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

Example: Web Server — Multi-threaded

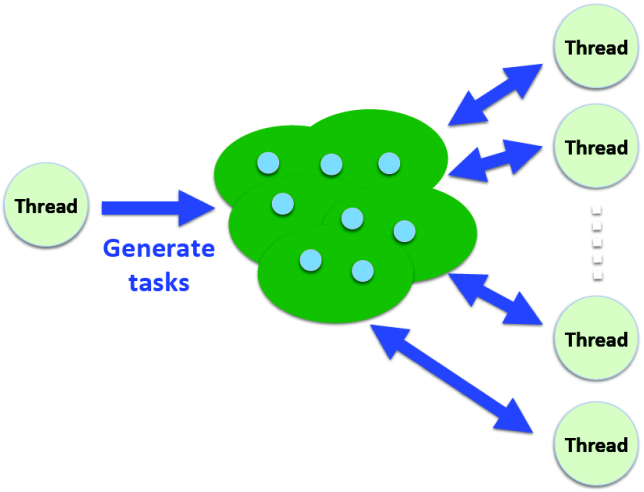
```
public class ThreadPerTaskWebServer {

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }

    private static void handleRequest(Socket connection) { ... }
}
```

Task-based Approach

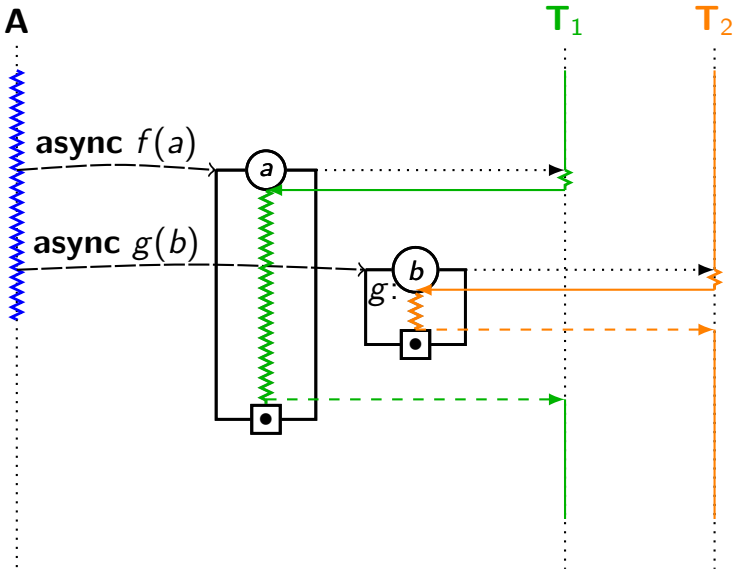
- Task: Well-defined, terminating, (non-trivial) sub-computation
- Executed by a *pool of threads*:



- Many names: *Bag-of-tasks*, *supervisor-worker*, ...
- Many frameworks: OpenMP, Java fork/join , .NET TPL, ...

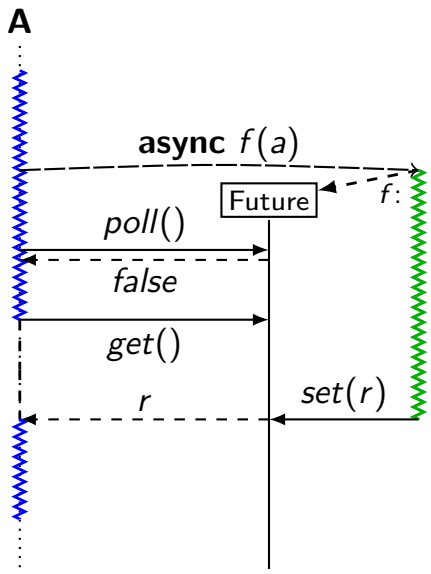
Graphics by Ruud van der Pas

Task Execution



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

Futures



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

Java Execution control

- Machinery (framework) to deal with common usages of threads

Notions

- A *task* is a (finite) piece of work represented by an `Runnable` object
- Tasks can be *submitted* to *executors*
- An *executor service* also allows for monitoring of submitted tasks
- A `Callable` object is a task that will compute a result
- A `Future` object represents the status of (cancelable) task
- A *thread pool* is an executor with:
  - A set of threads between a minimum and maximum number
  - A keep-alive time for idle threads
  - A *task queue*: None, bounded, unbounded
- If queue full, tasks may be *rejected* in different ways

Example: Web Server — using an Executor

```
public class TaskExecutionWebServer {
    private static final Executor exec = new SomeExector();

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = () -> handleRequest(connection);
            exec.execute(task);
        }
    }

    private static void handleRequest(Socket connection) { ... }
}
```

Some Simple Executor Implementations

- ```
  public class WithinThreadExecutor implements Executor {
      public void execute(Runnable r) {
          r.run();
      };
  }
  ```

- ```
  public class ThreadPerTaskExecutor implements Executor {
      public void execute(Runnable r) {
          new Thread(r).start();
      };
  }
  ```

Executor Service

- An `Executor` with more control of tasks
- Accepts `Callable<T>` tasks which computes results of type `T`
- A task submission generates a `Future<T>` to hold result
- The result of the future is obtained with the `get()` method
- A task may be *cancelled* through its future.

Termination Control

- An executor service may be *shut down*
- If shut down is *immediate*, pending tasks are returned
- Termination may be *awaited*

Thread Pools

- An `ExecutorService` that manages a pool of *worker threads*
- Many managing strategies.

Standard Types

- `newFixedThreadPool(`*size*`)` Keeps up to *size* threads ready
- `newCachedThreadPool()` Keeps threads alive for 60 sec
- `newSingleThreadExecutor()` Uses a single thread only
- `newScheduledThreadPool()` Allows tasks to be executed later

ThreadPoolExecutor

- Generic implementation which may be tailored to need
- Task queue parameters: *max size*, *rejection*
- Thread pool parameters: *max/min size*, *keep-alive-times*

Example: Web Server — using a Thread Pool

```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec =
                            Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = () -> handleRequest(connection);
            exec.submit(task);
        }
    }

    private static void handleRequest(Socket connection) { ... }
}
```

Example: Work division I

```
class SumTask implements Callable<Integer> {
    int low, high;
    int[] array;

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

    Integer call() {
        int sum = 0;
        for (int i = lo; i < hi; i++ ) { sum += arr[i]; }
        return sum;
    }
}
```

Example: Work division II

```
int sum(int [] arr) {
    Executor exec = Executors.newFixedThreadPool(N);
    final int chunk = arr.length/N; /* Assume multiplum */
    List<SumTask> tasks = new ArrayList<SumTask>();

    for (int i=0; i < N; i++) {
      tasks.add(new SumTask(arr, chunk*i, chunk*(i+1));
    }

    List<Future<Integer>> results = exec.invokeAll(tasks);

    int sum = 0;
    for (Future<Integer> res: results) { sum += res.get(); }

    exec. shutdown();
    return sum;
}
```

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 (sleep, synchronous I/O)
- Tasks should not do conditional synchronization
- Tasks may use mutual exclusion for small critical sections