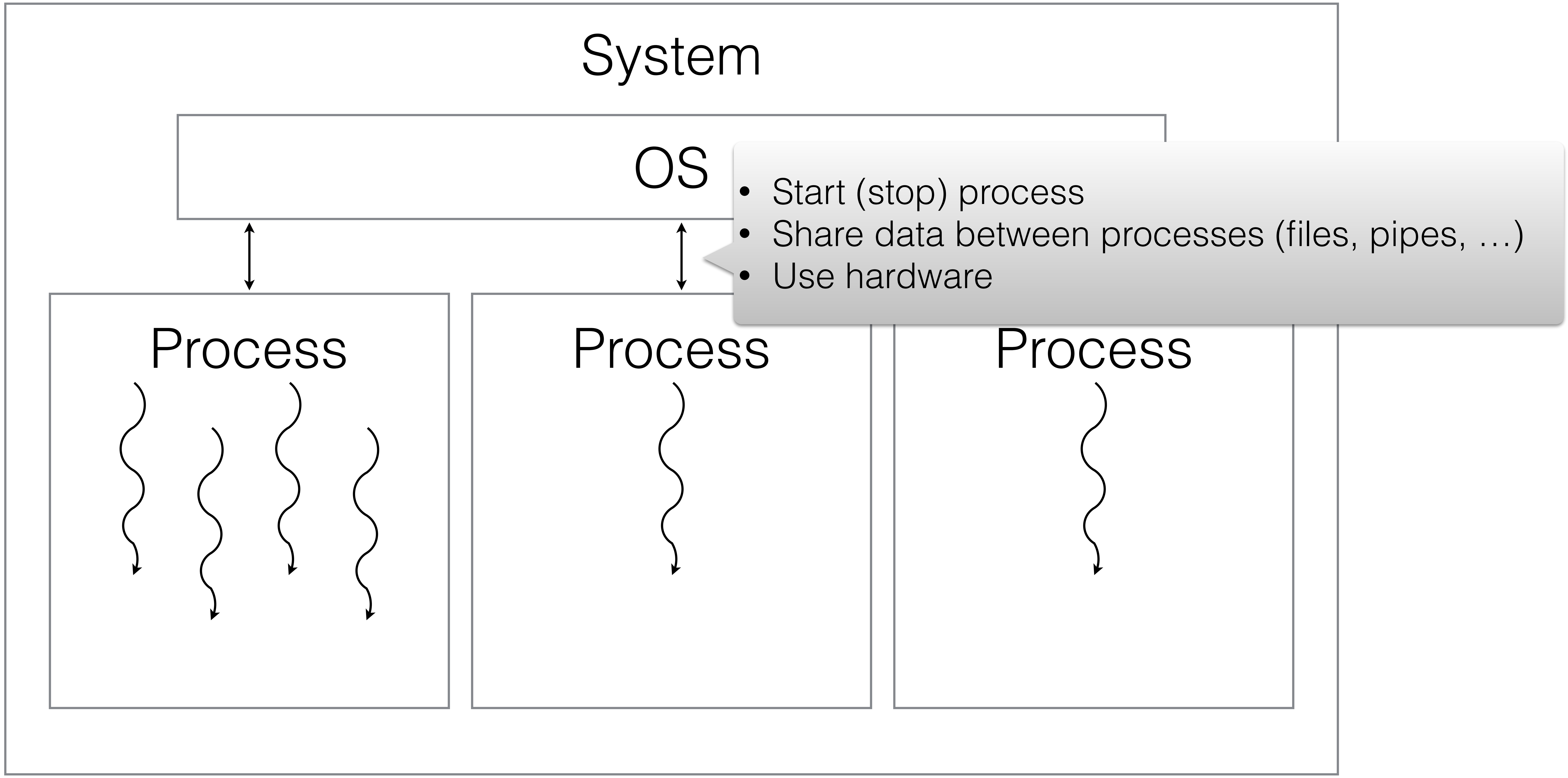
Multi-threading

Why?

- Concurrency / threading problems are hard to debug
- Might only occur in the field (not during testing)

Process vs Thread

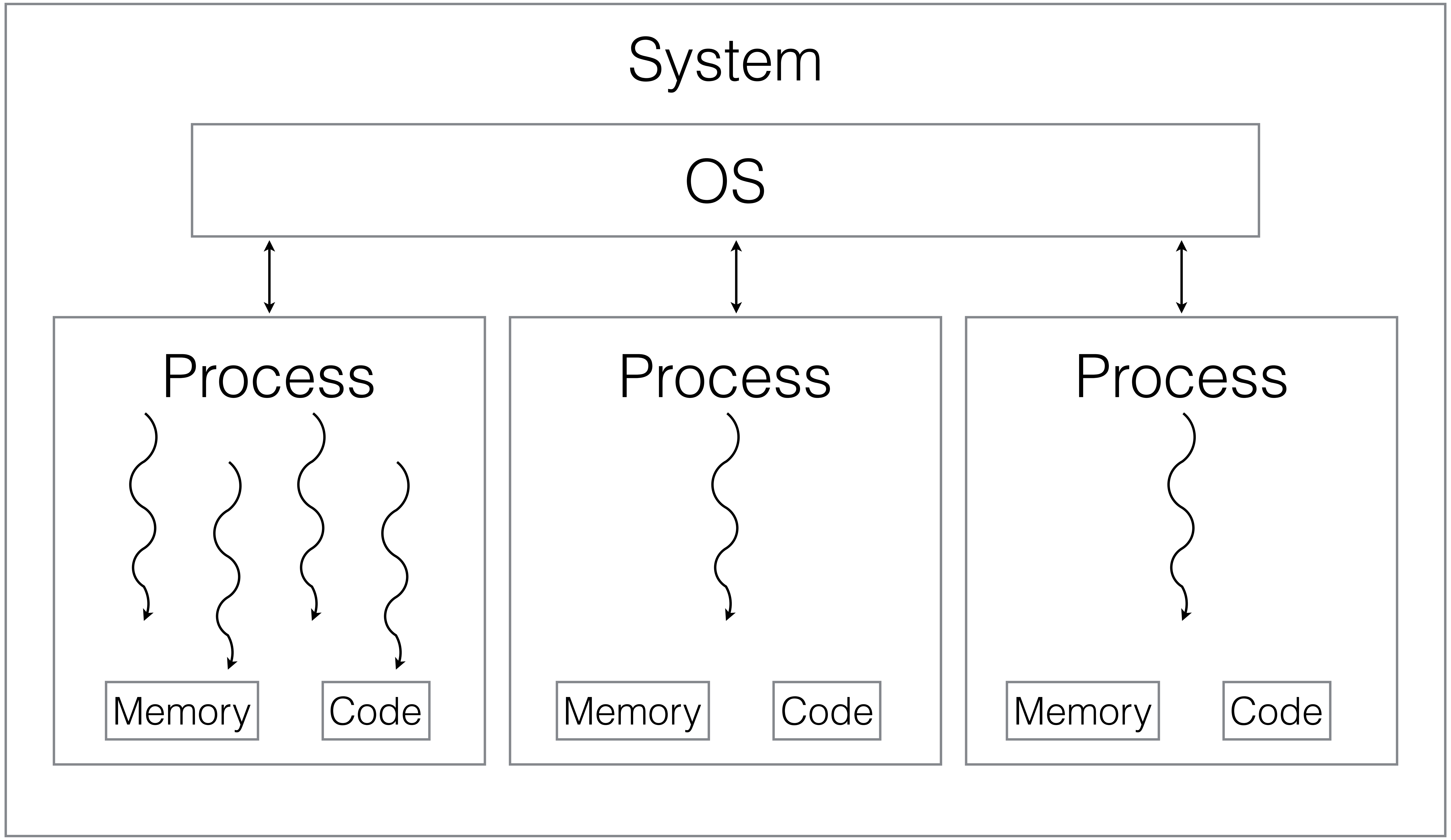
Process*



- Start (stop) process
- Share data between processes (files, pipes, …)
- Use hardware

* a.k.a: application, task

Thread*



* a.k.a: FreeRTOS task

Threads

- Use same code inside process
- Share memory inside process (globals & heap)
- Each thread has own
  stack, stack pointer & program counter

Why multiple threads?

Sometimes, code path needs to

- Wait specific time (delay, sleep)
- Wait for I/O, e.g.
  - User input
  - Listen for network data (web server)
  - Communication with hardware

Single-thread solution

Single-thread solution

- **Poll** every subsystem

```
public static void main(String[] args) {
    // initialize subsystems

    while (true) {
        for (SubSystem subSystem : subSystems) {
            subSystem.poll();
        }
    }
}
```

(many single-threaded embedded systems work like this)

Example

- Running your favorite IDE
  - Subsystem 1: GUI
  - Subsystem 2: Lengthy operation (updating your code index)

~~Subsystems need to be split up in small parts to be non-blocking~~

Suppose…

We can automatically **pause**
the lengthy operation when needed?

Preemptive threading model

- Perform **context switch** when needed (non-voluntary):
  - Save context* of thread which is paused
  - Load context* of thread which is resumed
- **Scheduler** decides when to switch

*Registers, stack pointer, program counter

Thread states

- **Inactive**: associated code is not running (anymore)
  (Java: new, terminated)

- **Runnable**: thread is ready to execute new code
  (Java: runnable)

- **Blocked**: thread is waiting some external event happens
  (Java: blocked, waiting, timed waiting)

OS scheduler (1)

- Keep track of **thread states**
  when a running thread enters **blocked** / **terminated** state, switch to another thread that is runnable

- Update thread states on **external events** (expired time, I/O, …)

- Perform **time slicing**
  when a thread has been running too long (e.g. 10ms), switch to another

OS scheduler (2)

- Enter **low-power idle**
  when there are no runnable threads anymore
- Schedule on **multiple cores**
- Thread **priorities**
- Schedule threads in **other processes**

Try it out

Create threads

```java
public static void main(String[] args) throws InterruptedException {
    final Thread t1 = new Thread(() -> {
        // thread code goes here
    });

    t1.start();

    // main thread continues here


    // this will block until the thread is finished
    t1.join();
}
```

Use `Thread.getState()` to poll the thread state

Try to reproduce all possible states (see the `State` enum)

Share data between threads

Thread-safety

Thread-safe code is code which you can use from multiple threads, with well-defined (and intended) behavior

What can go wrong?

some demos

Options to guarantee thread-safety

- Don’t use multiple threads
  (make sure a specific data is only used by 1 thread)

- Add object synchronization / locks / mutexes

- Immutability

- Atomicity

Locking

```java
private final SomeClass object1 = new SomeClass();
private final SomeClass object2 = new SomeClass();

public void myMethod1() {
    synchronized (this) {
        object1.doSomething();
    }
}

public void myMethod2() {
    synchronized (this) {
        object2.doSomething();
    }
}
```

```java
private final SomeClass object1 = new SomeClass();
private final SomeClass object2 = new SomeClass();

public synchronized void myMethod1() {
    object1.doSomething();

}

public synchronized void myMethod2() {
    object2.doSomething();
}
```

Locking

```java
public class SomeClass {
    public void doSomething() {
        // do something
    }

    public void doSomethingElse() {
        // do something else
    }
}

private final SomeClass object1 = new SomeClass();

public void myMethod1() {
    synchronized (object1) {
        object1.doSomething();
    }
}

public void myMethod2() {
    synchronized (object1) {
        object1.doSomethingElse();
    }
}
```

```java
public class SomeClass {
    public synchronized void doSomething() {
        // do something
    }

    public synchronized void doSomethingElse() {
        // do something else
    }
}

private final SomeClass object1 = new SomeClass();

public void myMethod1() {
    object1.doSomething();
}

public void myMethod2() {
    object1.doSomethingElse();
}
```

Dead-lock

```java
public class SomeClass {
    public synchronized void doSomething() {
        System.out.println("doSomething()");
    }
}

private final SomeClass object1 = new SomeClass();

public synchronized void myMethod1() {
    someFunction();
    object1.doSomething();
}

public void myMethod2() {
    synchronized (object1) {
        myMethod1();
    }
}
```

Volatile keyword

```
private volatile int someVariable = 0;
```

- Only if object synchronization is not suitable
- Makes sure the value is always read / written in main memory (not only in local cache)

Try it out

Make the code thread-safe

Inter-thread communication

Inter-thread communication

- BlockingQueue (Deque, Stack)
- Semaphore
- CountDownLatch (CyclicBarrier)
- wait() / notify() / notifyAll()

ArrayBlockingQueue

- Has a fixed capacity

Blocking

```
queue.put(someObject);

final SomeClass item = queue.take();
```

Return immediately

```
queue.offer(someObject);

final SomeClass item = queue.poll();
```

Blocking with timeout

```
queue.offer(someObject, 1, TimeUnit.SECONDS);

final SomeClass item = queue.poll(1, TimeUnit.SECONDS);
```

Semaphore

```
semaphore.acquire();

semaphore.release();
```

```
semaphore.tryAcquire();

semaphore.tryAcquire(1, TimeUnit.SECONDS);
```

Try it out

ArrayBlockingQueue, Semaphore

Creating threads is expensive

ThreadPoolExecutor

- Thread(s) block on incoming Runnables in a queue
- Rest of system can put Runnables in the queue

ScheduledThreadPoolExecutor

- Same as ThreadPoolExecutor
- But also functionality for scheduling tasks in the future

```java
final ScheduledFuture<SomeClass> future = executor.schedule(() -> {
    // some Callable
    return new SomeClass();
}, 5, TimeUnit.SECONDS);


// after some time
final SomeClass result = future.get();
```

Synchronous vs Asynchronous

Synchronous vs Asynchronous

- Synchronous operation: blocks until complete
- Asynchronous: only initiates the operation

Example: Request/Reply

```java
public void synchronousMethod() {
    try {
        final Reply reply = sendRequest();
        // do something with reply here

    } catch (TimeoutException e) {
        // handle timeout here
    }
}
```

```java
public void asynchronousMethod() {
    // send the request, but do not block
    sendRequest();
}

public void onReplyReceived(final Reply reply) {
    // might be called from another thread
}

public void onTimeout() {
    // might be called from another thread
}
```

Multi-threading in Renos

Multi-threading in Renos

- ZeroMQ message system
  Run loop in `MessageReceiver`

- Watchers
  `scheduler.getWatcher()`

- Schedulers
  `scheduler.getRealtime()`
  `scheduler.getBackground()`