

# Concurrent Systems

**Last updated:** January 15<sup>th</sup> 2019, at 12.58pm

## Contents

<b>1</b>	<b>This semester's course</b>	<b>2</b>
1.1	Concurrent systems . . . . .	2
1.1.1	Properties of concurrent systems . . . . .	2
1.1.2	Tools for concurrent systems . . . . .	2
1.2	Quantum computing . . . . .	2
1.3	Theoretical aspects . . . . .	2
1.4	Outcomes . . . . .	3
1.5	Lecture Plan . . . . .	3
<b>2</b>	<b>Introduction to Concurrent Systems</b>	<b>3</b>
2.1	Why programme concurrent systems? . . . . .	3
2.1.1	Efficiency . . . . .	3
2.1.2	Simplification . . . . .	5
2.1.3	Necessity . . . . .	5
2.2	Aspects of concurrent systems . . . . .	6
2.2.1	Necessary tools . . . . .	6
2.2.2	Properties . . . . .	6
<b>3</b>	<b>Concurrent processes in Java</b>	<b>6</b>
3.1	Defining process classes . . . . .	6
3.2	Defining process behaviour . . . . .	6
3.3	Creating a process . . . . .	7
3.4	Starting a thread . . . . .	7
3.5	Waiting for a thread to stop . . . . .	7
3.6	Sharing data between processes . . . . .	7
3.7	Some useful methods . . . . .	8
3.7.1	Access . . . . .	8
3.7.2	Control . . . . .	8
3.7.3	Priorities . . . . .	8

## **Reminder**

You should now have handed in part 2 of your logbook.

## **1 This semester's course**

### **1.1 Concurrent systems**

#### **1.1.1 Properties of concurrent systems**

- Critical sections
- Mutual exclusion
- Deadlock
- Starvation
- Liveness
- Loosely connectedness

#### **1.1.2 Tools for concurrent systems**

- Shared variables
- Semaphores
- Monitors

### **1.2 Quantum computing**

- Circuits as matrices and vectors
- Quantum systems
- Quantum circuits
- Quantum algorithms

### **1.3 Theoretical aspects**

- Correctness
- Complexity

## 1.4 Outcomes

1. Discuss the classification of algorithms according to efficiency and complexity
2. Prove code correct
3. Demonstrate a knowledge of the characteristics of a range of concurrency paradigms
4. Explain the difference between classical and quantum computing
5. Use a standard notation to analyse the efficiency and complexity of algorithms

## 1.5 Lecture Plan

Week	Topic
13	Introduction to Concurrent Systems
14	Dekker's Algorithm
15	Semaphores
16	Monitors
17	Modelling Circuits
— Guidance Week —	
19	Quantum Systems
20	Quantum Computing
21	Correctness
22	Complexity
23	Overspill/recap
24	

## 2 Introduction to Concurrent Systems

### 2.1 Why programme concurrent systems?

- Because they are efficient.  
Deterministic polynomial *vs.* nondeterministic polynomial
- Because they simplify programming.  
GUIs
- Because you have to.  
Operating systems.

#### 2.1.1 Efficiency

##### Sequential merge sort

### Algorithm

```
public void mergeSort() {
    int half; Sort left, right;
    if (size > 1) {
        half = size/2;
        left = new Sort(list,0,half-1);
        right = new Sort(list,half,size-1);
        left.mergeSort(); right.mergeSort();
        merge(left,right);
    }
}
```

### Complexity

- Assume `merge` of  $N$  items takes  $N$  “time units”  $t$ .
- How many merges?

$$n \left\{ \begin{array}{lcl} 1 \text{ merge} & \text{each } N & = 2^n \\ 2 \text{ merges} & \text{each } \frac{N}{2} & = 2^{n-1} \\ & \vdots & \\ 2^{n-1} \text{ merges} & \text{each } \frac{N}{2^{n-1}} & = 2 \\ 2^n \text{ merges} & \text{each } \frac{N}{2^n} & = 1 \end{array} \right. \left| \begin{array}{l} 1 \times N = Nt \\ 2 \times \frac{N}{2} = Nt \\ \\ 2^{n-1} \times \frac{N}{2^{n-1}} = Nt \\ 2^n \times \frac{N}{2^n} = Nt \end{array} \right.$$

So  $n \times Nt$ . What is  $n$ ?  $2^n = N \Rightarrow n = \log N$ .

So (sequential) mergesort  $tN \log N$ .

### Parallel merge sort

#### Algorithm

```
• public void mergeSort() throws InterruptedException {
    int half; Sort left, right; // Note: Sort extends Thread
    if (size > 1) {
        half = size/2;
        left = new Sort(list,0,half-1);
        right = new Sort(list,half,size-1);
        left.start(); right.start();
        left.join(); right.join();
        merge(left,right);
    }
}
```

### Complexity

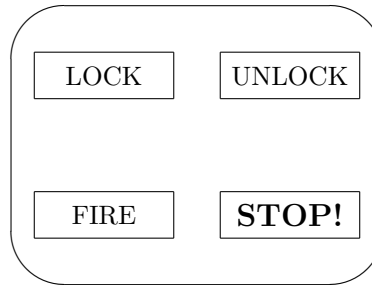
- Merges at each level can be executed in parallel

$$\begin{array}{rcl}
 1 \text{ merge} & \text{each } N = 2^n & \left| \begin{array}{l} 1 \times N = N \text{ } t \\ 1 \times \frac{N}{2} = \frac{N}{2} \text{ } t \end{array} \right. \\
 2 \text{ merges} & \text{each } \frac{N}{2} = 2^{n-1} & \\
 & \vdots & \\
 2^{n-1} \text{ merges} & \text{each } \frac{N}{2^{n-1}} = 2 & \left| \begin{array}{l} 1 \times \frac{N}{2^{n-1}} = \frac{N}{2^{n-1}} \text{ } t \\ 1 \times \frac{N}{2^n} = \frac{N}{2^n} \text{ } t \end{array} \right. \\
 2^n \text{ merges} & \text{each } \frac{N}{2^n} = 1 &
 \end{array}$$

So  $\sum_{i=0}^n \frac{N}{2^i} = N + \frac{N}{2} + \frac{N}{4} + \dots + 1 = 2N$

- So (parallel) mergesort:  $2Nt$ .

#### 2.1.2 Simplification



Sequential	Parallel
<code>while (true) {</code>	<code>while (true) {</code>
<code>  LOCK.listenTo();</code>	<code>  LOCK.listenTo()   </code>
<code>  UNLOCK.listenTo();</code>	<code>  UNLOCK.listenTo()   </code>
<code>  FIRE.listenTo();</code>	<code>  FIRE.listenTo()   </code>
<code>  STOP.listenTo();</code>	<code>  STOP.listenTo()   </code>
<code>}</code>	<code>}</code>

The sequential system imposes an ordering on the buttons. The code of the `listenTo`s may become complex in order to ensure that, for example, the **STOP!** button *always* prevents any of the other buttons from working. The parallel version may also require some complex code (see later weeks on e.g. critical sections) but is conceptually clearer.

#### 2.1.3 Necessity

Operating Systems

- I/O devices
- Interrupts

- Multi-tasking
- Networks

## 2.2 Aspects of concurrent systems

**Note:** A concurrent system is not necessarily truly parallel — timeslicing, interleaving.

### 2.2.1 Necessary tools

- Communication
- Synchronisation

### 2.2.2 Properties

- Complexity
- Correctness
- Granularity

## 3 Concurrent processes in Java

### 3.1 Defining process classes

A parallel process is an instance of a `Thread` — a `Thread` runs a `Runnable`.

- Either implement the `Runnable` class

```
class Process implements Runnable {...}
```

- or extend the `Thread` class

```
class Process extends Thread {...}
```

### 3.2 Defining process behaviour

```
public void run() {  
    ...  
}
```

### 3.3 Creating a process

- From a subclass of Thread

```
Process process = new Process();
```

- From an implementation of Runnable

```
Thread thread = new Thread(new MyRunnable());
```

Note: this does *not* start the thread running. Note also that named threads can be defined:

```
Thread process = new Process(threadGroup, "My process");
```

or

```
Thread thread = new Thread(threadGroup, new MyRunnable(), "My process");
```

### 3.4 Starting a thread

```
myThread.start();
```

Note: do *not* call run().

### 3.5 Waiting for a thread to stop

```
try {  
    myThread.join();  
} catch (InterruptedException e) {} ;
```

### 3.6 Sharing data between processes

- a non-static variable is unique to the instance

```
int belongsToPooh;
```

- a **static** variable is shared by all instances of the class

```
static int botherItsPigletsToo;
```

## 3.7 Some useful methods

### 3.7.1 Access

- `someThread.checkAccess()`  
Is the currently running thread allowed to modify `someThread`?
- `someThread.getId()` (returns a **long**)
- `someThread.getName()` (returns a `String`)

### 3.7.2 Control

- `join`:
  - `someThread.join()`  
Wait for `someThread` to die
  - `someThread.join(millis)`  
Wait at most `millis` ms for `someThread` to die (`millis` is **long**)
- **static void** `sleep(millis)`  
Currently executing thread sleeps for `millis` ms.
- **static void** `yield()`  
Currently executing thread temporarily allows another thread to execute.

### 3.7.3 Priorities

- **static void** `setPriority(int newPriority)`
- **int** `getPriority()`
- `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`