Concurrent Systems
Nebenläufige Systeme

II. Concurrency

Wolfgang Schröder-Preikschat

November 12, 2020
Agenda

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
Outline

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
discussion on two fundamental **abstract concepts**: concurrency (Ger. *Nebenläufigkeit*),
- designates the relation of causal independent events
- is related to events that have no mutual influence

causality (Ger. *Kausalität, Ursächlichkeit*)
- designates the relation between cause and effect
- is the causal chain or connection of two events
discussion on two fundamental **abstract concepts**: concurrency (Ger. *Nebenläufigkeit*),
- designates the relation of causal independent events
- is related to events that have no mutual influence

causality (Ger. *Kausalität, Ursächlichkeit*)
- designates the relation between cause and effect
- is the causal chain or connection of two events

**Definition (concurrent)**

Events occur or are concurrent if none is the cause of the other.
Subject Matter

- discussion on two fundamental abstract concepts: concurrency (Ger. *Nebenläufigkeit*),
  - designates the relation of causal independent events
  - is related to events that have no mutual influence
- causality (Ger. *Kausalität, Ursächlichkeit*)
  - designates the relation between cause and effect
  - is the causal chain or connection of two events

**Definition (concurrent)**

Events occur or are concurrent if none is the cause of the other.

- explanation of the relation of these concepts to resource sharing
  - differentiated with respect to various types of resources and sharing
  - classified as to appropriate or necessary synchronisation paradigms
Outline

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
Principle of Causality

causal chain of events related to some other event $e_i$:

- is cause of $e_i$
- is concurrent to $e_i$
- is effect of $e_i$

$A$, $B$ and $C$ denote some computation on a private or shared processor.
Principle of Causality

causal chain of events related to some other event $e_i$:

- $A$, $B$ and $C$ denote some computation on a private or shared processor
**Principle of Causality**

causal chain of events related to some other event $e_i$:

- $A$, $B$ and $C$ denote some computation on a private or shared processor
- an event is concurrent to another event ($e_i$) if it lies in the elsewhere of the other event ($e_i$)
- the event is neither cause nor effect of the other event ($e_i$)
Principle of Causality

causal chain of events related to some other event $e_i$:

- $A$, $B$ and $C$ denote some computation on a private or shared processor
- an event is concurrent to another event ($e_i$) if it lies in the elsewhere of the other event ($e_i$)
- the event is neither cause nor effect of the other event ($e_i$)
- as the case may be, it is cause/effect of other events (different from $e_i$) that are lying in the elsewhere (cf. dash-and-dot line)
computations can be carried out concurrently provided that:

- **general**
  - none requires a result of the other (cf. p. 10)
  - non-existent *data dependencies*
Order of Precedence

computations can be carried out concurrently provided that:

- **general**
  - none requires a result of the other (cf. p. 10)
  - non-existent *data dependencies*

- **special**
  - none depends on delays brought forth by the other
    - deadlines may be missed rarely or under no circumstances
    - periods may be stretched up to a certain limit or not at any time
  - non-existent *timing restrictions* $\leadsto$ *real-time processing*
interrelation of computations/events constrains concurrency

Event correlations v. Processing modes

\[
\begin{align*}
\text{“is cause of”} & \quad \mapsto \quad \text{sequential} \quad \text{(realised before/at run-time)} \\
\text{“is effect of”} & \quad \mapsto \quad \text{parallel} \quad \text{(realised in logical/real terms)}
\end{align*}
\]
interrelation of computations/events constrains concurrency

Event correlations v. Processing modes

```
“is cause of”  ”  sequential (realised before/at run-time)
“is effect of”
```

```
“is concurrent to”  ➞  parallel  (realised in logical/real terms)
```

decrease of the portion of **sequential code** is an important aspect
Limits in the Degree of Concurrency

- Amdahl’s Law [1]: speed-up ($su$) achievable by parallel processors
  - work load remains constant with the varying number of processors
  - aim at reducing overall computation time for a given fixed-size problem
Amdahl’s Law [1]: speed-up \((su)\) achievable by parallel processors

- work load remains constant with the varying number of processors
- aim at reducing overall computation time for a given fixed-size problem

\[
su = \frac{(r_s + r_p)}{(r_s + \frac{r_p}{n})} = \frac{1}{r_s + \frac{r_p}{n}}
\]

- \(r_s\) ratio of sequential code
- \(r_p\) ratio of parallel code, independent of \(n\)
- \(n\) number of processors
Amdahl’s Law [1]: speed-up ($su$) achievable by parallel processors
- Work load remains constant with the varying number of processors
- Aim at reducing overall computation time for a given fixed-size problem

$$su = \frac{r_s + r_p}{r_s + \frac{r_p}{n}}$$

- $r_s$ ratio of sequential code
- $r_p$ ratio of parallel code, independent of $n$
- $n$ number of processors

- Speed-up will be constrained by **data management housekeeping**
  - The nature of this overhead appears to be sequential
Adapting the Work Load

- Gustafson’s Law [4]: scaled speed-up \((ssu)\), “hands-on experience”
  - work load varies linearly with the number of processors
  - aim at getting better results for a given fixed computation time
Adapting the Work Load

- Gustafson’s Law [4]: scaled speed-up (ssu), “hands-on experience”
- work load varies linearly with the number of processors
- aim at getting better results for a given fixed computation time

\[ ssu = \frac{r_s + r_p \times n}{r_s + r_p} = r_s + r_p \times n = n + (1 - n) \times r_s \]

- \( r_p \): ratio of parallel code, scales with \( n \)
- \( r_s, n \): as with Amdahl’s Law

![Graph showing speedup vs. number of processors for different values of \( r_s \).]
Adapting the Work Load

Gustafson’s Law [4]: scaled speed-up (\(ssu\)), “hands-on experience”

- work load varies linearly with the number of processors
- aim at getting better results for a given fixed computation time

\[
ssu = \frac{r_s + r_p \times n}{r_s + r_p} = r_s + r_p \times n = n + (1 - n) \times r_s
\]

- \(r_p\) ratio of parallel code, scales with \(n\)
- \(r_s, n\) as with Amdahl’s Law

- data management housekeeping (serial part) becomes less important

\(^1\text{HPC}\)
Concurrent Operations of a Computation

- Operations can be concurrent if none needs the result of the other.

```c
int foo, bar;

int sample(int tupel[2]) {
    int subtotal, product;
    foo = tupel[0];
    bar = tupel[1];
    subtotal = foo + bar;
    product = bar * foo;
    return subtotal + product;
}
```

In computation:
- Which statements can be concurrent? 
  - (6, 7) and (9, 10)
- Which statements are not concurrent? 
  - (6, 7) and (9, 10)

Defined by the causal order (Ger. Kausalordnung) of the statements as far as the logical dimension of a program is concerned, but there is also a physical dimension, namely when it comes to the execution of that program by a real processor; level of abstraction.
Concurrent Operations of a Computation

operations can be concurrent if none needs the result of the other:

```c
int foo, bar;

int sample(int tupel[2]) {
    int subtotal, product;

    foo = tupel[0];
    bar = tupel[1];

    subtotal = foo + bar;
    product = bar * foo;

    return subtotal + product;
}
```
Concurrent Operations of a Computation

operations can be concurrent if none needs the result of the other:

```c
int foo, bar;

int sample(int tupel[2]) {
    int subtotal, product;

    foo = tupel[0];
    bar = tupel[1];

    subtotal = foo + bar;
    product = bar * foo;

    return subtotal + product;
}
```

in computation:
■ which statements can be concurrent?

■ which statements are not concurrent?
Concurrent Operations of a Computation

Operations can be concurrent if none needs the result of the other:

1
int foo, bar;

2

3
int sample(int tupel[2]) {
4   int subtotal, product;

5

6   foo = tupel[0];
7   bar = tupel[1];

8

9   subtotal = foo + bar;
10  product = bar * foo;
11
12  return subtotal + product;
13 }

In computation:

- Which statements can be concurrent?
  - 6 and 7
  - 9 and 10

- Which statements are not concurrent?
  - (6, 7) and (9, 10)
  - (9, 10) and 12
Concurrent Operations of a Computation

- operations can be concurrent if none needs the result of the other:

```c
int foo, bar;

int sample(int tupel[2]) {
    int subtotal, product;

    foo = tupel[0];
    bar = tupel[1];

    subtotal = foo + bar;
    product = bar * foo;

    return subtotal + product;
}
```

- defined by the causal order (Ger. *Kausalordnung*) of the statements

  - as far as the logical dimension of a program is concerned

- in computation:
  - which statements can be concurrent?
    - 6 and 7
    - 9 and 10
  - which statements are not concurrent?
    - (6, 7) and (9, 10)
    - (9, 10) and 12

- defined by the causal order (Ger. *Kausalordnung*) of the statements
Concurrent Operations of a Computation

- operations can be concurrent if none needs the result of the other:

```c
int foo, bar;

int sample(int tupel[2]) {
    int subtotal, product;
    foo = tupel[0];
    bar = tupel[1];
    subtotal = foo + bar;
    product = bar * foo;
    return subtotal + product;
}
```

- defined by the causal order (Ger. *Kausalordnung*) of the statements
- as far as the logical dimension of a program is concerned
- but there is also a physical dimension, namely when it comes to the execution of that program by a real processor \(\leadsto\) level of abstraction

- in computation:
  - which statements *can be* concurrent?
    - 6 and 7
    - 9 and 10
  - which statements *are not* concurrent?
    - (6, 7) and (9, 10)
    - (9, 10) and 12
Level of Abstraction

- A concurrent operation (in logical terms) at a higher level can be sequential (in real terms) at a lower level.

- A sequential operation (in logical terms) at a higher level can be “concurrent” (i.e., non-sequential in real terms) at a lower level.
Level of Abstraction

- a concurrent operation (in logical terms) at a higher level can be sequential (in real terms) at a lower level
  - the operation handles a resource that can be used only consecutively
    - a single memory area that is shared by multiple computations
    - a single communication bus that is shared by multiple processing units
  - simultaneous executions are constrained by the resource characteristic

a sequential operation (in logical terms) at a higher level can be “concurrent” (i.e., non-sequential in real terms) at a lower level
- the operation appears to be complex, consists of multiple sub-steps
  - the \( n \)-bit assignment on a \( n^2 \)-bit machine, with \( n = 16, 32, 64 \)
  - the addition of a number to a shared variable located in main memory
    - simultaneous execution of the sub-steps must be considered (cf. p.18)

This reveals a race condition, substantial critical situation:

- error in real-time processing, especially in case of hard deadlines.
Level of Abstraction

- a concurrent operation (in logical terms) at a higher level can be sequential (in real terms) at a lower level
  - the operation handles a resource that can be used only consecutively
    - a single memory area that is shared by multiple computations
    - a single communication bus that is shared by multiple processing units
  - simultaneous executions are constrained by the resource characteristic
- may result in a *performance penalty*, non-critical situation

© wosch, thoenig  CS (WS 2020/21, LEC 2)  Causality – Dimensions  11
a sequential operation (in logical terms) at a higher level can be “concurrent” (i.e., non-sequential in real terms) at a lower level

- the operation appears to be complex, consists of multiple sub-steps
  - the $n$-bit assignment on a $\frac{n}{2}$-bit machine, with $n = 16, 32, 64$
  - the addition of a number to a shared variable located in main memory

- simultaneous execution of the sub-steps must be considered (cf. p. 18)
Level of Abstraction

- A sequential operation (in logical terms) at a higher level can be “concurrent” (i.e., non-sequential in real terms) at a lower level.
  - The operation appears to be complex, consists of multiple sub-steps:
    - The $n$-bit assignment on a $\frac{n}{2}$-bit machine, with $n = 16, 32, 64$.
    - The addition of a number to a shared variable located in main memory.
  - Simultaneous execution of the sub-steps must be considered (cf. p. 18).

- Reveals a race condition, substantial critical situation.
Level of Abstraction

- a concurrent operation (in logical terms) at a higher level can be sequential (in real terms) at a lower level

→ may result in a *performance penalty*, non-critical situation but for...²

- a sequential operation (in logical terms) at a higher level can be “concurrent” (i.e., *non-sequential* in real terms) at a lower level

→ reveals a *race condition*, substantial critical situation: *error*

²real-time processing, especially in case of hard deadlines.
Resource Classification

permanent, limited
- reusable
  - preemptable
  - non-preemptable
- consumable
  - temporary, unlimited

resources are reusable, but always only of limited supply. They are acquired, occupied, used, and released (when no longer required). In-use resources are preemptable or non-preemptable, depending on whether allocation to another occupant is possible. When non-preemptable, they are exclusively owned by an occupant. Temporary resources are of unlimited supply, they are consumable, i.e. produced, received, used, and destroyed (when no longer required).

Also referred to as “persistent.”
permanent\(^3\) resources are **reusable**, but always only of limited supply
- they are acquired, occupied, used, and released (when no longer required)

  *in-use resources* are preemptable or non-preemptable, depending on whether allocation to another occupant is possible
- when non-preemptable, they are exclusively owned by an occupant

\(^3\)Also referred to as “persistent”.
Resource Classification

- **permanent, limited**
  - reusable
    - preemptable
    - non-preemptable
- **temporary, unlimited**
  - consumable

- temporary resources are of unlimited supply, they are **consumable**
  - i.e. produced, received, used, and destroyed (when no longer required)
## Resource Peculiarities

**hardware resources** as to be managed, e.g., by an operating system

<table>
<thead>
<tr>
<th>Reusable</th>
<th>Consumable</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>signal</td>
</tr>
<tr>
<td>memory</td>
<td></td>
</tr>
<tr>
<td>peripheral</td>
<td></td>
</tr>
<tr>
<td>CPU, FPU, GPU; MMU</td>
<td>IRQ, NMI, trap</td>
</tr>
<tr>
<td>RAM, scratch pad, flash</td>
<td></td>
</tr>
<tr>
<td>input, output, storage</td>
<td></td>
</tr>
</tbody>
</table>
Resource Peculiarities

- **software resources** as to be managed by any other program

<table>
<thead>
<tr>
<th>Reusable</th>
<th>Consumable</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>signal</td>
</tr>
<tr>
<td>data</td>
<td>notice</td>
</tr>
<tr>
<td></td>
<td>message</td>
</tr>
<tr>
<td></td>
<td>packet, stream</td>
</tr>
<tr>
<td>critical section/region</td>
<td></td>
</tr>
</tbody>
</table>
**Resource Peculiarities**

<table>
<thead>
<tr>
<th>reusable</th>
<th>consumable</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>signal</td>
</tr>
<tr>
<td>critical section/region</td>
<td>notice</td>
</tr>
<tr>
<td>data</td>
<td>message</td>
</tr>
<tr>
<td>variable, placeholder</td>
<td>packet, stream</td>
</tr>
</tbody>
</table>

- **software resources** as to be managed by any other program

  - reusable data resources are notably **container** for consumable resources
    - the latter must be contained in variables/placeholders to be processible
  - availability of the former constrains production/consumption of the latter
## Resource Peculiarities

- **hardware resources** as to be managed, e.g., by an operating system
  
<table>
<thead>
<tr>
<th>Reusable</th>
<th>Consumable</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>signal</td>
</tr>
<tr>
<td>CPU, FPU, GPU; MMU</td>
<td>IRQ, NMI, trap</td>
</tr>
<tr>
<td>memory</td>
<td></td>
</tr>
<tr>
<td>RAM, scratch pad, flash</td>
<td></td>
</tr>
<tr>
<td>peripheral</td>
<td></td>
</tr>
<tr>
<td>input, output, storage</td>
<td></td>
</tr>
</tbody>
</table>

- **software resources** as to be managed by any other program
  
<table>
<thead>
<tr>
<th>Reusable</th>
<th>Consumable</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>signal</td>
</tr>
<tr>
<td>critical section/region</td>
<td>notice</td>
</tr>
<tr>
<td>data</td>
<td>message</td>
</tr>
<tr>
<td>variable, placeholder</td>
<td>packet, stream</td>
</tr>
</tbody>
</table>

- Reusable and consumable resources imply different **use patterns**
Resource Use Patterns

- if so, **reusable resources** are subject to **multilateral** synchronisation

- **consumable resources** are subject to **unilateral** synchronisation
if so, reusable resources are subject to multilateral synchronisation

- provided that the following two basic conditions (i.e., constraints) apply:
  - i. resource accesses by computations may happen (quasi-) simultaneously
  - ii. simultaneous accesses may cause a conflicting state change of the resource

- simultaneous use of a shared resource this way must be coordinated
  - coordination may affect computations in a blocking or non-blocking manner

At the same level of abstraction, use of a shareable resource is exclusive in the blocking case or never refused in the non-blocking case.

© wosch, thoenig  CS (WS 2020/21, LEC 2)  Resource Sharing – Principles
Resource Use Patterns

- consumable resources are subject to unilateral synchronisation
- generally also referred to as logical or conditional synchronisation:
  - logical – as indicated by the “role playing” of the involved computations
  - conditional – as indicated by a condition for making computational progress
- use of a temporary resource follows a causal course of events or actions
  - by affecting producers in a non-blocking and consumers in a blocking way
Resource Use Patterns

- if so, reusable resources are subject to **multilateral** synchronisation

- consumable resources are subject to **unilateral** synchronisation

- simultaneous computations **overlap** in time, interfere with each other
  - they become critical in any case if they also overlap in (identical) place
Consolidating Example

Character Buffer of Limited Size

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

in which buffer is a reusable and item is a consumable resource
Consolidating Example

Character Buffer of Limited Size

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

in which buffer is a reusable and item is a consumable resource

which logical problems exist?
Consolidating Example

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

which logical problems exist?

- buffered items may be overwritten: **overflow**
- overlapping writes may go to the same memory location
- similar to overlapping reads, but reverse
- overlapping auto-increments may manifest wrong values

in which buffer is a **reusable** and item is a **consumable** resource
Consolidating Example

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

1. char buffer[80];
2. unsigned in = 0, out = 0;
3. void put(char item) {
   4.   buffer[in++ % 80] = item;
5. }
6. void get() {
7.   return buffer[out++ % 80];
8. }
9. in which buffer is a reusable and item is a consumable resource

which logical problems exist?

- buffered items may be overwritten: **overflow**
- values may be read from an empty buffer: **underflow**

© wosch, thoenig  CS (WS 2020/21, LEC 2)  Resource Sharing – Principles
Consolidating Example

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

■ which other problems exist?

in which buffer is a reusable and item is a consumable resource
Consolidating Example

Character Buffer of Limited Size

- assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

1. ```
   char buffer[80];
   unsigned in = 0, out = 0;
   ```

2. ```
   void put(char item) {
       buffer[in++ % 80] = item;
   }
   ```

3. ```
   char get() {
       return buffer[out++ % 80];
   }
   ```

- which other problems exist?
  - **overlapping writes** may go to the same memory location

- In which buffer is a **reusable** and item is a **consumable** resource
Consolidating Example

Character Buffer of Limited Size

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

which other problems exist?
- overlapping writes may go to the same memory location
- similar to overlapping reads, but reverse

in which buffer is a reusable and item is a consumable resource
Consolidating Example

Character Buffer of Limited Size

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

- **which other problems exist?**
  - **overlapping writes** may go to the same memory location
  - similar to **overlapping reads**, but reverse
  - **overlapping auto-increments** may manifest wrong values

in which buffer is a **reusable** and item is a **consumable** resource
Consolidating Example

Character Buffer of Limited Size

assuming that the following subroutines (put and get) are executed in any order and that they may also run simultaneously:

```c
char buffer[80];
unsigned in = 0, out = 0;

void put(char item) {
    buffer[in++ % 80] = item;
}

char get() {
    return buffer[out++ % 80];
}
```

in which buffer is a **reusable** and item is a **consumable** resource

put and get must be subject to uni- and multilateral synchronisation

they are **not** concurrent under the assumption that was made above
Serialisation of Simultaneous Computations

simultaneous computations or operations, resp., are in competition:
- **sharing** of the same **reusable** resource(s)
- **handover** of the same **consumable** resource(s)
simultaneous computations or operations, resp., are in competition:
- **sharing** of the same *reusable* resource(s)
- **handover** of the same *consumable* resource(s)

→ in either case hardware and, if applicable, software resources, too
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - **sharing** of the same **reusable** resource(s)
  - **handover** of the same **consumable** resource(s)
  → in either case hardware and, if applicable, software resources, too

- both aspects, in turn, apply against the background of the following:
  - i. the moment of an **simultaneous operation** is not predetermined
  - ii. the operation in question is complex (i.e., consists of multiple steps)
  - iii. the characteristic of this operation is its **divisibility** in temporal respect
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - **sharing** of the same **reusable** resource(s)
  - **handover** of the same **consumable** resource(s)
  - in either case hardware and, if applicable, software resources, too

- **conflict-prone operations** must go on **seriatim** (Ger. *nacheinander*)
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - **sharing** of the same **reusable** resource(s)
  - **handover** of the same **consumable** resource(s)

→ in either case hardware and, if applicable, software resources, too

- **conflict-prone operations** must go on **seriatim** (Ger. *nacheinander*)

  **off-line**
  - static scheduling based on control-flow and data dependencies
  - **analytical approach** that takes *a priori* knowledge as given
  - at run-time, dependable operations are implicitly synchronised
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - sharing of the same reusable resource(s)
  - handover of the same consumable resource(s)
  - in either case hardware and, if applicable, software resources, too
- both aspects, in turn, apply against the background of the following:
  - the moment of an simultaneous operation is not predetermined

- conflict-prone operations must go on seriatim (Ger. nacheinander)

  - off-line: static scheduling based on control-flow and data dependencies
  - analytical approach that takes a priori knowledge as given (v.s. i)
  - at run-time, dependable operations are implicitly synchronised
Simultaneous computations or operations, resp., are in competition:
- **Sharing** of the same reusable resource(s)
- **Handover** of the same consumable resource(s)
  - in either case hardware and, if applicable, software resources, too

**Conflict-prone operations** must go on *seriatim* (Ger. *nacheinander*)

**On-line**
- suitable explicit synchronisation of all dependable operations
- **Constructive approach** in shape of a **non-sequential program**
- based on either pessimistic or optimistic run-time assumptions
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - **sharing** of the same **reusable** resource(s)
  - **handover** of the same **consumable** resource(s)
  - in either case hardware and, if applicable, software resources, too

- **conflict-prone operations** must go on **seriatim** (Ger. *nacheinander*)

- the chosen synchronisation method should be **minimally invasive**
Divisibility in Temporal Respect

- when the steps of a complex operation may overlap at run-time
  - due to **simultaneous operation** (Ger. *Simultanbetrieb*)
Divisibility in Temporal Respect

by way of example an auto-increment operator (cf. p. 16):

as compiled from C to ASM (x86):
gcc -O3 -m32 -static -S

```c
in++
1 movl _in, %ecx
2 leal 1(%ecx), %eax
3 movl %eax, _in

out++
4 movl _out, %ecx
5 leal 1(%ecx), %eax
6 movl %eax, _out
```

non-critical overlapping execution of `in++` and `out++`

self-overlapping execution of `in++` or `out++`, resp.

the critical case may result in wrong reading (Ger. Zählerwert) of `in` or `out` in `in++` or `out++` are not concurrent to oneself, resp.: they are not re-entrant

Assuming that processor registers are private to each computation.
Divisibility in Temporal Respect

by way of example an auto-increment operator (cf. p.16):
- as compiled from C to ASM (x86): gcc -O3 -m32 -static -S

```
in++
1 movl _in, %ecx
2 leal 1(%ecx), %eax
3 movl %eax, _in
```

```
out++
4 movl _out, %ecx
5 leal 1(%ecx), %eax
6 movl %eax, _out
```

**non-critical**
- overlapping execution of `in++` and `out++`
- simultaneous operations work on different variables\(^5\)

\(^5\)Assuming that processor registers are private to each computation.
Divisibility in Temporal Respect

by way of example an auto-increment operator (cf. p.16):
- as compiled from C to ASM (x86): gcc -O3 -m32 -static -S

```c
in++
1  movl _in, %ecx
2  leal 1(%ecx), %eax
3  movl %eax, _in

out++
4  movl _out, %ecx
5  leal 1(%ecx), %eax
6  movl %eax, _out
```

critical
- self-overlapping execution of in++ or out++, resp.
- simultaneous operations work on the same variable

5Assuming that processor registers are private to each computation.
Divisibility in Temporal Respect

by way of example an auto-increment operator (cf. p.16):
- as compiled from C to ASM (x86): gcc -O3 -m32 -static -S

```
in++
1 movl _in, %ecx
2 leal 1(%ecx), %eax
3 movl %eax, _in

out++
4 movl _out, %ecx
5 leal 1(%ecx), %eax
6 movl %eax, _out
```

- the critical case may result in wrong reading (Ger. Zählerwert) of in/out
  - in++ or out++ are not concurrent to oneself, resp.: they are not re-entrant
Establishing of Synchronism

- assure a conflict-prone complex operation of (logical) **indivisibility**
Establishing of Synchronism

- assure a conflict-prone complex operation of (logical) **indivisibility**
- interpret the equivalent computation as **elementary operation** (ELOP)
  - an operation of indivisible cycle (Ger. *zeitlicher Ablauf*), apparently **atomic**

---

© wosch, thoenig  
CS (WS 2020/21, LEC 2)  
Resource Sharing – Synchronisation
indivisibility of a cycle is achieved through **synchronisation**,\(^6\) i.e.:

i. coordination of the cooperation and competition between processes

ii. calibration of real-time clocks or data in distributed systems

iii. sequencing of events along the causal order

\(^6\)(Gr. **sýn**: synced, **chrónos**: time)
indivisibility of a cycle is achieved through synchronisation, i.e.:  

i  coordination of the cooperation and competition between processes  
ii  calibration of real-time clocks or data in distributed systems  
iii  sequencing of events along the causal order  

6 (Gr. σύν: synced, χρόνος: time)
two fundamental approaches to synchronisation are distinguished:

**blocking**  ■ ensure synchronism at *operation start*

**non-blocking**  ■ ensure synchronism at *operation end*
two fundamental approaches to synchronisation are distinguished:

- **blocking**:
  - ensure synchronism at **operation start**
  - lock potential overlapping out in the first place
  - synchronised operation is made of sequential code
two fundamental approaches to synchronisation are distinguished:

- non-blocking
  - ensure synchronism at **operation end**
  - allow potential overlapping, achieve consistency afterwards
  - synchronised operation is made of non-sequential code
two fundamental approaches to synchronisation are distinguished:

- **blocking**
  - ensure synchronism at operation start

- **non-blocking**
  - ensure synchronism at operation end

both approaches come in a variety of solutions to the same problem
Varieties of Synchronisation

Relevant to Operating Systems

- Sequential code
- Inhibiting: interruption, continuation, preemption
- Blocking: semaphore (counting), lock, condition variable, mutex
- Multilateral
- Non-blocking: CAS, LL/SC, atomic R/W, TM
- Non-sequential code

The methods are more or less disruptive of the problematic operation: sequential code by a locking protocol, for the most part, the original code can be reused, pessimistic, overlapping is not a rare event.

Non-sequential code can be reprogrammed as a transaction, for the most part, the original code cannot be reused, optimistic, overlapping is a rare event.

Wherever applicable, downsizing sequential code is basic. i.a. Amdahl’s Law (cf. p.8) argues for non-blocking synchronisation.
Varieties of Synchronisation Relevant to Operating Systems

- the methods are more or less disruptive of the problematic operation
the methods are more or less disruptive of the problematic operation:

**sequential**
- bracket sequential code by a **locking protocol**
- for the most part, the original code can be reused

  ← *pessimistic*, overlapping is **not** a rare event
Varieties of Synchronisation

Relevant to Operating Systems

- the methods are more or less disruptive of the problematic operation:
  - non-sequential
    - reprogram sequential code as a transaction
    - for the most part, the original code cannot be reused
    - optimistic, overlapping is a rare event
wherever applicable, *downsizing sequential code* is basic
- i.a. Amdahl’s Law (cf. p. 8) argues for non-blocking synchronisation
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved

- inhibiting
  - prevents other computations from launching irrespective of the eventuality of co-occurrence
  - applies to consumable resources (i.e., signals), only running computations are not delayed

- blocking
  - delays computations subject to resource availability
  - takes effect only in case of co-occurrence (overlapping)
  - applies to reusable and consumable resources
  - running computations are possibly delayed

- non-blocking
  - may force non-dominantly running computations to repeat
  - takes effect only in case of co-occurrence (overlapping)
  - applies to reusable resources, only dominantly running computations are not delayed

- it bears repeating: downsizing sequential code is basic

- where possible, non-blocking synchronisation should be the first choice

- but even then: there is no all-in-one approach for every purpose...
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved:
  - **inhibiting**
    - prevents other computations from launching
    - irrespective of the eventuality of co-occurrence
    - applies to consumable resources (i.e., signals), only
  - running computations are not delayed

© wosch, thoenig  
CS (WS 2020/21, LEC 2)  
Resource Sharing – Synchronisation
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved:

**blocking**
- delays computations subject to resource availability
  - takes effect only in case of co-occurrence (overlapping)
  - applies to reusable and consumable resources
- running computations are possibly delayed

it bears repeating: downsizing sequential code is basic

...but even then: there is no all-in-one approach for every purpose...
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved:

  - non-blocking
    - may force non-dominantly running computations to repeat
      - takes effect only in case of co-occurrence (overlapping)
      - applies to reusable resources, only
    - dominantly running computations are not delayed
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved:

  - inhibiting: prevents other computations from launching irrespective of the eventuality of co-occurrence
  - blocking: delays computations subject to resource availability; takes effect only in case of co-occurrence (overlapping); applies to reusable and consumable resources; running computations are possibly delayed
  - non-blocking: may force non-dominantly running computations to repeat; takes effect only in case of co-occurrence (overlapping); applies to reusable resources; dominantly running computations are not delayed

- it bears repeating: *downsizing sequential code* is basic
  - where possible, non-blocking synchronisation should be the first choice
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved:
  - inhibiting  ■ prevents other computations from launching
  - blocking   ■ delays computations subject to resource availability
  - non-blocking ■ may force non-dominantly running computations to repeat

- it bears repeating: *downsizing sequential code* is basic
  ■ where possible, non-blocking synchronisation should be the first choice
  ■ but even then: there is no all-in-one approach for every purpose...
Outline

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
Abstract Concepts Revisited

understanding (Ger.) *Gleichzeitigkeit* in its various meanings

**concurrency**  ■ happening together in time and place [7]

**simultaneity**  ■ occurring, done, existing together or at the same time [7]

**synchronism**  ■ fact of being synchronous; simultaneous occurrence [7]
understanding (Ger.) *Gleichzeitigkeit* in its various meanings:

- **concurrency**
  - happening together in time and place [7]
  - designates the relation of causal independent events
  - when none computation depends on results of the other
Abstract Concepts Revisited

concurrency = simultaneity – synchronism

understanding (Ger.) *Gleichzeitigkeit* in its various meanings:

**simultaneity**
- occurring, done, existing together or at the same time [7]
- effect of a certain operation mode of a computing machine
- causes possibly critical overlapping of computations
Abstract Concepts Revisited

concurrency = simultaneity — synchronism

understanding (Ger.) *Gleichzeitigkeit* in its various meanings:

- **concurrency**
  - happening together in time and place [7]
  - designates the relation of causal independent events when none computation depends on results of the other

- **simultaneity**
  - occurring, done, existing together or at the same time [7]
  - effect of a certain operation mode of a computing machine causes possibly critical overlapping of computations

- **synchronism**
  - fact of being synchronous; simultaneous occurrence [7]
  - in respect of the multiple sub-steps of a complex operation
  - achieved through “ELOP-ifying” coherent instructions

**synchronism**
- fact of being synchronous; simultaneous occurrence [7]
- in respect of the multiple sub-steps of a complex operation
- achieved through “ELOP-ifying” coherent instructions
<table>
<thead>
<tr>
<th>concurrency</th>
<th>simultaneity</th>
<th>synchronism</th>
</tr>
</thead>
</table>

- **Simultaneity** includes concurrency, but not the other way round:
  - Concurrency implies unconstrained overlapping in time and place.
  - But simultaneity may also cause overlapping that must be constrained.
Abstract Concepts Revisited

concurrency = simultaneity − synchronism

Simultaneity includes concurrency, but not the other way round. Concurrency implies unconstrained overlapping in time and place, but simultaneity may also cause overlapping that must be constrained.

Synchronism ensures that overlapped complex operations do right:
- The individual sub-steps will be strictly executed interim (consecutively) or
- A transaction will take care for consistent (pseudo-) parallel execution.
the concept of (distant) simultaneity is not absolute, but depends on the **frame of reference** (Ger. *Bezugssystem*) an observer takes

- moving- and fixed-platform thought experiment [2, p. 768]:

> The simultaneity of two distant events means a different thing to two different observers if they are moving with respect to each other.
the reference frame when reflecting on simultaneous computations is the level of abstraction (cf. p. 11) of a particular program section
Relativity of Simultaneity

Physics figuratively

the reference frame when reflecting on simultaneous computations is the **level of abstraction** (cf. p. 11) of a particular program section

- a simplistic operation (++) at a higher level may translate to a complex operation (*read-modify-write*) at a lower level
  
  - while multiple invocations of the former will take place sequentially,\(^7\) the corresponding ones of the latter may come about non-sequentially
  
  - while multiple invocations of the latter discretely can be concurrent, their logical correlation to the former makes them possibly not concurrent

\(^7\)Due to the fact that each one refers to an ELOP (cf. p. 19), logically.
the reference frame when reflecting on simultaneous computations is the **level of abstraction** (cf. p. 11) of a particular program section
- a simplistic operation (++) at a higher level may translate to a complex operation (**read-modify-write**) at a lower level

- operations must be resolved **cross-level** (from “fixed platform” observed) in order to realise their ability for concurrency or need for synchronism
Résumé

Computations can be concurrent if none needs a result of the other. They must be free of data and control-flow dependencies in order to be concurrent, computations must be quasi-simultaneous through partial virtualisation (hardware multiplexing) or real simultaneous by multiprocessing (hardware multiplication). Both techniques will induce computations to overlap in time and place. Overlapping in time causes interference but is the lesser of two evils. More critical is overlapping in place relating to the same resource, particularly with regard to the same (i.e., shared) memory area. Critical overlapping must be counteracted through synchronisation, i.e., coordination of the cooperation and competition between processes. Here: uni- or multilateral synchronisation, depending on the resource type. Synchronisation ensures for indivisibility of a computation cycle. At the outset: physical, in blocking manner, by being pessimistic. At the road's end: logical, in non-blocking manner, by being optimistic.
Résumé

- computations can be **concurrent** if none needs a result of the other
- they must be free of data and control-flow dependencies
Résumé

- Computations can be **concurrent** if none needs a result of the other.
- In order to be concurrent, computations must be **simultaneous**:
  - Quasi-simultaneous through partial virtualisation (hardware multiplexing)
  - Or real simultaneous by multiprocessing (hardware multiplication)
- Both techniques will induce computations to overlap in time and place.

Overlapping in time causes interference but is the lesser of two evils.

More critical is overlapping in place, relating to the same resource, particularly with regard to the same (i.e., shared) memory area.

Critical overlapping must be counteracted through **synchronisation**. i.e., coordination of the cooperation and competition between processes. Here: uni- or multilateral synchronisation, depending on the resource type.

Synchronisation ensures for **indivisibility** of a computation cycle:
- At the outset: physical, in blocking manner, by being pessimistic
- At the road’s end: logical, in non-blocking manner, by being optimistic
Résumé

- Computations can be **concurrent** if none needs a result of the other.

- In order to be concurrent, computations must be **simultaneous**.

- **Overlapping** in time cause interference but is the lesser of two evils.
  - More critical is overlapping **in place** relating to the same resource.
  - Particularly with regard to the same (i.e., shared) memory area.

© wosch, thoenig  
CS (WS 2020/21, LEC 2)  
Summary
Résumé

- Computations can be **concurrent** if none needs a result of the other.
- In order to be concurrent, computations must be **simultaneous**.

- **Overlapping** in time cause interference but is the lesser of two evils.

- Critical overlapping must be counteracted through **synchronisation**:
  - I.e., coordination of the cooperation and competition between processes.
  - Here: uni- or multilateral synchronisation, depending on the resource type.
computations can be **concurrent** if none needs a result of the other

in order to be concurrent, computations must be **simultaneous**

**overlapping** in time cause interference but is the lesser of two evils

critical overlapping must be counteracted through **synchronisation**

synchronisation ensures for **indivisibility** of a computation cycle
- at the outset: physical, in blocking manner, by being pessimistic 😞
- at the road’s end: logical, in non-blocking manner, by being optimistic 😊
[1] **Amdahl, G. M.**:
Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities.

[2] **Comstock, D. F.**:
The Principle of Relativity.
In: *Science 31* (1910), Mai 20, Nr. 803, S. 767–772.
http://dx.doi.org/10.1126/science.31.803.767. –
DOI 10.1126/science.31.803.767

[3] **Dijkstra, E. W.**:
Forschungsbericht. –

[4] **Gustafson, J. L.**:
Reevaluating Amdahl’s Law.
In: *Communications of the ACM 31* (1988), Mai, Nr. 5, S. 532–533
*On Deadlock in Computer Systems.*  
Ithaca, NY, USA, Cornell University, Diss., 1971

Some Deadlock Properties of Computer Systems.  
In: *ACM Computing Surveys* 4 (1972), Sept., Nr. 3, S. 179–196

*Webster’s New World Dictionary.*  
Simon & Schuster, Inc., 1988
bounded buffer using a counting semaphore \([3]\) for unilateral and an ELOP (x86) for multilateral synchronisation

```c
typedef int semaphore_t;
extern void P(semaphore_t*);
extern void V(semaphore_t*);
semaphore_t free = 80;
semaphore_t empty = 0;

static inline int fai(int *ref) {
    int aux = 1;
    asm volatile("lock; xaddl %0,%1"
                 : "=r" (aux), "=m" (*ref)
                 : "0" (aux), "m" (*ref));
    return aux;
}

char buffer[80];
unsigned in = 0, out = 0;
void put(char item) {
    P(&free);
    buffer[fai(&in) % 80] = item;
    V(&empty);
}

char get() {
    char item;
    P(&empty);
    item = buffer[fai(&out) % 80];
    V(&free);
    return item;
}
```

- **free** controls the number of unused buffer entries
  - \(P\) prevents from buffer overflow, \(V\) signals reusable resource
- **empty** controls the number of used buffer entries
  - \(P\) prevents from buffer underflow, \(V\) signals consumable resource
- **fai** indivisibly *fetch and increment* specified counter variable

© wosch, thoenig  CS (WS 2020/21, LEC 2)  Addendum – Synchronisation  28