Concurrent Systems

Nebenläufige Systeme

II. Concurrency

Wolfgang Schröder-Preikschat

October 16, 2014
Agenda

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
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
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.
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 concurrent to $e_i$
- is effect of $e_i$
- is cause 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 $e_i$:

  - is cause of $e_i$
  - is effect of $e_i$

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

- A 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 ($e_i$).

![Causal Chain Diagram]

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

non-existent timing restrictions; real-time processing

interrelation of computations/events constrains concurrency

"is cause of" ↦ → sequential (realized before/at run-time)

"is concurrent to" ↦ → parallel (realized in logical/real terms)

decrease of the portion of sequential code is an important aspect
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** $\sim$ *real-time processing*
Order of Precedence

- Computations can be carried out concurrently provided that:
  - General: None requires a result of the other (cf. p. 10)
  - 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 → real-time processing

- Interrelation of computations/events constrains concurrency

Event correlations v. Processing modes

- “Is cause of” \(\leadsto\) Sequential (realised before/at run-time)
- “Is effect of” \(\leadsto\) Parallel (realised in logical/real terms)
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 \( \sim \) real-time processing

interrelation of computations/events constrains concurrency

Event correlations v. Processing modes

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

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

decrease of the portion of sequential code is an important aspect
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

\[
su = \frac{rs + rp}{rs + rp/n}
\]

\(rs\) ratio of sequential code, independent of \(n\)

\(rp\) ratio of parallel code,

\(n\) number of processors

\(su\) speed-up will be constrained by data management housekeeping

The nature of this overhead appears to be sequential.
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

$$su = \frac{(rs + rp)}{(rs + \frac{rp}{n})}$$

$$= \frac{1}{rs + \frac{rp}{n}}$$

$rs$ ratio of sequential code

$rp$ ratio of parallel code, independent of $n$

$n$ number of processors

![Graph showing speedup vs. number of processors for different $rp$ values]
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

- speed-up will be constrained by **data management housekeeping**
- the nature of this overhead appears to be sequential
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

\[
\text{ssu} = \frac{r_s + r_p \times n}{r_s + r_p}
\]

\[
\begin{align*}
ssu & = r_s + r_p \times n \\
& = n + (1 - n) \times r_s
\end{align*}
\]

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

in practise, the problem size scales with the number of processors: \(\text{HPC}^1\)

\(^1\text{High Performance Computing}\)
Concurrent Operations of a Computation

operations can be concurrent if none needs the result of the other
Concurrent Operations of a Computation

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

```
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:

```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 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:

```
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 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
- as far as the logical dimension of a program is concerned
Concurrent Operations of a Computation

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

```java
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 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:
- 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 \(\rightsquigarrow\) level of abstraction
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 may result in a performance penalty.

- 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 an n²-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:
      - 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
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).
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
  - 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 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
  - 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: error

²Real-time processing, especially in case of hard deadlines.
Outline

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
Resource Classification

- **permanent, limited**
  - reusable
    - preemptable
    - non-preemptable

- **temporary, unlimited**
  - consumable

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".

© wosch  CS (WS 2014, LEC 2)  Resource Sharing – Principles
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 resource

reusable

preemptable non-preemptable

consumable
temporary, unlimited

3 Also referred to as "persistent".

permanent resources are reusable, but always only of limited supply, they are acquired, occupied, used, and released (when no longer required) when non-preemptable, they are exclusively owned by an occupant, i.e. produced, received, used, and destroyed (when no longer required)

in-use resources are preemtatable or non-preemptable, depending on whether allocation to another occupant is possible 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><strong>processor</strong></td>
<td><strong>signal</strong></td>
</tr>
<tr>
<td>CPU, FPU, GPU; MMU</td>
<td>IRQ, NMI, trap</td>
</tr>
<tr>
<td><strong>memory</strong></td>
<td></td>
</tr>
<tr>
<td>RAM, scratch pad, flash</td>
<td></td>
</tr>
<tr>
<td><strong>peripheral</strong></td>
<td></td>
</tr>
<tr>
<td>input, output, storage</td>
<td></td>
</tr>
</tbody>
</table>

Reusable data resources are notably containers for consumable resources. The availability of the former constrains production/consumption of the latter. Reusable and consumable resources imply different use patterns.
### 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>
Resource Peculiarities

- **hardware resources** as to be managed, e.g., by an operating system
  - **reusable**
    - processor: CPU, FPU, GPU; MMU
    - memory: RAM, scratch pad, flash
    - peripheral: input, output, storage
  - **consumable**
    - signal: IRQ, NMI, trap

- **software resources** as to be managed by any other program
  - **reusable**
    - code: critical section/region
    - data: variable, placeholder
  - **consumable**
    - signal: notice
    - message: packet, stream

- 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 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

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.
Resource Use Patterns

- 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

---

4At the same level of abstraction, use of a shareable resource is exclusive in the blocking case or never refused in the non-blocking case.
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
  - provided that the following two basic conditions (i.e., constraints) apply:
    1. resource accesses by computations may happen (quasi-) simultaneously
    2. 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

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

---

4 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.
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;
}

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

- which logical problems exist?

- 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:

```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**

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?

- Buffered items may be overwritten: overflow
- Values may be read from an empty buffer: underflow

Overlapping writes may go to the same memory location similar to overlapping reads, but reverse overlapping auto-increments may manifest wrong values.

Put and get must be subject to uni- and multilateral synchronisation they are not concurrent under the assumption that was made above.
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?

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];
2 unsigned in = 0, out = 0;

4 void put(char item) {
5     buffer[in++ % 80] = item;
6 }

8 char get() {
9     return buffer[out++ % 80];
10 }
```

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];
}
```

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

■ 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
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 logical problems exist?
  - buffered items may be overwritten: overflow
  - values may be read from an empty buffer: underflow

- 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

- put and get must be subject to uni- and multilateral synchronisation
  - they are not concurrent under the assumption that was made above
Simultaneous computations or operations, resp., are in competition:
- they compete for the **sharing** of the same reusable resource(s)
- they compete for the **handover** of the same consumable resource(s)
Serialisation of Simultaneous Computations

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

→ in either case hardware resources 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 (v.s. i)
- 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

The chosen synchronisation method should be **minimally invasive**
Serialisation of Simultaneous Computations

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

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

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

- simultaneous computations or operations, resp., are in competition:
  - they compete for the **sharing** of the same reusable resource(s)
  - they compete for the **handover** of the same consumable resource(s)
  - in either case hardware resources 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

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

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

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

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

- **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
Serialisation of Simultaneous Computations

- simultaneous computations or operations, resp., are in competition:
  - they compete for the **sharing** of the same reusable resource(s)
  - they compete for the **handover** of the same consumable resource(s)
  - in either case hardware resources 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

- **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:

- They compete for the **sharing** of the same reusable resource(s).
- They compete for the **handover** of the same consumable resource(s).

In either case, hardware resources and, if applicable, software resources too.

Both aspects, in turn, apply against the background of the following:

- The moment of a **simultaneous operation** is not predetermined.
- The operation in question is complex (i.e., consists of multiple steps).
- The characteristic of this operation is its **divisibility** in temporal respect.

**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.

**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.

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*)

---

© wosch  CS (WS 2014, LEC 2)  Resource Sharing – Competition  18 – 28
Divisibility in Temporal Respect

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

- when the steps of a complex operation may overlap at run-time
  - due to **simultaneous operation** (Ger. *Simultanbetrieb*)
- by way of example an auto-increment operator (cf. p. 16):
  - as compiled from C to ASM (x86): `gcc -O3 -m32 -static -S`

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

when the steps of a complex operation may overlap at run-time

- due to **simultaneous operation** (Ger. *Simultanbetrieb*)

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

- as compiled from C to ASM (x86): `gcc -03 -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++`
- simultaneous operations work on different variables

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


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

when the steps of a complex operation may overlap at run-time

- due to **simultaneous operation** (Ger. *Simultanbetrieb*)

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

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

\[
\begin{align*}
in++ & \\
1 & \text{ movl } _\text{in}, \%ecx \\
2 & \text{ leal } 1(\%ecx), \%eax \\
3 & \text{ movl } \%eax, _\text{in} \\
\end{align*}
\]

\[
\begin{align*}
out++ & \\
4 & \text{ movl } _\text{out}, \%ecx \\
5 & \text{ leal } 1(\%ecx), \%eax \\
6 & \text{ movl } \%eax, _\text{out} \\
\end{align*}
\]

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

**critical**
- self-overlapping execution of \text{in}++ or \text{out}++, resp.
- simultaneous operations work on the same variable\(^5\)

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

\(^5\)Assuming that processor registers are private to each computation.
Establishing of Synchronism

cf. p. 28

- 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**
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**
- indivisibility of a **cycle** is achieved through **synchronisation**,\(^6\) i.e.:
  1. coordination of the cooperation and competition between processes
  2. calibration of real-time clocks or data in distributed systems
  3. sequencing of events along the causal order

\(^6\)(Gr. *sýn*: synced, *chrónos*: time)
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**

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)
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**

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

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

**non-blocking**

  ■ ensure synchronism at **operation end**
  
  ■ allow potential overlapping, achieve consistency afterwards
  
  ■ synchronised operation is made of non-sequential code

---

\(^6\)(Gr. *sýn*: synced, *chrónos*: time)
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**

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

- 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
  - **non-blocking**
    - ensure synchronism at **operation end**
    - allow potential overlapping, achieve consistency afterwards
    - synchronised operation is made of non-sequential code

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

\(^6\)(Gr. *sýn*: synced, *chrónos*: time)
Varieties of Synchronisation

Relevant to Operating Systems

synchronisation

unilateral

sequential code

inhibiting

interruption
continuation
preemption

blocking

semaphore
(locking)

lock
condition variable
mutex

non-blocking

CAS
LL/SC
atomic R/W
TM

multilateral

non-sequential code

Casella

softwaring

bracketing 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 reprogramming sequential code as a transaction, ↪ →

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

© wosch CS (WS 2014, LEC 2) Resource Sharing – Synchronisation

20 – 28
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
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
Varieties of Synchronisation

Relevant to Operating Systems

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

**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 synchronization

© wosch CS (WS 2014, LEC 2) Resource Sharing – Synchronisation
Synchronisation Behaviour

- effect of synchronisation procedures on the computations involved
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, only
  - running computations are not delayed
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
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 and takes effect only in case of co-occurrence (overlapping)
  - applies to reusable and consumable resources, only 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, 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
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, 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...
Outline

Preface

Causality
  Interdependencies
  Dimensions

Resource Sharing
  Principles
  Competition
  Synchronisation

Summary
understanding (Ger.) *Gleichzeitigkeit* in its various meanings
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
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
Understanding (Ger.) \textit{Gleichzeitigkeit} in its various meanings:

- **Concurrency**
  - happening together in time and place \cite{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 \cite{7}
  - effect of a certain operation mode of a computing machine causes possibly critical overlapping of computations
  - includes concurrency, but not the other way round

- **Synchronism**
  - fact of being synchronous; simultaneous occurrence \cite{7}
  - in respect of the multiple sub-steps of a complex operation
  - achieved through “ELOP-ifying” coherent instructions
  - ensures that overlapped complex operations do right, the individual sub-steps will be strictly executed (consecutively) or a transaction will take care for consistent (pseudo-) parallel execution
understanding (Ger.) \textit{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

Simultaneity includes concurrency, but not the other way round

- concurrency implies unconstrained overlapping in time and place
- but simultaneity my also cause overlapping that must be constrained
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

**simultaneity** includes concurrency, but not the other way round

- concurrency implies unconstrained overlapping in time and place
- but simultaneity my 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 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]:

\[
\text{The simultaneity of two distant events means a different thing to two different observers if they are moving with respect to each other.}
\]

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

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

---

7Due to the fact that each one refers to an ELOP (cf. p. 19), logically.
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 **simultaneous**
- quasi-simultaneous through partial virtualization (hardware multiplexing)
  - or real simultaneous by multiprocessing (hardware multiplication)
- both techniques will induce computations to overlap in time and place

**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

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 😍
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 **simultaneous**
- quasi-simultaneous through partial virtualization (hardware multiplexing)
  - or real simultaneous by multiprocessing (hardware multiplication)
- both techniques will induce computations to overlap in time and place

**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

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

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

- Quasi-simultaneous through partial virtualization (hardware multiplexing)
- Or real simultaneous by multiprocessing (hardware multiplication)
- Both techniques will induce computations to overlap in time and place

**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

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
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 **simultaneous**
- quasi-simultaneous through partial virtualization (hardware multiplexing) or real simultaneous by multiprocessing (hardware multiplication)
- both techniques will induce computations to overlap in time and place

**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

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
- in order to be concurrent, computations must be **simultaneous**
  - quasi-simultaneous through partial virtualization (hardware multiplexing)
    - or real simultaneous by multiprocessing (hardware multiplication)
  - both techniques will induce computations to overlap in time and place
- **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
- 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
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 **simultaneous**
- quasi-simultaneous through partial virtualization (hardware multiplexing)
  or real simultaneous by multiprocessing (hardware multiplication)
- both techniques will induce computations to overlap in time and place

**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

**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 😊

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

Forschungsbericht. –  

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
**Consolidating Example Revisited**

**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;
}
```

- **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