Concurrent Systems

Nebenläufige Systeme

II. Concurrency

Wolfgang Schröder-Preikschat

November 12, 2020

Agenda

Preface

Causality
- Interdependencies
- Dimensions

Resource Sharing
- Principles
- Competition
- Synchronisation

Summary

Outline

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)

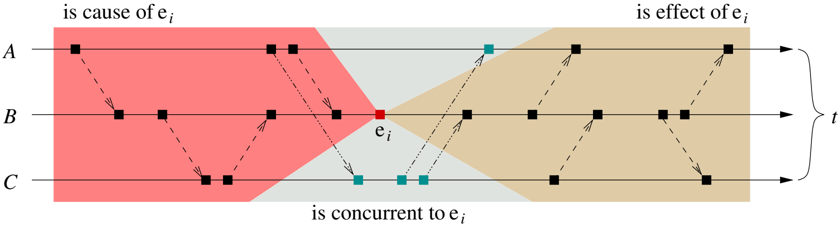
Events occur or are concurrent if none is the cause of the other.

- 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

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)

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

$$\left.\begin{array}{l}\text{“is cause of”}\\ \text{“is effect of”}\end{array}\right\} \mapsto \text{sequential (realised before/at run-time)}$$

$$\text{“is concurrent to”} \mapsto \text{parallel (realised in logical/real terms)}$$

$\hookrightarrow$ decrease of the portion of **sequential code** is an important aspect

Limits in the Degree of Concurrency

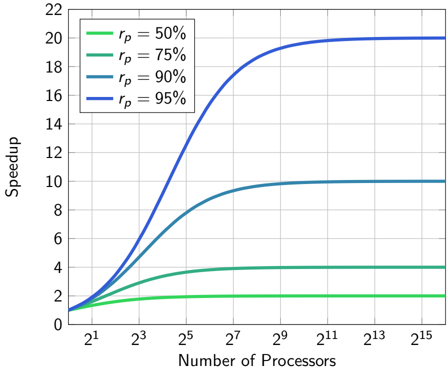
- Amdahl's Law [1]: speed-up (*su*) achievable by parallel processors
  - work load remains constant with the varying number of processors
  - aim at reducing overall computation time for a given fixed-size problem

$$su = (r_s + r_p)/(r_s + \frac{r_p}{n}) = \frac{1}{r_s + \frac{r_p}{n}}$$

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

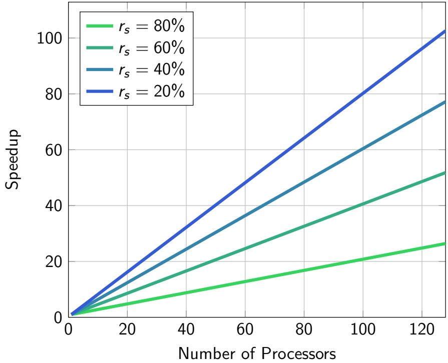
Adapting the Work Load

- 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

$$
\begin{aligned}
ssu &= \frac{r_s + r_p \times n}{r_s + r_p} \\
&= r_s + r_p \times n \\
&= n + (1 - n) \times r_s
\end{aligned}
$$

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: **HPC**[1]

[1] High Performance Computing

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

- 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
  - 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)

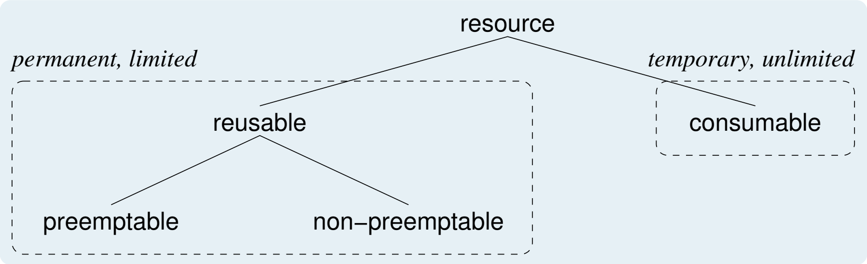
↪ reveals a *race condition*, substantial critical situation: **error**

[2] real-time processing, especially in case of hard deadlines.

Outline

Resource Classification

analogically with [5, 6]



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

[3] Also referred to as "persistent".

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
- reusable and consumable resources imply different **use patterns**

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[4]
- 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

- which logical problems exist?
  - buffered items may be overwritten: overflow
  - values my 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
- `put` and `get` must be subject to uni- <u>and</u> multilateral synchronisation
  - they are <u>not</u> 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)

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

`in++`

```
movl _in, %ecx
leal 1(%ecx), %eax
movl %eax, _in
```

`out++`

```
movl _out, %ecx
leal 1(%ecx), %eax
movl %eax, _out
```

non-critical
- overlapping execution of `in++` and `out++`
- simultaneous operations work on different variables[5]

critical
- self-overlapping execution of `in++` or `out++`, resp.
- simultaneous operations work on the same variable[5]

- 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**

[5] Assuming that processor registers are private to each computation.

Establishing of Synchronism

cf. p. 28

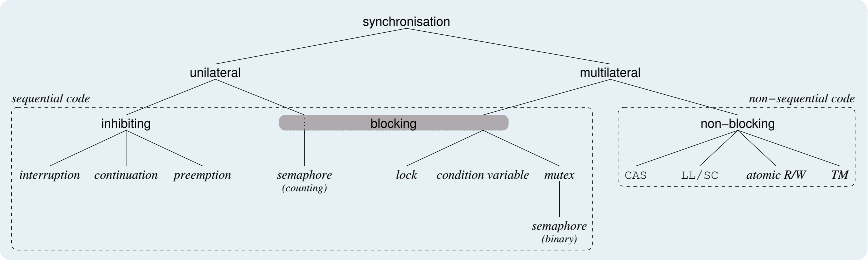
- assure a conflict-prone complex operation of (logical) **indivisibility**
  - interpret the equivalent computation as **elementary operation** (ELOP)
    - an operation of indivisible cycle (Ger. *zeitlicher Ablauf*), apparently **atomic**
- indivisibility of a *cycle* is achieved through **synchronisation**,[6] 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



- 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 <u>not</u> 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 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. . .

Outline

Abstract Concepts Revisited

- 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 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

Relativity of Simultaneity

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

[7] Due to the fact that each one refers to an ELOP (cf. p. 19), logically.

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 virtualisation (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 ☺

Reference List I

[1] Amdahl, G. M.:
Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities.
In: *Proceedings of the AFIPS Spring Joint Computer Conference (AFIPS 1967 (Spring)*, AFIPS Press, 1967, S. 483–485

[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.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[4] Gustafson, J. L.:
Reevaluating Amdahl's Law.
In: *Communications of the ACM* 31 (1988), Mai, Nr. 5, S. 532–533

Reference List II

[5] HOLT, R. C.:
On Deadlock in Computer Systems.
Ithaca, NY, USA, Cornell University, Diss., 1971

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

[7] NEUFELDT, V. (Hrsg.) ; GURALNIK, D. A. (Hrsg.):
Webster's New World Dictionary.
Simon & Schuster, Inc., 1988

Consolidating Example Revisited

cf. p. 16

- **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