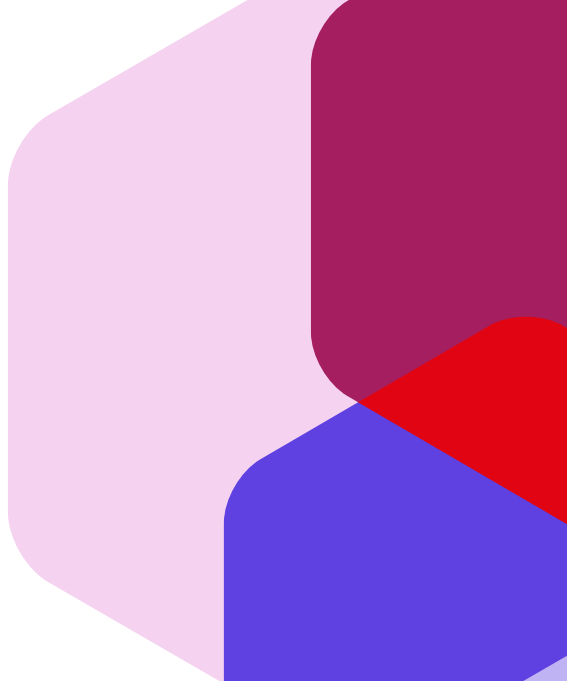


Principles, Algorithms and Models of Concurrency and Synchronization

01 Introduction



What do you prefer?

Prof. Dr. David Sabel

- Room C 031
- david.sabel@hs-rm.de

Resources: StudIP → ILIAS

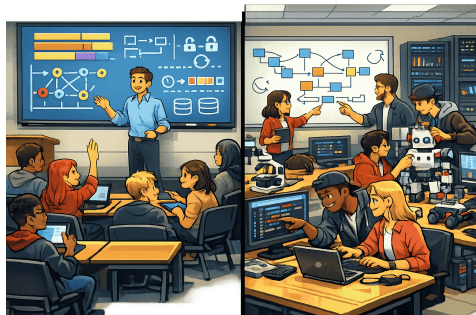
- lecture notes
- slides
- recordings
- exercises
- additional stuff

Lecture

- Tuesday 10:00 - 11:30

Practical part

- Tuesday 11:45 - 13:15



Exam: Practice (Praktische Tätigkeit)

Mandatory parts (both must be accepted to pass the exam):

- ① Group Research Task: Research and give a 15-minute talk on concurrency / synchronization / communication in your favourite programming language
- ② Group Project: Implement a software project that uses concurrency and synchronization using the concepts of the lecture.

Grades:

- (individual) grade is given for the group research task
- a bonus of up to 1 may be applied (after passing the exam)

Bonus:

- weekly exercise sheets: solved as homework, discussed in the practical part. For each (serious) presentation of your solution: bonus of 0.3 points
- group research task: may give a bonus between 0.0 and 0.7
- Final bonus: $\min(1, \text{sum of obtained bonuses})$

Registration for the Exam

- Registration via COMPASS
- Early Deadline: 11.05.2026

Introduction: Terms, notations, assumptions

Synchronization of processes and **mutual exclusion**

- Mutual exclusion under various assumptions of memory operations
- Complexity results: How many memory objects are needed, etc.
- Stronger memory objects: Test-and-set bits, RMW objects, CAS objects, ...
- Consensus and the consensus number

Programming primitives for concurrent programming:

- Semaphores, monitors, channels
- Solution of classic problems with the primitives

Concurrent access to multiple resources:

- Deadlocks, Deadlock avoidance (two-phase locking protocol), Deadlock prevention (banker's algorithm), Transactional memory

The π -calculus as a message-passing model

On mutual exclusion algorithms and constructs of concurrent programming

Ben-Ari, M. (2006). *Principles of concurrent and distributed programming*. Addison-Wesley.

Taubenfeld, G. (2006). *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall.

Reppy, J. H. (2007). *Concurrent Programming in ML*. Cambridge University Press.

Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.

Raynal, M. (2013). *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer.

On the π -calculus:

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press.

Sangiorgi, D. & Walker, D. (2001). *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

Why Concurrent Programming Matters?

Difference: sequential – concurrent

Areas of application for concurrent programming

Difficulties in concurrent programming



Sequential programs:

- Sequence of (machine) commands
- Commands are executed one after the other
- Each command changes main memory, registers, etc.
- Execution **deterministic**

Concurrent (parallel) programs:

- Multiple commands are executed **simultaneously**.
- Simultaneous can also mean **quasi-parallel**, i.e., sequential in reality.
- Execution is generally **non-deterministic**.
- More formal definition: later.

Real systems are almost never purely sequential

Example: operating systems:

- From the user's perspective: different tasks are performed simultaneously.
- E.g., printing, surfing, moving the mouse, listening to music (all at the same time).
- Implementation: unthinkable without concurrent programming.

Real systems are almost never purely sequential (2)

Example: web programming:

- Client-server model: client requests service from server
- e.g. WWW server: client requests web page
- Necessary: server serves multiple clients simultaneously
- Crash of one client does not cause other clients to hang

Concurrent programming is needed

Even on single-processor systems:

- Concurrency sometimes allows resources to be used more efficiently
- Example: 2 tasks
 - Long computation
 - Write large file
- Writing files does not burden the CPU (speed is determined by the hard disk/bus)
- Concurrency: If the hard disk is busy, then compute.
- Concurrent programming as a structuring principle

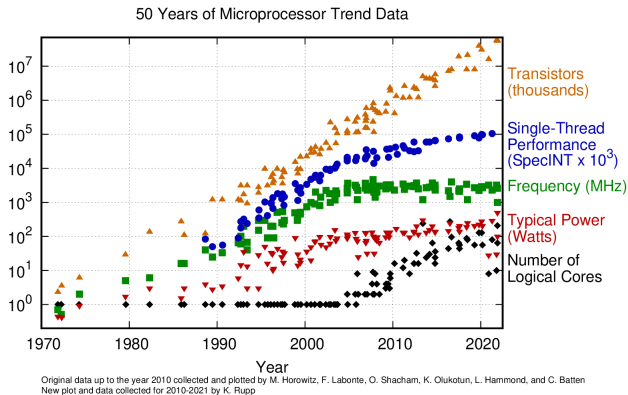
Concurrent programming is needed (2)

Stubborn problems and large problem instances:

- Computing through massive parallelization / distributed computing
- High-performance computers, “GRID computing”

Concurrent programming is needed (3)

Hardware development:



Source: <https://github.com/karlrupp/microprocessor-trend-data>

- Clock frequencies at the limits of their physical capabilities
- Instead of increasing clock speeds, increasing the number of processors
- Multicore architectures
- Architecture requires adapted (parallelized) programs
- Article by Herb Sutter: Multiprocessor systems require a paradigm shift for programming languages

What makes parallel programming so difficult?

- Parallelization of sequential algorithms is often not obvious.
- Main problem: The programs running in parallel must work together correctly.
- Exchange data (**communication**)
- Wait for each other (**synchronization**)

Example: Account Management

(The value of `account` before execution is 100)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$

Example: Account Management (2)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$

Execution (P1),(P2),(Q1),(Q2)
 $\text{account} = 100$

Example: Account Management (2)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

Execution (P1),(P2),(Q1),(Q2)

$\text{account} = 100$

Example: Account Management (2)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

Execution (P1), (P2), (Q1), (Q2)

$\text{account} = 90$

Example: Account Management (2)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$
 $X = 90$

Execution (P1),(P2),(Q1),(Q2)
 $\text{account} = 90$

Example: Account Management (2)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$
 $X = 90$

Execution (P1),(P2),(Q1),(Q2)
 $\text{account} = 100$

Example: Account Management (3)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$

Execution (Q1),(P1),(P2),(Q2)
 $\text{account} = 100$

Example: Account Management (3)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$
 $X = 100$

Execution (Q1),(P1),(P2),(Q2)
 $\text{account} = 100$

Example: Account Management (3)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1), (P1), (P2), (Q2)

$\text{account} = 100$

Example: Account Management (3)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1),(P1),(P2),(Q2)

$\text{account} = 90$

Example: Account Management (3)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1),(P1),(P2),(Q2)

$\text{account} = 110$

Example: Account management (4)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$

Execution (Q1),(P1),(Q2),(P2)
 $\text{account} = 100$

Example: Account management (4)

process P :

(P1) $X := \text{account};$
(P2) $\text{account} := X - 10;$

process Q :

(Q1) $X := \text{account};$
(Q2) $\text{account} := X + 10;$
 $X = 100$

Execution (Q1), (P1), (Q2), (P2)
 $\text{account} = 100$

Example: Account management (4)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1), (P1), (Q2), (P2)

$\text{account} = 100$

Example: Account management (4)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1),(P1),(Q2),(P2)

$\text{account} = 110$

Example: Account management (4)

process P :

(P1) $X := \text{account};$

(P2) $\text{account} := X - 10;$

$X = 100$

process Q :

(Q1) $X := \text{account};$

(Q2) $\text{account} := X + 10;$

$X = 100$

Execution (Q1),(P1),(Q2),(P2)

$\text{account} = 90$

Example: Account Management (5)

Analysis shows:

Order	Value of account afterwards
(P1),(P2),(Q1),(Q2)	100
(Q1),(Q2),(P1),(P2)	100
(P1),(Q1),(P2),(Q2)	110
(Q1),(P1),(P2),(Q2)	110
(P1),(Q1),(Q2),(P2)	90
(Q1),(P1),(Q2),(P2)	90

Conclusion:

- Incorrectly programmed.
- Traditional debugging does not work.
- Results may vary from execution to execution.

Assumption: There are processes generating data and processes consuming data

Different variants:

- Multiple consumers and one producer
- Multiple producers and one consumer
- Multiple producers and multiple consumers

Consumer-Producer Problems

Assumption: There are processes generating data and processes consuming data

Different variants:

- Multiple consumers and one producer
- Multiple producers and one consumer
- Multiple producers and multiple consumers

Wanted:

Data structures to ensure **safe** exchange of data between consumers and producers.

Therefore: Concurrent programming requires new data structures.

Terms of Concurrent Programming

Parallelism and concurrency

Parallel program:

- Calculation on multiple processors
- Simultaneous, overlapping

Concurrent programming:

- Execution on one or multiple processors
- Potentially, **all** execution sequences are possible.

Concurrent and distributed systems

Distributed system:

- System consisting of multiple processors (often geographically separated)
- No shared memory
- Data exchange: Only via sending and receiving messages.

Concurrent system:

- Calculation on one or more processors
- Shared memory possible

Processes and threads

- **process** is used in theory (e.g., in π -calculus)
- often distinguished based on control:
 - operating system handles **processes**
 - programs administer **threads**
 - where a program is an operating system process

Processes and threads

- **process** is used in theory (e.g., in π -calculus)
- often distinguished based on control:
 - operating system handles **processes**
 - programs administer **threads**
 - where a program is an operating system process
- we do not strictly separate the two terms.

Processes and threads

- **process** is used in theory (e.g., in π -calculus)
- often distinguished based on control:
 - operating system handles **processes**
 - programs administer **threads**
 - where a program is an operating system process
- we do not strictly separate the two terms.
- **multi-threading** = feature of programming languages: constructs for administration of threads

Processes and threads

- **process** is used in theory (e.g., in π -calculus)
- often distinguished based on control:
 - operating system handles **processes**
 - programs administer **threads**
 - where a program is an operating system process
- we do not strictly separate the two terms.
- **multi-threading** = feature of programming languages: constructs for administration of threads
- related: **multi-tasking** = ability to perform multiple tasks quasi-parallel

Message-passing and shared-memory models

Message-passing model:

- processes do not have shared memory
- communication occurs exclusively via sending and receiving messages
- suitable for distributed systems

Shared memory model:

- processes use shared memory
- communication takes place directly via the shared memory

Synchronous and asynchronous communication

synchronous:

- communication between sender and receiver takes place 'without time'
- example: telephone

asynchronous:

- sending and receiving do not have to take place simultaneously.
- often more natural.
- example: writing letters or emails is asynchronous.

Model Assumptions

Concurrent Programs: Pseudo-Code, Operational Behavior: Interleaving Assumption, Fairness

- **concurrent program:** *finite set of processes*
 - **process:** *sequential program consisting of atomic computation steps*
- assumption for all our models
 - definition of **computation step** depends on the model!

Example:

program with 3 processes

process P

(P1) $X := 10;$

(P2) while $X > 0$ do

(P3) $X := X + 1;$

process Q

(Q1) $Y := 3;$

(Q2) while $Y > 0$ do

(Q3) $X := Y - 1;$

(Q4) $Y := X;$

process R

(R1) $X := 0;$

(R2) $Y := 0;$

Interleaving assumption

Execution of a concurrent program:

*Sequence of atomic computation steps of the processes,
which **can be arbitrarily interleaved**.*

- Only one computational step **at a time**, i.e., two steps **never overlap**
- (Almost) arbitrary interleaving (**interleaving**) of computation steps of the individual processes
- (Almost) all execution sequences allowed
- “Almost” will be explained later

Example

program with 3 processes		
process P	process Q	process R
(P1) $X := 10;$	(Q1) $Y := 3;$	(R1) $X := 0;$
(P2) while $X > 0$ do	(Q2) while $Y > 0$ do	(R2) $Y := 0;$
(P3) $X := X + 1;$	(Q3) $X := Y - 1;$	
	(Q4) $Y := X;$	

a possible execution:

Step	Process	Operation performed	Memory
			$X \mapsto 0, Y \mapsto 0$
1	P1	$X := 10$	$X \mapsto 10, Y \mapsto 0$
2	P2	while $X > 0$?	$X \mapsto 10, Y \mapsto 0$
3	R1	$X := 0$	$X \mapsto 0, Y \mapsto 0$
4	P3	$X := \underline{X + 1}$	$X \mapsto 0, Y \mapsto 0$
5	P3	$X := 1$	$X \mapsto 1, Y \mapsto 0$
6	Q1	$Y := 3$	$X \mapsto 1, Y \mapsto 3$

...

Notation often simplified: $(P1), (P2), (R1), (P3), (P3), (Q1), \dots$

Why is the Interleaving Assumption Useful?

granularity

- computation steps can be defined arbitrarily (i.e., arbitrarily small!)

Why is the Interleaving Assumption Useful?

granularity

- computation steps can be defined arbitrarily (i.e., arbitrarily small!)

multitasking systems

- single-processor systems with multitasking perform interleaving
- real implementations usually use techniques (e.g., time slices) that do not admit every sequence
- “process executes steps for a period of time, then switches.”
- but: ‘for a period of time’ is problematic:
 - difficult to define formally, complicated
 - new (faster) hardware requires model adaptation
 - consideration of environmental influences (e.g., phase fluctuations)
- all sequences also include the smaller set of real sequences
(correctness for all implies correctness for a part)

Why is the Interleaving Assumption Useful? (2)

multiprocessor systems

- assumption seems unrealistic, since parallel / overlapping computation steps are possible
- **but:** shared resources (e.g., memory) are protected from parallel access at the hardware level and enforce sequentialization
- resource-autonomous steps are still parallel
- but this is not visible!

- earlier: ... **almost** arbitrary interleaving ...
- the restriction comes from the fairness assumption
- assumptions of fairness vary in strength
- our concept is relatively weak

Fairness assumption

Every process for which a computation step is possible performs this step in the overall evaluation sequence after a finite number of computation steps.

Example of Fairness

(X initially 0)

process P :

(P1) if $X \neq 10$ then goto (P1);

process Q :

(Q1) $X := 10$;

Evaluation sequence that does not observe fairness:

$(P1), (P1), (P1), (P1), (P1), (P1), \dots$ and so on infinitely

Note: The fairness assumption only makes a statement about
infinite evaluations

Evaluation sequences taking fairness into account

$\underbrace{(P1), (P1), \dots, (P1)}_{n\text{-times}} (Q1) (P1)$ for any $n \in \mathbb{N}_0$

Example of Fairness (2)

(X starts at 0)

process P :

(P1) if $X \neq 10$ then goto (P1);

process Q :

(Q1) $X := 10$;

Conclusion:

Under the fairness assumption, the following applies:

The above program **always terminates**

(Since (Q1) **must** be executed at some point)

Another Example of Fairness

(The value of X before execution is 0)

process P :

(P1) if $X \neq 10$ then goto (P1);
(P2) goto (P1)

process Q :

(Q1) $X:=10$;

Permitted sequences

$\underbrace{(P1), (P1), \dots, (P1)}_{n\text{-times}}, (Q1), \underbrace{(P1), (P2), (P1), (P2), \dots}_{\text{infinitely often}}$ with $n \in \mathbb{N}_0$

Prohibited sequences

$(P1), (P1), (P1), \dots$ infinitely long

Known processes

- a program consists of a **fixed number** of processes
- no new processes are created by the program
- processes are identifiable

Note:

We might not keep this assumption for the entire lecture

Programming language

- we assume an imperative small pseudo-programming language
- commands: `if-then`, `goto`, assignments `$X := 10$` , `while` loops, etc.
- each numbered line is executed **atomically**
- we will define permitted operations on memory