Einführung in die Informatik 2
Sommersemester 2006

Prof. Dr. Alois Knoll

TU München

Lehrstuhl VI Robotics and Embedded Systems

Informatik 2: Organisation

Bestandteile der Vorlesung

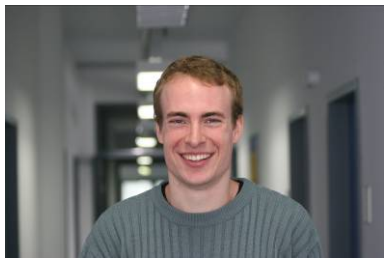
- Vorlesung:
  - Freitag 11:30-13:00 MW 0001
  - 4 ECTS Punkte
- Übung:
  - zweistündige Tutorübung jede zweite Woche
  - Beginn: 16./17.05.06
  - Die Anmeldung erfolgt über die Grundstudiumsseite https://grundstudium.in.tum.de
  - Wiederholer für das Praktikum melden sich bitte auch über die Grundstudiumsseite an.
- Prüfung:
  - Schriftliche Klausur am Ende des Sommersemesters.

Team

Prof. Dr. Alois Knoll

Dr. Markus Schneider

Übungsleitung



Dipl.-Inf. Christian Buckl

Technik, Übungsleitung

Weitere Informationen

- Homepage der Vorlesung:

  http://www6.in.tum.de/tum6/lectures/courses/ss06/info2
- Beim jeweiligen Tutor
- Übungsleitung
  - Email: schneima@in.tum.de, buckl@in.tum.de
- Newsgroup:
  - tum.info.info12

Informatik 2: Vorlesungsstruktur und -inhalt

Vorlesungsstruktur und -inhalt I

- Information und Codierung
  - Zum Begriff der Information
    - Repräsentation von Information
    - Interpretation von Repräsentation
  - Shannonsche Informationstheorie
    - Grundlagen: Codierung, Entropie
    - Codierungsverfahren: Huffman, LZW
  - Kryptologie
    - Historie, Konzepte
    - Zahlentheoretische Techniken, z.B. RSA-Algorithmus
    - Anwendungen: ssh, pgp

Vorlesungsstruktur und -inhalt II

- Grundlegende Konzepte des nebenläufigen Programmierens
  - Schwer- und leichtgewichtige Prozesse, Prozesszustände und Operationen auf Prozessen
  - Aktionen und Ereignisse
  - Praktische Umsetzung der Konzepte
  - Spezifikation des zeitlichen Verhaltens und Einplanungen: Synchrone und asynchrone Ablaufmodelle
- Synchronisation und Kommunikation
  - Semaphore, Monitore
  - Erzeuger-Verbraucher, Leser-Schreiber Probleme
  - Synchrone/asynchrone Kommunikation
  - Rendezvous als Beispiel für aktionsorientierte Kommunikation

Zum Begriff der Information

Zum Begriff der Information

- Informatik ist die Wissenschaft der **systematischen Verarbeitung von Information**
- Zunächst zu definierende Begriffe:
  - **Signal**: Elementar feststellbare Veränderung (Lichtblitz, Tonhöhe,…) eines physikalischen Parameters
  - **Datum**: Signal, dargestellt durch digitale Zeichen (siehe DIN **ISO/IEC 2382** Text und Daten)
  - **Nachricht**: Folge von Signalen bestimmter (physikalischer) Darstellung (oder *Repräsentation*) einschließlich zeit-räumlicher Anordnung zur Übertragung von Information
  - **Information**: Einer vom Empfänger einer Nachricht zugeordnete Bedeutung (die sein *Wissen* erweitert)

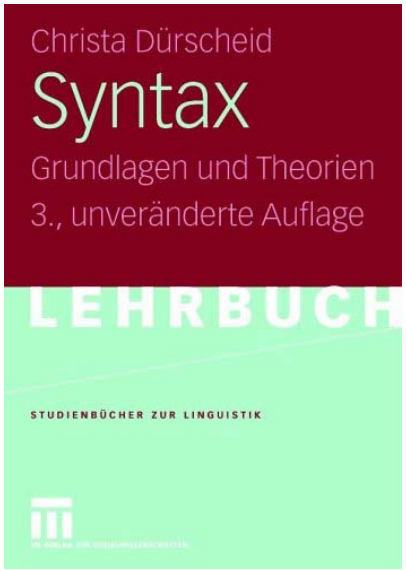
Interpretation von Daten / Nachrichten

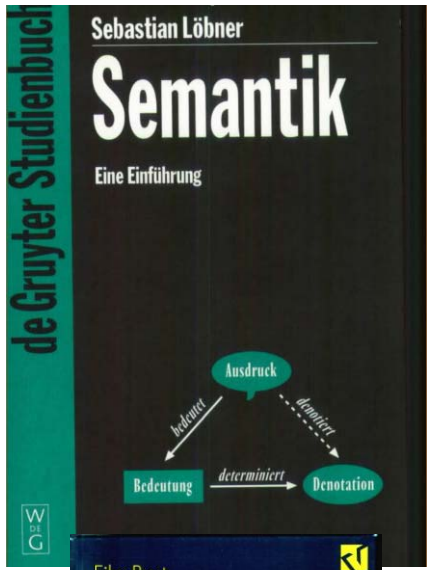
- Eine Nachricht wird durch eine **Interpretationsvorschrift** zur Information.
- Ein **Datum** ist eine konkrete Information zusammen mit ihrer Repräsentation.
- Es gibt keine (abstrakte) Information ohne (reale) Nachrichten/ Repräsentation.
- Information bedarf der Repräsentation, sonst kann sie nicht gespeichert oder verarbeitet werden.
- Dieselbe Information kann man durch verschiedene Nachrichten darstellen.
- Dieselbe Nachricht kann verschiedene Informationen vermitteln.

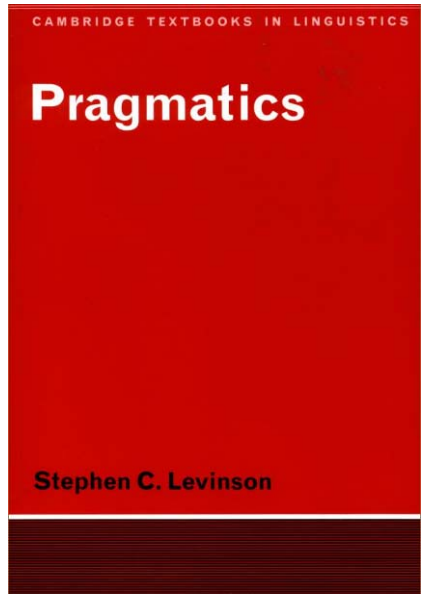
Weitere zentrale Begriffe

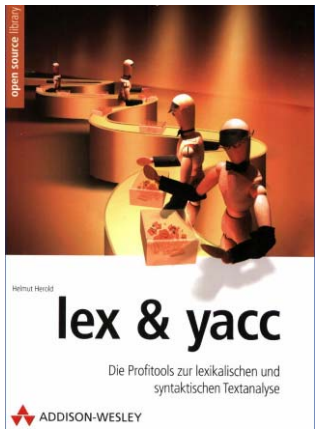
- **Codierung**: Umwandlung der Repräsentation einer Nachricht.
  - Beispiel: Strichzeichen II oder Wort „zwei“ steht für Information: Zahl 2.
- **Informationsstruktur**: (***I****: Repr* → *Info*); die Interpretationsvorschrift ***I*** ordnet jeder Repräsentation *Repr* eine Information *Info* zu. Das Paar (*I*, *Info*) wird als **Semantik** von *Repr* bezeichnet. Interpretationsvorschriften sind ihrerseits Nachrichten!
  - Beispiel: Geometrische Figuren VII werden durch Interpretationsvorschrift ***I*** (**"**VII **"**) „römische Zahl“ zur Zahl 7. Also I „römische Zahl“: VII → 7
- Demgegenüber ist die **Syntax** einer Information die Vorschrift, wie die einzelnen Bausteine zusammengesetzt werden dürfen.
- Die **Pragmatik** beschreibt, welchen Zweck die Information hat bzw. welche Handlungen sich aus der Nachricht ergeben (sollen).
- **Semiotik** (Lehre von den Zeichen und Zeichenprozessen) ist Syntax + Semantik + Pragmatik.

Literatur


Christa Dürscheid
Syntax
Grundlagen und Theorien
3., unveränderte Auflage
LEHRBUCH
STUDIENBÜCHER ZUR LINGUISTIK


Sebastian Löbner
Semantik
Eine Einführung
de Gruyter Studienbuch
Ausdruck
bedeutet
denotiert
Bedeutung
determiniert
Denotation


CAMBRIDGE TEXTBOOKS IN LINGUISTICS
Pragmatics
Stephen C. Levinson


open source library
Helmut Herold
lex & yacc
Die Profitools zur lexikalischen und syntaktischen Textanalyse
ADDISON-WESLEY


Alfred V. Aho/Ravi Sethi/
Jeffrey D. Ullmann
Compiler-
bau Teil 1
2. Auflage
Oldenbourg


Eike Best
Semantik
Theorie sequentieller und
paralleler Programmierung
vieweg
Lehrbuch Informatik

Datenverarbeitung

- **Datenverarbeitung**: **Bedeutungstreue** Manipulation oder **Umcodierung** von Repräsentationen, d.h. Manipulation muss in Übereinstimmung mit der Bedeutung vorgenommen werden.
  - Beispiel 1: Nachricht „**hallo du**“. Interpretation als Menge von Zeichen.
- Unter dieser Interpretationsvorschrift trägt die Verarbeitung zu einer lexikographisch geordnete Zeichenfolge „ **adhllou**“ die gleiche Information. Unter den Konventionen/Interpretationsvorschriften der deutschen Sprache ist eine solche Verarbeitung nicht bedeutungstreu.
  - Beispiel 2: Die bedeutungstreue Umcodierung von Temperaturangaben. ***C***(*f*) = 5/9*(*f*-32). *f* ist die nach der Messvorschrift „Fahrenheit“ gemessene Temperatur, ***C***(f) die Umcodierung auf die Temperaturskala „Celsius“. 100 ^{O}C entspricht 212 F und beide tragen die Information „kochendes Wasser“.
- **Problem**: Wie kann man die Interpretationsvorschriften formalisieren, wie kann man eindeutig beschreiben, was sie tun (etwa im Kopf des Empfängers?)
- Wesentlich: Konventionen – die Interpretation ***I*** kommt letztlich dadurch zustande, dass sich alle Mitglieder einer Gesellschaft darauf geeinigt haben. Einfaches Beispiel: Rotes Licht (Ampel) bedeutet für Westeuropäer: Straßenrichtung gesperrt

Information in der Informatik

- In der Informatik hauptsächlich interessant: Semantik von Programmiersprachen bzw. den mit ihnen zu formenden Ausdrücken.
- Hier kann man nicht der Interpretation jedes Ausdrucks eine langwierige Diskussion im sozialen Umfeld vorangehen lassen. Außerdem muss man sich darauf verlassen können, daß alle unter einem Ausdruck dasselbe verstehen, also dieselbe Wirkung erwarten. Deshalb:
- **Formale** Semantikdefinition einer Informationsstruktur (***I****: Repr* → *Info*) wird vorgenommen durch Rückführung auf eine als bekannt vorausgesetzte Struktur (***I***0*: Repr0* → *Inf*).
  - Beispiel:
    I0: Strichzeichen **"IIIIIIIIIIIIIII"** wird als Zahl 15 interpretiert und dies als allgemeinverbindlich angesehen.
    I: Für die Betrachtung anderer Zahlenformate genügt es dann, die entsprechende Umcodierung angeben zu können: ***C***("01111") = **"IIIIIIIIIIIIIII"**

Möglichkeiten zur Beschreibung von Semantik I

- Es gibt vier wichtige Ansätze zur Beschreibung der Semantik von Programmiersprachen: Übersetzersemantik, Operationale Semantik, Denotationale Semantik, Axiomatische Semantik
- **Übersetzersemantik**: Bedeutung von Konstrukten einer Programmiersprache **N** wird auf die Bedeutung der Konstrukte einer bekannten (einfacheren, d.h. ausdrucksschwächeren) Programmiersprache $\mathbf{N}_0$ zurückgeführt. Dazu gibt man einen Übersetzer $\mathbf{N} \rightarrow \mathbf{N}_0$ an. Problem wird allerdings in gewisser Weise nur verlagert (Semantik von $\mathbf{N}_0$ muss nach wie vor definiert werden).
- Beispiel: Umsetzung eines Schleifenkonstrukts in "C" in einen "LOOP"-Ausdruck des Prozessors 8086. Wie wird dieses definiert?

```
for (initial_value;
     test; increment) {}
```

→

LOOP - Decrement CX and Loop if CX Not Zero

Usage: LOOP label
Modifies flags: None

Decrements CX by 1 and transfers control to "label" if CX is not Zero. The "label" operand must be within -128 or 127 bytes of the instruction following the loop instruction

Operands		Clocks 808x	286	386	486	Size Bytes
label:	jump	18	8+m	11+m	6	2
	no jump	5	4	?	2	

Möglichkeiten zur Beschreibung von Semantik II

- **Operationale Semantik**: Angabe eines Interpreters für **N**, der aus Eingabedaten durch schrittweise Abarbeitung von Programmen Ausgabedaten erzeugt.
- Vorstellung: Interpreter "läuft" auf abstrakter Maschine; operationale Semantik zeigt, wie diese Maschine auf Programmkonstrukte in **N** reagiert und wie sich dabei der Speicher verändert.
- Abstrakte Maschine wird typischerweise elementar definiert mit arithmetischen Ausdrücken, Zuweisung, bedingter Verzweigung.

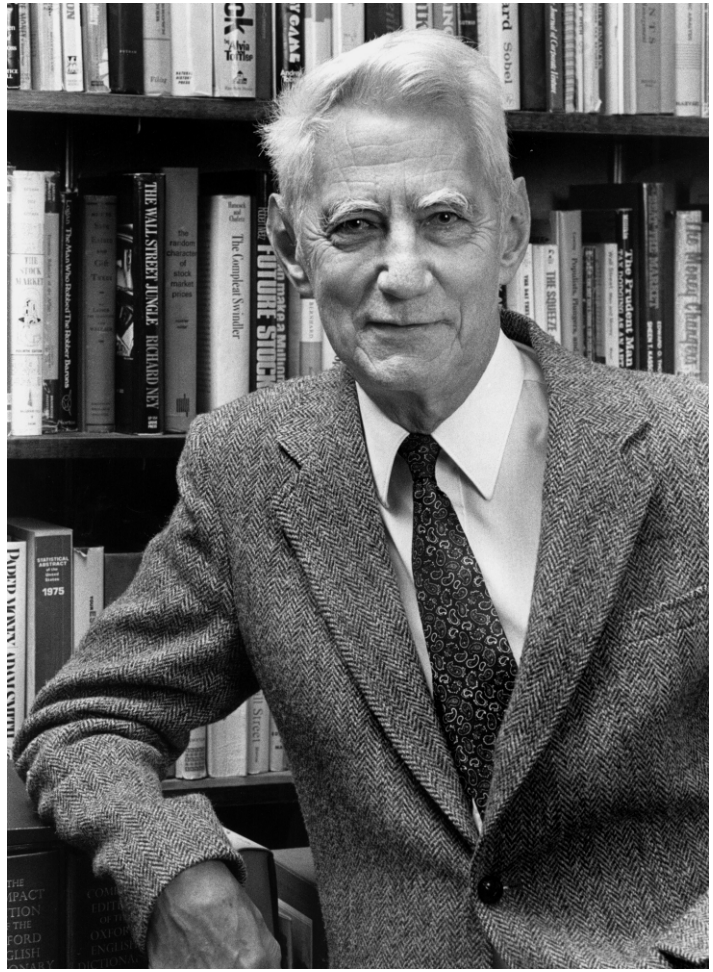
Möglichkeiten zur Beschreibung von Semantik III

- **Denotationale Semantik**: Beschreibung der *Wirkung*, die Anweisungen auf Zustände (= Variablenbelegungen) haben.
- Sei A_0 eine Anweisung, *z* ein Zustand aus der Menge aller möglichen Zustände *Z* und *z'* der auf *z* nach Ausführung der Anweisung folgende Zustand.
- Dann ist die *Wirkung* eine Abbildung $Z \rightarrow Z$ der Form: **F** $[A_0]$: $z \rightarrow z'$ (semantische Funktion).
- Beispiel: gegeben zwei Variablen *m* und *n* mit *m* = 5 und *n* = 6. Dann ist die Wirkung der Anweisung **m := n + 1** wie folgt: **F** [**m:= n + 1**] (5,6) = (7,6).

Möglichkeiten zur Beschreibung von Semantik

- **Axiomatische Semantik**: Angegeben werden die Eigenschaften von Zuständen vor und nach der Ausführung einer Aktion als Prädikate („Zusicherungen“).
- Form: {**P**} A_0 {**Q**}. **P** ist die Vorbedingung, **Q** die Nachbedingung, A_0 (Folge von) Anweisung(en).
- Beispiel:
- { **x=n** } y:=1; while x≠1 do (y:=y*x; x:=x-1) { **y=n!** ∧ **n>0** }

Shannonsche Informationstheorie

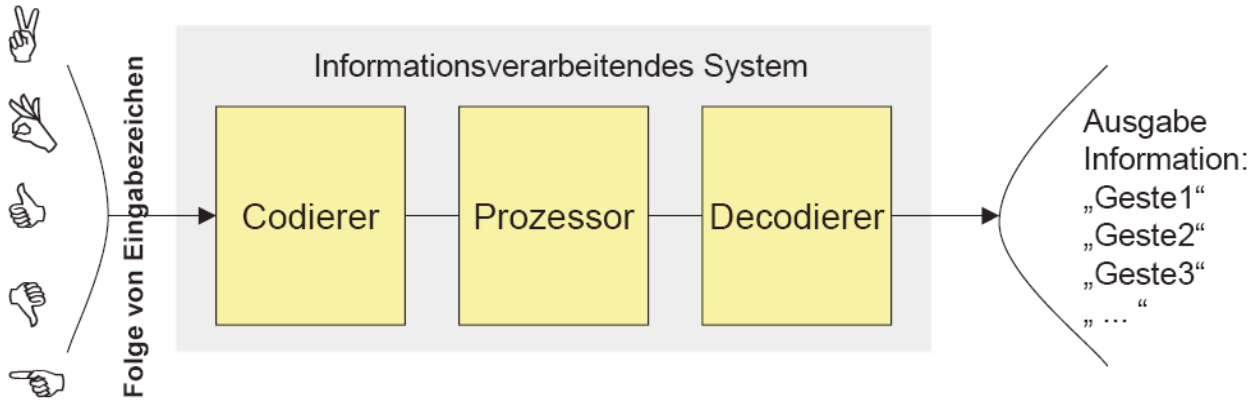


Grundlagen Codierung

- Bei der Codierung für technische Systeme ist man bestrebt einfache, kompakte und ggf. auch störsichere Repräsentationen für Nachrichten zu finden.
- Ziele:
  - Kurze Codewörter (um Speicherplatz zu sparen), Beispiel: 7 ist kürzer als |||||||, wenn jedes Zeichen gleich viel Speicherplatz verbraucht.
  - Verwendung von Redundanz (über das notwendige Maß hinausgehende Mittel, hier Informationen) zur Fehlererkennung und -korrektur.
- Insgesamt werden zwei Klassen der Codierung unterschieden:
  - für interne Repräsentation
  - für Informations*übertragung*

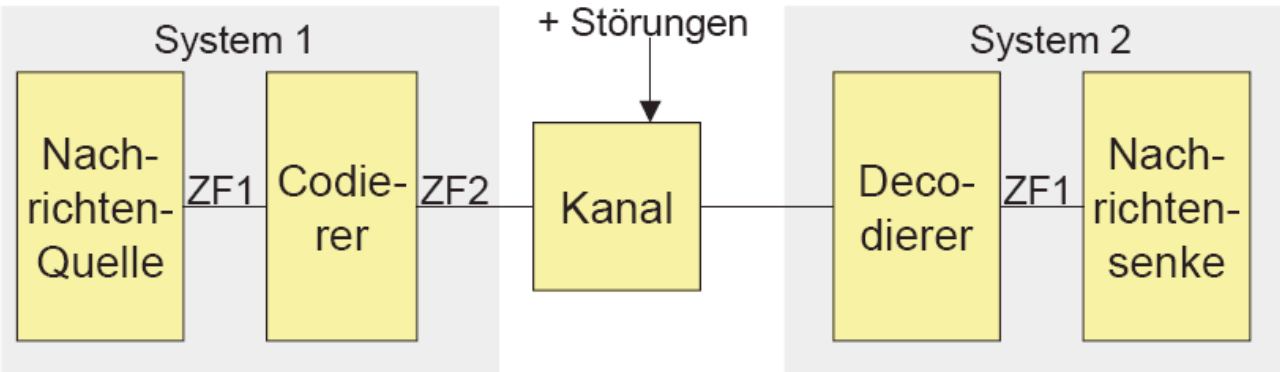
Codierung für interne Repräsentation

- Die Codierung wandelt eine Zeichenfolge in eine neue Zeichenfolge um, die für die interne Repräsentation geeignet ist (und z.B. möglichst keine Redundanz mehr enthält) → **Quellencodierung**.
- Beispiel (Gestenerkennung mit Hilfe einer Kamera):



- Der Prozessor arbeitet auf der internen Repräsentation, Codierung ist hier beispielsweise die Umsetzung in Pixel in der Kamera) und der Decodierer wandelt schließlich das Ergebnis in eine geeignete Form um.

Codierung für Informationsübertragung



- Der Codierer wandelt die Zeichenfolge ZF1 in eine für die *Übertragung geeignete Zeichenfolge* ZF2 um → **Kanalcodierung** (etwa durch gezieltes Hinzufügen von Redundanz)
- Der Decodierer macht die Umwandlung rückgängig und eliminiert dabei die Störungen durch die Nutzung der Redundanz.
- Bei Übertragung wird zwischen *paralleler Übertragung* (eine oder mehrere Zeichen werden gleichzeitig übertragen) und *serieller Übertragung* (die Zeichen werden in Untereinheiten zerlegt, die nacheinander übertragen werden) unterschieden.

Grundlagen Codierungstheorie

- Definitionen:
  - Symbol- bzw. Zeichenmenge $\sum$, bei linearer Ordnung heißt $\sum$ auch Alphabet.
  - Elementarer Zeichenvorrat B={L,O} oder B={1,0} oder B={H,L} bestehend aus zwei Binärzeichen oder **bi**nary digi**t**.
  - Die Menge $\sum^*$, die Menge aller Wörter über $\sum$.
  - Die Menge B^*, die Menge aller Binärwörter.
  - Die Menge B^n, die Menge aller Binärwörter aus genau *n* Binärzeichen.
- Eine Abbildungsvorschrift
  - $c: \sum \rightarrow B$ für einzelne Zeichen oder
  - $c: \sum^* \rightarrow B^*$ für Zeichenfolgen

  wird Code genannt.
- In der Rechentechnik werden zumeist Binärcodierungen für Alphabete verwendet, also Codierungen der Form $c: \sum \rightarrow B^*$.

Beispiel I: ASCII Tabelle

- ASCII (American Standard Code for Information Interchange
- Eigenschaften: Code fester Länge, d.h. jedes Zeichen wird mit der gleichen Anzahl von Binärzeichen codiert.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Beispiel II: Binärzahlen

- Binärzahlen sind direkte Codes: jede Binärstelle besitzt ein festes Gewicht. Mit den Multiplikatorfunktionen w(L)=1 und w(O)=0 wird jeden Binärwort d=<$d_n d_{n-1} \ldots d_0$> mit $d_i \in \{L,O\}$ durch

$$z = \sum_{i=0}^{n} g_i w(d_i)$$

  eine Zahl zugeordnet.
- Die ersten zehn Ziffern des Dezimalsystems können damit jeweils 4-bit langen Binärwörtern zugewiesen werden.
- Da mit 4-bit langen Binärwörtern auch alle Ziffern von 0 bis 15 dargestellt werden können, wird häufig das "Hexadezimalsystem" (griech.) oder "Sedezimalsystem" (lat.) verwendet.

a	c_D(a)
0	0000
1	000L
2	00L0
3	00LL
4	0L00
5	0L0L
6	0LL0
7	0LLL
8	L000
9	L00L
A	L0L0
B	L0LL
C	LL00
D	LL0L
E	LLL0
F	LLLL

Beispiel III: Darstellung von negativen Zahlen

- Beobachtung: zum Darstellen von ganzen Zahlen wird mindestens ein Bit zur Information benötigt, ob die Zahl negativ oder positiv ist.
- Erster Ansatz: negative Zahlen werden durch das Invertieren sämtlicher Bits der Darstellung der entsprechenden positiven Zahl dargestellt (**1er-Komplement**).

  z.B. 3: 0011 und -3: 1100
- *Problem 1*: der Aufbau einer Recheneinheit zur Addition und Subtraktion von Zahlen in dieser Darstellung wird deutlich komplizierter.
- *Problem 2*: die 0 kann auf zwei verschiedene Arten dargestellt werden: 0000 und 1111.
- Lösung: 2er-Komplement: Das **2er-Komplement** einer negativen Zahl ergibt sich aus dem 1er-Komplement und einer Addition mit 1 (modulo Stellenzahl).

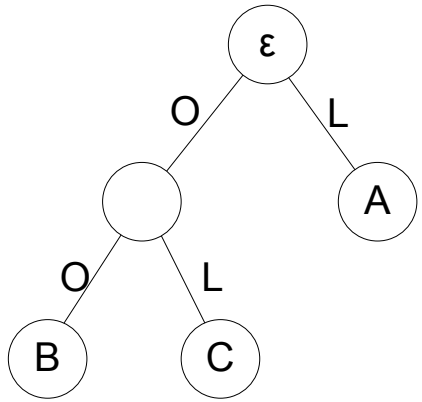
  z.B. 3: 0011 und -3: 1101
- Beide oben genannte Probleme werden mit dem 2er-Komplement gelöst.

Unterscheidung Codes fester Länge vs. Codes mit variabler Länge

- Codes mit fester Länge bilden jedes Eingabezeichen auf ein Binärwort fester Länge ab, also c: $A \rightarrow B^n$.
- Codes mit variabler Länge versuchen häufig vorkommende Eingabezeichen (z.B. Zeichen 'e' in der deutschen Sprache) mit kurzen Binärwörtern zu codieren, seltene Zeichen werden dagegen mit längeren Wörtern codiert. Historisches Beispiel: Morse-Code.
- Bewertung von Codes variabler Länge:
  - Ziel "Kurze Codewörter": bei entsprechender Wahl der Codierung kann die durchschnittliche Länge des Codes bei Verwendung von Codes variabler Länge verringert werden.
  - Problem: Werden mehrere codierte Zeichen in Folge übertragen, so ist unter Umständen eine Trennung der einzelnen Zeichen schwierig. Insbesondere Störungen, wie Bitfehler (Veränderung eines einzelnen Bits) sind dann schwieriger zu korrigieren.

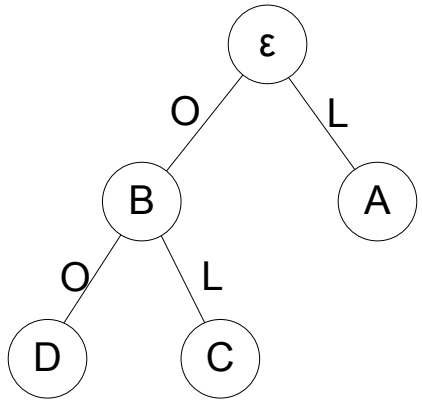
Beispiele für Codes variabler Länge

a) $A \rightarrow L$
$B \rightarrow OO$
$C \rightarrow OL$



Darstellung als **Codebaum**

b) $A \rightarrow L$
$B \rightarrow O$
$C \rightarrow OL$
$D \rightarrow OO$



Problem: Mehrdeutigkeiten bei der Decodierung

- Codiert man das Wort AABCDA mit dem Code aus dem vorigen Beispiel b, so erhält man LLOOLOOL.
- Dieses Wort kann wiederum zu AABCDC aber auch zu AADADA, AADADA, … dekodiert werden.

- Fano (*1917) beschrieb eine hinreichende Voraussetzung zur Eindeutigkeit der Codierungsabbildung: Kein Codewort darf Präfix eines anderen Codewortes sein (**Fano-Bedingung**).

⇒ Im Codebaum dürfen die Eingabezeichen nur in den Blättern stehen.

Ermittlung eines geeigneten Codes I

- **Ziel:** Nachrichten einer Quelle sollen durch Folge von Wörtern so codiert werden, daß die Gesamtlänge möglichst kurz ist.
- **Vorgehen:** Die Häufigkeit, mit der einzelne Zeichen von der Quelle emittiert werden, entscheidet über die Länge des verwendeten Codewortes.
- **1.Fall:**

  Die Auftrittswahrscheinlichkeiten sind nicht bekannt: dann ist es nur möglich, allen Zeichen gleichlange Codewörter zuzuordnen.
  Beispiel: ASCII-Code mit 256 Zeichen (jedes Zeichen wird mit 8 bit, 2^8=256, codiert).
- **Allgemein gilt**: Um ein Zeichen aus einem Zeichenvorrat *A* mit *M* Elementen mit einer eindeutigen Binärcodierung gleicher Länge zu codieren, werden ld *M* bit benötigt.

Ermittlung eines geeigneten Codes II

- Für Fall 2 (Berücksichtigung von Auftrittshäufigkeiten) müssen zunächst folgende Begriffe definiert werden:
  - Die **Auftrittswahrscheinlichkeit** p_a (relative Häufigkeit) eines Zeichens ist die Anzahl n der Auftritte des Zeichens in einer Zeichenfolge der Länge N in Relation zur Länge für N$\rightarrow\infty$, also $p_a = \lim_{N\to\infty} \frac{n}{N}$
  - Der **Entscheidungsgehalt** eines Zeichens ist die Anzahl der Schritte, die nötig sind, um das Zeichen aus dem Zeichenvorrat zu isolieren.
  - Der **Informationsgehalt** I_a eines Zeichens *a* steht im umgekehrten Verhältnis zu seiner Auftrittswahrscheinlichkeit. Er wird definiert als: I_a:=ld (1/p_a)= -ld p_a [bit].
  - Der von einem Zeichen der Quelle zu erwartende *mittlere Informationsgehalt* wird als **Entropie** *H* bezeichnet. Sie ist der Erwartungswert $H=E\{I_i\}=-\sum p_i \text{ ld } p_i$, wo *i* die Anzahl der Zeichen im Zeichenvorrat ist.

Bedeutung der Entropie

- Je höher die Entropie der Quelle ist, umso größer ist der Informationsgehalt der von ihr ausgegebenen Zeichen.
- Die Entropie ist ein "Maß für die Menge an Zufallsinformation, die in einer Informationsfolge steckt".
- Benennung erfolgte aufgrund formaler Strukturgleichheit der Formeln mit denen für die Berechnung der Entropie in der Thermodynamik
- Die Entropie nimmt ihr Minimum (min=0) an, falls die Quelle nur *ein* Zeichen aussendet.
- Die Entropie nimmt ihr Maximum an, falls alle Zeichen die gleiche Wahrscheinlichkeit haben → Die Wahrscheinlichkeit das nächste Zeichen korrekt vorhersagen zu können ist also am geringsten → **größter Informationsgehalt des Einzelzeichens**!
- Ist die Entropie gering, so reichen wenige Bits für die Codierung aus.
- Beispiel: Zwei Zeichen, a und b. p(a)=p(b)=0,5. Welchen Wert nimmt die Entropie an?

Strategie für kurzen Code

- **Strategie zur Codewortwahl**: Jedes Bit im Ausgabestrom sollte einen maximalen Informationsgehalt transportieren, d.h. die Wahrscheinlichkeit für den Auftritt einer '1' sollte nahe bei 0,5 liegen (ebenso die Wahrscheinlichkeit für eine '0').
- Damit Anleitung zur Konstruktion des Codebaums (Shannon-Fano):
  1. Sortieren der Zeichen nach ihrer Häufigkeit
  2. Aufteilung der Zeichen entlang dieser Reihenfolge in zwei Gruppen für den linken bzw. rechten Code-Teilbaum, so dass die summierte Auftrittswahrscheinlichkeit aller Zeichen beider Gruppen ungefähr (im Idealfall: exakt) gleich groß ist.
  3. Rekursiver Aufruf falls die Anzahl der Zeichen in einer entstandenen Gruppe größer als eins ist.

Beispiel: Shannon-Fano-Codierung

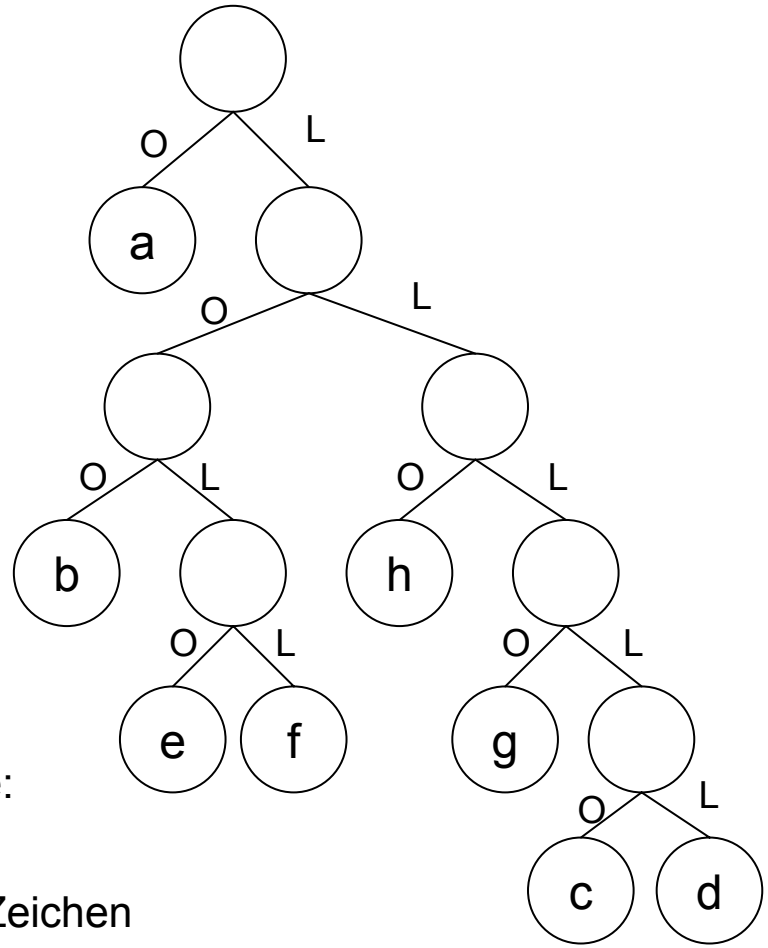
Zeichen i	a	b	c	d	e	f	g	h
P_i	0,5	0,1	0,05	0,03	0,09	0,08	0,07	0,08

Sortiert:

Zeichen i	a	b	e	f	h	g	c	d
P_i	0,5	0,1	0,09	0,08	0,08	0,07	0,05	0,03

Erwartungswert Bit/Zeichen:

Zeichen	P_i	Bit	Bit*P_i
a	0,5	1	0,5
b	0,1	3	0,3
c	0,05	5	0,25
d	0,03	5	0,15
e	0,09	4	0,36
f	0,08	4	0,32
g	0,07	4	0,28
h	0,08	3	0,24

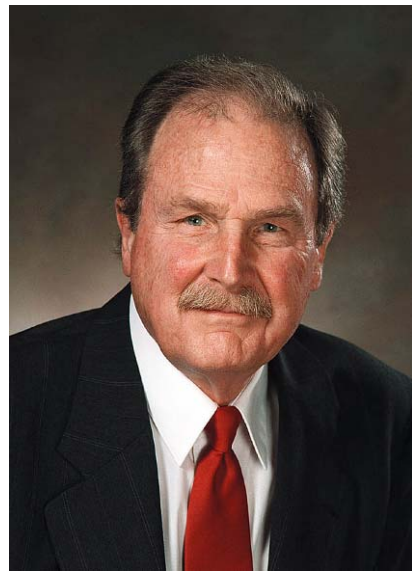


Also mittlere Zeichenlänge:
2,40 Bit/Zeichen

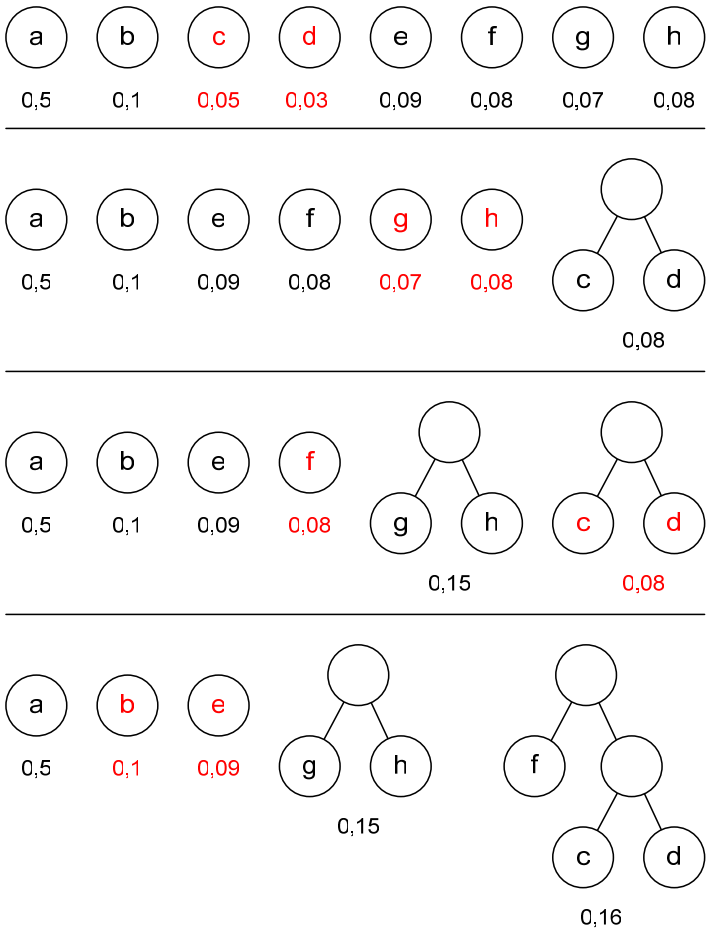
(bei Gleichverteilung 3bit/Zeichen

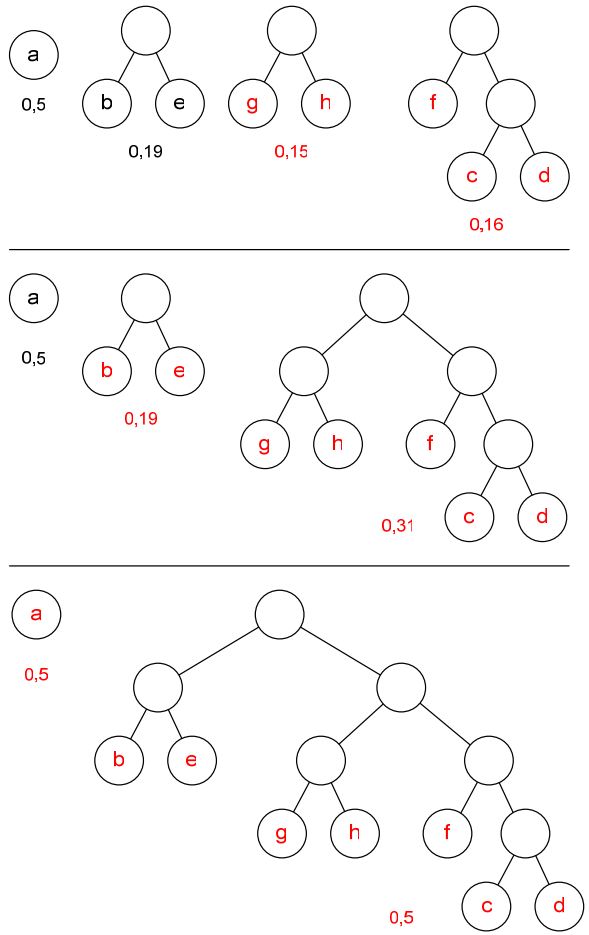
Huffman-Codierung I

- Bei genauer Betrachtung stellt man fest, daß die Shannon-Fano-Codierung nicht zwangsläufig optimal ist. So erhält man beispielsweise eine bessere Codierung, wenn man im Codebaum auf der vorherigen Folie die Blätter von 'h' und 'e' vertauscht.
- David A. Huffman (1925-1999) fand 1952 einen Algorithmus zur Erstellung eines optimalen Codebaums bei gegebenen Wahrscheinlichkeiten.
- Im Gegensatz zur Shannon-Fano-Codierung baut dieser Algorithmus den Baum *von den Blättern her* ("bottom-up") auf.
- Algorithmus:
  - Beginn: Für jedes Zeichen wird ein Baum bestehend aus einem Blatt eingeführt.
  - Schritt: Füge die Bäume mit den geringsten Wahrscheinlichkeiten zusammen.
  - Ende: nur noch ein Baum ist übrig.



Beispiel: Huffman-Codierung II

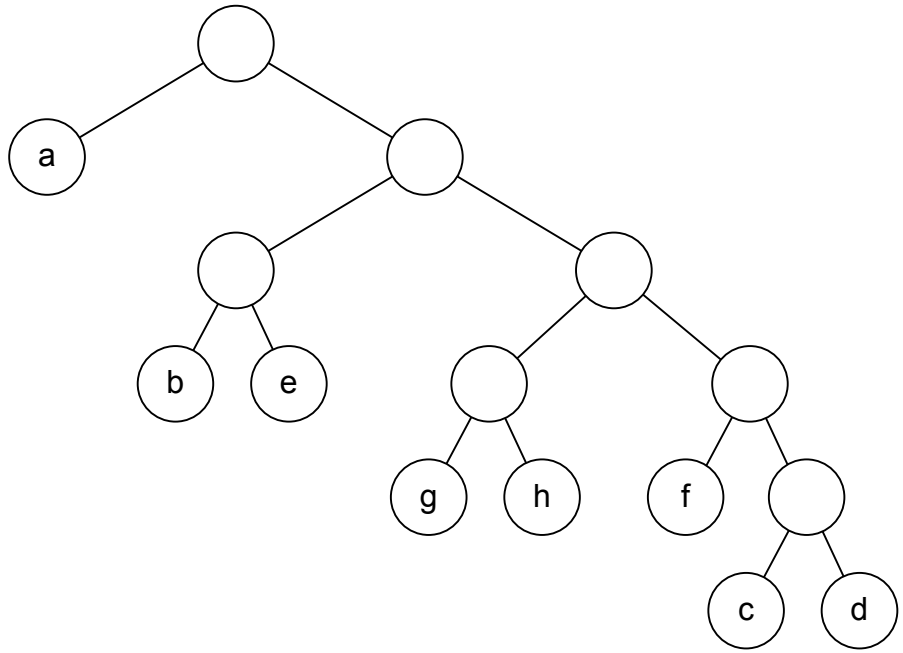




Huffman-Codierung III

Erwartungswert Bit/Zeichen:

Zeichen	P_i	Bit	Bit*P_i
a	0,5	1	0,5
b	0,1	3	0,3
c	0,05	5	0,25
d	0,03	5	0,15
e	0,09	3	0,27
f	0,08	4	0,32
g	0,07	4	0,28
h	0,08	4	0,32



Erwartungswert: 2,39 Bit/Zeichen

Huffman-Codierung IV

- Prinzipiell kommen zwei Vorgehensweisen bei der Huffman-Codierung zum Einsatz:
  - fester Codebaum: unterschiedliche Daten werden mit einem vorab bestimmten Codebaum codiert (z.B. ein Codebaum für die deutsche Sprache zur Komprimierung von deutschen Texten). Der Codebaum ist sowohl beim Empfänger als auch beim Sender im voraus bekannt.
  - dynamisch erzeugter Codebaum: zur Komprimierung von Daten
    - Die Daten werden auf die Häufigkeit der Zeichen analysiert und ein Codebaum erstellt.
    - Die Daten werden komprimiert und zusammen mit dem Codebaum übertragen / gespeichert.
    - Mit Hilfe des Codebaums werden die Daten dekomprimiert.
  - Einsatz der Huffman-Codierung u.a. in bestimmten Phasen von ZIP, JPEG.

Weitere Codierungsverfahren für Datenkompression

- Huffman-Codierung betrachtet nur Einzelzeichen. Man beobachtet jedoch, daß in vielen Fällen bestimmte Zeichen***folgen*** bzw. –gruppen häufiger vorkommen als andere. Ist dies der Fall, dann können vorteilhaft *lexikonbasierte Codierungsverfahren* eingesetzt werden.
- Problemstellungen:
  - Welche Zeichengruppen sollten betrachtet und in einem „Codebuch“-Eintrag gespeichert werden?
  - Wie wird der Text in die Zeichengruppen des Codebuchs aufgeteilt? Möglichkeiten:
    1. Codierer sucht, beginnend am Textanfang, den jeweils längsten Codebuch-Eintrag, der mit den nächsten Zeichen übereinstimmt.
    2. LFF (Longest Fragment First): der Codierer sucht über den gesamtem Text hinweg nach dem längsten Substring, der mit einem Eintrag im Codebuch übereinstimmt, anschließend den zweitlängsten, … bis der Text komplett abgearbeitet wurde.

Beispiel für Longest Fragment First

- Beispiel:

  Lexikon: {a,b,c,aa,aab,ab,baa,bccb,bccba}

  $\Rightarrow$ Die insgesamt 9 einzelnen Strings können durch jeweils 4 Bits codiert werden.

  Zu kodierender Text:

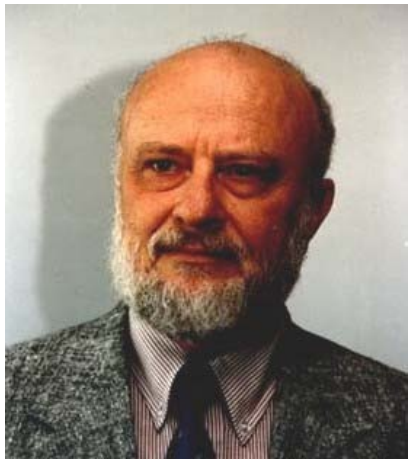
  "aaabccbaaaa" $\triangleq$ 11*8bits = 88 bits

  Bei Verwendung von LFF:

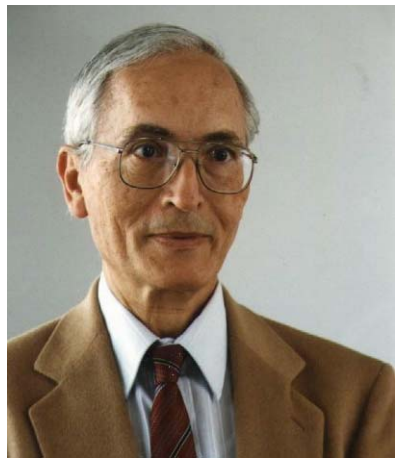
  5 Fragmente zu je 4 Bits $\Rightarrow$ 20 bit

Lempel-Ziv-Kompression



Abraham Lempel

- Lempel-Ziv-Kompression ist eine Familie von verlustfreien Kompressionsverfahren, die nach ihren Erfindern benannt wurden
- Vertreter:
  - LZ77:Grundlage von gzip(1977)
  - LZ78 (1978)
  - LZW: Lempel-Ziv-Welch (1984)

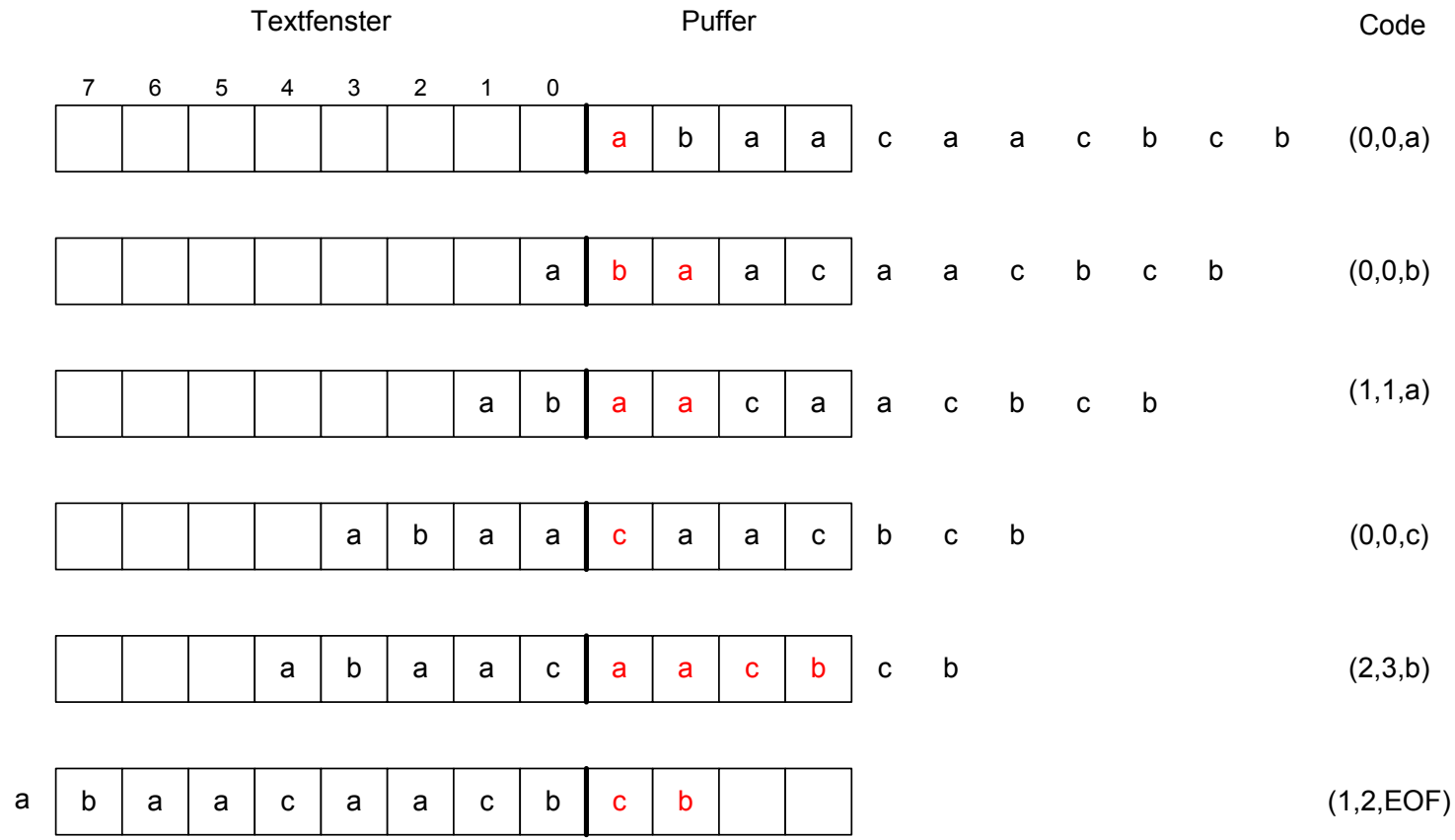


Jacob Ziv

LZ77

- LZ77 wird zum Teil auch *Sliding-Window-Algorithmus* genannt, weil der Text durch ein Fenster bearbeitet wird, das über den Text geschoben wird.
- Das Fenster besteht aus zwei Bestandteilen konstanter Größe: dem Textfenster und dem Puffer.
  - Textfenster: speichert den zuletzt eingelesenen Text.
  - Puffer: enthält aus dem Eingabestrom eingelesene, aber noch nicht codierte Zeichen.
  - Das Textfenster ist in der Regel deutlich größer als der Puffer.
- Algorithmus-Funktionsweise:
  - Im Textfenster wird nun ein möglichst langer Substring passend zum Anfang des Puffers gesucht.
  - Der Pufferanfang und das erste nachfolgende Zeichen wird nun mit folgenden Parametern codiert:
    - Anfangsadresse (offset) des Substrings im Puffer,
    - Länge des Substrings,
    - nach Pufferanfang folgendes Zeichen *sign*.

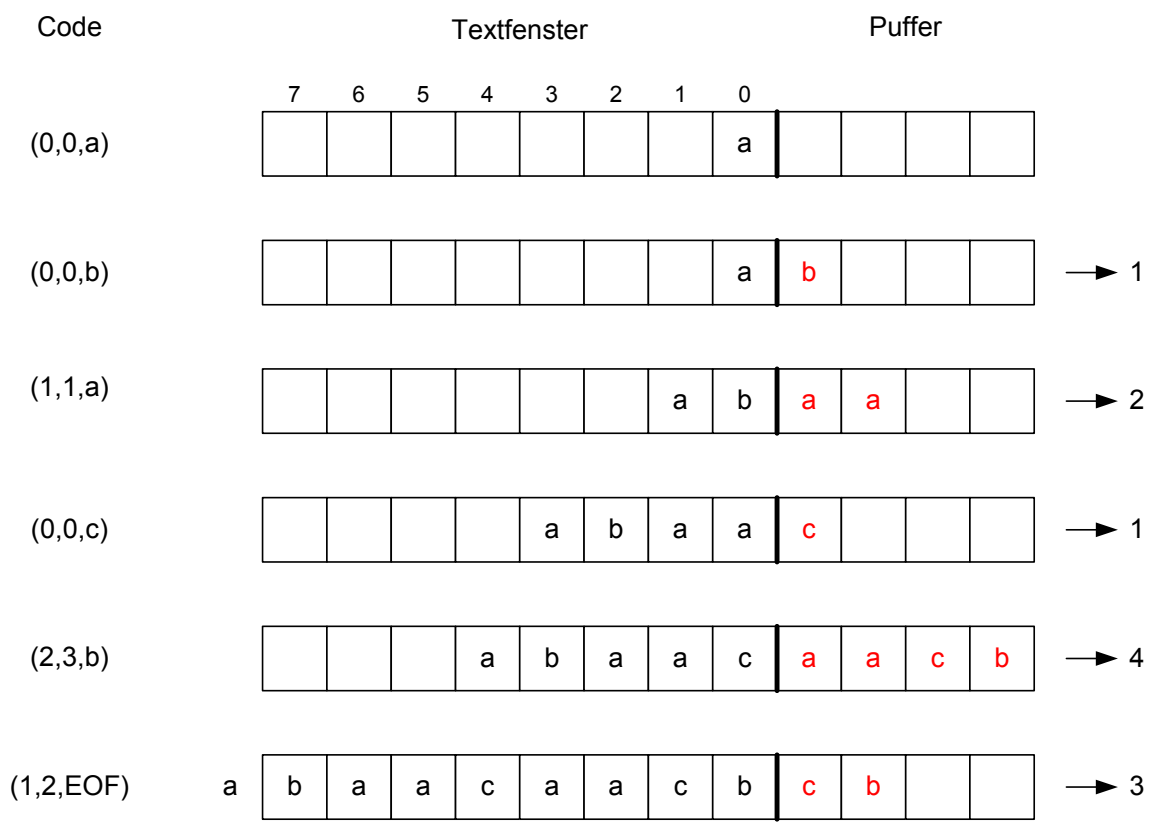
LZ77 – Kodierung: Beispiel



LZ77-Dekompression

- Die Dekompression kann analog erfolgen und es muss kein Codebuch übertragen werden (Codebuch implizit in den übertragenen Zeichen *sign*).
- Algorithmus:
  - Starte mit leerem Textfenster, Puffer und füge das erste Zeichen entsprechend dem Code in das Textfenster ein.
  - Für alle weiteren Codefragmente (*offset*,*length*,*sign*):
    - Kopiere den String der Länge *length* aus dem Textfenster beginnend an der Position *offset* in den Puffer und füge das Zeichen *sign* an.
    - Verschiebe das Fenster um (*length*+1).

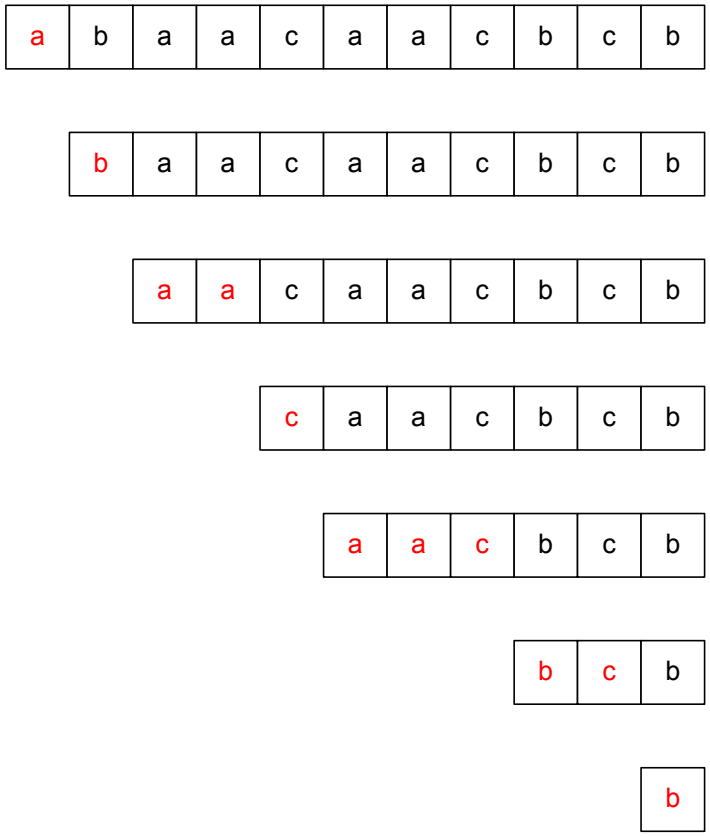
LZ77-Dekomprimierung: Beispiel



LZ78-Algorithmus

- Nachteile des LZ77-Algorithmus: Der Algorithmus kann Eingaben mit Zeichenfolgen, die sich in größeren Abständen (größer als die Länge des Fensters) wiederholen, schlecht komprimieren.
- Problemlösung: Verwendung von Wörterbuchern im LZ78-Algorithmus.
- Während der Kompression werden ständig neue Zeichenfolgen im Wörterbuch abgespeichert und Ausgaben der Form (*i*;*K*()) generiert.
- *i* ist dabei der Index der längsten Zeichenfolge im Eingabetext, die schon im Wörterbuch ist, *K*() der Code für das im Eingabetext darauf folgende Zeichen, bzw. der Index für das Zeichen, falls dieses bereits im Wörterbuch enthalten ist.

LZ78 – Kodierung: Beispiel



Code
(0,K(a))
(0,K(b))
(1,1)
(0,K(c))
(3,4)
(2,4)
(2,K(EOF))

Wörterbuch

Index	String
0	ε
1	a
2	b
3	aa
4	c
5	aac
6	bc

LZW-Algorithmus

- Das LZW-Verfahren, von Terry Welch 1984 entwickelt, modifiziert LZ78 leicht.
- Die Kompression erfolgt mittels Wörterbüchern, welches zu Beginn mit allen Einzelzeichen gefüllt ist. Auch beim LZW-Algorithmus muss das Wörterbuch nicht übertragen werden, sondern kann "on-the-fly" während der Dekompression erstellt werden.
- Typischerweise werden die Einträge des Wörterbuchs über einen 12bit Index angesprochen, es können also 4096 Einträge gespeichert werden. Die ersten 256 Einträge sind dabei für die Einzelzeichen reserviert.
- Der ausgegebene Code gibt nun nur noch den entsprechenden Index an und nicht mehr zusätzlich noch das nachfolgende Zeichen.
- Ein Applet zum LZ77 ist unter http://www-fs.informatik.uni-tuebingen.de/~reinhard/datkom/LZW_Applet.html verfügbar, ein Applet zu den anderen Verfahren unter http://graphics.ethz.ch/teaching/infotheory/Downloads/Applet_work_fullscreen.html
- Der LZW-Algorithmus wird u.a. zur Kompression von gif-Bildern eingesetzt.

LZW-Algorithmus

- Pseudo-Code:

```
InitializeStringTable;
pre = EMPTY;
while ( c = nextCharacter() )
{
    if (inTable(pc))
        pre = pre+c;
    else
    {
        outputCode(pre);
        insertInTable(pre+c);
        pre = c;
    }
}
outputCode(pre);
```

Zeichen	w	Z	Wörterbuch Eintrag	Index	Ausgabe
a		a			
b	a	b	ab	256	a
a	b	a	ba	257	b
a	a	a	aa	258	a
c	a	c	ac	259	a
a	c	a	ca	260	c
a	a	a			
c	aa	c	aac	261	258
b	c	b	cb	262	c
c	b	c	bc	263	b
b	c	b			
EOF	cb				262

LZW-Algorithmus: Dekompression

- Pseudo-Code:

```
InitializeStringTable;
c = getCode();
outputString(translate(c));
old = c;
while(c = getCode())
{
    if(inTable(c))
    {
        outputString(translate(c));
        insertInTable(translate(old)+
                        getFirstChar(translate(c)))
    }
    else
    {
        s = translate(old) +getFirstChar(translate(old));
        outputString(s);
        insertInTable(s);
    }
    old = c;
}
```

			Wörterbuch		
Code	old	c	Eintrag	Index	Ausgabe
a		a			a
b	a	b	ab	256	b
a	b	a	ba	257	a
a	a	a	aa	258	a
c	a	c	ac	259	c
258	c	258	ca	260	aa
c	258	c	aac	261	c
b	c	b	cb	262	b
262	b	262	bc	263	cb

Burrows-Wheeler-Transformation BWT

- Fragestellung: Häufig kommen bestimmte Kombinationen von Zeichen in Texten vor. Kann man dies ausnutzen?
  - Beispiel deutsche Sprache:
    - Große Buchstaben stehen oft nach einem Leerzeichen.
    - Nach einem 'q' steht häufig ein 'u'.
- Grundidee: Erzeugung einer Permutation der Eingabedaten, so dass Zeichen mit ähnlichem Kontext nahe beieinander liegen.
- Die Burrows-Wheeler-Transformation ermöglicht eine solche Permutation und wandelt so einen Text in einen leichter zu komprimierenden Text (reversibel) um.
- Durch Kombination der BWT mit anderen Kompressionsalgorithmen (z.B. MTF/Huffman) können so die Ergebnisse besser komprimiert werden.
- Beispiel: Bzip2 verwendet BWT und Huffman.

BWT- Algorithmus

- Die Burrows-Wheeler-Transformation besteht aus drei Schritten:
  1. Erzeugung aller Rotationen des zu komprimierenden Textes: in jedem Schritt wird das letzte Zeichen (L) vorne an das erste Zeichen (Z) angehängt.
     Ergebnis: Tabelle bestehend aus *n* Zeilen und Spalten, wobei *n* die Länge des zu komprimierenden Textes ist.
  2. Alphabetische Sortierung aller Zeilen.
  3. Zur Komprimierung wird als Ergebnis der Burrows-Wheeler-Transformation die letzte Spalte (L), sowie die Zeilennummer, die den ursprünglichen Text enthält, verwendet.

BWT-Beispiel: Erzeugung der Permutationen

Index	F										L
0	A	B	R	A	K	A	D	A	B	R	A
1	A	A	B	R	A	K	A	D	A	B	R
2	R	A	A	B	R	A	K	A	D	A	B
3	B	R	A	A	B	R	A	K	A	D	A
4	A	B	R	A	A	B	R	A	K	A	D
5	D	A	B	R	A	A	B	R	A	K	A
6	A	D	A	B	R	A	A	B	R	A	K
7	K	A	D	A	B	R	A	A	B	R	A
8	A	K	A	D	A	B	R	A	A	B	R
9	R	A	K	A	D	A	B	R	A	A	B
10	B	R	A	K	A	D	A	B	R	A	A

BWT-Beispiel: Sortieren der Permutationen

Index	F										L
0	A	A	B	R	A	K	A	D	A	B	**R**
1	A	B	R	A	A	B	R	A	K	A	**D**
2	A	B	R	A	K	A	D	A	B	R	**A**
3	A	D	A	B	R	A	A	B	R	A	**K**
4	A	K	A	D	A	B	R	A	A	B	**R**
5	B	R	A	A	B	R	A	K	A	D	**A**
6	B	R	A	K	A	D	A	B	R	A	**A**
7	D	A	B	R	A	A	B	R	A	K	**A**
8	K	A	D	A	B	R	A	A	B	R	**A**
9	R	A	A	B	R	A	K	A	D	A	**B**
10	R	A	K	A	D	A	B	R	A	A	**B**

$\Rightarrow$ Ergebnis der BWT ist "RDAKRAAAABB" und 2.

Rücktransformation der BWT

- Auf dem ersten Blick erscheint es nicht möglich aus dem Ergebnis den ursprünglichen Text wiederherzustellen.
- **Überlegung**:
  - Da in der Tabelle der BWT in jeder Spalte die gleichen Zeichen (auch mit gleicher Anzahl) stehen, kann die F-Spalte einfach durch alphabetische Sortierung der L-Spalte wiederhergestellt werden.
  - Durch eine Rotation nach rechts wird die L-Spalte zur F-Spalte (und die F-Spalte zur 2. Spalte). Die Reihenfolge der Zeilen ist aber nun nicht mehr konsistent. $\Rightarrow$ Sortierung nötig.
  - Durch wiederholtes Einfügen der L-Spalte und weiteres Rotieren und Sortieren kann die Tabelle komplett wieder hergestellt werden.
- Anmerkung: Ein effizienterer Algorithmus zur Wiederherstellung des Textes wird beschrieben unter:

  http://www.data-compression.info/JuergenAbel/Preprints/Preprint_Verlustlose_Datenkompression_BWT.pdf

BWT-Rücktransformation

Index	F	L
0		**R**
1		**D**
2		**A**
3		**K**
4		**R**
5		**A**
6		**A**
7		**A**
8		**A**
9		**B**
10		**B**

BWT-Rücktransformation

Index	F	L
0	R	**R**
1	D	**D**
2	A	**A**
3	K	**K**
4	R	**R**
5	A	**A**
6	A	**A**
7	A	**A**
8	A	**A**
9	B	**B**
10	B	**B**

Einfügen von L

BWT-Rücktransformation

Index	F	L
0	A	**R**
1	A	**D**
2	A	**A**
3	A	**K**
4	A	**R**
5	B	**A**
6	B	**A**
7	D	**A**
8	K	**A**
9	R	**B**
10	R	**B**

Sortieren von F

BWT-Rücktransformation

Einfügen von L

Rotieren nach Rechts

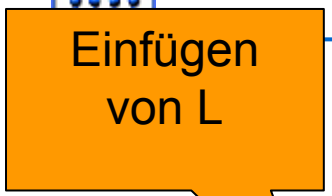
Index	F		L
0	R	A	**R**
1	D	A	**D**
2	A	A	**A**
3	K	A	**K**
4	R	A	**R**
5	A	B	**A**
6	A	B	**A**
7	A	D	**A**
8	A	K	**A**
9	B	R	**B**
10	B	R	**B**

BWT-Rücktransformation

Index	F		L
0	A	A	**R**
1	A	B	**D**
2	A	B	**A**
3	A	D	**K**
4	A	K	**R**
5	B	R	**A**
6	B	R	**A**
7	D	A	**A**
8	K	A	**A**
9	R	A	**B**
10	R	A	**B**

Sortieren

BWT-Rücktransformation



Index	F			L
0	R	A	A	**R**
1	D	A	B	**D**
2	A	A	B	**A**
3	K	A	D	**K**
4	R	A	K	**R**
5	A	B	R	**A**
6	A	B	R	**A**
7	A	D	A	**A**
8	A	K	A	**A**
9	B	R	A	**B**
10	B	R	A	**B**

Rotieren
nach Rechts

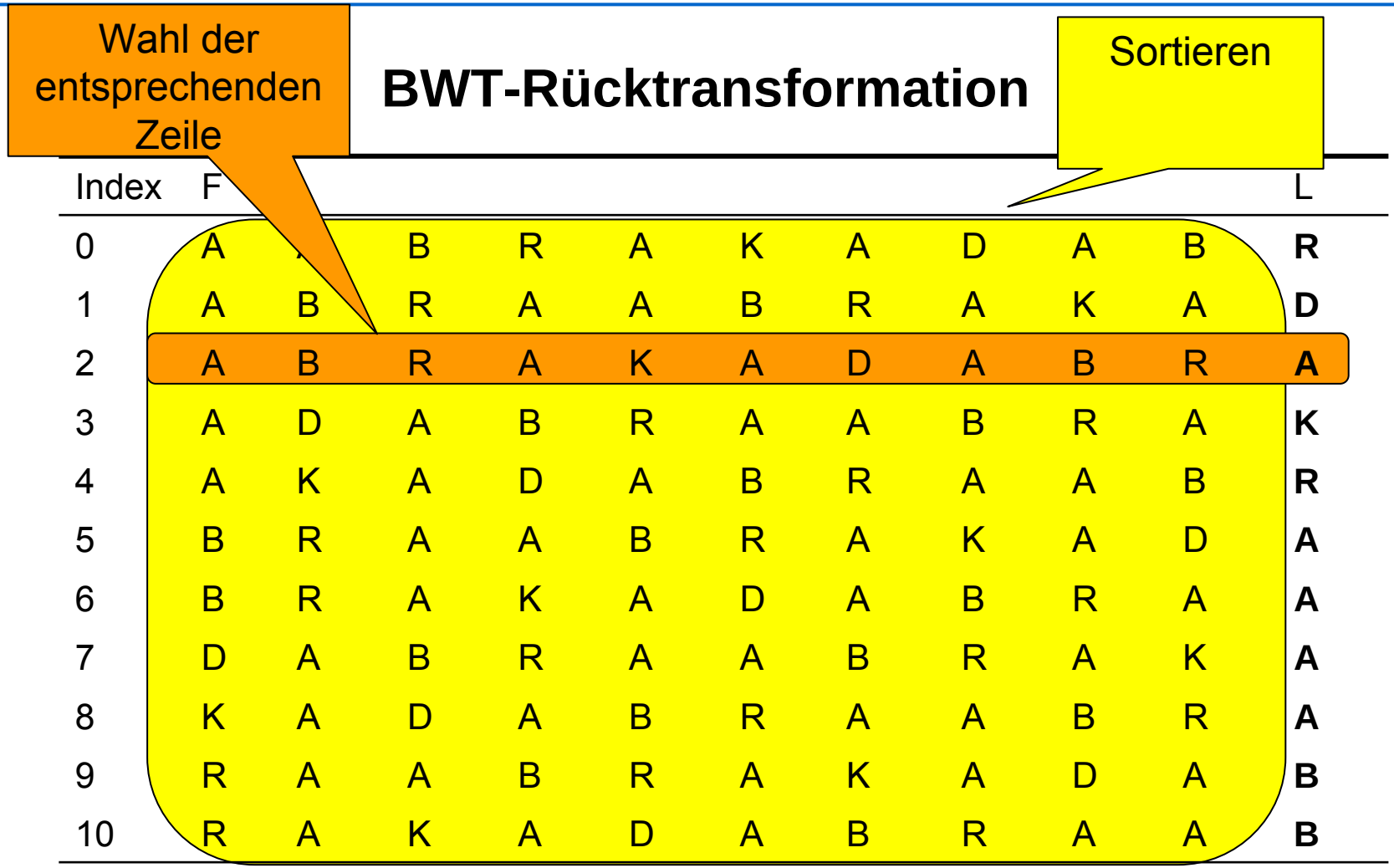
BWT-Rücktransformation

Index	F			L
0	A	A	B	**R**
1	A	B	R	**D**
2	A	B	R	**A**
3	A	D	A	**K**
4	A	K	A	**R**
5	B	R	A	**A**
6	B	R	A	**A**
7	D	A	B	**A**
8	K	A	D	**A**
9	R	A	A	**B**
10	R	A	K	**B**

Sortieren

…

BWT-Rücktransformation



Move-To-Front- (MTF-) Recodierung

- Ergebnis der BTW-Codierung ist nun ein String, bei dem viele gleiche Zeichen aufeinander folgen.
- Die Zeichenhäufigkeit hat sich jedoch noch nicht verändert, weshalb eine Huffman-Codierung noch keine Vorteile bringt.
- Die Move-To-Front-Recodierung bringt eine Veränderung der Zeichenhäufigkeiten mit sich und senkt die Entropie. Dadurch wird die Komprimierung erfolgreicher.
- Grundidee: Anstelle der einzelnen Zeichen wird nun der Abstand zum letzten gleichen Zeichen gespeichert.
  $\Rightarrow$ Folgen in dem Text nun viele gleiche Zeichen aufeinander, so wird sehr oft eine 0 gespeichert.

MTF-Algorithmus

- Initialisiere eine Tabelle, die alle Zeichen enthält.
- Für jedes Zeichen aus dem Eingabetext:
  - Ersetze das Zeichen durch die Position in der Tabelle
  - Setze das Zeichen in der Tabelle an die 0.Position
- Beispiel: "RDAKRAAAABB"
- Resultat: 43243200040
- Das Ergebnis lässt sich nun sehr gut mit dem Huffman-Verfahren komprimieren
- Das Verfahren (BWT+MTF) funktioniert besonders gut bei sehr langen Texten.

Lauflängencodierung

- Unter anderem im Kontext der Burrows-Wheeler-Komprimierung wird ein weiteres Kompressionsverfahren eingesetzt: die **Lauflängencodierung** (run-length encoding, RLE).
- Dieses sehr einfach Verfahren ist vor allem dann sehr effizient, wenn in den zu codierenden Daten Sequenzen von gleichen Werten enthalten sind.
- Anstelle der Wiederholung der gleichen Werte wird die Anzahl der gleichen Zeichen mit codiert:
  "AAABBBBAAAAA" $\Rightarrow$ "A3B4A5"
- Problem 1: Diese Form der Komprimierung kann im schlimmsten Fall auch zu einer Vergrößerung der Daten führen.
  z.B. "ABA" $\Rightarrow$ "A1B1A1"
- Problem 2: Ein weiteres Problem kann die Schwierigkeit der Interpretation von Daten sein: kann "510211" zu "5001" oder zu "55555555552222222222" dekodiert werden.
- Eine Lösung der Probleme ist die Trennung von Quelldaten und Kontrolldaten z.B. durch die Einführung von Sonderzeichen.

Informationstheorie

Fehlererkennung und -korrektur

Fehlererkennung und -korrektur

- Problem: Bei der Übertragung von Codewörtern auf störanfälligen Kanälen kann es zu Fehlern kommen.
- Da jede Bitstelle des Quellentextes mit *Nutz*informationen belegt ist, bedeutet schon ein einzelner Bitfehler eine Verfälschung.
- Abhilfe ist durch den Einsatz von zusätzlichen Bitstellen (**Redundanz**) möglich. (Redundanz bedeutet den Einsatz von zusätzlichen technischen Mitteln, die über den Einsatz im Nutzbetrieb hinausgehen).
- Zum Umgang mit Fehlern stehen grundsätzlich zwei Mechanismen zur Verfügung:
  - **Fehlererkennung**: fehlerhafte Codewörter werden erkannt ⇒ es wird nicht auf fehlerhaften Daten weitergearbeitet, der Empfänger kann den Fehler dem Sender mitteilen, dieser sendet daraufhin die Daten erneut.
  - **Fehlerkorrektur**: neben der Erkennung des Fehlers kann auch der korrekte Wert direkt aus dem übertragenen Code errechnet werden. Eine erneute Übertragung ist nicht nötig.

Hamming-Distanz

- Für die Methodik der Fehlererkennung bzw. Fehlerkorrektur ist zunächst ein Maß für die Ähnlichkeit zweier Binärwörter wichtig:
- R.W. Hamming führte dazu die Hamming-Distanz (Hamming-Abstand) ein, die definiert ist als:

$$h(w_1,w_2)= \sum_{i=1}^{n} d_i, \text{ mit } d_i = 1 \text{ falls } a_i \neq b_i, 0 \text{ sonst}$$

und $w_1=a_1a_2\ldots a_n$, $w_2=b_1b_2\ldots b_n$

oder einfach die Anzahl der Bitstellen, in denen sich zwei Wörter unterscheiden.

- Die Hamming-Distanz eines Codes ist der minimale Abstand zwischen zwei beliebig gepaarten Code-wörtern.

Richard Wesley Hamming

1915-1998

Fehlererkennung und -korrektur

- Bei einer Codierungsabbildung mit einer Hamming-Distanz von *n* können alle Verfälschungen mit *n*-1 Bitfehlern *erkannt* werden.
  Begründung: Da sich alle korrekten Codewörter um mindestens n Bit voneinander entscheiden, kann durch Veränderung von *n*-1 Stellen kein korrektes Wort in ein anderes korrektes Wort überführt werden.
- Unter der Annahme, dass höchstens (*n*-1)/2 Bitfehler auftreten, können alle Verfälschungen *korrigiert* werden.
  Begründung: Es gibt nur ein korrektes Wort in das das als verändert erkannte Codewort mit maximal (*n*-1)/2 Veränderungen von Bitstellen überführt werden kann. Andernfalls hätten die beiden korrekten Wörter einen Hammingabstand <*n*.
- **Beispiel**: Sei ein Code mit Hamming-Abstand 3 gegeben. Alle Codewörter, bei denen maximal 2 Bits verfälscht wurden, können erkannt werden. Unter der Annahme, daß maximal ein Bit verfälscht wurde, können alle Fehler korrigiert werden.
- **Beobachtung**: Zur Fehlerkorrektur ist gegenüber der Fehlererkennung ein stark erhöhtes Maß an Redundanz nötig.

Fehlererkennung I: Paritätsbits

- Ein einfaches Verfahren zur Fehlererkennung/Fehlerkorrektur ist der Einsatz von Paritätsbits.
- Paritätsbits geben an, ob die Anzahl der 1er in einem Block gerade (parity=0) oder ungerade (parity=1) ist.
- Man unterscheidet dabei zwischen:
  - LRC (longitudinal redundancy check): Paritätsbit für jede Zeile
  - VRC (vertical redundancy check): Paritätsbit für jede Spalte
- Bei gleichzeitigen Einsatz von VRC und LRC ist auch eine Korrektur möglich, so lange pro Zeile und Spalte höchstens ein Bit verfälscht wurde.
- Aber: nur Fehlererkennung trotz hohen Redundanzgrads (37,5% im Beispiel, bei dem jeweils Blöcke von 4 Codewörtern durch VRC und LRC gesichert werden)

								LRC
1	0	1	1	0	1	0	1	1
0	1	1	0	0	1	0	0	1
0	0	0	1	1	0	1	1	0
1	1	1	0	0	1	0	0	0
VRC	0	0	1	0	1	1	1	0

Fehlererkennung II: CRC (Cyclic redundancy check)

- Beobachtung: Neben Einzelfehlern sind auch Serienfehler (so genannte Bursts) sehr häufig.
- Gesucht: Codes und effizientes Verfahren, mit denen Verfälschungen mit hoher Wahrscheinlichkeit erkennt und ggf. auch korrigieren kann.
- Sehr populär: *zyklische binäre Codes* und darauf aufsetzend das Verfahren CRC – Cyclic Redundancy Check, z.B. bei Ethernet, (Floppy) Disks, ...

Cyclic Redundancy Check

- Grundidee:
  - Codewörter X_i werden durch Linearkombination von Generatorwörtern G_i erzeugt
  - Generatorwörter G_i werden durch Stellenverschiebung aus einem *Generatormuster* abgeleitet
  - Codewörter haben *m* Nachrichtenstellen, *k* Kontrollstellen und $n = m + k$ Gesamtstellen
- Beispiel:

			1	0	1	1	Generatormuster
x_7	x_6	x_5	x_4	x_3	x_2	x_1	
0	0	0	1	0	1	1	G_1
0	0	1	0	1	1	0	G_2
0	1	0	1	1	0	0	G_3
1	0	1	1	0	0	0	G_4
u^6	u^5	u^4	u^3	u^2	u^1	u^0	

Cyclic Redundancy Check

- Formalisierung:
  - Zuordnung von Wertigkeiten u^{n-1}, ..., u^1, u^0 zu den Stellen x_n, ..., x_1 des Codewortes X_i
  - Darauf aufbauend rein formale Interpretation der Binärwörter X_i und G_i als Polynome vom Grad k bzw. n–1, bei denen die Werte der Koeffizienten mit den einzelnen Bitstellen übereinstimmen
- Also Darstellung als Polynome:

$$G(u) = g_k u^k + g_{k-1} u^{k-1} + \ldots + g_1 u^1 + g_0 \text{ (Generatorpolynom mit Grad } k)$$

$$X(u) = x_n u^{n-1} + x_{n-1} u^{n-2} + \ldots + x_2 u^1 + x_1 u^0 \text{ (Codewortpolynom mit Grad } n-1)$$

Cyclic Redundancy Check

- Voriges Beispiel in Polynomdarstellung:
  - Generatorpolynom ist vom Grad $k = 3$, $G(u) = u^3 + u^1 + 1$
  - Verschiebung des Musters um eine Stelle über Multiplikation mit u

u^6	u^5	u^4	u^3	u^2	u^1	u^0	Wertigkeiten
			1	0	1	1	$G(u)$
		1	0	1	1		$G(u) \cdot u^1$
	1	0	1	1			$G(u) \cdot u^2$
1	0	1	1				$G(u) \cdot u^3$

  - Codewort ist z.B. die Summe aus zweiter und vierter Zeile, also $X(u) = G(u)\ (u^3 + u^1)$, das entspricht einer Bitfolge 1001110

Cyclic Redundancy Check

- Da Generatorworte immer das Generatorpolynom *G(u)* enthalten, sind auch gültige Codewörter dadurch ausgezeichnet, daß sie ein Vielfaches des Generatorpolynoms *G(u)* sind (d.h. durch Polynommultiplikation aus diesem hervorgegangen sind).
- $G(u)$ ist also Teiler aller $X(u)$, d.h. es gilt: $X(u) / G(u) = Q(u)$, wobei diese Polynomdivision immer ohne Rest aufgeht.
- Kriterium für die Fehlererkennung: alle Wörter, bei denen diese Division nicht ohne Rest aufgeht, werden als fehlerhaft zurückgewiesen.
- Das Generatorpolynom ist also Teiler aller gültigen Codeworte und jedes zyklisch verschobene Codewort ist wieder gültig (daher auch der Name).

CRC - Fehlererkennung

- Typische Fragestellungen:
  - Wie erzeugt man Codewörter, und wie können die eigentlichen Daten anschließend wieder extrahiert werden?
  - Wie muss das Generatorpolynom gewählt werden um einen möglichst hohen Fehlererkennungsgrad zu erreichen?

Cyclic Redundancy Check

- Vereinbarung: alle Operationen auf Codewörtern werden modulo-2 durchgeführt, d.h. folgende Operationen sind definiert:

  Multiplikation: 0*0=0, 0*1=0, 1*0=0, 1*1=1

  Addition: 0+0=0, 0+1=1, 1+0=1, 1+1=0

  Subtraktion: 0-0=0, 0-1=1, 1-0=1, 1-1=0

  $\Rightarrow$ kein Unterschied zwischen Subtraktion und Addition

- Idee zur Berechnung der Kontrollstellen: Division des Codewortes mit angehängten (zu null gesetzten) Kontrollstellen durch Generatorpolynom. Falls Rest: Abzug dieses Restes, Ergebnis ist teilbar. Also:
  1. Schritt: Anhängen von *k* Kontrollstellen $\{0\}^k$ an das Binärwort (vorläufiges Codewort)
  2. Schritt: Division des vorläufigen Codeworts durch das Generatorpolynom G(u) und Berechnung des Restes
  3. Schritt: Subtraktion (bzw. Addition, siehe vorherige Folie) des Restes vom vorläufigen Codewort. Ergebnis ist endgültiges Codewort

Cyclic Redundancy Check

- Beispiel zur Berechnung der Kontrollstellen: $k = 3$, $G(u) = u^3 + u^1 + 1$, $m = 4$ Nachrichtenstellen, $n = k + m$ Gesamtstellen.
- Seien als Nachrichtenstellen gewählt: (**1001**). Also Codewort: (**1001???**).
- Polynomdivision:

$$
\begin{array}{cccccccc}
 & \overbrace{}^{X(u)} & & & & & \overbrace{}^{G(u)} & & \overbrace{}^{Q(u)} \\
(u^6 & u^5 & u^4 & u^3 & u^2 & u^1 & u^0) & / \quad (u^3\ u^2\ u^1\ u^0) & = & (u^3\ u^2\ u^1\ u^0) \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & / \quad 1\ 0\ 1\ 1 & = & 1\ 0\ 1\ 0 \\
1 & 0 & 1 & 1 & & & & & & \\
\hline
 & 0 & 1 & 0 & 0 & & & & & \\
 & 0 & 0 & 0 & 0 & & & & & \\
\hline
 & & 1 & 0 & 0 & 0 & & & & \\
 & & 1 & 0 & 1 & 1 & & & & \\
\hline
 & & & 0 & 1 & 1 & 0 & & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
\hline
 & & & & 1 & 1 & 0 & \longleftarrow \text{Rest der Division} & &
\end{array}
$$

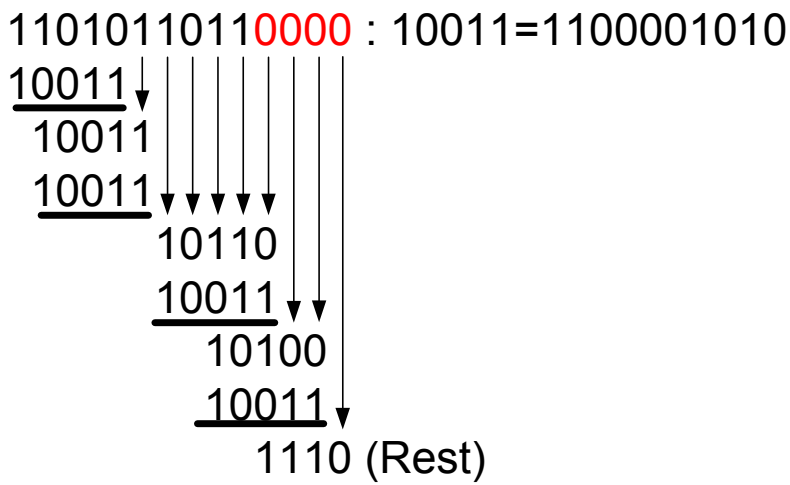
- Also Codewort: (**1001000**) – (**0000110**) = (**1001110**)

Weiteres CRC-Beispiel

Daten: 1101011011

Generatorpolynom: u^4+u+1

CRC-Codewort:
1101011011**1110**



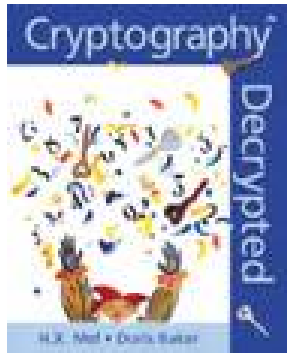
Vorteile des CRC-Algorithmus

- Mit einem geringen Maß an Redundanz ist eine vergleichsweise hohe Fehlererkennungsrate erreichbar.
- Erkannte Fehler sind bei guter Wahl von G mit Grad *r*:
  - alle Einzelfehler
  - jede ungerade Anzahl von verfälschten Bits (falls G den Faktor (u+1) enthält)
  - sowie alle Bündelfehler (Fehler, die eine Reihe von benachbarten Bits betreffen) der Länge $\leq r$.
- Beispiele für Generatorpolynome sind:
  - CRC-16: $u^{16}+u^{15}+u^2+1$
  - CRC-CCITT, HDLC: $u^{16}+u^{12}+u^5+1$
  - Ethernet: $u^{32}+u^{26}+u^{22}+u^{16}+u^{12}+u^{11}+u^{10}+u^8+u^7+u^5+u^4+u^2+u+1$
- Die Polynomdivision unter modulo-2-Arithmetik lässt sich sehr einfach direkt in der Hardware realisieren $\Rightarrow$ effiziente Umsetzung

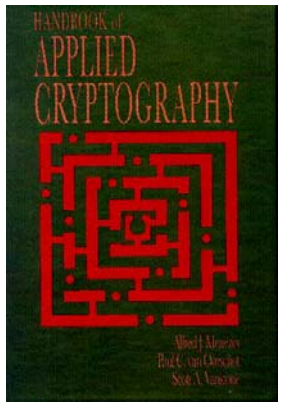
Kryptographie

Literatur

- F.L. Bauer: „Entzifferte Geheimnisse“



- H. Mel, D. Baker, S. Burnett: „Cryptography Decrypted“



- A. Menezes, P. van Oorschot, S. Vanstone: „Handbook of Applied Cryptography“ http://www.cacr.math.uwaterloo.ca/hac/

Kryptographie: Definition und Zweck

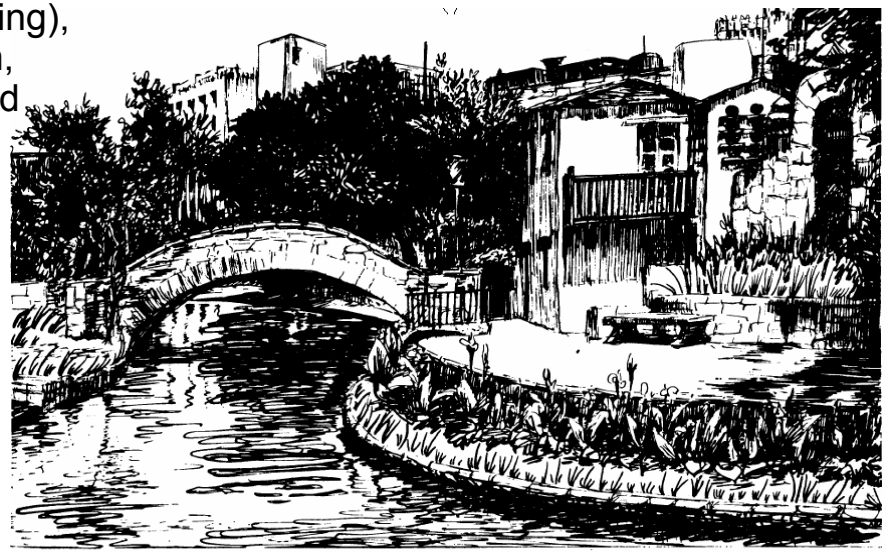
- **Kryptographie** ist die mathematische Teildisziplin, die sich mit Methoden und Mechanismen beschäftigt, Information zu sichern, etwa zum Zweck
  - des Verhinderns von Abhören (Vertraulichkeit, *confidentiality*),
  - der Sicherstellung von Datenintegrität (Sicherung gegen Veränderung, *integrity*),
  - der Authentifizierung einer Person/Institution/Gerät/Nachrichtenquelle (sie/es ist wirklich die/das, der sie/es vorgibt zu sein, *authentication*) oder einer Nachricht (sie hat die korrekten Absender-/Zeitangaben),
  - der Autorisierung von Berechtigungen (sichere Erteilung einer Erlaubnis durch A an B, Dinge zu tun, die A erlauben kann, *authorisation*),
  - der Zertifizierung (Beglaubigung, Sicherstellung von Korrektheit durch eine vertrauenswürdige Person/Institution, *certification*),
  - der dauerhaften Verpflichtung (nicht-Entlassung aus eingegangenen Verpflichtung über vetrauenswürdigen Dritten, *non-repudiation*).
- **Kryptanalyse** ist die mathematische Teildisziplin, die zum Ziel hat, die Verfahren der Kryptographie wirkungslos zu machen.
- **Kryptologie** ist der Oberbegriff für Kryptographie und Kryptanalyse.

Meilensteine Kryptographie

- Seit tausenden von Jahren immer neue Verfahren, die mehr oder weniger sicher sind
- Bislang nur ein einziges sicheres Verfahren entdeckt: **Vernam-Kodierung** als „one-time-pad“), das neue, völlig zufällige Schlüssel für jede Nachricht verwendet (mit mindestens der Nachrichtenlänge als Schlüssellänge).
- Neuer Schub durch **Digitalisierung**, die die identische Replikation von Nachrichten erlaubt (Kopie ist nicht mehr vom Original zu unterscheiden)
- Arbeiten von Feistel (IBM) Anfang der siebziger Jahre führten zur Feistel-Kodierung und später zum Data Encryption Standard DES (1977), bis heute (als Triple-DES) in Gebrauch (gegenwärtig Ablösung durch Advanced Encrpytion Standard AES)
- Wesentlicher zweiter Durchbruch war das Postulat der asymmetrischen (public key) Verschlüsselung nach Diffie-Hellmann, die den Schlüsselaustausch über abhörbare Kanäle erlaubte (public key encryption) (1976)
- Erste praktische Umsetzung dieses Verfahren durch Rivest, Shamir und Adleman (RSA) (1978)
- Nächster Meilenstein war die Verabschiedung der Standards für die digitale Unterschrift (Signatur), basierend auf RSA (1991) sowie die public key infrastructure (PKI) seit Anfang der neunziger Jahre, deutsches Gesetz für elektronische Signatur (1997)

Steganographie

- **Steganographie** zielt darauf ab, die bloße Existenz geheimer Information in einer Nachricht zu verbergen. Man unterscheidet in:
  - Technische Steganographie: Geheimtinte, micro-dots, …
  - Linguistische Stegonografie: a) Nachricht wird in „offener“ Nachricht verschleiert (bspw. jeder dritte Buchstaben), b) Nachricht wird in (un)sichtbarer Modifikation versteckt (Beispiel: Modifikation von Bildpunkten in einem Fernsehbild)
  - Damit aber auch möglich: Einbauen von unsichtbaren „Wasserzeichen“ (Watermarking), um Herkunft eines Dokuments zu kodieren, ohne daß der Eigentümer dies bemerkt (und ggf. vor Weitergabe löschen kann)



Beispiele Steganographie

Worthie Sir John:— Hoṗe, thăt is ye beste comfort of ye afflicted, caṅnot much, I fĕar me , heĺp you now. Thăt I would saye to you, is t̆his only: if ĕver I may be able to requite that I do owe you, stănd not upon asking me. 'Tiš not much that I can do: but̆ what I can do, beĕ ye verie sure I wille. I kn̆owe that, if d̆ethe comes, if ŏrdinary men fear it, it f̆rights not you, acc̆ounting it for a high honour, to h̆ave such a rewarde of your loyalty. Prăy yet that you may be spared this soe bitter, cuṗ. I fĕar not that you will grudge any sufferings; onl̆y if bie submission you can turn them away, 'tiš the part of a wise man. Tel̆l me, an ĭf you can, to d̆o for you anythinge that you wolde have done. Thĕ general goes back on Wednesday. Reštinge your servant to command. — R. T.

Abb. 13. Nachricht an *Sir John Trevanion: Panel at east end of chapel* slides (dritter Buchstabe nach Interpunktionszeichen)

Beispiele Steganographie

Abb. 9. Geheimzeichen für „Polizei freundlich“ und „Polizei unfreundlich“ und andere Botschaften (Mittelwesten der Vereinigten Staaten, erste Hälfte 20. Jh.)

Beispiele Steganographie

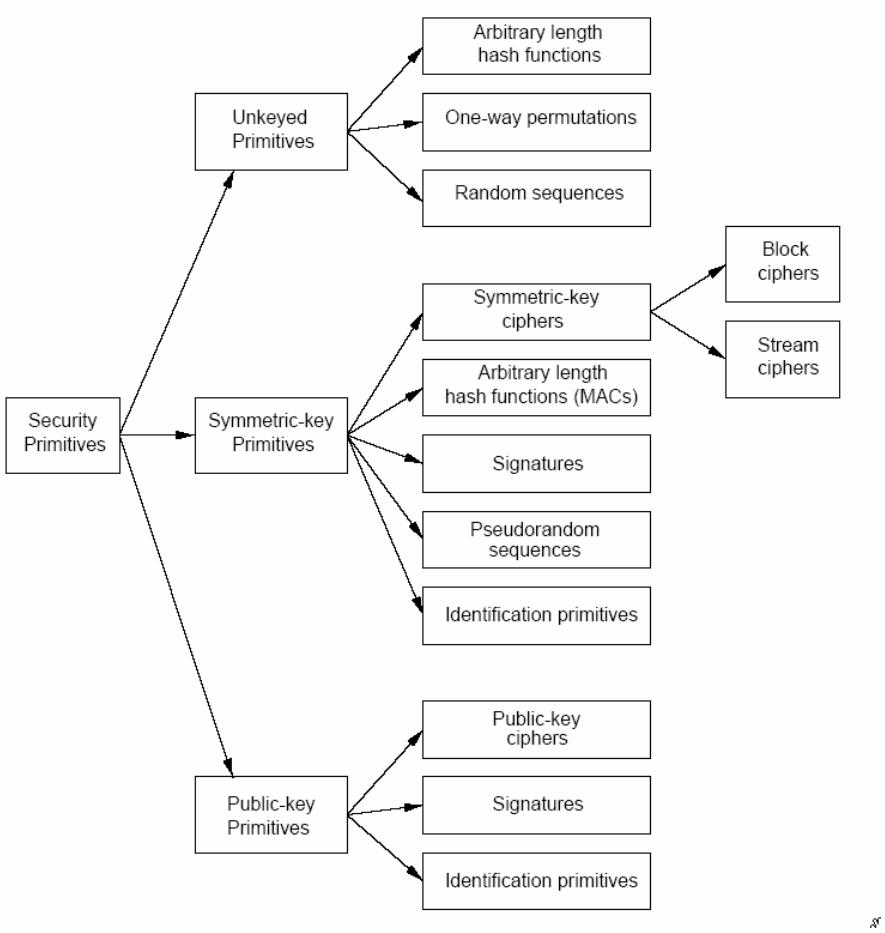
In Königsberg i. Pr. gabelt sich der Pregel und umfließt eine Insel, die *Kneiphof* heißt. In den dreißiger Jahren des achtzehnten Jahrhunderts wurde das Problem gestellt, ob es wohl möglich wäre, in einem Spaziergang jede der sieben Königsberger Brücken genau einmal zu überschreiten.

Daß ein solcher Spaziergang unmöglich ist, war für L. EULER der Anlaß, mit seiner anno 1735 der Akademie der Wissenschaften in St. Petersburg vorgelegten Abhandlung *Solutio problematis ad geometriam situs pertinentis* (Commentarii Academiae Petropolitanae 8 (1741) 128-140) einen der ersten Beiträge zur Topologie zu liefern.

Das Problem besteht darin, im nachfolgend gezeichneten Graphen einen einfachen Kantenzug zu finden, der alle Kanten enthält. Dabei repräsentiert die Ecke vom Grad 5 den Kneiphof und die beiden Ecken vom Grad 2 die Krämerbrücke sowie die Grüne Brücke.

„nieder mit dem sowjetimperialismus“

Strukturierung kryptographischer Grundtechniken



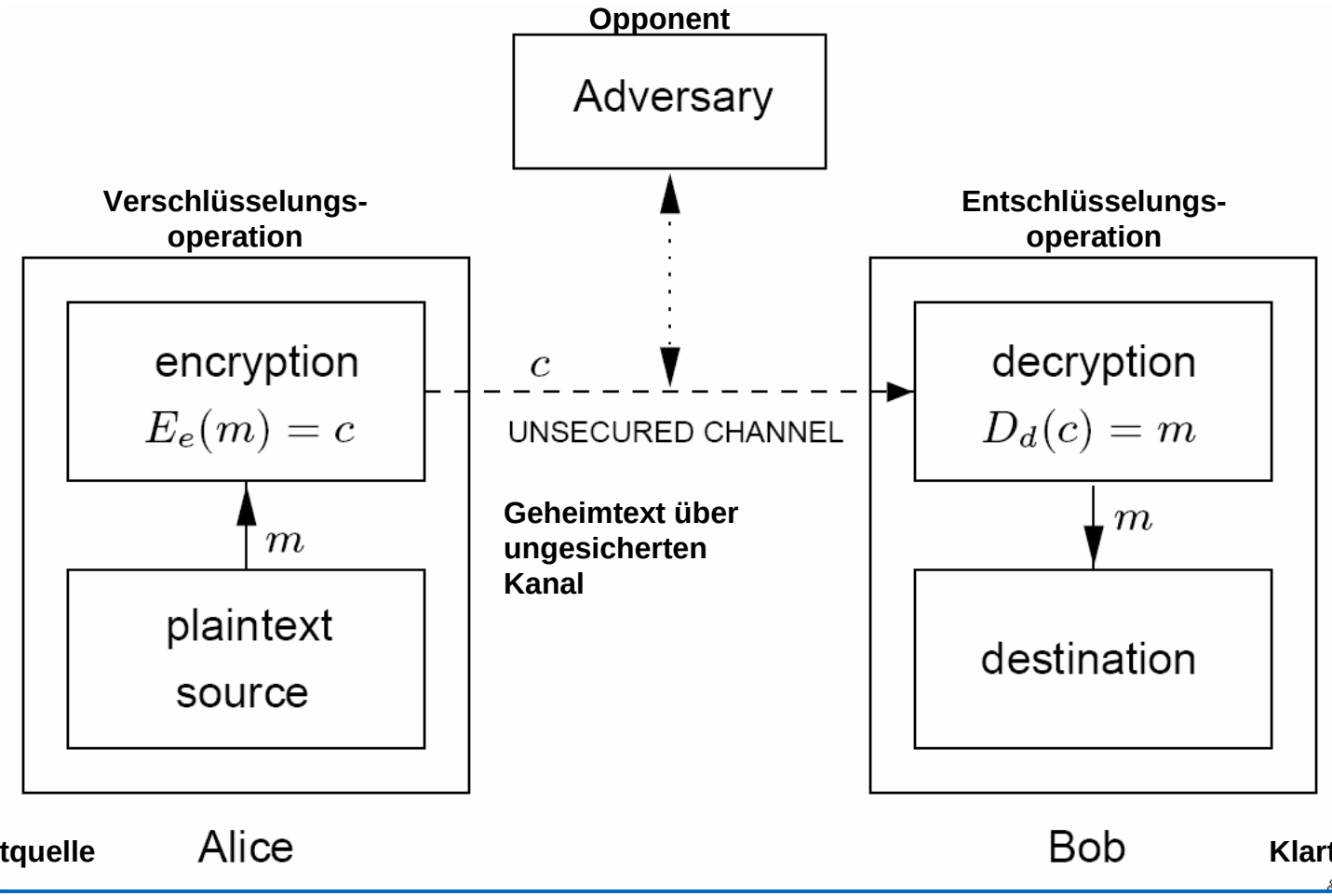
Kryptographie

Symmetrische Verschlüsselungsverfahren

Definitionen

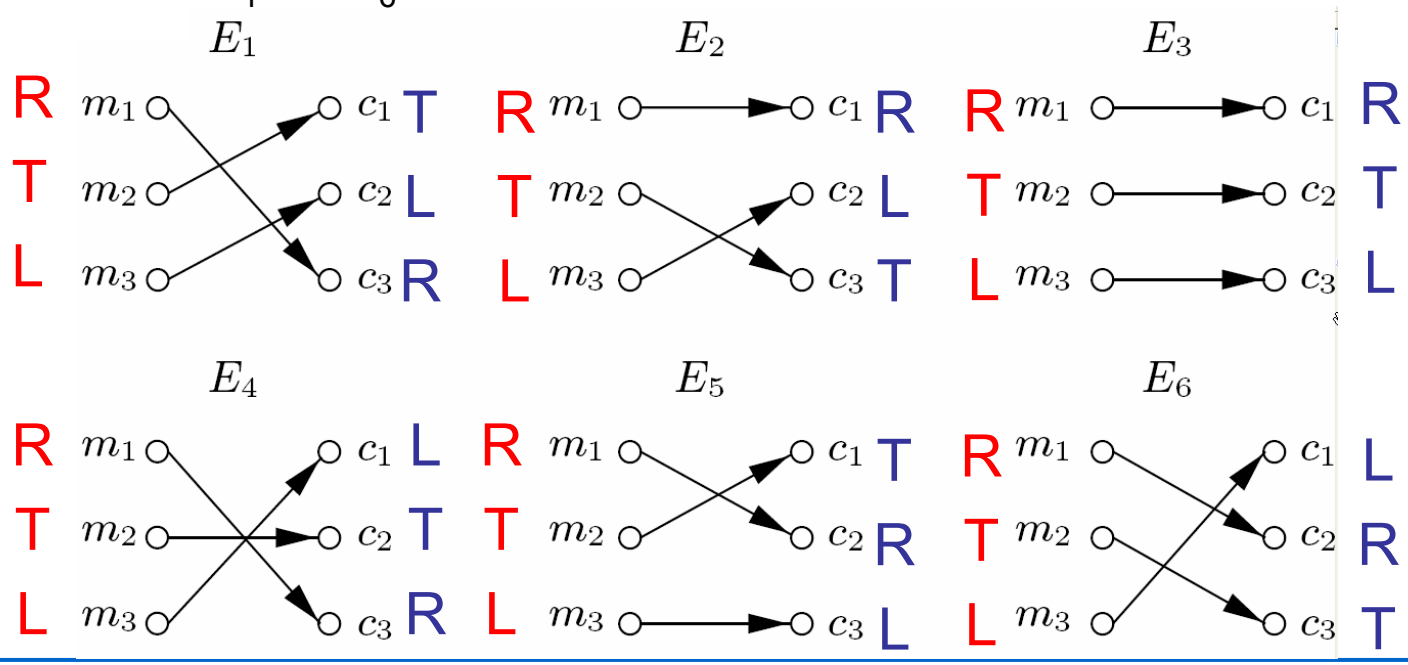
- ***M***: Zeichenmenge für Klartext, z.B. alle Buchstaben des Alphabets A…Z, oder 8-bit Binärzahlen x00…xFF
- ***M****: Klartextraum, alle möglichen Folge aus Elementen von ***M***
- ***C***: Zeichenmenge für Geheimtext, z.B. alle Buchstaben des Alphabets A…Z, oder 8-bit Binärzahlen x00…xFF
- ***C****: Geheimtextraum, alle möglichen Folge aus Elementen von ***C***
- ***K***: Schlüsselmenge aus allen möglichen Schlüsseln $e \in \mathbf{K}$
- E_e: Chiffriertransformation, in Abhängigkeit vom Schlüssel *e* wird Klartext auf Geheimtext abgebildet, also E_e: $\mathbf{M^*} \rightarrow \mathbf{C^*}$
- D_e: Dechiffriertransformation, in Abhängigkeit vom Schlüssel *e* wird Geheimtext auf Klartext abgebildet, also E_e: $\mathbf{C^*} \rightarrow \mathbf{M^*}$

Einordnung in kryptographisches Gesamtsystem



Beispiel für Codiervorschrift I

- **Permutierung**: Vertauschen (*Transposition*) der Folge von Elementen der Klartextelemente ergibt Geheimtextnachricht. Hat die Menge der Klartextsymbole 3 Elemente, ergeben sich sechs Möglichkeiten zur Anordnung Geheimtext-symbole. Sechs Schlüssel (1…6) ergeben sechs unterschiedliche Chiffrier-transformationen E_1 bis E_6.



Beispiel für Codiervorschrift II

- **monoalphabetische Substitution:** Ersetzung von Buchstaben z.B. durch jeweils den *e*-nächsten im Alphabet, wo der Schlüssel eine Zahl zwischen 1 und 25 ist. Beispiel für *e* = 3:

- Klartext ***M**** und Geheimtext ***C****:
  - ***M**** = THISC IPHER ISCER TAINL YNOTS ECURE
  - ***C**** = WKLVF LSKHU LVFHU WDLQO BQRWV HFXUH
- Weiteres hier illustriertes Konzept **Block-Cipher (Blockverschlüsselung)**: Klartext wird in Gruppen zu (hier fünf) Symbolen zusammengefaßt, die jeweils zusammen kodiert werden, auch in Abhängigkeit von der Stellung im Block (bei der hier gewählten Transformation allerdings Zeichen für Zeichen).
- Gegenteil **Stream-Cipher (Stromverschlüsselung)**: Es werden keine Blöcke gebildet, jedes Zeichen wird ohne Berücksichtigung anderer Zeichen und/oder der Position im Block für sich betrachtet und verschlüsselt.

Beispiel für Codiervorschrift III

- **polyalphabetische Substitution:** Blockbildung und **Ersetzung** der Buchstaben im Block in Abhängigkeit von ihrer Stellung im Block. Sei Blocklänge drei, dann werde *erster* Buchstabe im Block durch den dritten Folgebuchstaben im Alphabet ersetzt, der *zweite* durch den siebten Folgebuchstaben und der *dritte* durch den zehnten Folgebuchstaben.

- Klartext ***M**** und Geheimtext ***C****:

  M* = `THI SCI PHE RIS CER TAI NLY NOT SEC URE`

  C* = `WOS VJS SOO UPC FLB WHS QSI QVD VLM XYO`

Beispiel für Codiervorschrift IV

- **Transposition auf Blöcken:** Blockbildung und **Vertauschung der Reihenfolge** der Buchstaben im Block (siehe Beispiel I). Beispielsweise wird erster Buchstabe letzter, der zweite und dritte werden jeweils eine Position nach links verschoben (E_1 in Beispiel I).

- Klartext ***M**** und Geheimtext ***C****:

  M* = `THI SCI PHE RIS CER TAI NLY NOT SEC URE`

  C* = `HIT CIS HEP ISR ERC AIT LYN OTN ECS REU`

Produktverschlüsselung

- Einfache Ersetzung (Substitution) und Vertauschung (Transposition) im Block sind für sich genommen *keine* brauchbaren Verschlüsselungs-techniken: Durch die einfache Analyse von Zeichenauftrittshäufigkeiten (und Vergleich mit denen der Klartextsprache) läßt sich der Code meist einfach brechen.
- Die Kombination (Hintereinanderausführung, Produktbildung) dieser beiden Techniken führt allerdings zu starker Verschlüsselung.
- Eine Produktverschlüsselung ist also allgemein $c_i = E_{e1}(E_{e2}(\ldots(E_{et}(m_i)\ldots))$, wobei $t \geq 2$
- Die Hintereinanderausführung von Substitution und Vertauschung wird als *Runde* bezeichnet.
- Heute üblich ist die mehrfache Ausführung von Runden auf dem Klartext, um zum Geheimtext zu kommen.
- Für jede Runde wird ein getrenntes Schlüsselpaar (beim Sender und Emp-fänger) genutzt.

Beispiel: Produktverschlüsselung

- 8-bit Klartextstrings ($\boldsymbol{M} = \{0,1\}$; $\boldsymbol{M^*} = \{0,1\}^8$)
  Beispiel: $m = m_1m_2m_3m_4m_5m_6m_7m_8 = 10110001$
- 8-bit Geheimtextstrings ($\boldsymbol{C} = \{0,1\}$; $\boldsymbol{C^*} = \{0,1\}^8$)
- 8-bit Schlüssel $\boldsymbol{K} = \{0,1\}^8$
  Beispiel: $k = k_1k_2k_3k_4k_5k_6k_7k_8 = 10001001$
- **1. Schritt**: *Substitution*, hier ausgeführt durch modulo-2 Addition (= bitweise XOR) der 8-bit Zahl des Schlüssels k zum Klartextzeichen m, bezeichnet mit $c_0 = E_k(m) = k \oplus m$
  mit den Beispielzahlen: $c_0 = (10110001) \oplus (10001001) = (00111000)$
- **2. Schritt**: *Transposition*, Vertauschung der Bitfolge in dem gerade gewonnenen Zeichen c_0, abhängig oder unabhängig vom Schlüssel. Sei $E(m_1m_2m_3m_4m_5m_6m_7m_8) = m_4m_1m_5m_8m_3m_2m_6m_7$, dann wird mit den Beispielzahlen: $c = E(c_0) = E(E_k(m)) = (10101000)$
- Nach einer Runde wird in diesem Beispiel also aus 10110001 die Bitfolge 10101000

Enigma

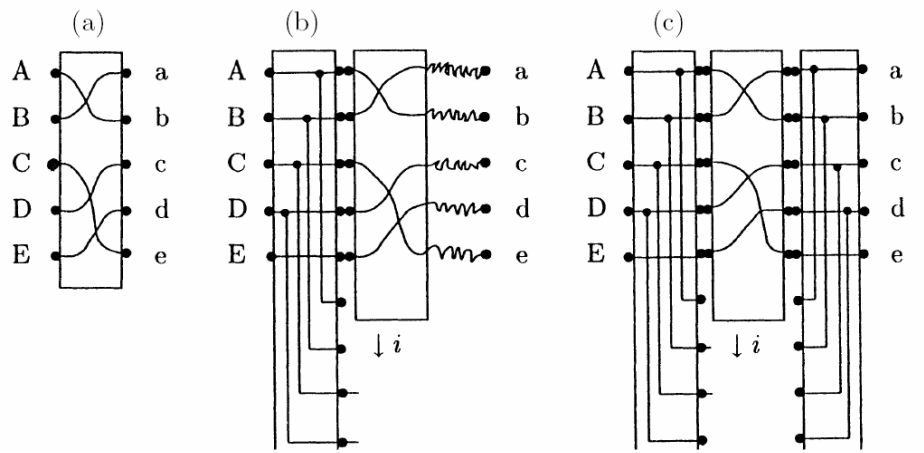
Achtung! Schlüsselmittel dürfen nicht unversehrt in Feindeshand fallen. Bei Gefahr restlos und frühzeitig vernichten.

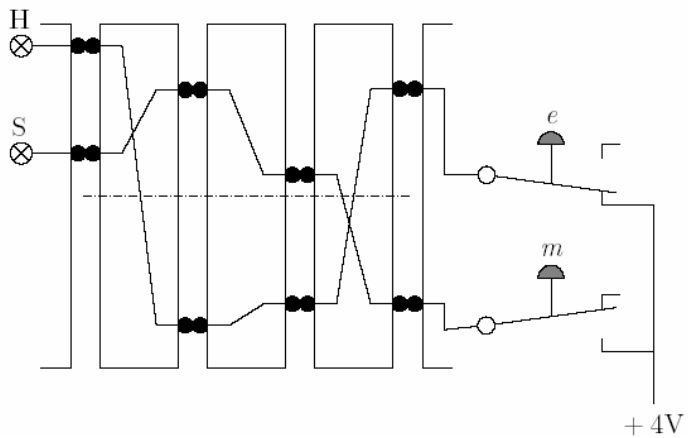
	Monats-tag	Walzenlage			Ringstellung			Steckerverbindungen										Zusatzsteckerverbindungen		Kenngruppen			
								an der Umkehrwalze				am Steckerbrett 1 2 3 4 5 6 7 8 9 10						1 bis 200	2 bis 260				
2744	31	III	V	IV	17	11	04	BG	TV	TW	BI	UY	GP	CK	JQ	DL	RV	EM	AH	NS	FQ	kim	pwh	sbx	csw
2744	30	I	IV	V	08	17	21			LS	DH	MT	EO	AP	UZ	FQ	WY	BK	GR	CI	JN	uaq	omn	ume	duf
2744	29	V	II	III	11	14	05	UX	IM	DO	JW	CN	IV	PZ	BM	HU	AL	FR	KX	EQ	GT	don	cqo	xum	bpg
2744	28	II	IV	V	02	20	16	SV	EP	NT	HK	BW	EP	LQ	AU	CY	FJ	CX	GI	DZ	MR	lui	pyg	sby	dtq
2744	27	III	V	IV	18	13	22	NQ	CO	HM	GV	KZ	AI	DQ	NR	ES	BL	OU	FT	CP	JY	cmy	fqr	sci	bur
2744	26	I	III	II	24	10	01	GL	BP	GW	AQ	MO	FV	FS	DI	RU	JZ	BN	EH	KT	CL	kbj	yaq	udm	cnz
2744	25	IV	I	III	04	25	23	HK	AD	LT	DR	QX	AG	IN	EU	BJ	KP	FW	CM	SZ	HO	kqx	yar	vdb	coa
2744	24	V	III	I	09	19	06			GL	MY	CR	HN	JX	DT	AF	PU	IQ	BO	EW	KS	cmz	aoj	zod	auh
2744	23	IV	I	V	15	03	19			IT	DV	HQ	AJ	MU	EX	KO	OS	FY	LN	BP	GZ	kra	yas	xun	cob
2744	22	I	V	III	12	26	07			EY	JL	AK	NV	FZ	GT	HP	MX	BQ	GS	DW	IO	jdm	uhf	xuo	bph
2744	21	III	IV	II	15	09	12			JP	DY	QS	HL	AE	NW	CU	IK	FX	BR	MV	GO	jpf	aok	iys	btx
2744	20	IV	II	I	02	22	05			HT	NP	AM	DX	GJ	KQ	BS	OV	ER	CW	IU	FL	boy	wac	uow	cse
2744	19	V	I	II	08	19	17			GM	OX	BT	QU	DP	HJ	FK	SW	AN	EL	CY	IR	xjc	wad	unj	ctd
2744	18	III	IV	I	11	21	01			KW	IP	DM	SV	JR	CX	EN	AZ	QT	BU	FH	GY	kpn	rzi	vcm	bpo
2744	17	I	V	II	18	23	14			BV	HW	AR	NX	DS	PT	CZ	FI	LY	EJ	GK	MQ	kdx	crq	vcn	cod
2744	16	III	IV	V	16	04	07			LU	CV	FM	KR	BY	GN	QW	DJ	PS	AO	EI	HX	lgx	jri	uob	aur
2744	15	V	III	IV	24	13	10			HZ	NQ	AD	TV	IX	KM	BG	LO	CE	RY	JU	PP	wpt	vhy	zoe	aus
2744	14	I	IV	II	06	20	25			FN	UY	CJ	IW	LP	AS	DK	GQ	MO	BZ	ET	HR	wog	hxi	zxi	bpi
2744	13	III	II	I	03	26	18			KR	IZ	AT	NV	BH	MP	CG	OY	ES	DF	UW	LQ	lqv	iqb	zsy	coe
2744	12	II	IV	III	04	11	15			DT	JV	HS	CI	AY	KU	EN	FQ	LR	BW	MP	GO	zic	myt	zof	dtr
2744	11	V	I	IV	16	07	02			JS	PW	AV	QX	DN	IZ	KM	CO	EG	PL	HY	BR	inf	zbm	krz	dug
2744	10	IV	III	II	20	12	14			FS	CQ	JO	PR	AW	HV	EZ	KN	DU	GT	IL	BY	ink	acu	zxj	cnu
2744	9	III	II	V	06	18	10			HK	TZ	MX	LW	GQ	AD	NY	BE	CS	JP	RV	IO	efm	pmi	snw	cof
2744	8	V	I	III	01	21	17			GU	SW	BP	RX	EV	OT	LQ	CH	IP	KY	JM	NZ	imy	rjw	tjm	cog
2744	7	II	V	I	25	08	23			CX	AZ	DV	KT	HU	LW	GF	EY	MR	PQ	IN	OS	inv	rkc	snx	bpj
2744	6	IV	II	V	13	26	03			DV	LP	NQ	GZ	OS	PK	EW	MR	IT	HX	UY	BJ	yvu	hsb	swq	aut
2744	5	III	I	II	24	19	22			SY	EK	NZ	OR	CG	JM	QU	PV	BI	LW	TX	DP	seu	iqe	swr	auv
2744	4	II	IV	I	17	05	09			BD	GV	AX	KP	EM	PN	CW	RU	HO	JT	IL	QS	zfj	hxj	zxk	dpt
2744	3	V	II	IV	20	16	11			JT	NW	DU	EO	KV	BY	FS	HQ	IM	LX	GP	CR	clx	zbn	zxa	buk
2744	2	II	III	V	14	03	19			RW	OQ	GI	AZ	BJ	MS	CU	DH	FY	BF	LV	TX	ljs	jre	zpq	coh
2744	1	III	I	IV	18	24	15			NP	JV	LY	IX	KQ	AO	DZ	CR	FT	EM	GS	HW	plf	dgw	tjn	cnv

- **Elektromechanische Realisierung** der kryptographischen Primitiva Substitution und Transposition mit Hilfe von Schlüsseln in der **Enigma**
- Prinzip war bekannt, trotzdem Entschlüsselung (fast) unmöglich

Arbeitsprinzip Enigma

- Schlüssel repräsentiert in unterschiedlichen Walzenlagen, Ringstellungen und Steckverbindungen
- Druck auf Taste (Klartextzeichen) führt zu Aufleuchten korrespondierender Lampe (korrespondierendes Geheimtextzeichen)
- Nach Tastendruck Weiterschaltung der Walzen





Stromverschlüsselung (Stream Cipher)

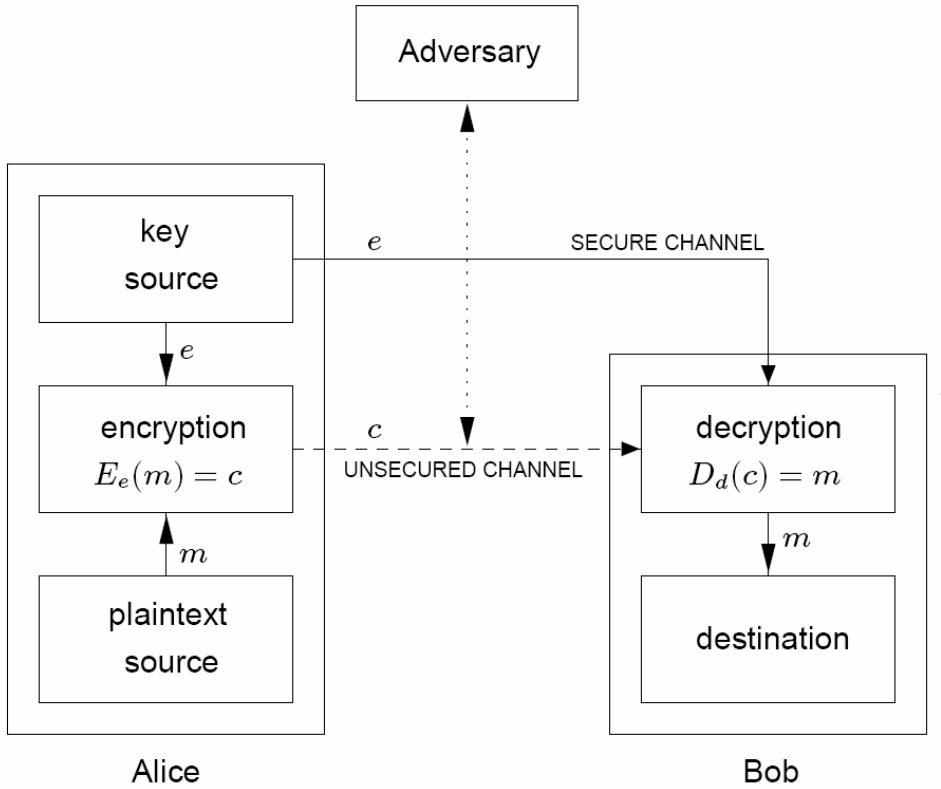
- Sonderfall der Blockverschlüsselung mit Blocklänge 1, also nur Transposition und keine Vertauschung
- Jedes Zeichen wird für sich betrachtet und verändert, deshalb keine Speicherung erforderlich
- Günstig für fehleranfällige Übertragungsstrecken, weil keine Fehlerfortpflanzung
- Verschlüsselung des Klartexts $m_1m_2m_3\ldots m_i\ldots$ mit *Stromschlüssel k* = $k_1k_2k_3\ldots k_i\ldots$ durch Substitution (über Addition, XOR) jedes Zeichens zum Geheimtextstrom $c_1c_2c_3\ldots c_i\ldots$ wobei: $c_i = E_{ki}(m_i)$ (Chiffrierabbildung ändert sich von Zeichen zu Zeichen).

Vernam-Codierung

- Spezialfall der Stromverschlüsselung
  - Klartextalphabet $\boldsymbol{M} = \{0,1\}$ und auf den Klartextfolgen der Zusammenhang Klartext und Geheimtext: $c_i = k_i \oplus m_i$
  - Also: Jedes Bit des Klartexts wir mit dem korrespondierenden Bit des Schlüssels „verodert“.
- Man kann zeigen: Wenn der Schlüssel eine *echte Zufallsfolge* ist und er nur ein einziges Mal verwendet wird, ist dieses System der Codierung **nicht zu brechen**. Aber
  - Schlüsseltransport ist aufwendig
  - Die Erzeugung echter Zufallsfolgen (keine Pseudofolgen!) ist schwierig

Symmetrische Verschlüsselung

Sender und Empfänger haben jeweils (den gleichen) Schlüssel(satz)



Unabdingbar: Schlüsselaustausch über **gesicherten Kanal.**

Vor- und Nachteile symmetrischer Verschlüsselung

- **Vorteile:**
  - Operationen sind sehr einfach, mit heute sehr billigen Hardwarelösungen lassen sich GBytes/s verschlüsseln, Software ebenfalls bereits sehr schnell
  - Schlüssel sind kurz (56/64 bis 256 bit)
  - Breiter Verwendungsbereich für gesicherte Übertragung, für digitale Signatur, für Gewährleistung Dokumentenechtheit
- **Nachteile**:
  - Schlüssel sind auf beiden Seiten geheim zu halten
  - Management der vielen Schlüsselpaare in großen Netzwerken ist schwierig und erfordert für Effizienz eine absolut vertrauenswürdige Instanz (trusted third party, TTP)
  - Schlüssel sollten sehr häufig gewechselt werden, verursacht Aufwand

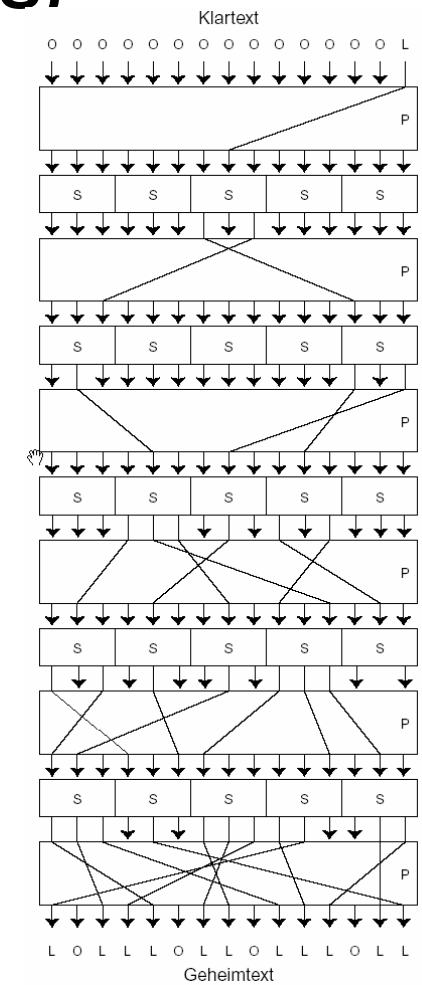
Kryptographie

Standardverfahren der symmetrischen Verschlüsselung

Data Encryption Standard (DES)

Grundlage des DES: Produktchiffrierung nach Feistel („Lucifer“-Blockverschlüsselung, 1973):

- Folge von Transpositionen im Block (Permutationen P) und Substitutionen in „*Substitutionsboxen*“ S
- Hier: Klartext mit einem **L** wird umgesetzt in Geheimtext mit elf **L**
- Wichtig: Kleiner Unterschied im Klartext muß zu großem Unterschied im Geheimtext führen
- Wesentliche Aufgabe der Substitution ist es, Konfusion zu erzeugen, d.h. den Zusammenhang zwischen Schlüssel und Geheimtext zu verschleiern
- Wesentliche Aufgabe der Transposition ist es, die Klartextbits so über den Geheimtext zu „verschmieren“, daß sich die Textredundanz möglichst gleichmäßig über den Geheimtext verteilt

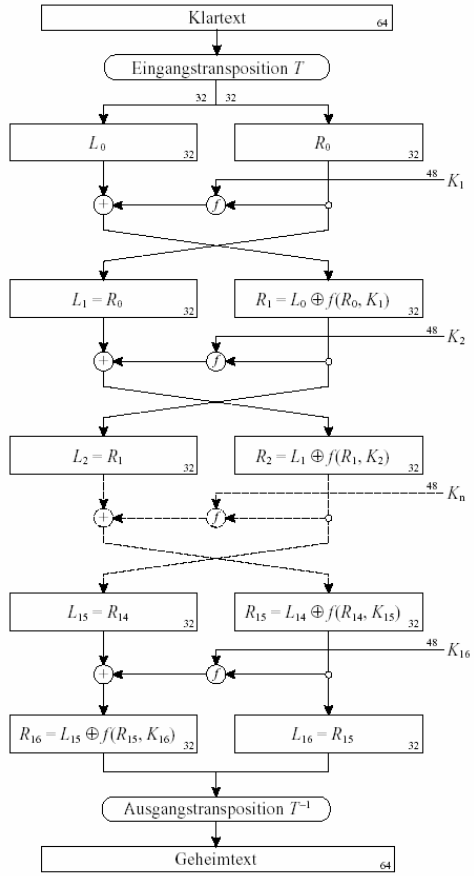


DES-Chiffrierung

1. 8 Byte Klartextblock wird schlüsselunabhängiger Eingangstransposition T unterworfen
2. Aufteilung in zwei 4-Byte Blöcke L_0 und R_0
3. Dann 16 Runden mit $L_i = R_{i-1}$ und $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$ (mit „⊕“:Addition modulo-2, XOR)
4. Schlüsselunabhängige Ausgangstransposition T^{-1}

Die K_i sind Schlüssel zu je 48 bit, die aus dem Chiffrierungsschlüssel zu 56 bit durch dessen Aufteilung in zwei Teilschlüssel à 28 bit und rundenabhängiger Verschiebung um ein oder zwei bit in die 48 bit Schlüssel transformiert werden.

Dechriffierung: Exakt gleiches Vorgehen wie bei der Chiffrierung. Schlüssel kommen in umgekehrter Reihenfolge zum Einsatz. (Vertauschung und Verarbeitung nach obiger Vorschrift sind involutorisch). Algorithmus ist schlüssel-symmetrisch, zum Chiffrieren und Dechiffrieren dient der gleiche Schlüssel.

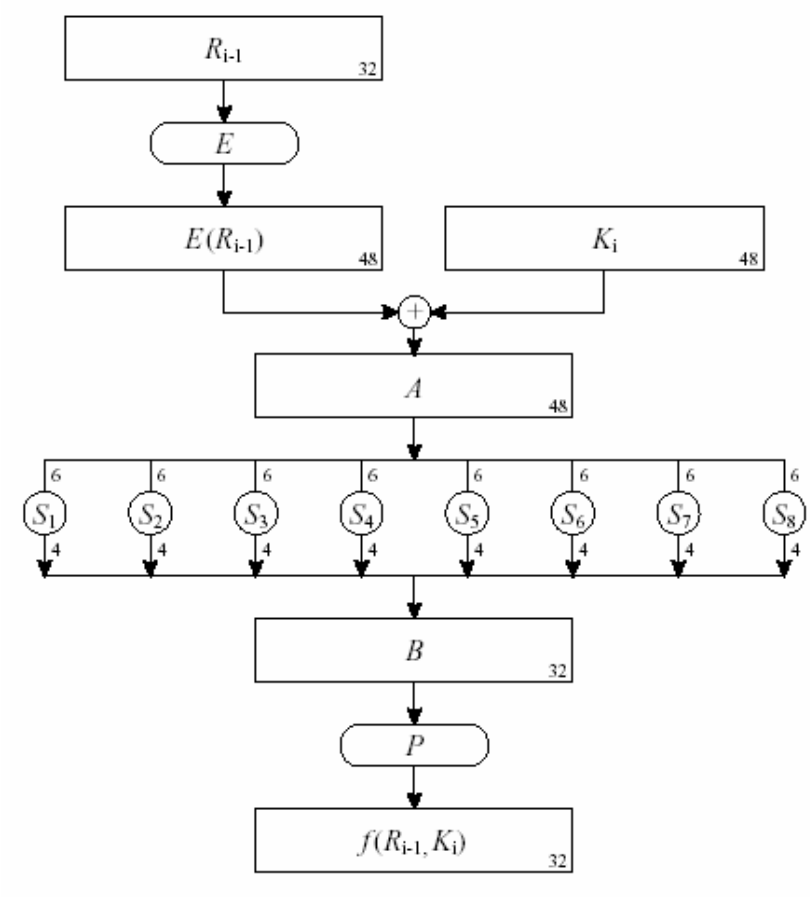


DES: Verschlüsselungsfunktion

- Expansion des 32-bit-Blocks R_{i-1} über das Expansionsmodul E in 48-bit-Blöcke und Addition mod-2 zu K_i.
- Aufspaltung durch A in 8 sechs-bit-Blöcke, die in die Substitutionsboxen S1...S8 eingespeist werden
- Jede S-Box definiert über Substitutionstabelle mit 16 Spalten und 4 Zeilen, z.B.:

S_1:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

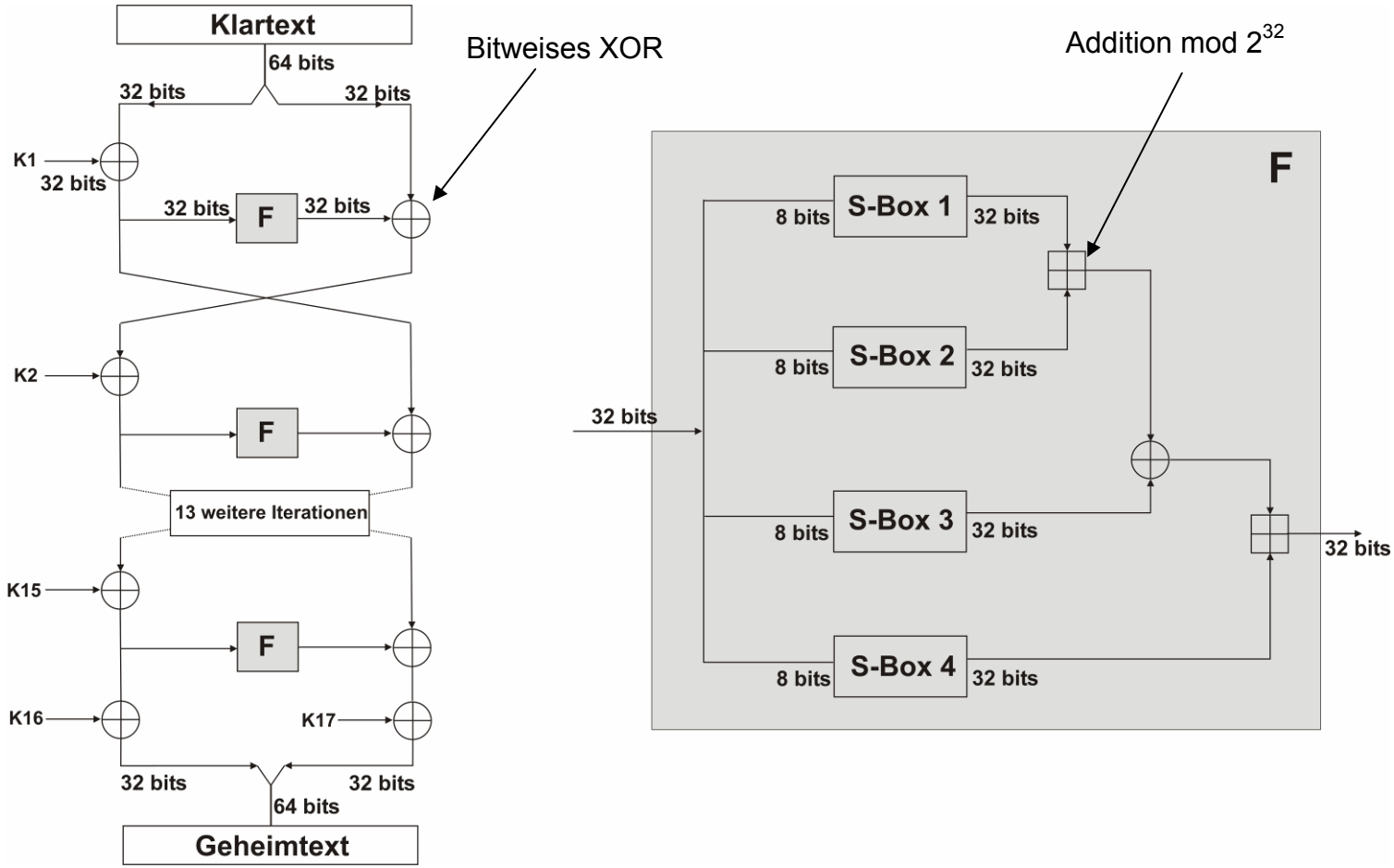
- Bits 1 und 6 (Zählung von links) bestimmen die Zeile, Bits 2...5 die Spalte der Substitutionstabelle, Ausgabe sind 4-bit-Wörter
- Zusammenfassung in B und abschließende Transposition P



DES und Blowfish

- Vorteile von Verfahren wie DES: sehr einfache Umsetzung in Hardware → Stromverarbeitung von Daten mit hoher Geschwindigkeit möglich
- Probleme bei der Verwendung von DES:
  1. Patente lassen keine unbeschränkte Nutzung zu
  2. 56-bit-Schlüssel heute nicht mehr sicher genug
- Lösung:
  - a) Triple-DES (3DES): Mehrfachausführung von DES
  - b) Andere Verfahren, z.B. Blowfish: patentfrei.

Blowfish (von B. Schneier)



Zusammenfassung: symmetrische Verschlüsselung

- **Allgemein:**
  - Unterscheidung in Blockverschlüsselung und Stromverschlüsselung
  - Sicherheit rührt im wesentlichen von Schlüssel(wechsel) her, Geheimhaltung des Verfahrens bietet keinen (oder höchstens vorübergehenden) Schutz
- **Für DES und ähnliche Verfahren:**
  - Eine Verschlüsselungsrunde besteht aus Transposition und Substitution
  - Zur Definition eines bestimmten Verfahrens gehören:
    - Initiale Aufteilung der Klartextbits und Festlegung Datenfluß
    - Schlüssel bzw. Schlüsseltabelle, falls unterschiedliche Schlüssel in jeder Runde
    - Definition der Substitutionsboxen (S-Boxen), der Verschaltung der Expansionsboxen (DES: E und A) und der Transpositionsglieder (Permutationstabellen, DES: B und P)

Kryptographie

Asymmetrische Verschlüsselungsverfahren

Asymmetrische Verschlüsselungsverfahren

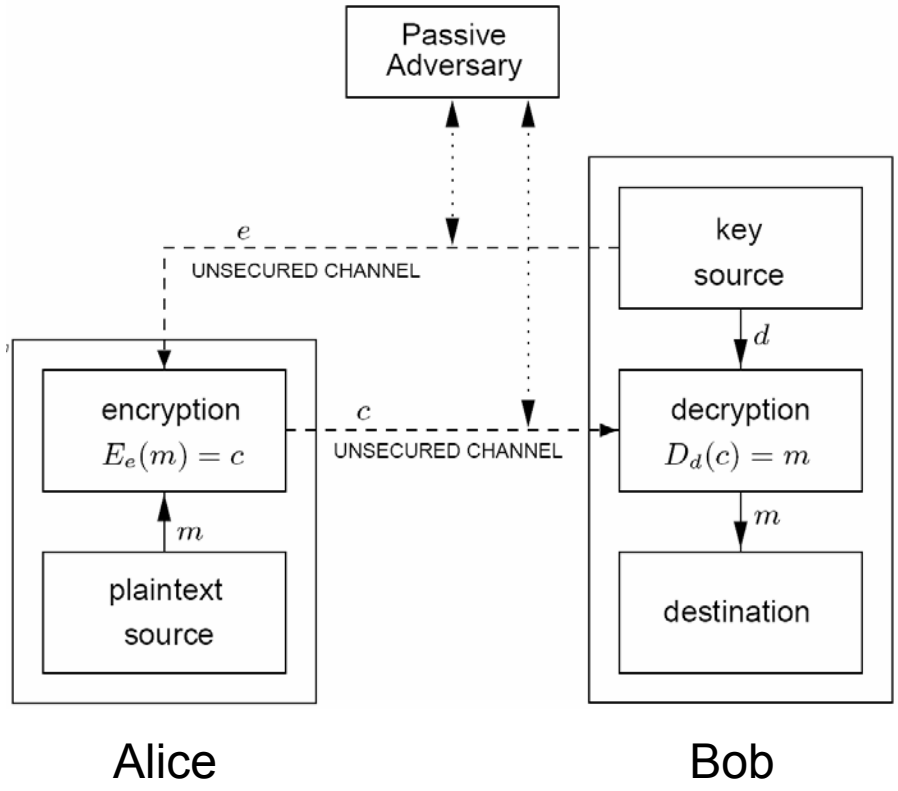
- Komplementär zur Technik der symmetrischen Verschlüsselung, die einen sicheren Kanal zur Schlüsselübertragung benötigt, gibt es die Technik der asymmetrischen (public-key) Verschlüsselung, die einen Schlüsselaustausch über unsichere Kanäle ermöglicht.
- Wichtige Einsatzgebiete für PKE sind:
  - Effiziente Schemata für digitale Unterschriften
  - Bereitstellung weiterer grundlegender Techniken für eine public-key-infrastructure (PKI)
  - **Schlüsselmanagement**

Motivation und Grundlagen

- **Grundidee:**
  - Erzeugung zweier Schlüssel, des **public key *e*** und des **private key *d***
  - Mit Hilfe von *e* kann eine Nachricht kodiert werden, und mit Hilfe von *d* wird sie dekodiert
  - Wird *e* frei verteilt, kann damit jedermann Nachrichten *ver*schlüsseln, aber nur der Besitzer von *d* kann sie wieder *ent*schlüsseln
  - *Analogie*: Paketautomaten von DHL – Jeder kann von DHL einen Schlüssel (Zahlenkombination) bekommen, mit der er Pakete in den Automaten legen darf. Nur der Abholer hat den Schlüssel, um sie wieder aus dem Automaten zu entnehmen
- **Formal**:
  - E_e Chiffriertransformation mit $e \in \boldsymbol{K}$, *wo* ***K***: Schlüsselmenge
  - D_d *Dec*hiffriertransformation mit $d \in \boldsymbol{K}$
  - Ein Paar (E_e , D_d) muß dann die Eigenschaft haben, daß es nicht möglich ist, den Klartext $m \in \boldsymbol{M^*}$ zu finden, wenn der Geheimtext *c* gegeben und die Transformation E_e ebenfalls bekannt ist ($E_e(m) = c \in \boldsymbol{C^*}$)
  - Also entscheidende Forderung: Wenn *e* gegeben, darf daraus *d* nicht ableitbar sein. Wenn aber *d* gegeben, ist daraus mit *e* chiffrierte Nachricht zu entschlüsseln

Ablauf der asymmetrischen Verschlüsselung

- Alice will Bob abhörsicher eine Nachricht senden
- Dazu erzeugt Bob ein Schlüsselpaar (*e*,*d*) und schickt *e* über einen ungesicherten Kanal an Alice (*d* muß er geheimhalten)
- Alice verschlüsselt ihre Nachricht *m* mit *e* und kann sie jetzt über einen ungesicherten Kanal an Bob senden
- Niemand außer Bob kann *m* entschlüsseln, auch Alice nicht
- Will B an A eine Nachricht senden, läuft der Vorgang komplementär ab
- Grundlage von SSH, https (SSL)



Idee der asymmetrischen Verschlüsselung

- Schlüsselerzeugung durch Verwendung einer Funktion, deren Funktionswert sehr leicht zu berechnen, aber deren Umkehrung rechnerisch aufwendig ist (sogenannte **Einweg-Funktionen**).
- Beispiele:
  i. Primfaktorenzerlegung einer großen Zahl ist schwer, während die Multiplikation zweier Zahlen leicht ist (siehe RSA).
  ii. Bestimmung der Parameter des diskreten Logarithmus.
- Wichtig ist, dass die Funktionen jedoch mit Zusatzinformationen wieder einfach zurückzurechnen ist (sogenannte Falltürfunktionen / trap door function).

Beispiel für Einweg-Funktionen (und Falltüren)

- Die Verschlüsselung eines Textes wird wie folgt vorgenommen: für jeden Buchstaben X wird irgendein Name Name, der mit diesen Buchstaben beginnt aus dem Telefonbuch einer Stadt Y genommen; die dazugehörige Telefonnummer gilt als Ergebnis der Verschlüsselung.
- Beispiel: Text: „info“, Stadt „München“:

i	→	Ivanov	→	4982063
n	→	Naumann	→	8560773
f	→	Fendel	→	5377204
o	→	Ontler	→	3305153

- Eine Entschlüsselung ist nahezu unmöglich, wenn das Telefonbuch nur in Papierform vorhanden ist.
- Besitzt man jedoch die Möglichkeit zur maschinellen inversen Suche, so ist die Entschlüsselung sehr einfach.
- **Anmerkung:** Prinzipiell ist die Funktionsumkehrung von Einweg-Funktionen möglich, allerdings mit hohem Rechen- und Speicheraufwand. Aufgrund des technischen Fortschrittes werden aber früher „unlösbare“ Aufgaben durchführbar ⇒ die Parameter zur Verschlüsselung müssen also ständig den neuen Gegenheiten angepasst werden.

Beispiel Einwegfunktion (RSA-Problem):

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(X)	0	1	8	12	4	5	6	13	2	9	10	11	3	7	14

- $f(X) = X^e \bmod N$
- $p = 3, q = 5, N = pq = 15$
- $X = \{0,1, 2, 3, ..., 14\}$
- $e = 3$
- $f(X) = X3 \bmod 15$

Kryptographisches Hilfsmittel: Modulare Inverse

- Analog zur multiplikativen Inversen aus der Schulmathematik (1/8 ist die Inverse zu 8 bezüglich der Multiplikation: 1/8 * 8 = 1) ist auch die Inverse *d* einer Zahl *e* bezüglich dem Modulus *N* derart definiert, daß *e***d* mod *N* = 1
- So ist beispielsweise 7 die Inverse von 3 bezüglich Modulo 20, denn 3*7 mod 20 = 1.
- Kryptographisch interessante Eigenschaft: das Ergebnis des Produktes einer Zahl *x* (kleiner dem Modulus *N*) mit dem Produkt eines inversen Paares (*e*,*d*) ist die Identität.
- Beispiel: (*x**3) mod 20 = *y* und *y**7 mod 20 = *x*
- Man kann also die zu verschlüsselnde Zahl *x* mit *e* multiplizieren und das Ergebnis modulo *N* übertragen.
  Zur Entschlüsselung wird *x* mit *d* multipliziert und mod *N* berechnet.

Kryptographisches Hilfsmittel: Modulare Inverse

- Beispiel zur modularen Inversen: e = 22, d = 23, N = 101
- Es ist: 22*23 mod 101 = 506 mod 101 = 1
- Zu verschlüsseln seien {8, 9, 25}

x	x*22	y = x*22 mod 101	y*23	y*23 mod 101
8	176	75	1725	8
9	198	97	2231	9
25	550	45	1035	25

Kryptographisches Hilfsmittel: Modulare Inverse

- Analog zur multiplikativen Inversen aus der Schulmathematik (1/8 ist die Inverse zu 8 bezüglich der Multiplikation: 1/8 * 8 = 1) ist auch die Inverse *d* einer Zahl *e* bezüglich dem Modulus *N* derart definiert, daß *e***d* mod *N* = 1
- So ist beispielsweise 7 die Inverse von 3 bezüglich Modulo 20, denn 3*7 mod 20 = 1.
- Kryptographisch interessante Eigenschaft: das Ergebnis des Produktes einer Zahl *x* (kleiner dem Modulus *N*) mit dem Produkt eines inversen Paares (*e*,*d*) ist die Identität.
- Beispiel: (*x**3) mod 20 = *y* und *y**7 mod 20 = *x*
- Man kann also die zu verschlüsselnde Zahl *x* mit *e* multiplizieren und das Ergebnis modulo *N* übertragen.
  Zur Entschlüsselung wird *x* mit *d* multipliziert und mod *N* berechnet.

Kryptographisches Hilfsmittel: Modulare Inverse

- Beispiel zur modularen Inversen: e = 22, d = 23, N = 101
- Es ist: 22*23 mod 101 = 506 mod 101 = 1
- Zu verschlüsseln seien {8, 9, 25}

x	x*22	y = x*22 mod 101	y*23	y*23 mod 101
8	176	75	1725	8
9	198	97	2231	9
25	550	45	1035	25

Kryptographisches Hilfsmittel: Modulare Inverse

- Bestimmmung der modularen Inversen *d* zu *e* mod *N* durch den sogenannten erweiterten Euklidischen Algorithmus eEA (Berlekamp Algorithmus)
- Voraussetzung: *e* und *N* sind teilerfremd, d.h. ggT(*e*,*N*) = 1.
- Bestimme mit dem eEA eine Darstellung 1 = *de* + *fN* = ggT(*e*,*N*).
- Dann gilt modulo *N*: 1 = *de* + *fN* = *de*, denn *fN* mod *N* = 0 (bzw. *fN* $\equiv$ 0 (mod *N*)).
- Beispiel: Gesucht sei die Inverse von 2 mod 5. Dazu Linearkombination 1 = 3*2 - 1*5, also ist 3 die gesuchte Inverse: 2 * 3 = 6 mod 5 = 1
- Dies ist das prinzipielle Vorgehen bei der Konstruktion von geheimem Schlüssel (bei gewähltem *N* und *e*)
- Eine Beschreibung des eEA mit Rechner ist unter http://johannes-bauer.com/thi/eea.php

RSA: Verwendung von Primzahlen und mod. Inversen

- Ronald **R**ivest, Adi **S**hamir und Leonard **A**dleman haben den De-facto Standard der asymmetrischen Verschlüsselung erfunden: das RSA-Verfahren.
- Im RSA-Verfahren werden große Primzahlen (p,q) und ihr Produkt $N = p*q$ verwendet.
- Zur Verschlüsselung wird als öffentlicher Schlüssel N samt einer weiteren aus p und q ableitbaren Zahl e verwendet:

  $$C=M^e \bmod N$$

  wobei M der Klartext und C der verschlüsselte Text ist.
- Zum Entschlüsseln wird der geheime Schlüssel d benötigt:

  $$M=C^d \bmod N$$

- **Bemerkung**: Zur Verschlüsselung des Textes muss dieser als Zahl interpretiert werden (ASCII). Zudem muss in jedem Fall $M < N$ gelten. Dies kann durch Aufteilung des Textes erreicht werden.

RSA: Berechnung von *e* und *d*

- Berechnung der Eulerschen Funktion*: $\varphi(N) = (p-1) * (q-1)$ mit *N=p*q,* diese Funktion kann nur bei Kenntnis von *p* und *q* effizient berechnet werden. Berechnung von *d* mit $\varphi(N)$ (zweiter Modulus bei RSA). Wenn *N* bekannt, ist $\varphi(N)$ nur durch Faktorisierung von *N* zu berechnen!
- **Vorgehen**:
  1. Wahl einer Zahl *e*, für die $1 < e < \varphi(N)$ gilt und die teilerfremd zu $\varphi(N)$ ist, z.B. eine Primzahl, also ggt(*e*, $\varphi(N)$) = 1
  2. Berechnung der Zahl *d*, so daß *e*d* kongruent 1 bezüglich des Modulus $\varphi(N)$ ist, also $(e * d)\ mod\ \varphi(N) = 1$
- **Bemerkung**: *a* ist kongruent zu *b* bezüglich des Modulus *c*, falls gilt:

$$a\ mod\ c = b\ mod\ c$$

die Reste beider Zahlen bezüglich des Teilers sind gleich.

*Motivation: Satz von Euler (siehe hinten)
Heute wird anstelle der Eulerschen Funktion beim RSA-Verfahren die Carmichaelsche Funktion $\psi(n)$ verwendet. Die Carmichaelsche Funktion ist definiert als $\psi(n)$=kgV(p-1,q-1).

RSA: Existenz von *d*

- **Satz**: Sind zwei Zahlen *a*,*b* $\in\mathbb{Z}$ teilerfremd, so gibt es eine (positive) ganze Zahl *c*, für die gilt: $b * c \; mod \; a = 1$

  b ist modulo *a* invertierbar und *c* nennt man die modulare Inverse von *b*.
- Beweis:
  1. Schritt: Es gilt ggT(*e*, $\varphi(N)$)=1
  2. Schritt: Wegen der Vielfachsummendarstellung des ggT, gibt es ein x,y$\in\mathbb{Z}$ mit
  $$ggT(e, \varphi(N)) = x * e + y * \varphi(N)$$
  3. Schritt: Aufgrund 1 und 2 gilt:
  $$1 = ggT(e, \varphi(N)) = x * e + y * \varphi(N)$$
  und damit auch
  $$1 \; mod \; \varphi(N) = (x * e + y * \varphi(N)) \; mod \; \varphi(N)$$
  $$1 = (x * e) \; mod \; \varphi(N) + (y * \varphi(N)) \; mod \; \varphi(N)$$
  $$1 = (x * e) \; mod \; \varphi(N)$$

das *x* entspricht also unserem gesuchten *d* (*d* kann durch den erweiterten euklidischen Algorithmus berechnet werden)

RSA: Beispiel

Wir wählen *p*=37 und *q*=41

⇒ N=*p***q*=1517

⇒ Die Eulersche Funktion ist $\varphi(N) = (p-1) * (q-1) = 1440$

e können wir frei wählen, z.B. *e*=43

⇒ *d* können wir über den erweiterten euklidischen Algorithmus berechen (wichtig ist dabei die Kenntnis von *p* und *q*)
Ergebnis: *d*=67

Test: $(e * d)\ mod\ \varphi\ =\ (43 * 67)\ mod\ 1440\ =\ 1$

⇒ Zur Verschlüsselung wird nun (N,*e*) verwendet, zur Entschlüsselung (N,*d*).

RSA: Beweis der Korrektheit I

- Das Ergebnis von Verschlüsseln des Klartextes *M* und Entschlüsseln des Resultates muss wieder der Klartext sein.
- Aus diesem Grund muss gelten:

$$M^{e*d} \, mod \, N = M$$

- **Beweis** durch Satz von Euler: Sei *M* eine natürliche Zahl, die teilerfremd zu den beiden (verschiedenen) Primzahlen *p* und *q* ist. Dann gilt: $M^{(p-1)(q-1)} mod \, p * q = 1$
- *M* und die Primzahlen *p* und *q* sind auf jeden Fall teilerfremd, ansonsten würde $M > p*q = N$ gelten (dieser Fall wurde ausgeschlossen, siehe Folie 124).

RSA: Beweis der Korrektheit II

1. Es gilt: $e*d$ mod $(p-1)(q-1) = 1$, siehe Wahl von d.
2. Feststellung: Es gibt eine Zahl k mit $e*d = k*(p-1)(q-1)+1$.

Daraus folgt:

$$M^{e*d} \; mod \; p*q =$$
$$M^{k*(p-1)(q-1)+1} \; mod \; p*q =$$
$$(M^{(p-1)(q-1)})^k * M \; mod \; p*q =$$
$$1^k * M \; mod \; p*q =$$
$$M \; mod \; p*q = M$$

- Eine anschauliche Erklärung von RSA findet man unter: http://www.matheprisma.uni-wuppertal.de/Module/RSA/index.htm

RSA: Beispiel - Fortsetzung

- Folgender Text soll verschlüsselt werden: „RSA Funktioniert“
- Zunächst müssen wir den einzelnen Buchstaben eine Zahl zuweisen:

  z.B. Leerzeichen = 00; A=01; … Z=26

  (Groß- und Kleinschreibung, sowie Sonderzeichen werden ignoriert)
- Der Text wird durch folgenden String repräsentiert:
  18 19 01 00 06 21 14 11 20 09 15 14 09 05 18 20
- Der Text muss nun in zu codierende Blöcke unterteilt werden, jeder Block muss kleiner als *N* sein (in unserem Fall eignen sich 3er-Blöcke), daraus ergibt sich (eine Null wird angehängt um den 3er Block zu füllen):
  181 901 000 621 141 120 091 514 090 518 200
- **Bemerkung**: In der Realität sind die Blöcke natürlich so groß gewählt, dass der Code nicht mittels Zeichen- bzw. Substringwahrscheinlichkeiten geknackt werden kann.

RSA: Beispiel - Fortsetzung

- Nun kann der Text mit (N=1517,e=43) verschlüsselt werden, dabei wird das Ergebnis der einzelnen Blockverschlüsselungen immer mit führenden Nullen auf 4 Stellen aufgefüllt

  181 901 000 621 141 120 091 514 090 518 200

  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

  0895 1475 0000 1118 1076 0793 0237 0562 1455 1258 1515

- Mit Hilfe des Schlüssels (N=1517, d=67) kann der Code nun wieder entschlüsselt werden.

  0895 1475 0000 1118 1076 0793 0237 0562 1455 1258 1515

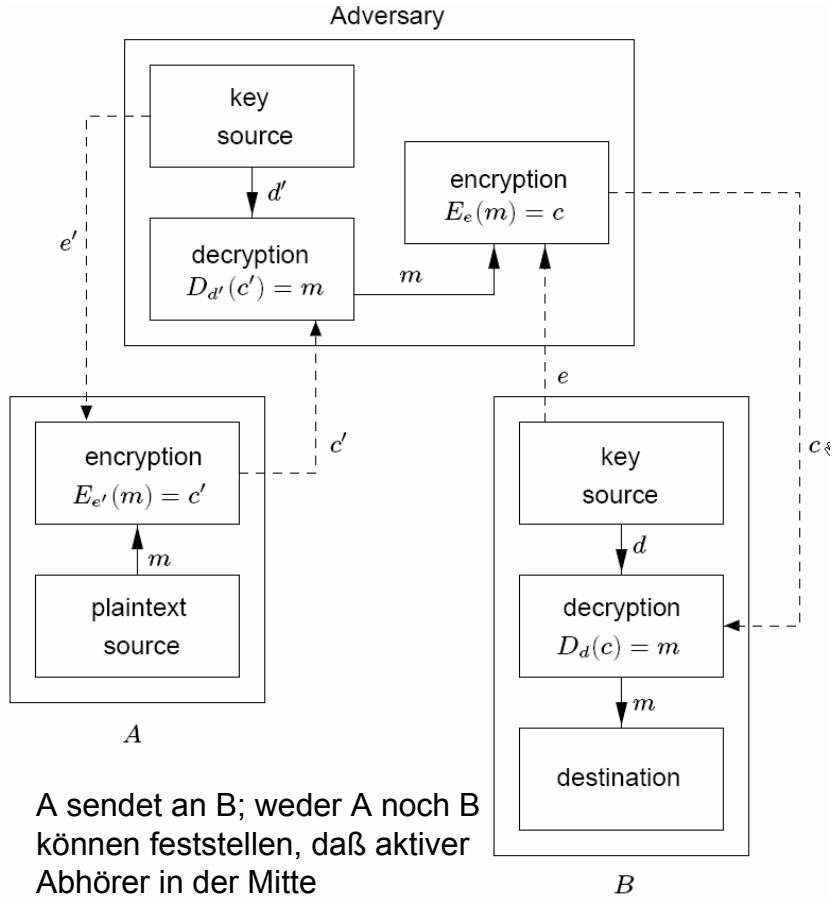
  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

  181 901 000 621 141 120 091 514 090 518 200

⇒ RSA funktioniert

Eigenschaften der asymmetrischen Verschlüsselung

- PKE ermöglicht die abhörsichere Kommunikation zwischen zwei Parteien über unsichere Kanäle, ohne daß diese jemals geheime Schlüssel über einen sicheren Kanal austauschen müssen.
- Einschränkung: Alice weiß, daß sie sicher kommuniziert, **aber nicht, mit wem**. Man benötigt einen vertrauenswürdigen Dritten, der zusichert, daß Bob auch der ist, der er vorgibt zu sein (bei symmetrischer Verschlüsselung benötigt man einen Dritten (Kurier), der dies sicherstellt und *zusätzlich* gewährleistet, daß der Schlüssel geheim bleibt)
- Veranschaulichung des Problems durch „man-in-the-middle“-attack



A sendet an B; weder A noch B können feststellen, daß aktiver Abhörer in der Mitte

Probleme der asymmetrischen Verschlüsselung und Lösungen

- Identität:
  - Sicherer Weg zur Feststellung der Identität: Verfälschungs-gesicherte Übermittlung eines Zertifikats einer TTP (trusted third party), daß ferne Partei die ist, für die sie sich ausgibt.
  - Hilfslösung (bei SSH): Programm fragt beim ersten Kommunikationsversuch den Nutzer, ob der ferne Rechner als der erscheint, der sein soll (IP-Adresse). Damit wird letzten Endes DNS als TTP „mißbraucht“.
- Geschwindigkeit:
  - Asymmetrische Verfahren sind im Vergleich zu symmetrischen Verfahren deutlich langsamer (RSA ist um einen Faktor von mindestens 1000 langsamer als 3DES oder AES)

  $\Rightarrow$ Verwendung von **hybriden** Verfahren: zum Schlüsselaustausch für symmetrische Verfahren werden asymmetrische Verfahren eingesetzt.

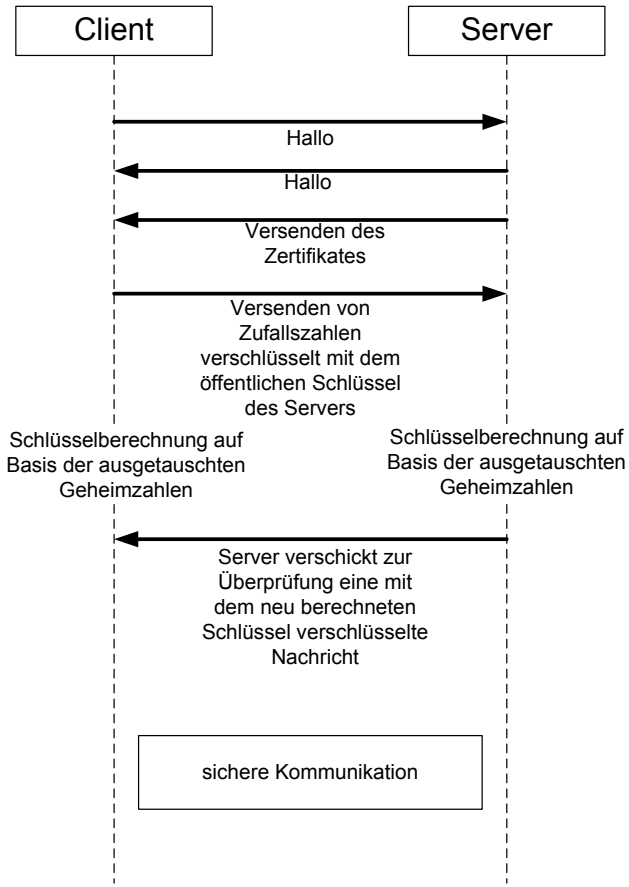
Hybride Verfahren: Beispiel SSL

- Aufgabe von SSL (Secure Socket Layer): Aufbau einer gesicherten Verbindung zwischen Client und Server
- Funktionsmerkmale von SSL sind:
  - Möglichkeit zur Vereinbarung von zu verwendenden Verschlüsselungs- und Authentifizierungs-Algorithmen (wichtig, da in einem offenen Netzwerk nicht alle die gleiche Client-Software enthalten)
  - Sichere Kommunikation zwischen beliebigen Partnern ohne Notwendigkeit eines sicheren Kanals: Beim Aufbau der Verbindung wird asymmetrische Verschlüsselung zum Austausch von Schlüsseln genutzt und erfolgt im Rahmen eines Handshake-Protokolls (siehe nächste Folie).
- Die eigentliche Kommunikation wird durch symmetrische Verschlüsselung geschützt.
- Typischerweise wird während des Aufbaus zumindest der Server durch Verwendung von Zertifikaten authentifiziert.

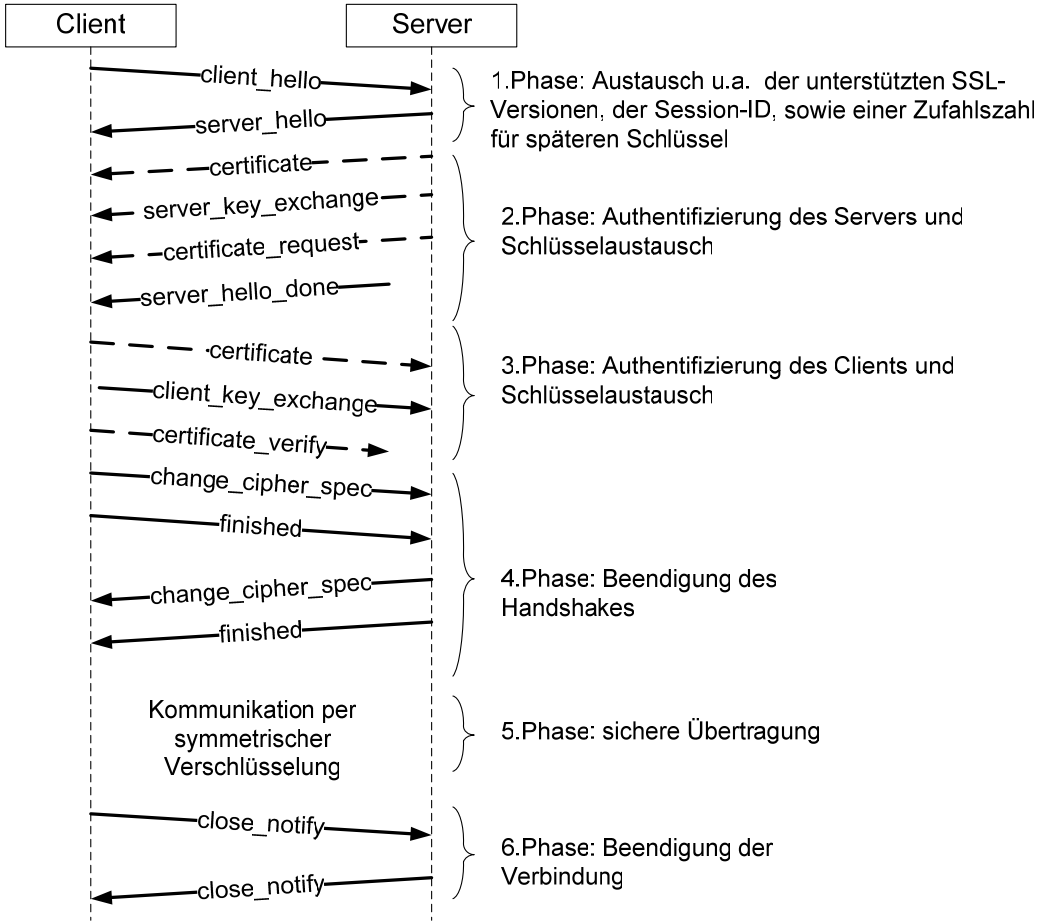
SSL: Handshakeprotokoll

- Das Handshake-Protokoll dient zur Einrichtung einer sicheren Verbindung und erlaubt den Austausch der zur Verschlüsselung und Authentifizierung notwendigen Optionen und Parametern.
- Das Handshake-Protokoll kann auch während einer etablierten sicheren Verbindung aufgerufen werden um Parameter zu ändern.
- Das Protokoll besteht insgesamt aus 4 Phasen:
  1. Einleitung des Handshakes inklusive Austausch von unterstützten Protokollversionen, der Sitzungs-ID, Verschlüsselungsmethode (z.B. RSA), Komprimierungsmethoden und Zufallswerte (zur Bildung des späteren Verschlüsselungscodes).
  2. Identifizierung des Servers gegenüber dem Client durch ein Zertifikat, wie z.B. X.509-Zertifikat (Phase optional)
  3. Identifizierung des Clients gegenüber dem Server durch ein Zertifikat (Phase optional)
  4. Abschluss des Handshakes und Berechnung des Codes für die symmetrische Verschlüsselung durch Verwendung der Hashfunktionen SHA-1 und MD5.

SSL: Handshake (vereinfacht)



SSL: Handshake-Nachrichten



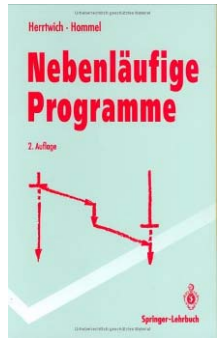
Beurteilung von SSL

- Einsatz: SSL wird vor allem für https eingesetzt, weitere Einsatzgebiete viele andere Applikationsprotokolle, wie z.B. ftp, telnet
- Werden keine Zertifikate zur Identifizierung eingesetzt, schützt SSL nicht vor einem Man-In-The-Middle-Angriff. Der Angreifer kann mit beiden Seiten Schlüssel austauschen und so die ausgetauschten Daten mitprotokollieren.

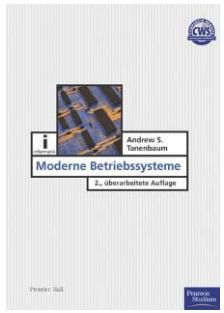
Grundlegende Konzepte der nebenläufigen Programmierung

Literatur

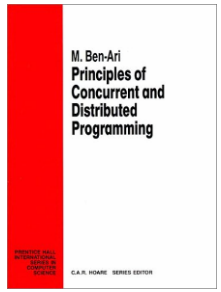
- R.G. Herrtwich, G. Hommel: „Nebenläufige Programme“



- A. S. Tanenbaum: „Moderne Betriebsysteme“



- M. Ben-Ari: „Principles of Concurrent and Distributed Programming“



Bisherige Annahmen

- Bisher wurden in der Vorlesung folgende Annahmen gemacht:
  1. Ein Rechner führt (zu jedem Zeitpunkt) genau ein Programm aus.

     vs. nebenläufige Systeme
  2. Das Programm wird auf einem Rechner ausgeführt.

     vs. verteilte Systeme
  3. Die Ausführung des Programms wurde unabhängig von der Startzeit und der Endzeit betrachtet, nur die Resultate des Programmlaufs waren interessant.

     vs. Echtzeit-Systeme
- In den folgenden Kapiteln werden die Konsequenzen betrachtet, wenn diese Annahmen, insbesondere 1., nicht mehr erfüllt werden.

Nebenläufigkeit

- Als nebenläufig werden Systeme bezeichnet, wenn zu einem Zeitpunkt mehrere Prozesse gleichzeitig ausgeführt werden.
- Jeder Prozess P_i besteht auf einer sequentiellen Aneinanderreihung einzelnen Anweisungen/Ereignissen.

$$P_1=e_{11}; e_{12}; e_{13};\ldots$$

$$\ldots$$

$$\| P_n=e_{n1}; e_{n2}; e_{n3};\ldots$$

- Prozesse können sich dabei in ihrer Ausführung gewollt (z.B. Prozess B wartet auf ein Ergebnis von Prozess A) und ungewollt (z.B. Konkurrenz um Betriebsmittel wie die CPU) beeinflussen.

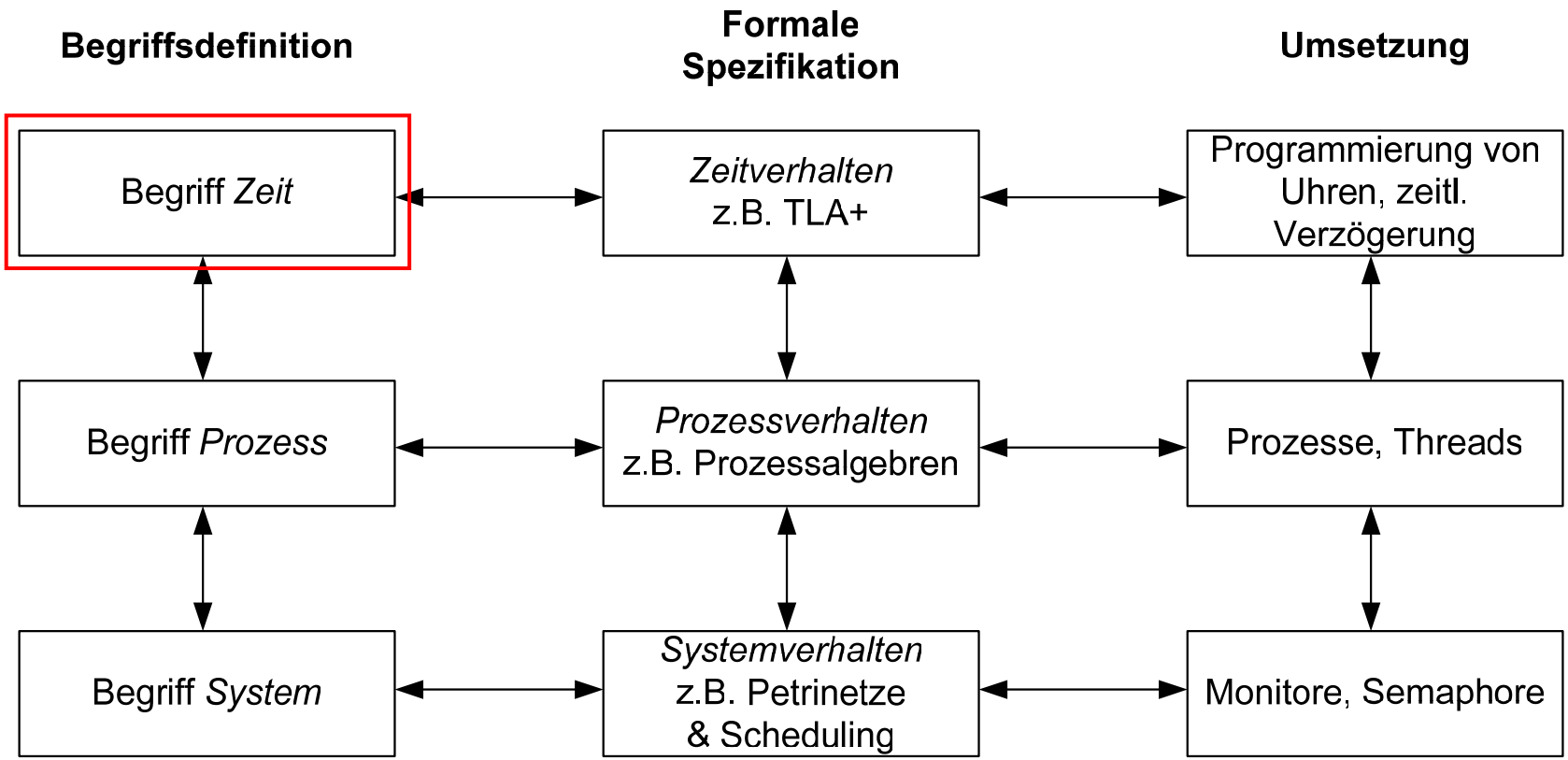
Parallelität vs. Nebenläufigkeit

- Zwei Prozesse/Ereignisse werden als **parallel** (gleichzeitig) bezeichnet, wenn ihre Ausführung nicht voneinander abhängt und keinerlei Wirkungszusammenhänge zwischen den beiden Prozessen/Ereignissen bestehen.
- Beispiele:
  - Während einer Klausur bearbeiten die Studenten (hoffentlich) parallel die Aufgaben.
  - Sobald einer der beiden Studenten abschreibt, ist die Bearbeitung nebenläufig.
- Beispiele in Bezug auf Rechner:
  - Besitzt ein Rechner nur eine CPU, so können zwei Prozesse auf diesem Rechner nur nebenläufig, jedoch nicht parallel ausgeführt werden.
  - Zwei Rechner ohne Vernetzung arbeiten vollständig parallel.

Interessante Fragestellungen bei nebenläufigen Systemen

- Parallelität vs. Nebenläufigkeit:
  - Welche Ereignisse/Aktionen können gleichzeitig (parallel) ablaufen, welche Ereignisse/Aktionen sind demgegenüber voneinander abhängig?
- Interaktion zwischen zwei Prozessen:
  - Wie können zwei (ansonsten unabhängige) Prozesse interagieren/ kommunizieren?
  - Wie können Abhängigkeiten zwischen einzelnen Ereignissen/Aktionen verschiedener Prozesse ausgedrückt werden (Synchronisation)?
- Determinismus von nebenläufigen Systemen
  - Zeigt ein nebenläufiges System bei gleichen Eingaben immer das gleiche Verhalten oder hängt das Ergebnis vom (zufälligen) zeitlichen Auftreten der einzelnen Ereignisse/Aktionen ab?

Themengebiete Nebenläufigkeit



Der Begriff der Zeit

- "Die **Zeit** ist die fundamentale Größe, über die sich zusammen mit dem Raum die Dauer von Vorgängen und die Reihenfolge von Ereignissen bestimmen läßt. Da sie sich bislang nicht auf grundlegendere Phänomene zurückführen lässt, wird sie über Verfahren zu ihrer Messung definiert, wie es auch bei Raum und Masse der Fall ist." (Wikipedia)
- Nach dem SI (Système International d'unités) wird die Zeit in Sekunden (Einheitenzeichen "s", *nicht* "sec") gemessen.
- Eine Sekunde ist das 9.192.631.770-fache der Periodendauer der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Cäsium-Atomen (genau: des Nuklids ^{133}Cs) entsprechenden Mikrowellen-Strahlung

Abhängigkeit von der Zeit

- Oftmals sind einzelne Arbeiten im Alltag nicht nur abhängig vom Ergebnis, sondern auch von der Zeit:
  - Kellner:
    - Der Kellner sollte die Gäste im Restaurant möglichst schnell bedienen. Kommt es zu langen Verzögerungen, so kann es zu Verdienstausfällen (in Form von geringeren oder keinem Trinkgeld) führen.

  - Airbag:
    - Ein Airbag muss sich im Fall eines Unfalls in weniger als 40 ms auffüllen. Sollte sich der Vorgang verzögern, so kann dies im schlimmsten Fall zum Tod des Passagiers führen.

- Das Forschungsgebiet der Echtzeitsysteme beschäftigt sich mit diesem Thema im Bereich der Informatik:
  - **Weiche** Echtzeitsysteme sind Systeme bei denen verspätete Ergebnisse noch nutzbar sind, sich die Dienstgüte jedoch verschlechtert.
  - Bei **harten** Echtzeitsystemen führt eine Verletzung einer Prozessfrist zu einem fatalen Ergebnis (z.B. Verletzung oder Tod von Personen).
  - Ob ein Echtzeitsystem also hart oder weich ist, hängt nicht von absoluten Zeiten (bzw. der Rechnergeschwindigkeit), sondern vielmehr von Reaktionszeiten und den Folgen ihrer Überschreitung.

Zur Zeit

Isaac Newton: „Die absolute, wahre und mathematische Zeit verfließt an sich und vermöge ihrer Natur gleichförmig und ohne Beziehung auf irgendeinen äußeren Gegenstand“

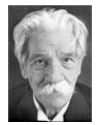
Jean-Luc Picard: „Jemand hat mir mal gesagt, die Zeit würde uns wie ein Raubtier ein Leben lang verfolgen. Ich möchte viel lieber glauben, dass die Zeit unser Gefährte ist, der uns auf unserer Reise begleitet und uns daran erinnert, jeden Moment zu genießen, denn er wird nicht wiederkommen. Was wir hinterlassen ist nicht so wichtig wie die Art, wie wir gelebt haben. Denn letztlich [...] sind wir alle nur sterblich.“

Konrad Adenauer: „Man darf niemals 'zu spät' sagen. Auch in der Politik ist es niemals zu spät. Es ist immer Zeit für einen neuen Anfang.“

Bill Gates: „Ich beneide Menschen, die mit 3 oder 4 Stunden Schlaf in der Nacht prächtig zurechtkommen. Sie haben so viel mehr Zeit zu arbeiten, zu lernen und zu spielen.“

William Shakespeare: „Jedes Ding hat seine Zeit“

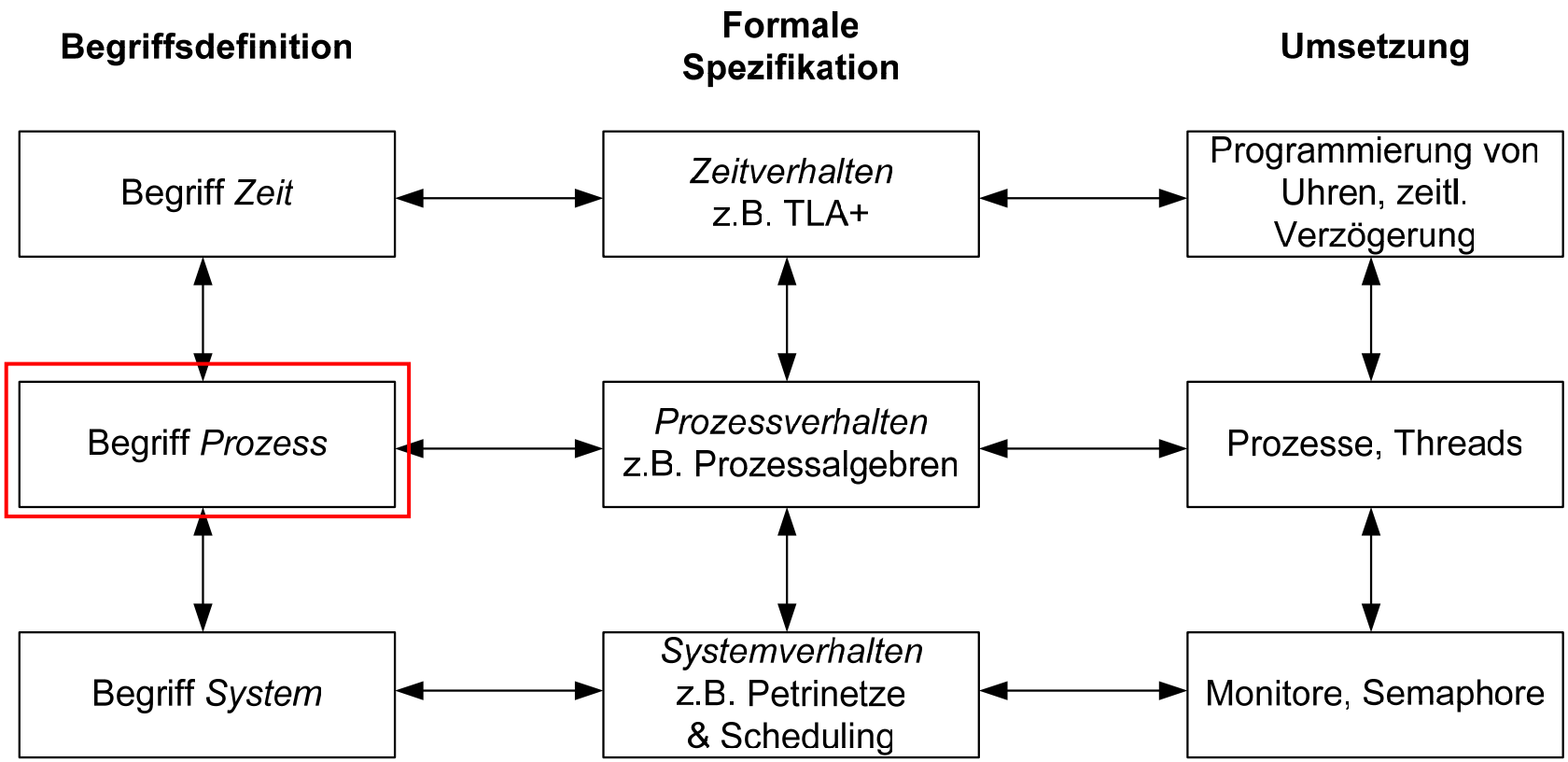
John Steinbeck: „Man verliert die meiste Zeit damit, daß man Zeit gewinnen will“



Albert Schweitzer: „Niemand kann vor seiner Zeit davonlaufen“

Johann Wolfgang von Goethe: „Wir haben genug Zeit, wenn wir sie nur richtig anwenden“

Themengebiete Nebenläufigkeit



SFB 453
Wirklichkeitsnahe Telepräsenz und Teleaktion

Allgemeine Definition des Prozessbegriffs

- **Definition**: Ein Prozess ist der definierte Ablauf von Zuständen eines Systems.
- Einfacher: "*Wenn Du nicht weißt, was es tut, dann nenne es einen Prozeß – wenn Du nicht weißt, was es ist, nenn' es ein System*!"
- Häufig wird auch der Begriff des diskreten Prozesses verwendet: Ein **diskreter** Prozess wird durch die Reihenfolge von Ereignissen, die in einem kausalen Zusammenhang stehen beschrieben. Ereignisse sind Folgen der Durchführung von Aktionen.
- Sei die Menge *E* das "Universum" der möglichen Ereignisse und sei *A* die Menge der möglichen Aktionen.
- Ein diskreter Prozess wird durch die Menge E_P, eine lineare Ordnungsrelation $\leq$ über die Menge E_P, sowie eine Funktion α zur Zuordnung von Aktionen zu jedem Ereignis.

Beispiel für Prozesse: Paternoster

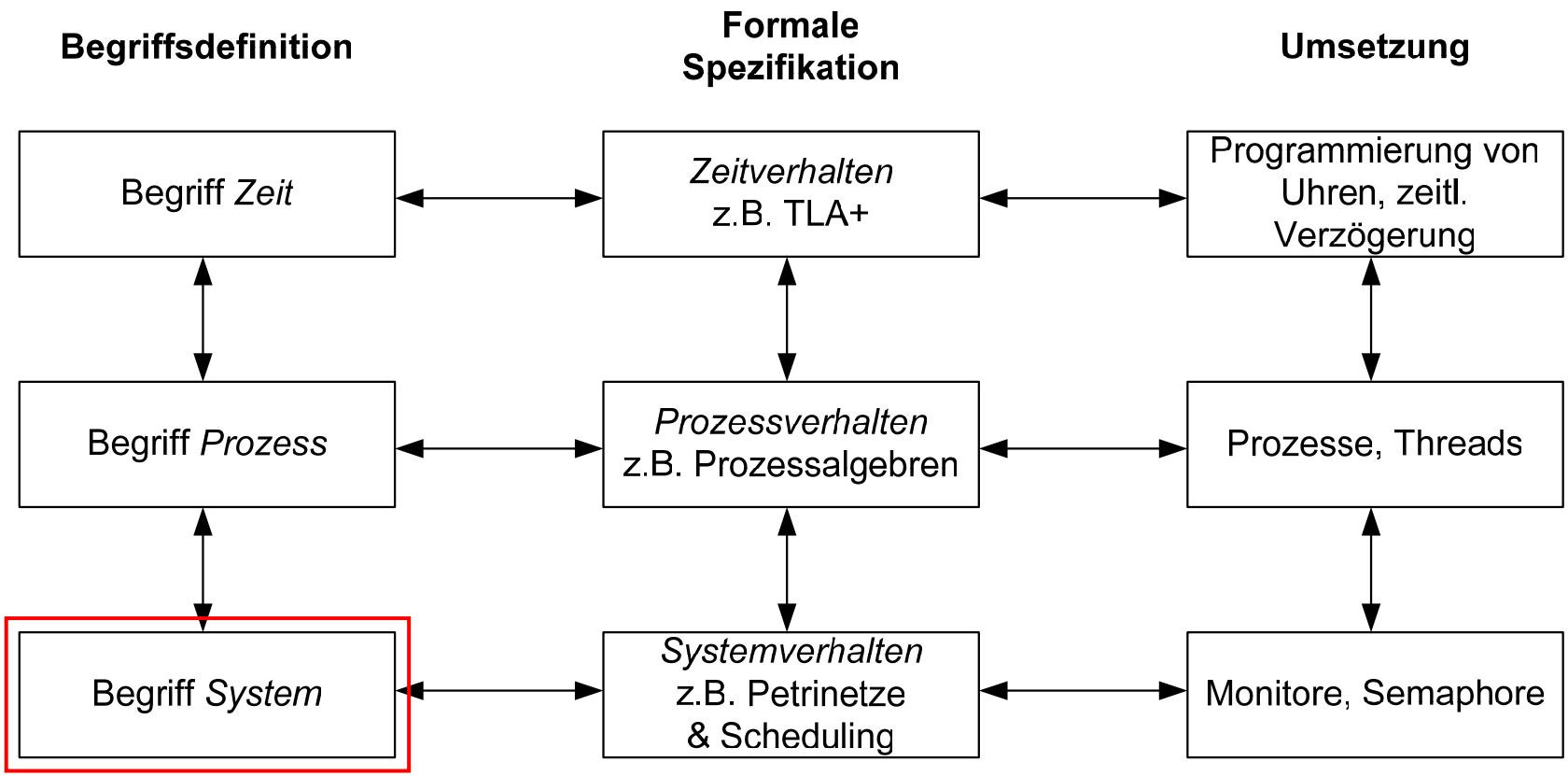
- Prozess: „Kabine“
  e_{K1} : „Kabine erreicht Erdgeschoss“
  e_{K2} : „Kabine verlässt Erdgeschoss“
  e_{K3} : „Kabine erreicht 1.Stock“
  e_{K4} : „Kabine verlässt 1.Stock“
  e_{K5} : „Kabine erreicht 2.Stock“
  e_{K6} : „Kabine verlässt 2.Stock“
  e_{K7} : „Kabine erreicht 2.Stock“
  e_{K8} : „Kabine verlässt 2.Stock“
  e_{K9} : „Kabine erreicht 1.Stock“
  e_{K10} : „Kabine verlässt 1.Stock“
  e_{K11} : „Kabine erreicht Erdgeschoss“
  …
- Prozess „Benutzer“
  e_{B1} : „Einsteigen in Kabine im EG“
  e_{B2} : „Auf die Uhr schauen“
  e_{B3} : „Aussteigen im 2. Stock“

  In dem Beispiel wird von einer impliziten Ordnung ausgegangen, also $e_{K1}<e_{K2}<e_{K3}$…

Aktion

Ereignis

Themengebiete Nebenläufigkeit



Zum Begriff "System"

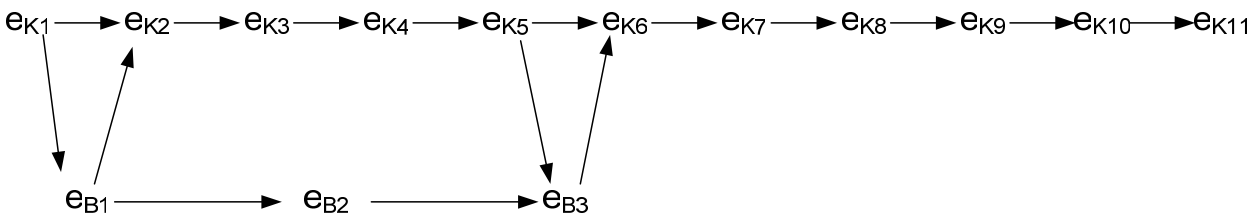
- **System** (v. griech. *systema* = das Gebilde, Zusammengestellte, Verbundene) bezeichnet ein Gebilde, dessen wesentliche Elemente (Teile) so aufeinander bezogen sind und in einer Weise wechselwirken, dass sie (aus einer übergeordneten Sicht heraus) als aufgaben-, sinn- bzw. zweckgebundene Einheit (d.h. als Ganzes) angesehen werden.
- In unserem Zusammenhang bestehen Systeme aus einzelnen Komponenten; die Beziehungen (Relationen) zwischen den Elementen sind Wirkungen von Austauschprozessen, wie zum Beispiel Stoff-, Energie- oder Informationsflüssen.
- Systeme können auch als Modell der Realität verstanden werden.

Beschreibung von Systemen für die Informatik: Relationen

- Die Prozesse des Systems "Paternoster" müssen gewisse Bedingungen erfüllen. So muss z.B. das Ereignis e_{B1} nach dem Ereignis e_{K1}, aber vor dem Ereignis e_{K2} erfolgen.
- Neben der linearen Ordnung innerhalb der beiden Prozesse „Lift“ und „Benutzer“ muss also noch eine weitere Relation angegeben werden, um den zeitlichen Ablauf zwischen den Ereignissen der verschiedenen Prozesse zu spezifizieren.
- Dies kann in Form einer Relation < erfolgen:
  $e_{K1} < e_{K2} < e_{K3} < e_{K4} < e_{K5} < e_{K6} < e_{K7} < e_{K8} < e_{K9}$
  $e_{B1} < e_{B2} < e_{B3}$
  $e_{K1} < e_{B1} < e_{K2}$
  $e_{K5} < e_{B3} < e_{K6}$
- **Anmerkung**: Die oben aufgeführten Beziehungen beschreiben nur die direkten Beziehungen, die eigentliche Relation ergibt sich aus der transitiven Hülle dieser Beziehungen (es gilt also z.B. auch $e_{K1} < e_{K3}$).

Beschreibung von Systemen: Graphen

- Die Spezifikation durch eine Relation kann relativ schnell unübersichtlich werden.
- Eine schönere Möglichkeit, die Relation darzustellen ist die Notation durch gerichtete Graphen.
- Graph für das vorangegangene Beispiel:



- Interpretation des Graphen:
  - Ein Ereignis *x* darf erst dann stattfinden, wenn für alle anderen Ereignisse $y \in E$ gilt, dass *y* schon stattgefunden hat oder aber dass es keinen Pfad von *y* nach *x* im Graphen gibt.
  - Zwei Ereignisse, die durch keinen Pfad miteinander verbunden sind, z.B. e_{B2} und e_{K3}, sind parallel bzw. nebenläufig.

Folgerungen

- Das Beispiel des Paternosters zeigt, daß sequentielle Programme, bei denen es immer exakt eine Reihenfolge der einzelnen Ereignisse gibt, im Alltag nicht realistisch sind: e_{B2} kann vor oder nach e_{K3} ausgeführt werden.
- Auch im Bezug auf Computersysteme sind nebenläufige Berechnungen zum Alltag geworden:
  - Nebenläufige Systeme:
    - Ausführung mehrerer Anwendungen
    - Graphische Oberflächen (Mausbewegung und Tastatureingaben haben unabhängig von ausgeführter Anwendung ihre eigene Rechenzeit)
  - Parallele Systeme:
    - Internet
- Entsteht bei der Ausführung von nebenläufigen Prozessen eine gemischte Sequenz von Anweisungen dieser Prozesse so spricht man von **Verzahnung** (**Interleaving**).

Probleme der nebenläufigen Programmierung

Klassische Probleme aus dem Alltag

- Im folgenden werden verschiedene Probleme aus dem Alltag angeführt, um die klassischen Aufgabenstellungen der nebenläufigen Programmierung zu motivieren.
- Wir geben zunächst einen Überblick in die Problematik ohne spezifische Lösungsansätze zu erarbeiten.
- Danach werden die Probleme klassifiziert und – falls möglich – jeder Klasse die typische Problemstellung aus der Informatik zugeordnet.
- Die einzelnen Lösungsansätze werden schließlich diskutiert.

Beispiel 1: (Nicht-deterministisches) Bank-Konto

- Eine schwerwiegende Konsequenz des Verzichts auf Sequentialität ist die mögliche Entstehung von Nichtdeterminismus.
- Ein System ist **deterministisch**, wenn es bei gleichen Eingaben immer mit dem gleichen Ergebnis endet.
- Klassisches Beispiel für Nichtdeterminismus (Verwaltung des Kontostands `account`):

Prozess A
```
x=readAccount(account);
x=x+500;
writeAccount(x,account);
```

Prozess B
```
y=readAccount(account);
y=y-200;
writeAccount(y,account);
```

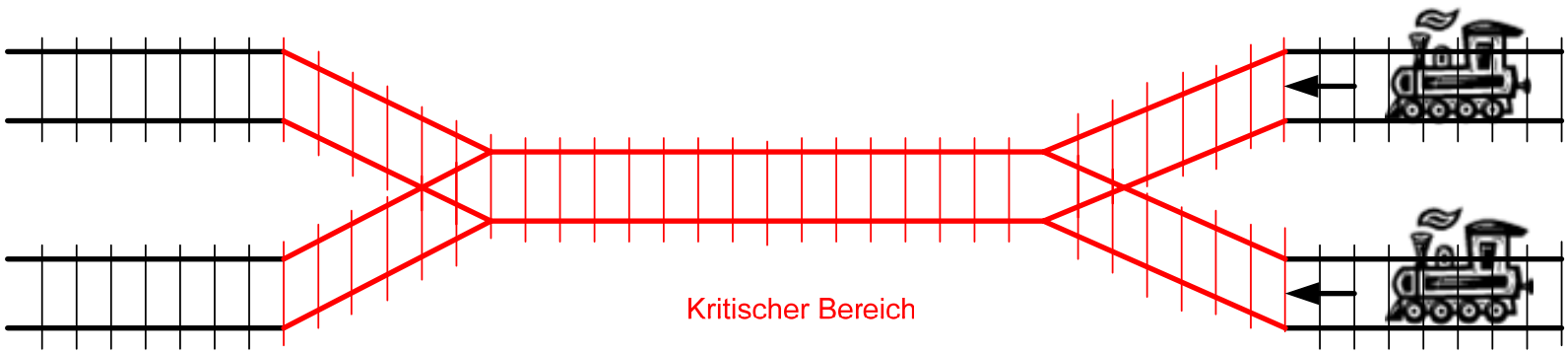
- Abhängig von der Ausführungsreihenfolge der einzelnen Anweisungen kann der Kontostand variieren.
- Lösungen zu diesem Problem später.

Klasse Beispiel 1: "Leser-Schreiber-Problem"

- Arbeiten zwei Prozesse auf gleichen Daten, wobei zumindest ein Prozess die Daten auch verändert, so kann es zu unterschiedlichen Resultaten kommen (wie gerade gezeigt).
- Diese Problemklasse wird als **Leser-Schreiber-Problem** bezeichnet.
- Grundsätzlich werden zu diesem Problem folgende Annahmen gemacht:
  - Es dürfen mehrere Prozesse gleichzeitig die Daten lesen.
  - Will ein Prozeß die Daten verändern, so darf für die Dauer dieser Veränderung kein anderen Prozeß auf die Daten (lesend oder schreiben) zugreifen.
  - Häufig (nicht immer!) wird dem Schreiber eine höhere Priorität eingeräumt, er soll also vom System bevorzugt werden.

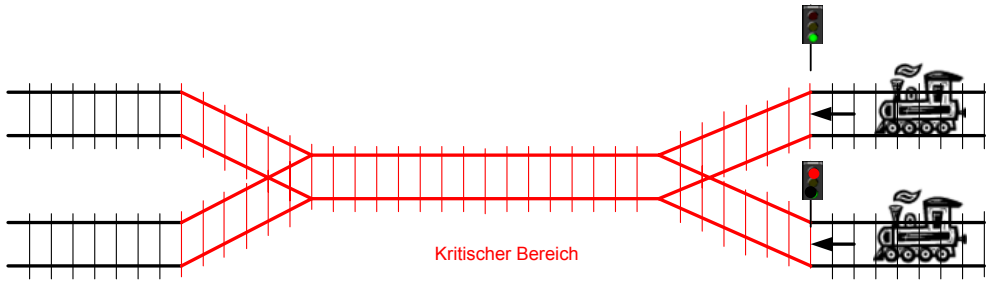
Beispiel 2: Eingleisige Strecke bei der Bahn

- Existiert auf einer Eisenbahn-Strecke nur ein Gleis, so muss sichergestellt werden, dass immer nur ein Zug sich in jedem Streckenabschnitt befindet.
- Hier wollen zwei Prozesse (Lok 1 und Lok 2) das gleiche Betriebsmittel (Gleis) benutzen. Dies geht offensichtlich nur nacheinander.
- Abschnitte, in denen mehrere Prozesse um ein Betriebsmittel konkurrieren, bezeichnet man als *kritische Bereiche*.

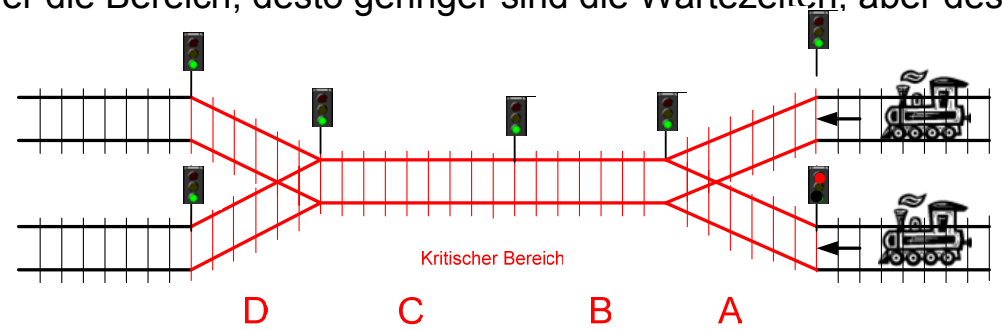


Klasse 2: wechselseitiger Ausschluß in kritischen Bereichen

- Die Klasse dieser Probleme wird durch **wechselseitigen Ausschluss** (nur ein Prozess darf sich gleichzeitig in dem kritischen Bereich befinden) gelöst.
- Zur Lösung des Problems werden im Alltag Signale verwendet

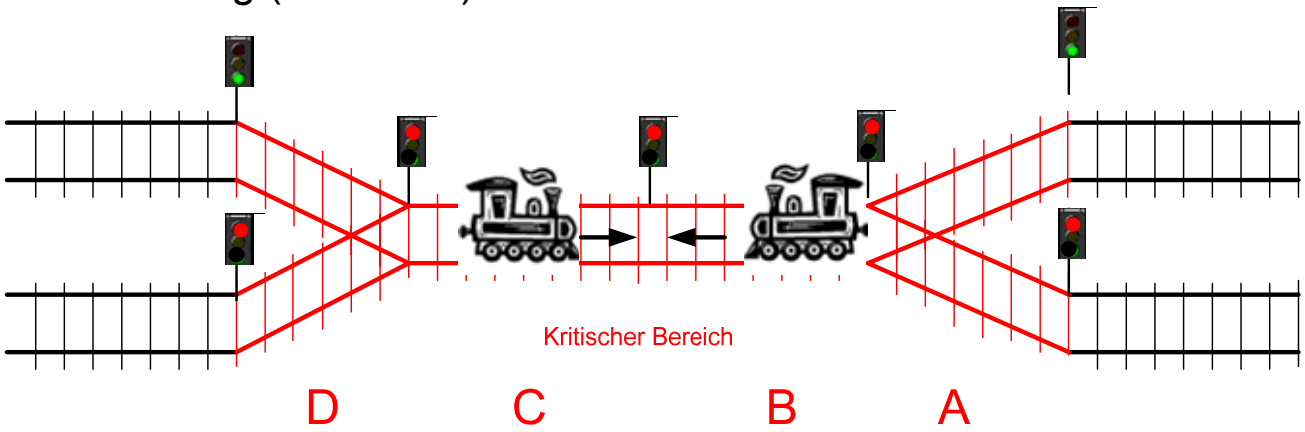


- Um Verzögerungen zu minimieren, wird versucht, die kritischen Bereiche in sinnvolle Größen zu zerlegen (je kleiner die Bereich, desto geringer sind die Wartezeiten, aber desto mehr Signale sind notwendig).



Beispiel 3: beidseitig befahrene Bahn-Strecke

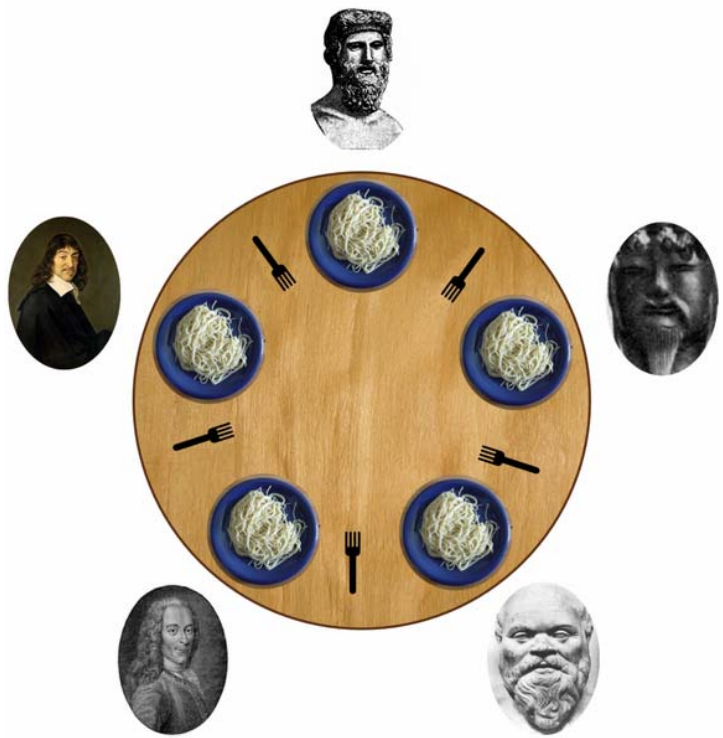
- Überprüft ein Zug immer nur den vor ihm liegenden Gleisabschnitt, so kann es zu einer Verklemmung (Deadlock) kommen:



- Besitzt ein System mehrere Betriebsmittel und werden von den einzelnen Prozessen mehrere Betriebsmittel und in unterschiedlicher Reihenfolge angefordert, so kann es zu Verklemmungen kommen.
- Kann ein einzelner Prozess das von ihm angeforderte Betriebsmittel nicht bekommen (weil z.B. andere Prozesse immer bevorzugt werden), so **verhungert** er.

Klasse 3: Verklemmungen

- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und **sterben den Hungertod (starvation)**.



Beispiel 4a (Scheduling): Fahrplan-Erstellung

- Ein weiteres sehr interessantes Problem ist die Fahrplanerstellung, denn in der Informatik existieren an vielen Stellen die gleichen Probleme wie bei der Fahrplanerstellung.
- Problem:
  - Verschiedene Prozesse (Züge) müssen sich gemeinsame Betriebsmittel (Gleise bzw. Gleisabschnitte) teilen.
  - Es dürfen keine Verklemmungen vorkommen.
  - Die Fahrplanerstellung soll fair sein (jeder Zug soll fahren dürfen).
  - Der Durchsatz sollte möglichst hoch sein.
- Sind die Prozesse und die benötigten Betriebsmittel (inklusive Dauer der Belegung) bekannt und wird ein Ausführungsplan (Fahrplan) erstellt, so spricht man von **Plansuche**, die Vergabe des Betriebsmittel an die Prozesse wird als **Scheduling** bezeichnet.

Beispiel 4b (Scheduling): Arztpraxis

- Die Terminplanung in einer Arztpraxis ist ebenfalls ein Scheduling-Problem:
  - Prozesse sind die einzelnen Untersuchungen
  - Betriebsmittel ist der Arzt
- Die Terminplanung lässt sich aus folgenden Gründen jedoch nicht per Planung lösen:
  - Die Anzahl der Untersuchungen ist im vorab nicht bekannt (einzelne Untersuchungen können abgesagt werden, es können Notfälle dazwischenkommen).
  - Die Dauer der Belegung ist nicht vorab bekannt.
  - Die Priorität einer Untersuchung kann sich ändern (je nach Untersuchungszwischenergebnissen).

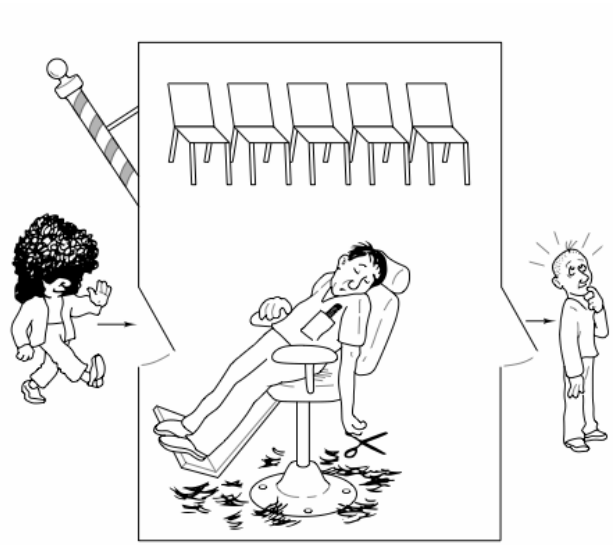
⇒ Hier ist eine dynamische Ein- bzw. Umplanung nötig, man spricht von **dynamischem Scheduling**.

Beispiel 5a: Taxistand

- Ein Taxistand ist ein Beispiel für eine weitere Problemklasse in der Informatik, dem **Producer-Consumer-Problem**.
- Szenario:
  - Die Zentrale (Prozess Producer) schickt laufend Taxis an den Taxistand, so lange an diesem noch Taxis Platz haben. Ist der Taxistand vollständig belegt, so muss die Zentrale warten, bis wieder ein Platz frei wird.
  - Die Kunden (Prozesse Consumer) gehen zu dem Taxistand und steigen in ein Taxi an, falls eines wartet. Andernfalls müssen sie selber warten.
- Allgemein lässt sich das Producer-Consumer-Problem wie folgt spezifizieren:
  - Der Producer-Prozess füllt die Produktionsplätze mit seinen Produkten auf. Sobald kein Platz leer ist, legt sich der Prozess schlafen, bis wieder ein Produkt entnommen wird.
  - Der Consumer-Prozess konsumiert die Produkte, sofern Produkte vorhanden sind. Ist kein Produkt vorhanden, so legt sich der Prozess schlafen, bis ein neues Produkt produziert wurde.

Beispiel 5b: Sleeping Barber

- Konstrukt für die Informatik: Sleeping Barber Problem
  - In einem Friseurladen gibt es einen Friseur, einen Friseurstuhl, sowie *N* Wartestühle
  - Der Friseur schneidet einem Kunden die Haare. Falls kein Kunde vorhanden ist, legt er sich schlafen.
  - Die Kunden überprüfen beim Betreten des Ladens, ob noch ein Wartestuhl frei ist und setzen sich auf diesen. Schläft der Friseur, so wecken sie ihn auf, andernfalls warten sie. Ist kein Wartestuhl frei, so verlassen sie den Friseurladen.
- Wird das Problem nicht richtig implementiert so kann es zu einer Verklemmung kommen oder ein Kunde kann auch verhungern (falls immer ein anderer Kunde vor ihm bedient wird).

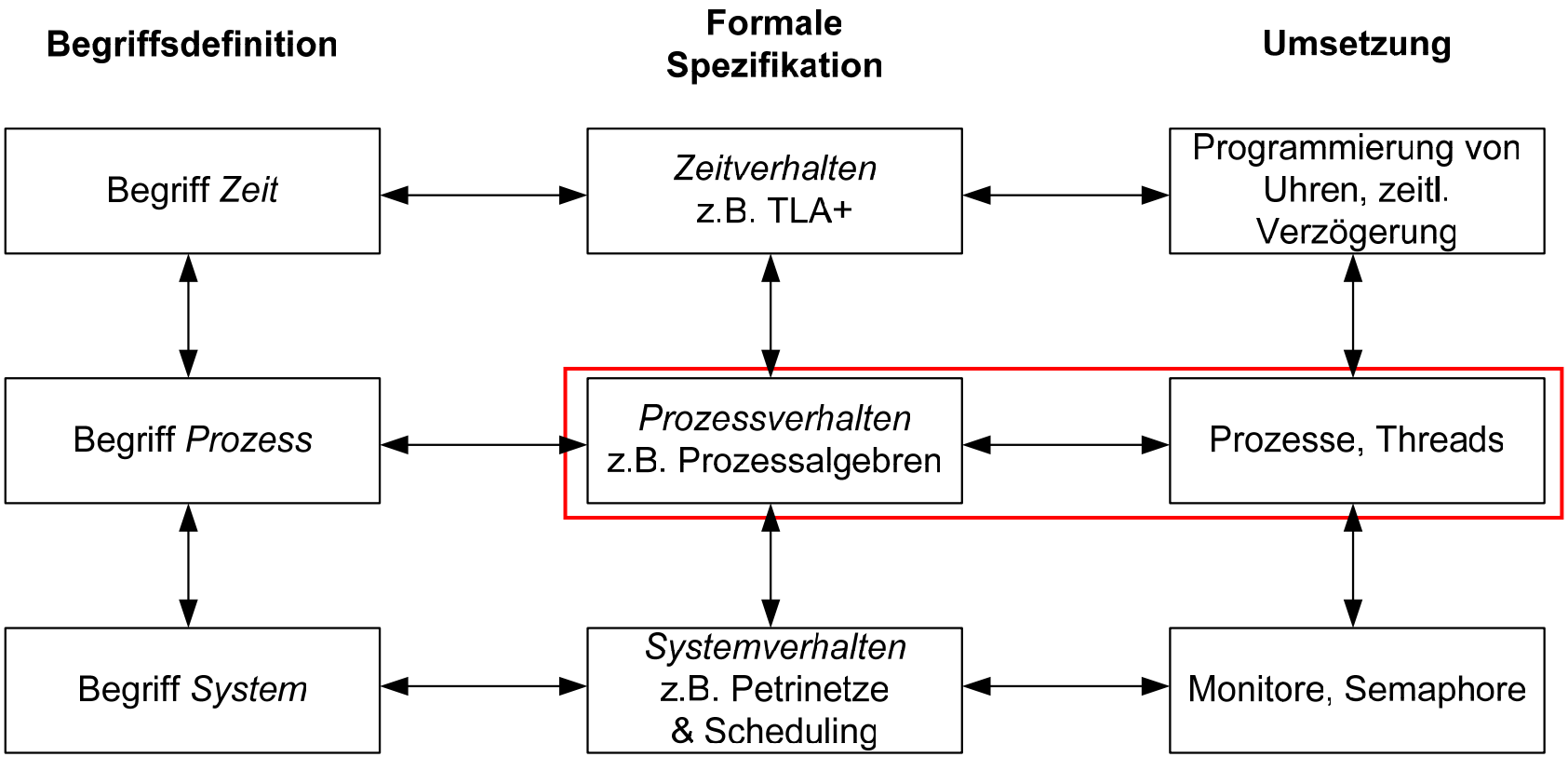


Beispiel 6: Synchronisation

- Aus dem Alltag kennen wir das Problem der Synchronisation:
  - Zeitsynchronisation:
    - Unsere Uhren müssen in regelmäßigen Abständen synchronisiert / aufeinander abgestimmt werden.
    - Befinden sich zwei Personen in unterschiedlichen Zeitzonen und wollen sie miteinander telefonieren, so müssen sie neben der Vereinbarung der Uhrzeit auch die relevante Zeitzone festlegen.
  - Prozesssynchronisation:
    - Vorlesung: Die Prozesse „Professor hält Vorlesung“ und „Student hört Vorlesung“ müssen aufeinander abgestimmt werden.
    - Kreuzung: Kommen zwei Autos (Prozesse) an eine Kreuzung ohne Ampel, so müssen sich die beiden Prozesse synchronisieren um einen Unfall zu vermeiden.
- Prozeßsynchronisation bezeichnet also die Koordinierung des Ablaufs nebenläufiger Systeme. Zeitsynchronisation ist ein Spezialfall der Prozeß-synchronisation. Zur Koordinierung können verschiedene Hilfsmittel, wie Nachrichten oder Uhren benutzt werden.

Prozesse, Threads

Themengebiete Nebenläufigkeit



Prozeß (aus Betriebssystemsicht)

- **Definition**: Ein BS-Prozess ist ein Programm in Ausführung, inklusive des Programmzählers, den Werten der Register und der Belegung der Variablen.
- In den meisten Betriebssystemen wird einem Prozeß ein eigener Adreßraum bei der Initialisierung zugewiesen. Prozesse können in diesem Fall nicht auf Datenwörter im Adressraum eines anderen Prozesses zugreifen.
- Prozesse werden so behandelt, als würden sie eine eigene virtuelle CPU besitzen, tatsächlich schaltet die CPU zwischen den einzelnen Prozessen hin und her $\Rightarrow$ Zu einem bestimmten Zeitpunkt kann maximal ein Prozess auf der CPU ausgeführt werden.
- Welcher Prozess wann und wie lange auf der CPU ausgeführt werden kann, entscheidet ein **Scheduler**.
- Ein Prozess kann sich in verschiedenen Zuständen befinden (wartend, laufend, blockiert,…).

Prozesse in Ada und Unix

Ada:

```
task Process1;

task body Process1 is
   Count: Natural := 0;
begin
  loop
    Put_Line("Process1 within
    loop");
  exit when Count=10;
  end loop;
end Process1;
```

Unix:

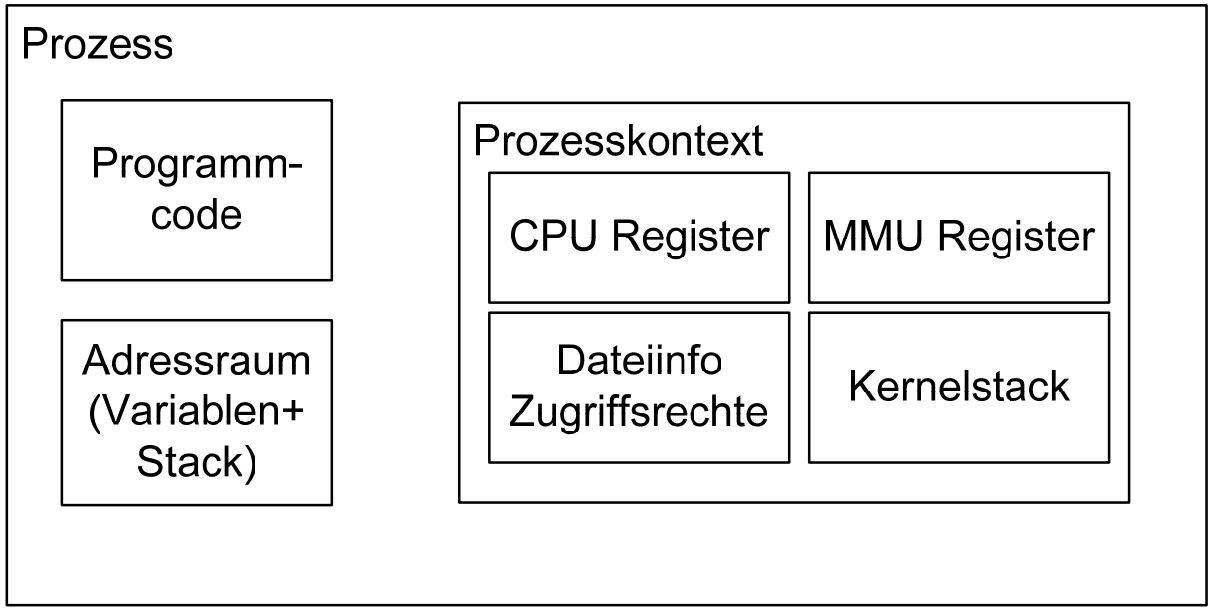
```
int main (void)
{
    pid_t pid;
    switch (pid=fork())
    {
      case -1:
            printf("error in fork\n");
      case 0:
            printf("Hello1\n");
            sleep(1); //wait for 1 second
            printf("I am the child process\n");
            sleep(2);
            printf("Good bye1\n");
      default:
            printf("Hello2\n");
            sleep(2); //wait for 1 second
            printf("I am the parent process\n");
            sleep(3);
            printf("Good bye2\n");
     }
}
```

Prozesse in Modula-2

```
MODULE Example;
CONST
    StackSize=200;
VAR
    P1,P2,Main : PROCESS;
    Stack1,Stack2 : ARRAY[1..StackSize] OF WORD;
PROCEDURE Proc2;
    VAR Count: CARDINAL;
    BEGIN
     Count:=0;
     LOOP
         Count:=Count+1;
         IF Count=10 THEN
             WriteString('Finished');
             WriteLn;
             Transfer(P2,Main);
         ELSE
             WriteString('Loop ');
             WriteCard(Count,2);
             Write(' of p2);
             TRANSFER(P2,P1);
         END
     END
    END Proc2;
```

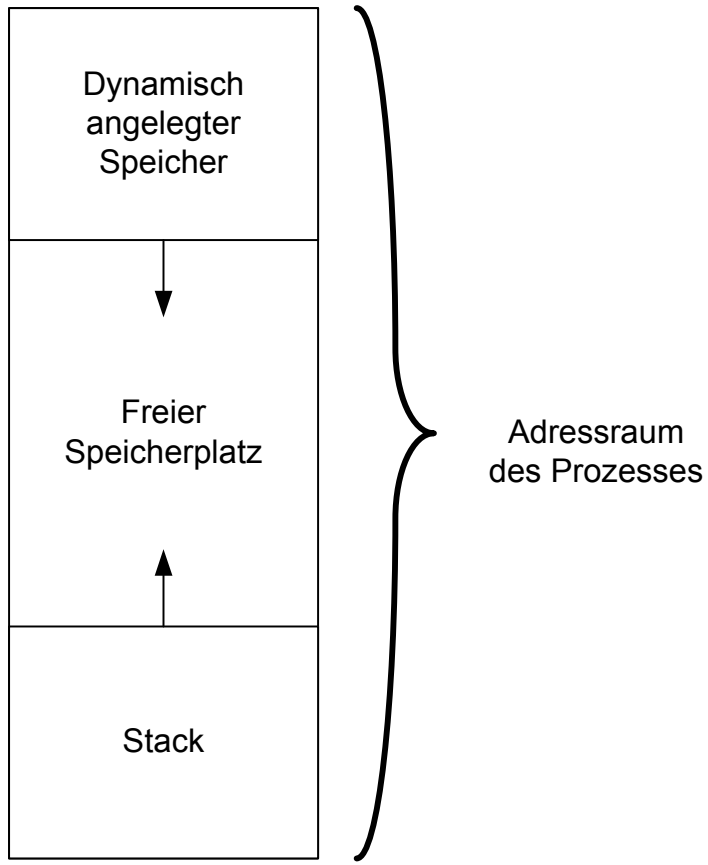
```
PROCEDURE Proc1;
    VAR Count: CARDINAL;
    BEGIN
     Count:=0;
     LOOP
         Count:=Count+1;
             WriteString('Loop ');
             WriteCard(Count,2);
             Write(' of p1');
             TRANSFER(P1,P2);
         END
     END Proc1;

BEGIN
    WriteString('Begin');
    WriteLn;
    NEWPROCESS(Proc1,ADR(Stack1),SIZE(StackSize),P1);
    NEWPROCESS(Proc2,ADR(Stack2),SIZE(StackSize),P2);
    TRANSFER(Main,P1);
    WriteString('End');
    WriteLn;
END Example.
```

Prozess: Bestandteile



Prozess: Adressraum

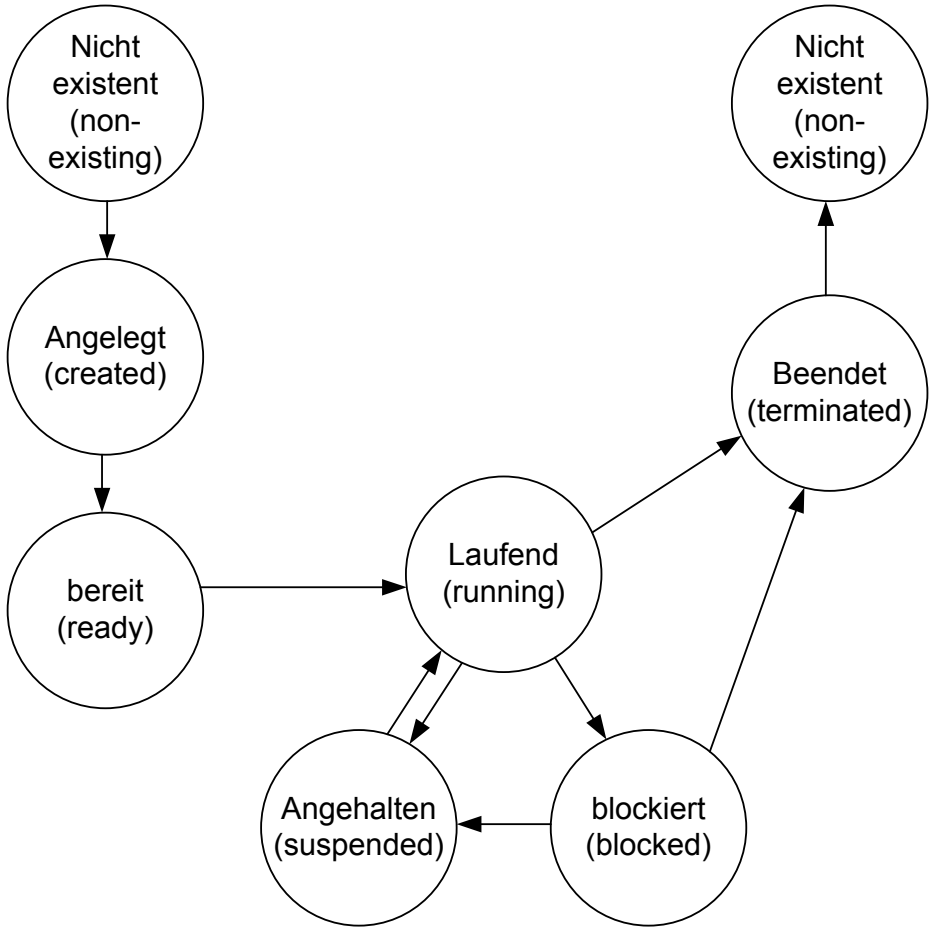
- Jedem Prozess wird zur Initialisierung ein Speicherplatz für seine Daten zugewiesen:
  - Fordert der Benutzer dynamisch Speicherplatz (z.B. in C mit der Funktion `malloc()`)an, so wird dieser an einem Ende des Speicherplatzes angelegt (Heap, Haldenspeicher).
  - Das andere Ende des Speicherplatzes wird für den *Stack* verwendet, um Platz für Parameter und Rückgabewerte beim Aufruf von Funktionen zu schaffen.
  - Beide Speicherbereiche wachsen aufeinander zu, im Falle einer Überschneidung kommt es zu dem Fehler *stack overflow*.



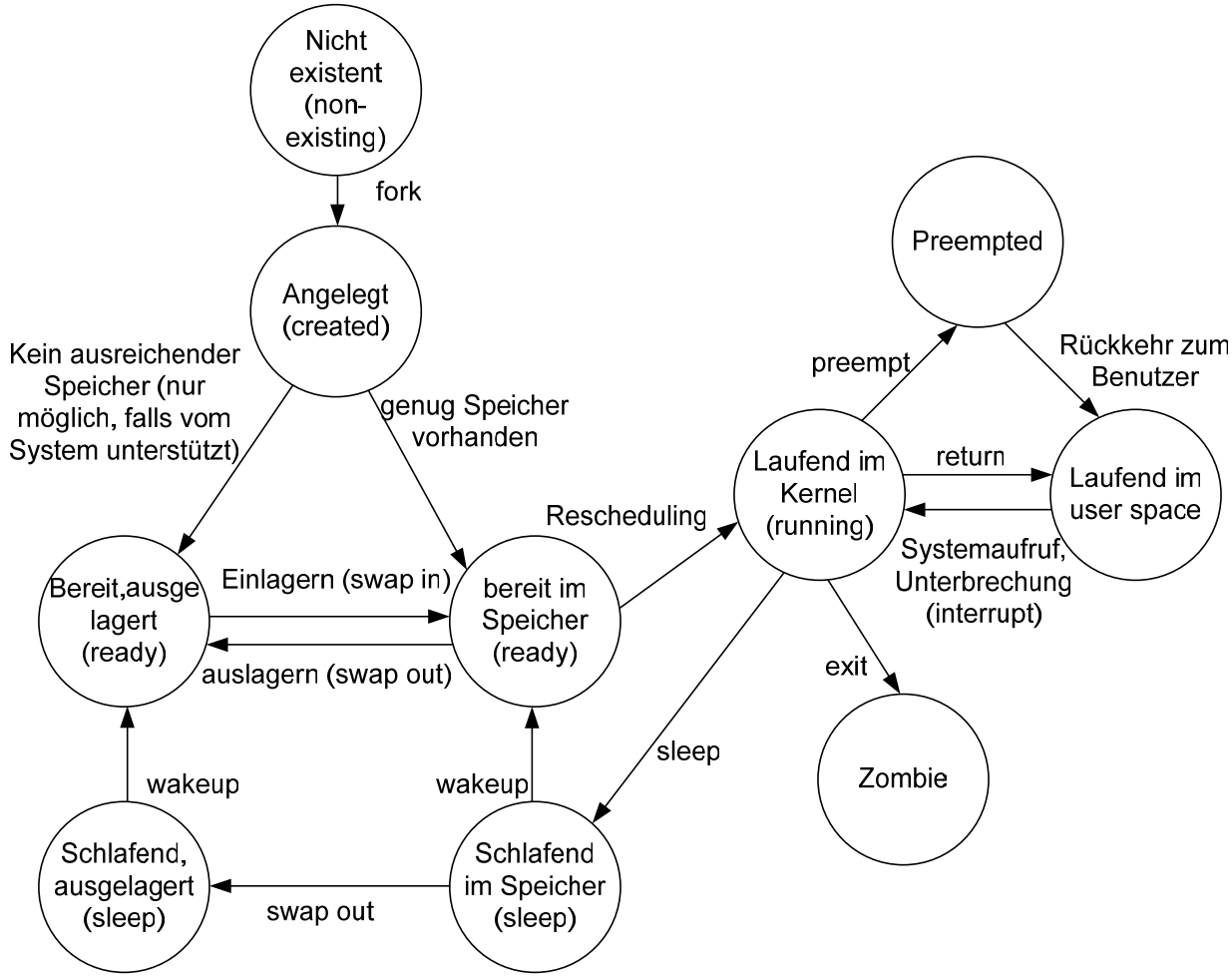
Prozesskontext

- Im Prozesskontext sind alle Informationen gespeichert, die nötig sind, um den Prozess zu verwalten (z.B. Scheduling, Rechteverwaltung, Prozesszustand…).
- Die Daten werden in einem Prozesskontrollblock (PCB) gespeichert. Dieser enthält:
  - Eindeutige Prozess-ID
  - Name des Benutzers, dem der Prozess zugeordnet ist
  - Priorität des Prozesses (siehe Scheduling)
  - Der momentane Prozesszustand (wartend/laufend/laufbereit,…)
    - Falls rechnend: der ausführende Prozessorkern
    - Falls wartend: Ereignis auf das der Prozess wartet
  - Inhalte der Register
  - Verweis auf den Adressraum des Prozesses
  - Programmstatuswort (PSW): enthält beispielsweise den Zustand der Flags des Prozessors

Prozesszustände (allgemein)



Prozeßzustände in Unix



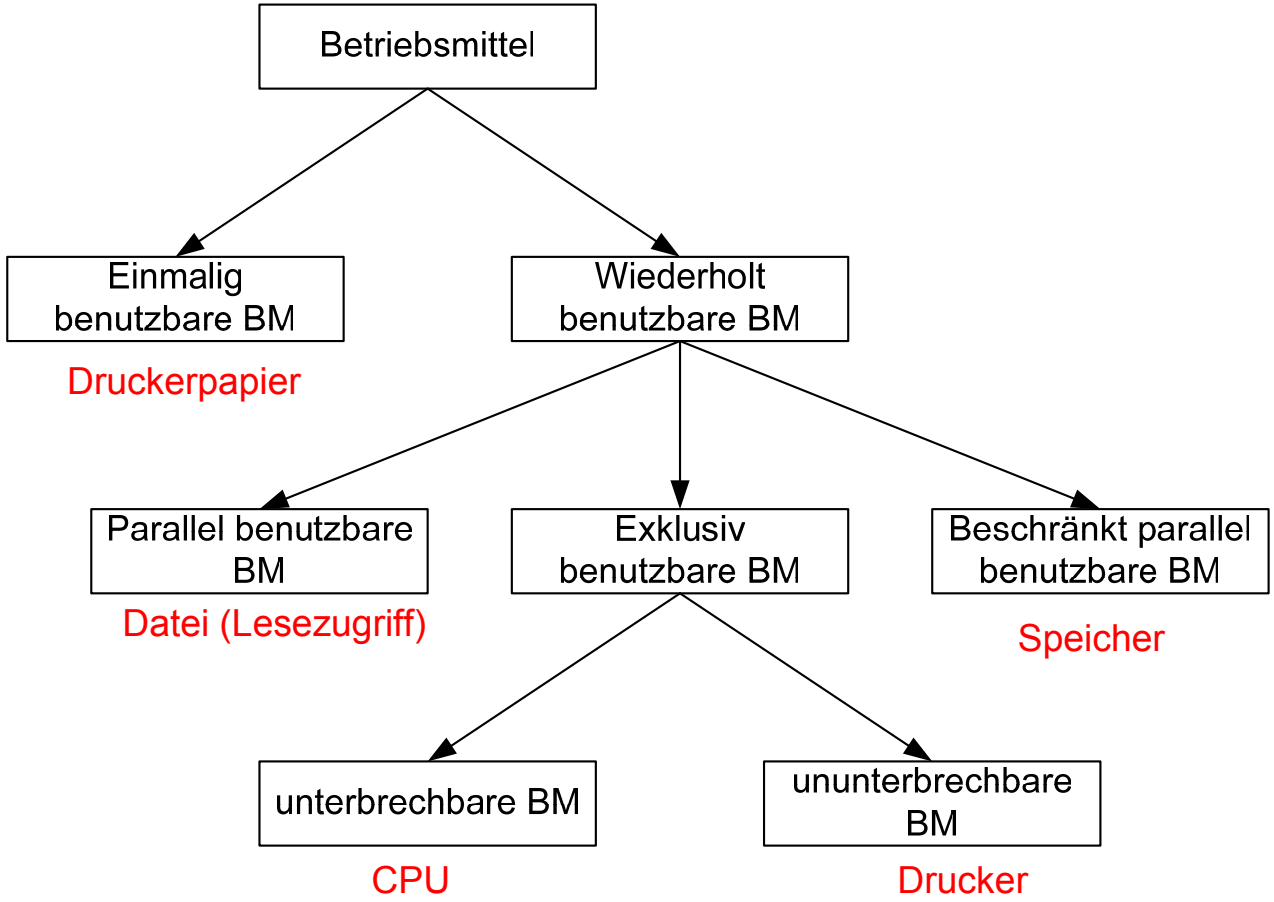
Ausführung von Prozessen

- Ein Prozess benötigt zur Ausführung verschiedene Ressourcen / Betriebsmittel:
  - Prozessorzeit
  - Speicher
  - weitere Betriebsmittel (z.B. spezielle Hardware wie CD-Laufwerk, Netzwerkkarte, Tastatur, Maus, Monitor)
- Die Ausführungszeit ist neben dem Programm auch von folgenden Parametern abhängig:
  - Leistungsfähigkeit des Prozessors
  - Verfügbarkeit der Betriebsmittel
  - Eingabeparametern
  - Verzögerungen durch andere Prozesse (Konkurrenz um Betriebsmittel)
  - bei interaktiven Programmen: von externen Ereignissen, wie z.B. Benutzereingaben

Betriebsmittel

- Zur Ausführung benötigt ein Prozess diverse Systemressourcen (Betriebsmittel). Beispiele: Prozessor, Hauptspeicher, Dateien, Drucker.
- Die Regelung des Zugriffs auf diese Systemressourcen übernimmt in der Regel das Betriebssystem.
- Entwickler müssen sich im Zusammenhang mit Betriebsmitteln mit diversen Problemen auseinandersetzen:
  - Konfliktvermeidung, falls das BM nicht von dem Betriebsmittel verwaltet wird.
  - Vermeidung von Verklemmungen (Deadlocks) bei wechselseitiger Anfordern von Ressourcen.

Betriebsmittel: Klassifikation



Ursachen für Prozeßwechsel

- Höher priorisierte Prozesse werden laufbereit
- Unterbrechungen/Interrupts
  - Timer, Ablauf der Zeitscheibe
  - Ereignisse bezüglich I/O
- Speicherfehler: die gewünschte Adresse ist ausgelagert (bei Benutzung von virtuellem Speicher), sie muss zunächst wieder eingelesen werden
- Beendigung des Prozesses: siehe spätere Folie
- Ausführung eines Systemaufrufs

Prozesswechsel (Kontextwechsel)

- Bevor auf der CPU ein neuer Prozess ausgeführt werden kann, müssen verschiedene Schritte durchgeführt werden:
  1. Sichern des Prozessorzustandes inklusive Programmzähler und Register
  2. Aktualisierung des Prozesskontrollblocks
  3. Einordnung des Prozesses in die geeignete Warteschlange (laufbereit, blockiert)
  4. Auswahl des neu auszuführenden Prozesses
  5. Aktualisierung des Prozesskontrollblocks des neuen Prozesses
  6. Laden des Memory-Managements für den neuen Prozess
  7. Wiederherstellung des Prozessor-Kontextes des neuen Prozesses

Prozessbeendigung

- Prozesse existieren auf einem Rechner nur für eine bestimmte Zeit.
- Ein Prozess kann aus folgenden Gründen beendigt werden:
  1. Normales Beenden: nach Abarbeitung der Anweisungen beendet der Prozess freiwillig (häufigste Form der Beendigung)
  2. Freiwilliges Beenden aufgrund eines Fehlers: während der Prozessausführung wird ein Fehler entdeckt und der Prozess beendet die Ausführung freiwillig. Beispiel: fehlerhafte Eingabe eines Parameters durch den Benutzer
  3. Erzwungenes Beenden aufgrund eines Fehlers: während der Prozessausführung tritt ein Fehler auf der nicht von der Anwendung behandelt werden kann. In diesem Fall wird der Prozess durch das Betriebssystem beendet.
     Beispiele: Division durch Null, Zugriff auf nicht gültige Speicherbereiche, Überlauf des Speichers (stack overflow)
  4. Beendigung durch anderen Prozess: ein berechtigter Prozess sendet ein Beendigungssignal (kill) an den Prozess.

Threads: Motivation I

- Feststellung: Häufig müssen unterschiedliche Berechnungen im gleichen Anwendungskontext erfolgen. Beispiele:
  - Reaktion auf Benutzereingaben trotz Berechnungen: Obwohl die Anwendung rechnet oder blockiert ist, soll auf Benutzereingaben reagiert werden (z.B. Benutzer soll Berechnung auch abbrechen können).
  - Unterschiedliche Berechnungen benötigen verschiedene Betriebsmittel und es ist im voraus nicht festzulegen, welche Berechnung zuerst durchgeführt werden kann.
- Diese Beispiele können auch mit Prozessen gelöst werden, allerdings ergeben sich diverse Schwierigkeiten:
  - Häufige Prozesswechsel führen zu zeitlichen Verzögerungen (aufgrund der Notwendigkeit, den gesamten Prozesskontext zu sichern)
  - Die Kommunikation zwischen den Prozessen ist kompliziert. Die Prozesse arbeiten auf den gleichen Daten, besitzen aber keinen gemeinsamen Adressraum $\Rightarrow$ erhöhtes Maß an Kommunikation um die Daten konsistent zu halten.

Threads: Motivation II

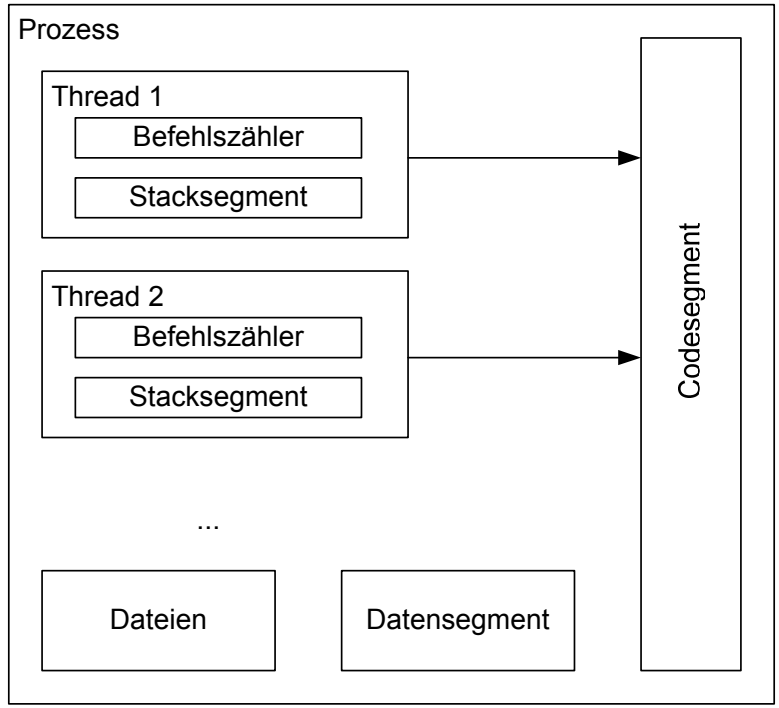
- Weiteres Probleme bei der Benutzung von Prozessen:
  - Mehrprozessor-Problematik: Wie kann eine Anwendung die Verfügbarkeit von mehreren Prozessoren (Multiprozessorsystem) ausnützen.
- Probleme bei der Implementierung von Client-Server-Anwendungen in Bezug auf den Server:
  - Programmierung des Servers als mehrere Prozesse möglich, aber hoher Aufwand zur Prozesserzeugung, Prozesswechsel und Kommunikation
  - Programmierung als ein Prozess: schwierige Verwaltung der einzelnen Clientanfragen, schlechte Performanz, da keine Parallelität möglich und keine blockierenden Aufrufe

Thread-Modell

- Das bisher verwendete Prozessmodell umfasst zwei eigenständige Konzepte: die Bündelung von Ressourcen und die (sequentielle) Ausführung des Programmcodes.
- In den meisten Betriebsystemen werden diese Konzepte getrennt:
  - Der Prozess dient der Zusammenfassung von Ressourcen, u.a.:
    - den Adreßraum
    - Rechte für verschiedene Dateien und IO-Geräte
    - Signal- und Alarminformationen
    - weitere Verwaltungsinformationen.
  - Der Thread (Ausführungsfaden) dient dagegen zur Ausführung des Programmcodes und enthält:
    - den Befehlszähler,
    - Registerinformationen
    - den Programmstack.
- Jeder Thread muß einem Prozeß zugeordnet werden können, ein Prozeß kann jedoch mehrere Threads besitzen.

Threads

- Threads werden auch *leichtgewichtige Prozesse* genannt.
- Threads werden immer einem Prozess zugewiesen und teilen sich mit diesem den Adressraum (Datensegment) und die Datei- und IO-Zugriffsrechte, besitzen jedoch einen eigenen Stack.
- Vorteil:
  - Schnellerer Wechsel möglich im Vergleich zum Umschalten zwischen schwergewichtigen Prozessen
  - Kommunikation zwischen Threads erleichtert durch gemeinsame Daten (shared memory)
- Nachteil:
  - Möglichkeit von Konflikten durch die gemeinsame Nutzung von Daten



Threads in Java

```
class MyThread extends Thread {
     private String msg;
     Mythread(String msg){this.msg=msg;}
     public void run() {
            for(int i=0; i<10; i++) {
                        try {
                                    sleep(500);
                        }
                        catch(InterruptedException e) {
                        }
                        System.out.println(msg);
            }
     }
}
public class Test {
     public static void main(String[] args) {
            MyThread myThread1,myThread2;
            myThread1 = new MyThread("Thread 1");
            myThread2 = new MyThread("Thread 2");
            myThread1.start();
            myThread2.start();
     }
}
```

Threads in POSIX (Bibliothek für C)

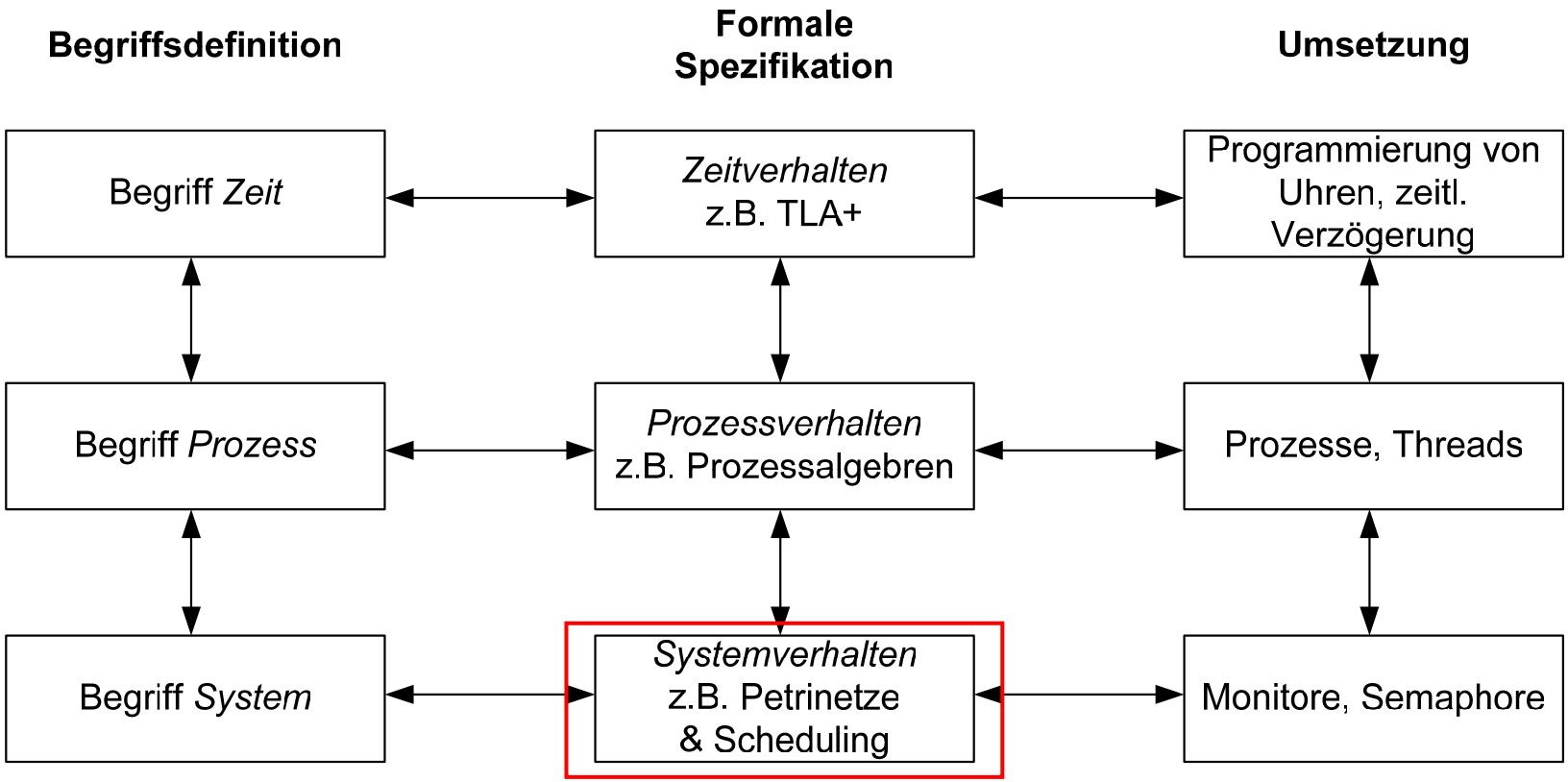
```
#include <stdio.h>
#include <pthread.h>

void* my_thread_function(void *ptr)
{
     int i;
     for(i=0;i<10;i++)
     {
            usleep(500000);
            printf("%s \n",(char*) ptr);
     }
}
int main (int argc, char** argv)
{
     pthread_t my_thread1,my_thread2;   /*process id*/
     char *msg1 = "Thread 1";
     char *msg2 = "Thread 2";
     pthread_create(&my_thread1,NULL,my_thread_function,(void*) msg1);
     pthread_create(&my_thread2,NULL,my_thread_function,(void*) msg2);
     pthread_join(my_thread1,NULL);
     pthread_join(my_thread2,NULL);
     return 0;
}
```

POSIX: Portable Operating System Interface for UniX ist eine Schnittstelle zwischen Anwendung und Betriebssystem mit dem Ziel die Portierung von Code zwischen unterschiedlichen Betriebssystemen zu erleichtern. POSIX ist in vier verschiedene Spezifikation untergliedert: Basisdefinitionen (POSIX.1), Shell und Hilfsprogramme (POSIX.2), Echtzeiterweiterungen (POSIX.4), Threaderweiterungen (POSIX.4a)

http://www.pasc.org/plato/

Scheduling

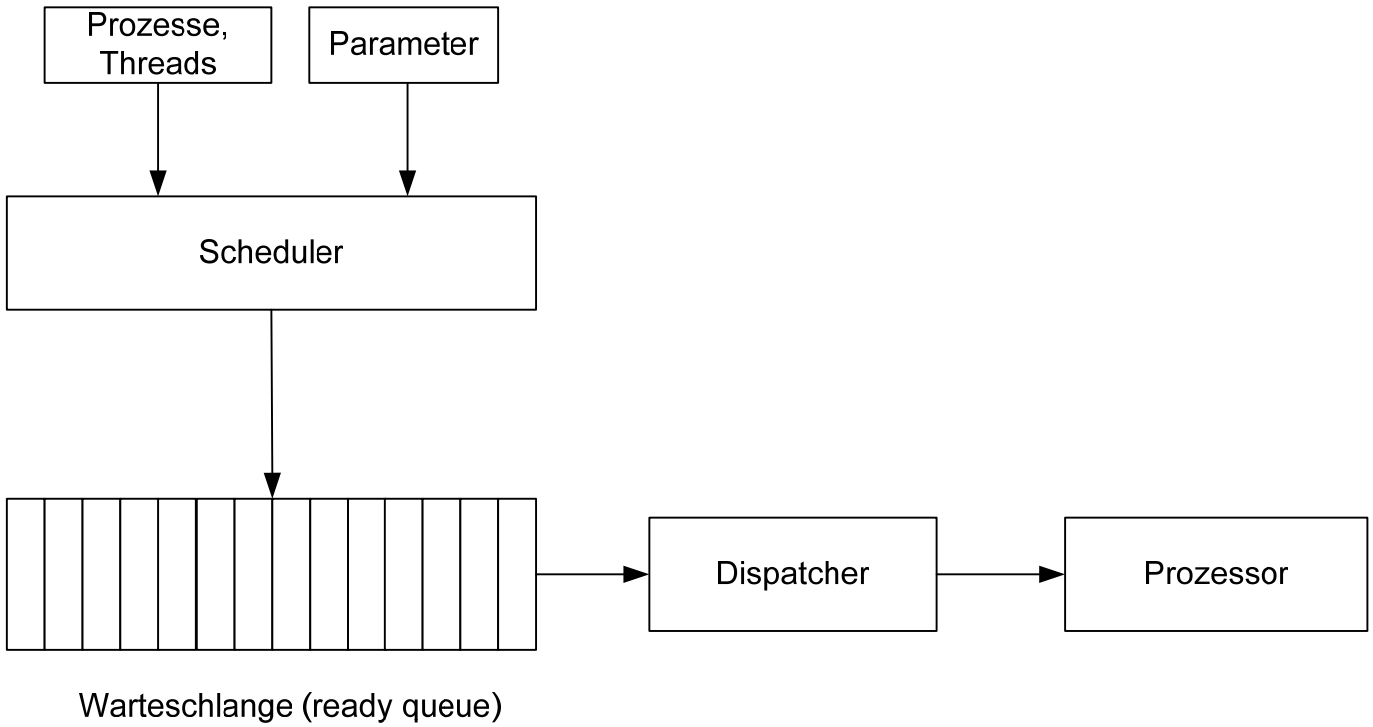
Themengebiete Nebenläufigkeit



Motivation & Definitionen

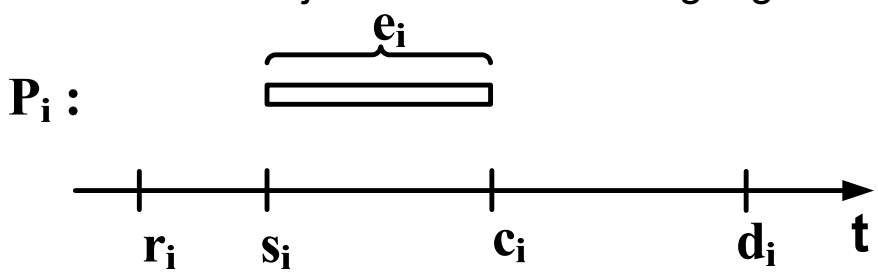
- Unterstützt ein Betriebssystem **Multitasking** (also die gleichzeitige Ausführung von mehr als einem Prozess), so muss festgelegt werden, wann welcher Prozess auf der CPU ausgeführt werden kann.
- Der **Scheduler** ist das Modul eines Betriebsystems, das die Rechenzeit an die unterschiedlichen Prozesse verteilt. Der ausgeführte Algorithmus wird als **Scheduling-Algorithmus** bezeichnet. Aufgabe des Schedulers ist also die langfristige Planung (Vergleich: Erstellung eines Fahrplans).
- Der **Dispatcher** (übersetzt: Einsatzleiter, Koordinator, Zuteiler) dient der Umsetzung eines Prozesswechsels:
  - Entzug der CPU vom aktiven Prozess
  - Zuteilung der CPU an den nächsten Prozess
  - Entscheidung über den nächsten Prozess fällt Scheduler

Scheduler & Dispatcher



Relevante zeitliche Parameter für das Scheduling

- In die Entscheidung des Schedulers fließen diverse Parameter bezüglich jedes Prozesses P_i ein:
  1. **Bereitzeit** (ready time) r_i: der früheste Zeitpunkt, an dem der Prozess dem Prozessor zugeteilt werden kann.
  2. **Startzeit** (start time) s_i: der tatsächliche Zeitpunkt, an dem zum ersten Mal der Prozessor dem Prozess zugeteilt wird.
  3. **Ausführungszeit** (execution time) e_i: Zeitdauer, die benötigt wird, um den Prozess auf dem Prozessor auszuführen.
  4. **Abschluzeit** (completion time) c_i: Zeitpunkt, zu dem die Ausführung des Prozesses beendet wird.
  5. **Frist** (deadline) d_i: Zeitpunkt zu dem die Ausführung beendet sein muss (relevant für Echtzeitsysteme).
- Nicht alle dieser Informationen sind jedem Scheduler zugänglich.



Faktoren bei der Planung

- Neben den zeitlichen Gegebenheiten müssen auch noch weitere Faktoren in die Planung eingezogen werden:
  - Art der Prozesse: Muss der Prozess
    - in regelmäßigen Abständen (**periodisch**, Beispiel: Abrufen von Emails alle 5 Minuten),
    - mehrfach, aber ohne Regelmäßigkeit (**sporadisch**, Beispiel: automatische Beantwortung von Emails)
    - oder einmalig (**nicht periodisch**, Initialisierung des Emailprogramms)

    ausgeführt werden?
  - Gemeinsame Nutzung von Ressourcen der einzuplanenden Prozesse
  - Vorrangrelationen, Beispiel: Prozess P_i muss vor dem Prozess P_j ausgeführt werden, da dieser ein Ergebnis von P_i benötigt.

Klassifikation von Planern

- Beim Scheduling wird zwischen verschiedenen Planungsarten unterschieden:
  - **offline** vs. **online**:
    - offline: Die Planung erfolgt vor Ausführung der Prozesse. Der Startzeitpunkt und Endzeitpunkt steht für jeden Prozess fest. Die Prozessmenge muss ebenfalls statisch sein.
    - online: Zur Laufzeit wird entschieden welcher Prozess ausgeführt werden soll.
  - **statische** vs. **dynamische** Planung: sind die Parameter (vor allem die Priorität eines Prozesses) veränderbar?
  - **präemtiv** vs. **nicht präemtiv**: kann der Scheduler einem Prozess den Prozessor während der Ausführung entziehen?

Offline vs. Online Scheduling

- Mit offline-scheduling wird die Festlegung eines Ausführungsplanes zur Übersetzungszeit bezeichnet.
  - Dieser Ausführungsplan wird vom Dispatcher abgearbeitet.
  - Voraussetzung: Bereitzeiten, Ausführungszeiten und Abhängigkeit der einzelnen Prozesse müssen schon im Voraus bekannt sein.
  - Problem: Die Suche nach einem solchen Plan ist in der Regel (ohne entsprechende Annahmen) nicht effizient lösbar. Deshalb werden Algorithmen verwendet, die keine optimale Pläne, sondern gute Lösungen (v.a. Einhaltung von Fristen in Echtzeitsystemen) effizient berechnen können.
- Werden die Schedulingentscheidungen zur Betriebszeit / online getroffen, so ist das System flexibler.
  - Weitere Vorteile: Kenntnisse über die einzelnen Prozesse sind nicht unbedingt nötig.
  - Im Gegensatz zur Offline-Planung müssen Mechanismen zum wechselseitigen Ausschluß verwendet werden.
  - Weitere Nachteile: Rechenzeit muss für die Berechnungen zum Scheduling verwendet werden und die Garantie zur Einhaltung von Fristen bei Echtzeitsystemen ist schwieriger.

Statische vs. dynamische Planung

- Statische Planung zur Laufzeit:
  - Alle Entscheidungen basieren auf Parametern, die schon zur Übersetzungszeit festgelegt werden.
  - Das System kann auf Eingaben der Umwelt reagieren, allerdings ist die Flexibilität eingeschränkt.
- Dynamische Planung:
  - Während des Betriebs können die zur Entscheidung des Schedulings herangezogenen Parameter verändert werden.
  - Erhöhte Flexibilität
  - Problem: Korrekte Einstellung der Schedulingparameter ist kompliziert.

Präemtives Scheduling

- Grundsätzlich werden Schedulingverfahren danach unterschieden, ob sie einem laufenden Prozess die CPU entziehen kann oder ob ein Prozess die CPU behalten kann, bis er blockiert oder sich beendet.
- Vorteil: Präemtives Scheduling ist fairer (ein Prozess kann nicht den Rechner blockieren) und sicherer (wichtige Aufgaben können bevorzugt werden).
- Beim präemtiven Scheduling kann die CPU einem Prozess entzogen werden, u.a. wenn:
  - ein wichtigerer Prozess rechenbereit wird
  - nach Ablauf einer bestimmten Zeitdauer (aus Gründen der Fairness)
- Typischerweise werden den Prozessen Prioritäten zugewiesen, um wichtige Prozesse bevorzugen zu können.
- Nachteil: häufiges Umschalten reduziert die Leistung, da Prozesswechsel aufwendig sind.

Schedulingkriterien

- Schedulingverfahren werden nach diversen Kriterien beurteilt:
  - **Fairness**: gerechte Verteilung der Prozessorzeit
  - **Effizient**: Reduzierung der Freizeiten der CPU
  - **Antwortzeit**: interaktive Prozesse sollen möglichst schnell reagieren (z.B. Maussteuerung soll möglichst ruckelfrei funktionieren, selbst wenn gerade intensive Berechnungen durchgeführt werden)
  - **Verweilzeit**: beim Abarbeiten von Aufgaben (Batchbetrieb) sollen die einzelnen Aufgaben möglichst schnell erledigt werden.
  - **Durchsatz**: Maximierung der Anzahl der Aufträge, die innerhalb einer bestimmten Zeitspanne ausgeführt werden.
- Bei Echtzeitsystemen werden diese Kriterien dem Kriterium der Einhaltung der Fristen untergeordnet.

Schedulingverfahren

- **First-Come First-Served** (Kriterium: Effizienz, Einfachheit): das einfachste Verfahren. Der Prozess der zuerst rechenbereit ist, wird ausgeführt.
- **Shortest Job First** (Kriterium: Durchsatz): es wird immer der Prozess ausgeführt, der am schnellsten abgearbeitet werden kann. Problem: die Bearbeitungszeit muss bekannt sein.
- **Shortest Remaining Time Next** (Kriterium: Durchsatz): präemptive Variante von Shortest Job First. Es wird immer der Prozess mit der kürzesten Bearbeitungszeit ausgewählt. Wird ein neuer Prozess rechenbereit, so wird seine Bearbeitungszeit mit der verbleibenden Bearbeitungszeit des gerade ausgeführten Prozesses verglichen. Ist die Zeit des neuesten Prozesses kürzer, so wird diesem Prozess die CPU zugeteilt.
- **Round Robin** (Kriterium: Fairness): Jeder Prozess erhält reihum für eine bestimmte Zeitdauer (Quantum, typischer Wert: 20-50ms) die CPU, danach wird die CPU dem nächsten Prozess zugewiesen.

Prioritätenbasiertes Scheduling

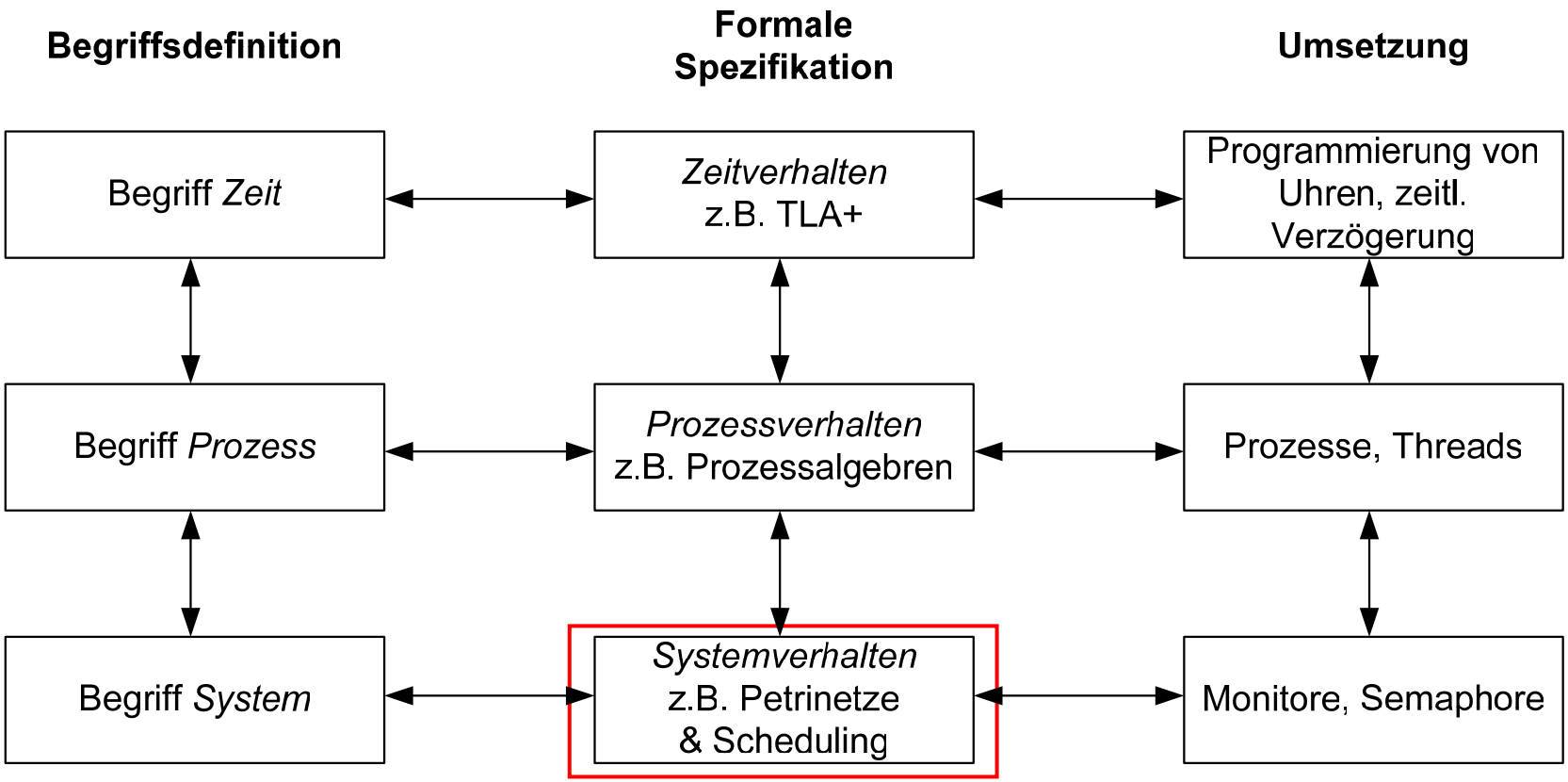
- Beim prioritätenbasierten Scheduling wird die Tatsache berücksichtigt, dass sich die Prozesse typischerweise in ihrer Wichtigkeit / Priorität unterscheiden.
- In bezug auf die Prioritäten kann zwischen statischen Prioritäten (falls sie schon zur Übersetzungszeit festgelegt werden) oder dynamischen Prioritäten (Änderung zur Laufzeit möglich) unterschieden werden.
- Prozesse höherer Priorität werden demnach bevorzugt ausgeführt, existieren mehrere Prozesse der gleichen Prioritätsklasse so wird entweder ein Prozess ausgeführt bis er sich beendet oder ein Prozess höherer Priorität rechenbereit wird (FIFO) oder die Prozessorzeit wird in Zeitscheiben eingeteilt die den Prozessen der höchsten gerade verfügbaren Prioritätsklasse reihum zugewiesen werden (Round Robin).
- **Problem**: Die Zuweisung der Priorität erfolgt durch den Entwickler. Die Abschätzung wie wichtig ein Prozess ist, fällt typischerweise schwer. Insbesondere kommt es zu Konflikten, wenn mehrere Entwickler Prozesse für ein System entwickeln (typisch: der eigene Prozess ist immer am wichtigsten).

Scheduling in Echtzeitsystemen

- Da in Echtzeitsystemen vor allem die Einhaltung der Fristen im Vordergrund steht werden im Gegensatz dazu vor allem die folgenden Schedulingverfahren verwendet:
  - **Earliest Deadline First (EDF)**: Es wird immer derjenige Prozess ausgeführt, der die früheste Frist hat. Auf einem Mehrprozessorsystem kann dieses Verfahren jedoch versagen (siehe Vorlesung Echtzeitsysteme).
  - Besser ist **Least Slack Time (LST)**: Es wird derjenige Prozess mit dem geringsten Spielraum ausgeführt. Als Spielraum wird die Zeitdauer bis zum Ablauf der Frist abzüglich der noch notwendigen Ausführungsdauer bezeichnet.
  - Da eine Implementierung beider Verfahren sehr aufwendig ist und darüber hinaus oftmals nicht vorhandene Kenntnisse (Laufzeit) notwendig ist, werden typischerweise doch wieder prioritätenbasierte Verfahren verwendet.
  - Handelt es sich bei dem Prozess um einen periodisch auszuführenden Prozess, so kann das **Rate Monotonic** Verfahren verwendet werden. Hier wird jedem Prozess eine Priorität zugewiesen, die proportional zur Ausführungsfrequenz des Prozesses ist.

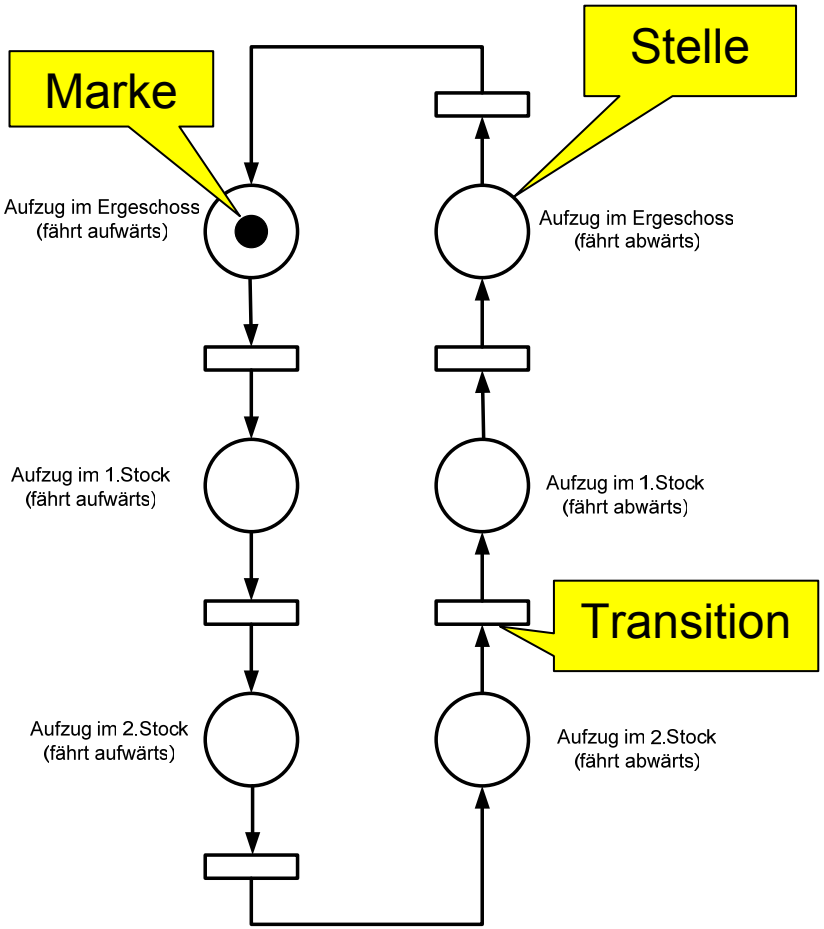
Modellierung von nebenläufigen Systemen

Themengebiete Nebenläufigkeit



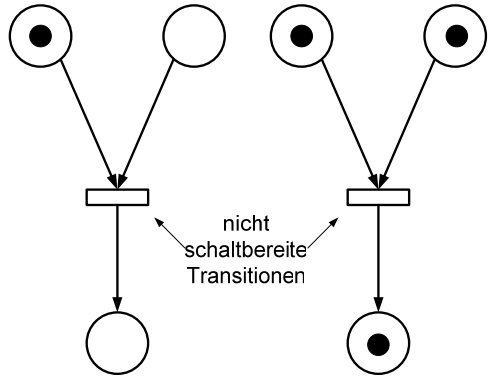
Petrinetze

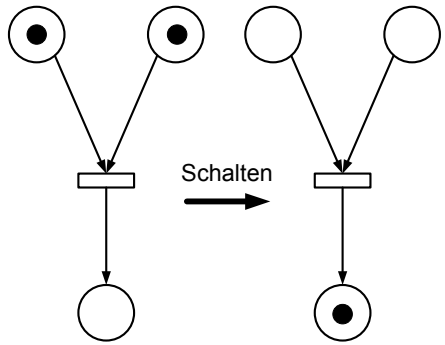
- Ein Ansatz zur Modellierung des Verhaltens von Systemen sind **Petrinetze** (1960 von Carl Adam Petri).
- Petrinetze sind gerichtete Graphen, deren Knoten abwechselnd **Stellen** und **Transitionen** sind.
- Stellen (dargestellt durch Kreise) symbolisieren dabei die Einzelzustände eines Systems, die Transitionen (dargestellt durch Rechtecke) dienen zur Spezifikation von Übergängen und ihren Bedingungen.
- **Marken** oder **Token** (dargestellt durch (gefärbte) ausgefüllte Kreise) definieren den Zustand des Gesamtsystems und können den einzelnen Stellen (vorerst maximal eine Marke pro Stelle) zugewiesen werden. Eine solche Belegung der Stellen heißt Markierung.
- Eine Stelle, die eine Marke enthält, heißt **belegt**.
- Häufig werden mit Petrinetzen unendlich lang laufende Systeme (wie z.B. der Paternoster) spezifiziert.
- http://www.informatik.uni-hamburg.de/TGI/PetriNets/



Übergänge in Petrinetzen

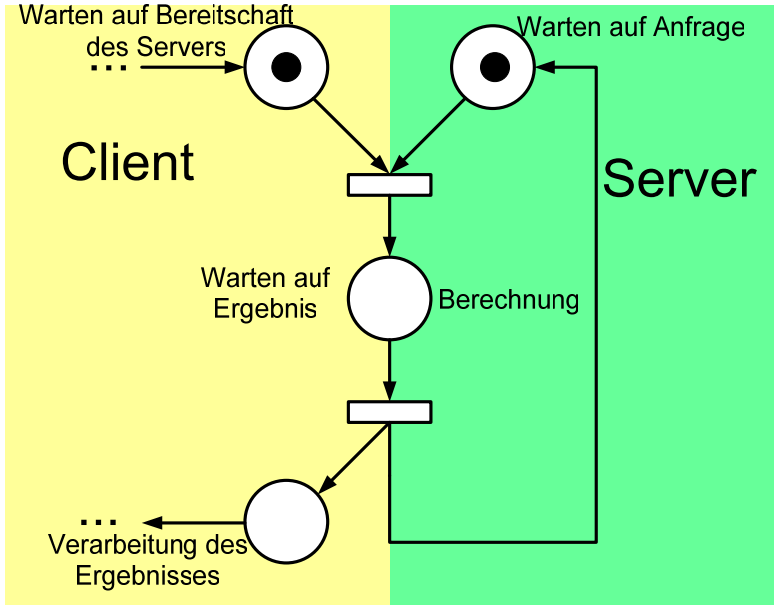
- Ein System kann seinen Zustand wechseln, indem eine „*Transition schaltet (feuert)*“.
- Eine Transition kann schalten (ist **aktiviert** bzw. **schaltbereit**), wenn jede Stelle, die eine eingehende Kante zur Transition besitzt, belegt ist und jede Stelle, die von der Transition mit einer Kante erreicht werden kann, frei ist.
- Eine aktivierte Transition kann zu einem beliebigen Zeitpunkt schalten (der Zeitpunkt wird nicht spezifiziert).
- Schaltet eine Transition, so werden die Marken der eingehenden Stellen gelöscht, in allen ausgehenden Stellen Marken erzeugt.
  $\Rightarrow$ Die Anzahl der Marken bleibt also nicht gleich (Merksatz: Marken bewegen sich nicht im Petrinetz, sie werden erzeugt und gelöscht).





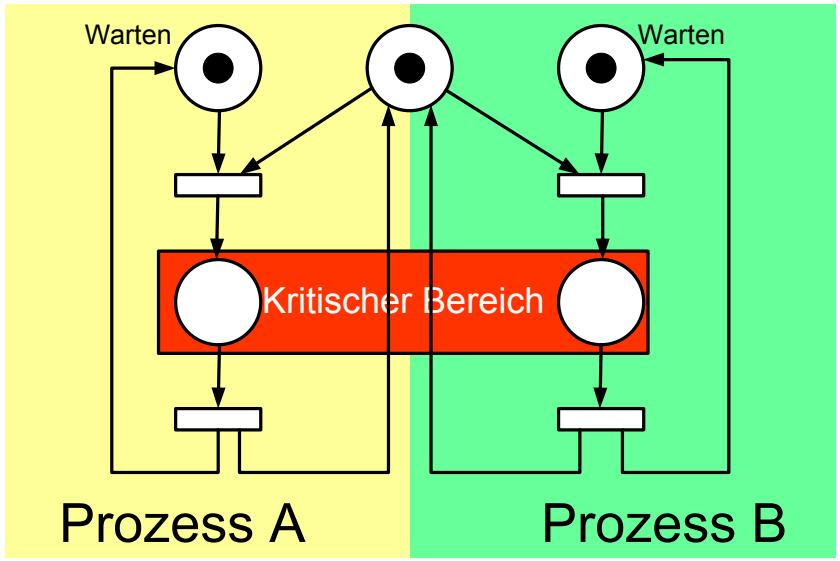
Modellierung von Synchronisationspunkten

- Die Abarbeitung von zwei Prozessen kann leicht über die Verwendung von Synchronisationspunkten miteinander abgestimmt werden.
- Beispiel: Client-Server Anwendung
  - Ein Client-Prozess stellt Anfragen an einen Serverprozess.
  - Der Server kann nur jeweils eine Anfrage gleichzeitig bearbeiten.
  - Der Server wartet immer auf die Anfrage eines Clients und berechnet dann das Ergebnis.
  - Während der Serverberechnungen blockiert der Client.
- Im Beispiel sind die beiden eingezeichneten Transitionen die Synchronisationspunkte.



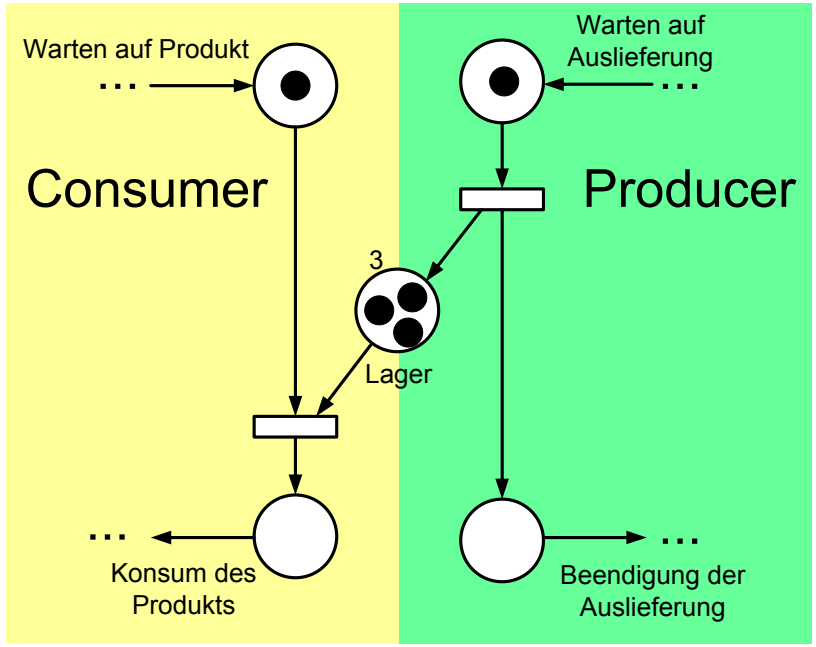
Modellierung von kritischen Bereichen

- Über die Verwendung von zusätzlichen Stellen und Marken können auch kritische Bereiche geschützt werden.
- Problemstellung
  - Zwei Prozesse greifen auf dieselben Betriebsmittel/Daten zu.
  - Dieser zeitgleiche Zugriff kann zu Inkonsistenzen führen.
- Bemerkung: eine Marke kann sich nicht aufteilen. Benötigen zwei Transitionen eine Marke gleichzeitig zum Schalten, so kann nur eine der beiden Transitionen schalten. Welche ist nicht spezifiziert $\Rightarrow$ Petrinetze sind nicht deterministisch.



Petrinetze mit mehreren Marken pro Stelle

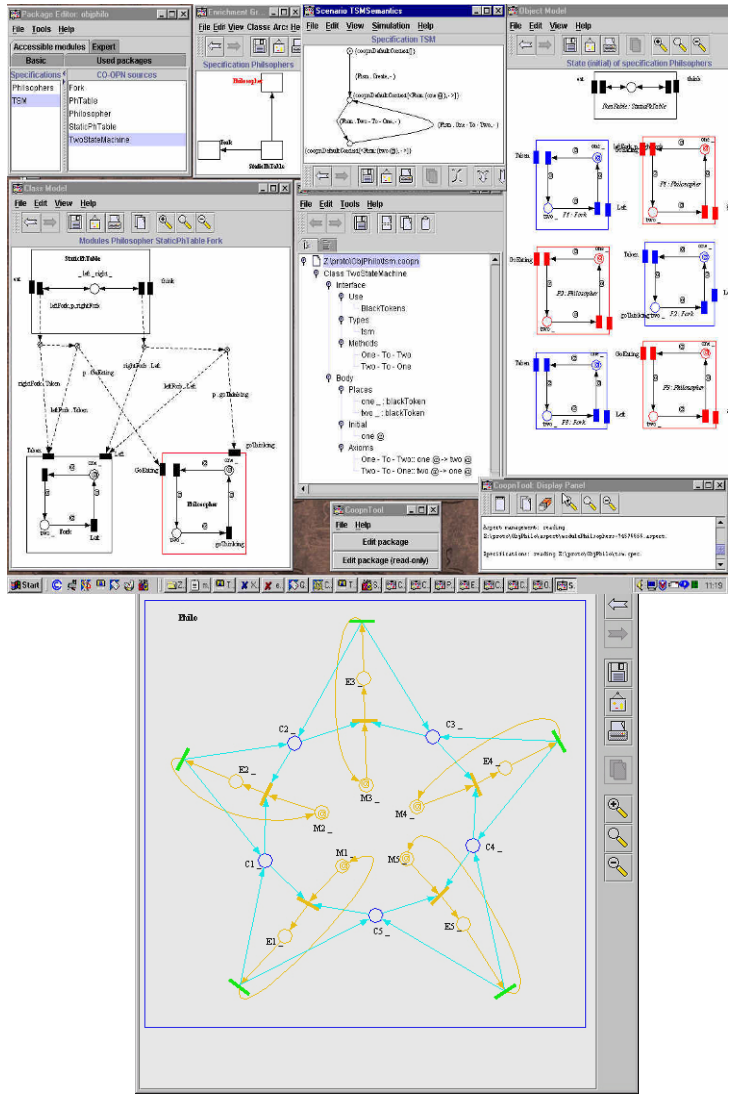
- Um z.B. das Erzeuger-Verbraucher-(Producer-Consumer)-Problem mit Hilfe von Petrinetzen einfach modellieren zu können ist es sinnvoll, wenn Stellen auch mehrere Marken aufnehmen können und man die Kapazität der Stellen explizit festlegen kann.
- Man verwendet folgende Modellierung:
  - Durch die Angabe einer Zahl bei einer Stelle kann die Kapazität spezifiziert werden (∞ bei unbeschränkter Kapazität).
  - Auch an den Kanten des Graphen können Zahlen angegeben werden, sie spezifizieren die Anzahl der Marken, die beim Schalten einer Transition gelöscht bzw. erzeugt werden.
  - Sind keine Zahlen angegeben, so ist von einer Kapazität von eins auszugehen.



Producer-Consumer-Problem mit Lagergröße 3 und vollem Lager

Eigenschaften von Petrinetzen

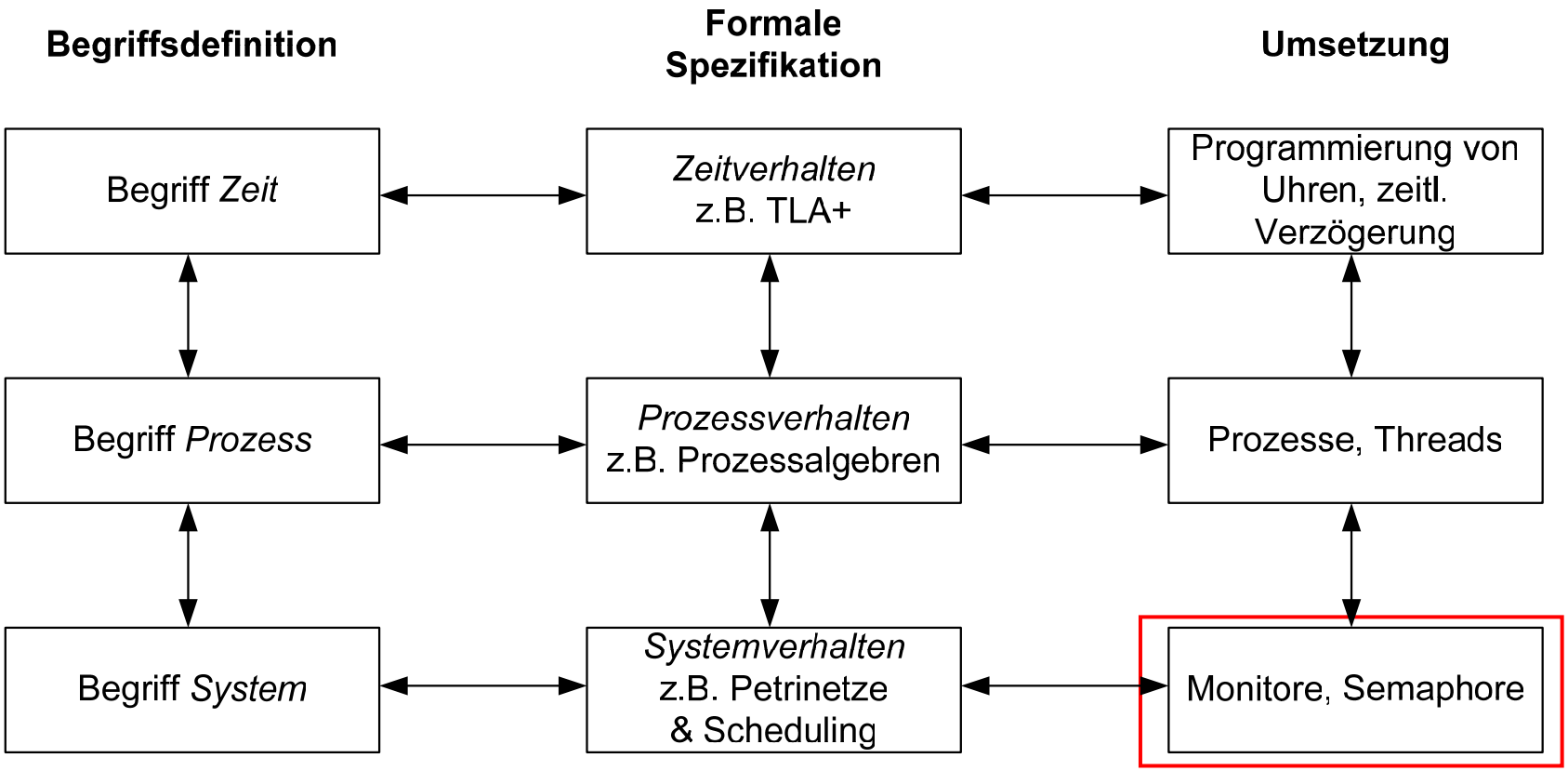
- Für Petrinetze gibt es eine große Zahl von Werkzeugen für unterschiedliche Anwendungsdomänen (von Rechnersimulation bis zu Produktionsplanungssystemen), die ein Modell einer Anwendung automatisch auf gewisse Eigenschaften untersuchen können, bspw.:
  - Eigenschaften von Transitionen für eine bestimmte Markierung:
    - Eine Transition ist **tot**, wenn aus der aktuellen Markierung keine Folgemarkierung erreicht werden kann, so daß die Transition aktiviert wird.
    - Eine Transition ist **aktivierbar**, wenn es eine Folgemarkierung gibt, so daß die Transition aktiviert ist (die Transition kann mindestens einmal schalten).
    - Eine Transition ist **lebendig**, falls sie in jeder erreichbaren Markierung aktivierbar ist (die Transition also beliebig oft schalten kann, Fairness).
  - Ein Petrinetz unter einer bestimmten Markierung ist
    - **tot**, wenn alle Transitionen tot sind.
    - **todesgefährdet**, falls eine Folgemarkierung existiert, so daß das Netz tot ist.
    - **Verklemmungsfrei** (**schwach lebendig**), wenn es nicht todesgefährdet ist.
    - (**Stark**) **lebendig**, wenn alle Transitionen lebendig sind.



Interprozesskommunikation

Synchronisation und Kommunikation

Themengebiete Nebenläufigkeit



Kritische Bereiche

- Wie bereits gesehen, ist ein großes Problem der nebenläufigen Programmierung die Existenz *kritischer Bereiche*.
- Um einen solchen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
  - Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
  - Wird verhindert, daß mehrere (unterschiedlich viele) Instanzen unterschiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem Leser-Schreiber-Problem (so dürfen beispielsweise mehrere Instanzen der Klasse `Leser` auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse `Schreiber` benötigen den exklusiven Zugriff).
- Aus dem Alltag sind diverse Mechanismen zum Schutz solcher Bereiche bekannt:
  - Signale im Bahnverkehr
  - Ampeln zum Schutz der Kreuzung
  - Schlösser für einzelne Räume
  - Vergabe von Tickets

Schutz kritischer Bereiche: Semaphor

- Semaphore (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariablen, sowie den Funktionen `down()`oder `wait()` (bzw. `P()`, von probeer te verlagen) und `up()` oder `signal()` (bzw. `V()`, von verhogen).

```
Init(Semaphor s, Int v)
{
  s = v;
}

V(Semaphor s)
{
  s = s+1;
}

P(Semaphor s)
{
  while (s <= 0) {} ;  // Blockade, unterschiedliche Implementierungen
  s = s-1 ;            // sobald s>0 belege eine Ressource
}
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muß er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- Solange der Bereich belegt ist (Wert des Semaphors <=0), wird der aufrufende Prozeß blockiert.

Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto auch beim Zugriff von zwei Prozessen konsistent gehalten werden:

Prozess A

```
P(semAccount);
x=readAccount(account);
x=x+500;
writeAccount(x,account);
V(semAccount);
```

Prozess B

```
P(semAccount);
y=readAccount(account);
y=y-200;
writeAccount(y,account);
V(semAccount);
```

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.

Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren (Realisierung nicht 100% wirklichkeitsgetreu): Damit ein Webserver nicht mit komplexen Anfragen überlastet wird, wird eine Höchstanzahl an aktuell bearbeiteten Anfragen festgelegt, z.B. 100.

Webserver

```
semaphore sem;
init(sem,100);
…
executeService;
```

Jede Anfrage führt zur Erzeugung eines Threads auf dem Server mit folgendem Code:

```
P(sem);
while(!finished)
{… query() …}
V(sem);
```

Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
  $\Rightarrow$ Die Funktionen `up()` und `down()` dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
  1. Kurzfristige Blockade der Prozeßwechsel während der Bearbeitung der Funktionen `up()` und `down()`. Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen** (**Interrupts**) ausgelöst.
  2. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den Inhalt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ. $\neq 0$, z.B. 1 für belegt) in das Speicherwort.

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung und Freigabe des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
  - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozeß zugreifen kann. Oder: Gemeinsames Objekt, in dem jede Methode einen kritischen Abschnitt darstellt.
  - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
  - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
  - Signalisierung innerhalb des Monitors festgelegt, dies braucht dem Programmierer nicht zu kümmern

Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung**: normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphor implementiert.
  Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

    synchronized public void up() {
        value++;
        notify();
    }

    synchronized public void down() {
        while(value==0) wait();
        value- -;
    }
}
```

Bemerkung zu Verklemmungen / Deadlocks

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe speisende Philosophen.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
  1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl}, die entweder frei sind oder genau einem Prozess zugeordnet sind.
  2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
  3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
  4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.

Interprozesskommunikation

- Neben der Umsetzung vom wechselseitigen Ausschluß können Semaphore auch zur Signalisierung zwischen Prozessen verwendet werden.
- Beispiel: ein Prozeß **Worker** wartet auf einen Auftrag (abgespeichert z.B. in einem char-Array job) durch einen Prozess **Contractor**, bearbeitet diesen und wartet im Anschluß auf den nächsten Auftrag:

Worker:

```
while(true)
{
        down(sem); /*wait for
                    next job*/
        execute(job);
}
```

Contractor:

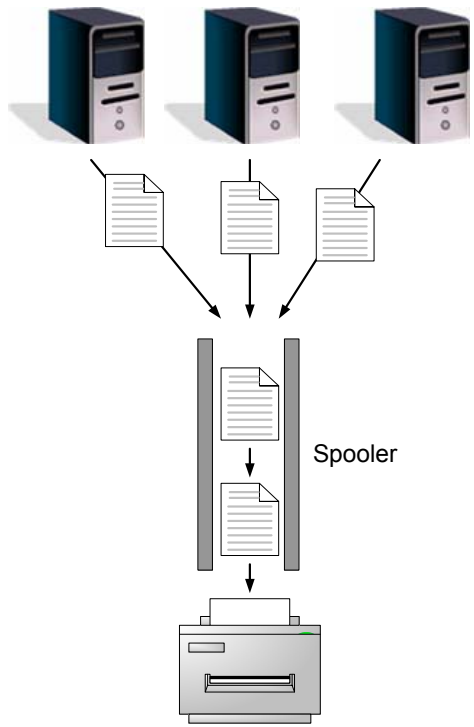
```
...
job=...  /*create new job and save
         address in global variable*/
up(sem); /*signal new job*/
...
```

Nachrichtenwarteschlangen: Motivation

- Problematisch an der Implementierung des Beispiels auf der letzten Folie ist, dass der Zeiger auf den Auftrag `job` nicht geschützt ist und es so zu fehlerhaften Ausführungen kommen kann.
- Durch Verwendung eines zusätzlichen Semaphors kann dieses Problem behoben werden.
- Ist die Zeit zwischen zwei Aufträgen zu kurz um die rechtzeitige Bearbeitung sicherzustellen, so kann es zu weiteren Problemen kommen:
  - Problem 1: Der Prozess **Contractor** muss warten, weil der Prozeß **Worker** den letzten Auftrag noch bearbeitet.
  - Problem 2: Der letzte Auftrag wird überschrieben, falls dieser noch gar nicht bearbeitet wurde. Abhängig von der Implementierung des Semaphors könnte dann der neue Auftrag zudem zweifach ausgeführt werden.

Nachrichtenwarteschlangen

- Notwendig ist also eine Methode, die eine Zwischen-speicherung von Aufträgen/Nachrichten ermöglicht.
- Der schreibende/sendende Prozess sollte nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist.
- Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher sollte der aufrufende Prozess blockiert werden bis eine neue Nachricht eintrifft.
- Beim Empfang sollte zunächst auf die älteste Nachricht zugegriffen werden.
- Eine solche Datenstruktur wird in der Informatik **Nachrichtenwarteschlange** (**message queue**) genannt.
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.

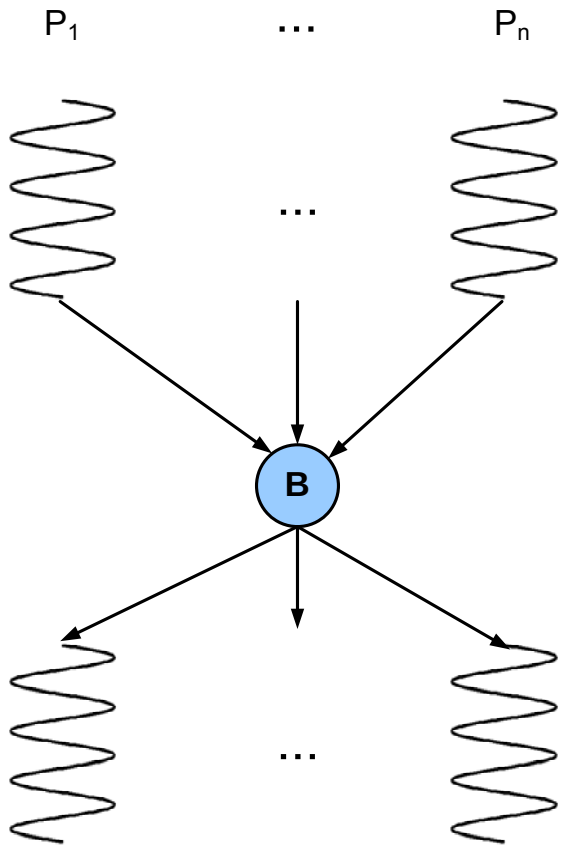


Synchrone vs. asynchrone Kommunikation

- Nachrichtenwarteschlangen erlauben eine *asynchrone Kommunikation*. Sender und Empfänger wissen jeweils nichts über den Ausführungsstand des Partners und sind somit entkoppelt.
- Müssen der Sender und der Empfänger jeweils aufeinander warten, so spricht man von *synchroner Kommunikation*.
- Vorteile synchroner Kommunikation:
  - Es kann zu keinem Speicherüberlauf/Überlastung des Empfängers kommen.
  - Antworten auf Nachrichten können einfach zurückgegeben werden.
- Vorteile asynchroner Kommunikation:
  - Es kommt zu keinen unnötigen Verzögerungen.

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i \in M$ erreichen müssen, bevor irgendein Prozess aus *M* die Berechnung über diesen Punkt hinaus fortfahren kann.
- Der Spezialfall für $|M|=2$ wird als Rendezvous bezeichnet.
- Barrieren können mit Hilfe von Semaphoren implementiert werden.



Konzept für synchrone Kommunikation: Rendezvous

- Ein Rendezvous dient zur Synchronisation von zwei Prozessen, z.B. zum Zweck der Übertragung von Daten.
- Beispiel für Rendezvous: Programmiersprache Ada
  - Die Deklaration der von einem Prozessen angebotenen Rendezvous (Empfänger) erfolgt über das Schlüsselwort `entry` in der Prozeßdeklaration.
  - Mit dem Schlüsselwort `accept` wird der Empfänger bis zur erfolgreichen Durchführung des Rendezvous verzögert.
  - Der Sender kann am Rendezvous über den Aufruf des entsprechenden Namens teilnehmen.
- Eine ausführliche Beschreibung ist unter http://www.ada-deutschland.de/AdaTourCD2004/ada_dokumentation/parallelprozesse/10_6_rendezvous.html zu finden.

Beispiel: Realisierung eines gemeinsamen Speichers (Leser-Schreiber-Problem mit Schreiberpriorität)

- Grundgerüst des Codes:
  - Deklaration eines generischen Datentyps `item`
  - Das Paket `sharedmemory` biete nach außen die beiden Funktionen `read` und `write` an.
  - Intern wird der Speicher in der Variablen `item` gesichert.
  - Zusätzlich besitzt das Paket einen Prozess `control`, der den Zugriff auf die Variable `value` überwacht.

```
GENERIC
         TYPE item IS PRIVATE

PACKAGE sharedmemory IS
         PROCEDURE read(x: OUT item)
         PROCEDURE write(x: IN item)
END;

PACKAGE BODY sharedmemory IS
         value: item;

  TASK control IS
                    ...
  END control;
  PROCEDURE read(x:OUT item) IS
  BEGIN
                    ...

  PROCEDURE write(x:IN item) IS
  BEGIN
                    ...
END shared memory;
```

Beispiel: Fortsetzung

- Schnittstelle des Prozesses `control`: der Prozess bietet insgesamt drei Funktionen als Rendezvous an: `start`, `write`, `stop`
- Die Prozedur `read` benutzt die Schnittstelle `start` zum Signalisierung des Lesebeginns und `stop` zur Signalisierung der Beendigung.
- Die Prozedur `write` benutzt die Schnittstellenfunktion `write`.
- Unterschied zwischen `read` und `write`: mehrere Leser dürfen gleichzeitig auf die Daten zugreifen, aber nur ein Schreiber.

```
TASK control IS
        ENTRY start;
        ENTRY stop;
        ENTRY write(x:in item);
END control;

PROCEDURE read(x:OUT item) IS
BEGIN
        control.start;
        x:=value;
        control.stop;
END read;

PROCEDURE write(x:IN item) IS
BEGIN
        control.write(x);
END write;
```

Beispiel (Fortsetzung): Code des Prozesses `control`

- Die Anzahl der aktuellen Leser wird in der Variable `readers` gespeichert
- Bevor ein Prozess lesend auf den Speicher zugreifen darf, muß er erstmalig beschrieben werden
- Im Anschluß führt der Prozess eine Endlosschleife mit folgenden Möglichkeiten aus:
  1. Falls kein Schreiber auf den Schreibzugriff wartet (`WHEN write'count=0`), so wird ein Schreibwunsch akzeptiert und die Anzahl der Leser erhöht, sonst wird der Wunsch bis zur Ausführung des Schreibwunsches verzögert (**Schreiberpriorität**).
  2. Beendet ein Leser den Zugriff, so wird die Anzahl erniedrigt.
  3. Falls kein Leser mehr aktiv ist (`WHEN readers=0`), werden Schreibwünsche akzeptiert, ansonsten wird dieser verzögert.

```
TASK BODY control IS
    readers: integer :=0;
BEGIN
    ACCEPT write(x:IN item) DO
        value:=x;
    END;
    LOOP
        SELECT
            WHEN write'count=0 =>
            ACCEPT start;
            readers:=readers+1;
        OR
            ACCEPT stop;
            reader:=readers-1;
        OR
            WHEN readers=0 =>
            ACCEPT write(x:IN item) DO
            value:=x;
        OR
            DELAY 3600.0;
            exit;
        END SELECT;
    END LOOP;
END control;
```

Abschlußbemerkungen

Einige Bemerkungen zum Abschluß

Was uns wichtig erscheint ...

- Die Informatik hat eine lange, beeindruckende Geschichte
- Die Informatik ist – neben den "Lebenswissenschaften" die *Leitwissenschaft* des 21. Jahrhunderts
- Trotzdem: die "Community" bzw. die ihre öffentliche Wahrnehmung ist noch (viel zu) schwach entwickelt

Was kann man tun?

- Mehr Öffentlichkeitsarbeit, mehr Werbung, mehr ... ?
- Mehr vorzeigbare Projekte ... auch während des Studiums!


http://www.tufast.de/
Erste Schritte
Aktuelle Nachrichten ...
TU fast
STUDENT RACING TEAM DER TU-MÜNCHEN
Home
News
Projekt
Technik
Rennwagen
Team
Media
Partner
Presse
FAQ
Links
Impressum
Feedback
Home
Formula Student 2006 - Teil 3
Samstag, 08 Juli 2006
That's racing! Nach einer hervorragend verlaufenden Rennwoche hat uns im wichtigsten Rennen, dem Endurance (Langstreckenrennen 22km) das Glück verlassen. Leider hat sich der Endschalldämpfer aufgrund der starken Vibrationen aus seiner Halterung gelöst und ist vom Fahrzeug gebrochen. Darauf musste uns die Rennleitung aus dem Rennen nehmen.
Das ist um so enttäuschender, als doch Florian Mühlbauer gerade dabei war, sich einzufahren und diese Einfahrrunden bereits den restlichen Favoriten das fürchten lehren konnten.
Das Potential, das der nb06 bietet ist berauschend. Man spürt seinen Durst darauf, es endlich komplett auszuschöpfen. Wir freuen uns auf den Deutschlandwettbewerb im August, wenn unser schwarzes Goldstück die Luft am Hockenheimring zum Brennen bringt.
Wettbewerbsphotos
Donnerstag, 06 Juli 2006
Dank der guten Internetverbindung konnten wir bereits einige Bilder auf die Homepage laden. Die Galerie ist zu finden unter Media / Bilder -> Wettbewerbe -> Formula Student 2006 bzw. hier. Viel Spaß beim Anschauen. Schöne Grüße aus England!
Formula Student 2006 - Teil 2
Donnerstag, 06 Juli 2006
Geschafft! Unser nb06 ist erfolgreich durch das Scruitineering gekommen. Außerdem ist noch ein Erfolg beim Noise Test zu verbuchen. Hier haben wir bei 10.000 rpm eine Lautstärke von 109.8 dB bei erlaubten 110 dB erreicht. Auch die anderen technischen Abnahmen (Tilt Table und Brake Test) sind erfolgreich verlaufen, so dass wir jetzt mit einem regelkonformen und rennbereiten Fahrzeug in den Wettbewerb starten können.
Zeitgleich haben wir mit den Static Events (Design Report, Cost Report und Presentation) begonnen. Hier wird unser Auto von den Ingenieuren auf intelligente Konstruktion, kostengünstige Produktion und überzeugende Kostenkalkulation begutachtet. Alle Wettbewerbe sind gewinnbringend verlaufen und wir freuen uns auf einen spannenden morgigen Tag mit den Dynamic Events.
Außerdem haben wir noch ein weiteres Etappenziel erreicht. Wir sind in der Endausscheidung in der Disziplin "Best use of composites" (sinnvollster Gebrauch von Verbundwerkstoffen) und damit unter den besten 6 Teams im Feld.
Formula Student 2006 - Teil 1
Donnerstag, 06 Juli 2006
Welcome to England – der Zeitplan ist wie geplant aufgegangen. Abfahrt Montagabend in Garching, Ankunft Dienstag auf dem Zeltplatz bei Bruntingthorpe. Der LKW wurde als mobile Werkstatt eingerichtet und die Mechaniker konnten dem nb06 über Nacht noch den letzten Schliff verpassen. Mittwoch Früh haben wir auf dem Wettbewerbsgelände Stellung bezogen und uns in unserer Box eingerichtet.
Dieser Mittwoch hielt für uns außerdem das Scruitineering (technische Abnahme) bereit. Hierbei werden die Fahrzeuge aller Teams von erfahrenen Ingenieuren auf ihre Regelkonformität und Renntauglichkeit geprüft. Das war bereits die erste Gelegenheit, bei der wir durch professionelles Auftreten und ein liebevoll konstruiertes Fahrzeug das erste Lob einfahren konnten. Zwar ist es uns nicht beim ersten
Downloads
Werbekatalog nb05
nb05 vs BMW M6
nb05 vs BMW M6
BMW Group
LuK
iteris
ZF
TECHNISCHE UNIVERSITÄT MÜNCHEN
formula student
I MECH E
FORMULA STUDENT


http://atknoll1.informatik.tu-muenchen.de:8080/tum6/
Go
Erste Schritte
Aktuelle Nachrichten ...
Fakultät für Informatik
der Technischen Universität München
Informatik VI: Robotics and Embedded Systems
Prof. Dr. A. Knoll , Prof. Dr. D. Burschka , Prof. Dr. J. Schmidhuber , Prof. Dr. G. Hirzinger
HOME | CONTACT | PEOPLE | RESEARCH | COURSES | PUBLICATIONS | JOIN IN
Suche
CONTACT
PEOPLE
RESEARCH
COURSES
Lectures
Seminars
Lab Courses
SEP Projects
Theses (BA/MA)
PUBLICATIONS
JOIN IN
Impressum/Legal
The Robotics and Embedded Systems group was formed in August 2001. Its primary mission is research and education of machines for perception, cognition, action and control -- extensively construed. It is organized into four research areas:
Human Robot Interaction and Service Robotics including work on the integration of speech, language, vision and action; programming service robots; development of new application scenarios for sensor-based service robots; robot systems for education;
Medical Robotics covering all aspects of manipulator and instrument control for complex surgical procedures, e.g. visualisation of all types of patient data, haptic feedback for delicate handling, skill transfer, shared control, multi-manipulator cooperation;
Cognitive Robotics encompassing a comprehensive area of topics ranging from sensor models by the way of individual sensor processing entities (e.g. for high-speed face tracking) to high-level cognitive skills for navigation, adaptation, learning;
Embedded Systems are investigated with special emphasis on fault tolerance and
Highlights/Press
Tag der Wissenschaft (SZ vom 25.10.2004)
Roboter in der Medizin (FOCUS 43/2004)
Robo Sapiens (CHIP 3/2005)
Lehrjahre der Gefühle (SZ 16.3.2005)
Die Roboter der Zukunft (AZ 24./25.3.2005)
Dienstbare Roboter (Financial Times 10.6.2005)
Japans Roboter eilen davon (SZ 11.10.2005)
Best Paper Award at MICCAI 2005 für Prof. Burschka
Best Paper Award at GECCO 2005 für Prof. Schmidhuber
Courses SS06

Fakultät für Informatik
der Technischen Universität München
Informatik VI: Robotics and Embedded Systems
Prof. Dr. A. Knoll , Prof. Dr. D. Burschka , Prof. Dr. J. Schmidhuber , Prof. Dr. G. Hirzinger

HOME | CONTACT | PEOPLE | RESEARCH | COURSES | PUBLICATIONS | JOIN IN

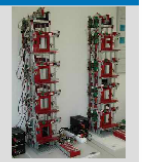
Position: > Join in Suche

CONTACT
PEOPLE
RESEARCH
COURSES
Lectures
Seminars
Lab Courses
SEP Projects
Theses (BA/MA)
PUBLICATIONS
JOIN IN
Impressum/Legal

Entry Level Research Jobs

Introduction to Real-Time Programming

Contact: Dipl.-Inf. Christian Buckl



Small projects and exercises are offered to get an introduction to real-time programming (e.g. threads,semaphors, simple scheduling algorithms) and distributed sytems (e.g. network programming, signals).

Requirements: Programming skills in C or C++ useful

Further information here .

Introduction to AdHoc Sensor Networks (ZigBee)

Contact: Dipl.-Inf. Christian Buckl

Small projects and exercises are offered to get an introduction into programming small sensor networks. We are using ZigBee as hardware.

Requirements: Programming skills in C or C++ useful

Introduction to Embedded Systems

Contact: Dipl.-Inf. Christian Buckl

Small projects and exercises are offered to learn more about programming embedded systems. We use IPC@CHIP by Beck IPC as hardware.

Requirements: Programming skills in C or C++ useful

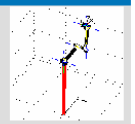
Railway Time-Table

Contact: Dr. Gerhard Schrott

In writing your own time-table for our model railway you learn everything on finite automata and train your logical abstract thinking.

Requirements: enthusiasm for model railways

Introduction to Robotics



Contact: Dipl.-Inf. Markus Rickert

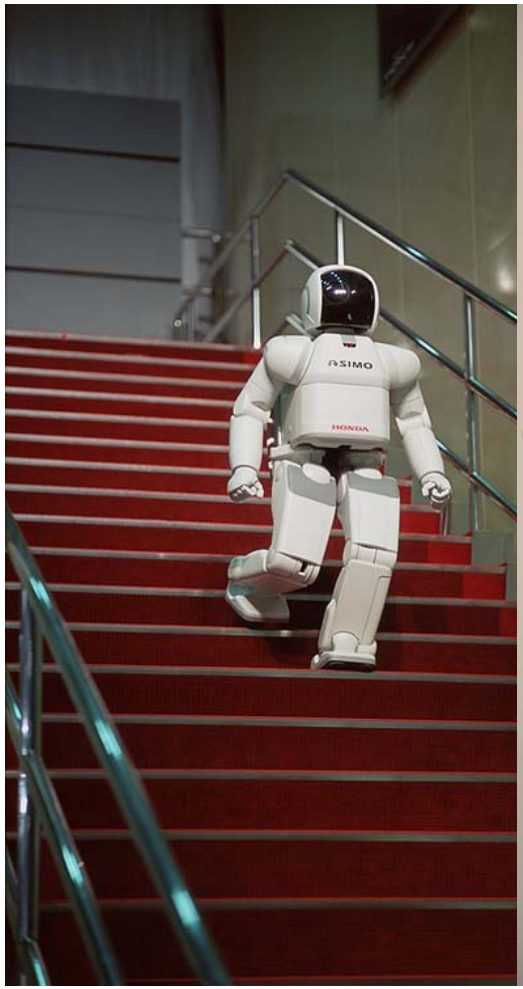
Learn about robotics by doing small projects and exercises. Topics include hardware communication, kinematics, trajectory generation etc.

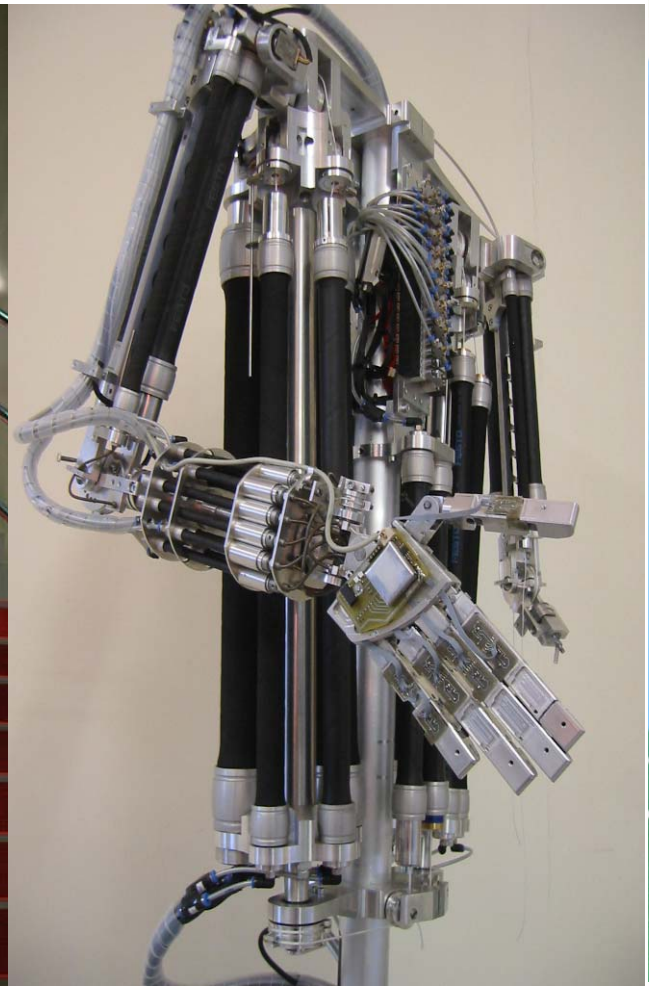
Requirements: Programming skills in C or C++ useful

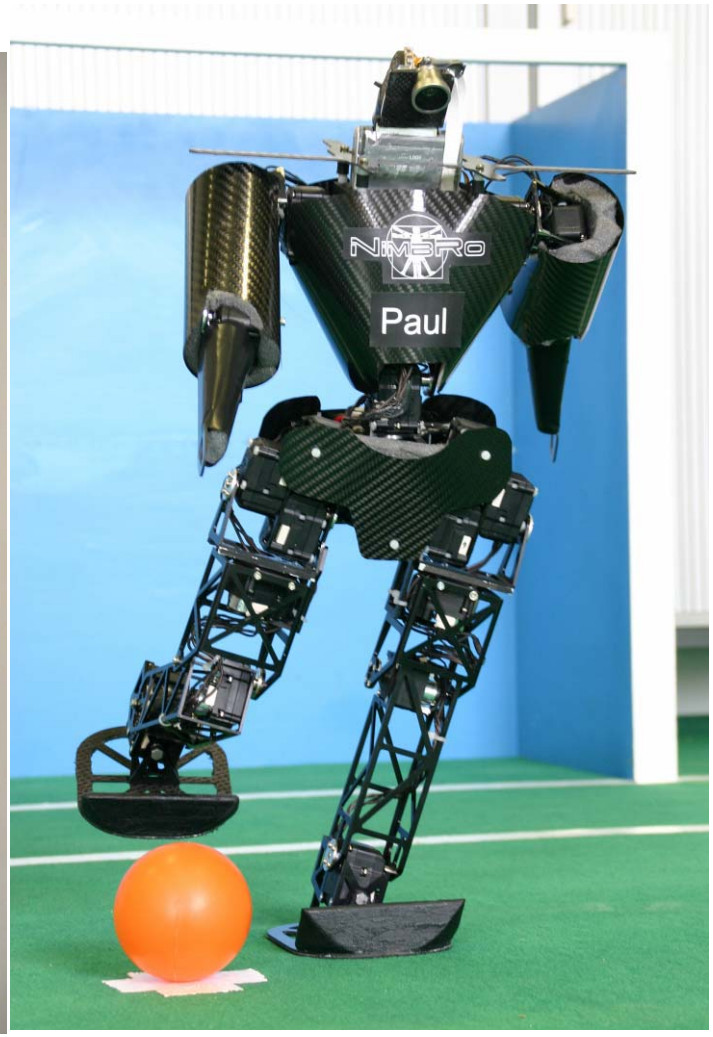
Introduction to 3D Computer Graphics and Physics Simulations

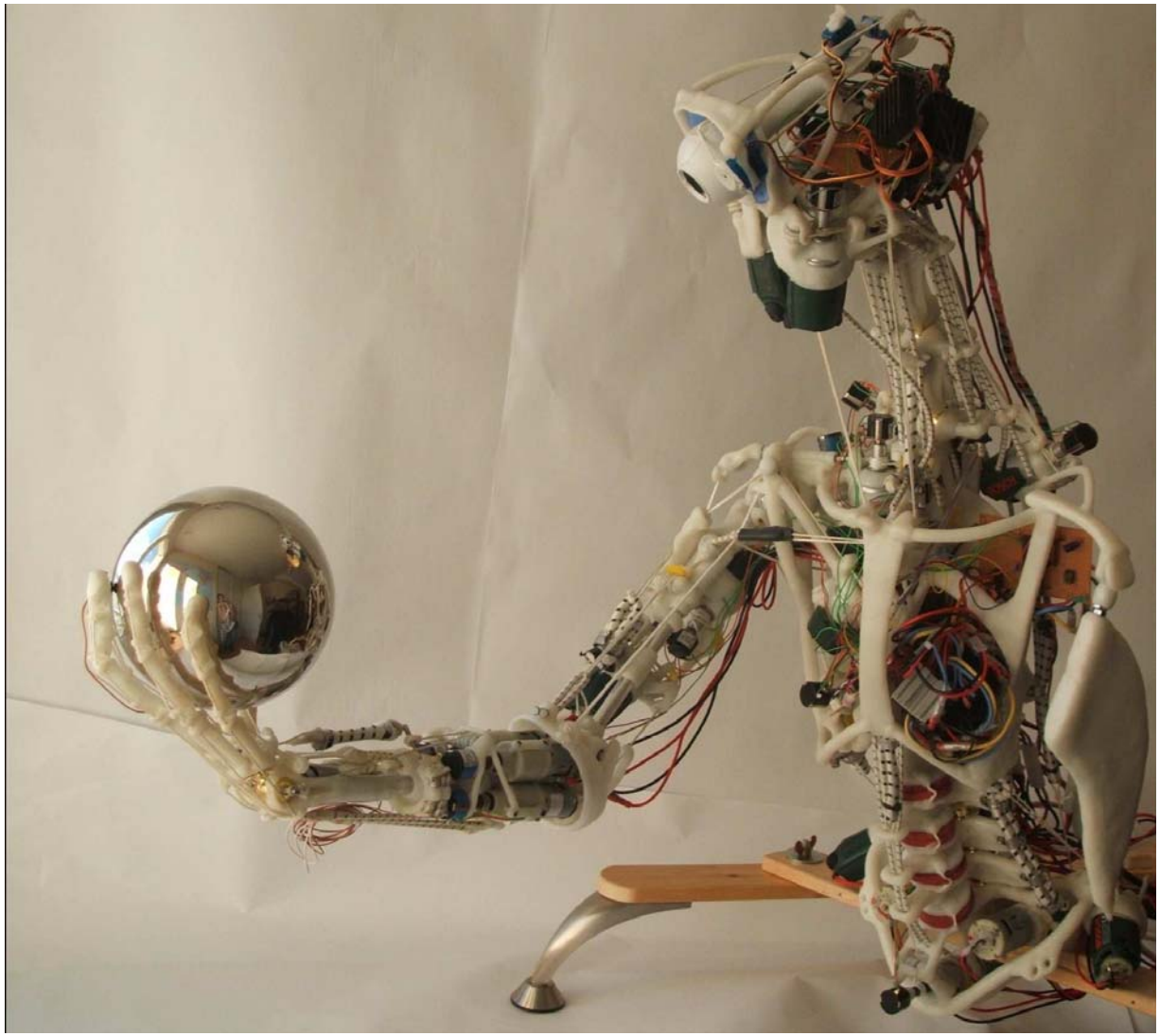
Contact: Dipl.-Inf. Markus Rickert

Learn about implementing basic 3D visualizations. Experiment with physics simulations.


ASIMO
HONDA




Paul



- Bei Interesse bitte mail an schrott@in.tum.de, Subject "Humanoiden-Projekt"

Abschlussbemerkungen - Klausur

Der Prozessbegriff

- In der Vorlesung wurde der Prozess als sequentielle Abfolge von Ereignissen (Anweisungen) eingeführt.
  - ⇒ klare Definition von Prozess, System
  - ⇒ die Ordnungsrelation über die Ereignisse ist für einen Prozess total
  - ⇒ ein *abstrakter* Prozess / ein Vorgang wie das „Anziehen“, kann durch eine konkrete Ordnung der Einzelereignisse unter Einhaltung der Randbedingungen in einen (von evtl. mehreren) gültigen Prozessen überführt werden.
- In der Informatik wird zum Teil (z.B. Broy, Übungsaufgabe) auch ein etwas anderer Prozessbegriff verwendet:
  - ⇒ Prozess und Systembegriff verschwimmen
  - ⇒ die Ordnungsrelation über die Ereignisse ist für einen Prozess partiell
  - ⇒ Im Broy´schen Modell kann ein Prozess aus mehreren parallelen Prozessen bestehen.
  - ⇒ Die Sequentialisierung eines Prozesses entspricht dem Finden eines gültigen Prozesses für einen *abstrakten* Prozess in der Begriffswelt der Vorlesung.

Klausur

- Themengebiete:
  - 1 Aufgabe Transferfragen
  - 2 Aufgaben zu Codierung und Verschlüsselung
  - 2 Aufgaben zu Nebenläufigkeit
- Umsetzung:
  - 2 Aufgaben Theorie, 3 Aufgaben Implementierung (Java,OCaml,Pseudocode)
- Punkte:
  - Insgesamt 50 Punkte erreichbar, wobei nur 40 Punkte benötigt werden.
- Hilfsmaterial:
  - Skripten (sehr **empfehlenswert**: letzte Version des kompletten Vorlesungsfoliensatzes), Übungsunterlagen, Fachliteratur… erlaubt
  - Technische Hilfsmittel sind mit Ausnahme eines nicht-programmierbaren Taschenrechners **nicht** erlaubt.
  - Arbeitsblätter sind nur die während der Klausur ausgeteilten Blätter

Was wollten wir Ihnen in dieser Vorlesung beibringen?
(Und was fragen wir Sie in der Klausur?)

- Komprimierung/Verschlüsselung:
  - Worauf basieren Verfahren zur Codierung/Komprimierung (Zeichen-/Substringhäufigkeiten) und wie funktionieren sie?
  - Wie funktionieren symmetrische/asymmetrische Verschlüsselungsverfahren?
  - Was sind Vor- und Nachteile dieser Verfahren? Was sind Kriterien für die Verwendung einzelner Verfahren?
  - Wie können Fehler in den Daten erkannt und behoben werden? Verfahren, Einsatzgebiete, Vor- und Nachteile?
  - Wie setzt man einzelne der genannten Verfahren praktisch um. Wo sind Fallstricke?

Was wollten wir Ihnen in dieser Vorlesung beibringen? II
(Und was fragen wir Sie in der Klausur?)

- Nebenläufigkeit:
  - Welche Problemklassen gibt es in der nebenläufigen Programmierung?
  - Welche Mechanismen/Lösungen gibt es in der Informatik?
  - Wie werden einzelne Probleme durch die in der Vorlesung erwähnten Mechanismen gelöst?
  - Allgemeine Fragen zum Prozess- und Systembegriff.
  - Umsetzung von Prozessen und Threads in einzelnen Programmiersprachen.

Implementierungshinweis

- Umgang mit Funktionen mit mehreren Parametern als Rückgabewert (Tupel) in Ocaml.
- Beispiel:
  `splitList list -> (smaller,pivot,bigger)`
  Die Funktion spaltet eine Liste in zwei Listen und ein Pivotelement auf, wobei die erste Liste `smaller`, alle Elemente enthält, die kleiner als das Pivotelement sind und die zweite Liste `bigger` alle Elemente größer dem Pivotelement enhält.
- **Fragestellung**: wie kann man in einer aufrufenden Funktion, die einzelnen Rückgabewerte extrahieren:
  … `let (s,p,b)=splitList list in` …
  `s`, `p` und `b` enthalten nun genau diese Rückgabewerte.