Technische Universität München
Institut für Informatik
Lehrstuhl für Effiziente Algorithmen

Skript zur Vorlesung
Einführung in die Informatik III

Prof. Ernst Mayr

27. Oktober 1995

Das Skript wurde mit den Satzsystem TEX
und Xfig von H. Lutzenberger erstellt.

Inhaltsverzeichnis

1. Grundlagen

1.1 Einleitung, Ziele der Vorlesung

Die Vorlesung "Einführung in die Informatik III" befaßt sich — einer echt Münchner Tradition folgend — mit den Grundlagen des maschinennahen Programmierens. Es wird angenommen, daß die Studenten in den beiden vorangegangenen Einführungsvorlesungen die grundlegenden mathematischen, logischen und programmiertechnischen Kenntnisse erworben und eingeübt haben. Die Ziele dieser Vorlesung sind eine Einführung in die

- grundsätzliche Struktur eines Rechners (Zentraleinheit, Speicher, Peripherie),
- grobe Struktur von Betriebssystemen verschiedener Kategorien,
- das Kennenlernen der wesentlichen, in der maschinennahen Programmierung relevanten Programmierstrukturen (d.h. Datenstrukturen und Algorithmen),
- Grundlagen der Synchronisation verteilter Abläufe,
- Struktur von Programmiersprachen und formalen Sprachen und ihre
- Übersetzung in maschinennähere Sprachen.

Zu den beiden letzten Punkten ist zu bemerken, daß hierbei ein fließender Übergang zur vierten Vorlesung in der Einführungsreihe erfolgt, so daß Teile dieser Themen erst im nächsten Semester eingehend behandelt werden.

Zunächst jedoch werden in den folgenden Abschnitten noch einige wesentliche Grundlagen behandelt, die für die Modellbildung und die mathematische Analyse der erzielten Resultate unerläßlich sind. Es handelt sich dabei zuerst um grundlegende Begriffe im Zusammenhang mit dem (asymptotischen) Wachstumsverhalten von Funktionen, sodann um Methoden, solches Wachstumsverhalten aus Rekursionsgleichungen abzuleiten, wie sie bei der praktischen Analyse von Algorithmen häufig auftreten, sowie um grundsätzliche Eigenschaften der Zahldarstellung in binären Rechenanlagen. Den Abschluß dieses einleitenden Kapitels bildet eine Darstellung, der unserer weiteren Diskussion zugrundeliegenden Rechnerarchitektur.

1.2 Wachstumsverhalten von Funktionen

Es sei $\mathbb{R}^+$ die Menge der nichtnegativen reellen Zahlen. Wir betrachten zunächst Funktionen von $\mathbb{R}^+$ nach $\mathbb{R}^+$. Sie dienen gewöhnlich dazu, die Komplexität (das kann der Zeit- oder Platzbedarf, aber auch irgendein anderes Komplexitätsmaß sein) von Algorithmen oder Problemen auszudrücken, z.B. zur Abschätzung in der Form von *oberen* oder *unteren* Schranken.

1.2.1 Definitionen

Wir sagen

(a) $f = O(g)$ falls $(\exists c > 0\, \exists r > 0\, \forall x \geq r)[f(x) \leq c \cdot g(x)]$

(aber auch, *unter gewissen Voraussetzungen*, $\lim_{x\to\infty} \frac{f(x)}{g(x)} \leq c$)

(b) $f = o(g)$ falls $(\forall c > 0\, \exists r > 0\, \forall x \geq r)[f(x) \leq c \cdot g(x)]$

(aber auch, *unter gewissen Voraussetzungen*, $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$)

(c) $f = \Omega(g)$ falls $(\exists c > 0\, \forall r > 0\, \exists x \geq r)[f(x) > c \cdot g(x)]$

(für Funktionen auf $\mathbb{N}$: $f(n) > cg(n)$ für *unendlich viele* n)

(d) $f = \omega(g)$ falls $(\forall c > 0\, \forall r > 0\, \exists x \geq r)[f(x) > c \cdot g(x)]$

(e) $f = \Theta(g)$ falls $f = O(g)$ und $g = O(f)$.

Die soeben definierten Symbole O, o, Ω, ω und Θ heißen auch Landau-Symbole. Mehr dazu kann der interessierte Leser in [Knu76] finden.

1.2.2 Variationen

Wir haben die oberen Schranken (also O und o) so definiert, daß die entsprechenden Ungleichungen für alle Argumente größer einer geeignet wählbaren Schranke gelten. Falls wir Funktionen auf $\mathbb{N}$ betrachten, kann man dafür auch sagen, daß die Ungleichungen *fast überall* bzw. *almost everywhere* gelten.

Die unteren Schranken (also Ω und ω) haben wir dagegen so definiert, daß die entsprechenden Ungleichungen lediglich für *beliebig große* Argumente gelten müssen (man beachte: es heißt hier nicht *"alle beliebig großen Argumente"*!!). Bei Funktionen auf $\mathbb{N}$ ist das gleichbedeutend damit, daß die Ungleichungen *unendlich oft* bzw. *infinitely often* gelten.

Damit sollte die Bedeutung der folgenden Schreibweisen klar sein:

$$f = \Omega_\infty(g) \text{ , oder}$$

$$
\begin{aligned}
f &= \Omega_{i.o.}(g)\ , \text{ oder} \\
f &= \Omega_{u.o.}(g)\ , \text{ bzw.} \\
f &= \Omega_{a.e.}(g)\ , \text{ oder} \\
f &= \Omega_{f.ü.}(g)\ .
\end{aligned}
$$

Entsprechende Variationen gibt es natürlich auch für ω, und es muß darauf hingewiesen werden, daß der Gebrauch in der Literatur nicht einheitlich ist. Es werdem Ω und ω oft im Sinne von $\Omega_{a.e.}$ bzw. $\omega_{a.e.}$ verwendet. Wir haben es nicht getan, einmal um *Dualität* zwischen O und ω (und auch zwischen o und Ω) zu erzielen, zum anderen auch, weil wir O und o gewöhnlich für obere Schranken benützen, die asymptotisch gelten, während wir mit Ω und ω gewöhnlich untere Schranken ausdrücken, für die es schon reicht, wenn die entsprechenden Ungleichungen (bei Funktionen auf $\mathbb{N}$) an unendlich vielen Stellen gelten.

Manchmal werden $O(\cdot)$ und die anderen Symbole auch als Klassen von Funktionen definiert, und es wird dann eine mengentheoretische Schreibweise verwendet:

$$f \in O(g) \quad \Longleftrightarrow \quad f = O(g)\ .$$

Wir können die Landau-Symbole natürlich auch für Funktionen mit negativen Funktionswerten verwenden, indem wir statt f die Funktion $|f|$ betrachten.

1.2.3 Beispiele

(a) $n = o(n \cdot \log n)$

(b) $n^{1000} = o(2^{\sqrt[1000]{n}})$

(c) $f(n) = 2^{O(n)}$ impliziert **nicht** $f(n) = O(2^n)$!!

1.2.4 Sprechweisen

Wir sagen

(a) f ist logarithmisch falls $f(n) = O(\log n)$,
(b) f ist polylogarithmisch falls $f(n) = O(\log^k n)$ für ein $k > 0$,
(c) f ist linear falls $f(n) = O(n)$,
(d) f ist polynomiell falls $f(n) = O(n^k)$ für ein $k > 0$,
(e) f ist quasipolynomiell falls $f(n) = 2^{O(\log^k n)}$ für ein $k > 0$,
(e') f ist expolylogarithmisch gleichbedeutend mit "quasipolynomiell",
(f) f ist exponentiell falls $f(n) = \Omega(2^{n^\epsilon})$ für ein $\epsilon > 0$.

Man beachte hier, daß wir alle Sprechweisen bis auf die letzte als obere Schranken eingeführt haben. Der aktuelle Gebrauch kann davon natürlich, je nach den gegebenen Umständen, abweichen.

1.2.5 Warum asymptotisches Verhalten?

Wenn wir das Verhalten von Algorithmen, Datenstrukturen oder Problemen mit Hilfe von Funktionen beschreiben, die etwa den Resourcenbedarf in Abhängigkeit zum Beispiel von der Problemgröße beschreiben, so sind damit immer gewisse Unwägbarkeiten verbunden. Ein solcher Faktor ist die Tatsache, daß z.B. der Zeitbedarf eines Algorithmus natürlich von der Taktfrequenz des verwendeten Rechners abhängt, die sich wiederum mit dem technologischen Fortschritt ändern kann. Um allgemeingültige Aussagen zu bekommen, wird man daher oft von solchen Faktoren abstrahieren und sie in der asymptotischen Notation "verstecken". Auf diese Art sind dann auch *robustere* Aussagen über das Verhalten eines Algorithmus oder die Komplexität eines Problems möglich, also Aussagen, die auch unter (vernünftigen) Modifikationen des zugrundeliegenden Modells weiterhin gültig bleiben.

Trotzdem ist es bei der konkreten Analyse von Algorithmen immer auch notwendig, Klarheit über die "versteckten Faktoren" zu bekommen, insbesondere dann, wenn mehrere Algorithmen, selbst mit unterschiedlichem asymptotischen Verhalten, miteinander verglichen werden sollen. Ein asymptotisch besseres Verfahren ist nicht unbedingt auch das bessere Verfahren für einen vorgegebenen, eingeschränkten Problembereich!

Das asymptotische Wachstumsverhalten gibt jedoch auch Hinweise darauf, wie sich der Bereich der "vernünftig lösbaren Problemgrößen" ändert, falls die Rechnergeschwindigkeit oder die Speichergröße sich etwa technologieabhängig ändern. Auch solche Abschätzungen sind für die Beurteilung der Güte und Zweckmäßigkeit von Verfahren oft unerläßlich.

1.3 Lösung von Rekursionsgleichungen

Bei der Komplexitätsanalyse von Algorithmen und anderen Methoden erhalten wir oft Gleichungen oder Abschätzungen, in denen der Aufwand für eine gewisse Problemgröße ausgedrückt wird, in Abhängigkeit vom Aufwand desselben Algorithmus für kleinere Problemgrößen. Dies tritt typischerweise bei rekursiven Verfahren auf.

1.3.1 Beispiele

1. Wir betrachten die Anzahl $V(n)$ der Vergleiche, die `MergeSort` im schlechtesten Falle (also *worst case*) beim Sortieren von n Schlüsseln durchführt, wobei wir der Bequemlichkeit halber annehmen, daß n eine Zweierpotenz ist. Dann gilt offensichtlich:

$$\begin{aligned} V(2) &= 1\,, \\ V(n) &= 2 \cdot V(\frac{n}{2}) + n - 1. \end{aligned}$$

2. Wir betrachten *vollständige Binärbäume* der Tiefe h (wobei wir festlegen, daß der aus einem Knoten bestehende triviale Binärbaum die Tiefe 0 hat) und wollen die Anzahl

der Knoten in einem solchen Baum bestimmen. Dann können wir einerseits sagen, daß sich diese Anzahl als die Summe der Anzahl der Knoten in einem vollständigen Binärbaum der Tiefe $h-1$ plus der Anzahl der Blätter ergibt (wie man leicht sieht, ist letztere gleich 2^h; ein sehr einfacher Induktionsbeweis genügt). Andererseits können wir feststellen, daß der vollständige Binärbaum der Tiefe h aus seiner Wurzel und zwei vollständigen Binärbäumen der Tiefe $h-1$ besteht. Entsprechend dieser beiden Sichtweisen erhalten wir die beiden folgenden Rekursionsgleichungen:

(α)

$$\begin{aligned} n_0 &= 1 \\ n_h &= n_{h-1} + 2^h \qquad \text{für } h \geq 1 \end{aligned}$$

(β)

$$\begin{aligned} n_0 &= 1 \\ n_h &= 2n_{h-1} + 1 \qquad \text{für } h \geq 1 \end{aligned}$$

Dabei ist n_h jeweils die Anzahl der Knoten im vollständigen Binärbaum der Tiefe h. Wir fordern den Leser auf zu überprüfen, daß beide Rekursionsgleichungen zur selben Lösung führen, nachdem er die nun folgenden Lösungsmethoden studiert hat!

1.3.2 Erraten der Lösung

Eine Lösungsmethode besteht darin, die Lösung (oder wesentliche Teile davon, etwa in Form eines Ansatzes; der Rest läßt sich eventuell aus den vorliegenden Rekursionsgleichungen ableiten) zu erraten und dann durch einen Induktionsbeweis zu verifizieren. Diese Methode setzt natürlich reichhaltige Erfahrung im Umgang mit Rekursionsgleichungen und dem Wachstumsverhalten von Funktionen voraus.

1.3.3 Die Multiplikatorenmethode

Es sei folgende Rekursionsgleichung gegeben (man vergleiche das einleitende Beispiel für `MergeSort` für eine im wesentlichen ähnliche Gleichung):

$$\begin{aligned} f_0 &= 1 \\ f_n &= 2f_{n-1} + n \qquad \text{für } n \geq 1. \end{aligned}$$

Indem wir absteigende Werte von n betrachten und die jeweiligen Gleichungungen mit geeigneten Faktoren multiplizieren, entwickeln wir daraus das folgende Tableau:

$$
\begin{aligned}
f_n &= 2f_{n-1} + n && /\cdot 1 \\
f_{n-1} &= 2f_{n-2} + (n-1) && /\cdot 2 \\
f_{n-2} &= 2f_{n-3} + (n-2) && /\cdot 4 \\
&\vdots && \\
f_{n-i} &= 2f_{n-i-1} + (n-i) && /\cdot 2^i \\
&\vdots && \\
f_1 &= 2f_0 + 1 && /\cdot 2^{n-1}
\end{aligned}
$$

Daraus ergibt sich, indem wir jede Gleichung mit dem rechts angegebenen Faktor multiplizieren, die Gleichungen aufaddieren und rechts und links gleiche Terme wegstreichen, die folgende Gleichung:

$$
\begin{aligned}
f_n &= 2^n \cdot f_0 + \sum_{i=0}^{n-1} (n-i) \cdot 2^i \\
&= 2^n + \sum_{i=0}^{n-1} \sum_{j=0}^{i} 2^j \\
&= 2^n + \sum_{i=0}^{n-1} (2^{i+1} - 1) \\
&= 2^{n+1} + 2^n - n - 2
\end{aligned}
$$

Die Umformung zur dritten Zeile ergibt sich dabei zum Beispiel, indem wir die Doppelsumme in der zweiten Zeile wie folgt graphisch darstellen:

$$
\begin{array}{r|cccccc}
n-1 & 2^0 & & & & & \\
 & & 2^1 & & & & \\
 & & & 2^2 & & & \\
 & \vdots & \vdots & \vdots & \ddots & & \\
 & & & & & & \\
0 & 2^0 & 2^1 & 2^2 & & \cdots & 2^{n-1} \\
\hline
 & i=0 & & & & & i=n-1
\end{array}
$$

Statt über die Spalten summieren wir nun über die Diagonalen.

1.3.4 Die charakteristische Gleichung

Wir betrachten die folgende Rekursionsgleichung:

$$f_0 \;=\; 0,\; f_1 \;=\; 1\,,$$

$$(*) \qquad f_n = f_{n-1} + f_{n-2} \quad \text{für } n \geq 2\,.$$

Die Gleichung $(*)$ stellt eine *lineare* (da die zu definierenden Werte f_i nur linear auftreten), *homogene* (da keine Summanden ohne einen f_i-Term auftreten) Rekursionsgleichung *zweiter Ordnung* (da f_n mittels f_{n-2} definiert wird) dar.

Wir versuchen den folgenden Lösungsansatz:

$$f_n = a^n \text{ für ein } a \neq 0\ .$$

Falls sich f_n auf diese Art darstellen läßt, muß $a^n - a^{n-1} - a^{n-2} = 0$ gelten, also, da $a \neq 0$,

$$\begin{aligned} a^2 - a - 1 &= 0\,, \\ a_{1/2} &= \frac{1 \pm \sqrt{5}}{2}\,. \end{aligned}$$

Falls $f_n = a_1^n$ und $f_n = a_2^n$ Lösungen von $(*)$ sind, ist aber auch, wie man leicht durch Einsetzen bestätigt, jede Linearkombination

$$f_n = c_1 a_1^n + c_2 a_2^n$$

mit beliebigen Konstanten c_1 und c_2 eine Lösung von $(*)$. Wir benützen nun die Anfangsbedingungen für f_0 und f_1, um diese Konstanten zu bestimmen. Durch Einsetzen ergibt sich das folgende lineare Gleichungssystem:

$$\begin{array}{lrcl} n = 0: & c_1 + c_2 & = & 0 \\ n = 1: & c_1 \frac{1+\sqrt{5}}{2} + c_2 \frac{1-\sqrt{5}}{2} & = & 1 \end{array}$$

Aus diesem System bestimmen sich die Lösungen

$$c_1 = -c_2 = 1/\sqrt{5}\,.$$

Damit haben wir folgende Lösung für f_n:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Allgemein gilt folgender Satz:

Satz 1 *Sei $p(x) = 0$ die charakteristische Gleichung für die lineare, homogene Rekursionsgleichung k-ter Ordnung*

$$(*) \qquad p_0 f_n + p_1 f_{n-1} + \ldots + p_k f_{n-k} = 0 \quad \textit{für } n \geq k$$

mit konstanten Koeffizienten p_i, also

$$p(x) = \sum_{i=0}^{k} p_{k-i} x^i = 0\,.$$

Seien r_i, $i = 1, \ldots, m$, die Wurzeln von p in den komplexen Zahlen, jede mit Vielfachheit q_i. Dann ist die allgemeine Lösung von $(*)$ durch

$$f_n = \sum_{i=1}^{m} (r_i^n \sum_{j=0}^{q_i-1} c_{ij} n^j)$$

gegeben, wobei die c_{ij} beliebige Konstanten sind (deren Wert dann natürlich durch die gegebenen Anfangsbedingungen eingeschränkt wird).

Beweis: Für einen Beweis dieses Satzes verweisen wir auf [Liu68]. ∎

Ist eine gegebene lineare Rekursionsgleichung (mit konstanten Koeffizienten) *inhomogen*, so versuchen wir zunächst mit anderen Methoden, z.B. geeigneten Multiplikatoren, daraus eine homogene lineare Rekursionsgleichung zu gewinnen, auf die dann die obige Lösungsmethode angewandt werden kann. Für eine ausführlichere Darstellung vergleiche man [Lue80].

1.3.5 Erzeugendenfunktionen

Definition 1.3.1 *Für eine Folge $(f_n)_{n\geq 0}$ ist die Erzeugendenfunktion $F(z)$ die wie folgt definierte Potenzreihe:*

$$F(z) = \sum_{n\geq 0} f_n z^n\,.$$

Seien nun $F(z) = \sum_{n\geq 0} f_n z^n$ und $G(z) = \sum_{n\geq 0} g_n z^n$, und es sei auch daran erinnert, daß

$$\frac{1}{1-z} = \sum_{n\geq 0} z^n\,.$$

Indem wir einfache Regeln für das Rechnen mit Potenzreihen benutzen, ergibt sich folgende Tabelle:

Erzeugendenfunktion	$n$tes Element der zugehörigen Folge
cF	cf_n
$F+G$	$f_n + g_n$
$F \cdot G$	$\sum_{i=0}^{n} f_i g_{n-i}$
$z^k F$	**if** $n < k$ **then** 0 **else** f_{n-k}
$F(z)/(1-z)$	$\sum_{i=0}^{n} f_i$
$z\frac{dF}{dz}(z)$	nf_n
$\int_0^z F(t)dt$	**if** $n = 0$ **then** 0 **else** f_{n-1}/n
$F(cz)$	$c^n f_n$

Beispiel:

Wir setzen

$$\begin{aligned} F(z) &= \sum_{n\geq 0} 2^n z^n = \frac{1}{1-2z}\,, \\ G(z) &= \sum_{n\geq 0} n z^n = \frac{z}{(1-z)^2}\,, \end{aligned}$$

und erhalten damit

$$F(z)\cdot G(z) = \frac{z}{(1-z)^2(1-2z)} = \sum_{n\geq 0}\sum_{i=0}^{n}(n-i)2^i z^n\,.$$

Indem wir $z/((1-z)^2(1-2z))$ in Partialbrüche zerlegen [Formelsammlung], erhalten wir

$$\frac{z}{(1-z)^2(1-2z)} = \frac{-2}{1-z} - \frac{z}{(1-z)^2} + \frac{2}{1-2z}\,,$$

und damit mit Hilfe der obigen Tabelle

$$\sum_{i=0}^{n}(n-i)2^i = 2^{n+1} - n - 2\,,$$

in Übereinstimmung (natürlich!) mit dem mittels der Multiplikatorenmethode abgeleiteten Resultat.

1.3.6 Transformation des Definitions- bzw. Wertebereichs

Eine reellwertige Folge $(f_n)_{n\geq 0}$ ist eine Abbildung $\mathbb{N} \to \mathbb{R}$. Eine Transformation der Folgenwerte wird daher auch als eine Transformation des Wertebereichs und eine Transformation der Indizes als eine Transformation des Definitionsbereichs bezeichnet.

Beispiel einer Transformation des Wertebereichs:

$$\begin{aligned} f_0 &= 1,\ f_1 = 2\,, \\ f_n &= f_{n-1}\cdot f_{n-2} \qquad \text{für } n \geq 2\,. \end{aligned}$$

Indem wir

$$g_n = \log f_n$$

setzen, erhalten wir

$$\begin{aligned} g_0 &= 0,\ g_1 = 1\,, \\ g_n &= g_{n-1} + g_{n-2} \qquad \text{für } n \geq 2\,. \end{aligned}$$

Für diese Rekursionsgleichung haben wir die Lösung bereits ermittelt: g_n ist die n-te Fibonacci-Zahl F_n; Einsetzen ergibt

$$f_n = 2^{F_n} \,.$$

Beispiel einer Transformation des Definitionsbereichs:

$$\begin{aligned} f_1 &= 1\,, \\ f_n &= 3f_{n/2} + n \qquad \text{für } n = 2^k > 1\,. \end{aligned}$$

Hier substituieren wir den Index und definieren

$$g_k = f_{2^k}$$

und erhalten damit die folgende Rekursion für die g_k:

$$\begin{aligned} g_0 &= 1\,, \\ g_k &= 3g_{k-1} + 2^k \qquad \text{für } k \geq 1\,. \end{aligned}$$

Diese Rekursionsgleichung können wir mit den früher diskutierten Methoden lösen. Wir erhalten

$$g_k = 3^{k+1} - 2^{k+1}\,,$$

und damit

$$f_n = 3n^{\log 3} - 2n\,.$$

Zum Abschluß dieses Abschnitts geben wir eine geschlossene Form der Lösung für einen häufigen Fall von Rekursionsgleichungen.

Satz 2 *Sei die Funktion T durch die folgende Rekursion definiert:*

$$\begin{aligned} T(n) &= s \cdot T(\frac{n}{m}) + a \cdot n^e \qquad \textit{für } n = m^k > 1\,, \\ T(1) &= d \end{aligned}$$

(wobei $a > 0$, $e \geq 0$, $s > 0$, und $m \geq 2$, $m \in \mathbb{N}$). Dann gilt:

$$T(n) = \begin{cases} \Theta(n^e) & \textit{falls } s < m^e\,, \\ \Theta(n^e \log_m n) & \textit{falls } s = m^e\,, \\ \Theta(n^{\log_m s}) & \textit{falls } s > m^e\,. \end{cases}$$

Beweis: Indem wir die Rekursionsgleichung teleskopartig expandieren, erhalten wir, mit $n = m^k$:

$$\begin{aligned} T(n) &= an^e + s\left(a(n/m)^e + s\left(a(n/m^2)^e + s\left(\ldots + s(a(n/m^{k-1})^e + sd)\ldots\right)\right)\right) \\ &= an^e \sum_{i=0}^{k-1} \left(\frac{s}{m^e}\right)^i + ds^k\,. \end{aligned}$$

Da $k = \log_m n$, folgt daraus

$$T(n) \;=\; an^e \sum_{i=0}^{\log_m n - 1} \left(\frac{s}{m^e}\right)^i + ds^{\log_m n} \,.$$

Aus dieser geometrischen Summe ergeben sich unmittelbar die drei Teilbehauptungen des Satzes, wenn wir bedenken, daß, falls $s < m^e$, die geometrische Summe bis auf einen konstanten Faktor gleich dem ersten Term und, falls $s > m^e$, wiederum bis auf einen konstanten Faktor gleich dem letzten Term ist. ▮

Für eine weitere einführende Diskussion von Lösungsmethoden für Rekursionsgleichungen sei auf die nachfolgend aufgeführte Literatur verwiesen, insbesondere auf [Mil60, Rei77, Rio58, Rio68].

1.4 Zahldarstellungen

Das Thema der Zahldarstellung in Rechnern ergibt sich aus der Tatsache, daß die Speicherzellen und Register eines Rechners jeweils nur eine endliche, maximale Anzahl von "Ziffern" einer Zahl speichern können, die durch die *Wortlänge* der Register bzw. des Speichers vorgegeben ist. In der Praxis auftretende Wortlängen sind z.B. 16, 32, 36, 64 und 80 Bit. Wir verwenden in diesem Abschnitt die Bezeichnung L für die Wortlänge.

1.4.1 Radixschreibweise

Sei $R > 1$, $R \in \mathbb{N}$ eine gegebene *Basis*, und sei

$$\{0, 1, \ldots, R-1\}$$

die Menge der *Ziffern* zur Basis R. Zahlen werden dann dargestellt durch Zeichenreihen

$$z = z_{n-1} z_{n-2} \ldots z_1 z_0 \in \{0, \ldots, R-1\}^+ \,,$$

und der Wert der dargestellten Zahl ist gegeben durch

$$\mathrm{Wert}(z) = \sum_{i=0}^{n-1} z_i R^i \,.$$

Wir sagen auch, daß für $i = 0, \ldots, n-1$ der Stellenwert der i-ten Stelle (in der obigen Schreibweise von rechts) gleich R^i ist.

Bemerkungen:

1. Im Hexadezimalsystem (also $R = 16$), das von sprachlichen Puristen auch oft als *Hexadekadisches System* bezeichnet wird, werden die Ziffern gewöhnlich mit

   $$0, 1, \ldots, 9, A, \ldots, F$$

   bezeichnet.

2. In der Assemblersprache für die Maschine MI können Zahlen zur Basis 2, 10 und 16 dargestellt werden. Es gilt:

$$\begin{array}{rcl} \langle\textbf{Dezimalzahl}\rangle & ::= & \{0,1,\ldots,9\}^{1,\infty}\,; \\ \langle\textbf{Binärzahl}\rangle & ::= & \text{B'}\{0,1\}^{1,\infty}\text{'}\,; \\ \langle\textbf{Hexadezimalzahl}\rangle & ::= & \text{H'}\{0,1,\ldots,9,A,\ldots,F\}^{1,\infty}\text{'}\,. \end{array}$$

3. In der Programmiersprache C können Zahlen zur Basis 8, 10 und 16 dargestellt werden. Es gilt

$$\begin{array}{rcl} \langle\textbf{Dezimalzahl}\rangle & ::= & 0|\{1,\ldots,9\}\{0,1,\ldots,9\}^{0,\infty}\,; \\ \langle\textbf{Oktalzahl}\rangle & ::= & 0\{0,\ldots,7\}^{0,\infty}\,; \\ \langle\textbf{Hexadezimalzahl}\rangle & ::= & 0\text{x}\{0,1,\ldots,9,A,\ldots,F\}^{1,\infty}\,. \end{array}$$

   Bei Hexadezimalzahlen können statt X, A, ..., F auch x, a, ..., f verwendet werden.

4. Der Microsoft-Assembler MASM (für MS-DOS) läßt Zahldarstellungen zur Basis 2, 8, 10 und 16 zu. Es gilt

$$\begin{array}{rcl} \langle\textbf{Dezimalzahl}\rangle & ::= & \{0,1,\ldots,9\}^{1,\infty}\{D\}^{0,1}\,; \\ \langle\textbf{Binärzahl}\rangle & ::= & \{0,1\}^{1,\infty}\text{B}\,; \\ \langle\textbf{Oktalzahl}\rangle & ::= & \{0,\ldots,7\}^{1,\infty}\{\text{O}|\text{Q}\}\,; \\ \langle\textbf{Hexadezimalzahl}\rangle & ::= & \{0,\ldots,9\}\{0,1,\ldots,9,A,\ldots,F\}^{0,\infty}\text{H}\,. \end{array}$$

   Auch hier können wieder statt der Großbuchstaben die entsprechenden Kleinbuchstaben verwendet werden.

5. Manchmal wird die Basis auch durch einen (in Dezimalschreibweise dargestellten) Index angegeben, so daß zum Beispiel

$$10000_{16} = 10000000000000000_2 = 65536_{10}\,.$$

Es sollte an dieser Stelle auch darauf hingewiesen werden, daß diese Liste von Darstellungsformen keineswegs vollständig ist.

1.4.2 Natürliche Zahlen

Die Wortlänge sei L, die Basis 2. Dann gilt für die Binärzahl $b_{L-1}b_{L-2}\ldots b_0$, als nichtnegative Zahl interpretiert:

$$\text{Wert}(b_{L-1}b_{L-2}\ldots b_0) = \sum_{i=0}^{L-1} b_i 2^i\,.$$

1.4.3 Ganze Zahlen

Man unterscheidet im wesentlichen die drei folgenden Darstellungsarten:

1. Vorzeichen-Betrag-Darstellung
2. Einer-Komplement-Darstellung
3. Zweier-Komplement-Darstellung

Da wir auch das Vorzeichen von ganzen Zahlen darstellen wollen, bleiben bei einer Wortlänge von L Bits für den Betrag der Zahl nur mehr $L-1$ Bits übrig. Sei nun $n = b_{L-2} \ldots b_0$ eine *positive* Binärzahl. Man beachte, daß wir hier n sowohl zur Bezeichnung der Zahl als auch ihrer Binärdarstellung verwenden. Falls nötig, verwenden wir für eine natürliche Zahl $n < 2^d$ auch $bin_d(n)$ für die dazugehörige Binärdarstellung der Länge d.

Die wesentlichen Eigenschaften der drei oben genannten Darstellungsarten lassen sich wie folgt darstellen:

	$n > 0$	$n = 0$	$-n,\ n > 0$
Vorzeichen-Betrag	$0b_{L-2} \ldots b_0$	$\substack{0 \\ 1} 0 \ldots 0$	$1b_{L-2} \ldots b_0$
Einer-Komplement	$0b_{L-2} \ldots b_0$	$00 \ldots 0$ $11 \ldots 1$	$1\overline{b_{L-2}} \ldots \overline{b_0}$
Zweier-Komplement	$0b_{L-2} \ldots b_0$	$00 \ldots 0$	$1\, bin(2^{L-1} - n)$

Bei der Vorzeichen-Betrag- und bei der Einer-Komplement-Darstellung ist also die Darstellung der 0 nicht eindeutig. Bei der Zweier-Komplement-Darstellung, die in der Praxis am häufigsten (bei Festkommazahlen) verwendet wird, gilt für den Wertebereich

$$-2^{L-1} \leq z \leq 2^{L-1} - 1\,.$$

Er ist also nicht symmetrisch. Für den Wert z der Binärzahl $b_{L-1}b_{L-2} \ldots b_0$ gilt in der Zweier-Komplement-Darstellung

$$\begin{aligned} z &= \sum_{i=0}^{L-1} b_i 2^i - b_{L-1} \cdot 2^L \\ &= \sum_{i=0}^{L-2} b_i 2^i - b_{L-1} \cdot 2^{L-1}\,. \end{aligned}$$

Wir können die Zweier-Komplement-Darstellung also auch so interpretieren, daß wir für $i = 0, \ldots, L-2$ der i-ten Stelle jeweils die Wertigkeit 2^i, der $L-1$-ten Stelle jedoch die Wertigkeit -2^{L-1}, also das Negative ihrer normalen Wertigkeit, zuordnen.

Wenn wir nun mit $\bar{n}$ das bitweise Komplement der L-Bit-Zahl $n \in \mathbb{N}$ ausdrücken (d.h. $\bar{n}$ stellt $-n$ im Einer-Komplement dar!), dann gilt $n + \bar{n} = 2^L - 1$. Daraus folgt

$$2^L - n = \bar{n} + 1\,,$$

d.h. wir erhalten die Zweier-Komplement-Darstellung von $-n$, indem wir zur Einer-Komplement-Darstellung 1 addieren!

Wir betrachten ein Beispiel für $L = 8$:

$n = 21$	0	0	0	1	0	1	0	1
Einer-Komplement $\bar{n}$	1	1	1	0	1	0	1	0
$+1$								1
$-21 = 107 - 128$	1	1	1	0	1	0	1	1

Es gilt also allgemein für $2^{L-1} > n \geq 0$:

$$\begin{array}{rc} & n \\ + & \bar{n} \\ \hline & 2^L - 1 \end{array} \qquad \begin{array}{l} 0b_{l-2}\ldots b_0 \\ 1\overline{b_{L-2}}\ldots\overline{b_0} \end{array}$$

d.h.:

$$n + \bar{n} = 2^L - 1 \qquad \Rightarrow \qquad 2^L - n = \bar{n} + 1$$

Eine Übersicht über die Darstellung ganzer Zahlen zeigt die folgende Tabelle:

	VZ	Signifikand		
positiv	0	11...11		
	⋮	⋮		
	0	00...01		
Null	0	00...00		
negativ	1	11...11		
	⋮	⋮		
	1	00...01		
	1	00...00	„unbestimmt“ z.B. bei unmaskierter Unterbrechung	
	b_{L-1}	$\underbrace{b_{L-2}\ldots b_0}$	*Intel*	*MI*
		15bits	Wort-Integer	H
		31bits	Short-Integer	W
		63bits	Long-Integer	

Gebräuchliche Werte für L sind:

MI: 8, 16, 32
Intel: 8, 16, 32, 64, $\boxed{80}$

1.4.4 Gleitpunktzahlen

In der Norm IEEE 754 wird die Darstellung der Gleitpunktzahlen wie folgt definiert:

$L-1$	$L-2 \quad\quad n$	$n-1 \quad\quad 0$
VZ	geshifteter Exponent	Signifikand
v	e	f

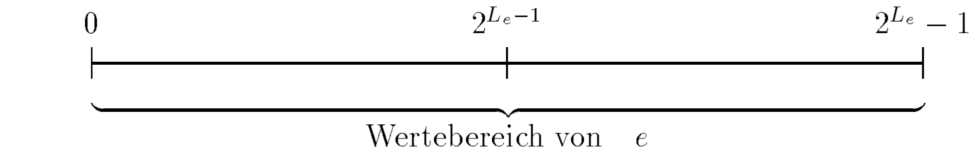
Den Wert einer Gleitpunktzahl berechnet man mit Hilfe der Gleichung:

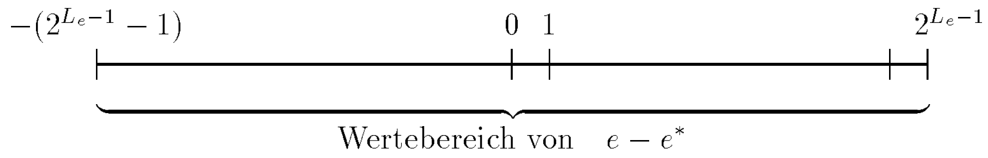
$$\begin{aligned} \text{Wert} &= \text{Vorzeichen} \cdot \text{Mantisse} \cdot 2^{\text{Exponent}} \\ &= v \cdot (1+f) \cdot 2^{e-e^*} \end{aligned}$$

wobei

$$\begin{aligned} v &= (-1)^{b_{L-1}} \\ f &= \sum_{i=0}^{n-1} b_{n-1-i} 2^{-1-i} \\ e &= \sum_{i=n}^{L-2} b_i 2^{i-n} \\ e^* &= 2^{L-n-2} - 1 \\ &= 2^{L_e-1} - 1 \qquad (\text{mit} \quad L_e = L-n-1 = \#\ \text{Bits für den Exponenten.}) \end{aligned}$$

Damit ergeben sich für den geshifteten Exponenten e und den Exponenten $e-e^*$ die Wertebereiche:





Um Platz einzusparen speichert man für die Mantisse nur den Wert $f = \text{Mantisse} - 1$ ab, d.h.

$$\text{Mantisse} = 1 + f \quad \text{mit} \quad 0 \le f < 1$$

.

Definition 1.4.1 *Diese Darstellung heißt normalisierte Gleitpunkt(komma)zahl (GPZ).*

Beispiel:

$$L = 32, \quad L_e = 8 \qquad \Rightarrow \qquad e^* = 2^{L_e - 1} - 1 = 127$$

$$\begin{aligned} & 1 \vdots 10000001 \vdots 1010 \ldots 0 \\ \hat{=} \; & (-1) \cdot (1 + \frac{1}{2} + \frac{1}{8}) \cdot 2^{129-127} \\ = \; & -(1 + \frac{1}{2} + \frac{1}{8}) \cdot 2^2 = -6,5 \end{aligned}$$

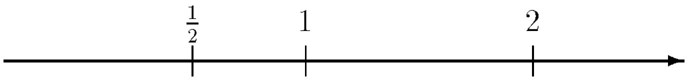
Binärdarstellung: $\quad -1.1010 \ldots 0 \cdot 2^2 = -110.10 \ldots 0$

Beobachtungen:

1. $$\begin{aligned} e_{max} &= 2^{L_e} - 1 & \Rightarrow \quad e_{max} - e^* &= 2^{L_e} - 1 - 2^{L_e - 1} + 1 = 2^{L_e - 1} \\ e_{min} &= 0 & \Rightarrow \quad e_{min} - e^* &= -(2^{L_e - 1} - 1) \end{aligned}$$

2. Die Mantisse (hier: $v \cdot (1 + f)$) besitzt Vorzeichen - Betrag - Darstellung.

3. Die Anzahl der reellen (rationalen) Zahlen zwischen $\frac{1}{2}$ und 1 ($e - e^* = -1$), bzw. zwischen 1 und 2 ($e - e^* = 0$), die durch normalisierte GPZ darstellbar sind, ist gleich.



4. Normalisierte GPZ sind unter den Rechenoperationen + , - , · , / *nicht* abgeschlossen. Es kann zum Verlust von Genauigkeit (Stellenverlust) in der Mantisse und zum Unter- / Überlauf des Exponenten kommen. Um einen Unter-/ Überlauf des Exponenten zu erkennen, schließen wir e_{max} und e_{min} als Werte für den geshifteten Exponenten e in der *normalisierten* GP-Darstellung aus!

Bei der Darstellung von Gleitpunktzahlen unterscheidet man zwischen 5 **Fällen:**

1. $0 < e < e_{max}$: Normalfall
   $z = (-1)^{b_{L-1}} \cdot (1 + f) \cdot 2^{e - e^*}$

2. $e = 0, \quad f = 0$: Null
   $z = \pm 0$

3. $e = 0, \quad f \neq 0:$ Unterlauf (denormalisierte GPZ)
$z = (-1)^{b_{L-1}} \cdot f \cdot 2^{0-e^*+1}$

4. $e = e_{max}, \quad f = 0:$ Überlauf $\quad z = \pm\infty$

5. $e = e_{max}, \quad f \neq 0:$ NaN ($\hat{=}$ not a number, d.h. „undefiniert")

Eine Übersicht über die Darstellung von GPZ zeigt die folgende Graphik:

	VZ	Gesch.Exp.(e)	Signifikand	
NaN	0	11…11	11…11	
	⋮	⋮	⋮	
	0	11…11	00…01	
$+\infty$	0	11…11	00…00	
norm. GPZ positiv	0	11…10	11…11	
	⋮	⋮	⋮	
	0	00…01	00…00	$\hat{=} 1 \cdot 2^{1-e^*}$
denorm. GPZ positiv	0	00…00	11…11	$\hat{=} (1-\varepsilon)2^{-e^*+1}$
	⋮	⋮	⋮	
	0	00…00	00…01	
$+0$	0	00…00	00…00	
-0	1	00…00	00…00	
denorm. GPZ negativ	1	00…00	00…01	
	⋮	⋮	⋮	
	1	00…00	11…11	
norm. GPZ negativ	1	00…01	00…00	
	⋮	⋮	⋮	
	1	11…10	11…11	
$-\infty$	1	11…11	00…00	
NaN	1	11…11	00…01	
	⋮	⋮	⋮	
	1	11…11	10…00	$\leftarrow$ „undefiniert" bei mask. Interrupt
	⋮	⋮	⋮	
	1	11…11	11…11	

In der Praxis vorkommende Formate sind:

	L	L_e	e_{max}	e^*	n	Genauigkeit (Dezimalziffern)
kurze GPZ(F)	32	8	255	127	23	6 - 7
lange GPZ(D)	64	11	2047	1023	52	15 - 16
Coprozessor	80	15	32767	16383	64[†]	19

[†]enthält gesamte Mantisse

1.4.5 Redundante Zahldarstellung

In dieser Darstellung besitzen die Zahlen die Form $\sum_{i=0}^{n} b_i \cdot 4^i$, d.h. sie werden zur Basis 4 dargestellt. Die b_i's können Werte aus der Menge $\{-3, -2, -1, 0, 1, 2, 3\}$ annehmen.

Beispiele:

1 -1 $\hat{=} 1 \cdot 4^1 + (-1) \cdot 4^0 = 3$
2 -2 1 0 $\hat{=} 2 \cdot 4^3 + (-2) \cdot 4^2 + 1 \cdot 4^1 + 0 \cdot 4^0 = 100$.

Der Vorteil dieser Darstellung ist darin begründet, daß es zu keinem kaskadierenden Überlauf bei der Addition/Subtraktion kommt. Man beginnt die Addition von $a+b$ wie gewohnt. Es werden a_i und b_i addiert und als $4 \cdot c_i + s_i$ dargestellt. Wegen der Redundanz der Zahldarstellung gibt es für c_i und s_i bei festem Wert von $a_i + b_i$ verschiedene Möglichkeiten. Wir wählen die folgende Darstellung:

$a_i + b_i$	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
s_i	-2	-1	0	1	-2	-1	0	1	2	-1	0	1	2
c_i	-1	-1	-1	-1	0	0	0	0	0	1	1	1	1

Berechnung von $a + b$:

1. Bestimme die Ziffern s_i, c_i nach obiger Tabelle
2. Shifte c um eine Stelle nach links und addiere dieses geshiftete c mit s Stelle für Stelle unter Verwendung der Ziffern -3,...,+3. Dabei entsteht kein neuer Übertrag!

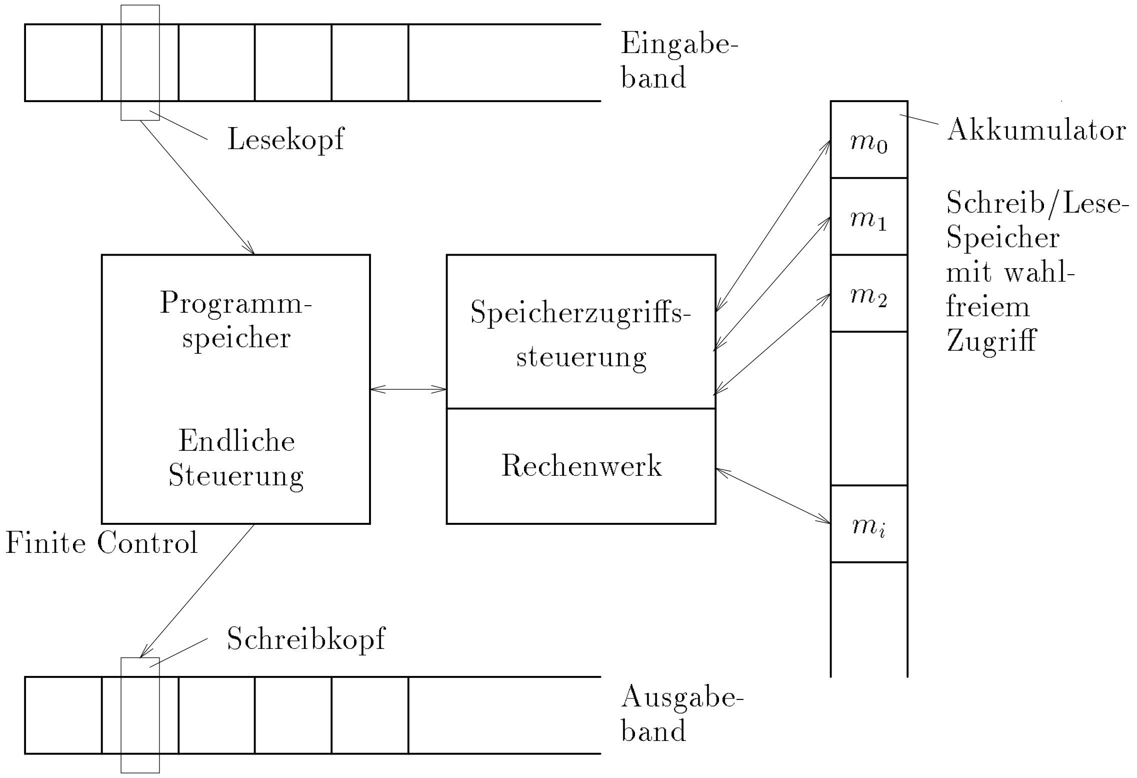
Beispiel:

Addition von 202 $\hat{=}$ 3 1 -2 2 und 55 $\hat{=}$ 313.

202	0	3	1	-2	2
55	0	0	3	1	3
$a + b$	0	3	4	-1	5
s	0	-1	0	-1	1
c	0	1	1	0	1
geschiftetes c	1	1	0	1	0
$s + c$ ($\hat{=}$257)	1	0	0	0	1

1.5 Die Registermaschine (RAM)

Schematische Darstellung einer RAM: Random Access Machine



Datentytpen:

1. Jede Speicherzelle m_i kann eine (beliebige) ganze Zahl darstellen (idealisiert!)
2. Jede Zelle des Eingabe- bzw. Ausgabebandes enthält ein Zeichen aus dem Zeichenvorrat $\{\ 0,\ 1,\ \not{b}\}$

Befehlsatz(kanonisch):

Es bezeichne b den Befehlszähler, m_0 den Akkumulator.

1. Eingabe/Ausgabe
   `LOAD i` : $m_0 \leftarrow m_i, \quad b \leftarrow b+1$
   `STORE i` : $m_i \leftarrow m_0, \quad b \leftarrow b+1$

2. Arithmetische Instruktionen

$$
\begin{array}{lll}
\texttt{ADD i} & : & m_0 \leftarrow m_0 + m_i, \quad b \leftarrow b+1 \\
\texttt{SUB i} & : & m_0 \leftarrow \begin{cases} m_0 - m_i & \text{falls } m_i < m_0 \\ 0 & \text{sonst} \end{cases}, \quad b \leftarrow b+1 \\
\texttt{MULT i} & : & m_0 \leftarrow m_0 \cdot m_i, \quad b \leftarrow b+1 \\
\texttt{DIV i} & : & m_0 \leftarrow \lfloor m_0 / m_i \rfloor, \quad b \leftarrow b+1
\end{array}
$$

3. Verzweigungsstrukturen

$$
\begin{array}{lll}
\texttt{GOTO i} & : & b \leftarrow i \\
\texttt{IF } m_o \odot l \texttt{ GOTO i} & : & b \leftarrow \begin{cases} i & \text{falls } m_0 \odot l \\ b+1 & \text{sonst} \end{cases} \\
\texttt{END} & : & b \leftarrow b
\end{array}
$$

4. Einbringen von Konstanten
   `CLOAD, CADD, CSUB, CMULT, CDIV`: Ersetze m_i durch die Konstante i.

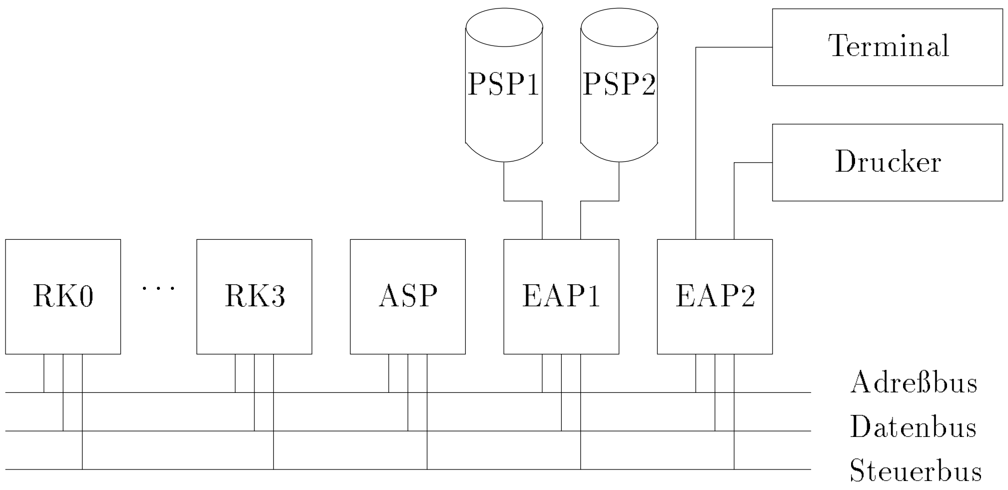
5. Indirekte Adressierung
   `INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT, INDDIV`: Ersetze m_i durch m_{m_i} den Inhalt des Registers m_i.

Die Mächtigkeit der RAM entspricht der Mächtigkeit Turing Maschine. Hierzu betrachte man:

Die Church'sche These:

Church postulierte schon 1936, daß die Klasse der turingberechenbaren Funktionen mit der Klasse aller im intuitiven Sinn berechenbaren Funktionen identisch sei. Diese Behauptung ist nicht beweisbar, da sich die Klasse aller im intuitiven Sinn berechenbaren Funktionen nicht formalisieren läßt.

1.6 Die Maschine MI



1.6.1 Rechnerkern

Der Rechnerkern der *MI* besteht aus dem Leitwerk und dem Rechenwerk.

Aufgaben des Leitwerks:

1. Befehl aus dem Speicher holen
2. Befehl dekodieren
3. falls nötig Operanden holen
4. Befehl ausführen bzw. Ausführung veranlassen
5. Ergebnis abspeichern
6. Befehlszähler fortschalten

Rechenwerk:

Es stehen 16 *allgemeine Register* $R_0 \ldots R_{15}$ zur Verfügung. Den Registern R_{14} und R_{15} kommt dabei eine spezielle Bedeutung zu. Das Register R_{14} wird als Kellerpegel SP(„stack-pointer") verwendet, das Register R_{15} als Befehlszähler $PC(BZ)$.

Spezielle Register

RKMAP	Modus-Register
P0B P1B SB	Anfangsadresse für Seitentafeln
P0L P1L SL	Länge der Seitentafeln

Modi:

Der Rechnerkern der *MI* kennt zwei verschiedene Arbeitszustände, den *Systemkernmodus* (kernel mode) und den *Benutzermodus* (user mode).

Adressierungsarten:

- direkte oder reale Adressierung
- virtuelle oder Seitenadressierung

1.6.2 Arbeitsspeicher (ASP)

- byte-weise adressiert

- Abspeicherung der Datentypen:

B	Byte	=	8 Bit
H	Halbwort	=	16 Bit
W	Wort	=	32 Bit
F	kurze GPZ(float)	=	32 Bit
D	lange GPZ(double)	=	64 Bit

  - bei *Intel*: niederwertigstes Byte zuerst (big-endian)
  - bei *Mi*: höchstwertige Byte zuerst (little-endian)

- Maschinenadressen:

$$
\begin{array}{l}
\left.\begin{array}{l}
\left.\begin{array}{l}\texttt{0000 0000}\\ \quad\vdots \\ \texttt{000F FFFF}\end{array}\right\}\text{1MB}\\
\quad\vdots \\
\texttt{1FFF FFFF}
\end{array}\right\}\text{512MB (max.)}\\
\left.\begin{array}{l}\texttt{2000 0000}\\ \quad\vdots \\ \texttt{7FFF FFFF}\end{array}\right\}\text{E/A-Register}\\
\left.\begin{array}{l}\texttt{8000 0000}\\ \quad\vdots \\ \texttt{FFFF FFFF}\end{array}\right\}\text{unzulässig}
\end{array}
$$

1.6.3 Bus

Die Kommunikation der verschiedenen Komponenten verläuft über den Bus. Die dabei verwendeten Mechanismen soll folgendes Beispiel verdeutlichen.

Es sollen Daten vom ASP über den Bus zum RK übertragen werden:

1. Der RK legt die ASP-Adresse des gewünschten Datums auf den Adreßbus.
2. Der RK legt das Signal „Lesen“ auf eine bestimmte Leitung im Steuerbus.
3. Der RK legt das Signal „Adresse gültig“ ebenfalls auf eine bestimmte Leitung im Steuerbus. Damit wird erreicht, daß die Adresse erst gelesen wird, wenn alle Signale stehen.
4. Der ASP ist passiv, „hört“ aber auf Adreß- und Steuerbus mit.
5. Der ASP erkennt daß eine gültige Adresse am Adressbus anliegt und prüft, ob diese in seinem Adressraum liegt. Falls nein, erfolgt von Seiten des ASP keine Aktion, andernfalls werden folgende Schritte ausgeführt.
6. Der ASP prüft das Transportrichtungssignal im Steuerbus und erkennt, daß es sich hier um einen Lesezugriff handelt.

7. Der ASP legt das adressierte Datum auf den Datenbus.
8. Der ASP legt das Signal „Daten bereit“ auf eine bestimmte Leitung im Steuerbus
9. Der RK hat bis jetzt auf das Signal „Daten bereit“ gewartet und übernimmt nun das Datum vom Adressbus
10. Der RK legt das Signal „Daten übernommen“ auf eine bestimmte Leitung im Steuerbus
11. Der ASP deaktiviert daraufhin den Datenbus und das Signal „Daten bereit“
12. Sobald das Signal „Daten bereit“ vom ASP weggenommen wird, deaktiviert der RK die von ihm verwendeten Adreß- und Steuerleitungen. Der Bus ist wieder im Grundzustand.

Definition 1.6.1 *Ein solcher Ablauf aus Aktions- und Kommunikationsschritten zwischen Kommuniktionspartnern heißt Protokoll. Das Protokoll definiert auch Syntax und Semantik der auszutauschenden Information.*

Bemerkungen

- Der oben geschilderte Ablauf heißt Busprotokoll.
- Das obige Busprotokoll macht keine Annahmen über die Geschwindigkeit der Arbeit bei den beteiligten Partnern (asynchrones Arbeiten ohne Zeitgrenze). Deshalb sind immer wieder Quittungen eines Partners erforderlich, in denen er mitteilt, daß er einen Schritt abgeschlossen hat.
- Protokolle spielen ersichtlich eine erhebliche Rolle für effiziente Kommunikation und Kooperation. Standardisierungsbedarf!
- Die Einteilung in IWe(Verteilung der zentralen Aufgabe des Rechners) und die vorgesehenen Kanäle sollen in einem zeitlich gut geplanten Ausgleich stehen (Architektur).

1.7 Bibliographie

[AHU74] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Mass., 1974

[Knu76] Knuth, D.E.: Big Omicron and Big Omega and Big Theta.
ACM Sigact News **8**, 2, p. 18

[Lue80] Lueker, G.S.: Some Techniques for Solving Recurrences.
ACM Computing Surveys **12**, 4 (1980), p. 419–436

[Liu68] Liu, C.L.: Introduction to Combinatorial Mathematics.
McGraw-Hill, New York, N.Y., 1968

[Mil60] Milne-Thomson, L.M.: The Calculus of Finite Differences.
Macmillan, London, 1960

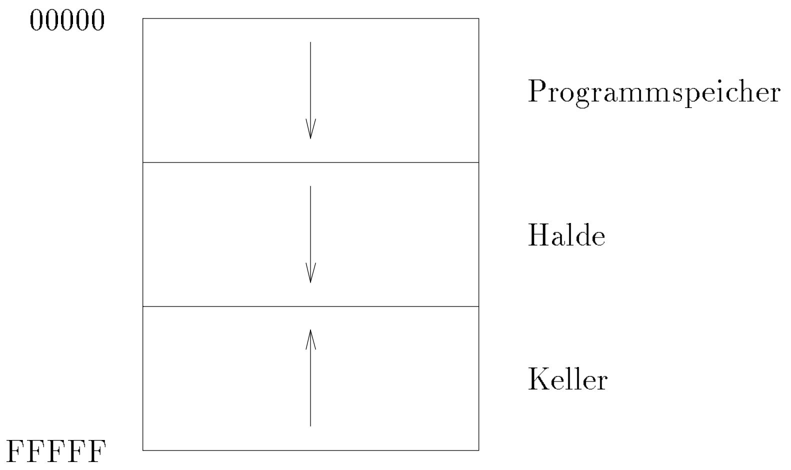
[Rei77] Reingold, E.M., Nievergelt, J., Deo, N.: Combinatorial Algorithms: Theory and Practice.
Prentice-Hall, Englewood Cliffs, N.J., 1977

[Rio58] Riordan, J.: An Introduction to Combinatorial Analysis.
J. Wiley, New York, N.Y., 1958

[Rio68] Riordan, J.: Combinatorial Identities.
J. Wiley, New York, N.Y., 1968

2. Architektur mit realer Adressierung

Das Speichermodell bei 1MB Arbeitsspeicher:



2.1 Adressierungsarten

Beispiel:

`ADD W` α_1, α_2, α_3

„addiere die beiden ganzen Zahlen, die in den jeweils 4 Bytes beginnend bei der durch α_1 bzw. α_2 gegebenen Adressen gespeichert sind, und speichere das Resultat in den 4 Bytes beginnend bei der durch α_3 gegebenen Adresse.“

Allgemein:

$$\begin{aligned}
\langle\textbf{Maschinenbefehl}\rangle \quad &::= \quad \langle\text{Operationsteil}\rangle \,| \\
& \quad \langle\text{Operationsteil}\rangle_\sqcup \\
& \quad \{\text{Operationsspezifikator}\}^{1,4} \\
\langle\textbf{Operationsteil}\rangle \quad &::= \quad \langle\text{Transportbefehl}\rangle \,|\ldots
\end{aligned}$$

Operationsteil:

- Opcode
- Kennung (LOP, Länge des Operanden, B, H, W, F, D)
- Anzahl der Operanden (nicht im Assemblersource; wird vom Assembler hinzugefügt)

Operandenspezifikator:

Im folgenden soll folgende Notation verwendet werden.

α_i	i-te Operandenadresse
$S[\alpha_i]$	Inhalt der Adresse α_i
α_3 sonst α_2	α_3 falls definiert, sonst α_2

⟨**Operandenspezifikator**⟩	::=		
		⟨absolute Adresse⟩|	
		⟨direkter Operand⟩|	„immediate"
		⟨Registeradressierung⟩|	
		⟨indirekte Adressierung⟩|	
		⟨indizierte relative Adressierung⟩|	
		⟨indizierte indirekte Adressierung⟩|	
		⟨Kelleradressierung⟩	

2.1.1 Absolute Adresse

⟨**absolute Adresse**⟩	::=	⟨ganze Zahl⟩|
		$\langle\text{Identifikator}\rangle\{\langle\text{VZ}\rangle\langle\text{Summand}\rangle\}^{0,\infty}$
⟨**ganze Zahl**⟩	::=	$\{\langle\text{VZ}\rangle\}^{0,1}\{\langle\text{Summand}\rangle\}^{1,\infty}_{\langle\text{VZ}\rangle}$

Beispiele:

1. `MOVE W 93, a-3`
   $\hat{=}$ S(a-3) wird s[93]

2. `MOVE B H'FF', 255`
   $\hat{=}$ S(255) wird s[255]

2.1.2 Direkter Operand

Ein direkter Operand (immediate operand) steht direkt im Befehl selbst, es wird daher keine Adresse für den Operanden definiert.

Notation:

$\mathbf{I}_\sqcup\langle\text{ganze Zahl}\rangle$

Beispiele:

1. `MOVE W I 0, 0`
2. `MOVE B I 'F', 0`
3. `MOVE B I H'F', 0`

2.1.3 Registeradressierung

Notation:

$$\begin{array}{lcl} \langle\textbf{Registeradressierung}\rangle & ::= & \langle R_x\rangle \\ \langle R_x\rangle & ::= & \mathrm{R0|R1|\ldots|R15|SP|PC} \end{array}$$

Beispiele:

1. `MOVE W R1, R0`
2. `MOVE B R1, R0` (nur das niederwertigste Byte wird kopiert!)

2.1.4 Relative Adressierung

Notation:

$\langle\textbf{relative Adressierung}\rangle ::= \{\langle\text{ganze Zahl}\rangle+\}^{0,1}!\langle R_x\rangle$

Die Adresse ist der *Inhalt* des Registers R_x, + ganze Zahl als Offset.

Beispiele:

1. `MOVE W R0, -4 + !SP`
2. `MOVE H R1, -6 + !SP`

2.1.5 Indirekte Adressierung

Notation:

$\langle\textbf{indirekte Adressierung}\rangle ::= !\langle\text{relative Adressierung}\rangle|!!\langle R_x\rangle$

Die Adresse ist der Inhalt des durch die innere Relativadressierung bestimmten Wortes.

Beispiel:

1. `MOVE W R0, !(-4+!SP)`

2.1.6 Indizierte relative Adressierung

Notation:

⟨**indiziert relative Adressierung**⟩	::=	⟨relative Adressierung⟩⟨Indexangabe⟩
⟨**Indexangabe**⟩	::=	$/\langle R_y \rangle$

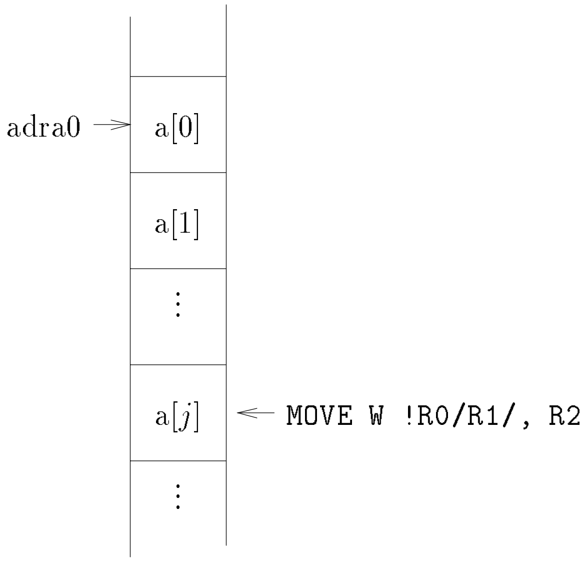
Die Adresse ist die durch die relative Adressierung bestimmte Adresse + (Inhalt von R_y)·LOP. LOP bezeichnet dabei die Länge des Operanden. Es gilt:

Kennung Operand	LOP
B	1
H	2
W	4
F	4
D	8

Beispiel:

Ab der Adresse adra0 ist eine Reihung mit den Elementen a[0], a[1], ...,a[n] in aufeinanderfolgenden Speicherzellen abgelegt. Die Elemente der Reihung sind Worte. In R0 wird die Adresse adra0 abgelegt. Um nun das j-te Element der Reihung ansprechen zu können wird der Index j in Register R1 abgelegt. Der Zugriff erfolgt dann über den Befehl:

```
MOVE W !R0/R1/,R2
```



2.1.7 Indizierte indirekte Adressierung

Notation:

⟨**Indizierte indirekte Adressierung**⟩ ::= ⟨indirekte Adressierung⟩ ⟨Indexangabe⟩

2.1.8 Kelleradressierung

Wie aus dem Namen schon ersichtlich dient diese Adressierungsart vorwiegend dem Zugriff auf kellerartig organisierte Datenstrukturen.

Notation:

⟨**Kelleradressierung**⟩ ::= -!$\langle R_x \rangle$|!$\langle R_x \rangle$+

-!$\langle R_x \rangle$	$\hat{=}$	push
!$\langle R_x \rangle$+	$\hat{=}$	pop

2.2 Maschinenbefehle

2.2.1 Befehlszyklus

Der Rechnerkern der *MI* führt permanent ein Mikroprogramm, in Form einer Endloschleife aus. Dieses bezeichnen wir als Befehlszyklus. Bei jedem Schleifendurchlauf wird ein Maschinenbefehl der *MI* ausgeführt. Die Berechnung der Operandenadresse erfolgt im

Rechnerkern. In dem nachfolgend dargestellen Befehlszyklus fehlen noch einige Details, wie z.B. Unterbrechungen.

```
{
Initialisierung; z.B. PC, RK
while (true) {
   hole Operationsteil
   PC++
   i = Zahl der Operanden
   switch(Operationsteil) {
      ⋮
      q: {
         for (j=1;j≤i;j++) {
            hole 1.Byte des j-ten Operanden
            PC++
            hole nächstes Byte des j-ten Operanden, falls es existiert
            falls Byte vorhanden war: PC++
            ⋮
            letztes Byte des j-ten Operanden holen
            PC++
            werte Adressierung für j-ten Operanden aus
            }/*for*/
         führe Befehl q aus
         }/*switch q*/
      ⋮
      }/*switch*/
   }/*while*/
}
```

Bemerkungen

- Variabel lange Befehle ergeben eine flexible Programmierung.
- Platzeinsparung durch die Verwendung von Bytes als Grundeinheit (wenig Verschnitt).
- Nachteil: Die Mikroprogramme werden aufwendig (vgl. schematischen Befehlszyklus). Die Folgen sind Performanceverluste und eine Verteuerung des Chip-Designs.

2.2.2 Transportbefehle

$$\langle \textbf{Transportbefehl} \rangle ::= \begin{array}{l} \texttt{MOVE}_\sqcup\langle \text{bhwfd}\rangle \\ \mid \texttt{CLEAR}_\sqcup\langle \text{bhwfd}\rangle \\ \mid \texttt{MOVEN}_\sqcup\langle \text{bhwfd}\rangle \\ \mid \texttt{MOVEC}_\sqcup\langle \text{bhwfd}\rangle \\ \mid \texttt{MOVEA} \\ \mid \texttt{CONV} \\ \mid \texttt{PUSHR} \\ \mid \texttt{POPR} \end{array}$$

$\langle \textbf{bhwfd} \rangle ::= \texttt{B} \mid \texttt{H} \mid \texttt{W} \mid \texttt{F} \mid \texttt{D}$

2.2.3 Logische Operationen

$$\langle \textbf{bitweise boolsche Operation} \rangle ::= \begin{array}{l} \texttt{OR}_\sqcup\langle \text{bhw}\rangle \\ \mid \texttt{ANDNOT}_\sqcup\langle \text{bhw}\rangle \\ \mid \texttt{XOR}_\sqcup\langle \text{bhw}\rangle \end{array}$$

$\langle \textbf{bhw} \rangle ::= \texttt{B} \mid \texttt{H} \mid \texttt{W}$

2.2.4 Arithmetische Operationen

$\texttt{ADD}_\sqcup\{\texttt{B} \mid \texttt{H} \mid \texttt{W} \mid \texttt{F} \mid \texttt{D}\}\ _\sqcup\{\alpha_1, \alpha_2 | \alpha_1, \alpha_2\alpha_3\}$
Wirkung: $S[\alpha_2|\alpha_3] := S[\alpha_2] + S[\alpha_1]$
$\texttt{SUB}_\sqcup\{\texttt{B} \mid \texttt{H} \mid \texttt{W} \mid \texttt{F} \mid \texttt{D}\}\ _\sqcup\{\alpha_1, \alpha_2 | \alpha_1, \alpha_2\alpha_3\}$
Wirkung: $S[\alpha_2|\alpha_3] := S[\alpha_2] - S[\alpha_1]$
$\texttt{MULT}_\sqcup\{\texttt{B} \mid \texttt{H} \mid \texttt{W} \mid \texttt{F} \mid \texttt{D}\}\ _\sqcup\{\alpha_1, \alpha_2 | \alpha_1, \alpha_2\alpha_3\}$
Wirkung: $S[\alpha_2|\alpha_3] := S[\alpha_2] \cdot S[\alpha_1]$
$\texttt{DIV}_\sqcup\{\texttt{B} \mid \texttt{H} \mid \texttt{W} \mid \texttt{F} \mid \texttt{D}\}\ _\sqcup\{\alpha_1, \alpha_2 | \alpha_1, \alpha_2\alpha_3\}$
Wirkung: $S[\alpha_2|\alpha_3] := S[\alpha_2] / S[\alpha_1]$

2.2.5 Sprungbefehle

Unbedingter Sprung

$\texttt{JUMP}_\sqcup\alpha_1$
Wirkung: PC $:= \alpha_1$

Bedingter Sprung

Im Statusregister des Rechnerkerns befinden sich die Bedingungscodes:

RES_C	Übertrag (carry)
RES_V	Überlauf (overflow)
RES_Z	Ergebnis Null (zero)
RES_N	Ergebnis negativ (negativ)

Diese werden durch arithmetische und logische Operationen, sowie bei Transportoperationen entsprechend den Ergebnissen gesetzt. Die Flags RES_Z und RES_N können ebenfalls durch den Vergleichsbefehl gesetzt werden.

`CMP␣{B | H | W | F | D}␣`α_1, α_2
Wirkung: $RES_Z := (S[\alpha_1] = S[\alpha_2])$
$RES_N := (S[\alpha_1] \leq S[\alpha_2])$

Aus der Kombination verschiedener Bedingungscodes ergeben sich folgende bedingte Sprungbefehle:

Befehl		Bedingung
`JEQ`	Sprung, falls Ergebnis Null	RES_Z
`JNE`	Sprung, falls Ergebnis nicht Null	$\neg RES_Z$
`JGT`	Sprung, falls Ergebnis größer Null	$\neg RES_Z \wedge \neg RES_N$
`JGE`	Sprung, falls Ergebnis größer gleich Null	$\neg RES_N$
`JLT`	Sprung, falls Ergebnis kleiner Null	RES_N
`JLE`	Sprung, falls Ergebnis kleiner gleich Null	$RES_N \vee RES_Z$
`JC`	Sprung, falls Übertrag	$RES_C = 1$
`JNC`	Sprung, falls kein Übertrag	$RES_C = 0$
`JV`	Sprung, falls Überlauf	$RES_V = 1$
`JNV`	Sprung, falls kein Überlauf	$RES_V = 0$

2.2.6 Schiebebefehle

Es stehen zwei Schiebebefehle zur Verfügung, die jeweils auf Wortoperanden angewendet werden. Es sind dies der arithmetische Befehl `SH` (shift), der überlaufsensitiv arbeitet sowie der logische Befehl `ROT` (rotate), der die Bits eines Wortes zyklisch vertauscht.

`SH␣`$\alpha_1, \alpha_2, \alpha_3$
Wirkung:

- falls $S[\alpha_1] \geq 0$: $S[\alpha_3] := S[\alpha_2]$ um $S[\alpha_1]$ Bits nach links geschoben und mit Nullen aufgefüllt (Overflow möglich)
- falls $S[\alpha_1] < 0$: $S[\alpha_3] := S[\alpha_2]$ um $S[\alpha_1]$ Bits nach rechts geschoben und mit dem ursprünglich höchstwertigen Bit aufgefüllt (Vorzeichen bleibt dadurch erhalten)

`ROT␣`$\alpha_1, \alpha_2, \alpha_3$
Wirkung:

- falls $S[\alpha_1] \geq 0$: $S[\alpha_3] := S[\alpha_2]$ um $S[\alpha_1]$ Bits zyklisch nach links geschoben

- falls $S[\alpha_1] < 0$: $S[\alpha_3] := S[\alpha_2]$ um $S[\alpha_1]$ Bits zyklisch nach rechts geschoben

2.2.7 Bitfeldbefehle

Mit Hilfe der Bitfeldbefehle lassen sich aus Registern, oder einem fortlaufenden Speicherbereich von maximal Wortlänge bestimmte Teilstücke (Bitfelder) setzen oder extrahieren. Ein Bitfeld wird definiert durch die Angabe einer Basisadresse A, eines möglicherweise auch negativen Displacements P, sowie einer Länge S im Bereich $0 \leq S \leq 32$. Bei Registeradressierung muß das Bitfeld in zwei aufeinanderfolgenden Registern enthalten sein.

`EXTS␣`$\alpha_1, \alpha_2, \alpha_3, \alpha_4$	$P := S[\alpha_1]$ (extract bitfield signed) $S = S[\alpha_2]$ $A := \alpha_3$ $S[\alpha_4] :=$ auf 32 Bit vorzeichenerweitertes Wort (integer) gemäß dem bezeichneten Bitfeld
`EXT␣`$\alpha_1, \alpha_2, \alpha_3, \alpha_4$	$P := S[\alpha_1]$ (extract bitfield) $S = S[\alpha_2]$ $A := \alpha_3$ $S[\alpha_4] :=$ mit Nullen auf 32 Bit erweitertes Wort (integer) gemäß dem bezeichneten Bitfeld
`INS␣`$\alpha_1, \alpha_2, \alpha_3, \alpha_4$	$P := S[\alpha_2]$ (insert bitfield) $S = S[\alpha_3]$ $A := \alpha_4$ bezeichnetes Bitfeld $:= S[\alpha_1]$
`FINDC␣`$\alpha_1, \alpha_2, \alpha_3, \alpha_4$	$P := S[\alpha_1]$ (find first bit cleared/set) $S = S[\alpha_2]$ $A := \alpha_3$ $S[\alpha_4] :=$ Nummer des ersten gefundenen Bits bzw. $P + S$

2.3 Speicherverwaltung

Da auch *MI* -Programme modular aufgebaut werden sollen, müssen Konventionen vereinbart werden, nach denen die Module oder Prozeduren, aus denen ein *MI* -Programm aufgebaut ist, miteinander kommunizieren sollen. Des weiteren muß noch eine Übereinkunft darüber getroffen werden, wie die gängigen Datenstrukturen, wie z.B. Felder oder Strukturen, auf den linearen Programmadreßraum abgebildet werden sollen.

2.3.1 Beispiele

int i;

int a[20];
struct listel {
  char index,
  listel *left,
  listel *right } s;

(α) i: Adresse = Speicher
```
MOVE W R0,i
ADD W I1,i
```

(β) Anfangsadresse von a in R0,
```
MOVEA i,R1
MOVE W !R0/R1/,R2
```

(γ)
```
MOVEA s,R1
MOVE W 5+!R1,R2
```

2.3.2 Konventionen bei Prozeduraufruf

Der Aufruf von Unterprogrammen, sprich Prozeduren und Funktionen, in *MI*-Assembler erfolgt mittels dem Befehl:

`CALL`$_\sqcup\alpha_1$

Der `CALL`-Befehl legt die Rücksprungadresse auf dem Stack ab und verzweigt danach an die Programmadresse α_1. Die Rückkehr aus einem Unterprogramm erfolgt mit Befehl `RET`.

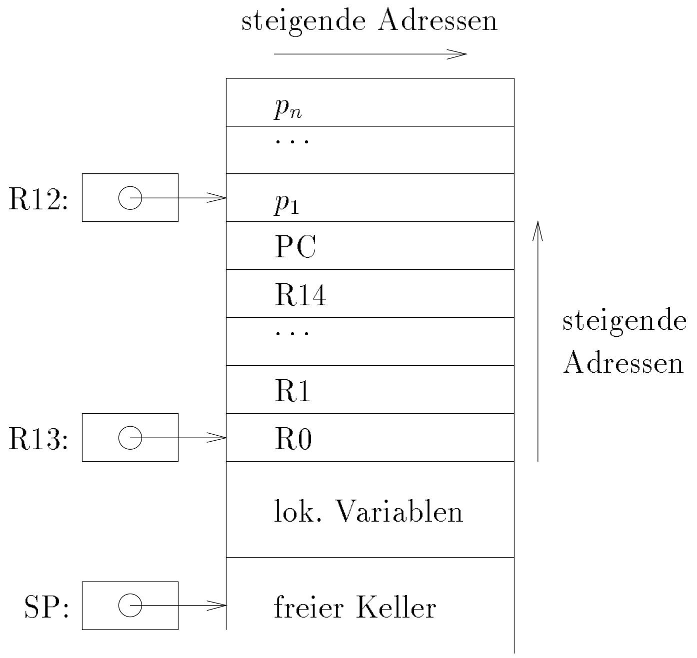
Da der `CALL`-Befehl lediglich die Sicherung der Rücksprungadresse und die Verzweigung übernimmt, sind vom Benutzer weitere Konventionen zu vereinbaren. Diese Konventionen regeln die Einrichtung des lokalen Datenraums des Unterprogramms, sowie die Rettung des Registersatzes des Rechnerkerns und die Ablage der Parameter des Unterprogramms.

Für die Vorlesung treffen wir folgende Konventionen:

- Die Paramter des Unterprogramms stehen im Keller.
- Bei Funktionen wird der Rückgabewert als fiktiver Parameter $p_{(n+1)}$ ebenfalls im Keller abgelegt.
- Die Parameter werden in umgekehrter Reihenfolge in den Keller geschrieben. Unmittelbar vor dem Unterprogrammaufruf zeigt der Kellerpegel also auf den ersten Parameter.
- Unterprogramme, welche Register des Rechnerkerns verwenden, müssen diese bei Eintritt in das Unterprogramm sichern. Dabei ist grundsätzlich der gesamte Registersatz zu retten.

- Lokale Variablen werden im Keller abgelegt. Den Bereich der lokalen Variablen im Keller bezeichnen wir als lokalen Datenraum.
- Die Adresse des letzten Bytes vor dem lokalen Datenraum bezeichnen wir als Basisadresse des lokalen Datenraumes.
- Die Register R12 und R13 stehen nicht zur freien Verfügung und haben folgende Bedeutung:
  - R12 zeigt im Keller auf den Bereich der Parameter. Es enthält folglich die Ablageadresse des Parameters p_1.
  - R13 enthält die Basisadresse des lokalen Datenraumes.

Damit ergibt sich beim Unterprogrammaufruf folgende Struktur im Keller:



Mit den getroffenen Konventionen lassen sich folgende Gerüste zum Unterprogrammaufruf angeben:

`Aufrufer`:

...
push letzten Parameter p_n bzw. $p_{(n+1)}$;
...
push ersten Parameter p_1;
`CALL proz`
pop alle Parameter, bzw. brichtige den Kellerpegel;
...

```
proz:
  PUSHR                         – Registersatz retten
  MOVEA 64+!SP,R12              – Adresse des ersten Parameters nach R12
  MOVE W SP,R13                 – Basisadresse des lokalen Datenraumes nach R13
  lokalen Datenraum anlegen;
  ...
  MOVE W R13,SP                 – Basisadresse lokaler Datenraum nach SP
                                  falls in der Prozedur der Keller nicht
                                  völlig abgebaut wurde
  POPR                          – Registersatz restaurieren
  RET                           – Rücksprung
```

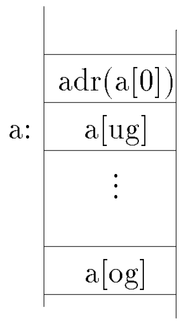
2.3.3 Adressierung eindimensionaler Felder

Sei a ein eindimensionales Feld der Form a: **array** $[ug..og]$ **of integer**, so berechnet sich die Adresse des i-ten Elements nach der Abbildungsvorschrift:

$$\begin{aligned}\mathrm{adr}(a[i]) &= \mathrm{adr}(a[ug]) + (i - ug) \cdot \mathrm{size(integer)} \\ &= \underbrace{(\mathrm{adr}(a[ug] - ug \cdot \mathrm{size(integer)})}_{\mathrm{adr}(a[0])} + i \cdot \mathrm{size(integer)}\end{aligned}$$

Die entsprechende Umsetzung in *MI*-Assembler verwendet die indiziert indirekte Adressierung und lautet wie folgt:

```
MOVEA a,R1
MOVE W i,R2
MOVE W !(-4+!R1)/R2/,R0
```



Bemerkungen

In obiger Variante des Feldzugriffes wird nicht überprüft, ob $i \in \{ug \dots og\}$. Zugriffe mit einem Index $i \notin \{ug \dots og\}$ führen folglich zu einem undefinierten Ergebnis anstelle eines Fehlers.

2.3.4 Adressierung mehrdimensionaler Felder

Mehrdimensionale Felder der Form a: **array** $[ug_1..og_1, ug_2..og_2, \ldots, ug_r..og_r]$ **of** foo; $lfoo :=$ size(foo) wird wie folgt adressiert:

row-major: $s_j := og_j - ug_j + 1$

$$\begin{aligned}
\mathrm{adr}(a[i_1, i_2, \ldots, i_r]) &= \mathrm{adr}(a[ug_1, \ldots, ug_r]) + \\
&\quad (\ldots((i_1 - ug_1) \cdot s_2 + (i_2 - ug_2)) \cdot s_3 + \ldots + i_r - ug_r) \cdot lfoo \\
&= \mathrm{adr}(a[0, \ldots, 0]) + ((\ldots(i_1 s_2 + i_2) \cdot s_3 + \ldots) + i_r) \cdot lfoo
\end{aligned}$$

mit

$$a[0, \ldots, 0] := a[ug_1, \ldots, ug_r] - (\ldots(ug_1 \cdot s_2 + ug_2)s_3 + \ldots + ug_r) \cdot lfoo$$

Bedingungen an die Indizes

$$ug_j \leq i_j \leq og_j \quad \text{oder} \quad \begin{array}{lcl} i_j - ug_j & \geq & 0 \\ i_j - ug_j & \leq & s_j \end{array}$$

Der Felddeskriptor

Um den Feldzugriff robust, einheitlich und effizient in Assembler formulieren zu können wird folgender Felddeskriptor eingeführt:

r
ug_1
s_1
ug_2
s_2
$\vdots$
ug_r
s_r
$lfoo$
raa

steigende Adressen

Die Elemente des Deskriptors sind Wörter und haben folgende Bedeutung:

r	Zahl der Indexpositionen
ug_p	untere Grenze für den Index p
s_p	Spanne: $s_p := og_p - ug_p + 1$
$lfoo$	size(foo)
raa	1. Anfangsadresse des Feldes a; adr(a[$ug_1, \ldots, ug_r$])
	2. Relativadresse bzgl. der Adresse des Deskriptors
	(3. Relativadresse bzgl. des lokalen Datenraumes)

function adr(adeskr, ifeld: adresse): adresse
adeskr: Adresse des Felddeskriptors
ifeld: Adresse von $[i_1, i_2, \ldots, i_r]$

```
adr:
    PUSHR
    MOVEA 64+!SP,R12
    MOVE W SP,R13
    CLEAR W R0
    MOVE W !R12,R1               -- R1 := adeskr
    MOVE W 4+!R12,R2             -- R2 := ifeld
    MOVE W !R1+,R3               -- R3 := r
    JUMP einstieg
loop:
    MULT W 4+!R1,R0
einstieg:
    MOVE W !R2+,R4               -- R4 := i_j
    SUB W !R1+,R4                -- i_j − ug_j
    JLT fehler
    CMP W R4,!R1+                -- i_j − ug_j ≤ s_j
    JGE fehler
    ADD W R4,R0
    SUB W I1,R3                  -- r := r − 1
    JGT loop
fertig:
    MULT W !R1+,R0               -- ·lfoo
    ADD W !R1+,R0                -- +raa
    ADD W !R12,R0                -- +adeskr
    MOVE W R0,8+!R12
    MOVE W R13,SP
    POPR
    RET
```

2.3.5 Adressierung im lokalen Datenraum

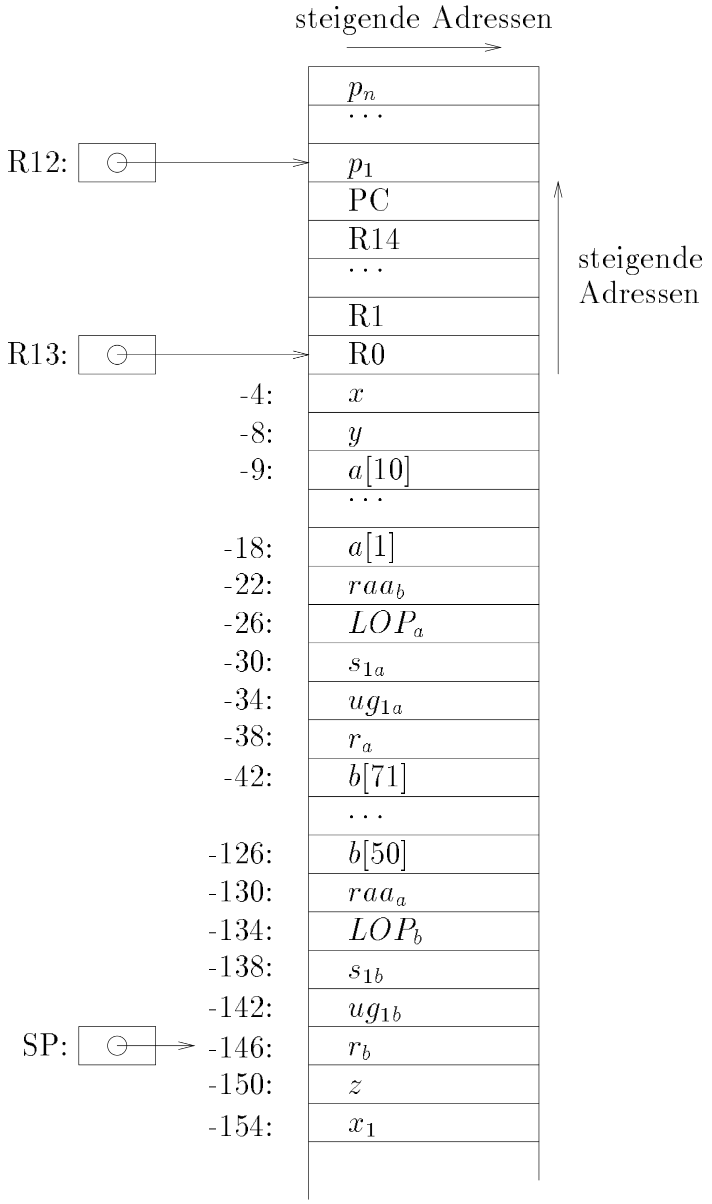
Folgendes Beispiel soll die Adressierung der lokalen Variablen eines Unterprogramms, gemäß der getroffenen Konventionen, verdeutlichen.

Gegeben sei die Prozedur:

procedure proz(...);
var x, y: integer
a: array[1..10] of char;
b: array[50..71] of integer;
z, x1: integer;
begin
...
$a[x] := f((b[x1])^2 + y + z)$;
...
end

Die Funktion f realisiere dabei die Abbildung von integer $\rightarrow$ char. Der lokale Datenraum für die Prozedur wird im Einklang mit den getroffenen Konventionen im Keller mit absteigenden Adressen aufgebaut. Zu beachten ist, daß aufgrund der Invarianz der Speicherabbildung, entgegen der Laufrichtung des Speicherraumes, Felder aufsteigend angeordnet werden. Das Feldelement $a[ug_1, \ldots, ug_r]$ besitzt folglich die niedrigste Speicheradresse.

Für unsere Beispielprozedur ergibt sich daraus folgender Kelleraufbau:




```
proz:
     PUSHR                        -- Sichern der Register
     MOVEA 64+!SP,R12             -- Adresse erster Parameter nach R12
     MOVE W SP,R13                -- Basisadresse lokaler Datenraum nach R13
     lokalen Datenraum ablegen;
     ...
     CLEAR W -!SP                 -- push Ergebnisparameter p3 = adel
     MOVEA -4+!R13,-!SP           -- push Index x
     MOVEA -38+!R13,-!SP          -- push Felddeskriptor a
     CALL AdresseFeldelement
     ADD W I 8,SP                 -- pop,pop jetzt a[x] oben im Keller
     CLEAR W -!SP                 -- push Ergebnisparameter p3 = adel
```

```
MOVEA -154+!R13,-!SP        -- push Index x1
MOVEA -146+!R13,-!SP        -- push Felddeskriptor b
CALL AdresseFeldelement
ADD W I 8,SP                -- pop,pop jetzt b[x1] oben im Keller
MULT W !!SP,!!SP,!SP        -- b[x1]² statt b[x1] im Keller
ADD W -8+!R13,!SP           -- +y
ADD W -150+!R13,!SP+,R0     -- +z, Ergebnis in R0,pop
MOVE B R0,!!SP              -- letztes Byte nach a[x]
ADD W I 4,SP                -- pop a[x] auch noch
...
MOVE W R13,SP               -- Basisadresse lokaler Datenraum nach SP
POPR                        -- Register restaurieren
RET
```

Wie aus dem Beispiel deutlich wird, verwendet man zur Adressierung lokaler Daten die relative Adressierung. Durch die von uns getroffenen Konventionen wird das Register R13 dabei immer als Basiszeiger für den lokalen Datenraum verwendet. Die einzelne Variable wird dann über ihren Offset zum Basiszeiger adressiert.

2.3.6 Parameterübergabe

(α) Allgemeines

- Variable Anzahl von Parametern
- Informationen (z.B. Typ) über einzelne Parameter

(β) Arten der Parameterübergabe

- call by value
- call by reference
- call by name

call by value

call by value bedeutet, daß der Wert des Parameters übergeben wird. Es wird folglich eine lokale Kopie des Parameters erzeugt, auf welcher das Unterprogramm arbeitet. Veränderungen des Parameters im Unterprogramm haben dadurch keinen Einfluß auf den ursprünglichen aktuellen Parameter im Aufrufer.

Es sind zwei Arten der Realisierung üblich:

1. Die Kopie erfolgt durch das aufgerufene Unterprogramm. Im Versorgungsblock wird die Adresse des aktuellen Parameters übergeben. Die Kopie wird im lokalen Datenraum des Unterprogramms angelegt. Nachteilig an dieser Variante ist der Umstand, daß auf den Parameterpositionen keine Ausdrücke stehen dürfen.

2. Die Kopie erfolgt durch den Aufrufer. Im Versorgungsblock wird der Wert des Parameters übergeben. Das aufgerufene Unterprogramm arbeitet auf der Kopie im Versorgungsblock. Hier können Ausdrücke auf Parameterpositionen benutzt werden. Diese Variante entspricht den für die Vorlesung getroffenen Konventionen zum Unterprogrammaufruf.

call by reference

Hier wird eine Referenz auf den Parameter übergeben. Praktisch bedeutet es, daß nicht eine Kopie des Parameters, sondern dessen Adresse, bei Feldern die Adresse des Felddeskriptors, an das Unterprogramm übergeben wird. Das Unterprogramm arbeitet also auf dem „Original“, folglich verändern Zuweisungen an den Parameter dessen Wert über das Unterprogramm hinaus. Variablenparameter werden deshalb als Ergebnisparameter verwendet. Bei Variablenparametern darf der aktuelle Parameter kein Ausdruck sein.

Da bei Variablenparametern nur eine Adresse übergeben wird und keine Kopie angelegt wird, verwendet man sie aus Gründen der Effizienz zur Übergabe von Feldern oder Strukturen, selbst falls diese nicht vom Unterprogramm verändert werden sollen. Dieser Programmierweise trägt z.B. C++ Rechnung und stellt dem Programmierer die Möglichkeit zur Verfügung Referenzparameter als Konstanten zu erklären, so wird verhindert, daß ungewollt Parameter verändert werden.

call by name

Da *call by name* sehr komplex ist, wird es von modernen Programmiersprachen nicht mehr unterstützt.

Die Zielsetzung bei dieser Art der Parameterübergabe ist, daß die Wirkung des Prozedurablaufs so ist, als wenn bei jedem Auftreten eines formalen Parameters in der der Quelle dieser dynamisch textuell durch den aktuellen Parameter ersetzt würde.

Zur Realisierung muß der Wert des aktuellen Ausdrucks neu berechnet werden, falls der zugehörige formale Parameter auf der rechten Seite auftritt. Tritt der Parameter auf der linken Seite einer Zuweisung auf, so muß die Adresse des aktuellen Parameters jedesmal neu berechnet werden.

Man übergibt somit für jeden Parameter die Adresse einer Parameterprozedur. Der Aufruf dieser Prozedur liefert:

- den Wert des Parameters (sofern definiert)
- die Adresse des Parameters, falls definiert (nicht definiert z.B bei Ausdrücken auf Parameterpositionen)
- Nähere Spezifikationen, z.B. Art des Parameters.

Beispiel:

```
int   k = 2;
proc  foo(a,b: int);
      a := 3;
      print(b,k);
int   f[5]=[0,10,20,30,40];

foo(k,f[k]); print(k);
```

call by value	call by reference	call by name
20,2	20,3	30,3
2	3	3

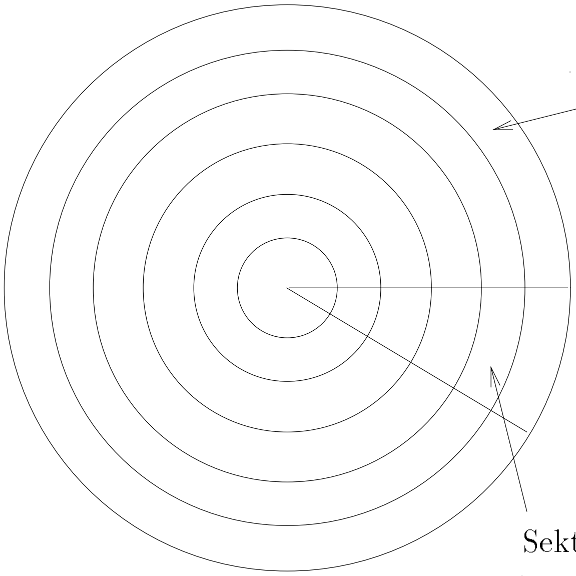
2.4 Ein-/Ausgabe

2.4.1 Der Plattenspeicher

Plattenspeicher werden über Ein/Ausgabe-Prozessoren(EAP) angeschlossen. Bei der MI sind zwei Plattenspeicherlaufwerke mit den Nummern 0 und 1 an einen EA-Prozessor (EAP1, Plattenspeichersteuerung) angeschlossen. Die Arbeitsweise der Plattenspeichersteuerung wird durch geeignete Besetzung spezieller Steuerregister geregelt. Die Adressen der Register liegen im Maschinenadreßraum im Adreßbereich von `H'2000 0000` bis `H'3FFF FFFF` (*Memory-Mapped-IO*).

Aufbau des Plattenspeichers

- Spuren (konzentrische Kreise); ihnen entspricht jeweils eine Stellung des Schreib-/-Lesekopfes
- 2 Speicherflächen
- Sektoren (Abschnitte auf den Spuren); diese entsprechen der kleinsten Transporteinheit.



Funktionen des Plattenspeichers

1. Schreibkopf positionieren
2. Daten schreiben
3. Daten lesen
4. Prüflesen
5. Sektorkopf lesen

Ein Sektor besteht aus einem Vorspann, dem Sektorkopf und den eigentlichen Daten. Im Falle der MI enthält der Sektorkopf Spur- und Sektornummer. Mittels des Sektorkopfes orientiert sich die Plattensteuerung über die aktuelle Position des Schreib-/Lesekopfes.

Die EA-Prozessoren arbeiten parallel zu den Rechnerkernen der MI und werden als gemeinsames Betriebsmittel von allen Prozessen genutzt. Es besteht folglich Synchronisationsbedarf. Deshalb werden folgende Konventionen eingeführt:

- Falls das Bit *Auftrag beendet* gesetzt ist, können die EAP-Register durch das Betriebssystem frei verändert werden.
- Nachdem alle Plattenspeichersteuerregister korrekt belegt sind, darf der Auftrag durch löschen des Bits *Auftrag beendet* gestartet werden.
- Nach Beendigung des Auftrags setzt der EAP das Bit *Auftrag beendet*.

Eine Möglichkeit das Ende eines Transportauftrages zu erkennen wäre es, zyklisch das Bit *Auftrag beendet* abzufragen. Dieses Verfahren nennt man *polling* bzw. *geschäftiges Warten*. Da die Ausführungszeit eines Transportauftrages an den Plattenspeicher jedoch sehr viel größer ist als die Ausführungszeit eines Befehls im Rechnerkern, bei der MI etwa um den Faktor 10^5, ist es wenig sinnvoll diese Rechenzeit in einer Warteschleife zu vergeuden. Insbesondere im Hinblick auf eine Multitaskingumgebung ist dieses Verfahren sehr unvorteilhaft. Deshalb besteht die Möglichkeit das Ende eines Plattenauftrages an die Auslösung einer Unterbrechung zu koppeln. Der Rechnerkern steht dann während des Plattenauftrags für andere Aufgaben zur Verfügung.

Die Register des Plattenspeichers

Name	Kennung	Adresse	Bezeichnung
STR	H	`H'203F F900`	Steuerregister
SARL	H	`H'203F F902`	Speicheradressregister, low
SARH	H	`H'203F F904`	Speicheradressregister, high
PAR	H	`H'203F F906`	Plattenspeicheradressregister
MZR	H	`H'203F F908`	Mehrzweckregister

Das Steuerregister (STR)

Aufbau des Steuerregisters:

Bits	Zugriff	Bedeutung
11-14	R	Fehlercode
9,10	RW	Nummer des RK, der unterbrochen werden soll falls $b_1$1 gesetzt
6,7	RW	Nummer des Laufwerks
5	RW	gesetzt, falls Auftrag beendet
4	RW	Unterbrechung gewünscht (falls gesetzt)
1-3	RW	Funktionen
0	R	EAP abrbeitsbereit (falls gesetzt)

Dabei bedeuten:

R	Lesezugriff zulässig (read)
W	Schreibzugriff zulässig (write)

Mögliche Fehlercodes:

Fehlernr.	Bedeutung
0	kein Fehler
1	Lesefehler bei den Daten
2	Lesefehler beim Sektorkopf
3	Sektorkopf nicht gefunden
4	Speicherzelle im ASP nicht vorhanden
5	unbekannte Funktion
6	falsche Laufwerksangabe
7	Schreib-/Lesekopf falsch positioniert
8	Daten passen nicht auf Spur, Sektornummer zu groß
9	falsche Distanz beim Positionieren des Schreib-/Lesekopfes

Vor Beginn einer Kommandoausführung löscht der EAP alle Fehlerbits.

Speicheradreßregister (SAR)

Das Speicheradreßregister besteht aus zwei Teilen: SARL und SARH. Es enthält, in High- und Low-Bytes aufgeteilt, die Speicheradresse (im Arbeitsspeicher), ab der sich der Pufferbereiches für die zu übertragenden Daten befindet.

Plattenspeicheradreßregister (PAR)

Das PAR hat, abhänig von der auszuführenden Funktion des Plattenspeichers, folgende Bedeutung:

- Beim Daten Schreiben, Lesen oder Prüflesen:

Bit		
7-15	Spurnummer	(0,...,511)
6	Oberfläche	(0,1)
0-5	Sektornummer	(0,...,39)

  Nach der Übertragung eines Sektors wird die Sektornummer inkrementiert.

- Beim Sektorkopf Lesen: PAR ohne Bedeutung.

- Beim Schreib-/Lesekopf positionieren:

Bit	
7-15	Anzahl der Spuren um die der Kopf bewegt werden soll $\lvert$alte Spur – neue Spur$\rvert$
0	Bewegungsrichtung: $0 \hat{=}$in Richtung größerer Spurnummern $1 \hat{=}$in Richtung kleinerer Spurnummern

Das Mehrzweckregister (MZR)

Das MZR hat, abhänig von der auszuführenden Funktion des Plattenspeichers, folgende Bedeutung:

- Beim Daten Schreiben, Lesen oder Prüflesen:

  Das MZR enthält $-n$, falls n Bytes noch zu übertragen sind. Nach jedem übertragenem Byte wird MZR inkrementiert. Im Fehlerfall kann so festgestellt werden, wieviele Bytes bereits übertragen worden sind.

  Bemerkung:
  Es kann maximal eine Spur gelesen werden, also 10 KB.

- Beim Sektorkopf Lesen:

Bit		
7-15	gelesene Spurnummer	(0,...,511)
6	ohne Bedeutung	
0-5	gelesene Sektornummer	(0,...,39)

- Beim Arm Positionieren ist das MZR ohne Bedeutung

2.4.2 Asynchrone Datenübertragung

Der EA-Prozessor 2 realisiert die EA-Steuerung für 4 asynchrone Datenübertragungskanäle. Jeder Datenübertragungskanal besteht aus einem Eingabekanal und einem Ausgabekanal. Beide sind, falls das angeschlossene Gerät dies erlaubt, gleichzeitig betreibbar. Der Ablauf des EA-Vorgangs wird wieder durch Register, die geeignet besetzt werden müssen, gesteuert.

Diese Register sind:

Name	Bedeutung
ESTR_i	Steuerregister für den Empfang von Zeichen über den Datenkanal i
EDR_i	Datenregister für den Empfang von Zeichen über den Datenkanal i
SSTR_i	Steuerregister für das Senden von Zeichen über den Datenkanal i
SDR_i	Datenregister für das Senden von Zeichen über den Datenkanal i

Die Adressen dieser 16-Bit-Register sind:

`H'203F FA00'`$+8 \cdot i$ für ESTR_i
`H'203F FA02'`$+8 \cdot i$ für EDR_i
`H'203F FA04'`$+8 \cdot i$ für SSTR_i
`H'203F FA06'`$+8 \cdot i$ für SDR_i

Das Empfangssteuerregister ESTR

Bits	Zugriff	Bedeutung
12	RW	Parität ist ungerade, falls gesetzt.
11	RW	Paritätsbit ist vorhanden, falls gesetzt.
		1. Bit wird als Paritätsbit interpretiert.
9,10	RW	Nummer des RK, der unterbrochen werden soll,
		falls Bit 6 gesetzt.
8	RW	Es werden 8-Bit Zeichen übertragen, falls Bit 8 gesetzt ist,
		sonst 7-Bit-Zeichen
		Falls Bit 11 = Bit 8 = 1, so überwiegt die Wirkung von Bit 11
7	R	In EDR ist ein Zeichen abholbereit
		Bit 7 wird gelöscht, falls EDR gelesen wird.
6	RW	Unterbrechung gewünscht, falls Zeichen in EDR bereit.
5	RW	Zwei Stoppschritte (falls Bit 5 = 0: 1 Stoppschritt)
1-4	RW	Schrittgeschwindigkeit:
		Hat Bit 1 ... Bit 4 den Wert n, so entspricht das
		einer Schrittgeschwindigkeit von $150 \cdot 2^n$ baud.
		mit $0 \leq n \leq 7$.
0	RW	Der Kanal ist empfangsbereit gesetzt.

Das Empfangsdatenregister EDR

Das Register kann nur gelesen werden. Es enthält rechtsbündig das gelesene Zeichen. Falls weniger als 8 Binärziffern gelesen werden, wird das Zeichen von links mit Nullen zum Byte aufgefüllt. Werden die Daten schneller übertragen als gelesen, so werden nicht gelesene Daten überschrieben und gehen verloren.

Das linke Byte enthält Fehlermeldungen:

Bits	Bedeutung
15	Bit 12 $\vee$ Bit 13 $\vee$ Bit 14
14	Mindestens ein Zeichen ging verloren,
	da EDR nicht genügend schnell gelesen wurde.
13	Beim Lesen des Zeichens folgte nach einem Zeichen kein Stopschritt.
12	Beim Lesen des Zeichens trat ein Paritätsfehler auf.

Die Fehlermeldungen werden unmittelbar nach dem Lesen von EDR gelöscht.

Das Sendestatusregister SSTR

Bits	Zugriff	Bedeutung
12	RW	Parität ist ungerade, falls gesetzt.
11	RW	Paritätsbit ist vorhanden, falls gesetzt.
		1. Bit wird als Paritätsbit interpretiert.
9,10	RW	Nummer des RK, der unterbrochen werden soll,
		falls Bit 6 gesetzt.
8	RW	Es werden 8-Bit Zeichen übertragen, falls Bit 8 gesetzt ist,
		sonst 7-Bit-Zeichen
		Falls Bit 11 = Bit 8 = 1, so überwiegt die Wirkung von Bit 11
7	R	Zeichen in SDR wurde übertragen
		Bit 7 wird gelöscht, falls SDR beschrieben wird.
6	RW	Unterbrechung gewünscht, falls Zeichen in SDR bereit.
5	RW	Zwei Stoppschritte (falls Bit 5 = 0: 1 Stoppschritt)
1-4	RW	Schrittgeschwindigkeit:
		Hat Bit 1 ... Bit 4 den Wert n, so entspricht das
		einer Schrittgeschwindigkeit von $150 \cdot 2^n$ baud.
		mit $0 \leq n \leq 7$.
0	RW	Der Kanal ist sendebereit gesetzt.

Das Sendedateregister SDR

Ist SSTR[b_8] = 1, so sind $b_8 \ldots b_{15}$ ohne Bedeutung, und $b_0 \ldots b_7$ enthalten das zu übertragende Zeichen.
Ist SSTR[b_7] = 0, so sind $b_7 \ldots b_{15}$ ohne Bedeutung, und $b_0 \ldots b_6$ enthalten das zu übertragende Zeichen.

3. Grundlagen des Betriebssystems

3.1 Einleitung

- Prozesse
- Modi des Rechnerkerns. Der Rechnerkern der MI besitzt zwei unterschiedliche Modi, den Systemmodus KM (kernel mode) und den Benutzermodus UM (user mode).
- Ein Prozeß hat drei mögliche Arbeitszustände:
  - rechnend: der Prozeß verfügt über einen RK
  - rechenbereit: der Prozeß könnte rechnen, hat aber keinen RK
  - wartend: der Prozeß kann nicht rechnen, da er auf ein externes Ergebnis wartet

3.2 Prozeßkontrollblock (PCB)

Der Rechnerkernzustand eines rechnenden Prozesses ist durch den realen Rechnerkernzustand festgehalten. Der vollständige Rechnerkernzustand eines rechenbereiten oder wartenden Prozesses ist im sog. Prozeßkontrollblock abgelegt. Ein PCB hat folgenden Aufbau:

+0	SP für Systemmodus	
+4	SP für Benutzermodus	
+8	R0	
	⋮	
+68	R15	
+72	PSW	Programmstatuswort
	P0B P0L P1B P1L	Informationen über Seitentabellen (wird später erklärt)

Während der Zuordnung des Rechnerkerns an einen Prozeß steht die Maschinenadresse des zugehörigen PCB in dem Sonderregister PCBADR des Rechnerkerns.

Da der Prozeßkontrollblock eine Datenstruktur des Betriebssytems ist, kann die Ablage nicht im Benutzeradreßraum erfolgen. Da weiterhin die Prozesse nicht kellerartig aktiviert und deaktiviert werden, sondern in beliebiger Reihenfolge, kann der PCB auch nicht im Systemkeller abgelegt werden. PCBs werden daher in einer festen Liste (der Prozeßliste) des Betriebssystems abgelegt. In der Prozeßliste steht für jeden Prozeß ein Prozeßelement, das folgenden Aufbau hat:

+0	PCB	
+92	PID	Eindeutige Kennzeichnung des Prozesses
+96	BKZ	Benutzerkennzeichen
+100	PRIO	Ablaufpriorität
+104	Arbeitszustand	
+108	RKNR	
+112	Ursache Wartezustand	
+116	Nachfolger Warteliste	
+120	Vorgänger Prozeßliste	
+124	Nachfolger Prozeßliste	

3.3 Programmstatuswort (PSW)

Im PSW sind weitere Informationen, die der Rechnerkern über den Prozeß weiß, zusammengefaßt.

Bit	Name	Bedeutung
b_{31}	TP	trace pending bit
b_{26}	BEN.MAP EN	Adressierungsart im Benutzermodus
		0: direkte Adressierung
		1: virtuelle Adressierung
b_{25},b_{24}	CM	gegenwärtiger Arbeitsmodus (current mode)
		00: Systemmodus
		11: Benutzermodus
b_{23},b_{22}	PM	vorhergehender Arbeitsmodus (previous mode)
		Belegung wie oben
b_{20},...,b_{16}	IPL	gegenwärtige Ablaufpriorität
		(interrupt priority level)
b_5	IV	integer overflow trap disable
		(keine Unterbrechung bei Overflow, falls gesetzt)
b_4	T	trace bit
b_3	N	RES_N
b_2	Z	RES_Z
b_1	V	RES_V Überlauf
b_0	C	RES_C Übertrag

$b_7 \ldots b_0$ sind direkt durch Befehle veränderbar:

$$\begin{array}{lll} \texttt{SBPSW} & \alpha & \mathrm{PSW}_{7\ldots0} := \mathrm{PSW}_{7\ldots0} \vee S[\alpha] \\ \texttt{LBPSW} & \underbrace{\alpha}_{\mathrm{B}} & \mathrm{PSW}_{7\ldots0} := \mathrm{PSW}_{7\ldots0} \wedge \neg S[\alpha] \end{array}$$

3.4 Rechnerkernmodi

Eine wichtige Zielsetzung ist der Schutz der Daten von verschiedenen Modulen gegen einen unberechtigten Zugriff. Die Basis des Schutzes ist eine Einschränkung des Zugriffs auf Hardware-Komponenten. Es gibt also in einem System nicht vertrauenswürdige Benutzerprozesse mit eingeschränkten Rechten und das vertrauenswürdige Betriebssystem mit allen Rechten. Um im Rechnerkern die Situation unterscheiden zu können, gibt es zwei Arbeitsmodi im Rechnerkern:

Arbeitsmodus	Adressierung	
System	direkt	Betriebssystemkern
	virtuell	Teile des Betriebssystems
Benutzer	direkt	niemals!
	vituell	Prozesse

Anforderungen an die Sicherheit:

Nur im Systemmodus (also mit sogenannten priviligierten Befehlen) können verändert werden:

- der Arbeitsmodus
- der Adressierungsmodus
- die Seitentabellen und die zugehörigen Register
- die Adresse des gültigen PCB (Register PCBADR)
- die Adresse des gültigen SCB (Register SCBADR)

3.5 Privilegierte Befehle

- `HALT`
- Laden des PCB: `LPCB`

  Die Adresse steht in PCBADR. $\underbrace{\text{PC und PSW}}_{\text{im PCB}}$ werden nicht in PC und PSW übernommen, sondern auf den Keller gepuscht.

SP+4	PSW
SP	PC

- Speichern des PCB: `SPPCB`

  Zieladresse steht in PCBADR. PC und PSW werden nicht aus den Registern, sondern vom Keller genommen!

- `SP` $\langle SR \rangle$ α

  Speichere Sonderregister nach α

- `L` $\langle SR \rangle$ α

  Lade Sonderregister nach α

Sonderregister

IPL	Ablaufpriorität
MAPEN	MI mit einem Rechnerkern 0: direkte Adressierung 1: virtuelle Adressierung
RKMAP[31...16]	Nummer des Rechnerkerns
RKMAP[15...0]	Adressierungsmodus 0: direkte Adressierung 1: virtuelle Adressierung
SB	Adresse Anfang Seitentafel im Systembereich
SL	Länge Seitentafel für Systembereich
PCBADDR	Adresse des Prozeßkontrollblocks
SCBADR	Adresse des Systemkontrollblocks

Weitere priviligierte Befehle

- Rechnerkernalarm:

  `RKALARM` α Erzeuge einen Rechnerkernalarm für RK S[α]

- Systemaufruf (auch im Benutzermodus zulässig):

  `CHMK` α Aufruf des Systemdienstes S[α]

  Wirkung des Befehls `CHMK`:

  1. PSW und PC werden auf den Systemkeller gepusht.
  2. Der Operand S[α] gibt die Nummer des aufgerufenen Systemdienstes an. Diese Nummer wird ebenfalls im Systemkeller abgelegt und kann vom Systemkern ausgewertet werden. Weitere Parameter für den Dienst werden in einem Versorgungsblock bereitgestellt. Der Ort dieses Versorgungsblocks muß verabredet werden. Achtung bei Adreßübergaben: Aufrufer und Systemkern evtl. in getrennten Adreßräumen!

     Systemkeller:

PSW	SP+8
PC	SP+4
S[α]	SP

  3. Der vorhergehende Arbeitsmodus im PSW wird mit dem gegenwärtigen Arbeitsmodus besetzt..
  4. Der aktuelle Arbeitsmodus im PSW wird auf B'11' (Systemmodus) gesetzt.
  5. Der PC wird auf die Adresse für die Behandlung von Systemdiensten gemäß dem Systemkontrollblock eingestellt (Systemdienstanfangsbehandlung).

Systemkontrollblock (SCB)

Unterbr.Nr.	Relativadr. im SCB	Priorität	Art d. Unterbrechung
1	+0	31	katastrophaler Fehler
2	+4	(übernommen)	Befehlsalarm
3	+8	(übernommen)	Speicherschutzalarm
4	+12	(übernommen)	Seite-fehlt-Alarm (page fault)
5	+16	(übernommen)	Trace
6	+20	(übernommen)	arithmetischer Alarm
7	+24	(übernommen)	`CHMK`
8	+28	(übernommen)	-reserviert-
9	+32	25	Rechnerkernalarm
10	+36	24	Weckeralarm (nur 4-RK-MI)
11	+40	23	Plattenspeicher
12	+44	22	Kanal 0 Empfänger
13	+48	22	Kanal 0 Sender
14	+52	21	Kanal 1 Empfänger
15	+56	21	Kanal 1 Sender
16	+60	20	Kanal 2 Empfänger
17	+64	20	Kanal 2 Sender
18	+68	19	Kanal 3 Empfänger
19	+72	19	Kanal 3 Sender

- Der Befehl `REI` (return from interrupt)
  Es werden die Register PC und PSW gemäß den Werten im Systemkeller besetzt. Dies entspricht der Einstellung des alten Zustandes des Aufrufers. Der Aufrufer wird also fortgesetzt.

  Der Systemkeller vor `REI`:

PSW	SP+4
PC	SP

  PC und PSW werden vom Systemkeller gepoppt:

  PC
  tmp1 := S[SP]; SP++
  tmp2 := S[SP]; SP++
  PSW

  Falls : tmp2.CM $<$ aktueller PSM.CM oder
  tmp2.IPL $> 0 \wedge$ tmp2.CM $\neq$ Systemmodus oder
  tmp2.PM $<$ tmp2.CM oder
  tmp2.IPL $>$ aktueller PSW.IPL

  dann : Befehlsalarm

  ansonsten: akt. PSW := tmp2; PC := tmp1

3.6 Unterbrechungen

Behandlung von Unterbrechungen im Befehlszyklus

```
doit:   if tracepending goto doit3 fi
        tracepending := trace
        führe Befehl aus
        setze PC neu
        if interner Alarm then
doit2:     push PC, PSW auf Systemkeller
           setze PSW new
           setze PC neu (gemäß Alarmart)
           goto doit
        fi
        if tracepending == true then
doit3:  tracepending:=false
           erzeuge Alarm
           goto doit2
        fi
        if externer Alarm then
           wähle einen von höchster Priorität aus (IPL')
           if IPL' > IPL then goto doit2 fi
        fi
        goto doit
```

3.7 Synchronisation

3.7.1 Einleitung

In jeder Multitasking- oder Multiprozessorumgebung besteht die Notwendigkeit, daß sich verschiedene Prozesse Daten miteinander teilen. Bei statischen Daten ist dies kein Problem. Bei variablen Daten hingegen muß es eine Möglichkeit geben, den Zugriff auf die Daten zu synchronisieren. Dabei ist der Begriff synchron dahingehend zu verstehen, daß eine Taktung hinsichtlich relevanter Ereignisse vorliegt.

Als Beispiel betrachte man eine verkettete Liste, in die ein Prozeß einen neuen Knoten einfügt, während ein anderer Prozeß dieselbe Liste lesend durchläuft. Ohne eine Form der Synchronisation würde der lesende Prozeß unter Umständen einen ungültigen Zeiger einlesen und verwenden. Eine Speicherverletzung wäre die Folge. Gleichzeitig entsteht durch das Fehlen einer Synchronisation ein Nichtdeterminismus, wie man sich an obigem Beispiel leicht vor Augen führen kann.

Ein adäquates Mittel zur Lösung dieser Aufgabe sind Semaphore. Ein Semaphor ist quasi ein Schlüssel zu einem Datum. Der Prozeß, der den „Schlüssel“ besitzt, kann ungehindert auf das Datum zugreifen. Jeder andere Prozeß, der zur selben Zeit auf das Datum zugreifen will, wird solange in einen Wartezustand versetzt, bis der Semaphor wieder frei ist. Zunächst jedoch werden noch einige grundlegende Begriffe behandelt, die zur Implementation und zum Verständnis von Semphoren notwendig sind.

3.7.2 Grundbegriffe und Beispiele

Nebenläufigkeit (concurrency)

Mehrere Prozesse, die gleichzeitig an der Lösung eines Problems arbeiten, werden als nebenläufig bezeichnet. Folglich sind nebenläufige Prozesse voneinander abhänig und beeinflussen sich gegenseitig.

Enge und lose Kopplung

Nebenläufigkeit kann auf folgende zwei Arten synchronisiert werden:

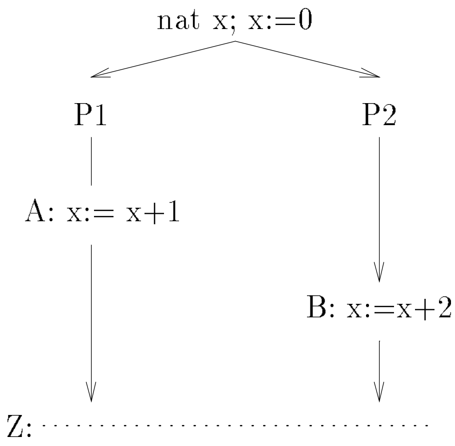
1. Die beteiligten nebenläufigen Prozesse verwenden gemeinsame Datenbereiche, auf welche alle beteiligten Prozesse zugreifen können. In diesem Fall spricht man von *enger Kopplung*.
2. Die nebenläufigen Prozesse besitzen eigene Datenbereiche, tauschen aber Informationen über ein Nachrichtensystem aus. In diesem Fall spricht man von *loser Kopplung*. Sie wird zum Beispiel bei modernen graphischen Benutzeroberflächen wie X-Windows zur Kopplung zwischen Anwenderprozeß und Benutzeroberfläche verwendet.

Bemerkung:

Die Begriffe Nebenläufigkeit und Synchronität sind immer auf eine konkrete Prozeßumgebung zu beziehen, da sie sich bei Verfeinerung oder Vergröberung ändern.

3.7.3 Beispiel für die Notwendigkeit von mutual exclusion

Die Prozesse P1 und P2 seien nebenläufig.



Sie greifen beide auf die gemeinsame Variable x zu. P1 verwendet den Rechnerkern 0, P2 verwendet den Rechnerkern 1.[*] Z sei nun ein beliebiger Zeitpunkt nach Ausführung der Anweisungen A und B. Es stellt sich nun die Frage, welchen Wert die Variable x zum Zeitpunkt Z besitzt.

Wie im Folgenden deutlich wird, ist das Ergebnis nicht deterministisch. Es können folgende Fälle auftreten:

1. P1 liest $x = 0$, erhöht, speichert $x := 1$;
   P2 liest $x = 1$, erhöht, speichert $x := 3$;
2. P2 liest $x = 0$, erhöht, speichert $x := 2$;
   P1 liest $x = 2$, erhöht, speichert $x := 3$;
3. P1 und P2 lesen $x := 0$;
   P1 erhöt und speichert $x := 1$;
   P2 erhöt und speichert $x := 2$;
4. P1 und P2 lesen $x := 0$;
   P2 erhöt und speichert $x := 2$;
   P1 erhöt und speichert $x := 1$;

Je nach zeitlichem Verlauf erhalten wir als Ergebnis 1,2 oder 3. Der durch die Asynchronität bedingte grundlegende Nichtdeterminismus der Nebenläufigkeit wirkt sich hier auf das Ergebnis aus. Dieser Nichtdeterminismus läßt sich nur dadurch verhindern, indem man sicherstellt, daß die Veränderung von x in beiden Prozeßsen unter *gegenseitigem Ausschluß*[†] erfolgt. Dann ist Fall 1. oder Fall 2. garantiert. Die übrigen Fälle sind ausgeschlossen.

[*] Das Problem ist nicht auf echte Parallelität beschränkt. Es stellt sich auch bei zeitlich verschränkter Arbeit mehrerer Prozesse auf einem Rechnerkern

[†] mutual exclusion

3.7.4 Synchronisationsbefehle `JBCCI` ,`JBSSI`

`JBCCI` ,`JBSSI` sind unteilbar (atomar):

- Operationen werden im Rechnerkern ohne Unterbrechungen ausgeführt ‡
- Kein anderer Rechnerkern darf während der Ausführung auf dieselbe Speicherzelle zugreifen §

`JBCCI` ,`JBSSI` sind nahe Verwandte der Klasse der *Test-und-Set-Befehle*:

`TSI` ⟨flag⟩,⟨register⟩	holt atomar ⟨flag⟩ ins ⟨register⟩ und setzt ⟨flag⟩ auf 1

Implementierung von PP und VV (als Prozeduren in MI):

```
PP(s):
w:   JBCCI I7,!(4+!SP),w          RET

VV(s):
     MOVE B I1,!(4+!SP)           RET
```

3.7.5 Semaphore

Der Semaphor wurde von E.W.Dijkstra, 1968 in „Cooperating Sequential Processes“ eingeführt.

Typen:

- binäre Semaphore (Wertebereich {0,1})
- allgemeine Semaphore (Wertebereich $\mathbb{N}$)

Operationen:

1. Initialisierung (einmalig)
2. P-Operation (PP-Operation bei binären Semaphoren)
3. V-Operation (VV-Operation bei binären Semaphoren)

‡einfach über den Befehlszyklus zu realisieren

§Lösung über die Hardware: z.B. daisy chain oder Bus arbitration

Semantik der Operationen

$$\left.\begin{array}{lll} 1. & \surd & \\ 2. & PP(s) & : \text{while(s==0); s := 0;} \\ 3. & VV(s) & : \text{s := 1;} \\ 2'. & P(s) & : \text{while(s==0); s}--; \\ 3'. & V(s) & : \text{s++;} \end{array}\right\} \text{jeweils atomar}$$

Realisierung eines allgemeinen Semaphors:

Mittels eines binären Semaphors `spv` wird der Zugriff auf den allgemeinen Semaphor `s` synchronisiert.

z.B. (in MI):

```
l1:
    JBCCI I7,spv,l1          -- PP(spv)
    CMP W I0,!(4+!SP)        -- s == 0
    JNE frei
belegt:
    MOVE B I1,spv            -- spv :=1; VV(spv)
    JUMP l1
frei:
    SUB W I1,!(4+!SP)        -- s--
    MOVE B I1,spv            -- spv := 1
    RET
l2:
    JBCCI I7,spv,l2          -- PP(spv)
    ADD W I1,!(4+!SP)        -- s++
    MOVE B I1,spv            -- VV(spv)
    RET
```

3.7.6 Organisation des Zugriffs auf den Betriebsystemkern

Nebenläufigkeit, gemeinsam benutzter Datenbereiche.

- der ganze Betriebsystemkern wird ein kritischer Bereich. Diese Variante ist zwar am einfachsten zu implementieren, sie ist aber auch die denkbar unökonomischste.
- Sperrung so fein granuliert wie möglich.
- Threads.

Aus Gründen der Vereinfachung wird in der Vorlesung die erste Variante verwendet.

Standardablaufsequenz im Betriebssystem

- Semaphor `SysFrei` regelt den Zugriff auf den BS-Kern
- Setze Unterbrechungspriorität auf „fast maximal". Dadurch werden keine externen und keine Unterbrechungen durch andere Benutzerprozesse mehr angenommen.
- Interne Unterbrechungen, die im selben Prozeß verursacht werden, während er im BS-Kern ist, sollen berücksichtigt werden.
- Eine Variable `NrRKImSystem` notiert, welcher RK gerade den BS-Kern benützt. Der Zugriff darauf wird mittels des Semaphors `NrRKFrei` synchronisiert.

Datenstruktur:

```
     SysFrei:  DD  B  1                    (bzw. 0)
    NrRKFrei:  DD  B  1
NrRKImSystem:  DD  W  -1
        NrRK:  DD  W  -1
 prozeßliste:  DD     (W (0)*23,(0)*9)*10
```

Unterbrechungsbehandlung:

```
uab:
    LIPL I 30                   -- nur katastophale Alarme und
                                -- interne Unterbrechungen kommen noch durch
    JBCCI I 7, SysFrei, systembelegt
systemfrei:
    JBCCI I 7, NrRkFrei, systemfrei
                                -- Zugriff auf NrRkImSystem koordinieren
                                -- Vermeidung von Verklemmung bei
                                -- Zugriff auf die beiden Semaphore
    SPRKMAP NrRkImSystem
    SH I -16, NrRkImSystem, NrRkImSystem
                                -- Nr. des RK rechtsbündig
    MOVE B I1, NrRkFrei         -- VV(NrRkFrei)
    CALL VerlangteAktion
    ADD W I4, SP                -- poppe zusätzlichen Parameter
                                -- bei Systemaufruf
    JUMP rechnerkernvergabe
systembelegt:
    JBCCI I7, NrRkFrei, uab     -- PP(NrRkFrei)
    SPRKMAP NrRk
    SH I -16, NrRk, NrRk        -- Nr. des Rk rechtsbündig
    CMP W NrRk, NrRkImSystem
    JEQ wiederbetreten
    MOVE B I1, NrRkFrei         -- VV(NrRkFrei)
```

```
    JUMP uab
wiederbetreten:                   -- Unterbrechung innerhalb
                                  -- Unterbrechungsbehandlung
    MOVE B I1, NrRkFrei           -- VV(NrRkFrei)
    CALL VerlangteAktion
    ADD W I4, SP                  -- poppe zus. Parameter
    REI


rechnerkernvergabe:
    SPPCB                         -- Ablegen des PCB des unterbrochenen
                                  -- Prozesses p (nach PCBADR) PC, PSW vom Keller
    setze p rechenbereit, falls geeignet
    wähle einen rechenbereiten Prozeß q aus
    MOVEA adr(PCB_q), R0
    LPCBADR R0
    setze q rechenend
    SPRKMAP 108+!R0               -- Position der RkNr im Prozeßlistenelement
    SH I-16, 108+!R0, 108+!R0     -- NrRk rechtsbündig
warte:
    JBCCI I7, NrRkFrei, warte     -- PP(NrRkFrei)
    MOVE W I-1, NrRKImSystem      -- kein Rk mehr im System
    MOVE B I1, NrRkFrei           -- VV(NrRkFrei)
    LPCB                          -- Laden des PCB_q; PC_q, PSW_q im Keller
    MOVE B I1, SysFrei            -- VV(SysFrei)
    REI                           -- PC_q, PSW_q werden geladen
```

Beachte: NICHT vorkommen darf:

$$
\begin{array}{ll}
\text{Prozeß 0} & \text{Prozeß 1} \\
\left.\begin{array}{ll}
\text{PP(SysFrei)} & \text{PP(NrRkFrei)} \\
\text{PP(NrRkFrei)} & \text{PP(SysFrei)}
\end{array}\right\} & \rightarrow \text{Verklemmung/Deadlock}
\end{array}
$$

Dies wird durch die gleiche Reihenfolge der PP-Befehle vermieden.

3.8 Video, Bildschirmein- und ausgabe

3.8.1 Terminals

Terminals mit serieller Schnittstelle

- Schnittstellen: V24, RS 232, UART (ein mit dem Rechner verbundener Baustein, welcher Bytes sequentialisiert.)
- übliche Baudraten: (75),300 (bei der Post), 2400, 9600, 19200, 115200

- intelligente Terminals mit eigenem Prozessor und Speicher: In diesem Bereich ist besonders das am MIT entwickelte X-Terminal von Bedeutung, da es die komplette Oberflächenverwaltung, wie z.B. die Fensterverwaltung, selbst durchführt und so den Rechner entlastet. Im Bereich der verteilten graphischen Benutzeroberflächen ist X der Standard.

Speicherabgebildete Terminals

- PC, Video-RAM (bei Segment B000 (SW), B800 (Farbe), A000 (VGA)):
  I m Gegensatz zu Terminals mit serieller Schnittstelle findet die Kommunikation hier über einen Speicherbereich statt, welchen sich Prozessor und Videosteuerwerk teilen. Diesen Speicherbereich, das Video-RAM, bezeichnet man als dual ported. Dieses Verfahren benötigt zwar weniger Resourcen, hat aber ansatzbedingt entscheidende Nachteile im Bereich verteilter Anwendungen, da Programm und Benutzeroberfläche auf demselben Rechner laufen müssen.
- Zeichen- und pixelorientiert

3.8.2 Tastatur

Bei den meisten Tastaturen wird die Tastennummer abgeliefert (scan code). Von der Tastatur-Unterbrechung (MS-DOS: INT9) werden diese Codes in (Standard-) Zeichendarstellungen wie ASCII, Extended ASCII oder ISO umgewandelt. Zusätzliche Flexibilität bieten Tastaturtreiber. Man unterscheidet grundsätzlich zwischen zwei Arbeitsweisen:

- unverarbeitender Modus (raw)
- verarbeitender Modus (cooked)

In der Regel wird die Eingabe gepuffert. D.h. die Unterbrechung erfolgt nicht sofort, sondern die Eingabe wird in einem Puffer gesammelt. Bei mehreren Prozessen kann so eine Differenzierung stattfinden. Jeder Prozeß erhält dabei seinen eigenen Eingabepuffer z.B. X-Windows.

3.8.3 Software-Unterstützung (Schnittstellen) für Konsol-E/A

- Speicherabbildung: Zeichen auf dem Bildschirm entsprechen Zeichen im VideoRAM. Dieses Verfahren ist zwar sehr schnell, aber nicht sehr portabel.
- Systemdienste (z.B. INT9, 10, 15, 16, ... in MS-DOS)

- ANSI-Steuersequenzen, z.B.

ESC := H'1B'	
ESC[r;cH	position cursor, row r, column c
ESC[2]	erase screen
ESC[n_1;n_2;...;n_rm	set attributes n_1;n_2;...;n_r
ESC[k	erase to end of line
ESC['k string' p	redefine key k to produce string

- die curses-Bibliothek[¶] stellt Datenstrukturen und Prozeduren (Funktionen) in C zur terminalunabhänigen Manipulation von Bildschirmen zur Verfügung. Die Änderungen erfolgen auf einem virtuellen Bildschirm, welcher auf den realen Bildschirm abgebildet wird.
- Grafikbibliotheken z.B. CORE, GKS, PEX, u.a.
- GUI (Graphische Benutzeroberflächen): X, Motif, NeWS(entspricht X + Postscript)

[¶]siehe `man curses`

4. Virtuelle Adressierung

4.1 Einleitung

Motivation zur Einführung virtuellen Speichers ist der Wunsch mehrere Programme gleichzeitig, quasiparallel, arbeiten zu lassen.

Bei der Verwendung direkter Adressierung ergäbe sich zum Beispiel folgende Speicherbelegung:

System	U_1	U_2	U_3	U_4

Stirbt nun zum Beispiel der Prozeß U_2 so entsteht eine Lücke im Speicher. Wie leicht nachzuvollziehen ist, ist es höchst unwahrscheinlich, daß ein neuer Prozeß U_5 genau diese Lücke füllt. Offensichtlich führt eine deratige Speicherverwaltung auf Dauer zu einer starken Fragmentierung des Arbeitsspeichers. Dabei kann durchaus die kroteske Situation auftreten, daß der gesamte freie Speicher zwar für einen neuen Prozeß ausreichen würde, der Speicher aber in viele kleine Teile zerstückelt ist und der Prozeß aus Speichermangel nicht gestartet werden kann. Dieses Problem wird noch verschärft, falls Prozesse dynamisch Daten anlegen und wieder freigeben.

Eine Bereinigung (Garbage Collection) des Arbeitsspeichers ist nicht möglich, da alle Prozesse auf einen bestimmten Speicherraum fixiert sind und folglich nicht im Speicher verschiebbar sind.

Da zu einer bestimmten Zeit immer nur ein kleiner Bereich eines Programmes bearbeitet wird, liegt es nahe zum Zweck einer besseren Speicherausnutzung die „brachliegenden“ Teile des Programms auf einen Hintergrundspeicher auszulagern. So wird es möglich auch Prozesse auf einer Maschine abzuarbeiten, deren Speicherbedarf die Größe des auf der Maschine vorhandenen Arbeitsspeichers übersteigt.

Adressierungsarten:

1. direkte Adressierung: Programmadressen := Maschinenadressen

2. Basisadressierung: Maschinenadresse := Basis + Programmadresse

Swapping(Auslagern) und Speicherzuteilung

Das Auslagern eines "schlafenden" Prozesses auf die Festplatte (in den Swapbereich) kann mittels Listen geregelt werden:

1. Man hat eine Liste der freien bzw. belegten Speicherblöcke und geht nach einer der folgenden Methoden vor:

   - First Fit: erster passender Eintrag wird genommen
   - Best Fit: es wird das Loch gesucht, das am besten paßt
   - Worst Fit: das größte Loch wird genommen
   - Next Fit: die Liste wird jeweils vom letzten Punkt aus durchsucht

   Praxis: Worst Fit ist besser als Best Fit, da Best Fit dazu tendiert kleine Löcher zu erzeugen, während Worst Fit zu großen Löchern tendiert.

2. Buddy-System: Es gibt Teillisten mit jeweils allen Blöcken der Größe 1, 2, 4, 8, 16,…, 1024, 2048,…. Die Blöcke werden dann der Größe nach vergeben. Dieses Verfahren erfordert viel Verwaltungsaufwand.

4.2 Grundlagen der Organisation virtueller Speicher

Die Idee zur Verwendung eines virtuellen Speichers geht auf das Jahr 1959 zurück und wurde damals erstmals in England im *ATLAS*-Computer realisiert.

Programmadressen (linearer Prgrammadreßraum) Seiten (Größe ist Zweierpotenz)	$\Longrightarrow$ Abbildung	Maschinenadreßraum (linearer Maschinenadreßraum) Kacheln (gleiche Größe wie Seiten)

Programmadresse:

31 … 10	9 … 0	
s	r	Programmadresse

$\Downarrow$

k	r	Seitenkacheltabelle

Seitenkacheltabelle (SKT):

Die Abbildung der Seiten auf Kacheln ist injektiv und partiell. In der Seitenkacheltabelle sind zu jeder Kachel zusätzliche Informationen zur leichteren Verarbeitung gespeichert.

- Seite existiert (e) Dieses Flag wird benötigt, da es möglich ist, daß sich Lücken im Adreßraum befinden und folglich nicht jede Seite existiert.
- Seite geladen (v*)
- Zugriffsrechte (R)

  r : lesbar
  w : beschreibbar
  x : ausführbar
- Zugriffsbit (z)

  Das Zugriffsbit ist 1 falls auf die Seite zugegriffen wurde. Beim Laden einer Seite wird es auf 0 gesetzt. Es hilft dem Betriebssystem bei der Entscheidung welche Seiten ausgelagert werden sollen.
- Modifikationsbit (m)

  Das Modifikationsbit ist 1, falls eine Seite verändert wurde.

Arten der Organisation der SKT

- indizierte, lokale Darstellung (I). Jeder Prozeß besitzt seine eigene SKT. Diese Variante ist jedoch mit einem hohem Maß an Redundanz verbunden, da die meisten Einträge unbenutzt bleiben.
- assoziative, lokale Darstellung (AL). Jeder Prozeß speichert die Liste seiner Seiten-Kachel-Paare. Für nicht geladene Seiten wird kein Speicher verbraucht. Bei einer assoziativen Darstellung ist kein schneller, indizierter Zugriff über die Seitennummer mehr möglich. Es gibt aber besondere Speichertypen (Assoziativspeicher), die diesen Zugriff effizient durchführen können.
- assoziative, globale Darstellung (AG). Es gibt eine systemweite Seiten-Kachel-Liste.
- indizierte, globale Darstellung (IG). Es gibt eine systemweite SKT.

Vor- und Nachteile

	I	AL	AG	IG
Größe der SKT entspricht Programmadreßraum	j−	n+	n+	n+
Größe der SKT entspricht Anzahl der Kacheln	n−	j+	j+	j+
Tabellensuche nach Seite	n+	j−	j−	j−
Tabelle komprimierbar (bei Auslagerung)	n+	j−	j−	n+
Gemeinsame Seiten	j+	j+	j+	n−

*available

Mehrstufige SK-Tabellen

Da auf die SKT häufig zugegriffen wird, wird sie gewöhnlich in einem schnellen Cache-Speicher abgelegt. Diese Speicher sind allerdings teuer und von vergleichsweise geringer Kapazität. Da für große Hintergrundspeicher (im GByte Bereich) die zugehörige SKT schon mehrere MByte groß ist, kann im allgemeinen nicht die gesamte SKT im Cache gehalten werden. Aus diesem Grund verwendet man in der Praxis häufig mehrstufige SKT's. Das heißt, daß der Adreßraum, in dem die SKT gehalten wird, seinerseits wieder in Seiten unterteilt wird, über die wiederum in einer anderen SKT Buch geführt wird. Dieses Verfahren kann mehrfach iteriert sein.

4.3 Virtueller Speicher bei der MI

Überlegungen zur Seiten-/Kachelgröße

1. Je kleiner die Seite, desto schneller der Transfer ASP $\leftrightarrow$ Hintergrundspeicher.
2. Je kleiner die Seite, desto kleiner der Verschnitt.
3. Je kleiner die Seite, desto mehr relevante Daten enthält sie (prozentual).
4. Je größer die Seite, desto weniger relativer Overhead beim Transport ASP $\leftrightarrow$ Hintergrundspeicher.
5. Je größer die Seite, desto effizienter der Datentausch ASP $\leftrightarrow$ Hintergrundspeicher.
6. Je größer die Seite, desto weniger Seitenfehleralarme [†], vorausgesetzt die Menge der relevanten Daten ist auf großen Seiten größer.
7. Je größer die Seite, desto kleiner die SKT.
8. Je größer die Seite, desto mehr Puffer (Hardware) für den Transport ASP $\leftrightarrow$ Hintergrundspeicher.
9. Je größer die Busbreite (ASP $\leftrightarrow$ Plattensteuerwerk), desto größere Seiten sind möglich.

Die optimale Seitengröße ist folglich vom technischen Fortschritt abhänig. Die DEC-VAX verwendet 512 Byte/Seite. Heute sind zwischen 4 bis 8 kByte/Seite üblich.

[†] eigentlich: "Seite-fehlt-Alarm"

Name	Funktion
PCB.P0B	Adresse dder SKT Für den P0-Bereich
	(Adresse im Systembereich)
P0L	Länge
PCB.P1B	Adresse der SKT für den P1-Bereich
	(Adresse im Systembereich)
P1L	Länge
SB	Maschinenadresse der SKT für den Systembereich
SL	Länge (dieser Tabelle)
MAPEN/RKMAP	

Abbildung einer Adresse im P0-Bereich

4.4 Paging-Verfahren

Die zentrale Frage des Pagings ist:

Welche Seite soll aus dem ASP auf den HGSP ausgelagert werden?

1. Die ideale/optimale/visionäre Lösung:
   Ersetze die Seite, die erst am weitesten in der Zukunft wieder benötigt wird.

2. Not recently used (NRU)

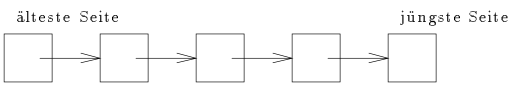
m	z
modifiziert	referenziert

   4 Fälle:

z	m
0	0
0	1
1	0
1	1

   Das z-Bit wird z.B. alle 20 msec gelöscht. Wenn eine Seite ausgelagert werden muß, wird eine Seite mit der niedrigsten Kategorie aus der Tabelle beliebig ausgewählt und ausgelagert.

3. FIFO

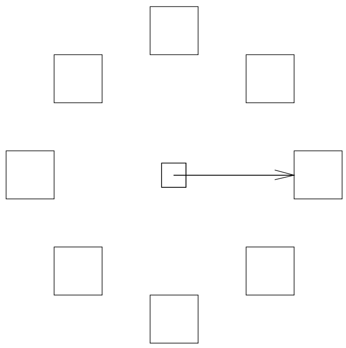
   Liste der Seiten in der Reihenfolge in der sie eingelagert werden. Die älteste Seite wird ausgelagert.

Vorteil	leicht zu implementieren.
Nachteil	häufig benutzte Seiten werden irgendwann auch ausgelagert.

4. Zweite Chance[‡]

   Liste wie NRU, mit z-Bit. Erstes Element, falls z=1 wird mit z=0 am anderen Ende wieder eingefügt. Iteriere für die weiteren Elemente der Liste. Die erste Seite mit z=0, die gefunden wird, wird ausgelagert.

5. Der Uhralgorithmus

   Der Uhralgorithmus arbeitet analog zum Verfahren 4., verwendet aber eine zyklische anstelle einer linearen Liste.

   z=0: Seite wird verdrängt
   z=1: z:=0, bewege Zeiger um eine Position

6. LRU (least-recently-used) Algorithmus.

   Es wird die Seite verdrängt, auf die am längsten nicht mehr zugegriffen wurde.

   Eine praktische Implementierung: Notiere zu jeder Seite im ASP, die „Uhrzeit“, wann zum letzten Mal auf sie zugegriffen wurde. Verdränge die Seite mit der kleinsten Uhrzeit.

[‡] Kombination der Ideen von NRU und FIFO

5. Prozesse und Betriebsmittelverwaltung

5.1 Prozesse

- Prozeß = Programm in Ausführung

  Im Gegensatz zur statischen Aufschreibung des Programmtextes

- sequentielle Prozesse

  Jeder Prozeß besteht aus einer Folge von hintereinander ausgeführten Aktionen

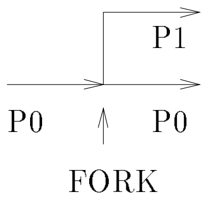
- quasiparallele Ausführung

  mehrere Prozesse werden „zugleich“ auf einem Rechnerkern ausgeführt, indem abwechselnd die Aktionsfolgen dieser Prozesse bearbeitet werden.

  → Mehrprogrammbetrieb, Multiprogramming, Time-Sharing

- Prozeßgenerierung

  z.B. Unix: Es gibt einen Ur-Prozeß, der sich durch den Systembefehl `FORK` aufspaltet.

  `FORK` ist das Mittel um neue Prozesse zu generieren; aus einem werden zwei.

  P0 und P1 stehen in einem Eltern ↔ Kind Verhältnis zueinander. P0 kann auf die beendigung von P1 warten, aber nicht umgekehrt.

- Prozeßkommunikation:
  - Unterbrechungen
  - kritische Bereiche
  - Semaphore

- Ereigniszähler
- Monitore
- Blockieren
- Nachrichten (Mailboxen)

Probleme:

- Zu einem Prozeß gehören zahlreiche Verwaltungsinformationen $\Rightarrow$ Der Context-Switch zwischen Prozessen ist aufwendig.
- Zur effizienten breitbandigen Kommunikation zwischen Prozessen ist gemeinsamer Speicher notwendig. Dies ist wegen getrennter Adreßräume im allgemeinen nicht möglich.

Die Lösung dieser Probleme ist die Abmagerung des Prozeßkonzeptes, sie führt zum Konzept der sogenannten Threads.

5.2 Threads oder leichtgewichtige Prozesse

Thread := sequentiell ausgeführter Programmteil,
ohne den großen Overhead von Prozessen.

Anstelle eines Prozesses treten ein oder mehrere Threads.

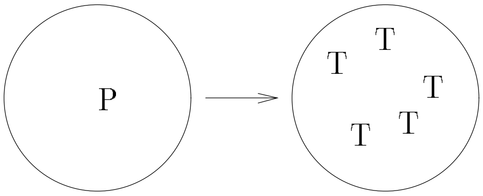
Vergleich zwischen Threads und Prozessen:

Threads	Prozesse
Programmzähler	Adreßraum
Stack	globale Variablen
Registersatz	geöffnete Dateien
abhängige threads	abhängige Prozesse
Zustand	Uhren
	Signale
	Semaphore
	Verwaltungsinformationen

Zusammenhängende Threads benutzen den gleichen Adreßraum.

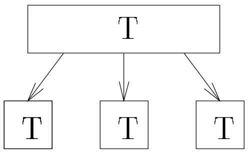
Arten des Zusammenwirkens

z.B. Datenbank-Server: bisher ein Prozeß, jetzt variable Anzahl kooperierender Threads



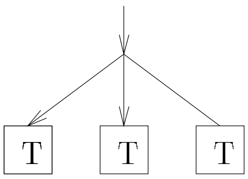
Als Methode der Zusammenarbeit kommen in Frage:

- Verteiler-Arbeiter:

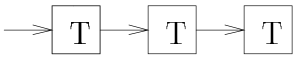
  Ein Verwalter verteilt ankommende Aufgaben auf die einzelnen Arbeits-Threads.

- Team

  Die Threads im Team sind alle gleichberechtigt. Eine ankommende Aufgabe wird vom nächsten bereiten Thread übernommen.

- Pipeline

  Jeder Thread der Gruppe hat sich auf eine andere Teilaufgabe spezialisiert.

Prozeß : rein sequentiell; blockiert, falls angeforderte Resourcen fehlen.

Threads : ein Thread immer sequentiell; Thread kann blockieren; aber restliche Threads laufen weiter. Die Threads werden untereinander parallel ausgeführt.

Vorteile Threads:

- schneller Kontextwechsel zwischen Threads einer Gruppe, da der Kontext klein ist.
- einfache/schnelle Kommunikation zwischen Threads einer Gruppe, durch einen gemeinsamen Adreßraum.

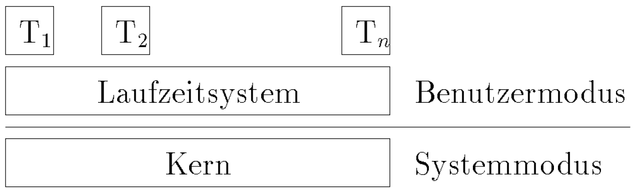
Problem:

Threads einer Gruppe benutzen einen gemeinsamen Adreßraum und sind daher nicht (wie Prozesse) voreinander geschützt. Dies kann jedoch akzeptiert werden, da Threads einer Gruppe im allgemeinen nur einem Benutzer gehören.

Thread-Pakete:

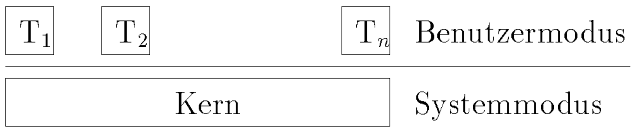
- Problem „globaler“ Variablen
- Implementierung des Thread-Konzeptes:
    a) Verwaltung im Benutzeradreßraum (1)
    b) Verwaltung im Kern (2)

(1)



Scheduling der Prozesse erfolgt wie gehabt im Kern, Scheduling und Verwaltung der Threads erfolgt innerhalb der Prozesse durch ein Laufzeitsystem. Der Betriebssystemkern muß nicht verändert werden.

(2)



Scheduling und Verwaltung aller Threads erfolgt im Betriebssystemkern. Gruppen von Threads ersetzen den Prozeß.

Eigenschaften:

	Variante (1)		Variante (2)
+	flexibel, da unabhängig vom Betriebssystem.	+	zuverlässig
+	effizient, da für Kontextwechsel kein Wechsel in den Systemmodus notwendig	+	besseres Scheduling, da alle Threads zentral verwaltet werden.
−	Thread blockiert durch Systemaufruf die restlichen Threads der Gruppe.		

Beobachtung: Threads enden gewöhnlich mit blockierenden Systemaufrufen (nur dann ist das Thread-Konzept sinnvoll!)
$\Rightarrow$ in der Praxis wird gewöhnlich Variante (2) benutzt.

5.3 Prozeß-Scheduling

Ziele (Meliorierung von ...):

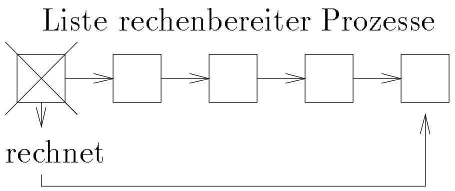
1. Fairneß
2. Auslastung der Resourcen (Rechnerkern, Hintergrundspeicher)
3. Antwortzeit
4. Verweilzeit
5. Durchsatz
6. Overhead (durch Scheduler und Kontextwechsel)

Arten:

1. Preemptive
2. Non-preemptive = run-to-completition

5.3.1 Round-Robin-Scheduling

Jeder rechenbereite Prozeß erhält für eine kurze Zeit den Rechnerkern. Die Prozesse werden nach der FIFO-Strategie behandelt



Es besteht ein Tradeoff zwischen der Antwortzeit eines Prozesses und dem Overhead; kleinere Zeitscheiben verringern die Antwortzeiten der Prozesse, erhöhen aber den Zeitanteil, den der Rechnerkern für die Verwaltung der Prozesse verbraucht.

Typische Größe der Zeitscheiben: 100 msec

5.3.2 Prioritätsscheduling

Jedem Prozeß wird eine Prioritätszahl zugeordnet. Der rechenbereite Prozeß mit der höchsten Priorität erhält den Rechnerkern.

Die Zuteilung der Priotätszahl erfolgt entweder:

statisch : jeder Prozeß besitzt eine festgelegte Priorität. Der Uhrtakt verringert die Priorität während des Rechnens eines Prozesses, um auch Prozesse mit niedrigerer Priorität zum Zug kommen zu lassen.

dynamisch : die Priorität eines Prozesses wird ständig angepasst.
$f :=$ ausgenützer Teil der letzten Zeitscheibe,
Priorität $:= \frac{1}{f}$
Wirkung: Interaktive Prozesse (mit großer Anzahl an I/O-Befehlen) werden vor langlaufenden Rechenprozessen bevorzugt, da interaktive Prozesse in der Regel ihre Zeitscheiben nicht voll ausnutzen. Dadurch wirkt das System für den Benutzer subjektiv schneller.

Prozesse gleicher Priorität werden nach dem Round-Robin-Verfahren behandelt.

5.3.3 Mehrere Schlangen

Schlangen mit verschieden langen Zeitscheiben.

Zum Beispiel:

Schlangen-Nr.:	1	Terminal
	2	Plattenspeicher, E/A
	3	Quantum voll aufgebraucht
	4	Quantum mehrmals voll aufgebraucht
zugehörige Zeitscheibe:	1	1 Einheit
	2	2 Einheiten
	3	4 Einheiten
	4	8 Einheiten

Warteschlangenpriorität der Schlange 1 ist hoch, die der Schlange 4 niedrig.

Wirkung: langlaufende Prozesse dürfen längere Zeit auf einmal rechnen, bekommen aber seltener Rechenzeit zugeteilt.

5.3.4 Shortest Job First

Wichtig vor allem im Batch-Betrieb. Der kürzeste Auftrag wird zuerst bearbeitet.

$\Rightarrow$ Die mittlere Verweildauer eines Auftrages wird minimiert.

Verwende Schätzung für die Laufzeit der nächsten Aktion. Fortschreibung des Schätzwertes am Ende einer Aktion:

$$S' := \alpha S + (1 - \alpha) \cdot (\text{Zeit für letzte Aktion})$$

$$\text{mit} \quad 0 \leq \alpha \leq 1 \quad \text{z.B.} \quad \alpha = \frac{1}{2}$$

Die dynamische Fortschreibung eines Schätzwertes wird als *Alterung* bezeichnet.

5.3.5 Garantiertes Scheduling

n Jobs, z.B. jeder Prozeß soll $\frac{1}{n}$ der Prozessorleistung erhalten. Bei jedem Uhrentakt werden die Prozesse nach der bisher verbrauchten Prozessorleistung sortiert. Der Prozeß mit der niedrigsten verbrauchten Prozessorleistung erhält den Rechnerkern.

5.3.6 Zweistufiges Scheduling

Bisherige Annahme:

Das Scheduling aller Prozesse ist gleich (zeit-)aufwendig.

Dies ist in der Realität nicht der Fall, da Prozesse auf Hintergrundspeicher ausgelagert werden können. Das Scheduling eines solchen Prozesses dauert länger als für einen nicht-ausgelagerten Prozeß.

Diesem Umstand trägt das zweistufige Scheduling Rechnung.

Untere Stufe:	Scheduling der Prozesse im ASP
Obere Stufe:	Ein-/Auslagern von Prozessen von/zum Hintergrundspeicher.

Kriterien für die obere Schicht:

- bisherige Verweildauer im ASP
- verbrauchte Rechenzeit
- benötigter Platz
- Priorität

Auf der unteren Stufe wird eines der beiden bereits erwähnten Verfahren eingesetzt. Der Einsatz der Kriterien in der oberen Stufe erfordert Feingefühl und muß auf die jeweilige Systemumgebung (z.B. Hardwarearchitektur) Rücksicht nehmen.

5.4 Betriebsmittelvergabe

Betriebsmittel kann sein: Speicher, Drucker, E/A-Kanäle, CD-ROM, Diskettenlaufwerke, Rechnerkern.

Man unterscheidet unterbrechbare und nicht unterbrechbare Betriebsmittel.

Zum Beispiel:

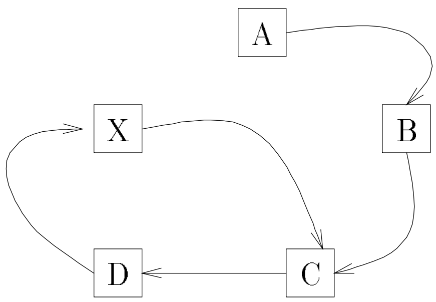
- Rechnerkern und Arbeitspeicher sind unterbrechbare Betriebsmittel
- Drucker, Diskettenlaufwerke sind im allgemeinen nicht unterbrechbare Betriebsmittel.

Unterbrechbar heißt folglich: das Betriebsmittel kann ohne Schaden einem Prozeß entzogen und ihm später wieder zugeteilt werden.

Verklemmung (Deadlock)

Def. : Es gibt eine nichtleere Menge von Prozessen von denen jeder auf ein Ereignis wartet, das nur von einem anderen Prozeß in der Menge ausgelöst werden kann.

Beispiel:



Das Beispiel ist kein minimaler Deadlock, da der Deadlock auch dann noch besteht, wenn A, B aus der Menge entfernt werden.

Voraussetzungen für Deadlocks

- wechselseitiger Ausschluß
- Prozesse fordern weitere Betriebsmittel an
- nicht unterbrechbare Betriebsmittel
- zyklische Wartebedingung

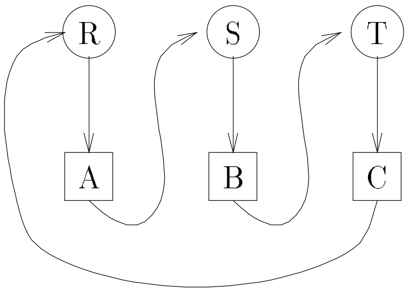
Diese vier Kritrien sind notwendige Vorraussetzungen für Deadlocks.

Graphische Darstellung

○	Betriebsmittel
□	Prozeß
(R) → [A]	Prozeß A belegt/besitzt Betriebsmittel R
[B] → (S)	Prozeß B fordert Betriebsmittel S an

Beispiel

Prozesse	anfordern		freigeben	
A	R_1	S_4	$\overline{R}$	$\overline{S}$
B	S_2	T_5	$\overline{S}$	$\overline{T}$
C	T_3	R_6	$\overline{T}$	$\overline{R}$



↯ Zyklus im Betriebsmittelgraph → Deadlock

Methoden zur Deadlock-Vermeidung

- Ignorieren des Problems
- Erkennen und Beheben
- Vorsichtigte Betriebsmittelvergabe
- Konzeptionelle Vermeidung einer der vier notwendigen Bedingungen für Deadlocks.

5.4.1 Die Vogel-Strauß-Methode

Vorkehrungen gegen Deadlocks werden üblicherweise nur getroffen, wenn ein Deadlock mit signifikanter Wahrscheinlichkeit auftritt; anderfalls nimmt man das Risiko in Kauf.

5.4.2 Erkennen und Beheben von Verklemmungen

Mehrere Betriebsmittelklassen, ein Betriebsmittel pro Klasse

Mehrere Betriebsmittel pro Klasse

Darstellung durch Betriebsmittel-Vektor $E = (E_1, E_2, \ldots, E_m)$. E_j = Anzahl der Exemplare des j-ten Betriebsmittels.

$$\text{Betriebsmittelmatrix} \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$

C_{ij} = Anzahl der Exemplare des j-ten Betriebsmittels, welche vom i-ten Prozeß belegt sind.

Betriebsmittelvektor $(A_1, \ldots, A_m)$
$A_j = E_j - \sum_{i=1}^{n} C_i j$ = Anzahl der nicht belegten Exemplare des j-ten Betriebsmittels.

$$\text{Betriebsmittelmatrix} \begin{pmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ R_{21} & R_{22} & \cdots & R_{2m} \\ \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & \cdots & R_{nm} \end{pmatrix}$$

R_{ij} = Anzahl der Exemplare des j-ten Betriebsmittel, die vom i-ten Prozeß angefordert werden.

Algorithmus (zur Erkennung von Verklemmungen)

while $\exists$ Anforderungsvektor $R_i \leq A$, i unmarkiert **do**
    $A = A + C_i$
    markiere i
od

Satz: Es gibt eine Verklemmung $\Leftrightarrow$ es bleibt ein Prozeß als unmarkiert übrig.

Beispiel:

CD-ROM

Plotter

E=(4, 2, 3, 1)

Bandgerät

Drucker

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \end{pmatrix}$$

Einziges i mit $R_i \leq A$ ist $i = 3$

$$\begin{aligned} \rightarrow \quad A &= \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix} \quad \text{dann } i = 2 \\ \rightarrow \quad A &= \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \quad \text{dann } i = 1 \\ \rightarrow \quad A &= \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \end{aligned}$$

Die gleiche Situation, aber mit $R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 \end{pmatrix}$ führt zur Verklemmung.

Deadlock-Behebung

- Unterbrechung; das heißt ein Prozeß wird unterbrochen, seine Betriebsmittel werden ihm entzogen.
- teilweise Wiederholung. Nach der Unterbrechung führt der Prozeß Teile seines Ablaufs vor der Unterbrechung erneut aus.
- Abbruch; ein Prozeß wird unterbrochen und anschließend komplett neu gestartet.

5.4.3 Deadlock-Vermeidung durch vorsichtige Betriebsmittelzuteilung

Wir unterscheiden zwischen sicheren und unsicheren Zuständen *.

*gegeben durch die Betriebsmittel-Belegungsmatrix

Def.: Ein Zustand heißt sicher, wenn kein Deadlock vorliegt und alle verbleibenden Betriebsmittelanforderungen erfüllt werden können.

Beispiel:

nur eine Klasse von Betriebsmitteln.

$$E = (10) \quad , \quad C = \begin{pmatrix} 3 \\ 2 \\ 2 \end{pmatrix} \quad , \quad A = (3) \quad , \quad R = \begin{pmatrix} 6 \\ 2 \\ 5 \end{pmatrix} \quad , \quad \text{sicher}$$

$$E = (10) \quad , \quad C = \begin{pmatrix} 4 \\ 2 \\ 2 \end{pmatrix} \quad , \quad A = (2) \quad , \quad R = \begin{pmatrix} 5 \\ 2 \\ 5 \end{pmatrix} \quad , \quad \text{unsicher}$$

Bemerkung

Unsicher heißt nicht unbedingt, daß ein Deadlock entstehen muß.

5.4.4 Systematische Deadlock-Vermeidung

1. Wechselseitiger Ausschluß: Spooling
2. Anforderung *weiterer* Betriebsmittel:
   - alle Betriebsmittel müssen am Anfang angefordert werden (insbesonders im Batch-Betrieb)
   - bei Anforderung weiterer Betriebsmittel müssen zuerst bereits belegte Betriebsmittel freigegeben werden.
3. Nichtunterbrechbar: äußerst problematisch
4. Zyklische Wartebedingung: die Betriebsmittel werden numerisch geordnet
   a) Anforderungen durch einen Prozeß nur in aufsteigender Reihenfolge bezüglich der Nummerierung
      $n_0 < n_1 < n_2 < \ldots < n_r \Rightarrow$ im Betriebsmittelgraphen kann es keinen Zyklus geben.
   b) Die Nummer jedes neu angeforderten Betriebsmittels muß größer sein als die maximale Nummer der aktuell belegten Betriebsmittel.

6. Dateisysteme

6.1 Einleitung

- Speicherung von Informationsmengen
- Speicherung über die Terminierung der Prozesse hinaus.
- „Gleichzeitiger“ Zugriff durch mehrere Prozesse.

6.2 Grundlagen

6.2.1 Dateinamen

Offensichtlich dienen Dateinamen dazu die unter diesem Namen abgelegten Daten wiederzufinden. Der Dateiname sollte daher eine Aussage über den darunter abgelegten Inhalt zulassen. Dabei tritt aber zwangsläufig das Problem der Eindeutigkeit auf.

Eine gängige Methode den Informationsgehalt des Dateinamens zu erhöhen ist die Verwendung von Erweiterungen. Mittels dieser Erweiterungen ist es möglich die Datei einer bestimmten Klasse zuzuordnen. Moderne Dateisysteme erkennen die Dateiklasse anhand der Struktur der in ihr enthaltenen Daten.

Beispiele für gänige Dateinamenerweiterungen:

Extension	Dateiart
.c	C-Quellcode
.mi	MI-Quellcode
.h	Header-Dateien
.tex	TeX- bzw. LaTeX-Dateien
.sty	Style-Datei für LaTeX
.dvi	Geräteunabhänige Druckdatei von TeX/LaTeX erzeugt
.ps	Postscript-Datei

6.2.2 Dateistruktur

a) Folge von Bytes

b) Folge von Blöcken (vgl. Lochkarten, Ausgabedateien für Zeilendrucker)

c) Folge von Records

record:	length	key	contents

6.2.3 Dateitypen

- gewöhnliche Dateien (ASCII, binär)
- spezielle zeichen-/blockorientierte Dateien
  (für E/A-Geräte, z.B. Drucker, Festplatte,...)

Kennzeichnung des Inhalts:

a) Erweiterung des Dateinamens

b) Verwendung einer „magic number" am Anfang der Datei (z.B. `MZ` für exe, `II` für .tiff, `%!` für Postscript)

Weitere Beispiele:

magic #
Textgröße
Dateigröße
BSS-Größe
Größe d. Symboltabelle
Einstiegsadresse
Flags
Text
Daten
Relokations- information
Symboltabelle

6.2.4 Dateizugriff

Grundsätzlich kann man zwischen zwei Arten des Dateizugriffs unterscheiden. Den sequentiellen und den wahlfreien Zugriff. Beim sequentiellen Zugriff können Dateien nur Byte für Byte gelesen und beschrieben werden. Diese Art des Zugriffs ist vor allem bei Bandlaufwerken von Bedeutung, da sich bei diesen bauartbedingt ein wahlfreier Zugriff kaum sinnvoll implementieren läßt. Beim wahlfreien Zugriff können über einen Cursor an jeder belibiegen Stelle der Datei Lese- und Schreiboperationen ausgeführt werden. Diese Art des Zugriffs ist zum Beispiel bei Festplatten möglich.

6.2.5 Dateioperationen

Operation	Beschreibung
create	Datei erzeugen
write	Daten schreiben
delete	Datei löschen
append	Datei erweitern
open	Datei öffnen
seek	Cursor in der Datei positionieren
close	Datei schließen
get/set attributes	lesen/setzen der Dateiattribute
read	Daten lesen
rename	Datei umbennen

6.2.6 Dateiattribute

Attribut	Beschreibung
Schutz	wer darf zugreifen
Paßwort	nötig für Zugriff
Erzeuger	wer hat die Datei geschrieben
Eigentümer	wem gehört die Datei
Read-only-Flag	Schreibsperre
Hidden-Flag	Datei verstecken
System-Flag	Systemdateien
Archive-Flag	Archivierbit
ASCII/Binary-Flag	ASCII-Datei oder nicht
Random-Access-Flag	wahlfreier Zugriff
Temp-Flag	Datei temporär
Lock-Bit	Datei gesperrt
Record-Length	Länge der Records bzw. Blöcke
Location-Key	Position des Schlüssels im Record
Length-Key	Länge des Schlüssels im Record
Erzeugungsdatum	
Zugriffsdatum	Zeitpunkt des letzten Zugriffs
Modifikationsdatum	Zeitpunkt des letzten schreibenden Zugriffs
Größe	Länge der Datei

6.2.7 Beispielprogramm

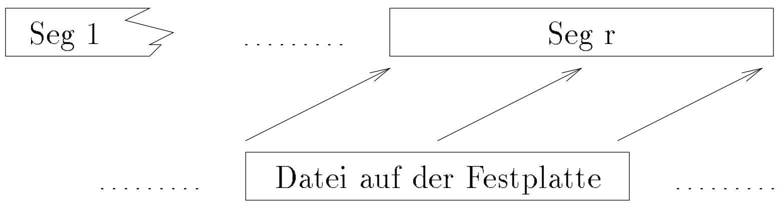
Implementierung eines einfachen Dateikopierbefehles:

`cp ⟨source⟩ ⟨dest⟩`

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 4096
#define MODE 0744 /*Zugriffsrechte: rwxr--r--*/

void main(int argc, char *argv[])
{
  int src, dst, in, out;
  char buf[BUF_SIZE];

  if (argc != 3) exit(1); /* falsche Anzahl von Parametern */
  /* Quelle öffnen und Zieldatei erzeugen */
```

```
  src = open(argv[1], O_RDONLY); /* Datei zum Lesen öffnen */
  if (src < 0) exit(2); /* Datei konnte nicht geöffnet werden */
  dst = open(argv[2], MODE);
                /* Datei mit angegebenen Zugriffsrechten erzeugen */
  if (dst < 0) exit(3); /* Datei konnte nicht erzeugt werden */
  /* Kopieren */
  while (1) {
    in = read(src, buf, BUF_SIZE);
    if (in <= 0) break; /* die Datei src ist leer */
    out = write(dst, buf, in);
    if (out <= 0) break; /* oder exit(4) */
    }/* kopiert Quelldatei in Stücken von 4K in Zieldatei */
  close(src); close(dst);
  exit(0);
}
```

6.2.8 Speicherabgebildete Dateien



Probleme:

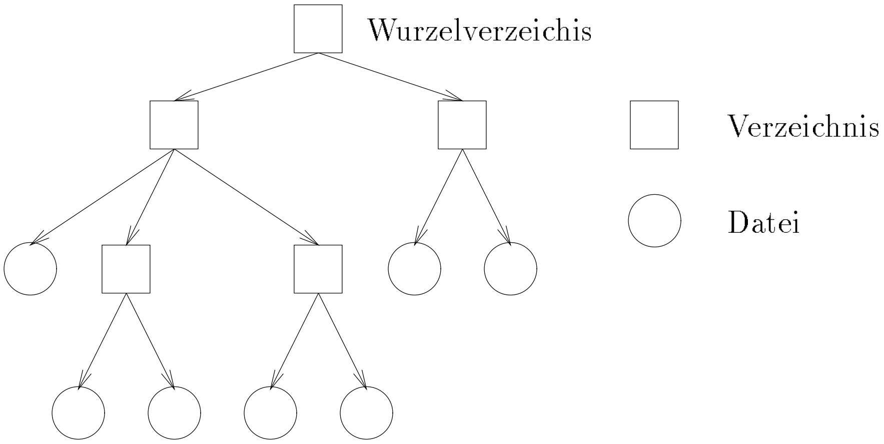
a) Exakte Länge der Datei

b) Kohärenz bei gleichzeitigem Zugriff durch mehrere Benutzer.

c) Dateilänge > Segment. Bei der in der Vorlesung benutzten Maschine MI ergeben sich hier keine Probleme. Bei Intel 80x86 entstehen bedingt durch das Paging Probleme.

6.3 Verzeichnisse

6.3.1 Verzeichnishierarchie



6.3.2 Pfadnamen

Man unterscheidet zwischen zwei Arten von Pfadnamen:

- relative Pfadnamen: ausgehend vom aktuellen Verzeichnis
- absolute Pfadnamen: ausgehend vom Wurzelverzeichnis

6.3.3 Operationen

Operation	Beschreibung
createdir	Verzeichnis erzeugen
deletedir	Verzeichnis löschen
opendir	Verzeichnis öffnen
closedir	Verzeichnis schließen
readdir	Verzeichnis lesen
link	einen Link auf ein Verzeichnis setzen
unlink	einen Link löschen

6.4 Implementierung

Bei der Implemetierung eines Dateisystems gilt es in erster Linie, einen möglichst guten Kompromiß zwischen den beiden konkurrierenden Zielen „hohe Flexibilität“ und „kurze

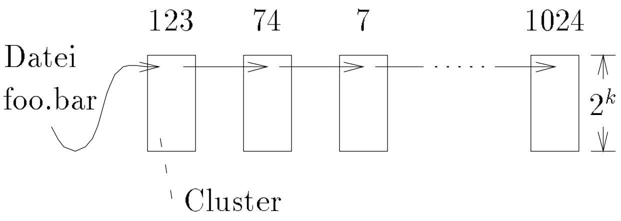
Zugriffszeiten“ zu finden.

6.4.1 Kontinuierliche Allokation

Bei der kontinuierlichen Allokation wird für eine Datei entsprechend ihrer Größe ein fortlaufender Speicherbereich reserviert. Da diese Methode einen großen Verschnitt verursacht, wird sie in der Praxis nicht eingesetzt.

6.4.2 Zuweisung von Speicher aus in einer Liste verknüpfter Cluster

Analog zur Hauptspeicherverwaltung kann man auch den Hintergrundspeicher in „Seiten“ konstanter Größe unterteilen, die hier *Cluster* genannt werden.



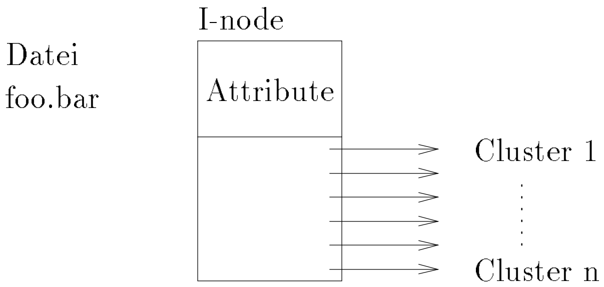
Ein Problem bei diesem Verfahren liegt darin begründet, daß wegen der hier notwendigen Verweisinformation der nutzbare Platz von „ungerader“ Größe ist ($< 2^k$). Da die Nutzdaten häufig in 2^k-Stücken vorliegen, entsteht unnötiger Verschnitt. Abhilfe schafft hier die Verwendung einer getrennten Liste (Tabelle), welche die Verküpfungsinformationen enthält. Man bezeichnet eine solche Liste als *File Allocation Table* (FAT). Bei kleiner Clusterzahl is dieses Verfahren sehr schnell, da die FAT im Speicher gehalten werden kann. Bei großer Clusterzahl wird die FAT jedoch „unhandlich“ groß.

Beispiel für eine FAT:

0	frei	
1	frei	
2	10	
3		
4	7	Datei foo.a beginnt mit Cluster 4
5		
6		
7	2	
8		
9		
10	12	
11		
12		

6.4.3 I-nodes (Index-Knoten)

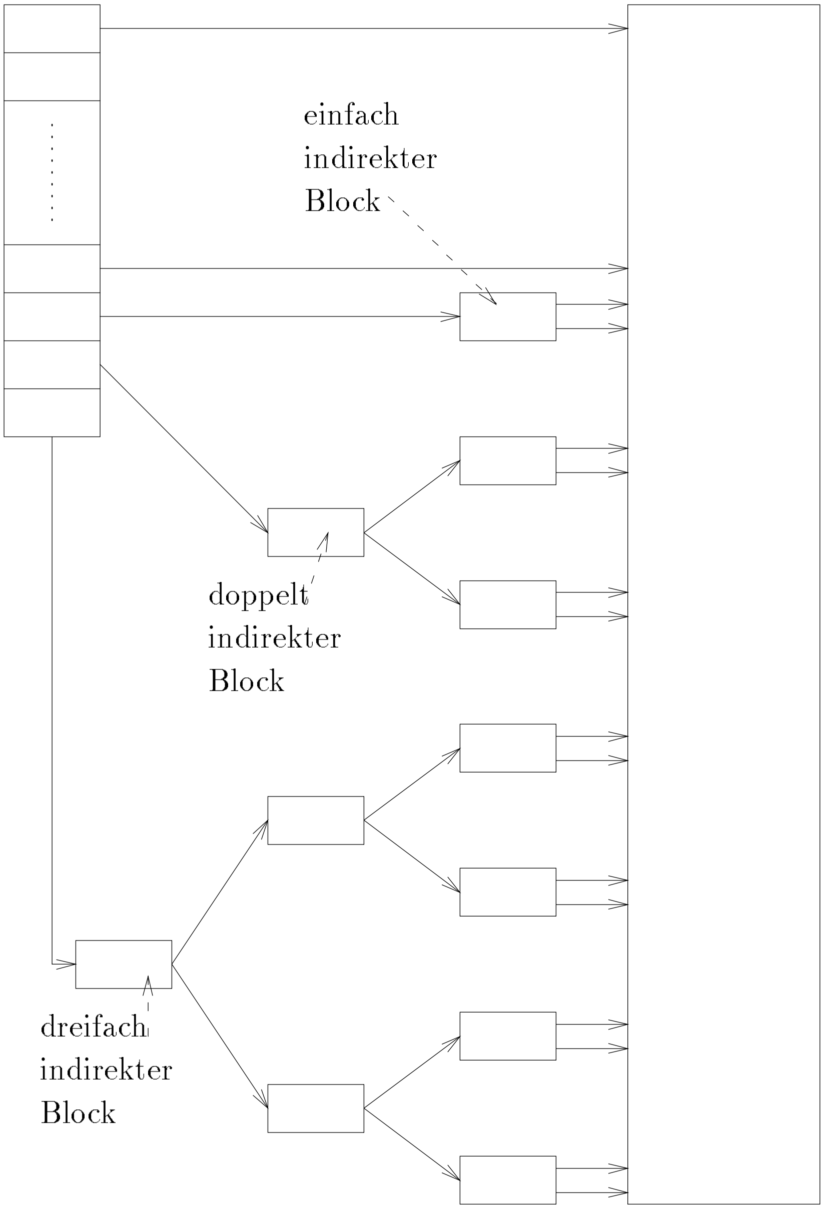
I-nodes sind eine Mischform zwischen Baum und Liste.



Sie bilden die Basis des Dateisystems von Unix. In ihnen sind alle wesentlichen Informationen, bis auf den Dateinamen, einer Datei gesammelt.

```
struct INode {
  u_short   ino_mode;       /* Dateityp und Zugriffsrechte */
  short     ino_nlink;      /* Anzahl der Namen der Datei */
  uid_t     ino_uid;        /* Benutzernummer des Dateieigentümers */
  gid_t     ino_gid;        /* Gruppennummer des Dateieigentümers */
  off_t     ino_size;       /* Größe der Datei in Byte */
  time_t    ino_atime;      /* Zeit des letzten Lesezugirffs */
  long      ino_atspare;
  time_t    ino_mtime;      /* Zeit des letzten Schreibzugriffs */
  long      ino_mtspare;
  time_t    ino_ctime;      /* Zeit der letzten Satusänderung */
  long      ino_ctspare;
  daddr_t   ino_db[12];     /* Blocknummern der ersten 12 Datenblöcke*/
  daddr_t   ino_ib[3];      /* Blocknummern der 3 indirekten Datenblöcke*/
  long      ino_blocks;     /* Größe der Datei in Blöcken */
  long      ino_gen;        /* Generationsnummer (NFS) */
};
```

Unix legt für jedes Gerät, jede Datei und jedes Verzeichnis einen I-Node an. Da statistische Untersuchungen ergeben haben, daß ein großteil der Dateien relativ klein ist, versucht das Dateisystem, die ersten Cluster einer Datei im ersten I-Node zu speichern. Da in einem I-Node aber nur begrenzt Platz ist, ist diese Methode für große Dateien nicht praktikabel. Wächst eine Datei nun über die Größe, welche in einem I-Node gespeichert werden kann, hinaus, so belegt das Dateisystem einen neuen Cluster. Dieser Cluster wird als Speicher für Zeiger auf die Cluster welche die Daten der Datei enthalten verwendet.



6.4.4 Verzeichnisse

1. DOS
   DOS verwendet ein 32 Byte Record für jeden Verzeichniseintrag.

8	3	1	10	2	2	2	4
Name	Erw.	Attr	Reserv.	Uhrzeit	Datum	# 1. Cluster	Größe

   Das Wurzelverzeichnis ist ein Feld von begrenzter Größe, das Verzeichniseinträge enthält. Unterverzeichnisse werden wie Dateien gehandhabt. Dieses Verfahren ist zwar einfach zu implementieren, bedingt durch die Verwendung von Feldern fester Größe ist es jedoch nicht besonders flexibel.

6.5 Diskussion

Clustergröße

Das Mißverhältnis zwischen der Summe der mittleren Positionierungszeit (12 - 18 ms) und der mittlere Rotationsdauer (8 ms) zur Länge eines Maschinenbefehls spricht für die Verwendung großer Cluster. Je größer jedoch die Cluster gewält werden, desto größer wird der Verschnitt. Es liegt somit ein Tradeoff zwischen Performance und ökonomischer Ausnutzung des Plattenplatzes vor. Statistische Untersuchungen auf Unix ergaben eine durchschnittliche Dateigröße von 1kByte.

Eine Alternaive ist es, größere Cluster zu wählen, kleine Dateien aber zu großen zusammenzufassen und diese im Falle eines Zugriffs wieder auszupacken.

Verwaltung der freien Cluster

- Liste von freien Clustern.
- Bitmap von freien Clustern. In der Bitmap repräsentiert jedes Bit einen Cluster. Diese Strategie ist im Vergleich zur Liste wesentlich kompakter.

Verwaltung fehlerhafter Cluster

Fehlerhafte Cluster werden mit einer Datei belegt, welche nicht gelöscht und nicht gelesen werden kann. Eine Alternative hierzu ist die Verwendung von Reservezylindern auf der Platte. Der Controller oder, wie bei SCSI-Festplatten, die Platte selbst, hält sich eine Liste der fehlerhaften Cluster und ersetzt diese durch Reservecluster.

Backups

7. Maschinenmodelle, Formale Sprachen

7.1 Motivation

Bereits betrachtet wurde die MI. Sie könnte Grundlage für ein formales Modell sein, sie besitzt aber Beschränkungen wie, endlichen Speicher, feste endliche Wortlängen und keine formale vollständige Spezifikation. Um maschinelles Rechnen genauer untersuchen zu können sind formale Modelle notwendig.

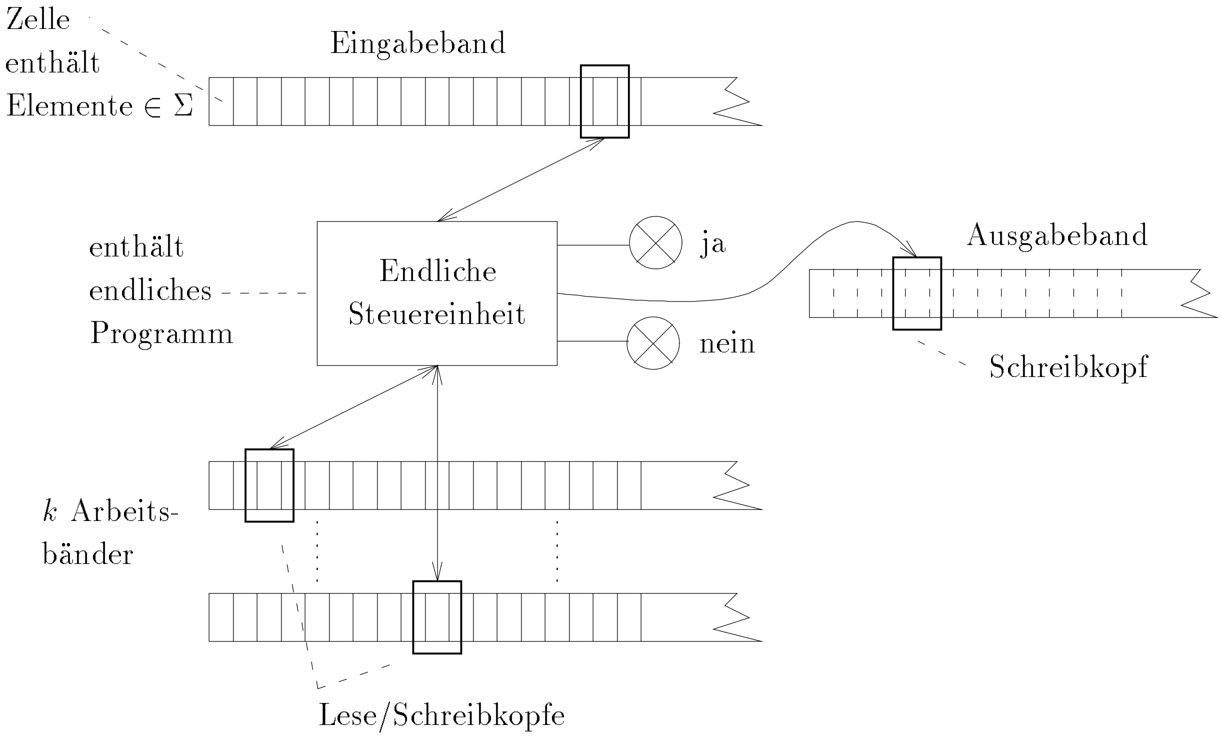
Zwei Vorgehensweisen:

1. Nachweis der Möglichkeit:
   Zur Lösung eines Problems wird ein Algorithmus gefunden dessen Komplexität und Eigenschaften nachgewiesen werden können.
2. Nachweis der Unmöglichkeit:
   Es ist vielfach ebenso interessant zu zeigen, daß es für bestimmte Probleme keinen Algorithmus gibt. Wir benötigen einen Formalismus (bzw. Modell), mit dem wir auch solches nachweisen können.

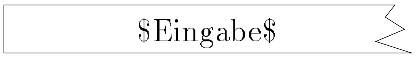
7.2 Die (Mehrband-)Turing-Maschine

Ein formales Modell eines Algorithmus sollte bestimmte Eigenschaften haben: Erstens sollte jeder Algorithmus endlich beschreibbar sein. Zweitens sollte der Algorithmus aus diskreten Schritten bestehen, von denen jeder maschinell ausführbar ist. Ein solches Modell wurde 1936 von Alan Turing vorgestellt.

Das folgende Bild zeigt eine (Mehrband-/ K-Band-)Turing-Maschine:



- Die Bänder (Eingabeband, Ausgabeband und die k Arbeitsbänder) sind in Zellen unterteilt, wobei jede Zelle genau ein Element aus einer endlichen Menge von Bandsybolen enthalten kann. Jedes Band hat ein am weitesten links stehendes Feld, ist aber unendlich zur rechten Seite.
- Die Eingabe auf dem Eingabeband wird aus den Elementen eines Alphabets Σ zusammengesetzt und durch \$-Zeichen $\in \Sigma$ umschlossen.



- Zusätzlich gibt es blank Zeichen $\not{b} \in \Sigma$ oder $\sqcup \in \Sigma$ um unbeschriebene Zellen anzuzeigen.
- Der Lesekopf kann in jedem Schritt um eine Zelle nach links oder rechts bewegt werden. Es kann immer nur ein Zeichen gelesen werden, nämlich das Zeichen in der Zelle, die sich gerade unter dem Lesekopf befindet. Der Kopf kann während einer Rechnung beliebig weit nach rechts wandern.
- Das Alphabet der Arbeitsbänder sei Σ' mit $\not{b} \in \Sigma'$.
- Arbeitsbänder können beschrieben oder gelesen werden.
- Das Alphabet des Ausgabebandes sei Γ mit $\not{b} \in \Gamma$
- Auf das Ausgabeband kann nur geschrieben werden, das heißt der Ausgabekopf bewegt sich nur nach rechts.

Um eine Funktion zu berechnen, werden Argumente auf das Eingabeband gelegt. Nach der Berechnung schreibt die Maschine das Resultat auf das Ausgabeband. Manchmal reicht es aus, als Resultat nur „ja“ oder „nein“ zu erhalten. Deshalb besitzt die Maschine zusätzlich zwei Lämpchen (rot und grün) welche von der Maschine am Ende einer Berechnung aktiviert werden können.

Ablauf eines Arbeitschrittes:

1. Lesen des Zeichens am Eingabekopf und der Zeichen unter den Arbeitsköpfen.
2. Schreiben auf die Arbeitsbänder und das Ausgabeband.
3. Bewegen der Köpfe, jeweils nach rechts oder links, (Ausgabekopf nur nach rechts) oder nicht bewegen.

Formale Definition:

Eingabealphabet	Σ	
Arbeitsalphabet	Σ'	k Arbeitsbänder
Ausgabealphabet	Γ	
Zustandsmenge	Q	
Übergangsfunktion	$f:$	$Q \times \Sigma \times \Sigma'^k \longrightarrow Q \times \{r,l,n\} \times (\Sigma' \times \{r,l,n\})^k \times (\Gamma \cup \{\varepsilon\})$

r	$\sim$	Kopf nach rechts
l	$\sim$	Kopf nach links
n	$\sim$	Kopf nicht bewegen
ε	$\sim$	keine Ausgabe auf das Ausgabeband

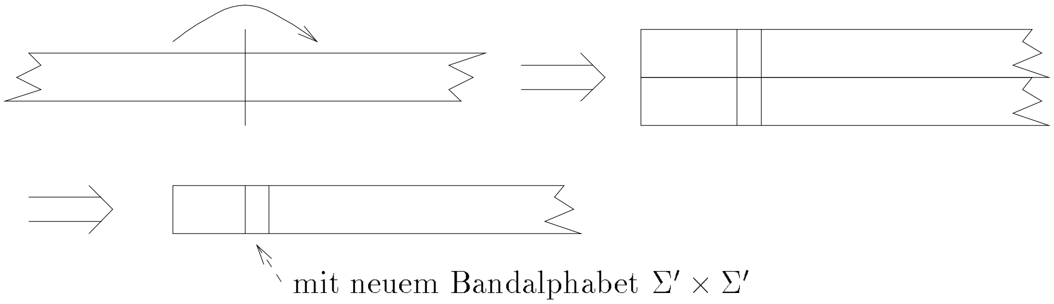
akzeptierende Endzustände	$F_a \subseteq Q$
nicht-akzeptierende Endzustände	$F_r \subseteq Q$
(verwerfend, zurückweisend)	

mit $F_a \cap F_r = \emptyset$

Erreicht die Maschine einen Zustand aus F_a so leuchtet das „ja“-Lämpchen, erreicht sie einen Zustand aus F_r leuchtet das „nein“-Lämpchen. In beiden Fällen hält die Maschine an.

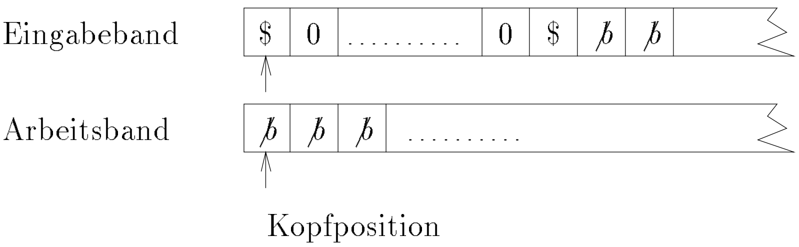
Häufig verwendete Varianten der Turingmaschine:

- In der Literatur wird oft ein Modell mit beidseitig unendlichen Arbeitsbändern verwendet. Beide Modelle sind jedoch gleichwertig.



- Eine weitere Variante ist die Verwendung mehrdimensionaler Bänder. In Bezug auf die Berechenbarkeit leisten solche Turingmaschinen aber auch nicht mehr. Ein Unterschied besteht jedoch in der Geschwindigkeit.

Beispiel:



Übergangsfunktion:

$$
\begin{array}{lll}
\delta: & q_0\$\not b & q_1 n\$r \\
& q_1\$\not b & q_1 r0n \\
& q_1 0* & q_1 r*n \\
& q_1\$* & q_f n*n \\
& q_1 1* & q_2 n*n \\
& q_2 10 & q_3 n1l \\
& q_2 11 & q_2 n1r \\
& q_2 1\not b & q_3 n1l \\
& q_3 1* & q_3 n*l \\
& q_3 1\$ & q_1 r\$r
\end{array}
\qquad \text{mit} * \in \{0,1\}
$$

Anfangszustand q_0, Endzustand q_f

Quiz: Was wird auf dem Arbeitsband erzeugt für Eingaben $\$\{0,1\}^*\$\ \not b \ldots$?

Lösung: Die Maschine zählt die Anzahl der Einsen in der Eingabe und legt die Anzahl als Binärzahl ab, wobei das niederwertigste Bit ganz links steht.

Definition 7.2.1 *Sei Σ ein endliches Alphabet, M eine Turing-Maschine (TM).*

a) Eine (Formale) Sprache L ist eine Teilmenge von Σ^*.

b) M akzeptiert *L wenn gilt:*
$\forall x \in L$: *M hält auf der Eingabe x in einem akzeptierenden Endzustand.*
$\forall x \notin L$: *M hält auf der Eingabe x* nicht *in einem akzeptierenden Endzustand.*

c) M erkennt *L wenn gilt:*
$\forall x \in L$: *M hält bei Eingabe x in einem akzeptierenden Endzustand.*
$\forall x \notin L$: *M hält bei Eingabe x in einem* nicht-akzeptierenden *Endzustand.*[*]

d) $L(M)$ ist die von M akzeptierte Sprache.

Der momentane Zustand der Turing-Maschine heißt *Konfiguration*:

- Zustand
- Inhalt der Arbeitsbänder + Kopfposition der Arbeitsbänder
- Kopfposition des Eingabebandes

Eine *Rechnung* einer TM ist eine Folge von Konfigurationen $C_0, C_1, C_2, \ldots$ wobei gilt:

- C_0 ist Anfangskonfiguration
- C_i geht in einem Schritt aus C_{i-1} hervor, für $i > 1$.

Eine *terminierende Rechnung* ist eine endliche Folge von Konfigurationen.

Bisher:

Übergang wurde durch eine Funktion beschrieben. Ersetzt man δ durch eine Übergangs*relation* so erhält man eine nichtdeterministische Turing-Maschine. Jede Konfiguration kann sich in eine endliche Menge von Konfigurationen aufspalten.

Definition 7.2.2 *Sei M eine (deterministische) k-Band Turingmaschine, x eine Eingabe für M.*

a) Der Zeitbedarf $T(x)$ von M auf Eingabe x ist die Anzahl der Schritte, die M bis zum Halten ausführt.

---

[*] **Unterschied:**

- Wenn M L erkennt, dann hält die Maschine in jedem Fall, entweder mit leuchtender „ja“ oder „nein“ Lampe.
- Wenn M L nur akzeptiert, so kann M auch unendlich laufen, wir wissen nicht, ob M doch noch anhält.

b) Unter der (worst-case)-Zeitkomplexität von M auf Eingaben der Länge n ($n \in \mathbb{N}$) verstehen wir

$$T(n) := \max_{x \in \Sigma^n}\{T(x)\}$$

c) Unter der durchschnittlichen (average-case) Zeitkomplexität von M verstehen wir

$$T_{av}(n) := \frac{1}{|\Sigma|^n} \sum_{x \in \Sigma^n} T(x)$$

d) Unter dem Platzbedarf $S(x)$ der TM auf der Eingabe x verstehen wir die Anzahl der Zellen auf den Arbeitsbändern, die während der Rechnung von M auf x besucht werden.[†‡]

$$\textit{(worst-case) Platzbedarf} \quad S(n) := \max_{x \in \Sigma^n}\{S(x)\}$$

$$\textit{durchschnittlicher (average-case) Platzbedarf} \quad S_{av}(n) := \frac{1}{|\Sigma|^n} \sum_{x \in \Sigma^n} S(x)$$

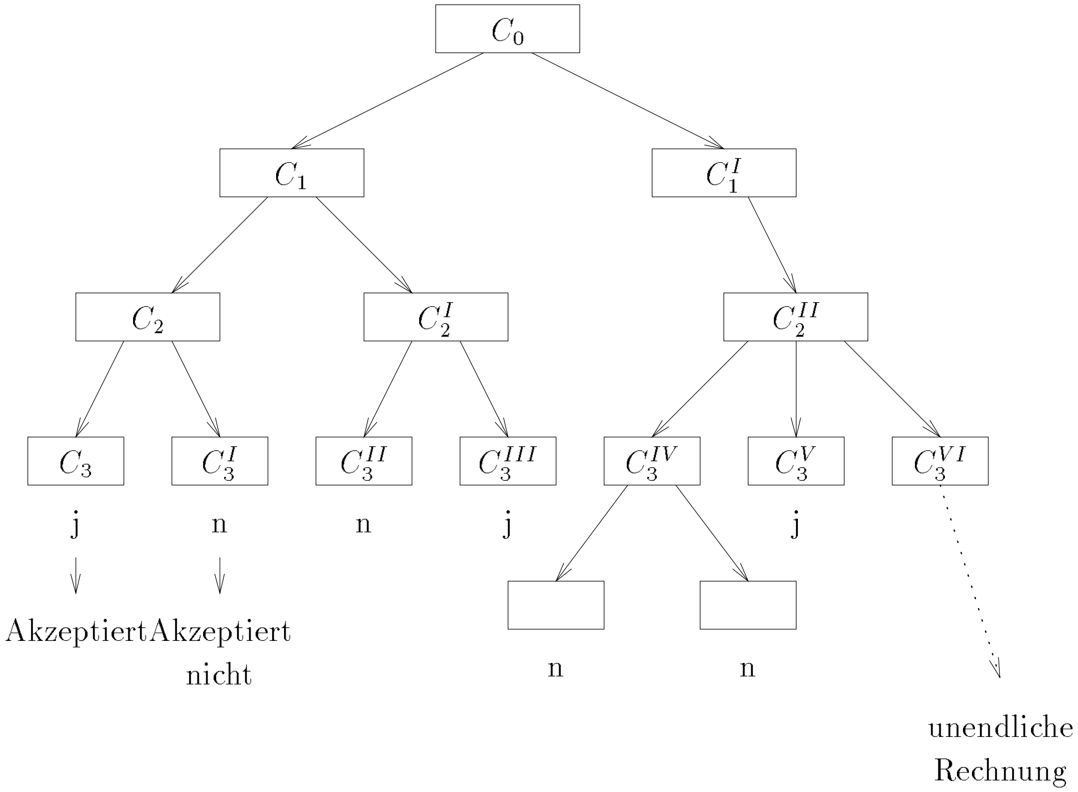
Anmerkung zum Definitionsbereich der Übergangsfunktion δ:

δ muß nicht unbedingt total sein. Man stellt sich vor, daß alle nicht definierten Übergänge in einen Fangzustand münden, den die TM nicht mehr verläßt.

7.3 Nicht-deterministische Rechnung

Die Übergangs*funktion* δ wird durch eine Übergangs*relation* δ ersetzt. δ ist nach wie vor endlich. Für eine gegebene Konfiguration hat die nicht-deterministische Turing-Maschine eine endliche Zahl von nächsten Konfigurationen zur Auswahl. Nach wie vor startet die TM in *einem* Anfangszustand. Zur Darstellung aller möglichen Konfigurationsübergänge einer nicht-deterministischen TM N dient der sog. *Berechnungsbaum*:

[†] Es werden nur Bandzellen auf Arbeitsbändern gezählt, nicht die auf den Eingabe- und Ausgabeband!
[‡] Eine Zelle muß nicht unbedingt beschrieben werden um als Platzbedarf zu gelten



Hat die nicht-deterministische TM mehrere Möglichkeiten weiterzuarbeiten, so stellen wir uns vor, daß sie so oft kopiert wird wie nötig, um alle Möglichkeiten „gleichzeitig" weiterzuverfolgen.

Definition 7.3.1 *Eine nicht-deterministische Turing-Maschine N* akzeptiert *die Eingabe x genau dann, wenn es im Berechnungsbaum (von N) für x ein Blatt gibt, das einer akzeptierenden Endkonfiguration von N entspricht.*

$L(N)$:= Menge der von N akzeptierten Eingaben
$L(N)$ ist die von N akzeptierte Sprache.

Wichtig:

Eine nicht-deterministische Maschine N erkennt ihre Sprache nicht sondern *akzeptiert* sie, im Unterschied zur deterministischen TM. Der Begriff des „Erkennens" macht bei nicht-deterministischen Maschinen keinen Sinn mehr.

Beispiel:

Was passiert wenn man die Ergebnislampen „ja", „nein" vertauscht?

Bei einer deterministischen TM welche L *erkennt*, wird die TM mit den vertauschten Lampen $\overline{L}$ erkennen (das Komplement von L).

Wird von einer nicht-deterministischen TM N ein Wort x *akzeptiert*, so kann es von der TM mit vertauschten Lampen immer noch akzeptiert werden, nämlich genau dann wenn es für die Eingabe x im Berechnungsbaum von N ein Blatt mit nicht-akzeptierender Endkonfiguration gibt. Durch die Vertauschung der Lampen entspricht dieses Blatt nun einer akzeptierenden Endkonfiguration.

Definition 7.3.2 *Sei N eine nicht-deterministische Turing-Maschine, x eine Eingabe für N.*

$$\textit{Zeitbedarf von } N \textit{ auf } x := \begin{cases} \textit{Länge eines kürzesten akzeptierenden Pfades für } x \in L(N) \\ 0 \quad \textit{sonst} \quad \textit{(falls kein akzeptierender Pfad existiert)} \end{cases}$$

Durchschnittlicher Zeitbedarf: wie bei der deterministischen TM.

$$\textit{Platzbedarf} := \begin{cases} \textit{minimaler Platzbedarf eines akzeptierenden Pfades für } x \in L(N) \\ 0 \quad \textit{sonst} \end{cases}$$

Durchschnittlicher Platzbedarf: wie bei der deterministischen TM.

7.4 Rekursiv aufzählbare Sprachen

Definition 7.4.1 *Eine Sprache L heißt rekursiv aufzählbar genau dann, wenn es eine nicht-deterministische TM N gibt mit*

$$L = L(N)$$

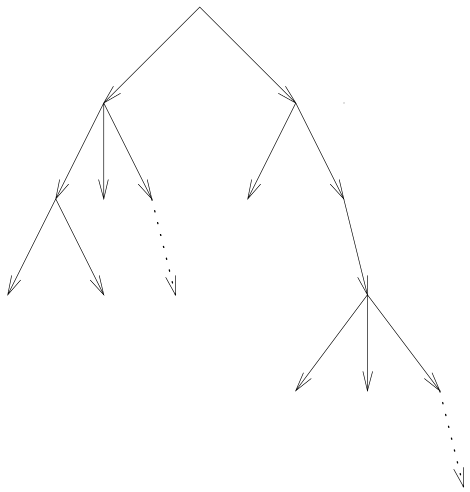
§ *Man bezeichnet rekursiv aufzählbare Sprachen auch als Chomsky-0 Sprachen.*

Satz 3 *L sei rekursiv aufzählbar $\Rightarrow$ es gibt eine deterministische TM M mit $L = L(M)$*

Beweis: Berechnungsbaum von N auf x ¶

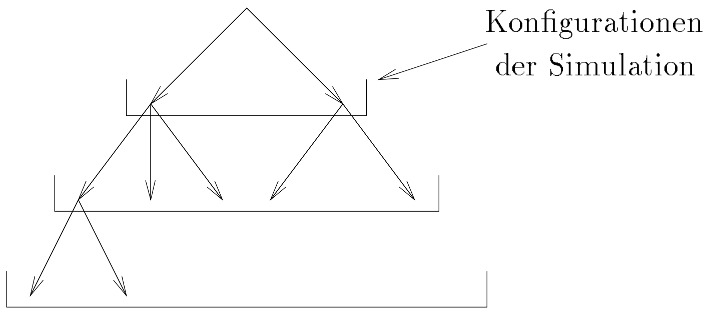
§rekursiv aufzählbar deshalb, weil es eine nicht-deterministische TM gibt, die die Sprache aufzählen kann.

¶Der Verzweigungsgrad könnte immer auf zwei beschränkt werden.



Prinzip: simuliere die nicht-deterministische TM durch eine deterministische TM.

Der Berechnungsbaum wird von der deterministischen TM abgelaufen, solange bis „j“ gefunden wird. Allerdings nicht im Depth-First-Verfahren, wegen der Gefahr in endlosen Pfaden stecken zu bleiben, sondern im Breadth-First-Verfahren (BFS).



Akzeptiere, wenn im Berechnungsbaum akzeptierendes Blatt gefunden wird.

$$L(M) = L(N)$$

$$x \in L(M) \Rightarrow x \in L(N)$$

$$x \in L(N) \Rightarrow x \in L(M)$$

∎

Die deterministische Simulation benötigt exponetiell mehr Zeit als die nicht-deterministische TM.

Platz und Zeitbedarf einer TM

Satz 4 *Sei $S(n) \geq \log n$, sei M eine (nicht-deterministische) K-Band-TM, die auf x hält und $S(|x|)$-platzbeschränkt ist. Dann gibt es ein $c > 0$, so daß M auf x $2^{c \cdot S(|x|)}$ zeitbeschränkt ist.*

Beweis: Anzahl der Konfigurationen von M mit $n = |x|$

$$\begin{aligned} &\leq |\Sigma'|^{S(n)} \cdot S(n)^k \cdot n \cdot |Q| \\ &\leq 2^{c \cdot S(n)} \end{aligned}$$

Wir nehmen an, daß die TM keine Konfiguration wiederholt. In diesem Fall nämlich hätte die TM eine unnötige Schleife gefahren.

$\Rightarrow$ die TM hat höchstens einen exponentiell höheren Zeit- als Platzbedarf. ∎

Satz 5 $T(n) \geq \Omega(S(n))$

Beweis: trivial. ∎

Also: $S(n) \leq T(n) \leq 2^{c \cdot S(n)}$ (mit $S(n) \geq \log n$) ist das einzige, was wir im Allgemeinen über die Laufzeit wissen.

Wir wissen auch bereits:

$$T_{nicht-det}(n) \leq T_{det}(n) \leq 2^{c \cdot T_{nicht-det}(n)}$$

$\Rightarrow$ Exponentieller Zwischenraum zwischen Zeit und Platz, sowie zwischen Determinismus und Nichtdeterminismus („exponential gap").

A. Eine kurze Einführung in die Grundlagen der Graphentheorie

A.1 Gerichtete Graphen

Definition : Ein gerichteter Graph $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Menge $E \subseteq V \times V$ von Kanten. *

Bemerkung : Die hier gegebene Definition läßt den Begriff der Mehrfachkante nicht zu. Eine mögliche Erweiterung in diesem Sinne wäre die Definition von E als Multimenge.

Definition :

a) Eine Kante der Form (v, v) (Anfangs- und Endpunkt sind identisch) heißt Schlinge
b) Ein Graph heißt einfach, wenn er keine mehrfachen Kanten † und keine Schlingen enthält.
c) Ein Pfad von v_0 nach v_r ist eine Folge von Knoten $v_0, \ldots, v_r$ mit $\{(v_i, v_{i+1}) | 0 \leq i < r\} \subseteq E$. Dabei heißt r die Länge des Pfades.
d) Ein Pfad $v_0, \ldots, v_r$ heißt einfach, wenn $\forall i, j : v_i = v_j \Rightarrow (i = j \vee (i = 0 \wedge j = r) \vee (i = r \wedge j = 0))$
e) Ein Kreis ist ein einfacher Pfad der Länge ≥ 1 mit identischen Anfangs- und Endpunkten.
f) $w \in V$ heißt von $v \in V$ aus erreichbar, wenn es einen Pfad von v nach w gibt.
g) Ein gerichteter Graph heißt stark zusammenhängend, wenn jeder Koten von jedem Knoten aus erreichbar ist.

A.2 Ungerichtete Graphen

Definition : Ein ungerichteter Graph $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Menge $E \subseteq \binom{V}{2} \cup \binom{V}{1}$ von Kanten, ($\binom{V}{n} := \{M \in 2^v | |M| = n\}$).

*Die Kante (v, w) wird dabei als von v nach w laufend interpretiert

†nach der hier gegebenen Definition gar nicht möglich, siehe Bemerkung.

Eine Kante $\{v,w\} \in \binom{V}{2}$ verbindet v mit w, eine Kante $\{v\} \in \binom{V}{1}$ ist eine Schlinge.

Definition : a) Ein Pfad von v_0 nach v_r bzw. von v_r nach v_0 ist eine Folge von Knoten $v_0, \ldots, v_r$ mit $\{\{v_i, v_{i+1}\} | 0 \leq i \leq r\} \subseteq E$. Dabei heißt r die Länge des Pfades.

b) Ein ungerichteter Graph heißt zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.

Definition : Sei $G = (V, E)$ ein gerichteter Graph. Dann ist $\tilde{G} = (V, \{\{v,w\} | (v,w) \in E\})$ der zugehörige ungerichtete Graph.

Definition : a) Ein ungerichteter Graph $G = (V, E)$ heißt bipartit, wenn
$\exists V_1, V_2 \subseteq V : V_1 \cap V_2 = \emptyset \wedge V_1 \cup V_2 = V \quad \wedge$
$E \cap \binom{V_1}{2} = E \cap \binom{V_2}{2} = E \cap \binom{V}{1} = \emptyset$

b) Ein gerichteter Graph heißt bipartit, wenn sein zugehöriger ungerichteter Graph bipartit ist.

Satz : Ein ungerichteter Graph ist genau dann bipartit, wenn er keinen Kreis ungerader Länge enthält.

Beweis : als Übung.

A.3 Bäume (ungerichtete Variante)

Definition : Ein Baum ist ein ungerichteter, zusammenhängender Graph, welcher keine Kreise enthält.

Lemma : Ein Baum ist ein einfacher Graph.

Satz : In einem Baum ist die Anzahl der Kanten genau um eins kleiner als die Anzahl der Knoten.

Beweis : als Übung.

Definition : Sei $G = (V, E)$ ein ungerichteter Graph. Der Graph $T = (V, \tilde{E})$ heißt Spannbaum von G, wenn er ein Baum ist und $\tilde{E} \subseteq E$.

Satz : Ein Graph hat genau dann einen Spannbaum, wenn er zusammenhängend ist.

Beweis : als Übung.