Softwareentwicklung und Betriebssysteme

Summer School Dritteinsteiger

FH-Prof. Dr. Fabian Knirsch

8.-18. Juli 2019

Intro

Über

Fabian Knirsch

fabian.knirsch@fh-salzburg.ac.at
Raum 421
www.en-trust.at/knirsch

Senior Lecturer
Studiengang Informationstechnik & Systemmanagement
Zentrum für sichere Energieinformatik

Benutzung von Laborrechnern

Benutzername: fh-gast
Kennwort: Welcome@FHS!

Achtung: Das Benutzerprofile wird nach dem Abmelden zurückgesetzt. Speichern Sie alle Daten auf einem externen Speicher!

Lehrveranstaltungsübersicht I

Prüfungsmodus

1. 15 Einheiten VL und LB, 2 Einheit Klausur
2. Anwesenheit
3. Übungsaufgaben
4. Abschließende Prüfung (90 min)
5. Folien und **eigene Mitschrift**

Erwartungen

1. selbstständiges Erarbeiten von Inhalten
2. aktive Mitarbeit
3. Diskussion

Lehrveranstaltungsübersicht I

Prüfungsmodus

1. 15 Einheiten VL und LB, 2 Einheit Klausur
2. Anwesenheit
3. Übungsaufgaben
4. Abschließende Prüfung (90 min)
5. Folien und **eigene Mitschrift**

Erwartungen

1. selbstständiges Erarbeiten von Inhalten
2. aktive Mitarbeit
3. Diskussion

Lehrveranstaltungsübersicht II

Datum	#Einheiten	Inhalte
8.7.2019	3	VL: Intro, Formale Sprachen, Automaten, Grundlagen C
11.7.2019	3	VL: Grundlagen C
12.7.2019	3	VL: AlgoDat, Komplexitätsbewertung, Betriebssysteme
16.7.2019	4	VL: Betriebssysteme
16.7.2019	2	LB: Präsentation der vorbereiteten Aufgaben
18.7.2019	2	KL

Tabelle: Organisation und Inhalte der Lehrveranstaltung

Voraussetzungen

1. Grundlagen C (Datentypen, Strukturen, Zeiger, C library)
2. Fähigkeit, in C komplexe Programme schreiben zu können
3. Algorithmen und Datenstrukturen (*-Sort, Heap, Stack, Tree)
4. Grundlagen Zustandsautomaten (Moore, Mealy)
5. Betriebssysteme (Aufgaben, Aufbau, Einteilung, Scheduling)

Referenzen

Literatur:

1. Wolf05: Wolf, J.: C von A bis Z – Das umfassende Handbuch, Galileo Computing, 2005[1]
2. Cormen03: Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2. Auflage, 1990, The MIT Press
3. Stallings03: Stallings, W.: Betriebssysteme, 4. Auflage, 2003, Pearson

Links:

1. `http://www.cplusplus.com/`
2. `http://www.en-trust.at/knirsch` (diese Folien)

[1] Als Open Book frei zugänglich `http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/`

Inhalt

1. Intro
2. Formale Sprachen
3. Automaten
4. Grundlagen Linux
5. Grundlagen C
6. Algorithmen und Datenstrukturen
7. Komplexitätsbewertung
8. Betriebssysteme

Formale Sprachen

Einführung

Gegeben ein Alphabet Σ, und die Kleenesche Hülle Σ^*: Eine formale Sprache λ ist die Teilmenge $\lambda \subseteq \Sigma^*$.

$$\Sigma = \{'\text{Hallo}', '\text{Welt}', ',', '\_', '!'\}$$

$$\Sigma^* = \{\epsilon, '\text{Hallo}', '\text{HalloWelt}', '\text{HalloWelt}', \ldots, '\text{Hallo, Welt!}', '\text{Hallo, Welt!}'\}$$

Gültiges Wort (λ): *Hallo, Welt!*

Weitere gültige Wörter: *HalloHallo*, *Welt!Hallo*, *!,,* ...

Einführung

Gegeben ein Alphabet Σ, und die Kleenesche Hülle Σ^*: Eine formale Sprache λ ist die Teilmenge $\lambda \subseteq \Sigma^*$.

$$\Sigma = \{'\text{Hallo}', '\text{Welt}', ',', '\_', '!'\}$$

$$\Sigma^* = \{\epsilon, '\text{Hallo}', '\text{HalloWelt}', '\text{HalloWelt}', \ldots, '\text{Hallo, Welt!}', '\text{Hallo, Welt!}'\}$$

Gültiges Wort (λ): *Hallo, Welt!*

Weitere gültige Wörter: *HalloHallo*, *Welt!Hallo*, *!,,* ...

Einführung

Gegeben ein Alphabet Σ, und die Kleenesche Hülle Σ^*: Eine formale Sprache λ ist die Teilmenge $\lambda \subseteq \Sigma^*$.

$$\Sigma = \{'\text{Hallo}', '\text{Welt}', ',', '\_', '!'\}$$

$$\Sigma^* = \{\epsilon, '\text{Hallo}', '\text{HalloWelt}', '\text{HalloWelt}', \ldots, '\text{Hallo, Welt!}', '\text{Hallo, Welt!}'\}$$

Gültiges Wort (λ): *Hallo, Welt!*

Weitere gültige Wörter: *HalloHallo*, *Welt!Hallo*, *!,,* . . .

Einführung

Gegeben ein Alphabet Σ, und die Kleenesche Hülle Σ^*: Eine formale Sprache λ ist die Teilmenge $\lambda \subseteq \Sigma^*$.

$$\Sigma = \{'\text{Hallo}', '\text{Welt}', ',', '\_', '!'\}$$

$$\Sigma^* = \{\epsilon, '\text{Hallo}', '\text{HalloWelt}', '\text{HalloWelt}', \ldots, '\text{Hallo, Welt!}', '\text{Hallo, Welt!}'\}$$

Gültiges Wort (λ): *Hallo, Welt!*

Weitere gültige Wörter: *HalloHallo*, *Welt!Hallo*, *!,*, ...

Formale Grammatik

Eine formale Grammatik ist ein 4-Tupel $G = \langle V, \Sigma, \Phi, \Sigma_0 \rangle$

- V Vokabular (endliche Menge)
- $\Sigma \subset V$ Alphabet mit Terminalsymbolen
- $\Phi \subset (V^* \backslash \Sigma^*) \times V^*$ Produktionsregeln
- $\Sigma_0 \in V \backslash \Sigma$ Startsymbol

Symbole:

- Terminalsysmbol: Kann nicht weiter durch eine Produktionsregel ersetzt werden
- Nicht-Terminalsymbol: Ersetzbare Symbole

EBNF

Erweiterte Backus-Naur-Form (EBNF) dient der Darstellung kontextfreier Grammatiken.

Kontextfreie Grammatik: Genau ein Nicht-Terminalsymbol wird auf Terminal- oder Nicht-Terminalsymbole abgebildet.

Beispiel für eine Grammatik in EBNF:

```
DIGIT = "0" | "1" | "2" | ... | "9";
SIGN = "+" | "-";
NUM = [SIGN] DIGIT {DIGIT};
```

Gültige Wörter: 1, -300, +123

Wie kann die Grammatik zur Darstellung von Fließkommazahlen erweitert werden?

```
NUM = [SIGN] DIGIT {DIGIT} "." DIGIT {DIGIT};
```

EBNF

Erweiterte Backus-Naur-Form (EBNF) dient der Darstellung kontextfreier Grammatiken.

Kontextfreie Grammatik: Genau ein Nicht-Terminalsymbol wird auf Terminal- oder Nicht-Terminalsymbole abgebildet.

Beispiel für eine Grammatik in EBNF:

```
DIGIT = "0" | "1" | "2" | ... | "9";
SIGN = "+" | "-";
NUM = [SIGN] DIGIT {DIGIT};
```

Gültige Wörter: 1, -300, +123

Wie kann die Grammatik zur Darstellung von Fließkommazahlen erweitert werden?

```
NUM = [SIGN] DIGIT {DIGIT} "." DIGIT {DIGIT};
```

EBNF

Erweiterte Backus-Naur-Form (EBNF) dient der Darstellung kontextfreier Grammatiken.

Kontextfreie Grammatik: Genau ein Nicht-Terminalsymbol wird auf Terminal- oder Nicht-Terminalsymbole abgebildet.

Beispiel für eine Grammatik in EBNF:

```
DIGIT = "0" | "1" | "2" | ... | "9";
SIGN = "+" | "-";
NUM = [SIGN] DIGIT {DIGIT};
```

Gültige Wörter: 1, -300, +123

Wie kann die Grammatik zur Darstellung von Fließkommazahlen erweitert werden?

```
NUM = [SIGN] DIGIT {DIGIT} "." DIGIT {DIGIT};
```

Automaten

Überblick

Ein Automat ist eine abstrakte Maschine und das Modell eines Rechners.

Eigenschaften

- Menge von Eingaben
- Menge von Ausgaben
- Menge von Operationen/Übergängen

Formale Beschreibung eines Automaten als N-Tupel, z.B. Turing-Maschine

$$M = \langle S, \Gamma, \gamma, \Sigma, \delta, S_0, F \rangle$$

(S Zustände, Γ Symbole, γ Blank-Symbol, Σ Eingabe-Symbole, δ Übergangsfunktion, S_0 Startzustand, F Final-Zustand)

Überblick

Ein Automat ist eine abstrakte Maschine und das Modell eines Rechners.

Eigenschaften

- Menge von Eingaben
- Menge von Ausgaben
- Menge von Operationen/Übergängen

Formale Beschreibung eines Automaten als N-Tupel, z.B. Turing-Maschine

$$M = \langle S, \Gamma, \gamma, \Sigma, \delta, S_0, F \rangle$$

(S Zustände, Γ Symbole, γ Blank-Symbol, Σ Eingabe-Symbole, δ Übergangsfunktion, S_0 Startzustand, F Final-Zustand)

Überblick

Ein Automat ist eine abstrakte Maschine und das Modell eines Rechners.

Eigenschaften

- Menge von Eingaben
- Menge von Ausgaben
- Menge von Operationen/Übergängen

Formale Beschreibung eines Automaten als N-Tupel, z.B. Turing-Maschine

$$M = \langle S, \Gamma, \gamma, \Sigma, \delta, S_0, F \rangle$$

(S Zustände, Γ Symbole, γ Blank-Symbol, Σ Eingabe-Symbole, δ Übergangsfunktion, S_0 Startzustand, F Final-Zustand)

Turingmaschine

Abstraktes Modell eines Rechners

- unendlich langes Speicherband mit Feldern denen ein Zeichen (oder ein Symbol für “leer”) zugewiesen werden kann
- Ein Lese- und Schreibkopf der Felder lesen und schreiben bzw. löschen kann

Kellerautomat

Automat mit Eingabeband und Keller (Stack), Beschreibung als 7-Tupel:

$$M = \langle S, \Sigma, \Gamma, \delta, \#, S_0, F \rangle$$

(S Zustände, Σ Eingabe-Alphabet, Γ Keller-Alphabet, δ Übergangsfunktion, $\#$ Anfangssymbol im Keller, S_0 Startzustand, F Final-Zustand)

Verwendung: Überprüfung, ob eine Eingabe zu einer bestimmten formalen Sprache gehört.

Zustandsautomat

Endlicher Automat mit Zuständen und Übergängen, Beschreibung als 7-Tupel:

$$M = \langle S, \Sigma, \Omega, \delta, \lambda, S_0, F \rangle$$

(S Zustände, Σ Eingabe-Alphabet, Ω Ausgabe-Alphabet, δ Übergangsfunktion, λ Ausgabe-Funktion, S_0 Startzustand, F Final-Zustand)

- Moore-Automat: Die Ausgabe hängt ausschließlich vom Zustand ab, $\lambda : S \to \Omega$
- Mealy-Automat: Die Ausgabe hängt vom Zustand und der Eingabe ab $\lambda : S \times \Sigma \to \Omega$

Zustandsautomat

Endlicher Automat mit Zuständen und Übergängen, Beschreibung als 7-Tupel:

$$M = \langle S, \Sigma, \Omega, \delta, \lambda, S_0, F \rangle$$

(S Zustände, Σ Eingabe-Alphabet, Ω Ausgabe-Alphabet, δ Übergangsfunktion, λ Ausgabe-Funktion, S_0 Startzustand, F Final-Zustand)

- Moore-Automat: Die Ausgabe hängt ausschließlich vom Zustand ab, $\lambda : S \to \Omega$
- Mealy-Automat: Die Ausgabe hängt vom Zustand und der Eingabe ab $\lambda : S \times \Sigma \to \Omega$

Grundlagen Linux

Konsole

- `./a` Programm a starten
- `./a &` Programm vom Terminal getrennt starten
- `a > b.txt` Standardausgabe von a wird in b.txt geschrieben
- `a >> b.txt` Standardausgabe von a wird an b.txt angehängt
- `a 2> b.txt` Fehlerausgabe von a wird in b.txt geschrieben
- `a > b.txt 2>&1` Standard- und Fehlerausgabe von a wird in b.txt geschrieben
- `a < b.txt` Standard-eingabe von a wird aus b.txt gelesen
- `a | b` Ausgabe von a wird an b weitergeleitet
- `a | tee out.txt` Ausgabe von a wird in out.txt ausgegeben und auf der Standardausgabe

Kombinationen sind möglich: `a < b.txt | c > d.txt`

Tastenkombinationen:

- `Strg+Z` Programm pausieren
- `Strg+C` Programm beenden

Konsole

- `./a` Programm a starten
- `./a &` Programm vom Terminal getrennt starten
- `a > b.txt` Standardausgabe von a wird in b.txt geschrieben
- `a >> b.txt` Standardausgabe von a wird an b.txt angehängt
- `a 2> b.txt` Fehlerausgabe von a wird in b.txt geschrieben
- `a > b.txt 2>&1` Standard- und Fehlerausgabe von a wird in b.txt geschrieben
- `a < b.txt` Standard-eingabe von a wird aus b.txt gelesen
- `a | b` Ausgabe von a wird an b weitergeleitet
- `a | tee out.txt` Ausgabe von a wird in out.txt ausgegeben und auf der Standardausgabe

Kombinationen sind möglich: `a < b.txt | c > d.txt`

Tastenkombinationen:

- `Strg+Z` Programm pausieren
- `Strg+C` Programm beenden

Konsole

- `./a` Programm a starten
- `./a &` Programm vom Terminal getrennt starten
- `a > b.txt` Standardausgabe von a wird in b.txt geschrieben
- `a >> b.txt` Standardausgabe von a wird an b.txt angehängt
- `a 2> b.txt` Fehlerausgabe von a wird in b.txt geschrieben
- `a > b.txt 2>&1` Standard- und Fehlerausgabe von a wird in b.txt geschrieben
- `a < b.txt` Standard-eingabe von a wird aus b.txt gelesen
- `a | b` Ausgabe von a wird an b weitergeleitet
- `a | tee out.txt` Ausgabe von a wird in out.txt ausgegeben und auf der Standardausgabe

Kombinationen sind möglich: `a < b.txt | c > d.txt`
Tastenkombinationen:

- `Strg+Z` Programm pausieren
- `Strg+C` Programm beenden

Dateioperationen I

- `ls [-al]` Liste aller Dateien
- `cp source target` Datei kopieren
- `mv source target` Datei verschieben
- `rm file` Datei löschen, `rm -r file` Unterverzeichnisse löschen
- `pwd` Aktuelles Arbeitsverzeichnis anzeigen
- `ln [-s]` (symbolischen) Link erstellen
- `cd [directory]` Verzeichnis wechseln
- `mkdir name` Verzeichnis erstellen
- `rmdir name` leeres Verzeichnis löschen
- `chown [-R] user file` Besitzer einer Datei ändern
- `chmod mode file` Zugriffsberechtigung ändern (mode: `u|g|o`, `+|-`, `r|w|x`)

Dateioperationen II

- `gzip [-d] files` Dateien (de-)komprimieren
- `tar -cf output.tar files` Archiv erstellen
- `tar -xf input.tar` Archiv extrahieren
- `tar -czf output.tar.gz files` Archiv erstellen und mit gzip komprimieren
- `tar -xzf input.tar.hz -C /outputdir` gzip-Archiv extrahieren

Dateiverarbeitung

- `find [../|/dir|/dir1 /dir2] -iname file` Suche nach einer Datei
- `cat file` Dateiinhalt auf Konsole ausgeben
- `grep "search" file` Suche in einer Datei
- `touch file` Erstellt eine leere Datei
- `nano file` Datei in einfachem Editor öffnen

Prozesse

- `top` Liste aller laufender Prozesse
- `ps [aux]` Liste eigener oder aller laufender Prozesse
- `kill pid` Programm mit PID beenden
- `bg` Programm in den Hintergrund schicken
- `fg` Programm in den Vordergrund holen

Netzwerk

- `ping ipaddress` Testen von IP-Verbindungen
- `traceroute ipaddress` Nachverfolgen von IP-Verbindungen
- `dig -x ipaddress` Reverse-DNS-Lookup
- `nslookup domain` DNS-Lookup
- `ifconfig` Netzwerkkonfiguration anzeigen
- `ifconfig eth0 ipaddress netmask mask` Netzwerkkonfiguration ändern
- `route add default gw ipaddress eth0` Standardgateway ändern
- `route -n` Routingtabelle anzeigen

System

- `df [-h]` Informationen über freien Speicherplatz (human readable)
- `du [-h]` Informationen über belegten Speicherplatz im aktuellen Ordner und dessen Unterordner (human readable)
- `date` Systemzeit/-datum anzeigen oder setzen
- `passwd username` Passwort ändern
- `sudu command` Befehl als root-Benutzer ausführen
- `su [username]` In einer Sitzung als anderer Benutzer anmelden
- `shutdown -h now` System herunterfahren
- `reboot` System neu starten
- `clear` Konsolenausgabe löschen

C-Compiler

Empfohlene Compiler-Einstellungen:

```
gcc -g -Wall -Wextra -Werror program.c -o program
```

- `-g` Debugging-Symbole in der Ausgabedatei
- `-Wall -Wextra` Warnungen anzeigen
- `-Werror` Bei Fehler behandeln
- `-o` Setzt Namen der Ausgabedatei

Übungsaufgabe:

Erstellen Sie auf der Konsole ein neues Verzeichnis und darin mit nano eine Datei `main.c`. Schreiben Sie darin ein Programm in C das Ihren Namen ausgibt und kompilieren Sie dieses mit gcc. Prüfen Sie, ob alle Rechte zum Ausführen gesetzt sind und starten Sie das Programm auf der Konsole.

C-Compiler

Empfohlene Compiler-Einstellungen:
`gcc -g -Wall -Wextra -Werror program.c -o program`

- `-g` Debugging-Symbole in der Ausgabedatei
- `-Wall -Wextra` Warnungen anzeigen
- `-Werror` Bei Fehler behandeln
- `-o` Setzt Namen der Ausgabedatei

Übungsaufgabe:
Erstellen Sie auf der Konsole ein neues Verzeichnis und darin mit nano eine Datei `main.c`. Schreiben Sie darin ein Programm in C das Ihren Namen ausgibt und kompilieren Sie dieses mit gcc. Prüfen Sie, ob alle Rechte zum Ausführen gesetzt sind und starten Sie das Programm auf der Konsole.

Grundlagen C

Motivation

Programmiersprachen-Ranking Februar 2016[2]:

1	Java	21,145%
2	C	15,59%
3	C++	6,907%
4	...	$<$ 4,5%

- international standardisiert (ANSI, ISO)
- abgeschlossener Sprachumfang (C89: 32 keywords)
- Compiler für (fast) alle Plattformen (insbesondere μcs)
- beeinflusste C++, Objective-C, C#, Java, JavaScript, ...

[2] http://www.heise.de/developer/meldung/Programmiersprachen-Ranking-Februarausgabe-des-TIOBE-Index-legt-Fokus-auf-Java-Entwicklung-3091879.html

Grundlagen I

Geschichte

- Entwickelt von Dennis Ritchie bei den Bell Labs 1972
- ANSI/ISO-Standardisierung 1989 (C89, später C99, C11)

Eigenschaften

- Paradigma: imperativ, strukturiert
- Typisierung: statisch (aber weakly enforced, implizite Konvertierung)
- Verwendet Präprozessor
- hardwarenah, direkter Speicherzugriff über Adressen
- Keine Objektorientierung, kein Speichermanagement

Grundlagen I

Geschichte

- Entwickelt von Dennis Ritchie bei den Bell Labs 1972
- ANSI/ISO-Standardisierung 1989 (C89, später C99, C11)

Eigenschaften

- Paradigma: imperativ, strukturiert
- Typisierung: statisch (aber weakly enforced, implizite Konvertierung)
- Verwendet Präprozessor
- hardwarenah, direkter Speicherzugriff über Adressen
- Keine Objektorientierung, kein Speichermanagement

Grundlagen II

Ein kurzes C-Programm...

```
#include <stdio.h>

int main() {
  printf("Hello, World\n");
  return 0;
}
```

noch kürzer

```
int main(){}
```

Grundlagen II

Ein kurzes C-Programm...

```
#include <stdio.h>

int main() {
  printf("Hello, World\n");
  return 0;
}
```

noch kürzer

```
int main(){}
```

Darstellung im Maschinencode

Siehe: `https://godbolt.org/`

Präprozessor, Linker und Compiler

1. Präprozessor: entfernt Kommentare, Leerzeilen, ersetzt Text (Makros)
2. Compiler: Umwandlung des Quellcodes in binäre Objektdateien (*.o)
3. Linker: Verbindet die Objektdateien (z.B. mit Bibliotheken)

Bekannte C-Compiler: GNU GCC C, Microsoft Visual C++

Präprozessor und Makros I

Häufige Präprozessoranweisungen

```
#include <...>

#define ...

#ifdef ...
#else ...
#endif

#if ...
#endif
```

Beispiele

```
#define TEST

#ifdef TEST
  // code...
#endif

#define SQUARE(x)((x)*(x))

SQUARE(2)
```

Präprozessor und Makros I

Häufige Präprozessoranweisungen

```
#include <...>

#define ...

#ifdef ...
#else ...
#endif

#if ...
#endif
```

Beispiele

```
#define TEST

#ifdef TEST
  // code...
#endif

#define SQUARE(x)((x)*(x))

SQUARE(2)
```

Präprozessor und Makros II

Vorsicht bei der Verwendung doppelter #includes

Vorsicht bei der Verwendung von Makros

```
#define SQUARE(x)((x)*(x))
#define ADD(x,y) x+y

int x = 2;
int y = 3;
SQUARE(x++);
printf("%d\n",x);
int z = ADD(x,y)*2;
printf("%d\n",z);
```

Präprozessor und Makros II

Vorsicht bei der Verwendung doppelter #includes
Vorsicht bei der Verwendung von Makros

```
#define SQUARE(x)((x)*(x))
#define ADD(x,y) x+y

int x = 2;
int y = 3;
SQUARE(x++);
printf("%d\n",x);
int z = ADD(x,y)*2;
printf("%d\n",z);
```

Schlüsselwörter und Datentypen I

Schlüsselwörter

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

Datentypen

- Specifiers: char, int, float, double
- Modifier: signed, unsigned, short, long
- void
- enum, structure, union
- const

Die Größe von Datentypen ist in C nicht festgelegt, die Relation schon

Schlüsselwörter und Datentypen I

Schlüsselwörter

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

Datentypen

- Specifiers: char, int, float, double
- Modifier: signed, unsigned, short, long
- void
- enum, structure, union
- const

Die Größe von Datentypen ist in C nicht festgelegt, die Relation schon

Schlüsselwörter und Datentypen I

Schlüsselwörter

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

Datentypen

- Specifiers: char, int, float, double
- Modifier: signed, unsigned, short, long
- void
- enum, structure, union
- const

Die Größe von Datentypen ist in C nicht festgelegt, die Relation schon

Schlüsselwörter und Datentypen II

Die jeweils gültigen Grenzen stehen in `limits.h` und für Gleipunktzahlen in `float.h`

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

int main(void) {
    printf("int-Wert mindestens : %d\n", INT_MIN);
    printf("int-Wert maximal    : %d\n", INT_MAX);
    printf("unsigned int max.   : %u\n", UINT_MAX);
    printf("int benoetigt %d Byte (%d Bit) Speicher\n",
        sizeof(int), sizeof(int) * CHAR_BIT);
    return 0;
}
```

Listing übernommen aus (Wolf, J.: C von A bis Z – Das umfassende Handbuch, Galileo Computing, 2005[3])

[3] Als Open Book frei zugänglich `http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/`

void

`void` hat mehrere Bedeutungen:

1. Funktion ohne Rückgabewert

```
void f() {}
```

2. Funktion ohne Parameter

```
int f(void) {}
```

3. Typenlose Zeiger

```
void* p;
int f(void* arg) { ... }
```

4. Cast auf void, um Werte zu verwerfen

```
void f(int i) {
 (void)i; // Parameter wird nicht benutzt
 return 10;
}
```

Negative Zahlen

z.B. Verwendung eines “Vorzeichenbits”, besser: Zweierkomplement

Bei 8 Bit: $-128_{10} \ldots 127_{10}$, Allgemein: $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$
Berechnung für eine negative Zahl:

1. In Binärdarstellung umrechnen (ohne Vorzeichen)
2. Invertieren
3. 1 addieren

$-5 \rightarrow 00000101 \rightarrow 11111010 \rightarrow 11111011$

Bitweise Betrachtung:
$00001101 \rightarrow 1 + 4 + 8 = 13$
$11110011 \rightarrow -128 + 64 + 32 + 16 + 2 + 1 = -13$

Vorteil: Einfache Addition und Subtraktion ohne Fallunterscheidung

Negative Zahlen

z.B. Verwendung eines “Vorzeichenbits”, besser: Zweierkomplement

Bei 8 Bit: $-128_{10} \ldots 127_{10}$, Allgemein: $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$
Berechnung für eine negative Zahl:

1. In Binärdarstellung umrechnen (ohne Vorzeichen)
2. Invertieren
3. 1 addieren

$-5 \rightarrow 00000101 \rightarrow 11111010 \rightarrow 11111011$

Bitweise Betrachtung:
$00001101 \rightarrow 1 + 4 + 8 = 13$
$11110011 \rightarrow -128 + 64 + 32 + 16 + 2 + 1 = -13$

Vorteil: Einfache Addition und Subtraktion ohne Fallunterscheidung

Negative Zahlen

z.B. Verwendung eines “Vorzeichenbits”, besser: Zweierkomplement

Bei 8 Bit: $-128_{10} \ldots 127_{10}$, Allgemein: $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$
Berechnung für eine negative Zahl:

1. In Binärdarstellung umrechnen (ohne Vorzeichen)
2. Invertieren
3. 1 addieren

$-5 \rightarrow 00000101 \rightarrow 11111010 \rightarrow 11111011$

Bitweise Betrachtung:
$00001101 \rightarrow 1 + 4 + 8 = 13$
$11110011 \rightarrow -128 + 64 + 32 + 16 + 2 + 1 = -13$

Vorteil: Einfache Addition und Subtraktion ohne Fallunterscheidung

Negative Zahlen

z.B. Verwendung eines “Vorzeichenbits”, besser: Zweierkomplement

Bei 8 Bit: $-128_{10} \ldots 127_{10}$, Allgemein: $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$
Berechnung für eine negative Zahl:

1. In Binärdarstellung umrechnen (ohne Vorzeichen)
2. Invertieren
3. 1 addieren

$-5 \rightarrow 00000101 \rightarrow 11111010 \rightarrow 11111011$

Bitweise Betrachtung:
$00001101 \rightarrow 1 + 4 + 8 = 13$
$11110011 \rightarrow -128 + 64 + 32 + 16 + 2 + 1 = -13$

Vorteil: Einfache Addition und Subtraktion ohne Fallunterscheidung

Negative Zahlen

z.B. Verwendung eines "Vorzeichenbits", besser: Zweierkomplement

Bei 8 Bit: $-128_{10} \ldots 127_{10}$, Allgemein: $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1} - 1$
Berechnung für eine negative Zahl:

1. In Binärdarstellung umrechnen (ohne Vorzeichen)
2. Invertieren
3. 1 addieren

$-5 \rightarrow 00000101 \rightarrow 11111010 \rightarrow 11111011$

Bitweise Betrachtung:
$00001101 \rightarrow 1 + 4 + 8 = 13$
$11110011 \rightarrow -128 + 64 + 32 + 16 + 2 + 1 = -13$

Vorteil: Einfache Addition und Subtraktion ohne Fallunterscheidung

Gleitpunkttypen float, double, long double I

Typische Werte für PC:

	Bit	Wertebereich	Genauigkeit
float	32	$1.5 10^{-45} \ldots 3.4 10^{38}$	7 - 8
double	64	$5.0 10^{-324} \ldots 1.7 10^{308}$	15 - 16
long double	80	$1.9 10^{-4951} \ldots 1.1 10^{4932}$	19 - 20

Darstellung meist nach ANSI/IEEE Std 754-1985
Beispiel 32 Bit float: $f = m \cdot b^e$, z.B. $b = 2$

Vorzeichen s	Exponent e	Mantisse m
31	30 ... 23	22 ... 0

Gleitpunkttypen float, double, long double II

Berechnung nach IEEE ANSI/IEEE Std 754-1985:

1. Vorkommazahl umrechnen
2. Nachkommazahl umrechnen
3. Mantisse ermitteln
4. Exponent umrechnen (Bias bei 32 Bit: 127)
5. Vorzeichen bestimmen
6. Gleitkommazahl bilden

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

Gleitpunkttypen float, double, long double II

Berechnung nach IEEE ANSI/IEEE Std 754-1985:

1. Vorkommazahl umrechnen
2. Nachkommazahl umrechnen
3. Mantisse ermitteln
4. Exponent umrechnen (Bias bei 32 Bit: 127)
5. Vorzeichen bestimmen
6. Gleitkommazahl bilden

$20,24_{10} \rightarrow 0\,10000011\,01000011110101110110011$

Gleitpunkttypen float, double, long double III

Besondere Interpretationen:

1. $+0 : s = 0, e = \{0, \ldots, 0\}, m = \{0, \ldots, 0\}$
2. $-0 : s = 1, e = \{0, \ldots, 0\}, m = \{0, \ldots, 0\}$
3. $+\infty : s = 0, e = \{1, \ldots, 1\}, m = \{0, \ldots, 0\}$
4. $-\infty : s = 1, e = \{1, \ldots, 1\}, m = \{0, \ldots, 0\}$
5. NaN: $s = *, e = \{1, \ldots, 1\}, m = *$ (nicht $\{0, \ldots, 0\}$)

Beispiel: Umrechnung nach IEEE Std 754-1985 I

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(1) Vorkommazahl umrechnen:

$20/2 =$	10	0	letztes Bit
$10/2 =$	5	0	
$5/2 =$	2,5	1	
$2/2 =$	1	0	
$1/2 =$	0,5	1	

Beispiel: Umrechnung nach IEEE Std 754-1985 II

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(2) Nachkommazahl umrechnen:

$$
\begin{array}{rrcl}
0{,}24 \cdot 2 = & 0{,}48 & 0 & \text{erstes Bit} \\
0{,}48 \cdot 2 = & 0{,}96 & 0 & \\
0{,}96 \cdot 2 = & 1{,}92 & 1 & \\
0{,}92 \cdot 2 = & 1{,}84 & 1 & \\
0{,}84 \cdot 2 = & 1{,}68 & 1 & \\
0{,}68 \cdot 2 = & 1{,}36 & 1 & \\
0{,}36 \cdot 2 = & 0{,}72 & 0 & \\
\multicolumn{4}{c}{\ldots} \\
0{,}6 \cdot 2 = & 1{,}2 & 1 &
\end{array}
$$

Beispiel: Umrechnung nach IEEE Std 754-1985 III

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(3) Mantisse ermitteln

$10100,001111\ldots1 \rightarrow 1,0100\ 001111\ldots1$

Komma um vier Stellen nach Links verschieben, vorne steht immer eine 1

(4) Exponent umrechnen (Bias bei 32 Bit: 127)

Neuer Exponent = Exponent + Bias = 4 + 127 = 131

131/2 =	65,5	1	letztes Bit
65/2 =	32,5	1	
32/2 =	16	0	
16/2 =	8	0	
8/2 =	4	0	
4/2 =	2	0	
2/2 =	1	0	
1/2 =	0,5	1	

Beispiel: Umrechnung nach IEEE Std 754-1985 III

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(3) Mantisse ermitteln

$10100,001111\ldots1 \rightarrow 1,0100\ 001111\ldots1$

Komma um vier Stellen nach Links verschieben, vorne steht immer eine 1

(4) Exponent umrechnen (Bias bei 32 Bit: 127)

$$\text{Neuer Exponent} = \text{Exponent} + \text{Bias} = 4 + 127 = 131$$

$$\begin{array}{rrll} 131/2 = & 65{,}5 & 1 & \text{letztes Bit} \\ 65/2 = & 32{,}5 & 1 & \\ 32/2 = & 16 & 0 & \\ 16/2 = & 8 & 0 & \\ 8/2 = & 4 & 0 & \\ 4/2 = & 2 & 0 & \\ 2/2 = & 1 & 0 & \\ 1/2 = & 0{,}5 & 1 & \end{array}$$

Beispiel: Umrechnung nach IEEE Std 754-1985 IV

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(5) Vorzeichen bestimmen

positiv $\rightarrow 0$

negativ $\rightarrow 1$

(6) Gleitkommazahl bilden

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

Beispiel: Umrechnung nach IEEE Std 754-1985 IV

$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

(5) Vorzeichen bestimmen

positiv $\rightarrow 0$
negativ $\rightarrow 1$

(6) Gleitkommazahl bilden
$20,24_{10} \rightarrow 0\ 10000011\ 01000011110101110110011$

Zeiger

Enthalten eine Adresse, Zeigen auf eine Speicherstelle

```c
int a = 2;
int* b = &a;
printf("%d, %d", a, *b);
```

Vorsicht bei Deklaration und Initialisierung

```c
// Deklariert beides nur einen Zeiger
int* ptr1, ptr2;
int *ptr1, ptr2;

// Initialisiert mit NULL
int* ptr3 = NULL;
```

Zeigerarithmetik (siehe Arrays und Zeiger)

Zeiger

Enthalten eine Adresse, Zeigen auf eine Speicherstelle

```
int a = 2;
int* b = &a;
printf("%d, %d", a, *b);
```

Vorsicht bei Deklaration und Initialisierung

```
// Deklariert beides nur einen Zeiger
int* ptr1, ptr2;
int *ptr1, ptr2;

// Initialisiert mit NULL
int* ptr3 = NULL;
```

Zeigerarithmetik (siehe Arrays und Zeiger)

Zeiger

Enthalten eine Adresse, Zeigen auf eine Speicherstelle

```
int a = 2;
int* b = &a;
printf("%d, %d", a, *b);
```

Vorsicht bei Deklaration und Initialisierung

```
// Deklariert beides nur einen Zeiger
int* ptr1, ptr2;
int *ptr1, ptr2;

// Initialisiert mit NULL
int* ptr3 = NULL;
```

Zeigerarithmetik (siehe Arrays und Zeiger)

Operatoren

Unäre

```
*, &, ++, --, +=, -=, ...
z.B.: a++, --a
```

Binäre

```
+, -, *, /, %, ...
```

Ternäroperator

```
(a > b) ? a : b
(a > ((x == y) ? x : 0) ? (a == y) : b;
```

Bitoperatoren

```
unsigned int a = 0xFF;
unsigned int b = 0xAA;

unsigned int c = a ^ b;
unsigned int d = c << 4;
```

Operatoren

Unäre

```
*, &, ++, --, +=, -=, ...
z.B.: a++, --a
```

Binäre

```
+, -, *, /, %, ...
```

Ternäroperator

```
(a > b) ? a : b
(a > ((x == y) ? x : 0) ? (a == y) : b;
```

Bitoperatoren

```
unsigned int a = 0xFF;
unsigned int b = 0xAA;

unsigned int c = a ^ b;
unsigned int d = c << 4;
```

Operatoren

Unäre

```
*, &, ++, --, +=, -=, ...
z.B.: a++, --a
```

Binäre

```
+, -, *, /, %, ...
```

Ternäroperator

```
(a > b) ? a : b
(a > ((x == y) ? x : 0) ? (a == y) : b;
```

Bitoperatoren

```
unsigned int a = 0xFF;
unsigned int b = 0xAA;

unsigned int c = a ^ b;
unsigned int d = c << 4;
```

Operatoren

Bei der Verwendung von Operatoren Präzedenzreihenfolge beachten!
Zur Sicherheit immer klammern.

1	a++, a−−, Funktionsaufruf (), Array Element [], ->, .
2	++a, −−a, !, ~, unäres -, unäres +, Adresse &, Indirektion *, sizeof, cast (type)
3	*, /, %
4	+, -
5	<<, >>
6	<, <=, >, >=
7	==, !=
8	Bitweise &
9	^
10	|
11	&&
12	||
13	?;
14	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
16	,

Arrays I

Arrays

```
double measurements[128];
measurement[0] = 3.0;
printf("%f\n", measurement[127]);
```

Arrays und Zeiger

```
double measurements[128];
double* ptr = measurements; // double* ptr = &measurements[0]
for (int i=0; i < 128; i++)
  printf("%f\n", *(ptr+i));
```

Array mit Funktions-Zeiger

```
int square(int a) { return a*a; }

int (*functions[2])(int);
functions[0] = square;
printf("%d\n", functions[0](9));
```

Arrays I

Arrays

```
double measurements[128];
measurement[0] = 3.0;
printf("%f\n", measurement[127]);
```

Arrays und Zeiger

```
double measurements[128];
double* ptr = measurements; // double* ptr = &measurements[0]
for (int i=0; i < 128; i++)
  printf("%f\n", *(ptr+i));
```

Array mit Funktions-Zeiger

```
int square(int a) { return a*a; }

int (*functions[2])(int);
functions[0] = square;
printf("%d\n", functions[0](9));
```

Arrays I

Arrays

```
double measurements[128];
measurement[0] = 3.0;
printf("%f\n", measurement[127]);
```

Arrays und Zeiger

```
double measurements[128];
double* ptr = measurements; // double* ptr = &measurements[0]
for (int i=0; i < 128; i++)
  printf("%f\n", *(ptr+i));
```

Array mit Funktions-Zeiger

```
int square(int a) { return a*a; }

int (*functions[2])(int);
functions[0] = square;
printf("%d\n", functions[0](9));
```

Arrays II

Mehrdimensionale Arrays

```
int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; // [Rows][Cols]
```

	Col 0	Col 1	Col 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]
Row 2	a[2][0]	a[2][1]	a[2][2]

$N = \text{Rows} \cdot \text{Cols}$

Initialisierung:

```
int b[2][3][4] = {{{1,1,1,1},{2,2,2,2},{3,3,3,3}},
                  {{4,4,4,4},{5,5,5,5},{6,6,6,6}}};
int c[2][3] = {{1,2,3},{4,5,6}};
int d[][3] = {{1,2,3},{4,5,6}};
int e[2][3] = {1,2,3,4,5,6};
```

Arrays II

Mehrdimensionale Arrays

```
int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; // [Rows][Cols]
```

	Col 0	Col 1	Col 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]
Row 2	a[2][0]	a[2][1]	a[2][2]

$N = \text{Rows} \cdot \text{Cols}$

Initialisierung:

```
int b[2][3][4] = {{{1,1,1,1},{2,2,2,2},{3,3,3,3}},
                  {{4,4,4,4},{5,5,5,5},{6,6,6,6}}};
int c[2][3] = {{1,2,3},{4,5,6}};
int d[][3] = {{1,2,3},{4,5,6}};
int e[2][3] = {1,2,3,4,5,6};
```

Arrays III

Mehrdimensionale Arrays sind ein zusammenhängender Speicherblock und können auch als solcher adressiert werden.

```
int arr[2][3] = {1,2,3,4,5,6};
int* p = arr;
int val = *(p+3);
```

Nicht verwechseln mit Zeiger auf dynamisch allozierte Arrays, hier wird kein zusammenhängender Speicherbereich reserviert:

```
int* arr[2];
arr[0] = (int*)malloc(3*sizeof(int));
arr[1] = (int*)malloc(3*sizeof(int));
arr[0][1] = 1;
```

Arrays III

Mehrdimensionale Arrays sind ein zusammenhängender Speicherblock und können auch als solcher adressiert werden.

```
int arr[2][3] = {1,2,3,4,5,6};
int* p = arr;
int val = *(p+3);
```

Nicht verwechseln mit Zeiger auf dynamisch allozierte Arrays, hier wird kein zusammenhängender Speicherbereich reserviert:

```
int* arr[2];
arr[0] = (int*)malloc(3*sizeof(int));
arr[1] = (int*)malloc(3*sizeof(int));
arr[0][1] = 1;
```

Arrays IV

Besonderheiten bei der Initialisierung:

```
int arr[10] = {2}; // 2,0,0,...,0
int arr[10] = {[0]=2, [1]=3}; // 2,3,0,...,0
int arr[10] = {1,2,[2]=3}; // 1,2,3,0,...,0
```

Verwendung von `memset`:

```
#include <string.h>
int arr[10];
memset(arr, 0, sizeof(arr)); // mit 0 initialisieren
```

Achtung Spezialfall!

`void *memset(void *s, int c, size_t n);`

belegt die ersten n **Bytes** vom Speicher auf den s Zeigt mit dem Wert vom **Byte** c

```
int arr[10];
memset(arr, 1, 2*sizeof(int)); // die ersten beiden int im Array
                               // werden zu: 0x01010101 (4 Byte int)
```

Arrays IV

Besonderheiten bei der Initialisierung:

```
int arr[10] = {2}; // 2,0,0,...,0
int arr[10] = {[0]=2, [1]=3}; // 2,3,0,...,0
int arr[10] = {1,2,[2]=3}; // 1,2,3,0,...,0
```

Verwendung von `memset`:

```
#include <string.h>
int arr[10];
memset(arr, 0, sizeof(arr)); // mit 0 initialisieren
```

Achtung Spezialfall!

```
void *memset(void *s, int c, size_t n);
```

belegt die ersten n **Bytes** vom Speicher auf den s Zeigt mit dem Wert vom **Byte** c

```
int arr[10];
memset(arr, 1, 2*sizeof(int)); // die ersten beiden int im Array
                               // werden zu: 0x01010101 (4 Byte int)
```

Arrays IV

Besonderheiten bei der Initialisierung:

```
int arr[10] = {2}; // 2,0,0,...,0
int arr[10] = {[0]=2, [1]=3}; // 2,3,0,...,0
int arr[10] = {1,2,[2]=3}; // 1,2,3,0,...,0
```

Verwendung von `memset`:

```
#include <string.h>
int arr[10];
memset(arr, 0, sizeof(arr)); // mit 0 initialisieren
```

Achtung Spezialfall!

`void *memset(void *s, int c, size_t n);`

belegt die ersten n **Bytes** vom Speicher auf den s Zeigt mit dem Wert vom **Byte** c

```
int arr[10];
memset(arr, 1, 2*sizeof(int)); // die ersten beiden int im Array
                               // werden zu: 0x01010101 (4 Byte int)
```

Strings

Strings: Ein Zeichenarray mit 0-Character terminiert. Compiler terminiert implizit.

```
char c = 'A';
char str1[] = "Hallo, Welt"; // Laenge implizit festgelegt mit 12
char str2[15] = "Hallo, Welt"; // Laenge 15, 3 Zeichen unbenutzt
char* str = "Hallo, Welt"; // Zeigt auf Adr. von 'H', implizit const
```

Stringoperationen am besten mit `string.h`, z.B.:

- `strcpy`
- `strcat`
- `strcmp`
- `strlen`

Welchen Wert haben `n` und `p`?

```
int n = strlen("Hallo,\n\0Welt");
int p = sizeof("Hallo,\n\0Welt");
```

Strings

Strings: Ein Zeichenarray mit 0-Character terminiert. Compiler terminiert implizit.

```
char c = 'A';
char str1[] = "Hallo, Welt"; // Laenge implizit festgelegt mit 12
char str2[15] = "Hallo, Welt"; // Laenge 15, 3 Zeichen unbenutzt
char* str = "Hallo, Welt"; // Zeigt auf Adr. von 'H', implizit const
```

Stringoperationen am besten mit `string.h`, z.B.:

- `strcpy`
- `strcat`
- `strcmp`
- `strlen`

Welchen Wert haben `n` und `p`?

```
int n = strlen("Hallo,\n\0Welt");
int p = sizeof("Hallo,\n\0Welt");
```

Strings

Strings: Ein Zeichenarray mit 0-Character terminiert. Compiler terminiert implizit.

```
char c = 'A';
char str1[] = "Hallo, Welt"; // Laenge implizit festgelegt mit 12
char str2[15] = "Hallo, Welt"; // Laenge 15, 3 Zeichen unbenutzt
char* str = "Hallo, Welt"; // Zeigt auf Adr. von 'H', implizit const
```

Stringoperationen am besten mit `string.h`, z.B.:

- `strcpy`
- `strcat`
- `strcmp`
- `strlen`

Welchen Wert haben `n` und `p`?

```
int n = strlen("Hallo,\n\0Welt");
int p = sizeof("Hallo,\n\0Welt");
```

Eigene Datentypen I

C erlaubt die Definition eigener Datentypen mit `struct` und `union`

struct:

```
struct Node {
 struct Node* left;
 struct Node* right;
 int val;
};
typedef struct Node NODE;
```

Verwendung:

```
NODE root;
NODE* left = (NODE*)malloc(sizeof(NODE));
NODE* right = (NODE*)malloc(sizeof(NODE));

left->val = 1;
right->val = 3;

root.val = 2;
root.left = left;
root.right = right;
```

Eigene Datentypen I

C erlaubt die Definition eigener Datentypen mit `struct` und `union`

struct:

```
struct Node {
 struct Node* left;
 struct Node* right;
 int val;
};
typedef struct Node NODE;
```

Verwendung:

```
NODE root;
NODE* left = (NODE*)malloc(sizeof(NODE));
NODE* right = (NODE*)malloc(sizeof(NODE));

left->val = 1;
right->val = 3;

root.val = 2;
root.left = left;
root.right = right;
```

Eigene Datentypen II

Varianten der Initialisierung:

```
struct Data {
 char givenname[30];
 char surname[30];
 int birthyear;
};

struct Data person = {"Dennis", "Ritchie", 1941};
// oder (C99)
struct Data person = {.givenname="Dennis",
                      .surname="Ritchie",
                      .birthyear=1941};
// oder (C99)
struct Data person = {.surname="Ritchie",
                      1941,
                      .givenname="Dennis"};
```

Eigene Datentypen III

union:

Jeder Member startet bei der selben Speicheradresse

```
union Info {
 char name[100];
 int age;
};
```

Speicherbelegung:

- `struct` belegt für jedes Feld Speicher
- `union` belegt den Speicher des größten Feldes

Achtung

- Unions haben keine Typprüfung
- Es muss sichergestellt werden, dass zu einem Zeitpunkt nur genau ein Feld verwendet wird

Eigene Datentypen III

union:

Jeder Member startet bei der selben Speicheradresse

```
union Info {
 char name[100];
 int age;
};
```

Speicherbelegung:

- `struct` belegt für jedes Feld Speicher
- `union` belegt den Speicher des größten Feldes

Achtung

- Unions haben keine Typprüfung
- Es muss sichergestellt werden, dass zu einem Zeitpunkt nur genau ein Feld verwendet wird

Eigene Datentypen III

union:

Jeder Member startet bei der selben Speicheradresse

```
union Info {
 char name[100];
 int age;
};
```

Speicherbelegung:

- `struct` belegt für jedes Feld Speicher
- `union` belegt den Speicher des größten Feldes

Achtung

- Unions haben keine Typprüfung
- Es muss sichergestellt werden, dass zu einem Zeitpunkt nur genau ein Feld verwendet wird

Lesen von Deklarationen

Nach Kernighan/Ritchie: *"go right when you can, go left when you must"*[4]

```
int** f();
long** g[2];
const char* const c;
int* (*(*h[2])());
```

f ist eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt
g ist ein Array mit 2 Elementen vom Typ Zeiger auf Zeiger auf long
c ist ein Zeiger vom Typ const auf ein Element const char
h ist ein Array mit zwei Elementen vom Typ Zeiger auf eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

[4] `http://www.unixwiz.net/techtips/reading-cdecl.html`

Lesen von Deklarationen

Nach Kernighan/Ritchie: *"go right when you can, go left when you must"*[4]

```
int** f();
long** g[2];
const char* const c;
int* (*(*h[2])());
```

f ist eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

g ist ein Array mit 2 Elementen vom Typ Zeiger auf Zeiger auf long

c ist ein Zeiger vom Typ const auf ein Element const char

h ist ein Array mit zwei Elementen vom Typ Zeiger auf eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

[4] http://www.unixwiz.net/techtips/reading-cdecl.html

Lesen von Deklarationen

Nach Kernighan/Ritchie: *"go right when you can, go left when you must"*[4]

```
int** f();
long** g[2];
const char* const c;
int* (*(*h[2])());
```

f ist eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt
g ist ein Array mit 2 Elementen vom Typ Zeiger auf Zeiger auf long
c ist ein Zeiger vom Typ const auf ein Element const char
h ist ein Array mit zwei Elementen vom Typ Zeiger auf eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

[4] http://www.unixwiz.net/techtips/reading-cdecl.html

Lesen von Deklarationen

Nach Kernighan/Ritchie: *"go right when you can, go left when you must"*[4]

```
int** f();
long** g[2];
const char* const c;
int* (*(*h[2])());
```

f ist eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt
g ist ein Array mit 2 Elementen vom Typ Zeiger auf Zeiger auf long
c ist ein Zeiger vom Typ const auf ein Element const char
h ist ein Array mit zwei Elementen vom Typ Zeiger auf eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

[4] http://www.unixwiz.net/techtips/reading-cdecl.html

Lesen von Deklarationen

Nach Kernighan/Ritchie: *"go right when you can, go left when you must"*[4]

```
int** f();
long** g[2];
const char* const c;
int* (*(*h[2])());
```

f ist eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt
g ist ein Array mit 2 Elementen vom Typ Zeiger auf Zeiger auf long
c ist ein Zeiger vom Typ const auf ein Element const char
h ist ein Array mit zwei Elementen vom Typ Zeiger auf eine Funktion die einen Zeiger auf einen Zeiger auf int zurückgibt

[4] http://www.unixwiz.net/techtips/reading-cdecl.html

Beispiel

Beispiel für eine Klausurfrage

Erklären Sie folgenden Code:

```
int* y[2];
int* (*f[2])(int);
```

Lösung:

Zeile 1: y ist ein Array mit zwei Zeiger vom Typ Zeiger auf int

Zeile 2: f ist ein Array mit Zwei Elementen vom Typ Zeiger auf eine Funktion die int als Argument hat und einen Zeiger auf int zurückgibt.

Beispiel

Beispiel für eine Klausurfrage
Erklären Sie folgenden Code:

```
int* y[2];
int* (*f[2])(int);
```

Lösung:
Zeile 1: y ist ein Array mit zwei Zeiger vom Typ Zeiger auf int
Zeile 2: f ist ein Array mit Zwei Elementen vom Typ Zeiger auf eine Funktion die int als Argument hat und einen Zeiger auf int zurückgibt.

Sichtbarkeit und Lebensdauer von Variablen

- local
- local static
- global - im gesamten Programm sichtbar
- global static - nur in der Datei sichtbar wo deklariert

```
int a;
void f() {
 int b;
}

void g() {
 int c = a; // OK
 int d = b; // Error: out of scope
}

void h() {
 for(int i=0; i<10; i++) {
  static int counter = 0;
  printf("%d\n", counter++);
 }
}
```

Sichtbarkeit und Lebensdauer von Variablen

- local
- local static
- global - im gesamten Programm sichtbar
- global static - nur in der Datei sichtbar wo deklariert

```
int a;
void f() {
 int b;
}

void g() {
 int c = a; // OK
 int d = b; // Error: out of scope
}

void h() {
 for(int i=0; i<10; i++) {
  static int counter = 0;
  printf("%d\n", counter++);
 }
}
```

Stack und Heap / Dynamischer Speicher

Auto-Variablen werden am Stack abgelegt → Lebensdauer ist der Scope

```
void f() {
 int a;
 auto int b;
 for(...) {
  int c;
  c = a + b // OK
  while(...) int d;
  c = d; // nicht OK
 }
}
```

Dynamisch allozierte Variablen werden am Heap abgelegt → Lebensdauer ist das Programm

```
void f() {
 int* a = (int*)malloc(sizeof(int));
 if (a == NULL) return; // Fehlerbehandlung
 // ...
 free(a);
}
```

"jedes malloc braucht ein free"

Stack und Heap / Dynamischer Speicher

Auto-Variablen werden am Stack abgelegt → Lebensdauer ist der Scope

```
void f() {
 int a;
 auto int b;
 for(...) {
  int c;
  c = a + b // OK
  while(...) int d;
  c = d; // nicht OK
 }
}
```

Dynamisch allozierte Variablen werden am Heap abgelegt → Lebensdauer ist das Programm

```
void f() {
 int* a = (int*)malloc(sizeof(int));
 if (a == NULL) return; // Fehlerbehandlung
 // ...
 free(a);
}
```

"jedes malloc braucht ein free"

Stack und Heap / Dynamischer Speicher

Auto-Variablen werden am Stack abgelegt → Lebensdauer ist der Scope

```
void f() {
 int a;
 auto int b;
 for(...) {
  int c;
  c = a + b // OK
  while(...) int d;
  c = d; // nicht OK
 }
}
```

Dynamisch allozierte Variablen werden am Heap abgelegt → Lebensdauer ist das Programm

```
void f() {
 int* a = (int*)malloc(sizeof(int));
 if (a == NULL) return; // Fehlerbehandlung
 // ...
 free(a);
}
```

"jedes malloc braucht ein free"

Dynamischer Speicher I

Dynamische Arrays

```
int size = 10;
int* a = (int*)malloc(size*sizeof(int));

if (a == NULL) return -1;
```

Vergrößern des Arrays

```
int newsize = 12;
int* b = (int*)malloc(newsize*sizeof(int));
if (b == NULL) return -1;

for(int i=0;i<size;i++)
 b[i] = a[i];

free(a);
```

Dynamischer Speicher I

Dynamische Arrays

```
int size = 10;
int* a = (int*)malloc(size*sizeof(int));

if (a == NULL) return -1;
```

Vergrößern des Arrays

```
int newsize = 12;
int* b = (int*)malloc(newsize*sizeof(int));
if (b == NULL) return -1;

for(int i=0;i<size;i++)
 b[i] = a[i];

free(a);
```

Dynamischer Speicher II

Vergrößern des Arrays mit `realloc`

```
int size = 2;
int* b = (int*)malloc(size*sizeof(int));
if (b == NULL) return -1;

int newsize = 12;
b = (int*)realloc(b, newsize*sizeof(int));
```

- Funktioniert auch mit newsize $<$ size, gibt den restlichen Speicher frei
- Legt ein neues Array mit neuer Größe an, kopiert den Inhalt und gibt das alte Array frei

Auch: `calloc`, nimmt Anzahl und Größe und initialisiert mit 0.

Dynamischer Speicher II

Vergrößern des Arrays mit `realloc`

```
int size = 2;
int* b = (int*)malloc(size*sizeof(int));
if (b == NULL) return -1;

int newsize = 12;
b = (int*)realloc(b, newsize*sizeof(int));
```

- Funktioniert auch mit newsize $<$ size, gibt den restlichen Speicher frei
- Legt ein neues Array mit neuer Größe an, kopiert den Inhalt und gibt das alte Array frei

Auch: `calloc`, nimmt Anzahl und Größe und initialisiert mit 0.

Dynamischer Speicher III

Bei modernen Betriebssystemen hat jeder Prozess einen eigenen, virtuellen Speicherbereich

Freie Speicherbereiche, z.B.: [10,4,21,18,7,9,12,15]

```
malloc(10);
malloc(12);
malloc(9);
```

1. First-Fit: [*10*,4,*21*→*12*→*9*,18,7,9,12,15]
2. Next-Fit: [*10*,4,*21*→*12*,*18*→*9*,7,9,12,15]
3. Best-Fit: [*10*,4,20,18,7,*9*,*12*,15]
4. Worst-Fit: [10,4,*21*→*10*,*18*→*12*,7,9,12,*15*→*9*]
5. Quick-Fit (siehe Abschnitt Betriebssysteme)
6. Buddy-Verfahren (siehe Abschnitt Betriebssysteme)

Kontrollstrukturen I

Verzweigung mit `if`, `else if`, `else`

if (a==1)	if(a)
if (a==0)	if(!a)
if (a>b)	if (!(a<=b))
if ((a-b)==0)	if(!(a-b))

`if` wertet einen Ausdruck aus, z.B.
a = 1, b = 7, c=2

```
if(a&&(((b>c)+1)%c));
```

0 wird als false interpretiert, $\neq 0$ als true

Kontrollstrukturen II

```
while(condition == 1);
```

```
do ; while(condition == 1);
```

```
for(;condition==1;) ;
```

- Verlassen der Schleifen mit `break`

```
while(1) if (condition==1) break;
```

- Beenden des aktuellen Schleifendurchlaufs mit `continue`
- Direkte Sprünge mit `goto` (nicht empfohlen)
- Beenden des Programms mit `exit`
- Beenden der Funktion mit `return`

Kontrollstrukturen II

```
while(condition == 1);
```

```
do ; while(condition == 1);
```

```
for(;condition==1;) ;
```

- Verlassen der Schleifen mit `break`

```
while(1) if (condition==1) break;
```

- Beenden des aktuellen Schleifendurchlaufs mit `continue`
- Direkte Sprünge mit `goto` (nicht empfohlen)
- Beenden des Programms mit `exit`
- Beenden der Funktion mit `return`

Kontrollstrukturen III

Gefahr bei der Verwendung von goto

```
void f() {
 goto x;

 for(int i=0;i<10;i++)
  x: printf("%d", i);
}
```

Kontrollstrukturen IV

```
switch (condition) {
  condition1: break;
  condition2: break;
  ...
  default: ;
}
```

`default` und `break` sind optional

Funktionen I

Call by value:

```
void f(int a) { a++; }

int x = 2;
f(x);
```

Call by reference:

```
void f(int* a) { (*a)++; }

int x = 2;
f(&x);
```

Welchen Wert hat x nach dem Funktionsaufruf?

Funktionen II

Achtung! Auch Zeiger werden intern “by value” übergeben:

```
void f(char* c) {
 c = (char*)malloc(10*sizeof(char));
 strcpy(c, "Test");
}

char* str = NULL;
f(str);
```

```
void f(char** c) {
 *c = (char*)malloc(10*sizeof(char));
 strcpy(*c, "Test");
}

char* str = NULL;
f(&str);
```

Welchen Wert hat str nach dem Funktionsaufruf?

Funktionen II

Achtung! Auch Zeiger werden intern "by value" übergeben:

```
void f(char* c) {
 c = (char*)malloc(10*sizeof(char));
 strcpy(c, "Test");
}

char* str = NULL;
f(str);
```

```
void f(char** c) {
 *c = (char*)malloc(10*sizeof(char));
 strcpy(*c, "Test");
}

char* str = NULL;
f(&str);
```

Welchen Wert hat str nach dem Funktionsaufruf?

Rekursion

Funktion ruft sich selbst auf → Abbruchbedingung erforderlich

```
int factorial(int a) {
 if (a > 0) return a*factorial(a-1); // a am Stack
 else return a;
}

factorial(5);
```

Eine Funktion f ist **endrekursiv**, wenn der rekursive Aufruf von f der letzte Befehl in der Funktion ist.

```
int factorial(int accu, int a) {
 if (a > 0) return factorial(accu*a, a-1); // a nicht mehr am Stack
 else return accu;
}

factorial(1,5);
```

Rekursion

Funktion ruft sich selbst auf $\rightarrow$ Abbruchbedingung erforderlich

```
int factorial(int a) {
 if (a > 0) return a*factorial(a-1); // a am Stack
 else return a;
}

factorial(5);
```

Eine Funktion f ist **endrekursiv**, wenn der rekursive Aufruf von f der letzte Befehl in der Funktion ist.

```
int factorial(int accu, int a) {
 if (a > 0) return factorial(accu*a, a-1); // a nicht mehr am Stack
 else return accu;
}

factorial(1,5);
```

Kommandozeilenargumente

Programm `test`:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
 for(int i=0; i<argc; i++)
  printf("%s\n", argv[i]);
 return 0;
}
```

Ausgabe für `./test Hallo Welt`:

```
test
Hallo
Welt
```

→ erstes Argument (argv[0]) ist der Programmname

Kommandozeilenargumente

Programm `test`:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
 for(int i=0; i<argc; i++)
  printf("%s\n", argv[i]);
 return 0;
}
```

Ausgabe für `./test Hallo Welt`:

```
test
Hallo
Welt
```

→ erstes Argument (argv[0]) ist der Programmname

qsort

stdlib stellt eine generische Implementierung für Quick-Sort bereit

```
void qsort(void* base, size_t num, size_t size,
           int (*cmp)(void* elem1, void* elem2));
```

Beispiel:

```
#include <stdlib.h>
#define NUM 100

mycmp(const void*, const void*);

int main() {
 int values[NUM];
 for(int i=0; i<NUM; i++) values[i] = rand();

 qsort(values, NUM, sizeof(int), &mycmp);
 return 0;
}

int mycmp(const void* a, const void* b) {
 return (*(const int*)a - *(const int*)b);
}
```

qsort

stdlib stellt eine generische Implementierung für Quick-Sort bereit

```
void qsort(void* base, size_t num, size_t size,
           int (*cmp)(void* elem1, void* elem2));
```

Beispiel:

```
#include <stdlib.h>
#define NUM 100

mycmp(const void*, const void*);

int main() {
 int values[NUM];
 for(int i=0; i<NUM; i++) values[i] = rand();

 qsort(values, NUM, sizeof(int), &mycmp);
 return 0;
}

int mycmp(const void* a, const void* b) {
 return (*(const int*)a - *(const int*)b);
}
```

bsearch

stdlib stellt eine generische Implementierung für Binary-Search bereit

```
void *bsearch(const void *key, const void *array,
              size_t n, size_t size,
              int (*cmp)(const void* elem1, const void* elem2));
```

Achtung! Die Elemente müssen sortiert vorliegen. Beispiel:

```
#include <stdlib.h>
#define NUM 100

mycmp(const void*, const void*);

int main() {
 int values[NUM] = {1,3,5,8,10,11, ..., 100}; int find=8;
 int* item = (int*)bsearch(&find, values, NUM, sizeof(int), &mycmp);
 // item ist NULL oder ein Zeiger zum gesuchten Element
 return 0;
}

int mycmp(const void* a, const void* b) {
 return (*(const int*)a - *(const int*)b);
}
```

bsearch

stdlib stellt eine generische Implementierung für Binary-Search bereit

```
void *bsearch(const void *key, const void *array,
              size_t n, size_t size,
              int (*cmp)(const void* elem1, const void* elem2));
```

Achtung! Die Elemente müssen sortiert vorliegen. Beispiel:

```
#include <stdlib.h>
#define NUM 100

mycmp(const void*, const void*);

int main() {
 int values[NUM] = {1,3,5,8,10,11, ..., 100}; int find=8;
 int* item = (int*)bsearch(&find, values, NUM, sizeof(int), &mycmp);
 // item ist NULL oder ein Zeiger zum gesuchten Element
 return 0;
}

int mycmp(const void* a, const void* b) {
 return (*(const int*)a - *(const int*)b);
}
```

Pseudozufallszahlen

*Pseudo*zufallszahlen erzeugen mit C: `rand` erzeugt eine Pseudozufallszahl zwischen 0 und `RAND_MAX`.

```
rand() % 100; // Pseudozufallszahl zwischen 0 und 99
rand() % 100 + 1; // Pseudozufallszahl zwischen 1 und 100
```

```
#include <stdlib.h>
#include <time.h>

int main() {
 srand(time(NULL));

 for(int i=0;i<100;i++)
  printf("%d", rand() % 100 + 1);
}
```

Dateien öffnen

```c
#include <stdio.h>
FILE* file = fopen("file.txt", "r");
if (file == NULL) ; // Fehler
```

Modi zum Öffnen einer Datei

1. `r` öffnet die Datei zum Lesen
2. `r+` öffnet die Datei zum Lesen und Schreiben
3. `w` legt eine Datei zum Ändern an
4. `w` legt eine Datei zum Ändern an, wenn diese bereits existiert wird sie gelöscht
5. `a` öffnet eine Datei zum Schreiben (nur Anhängen)
6. `a` öffnet eine Datei zum Schreiben (nur Anhängen), wenn diese nicht existiert wird sie angelegt

Es kann auch zwischen Binär- (`b`) und Textdateien (`t`) unterschieden werden. Bei letzteren wird der Zeilenumbruch (`\n` → `\r\n` unter Windows) automatisch konvertiert.

Dateien lesen und schreiben I

Zeichenweise lesen, gibt einen `char` oder `EOF`

```
int fgetc(FILE * fp);
char *fgets(char *buf, int n, FILE *fp);
```

Zeichenweise schreiben

```
int fputc(int c, FILE *fp);
int fputs(const char *s, FILE *fp);
```

Blockweise oder formatiert lesen

```
size_t fread(void *ptr, size_t size,
             size_t count, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
```

Blockweise oder formatiert schreiben

```
size_t fwrite(const void *ptr, size_t size,
              size_t count, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
```

Dateien lesen und schreiben I

Zeichenweise lesen, gibt einen `char` oder `EOF`

```
int fgetc(FILE * fp);
char *fgets(char *buf, int n, FILE *fp);
```

Zeichenweise schreiben

```
int fputc(int c, FILE *fp);
int fputs(const char *s, FILE *fp);
```

Blockweise oder formatiert lesen

```
size_t fread(void *ptr, size_t size,
             size_t count, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
```

Blockweise oder formatiert schreiben

```
size_t fwrite(const void *ptr, size_t size,
              size_t count, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
```

Dateien lesen und schreiben II

```c
#include <stdio.h>

int main () {
  FILE* file = fopen("test.txt", "wt");
  char write_buffer[] = {'H','a','l','l','o',',',' ',
                         'W','e','l','t','!'};
  fwrite(write_buffer , sizeof(char), sizeof(write_buffer), file);
  fprintf(file, "Hallo, Welt%c", write_buffer[11]);
  fclose (file);
  return 0;
}
```

Optimierendes Programmieren

1. global und local static Variablen haben Performancevorteile
2. Funktionsprototypen für bessere Compileroptimierung, kein Rückgabewert wenn nicht unbedingt notwendig
3. Datentypen so sparsam wie möglich verwenden
4. Loop unrolling[5]

```
for(int i=0;i<3;i++) f(i); -> f(0); f(1); f(2);
```

5. `switch` besser als mehrere `if ... else`
6. Divisionen und Multiplikationen von Zweierpotenzen mit Bit shifting
7. `register` Variablen verwenden

[5] macht i.d.R. der Compiler

Sicheres Programmieren

Vermeidung von

1. Buffer Overflow
2. Memory Leaks
3. ...

z.B.: Sichere String-Verarbeitung: `strncpy` besser als `strcpy`

Beispiel

Beispiel für eine Klausurfrage

Gegeben ist eine iterative Implementierung zur Berechnung von $f(x, n) = x^n$.

Schreiben Sie eine endrekursive Variante zur Berechnung dieser Funktion.

```
int f(int x, int n) {
 int y = x;
 for (int i=1; i<n; i++) y = y * x;
 return y;
}

f(3,4);
```

Lösung:

```
int f(int accu, int x, int n) {
 if (n == 0) return accu;
 else return f(accu * x, x, n-1);
}

f(1,3,4);
```

Beispiel

Beispiel für eine Klausurfrage

Gegeben ist eine iterative Implementierung zur Berechnung von $f(x, n) = x^n$.

Schreiben Sie eine endrekursive Variante zur Berechnung dieser Funktion.

```
int f(int x, int n) {
 int y = x;
 for (int i=1; i<n; i++) y = y * x;
 return y;
}

f(3,4);
```

Lösung:

```
int f(int accu, int x, int n) {
 if (n == 0) return accu;
 else return f(accu * x, x, n-1);
}

f(1,3,4);
```

Übungsaufgabe 1

Schreiben Sie ein C-Programm *tictactoe* mit dem das Spiel "Tic Tac Toe" gespielt werden kann. Zwei Spieler ('x' und 'o') sollen über die Kommandozeile auf einem 3·3-Feld abwechselnd setzten können, z.B. indem das Feld zuerst über die horizontale Position (1 . . . 3) und danach über die vertikale Position (1 . . . 3) angegeben wird. Das Programm erkennt, sobald ein Spieler gewonnen hat und gibt den aktuellen Stand des Spielfeldes geeignet grafisch aus.

Beispielaufruf:

```
> tictactoe
> 1. Zug 'x' horizontal: 2
> 1. Zug 'x' vertikal: 1
> ---------
> |  |x|  |
> ---------
> |  |  |  |
> ---------
> |  |  |  |
> ---------
```

Übungsaufgabe 2

Schreiben Sie ein C-Programm *sortbookdata* zum Einlesen, Sortieren und Speichern der Datei *bookdata.xml*[6]. Die Daten sollen in einer geeigneten Datenstruktur zwischengespeichert werden. Das Programm nimmt drei Argumente entgegen: Quelldatei, Zieldatei, Sortierung. Sortierung kann *Titel*, oder *Autor* sein. Die Zieldatei soll nach dem im Feld Sortierung angegebenen Attribut sortiert sein.

Beispielaufruf:

```
> sortbookdata bookdata.xml out.xml Autor
```

[6] `http://www.users.fh-salzburg.ac.at/~fhs31108/teaching/bookdata.xml`

Übungsaufgabe 3

Schreiben Sie ein C-Programm `feistel` zur Ver- und Entschlüsselung einer Buchstabenfolge mit der Feistel-Chiffre. Diese Chiffre ist ein einfaches symmetrisches Verfahren. Der Klartext M wird in zwei gleich große Blöcke $M = (L_1, R_1)$ aufgeteilt, jede Runde hat einen Rundenschlüssel K_i. Die $i = 1 \ldots n$ Runden für die Verschlüsselung sind wie folgt:

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

Recherchieren Sie in der Literatur eine geeignete Funktion F sowie den Algorithmus zur Entschlüsselung.

Beispielaufrufe:

```
> feistel -enc KLARTEXT GEHEIMER_SCHLUESSEL
> feistel -dec CHIFFRE GEHEIMER_SCHLUESSEL
```

Übungsaufgabe 4

Schreiben Sie ein C-Programm `calc`, das die folgende über EBNF definierte Grammatik parsen und das Ergebnis eines Textes in dieser Grammatik berechnen kann. Lesen Sie dazu einen Text in dieser Grammatik über eine Datei oder über `stdin` ein, die Ausgabe kann auf `stdout` erfolgen.

Grammatik:

```
DIGIT = "0" | "1" | ... | "9";
SIGN = "+" | "-";
NUM = [SIGN] DIGIT {DIGIT};
REGISTER = "A" | "B" | ... | "Z";

ADD = "ADD " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
SUB = "SUB " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
MUL = "MUL " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
DSP = "DSP " REGISTER;
COMMAND = (ADD | SUB | MUL | DSP) "\n";
PROGRAM = {COMMAND};
```

Übungsaufgabe 5

Schreiben Sie ein C-Programm `exprand`, das N exponentialverteilte Pseudozufallszahlen für ein gegebenes λ erzeugt und diese zeilenweise in eine Datei schreibt. Verwenden Sie dazu die Inversionsmethode und lesen sie N und λ als Kommandozeilenargumente ein.

Die Exponentialverteilung hat folgende Verteilungsfunktion:

$$F(x) = 1 - \exp(\lambda x)$$

Überprüfen Sie, ob die Pseudozufallszahlen der gewünschten Verteilung folgen, indem Sie die erzeugte Datei in (z.B.) Matlab einlesen und die Verteilung visualisieren (ohne statistischen Test).

Übungsaufgabe 6

Schreiben Sie ein C-Programm `hashtest`, das eine Hash-Tabelle mit N einträgen für Tupel aus (österreichischer) Postleitzahl und Ortsname implementiert. Die Postleitzahl wird dabei als Schlüssel verwendet. Recherchieren Sie in der Literatur und entwerfen Sie eine geeignete Hash-Funktion $H(\cdot)$, um eine Postleitzahl auf einen Index $1, \ldots, N$ abzubilden. Lesen Sie eine Datei mit Postleitzahlen[7] ein und bauen Sie eine Hash-Tabelle mit $N = 1000$ Einträgen auf. Probieren Sie verschiedene Varianten der Funktion H und bewerten Sie deren Qualität anhand der Verteilung der Einträge und der Anzahl der auftretenden Kollisionen.

```
5020 -> Salzburg
1010 -> Wien
...
```

[7] `https://www.post.at/geschaeftlich_werben_produkte_und_services_adressen_postlexikon.php`, Download des PLZ-Vrezeichnisses als Excel-Datei, die ersten beiden Spalten als CSV-Datei speichern und im Programm einlesen.

Algorithmen und Datenstrukturen

Einführung

Datenstrukturen

- Speichern Daten strukturiert
- Effizienter und einfacher Zugriff

Algorithmen

- Verändern oder verarbeiten Daten
- Arbeiten mit Datenstrukturen

Wichtige Datenstrukturen

1. Stack
2. List
3. Tree
4. Queue
5. Hash-Table

Einführung

Datenstrukturen

- Speichern Daten strukturiert
- Effizienter und einfacher Zugriff

Algorithmen

- Verändern oder verarbeiten Daten
- Arbeiten mit Datenstrukturen

Wichtige Datenstrukturen

1. Stack
2. List
3. Tree
4. Queue
5. Hash-Table

Einführung

Datenstrukturen

- Speichern Daten strukturiert
- Effizienter und einfacher Zugriff

Algorithmen

- Verändern oder verarbeiten Daten
- Arbeiten mit Datenstrukturen

Wichtige Datenstrukturen

1. Stack
2. List
3. Tree
4. Queue
5. Hash-Table

Stack

Zwei Operationen

```
push(.);
pop();
```

Einfacher UPN-Rechner mit einem Stack (Pseudocode):

```
stack = array[10]; len = 0;
push(val) { stack[len++] = val; }
pop() {
  if(len > 1) {
    val = stack[--len];
    if (val == '+') push(pop() + pop());
    else if (val == '*') push(pop() * pop());
    else return val;
  } else {
    print(pop());
  }
}
main() {
 push(1); push(3); push(+); push(2); push(*); pop();
}
```

Ausgabe: 8

Stack

Zwei Operationen

```
push(.);
pop();
```

Einfacher UPN-Rechner mit einem Stack (Pseudocode):

```
stack = array[10]; len = 0;
push(val) { stack[len++] = val; }
pop() {
  if(len > 1) {
    val = stack[--len];
    if (val == '+') push(pop() + pop());
    else if (val == '*') push(pop() * pop());
    else return val;
  } else {
    print(pop());
  }
}
main() {
 push(1); push(3); push(+); push(2); push(*); pop();
}
```

Ausgabe: 8

Liste

Lineare, dynamische Datenstruktur, Elemente zeigen aufeinander, jedes Element hat genau einen Nachfolger

- einfach verkettet (Zeiger zum Nachfolger)
- doppelt verkettet (Zeiger zum Vorgänger und Nachfolger)

```
struct listitem {
 Data* data;
 struct listitem* next;
 struct listitem* prev;
}
typedef struct listitem ListItem;
```

Eigenschaften:

- Effizientes Einfügen am Anfang oder Ende der Liste (mit Ende-Zeiger)
- Ineffizientes Suchen

Skip-Liste

Aufbau wie eine einfache Liste, aber mit zusätzlichen Skip-Pointern, d.h. Zeiger von einem Element zu einem anderen, wobei Einträge übersprungen werden können.

```
struct listitem {
 Data* data;
 struct listitem* next;
 struct listitem* prev;
 struct listitem* skipPointer;
}
typedef struct listitem ListItem;
```

Baum I

Bildet Daten in einer Hierarchie ab

- Knoten und Kanten als Verweis auf Knoten
- Jeder Knoten hat ein Elternelement
- Spezialfall: Binärbaum (jeder Knoten hat max. 2 Kinder)

```
struct node {
 Data* data;
 struct nod* left;
 struct nod* right;
}
typedef struct node Node;
```

Eigenschaften:

- Effizientes Suchen mit Höhe h: $O(h)$
- Baum kann "schief" oder balanciert sein
- Einfache Traversion

Baum II

Für iteratives Traversieren Erweiterung um Eltern-Zeiger:

```
struct node {
 Data* data;
 struct nod* left;
 struct nod* right;
 struct nod* parent;
}
typedef struct node Node;
```

Wahl des Wurzelknotens

Unbalancierte/entartete Bäume: Als Wurzel des Baumes sollte kein sehr kleiner oder sehr großer Wert verwendet werden, da Bäume sonst schief werden

ggf. muss ein Baum reorganisiert werden

Baum II

Für iteratives Traversieren Erweiterung um Eltern-Zeiger:

```
struct node {
 Data* data;
 struct nod* left;
 struct nod* right;
 struct nod* parent;
}
typedef struct node Node;
```

Wahl des Wurzelknotens

Unbalancierte/entartete Bäume: Als Wurzel des Baumes sollte kein sehr kleiner oder sehr großer Wert verwendet werden, da Bäume sonst schief werden

ggf. muss ein Baum reorganisiert werden

Traversieren eines Baums

Systematisches Untersuchen der Knoten, leicht rekursiv implementierbar

- Preorder

```
void traverse(NODE* node) {
 // node->data untersuchen
 traverse(node->left);
 traverse(node->right);
}
```

- Inorder

```
void traverse(NODE* node) {
 traverse(node->left);
 // node->data untersuchen
 traverse(node->right);
}
```

- Postorder

```
void traverse(NODE* node) {
 traverse(node->left);
 traverse(node->right);
 // node->data untersuchen
}
```

Warteschlange

Häufig benötigte Datenstruktur zum Zwischenspeichern von Daten in einer Reihenfolge (siehe Betriebssysteme, Warten auf eine Ressource). Die Implementierung kann als Ringpuffer erfolgen.

Im Gegensatz zu Listen erfolgt kein wahlfreier Zugriff!

Ringpuffer

- Feste Größe mit N Plätzen
- `enqueue`: Zeigt auf nächsten freien Platz zum Einfügen eines Elements
- `dequeue`: Zeigt auf das vorderste Element

Bei Überlauf: Überscheiben der Daten, Ausnahmesituation signalisieren oder zusätzlichen Speicher anfordern

Auch Priority-Queue: Elemente haben einen Prioritäts-Wert der die Reihenfolge der Abarbeitung bestimmt

Warteschlange

Häufig benötigte Datenstruktur zum Zwischenspeichern von Daten in einer Reihenfolge (siehe Betriebssysteme, Warten auf eine Ressource). Die Implementierung kann als Ringpuffer erfolgen.

Im Gegensatz zu Listen erfolgt kein wahlfreier Zugriff!

Ringpuffer

- Feste Größe mit N Plätzen
- `enqueue`: Zeigt auf nächsten freien Platz zum Einfügen eines Elements
- `dequeue`: Zeigt auf das vorderste Element

Bei Überlauf: Überscheiben der Daten, Ausnahmesituation signalisieren oder zusätzlichen Speicher anfordern

Auch Priority-Queue: Elemente haben einen Prioritäts-Wert der die Reihenfolge der Abarbeitung bestimmt

Warteschlange

Häufig benötigte Datenstruktur zum Zwischenspeichern von Daten in einer Reihenfolge (siehe Betriebssysteme, Warten auf eine Ressource). Die Implementierung kann als Ringpuffer erfolgen.

Im Gegensatz zu Listen erfolgt kein wahlfreier Zugriff!

Ringpuffer

- Feste Größe mit N Plätzen
- `enqueue`: Zeigt auf nächsten freien Platz zum Einfügen eines Elements
- `dequeue`: Zeigt auf das vorderste Element

Bei Überlauf: Überscheiben der Daten, Ausnahmesituation signalisieren oder zusätzlichen Speicher anfordern

Auch Priority-Queue: Elemente haben einen Prioritäts-Wert der die Reihenfolge der Abarbeitung bestimmt

Hash-Table I

Implementiert ein assoziatives Array, d.h. speichert Key-Value-Pairs. Mit einer Hash-Funktion H wird ein eindeutiger Platz für ein Element m in einer Liste mit N Elementen berechnet: $p = H(m) \mod N$
Herausforderung: Geeignete Wahl der Hash-Funktion, sodass (idealerweise)

- keine Kollisionen entstehen
- der gesamte Platz in der Tabelle gleichmäßig ausgenutzt wird

Strategie bei Kollision:
(i) Wahl einer anderen, freien Stelle (Sondierung)
- linear: Verschieben um festes Intervall $H(m) + \Delta \mod N$
- quadratisch: Verschieben um Quadrat des Intervalls $H(m) + \Delta^2 \mod N$ (vereinfacht)
- hash: Eine Hashfunktion liefert das Intervall $H(m) + H'(m)\Delta \mod N$

(ii) Erstellen von Buckets
- Ein Key kann mehrere Values aufnehmen

Hash-Table I

Implementiert ein assoziatives Array, d.h. speichert Key-Value-Pairs. Mit einer Hash-Funktion H wird ein eindeutiger Platz für ein Element m in einer Liste mit N Elementen berechnet: $p = H(m) \mod N$
Herausforderung: Geeignete Wahl der Hash-Funktion, sodass (idealerweise)

- keine Kollisionen entstehen
- der gesamte Platz in der Tabelle gleichmäßig ausgenutzt wird

Strategie bei Kollision:
(i) Wahl einer anderen, freien Stelle (Sondierung)

- linear: Verschieben um festes Intervall $H(m) + \Delta \mod N$
- quadratisch: Verschieben um Quadrat des Intervalls $H(m) + \Delta^2$ $\mod N$ (vereinfacht)
- hash: Eine Hashfunktion liefert das Intervall $H(m) + H'(m)\Delta \mod N$

(ii) Erstellen von Buckets

- Ein Key kann mehrere Values aufnehmen

Hash-Table II

Geeignete Wahl von H: Beispiel Matrikelnummer 1410555001, es stehen N Plätze zur Verfügung

$$H(m) = m \mod N$$

z.B.: $N = 997$ (Primzahl!), da mit $\Delta < N$ und $k_i = z + i\Delta$ Kollisionen auftreten wenn

$$k_i \mod N = k_j \mod N \Leftrightarrow$$

$$z + i\Delta \mod N = z + j\Delta \mod N \Leftrightarrow$$

$$i = j + mN, m \in \mathbb{Z}$$

(Siehe `http://www.cs.vu.nl/~tcs/ds/lecture6.pdf`)

Hash-Table II

Geeignete Wahl von H: Beispiel Matrikelnummer 1410555001, es stehen N Plätze zur Verfügung

$$H(m) = m \mod N$$

z.B.: $N = 997$ (Primzahl!), da mit $\Delta < N$ und $k_i = z + i\Delta$ Kollisionen auftreten wenn

$$k_i \mod N = k_j \mod N \Leftrightarrow$$

$$z + i\Delta \mod N = z + j\Delta \mod N \Leftrightarrow$$

$$i = j + mN, m \in \mathbb{Z}$$

(Siehe `http://www.cs.vu.nl/~tcs/ds/lecture6.pdf`)

Hash-Table III

Weitere Varianten für die Wahl von H:

- Mid-square method: $H(12) = 4$: $12 \cdot 12 = 1\mathbf{4}4$, $H(5020) = 200$: $5020 \cdot 520 = 25\mathbf{200}400$
- Multiplicative method: $H(m) = \lfloor N(mA - \lfloor mA \rfloor) \rfloor$, $A \in [0, 1]$
- ...

O-Notation

Die O-Notation beschreibt die asymptotische obere Grenze. Für eine Funktion $g(n)$ notiert $O(g(n))$ die Menge an Funktionen

$$\begin{aligned}\Theta(g(n)) = \{f(n) : &\text{ es existieren positive Konstanten } c_1, c_2 \text{ und } n_0, \text{ so dass} \\ &0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{für alle} n \geq n_0\}\end{aligned} \tag{1}$$

(Cormen T. H., Leiserson, C. E., Rivestm R. L., Stein, C., Introduction to Algorithms, 2nd Ed., 2003, The MIT Press.)

Komplexitätsvergleich

Operation	Array	List	Tree (balanced)	Hash-Table
Einfügen/Löschen am Anfang		$O(1)$	$O(\log n)$	$O(1)$ bis $O(n)$
Einfügen/Löschen am Ende		$O(1)$	$O(\log n)$	$O(1)$ bis $O(n)$
Einfügen/Löschen in der Mitte		$O(1)$	$O(\log n)$	$O(1)$ bis $O(n)$
Suchen	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$ bis $O(n)$
Zugriff auf beliebiges Element	$O(1)$	$O(n)$	$O(n)$	$O(n)$

Tabelle: Komplexitäsvergleich, n ist die Anzahl der Elemente in der Datenstruktur (`http://bigocheatsheet.com/`)

Bubble-Sort

Einfacher (wenig effizienter $O(n^2)$ im Average- und Worst-Case) Sortieralgorithmus:

1. Vergleiche zwei Elemente, wenn rechts größer links, dann tauschen
2. Gehe eins weiter und wiederhole Schritt (1), wenn am Ende der Liste angelangt, starte von vorne
3. Wiederhole Schritte (1-2) bis keine Elemente mehr getauscht werden

Eigenschaften:

- Bubble-Sort arbeitet in-place
- für kleine, großteils bereits sortierte Eingaben sinnvoll

Bubble-Sort

Einfacher (wenig effizienter $O(n^2)$ im Average- und Worst-Case) Sortieralgorithmus:

1. Vergleiche zwei Elemente, wenn rechts größer links, dann tauschen
2. Gehe eins weiter und wiederhole Schritt (1), wenn am Ende der Liste angelangt, starte von vorne
3. Wiederhole Schritte (1-2) bis keine Elemente mehr getauscht werden

Eigenschaften:

- Bubble-Sort arbeitet in-place
- für kleine, großteils bereits sortierte Eingaben sinnvoll

Quick-Sort

Schnellerer ($O(n \cdot \log(n))$ im Average- und $O(n^2)$ Worst-Case) Sortieralgorithmus nach dem Prinzip *divide and conquer*:

1. Wähle ein Pivot-Element
2. Ordne alle Elemente kleiner dem Pivot-Element davor an und alle Element größer dem Pivot-Element danach (Partitionierung)
3. Wiederhole Schritte (1-2) jeweils mit den Elementen links dem Pivot-Element und den Elementen rechts dem Pivot-Element
4. Wiederhole Schritte (1-3) bis jedes Element Pivot-Element war

Eigenschaften:

- Quick-Sort arbeitet in-place
- unterschiedliche Strategien für die Partitionierung

Quick-Sort

Schnellerer ($O(n \cdot \log(n))$ im Average- und $O(n^2)$ Worst-Case) Sortieralgorithmus nach dem Prinzip *divide and conquer*:

1. Wähle ein Pivot-Element
2. Ordne alle Elemente kleiner dem Pivot-Element davor an und alle Element größer dem Pivot-Element danach (Partitionierung)
3. Wiederhole Schritte (1-2) jeweils mit den Elementen links dem Pivot-Element und den Elementen rechts dem Pivot-Element
4. Wiederhole Schritte (1-3) bis jedes Element Pivot-Element war

Eigenschaften:

- Quick-Sort arbeitet in-place
- unterschiedliche Strategien für die Partitionierung

Komplexitätsbewertung

Analyse von Algorithmen I

Ziel: Bestimmen der Ressourcen (Rechenzeit) die ein Algorithmus benötigt
Annahme: *Random Access Machine*, zeilenweises Ausführen des Programms, keine Parallelisierung
(Cormen T. H., Leiserson, C. E., Rivest R. L., Stein, C., Introduction to Algorithms, 2nd Ed., 2003, The MIT Press.)

Insertion-Sort(A)

for $j \leftarrow 2 \rightarrow length[A]$ **do**
  $key \leftarrow A[j]$
  $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1..j-1]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] > key$ **do**
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  end while
  $A[i+1] \leftarrow key$
end for

Analyse von Algorithmen I

Ziel: Bestimmen der Ressourcen (Rechenzeit) die ein Algorithmus benötigt
Annahme: *Random Access Machine*, zeilenweises Ausführen des Programms, keine Parallelisierung
(Cormen T. H., Leiserson, C. E., Rivest R. L., Stein, C., Introduction to Algorithms, 2nd Ed., 2003, The MIT Press.)

Insertion-Sort(A)

for $j \leftarrow 2 \rightarrow length[A]$ **do**
  $key \leftarrow A[j]$
  ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] > key$ **do**
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  end while
  $A[i+1] \leftarrow key$
end for

Analyse von Algorithmen II

Insertion-Sort(A)

for $j \leftarrow 2$ **to** $length[A]$ **do**	$c_1,\ n$
$key \leftarrow A[j]$	$c_2,\ n-1$
	$c_3,\ 0$
$i \leftarrow j-1$	$c_4,\ n-1$
while $i > 0$ and $A[i] > key$ **do**	$c_5,\ \sum_{j=1}^{n} t_j$
$A[i+1] \leftarrow A[i]$	$c_6,\ \sum_{j=1}^{n}(t_j-1)$
$i \leftarrow i-1$	$c_7,\ \sum_{j=1}^{n}(t_j-1)$
end while	
$A[i+1] \leftarrow key$	$c_8,\ n-1$
end for	

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

$$n = length[A]$$

Analyse von Algorithmen II

Insertion-Sort(A)

for $j \leftarrow 2$ **to** $length[A]$ **do**	$c_1,\ n$
$key \leftarrow A[j]$	$c_2,\ n-1$
	$c_3,\ 0$
$i \leftarrow j-1$	$c_4,\ n-1$
while $i > 0$ and $A[i] > key$ **do**	$c_5,\ \sum_{j=1}^{n} t_j$
$A[i+1] \leftarrow A[i]$	$c_6,\ \sum_{j=1}^{n}(t_j-1)$
$i \leftarrow i-1$	$c_7,\ \sum_{j=1}^{n}(t_j-1)$
end while	
$A[i+1] \leftarrow key$	$c_8,\ n-1$
end for	

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

$n = length[A]$

Analyse von Algorithmen III

Die Laufzeit ist abhängig von der *Art* der Eingabe

- Best case: Array A ist bereits sortiert
  $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = an + b$
- Worst case: Array A absteigend sortiert
  $T(n) = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8) = an^2 + bn + c$

Ausführung in konstanter Zeit

Wichtig für sicherheitskritische Anwendungen: Algorithmus terminiert in konstanter Zeit, unabhängig von Eingabe und Ergebnis

```
int check_password(const char* input, const char* secret) {
 for (int i=0; i < strlen(input) && i < strlen(secret); i++) {
  if (input[i] != secret[i])
   return 0;
 }
 return 1;
}
```

```
int check_password(const char* input, const char* secret) {
 int ret = 1;
 for (int i=0; i < strlen(input) && i < strlen(secret); i++)
  ret &= !!(input[i] != secret[i]);
 return ret;
}
```

Wie sieht ein möglicher Angriff bei obigen Varianten aus?

Ausführung in konstanter Zeit

Wichtig für sicherheitskritische Anwendungen: Algorithmus terminiert in konstanter Zeit, unabhängig von Eingabe und Ergebnis

```
int check_password(const char* input, const char* secret) {
 for (int i=0; i < strlen(input) && i < strlen(secret); i++) {
  if (input[i] != secret[i])
   return 0;
 }
 return 1;
}
```

```
int check_password(const char* input, const char* secret) {
 int ret = 1;
 for (int i=0; i < strlen(input) && i < strlen(secret); i++)
  ret &= !!(input[i] != secret[i]);
 return ret;
}
```

Wie sieht ein möglicher Angriff bei obigen Varianten aus?

Betriebssysteme

Überblick

Die Inhalten und Abbildungen in diesem Abschnitt sind mit freundlicher Genehmigung übernommen aus der Vorlesung *Betriebssysteme* von DI(FH) DI Roland J. Graf, MSC(GIS)
`http://www.users.fh-salzburg.ac.at/~rgraf/`

- Definition, Aufgaben und Komponenten
- Architektur und Strukturen
- Systemaufrufe und Interrupts
- Speicherverwaltung
- Prozesse und Threads
- Scheduling

Überblick

Die Inhalten und Abbildungen in diesem Abschnitt sind mit freundlicher Genehmigung übernommen aus der Vorlesung *Betriebssysteme* von DI(FH) DI Roland J. Graf, MSC(GIS)
`http://www.users.fh-salzburg.ac.at/~rgraf/`

- Definition, Aufgaben und Komponenten
- Architektur und Strukturen
- Systemaufrufe und Interrupts
- Speicherverwaltung
- Prozesse und Threads
- Scheduling

Definition

Definition nach DIN 44300:
Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Engere Definition, z.B. nach A. S. Tanenbaum:
Editoren, Compiler, Assembler, Binder und Kommandointerpreter sind definitiv nicht Teil des Betriebssystems, auch wenn sie bedeutsam und nützlich sind.

Definition

Definition nach DIN 44300:
Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Engere Definition, z.B. nach A. S. Tanenbaum:
Editoren, Compiler, Assembler, Binder und Kommandointerpreter sind definitiv nicht Teil des Betriebssystems, auch wenn sie bedeutsam und nützlich sind.

Aufgaben und Klassifikation

1. Ressourcenverwaltung
2. Prozessmanagement
3. Abstraktion der Hardware
4. Reduktion der Komplexität
5. Schutz des Systems
6. Fehlerbehandlung
7. Bereitstellen einheitlicher Schnittstellen (z.B. POSIX, IEEE Std.1003.1-2008)

Mögliche Klassifikationen: batch processing, interactive/dialog processing, network, realtime, mainframe, embedded, single/multi tasking, single/multi user, ...

Aufgaben und Klassifikation

1. Ressourcenverwaltung
2. Prozessmanagement
3. Abstraktion der Hardware
4. Reduktion der Komplexität
5. Schutz des Systems
6. Fehlerbehandlung
7. Bereitstellen einheitlicher Schnittstellen (z.B. POSIX, IEEE Std.1003.1-2008)

Mögliche Klassifikationen: batch processing, interactive/dialog processing, network, realtime, mainframe, embedded, single/multi tasking, single/multi user, ...

Komponenten

Prozessor (CPU):

- holt Befehle aus dem Speicher und führt sie aus
- hat fest definierten Befehlssatz
- interne Register zur Verarbeitung, z.B. Akkumulator, Adressregister, Instruction Pointer, Status Register

Speicher

Unterteilung nach Permanenz, Zugriffsgeschwindigkeit und Kapazität

- Register ($<$ 1 kB, ca. 1ns)
- Cache (4 MB, ca. 2ns)
- Arbeitsspeicher (einige GB, ca. 10ns)
- Fetsplatten (einige GB – einige TB, einige ms)

Ein-/Ausgabegeräte

Controller

- steuert das Gerät, besitzt Schnittstelle zum Betriebssystem
- verdeckt und abstrahiert die Schnittstelle zum Gerät, setzt Befehle um

Gerät

- bietet die Funktionalität der Hardware, meist einfache Schnittstellen
- wird vom Controller verdeckt und angesprochen

Zugriff auf Controller

Busy Waiting

1. Programm benutzt Systemaufruf
2. System ruft Treiber auf
3. Treiber wartet in Endlosschleife auf die Daten

→ Gerät und CPU sind blockiert

Interrupt

1. Treiber steuert den Controller an und wartet auf einen Interrupt der das Arbeitsende signalisiert

→ Gerät ist blockiert, CPU nicht

Direct Memory Access (DMA)

1. Spezieller Chip zwischen Controller und Speicher
2. DMA-Chip löst Interrupt aus

Zugriff auf Controller

Busy Waiting

1. Programm benutzt Systemaufruf
2. System ruft Treiber auf
3. Treiber wartet in Endlosschleife auf die Daten

→ Gerät und CPU sind blockiert

Interrupt

1. Treiber steuert den Controller an und wartet auf einen Interrupt der das Arbeitsende signalisiert

→ Gerät ist blockiert, CPU nicht

Direct Memory Access (DMA)

1. Spezieller Chip zwischen Controller und Speicher
2. DMA-Chip löst Interrupt aus

Zugriff auf Controller

Busy Waiting

1. Programm benutzt Systemaufruf
2. System ruft Treiber auf
3. Treiber wartet in Endlosschleife auf die Daten

$\rightarrow$ Gerät und CPU sind blockiert

Interrupt

1. Treiber steuert den Controller an und wartet auf einen Interrupt der das Arbeitsende signalisiert

$\rightarrow$ Gerät ist blockiert, CPU nicht

Direct Memory Access (DMA)

1. Spezieller Chip zwischen Controller und Speicher
2. DMA-Chip löst Interrupt aus

Betriebssystemkomponenten: Bootloader

Bootloader

- wird durch die Firmware (z.B. BIOS) geladen
- startet das Betriebssystem
- Beim IBM-PC wird der Bootloader aus dem master boot record geladen, d.h. max. 446 Bytes

Varianten

- Multistage Bootloader: Wenn Programm nicht im Bootsektor platz hat
- Chain Loader: z.B. für Multi-Boot, Systemauswahl, etc.

Bekannte Bootloader: Bootmgr, GRUB (Grand Unified Bootloader)

Betriebssystemkomponenten: Bootloader

Bootloader

- wird durch die Firmware (z.B. BIOS) geladen
- startet das Betriebssystem
- Beim IBM-PC wird der Bootloader aus dem master boot record geladen, d.h. max. 446 Bytes

Varianten

- Multistage Bootloader: Wenn Programm nicht im Bootsektor platz hat
- Chain Loader: z.B. für Multi-Boot, Systemauswahl, etc.

Bekannte Bootloader: Bootmgr, GRUB (Grand Unified Bootloader)

Betriebssystemkomponenten: Kernel

Kernel

- verwaltet die Hardware des Computers (Interrupts, Zuteilung des Adressraums)
- verwaltet Programme zum Starten und Ausführen des Betriebssystems (Prozesswechsel/Scheduling, Prozesssynchronisation)
- verwaltet die Konfiguration des Systems

Linux-Kernel 0.0.1: `https://www.kernel.org/pub/linux/kernel/Historic/linux-0.01.tar.gz`

Betriebssystemkomponenten: Kernel

Kernel

- verwaltet die Hardware des Computers (Interrupts, Zuteilung des Adressraums)
- verwaltet Programme zum Starten und Ausführen des Betriebssystems (Prozesswechsel/Scheduling, Prozesssynchronisation)
- verwaltet die Konfiguration des Systems

Linux-Kernel 0.0.1: `https://www.kernel.org/pub/linux/kernel/Historic/linux-0.01.tar.gz`

Betriebssystemkomponenten: Gerätetreiber

Device Driver

- Programm, das Hardware steuert (z.B. Netzwerk, Grafik, Audio, ...)
- eine standardisierte Softwareschnittstelle des Betriebssystems stellt die Austauschbarkeit von Hardware sicher

Programming Libraries

- Sammlung von Hilfsmodulen und -funktionen

Utilities

Hilfsprogramme zur Konfiguration des Systems

- Benutzer, Rechte
- Hardware
- Festplatten, Partitionen
- Netzwerkverbindung
- ...

Betriebssystemkomponenten: Gerätetreiber

Device Driver

- Programm, das Hardware steuert (z.B. Netzwerk, Grafik, Audio, ...)
- eine standardisierte Softwareschnittstelle des Betriebssystems stellt die Austauschbarkeit von Hardware sicher

Programming Libraries

- Sammlung von Hilfsmodulen und -funktionen

Utilities

Hilfsprogramme zur Konfiguration des Systems

- Benutzer, Rechte
- Hardware
- Festplatten, Partitionen
- Netzwerkverbindung
- ...

Betriebssystemkomponenten: Gerätetreiber

Device Driver

- Programm, das Hardware steuert (z.B. Netzwerk, Grafik, Audio, ...)
- eine standardisierte Softwareschnittstelle des Betriebssystems stellt die Austauschbarkeit von Hardware sicher

Programming Libraries

- Sammlung von Hilfsmodulen und -funktionen

Utilities

Hilfsprogramme zur Konfiguration des Systems

- Benutzer, Rechte
- Hardware
- Festplatten, Partitionen
- Netzwerkverbindung
- ...

Architektur

Die Strukturierung erfolgt in Schichten für eine zunehmende Abstraktion der Hardware

- unterste Schicht verwaltet die realen Betriebsmittel (BIOS)
- jede Schicht bietet der darüberliegenden Dienste/Funktionen an
- Kommunikation mit benachbarten Schichten über Schnittstellen/Protokolle

Single User und Single Tasking Systeme
Der Zugriff über Schichten hinweg kann erlaubt sein, z.B. direktes Ansteuern von Grafiktreiber oder Drucker

Multi User und Multi Tasking Systeme
Schutz und Integrität der Ressourcen (CPU, Speicher, etc.) muss gewährleistet sein und Applikationen dürfen sich nicht beeinflussen

Kernel

Der Kernel liegt im Hauptspeicher und umfasst die am häufigsten verwendeten Funktionen

- Monolithischer Kernel
- Mikrokernel
- Hybridkernel

Kernel: Monolithischer Kernel

Verwendung bei Unix/Linux, DOS, OS/2: Das Betriebssystem ist eine Menge von Prozeduren die sich gegenseitig aufrufen können. Der Kernel ist minimal modularisiert und beinhaltet Speicher-, Prozessmanagement und Treiber.

- Keine Abgrenzung, Fehler wirken sich auf den ganzen Kernel/das Betriebssystem aus
- Einfach, Performant, wenig Overhead bei Funktionsaufrufen

Kernel: Mikrokernel

Verwendung bei Minix, Mach: Minimaler Kernel für Speicher- und Prozessmanagement und Mittel für Synchronoisation und Kommunikation. Betriebssystemkomponenten sind eigene Prozesse mit eigenem Speicherbereich (z.B. User Mode)

- Overhead für Kommunikation und Kontextwechsel, Synchronisationsaufwand und oft aufwendigere Treiber
- Treiber im User Mode, stabiler und einfacher Kernel mit wenig Seiteneffekten

Kernel: Hybridkernel

Verwendung bei Windows, MAC OS X, BeOS: Mischung aus Mikrokernel und monolithischem Kernel mit einzelnen Komponenten im Kernel.

- Versucht die Vorteile beider Ansätze zu vereinen
- Auch Makrokernel genannt

Privilegierung und Ringe

Bei modernen Betriebssystemen läuft ein Prozess mit einer bestimmten Privilegierung. Das wird durch die CPU unterstützt.
Ein Prozess in einem Ring hat

1. einen eingeschränkten Befehlssatz (CPU)
2. einen eingeschränkten Zugriff auf die Hardware (z.B. Speicher)

x86-Prozessoren unterstützen 4 Ringe (0–3). Verwendet werden meist nur Ring 0 und 3.

- Ring 0: Kernel Mode, uneingeschränkter Hardwarezugriff
- Ring (1–)3: User Mode, keine Beeinflussung anderer Prozesse

Privilegierung und Ringe / Kontextwechsel

Nach dem Start lädt die CPU den Kernel in den privilegierten Modus.
Programme laufen in den äußeren Ringen.
(Virtualisierung: Betriebssystem wird in Ring 1 verschoben, oder eigene CPUs mit Ring -1)

System Call

Ein Aufruf einer Betriebssystemfunktion aus einer Applikation → Umschalten vom User Mode in den Kernel Mode (Kontextwechsel)

Privilegierung und Ringe / Kontextwechsel

Nach dem Start lädt die CPU den Kernel in den privilegierten Modus.
Programme laufen in den äußeren Ringen.
(Virtualisierung: Betriebssystem wird in Ring 1 verschoben, oder eigene CPUs mit Ring -1)

System Call
Ein Aufruf einer Betriebssystemfunktion aus einer Applikation $\rightarrow$ Umschalten vom User Mode in den Kernel Mode (Kontextwechsel)

Kontextwechsel I

Der Kontextwechsel erfolgt hardwareunterstützt über Interrupts:

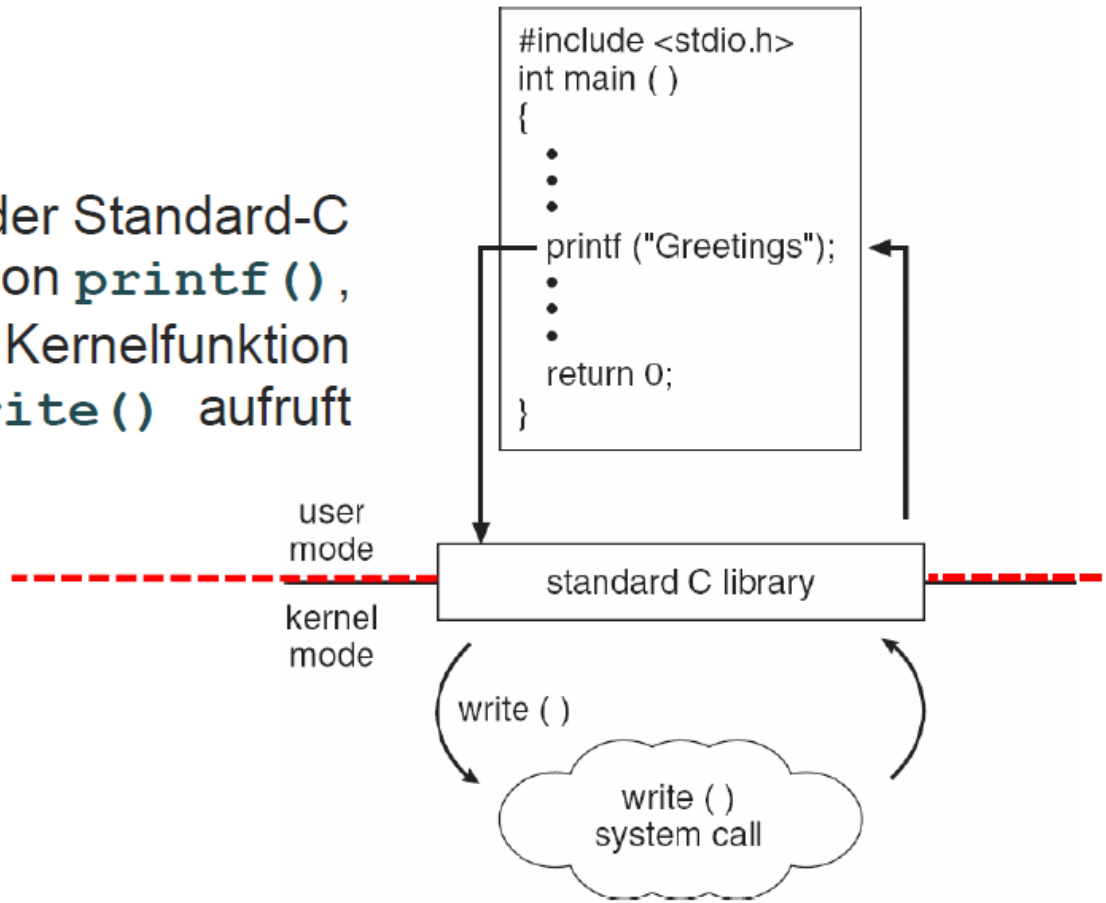
1. Parameter für den syscall werden am Stack oder in ein Register der CPU gelegt
2. Ein Softwareinterrupt wird ausgelöst (z.B.: DOS, INT 21h)
3. CPU unterbricht die laufende Funktion, Instruktionsadresse und Status werden am Stack gespeichert
4. CPU springt zur Einsprungadresse ins Betriebssystem (wird beim Initialisieren festgelegt)
5. Betriebssystem arbeitet die Interrupt-Routine ab und kehrt zur vorherigen Funktion zurück

Ein Sytemaufruf unterbricht immer die aktuelle Funktion (trap door), Kontextwechsel sind teuer!

Kontextwechsel II

Beispiel

Aufruf der Standard-C Library Funktion **`printf()`**, welche den Kernelfunktion **`write()`** aufruft



Prozessspeicher

Der Speicher eines Prozesses ist aufgeteilt in

1. Textsegment (Instruktionen in Maschinencode)
2. Datensegment (globale Variablen, Konstanten)
3. BSS-Segment (Block Started by Symbol, Speicher für nicht initialisierte globale Variablen)
4. Prozess-Stack für Zugriff im User Mode (normale Funktionsaufrufe, Parameter, Rücksprungadresse)
5. Prozess-Stack für Zugriff im Kernel Mode

Virtueller Speicher

Probleme bei gleichzeitiger Ausführung mehrerer Programme

- Hauptspeicher muss geteilt werden
- Auslagern von Hauptspeicher auf die Festplatte ($\rightarrow$ neuer Speicherbereich beim erneuten Laden)

Lösung

Jedes Programm hat einen virtuellen Speicherbereich. Das Betriebssystem übersetzt mittels Relozierungstabellen die realen Adressen in virtuelle Adressen (relocation).

Virtueller Speicher

Probleme bei gleichzeitiger Ausführung mehrerer Programme

- Hauptspeicher muss geteilt werden
- Auslagern von Hauptspeicher auf die Festplatte ($\rightarrow$ neuer Speicherbereich beim erneuten Laden)

Lösung
Jedes Programm hat einen virtuellen Speicherbereich. Das Betriebssystem übersetzt mittels Relozierungstabellen die realen Adressen in virtuelle Adressen (relocation).

Primär-/Sekundärspeicher

Primärspeicher
kurzzeitige Ablage der Daten während des Betriebs (Haupt- oder Arbeitsspeicher)

- Direkt adressierbar
- Organisation über Adressen
- Datenquelle für Instruktionen und Operanden für die CPU
- Realisiert als RAM oder ROM

Sekundärspeicher
langfristige, permanente Ablage der Daten

- Indirekt adressierbar (über Driver)
- Logische Organisation (Dateisystem)
- Permanente Speicherung von Programmen und Daten

Primär-/Sekundärspeicher

Primärspeicher
kurzzeitige Ablage der Daten während des Betriebs (Haupt- oder Arbeitsspeicher)

- Direkt adressierbar
- Organisation über Adressen
- Datenquelle für Instruktionen und Operanden für die CPU
- Realisiert als RAM oder ROM

Sekundärspeicher
langfristige, permanente Ablage der Daten

- Indirekt adressierbar (über Driver)
- Logische Organisation (Dateisystem)
- Permanente Speicherung von Programmen und Daten

Cache-Speicher

Ziel: Schnellstmöglicher Speicherzugriff für die CPUS $\rightarrow$ Sinnvoll, wenn CPU schneller als Hauptspeicherzugriff

- Prozessorintern oder prozessorextern realisiert
- Puffert Teile des Hauptspeichers
- Effizient bei wiederholtem Zugriff auf gleiche Adressen

Einflussgrößen:

- Größe des Cache-Speichers
- Organisation
- Programmstruktur (z.B. optimierte Organisation von Array-Daten und Programmflüssen)

Registerzugriffe und Operationen werden durch den Cache nicht beschleunigt

Cache-Speicher

Ziel: Schnellstmöglicher Speicherzugriff für die CPUS $\rightarrow$ Sinnvoll, wenn CPU schneller als Hauptspeicherzugriff

- Prozessorintern oder prozessorextern realisiert
- Puffert Teile des Hauptspeichers
- Effizient bei wiederholtem Zugriff auf gleiche Adressen

Einflussgrößen:

- Größe des Cache-Speichers
- Organisation
- Programmstruktur (z.B. optimierte Organisation von Array-Daten und Programmflüssen)

Registerzugriffe und Operationen werden durch den Cache nicht beschleunigt

Cache-Speicher

Ziel: Schnellstmöglicher Speicherzugriff für die CPUS $\rightarrow$ Sinnvoll, wenn CPU schneller als Hauptspeicherzugriff

- Prozessorintern oder prozessorextern realisiert
- Puffert Teile des Hauptspeichers
- Effizient bei wiederholtem Zugriff auf gleiche Adressen

Einflussgrößen:

- Größe des Cache-Speichers
- Organisation
- Programmstruktur (z.B. optimierte Organisation von Array-Daten und Programmflüssen)

Registerzugriffe und Operationen werden durch den Cache nicht beschleunigt

Cache-Speicher

Die Leistung eines Cache-Speichers ist über die Zugriffszeit definiert:

$$T_{eff} = hT_c + (1 - h)T_h$$

T_{eff} ... effektive Zugriffszeit
T_c ... Zugriffszeit des Cache-Speichers
T_h ... Zugriffszeit des Hauptspeichers
h ... hit rate (Trefferrate), Verhältnis der Anzahl der Treffer zu den Gesamtzugriffen (0, alle Zugriffe auf Hauptspeicher, 1 alle Zugriffe in den Cache)

Heap und Stack

Statische Daten

- Lebensdauer ist die Programmlaufzeit
- Liegen im Datensegement auf dem Heap (globale Variablen, statische Variablen)

Dynamische Daten

- kurze Lebensdauer, z.B: Funktion
- Liegen am Stack (z.B.: lokale (auto) Variablen), Speicher wird implizit freigegeben
- Liegen am Heap (z.B. malloc(), dynamisch allozierter, zusammenhängender Speicherbereich, explizite Freigabe erforderlich)

Heap und Stack

Statische Daten

- Lebensdauer ist die Programmlaufzeit
- Liegen im Datensegement auf dem Heap (globale Variablen, statische Variablen)

Dynamische Daten

- kurze Lebensdauer, z.B: Funktion
- Liegen am Stack (z.B.: lokale (auto) Variablen), Speicher wird implizit freigegeben
- Liegen am Heap (z.B. malloc(), dynamisch allozierter, zusammenhängender Speicherbereich, explizite Freigabe erforderlich)

Heapmanagement

Aus einem großen Speicherbereich werden durch die Laufzeitbibliothek oder das Betriebssystem dynamisch kleinere Speicherbereiche variabler Größe bereitgestellt (malloc/free, new/delete, . . .)

Anforderungen:

- Flexible Zuordnungsgröße
- Zusammenhängende Bereiche
- Schnelle Zuordnung
- Maximale Speicherausnutzung, keine Lücken
- Adressraumausrichtung (z.B. CPU Einschränkung auf durch 2 oder 4 teilbare Adressen)

Problem: Fragmentierung

Heapmanagement

Aus einem großen Speicherbereich werden durch die Laufzeitbibliothek oder das Betriebssystem dynamisch kleinere Speicherbereiche variabler Größe bereitgestellt (malloc/free, new/delete, ...)

Anforderungen:

- Flexible Zuordnungsgröße
- Zusammenhängende Bereiche
- Schnelle Zuordnung
- Maximale Speicherausnutzung, keine Lücken
- Adressraumausrichtung (z.B. CPU Einschränkung auf durch 2 oder 4 teilbare Adressen)

Problem: Fragmentierung

Heapmanagement

Aus einem großen Speicherbereich werden durch die Laufzeitbibliothek oder das Betriebssystem dynamisch kleinere Speicherbereiche variabler Größe bereitgestellt (malloc/free, new/delete, ...)

Anforderungen:

- Flexible Zuordnungsgröße
- Zusammenhängende Bereiche
- Schnelle Zuordnung
- Maximale Speicherausnutzung, keine Lücken
- Adressraumausrichtung (z.B. CPU Einschränkung auf durch 2 oder 4 teilbare Adressen)

Problem: Fragmentierung

Heapmanagement

Freie Speicherbereiche, z.B.: [10,4,20,18,7,9,12,15]

```
malloc(10);
malloc(12);
malloc(9);
```

1. First-Fit: [*10*,4,*20*→*12*→9,18,7,9,12,15]
2. Next-Fit: [*10*,4,*20*→12,*18*→9,7,9,12,15]
3. Best-Fit: [*10*,4,20,18,7,*9*,*12*,15]
4. Worst-Fit: [10,4,*20*→10,*18*→12,7,9,12,*15*→9]
5. Quick-Fit
6. Buddy-Verfahren

Swapping

Arbeitsspeicher ist beschränkt → Strategie für Vorgehen bei knappem Speicher erforderlich

Lösung: Das Datensegment eines inaktiven Prozesses wird ausgelagert

- Speicheradressen sollen sich nicht ändern
- Fragmentierung durch unterschiedliche Partitionsgrößen

Swapping lagert Teile oder ganze Prozesse zusammenhängend auf die Festplatte aus
(Einfaches) Paging lagert Teile fester Größe (Seiten) aus, Hauptspeicher wird aufgeteilt in Seitenrahmen

Swapping

Arbeitsspeicher ist beschränkt $\rightarrow$ Strategie für Vorgehen bei knappem Speicher erforderlich

Lösung: Das Datensegment eines inaktiven Prozesses wird ausgelagert

- Speicheradressen sollen sich nicht ändern
- Fragmentierung durch unterschiedliche Partitionsgrößen

Swapping lagert Teile oder ganze Prozesse zusammenhängend auf die Festplatte aus
(Einfaches) Paging lagert Teile fester Größe (Seiten) aus, Hauptspeicher wird aufgeteilt in Seitenrahmen

Virtueller Speicher

Unterscheidung:

1. Virtueller Speicher: direkt ansprechbarer Adressraum, wird im Programm verwendet und im Debugger angezeigt
2. Realer Speicher: physischer Speicherraum, Speicherstellen im Hauptspeicher die über den Bus ausgewählt werden

Umsetzung erfolgt über Tabellen in der Memory Management Unit (MMU)

Eigenschaften:

- Jeder Prozess hat einen privaten Adressraum und ist scheinbar alleine
- Jeder Prozess erhält wenn aktiv wieder die selben (virtuellen) Adressen
- Größe des Adressraums vom Hauptspeicher unabhängig
- Gegenseitiger Schreib- und Leseschutz

Virtueller Speicher

Unterscheidung:

1. Virtueller Speicher: direkt ansprechbarer Adressraum, wird im Programm verwendet und im Debugger angezeigt
2. Realer Speicher: physischer Speicherraum, Speicherstellen im Hauptspeicher die über den Bus ausgewählt werden

Umsetzung erfolgt über Tabellen in der Memory Management Unit (MMU)

Eigenschaften:

- Jeder Prozess hat einen privaten Adressraum und ist scheinbar alleine
- Jeder Prozess erhält wenn aktiv wieder die selben (virtuellen) Adressen
- Größe des Adressraums vom Hauptspeicher unabhängig
- Gegenseitiger Schreib- und Leseschutz

Adressumsetzung

Memory Management Unit (MMU)

- führt bei jedem Speicherzugriff eine Adressumsetzung durch (virtuell auf real)
- für jeden Prozess wird eine eigene Umsetzungstabelle verwaltet

Segmentbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{logisch}} + A_{\text{Segmentstart}}$

Seitenbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{relativ}} + A_{\text{Seitengroesze}}\,p$

p . . . Seitenrahmennummer

Adressumsetzung

Memory Management Unit (MMU)

- führt bei jedem Speicherzugriff eine Adressumsetzung durch (virtuell auf real)
- für jeden Prozess wird eine eigene Umsetzungstabelle verwaltet

Segmentbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{logisch}} + A_{\text{Segmentstart}}$

Seitenbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{relativ}} + A_{\text{Seitengroesze}}\, p$

p . . . Seitenrahmennummer

Adressumsetzung

Memory Management Unit (MMU)

- führt bei jedem Speicherzugriff eine Adressumsetzung durch (virtuell auf real)
- für jeden Prozess wird eine eigene Umsetzungstabelle verwaltet

Segmentbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{logisch}} + A_{\text{Segmentstart}}$

Seitenbasierte Umsetzung

- Aufteilung der Prozesse in Segmente mit eindeutiger ID
- Adressierung: $A_{\text{physisch}} = A_{\text{relativ}} + A_{\text{Seitengroesze}}p$

p . . . Seitenrahmennummer

Prozesse und Threads

Begriffe:

- Programm: Verfahrensvorschrift (ausführbare Datei im Dateisystem)
- Process/Task: Ein Programm in Ausführung (hat Befehlszähler, Register, Variablenbelegung)
- Thread: Parallel ablaufende Aktivität innerhalb eines Prozesses (jeder Thread gehört zu einem Prozess)
- Job/Session: Voneinander unabhängig laufende Anwendungen, die aus einer Gruppe von Prozessen bestehen

Parallelverarbeitung

Parallelverarbeitung: Verschiedene Aktivitäten finden gleichzeitig statt

1. rein seriell
2. rein parallel
3. seriell/parallel
4. gemischt

Varianten der Parallelität:

1. Process: jeder Prozess teilt sich den Adressraum nur mit dem Betriebssystem, Prozesse sind isoliert
2. Thread: teilen sich den Adressraum innerhalb eines Prozesses, sind nicht voneinander isoliert

Je nach Betriebssystem werden verschiedene Kombinationen unterstützt (1P/1T, 1P/nT, nP/1T, nP/nT)

Parallelverarbeitung

Parallelverarbeitung: Verschiedene Aktivitäten finden gleichzeitig statt

1. rein seriell
2. rein parallel
3. seriell/parallel
4. gemischt

Varianten der Parallelität:

1. Process: jeder Prozess teilt sich den Adressraum nur mit dem Betriebssystem, Prozesse sind isoliert
2. Thread: teilen sich den Adressraum innerhalb eines Prozesses, sind nicht voneinander isoliert

Je nach Betriebssystem werden verschiedene Kombinationen unterstützt (1P/1T, 1P/nT, nP/1T, nP/nT)

Prozesse I

Ein Prozess ist ein Programm in Ausführung, umfasst auch Befehlszähler, Register und Variablenbelegung und beschreibt zusammen mit den Statusinformationen eine Instanz.

↓	Stack	(bis 0xF...F)
	freier Speicher	
↑	Daten	
	Text	(ab 0x0...0)

Tabelle: Segmente eines Prozesses

Prozesse II

Auf einem Prozessor(-kern) ist zu einem Zeitpunkt genau ein Prozess aktiv
$\rightarrow$ quasi-/pseudoparallele Ausführung durch den Scheduler (time sharing)

Scheduler Teilt einem Prozess CPU-Zeit (time slot, time slice) zu oder entzieht diese.

Prozesse II

Auf einem Prozessor(-kern) ist zu einem Zeitpunkt genau ein Prozess aktiv
$\rightarrow$ quasi-/pseudoparallele Ausführung durch den Scheduler (time sharing)

Scheduler Teilt einem Prozess CPU-Zeit (time slot, time slice) zu oder entzieht diese.

Prozesse III

Multi Processing

- Jeder Prozess hat eine eigene, virtuelle CPU
- Jeder Prozess hat einen eigenen, getrennten Adressraum
- Prozesse haben Prioritäten
- Prozesse können unabhängig arbeiten oder Abhängigkeiten haben (kooperierende Prozesse)
- Prozesse können kommunizieren
- Prozesse können andere Prozesse erzeugen
- Bei der Prozessumschaltung wird der aktuelle Zustand und Metainformationen (als Process Control Block) in der Prozesstabelle gesichert

Prozesse IV

Process Control Block

- Process Identifier (PID)
- Process State
- Program Counter
- CPU Register
- CPU Scheduling Information
- Memory Management Information
- I/O State Information
- Account Information

Prozesse V

Ablauf beim Prozesswechsel

1. Kontextsicherung: Sicherung der Register in einen PCB
2. Auswahl: Selektion des nächsten Prozesses durch den Scheduler
3. Kontextwiederherstellung: Rückkopieren der Register aus einem PCB

Prozesserzeugung I

Prozesserzeugung

- bei Systemstart (z.B. UI, Daemons/Services)
- durch Erzeugung durch einen anderen Prozess
- durch den Benutzer
- durch Batchbetrieb (z.B. Cronjob)

Prozessbeendigung

- durch normales Terminieren (freiwillig)
- aufgrund eines Fehlers (freiwillig oder unfreiwillig)
- Terminierung durch einen anderen Prozess (unfreiwillig)

Prozesserzeugung II

Prozessstartformen

- Prozessverkettung (chain)

- Prozessvergabelung (fork)

- Prozesserzeugung (create)

Prozesserzeugung unter Unix

Unter Linux/Unix gibt es Prozesshierarchien

- Jeder Prozess hat einen Elternprozess
- Jeder Prozess hat 0 bis n Kindprozesse
- Durch Erzeugung eines Kindprozesses entsteht eine Prozesshierarchie
- Ein Prozess mit allen Nachkommen bildet eine Prozessfamilie
- Prozesse können nicht enterbt werden
- Bei Terminierung mit einem Signal erhalten auch alle Nachkommen dieses Signal

Prozessbeendigung

Prozessvereinigungsformen

- Prozessvereinigung (join, exit)

- Prozesstreffen (wait, exit)

Beispiel mit `fork()`

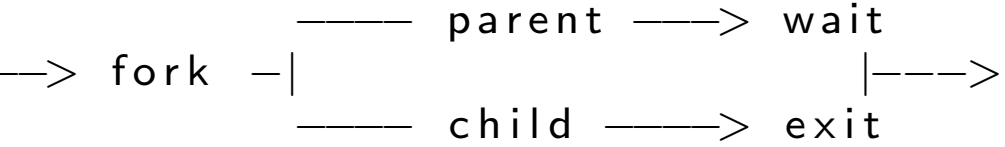
Beispiel mit `fork()` (Unix-Systemaufruf)

```
#include <unistd.h>
...
void f() {
 int status;
 pid_t pid;

 pid=fork();
 if (pid==0) {
   // Kindprozess ab fork-Aufruf und bis exit
   ...
   _Exit(0);
 } else {
   // Elternprozess
   ...
   pid = wait(&status);
 }
}
```

Beispiel mit `fork()`

Beispiel mit `fork()` (Unix-Systemaufruf)

```
#include <unistd.h>
...
void f() {
 int status;
 pid_t pid;

 pid=fork();
 if (pid==0) {
   // Kindprozess ab fork-Aufruf und bis exit
   ...
   _Exit(0);
 } else {
   // Elternprozess
   ...
   pid = wait(&status);
 }
}
```



Zombieprozess

Zombie: "a person who moves very slowly and is not aware of what is happening especially because of being very tired" – Merriam Webster[8]

Ein Zombieprozess führt keinen Code mehr aus, belegt aber Einträge in der Prozesstabelle. Eine Auflösung kann nur durch `wait()` im Elternprozess aufgelöst werden.

1. Elternprozess wartet auf Ende des Kindprozesses
2. Kindprozess terminiert bevor Elternprozess wartet (zwischen `_Exit()` im Kindprozess und `wait()` im Elternprozess entsteht ein Zombieprozess)

[8] http://www.merriam-webster.com/dictionary/zombie

Zombieprozess

Zombie: "a person who moves very slowly and is not aware of what is happening especially because of being very tired" – Merriam Webster[8]

Ein Zombieprozess führt keinen Code mehr aus, belegt aber Einträge in der Prozesstabelle. Eine Auflösung kann nur durch `wait()` im Elternprozess aufgelöst werden.

1. Elternprozess wartet auf Ende des Kindprozesses
2. Kindprozess terminiert bevor Elternprozess wartet (zwischen `_Exit()` im Kindprozess und `wait()` im Elternprozess entsteht ein Zombieprozess)

[8] http://www.merriam-webster.com/dictionary/zombie

Threads

Threads nutzen gemeinsame Ressourcen in einem Prozess (insbes. gemeinsamer Adressraum)

Pro Prozess vorhanden	Pro Thread vorhanden
Adressraum	Programmzähler
Globale Variable	Register
Geöffnete Dateien	Stapel

Threadwechsel ist einfacher als Prozesswechsel und ermöglicht eine bessere Nutzung der Rechenzeit durch Verringerung der Leerzeiten (z.B. bei I/O-Aktivitäten)

Threads

Threads nutzen gemeinsame Ressourcen in einem Prozess (insbes. gemeinsamer Adressraum)

Pro Prozess vorhanden	**Pro Thread vorhanden**
Adressraum	Programmzähler
Globale Variable	Register
Geöffnete Dateien	Stapel

Threadwechsel ist einfacher als Prozesswechsel und ermöglicht eine bessere Nutzung der Rechenzeit durch Verringerung der Leerzeiten (z.B. bei I/O-Aktivitäten)

Threads

Threads nutzen gemeinsame Ressourcen in einem Prozess (insbes. gemeinsamer Adressraum)

Pro Prozess vorhanden	**Pro Thread vorhanden**
Adressraum	Programmzähler
Globale Variable	Register
Geöffnete Dateien	Stapel

Threadwechsel ist einfacher als Prozesswechsel und ermöglicht eine bessere Nutzung der Rechenzeit durch Verringerung der Leerzeiten (z.B. bei I/O-Aktivitäten)

Beispiel mit pthread_create()

Beispiel mit pthread_create() (Unix-Systemaufruf)

```
#include <pthread.h>
...
void* thread_function(void* arg) {
  ...
  return arg;
}

void f() {
  pthread_t id;
  int status, *presult;
  status = pthread_create(&id, NULL, thread_function, NULL);
  if (status == 0) ; // gestartet
  ...
  pthread_join(id, (void**)&presult);
  ...
}
```

Siehe http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_create.html und http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_join.html

Scheduling I

CPU-Scheduling: Legt fest, welcher Prozess als nächster die CPU erhält.

Prozesszustände: Bereit (ready), Laufend (running), Wartend (waiting)

- waiting: Prozess wartet in einer Warteschlange auf Ressource
- ready: Liste von Prozessen die auf die CPU warten
- running: Prozess wird in der CPU bearbeitet

Wird einem Prozess unfreiwillig der Prozessor entzogen (z.B. durch einen Interrupt) wechselt er an den Beginn der Bereitliste.
Wird einem Prozess zeitbedingt der Prozessor entzogen, so wechselt er an das Ende der Bereitliste.
Wartet ein Prozess auf eine Ressource, wechselt er in die Warteschlange der Ressource.

Scheduling I

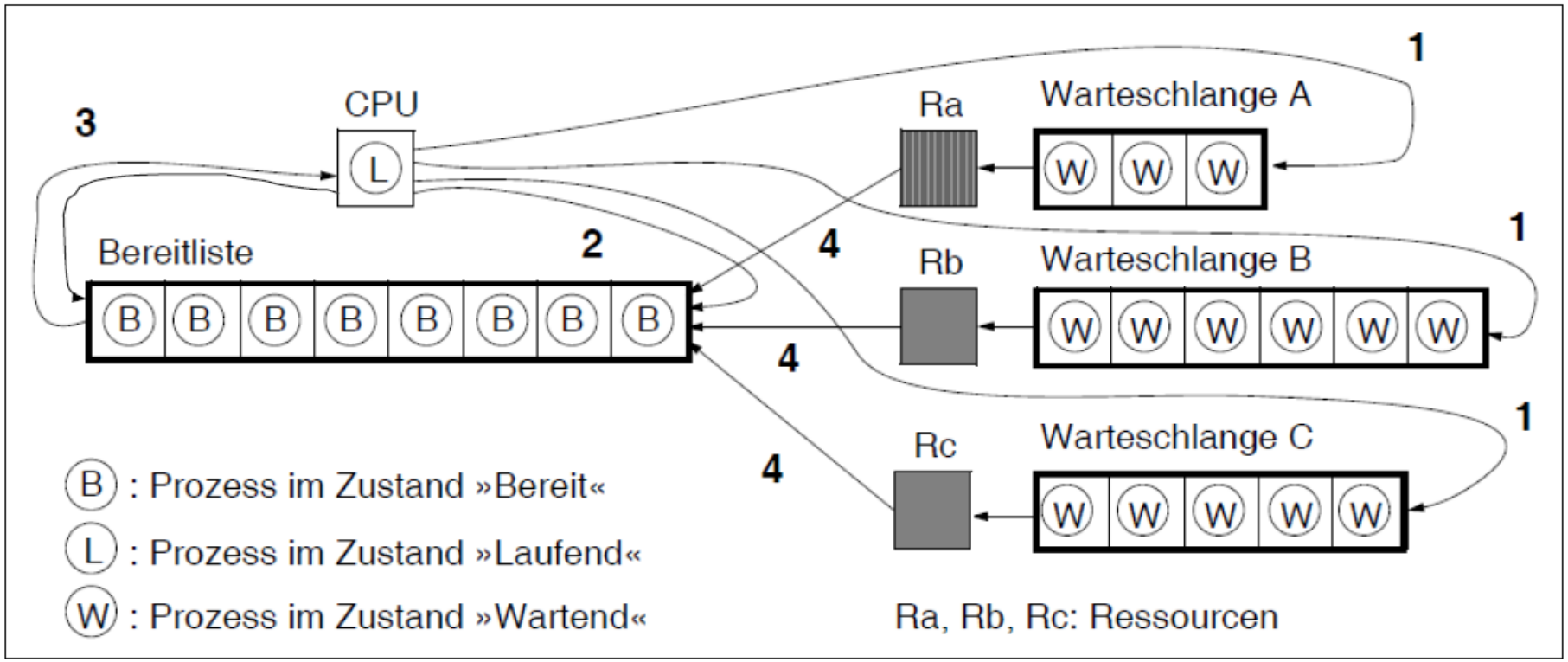
CPU-Scheduling: Legt fest, welcher Prozess als nächster die CPU erhält.

Prozesszustände: Bereit (ready), Laufend (running), Wartend (waiting)

- waiting: Prozess wartet in einer Warteschlange auf Ressource
- ready: Liste von Prozessen die auf die CPU warten
- running: Prozess wird in der CPU bearbeitet

Wird einem Prozess unfreiwillig der Prozessor entzogen (z.B. durch einen Interrupt) wechselt er an den Beginn der Bereitliste.
Wird einem Prozess zeitbedingt der Prozessor entzogen, so wechselt er an das Ende der Bereitliste.
Wartet ein Prozess auf eine Ressource, wechselt er in die Warteschlange der Ressource.

Scheduling II



Scheduling III

Zuteilungsstrategien:

1. Non-preemptive: Prozess läuft solange bis er auf ein Ein-/Ausgabegerät wartet, auf einen anderen Prozess wartet oder er freiwillig auf den Prozessor verzichtet
   $\rightarrow$ keine Prozessumschatung bei Systeminterrupt oder wenn anderer Prozess ablaufbereit
2. Preemptive: Prozess läuft solange bis er auf ein Ein-/Ausgabegerät wartet, auf einen anderen Prozess wartet oder er freiwillig auf den Prozessor verzichtet
   Zusätzlich: Bei Systeminterrupt wenn Zeitquantum erschöpft und ablaufbereite Prozesse warten oder ein auf Ein-/Ausgabedaten wartender Prozess bevorzugt werden soll

Scheduling IV

Anforderungen aus Anwendersicht:

- Minimierung der Durchlaufzeit (turnaround time)
- Minimierung der Antwortzeit (response time)
- Einhaltung des Endtermins (deadline)

Strategien:

- First Come First Serve (FCFS oder FIFO-Strategie)
- Shortest Job First (SJF)
- Shortest Remaining Time (SRT)
- Round Robin (RR)
- Multi Level (ML)
- Multi Level Feedback (MFL)

Scheduling IV

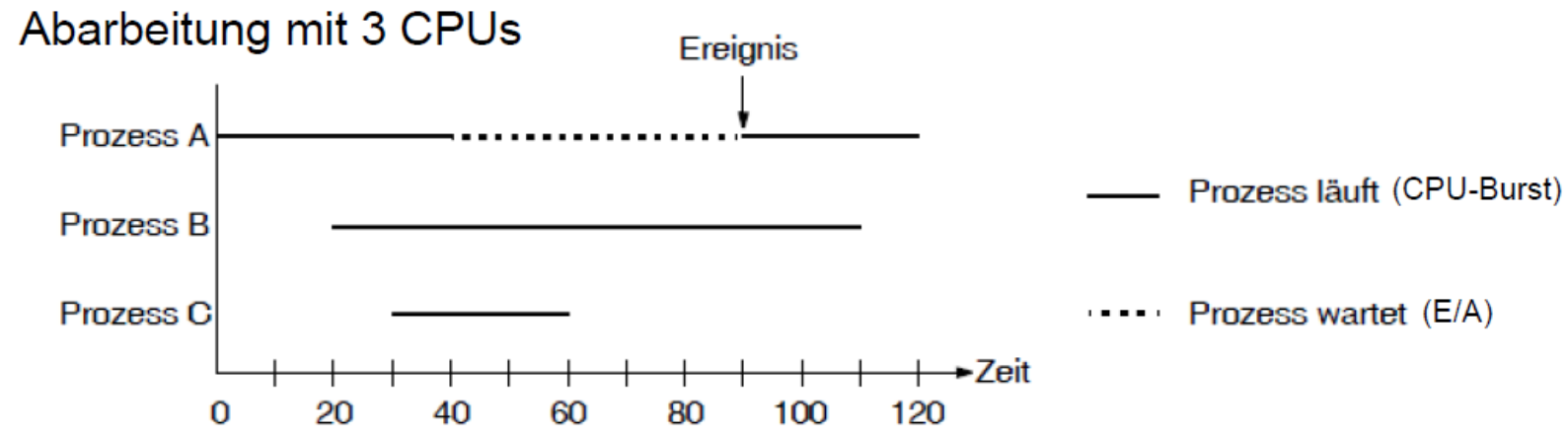
Anforderungen aus Anwendersicht:

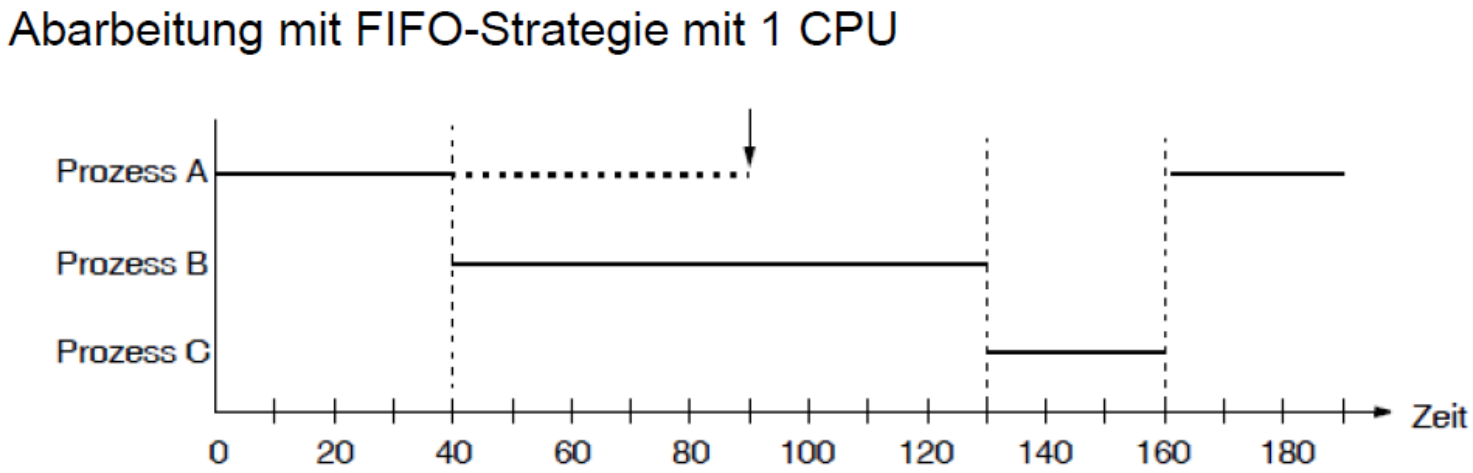
- Minimierung der Durchlaufzeit (turnaround time)
- Minimierung der Antwortzeit (response time)
- Einhaltung des Endtermins (deadline)

Strategien:

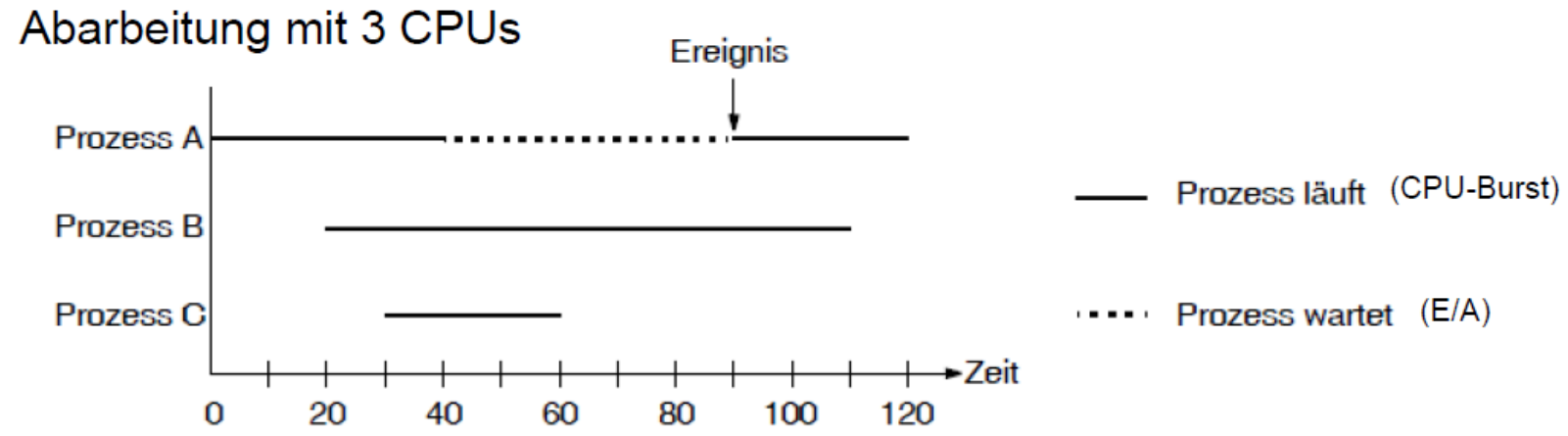
- First Come First Serve (FCFS oder FIFO-Strategie)
- Shortest Job First (SJF)
- Shortest Remaining Time (SRT)
- Round Robin (RR)
- Multi Level (ML)
- Multi Level Feedback (MFL)

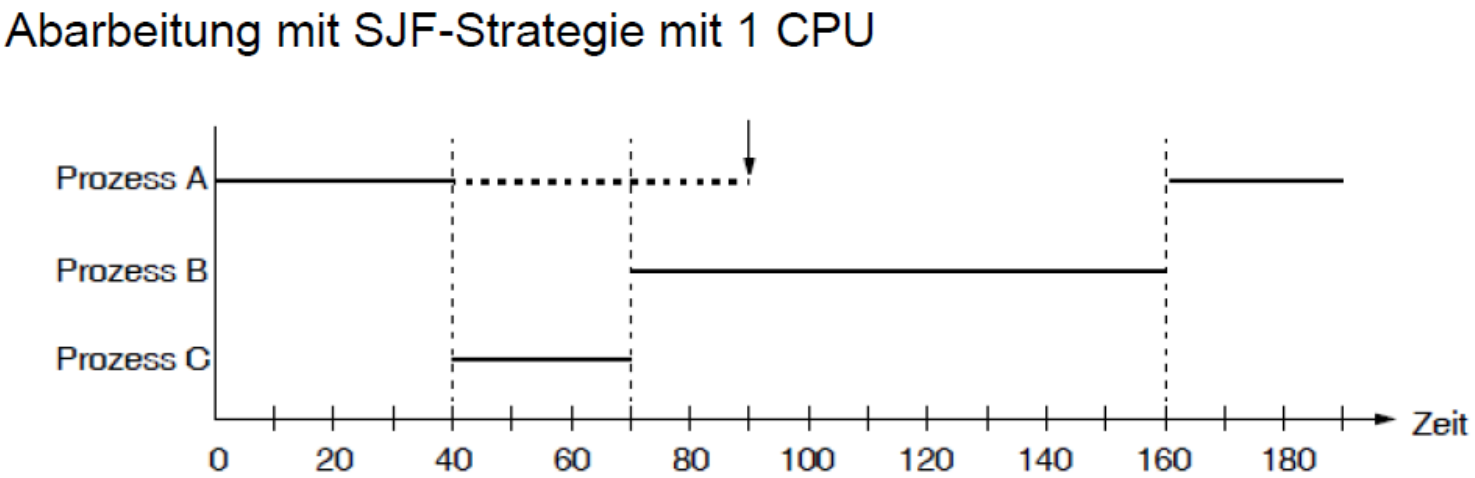
FCFS/FIFO-Strategie



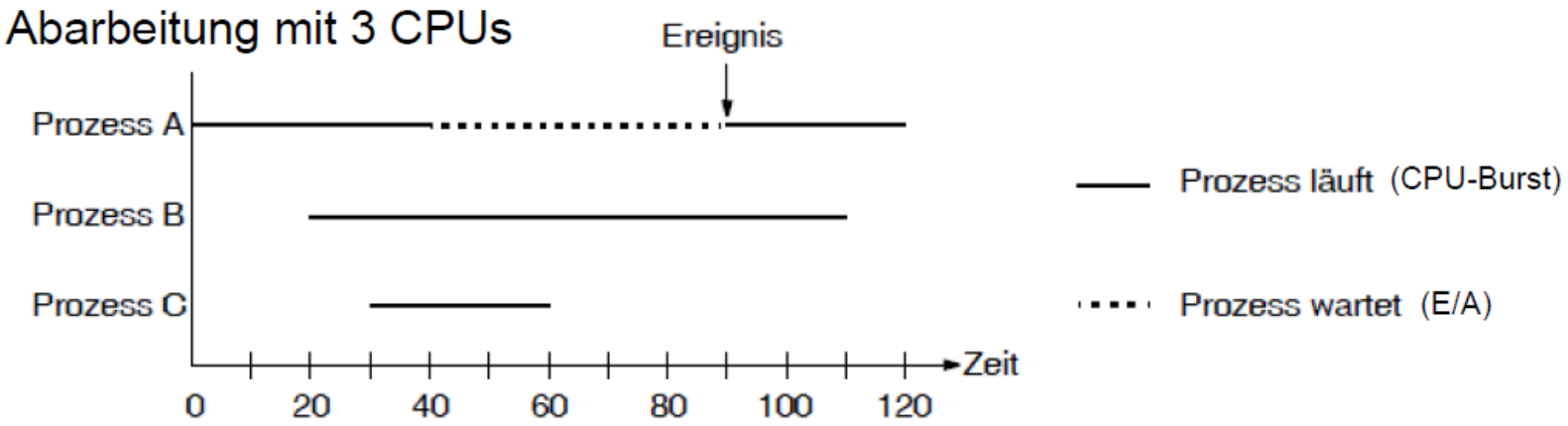


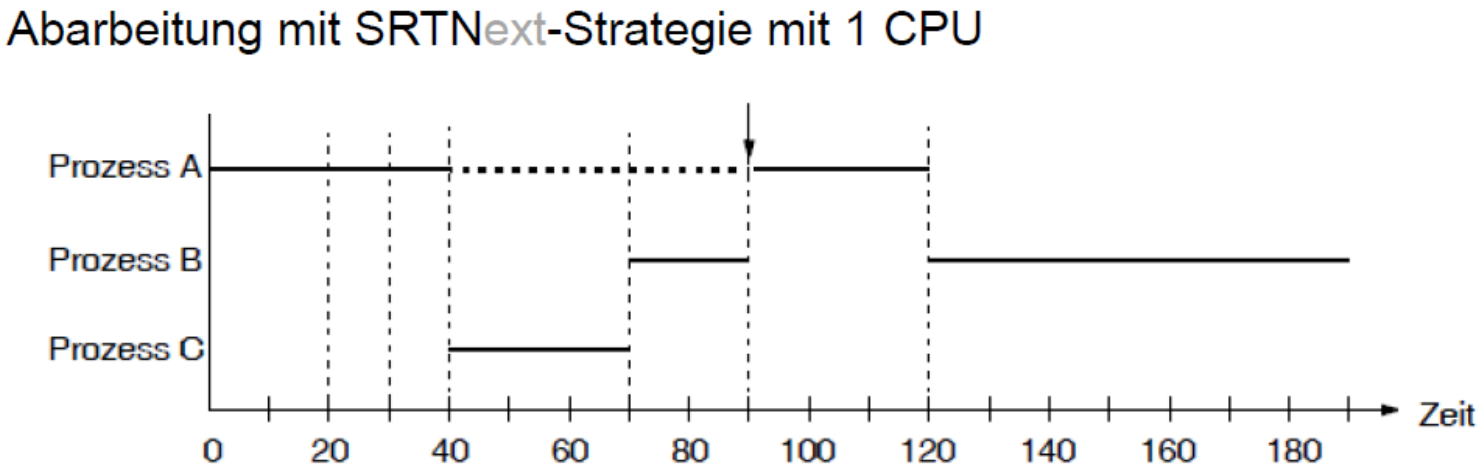
SJF


Abarbeitung mit 3 CPUs
Ereignis
Prozess A
Prozess B
Prozess C
Zeit
0
20
40
60
80
100
120
Prozess läuft (CPU-Burst)
Prozess wartet (E/A)

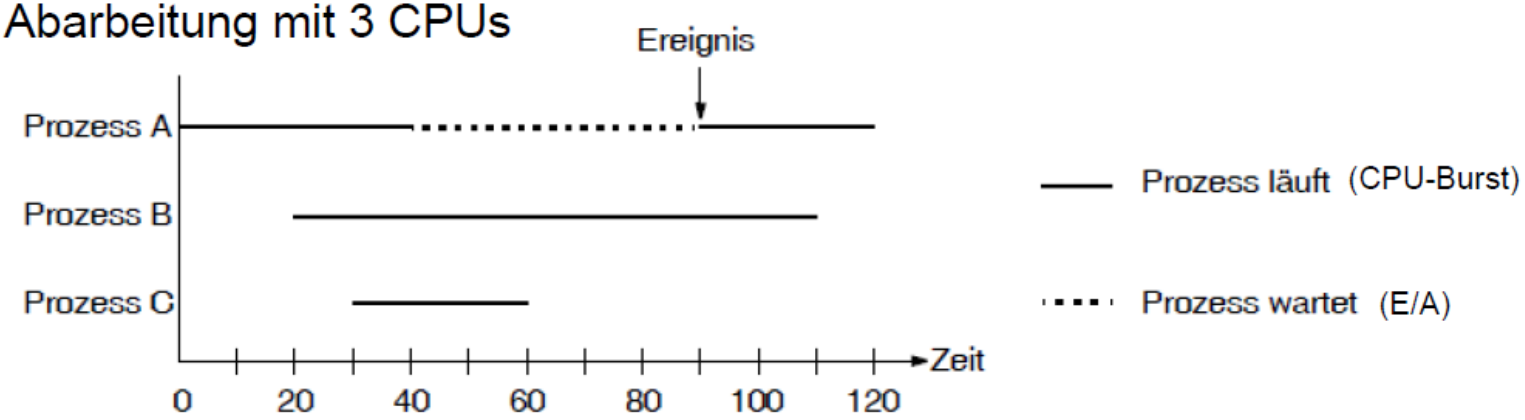

Abarbeitung mit SJF-Strategie mit 1 CPU
Prozess A
Prozess B
Prozess C
Zeit
0
20
40
60
80
100
120
140
160
180

SRT

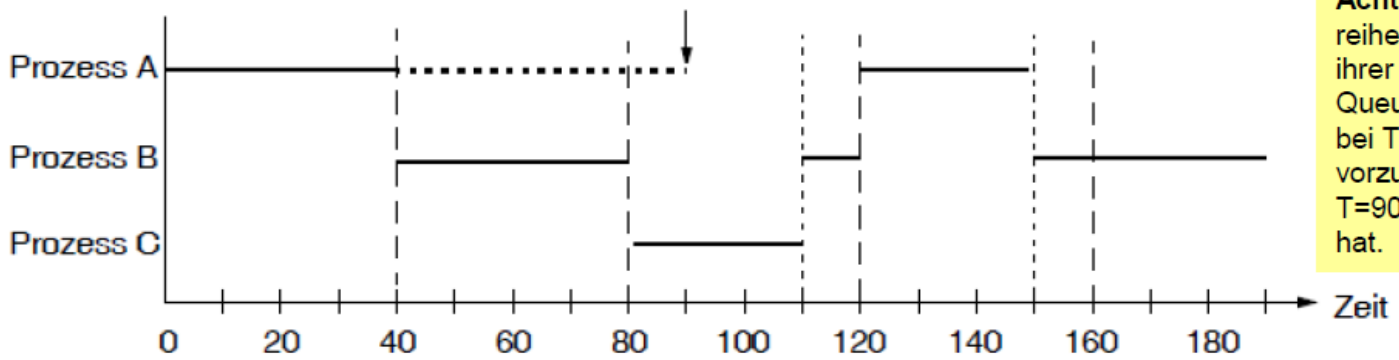

Abarbeitung mit 3 CPUs
Ereignis
Prozess A
Prozess B
Prozess C
Zeit
0
20
40
60
80
100
120
Prozess läuft (CPU-Burst)
Prozess wartet (E/A)


Abarbeitung mit SRTNext-Strategie mit 1 CPU
Prozess A
Prozess B
Prozess C
Zeit
0
20
40
60
80
100
120
140
160
180

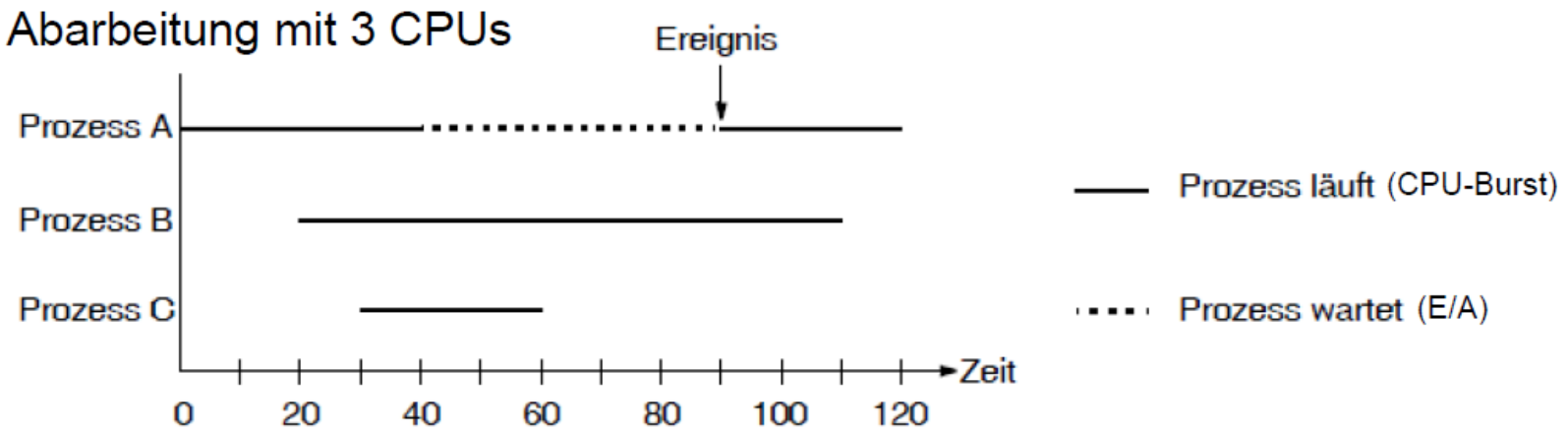
Abarbeitung mit 3 CPUs

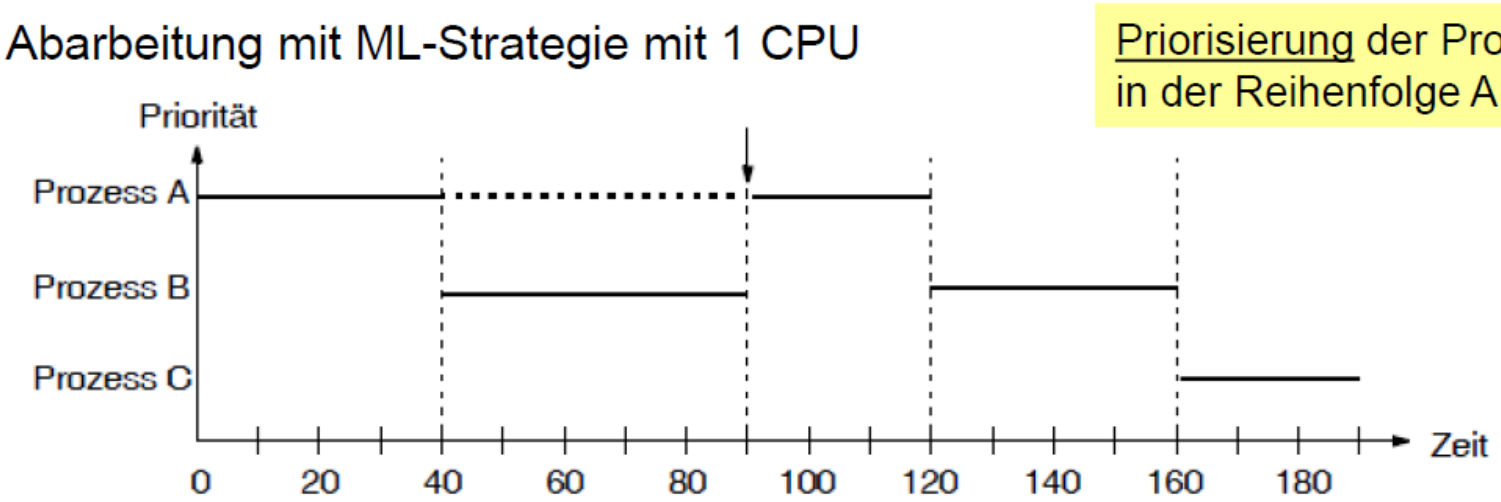


Abarbeitung mit RR-Strategie mit 1 CPU (mit Zeitquantum = 40 Zeiteinheiten)



Achtung: Die Prozesse reihen sich entsprechend ihrer Bereitschaft in die Queue ein; deshalb wird bei T=110 Prozess B bevorzugt, da A sich erst bei T=90 in die Queue gereiht hat.





Priorisierung der Prozesse in der Reihenfolge A, B, C

Synchronisation von Prozessen und Threads

Problem: Parallele Aktivitäten mit gleichzeitigem Zugriff auf gemeinsame Ressource (shared data structures, shared files, shared hardware)

Probleme

- verlorene Aktualisierung (lost update problem)
- inkonsistente Abfrage (inconsistent read)
- Synchronisatiosnprobleme

Lösung

- Absicherung mit Selbstverwaltung (z.B. über boolsche Variablen in einer Applikation)
- Absicherung mit Systemmitteln (Semaphoren)

Synchronisation von Prozessen und Threads

Problem: Parallele Aktivitäten mit gleichzeitigem Zugriff auf gemeinsame Ressource (shared data structures, shared files, shared hardware)

Probleme

- verlorene Aktualisierung (lost update problem)
- inkonsistente Abfrage (inconsistent read)
- Synchronisatiosnprobleme

Lösung

- Absicherung mit Selbstverwaltung (z.B. über boolsche Variablen in einer Applikation)
- Absicherung mit Systemmitteln (Semaphoren)

Synchronisation von Prozessen und Threads

Problem: Parallele Aktivitäten mit gleichzeitigem Zugriff auf gemeinsame Ressource (shared data structures, shared files, shared hardware)

Probleme

- verlorene Aktualisierung (lost update problem)
- inkonsistente Abfrage (inconsistent read)
- Synchronisatiosnprobleme

Lösung

- Absicherung mit Selbstverwaltung (z.B. über boolsche Variablen in einer Applikation)
- Absicherung mit Systemmitteln (Semaphoren)

Arten der Interprozesskommunikation

Interprozesskommunikation ist der Oberbegriff für Synchronisation und Kommunikation

Synchronisation

- Semaphore
- Signale

Kommunikation

- speicherbasierte Verfahren über gemeinsamen Speicher
- nachrichtenbasierte Verfahren (synchron oder asynchron)

Arten der Interprozesskommunikation

Interprozesskommunikation ist der Oberbegriff für Synchronisation und Kommunikation

Synchronisation

- Semaphore
- Signale

Kommunikation

- speicherbasierte Verfahren über gemeinsamen Speicher
- nachrichtenbasierte Verfahren (synchron oder asynchron)

Arten der Interprozesskommunikation

Interprozesskommunikation ist der Oberbegriff für Synchronisation und Kommunikation

Synchronisation

- Semaphore
- Signale

Kommunikation

- speicherbasierte Verfahren über gemeinsamen Speicher
- nachrichtenbasierte Verfahren (synchron oder asynchron)

Absicherung mit Selbstverwaltung I

Prozess A

```
extern bool bCritical;
if (bCritical == false) {
  bCritical = true;
  value = value + 4.56;
  bCritical = false;
}
```

Prozess B

```
extern bool bCritical;
if (bCritical == false) {
  bCritical = true;
  value = value - 3.21;
  bCritical = false;
}
```

- Zugriff auf gemeinsamen Code und Variable (critical section)
- Anwendungsfall, bei dem das Resultat davon abhängt wann wecher Thread/Prozess läuft (race codition)

→ Lösung durch wechselseitigen Ausschluss (mutual exclusion)

Absicherung mit Selbstverwaltung I

Prozess A

```
extern bool bCritical;
if (bCritical == false) {
 bCritical = true;
 value = value + 4.56;
 bCritical = false;
}
```

Prozess B

```
extern bool bCritical;
if (bCritical == false) {
 bCritical = true;
 value = value - 3.21;
 bCritical = false;
}
```

- Zugriff auf gemeinsamen Code und Variable (critical section)
- Anwendungsfall, bei dem das Resultat davon abhängt wann wecher Thread/Prozess läuft (race codition)

→ Lösung durch wechselseitigen Ausschluss (mutual exclusion)

Absicherung mit Selbstverwaltung II

Prozess A

```
EnterCriticalSection(1);
value = value + 4.56;
LeaveCriticalSection(1);
```

Prozess B

```
EnterCriticalSection(1);
value = value - 3.21;
LeaveCriticalSection(1);
```

- Wechselseitiger Ausschluss durch Schutzfunktionspaar (mutual exclusion)
- Serialisierung innerhalb der critical section, nur ein Thread darf zu einem Zeitpunkt im kritischen Bereich sein

Absicherung mit Selbstverwaltung II

Prozess A

```
EnterCriticalSection(1);
value = value + 4.56;
LeaveCriticalSection(1);
```

Prozess B

```
EnterCriticalSection(1);
value = value - 3.21;
LeaveCriticalSection(1);
```

- Wechselseitiger Ausschluss durch Schutzfunktionspaar (mutual exclusion)
- Serialisierung innerhalb der critical section, nur ein Thread darf zu einem Zeitpunkt im kritischen Bereich sein

Absicherung mit Systemmitteln

Absicherung mit speziellen Objekten bzw. Funktionen die Teil des Betriebssystemkerns oder systemnaher Funktionsbibliotheken sind

- Semaphore
- Mutex
- Critical Section Object

Absicherung über Meldungen

- Events
- Signals
- Messages
- Pipes

Absicherung mit Systemmitteln

Absicherung mit speziellen Objekten bzw. Funktionen die Teil des Betriebssystemkerns oder systemnaher Funktionsbibliotheken sind

- Semaphore
- Mutex
- Critical Section Object

Absicherung über Meldungen

- Events
- Signals
- Messages
- Pipes

Semaphore I

Semaphor: Betriebssystemobjekt zur Absicherung und Synchronisation von Prozessen und Threads

Bestandteile

- Identität: Instanzenbezeichner eines Semaphors
- Zähler: Markenzähler mir Initialwert
- Warteschlange: Liste der blockierten Prozesse
- P-Operation / Down-Operation: Inkrementiert den Markenzähler
- V-Operation / Up-Operation: Dekrementiert den Markenzähler

Semaphore I

Semaphor: Betriebssystemobjekt zur Absicherung und Synchronisation von Prozessen und Threads

Bestandteile

- Identität: Instanzenbezeichner eines Semaphors
- Zähler: Markenzähler mir Initialwert
- Warteschlange: Liste der blockierten Prozesse
- P-Operation / Down-Operation: Inkrementiert den Markenzähler
- V-Operation / Up-Operation: Dekrementiert den Markenzähler

Semaphore II

Die Bedeutung der Marke ist applikationsspezifisch. Die Operationen P und V bewirken, dass ein Prozess blockiert wird, wenn ein anderer die critical section ausführt.

Ein wartender Prozess wird aus der Bereitliste entfernt und in die Warteschlange des des Semaphors eingefügt.

Der laufende Prozess zwischen P und V wird höher priorisiert (rescheduling). Der Code zwischen P und V wird garantiert unteilbar durchlaufen.

Semaphore II

Die Bedeutung der Marke ist applikationsspezifisch. Die Operationen P und V bewirken, dass ein Prozess blockiert wird, wenn ein anderer die critical section ausführt.

Ein wartender Prozess wird aus der Bereitliste entfernt und in die Warteschlange des des Semaphors eingefügt.

Der laufende Prozess zwischen P und V wird höher priorisiert (rescheduling). Der Code zwischen P und V wird garantiert unteilbar durchlaufen.

Semaphore II

Die Bedeutung der Marke ist applikationsspezifisch. Die Operationen P und V bewirken, dass ein Prozess blockiert wird, wenn ein anderer die critical section ausführt.

Ein wartender Prozess wird aus der Bereitliste entfernt und in die Warteschlange des des Semaphors eingefügt.

Der laufende Prozess zwischen P und V wird höher priorisiert (rescheduling). Der Code zwischen P und V wird garantiert unteilbar durchlaufen.

Semaphore III

Typen von Semaphoren

- Binärer Semaphor (binary semaphor, mutex): genau 0 oder 1 Marken
- Zählsemaphor (counting semaphor, general semaphor): beliebig viele Marken
- spezieller Semaphor (spinlock): für aktives Warten

```
while(bCritical == true) {
  // warten...
}
bCritical = true;
// Resource modifizieren
bCritical = false;
```

Implementierungsproblematik bei Semaphoren

- Die Implementierung muss die Unteilbarkeit von P bis V sicherstellen
- Aktives Warten in einer Schleife oder passives Warten mit CPU-Freigabe
- Wartereihenfolge (FIFO oder Priorisierung)
- Mehrfachbelegung der gleichen Semaphor von einem Thread (kann zu deadlock führen)

Anforderungen an Semaphoren

- Wechselseitiger Ausschluss (mutual exclusion): zu einem bestimmten Zeitpunkt darf maximal ein Prozess im kritischen Bereich sein
- Garantierter Fortschritt (progress): nur Prozesse die sich beworben haben dürfen den kritischen Bereich ausführen
- Kein Verhungern (no livelock, no starvation, bounded waiting): jeder kommt in Endlicher Zeit zum Zug
- Effizienz (efficiency): kein unnötiger Verbrauch von Ressourcen (CPU-Zeit)

Beispiel: Ablaufsynchronisation I

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 1:

A

```
P(sync); // auf B warten
```

B

```
P(sync); // auf A warten
```

Problem: Synchronisation ist nicht symmetrisch (A wartet auf B) $\rightarrow$ Bedingungssynchronisation (condition synchronization)

Beispiel: Ablaufsynchronisation I

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 1:

A

```
P(sync); // auf B warten
```

B

```
P(sync); // auf A warten
```

Problem: Synchronisation ist nicht symmetrisch (A wartet auf B) → Bedingungssynchronisation (condition synchronization)

Beispiel: Ablaufsynchronisation I

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 1:

A

```
P(sync); // auf B warten
```

B

```
P(sync); // auf A warten
```

Problem: Synchronisation ist nicht symmetrisch (A wartet auf B) $\rightarrow$ Bedingungssynchronisation (condition synchronization)

Beispiel: Ablaufsynchronisation II

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 2:

A

```
V(sync_B) // B wecken
P(sync_A); // auf B warten
```

B

```
V(sync_A) // a wecken
P(sync_B); // auf A warten
```

Diese Lösung ist symmetrisch $\rightarrow$ Ablaufsynchronisation (barrier synchronization)

Beispiel: Ablaufsynchronisation II

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 2:

A

```
V(sync_B) // B wecken
P(sync_A); // auf B warten
```

B

```
V(sync_A) // a wecken
P(sync_B); // auf A warten
```

Diese Lösung ist symmetrisch → Ablaufsynchronisation (barrier synchronization)

Beispiel: Ablaufsynchronisation II

Zwei Threads (A, B) sollen an bestimmtem Punkt aufeinander warten

Lösungsvorschlag 2:

A

```
V(sync_B) // B wecken
P(sync_A); // auf B warten
```

B

```
V(sync_A) // a wecken
P(sync_B); // auf A warten
```

Diese Lösung ist symmetrisch $\rightarrow$ Ablaufsynchronisation (barrier synchronization)

Beispiel: POSIX-Mutex

```
#include <pthread.h>
...
pthread_mutex_t mtx; // Mutex-Objekt
pthread_mutex_init(&mtx, NULL); // Initialisierung
pthread_mutex_lock(&mtx); // kritischen Bereich betreten
...
pthread_mutex_unlock(&mtx); // kritischen Bereich verlassen
pthread_mutex_destroy(&mtx); // Mutex loeschen
```

Deadlock I

Deadlock: Eine Gruppe von Prozessen ist in einem dauerhaft blockierten Zustand, weil jeder Prozess dieser Gruppe auf ein Ereignis wartet, das nur ein anderer Prozess dieser Gruppe auflösen kann.

Ursachen:

- Nutzung gemeinsamer Ressourcen, die über einen Semaphor abgesichert sind; das Warteereignis ist die Freigabe der Ressource
- Dateisperren (file locks), das Warteereignis ist die Freigabe der Datei

Deadlock I

Deadlock: Eine Gruppe von Prozessen ist in einem dauerhaft blockierten Zustand, weil jeder Prozess dieser Gruppe auf ein Ereignis wartet, das nur ein anderer Prozess dieser Gruppe auflösen kann.

Ursachen:

- Nutzung gemeinsamer Ressourcen, die über einen Semaphor abgesichert sind; das Warteereignis ist die Freigabe der Ressource
- Dateisperren (file locks), das Warteereignis ist die Freigabe der Datei

Deadlock II

Vorausstezung für einen Deadlock ist das Auftreten aller vier Bedingungen:

1. Mutual Exclusion: Jede Ressource ist genau einem Prozess zugeteilt oder allgemein erhältlich und nicht abgesichert
2. Hold and Wait Condition: Prozess besitzt mindestens eine Ressource und verlangt eine weitere
3. No Preemption: Das Betriebssystem kann die zugeteilt Ressource nicht zurückfordern und der Prozess gibt sie nicht freiwillig zurück
4. Zyklische Wartebedingungen: Zyklische Kette von zwei oder mehreren Prozessen, die alle auf eine Ressource warten, die vom Nachfolger reserviert ist

Deadlock: Lösungsansätze I

Ignorieren (ignoring, detection and recovery)

- Keine Vorsichtsmaßnahmen vorgesehen
- Watchdog und Reset bei Auftreten

Vorbeugung (dead lock prevention)

- Vorbeugung während der Programmentwicklung
- Mutual Exclusions verwenden

Vermeidung (dead lock avoidance)

- Hold and Wait: Alle Ressourcena uf einmal reservieren
- No Preemption; Zwangsfreigabe durch Betriebssystem
- Zyklische Wartebedingung: Ressourcen fürfen nur in aufsteigender Reihenfolge reserviert werden

Zusammenfassung

- Wiederholung Grundlagen Linux
- Grundlagen C
- Algorithmen und Datenstrukturen
- Komplexitätsbewertung
- Betriebssysteme

Unterlagen zur Klausurvorbereitung nutzen!
Programmieren lernt man nur durch Programmieren!

Zusammenfassung

- Wiederholung Grundlagen Linux
- Grundlagen C
- Algorithmen und Datenstrukturen
- Komplexitätsbewertung
- Betriebssysteme

Unterlagen zur Klausurvorbereitung nutzen!
Programmieren lernt man nur durch Programmieren!