Inhalt

I. Organisation 5
II. Einführung in die Programmiersprache C 5
III. Vom C-Programm zum laufenden Prozess 9
Programme und Prozesse 10
Prozesse 10
IV. Einleitung 11
1. Fallstudie 11
Zusammenfassung 11
V. Rechnerorganisation 11
V.1 Schichtenstruktur 11
1. Semantische Lücke 11
2. Mehrebenenmaschinen 11
V.2 Assembliersprache 12
4. Zusammenfassung 12
V.3 Maschinenprogramme 13
1. Vorwort 13
2. Programmhierarchie 13
3. Organisationsprinzipien 13
4. Zusammenfassung 14
V.4Betriebssystemebene 14
1. Teilinterpretation 14
2. Programmunterbrechung 15
3. Laufzeitkontext 16
4. Nichtsequentialität 17
5. Zusammenfassung 18
VI.1 Massenspeicher 18
1. Datei 18
2. Dateisystem 19
3. Zusammenfassung 21
VI.2 Arbeitsspeicher 21
1. Vorwort 21
2. Adressräume 21
3. Speicherverwaltung 23
4. Zusammenfassung 23
VI.3 Betriebssystemkonzepte: Prozesse 23

1. Einführung 23
2. Grundlagen 23
3. Ausprägung 24
4. Zusammenfassung 25
VII.1 Stapelverarbeitung 25
2. Einprogrammbetrieb 25
3. Mehrprogrammbetrieb 26
4. Zusammenfassung 28
VII.2 Dialogverarbeitung 29
1. Mehrzugangsbetrieb 29
2. Systemmerkmale 29
3. Echtzeitbetrieb 32
4. Zusammenfassung 33
VIII Zwischenbilanz 34
IX.1Prozessverwaltung: Einplanungskriterien 37
1. Programmfaden 37
2. Arbeitsweisen 38
3. Gütemerkmale 39
4. Zusammenfassung 40
IX.2 Prozessverwaltung: Einplanungsverfahren 40
1. Einordnung 40
2. Verfahrensweisen 41
3. Zusammenfassung 44
IX.3 Prozessverwaltung: Einlastung 44
1. Vorwort 44
2. Koroutine 44
3. Programmfaden 45
4. Zusammenfassung und Anhang 46
X.1 Nichtsequentialität 47
1. Kausalitätsprinzip **Fehler! Textmarke nicht definiert.**
2. Konkurrenz und Koordination 47
3. Verfahrensweisen 48
X.2 Befehlssatzebene 50
1. Monitor 50
2. Bedingungsvariable 51
3. Beispiel **Fehler! Textmarke nicht definiert.**
4. Zusammenfassung 52

X.3 Maschinenprogrammebene 52
1. Vorwort 52
2. Verdrängungssperre 53
2.1 Konzept 53
3. Bedingungsvariable 53
3.1 Definition 53
3.2 Unterbrechungsprotokoll 54
3.3 Signalisierungsprotokoll 54
4. Semaphor 55
4.1 Definition 55
4.2 Implementierung 55
4.3 Varianten 55
5. Zusammenfassung 56
X.4 Befehlssatzebene 57
1. Vorwort 57
2. Unterbrechungssteuerung 57
2.1 Prinzip 57
2.2 Implementierung ..... **Fehler! Textmarke nicht definiert.**
3. Schlossvariable 57
3.1 Definition 57
3.2 Implementierung 57
3.3 Diskussion 58
4. Nichtblockierende Synchronisation 58
4.1. Motivation 58
4.2. Prinzip 58
1.2 Elementaroperation 59
1.3 Anwendung ..... **Fehler! Textmarke nicht definiert.**
1.4 Diskussion 59
5. Zusammenfassung 60
XI Betriebsmittelverwaltung 60
1. Betriebsmittel 60
2. Verklemmung 61
3. Zusammenfassung 62
XII.1 Speicherverwaltung: Adressraumkonzepte 63
1. Grundlagen 63
2. Adressräume 63
3. Zusammenfassung 66

XII.2 Speicherverwaltung: Zuteilungsverfahren 66
1. Platzierungsstrategie 66
2. Speicherverschnitt 68
3. Zusammenfassung 70
XII.3 Speichervirtualisierung 70
1. Ladestrategie 70
2. Ersetzungsstrategie 70
3. Zusammenfassung 73
XIII. Dateisystem 73
1. Medien 73
2. Speicherung von Dateien 74
3. Beispiele: Dateisysteme unter UNIX und Windows ..... **Fehler! Textmarke nicht definiert.**
4. Dateisysteme mit Fehlererholung 76
5. Datensicherung 77

Teil A

I. Organisation

II. Einführung in die Programmiersprache C

Primitive Datentypen in C

- Ganzzahlen/Zeichen: char, short, int, long, long long
- Fließkommzahlen: float(32), double(64), long double(128)
- Boolscher Datentyp: _Bool (C99)
- Vorangestellte Typ-Modifizierer verändern die Bedeutung:
    - Vorzeichenbehaftet: signed
    - Vorzeichenlos: unsigned
    - Konstant: const

Verbund-Datentyp / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit
- Strukturdeklaration: struct Person { char name[20]; int alter; };
- Definition einer Variablen vom Typ der Struktur: struct Person p1;
- Zugriff auf ein Element der Struktur: p1.alter = 20;

Kontrollstrukturen – Schleifensteuerung

- Break: bricht die umgebene Schleife ab
- Continue: bricht den aktuellen Schleifendurchlauf ab; setzt das Programm am Schleifenkopf fort

Funktionen

- Regeln
    - Werden global definiert
    - Main()
    - Rekursion ist zulässig
    - Müssen deklariert sein, bevor sie aufgerufen werden

Parameterübergabe an Funktionen

- Call-by-value: es wird eine Kopie des tatsächlichen Parameters übergeben
    - Funktion kann den Parameter lesen, kann den Wert der Kopie ändern, kann über den Parameter keine Ergebnisse mitteilen

C-Präprozessor

- Makro-Präprozessor: bearbeitet das Programm bevor es an den C-Compiler übergeben wird: #define Makroname Ersatztext
- Einfügen von Dateien: #include < Dateiname > (Standard Header-Datei) oder #include „Dateiname“ (Header-Datei des Benutzers). Header-Dateien werden einkopiert

Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien verteilt werden → logisch zusammengehörende Daten sollten zusammengefasst werden
- Jede C-Quelldatei kann separat übersetzt werden (Option -c)
- Das Kommando cc kann mehrere .c-Dateien übersetzen und das Ergebnis –zusammen mit .o-Dateien – binden:
  % cc –o prog prog.o f1.o f2.c f3.c

Globale Variablen

- Gültig im gesamten Programm. Werden außerhalb von Funktionen definiert.
- Wenn in anderen Modulen definiert, müssen sie vor dem ersten Zugriff bekanntgemacht werden: extern-Deklaration = Typ und Name bekanntmachen
- Wenn möglich vermeiden, da: Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren; Änderungen können vorgenommen werden ohne das sie erwartet werden; Programme schwer zu pflegen

Globale Funktionen

- Sind generell global
- Wenn in anderen Modulen definiert, müssen sie vor dem ersten Zugriff bekanntgemacht werden: Typ, Name und Parametertyp
- Bilden äußere Schnittstelle eines Moduls

Einschränkung der Gültigkeit auf ein Modul

- Zugriff (Variable und Funktion) auf Modul beschränken: static
- Bilden zusammen den Zustand eines Moduls, die Funktion operiert auf diesem Zustand
- Hilfsfunktionen immer static

Lokale Variablen

- Definiert innerhalb einer Funktion oder eines Blocks
- Bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenen Blocks gilt die jeweils letzte Definition
- Außerhalb ihres Blocks kann man nicht darauf zugreifen

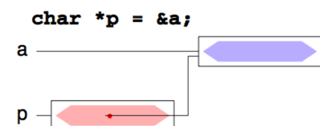
Lebensdauer von Variablen

- Bestimmt wie lang der Speicherplatz aufgehoben wird
- Zwei Arten
  - statisch (Static): bleibt für die gesamte Programmausführung
  - dynamisch (Automatic): nur für den Block
- Auto-Variablen:
  - Alle lokalen sind automatisch dyn. Variablen
  - Können durch beliebige Ausdrücke initialisiert werden
- Static-Variablen:
  - Speicher von Anfang bis Ende des Programms reserviert
  - Mit Schlüsselwort Static erhalten Lokale Variablen eine Lebensdauer über die gesamte Programmausführung hinweg
  - Können durch beliebige konstante Ausdrücke initialisiert wer

Zeiger(-Variablen)

Einordnung

- Konstante: Bezeichnung für einen Wert 'a' ≡ 0110 0001
- Variable: Bezeichnung für ein Datenobjekt a
- Zeiger-Variable: Bezeichnung einer Referenz auf ein Datenobjekt

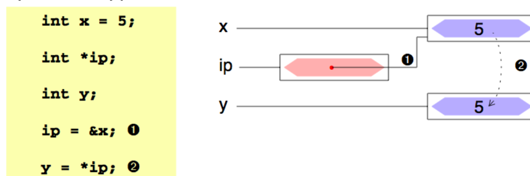


<u>Überblick</u>

- Eine Zeigervariable (pointer) enthält als Wert die Adresse einer anderen Variable → der Zeiger verweist auf die Variable
- Über die Adresse kann man indirekt auf die Variable zugreifen
- Bedeutung der Zeiger in C
  - Funktionen können ihre Aufrufparameter verändern (call-by-reference)
  - Dynamische Speicherverwaltung
  - Effizientere Programme
- Nachteile: Programmstruktur wird unübersichtlicher. Häufige Fellerquelle

<u>Definition von Zeigervariablen</u>

- Syntax : Typ *Name ;

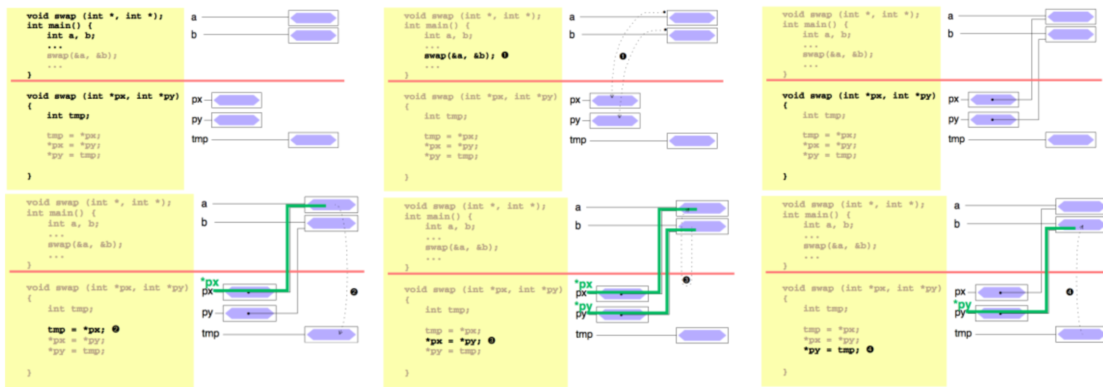


Adressoperatoren

- &x liefert Referenz auf den Inhalt der Variable x
- *x ermöglicht Zugriff auf den Inhalt der Variablen, auf die der Zeiger x verweist
  - * in Variablendefinition: ip ist eine Variabel vom Typ (int *), eine Variable die auf ein Objekt vom Typ (int) verweist
  - * als Operator in einem Ausdruck: ip ist eine Variable, die auf ein Objekt vom Typ (int) verweist, der Ausdruck *ip ermittelt den Inhalt dieses Objekts, also den int-Wert

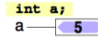
Zeiger als Funktionsargument

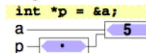
- Parameter und Zeiger werden by-value übergeben (Kopie)
- Die Funktion kann den tatsächlichen Parameter nicht verändern
- Über diesen Verweis kann die Funktion jedoch mit Hilfe des*-Operators auf die zugehörige Variable zugreifen und sie verändern → call-by-reference

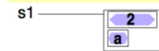


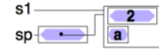
Zeiger als Struktur und Zusammenfassung

- „*“-Operator liefert die Struktur und „.*“Operator zum Zugriff auf Komponente
- Operatorenvorrang beachten: (*pstud).alter = 21;
- Syntaktisch schöner:
  pstud->alter = 21;





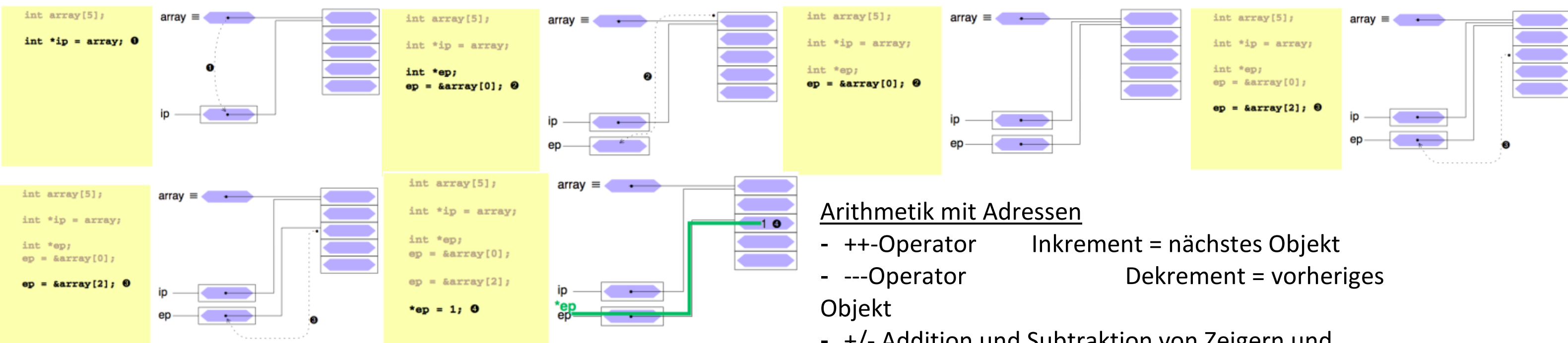




Zeiger und Felder

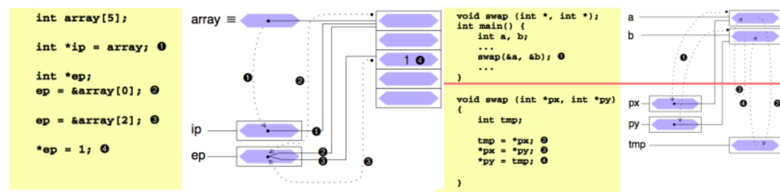
<u>Einführung</u>

- Ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes array = &array[0]
- Im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- Es gilt:



<u>Arithmetik mit Adressen</u>

- ++-Operator Inkrement = nächstes Objekt
- ---Operator Dekrement = vorheriges Objekt
- +/- Addition und Subtraktion von Zeigern und ganzzahligen Werten



<u>Zeigerarithmetik und Felder</u>

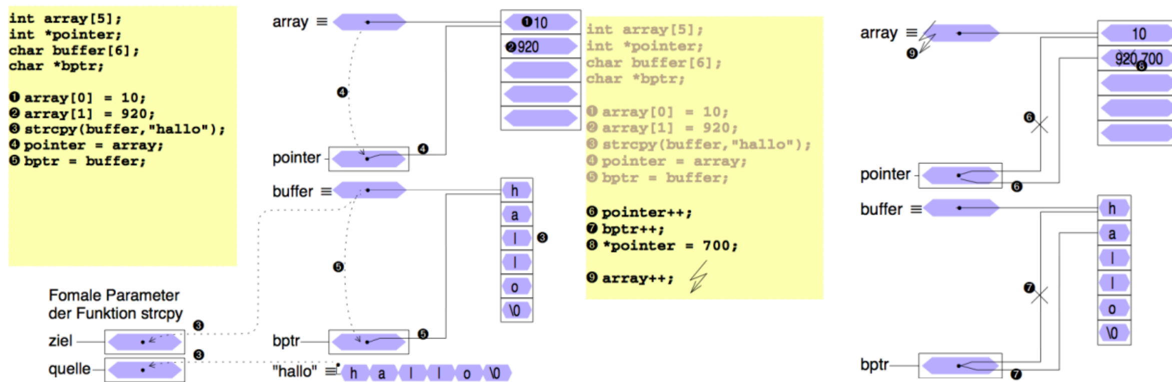
- Ein Feldname ist eine Konstante für die Adresse des Feldanfangs
- → Feldname ist ein ganz normaler Zeiger
- → aber keine Variablen → kein Modifikation erlaubt
- In Kombination mit Zeigerarithmetik lässt sich in C jede Feldoperation auf eine äquivalente Zeigeroperation abbilden

➤ für `int, array[N], *ip = array;` mit 0 ≤ i < N gilt:

```
        array ≡ &array[0] ≡ ip         ≡ &ip[0]
       *array ≡  array[0] ≡ *ip        ≡ ip[0]
*(array + i) ≡  array[i] ≡ *(ip + 1)  ≡ ip[i]
              array++ ≢ ip++
```

Fehler: array ist konstant!

- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden



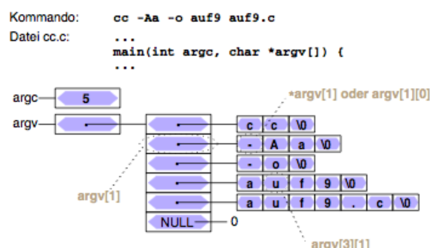
Dynamische Speicherverwaltung

- Felder können nur mit statischer Größe definiert werden
- Wird die Größe eines Feldes erst zur Laufzeit des Programms bekannt, kann der benötigte Speicherbereich dynamisch vom Betriebssystem angefordert werden: Funktion malloc: *malloc(size_t size)
    - Ergebnis: Zeiger auf den Anfang des Speicherbereichs
    - Zeicher kann danach wie ein Feld verwendet werden
- Dynamisch angeforderte Speicherbereiche können mit der free-Funktion wieder freigegeben werden : void free(void *ptr)
- Die Schnittstelle des Funktionen sind in der include Datei stdlib.h definiert

Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen, mit ‚\o' abgeschlossen sind
- Zeichenkette zur Initialisierung eines char-Feldes, Feldname ist konstanter Zeiger auf den Anfang
- Zeichenkette als char-Zeiger → Zeiger eine Variable, mit der Anfangsadresse der Zeichenkette
- Zuweisung char-Zeiger oder Zeichenkette an char-Zeiger bewirkt kein Kopieren von Zeichenketten
- Zeichenkette als tatsächlicher Parameter an Funktion → Kopie des Zeigers

Datenaufbau



Strukturen

Initialisieren von Strukturen

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {.re=0.5, .im=0.33};
```

Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden mit call by value
- Struktur können auch Ergebnis einer Funktion sein

Felder von Strukturen

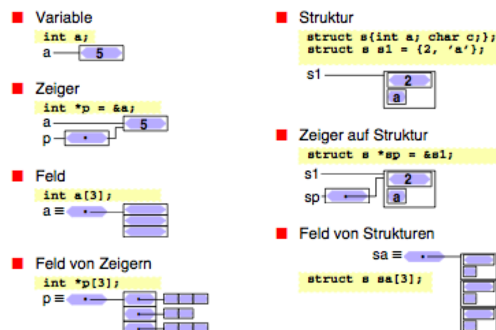
- Von Strukturen können –wie von normalen Datentypen- Felder gebildet werden

Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp

Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
- Variablendef.: <Rückgabetyp> (*<Variablenname>)(<Parameter>);



Ein-/Ausgabe

Standard Ein-/Ausgabe

stdin/stdout/stderr

Pipes: Standardausgabe eines Programms kann mit der Standardeingabe eines anderen verbunden werden:

- prog1 | prog2

automatische Pufferung: Eingabe von der Tastatur wird normalerweise von Betriebssystem zeilenweise zwischengespeichert und erst bein NEWLINE-Zeichen ('\n') übergeben

Öffnen und Schließen von Dateien

- Neben den Standart-E-/A-Kanälen kann ein Programm selbst weitere E-/A-Kanäle öffnen
- Öffnen eines E-/A-Kanals mit fopen: Ergebnis: Zeiger auf einen Datentyp FILE, der einen Dateikanal beschreibt
- Schließen eines E-/A-Kanals: int fclose(FILE *fp) schließt Kanal fp

Zeichenweise Lesen und Schreiben

- Lesen eines einzelnen Zeichens (geben int-Wert zurück)
  - Von der Standardeingabe int getchar()
  - Von einem Dateikanal int getc(FILE *fp)
- Schreiben eines einzelnen Zeichens
  - Auf die Standardausgabe int putchar(int c)
  - Auf einen Dateikanal int putc(int c, FILE *fp)
  - Schreiben das in c übergebene Zeichen
- Lesen einer Zeile von der Standardeingabe
  - Char *fgets(char *s, int n, FILE *fp)
  - liest Zeichen von Dateikanal fp in das Feld s bis entweder n-1 Zeichen gelesen wurden oder ´\n´ oder EOF gelesen wurde
- Schreiben einer Zeile
  - Int fputs(char *s, FILE *fp)
  - Schreibt Zeichen im Feld s auf Dateikanal fp

Formatierte Ausgabe

- Bibliotheksfunktion – Prototypen: printf, fprintf, sprintf, snprintf
- Format-Anweisung

%d, %i	int Parameter als Dezimalzahl ausgeben
%f	float Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
%e	float Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
%c	char-Parameter wird als einzelnes Zeichen ausgegeben
%s	char-Feld wird ausgegeben, bis '\0' erreicht ist

Formatierte Eingabe

- Bibliotheksfunktion: scanf, fscanf, sscanf
- Die Funktion lesen Zeichen von stdin, fp bzw. aus dem char-Feld s
- White space bildet jeweils die Grenze zwischen Daten, die interpretiert werden

III. Vom C-Programm zum laufenden Prozess

Übersetzen- Objektmodule

1. Schritt: Präprozessor

- Entfernt Kommentare, wertet Präprozessoranweisungen aus
- Zwischenergebnis kann mit cc –P datei.c als datei.i erzeugt werden oder mit cc –E datei.c ausgegeben werden

2. Schritt: Compilieren

- Übersetzt C-Code in Assembler
- Zwischenergebnis: cc –S datei.c als datei.s

3. Schritt: Assemblieren

- Erzeugt Maschinencode
- Standardisiertes Objekt-Dateiformat: ELF
- Zwischenergebnis: cc –c datei.c als datei.o

4. Schritt: Binden

- Programm ld: erzeugt ausführbare Datei
- Objekt-Datei werden zusammengebunden
- Nach fehlenden Funktionen wird in Bibliotheken gesucht

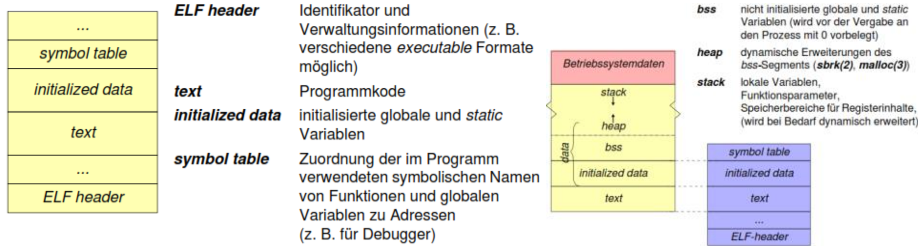
Binden und Bibliotheken

Statisch binden: Alle fehlenden Funktionen werden aus Bib genommen und in die ausführbare Datei einkopiert
Dynamisch Binden: Funktionen aus gemeinsam nutzbaren Bib werden nicht in die ausfährbare Datei einkopiert

Programme und Prozesse

Speicherorganisation eines Programms

- Definiert durch ELF-Format
- Wichtiges Element



Speicherorganisation eines Prozesses

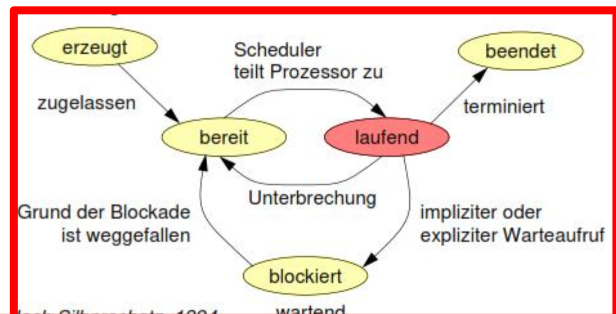
<u>Laden eines Programms</u>

- In eine konkrete Ausführungsumgebung kann ein Programm geladen werden
- Laden statisch gebundener Programme
  - Segmente der ausführbaren Datei werden in Speicher geladen
  - Speicher für Variablen wird bereitgestellt
  - Aufrufparameter werden in Stack kopiert
  - Main-Funktion wird angesprungen
- Laden dynamisch gebundener Programme
  - Spezielles Lade Programm ld.so
    - Segmente des ausführbaren Datei werden in Speicher geladen
    - Fehlende Funktion werden geladen
    - Noch offene Referenzen abgesättigt
    - Wenn notwendig werden Initialisierungsfunktionen aufgerufen
    - Parameter bereitgestellt
    - Main-angesprungen

Prozesse

Prozesszustände

- Erzeugt: wurde erzeugt besitzt aber noch nicht alle nötigen BM
- Bereit: Prozess besitzt alle nötigen BM und ist bereit
- Laufend: wird vom realen Prozessor ausgeführt
- Blockiert: Prozess wartet auf Ereignis; zum Warten wird er blockiert
- Beendet: Prozess ist beendet; Prozess muss aber im System bleiben



Prozesserzeugung (UNIX)

Erzeugen eines neuen UNIX-Prozesses:
Kind-Prozess ist eine perfekte Kopie des Vaters;
Unterschied: verschiede PIDs, ab fork() liefert verschiedene Werte für Vater & Kind

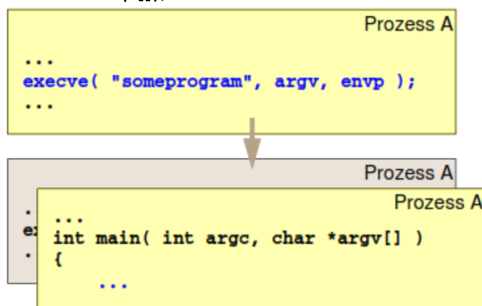
Vater
```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kind
```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Ausführen eine Programms (UNIX)

Prozess führt ein neues Programm aus

- int execve (const char *path, char *const argv[], char *const envp[]);



- vorheriges Programm endgültig beendet

Operationen auf Prozessen (UNIX)

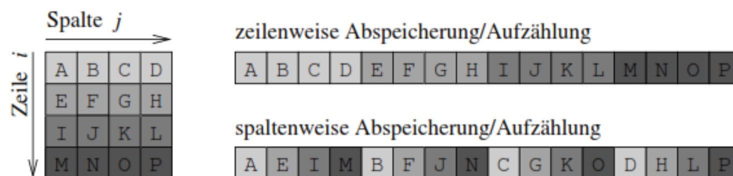
- Prozess beenden : void _exit (int status);
- Prozessidentifikator :
  - pid_t get(void); (eigene PID)
  - pid_t getppid(void); (PID des Vaterprozesses)
- Warten auf Beendigung eine Kindprozesses
  - pid_t wait(int *statusp);
  - wird solang blockiert bis Kindprozess terminiert

Teil B

IV. Einleitung

1. Fallstudie

Analyse



Virtueller Speicher

- Wenn z.B. gilt (Arbeitsspeicher) > (Hauptspeicher)
- Hauptspeicher ist Zwischenspeicher → Vordergrundspeicher
- Ungenutzte Bestände im Massenspeicher → Hintergrundsp.
- Problem: vgl: Zwischenspeicher

Betriebssystemarchitektur

- Schönheit, Stabilität, Nützlichkeit – Die drei Prinzipien von Architektur

Zusammenfassung

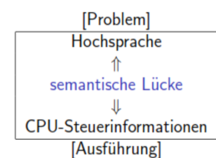
Betriebssystem

- Eine Menge von Programmen
- Verwaltet die Betriebsmittel einer Rechensystems
- Definiert sich über Funktionen

V. Rechnerorganisation

V.1 Schichtenstruktur

1. Semantische Lücke



- Verschiedenheit zwischen Quell- und Zielsprache
  - Faustregel:
    - Quellsprache → höheres Abstraktionsniveau
    - Zielsprache → niedriges Abstraktionsniveau

Fallstudie:

Verschiedenheit zwischen Quell- und Zielsprache

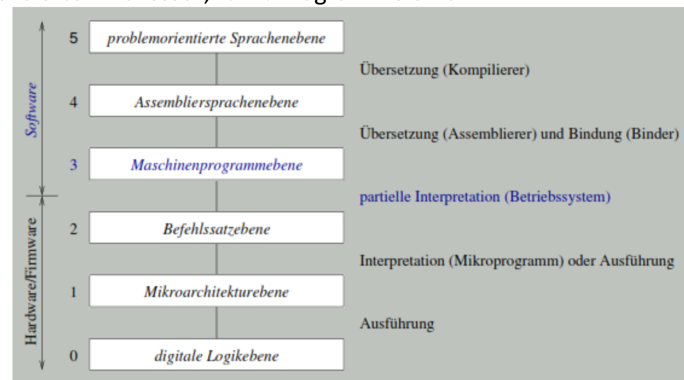
- Die Diskrepanz zwischen der vom Menschen skizzierten Lösung des Problems und dem dazu korrespondierenden, von einem Prozessor ausführbaren Maschinenprogramm ist beträchtlich
- Abstraktion half, sich auf das Wesentliche konzentrieren zu können: eine virtuell Maschine zur Berechnung die schrittweise abgebildet auf die reale Maschine „x86“

2. Mehrebenenmaschinen

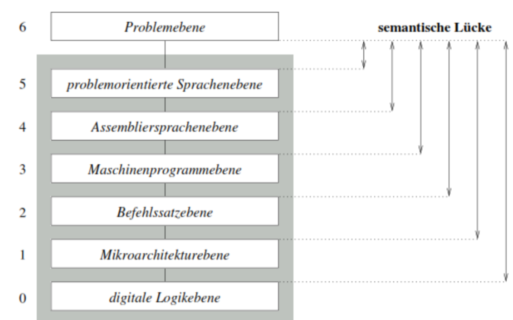
Maschinenhierarchie

- Aufgabenstellung → Programmlösung
  - Breite der semantischen Lücke variiert
  - Semantische Lücke schrittweise schließen: divide et impera („teile und herrsche“): einen Gegner in leichter besiegbare Untergruppen aufspalten

- Hierarchie virtueller Maschinen
- Programm sorgen für die Maschinenabbildung
    - Kompilator: ein Softwareprozessor, transformiert Programme in Quellsprache in semantisch äquivalente Programme einer Zielsprache
    - Interpret: ein in Hard-, Firm- oder Software realisierter Prozessor, führt Programme einer bestimmten Quellsprache „direkt“ aus



Maschinen und Prozessoren
- Hardware/Software-Hierarchie von Rechensystemen
- Softwaremaschine: Kompilierer / Assemblierer
    - Problemorientierte Programmiersprachebene
    - Assembliersprachebene
- Softwaremaschine: Teilinterpretierer →Betriebssystem
    - Maschinenprogrammebene (binärer Maschinenkode)
        - Legt Betriebsarten des Rechners fest, verwaltet Betriebsmittel und steuert bzw. überwacht die Abwicklung von Programmen
- Firm-/Hardwaremaschine: Interpretierer
    - Befehlssatzebene
    - Mikroarchitekturebene
- Hardwaremaschine: Ausführende
    - Digitale Logikebene (GST/ GTI)
- Abstraktionsniveau vs. Semantische Lücke



Entvirtualisierung
- Abbildung durch Übersetzung
    - Ebene $_5$ → Ebene $_4$ (Kompilierung): Befehle werden „1:N“ übersetzt, im Zuge der Transformation ggf. Optimierungsstufen durchlaufen
    - Ebene $_4$→ Ebene $_3$ (Assemblierung und Binden): Befehle „1:1“ übersetzt, symbolischen Maschinenkode auflösen
- Abbildung durch Interpretation
    - Ebene $_3$ → Ebene $_2$ (partielle Interpretation): Befehle werden typ- und zustandsabhängig verarbeitet, Befehl aktiviert ggf. ein Programm
    - Ebene $_2$→ Ebene $_1$ (Interpretation): Befehle als Folgen ausführen, Befehl löst Steueranweisung aus
- Übersetzung und Interpretation durch Prozessoren
    - Ebene $_5$: Kompilierer
    - Ebene $_4$: Assemblierer und Binder
    - Ebene $_3$: Betriebssystem
    - Ebene $_2$: Zentraleinheit (CPU)
- Zeitpunkte der Abbildungsvorgänge
    - Vor Laufzeit → statisch
    - Zur Laufzeit → dynamisch

V.2 Assembliersprache

4. Zusammenfassung

- Assembliersprache reflektiert das Programmiermodell einer CPU
    - Befehls-/Registersatz, Adressierungsarten
- Machschinenbefehle werden in "verständliche" Kürzel dargestellt
    - Mnemonik, symbolischer Maschinenkode
- Pseudobefehle steuern den Assemblier- und Bindevorgang
    - Objektmodule erzeugen, zum Lademodul zusammenfügen

Gründe für Programmierung in Assembliersprache
- Unzulänglichkeiten einer Hochsprache korrigieren

- Schwachstellen eines Kompilierers umgehen
- Elementarioperationen des Prozessors erzwingen
- Spezialbefehle der Hardware absetzen
- ...
- Funktionsweise einer CPU besser verstehen und beurteilen

V.3 Maschinenprogramme

1. Vorwort

<u>Betriebssystem = Programm der Befehlssatzebene</u>

BS Implementiert die Maschinenprogrammebene

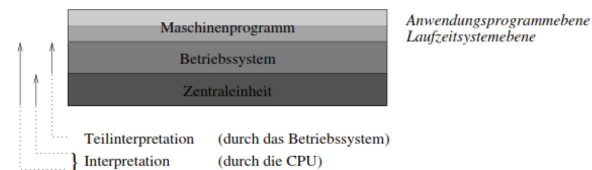
- zählt selbst nicht zur Klasse der Maschinenprogramme
- setzt normalerweise keine Systemaufrufe (an sich selbst) ab
- interpretiert eigentümliche Programme nur eingeschränkt partiell

Teilinterpretation von BS-Programmen

- bewirkt indirekt rekursive Programmausführungen im BS
- erfordert die Fähigkeit zum **Wiedereintritt** (re-entrance)
- ab einer bestimmten Ebene im BS ist dies unzulässig

<u>BS sollte es zulassen in der Ausführung eigentümlicher Programme unterbrochen werden zu können (nicht durch SystemCalls sondern durch Traps und Interrupts…)</u>

2. Programmhierarchie



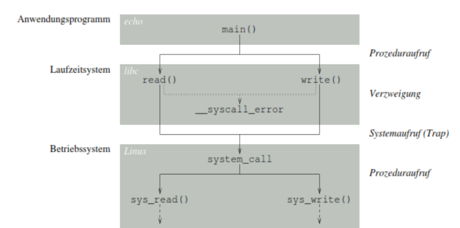
- Maschinensprache(n)
  - Maschinenprogramme setzen sich aus Anweisungen zusammen, die ohne Übersetzung von einem Prozessor ausführbar sind, werden durch Übersetzung generiert, repräsentieren sich technisch als Lademodul
  - Grundlagen für die Entwicklung von Maschinenprogrammen bilden Hoch- und vereinzelt Assembliersprachen auf Anwendungsprogramm-, Leufzeitsystem und Betriebssystemebene
- „Triumvirat" zur Ausführung von Anwendungsprogrammen
  - Maschinenprogramm = Anwendungsprogramm + Lauzeitsystem
    - selber Programmadressraums. normale Unterprogrammaufrufe aktivieren das Laufzeitsystem
  - Ausführungsplattform = Betriebssystem + Zentraleinheit(CPU)
    - zwischen den Ebenen erstreckt sich die Hard/Softwaregrenze
    - SystemCalls aktiveren das BS

Betriebssystemebene: Systemaufrufe verarbeiten

- Betriebssysteme realisieren einen Befehlsabruf- und –ausführungszyklus (fetch-execute cycle)
- Prozessorstatus des unterbrochenen Programms sichern
- Systemaufruf interpretieren
  1. Systemaufrufnummer abrufen
  2. auf Gültigkeit überprüfen und ggf. Fehlerbehandlung auslösen
  3. bei gültigem Operationskode, zugeordnete Systemfunktion ausführen
- Prozessorstatus wiederherstellen und zurückspringen
    - Beendigung der Teilinterpretation der CPU „mitteilen"

3. Organisationsprinzipien

Funktionen

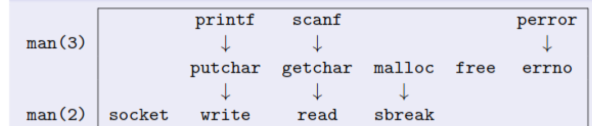


Zusammenspiel der Ebene des „echo"-Anwendungsfalls

Systemaufrufschnittstellen

- Aufrufstümpfe verbergen die technische Auslegung der Interaktion zwischen Anwendungsprogramm und BS
  - nach außen erschein ein Systemaufruf als normaler Prozeduraufruf
  - nach innen setzt ein Systemaufruf eine Programmunterbrechung ab
- Systemaufrufe sind spezielle „Prozedurfernaufrufe" die ggf. bestehende Schutzdomänen in kontrollierter Weise überwinden müssen
  - getrennte Adressräume für Anwendungsprogramm und BS
  - Ein-/Ausgabeparameter in Registern übergeben, „Trap" auslösen

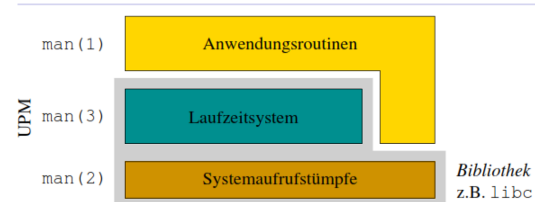
Laufzeitumgebung (runtime environment)
- Programmbausteine in Form eines zur Laufzeit zur Verfügung gestellten universellen Satzes von Funktionen und Variablen



Ensemble problemspezifischer Prozeduren
- Anwendungsroutinen (bsp. Funktion main(); setzen u.a. BS- oder Laufzeitsystemaufrufe ab)
- Laufzeitsystemfunktionen (Bibliotheksfkt. printf und malloc; setzt BS- oder (andere)Laufzeitsystemaufrufe ab)
- Systemaufrufstümpfe (Bibliotheksfkt. read und write; setzen Aufrufe an das BS ab - SystemCall -> synchrone Programmunterbrechung ~ Trap)
- Bilden zusammengebunden das Maschinenprogramm (Lademodul)

Komponenten

Grobstruktur von Maschinenprogrammen



- Statisch gebundenes Programm: Zum Ladezeitpunkt des Programms sind alle Referenzen aufgelöst;
    - Compiler und Assembler lösen lokale (interne) Referenzen auf
    - Binder löst globale (extern, .globl) Referenzen auf
- Schalter –static bei gcc(1) oder ld(1)
- Dynamisch gebundenes Programm: Bibliotheksfunktionen erst bei Bedarf einbinden; dynamische Bibliothek

4. Zusammenfassung

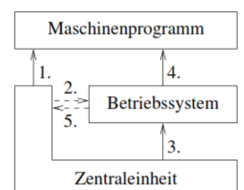
- Maschinenprogramme umfassen zwei Arten Eementaroperationen
    - (1) Maschinenbefehle der Befehlssatzebene und (2) Systemaufrufe
    - Befehle, die ohne Übersetzung von einem Prozessor ausführbar sind
- typisches Programm der Befehlssatzebene ist ein BS
    - das einen abstrakten Prozessor (Ebene(3)) implementiert
    - das einen Befehlsabruf- und –ausführungszyklus realisiert
- Teilinterpretation ist nicht strikt auf Maschinenprogramme bezogen
    - ein BS kann eigentümliche Programme teilinterpretieren (wenn diese etwa durch Traps oder Interrupts unterbrochen wurden)
    - dazu muss das BS allgemein zum Wiedereintritt fähig sein
- Maschinenprogramme als Ensemble problemspezifischer Routinen
    - Anwendungsroutinen, Laufzeitsystemfunktionen, Systemaufrufstümpfe
    - statisch/dynamisch zusammengebunden als Lademodul repräsentiert

V.4Betriebssystemebene

1. Teilinterpretation

Hybride Maschine
- Elementaroperationen der Maschinenprogrammebene
  Maschinenprogramme umfassen zwei Sorten von Befehlen
  1. Anweisung an das Betriebssystem (Ebene$_3$): explizit als Systemaufruf kodiert; implizit als Programmunterbrechung ausgelöst
  2. Anweisung an die CPU (Ebene$_2$)

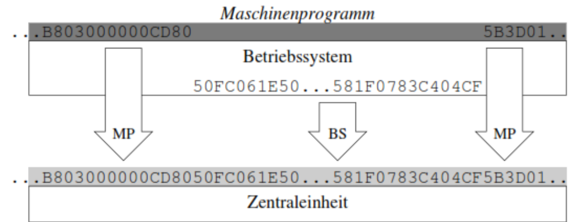
  Betriebssysteme fangen Ebene$_3$-Befehle ab, behandeln Ausnahmen



Ausführende Instanz ist immer die CPU, die nur Ebene$_2$-Befehle kennt
- Ebene$_3$-Befehle { werden „wahrgenommen", nicht ausgeführt / signalisieren eine Ausnahme (engl. *exception*)

- Ausführung von Maschinenprogrammen
    - Die Zentraleinheit interpretiert das Maschinenprogramm, setzt des Ausführung aus (Ausnahmesituation, Programmunterbrechung) startet das Betriebssystem und interpretiert die Programme des Betriebssystems befehlsweise

- Logischer Aufbau des Befehlsstroms für die Zentraleinheit



Zwischenzusammenfassung:

- Befehle der Maschinenprogrammebene, also Ebene$_3$-Befehle…
  - sind „normale" Befehle der Ebene$_2$, die die CPU ausführt
  - sind „ausnahmebedingte" Befehle, die das BS ausführt
- … implementieren z.B. Adressräume, Dateien, Prozesse
  - Interpreter dieser zusätzlichen Befehle ist das BS
- BS sind ausnahmsweise aktiv (werden von außerhalb aktiviert… Systemcall – programmiert; Ausnahmesituation – nicht programmiert) und deaktivieren sich immer selbst - programmiert

2. Programmunterbrechung

- Behandlung ist zwingend und grundsätzlich prozessorabhängig
  - vom realen und abstrakten Prozessor
  - genauer: von Zentraleinheit und Betriebssystem

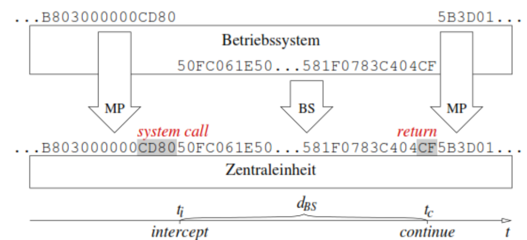
Trap

- Synchrone Programmunterbrechung - Trap
  - Unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
  - Systemaufruf, Adressraumverletzung, unbekanntes Gerät
  - Seitenfehler im Falle lokaler Ersetzungsstrategien
  - Trapvermeidung ohne Behebung der Ausnahmebedingung unmöglich

Trap — synchron, vorhersagbar, reproduzierbar

Ein in die Falle gelaufenes („getrapptes") Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und dem selben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle tappen.

- durch das Programm in Ausführung (≡ Prozess) selbst ausgelöst
  - bei Ablauf eines Maschinenbefehls des zugrundeliegenden Prozessors



- Unterbrechungsverzögerung: d_{BS} Ebene$_3$; begrenzt bei Echtzeitfähigkeit, unbegrenzt sonst

Interrupt

- Asynchrone Programmunterbrechung
  - Signalisierung externer Ereignisse
  - Beendigung einer DMA- bzw. E/A-Operation
  - Seitenfehler im Falle globaler Ersetzungsstrategien
  - Unterbrechungsverzögerung:
    d_{HW} Ebene konstan bei RICS variablen bei CISC;
    d_{BS} Trap BS kann d_{HW} beeinträchtigen

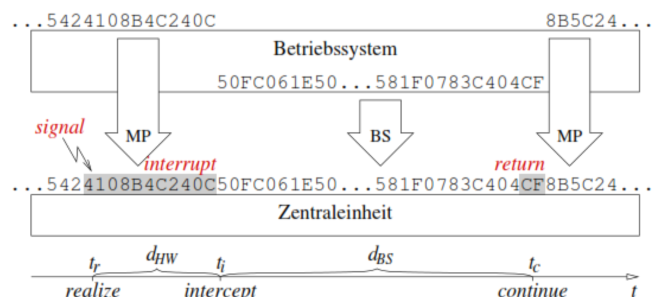
Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

Ein „externer Prozess" (z.B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programms.

Ob und ggf. an welcher Stelle das betreffende Programm unterbrochen wird, ist nicht vorhersehbar.

- durch einen anderen, externen (Soft-/Hardware-) Prozess ausgelöst

- Ausnahmesituationsbehandlung muss nebeneffektfrei verlaufen



Einordnung: Trap oder Interrupt?

- Division: Programmunterbrechung, wenn dann wird diese zufällig geschehen
- Programmierfehler der sich jedoch nicht zwingend auswirken muss: Unterbrechung verläuft synchron zum Programmablauf; die Unterbrechungsstelle im Programm ist vorhersagbar; der Zufall macht die Unterbrechung reproduzierbar
- Indirekte Adressierung: Programmunterbrechung trotz korrektem Text
- Seitenfehler vorbehaltlich eines virtuellen Speichers die Unterbrechung verläuft synchron zum Programmablauf

Diskussionsstoff: **Ersetzungsstrategie** (engl. *replacement policy*)

- optionales Merkmal der **Speicherverwaltung** eines Betriebssystems:
  lokal ⇝ Stelle vorhersagbar, Unterbrechung reproduzierbar . . . Trap?
  global ⇝ Stelle ¬ vorhersagbar, Unterbrechung ¬ reproduzierbar . Int.?

- die mögliche Unterbrechung ist „programmiert", sie wird vom Prozess selbst ausgelöst und bedeutet damit einen **Trap**!

3. Laufzeitkontext

Ausnahmen

- Ausnahmen von der normalen Programmausführung
  - Programmunterbrechungen werden durch Ereignisse hervorgerufen, die- aus Anwendungssicht- normalerweise unerwünscht sind:
    - Signale von der Peripherie
    - Wechsel der Schutzdomäne
    - Programmierfehler
    - Unerfüllbare Speicheranforderung
    - Einlagerung auf Anforderung
    - Warnsignale von der Hardware
  - Vorkehrungen zu Ereignisbehandlung sind im Betriebssystem unabdingbar, im Anwendungsprogramm dagegen nicht zwingend erforderlich: sind in beiden Fällen problemspezifisch auszulegen

Ausnahmebehandlung

- ***Wiederaufnahmemodell***: bei Behandlung der Ausnahmesituation führt zur Fortsetzung der Ausführung des unterbrochenen Programms; ein Trap kann, ein Interrupt muss so behandelt werden
- ***Beendigungsmodell***: die Behandlung der Ausnahmesituation führt zum Abbruch der Ausführung des unterbrochenen Programms; ein Trap kann, ein Interrupt darf niemals so behandelt werden
- Auslösung einer Ausnahme impliziert Kontextwechsel

Abrupter Wechsel des Laufzeitkontextes

- Programmunterbrechungen implizieren nicht-lokale Sprünge:

$$\text{vom} \left\{ \begin{array}{c} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\} \text{Programm zum} \left\{ \begin{array}{c} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\} \text{Programm}$$

- Aufrufe von und Rückkehr aus Unterbrechungsbehandlungsroutinen geschehen sporadisch und wechseln den Laufzeitkontext: erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung; Mechanismen liefert das behandelnde Programm/die tiefere Ebene
- Der Prozessorstatus unterbrochener Programme muss invariant sein

Sicherung/Wiederherstellung

- Prozessorstatus invariant halten
  - Ebene_2 (CPU) sicher bei Ausnahmen einen Zustand minimaler Größe: Statusregister und Befehlszeiger; möglicherweise aber auch den kompletten Registersatz; je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt
  - $\text{Ebene}_{3/5}$ (Betriebssystem/Kompilierer) sichert den restlichen Zustand

$$\text{d.h., alle} \left\{ \begin{array}{c} \text{dann noch ungesicherten} \\ \text{flüchtigen}^1 \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\} \text{CPU-Register}$$

  - Die zu ergreifenden Maßnahmen sind höchst prozessorabhängig
- Prozessorstatus sichern und wiederherstellen ungesicherte CPU
  - Train (trap/interrupt): Arbeitsregisterinhalte im RAM sichern und wiederherstellen; Unterbrechungsbehandlung durchführen; Ausführung des unterbrochenen Programms wieder aufnehmen
  - Beteiligte Prozessoren: CPU (Ebene_2), Betriebssystem (Ebene_3)
- Prozessorstatus sichern und wiederherstellen flüchtigen CPU

- Train: Inhalte flüchtiger Arbeitsregister sichern und wiederherstellen; Unterbrechungsbehandlung durchführen; Ausführung des unterbrochenen Programms wieder aufnehmen
- Beteiligte Prozessoren: CPU, Betriebssystem, Kompilierer

- Prozessorstatus sichern und wiederherstellen verwendeten CPU

Transparenz

- Unvorhersagbare Laufzeitvarianzen
    - Problem für determinierte Programme: lassen bei ein und derselben Eingabe verschiedene Abläufe zu; alle Abläufe liefern jedoch stets das gleiche Resultat; nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
    - Kritisch für echtzeitabhängige Programme:
        - Die Laufzeitumgebung dieser Programme muss echtzeitfähig sein
        - Nicht vergessen: Berücksichtigung aller Last- und Fehlerbedingungen

4. Nichtsequentialität

Konkurrenz

Nichtdeterministisches Programm

- Welche wheel-Werte gibt main() aus?

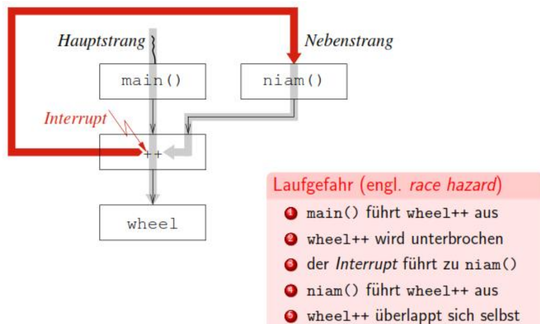
```
int wheel = 0;

main () {
    for (;;)
        printf("%u\n", wheel++);
}

void __attribute__ ((interrupt))
niam () {
    wheel++;
}
```

normalerweise fortlaufende Werte im Bereich[2] $[0, 2^{32} - 1]$, Schrittweite 1

ausnahmsweise dito, allerdings mit Schrittweite n, $0 \leq n \leq 2^{32} - 1$

- jenachdem, wie oft die Ausführung von niam() die von main() im kritischen Abschnitt wheel++ überlappt
- $n = 1$ impliziert nicht, dass keine Überlappung stattgefunden hat

Asynchronität von Programmunterbrechungen



Teilbarkeit von Operationen

- Wheel++: eine Elementaroperation der Ebene5; nicht zwingend auch eine Elop der Ebene4

	main()	niam()
Ebene 5	wheel++	

Ebene 4	movl _wheel,%edx
	leal 1(%edx),%eax
	movl %eax,_wheel
		incl _wheel

# Elop	3	1

Auszug nach gcc -O6 -S:

in main() teilbar

in niam() teilbar/unteilbar

- unteilbar nur bei Uni(kern)prozessoren

Unterbrechungsfall

- zeitliche Überlappung einer Veränderung des wheel-Wertes
    - niam()-Ausführung überlappt sich mit main()-Ausführung

Unterbrechungsbedingte Überlappungseffekte

niam()-Ausführung überlappt main()-Ausführung

_wheel	Befehls-folge	main()			niam()
		x86-Befehl	%edx	%eax	x86-Befehl
42	1	movl _wheel,%edx	42	?	
43	2				incl _wheel
43	3	leal 1(%edx),%eax	42	43	
43	4	movl %eax,_wheel	42	43	

Kritischer Abschnitt (engl. *critical section*) „++“

- zweimal wheel++ durchlaufen (main() und niam())
- zweimal gezählt, den Wert von wheel aber nur um eins erhöht

Wettlaufsituation

- Zwei Aktionen wetteifern um die Absicht, als erste ein Berechnungsergebnis herbeizuführen
- Eine Berechnung zeigt eine unerwartete kritische Abhängigkeit vom relativen Zeitverlauf von Ereignisses
- Ein potentielles Programm, sobald die Ausführung von Programmen nebenläufig möglich ist
- Programmabschnitte, die Wettlaufsituationen enthalten, bilden sogenannte kritische Abschnitte und sind speziell zu behandeln: physisch schützen oder logische Einheit

Koordinierung

Schutz vor unterbrechungsbedingten Überlappungen

- Lösungsansatz: sequentielle Ausführung erzwingen, Überlappungen vorbeugen; auf Elop eines tieferen Prozessors abbilden

- Abstraktion als grundsätzliche Vorgehensweise: Elop mit den funktionalen Eigenschaften des kritischen Abschnitts bilden: einen kritischen Abschnitt als Modul abkapseln; den modularisierten Programmtext passen synchronisieren

```
int wheel = 0;

main () {
    for (;;)
        printf("%u\n", fai(&wheel));
}

void __attribute__ ((interrupt))
niam () {
    fai(&wheel);
}
```

Optionale Merkmale der (System-) Software

`niam()`	• muss ggf. Überlappung durch sich selbst ins Kalkül ziehen • bei Verschachtelungen von Programmunterbrechungen • bei `niam()`-Ausführung in der Unterbrechungsbehandlung
`main()`	• muss ggf. Überlappung durch `niam()` ins Kalkül ziehen

Funktionale Abstraktion vom kritischen Abschnitt

- Randbedingung: Anweisung whell++ erhöht den Wert von wheel um 1 nachdem dieser als Ausdruckswert bestimmt wurde: Wert holen, dann inkrementieren; entsprechend funktioniert die benötigte Elop: int fai (int*)

Abbilden auf Komplexbefehl inc

```
int fai (int *ref) {
    int aux = *ref;

    asm volatile ("incl %0"
        : "=m" (*ref)
        :  "m" (*ref));

    return aux;
}
```

Abbilden auf Komplexbefehl add

```
int fai (int *ref) {
    int aux = *ref;

    asm volatile ("addl $1,%0"
        : "=m" (*ref)
        :  "m" (*ref));

    return aux;
}
```

Semantik: `return aux` liefert nicht zwingend `(*ref) - 1`

- Bestimmung des Ausdruckswerts und Veränderung der Variablen erfolgt durch Ausführung von (wenigstens) zwei Maschinenbefehlen
- dazwischen ist eine Unterbrechung möglich, die eine Überlappung der Ausführung der Anweisungsfolgen nach sich ziehen kann

Unterbrechungen sperren

```
int fai (int *ref) {
    int aux;

    asm volatile ("cli");
    aux = (*ref)++;
    asm volatile ("sti");

    return aux;
}
```

Spezialbefehl xadd nutzen

```
int fai (int *ref) {
    int aux = 1;

    asm volatile ("xaddl %0,%1"
        : "=g" (aux), "=m" (*ref)
        :  "0" (aux),  "m" (*ref));

    return aux;
}
```

Beachte: Ausführungsrechte und Arbeitsmodus einer CPU[a]

[a] Privilegierte Befehle sind nur im privilegierten Arbeitsmodus ausführbar.

`cli`	• privilegierter Maschinenbefehl, bedingt ausführbar	☹
`sti`	• privilegierter Maschinenbefehl, bedingt ausführbar	☹
`xadd`	• nicht-privilegierter Maschinenbefehl, unbedingt ausführbar	☺

`inc` bzw. `add`	• brauchbar, wenn die abweichende Semantik tolierbar ist • eine Entscheidung allein der Anwendungsebene
`cli/sti`	• nur gültig für Programme der Befehlssatzebene, beispielsweise Betriebssysteme • brauchbar, falls die Unterbrechungsverzögerung (S. 12) dadurch nicht zu stark beeinträchtig wird • eine Entscheidung maßgeblich der Anwendungsebene
`xadd`	• uneingeschränkt brauchbar

Beachte: Elop `fai()` allein garantiert noch keine korrekten Ausgaben

- die Operation garantiert nur konsistentes Zählen im Überlappungsfall
- Schrittweiten ungleich 1 bei der Ausgabe sind weiterhin möglich
  - bedingt durch Verdrängungslatenz bzw. Unterbrechungsfrequenz
- eine Lösung im Anwendungsprogramm selbst ist ggf. noch erforderlich
- Koordinierung paralleler Abläufe ist ein **querschneidender Belang**...

5. Zusammenfassung

- Maschinenprogramme werden durch **Teilinterpretation** ausgeführt
  - normalerweise laufen sie auf der Befehlssatzebene (CPU) ab
  - ausnahmeweise greift die Betriebssystemebene in die Ausführung ein
- **Programmunterbrechungen** leiten die Teilinterpretation ein
  - synchron zur Programmausführung TRAP
  - asynchron zur Programmausführung INTERRUPT
- die Behandlung von Unterbrechungen ist eine **Ausnahmesituation**
  - bedingt Sicherung und Wiederherstellung des Laufzeitkontextes
  - hat allgemein unvorhersagbare Laufzeitvarianzen zur Folge
- asynchrone Programmunterbrechungen bringen **Nichtsequentialität**
  - ggf. vorhandene kritische Abschnitte sollten modularisiert werden
  - das entstandene Modul der Wettlaufsituation entsprechend auslegen

VI.1 Massenspeicher

nicht-flüchtige Datenträger	Platte, Band, CD, DVD, ..., EEPROM
flüchtige Datenträger	RAM

1. Datei

Außensicht

- Arten von Dateien
  - Ausführbare Dateien: Binär- und Skriptprogramme: von einem Prozessor ausführbarer Programmtext und der Prozessor liegt in Hard-, Firm- und/oder Software vor

Binär	⇝	CPU, FPU, MCU, JVM, ..., Basic, Lisp, Prolog
Skript	⇝	`perl(1)`, `python(1)`, `{a,ba,c,tc}sh(1)`, `tcl(n)`

  - Nicht-ausführbare Dateien: Text-, Bild- und Tondaten von einem Prozessor verarbeitbare Programmdaten; der Prozessor legt in Form von Programmtext vor
- Bezeichnung von Dateien
  - Dateien sind „von außen" über symbolische Adressen erreichbar
    - Benutzerdefinierter Name von beliebiger aber maximaler Länge
    - Als Dateiname bekannter
  - „nach innen„ besitzt jede Datei einer numerische Adresse
    - Systemdefinierte Kennung einer Datenstruktur der Dateiverwaltung
    - Identifiziert den sogenannten Dateikopf
  - Tupel symbolische und numerische Dateiadresse bilden ein Paar
- Erweiterung eines Dateinamens
  - Dateinamensuffix eine meist durch eine Punkt vom Dateinamen abgegrenzt symbolische Erweiterung des Dateinamens

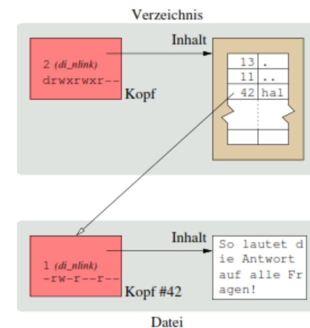
Suffix	Art	Programm	
.doc	Textdokumente	MS-Word	
.fm		Framemaker	maker(1)
.tex		LaTeX	latex(1)
.h	Programme	Präprozessor	cpp(1)
.c		Kompilierer	cc(1)
.s		Assemblierer	as(1)
.o		Binder	ld(1)

Innensicht

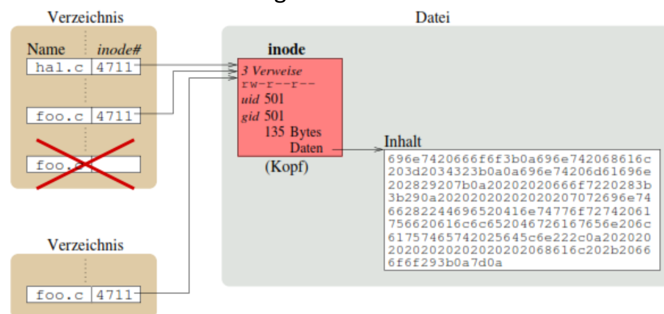
symbolische Adresse		numerische Adresse
Dateiname	~	Dateikopf
foo		4711

- Dateikopf und Dateikopfnummer
    - „inode“ enthält Dateiattribute: Eigentümer, Gruppenzugehörigkeit, Typ, Rechte, Zeitstempel, Anzahl der Verweise, Größe, Adresse der Daten auf dem Speichermedium
    - „inode number“ Index in einer Tabelle von Dateiköpfen: Die numerische Adresse der Datei
- Dateityp: Reguläre Datei und spezielle Datei
- Dateiverzeichnis: Katalog von symbolischen Namen definiert eine gemeinsamen Kontext implementier eine „Umsetzungstabelle“ speichert die Abbildung Dateiname → Dateikopfnummer

Bindung



- Verknüpfung von Dateiname und Dateikopf
    - Verzeichnisse sind „Spezialdateien“: die selbst einen Namen und Kopf haben; erreichbar über Verknüpfung; Name getrennt von Dateien gespeichert
- Referenzzähler
    - Buchführung über die Anzahl der Verknüpfungen zu einem Dateikopf geschieht über Verknüpfungszähler
    - di_nlink != 0 Datei wird referenziert
    - di_nlink == 0 Datei wird nicht referenziert
    - Veränderung des Verknüpfungszählerwertes erfolgt beim Eintragen bzw. Löschen von Verknüpfungen im Verzeichnis
    - Verzeichnis: mkdir(2)/rmdir(2); Datei: link(2)/unlink(2)
    - Namen müssen eindeutig sein



Systemfunktion

- UNIX Systemfunktionen
- Dateideskriptor: von der Dateiverwaltung implementierter eindeutiger Bezeichner einer geöffneten Datei; „nach außen“ meinst durch eine Ganzzahl repräsentiert

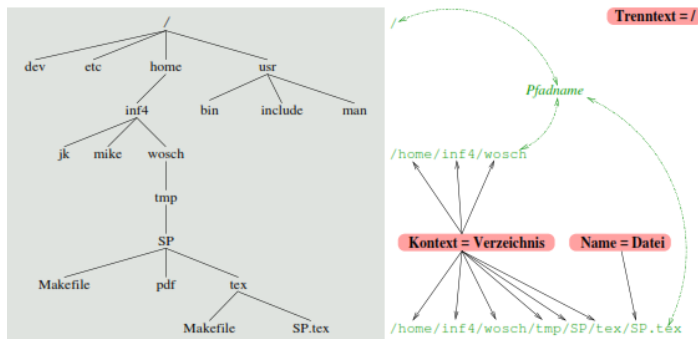
Linux, MacOS, SunOS

```
fd  = open(path, flags, mode)
num = read(fd, buf, nbytes)
num = write(fd, buf, nbytes)
off = lseek(fd, offset, whence)
ok  = close(fd)
ok  = stat(path, buf)
```

2. Dateisystem

Namensraum

- Aufbau von Namensräumen
    - Flache Struktur: definiert nur eine einzigen Kontext; Eindeutigkeit durch Namen gewährleistet
    - Hierarchische Struktur: definiert mehrere Kontexte; Eindeutigkeit durch Kontextnamen als Präfix; Sonderzeichen stehen für Separatoren
- Hierarchischer Namensraum

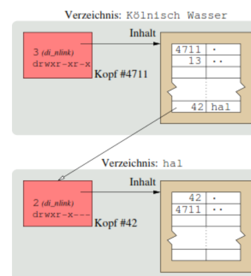


Formaler Aufbau eines (UNIX) Pfadnamens in EBNF [2]

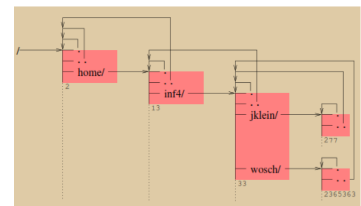
pathname	=	*resolver* | ([*resolver*], {*name*, *resolver*}, *name*);
resolver	=	{*separator*}–;
separator	=	"/";
name	=	{*character*}–;
character	=	*character set* – *separator*;
character set	=	ASCII;

z.B.: `/`, `.`, `..`, `foo`, `foo/bar`, `/foo`, `bar/`, `./bar/..`, `../foo/.//bar//`

- Navigation im Namensraum
  - Eindeutigkeit der symbolischen Adressen ist durch einen Pfad im Namensraum gegeben; Pfadname ist vollständiger Dateiname
- Spezielle Kontexte
  - Wurzelverzeichnis: wird vom System gesetzt
  - Arbeitsverzeichnis: ändert sich beim „Durchklettern" des Dateibaums
  - Heimatverzeichnis: wird vom System gesetzt
- Relative Adressierung von Kontexten
  - („dot"): aktuelles Arbeitsverzeichnis: benennt Verknüpfung zu selbigem Verzeichnis; erster Eintrag in jedem Verzeichnis
  - („dot dot"): aktuelles Elternverzeichnis: benennt die Verknüpfung zum übergeordneten Verzeichnis; zweiter Eintrag in jedem Verzeichnis
  - di_nlink # Verknüpfung die . referenzieren SunOS, Linux, speichert MacOSX

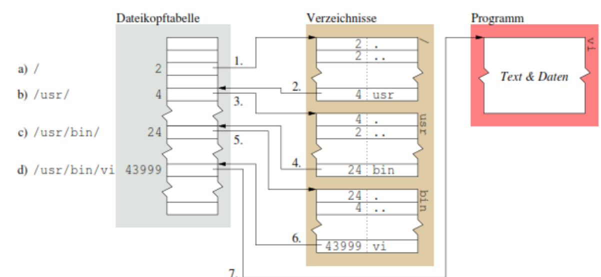


- Dynamische Datenstruktur „Dateibaum"
  - Verzeichnisname entsprechen einer Vorwärtsverkettung
  - . ist eine Selbstreferenz
  - .. entspricht einer Rückwärtsverkettung



Namensauflösung

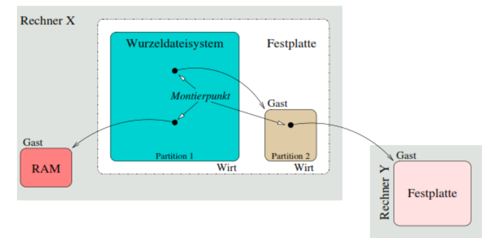
- Arten von Pfadnamen
  - Relativer Pfadname – vom gegenwärtigen Arbeitsverzeichnis ausgehend
  - Absoluter Pfadname – vom Wurzelverzeichnis ausgehend
- Bindung und Auflösung von Namen bzw. Pfadnamen
  - Namensbindung: bedeutet Abbildung der symbolischen Adresse in numerische Adresse, zum Erzeugungszeitpunkt; Pfandnamen mit Dateikopf assoziieren: creat, link
  - Namensauflösung bedeutet Umsetzung der symbolischen Adresse in numerische Adresse, zum Benutzerzeitpunkt; Dateikopf des Pfadnamens lokalisieren: open
- Auflösung von Namen bzw. Pfadnamen



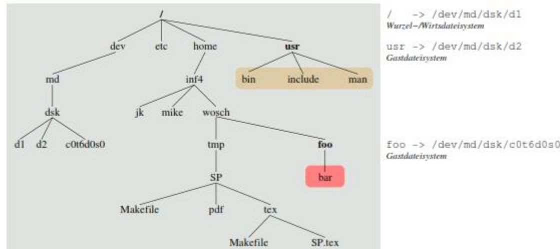
Querverweise

- Verwaltung von Dateien und Dateibäumen
  - Datenstrukturkomplex zur Verwaltung von Hintergrundspeicher: Dateisystemkopf (speichert Verwaltungsinfo, legt Grenzwerte des Dateisystems fest), Dateikopftabelle (Beschreibung von Dateien und Verzeichnisse), Datenblöcke (Speicherung der Inhalte)
  - Partition im Hintergrundspeicher: logische Unterteilung in einen Satz zusammenhängender Sektoren
- Montieren von Dateisystemen

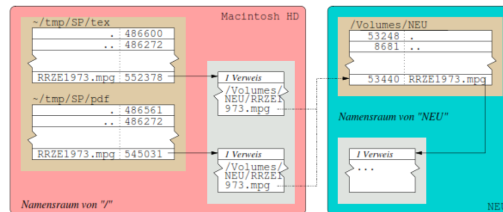
- Befestigungspunkt ist eine Stelle im Wirtsdateisystem, an der ein Gastdateisystem eingebunden werden kann
- Wirts- und Gastdateisystem bilden jeweils eigene Partitionen
- Ausgangspunkt ist das Wurzeldateisystem



- Hierarchiebildung von Dateisystemen
- Einbindung von Namensräumen



- Verweis hinein in einen anderen Namensraum
  - Ursprung ist der symbolische Name in Multics[1]



Systemfunktionen

- UNIX Systemfunktionen

Linux, MacOS, SunOS

```
fd = creat(path, mode)
    ↦ open(path, O_CREAT | O_TRUNC | O_WRONLY, mode)
ok = link(path1, path2)
ok = symlink(path1, path2)
ok = unlink(path)
ok = mkdir(path, mode)
ok = rmdir(path)
ok = mount(type, dir, flags, data)
ok = umount(dir, flags)
```

Linux, MacOS, SunOS

```
dirp = opendir(path)
dp   = readdir(dirp)
loc  = telldir(dirp)
void   seekdir(dirp, loc)
void   rewinddir(dirp, loc)
ok   = closedir(dirp)
```

- Vorsicht: readdir Implementierung ist eintrittsvariant

3. Zusammenfassung

- **Datei** als Abstraktion von Informationen tragenden Betriebsmitteln
  - das Dateiverzeichnis ist ein Katalog von Namen
  - ein Dateikopf führt Buch über die Systemdaten einer Datei
  - jedem Dateikopf ist eine Dateikopfnummer eindeutig zugeordnet
  - Referenzzähler verwalten Verzeichnis- und Dateiverknüpfungen
- **Dateisystem** als Ablageorganisation eines Datenträgers
  - Namensräume sind flach oder hierarchisch aufgebaut
  - unterschieden werden Wurze-. Heimat-, Eltern- Arbeitsverzeichnisse
  - symbolische Verknüpfung macht „physikalische Barrieren“ durchlässig
  - Gastdateisysteme lassen sich in Wirtsdateisysteme „montieren“

VI.2 Arbeitsspeicher

1. Vorwort

Begriff „Arbeitsspeicher“:

- Technische Informatik: flüchtiger Speicher eines Rechensystems, in Teilen auch nichtflüchtiger Speicher (Festwertspeicher, EEPROM), als Hauptspeicher unmittelbar von einer SPU ansprechbar
- Teilinterpretation von Speicherzugriffen: Massenspeicher; als virtueller Speicher mittelbar durch Betriebssysteme ansprechbar

2. Adressräume

Einführung

- Realer Adressraum (Hardware): ist durch Hardwarekonfiguration definiert; nicht jede Adresse ist gültig

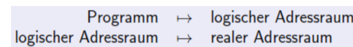
- Logischer Adressraum (Kompiler): abstrahiert von Hauptspeicher; alle Adressen gültig (zwischenschritt MMU, ist im Speicher)
- Virtueller Adressraum (Betriebssystem): auf Vorder- und Hintergrundspeicher abgebildet; erlaubt die Ausführung vollständig im RAM (Speicher oder Festplatte. Können auch Programme ausführen die teilweise ausgelagert sind)

Realer Adressraum

- Fossil: Toshiba Tecra 730CDT, 1996
    - Realer Adressraum verlangt hardwareabhängige Programme

Logischer Adressraum

- Ausführungsdomäne für Prozesse
    - Illusion von einem eigenen Adressraum für jedes im Hauptspeicher vollständig vorliegenden Programm
    - Adressabbildung erfolgt mehrstufig



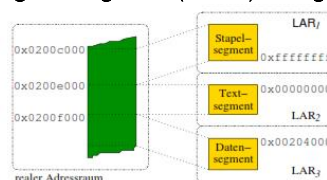
Programm	$\mapsto$	logischer Adressraum
logischer Adressraum	$\mapsto$	realer Adressraum

    - Logische Adressen sind mehrdeutig, reale dagegen eindeutig
- Abbildungszeitpunkte
    - Adressabbildung kann auf verschiedene Ebenen erfolgen

Entwicklungszeit	Programmierer(in)	Ebene $_6$	statisch
Übersetzungszeit	Kompilierer, Assemblierer	Ebene $_{5/4}$	
Bindezeit	Binder	Ebene $_4$	
Ladezeit	verschiebender Lader	Ebene $_3$	
Laufzeit	bindender Lader, MMU	Ebene $_{3/2}$	dynamisch

    - Zielkonflikt: Flexibilität vs. Effizienz: je später die Abbildung durchgeführt wird, desto höher das Abstraktionsniveau und geringer die Hardwareabhängigkeit/ höher des Systemaufwand und geringer des Spezialisierungsgrad
- Verantwortlichkeiten bei der Adressraumabbildung
    - Betriebssystem: Adressraumabbildung zur Ladezeit (Lader fordert Betriebsmittel an; Verwaltungsinfo für MMU werden aufgesetzt; neue Prozess wird der Einplanung zugeführt)
    - Hardware/MMU: Adressumsetzung zur Laufzeit (Verwendung der Deskriptoren gespeicherten Informationen)
    - Beachte: Verantwortung trägt alleine das Betriebssystem; MMU setzt das Vorgegeben Betriebssystem „gnadenlos“ um
- Segmentierung eines logischen Adressraums
    - Textsegment (Maschinenbefehle und andere Programmkonstanten; statische oder dynamische Größen; gemeinsam ausgelegt für mehrere Prozesse)
    - Datensegment (initialisiert Daten, globale Variablen; statisch oder dynamische Größe)
    - Stapelsegment (lokale Variablen, Hilfsvariablen und aktuelle Parameter; dynamische Größe)
- Ausrichtung von Segmenten
    - Segmente müssen angrenzen im logischen Adressraum liegen ggf. zur Bindezeit vom Binder ausgerichtet
    - Gründe: Mitbenutzung von Segmenten; differenzierter Schutz; dynamisches Binden und bindendes Laden; im virtuellen Speicher ablegen und halten
    - Hardwareabhängigkeit: sei N eine Anzahl von Bytes
    - Art der Abbildung: nach Seiten (N ist Seitengröße); nach Segmenten (N ist Größe des Segmentbestandteils)
    - Anfangsadresse ist Vielfaches von N

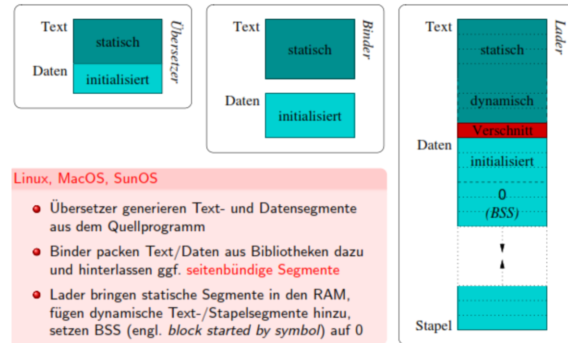
- Einrichtung der Segment (MMU) erfolgt zur Ladezeit



Kontrolliert durch das Betriebssystem ist die Mitbenutzung von Segmenten möglich, indem diese auf einen gemeinsamen Bereich im realen Adressraum abgebildet werden.

    - Verletzung der Segmentierung wird durch MMU verhindert und bewirkt eine Programmunterbrechung (seg_{log} Segmentanfang; Wert dieser Konstante wird zuvor passend ausgerichtet)
- Logischer Adressraum als Schutzdomäne

    - Adressraumisolation, eine Maßnahme zur Erhöhung von Sicherheit safety (Schutz von Menschen und Sachwerten vor dem Versagen technischer Systeme)/security (Schutz von Information vor „intelligenten“ Angreifern) in Rechensystemen, die im Mehrprogrammbetrieb gefahren werden
- UNIX Segmentierung



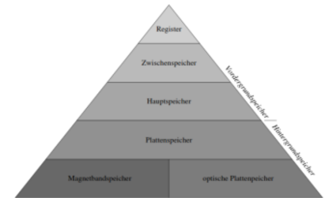
Virtueller Adressraum

Virtueller Speicher
- Illusion von einem eignen Adressraum für jedes unvollständig vorliegende Programm
- Adressabbildung erfolge mehrstufig: Programm → logisch; logisch → virtueller; virtueller → realer
- Virtuelle Adressen sind ebenso mehrdeutig wie logische Adressen

3. Speicherverwaltung

Speicherhierarchie

Speicherkonzepte und –medium
- Vordergrundspeicher: Hauptspeicher *RAM* (entsprechen im realen Adressraum; Zentralspeicher für Programmausführung; kann realen Adressraum überschreiten; kurzfristige Speicherung)
- Hintergrundspeicher: Massenspeicher *Band, CD, DVD* (über Rechnerperipherie angeschlossen; dient Datenablage virtueller Adressräume; ist größer als realer Adressraum; mittel- bis langfristige Speicherung)
- Funktional bringt Virtualisierung „Zugriffstransparenz“



Verwaltungshierarchie
- Arbeitsteilung von Laufzeit- und Betriebssystem
    - Laufzeitsystem: verwaltet den lokal vorrätigen Speicher einer logischen/virtuellen Adressraums
    - Betriebssystem: verwaltet den global vorrätigen Speicher
    - Trennung von Belangen: Aufteilung von Software nach unterschiedlichen Merkmalen, die sich funktional so wenig wie nur möglich überlappen

4. Zusammenfassung
- Arbeitsspeicher kann auch Massenspeicher beinhalten
    - unmittelbar von einer CPU ansprechbar: Hauptspeicher
    - mittelbar durch Betriebssysteme ansprechbar: virtueller Speicher
- Adressräume bilden verschiedene Ausführungsdomänen
    - realer Adressraum: gibt Aufbau/Struktur und Größe des Hauptspeichers vor
    - logischer Adressraum: abstrahiert von Aufbau/Struktur des Hauptspeichers
    - virtueller Adressraum: abstrahiert von der Größe des Hauptspeichers
- Speicherverwaltung erfolgt arbeitsteilig auf zwei Ebenen
    - auf der Maschinenprogrammebene durch das Laufzeitsystem
    - auf der Befehlssatzebene durch das Betriebssystem

VI.3 Betriebssystemkonzepte: Prozesse

1. Einführung

Gerichteter Ablauf eines Geschehens
- Betriebssysteme bringen Programme zur Ausführung, dazu Prozesse erzeugt, bereitgestellt und begleitet werden: Prozess ohne Programm nicht möglich; ein Programm beschreibt die Art des Ablaufs eines Prozesses (sequentiell oder parallel); Arten besteht Programmlauf aus Aktionen

2. Grundlagen

Programm
- Maschinenprogrammebene
    - Adressbereich und virtuelle Maschine SMC

- Nichtsequentielles Programm
  - Definition: Ein Programm P, das Aktionen spezifiziert, die parallel Abläufe in P selbst zulassen
  - Trotz Aktionen für Parallelität, sequentielle Programmabläufe
    - „Betriebssystem“ ist Inbegriff für „nichtsequentielles Programm“

Prozess

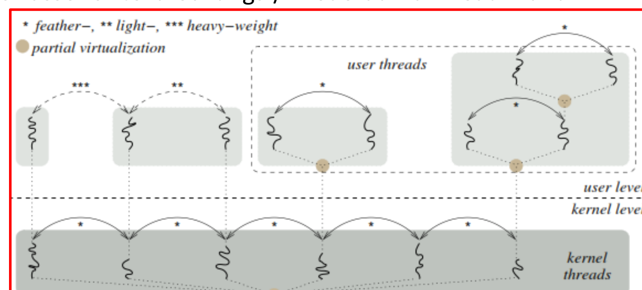
Definition Programmablauf: Ein Programm in Ausführung durch einen Prozessor

- Unteilbarkeit: Ganzheit oder Einheit einer Aktionsfolge, deren Einzelaktionen alle scheinbar gleichzeitig stattfinden
- Sequentieller Prozess
  - Definition: Ein Prozess der sich aus einer Folge von zeitlich nicht überlappenden Aktionen zusammensetzt
  - ! Ausführungsfaden ist ungleich Thread: Annahme zur technischen Umsetzung der Aktionsfolge werden nicht getroffen und sind hier auch ohne Belang. Ein Thread ist nur eine technische Option, um die Inkarnation eines sequentiellen Prozesses zu realisieren.
- Nichtsequentieller Prozess
  - Ein auch als „parallel“ bezeichneter Prozess,, der sich aus einer Folge von Aktionen zusammensetzt, die sich zeitlich überlappen können
- Gleichzeitiger Prozess (concurrent processes)
  - Definition: Wenigsten zwei aktionsfolgen einer nichtsequentiellen Prozesses, die sich zeitlich überlappen
  - sind sie unabhängig -> Prozesse heißen nebenläufig
  - gleichzeitige Prozesse stehen miteinander im Wettstreit (teilen sich den Prozessorkern, Speicher, Bus/Gerät; dadurch entsteht Interferenz)
- Gekoppelte Prozesse (interacting processes)
  - Definition: Gleichzeitige Prozesse, die interagierend auf gemeinsame Variablen zugreifen oder Betriebsmittel gemeinsam benutzen
  - geraten in Konflikt, wenn mind. einer der Prozesse… (eine gemeinsame Variable ändert oder ein gemeinsames unteilbares Betriebsmittel belegt)
  - daraus kann eine Wettlaufsituation entstehen
  - Konflikte werden durch Synchronisationsmaßnahmen unterbunden (blockierend, nicht-blockierend, reduzierend)
  - Koordination der Kooperation und Konkurrenz zwischen Prozessen

3. Ausprägung

Physisch

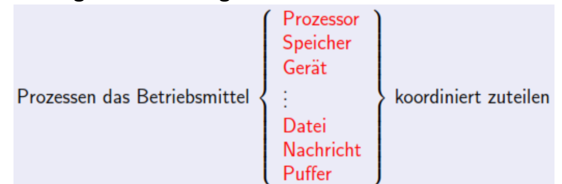
- Konsistenz: Koordination gekoppelter Prozesse
  - Beschaffenheit in Bezug auf sich überlappende Aktionsfolgen
- Koordinationsmittel: Austausch von Zeitsignalen
  - Semaphor: „nichtnegative ganze Zahl“ für die – nach der ursprünglichen Definition - zwei unteilbare Operationen definiert
  - P(s) koppelt den Prozess; V(s) Bereitstellung des blockierten Prozess
- Verortung: Betriebssystem- vs. Anwendungsebene
  - Prozesse sind verschiedenartig verankert: innerhalb (ursprünglich im Betriebssystem) und oberhalb (optional im Laufzeit- oder gar Anwendungssystem)
- Gewichtsklasse: Unterbrechungs-/Wiederaufnahmeaufwand



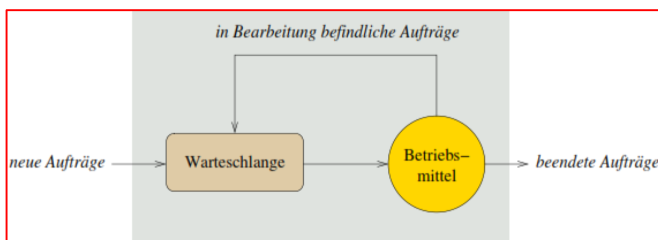
Logisch

- Planung der Einlastung

- Prozesseinplanung stellt sich im allgemein zwei grundsätzliche Fragestellungen: zu welchem Zeitpunkt sollen Prozesse eingespeist werden? In welcher Reihenfolge sollen eingespeiste Prozesse ablaufen?
  - Einplanungsalgorithmus: Implementiert die Strategie, nach der ein von einem Rechnersystem zu leistender Ablaufplan zur Erfüllung der jeweiligen Anwendungsanforderungen entsprechend aufzustellen und zu aktualisieren ist
- Reihenfolgebestimmung
  - Ablauf zu Betriebsmittelzuteilung erstellen



- Einplanungsalgorithmen
  - Ein einzelner Einplanungsalgorithmus ist charakterisiert durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden



- Warteschlangentheorie
  - Betriebssysteme durch die „theoretische/mathematische Brille“ gesehen
  - Einplanungsverfahren stehen und fallen mir Vorgabe der Zieldomäne
  - Scheduling ist ein Querschittsbelang
- Verarbeitungszustände
  - Prozessverarbeitung impliziert die Verwaltung mehrerer Warteschlangen

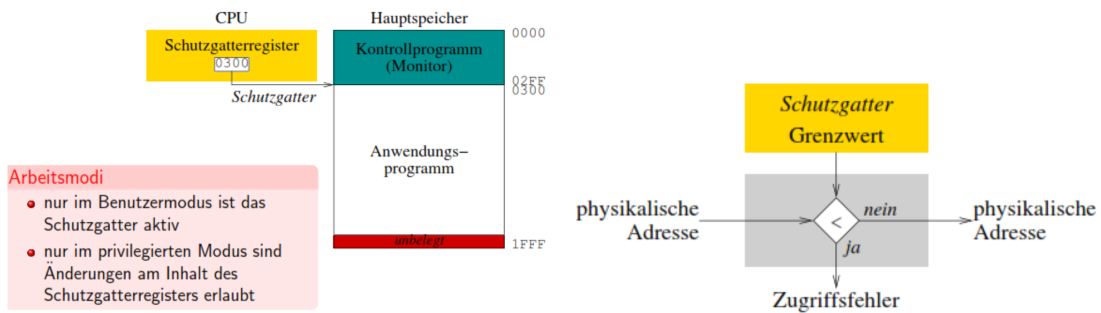
4. Zusammenfassung

- Programm: für eine Maschine konkretisierte Form eines Algorithmus
  - sequentiell – Aktionen, die nur serielle Abläufe in P zulassen
  - nichtsequentiell – Aktionen, die parallele Abläufe in P selbst zulassen
  - Aktion – Ausführung einer Maschinenanweisung
- Prozess: ein Programm in Ausführung durch einen Prozessor
  - sequentiell – eine Folge von zeitlich nicht überlappenden Aktionen
  - nichtsequentiell – eine Folge von zeitlich überlappenden Aktionen
  - gleichzeitig – unabh. zeitl. überlappende Aktionsfolgen
  - gekoppelt – abh. zeitl. überlappende Aktionsfolgen
- Planer: legt Zeitpunkt und Reihenfolge von Prozessen fest
  - Prozessinkarnationen werden Verarbeitungszuständen zugeschrieben
    - Zustandsübergänge (in EBNF): {{bereit, laufen} – , [blockier]}
  - UNIX: verdeängend, nicht-deterministisch, gekoppelt, Zeitmultiplex
- Betriebssystem: Inbegriff für ein nichtsequentielles Programm
  - asynchrone Programmunterbrechung (interrupt)
  - Simultanverarbeitung (multiprocessing) sequentieller Programme

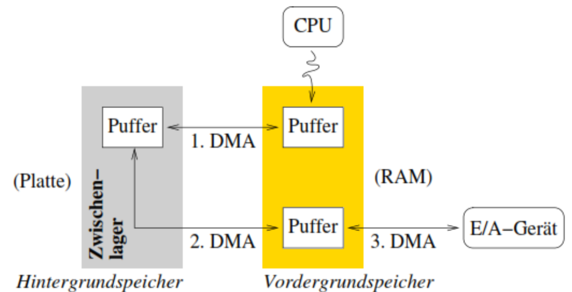
VII.1 Stapelverarbeitung

2. Einprogrammbetrieb

- Adressraumschutz durch Abteilung
  - Schutzgatter trennt den vom residenten Steuerprogramm und vom transienten Programm belegten Arbeitsspeicherbereich
  - Speicherverwaltung ist statisch, geschieht vor Programmausführung
  - Problem: Überbelegung des Arbeitsspeichers
- Schutzgatterregister



- Dedizierte Rechner/Geräte für verschiedene Arbeitsphasen
- Überlappte Ein-/Ausgabe
  - Speicherdirektzugriff: unabhängig von CPU, durch Kanalprogramm
  - Fall für asynchrone Programmunterbrechung
  - Problem: Leerlauf beim Auftragswechsel
- Überlappte Auftragsverarbeitung
  - Verarbeitungsstrom sequentiell auszuführender Programme/Aufträge
  - Auftragseinplanung liefert Abarbeitungsreihenfolge
  - Problem: Arbeitsspeicher, Monopolisierung der CPU, Leerlauf bei Ein-/Ausgabe
- Abgesetzte Ein-/Ausgabe
  - Pufferbereiche zur Entkopplung von Berechnung und Ein-/Ausgabe
  - Programmausführung in eine periodische Abfolge von zwei Phasen: CPU-Stoß (vergleichsweise schnell) und E/A-Stoß (vergleichsweise langsam)
  - Problem: Leerlauf im Wartezustand



3. Mehrprogrammbetrieb

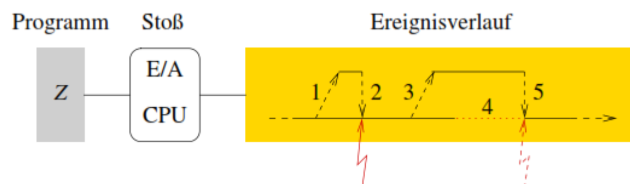
Multiplexverfahren

Maßnahme zur Leistungssteigerung

- Multiprogrammbetrieb und Simulationsbetrieb: Multiplexen der CPU, Abschotten der sich in Ausführung befindlichen Programme und Überlagerung unabhängiger Programmteile

Überlappte Ein-/Ausgabe im Einpro(gramm/zess)betrieb

- Problem:
  Durchsatz, Auslastung

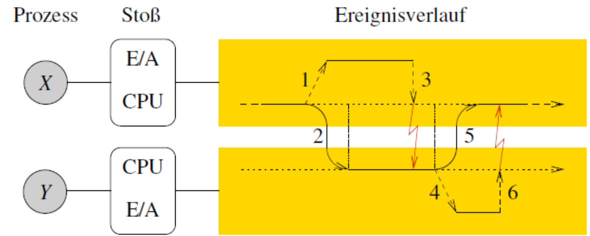


1. Programm$_z$ löst einen E/A-Stoß nebenläufig zum CPU-Stoß aus
2. ein Interrupt zeigt die Beendigung des E/A-Stoßes an
3. die Unterbrechungsbehandlung startet einen weiteren E/A-Stoß
4. Programm$_z$ muss auf E/A warten, sein CPU-Stoß endet
5. die CPU läuft leer, bis der E/A-Stoß zum Abschluss kommt

Multiplexen des Prozessors

- Die CPU "gleichzeitig" von mehreren Programmen benutzen
- Wartezeit von Programm(x) wird als Laufzeit von Programm(y) genutzt
  - aktives warten vermeiden
  - passives Warten betreiben (CPU einem anderen lauffähigen Programm zuteilen)
- **Prozess**: "Programm in Abarbeitung"
  - für jedes zu ladende Programm wird ein Prozess erzeugt
- Einplanung und Einlastung von Prozessen regeln die Verarbeitung lauffähiger Programme.

- BS → **mehrfädiges Programm**



Kooperation von Prozess$_x$ und Prozess$_y$, nach erfolgter Einplanung:

1. Prozess$_x$ löst einen E/A-Stoß und beendet seinen CPU-Stoß
2. Prozess$_y$ wird eingelastet und beginnt seinen CPU-Stoß
3. Beendigung des E/A-Stoßes von Prozess$_x$ unterbricht Prozess$_y$
   - die Unterbrechungsbehandlung überlappt sich mit Prozess$_y$
4. Prozess$_y$ löst einen E/A-Stoß und beendet seinen CPU-Stoß
5. Prozess$_x$ wird eingelastet und beginnt seinen CPU-Stoß
6. Beendigung des E/A-Stoßes von Prozess$_y$ unterbricht Prozess$_x$
   - die Unterbrechungsbehandlung überlappt sich mit Prozess$_x$

Problem
- Adressraumschutz, Koordination

Schutzvorkehrungen

Adressraumschutz – **Abschottung** von Programmen durch Eingrenzung/Segmentierung

- **Bindungszeitpunkt** von Programm- und Arbeitsspeicheradressen beeinflusst Modell zur Abschottung und umgekehrt:
- **vor Laufzeit** – Schutz durch *Eingrenzung*
  - Programme laufen im physikalischen Adressraum;
  - ein **verschiebender Lader** besorgt die Bindung zum *Ladezeitpunkt*,
  - die CPU überprüft die generierten Adressen zur Ausführungszeit
- **zur Laufzeit** Schutz durch **Segmentierung**
  - jedes Programm läuft auf eigenem log. Adressraum
  - Lader bestimmt Bindungsparameter
  - CPU/MMU besorgt Bindung zur Ausführungszeit (dyn. Programmumlagerung)
- In beiden Fällen richtet der Binder Programme relativ zu einer logischen Angangsadresse aus, meistens zur Basis 0

Adressraumschutz durch **Eingrenzung**

Begrenzungsregister legen Unter-/Obergrenze eines Programms im physikalischen Adressraum fest

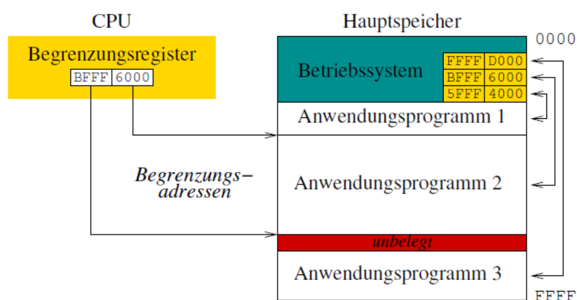
- für jedes Programm wird ein solches Registerpaar verwaltet (Softwareprototypen d. Adressüberprüfungshardware)
- bei Prozesseinlastung erfolgt die Abbildung auf die Hardwareregister

Speicherverwaltung ist statisch, geschieht vor der Programmausführung

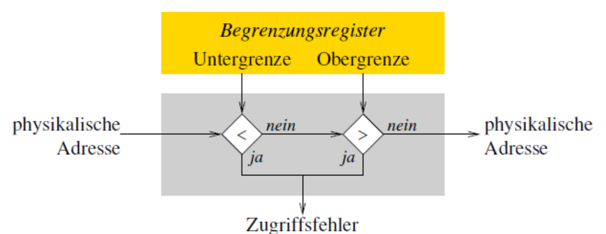
- der verschiebende Lader legt die Lage im Arbeitsspeicher fest
- zur Programmlaufzeit ist zusätzlicher Speicher nur bedingt zuteilbar
  - spätestens zum Ladezeitpunkt muss weiterer Bedarf bekannt sein

Problem: Fragmentierung, Verdichtung

Begrenzungsregister



Zugriffsfehler führt zum Abbruch (Trap)

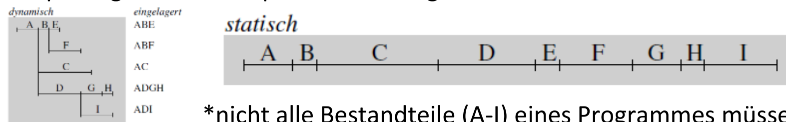


Dynamisches Laden

Überlagerung von Programmteilen im Arbeitsspeicher

- werden nur bei Bedarf nachgeladen (Nachladen ist programmiert, d.h. in den Programmen festgelegt; Entscheidung fällt zur Programmlaufzeit)
- Problem: finden der zur Laufzeit "optimalen" Überlagerungsstruktur

Einsparung von Arbeitsspeicher zur Programmlaufzeit

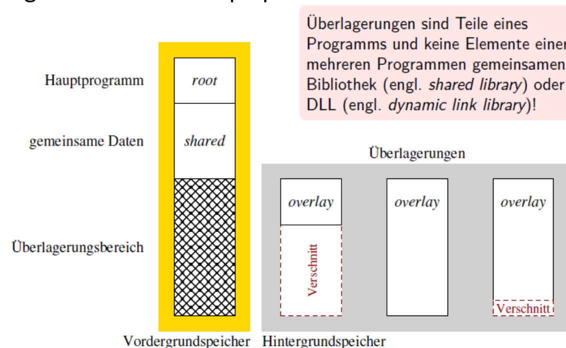


*nicht alle Bestandteile (A-I) eines Programmes müssen gleichzeitig im Arbeitsspeicher vorliegen, damit das Programm auch ausführbar ist
*wann welches Programmteil zur Ausführung gelangt, legt der dyn. Ablauf des Programms selbst fest

Überlagerungsstruktur eines Programms

- Programme die Überlagerungen enthalten sind grob dreigeteilt
  1. ein arbeitsspeicherresidenter Programmteil – root program
  2. mehrere in Überlagerungen zusammengefasste Unterprogramme
  3. ein gemeinsamer Datenbereich – common section
- Überlagerungen sind das Ergebnis einer **Programmzerlegung** zur **Programmier-, Übersetzungs-** und/oder **Bindezeit**
  - manuelle bzw. automatisierte Abhängigkeitsanalyse des Programms
  - Anzahl und Größte von Überlagerungen werden statisch festgelegt
  - lediglich aktiveren (d.h. laden) von Überlagerungen ist dynamisch

Organisation des Hauptspeichers



Simultanverarbeitung → Verarbeitung von mehreren Auftragsströme

Verarbeitungsströme sequentiell aufzuführender Programme/Aufträge

- pseudo-parallele Abarbeitung mehrerer Stapel im Multiplexverfahren
  - sequentielle Ausführung der Programme desselben Auftragsstapels
  - überlappende Ausführung der Programme verschiedener Auftragsstapel
- "wer [innerhalb des Stapels] zuerst kommt, mahlt zuerst"...

Stapelwechsel verlaufen kooperativ oder kommen verdrängend (cooperative, preemptive) zustande

- kooperativ bei programmierter I/O und Programmende
  - d.h. bei Beendigung des CPU-Stoßes eines Progresses
  - verdrängend bei Ablauf einer Frist (time limit)

Problem: interaktionsloser Betrieb, Mensch/Maschine-Schnittstelle

4. Zusammenfassung

Stapelbetrieb meint die sequentielle Abwicklung von Aufgaben

- anfangs manuelle Bestückung des Rechners durch die Programmierer
- später automatisierte Bestückung mittel **Kommandointerpretierer**
  - Programmierer organisierten ihren Auftragsstapel mittels Steuerkarten
  - Operateure übern{a,e}hmen die manuelle Bestückung des Rechners
- **Aufgabenverteilung** dehnt(e) sich auch systeminterne Abläufe aus
  - abgesetzter Betrieb, überlappte Ein-/Ausgabe
  - überlappte Auftragsverarbeitung, abgesetzte Ein-/Ausgabe

Mehrprogrammbetrieb hält mehrere Programme im Hauptspeicher

- der Prozessor wird im Multiplexverfahren betrieben: **Prozesse**
- Schutzvorkehrungen schotten Programmadressräume voneinander ab
- Hauptspeicherknappheit wird durch dynamisches Laden begegnet

- das Rechensystem damit sicher in **Simultanbetrieb** fahren können

Beachte
- Operateure bek{a, om}men interaktiven Zugang zum Rechensystem
- Anwerder arbeit(et)en mit dem Rechensystem interaktionslos

VII.2 Dialogverarbeitung

1. Mehrzugangsbetrieb

Dialogorientiertes Monitorsystem

Programme "im Vordergrund" starten und "im Hintergrund" verarbeiten
- interaktiv Aufträge annehmen, ausführen und dabei überwachen
- zur selben Zeit sind mehrere Programme parallel aktiv
- im Ergebnis werden mehrere Aufgaben "gleichzeitig" bearbeitet"

Mischbetrieb
- E/A-intensive Programme im Vordergrund ausführen
- CPU-intensive Programme im Hintergrund ausführen

Vordergrund	Hintergrund
Echtzeitbetrieb	Dialogbetrieb
Dialogbetrieb	Stapelbetrieb

Problem
- Hauptspeicher(größe)

Teilnehmerbetrieb

Dialoge können eigene Dialogprozesse absetzen

Zeitscheibe – time slice als "Betriebsmittel"
- jeder Prozess bekommt die CPU nur für eine Zeitspanne zugeteilt -> time sharing
  - Prozesse berechtigen, die CPUfür eine Zeit zur Abrarbeitung der ihnen zugeordneten Programme zu nutzen
- endet die Zeitspanne, werden die Prozesse neu eingeplant (ggf. Verdrängung des laufenden Prozesses. CPU wird zu Gunsten eines anderen Prozesses entzogen)

Zeitgeber – timer sorgen für zykl. Programmunterbrechungen
- das Intervall ist durch die Zeitscheibenlänge vorgegeben
- Monopolisierung der CPU ist nicht mehr möglich: CPU-Schutz

Problem: Arbeitsspeicher, Einplanung, Einlastung, I/O, Sicherheit

Teilhaberbetrieb

Dialoge teilen sich einen Dialogprozess
- Bedienung mehrere Benutzer durch ein Programm bzw. einen Prozess
  - gleichartige, bekannte Vorgänge an vielen Daten-/Dialogstationen
  - Wiederverwendung derselben residenten Dialogprozessinkarnation
- Endbenutzerdienst mit bestem, definiertem Funktionsangebot
  - Kassen, Bankschalter, Auskunft-/Buchungssyst.; **Transaktionssysteme**
- **Klient/Anbieter System** (client/server system)
  - sehr viele Dienstnehmer (service user),
  - einen/wenige Diensgeber (service provider)
- Problem: Antwortverhalten (weiche/feste Echtzeit), Durchsatz

2. Systemmerkmale

Multiprozessoren

SMP – symmetric multiprocessing

symmetrisch aufgebauter Multiprozessor
- Zwei oder mehr gleiche Prozessoren, die über ein gemeinsames Verbindungssystem gekoppelt sind
- jeder Prozessor hat gleichberechtigten Zugriff auf Hauptspeicher und I/O-Geräte – shared-memory access
- Zugriff auf alle Speicherzellen ist für alle Prozessoren gleichförmig - uniform memory access (UMA)
- Prozessoren bilden homogenes System und werden von demselben Betriebssystem verwaltet
- Problem: Koordination, Skalierbarkeit

SMP – shared – memory processor

Speichergekoppelter Multiprozessor

- Ein Parallelrechnersystem, bei dem sich alle Prozessoren denselben Arbeitsspeicher teilen, nicht jedoch gleichberechtigten Zugriff darauf haben müssen – sie können zusammen ein heterogenes System bilden
- *asymmetrischer SMP*
  - hardwarebedingter assym. Multiprozessorbetrieb
  - Programme/Betriebssystem sind ggf. prozessorgebunden
- *symmetrischer SMP*
  - Anwendungen geben die gewünschte Art des Parallelbetriebs vor
  - das Betriebssystem legt die Multiprozessorbetriebsart fest
- Problem: Koordination, Skalierbarkeit, Anpassbarkeit

Parallelverabeitungen
N Prozessoren können....

- N verschiedene Programme, N Prozessoren verschiedener Programme, N Prozesse desselben Programms

...parallel ausführen und jeder der N Prozessoren kann...

- M verschiedene Programme, M Prozesse verschiedener Programme, M Fäden desselben Programms

... pseudo-parallel (nebenläufig) im Multiplexbetrieb ausführen
(in einem Programm können mehrere Fäden/threads gleichzeitig aktiv sein)

Schutzvorkehrungen

Sicherheit – Schutz vor nicht autorisierten Zugriffen
Adressraumisolation: Schutz -> Eingrenzung/Segmentierung

- Zugriffsfehler führen zum Abbruch der Programmausführung. Die CPU bzw MMU fängt, das BS bricht ab
- i.A. keine selektive Zugriffskontrolle möglich und sehr grobkörnig

Prozessen eine **Befähigung** zum Zugriff erteilen:

- Verschiedene Subjekten unterschiedliche Zugriffsrechte (ausführen, lesen, schreiben, ändern) auf dasselbe Objekt (Datum, Datei, Gerät, Prozedur, Prozess) einräumen

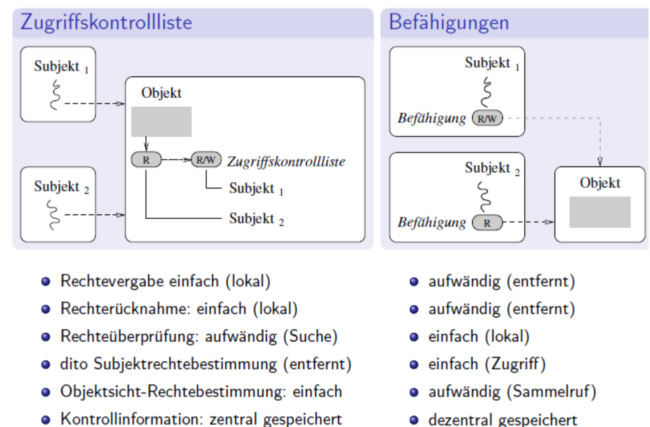
Alternative: Zugriffskontrollliste (access control list, ACL)
Problem: verdeckter Kanal (covered channel)

Zugriffsmatrix – access matrix

Einträge spezifizieren individuelle Zugriffsrechte der Subjekte auf Objekte: R read, W write, X execute

Implementierungsoptionen

- Totalsicht (in der dem Modell entsprechenden Tabellenform, ineffizient, wegen der i.d.R. dünn besetzten Matrix) :(
- Objektsicht (in der Form einer objektbezogenen **Zugriffskontrollliste**, spaltenweise Speicherung der Zugriffsmatrix) :)
- Subjektsicht (in Form subjektbezogener **Befähigungen**, zeilenweise Speicher und der Zugriffsmatrix) :)



Ringschutz – ring-protected paged segmentation

Multics, Maßstab in Bezug auf Adressraum-/Speicherverwaltung

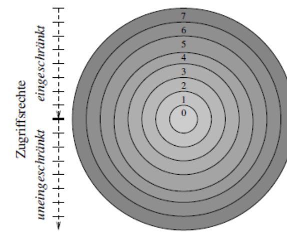
1. jede im System gespeicherte abrufbare Inforamtion ist direkt von einem Prozessor adressierbar und jeder Berechnung referenzierbar
2. jede Referenzierung unterliegt einer durch **Hardwareschutzringe** implentierten mehrstufigen Zugriffskontrolle

"trap on use", außnahmebedingtes dynamisches Binden/Laden

- Textsegmente werden bei Bedarf automatisch geladen
- mögl. synchr. Programmunterbrechung beim Prozeduraufruf

Problem: hochkomplexes System (Hardware/Software)

Schutzringe: Multics



Verwendung der Schutzringe:

0–3 Betriebssystem
- 0–1 Hauptsteuerprogramm
- 2–3 Dienstprogramme

4–7 Anwendungssystem
- 4–5 Benutzerprogramme
- 6–7 Subsysteme

Ringwechsel, Zugriffe
- kontrolliert durch die Hardware
- ggf. Teilinterpretation

Problem
- Schichtenstruktur, Ringzuordnung: **funktionale Hierarchie** [10]

Speicherverwaltung

Arbeitsspeicher und der Grad an Mehrprogrammbetrieb

Anzahl x Größe arbeitsspeicherresidenter Text-/Datenbereiche begrenzt die Anzahl der gleichzeitig zur Ausführung vorgehaltenen Programme

- variabler Wert, abh. von Struktur/Organisation der Programme
- eine Variable auch in Bezug auf die Fähigkeiten der Programmierer

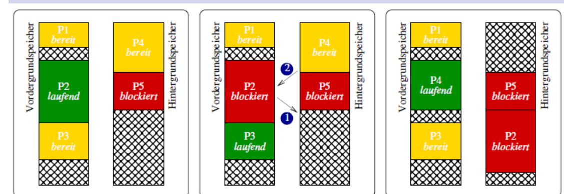
Umlagerung von solche Bereichen **gegenwärtig nicht ausführbereiter Programme** (swapping) verschiebt die Grenze nach hinten

- schafft Platz für ein oder mehrere andere (zusätzliche) Programme
- lässt mehr Programme zu, als insgesamt in den Arbeitsspeicher passt

Auslagerung von Bereichen sich in Ausführung befindlicher Programme (paging, segmentation) liefert weiteren Spielraum

Umlagerung nicht ausführbereiter Programme

Funktion der mittelfristigen Einplanung (engl. *medium-term scheduling*) von Prozessen



Ausgangssituation:
- P[1-3] im RAM
- P2 belegt die CPU

Umlagerung P2 & P4:
1. *swap out*
2. *swap in*

Resultat:
- P[134] im RAM
- P4 belegt die CPU

Problem
- Fragmentierung, Verdichtung, Körnigkeit

Umlagerung ausführbereiter Programme

Prozesse laufen trotz Hauptspeichermangel ab: **virt. Speicher**

- nicht benötigte Programmteile liegen im Hintergrundspeicher (on demand nachgeladen. evtl Folge andere Programmteile verdrängen)
- Zugriff auf ausgelagerte Teile bedeutet **partielle Interpretation**
  (log. bleibt das unterbrochene Programm weiter in Ausführung, physisch wird es jedoch im Zuge der Einlagerung blockieren)
- Aus- und Einlagerung wechseln sich mehr oder weniger intensiv ab

Der Arbeitsspeicherbedarf eines/mehrerer scheinbar (virt.) komplett im Arbeitsspeicher liegender und laufender/laufbereiter Programme kann die Größe des wirklichen (phys.) Arbeitsspeichers weit überschreiten.
Problem: Lade- und Ersetzungsstrategien

Körnigkeit (granularity) der Umlagerungseinheiten

Programmteile -> Pages oder Segmente

Seitenüberlagerung – paging (Adressraumteile fester Größe – Seiten, werden auf entsprechend große Teile – Seitenrahmen des Haupt-/Hintergrundspeichers abgebildet)

Segmentierung – segmentation (Adressraumteile var. Größe – Segmente werden auf entspr. große Teile des Haupt/Hintergrundspeichers abgebildet)

seitennummerierte Segmentierung – paged segmentation (Kombination beider Verfahren: Segmente in Seiten untergliedern)

Problem: interne Fragmentierung (Pages), externe Fragmentierung (Segmente)

Automatische Überlagerung durch partielle Interpretation

ähnl. zur Überlagerungstechnik, jedoch...

- Seiten-/Seg.anforderungen sind nicht in Anwendungen programmiert -> BS
  - Anforderungen stellt stellvertretend ein Systemprogramm
  - Ladeanweisungen sind so vor dem Anwendungsprogramm verborgen

- Zugriffe auf ausgelagerte Seiten/Segmente werden von der Befehlssatzebene abgefangen und ans BS weitergeleitet
    - Anwendungsprogramm wird von CPU/MMU unterbrochen
    - Zugriff wird vom BS partiell interpretiert
- Wiederholungen des unterbrochenen Maschinenbefehls sind in Betracht zu ziehen und vom BS zu veranlassen

Problem: Komplexität, Determiniertheit

Universalität

Universalbetriebssystem – "Allgemeinzweck" (general purpose operating system)

Verbesserung von Benutzerzufriedenheit:

- Selbstvirtualisierung, Konzentration auf das Wesentliche

Mulics – Multiplexed...; UNICS – Uniplexed Information and Computing System

3. Echtzeitbetrieb

Prozesssteuerung

Verarbeitung von Programmen in Echtzeit

Zustandsänderung von Programmen ist eine Funktion der **realen Zeit**

- korrektes Verhalten des Systems hängt nicht nur von den logischen Ergebnissen von Berechnungen ab
- zusätzlicher Aspekt ist der **physikalische Zeitpunkt** der Erzeugung und Verwendung der Berechnungsergebnisse

Echtzeitfähigkeit bedeutet Rechtzeitigkeit

Geschwindigkeit liefert keine Garantie, um rechtzeitig Ergebnisse von Berechnungen abliefern und Reaktionen darauf auslösen zu können.

- die im Rechensystem verwendete Zeitskala muss mit der durch die Umgebung vorgegebenen identisch sein
- **"Zeit" ist keine intrinsische Eigenschaft des Rechensystems**

Determiniertheit und Determinismus

- bei ein und derselben Eingabe sind verschiedene Abläufe zulässig, alle Abläufe liefern jedoch stets das gleiche Resultat
- zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
- die **Terminvorgaben** gelten als weich, fest oder hart

Echtzeitbedingungen

Terminvorgaben

Festgelegt durch die Umgebung ("externe Prozesse") des Rechensystems

weich – soft auch "schwach"

- Ergeb. einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist weiterhin von Nutzen, verliert aber Wert (Terminverletzung ist tolerierbar, Ton-/Bildrate)

fest – firm auch "stark"

- Ergebnis der nicht geleisteten Arbeit ist wertlos & wird verworfen (Übungsaufg.)
- Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch

hart auch "strikt"

- das Versäumnis eines fest vorgegebenen Termins kann eine "Katastrophe" hervorrufen
- Terminverletzung ist keinesfalls tolerierbar (Airbag)

Problem: Termineinhaltung unter allen Last- und Fehlerbedingungen

Terminvorgaben: Fest <=> Hart

fest/hart -> Terminverletzung ist nicht ausgeschlossen

- die Terminverletzung wird vom BS erkannt

fest -> plangemäß weiterarbeiten	hart -> sicheren Zustand finden
- das BS bricht den Arbeitsauftrag ab	- das BS löst eine **Ausnahemsituation** aus
- der nächste Arbeitsauftrag wird gestartet	- die Ausahmebehandlung führt zum sicheren Zustand
- ist transparent für die Anwendung	- ist **intransparent für die Anwendung**

4. Zusammenfassung

- **Mehrzugangsbetrieb** ermöglicht Arbeit und Umgang mit einem Rechensystem über mehrere Dialogstationen
    - im ***Teilnehmerbetrieb*** setzen Dialogstationen eigene Dialogprozesse ab
    - im ***Teilhaberbetrieb*** teilen Dialogstationen einen Dialogprozess
- wichtige **Systemmerkmale** zur Unterstützung dieser Betriebsart
    - Parallelverarbeitung durch (speichergekoppelte) ***Multiprozessoren***
    - über bloße Adressraumisolation hinausgehende ***Schutzvorkehrungen***
    - auf Programm(teil)umlagerung ausgerichtete ***Speicherverwaltung***
    - dem Allgemeinzweck gewidmete ***Universalbetriebssysteme***
- **Echtzeitbetrieb –** querschneidende Betriebsart des Rechensystems
    - Zustandsänderung von Programmen wird zur Funktion der ***realen Zeit***
        - "Zeit" ist keine intrinsische Eigenschaft des Rechensyst. mehr
    - "externe Prozesse" definieren ***Terminvorgaben***, die einzuhalten sind
        - die ***Echtzeitbedingungen*** dabei gelten als weich, fest oder hart

VIII Zwischenbilanz

Grundlagen von Betriebssystemen

Vorgänge innerhalb von Rechensystemen **ganzheitlich** verstehen

- **Zusammenspiel** { Hardware ↔ Software; ⋮; Anwendung ↔ Betriebssystem; ⋮; Treiber ↔ Gerät } **begreifen**

Grundzüge imperativer Systemprogrammierung (in C)

- im Kleinen praktizieren ⇝ Dienstprogramme
- im Großen erfahren ⇝ Betriebssysteme

Motivation

Rückgrat eines jeden Rechensystems

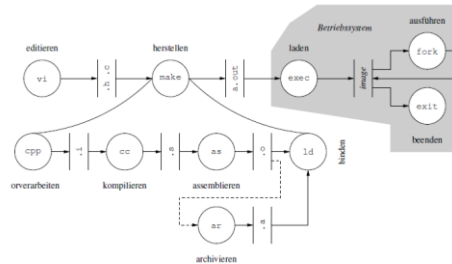
Betriebssysteme sind unerlässliches Handwerkszeug der Informatik
nicht alle müssen ein solches Handwerkszeug bauen/pflegen können
alle müssen jedoch mit dem Begriff/Produkt umgehen können

Betriebssysteme zu verstehen hilft, Phänomene zu begreifen

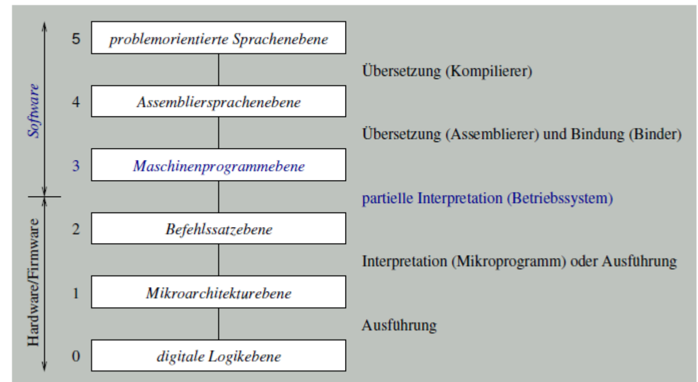
- unterschiedliches Systemverhalten erklären zu können
- Eigenschaften und Fehler auseinanderhalten zu können

Betriebssysteme immer im Anwendungskontext beurteilen:

- kein einzelnes System ist für alle möglichen Zwecke optimal geeignet



Betriebssystem: **abstrakter Prozessor/virtuelle Maschine** für Programme der Ebene 3



Unterbrechungen und Ausnahmesituationen

Teilinterpretation

Programmunterbrechungen zeigen Ausnahmebedingungen an und bedeuten die **partielle Interpretation** von Maschinenprogrammen:

Trap synchron, vorhersagbar, reproduzierbar
Interrupt asynchron, unvorhersagbar, nicht reproduzierbar

- macht determinierte Programme nicht-deterministisch
- **Nebenläufigkeit**, **kritischer Abschnitt**

Ausnahmebehandlung bringt Kontextwechsel mit sich, die **abrupte Zustandswechsel** das ausführenden Prozessors bewirken:

- vom unterbrochenen Programm zum behandelnden Programm ↓BS
- vom behandelnden Programm zum unterbrochenen Programm BS↑

Hardware und Software sind (funktional) äquivalent: Emulation

- die Nachahmung der Eigenschaften von Hardware durch Software

Adressraum

Ausführungs- und Schutzdomäne von Programmen

physikalischer Adressraum (Hardware) . . . . Ebene $_2$
- ist durch die jeweils gegebene Hardwarekonfiguration definiert
- nicht jede Adresse ist gültig, zur Programmspeicherung verwendbar

logischer Adressraum (Kompilierer, Binder, Betriebssystem) . . . Ebene $_{5/4/3}$
- abstrahiert von Aufbau/Struktur des Haupt- bzw. Arbeitsspeichers
- alle Adressen sind gültig und zur Programmspeicherung verwendbar

virtueller Adressraum (Betriebssystem) . . . . Ebene $_3$
- auf Vorder- und Hintergrundspeicher abgebildeter log. Adressraum
- erlaubt die Ausführung unvollständig im RAM liegender Programme

Speicher

Zusammenspiel aneinander angepasster Funktionen zu gegenseitigem Nutzen

Laufzeitsystem (bzw. Bibliotheksebene) verwaltet den lokal vorrätigen Speicher eines logischen/virtuellen Adressraums
- Speicherblöcke können von sehr feinkörniger Struktur/Größe sein
  - einzelne Bytes bzw. Verbundobjekte
- Verfahrensweisen orientieren sich (mehr) an Programmiersprachen

Betriebssystem verwaltet den global vorrätigen Speicher (d.h. den bestückten RAM-Bereich) des physikalischen Adressraums
- Speicherblöcke sind üblicherweise von grobkörniger Struktur/Größe
  - z.B. eine Vielfaches von Seiten
- Verfahrensweisen fokussieren auf Benutzer- bzw. Systemkriterien

Datei

Abstraktion von Informationen (über-) tragenden Betriebsmitteln

Aufbewahrungsmittel für zu speichernde Informationen
- kurz-, mittel-, langfristige Speicherung
- bleibende Speicherung (persistente Daten)

Kommunikationsmittel für kooperierende Prozesse
- gemeinsamer (externer) Speicher
- Weiterleitung von Informationen

Abstraktionsmittel für den Betriebsmittelzugang
- Hardware: CPU, RAM, Peripherie, . . .
- Software: Adressräume, Prozesse, . . .

Abbildung symbolische Adresse $\mapsto$ numerische Adresse:
- einen Dateinamen auf auf eine Dateikopfnummer abbilden
- ein Dateiverzeichnis (auch) als Umsetzungstabelle verstehen

Namensraum

Namen Kontexte zuordnen

Namensräumen eine Struktur aufprägen und dadurch einem Namen in „benutzerfreundlicher Weise“ eine eindeutige Bedeutung geben können:

flache Struktur eines einzigen Kontextes

hierarchische Struktur mehrerer Kontexte (d.h. flacher Strukturen)
- **hierarchischer Namensraum**
  - Pfadnamen zur Navigation im Namensraum
  - spezielle Kontexte (UNIX-artiger Systeme)
  - Bindung und Auflösung von Namen
- **Hierarchie von Namensräumen**
  - Montieren von Dateisystemen

Dateisysteme und Namensräume sind (logisch) verschiedene Dinge:
- das eine organisiert den Hintergrundspeicher (zur Dateiablage)
- das andere dient der Identifikation von Objekten (nicht nur Dateien)

Prozess

Abstraktes Gebilde vs. Identität einer Programmausführung

Gewichtsklasse eine Frage der Isolation von Adressräumen

Federgewicht keine Isolation
- der „reine“ Kontrollfluss: Faden

Leichtgewicht vertikale Isolation
- vom Betriebssystemadressraum

Schwergewicht horizontale Isolation
- von allg. Programmadressräumen

Einplanung Reihenfolgen festlegen, Aufträge sortieren
- Ablaufplan zur Betriebsmittelzuteilung erstellen
- Ablaufzustände von Prozessen fortschreiben

- charakteristische Eigenschaften der Einplanung/Einlastung von UNIX

Koordinationsmittel

Sequentialisierung nicht-sequentieller Programme

Semaphor abstrakter Datentyp zur Signalisierung von Ereignissen
- unteilbare Operationen auf eine Koordinationsvariable
  - **P** und **V** manipulieren eine nicht-negative ganze Zahl
- zur blockierenden Synchronisation gleichzeitiger Prozesse

Botschaft Synchronisation kombiniert mit Datentransfer
- Primitiven (Semantiken) zum Botschaftenaustausch
  - {*no-wait*, *synchronization*, *remote-invocation*} *send*
- Rollenspiele bei der Interprozesskommunikation
  - gleich- vs. ungleichberechtigte Kommunikation
- Kommunikationsendpunktadressen und Verbindungen

Betriebsmittel vs. synchrone/asynchrone bzw. blockierende IPC:

konsumierbares Betriebsmittel Nachricht (bzw. Botschaft)

wiederverwendbares Betriebsmittel Nachrichtenpuffer

a. Betriebsarten

Echtzeitbetrieb
Zeitabhängige Systeme

- die im Rechensystem verwendete Zeitskala muss mit der durch die Umgebung vorgegebenen identisch sein
- **Zeit ist keine intrinsische Eigenschaft des Rechensystems**

weich auch „schwach“ . . . . *soft*
- Terminverletzung ist tolerierbar

fest auch „stark“ . . . . *firm*
- Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch

hart auch „strikt“ . . . . *hard*
- Terminverletzung ist keinesfalls tolerierbar, Ausnahmefall

- querschneidender Belang der gesamten Systemsoftware + Anwendung

Mehrzugangsbetrieb
Interaktive Systeme

Dialogbetrieb Dialogstationen
- mehrere Benutzer gleichzeitig bedienen können

Hintergrundbetrieb Mischbetrieb
- Programme im Vordergrund starten

Teilnehmerbetrieb Zeitscheibe, *Timesharing*
- eigene Dialogprozesse absetzen können

Teilhaberbetrieb residente Dialogprozesse
- sich gemeinsame Dialogprozesse teilen können

Multiprozessorbetrieb Parallelrechner, SMP
- Parallelverarbeitung von Programmen

- Umlagerung (*Swapping*) kompletter Programme, virtueller Speicher

2. SP2

a. Ausblick

Vertiefung
Ausgewählte Kapitel der Systemprogrammierung

Prozessverwaltung
- Einplanung (klassisch, Fallstudien)
- Koroutinen, Programmfäden, Einlastung

Koordination
- ein-/mehrseitig, blockierend/nicht-blockierend
- Verklemmungen (Gegenmaßnahmen, Auflösung)

Speicherverwaltung
- Adressräume, MMU (Pentium)
- Disziplinen, virtueller Speicher, Arbeitsmenge

Dateiverwaltung
- Organisation des Hintergrundspeichers
- Datenverfügbarkeit (RAID)

Was noch wichtig ist – aber den Rahmen von SP2 sprengen würde. . .

Sicherheit
- Zugriffsmatrix, Befähigung, Zugriffskontrollliste
- Bell/LaPadula

Architektur
- Monolith, geschichtetes System, Minimalkerne
- Selbst-/Paravirtualisierung

TEIL C

IX.1Prozessverwaltung: Einplanungskriterien

1. Programmfaden

Grundsätzliches

Ablaufplanung von Threads erfolgt betriebsmittelorientiert und ist ggf. ereignis- oder zeitgesteuert.

- Laufbereitschaft hängt von erforderlichen BM und deren Verfügbarkeit ab
- Bereitstellung von BM kann sofortige Einplanung von Threads bewirken
- oder die Einplanung erfolgt in fest vorgegebenen Zeitintervallen

Einplanung ist der Vorgang der Reihenfolgebildung von Aufträgen und
Einlastung ist der Moment der Zuteilung von BM!!!

Fadenverläufe

Stoßbetrieb (burst mode)

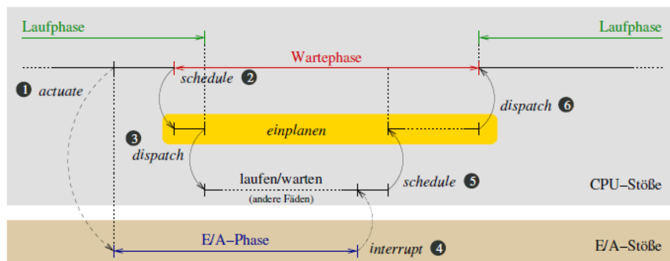
Laufphase: CPU-Stoß

- aktive Phase eines Threads (Rechenphase) in der alle zur Ausführung erforderlichen BM verfügbar sind
- Thread ist eingelastet, ihm wurde die CPU zugeteilt

Wartephase: E/A-Stoß

- inaktive Phase eines Threads (nicht alle erforderl. BM sind verfügbar)
- auf I/O warten bedeutet auf BM warten
  - konsumierbare BM: Eingabedaten, Nachrichten, Signale
  - wiederverwendbare BM: Puffer, Geräte, ..., die CPU
- die BM werden letztlich durch andere Fäden bereitgestellt (E/A-Gerät kann dabei als "externer Faden" betrachtet werden)

Lauf-, E/A- und Wartephasen von Fäden



- ein Faden läuft physisch solange, bis er den Prozessor freiwillig abgibt[a]
- logisch kann er sich zuvor jedoch bereits in der Wartephase befinden

[a]Fäden geben den Prozessor immer von selbst ab. Sie können gezwungen werden, dies dann auch zu gegebener Zeit zu tun ⇝ Verdrängung (S. 18).

Fäden durchlaufen (im BS) einen Kontrollfluss zur **Einplanung** und **Einlastung** anderer Fäden:

1. der laufende Faden stößt einen E/A-Vorgang an (*actuate*)
2. er wartet passiv auf die Beendigung der Ein-/Ausgabe (*schedule*)
   - Anforderung eines wiederverwend-/konsumierbaren Betriebsmittels
3. und lastet einen eingeplanten, laufbereiten Faden ein (*dispatch*)
4. die Beendigung der Ein-/Ausgabe wird signalisiert (*interrupt*)
   - Bereitsstellung des konsumierbaren Betriebsmittels „Signal“
5. der auf dieses Ereignis wartende Faden wird eingeplant (*schedule*)
6. der Faden wird eingelastet, sobald er an der Reihe ist (*dispatch*)

Sonderfall: ein Faden plant und lastet sich selbst ein, wenn sich die CPU mangels anderer laufbereiter Fäden im Leerlauf (engl. *idle state*) befindet

Leistungsoptimierung

Fäden als Mittel zum Kaschieren von Totzeiten

- Überlappung von Lauf- und Wartephasen erhöht die Rechenauslastung (Wartephase als Laufphase für andere Threads nutzen, Stöße anderer Fäden zum "Auffüllen" von Wartephasen nutzen)
- Auslastung von CPU <u>und</u> Peripherie steigert sich
  - CPU kann nur einen CPU-Stoß verarbeiten aber es können mehrere E/A-Stöße laufen
  - CPU und E/A-Geräte sind andauernd mit Arbeit beschäftigt
- bei weniger Prozessoren als Fäden, sind Fäden zu serialisieren

2. Arbeitsweisen

Ebenen

langfristige Einplanung [s-min]

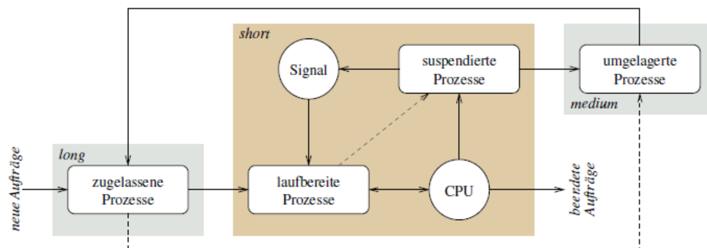
- Lastkontrolle, Grad an Mehrprogrammbetrieb einschränken
- Programme laden und/oder zur Ausführung zulassen
- Prozesse der mittel- bzw. kurzfristigen Einplanung zuführen

mittelfristige Einplanung [ms-s]

- Teil der Umlagerungsfunktion (swapping)
- Programme vom Hinter- in den Vordergrundspeicher bringen
- Prozesse der langfristigen Einplanung zuführen

kurzfristige Einplanung [µs-ms}

- Einlastungsreihenfolge der Prozesse festlegen – obligatorisch
- Voraussetzung für Mehrprozessbetrieb (laufbereite Prozesse brauchen zuteilung des wiederverwendbaren BM "CPU" -> Start ihrer Laufphase und suspendierte Prouesse warten auf konsumierbare BM "Signal" -> Ende ihrer Wartephase)



Ebenenübergänge

Prozesse haben in abh. von der Einplanungsebene zu einem Zeitpunkt einen logischen Zustand: kurzfristig (bereit, laufend, blockiert), mittelfristig (schwebend bereit, schwebend blockiert), langfristig (erzeugt, gestoppt, beendet)

Anmerkung: Anwendungsfälle legen fest, welche der Einplanungsebenen von einem BM wirklich zur Verfügung zu stellen sind, nicht umgekehrt.

<u>kurzfristige Einplanung (Festlegung der Prozesszuteilungsreihenfolge)</u>

BS bietet Mehrprozessbetrieb auf Basis der Serialisierung von Programmfäden:

- **bereit** zur Ausführung durch den Prozessor (CPU).
  - der Prozess ist auf der Bereitliste
  - das Einplanungsverfahren bestimmt die Listenposition
- **laufend**, erfolgte Zuteilung des BM "CPU"
  - der Prozess vollzieht seinen CPU-Stoß
  - zu einem Zeitpunkt pro CPU nur ein laufender Prozess
- **blockiert** auf ein bestimmtes Ereignis
  - der Prozess erwartet die Zuteilung eines BMs (mit Ausnahme des BMs "CPU")
  - ggf. vollzieht der Prozess auch seinen E/A-Stoß

Spezialfall: blockiert -> laufend ~> voll verdrängend

<u>mittelfristige Einplanung (Festlegung der Umlagerungsreihenfolge)</u>

BS implementiert die Umlagerung (swapping) von kompletten Programmen bzw. logischen Adressräumen:

- schwebend bereit
  - Adressraum des Prozesses ist ausgelagert (verschoben in den Hintergrundspeicher; "swap-out" ist erfolgt; "swap-in" wird erwartet)
  - die Einlastung des Prozesses ist außer Kraft (genauer: aller Fäden d. Adressraums)
- schwebend blockiert
  - ausgelagerter ereigniswartender Prozess
  - Ereigniseintritt -> schwebend bereit

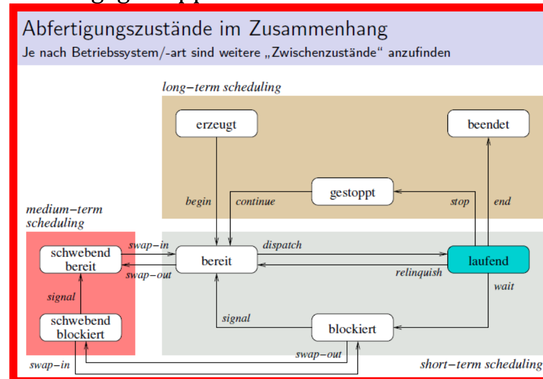
Variante: schwebend bereit -> bereit ~> langfristige Einplanung

<u>langfristige Einplanung (Festlegung der Zulassungsreihenfolge)</u>

BS verfügt über Funktionen zur Lastkontrolle und steuert den Grad an Mehrprogrammbetrieb:

- erzeugt und fertig zur Programmverarbeitung
    - Prozess ist "inkarniert", Programm wurde zugeordnet
    - ggf. steht die Speicherzuteilung noch aus
- gestoppt und erwartet seine Fortsetzung
    - Prozess wurde angehalten (z.B. ^Z bzw. kill(2))
    - Gründe: Überlastung, Verklemmungsvermeidung...
- beendet und erwartet seine Entsorgung
    - Prozess ist terminiert, BM-freigabe erfolgt
    - ggf. muss ein andere Prozess den "Kehraus" vollenden

Achtung: gestoppt werden können auch bereite/blockierte Prozesse



Verdrängung

Einplanung bzw Umplanung:

- nachdem ein Prozess erzeugt worden ist: *begin*
- wenn ein Prozess freiwillig die CPU abgibt: *relinquish*
- falls das von einem Prozess erwartete Ereignis eingetreten ist: *signal*
- sobal ein Prozess wieder aufgenommen werden kann: *continue*

Pozesse können dazu gedrängt werden, die CPU freiwillig abzugeben (sofern **verdrängende Prozesseinplanung** erfolgt)

Verdrängende Prozesseinplanung (Ereigniseintritt -> Einplanung -> Einlastung)
Verdrängung des laufenden Prozesses von der CPU bedeutet folgendes:

1. Ereignis tritt ein, dessen Behandlungsverlauf zum Planer führt
2. der laufende Prozess wird eingeplant
3. der einzulastende Prozess wird ausgewählt und (wieder) aufgenommen

Einplanung und Einlastung von Prozessen erfolgt nicht immer zeitnah zum Ereigniseintritt

- Verdrängung eines Prozesses verzögert sich ggf. unbestimmt lang
- Ursache dafür ist u.a. die Architektur von Betriebssystem(kern)en

ggf. entstehende Latenzzeiten können Anwendungen beeinträchtigen
Latenzzeiten und Determinismus

Einplanungslatenz ist unvermeidbar, nicht jedoch ihre Unbestimmtheit – deterministische Einplanung:

- zu jedem Zeitpunkt ist der nachfolgende Schritt eindeutig festgelegt
- die Latenzzeit ist konstant oder mit fester oberer Schranke variabel

Einlastungslatenz, die Zeitspanne zw. Einplanung und Verdrängung – führt zur weiteren Differenzierung:

- *verdrängend* ∼ "programmierte Verdrängung"
    - Einlastung nur an bestimmten Stellen freigegeben; **Verdrängungspunkte**
- *voll verdrängend* ≡ Einlastung jederzeit erlaubt

Unterbrechungslatenz, die Zeitspanne zw. Signalisierung und Behandlungsbeginn einer Programmunterbrechung

3. Gütemerkmale

Dimensionen der Prozesseinplanung
benutzerorientierte Kriterien fokussieren auf **Benutzerdienlichkeit**
systemorientierte Kriterien haben **Systemperformanz** im Vordergrund

- ➔ gute Systemperformanz ist auch der Benutzerdienlichkeit förderlich

Benutzerorientierte Kriterien

- Antwortzeit: Minimierung der Zeitdauer von der Auslösung eines Systemaufrufs bis zur Entgegenahme der Rückantwort, bei gleichzeitiger Maximierung der Anzahl interaktiver Prozesse
- Durchlaufzeit: Minimierung der Zeitdauer vom Start eines Prozesses bis zu seiner Beendigung, d.h. der effektiven Prozesslaufzeit und aller anfallenden Prozesswartezeiten
- Termineinhaltung: Starten und/oder Beendigung eines Prozesses (bis) zu einem fest vorgegebenen Zeitpunkt
- Vorhersagbarkeit: Deterministische Ausführung des Prozesses unabh. von der jeweils vorliegenden Systemlast

Systemorientierte Kriterien

- Durchsatz: Maximierung der Anzahl vollendeter Prozesse pro vorgegebener Zeiteinheit, d.h., der (im System) geleisteten Arbeit
- Prozessauslastung: Maximierung des Prozentanteils der Zeit, während der die CPU Prozesse ausführt, d.h., "sinnvoll" Arbeit leistet
- Gerechtigkeit: Gleichbehandlung der auszuführenden Prozesse und Zusicherung, den Prozessen innerhalb gewisser Zeiträume die CPU zuzuteilen
- Dringlichkeit: Vorzugbehandlung des Prozesses mit der höchsten (statischen/dynamischen) Priorität
- Lastausgleich: Gleichmäßige BM-auslastung; ggf. auch Vorzugsbehandlung der Prozesse, die stark belastete BM eher selten belegen

Betriebsart vw. Einplanungskriterien

Prozesseinplanung impliziert eine bestimmte Betriebsart und umgekehrt

- allgemein: Gerechtigkeit, Lastausgleich
  - Durchsetzung der jeweiligen Strategie
- Stapelbetrieb: Durchsatz, Durchlaufzeit, Prozessauslastung
- Dialogbetrieb: Antwortzeit

- Echtzeitbetrieb: Dringlichkeit, Termineinhaltung, Vorhersagbarkeit
  - oft Konflikt mit Gerechtigkeit/Lastausgleich

4. Zusammenfassung

- **Einplanungseinheit** für die Prozessorvergabe ist der Faden
  - seine Lauf- und Wartephasen betreiben einen Rechner stoßartig
  - Fäden sind Mittel zum Kaschieren von Totzeiten anderer Fäden
- Betriebssysteme treffen **Zuteilungsentscheidungen** auf drei Ebenen:

long-term scheduling	Lastkontrolle des Systems
medium-term scheduling	Umlagerung von Programmen
short-term scheduling	Einlastungsreihenfolge von Prozessen/Fäden

- die **Entscheidungskriterien** haben verschiedene Dimensionen:

Benutzer	Antwort-/Durchlaufzeit, Termine, Vorhersagbarkeit
System	Durchsatz, Auslastung, Gerechtigkeit, Dringlichkeit, Lastausgleich

- Durchsetzung der Kriterien impliziert eine bestimmte **Betriebsart**
  - umgekehrt: Betriebsarten erwarten die Durchsetzung gewisser Kriterien

IX.2 Prozessverwaltung: Einplanungsverfahren

1. Einordnung

Klassifikation

Kooperativ vs. Präemptiv

cooperative scheduling (voneinander abhängige Prozesse)

- "unkooperative" Prozesse können die CPU *monopolisieren*
- während der Programmausführung müssen Systemaufrufe erfolgen (*Endlosschleifen ohne Systemaufrufe*im Anwendungsprogramm verhindern Prozesse anderer Anwendungsprogramme)
- alle Systemaufrufe müssen den Scheduler durchlaufen

preemtive scheduling (voneinander unabhängige Prozesse)

- Prozessen wird die CPU entzogen, zugunsten andere Prozesse

- der laufende Prozess wird ereignisbedingt von der CPU verdrängt (Endlosschleifen beeinträchtigen andere Prozesse nicht (bzw. kaum))
- die Ereignisbehandlung aktiviert (direkt/indirekt(den Scheduler
- Monopolisierung der CPU ist nicht möglich: **CPU-Schutz**

Deterministisch vs. Probabilistisch
deterministic scheduling (bekannter, exakt vorberechneter Prozesse)
- alle CPU-Stoßlängen und ggf. auch **Termine** sind bekannt
- die genaue Vorhersage der CPU-Auslastung ist möglich
- das System stellt die Einhaltung von Zeitgarantien sicher
- die **Zeitgarantien** gelten unabh. von der jeweiligen Systemlast

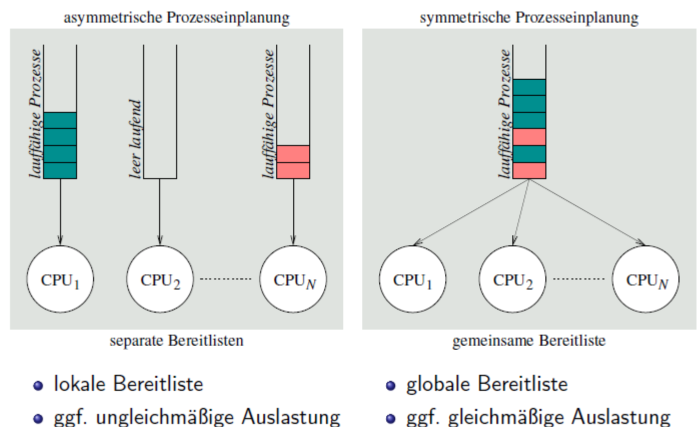
probabilistic scheduling (unbekannter Prozesse)
- exakte CPU-Stoßlängen sind unbekannt, ggf. auch Termine
- die CPU-Auslastung kann lediglich abgeschätzt werden
- das System kann Zeitgarantien weder geben noch einhalten
- Zeitgarantien sind durch die Anwendung sicherzustellen

Statisch vs. Dynamisch
offline scheduling (statisch, <u>vor</u> der Programmausführung)
- Komplexität verbietet Ablaufplanung im laufenen Betrieb
- Ergebnis der Vorberechnung ist ein vollständiger Ablaufplan
- die Verfahren sind zumeist beschränkt auf strikte Echtzeitsysteme

online scheduling (dynamisch, <u>während</u> der Programmausführung)
- Stapelsysteme, interaktive Systeme, verteilte Systeme
- schwache und feste Echtzeitsysteme

Asymmetrisch vs. Symmentrisch
assymetric scheduling (ist abhängig von Eigenschaften der Ebene$_{2/3}$)
- obligatorisch in einem assymentrischen Muliprozessorsystem (Prozesse sind an bestimmte Prozessoren gebunden)
- optional in einem symmetrischen Muliprozesorsystem (BS hat freie Hand über Prozessorvergabe)
- Prozesse in funktionaler Hinsicht ungleich verteilen (müssen)

symmetric scheduling (ist abhängig von Eigenschaften der Ebene$_2$)
- identische Prozessoren, alle geeignet zur Programmausführung
- Prozesse werden gleich auf die Prozessoren verteilt: Lastausgleich



2. Verfahrensweisen

Übersicht:
- kooperativ: FCFS (wer zuerst kommt, mahlt zuerst...) — gerecht
- verdrängend: RR, VRR (jeder gegen jeden...) — reihum
- probabilistisch: SPN(SJF), SRTF, HRRN (die Kleinen nach vorne...) — priorisierend
- mehrstufig: MLW, FB(MLFQ) (Rasterfahndung...)

Kooperativ

FCFS (first come, first served) Fair, einfach zu impl., ..., dennoch problematisch
- Prozesse nach ihrer Ankunftszeit einplanen und in dieser Reihenfolge verarbeiten
  - nicht-verdrängendes Verfahren, setzt kooperative Prozesse voraus
- Gerechtigkeit zu Lasten hoher Antwortzeit und niedrigem E/A-Durchsatz
  - suboptimal bei einem Mix von kurzen und langen CPU-Stößen

Prozesse mit { langen / kurzen } CPU-Stößen werden { begünstigt / benachteiligt } -

Problem: ***Konvoieffekt*** – kurze Prozesse bzw. CPU-Stöße folgen einem langen...

Verdrängend

RR (round robin) Verdrängendes FCFS, Zeitscheiben, CPU-Schutz

- Prozesse werden nach Ankunftszeit ein- und in regelmäßigen Zeitabständen (periodisch) umgeplant
  - verdrängendes Verfahren, nutzt periodische Unterbrechungen (Zeitgeber liefert asynchrone Programmunterbrechungen)
  - jeder Prozess erhält eine Zeitscheibe zugeteilt (obere Schranke für die CPU-Stoßlänge eines laufenden Prozesses)
- Verringerung der bei FCFS auftretenden Benachteiligung von Prozessen mit kurzen CPU-Stößen
  - die Zeitscheibenlänge bestimmt die Effektivität des Verfahrens (zu lang -> Degenerierung zu FCFS; zu kurz -> sehr hoher Mehraufwand)
  - Faustregel: etwas länger als die Dauer eines "typ. CPU-Stoßes"

RR-Konvoieffekt

Leistungsprobleme bei einem Mix von Prozessen)

- E/A-intensive Prozesse schöpfen ihre Zeitscheibe selten voll aus (sie beenden ihren CPU-Stoß freiwillig – vor Ablauf der Zeitscheibe)
- CPU-intensive Prozesse schöpfen ihre Zeitscheibe meist voll aus (sie beenden ihren CPU-Stoß unfreiwillig – durch Verdrängung)

Problem: CPU-Zeit ist zu Gunsten der CPU-intensiver Prozesse ungleich verteilt (E/A-intensive Prozesse werden schlechter bedient; E/A-Geräte sind schlecht ausgelastet; Varianz der Antwortzeit E/A-intensiver Prozesse ist groß)

VRR (virtual round robin) RR mit Vorzugswarteschlange und var. Zeitscheiben

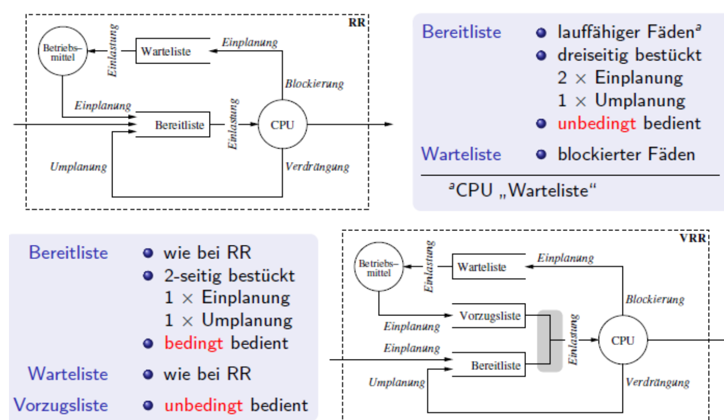
Prozesse werden mit Beendigung ihres E/A-Stoßes bevorzugt eingeplant, jedoch nicht (zwingend) bevorzugt/sofort eingelastet

- Einreihung in eine der Bereitliste vorgeschalteten Vorzugsliste
  - FIFO -> evtl. Benachteiligung hoch-interaktiver Prozesse; daher ...
  - aufsteigend sortiert nach dem Zeitscheibenrest eines Prozesses
- Umplanung bei Beendigung des jeweils laufenden CPU-Stoßes
  - die Prozesse auf der Vorzugsliste werden zuerst eingelastet;
  - sie bekommen die CPU für die Restdauer ihrer Zeitscheibe zugeteilt;
  - bei Ablauf dieser Zeitscheibe werden sie in die Bereitliste eingereiht

Vermeidung der bei RR möglichen Ungleichverteilung von CPU-Zeiten

- bevorzugt werden interaktive Prozesse mit kurzen CPU-Stößen
- erreicht durch struktullen Maßnahmen – nicht durch analytische

RR vs. VRR



SPN (shortest process next) Zeihreihen bilden, analysieren und verwerten

Prozesse werden nach ihrer *erwarteten Bedienzeit* eingeplant

- Grundlage dafür ist à priori Wissen über die *Prozesslaufzeiten*

Stapelbetrieb	Programmierer setzen Frist
Produktionsbetrieb	Erstellung einer Statistik durch Probeläufe
Dialogbetrieb	Abschätzung von CPU-Stoßlängen zur Laufzeit

- Abarbeitung einer aufsteigend nach Laufzeiten sortierten Bereitliste
  - Abschätzung erfolgt vor (statisch) oder zur (dynamisch) Laufzeit

Verkürzung von Antwortzeiten und Steigerung der Gesamtleistung des Systems auf Kosten länger laufender Prozesse

- ein *Verhungern* dieser Prozesse ist möglich

SRTF (shortest remaining time first) Verdrändendes SPN, Verhungerungsgefahr, Effektivität von VRR

Prozesse werden nach erwarteten Bedienzeit eingeplant und in unregel. Zeitabständen sporadisch umgeplant. Umplanung erfolgt ereignisbedingt und (ggf. voll) verdrängend. (z.B. bei Beendigung des E/A-Stoßes eines wartenden Prozesses. Allg: bei Aufhebung der Wartebedingung für einen Prozess)
Verdrängung führt zu besseren Antwort- und Durchlaufzeiten: ggü. VRR steht der Overhead zur CPU-Stoßlängenabschätzung

HRRN (highest response ration next) SRTF ohne Verhungern der Prozesse
Prozesse werden nach erwarteten Bedienzeit eingeplant und periodisch unter Berücksichtigung ihrer Wartezeit umgeplant

- in regemäßigen Zeitabständen wird ein Verhältniswert R berechnet: R = (w+s)/s -> w: aktuell abgelaufene Wartezeit eines Prozesses; s erwartete Bedienzeit
- periodische Aktualisierung aller Einträge in der Bereitliste
- ausgewählt wird der Prozess mit dem größten Verhältniswert R

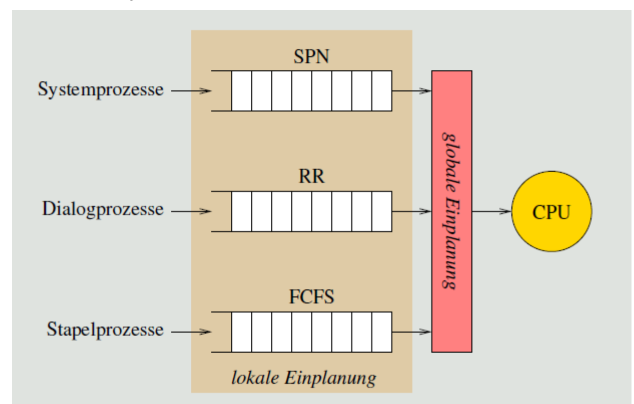
Alterung von Prozessen meint Anstieg der Wartezeit:
Alterung entgegenwirken beugt Verhungern vor!!!

Mehrstufig

MLW (multilevel queue) Unterstützt Mischbetrieb: Vorder- und Hintergrundbetrieb
Prozesse nach Typ einplanen

- Bereitliste in separate Listen aufteilen (zB. System-, Dialog- & Stapelprozesse)
- jede Liste eine lokale Einplanungsstrategie (SPN, RR und FCFS)
- zwischen den Listen globale Einplanungsstrategie definieren
  - statisch: Liste einer bestimmten Prioritätsebene fest zuordnen
  - dynamisch: die Listen im Zeitmultiplexverfahren wechseln

Prozessen Typen zuordnen ist eine statische Entscheidung (sie wird zum Zeitpunkt der Prozesserzeugung getroffen)



FB (feedback) Begünstigt kurze/interaktive Prozesse, ohne die relativen Stoßlängen kennen zu müssen

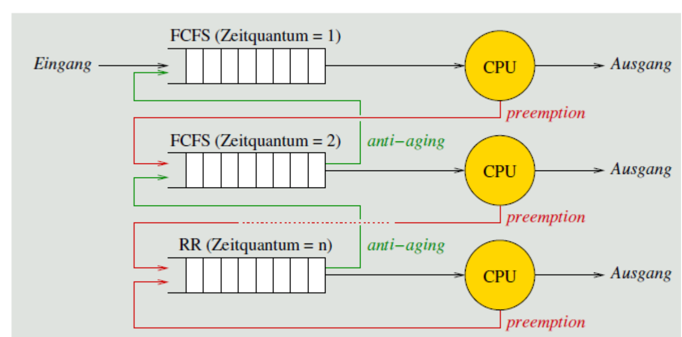
Prozesse werden nach ihrer Ankunftszeit ein- und in regelmäßigen Abständen umgeplant (periodisch)

- Hierarchie von Bereitlisten, je nach Anzahl der Prioritätsebenen
- je nach Ebene verschiedene Einreihungsstrategien und –parameter

Bestrafung:

- Prozesse mit langen CPU-Stößen fallen nach unten durch
- Prozesse mit kurzen CPU-Stößen laufen relativ schnell durch

FB: Bestrafung und Bewährung



multilevel feedback queue (MLFQ)

Priorisierend

Prioritäten setzende Verfahren
Statische Prioritäten (MLQ) vs. dyn. Prioritäten (VRR, SPN, SRTF, HRRN, FB)

Prozessvorrang (bevorzugte Einlastung v. Prozessen mit höherer Priorität)

- statisch: zum Zeitpunkt der **Prozesserzeugung** -> Laufzeitkonstante
  - wird im weiteren Verlauf nicht mehr veränderter zwingt eine deterministische Ordnung zw. Prozessen
- dyn. : zum Zeitpunkt der **Prozessausführung** -> Laufzeitvariable
  - Berechnung erfolgt durch das BS
  - erzwingt keine deterministische Ordnung zw. Prozessen

Echtzeitverarbeitung bedingt Prioritäten setzende Verfahren.
- jedoch nicht jedes solcher Verfahren eignet sich zum Echtzeitbetrieb
- Einplanung muss ein deterministisches Laufzeitverhalten liefern

Vergleich

Gegenüberstellung von Strategien und Verfahrensweisen
kooperativ/verdrängend vs. probabilistisch/deterministisch

	FCFS	RR	VRR	SPN	SRTF	HRRN	FB
kooperativ	✓			✓			
verdrängend		✓	✓		✓	✓	✓
probabilistisch				✓	✓	✓	
deterministisch	keine bzw. nicht von sich aus allein ⇝ EZS [5]						

MLQ umfasst Eigenschaften der in dem Verfahren vereinten Strategien
- Priorisierung von Strategien liefert Nuancen im Laufzeitverhalten
- speziellen Anwendungsanforderungen (teilweise) entgegenkommen:
  - z.B. FCFS priorisieren ⇝ „*number crunching*“ fördern

3. Zusammenfassung

- Prozesseinplanung unterliegt einer breit gefächerten Einordnung
  - kooperativ/verdrängend
  - deterministisch/probabilistisch
  - entkoppelt/gekoppelt
  - asymmetrisch/symmetrisch
- die entsprechenden Verfahrensweisen sind z.T. sehr unterschiedlich
  - FCFS: kooperativ
  - RR, VRR: verdrängend
  - SPN, SRTF, HRRN: probabilistisch
  - MLQ, FB (MLFQ): mehrstufig
- Prioritäten setzende Verfahren legen einen Prozessvorrang fest
  - dies betrifft die behandelten probabilistischen, mehrstufigen Verfahren
    - die allesamt nichtdeterministisch und damit nicht echtzeitfähig sind
  - echtzeitfähige Prozesseinplanung ist vor allem deterministisch
- die Fallstudien (s. Anhang) planen Prozesse probabilistisch ein...

IX.3 Prozessverwaltung: Einlastung

1. Vorwort

Programmfaden:
Einlastung zeitnah zur Ablaufplanung von Threads. ist ihr nachgeschaltet, ein Abfertiger führt die eingeplanten Fäden der CPU zur Verarbeitung zu. dazu nimmt er Aufträge vom Planer entgegen.
Umschalten der CPU bedeutet, zw. zwei Aktivitätsträgern desselben oder verschiedener Programme zu wechseln.

a. CPU-Stoß endet: der laufende Thread wird weggeschaltet
b. CPU-Stoß beginnt: laufbereiter Thread wird zugeschaltet

2. Koroutine

Konzept

- gleichberechtigtes Unterprogramm
- Ausführung beginnt an der letzten "Unterbrechungsstelle" (kooperativ)
- zw. aufeinanderfolgenden Ausführungen ist ihr Zustand invariant
- "zustandsbehaftete Prozedur"

Aktivierungskontext bleibt während Phasen der Inaktivität erhalten:
deaktivieren -> Koroutinenkontext "einfrieren" (sichern)
aktivieren -> Koroutinenkontext "auftauen" (wieder herstellen)

Routine vs. Koroutine

Merkmal	Routine	Koroutine
Aktivierung	mehrmals	einmal: initial
Unterbrechung	niemals	mehrmals
Fortsetzung	niemals	mehrmals
Beendigung	einmal: je Aktivierung	niemals

„Unsterblichkeit“
- da Koroutinen eine Aufrufhierarchie fehlt, können sie sich auch nicht von selbst beendigen
  - Routinen werden durch Unterprogrammaufrufe aktiviert, Rücksprung aus dem Unterprogramm beendigt somit auch ihre Ausführung
  - Koroutinen werden nicht durch Unterprogrammaufrufe aktiviert
- um sich zu beendigen müssen sie einer anderen Koroutine dazu einen diesbezüglichen Auftrag erteilen

Routine hat keinen Kontrollflusswechsel bei Aktivierung/Deaktivierung (Koroutine schon)

Spezialisierung/Generalisierung
- Routine spezifischer als Koroutine und diese generischer als Routine

Routinen können durch Koroutinen implementiert werden

Implementierung
Elementaroperationen: resume, invoke, launch,

3. Programmfaden

Aktivitätsträger
Koroutinen "mechanisieren" Programmfäden
Mehrprogrammbetrieb basiert auf Koroutinen des BS
- pro auszuführendes Programm gibt es (wenigstens) eine Koroutine
- ist eine Koroutine aktiv, so ist das ihr zugeordnete Programm aktiv
- ein anderes Programm ausführen -> Koroutine wechseln

Koroutinen sind (autonome) Aktivitätsträger des BS
- ihr Aktivierungskontext ist globale Variable des BS
- für jede Prozessinkarnation gibt es eine solche BS-variable

ein BS ist Inbegriff für das nicht-sequentielle Programm

Fadenarten
Fäden der Kernebene – Klassische Variante von Mehrprozessbetrieb
Prozessinkarnationen als $Ebene_3$-Konzept basieren auf Kernfäden
- egal. ob die Maschinenprogramme ein- oder mehrfädig ausgelegt sind
  - jeder Anwendungsfaden ist Kernfaden (umgekehrt nicht)
- Kernfäden ≠ Prozessinkarnation der $Ebene_3$
  - Maschinenprogramme verwenden Fäden
  - im Programmiermodell des BS manifestiert

Einplanung und Einlastung der Anwendungsprozesse sind Funktionen des BS-(Kern)s
- Erzeugung, Koordination, Zerstörung von Fäden -> Systemaufrufe

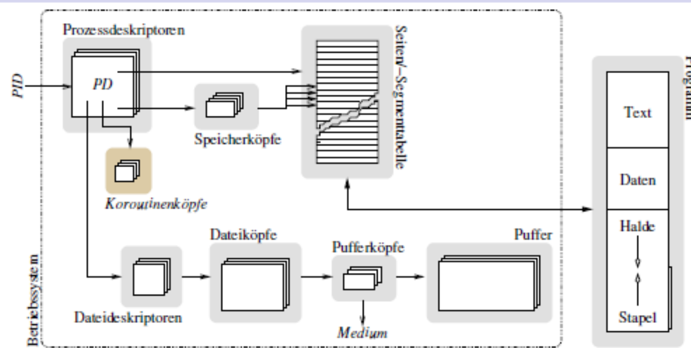
Fäden der Benutzerebene – Ergänzung oder Alternative zu Kernfäden
Prozessinkarnationen als $Ebene_{2/3}$-Konzept sind Benutzerfäden
- virtuelle Prozessoren bewirken die Ausführung der Maschinenprogramme
  - Benutzerfäden = Koroutinen -> $Ebene_2$
  - 1 Kernfaden f. ≥ 2 Benutzerfäden -> $Ebene_3$
- der Kern stellt ggf. Planeransteuerungen bereit
- Fäden realisiert durch $Ebene_{2/3}$-Programme

Einplanung und Einlastung der (federgewichtigen) Anwendungsprozesse sind keine Funktionen des BS(kern)s.
- Erzeugung, Koordination, Zerstörung von Fäden -> Prozeduraufrufe

Prozessdeskriptor
Prozesskontrollblock – Datenstruktur zur Verwaltung von Prozessinstanzen
Kopf eines Datenstrukturgeflechts zur Beschreibung und Verwaltung einer Prozessinkarnation und Steuerung eines Prozesses
- Prozessdeskriptor
- ein abstrakter Datentyp des Betriebssystem(kern)s

Softwarebetriebsmittel zur Verwaltung von Programmausführungen
- Kernfaden Variable des BS
- Benutzerfaden Variable des Anwendungsprogramms

Generische Datenstruktur
Logische Sicht eines Geflechts abstrakter Datentypen



einfädiger Prozess $\mapsto$ 1 Koroutinenkopf
mehrfädiger Prozess $\mapsto$ $N > 1$ Koroutinenköpfe

Prozesszeiger

Instanzvariable des Typs „Prozessdeskriptor"
Buchführung über den aktuell laufenden Prozess
Zeiger auf den Kontrollblock des laufenden Prozess: **Prozesszeiger**

- für jeden Prozessor(kern) ist solch ein Zeiger bereitzustellen
- innerhalb des BS ist somit jederzeit bekannt, welcher Prozess, Faden, welche Koroutine die CPU „besitzt"
- wichtige Funktion der Einlastung ist es, den Zeiger zu aktualisieren

Laufgefahr: Verdrängend arbeitende Prozesseinplanung

- Einlastung: Prozesszeiger- & Fadenumschaltung -> Verdrängung des dazw. laufenden Fadens ist kritisch (KA). Abh. von der Betriebsart zu schützen

4. Zusammenfassung und Anhang

- **Koroutinen** konkretisieren Prozesse, realisieren Prozessinstanzen
  - die Gewichtsklasse von Prozessen spielt eine untergeordnete Rolle
    - feder-, leicht-, schwergewichtige Prozesse basieren auf Koroutinen
  - ihr Aktivierungskontext überdauert Phasen der Inaktivität
    - gesichert („eingefroren") im jeder Koroutine eigenen Stapelspeicher
- **Programmfäden** (engl. *threads*) sind durch Koroutinen repräsentiert
  - unterschieden in zwei Fadenarten, je nach Ebene der Abstraktion:
    - Kernfaden implementiert durch Ebene$_2$-Programme
    - Benutzerfaden implementiert durch Ebene$_{2/3}$-Programme
  - Einlastung eines Fadens führt einen Koroutinenwechsel nach sich
- der **Prozessdeskriptor** ist Objekt der Buchführung über Prozesse
  - Datenstruktur zur Verwaltung von Prozess- und Prozessorzuständen
    - insbesondere des Aktivierungskontextes der Koroutine eines Prozesses
  - Softwarebetriebsmittel zur Beschreibung einer Programmausführung
- jeder Prozessor(kern) verfügt über einen eigenen **Prozesszeiger**

X.1 Nichtsequentialität

1. Konkurrenz und Koordination

Koordinierung

Serialisierung gleichzeitiger Aktivitäten

- blockierend: pessimistische Annahme der Überlappung
- nichtblockierend: optimistische Annahme keiner Überlappung

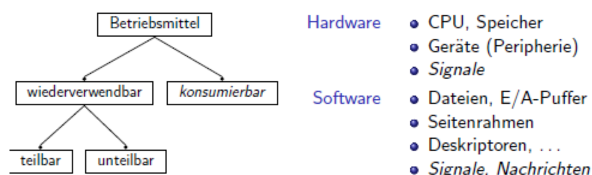
Synchronisation bezeichnet allg. das Herstellen von Gleichzeitigkeit

a) Koordination der Kooperation und Konkurrenz zw. Prozessen
b) Abgleich von Echtzeituhren (oder Daten) in verteilten Systemen
c) Sequentialisierung von Ereignissen entlang einer Kausalordnung

Herangehensweisen
analytisch (Prozesseinplanung: implizite Syn.)
konstruktiv (Programmiertech.: explizite Syn.)

Konkurrenz

Betriebsmittel und BMarten



Wettbewerb um BM
und Art der BM bedingen problemspez. Verfahren:

- einseitige Syn: konsumierbare BM
- mehrseitige Syn: wiederverwendbare BM

Unteilbarkeit
Umstand, der die Verteilung der BM auf mehrere Prozessoren oder Prozesse verhindert.

- unteilbar ist ein BM, wenn es zu einem Zeitpunkt von nur genau einem Prozessor/Prozess genutzt werden darf (Zugriffsoperationen darauf können/dürfen nicht zeitlich zerteilt werden und sie müssen atomar, d.h. als Elementaroperation ausgeführt werden)
- teilbar ist ein BM, wenn mehrere Prozessoren/Prozesse es gleichzeitig benutzen dürfen (es dem einen entzogen und einem anderen gegeben werden darf)

Konfliktfall bei der Benutzung von BM
Prozesse gegenseitig im Konflikt, wenn

- begrenzte # BM vorrätig
- diese BM unteilbar und von derselben Art sind
- Zugriff darauf gleichzeitig: gleichzeitige Prozesse

Prozesse sind im Wettstreit um BM, wenn einer das BM anfordert, das ein anderer besitzt

- der anfordernde Prozess blockiert und wartet auf Freigabe
- der das BM belegende Prozess löst den auf die Freigabe des BM wartenden Prozess aus, deblockiert ihn wieder

Wechselseitiger Ausschluss

- n-fach mehrseitige, für n-1 blockierend wirkende Synchronisation
- eine *Option* zum Schutz eines krit. Abschnitts

Wechselseitiger Ausschluss – Serialisierung von Prozessen mit begrenzen/unteilbaren BM
Sequenzialisierung der Zugriffe gleichzeitiger Prozesse per Protokoll:
Vergabe -> das BM sperren und dem Prozess zuteilen (gesperrtes BM erneut belegen -> anfordernde Prozess wird blockiert. blockierende Prozess erwartet Ereignis zur Freigabe des gesperrten BM -> ihm wird die CPU entzogen)
Freigabe -> das BM dem Prozess wieder entziehen

- sollten Prozesse die Freigabe dieses Betriebsmittels erwarten, wird es sofort der **Wiedervergabe** zugeführt; das bedeutet:
  (a) das Betriebsmittel entsperren und alle Prozesse deblockieren, die sich dann wiederholt um die Vergabe zu bemühen haben *oder*
  (b) einen Prozess auswählen und ihm das Betriebsmittel zuteilen
- nur der das Betriebsmittel „besitzende" Prozess kann es freigeben

Serialisierung gleichzeitiger Prozesse: Koordinierung
Synchronisation ≡ Koordination der Kooperation und Konkurrenz zw. Prozessen

- sich überlappen könnende Aktivitäten der Reihe nach ausführen (gleichzeitige Prozesse im krit. Abschnitt koordinieren)
- „der Reihe nach" -> Verzögerung gleichzeitiger Prozesse erzwingen
    - potentiell überlappende Prozesse verzögern, der Konflikt ist ungewiss
    - den überlappten Prozess verzögern, der Konflikt ist gewiss

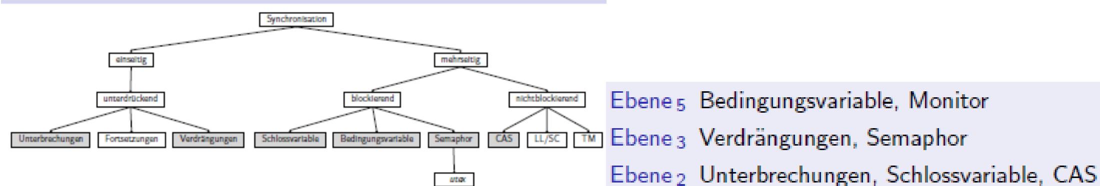
Synchronisationsverfahren

- wirken einseitig oder mehrseitig
    - unterdrückend, blockierend, nichtblockierend
        - behinderungs-, sperr-, wartefrei

2. Verfahrensweisen

Einordnung

Arten der Synchronisation



Einseitige Synchronisation – Uniliteral
Auswirkungen nur auf einen der beteiligten Prozesse
<u>Bedingungssynchronisation</u>

- der Ablauf des einen Prozesses ist abh. von einer Bedingung
- der andere Prozess erfährt keine Verzögerung in seinem Ablauf

<u>log. Synchronisation</u>

- die Maßnahme resultiert aus der log. Abfolge der Aktivität
- vorgegeben durch das „Rollenspiel" der beteiligten Prozesse

Beachte!
Andere Prozesse sind jedoch nicht gänzlich unbeteiligt (Veränderung einer Bedingung, auf die ein Prozess wartet, ist z.B. von einem anderen Prozess herbeizuführen)

Mehrseitige Synchronisation – Multiliteral
Auswirkungen haben die Verfahren ggf. auf alle beteiligten Prozesse:

- welche Prozesse verzögert werden, ist i.A. unvorhersehbar

Beispiel: Wechselseitiger Ausschluss

- erzwungene <u>seq. Ausführung</u> von Anweisungsfolgen
- im Regelfall zeitl. begrenzte, exkl. Betriebsmittelvergabe

Beispiel: krit. Abschnitt

- wettlaufintolerant (wechselseitiger Ausschluss, Verzögerung ggf. folg. gleichzeitiger Prozesse)
- wettlauftolerant (nichtseq. Ausführung, Verzögerung überlappter gl. Prozesse)

Betriebsmittelart und Synchronisationstechnik – Artenspez. Verfahren
<u>Notwendigkeit des Ausschlusses</u> gleichzeitiger Prozesse ist immer nu in folg. Fällen gegeben -> blockierende Synchronisation:

- beim Zugriff auf unteilbare wiederverwendbare BM
- bei Inanspruchnahme eines konsumierbaren BMs

<u>Schutz eines KA</u> bedingt sich in anderer Weise und ist auf zwei versch. Wege mögl.:

- pessimisitisch -> wechselseitiger Ausschluss (block. Syn.)
    - KA implementiert als seq. Programm
    - entsprechend der „herkömml. Sichtweise"
- optimisitsch -> nichtblockierende Syn.
    - KA implementiert als nichtseq. Programm
    - entsprechend der „alternativen Sichtweise"

Kritischer Abschnitt

Kritischer Abschnitt (KA) – Kennzeichnend für mehrseitige Synchronisation

KA

- gegenseitig ausschließende Aktivitäten (werden nie parallel ausgeführt; verhalten sich zueinander wie unteilbar)
- Anweisungen, deren Ausführung wechselseitigen Ausschluss erfordert

Eine Frage der Abstraktionsebene -> alternative Sichtweise

- der wechselseitige Ausschluss (von gleichzeitigen Prozessen) ist nicht zwingend, um einen KA zu schützen
- vielmehr gilt es sicherzustellen, dass die Ausführung eines solchen Abschnitts jederzeit ein *konsistentes Ergebnis* liefert

Atomarität KA

Beachte: Ein Programmabschnitt ist kritisch, wenn er bei Ausführung durch einen Prozessor vor dem Hintergrund gleichzeitiger Prozesse wenigstens eine Wettlaufsituation zeigt.

unteilbare KA: ist blockierend syn., wird von gl. Prozessen immer seq. durchlaufen
teilbare KA: ist nichtblockierend syn., wird von gl. Prozessen nichtseq. durchlaufen

KA durch prozedurale Abstraktion „verbergen"

- KA logisch als Elementaroperation (ELOP) auslegen -> Prozedur
- den ausgeklammerten Programmabschnitt problemgerecht schützen

Verhaltensregeln zum Durchlaufen KA

Betreten und Verlatten von KS steuern problemspezifische Protokolle

Eintrittsprotokoll

- regelt Belegung des KA durch einen Prozess (erteilt Prozess Zugangsberechtigung)
- bei bereits belegten Abschnitt: Wettstreitigkeit
  - blockierend: den eintreffenden Prozess am Eintritt hindern
  - nichtblockierend: den Abschnitt belegenden Prozess überlappen dürfen

Austrittsprotokoll

- regelt die Freigabe des KA durch einen Prozess
  - blockierend: ggf. am Eintritt gehinderte(n) Prozess(e) freigeben
  - nichtblockierend: den überlappten Prozess zurücksetzen/wiederholen
- Prozesse können den KA (wieder) belegen

die Vorgehensweise variiert mit dem jew. Synchronisationsverfahrne

Bedingter kritischer Abschnitt

Betreten des KA ist von einer Wartebedingung abh., die nicht erfüllt sein darf, um den Prozess fortzusetzen

- die Bedingung ist als Prädikat über die im KA enthaltenen bzw verwendeten Daten definiert
- typischerweise technisch realisiert durch eine Bedingungsvariable

Auswertung der Wartebedingung muss im Ka erfolgen

- bei Nichterfüllung der Bedingung wird der Prozess auf Eintritt eines zur Wartebedingung korrespondierenden Ereignisses blockiert
  - damit das Ereignis später signalisiert werden kann, muss der KA beim Schlafenlegen jedoch freigegeben werden
- bei (genauer: nach) Erfüllung/Signalisierung der Bedingung versucht der Prozess den KA wieder zu belegen
  - ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Lebendigkeit

Fortschrittsgarantien – Aussagen zur Lebendigkeit nichtseq. Programme

behinderungsfrei

- ein einzelner, in Isolation ablaufender Faden wird seine Operation in begrenzter Anzahl von Schritten beenden
- ein Faden läuft in Isolation ab, wenn alle anderen Fäden, die ihn behindern könnten, in der Ausführung zurückgestellt sind

sperrfrei

- jeder bei Ablauf eines Fadens auszuführende Schritt trägt dazu bei, dass das nichtseq. Programm insgesamt voranschreitet

- garantiert systemweiten Durchsatz, erlaubt jedoch Aushungerung eines einzelnen Fadens

wartefrei

- die Anzahl der zur Beendigung einer Operation bei Fadenabläufen auszuführenden Schritte ist begrenzt
- garantiert systemweiten Durchsatz und ist frei von Aushungerung
-

Zusammenfassung

Dualität von Koordinierungstechniken
Theorie vs. Praxis

Problem:	Methode:
Erhöhung von Parallelität	nichtblockierende Synchronisation
wechselseitiger Ausschluss	Schlossvariable
explizite Prozesssteuerung	Bedingungsvariable
bedingte Verzögerung	bedingter kritischer Abschnitt
Austausch von Zeitsignalen	Semaphor
Austausch von Daten	Nachrichtenpuffer

logisch betrachtet sind alle Methoden äquivalent, da jede von ihnen hilft, ein beliebiges Steuerungsproblem zu lösen

praktisch betrachtet sind die Methoden nicht äquivalent, da einige von ihnen für ein gegebenes Problem zu komplexen und ineffizienten Lösungen führen

- **Nebenläufigkeit** setzt voneinander unabhängige Prozesse voraus
  - bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen
  - schränkt sich ein aus Gründen von Daten- oder Zeitabhängigkeit
- gleichzeitige abhängige Prozesse implizieren **Koordinierung**
  - nämlich der Kooperation und Konkurrenz zwischen Prozessen
  - durch analytische (implizite) oder konstruktive (explizite) Techniken
- **Synchronisation** zeigt einen großen Facettenreichtum
  - klassifiziert nach der jeweiligen Auswirkung auf beteiligte Prozesse:
    - einseitig oder mehrseitig
    - unterdrückend, blockierend oder nicht-blockierend
    - behinderungs-, sperr- oder wartefrei
  - verschiedenen Abstraktionsebenen eines Rechensystems zugeordnet:
    - Hochsprachenebene: Bedingungsvariable, Monitor
    - Maschinenprogrammebene: Verdrängungssteuerung, Semaphor
    - Befehlssatzebene: Schlossvariable, Spezialbefehle (CPU)
- Aussagen zur „Lebendigkeit" nichtsequentieller Programme leiten sich aus den **Fortschrittsgarantien** der Synchronisationsverfahren ab

X.2 Befehlssatzebene

1. Monitor

Eigenschaften

Synchronisierter abstrakter Datentyp: Monitor

Datentyp mit impliziten Synchronisationseigenschaften:

- *mehrseitige* Synchronisation an der Monitorschnittstelle
  - wechselseitiger Ausschluss der Ausführung exportierter Prozeduren
  - realisiert mittels **Schlossvariablen** oder vorzugsweise Semaphoren
- *einseitige* Synchronisation innerhalb des Monitors
  - bei Bedarf, Bedingungssyn. abh. Prozesse
  - realisiert durch eine **Bedingungsvariable** und ihre Operationen
    wait – blockiert einen Prozess auf das Eintreten eines Signals/einer Bedingung und gibt den Monitor implizit frei
    signal – zeigt das Eintreten eines Signals/einer Bedingung an und deblockiert (genau einen oder alle) darauf blockierte Prozesse

Monitor ≡ (eine auf ein Modul bezogene) Klasse

Kapselung

- von mehreren Prozessen gemeinsam bearbeitete Daten müssen. analog zu Modulen, in Monitoren organisiert vorliegen
- als Konsequenz macht die Programmstruktur krit. Abschnitte explizit sichtbar

Datenabstraktion

- wie ein Modul, so kapselt auch ein Monitor für mehrere Funktionen Wissen über gemeinsame Daten
- Auswirkungen lokaler Programmänderungen bleiben begrenzt

Bauplan (blueprint)

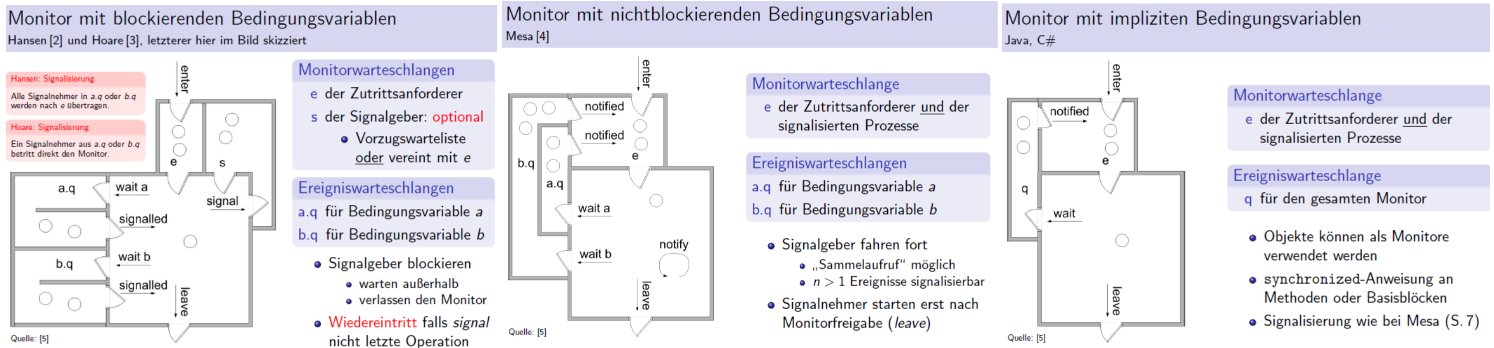
- wie eine Klasse, so beschreibt ein Monitor für mehrere Exemplare seines Typs den Zustand und das Verhalten
- er ist eine gemeinsam benutzte Klasse (shared class)

Klassenkonzept erweitert um Synchronisationssemantik (Monitor ≡ implizit syn. Klasse)

Monitorprozeduren

- schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse in ihrer Ausführung gegenseitig aus (erfolgreiche Prozeduraufruf sperrt Monitor; Prozedurrückkehr entsperrt Monitor)
- repräsentiert per Def. krit. Abschnitte, deren Integrität vom Kompilierer garantiert wird

- "Klammerung" krit. Abschnitte erfolgt automatisch
- Kompilierer erzeugt dafür notwendige Steueranweisungen



Architektur

2. Bedingungsvariable

Operationen

Signalisierung einer Fortführungsbedingung erwarten: wait
Wartebedingung festlegen

Monitorfreigabe als notwendiger Seiteneffekt beim Warten:
- andere Prozesse wären sonst am Monitoreintritt gehindert
- als Folge könnten die zu erfüllende Bedingung nie erfüllt werden
- schlafende Prozesse würde nie mehr erwachen -> Verklemmung

Monitordaten sind in einem konsistenten Zustand zu hinterlassen
- andere Prozesse betreten den Monitor während der Blockadephase
- als Folg sind (je nach Funktion) Zustandsänderungen zu erwarten
- vor Eintritt in die Wartephase muss der Datenzustand konsistent sein

Signalisierung einer Fortführungsbedingung: signal
Wartebedingung aufheben

Prozessblockaden in Bezug auf eine Wartebedingung werden aufgehoben
- warten Prozesse, müssen folgende Anforderungen erfüllt sein:
  - mind. ein an der Bedingungsvar. wartender Prozess wird deblockiert
  - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
- erwartet kein Prozess ein Signal, ist die Operation wirkungslos
  - d.h. Signale dürfen in Bedingungsvariablen nicht gespeichert werden

Lösungsansätze hierzu sind z.T. von sehr unterschiedlicher Semantik
- das betrifft etwa die Anzahl der deblockierten Prozesse:
  - alle auf die Bedingung wartenden oder genau nur einer
- große Unterschiede auch bzgl. **Besitzerwechsel** bzw. **Besitzwahrung**
  - "falsche Signalisierungen" werden toleriert oder nicht

Signalisierung

Besitzerwechsel: *signal and (urgent) wait*
Signalisierender Prozess gibt die Kontrolle über den Monitor ab, wird inaktiv

<u>alle</u> das Ereignis erwartenden Prozesse befreien
- alle Prozesse aus der Ereignis- in die Monitorwarteschlange bewegen
- bei Freigabe alle n Prozesse "bereit" setzen
- n-1 Prozesse reihen sich erneut in die Monitorwarteschlange ein

<u>höchstens einen</u> das Ereignis erwartenden Prozess befreien
- nur einen Prozess der Ereigniswarteschlange entnehmen

- den signalisierenden Prozess in die Monitorwarteschlange eintragen
- direkt vom signalisierenden zum signalisierten Prozess wechseln

Hoare: Neuauswertung der Wartebedingung entfällt
- Fortführungsbedingung des signalisierten Prozesses ist garantiert
    - seit der Signalisierung war kein anderer Prozess im Monitor
    - kein anderer Prozess konnte die Fortführugnsbed. entkräften
- der signalisierende Prozess bewirbt sich ggf. erneut um Monitorzutritt
    - "falsche Signalisierungen" werden nicht toleriert

Besitzwahrung: *signal and continue*
Signalisierender Prozess behält die Kontrolle über den Monitor, bleibt aktiv

einen oder alle das Ereignis erwartende Prozesse befreien
- Prozess(e) aus der Ereignis- in die Monitorwarteschlange bewegen
- bei Freigabe n ≥ 1 Prozesse "bereit" setzen

Mesa/Hoare: Gefahr von Prioritätsverletzung [4]
- bedingt durch die Auswahlentscheidung, die festlegt, welcher Prozess „bereit" wird bzw. der Ereigniswarteschlange entnommen werden soll
- Interferenz mit der Prozesseinplanung ist vorzubeugen/zu vermeiden

Mesa/Hansen: Neuauswertung der Wartebedingung erforderlich
- Fortführungsbedingung des signalisierten Prozesses nicht garantiert
    - ein anderer Prozess kann den Monitor zwischenzeitlich betreten haben
- signalisierte Prozesse bewerben sich erneut um den Monitorzutritt
    - „falsche Signalisierungen" (an den falschen Prozess) werden toleriert

3. Zusammenfassung

Monitorkonzepte im Vergleich
Hansen/Mesa vs. Hoare

Hansen und Mesa	Hoare
`while (free == NDATA) full.await();` `while (!free) null.await();`	`if (free == NDATA) full.await();` `if (!free) null.await();`
Prozessen wird nicht garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt	Prozessen wird garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt
• andere Prozesse können den Monitor betreten haben • Wartebedingung erneut prüfen • evtl. falsche Signalisierungen werden toleriert	• kein anderer Prozess konnte den Monitor betreten haben • Wartebedingung einmal prüfen • evtl. falsche Signalisierungen werden nicht toleriert

- ein Monitor ist ein **ADT** mit impliziten Synchronisationseigenschaften
    - mehrseitige Synchronisation von Monitorprozeduren
    - einseitige Synchronisation durch Bedingungsvariablen
- die **Architektur** lässt verschiedene Ausführungsarten zu
    - Monitor mit beid- oder einseitig blockierenden Bedingungsvariablen
- Unterschiede liegen vor allem in der **Semantik der Signalisierung**:
    - wirkt blockierend (Hansen, Hoare) oder nichtblockierend (Mesa) für den ein Ereignis signalisierenden Prozess
    - deblockiert einen (Hoare, Mesa) oder alle (Hansen, Mesa) auf ein Ereignis wartende Prozesse
    - die Fortführungsbedingung für den jeweils signalisierten Prozess wird garantiert (Hoare) oder nicht garantiert (Hansen, Mesa)
    - erfordert (Hansen, Mesa) oder erfordert nicht (Hoare) die erneute Auswertung der Wartebedingung bei Fortführung
    - ist falschen Signalisierungen gegenüber tolerant (Hansen, Mesa) oder intolerant (Hoare)

X.3 Maschinenprogrammebene

1. Vorwort

Prozesssynchronisation auf der Maschinenprogrammebene
Alleinstellungsmerkmal dieser Abstraktionsebene ist allgemein die durch ein BS erreichte funktionale Anreicherung der CPU, hier:
a) in Bezug auf die Einführung des Prozesskonzeptes und
b) hinsichtlich der Art und Weise der Verarbeitung von Prozessen

Techniken zur Synchronisation gleichzeitiger Prozesse können demzufolge auf Konzepte zurückgreifen, die die Befehlssatzebene nicht bietet
zu a) Prozessinkarnationen kontrolliert schlafen legen und aufwecken (X)
- Bedingungsvariable, Semaphor
- sleeping lock

zu b) Zeitpunkt der Einplanung o. Einlastung solcher Inkarnationen gezielt vorgeben (X)
- Verdrängungssperre

Mehrseitige Synchronisation kritischer Abschnitte
Schutz durch **Ausschluss gleichzeitiger Prozesse**

a) asynchrone Programmunterbrechung unterbinden, deren jeweilige Behandlung sonst einen gleichzeitigen Prozess impliziert
b) Verdrängung des laufenden Prozesses aussetzen, die anderenfalls die Einlastung eines gleichzeitigen Prozesses bewirken könnte(X)
c) gleichzeitige Prozesse allgemein zulassen, sie allerdings dazu bringen, die Entsperrung des KA eigenständig abzuwarten(X)

Alleingang (solo) eines Prozesses durch einen krit. Abschnitt sicherzustellen basiert auf ein und dasselbe

2. Verdrängungssperre

2.1 Konzept

Verdrängungsfreie kritische Abschnitte

NPCS – non-preemptive ciritical section

- Ereignisse, die zur Verdrängung eines sich in einem kritischen Abschnitt befindlichen Prozesses führen könnten, werden unterbunden

  enter
  - Flagge zeigen, dass Entzug des Prozessors <u>nicht</u> stattfinden darf
  - die möglichen Verdränung des laufenden Prozesses zurückstellen

  leave
  - Flagge zeigen, dass Entzug stattfinden darf
  - die ggr. zurückgestellte Verdrängungsanfodergung weiterleiten
- Aussetzen der Verdrängung des laufenden Prozesses ist durch (einfache) Maßnahmen an zwei Stellen der Prozessverwaltung möglich:
  1. Einplanung eines freigestellten Prozesses zurückstellen
  2. Einlastung eines zuvor eingeplanten Prozesses zurückstellen

Schutzvorrichtung – guard: „Aufgaben durchschleusen“

Bitschalter – flags, zum sperren/zurückstellen von Verdrängungen

Warteschlange zurückgestellter Verdrängungsanforderungen

3. Bedingungsvariable

3.1 Definition

<u>Bedingungsvariable – condition variable</u>

Konzepte für **bedingte krit. Abschnitte,** das zwei grundlegende Operationen definiert:

await – wait ⊨ Unterbrechungsprotokoll (X) -> S.11

- lässt einen Prozess auf das mit einer Bedingungsvariablen assoziierten Ereignisses innerhalb einer KA wartet:
  - gibt den gesperrten KA automatisch frei UND blockiert den lauf. Prozess auf die Bedingungsvariable
  - bewirbt einen durch Ereignisanzeige deblockierten Prozess erneut um den Eintritt in den KA

cause – signal: ⊨ Signalisierungsprotokoll (X)-> S.14

- zeigt das mit der Bedingungsvariable assoziierte Ereignis an
- deblockiert die gggf. auf das Ereignis wartenden Prozesse

Bedeutung: *Ermöglicht einem Prozess, innerhalb eines kritischen Abschnitts zu warten, ohne diesen während der Wartephase belegt zu halten*.

<u>Bedingter kritischer Abschnitt – conditional ciritical section – region</u>

Betreten des kritischen Abschnitts ist von einer Wartebedingung abhängig, die nicht erfüllt sein darf, um den Prozess fortzusetzen

- die Bedingung ist als Prädikat über die im krit. Abschnitt enthaltenen bzw verwendeten Daten definiert
- z.B. Fallunterscheidung, Abbruchbedingungen (Schleifen)

Auswertung der Wartebedingung muss im krit. Abschnitt erfolgen

- bei Nichterfüllung der Bedingungen wird der Prozess auf Eintritt eines zur Wartebedingung korrespondierenden Ereignisses blockiert
  - damit das Ereignis später signalisiert werden kann, muss der kritische Abschnitt beim Schlafengehen jedoch freigegeben werden
- bei (genauer: nach) Erfüllung/Signalisierung der Bedingung versucht der Prozess den kritischen Abschnitt wieder zu belegen

- ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Systemorientierte Schnittstelle

Datentyp mit optionaler Warteliste und assoziierter Sperrvariable

- die Sperrvariable (solo-t*) identifiziert einen krit. Abschnitt
    - der in cv_await zunächst freigegeben und später wieder betreten wird
- die Liste enthält die durch die Wartebedingung blockierten Prozesse
    - wodurch cv_cause schnell den zu deblockierenden Prozess finden kann

Beachte: cv_await und cv_cause kontrollieren denselben KA

- beide müssen aus demselben gesperrten KS heraus aufgerufen werden
- einem „schläfrigen Prozess" darf dabei das Wecksignal nicht entgehen

3.2 Unterbrechungsprotokoll

Entgangenes Wecksignal – lost wake-up

Laufgefahr: Angenommen, der laufende Prozess hat den KA freigegeben (CS_LEAVE ausgeführt) und wird dann von ps_sleep verdrängt:

1. da der KA nun frei ist, kann das „Ereignis" gate signalisiert werden, auf dessen Eintritt der Prozess mit ps_sleep passiv warten wollte
2. Der Signalzustellung ist dieses Vorhaben des Prozesses jedoch nicht bekannt, so geht diesem dann das „Ereignis" gate verloren
3. Nach Wiedereinlastung wird sich der Prozess in ps_sleep blockieren und sodann ggf. vergebens auf den Ereigniseintritt warten

Lösungsansatz: Abstraktions aufbrechen

Operation des Planers in zwei „elementaren" Anweisungsschritten:

1. das Ereignis, auf dessen Eintritt sich der Prozess schlafen legen will, im Prozessdeskriptor verbuchen (ps_allot)
2. den Prozess blockieren, ihm dabei die CPU entziehen, die sodann einem laufbereiten Prozess zugeteilt wird (ps_block)

Herangehensweise zur Vorbeugung entgangener Wecksignale

1. das erwartete Ereignis dem Planer noch vor CS_LEAVE bekanntgeben
2. die Prozessblockierung dem Planer nach dem CS_LEAVE anzeigen

„schläfrigen" Prozess disponieren

Lösungsansatz, der **besondere Vorsicht**im Planer erforderlich macht:

- nach CS_LEAVE kann die Fortsetzungsbedingung für einen noch laufenden Prozess signalisiert werden -> cv_cause
- der signalisierte Prozess kommt auf die Bereitliste, von der er sich durch ps_block ggf. selbst wieder entfernen und einlasten könnte

Analogie zum Sonderfall **„Leerlauf" –idle state**

- holst ein sich schlafen legender Prozess sich selbst von der Bereitliste, bleibt er eingelastet und kehrt aus ps_block zurück

3.3 Signalisierungsprotokoll

Fortsetzungsbedingung anzeigen – Wartebedingung aufheben

//if()… await()

- ➔ der Prozess ist nach dem if und wird dann unterbrochen bevor er ins await geht
- ➔ wenn die if-Bedingung in einem anderen Code der nicht im gleichen kritischen Abschnitt ist geändert wird, dann kann eine cause-Signalisierung die die selbe Sperrvariable wie das await hat, daran vorbeirauschen weil es das nie sieht

Laufgefahr, sollte die Aufhebung der Wartebedingung nicht aus dem cv_await umfassenden krit. Abschnitt heraus erfolgen:

- in dem Fall könnte cv_cause das „Ereignis" gate überlappen mit der Ausführung des krit. Abschnitts anzeigen
- krit. ist der Teilabshcnitt von Auswertung der Wartebedingung des KA bis Ausführung von cv_await bzw (dem ps_allot in) cv_allot
- !!! cv_await und cv_cause müssen paarweise dieselbe Bedingungsvariable (fate) bzw. denselben krit. Abschnitt bedienen

4. Semaphor

4.1 Definition

<u>Synchronisation durch Austausch von Zeitsignalen</u>

Semaphor – Zeichenträger

- eine „nicht negative ganze Zahl", für die – zwei **unteilbare Operationen** definiert sind:
    - P (erniedrigen, down, wait)
        - hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
        - ansonsten wird der Semaphor um 1 dekrementiert
    - V (erhöhen, up, signal)
        - inkrementiert den Semaphor um 1
        - auf den Semaphor ggf. blockierte Prozesse werden deblockiert
- ein abstrakter Datentyp zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen

4.2 Implementierung

<u>EWD (Edsgar Wybe Dijkstra) beim Wort genommen</u>

Programme für P und V bilden **kritische Abschnitte:**

1. gleichzeitiges Ausführen von P kann mehr Prozesse passieren lassen, als es der Semephorwert (sema) erlaubt
2. gleichzeitiges Zählen kann Werte hinterlassen, die nicht der wirklichen Anzahl der ausgeführten Operationen (P, V) entsprechen
3. gleichzeitiges Auswerten der Bedingung (P) und Hochzählen (V) kann das Schlafenlegen (ps_sleep()) von Prozessen bedingen, obwohl die Wartebedingung für die schon nicht mehr gilt („lost wake-up")

<u>Konsequenzen für eine Implementierung von P und V</u>

Pessimistischer Ansatz zum Schutz der kritischen Abschnitte, nämlich die **mehrseitige (blockierende) Synchronisation** von P und V

1. **wechselseitigeAusschluss** wird die Funktionen bei ihrer Ausführung nicht überlappen lassen, weder sich selbst noch gegenseitig
    - P und V sind durch ein gemeinsames „Schloss" zu schützen
2. Schlafenlegen eines Prozesses in P muss implizit die **Entsperrung** des krit. Abschnitts zur Folge haben
    - sonst wird kein V die Ausführung vollenden können
    - als Folge werden in P schlafende Prozesse niemals aufgeweckt
3. Aufwecken von Prozessen in V sollte bedingt erfolgen, und zwar falls wenigstens ein Prozess in P schlafengelegt wurde

Optimistischer Ansatz als (bessere) Alternative, die den Schutz von P und V durch *nichtblockierende Synchronisation* erreicht (äußerst knifflig).

<u>Mehrseitige Synchronisation von P und V</u>

4.3 Varianten

<u>Arten von Semaphoren – Instrumente zur Betriebsmittelvergabe, differenziert nach Wertebereichen der Semaphoren</u>

binäre Semaphore – binary Semaphore

- verwalrtet zu einem Zeitpunkt immer nur genau ein Betriebsmittel
    - wechselseitiger Ausschluss (mutual exclusion, mutex)
- vergibt **unteilbare Betriebsmittel** an Prozesse
- beseitzt den Wertebereich [0, 1]

zählender Semaphor – couting semaphore, general semaphore

- verwaltet zu einem Zeitpunkt mehr als ein Betriebsmittel
    - d.h. mehrere Betriebsmittelexemplare desselben Typs
- vergibt teil- bzw. **konsumierbare Betriebsmittel** an Prozesse
- besitzt den Wertebereich [0, N] für N Betriebsmittel

<u>Arten von Betriebsmittel (BM) – Semaphore und BMVerwarltung</u>

wiederverwendbare Betriebsmittel werdn angefordert und freigegeben

- ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (Puffer)
    - teilbar zu einer Zeit von mehreren Prozessen belegbar
    - unteilbar zu einer Zeit von einem Prozessor belegbar

- auch ein krit. Abschnitt ist solch ein BM
    - von jedem Typ gibt es jedoch nur ein einziges Exemplar

konsumierbare BM werden erzeugt und zerstört

- ihre Anzahl ist (log.) unbegrenzt: Signale, Nachrichten, Interrupts
    - Produzent kann beliebig viele davon erzeugen
    - Konsument zerstört sie wieder bei Inanspruchnahme
- Produzent und Konsument sind voneinander abhängig

Ausschließender Semaphor

Vergabe unteilbarer BM, Schutz krit. Abschnitte

```
semaphore_t lock = {1, 0};

int fai (int *ref) {
    int aux;

    P(&lock);
    aux = (*ref)++;
    V(&lock);

    return aux;
}
```

unteilbares Betriebsmittel „KA"

- von dem es nur ein Exemplar gibt
- der Initialwert des Semaphors ist 1

mehrseitige Synchronisation des KA

- die Reihenfolge gleichzeitiger Prozesse ist unbestimmt
- gleichzeitig können jedoch nicht mehrere Prozesse im KA sein

Syntaktischer Zucker

```
#define P(sema) wsp_prolaag(sema)
#define V(sema) wsp_verhoog(sema)
```

Signalisierender Semaphor

Vergabe konsumierbarer BM

konsumierbares BM

- ist vor dem Verbrauch zu erzeugen
- der Initialwert des Semaphors ist 0

einseitige Synchronisation

- nur einer von beiden beteiligten Prozessen wird ggf. blockieren
- nämlich der Konsument, wenn noch kein Datum verfügbar ist
- er ist später von dem Konsumenten wieder freizustellen

Bregrenzter Datenpuffer: max. ein Platz

- Daten gehen verloren, wenn die Prozesse nicht im gleichen Takt arbeiten:
    - Kosument* -> (Produzent -> Konsument)+

Semaphore „considered harmful"

- auf Semaphore basierende Lösungen sind komplex und fehleranfällig
    - Synchronisation: **Querschnittsbelang** nichtseq. Programme (krit. Abschnitte neigen dazu, mit ihren P/V-Operationen quer über die Software verstreut vorzuliegen)
    - das schützen gemeinsamer Variablen oder krit. Abschnitte kann leicht üebrsehen werden
- hohe Gefahr von **Verklemmung** von Prozessen
- linguistische Unterstützung rediziert Fehlermöglichkeit -> Monitor

5. Zusammenfassung

- Synchronisation in der Maschinenprogrammebene kann auf Konzepte von Betriebssystemen zurückgreifen
    - die den Zeitpunkt von Einplanung oder Einlastung gezielt beeinflussen
    - die Prozesse kontrolliert schlafen legen und wieder aufwecken
- durch eine **Verdrängungssperre** wird die Einplanung bzw. Einlastung von Prozessen erst verzögert wirksam
    - kritische Abschnitte werden verdrängungsfrei durchlaufen, aber
    - unabhängige gleichzeitige Prozesse werden unnötig zurückgehalten
- eine **Bedingungsvariable** ermöglicht Prozessen innerhalb eines KA zu warten, ohne diesen während der Wartephase belegt zu halten
    - ein Datentyp mit optionaler Warteliste und assoziierter Sperrvariable
    - *await* und *cause* müssen im selben gesperrten KA benutzt werden
- ein **Semaphor** ist ein kompositer Datentyp bestehend aus Zähl-, Sperr- und Bedingungsvariable
    - unterschieden wird zwischen binärem und zählendem Semaphor
    - ein *Mutex* ist ein binärer Semaphor mit Prozesseigentümerschaft

X.4 Befehlssatzebene

1. Vorwort

Alleinstellungsmerkmal dieser Abstraktionsebene sind die in der CPU manifestierten Fähigkeiten eines Rechnersystems.

a. in Bezug auf die Bereitstellung von Spezialbefehlen
b. Semantik dieser Befehle zur Prozessverarbeitung

Techniken zur Syn. gleichzeitiger Prozesse können auf einfache, elementare Konzepte zurückgreifen

- zu a. Möglichkeit, externe/interne Prozesse aussperren zu können (Unterbrechungssperre (X), Schlossvariable und Umlaufsperre (X))
- zu b. Möglichkeit, krit. Abschnitte so ausformulieren zu können, dass gleichzeitige Prozesse nicht ausgesperrt werden (nicht blockierende Syn. (X))

2. Unterbrechungssteuerung

2.1 Prinzip

Kontrolle asyn. Programmunterbrechungen

Ansatz: verhindern oder tolerieren -> Verzögerung der…
überlappenden Aktivität -> pessimistisches Verfahren
überlappten Aktivität-> optimistisches Verfahren

Unterbrechung sperren einfach – aber nicht immer zweckmäßig

- Faustregel: harte Syn. ist möglichst zu vermeiden !!!

Wiedersehen mit einem alten Problem: überlapptes Zählen

Wettlaufsituation: kritischer Abschnitt: ++; Laufgefahr vorbeugen, ELOP, unteilbares Zählen konstruktiv und problemadäquat sicherstellen)

Verhinderung vf. Tolerierung von *Interrupts*

3. Schlossvariable

3.1 Definition

engl. lock variable

Datentyp

definiert zwei Operationen:

aquire – lock ⊨ Eintrittsprotokoll

- verzögert einen Prozess, bis das zugehörige Schloss offen ist (offen – Prozess fährt unverzögert fort)
- verschließt das Schloss („von innen“), wenn es offen ist

release – unlock ⊨ Ausstrittsprotokoll

- öffnet ein Schloss, ohne den öffnenden Prozess zu verzögern

Implementierung dieses Konzepts werden auch als Schlossalgorithmen (lock algorithms) bezeichnet.

3.2 Implementierung

Prinzp – mit Problem(en)

- die Phase vom Verlassen der Kopfschleife bis zum Setzen der Schlossvariable ist kritisch
- gleichzeitige Prozesse können das Schloss geöffnet vorfinden, dann jeweils schließen und den krit. Abschnitt gemeinsam belegen
- Abprüfen und setzen (test and set, TAS) der Schlossvariablen (!!!)
    - muss als Elementaroperation und wirklich atomar ausgelegt sein

Umlaufsperre – spin lock

TAS kommt in zwei Varianten

a) int tas(lock_t *)
b) int tas(volatile bool *)

„Drehschloss“

```
void lv_acquire (lock_t *lock) {
    while (TAS(lock));
}
```

!Gefahr von Leistungsabfall

- pausenloses Schleifen allein nur mit TAS:
    a) erhöht das Risiko, Anforderungen von Programmunterbrechungen oder Prozessverdrängungen zu verpassen
        - die Unterbrechungs- bzw. Verdrängugnssperre ist fast nur noch gesetzt

b) hindert andere Prozessoren am Buszugang bzw. sorft für eine überaus hohe Last im Kohärenzprotokoll des Zwischenspeichers
    - der Prozessor führt das nurnoch „read-modify-write“-Zyklen durch
- das Problem verschärft sich massiv, wenn (viele) gleichzeitige Prozesse den Wettstret – contention um das „Drehschloss“ aufnehmen

<u>Sensitive Umlaufsperre</u>
- nur lesender Zugriff beim Warten
- Zwischenspeicherzeile gemeinsam benutzbar
- einmal „read-modity-write“ (TAS)
- beruhigend für den Busverkehr

Eigenschaften wie zuvor, zusätzlich:
- Wettstreit aus dem Wege gehend
- zusätzliche Wartezeit – back-off nach gescheitertem TAS
- prozessspezifisch, ansteigend

Eyponentielles Zurücktreten (exponential back-off)
- beschränkte Wartezeitverdopplung mit jedem gescheitertem Versuch

Nichtinvasives Warten

```
void lv_acquire (lock_t *lock) {
    do {
        while (lock->busy);
    } while (TAS(lock));
}
```

Zurücktretendes Warten

```
void lv_acquire(lock_t *lock) {
    while (true) {
        while (lock->busy);
        if (!TAS(lock)) break;
        lv_backoff(lock);
    }
}
```

3.3 Diskussion

<u>Aktives Warten (busy waiting)</u>

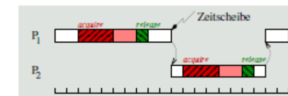
Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess..
- kann keine Änderungen der Bedingung herbeiführen, auf die er wartet
- behindert andere Prozesse, die sinnvolle Arbeit leisten könnten
- schadet damit letzlich auch sich selbst

Je länger der Prozess den Prozessor für sich behält, umso länger muss er darauf warten, dass andere Prozesse die Bedingung erfüllen, auf die er selbst wartet.
- in den meistne Fällen sind Effizienzeinbußen in Kauf zu nehmen
- es sei denn, jeder Prozess hat seinen eigenen realen Prozessor(kern)

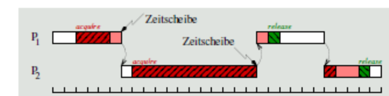
(!) Allg. ein nur bedingt effektives Verfahren:
- notwendige Bedingung: kurze Laufzeit des zu schützenden KA
- hinreichende Bed.: CPU-Abgabe nach gescheitertem Versuch bei gemeinsam betnutzten Prozessor(kern)

Aktives Warten ohne Prozessorabgabe

„Spin locking considered harmful“



	T_s	T_q	T_q/T_s
P_1	12	20	1.67
P_2	8	8	1.0



	T_s	T_q	T_q/T_s
P_1	12	24	2.0
P_2	17	23	1.35

Verbesserung: Prozessorabgabe in der Warteschleife (vgl. S. 34)

laufend ↦ bereit	in Laufbereitschaft bleiben	*ps_forgo*
laufend ↦ blockiert	schlafend die Schlossfreigabe erwarten	*ps_sleep*

4. Nichtblockierende Synchronisation

4.1. Motivation

<u>Blockierende Synchronisation „considered harmful“</u>

Leistung (engl. *performance*) insbesondere in SMP-Systemen [1]
- „*spin locking*“ reduziert ggf. massiv Busbandbreite

Robustheit (engl. *robustness*) „*single point of failure*“
- ein im kritischen Abschnitt scheiternder Prozess kann schlimmstenfalls das ganze System lahm legen

Einplanung (engl. *scheduling*) wird behindert bzw. nicht durchgesetzt
- un- bzw. weniger wichtige Prozesse können wichtige Prozesse „ausbremsen“ bzw. scheitern lassen
- Prioritätsverletzung, Prioritätsumkehr [3]
    - Mars Pathfinder [5, 2]

Verklemmung (engl. *deadlock*) einiger oder sogar aller Prozesse

4.2. Prinzip

<u>Optimistisches Verfahren der Synchronisation</u>

Nebenläufigkeit unterstützen, nicht einschränken wie beim wechselseitigen Ausschluss

Koordinierung sich einander ggf. überlappender Aktivitäten, ohne dabei **gleichzeitige Prozesse** auszuschließen
- toleriert (pseudo-) parallele Programmausführung
    - parallel: Multiprozessor, wirkliche Parallelität

- pseudopar.: Uniprozessor, Parallelität durch Unterbrechungen
- die Verfahren greifen auf nichtprivilegierte Befehle der ISA zurück
  - CISC: TAS, FAA, CAS bzw. CMPXCHG
  - RISC: LL/SC
- d.h.,sie funktionieren im Benutzer- wie auch im Systemmodus

Beachte(!)
- kein wechselseitiger Ausschluss =>**Verklemmungsvorbeugung**
- die *benutzten* Befehle müssen „echte" **Elementaroperatinonen**

Muster nichtblockierender Synchronisatoin (NBS)

erledige NBS mit CAS;
wiederhole
ziehe *lokale Kopie* des Inhalts der *Adresse* einer globalen Variablen;
verwende die Kopie, um einen neuen *lokalen Wert* zu berechnen;
versuche CAS: sichere den *lokalen Wert* an die *Adresse*, wenn ihr Inhalt immer noch mit dem Wert der *lokalen Kopie* identisch ist;
solange CAS scheitert;
basta.

pros
- Tolerierung beliebiger Überlappungsmuster
- transparent für die Einplanung: keine Prioritätsumkehr
- Verklemmungsvorbeugung: kein wechselseitiger Ausschluss
- Robustheit: keine hängenden Sperren bei Programmabbrüchen

cons
- Wiederverwendung sequentieller Altsoftware unmöglich
- Gefahr von Verhungerung (engl. *starvation*) in simplen Lösungen
- Entwicklung nebenläufiger Varianten im Regelfall nicht trivial

1.2 Elementaroperation

Bedingte Wertzuweisung: *Compare und Swap, CAS*
- ref: die Adresse des atomar, bedingt zu ändernden Speicherworts
- exp: der unter der Adresse ref erwartete alte Wert
- val: der unter der Adresse ref zu speichernde neue Wert
- die Speicherung erfolgt nur, wenn der unter ref gepsicherte Wert dem erwarteten Wert exp gleicht
- lag Gleichheit vor, liefert die Funktion true, anderenfalls false

1.3 Diskussion

Generationszähler „*considered harmful*"?

Abhilfe (engl. *workaround*) zum ABA-Problem ⇝ Bedingte Lösung
- die Effektivität des Lösungsansatzes steht und fällt mit dem für den Generationszähler definierten **endlichen Wertebereich**
  - dessen Auslegung letztlich vom jeweiligen Anwendungsfall abhängt
- **Überlappungsmuster** gleichzeitiger Prozesse haben Einfluss auf den für den Generationszähler zur Verfügung zu stellenden Wertebereich
  - bestimmt durch Zusammenspiel *und* Anzahl der wettstreitigen Fäden
  - ein Bit kann reichen, ebenso, wie ein *unsigned int* zu klein sein kann
- diese, dem jeweiligen Anwendungsfall zu entnehmenden Muster zu entdecken, ist zumeist schwer und nicht selten unmöglich

Vorbeugung (engl. *prevention*) muss zuerst kommen — sofern machbar:
- beliebige Überlappungsmuster konstruktiv (Entwurf) ausschließen
- oder auf Adressreservierungsverfahren der Hardware zurückgreifen
  - unterstützt nicht jede Hardware, ist nur typisch für RISC
  - z.B. ELOP-Paar *load linked, store conditional* (LL/SC) verwenden

Wiederholungsversuche — Aktives Warten?

Scheitern der etwa durch CAS[2] abzuschließenden **Transaktion** zieht die Wiederholung des kompletten Vorbereitungsvorgangs nach sich
- je höher der Grad an Wettstreitigkeit unter gleichzeitigen Prozessen, umso höher die Wahrscheinlichkeit, dass CAS scheitert
- ein zur Umlaufsperre sehr ähnliches Problem ergibt sich, das jedoch durch **sensitive Techniken** gleicher Art lösbar ist (vgl. S. 13–14)
  - Häufigkeit von „*read-modify-write*"-Zyklen pro Durchlauf minimieren
  - prozessspezifisches Zurücktreten vom erneuten Transaktionsversuch
  - variable Wartezeiten, um Konflikte bei Wiederholungen zu vermeiden

Unterschied zur Umlaufsperre
- gleichzeitige Prozesse müssen nicht untätig darauf warten, dass ein kritischer Abschnitt entsperrt wird und somit frei ist
- im Gegensatz zur Umlaufsperre kommen sie während ihrer Wartezeit mit ihren (lokalen) Berechnungen voran

[2] Für LL/SC-artige Elementaroperationen gilt dies ebenfalls.

5. Zusammenfassung

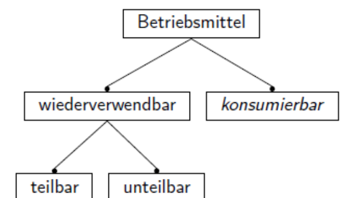
- **Unterbrechungssteuerung** ist leicht, aber nicht immer zweckmäßig
  - verlangt von den Prozessen besondere Rechte: **privilegierte Befehle**
  - unbeteiligte gleichzeitige Prozesse werden unnötig ausgesperrt
  - externe Ereignisse (*Interrupts*) können verloren gehen
  - ungeeignet für Multi(kern)prozessorsysteme: lokale Signifikanz
- die **Schlossvariable** kommt meist mit **Umlaufsperre** zum Einsatz
  - pessimistischer Ansatz zum Schutz kritischer Abschnitte: *leicht*
    - gleichzeitige Prozesse werden als sehr wahrscheinlich angenommen
    - bezogen auf den jeweils zu schützenden kritischen Abschnitt
  - im Regelfall bedingen Leistungsanforderungen **sensitive Verfahren**
    - Häufigkeit von „*read-modify-write*"-Zyklen pro Durchlauf minimieren
    - prozessspezifisches Zurücktreten vom erneuten Schließversuch mit TAS
    - variable Wartezeiten, um Konflikte bei Wiederholungen zu vermeiden
  - als blockierende Synchronisation besteht hohe **Verklemmungsgefahr**
- **nichtblockierende Synchronisation** ist frei von Verklemmungen
  - optimistischer Ansatz zum Schutz kritischer Abschnitte: *schwer*
  - die Verfahren müssen ebenfalls sensitiv für Plattformeigenschaften sein

XI Betriebsmittelverwaltung

1. Betriebsmittel

Klassifikation

Hardware: CPU, Speicher // Geräte (Peripherie) // Signale
Software: Dateien, E/A-Puffer // Seitenrahmen // Deskriptoren, ... // Signale, Nachrichten



Wettbewerb um Betriebsmittel

- konsumierbare BM (unbegrenzte #, einseitige Syn)
- wiederverwendbare BM (begrenzte #, mehrseitige Syn)

Konfliktfall bei BM-zuteilung

Prozesse befinden sich untereinander im **Konflikt**, wenn:

1. eine **begrenzte Anzahl gemeinsamer Betriebsmittel** vorrätig ist
2. diese Betriebsmittel **unteilbar** und von derselben Art sind
3. ein Zugriff darauf gleichzeitig geschieht: **gleichzeitige Prozesse** [3]

Prozesse sind im **Wettstreit** (engl. *contention*) um ein Betriebsmittel, wenn einer das Betriebsmittel anfordert, das ein anderer bereits besitzt

- der anfordernde Prozess blockiert und wartet auf die Freigabe des Betriebsmittels durch den Prozess, der das Betriebsmittel belegt
- der das Betriebsmittel belegende Prozess löst den auf die Freigabe des Betriebsmittels wartenden Prozess aus, deblockiert ihn wieder

Verwaltung

Ziel: Durchsetzung der vorgegebenen Betriebsstrategie und eine optimale Realisierung in Bezug auf relevante Kriterien

Aufgaben

Buchführung über die im Rechensystem vorhandenen BM
Steuerung der Verarbeitung von BManforderungen

- Entgegennahme, Überprüfung (z.B. der Zugriffsrechte)
- Einplanung der Nutzung angeforderter BM durch Prozesse
- Einlastung (Zuteilung) von BM
- Entzug oder Freigabe von Prozessen benutzter BM

BMentzug

- Zurücknahme der BM, die von einem "aus dem Ruder geratenen" Prozess belegt werden
- bei Visualisierung zusätzlich: Rückforderung und Neuzuteilung eines realen BM, wobei das zugehörige virt. BM dem Prozess zugeteilt bleibt

Verfahrensweisen

1. **statisch**

- vor Laufzeit oder vor dem Laufzeitabschnitt
- Anforderungen <u>aller</u>(im Abschnitt) benötigten Betriebsmittel
- Zuteilung der BM erfolgt ggf. lange vor ihrer eigentlichen Benutzung
- Freigabe <u>aller</u> belegten BM mit Laufzeit(abschnitt)ende
- !Risiko einer <u>*suboptimalen Auslastung*</u> der BM

2. **dynamisch**

- zur Laufzeit, in beliebigen Laufzeitabschnitten
- Anforderung des jeweils benötigten Betriebsmittels <u>bei Bedarf</u>
- Zuteilung des jeweiligen BM erfolgt "im Moment" seiner Benutzung
- Freigabe eines belegten BMs, sobald <u>kein Bedarf</u> mehr besteht
- ! Risiko der Verklemmung von abhängigen Prozessen

2. Verklemmung

Grundlagen

Stillstand von Prozessen bei aktivem Warten

live·lock ist ...

- ein *deadlock*-ähnlicher Zustand, in dem die involvierten Prozesse zwar nicht blockieren, sie aber auch keine wirklichen Fortschritte in der weiteren Programmausführung erreichen
- wenn die an der Verklemmung beteiligten Prozesse **wechselseitig aktiv** auf die Bereitstellung von Betriebsmitteln **warten**:
  - ohne Prozessorabgabe ↦ *busy waiting*
  - mit Prozessorabgabe, in Laufbereitschaft bleibend ↦ *lazy waiting*
- das „größere Übel", da dieser Zustand nicht eindeutig erkennbar ist und damit die Basis zur „Erholung" fehlt
  - die verklemmten Prozesse haben die **Einplanungszustände** „laufend" oder „bereit" (d.h., jeden anderen außer „blockiert")
  - die Unterscheidung von unverklemmten Prozessen ist kaum möglich

Verklemmung sind durch geschickte Prozesseinplanung verhinderbar – vorausgesetzt, die Prozesse wie auch ihre BManforderungen sind alle bekannt.

Voraussetzungen für Verklemmungen
Notwendige und hinreichende Bedingungen

notwendige Bedingungen (müssen erfüllt sein, damit die Aussage zutreffen kann)

1. exklusive Belegung von Betriebsmitteln („*mutual exclusion*")
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. Nachforderung von Betriebsmitteln („*hold and wait*")
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. kein Entzug von Betriebsmitteln („*no preemption*")
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

hinreichende Bedingung (muss erfüllt sein, damit die Aussage zutrifft bzw. „bewiesen" ist)

4. zirkulares Warten (engl. *circular wait*)
   - Existenz einer geschlossenen Kette wechselseitig wartender Prozesse

Beispiel

Speisende Philosophen: Synchronisationsaspekte [2]

mehrseitige Synchronisation

- **wechselseitiger Ausschluss** beim Gebrauch wiederverwendbarer, unteilbarer (nicht entziehbarer) Betriebsmittel
- keine zwei benachbarten Philosophen können gleichzeitig dasselbe Stäbchen gemeinsam benutzen

einseitige Synchronisation

- ein Philosoph muss warten, bis seine beiden Nachbarn ihm ein Stäbchen zur Verfügung gestellt haben

Randbedingungen

- jeder Philosoph fordert die Stäbchen nacheinander an
  - kein Philosoph legt ein Stäbchen zurück, wenn er feststellt, dass das andere bereits vom Nachbarn aufgenommen worden ist
- ein Philosoph kann seinem Nachbarn ein bereits aufgenommenes Stäbchen nicht entziehen

Praxisrelevanz:
auf mehrere Magnetbänder vorliegende Daten sortieren. einen kontinuierlichen Strom kodierter Infos umkodieren....

überall dort, wo eine von mehreren Prozessen benutzte Routine (mind.) zwei wiederverwendbare BM zugleich benötigt, diese aber nur nacheinander angefordert werden können

Gegenmaßnahmen

Vorbeugung

indirekte Methoden entkräften eine der Bedingungen 1-3

1. nicht-blockierende Verfahren verwenden
2. BManforderungen unteilbar (atomar) auslegen
3. BMentzug durch <u>Virtualisierung</u> ermöglichen (vir. Speicher/Geräte/Prozessoren)

direkte Methoden entkräften Bedingung 4

4. lineare/totale Ordnung von BMklassen einführen

Regeln, die das Eintreten von Verklemmung verhindern
Methoden, die zur Entwurfs- bzw. Implementierungszeit greifen

Vermeidung

Verhinderung von Wartezyklen durch strategische Maßnahmen:

- keine der drei notwendigen Bedingungen wird entkräftet

- fortlaufende Bedarfsanalyse schließt zirkulares Warten aus

Prozesse und ihre BManforderungen sind zu steuern:

- das System wird (laufend) auf "unsichere Zustände" hin überprüft
- Zuteilungsablehnung im Falle ungedeckten BMbedarfs
- anfordernde Prozesse nicht bedienen bzw. frühzeitig suspendieren
- BMnutzung einschränken -> "sichere Zustände"

Erkennung und Erholung

Verhinderung unsicherer Zustände

sicherer Zustand ist, wenn eine Folge der Verarbeitung vorhandener Prozesse existiert, in der alle BManforderungen erfüllbar sind

unsicherer Zustand ist, wenn eine solche Folge nicht existiert; Erkennung dieses Zustands z.B. durch:

- Betriebsmittelbelegungsgraph
  - damit Vorhersage über das Eintreten von Zyklen treffen -> $O(n^2)$
  - bei jeder BManforderung den Graphen überprüfen
- Bankiersalgorithmus
  - Prozesse beenden ihre Operationen in endlicher Zeit
  - BMbedarf aller Prozesse übersteigt Gesamtvorrat nicht
  - Prozesse definieren einen verbindlichen Kreditrahmen
  - BMzuteilung erfolgt variabel innerhalb des Rahmens
- die Verfahrensweisen führen Prozesse dem long-term scheduling zu

Verklemmungserkennung

Verklemmungen werden in Kauf genommen...

- nichts im System verhindert das Auftreten von Wartezyklen
- keine der vier Bedingungen wird entkräftet

Ansatz: **Wartegraph** erstellen und auf Zyklen hin untersuchne -> $O(n^2)$

- zu häufige Überprüfung verschwendet BM/Rechenleistung
- zu seltene Überprüfung lässt BM brach liegen

Zyklensuche geschieht zumeist in großen Zeitabständen, wenn..

- BManforderungen zu lange andauern
- die Auslastung der CPU trotz Prozesszunahme sinkt
- die CPU bereits pber einen sehr langen Zeitraum untätig ist

Verklemmungsauflösung – Erholungsphase nach der Erkennungsphase

Prozesse abbrechen und dadurch Betriebsmittel frei bekommen

- verklemmte Prozesse schrittweise abbrechen (gr. Aufwand)
- alle verklemmten Prozesse terminieren (gr. Schaden)

BM entziehen und mit dem "effektivsten Opfer" (?) beginnen

- der betreffende Prozess ist zurückzufahren/wieder aufzusetzen
- ein Aushungern der zurückgefahrenen Prozesse ist zu vermeiden

!Gratwanderung zwischen Schaden und Aufwand

- Schäden sind unvermeidbar und die Frage ist, wie sie sich auswirken

3. Zusammenfassung

Nachlese ...

Verfahren zum **Vermeiden/Erkennen** sind eher praxisirrelevant

- sie sind kaum umzusetzen, zu aufwändig und damit nicht einsetzbar
- zudem macht die — nach wie vor stark ausgeprägte — Vorherrschaft sequentieller Programmierung diese Verfahren wenig notwendig

Vorbeugung ist besser als heilen — notwendige Bedingungen entkräften:

1. durch **nicht-blockierende Synchronisation** kritischer Abschnitte
   - keine exklusive Belegung von kritischen Abschnitten durchführen
2. durch **Virtualisierung**, wenn 1. nicht möglich/unpraktisch ist
   - Prozesse beanspruchen/belegen nur virtuelle Betriebsmittel
   - der Trick besteht darin, in kritischen Momenten den Prozessen (ohne ihr Wissen) physische Betriebsmittel entziehen zu können
   - dadurch wird die Bedingung der Nichtentziehbarkeit entkräftet

Resümee

- **Betriebsmittel** zeigen sich als Entitäten von Hardware und Software
  - wiederverwendbar: begrenzt verfügbar: teilbar, unteilbar
  - konsumierbar: unbegrenzt verfügbar
- Ziele, Aufgaben und Verfahrensweise der **Betriebsmittelverwaltung**
  - Betriebsmittelanforderung frei von Verhungerung/Verklemmung
  - Buchführung der Betriebsmittel, Steuerung der Anforderungen
  - statische/dynamische Zuteilung von Betriebsmitteln
- für eine Verklemmung müssen **vier Bedingungen** gleichzeitig gelten
  - exklusive Belegung, Nachforderung, kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
  - nicht zu vergessen: Verklemmung bedeutet „*deadlock*" oder „*livelock*"
- die **Gegenmaßnahmen** sind:
  - Vorbeugen, Vermeiden, Erkennen & Erholen
  - die Verfahren können im Mix zum Einsatz kommen

XII.1 Speicherverwaltung: Adressraumkonzepte

1. Grundlagen

Fragmente

Herangehensweise zur Verwaltung des Arbeitsspeichers

statisch

- Arbeitsspeichergebiete maximaler, fester Größe
- Gefahr von Leistungsbegrenzung/-verlusten

dynamisch

- Arbeitsspeicherfragmente variabler Größe
- Zusammenspiel Laufzeit- und Betriebssystem

Zuteilungseinheiten

Hardwarevorgabe und Verschnitt – Zuteilungseinheiten als Vielfaches v. Bytes oder Seitenrahmen
Adressraumausprägung:
Seitennummerierung

- Fragmente – vielfaches v. Seitenrahmen.
- ggf. wird mehr Speicher als benötigt zugeteilt
- interne Fragmentierung des Seitenrahmens

Segmentierung

- Fragmente – vielfaches von Bytes -> Segmente
- ggf. ist ein passendes Stück nicht verfügbar
- externe Fragmentierung des Arbeitsspeichers

Verschnitt zu optimieren! anfallender Rest bei der Speicherzuteilung ist "abfall" (interne Fragm.) oder "Hohlräume" (externe Fragmentierung)

Strategien

Politiken bei der Speicherverwaltung

Speicherzuteilungsverfahren .... obligatorisch

Platzierungsstrategie (engl. *placement policy*)

- **wohin** im Arbeitsspeicher ist ein Fragment abzulegen?
  - dorthin, wo der Verschnitt am kleinsten/größten ist?
  - oder ist es egal, weil Verschnitt zweitrangig ist?

Speichervirtualisierung .... optional

Ladestrategie (engl. *fetch policy*)

- **wann** ist ein Fragment in den Arbeitsspeicher zu laden?
  - im Moment der Anforderung durch einen Prozess?
  - oder im Voraus, auf Grund von Vorabwissen oder Schätzungen?

Ersetzungsstrategie (engl. *replacement policy*)

- **welches** Fragment ist ggf. aus den Arbeitsspeicher zu verdrängen?
  - das älteste, am seltensten genutzte oder am längsten ungenutzte?

2. Adressräume

Überblick

physikalischer Adressraum (auch: realer/physischer Adressraum)

- **nicht linear** adressierbarer E/A- und Speicherbereich, dessen Größe der Adressbreite der CPU entspricht
  - 2^N Bytes, bei einer Adressbreite von N Bits
  - von Lücken durchzogen $\leadsto$ ungültige Adressen

logischer Adressraum

- **linear** adressierbarer Speicherbereich von 2^M Bytes bei einer Adressbreite von N Bits
  - $M = N$ z.B. im Fall einer *Harvard-Architektur*[1]
  - $M < N$ sonst

virtueller Adressraum

- logischer Adressraum, der 2^K Bytes umfasst
  - $K > N$ bei Speicherbankumschaltung, Überlagerungstechnik
  - $K \leq N$ sonst

[1] Getrennter Programm-, Daten- und E/A-Adressraum.

Physischer Adressraum

Logischer Adressraum

Segmentierung durch die Maschinenprogrammebene

Logische Unterteilung von Prozessadressräumen

- Ebene4-Programme (mind.) 2 Segmente logisch aufgeteilt
    - Text (Maschinenanweisungen, Programmkonstanten)
    - Daten (initialisierte Daten, globale Variablen, Halde)
- Ebene3-Programme (mind.) ein weiteres Segment
    - Stapel (lokale Var., Hilfsvar., aktuelle Parameter)

BS verwaltet diese Seg. im phys. Adressraum (ggf. mit Hilfe der MMU. Diese legt eine Organisationsstruktur auf den phys. Adressraum und unterteil ihn in Seiten fester oder Segmente variabler Länge)

Ausprägung v. Prozessadressräumen

eindimensional in Seiten aufgeteilt – paged

- Programmadresse A_p bildet Tupel (p, o)
- Seite -> Seitenrahmen des phys. Adressraums

$p = A_P \texttt{ div } 2^N \rightsquigarrow$ Seitennummer (engl. *page number*)
$o = A_P \texttt{ mod } 2^N \rightsquigarrow$ Versatz (engl. *byte offset*)

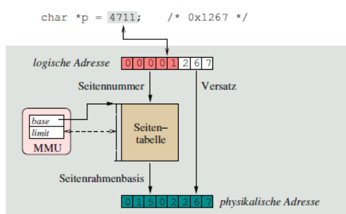
- mit 2^N gleich der Seitengröße (engl. *page size*) in Bytes

zweidimensional in Segmente aufgeteilt – segmented

- eine Programmadresse A_s bildet ein Paar (S,A)
- Segment -> Folge von Bytes/Seitenrahmen des phys. Adressraums

Adressumsetzung: seitenorientiert

Seitennummerierter Adressraum (engl. *paged address space*)

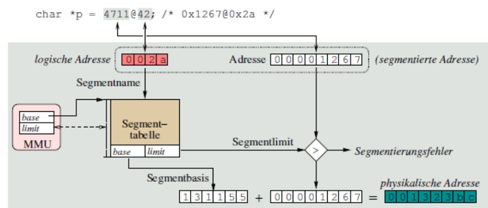


- **Seitennummer** ist Index in **Seitentabelle**
    - pro Prozess
    - dimensioniert durch die MMU
- ungültiger Index führt zum *Trap*

- die indizierte Adressierung (der MMU) liefert einen **Seitendeskriptor**
    - enthält die **Seitenrahmennummer**, die die Seitennummer ersetzt
    - entspricht der **Basisadresse des Seitenrahmens** im phys. Adressraum
- setzen der Basis-/Längenregister der MMU $\rightsquigarrow$ Adressraumwechsel

Adressumsetzung: explizit segmentorientiert

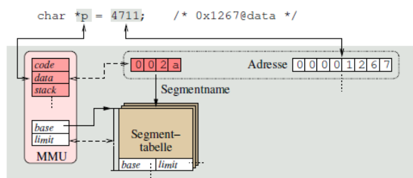
Segmentierter Adressraum (engl. *segmented address space*)



- **Segmentname** ist Index in die **Segmenttabelle** eines Prozesses
    - dimensioniert durch die MMU, ungültiger Index führt zum *Trap*
- indizierte Adressierung (der MMU) ergibt den **Segmentdeskriptor**
    - enthält **Basisadresse** und **Länge** des Segments (engl. *base/limit*)
    - $address_{phys} = address > limit \; ? \; (Trap, 0) : base + address$

Adressumsetzung: implizit segmentorientiert

Segmentregister bzw. Segmentselektor (engl. *segment selector*)



- je nach Art des Speicherzugriffs selektiert die MMU implizit das passende Segment

Befehlsabruf (engl. *instruction fetch*) $\mapsto$ *code* ☺

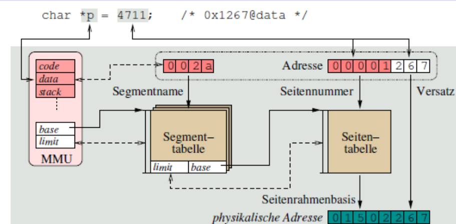
Operandenabruf (engl. *operand fetch*) aus Text-, Daten-, Stapelsegment

- Direktwerte $\mapsto$ *code* ☺
- globale/lokale Daten $\mapsto$ *data* $\equiv$ *stack* ☹

- Programme können weiterhin 1-dimensionale log. Adressen verwenden

Adressumsetzung: implizit segment- und seitenorientiert

Segmentierter seitennummerierter Adressraum (engl. *page-segmented address space*)



- x86

- **Reihenschaltung** von zwei Adressumsetzungseinheiten der MMU:
  Segmenteinheit löst eine segmentierte Adresse auf
    - adressiert und begrenzt die Seitentabelle

  Seiteneinheit generiert die physikalische Adresse
- jeder Prozess hat (mind.) eine Segment- und eine Seitentabelle

Seiten- bzw. Segmentdeskriptor – Abbildung steuernder Verbund

Adressumsetzung basiert auf Deskriptoren der MMU, die für jede Seite bzw. Segment die **Relokations- und Zugriffsdaten** verwalten

- die Basisadresse des Seitenrahmens/Segments im phys. Adressraum
- die Zugriffsrechte des Prozesses (read, write, execute)

Segmente sind von variabler, dynamischer Größe und benötigen daher zusätzliche Verwaltungsdaten ->**Segmentdeskriptor**

- die Segmentlänge, um Segmentverletzung abfangen zu können
- die Expansionsrichtung: Halde "bottom-up", Stapel "top-down"

Deskriptorprogrammierung erfolgt zur Programmlade- und –laufzeit

- bei Erzeugung/Zerstörung schwer- und leichtgewichtiger Prozesse
- bei Anforderung/Freigabe von Arbeitsspeicher

Seiten- bzw. Segmenttabelle – Adressraum beschreibende Datenstruktur

Deskriptoren des Adressraums eines Prozesses sind in einer *Tabelle im Arbeitsspeicher* zusammengefasst

- die Arbeitsmenge von Deskriptoren eines Prozesses wird im Zwischenspeicher gehalten (TLB der MMU)
- Adressraumwechsel als Folge eines Prozesswechels bedeutet:
  - zerstören der Arbeitsmenge (TLB "flush")
  - Tabellenwechsel (Zeiger umsetzen)

Basis-/Längenregister

- beschreibt eine Tabelle und damit exakt einen Prozessadressraum
- bei der Adressumsetzung wird eine Indexprüfung durchgeführt

Virtueller Adressraum

Adressraumabbildung einhergehend mit Ein-/Ausgabe – Integration von Vorder- und Hintergrundspeicher
Abstraktion von Größe und Örtlichkeit des verfügbaren Hauptspeichers
(nicht benötigtes ist ausgelagert. Hintergrundspeicher (z.B. Festplatte), Prozessadressraum z.B. über Rechnernetz verteilt)
Zugriff auf ausgelagerte Programmteile fängt der Prozessor ab: Trap

Seiten- bzw. Segmentdesktiptor (Forts.) – Zusätzliche Attribute
Adressumsetzung unterliegt einer **Steuerung**, die "transparent" für den zugreifenden Prozess die Einlagerung auslöst. present bit (0 ausgelagert, 1 eingelagert)

Seitenfehler: Aufwandsabschätzung von Einzelzugriffen
effektive Zugriffszeit auf den Hauptspeicher

- hängt stark ab von der Seitenfehlerwahrscheinlichkeit p und verhält sich direkt proportional zur Seitenfehlerrate:
  eat=(1-p)*pat + p*pft, $0 \leq p \leq 1$

- angenommen, folgende Systemparameter sind gegeben:
  - 50 ns Zugriffszeit auf den RAM (*physical access time, pat*)
  - 10 ms mittlere Zugriffszeit auf eine Festplatte (*page fault time, pft*)
  - 1 % Wahrscheinlichkeit eines Seitenfehlers ($p = 0,01$)
- dann ergibt sich:
  $eat = 0,99 \cdot 50\,\text{ns} + 0,01 \cdot 10\,\text{ms} = 49,5\,\text{ns} + 10^5\,\text{ns} \approx 0,1\,\text{ms}$

Einzelzugriffe sind im Ausnahmefall um den Faktor 2000 langsamer

mittlere Zugriffszeit

- hängt stark ab von der effektiven Seitenzugriffszeit und der Seitengröße:
  mat = (eat + (sizeof(page)-1)*pat)/sizeof(page)

- angenommen, folgende Systemparameter sind gegeben:
  - Seitengröße von 4 096 Bytes (4 KB)
  - 50 ns Zugriffszeit (*pat*) auf ein Byte im RAM
  - effektive Zugriffszeit (*eat*) wie eben berechnet bzw. abgeschätzt
- dann ergibt sich: $mat = (eat + 4\,095 \cdot 50\,\text{ns})/4\,096 \approx 74,41\,\text{ns}$

Folgezugriffe sind im Ausnahmefall um den Faktor 1,5 langsamer

Seitenüberlagerung „*Considered Harmful*"

Pro und Contra

Virtuelle Adressräume sind ...

vorteilhaft wenn übergroße bzw. gleichzeitig mehrere Programme in Anbetracht zu knappen Hauptspeichers auszuführen sind

ernüchternd wenn der eben durch die Virtualisierung bedingte Mehraufwand zu berücksichtigen ist und sich für ein gegebenes Anwendungsszenario als problematisch bis unakzeptabel erweisen sollte

Seitenfehler sind ...

- nicht wirklich transparent, wenn zeitliche Aspekte relevant sind
  - z.B. im Fall von Echtzeitverarbeitung oder Hochleistungsrechnen
- erst zur Laufzeit ggf. entstehende nichtfunktionale Eigenschaften

3. Zusammenfassung

Adressräume
Ebenen der Abstraktion

- der **physikalische Adressraum** enthält gültige & ungültige Adressen
  - ungültige Adressen: Zugriff führt zum Busfehler
  - gültige Adressen: Zugriff gelingt, ist jedoch zu bedenken...
    - reservierte Adressbereiche ⇝ Schutz
- der **logische Adressraum** enthält gültige Adressen
  - Zugriffsrechte der Prozesse stecken Gültigkeitsbereiche ab
    - Zugriff auf reservierte Adressen führt ggf. zum Schutzfehler
  - Prozesse sind in ihrem Programmadressraum abgeschottet, isoliert
    - Zugriff auf „fremde" freie Adressen führt zum Schutzfehler
- der **virtuelle Adressraum** enthält „flüchtige Adressen"
  - die Bindung der Adressen zu den Speicherzellen ist nicht fest
  - sie variiert phasenweise zwischen Vorder- und Hintergrundspeicher

Adressraumdeskriptoren
Seitennumerierte und segmentierte Adressräume

Abbildung steuernde **Verbunde** zur Erfassung einzelner Adressraumteile
- speichern Attribute von Seiten oder Segmente
  - d.h., Relokations- und Zugriffsdaten, Zugriffsrechte
- bilden Seiten fester Größe auf gleichgroße Seitenrahmen ab
  - seitennummerierter Adressraum
- bilden Segmente variabler Größe auf Byte- oder Seitenfolgen ab
  - segmentierter und ggf. seitennummerierter Adressraum

Abbildungstabellen fassen Deskriptoren (eines Adressraums) zusammen
- die Tabellen liegen im Arbeitsspeicher des Betriebssystems
  - der dafür erforderliche Speicherbedarf kann beträchtlich sein
- im TLB sind Arbeitsmengen von Deskriptoren zwischengespeichert
  - ein *Cache* der MMU, ohne dem Adressumsetzung ineffizient ist

Zugriffsfehler sind intransparente, nicht-funktionale Eigenschaften

Resümee

- **Arbeitsspeicherverwaltung** ist (a) statisch oder (b) dynamisch
  - (a) Arbeitsspeichergebiete maximaler, fester Größe
  - (b) Arbeitsspeicherfragmente variabler Größe
- dabei sind **Fragmente** Teile eines (log./virt.) Prozessadressraums
  - die im Arbeitsspeicher zu platzierenden Teile von Programmtext/-daten
  - für die freier Platz, sog. **Löcher**, im Arbeitsspeicher zu finden ist
- **Prozessadressräume** sind (a) physikalisch, (b) logisch, (c) virtuell
  - (a) lückenhafter, wirklicher Hauptspeicher
  - (b) lückenloser, wirklicher Hauptspeicher
  - (c) lückenloser, scheinbarer Hauptspeicher

  Arbeitsspeicher liegt im Vordergrund (a, b) bzw. Hintergrund (c)

Politiken der Speicherverwaltung	
Platzierungsstrategie	**wohin** ist ein Fragment abzulegen?
Ladestrategie	**wann** ist ein Fragment zu laden?
Ersetzungsstrategie	**welches** Fragment ist ggf. zu verdrängen?

XII.2 Speicherverwaltung: Zuteilungsverfahren

1. Platzierungsstrategie

Freispeicherorganisation

Verwaltung der "Hohlräume" im Arbeitsspeicher

Bitkarte von Fragmenten fester Größe
- eignet sich für seitennummerierte Adressräume
- grobkörnige Vergabe auf Seitenrahmenbasis
- alle freien Fragmente sind gleich gut

verkette Liste von Fragmenten variabler Größe (nicht so gut)

Freispeicher erscheint als Hohlräume/Löcher im RAM, die mit Programmtext/-daten auffüllbar sind. Als Bitkarten/verk. Liste implementierte Löcherliste.

Freispeicher(bit)karte – Erfassung freien Speichers fester Größe
- Fragmente des Arbeitsspeichers ist (mind.) ein Zustand zugeordnet, der durch einen Bitwert (0-belegt, 1-frei) repräsentiert wird. Suche, Belegung und Freigabe (Operationen zur Bitverarbeitung)
- Anforderungen von K Bytes zu erfüllen bedeutet, M Zuteilungseinheiten in Bitkarte zu suchen, deren Zustand "frei" anzeigt:
  - $M = \frac{K + sizeof(unit) - 1}{sizeof(unit)}$
  - mit unit definiert als char[N], d.h. allg. ein Bytefeld darstellend
    - N=1 im Falle segmentierter Adressräume
    - N=sozeof(pages) im Falle seitennummerierter Adressräume
- Abfall: M*sizeof(unit)-K ist nur möglich für sizeof(unit) > 1

Freispeicher(bit)karte: Gemeinkosten

Erfassung freier Fragmente beansprucht mehr oder weniger viel Speicher:

- z.B. ein System mit 1 GB Hauptspeicher und 4 KB Seitengröße
- die dazu passende Bitkarte hat eine Größe von 32 KB:

$$\begin{aligned} sizeof(bit\ map) &= 1\,\text{GB} \div 4\,\text{KB} \div 8\,\text{Bits} \\ &= 2^{30} \div 2^{12} \div 2^{3} = 2^{15}\,\text{Bytes} \end{aligned}$$

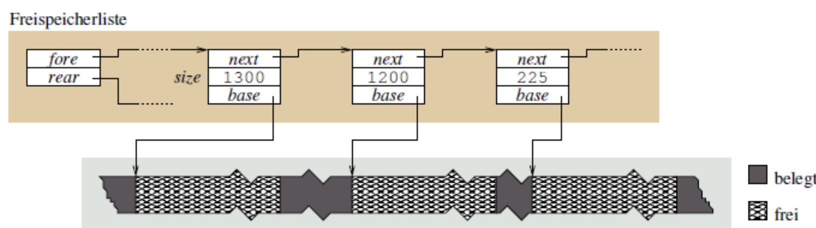
- der Bitkartenumfang variiert mit der Seiten(rahmen)größe
  - gleich gr. Speicher ⇝ ungleich gr. indirekte Speicherkosten
- die MMU erlaubt ggf. eine Feinabstimmung (engl. *tuning*)
  - durch einstell- bzw. programmierbare Seiten(rahmen)größen
  - je feinkörniger die Speicherzuteilung, desto größer die Bitkarte

Freispeicherliste – Erfassung freien Speichers variabler Größe

Löcherliste (engl. hole list) führt Buch über freie Speicherbereiche

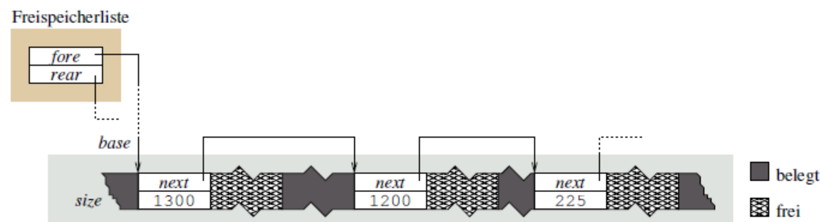
- ein Listenelement hält **Anfangsadresse** und **Länge** eines Bereichs (ein fr. Fragment)
- für die Liste ergeben sich **zwei grundlegende Repräsentationen:**
  - Liste und Löcher sind voneinander getrennt
  - Liste und Löcher sind miteinander vereint
- **strategische Überlegungen** bestimmen die Art der Listenverwaltung

Listenelemente als Löcherdeskriptoren



- die Liste und der Listenkopf liegen im BS-adressraum
- Listenmanipulation erfolgen innerhalb eines log./virt. Adressraums

Listenelemente als Löcher



- nur der Listenkopf liegt im BS-adressraum
- Listenmanipulation müssen ggf. Adressraumgrenzen überschreiten
  - die Listenoperationen laufen im BS-adressraum ab
  - der BS-adressraum ist ein log./virt. Adressraum
  - die Liste ist im phys. Adressraum: ***Adressraumumschaltung***

Löcher der Größe nach sortieren

best-fit verwaltet Löcher nach aufsteigenden Größen (Zielt ist es, den Verschnitt zu minimieren. erzeugt kl. Löcher am Anfang, erhält gr. Löcher am Ende)
worst-fit verwaltet Löcher nach absteigenden Größen (Ziel ist es, den Suchaufwand zu minimieren. Zerstört gr. Löcher am Anfang, macht kl. Löcher am Ende)

Ist die angeforderte Größe kleiner als das gefundene Loch, fällt Verschnitt an, der als verbleibendes Loch in die Liste neu einsortiert werden muss:

- d.h., **die Freispeicherliste ist ggf. zweimal zu durchlaufen**

Verfahrensweisen

Löcher der Größe nach sortieren: Größe = 2er-Potenz

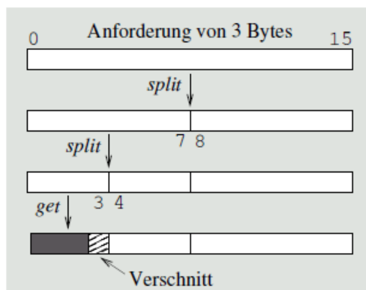
buddy verwaltet Löcher als aufsteigenden Größen

- das kleinste passende Loch $buddy_i$ der Größe 2^i suchen (i, Index in eine Tabelle von Adressen auf Löcher der Größe 2^i)
- $buddy_i$ entsteht durch sukzessive Splittung von $buddy_{j,j>i}$:

  - $2^n = 2*2^{n-1}$
  - zwei gleichgroße Blöcke, die Buddy des jeweils anderen sind
- der ggf. anfallende Verschnitt kann beträchtlich sein (schlimmstenfalls 2^i-1 bei 2^i+1 angeforderten Einheiten)
- ein Kompromiss zwischen best-fit und worst-fit
  - vergleichweise geringer Such- und Aufplitterungsaufwand
  - passt gut zum Laufzeitsystem, das in 2er-Potenzen anfordert

jeder Rest ist als Summe freier Buddies darstellbar, wie auch jede Dezimalzahl als Summe von 2er-Potenzen.

Löcher der Größe nach sortieren: *Buddy*-Verfahren



1. Block 2^4 teilen: $3 < 2^4/2$
2. Block 2^3 teilen: $3 < 2^3/2$
3. Block 2^2 vergeben: $3 \geq 2^2/2$
   - Verschnitt von $2^2 - 3 = 1$ Byte

Ob der Verschnitt als interne Fragmentierung zu verbuchen ist, hängt von der MMU ab.

Verschmelzung bei Speicherfreigabe wird zum "Kinderspiel"
- zwei freie Blöcke lassen sich verschmelzen, wenn sie *Buddies* sind
  - die Adresse von *buddies* unterscheiden sich nu in einer Bitposition
- zwei Blöcke der Größe 2^i sind genau dann *Buddies,* wenn sich ihre Adressen in Bitposition i unterscheiden

Löcher der Adresse nach sortieren

first-fit verwaltet Löcher nach aufsteigenden Adressen (Ziel ist es, den Verwaltungsaufwand zu minimieren. erzeugt kl. Löcher vorne, erhält gr. Löcher am Ende)
next-fit reihum Variante von *first-fit* (Ziel ist es, den Suchaufwand zu minimieren. Nähert sich einer Verteilung "gleichgroßer Löcher" – als Folge nimmt der Suchaufwand ab)

Ist die angeforderte Größe kleiner als das gefundene Loch, fällt Verschnitt an, der jedoch nicht als Restloch in die Liste einsortiert werden muss:
- d.h. **die Freispeicherliste ist nur einmal zu durchlaufen**

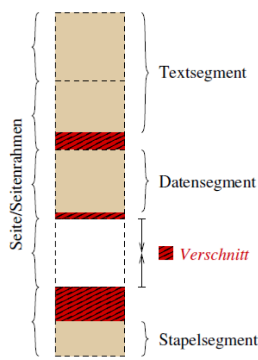
2. Speicherverschnitt

Fragmentierung

Verschnitt durch zuviel zugeteilte/nicht nutzbare Bereiche
Bruchstückbildung: Zerstückelung des Speichers in immer kleinere, verstreut vorliegende Bruchstücke
intern bei seitennummerierten Adressräumen -> Verschwendung
- Speicher wird in Einheitne gleicher, fester Größe vergeben
- der "lokale Verschnitt" ist nutzbar, dürfte aber nicht sein

extern bei segmentierten Adressräumen -> Veröist
- Speicher wird in Einheiten variabler Größe vergeben
- der "globale Verschnitt" ist ggf. nicht mehr zuteilbar
- *Verschmelzung und Kompaktifizierung* schaffen Abhilfe

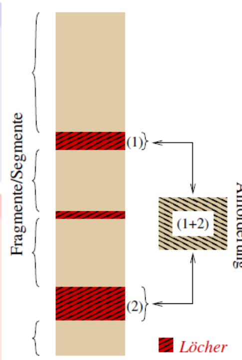
Interne Fragmentierung



Seitennummerierter Adressraum
- abzubildende Programmsegmente sind Vielfaches von Bytes
- der (log./virt.) Prozessadressraum ist aber ein Vielfaches von Seiten
- die jew. letzte Seite der Segmente ist ggf. nicht komplett belegt

Seitenlokaler Verschnitt
- wird vom Programm *logisch* nicht beansprucht
- ist vom Prozess *physisch* jedoch adressierbar
- da eine seitennummerierte MMU Seiten schützt, keine Segmente

Externe Fragmentierung



Segmentierter Adressraum
- die zu platzierenden Fragmente sind Vielfaches von Bytes
- sie werden 1:1 auf Segmente einer MMU abgebildet
- die jew. eine lineare Bytefolge im phys. Adressraum bedingen

Globaler Verschnitt
- die Summe von Löchern ist groß genug für die Speicheranforderung
- die Löcher liegen aber verstreut im phys. Adressraum vor und
- jedes einzelne Loch ist zu klein für die Speicheranforderung

Verschmelzung

Vereinigung eines Lochs mit angrenzenden Löchern

Verschmelzung von Löchern erzeugt ein großes Loch, die Maßnahme...
- beschleunigt Speicherzuteilung, *verringert externe Fragmentierung*
- erfolgt bei Speicherfreigabe oder scheiternder Speichervergabe

Löchervereinigung sieht sich mit vier Situationen konfrontiert, je nach dem, welche relative Lage ein Loch im Arbeitsspeicher hat:

1. zw. zwei zugeteilten Bereichen — keine Vereinigung möglich
2. direkt nach einem Loch — Vereinigung mit Vorgänger
3. direkt vor einem Loch — Vereinigung mit Nachfolger
4. zwischen zwei Löchern — Kombination von 2. und 3.

- der Aufwand variiert z.T. sehr stark mit dem Zuteilungsverfahren

Bezug zum Zuteilungsverfahren

Aufwand ist klein bei buddy, mittel bei first/next-fit, groß bei best/worst-fit

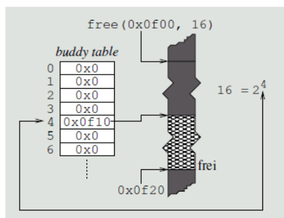
buddy anhand eines Bits der Adresse des zu verschmelzenden Lochs lässt sich leicht feststellen, ob sein Buddy bereits als Loch in der Tabelle verzeichnet ist

fist/next-fit beim Durchlaufen der Freispeicherliste (bei Freigabe) wird jeder Eintrag daraufhin überprüft, ob...
- Adresse plus Größe eines Eintrags gleich der Adresse des verschmelzenden Lochs ist
- Adresse plus Größe eines zu verschmelzenden Lochs der Adresse eines Eintrags entspricht

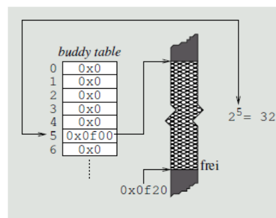
best/worst-fit ähnlich wie bei *first/next-fit*, jedoch kann im Gegensatz dazu nicht davon ausgegangen werden, dass bei einem angrenzenden Loch das Vorgänger-/Nachfolgeelement in der Liste das ggf. andere angrenzende Loch sein muss
- es muss weitergesucht werden

Vereinigung von Löchern: *Buddy*-Verfahren

Zwei Blöcke 2^i sind *Buddies*, wenn sich ihre Adressen in Bitposition i unterscheiden



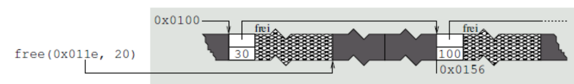
$0f00_{16} = 0000\,1111\,0000\,0000_2$
$0f10_{16} = 0000\,1111\,0001\,0000_2$
$16_{10} = 0000\,0000\,0001\,0000_2$



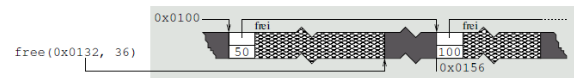
$0f00_{16} = 0000\,1111\,0000\,0000_2$
$0f20_{16} = 0000\,1111\,0010\,0000_2$
$32_{10} = 0000\,0000\,0010\,0000_2$

Vereinigung von Löchern: *first/next-fit*-Verfahren

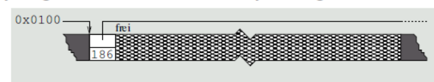
Freizugebender Block ist Nachfolger und/oder Vorgänger



1. das alte 30 Bytes große Loch kann um 20 Bytes vergößert werden



2. das alte 50 Bytes große Loch kann um 36 Bytes vergößert werden
3. das neue 86 Bytes große Loch kann um 100 Bytes vergößert werden



Kompaktifizierung

Auflösung externer Fragmentierung
Vereinigung des globalen Verschnitts

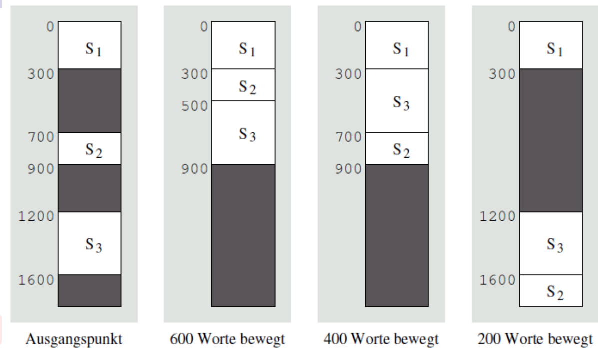
Segmente von (Bytes oder Seitenrahmen) werden so verschoben, dass am Ende ein einziges großes Loch vorhanden ist

- alle in der Freispeicherliste erfassten Löcher werden sukzessive verschmolzen, so dass schließlich nur noch ein Loch übrigbleibt
- durch **Umlagerung** (engl. *swapping*) kompletter Segmente bzw. Adressräume wird der Kopiervorgang „erleichtert"

Relokation der verschobenen Segmente/Seiten(rahmen) ist erforderlich

- das Betriebssystem implementiert logische/virtuelle Adressräume oder
- der Übersetzer generiert positionsunabhängigen Programmtext

- je nach Fragmentierungsgrad ein komplexes **Optimierungsproblem**



3. Zusammenfassung

- Zuteilung von Arbeitsspeicher ist Aufgabe der **Platzierungsstrategie**
  - die Erfassung freier Bereiche hängt u.a. ab vom Adressraummodell
    - (a) Seiten bzw. Seitenrahmen ⇝ Bitkarte oder Löcherliste
    - (b) Segmente ⇝ Löcherliste
  - Folge davon ist **interne** (a) oder **externe** (b) **Fragmentierung**
    - Speicherverschnitt durch zuviel zugeteilte bzw. nicht nutzbare Bereiche
- die **Zuteilungsverfahren** verwalten Löcher nach Größe oder Adresse
  - nach abnehmender Größe *worst-fit*
  - nach ansteigender { Größe *best-fit*, *buddy* / Adresse *first-fit*, *next-fit* }
- angefallener **Speicherverschnitt** ist zu reduzieren oder aufzulösen
  - (a) Verschmelzung von Löchern verringert externe Fragmentierung
    - beschleunigt die Speicherzuteilungsverfahren und
    - lässt die Speicherzuteilung im Mittel häufiger gelingen
  - (b) Kompaktifizierung der Löcher löst externe Fragmentierung auf
    - hinterlässt (im Idealfall) ein großes Loch
    - erfordert aber positionsunabhängige Programme oder log. Adressräume

XII.3 Speichervirtualisierung

1. Ladestrategie

Überblick

Einlagerung von Seiten/Segmenten – Spontanität oder vorauseilender Gehorsam
Einzelanforderung "on demand" -> present bit
(Seiten-/Segmentzugriff führt zum Trap -> page/segment fault. Ergebnis der Ausnahmebehandlung ist die Einlagerung der angeforderten Einheit)
Vorausladen "anticipatory" (**Heuristiken** liefern Hinweise über Zugriffsmuster -> Programmlokalität, Arbeitsmenge. Alternativ auch als **Vorabruf** im Zuge einer Einzelanforderung -> z.B. zur Vermeidung von Folgefehlern)

- ggf. fällt die **Verdrängung** (Ersetzung) von Seiten/Segmenten an

2. Ersetzungsstrategie

Überblick

Verdrängung eingelagerter Fragmente - Platz schaffen zur Einlagerung anderer Fragmente (d.h., Seiten oder Segmente)
Konsequenz zur Durchsetzung der Ladestrategie bei Hauptspeichermangel

- wenn eine Überbelegung des Hauptspeichers vorliegt
- aber auch im Falle (extensiver) externer Fragmentierung

OPT (optimales Verfahren) Verdrängung des Fragments, das am längsten nicht mehr verwendet werden wird – unrealistisch

- erfordert Wissen über die im weiteren Verlauf der Ausführung von Prozessen generierten Speicheradressen, was bedeutet: das Laufzeitverhalten von Prozessen ist im Voraus bekannt. Eingabewerte sind vorherbestimmt. asynchrone Programmunterbrechungen sind vorhersagbar
- bestenfalls ist eine gute Approximation möglich/umsetzbar

Zugriffsfehlerwahrscheinlichkeit und Zugriffsfehlerrate verringern

Praxistaugliche Herangehensweisen
Approximation des optimalen Verfahren greift auf Wissen aus der Vergangenheit/Gegenwart zurück, d.h., ersetzt wird...
FIFO (fist-in, first-out) das zuerst eingelagerte Fragment
- Fragmente entsprechend des Einlagerungszeitpunkts verketten

FLU (least frequently used) das am seltenste genutzte Fragment
- jeden Zugriff auf eingelagerte Fragmente zählen
- Alternative: MFU (most frequently used)

LRU (least recently used) das am längsten nicht mehr genutzte Fragment
- Zeitstempel, Stapeltechniken oder Referenzlisten einsetzen
- bzw. weniger aufwändig durch Referenzbits approximieren

zu ersetzende/verdrängende Fragmente sind vorzugsweise **Seiten**

Verfahrensweisen
Zählverfahren – Auswahlkriterium ist die Häufigkeit von Seitenreferenzen
Zähler-basierte Ansätze führen Buch darüber, wie häufig eine Seite innerhalb einer bestimmten Zeitspanne referenziert worden ist:
- im Seitendeskriptor ist dazu ein Referenzzähler enthalten
- der Zähler wird mit jeder Referenz zu der Seite inkrementiert
- aufwändige Implementierung, bei mäßiger Approximation von OPT

LFU ersetzt die Seiten mit dem kleinsten Zählerwert
- Annahme: aktive Seiten haben große Zählerwerte und weniger aktive bzw. inaktive Seiten haben kleine Zählerwerte
- große Zählerwerte können dann aber auch jene Seiten haben, die z.B. nur in der Initialisierungsphase aktiv gewesen sind

MFU ersetzt die Seiten mit dem größten Zählerwert
- Annahme: kürzlich aktive Seiten haben eher kl. Zählerwerte

Zeitverfahren – Auswahlkriterium ist die Zeitspanne zurückliegender Seitenreferenzen
LRU_{time} verwendet einen Zähler ("logische Uhr") in der CPU, der bei jedem Speicherzugriff erhöht und in den zugehörigen Seitendeskriptor geschrieben wird
- verdrängt die Seite mit dem kleinsten **Zeitstempelwert**

LRU_{stack} nutzt einen Stapel eingelagerter Seiten, aus dem bei jedem Seitenzugriff die betreffende Seiten herausgezogen und wieder oben drauf gelegt wird
- verdrängt die Seite am **Stapelboden**

LRU_{ref} führt Buch über alle zurückliegenden Seitenreferenzen
- verdrängt die Seite mit dem größten **Rückwärtsabstand**

entsprechen "OPT mit Blick in die Vergangenheit", anstatt Zukunft
- gute Approximation von OPT, bei sehr aufwändiger Implementierung

Approximation
LRU:
Alterung von Seiten (page aging) verfolgen
Referenzbit im Seitendeskriptor zeigt an, ob auf die zugehörige Seite zugegriffen wurde: 0->kein Zugriff; 1->Zugriff bzw. Einlagerung
- das Altern eingelagerter Seiten wird periodisch (Zeitgeber) bestimmt
- „kürzlich am wenigstens genutzte“ Seiten haben den Zählerwert 0

Schieberegister Bestimmung des Lebensalters einer Seite
page aging mit Ablauf eines Zeitquamtums, werden die Referenzbits eingelagerter Seiten des laufenden Prozesses in die Schieberegister seiner Seitendesk. übernommen.
Aufwand steigt mit Adressraumgröße des unterbrochenen Prozesses.

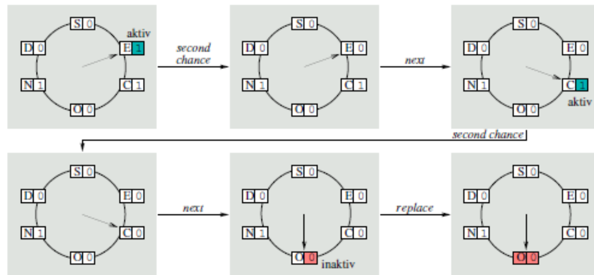
Seite eine zweite Chance einräumen Annahme: Ref. Seite sind vermeintlich aktiv
second chance:

- arbeitet im Grunde nach FIFO, berücksichtigt jedoch zusätzlich noch die Referenzbits der jeweils in Betracht zu ziehenden Seiten
- periodisch werden die Seitendeskriptoren des laufenden Prozesses untersucht

Referenzbit	Aktion	Bedeutung
1	Referenzbit auf 0 setzen	Seite erhält zweite Chance
0	—	Seite ist Ersetzungskandidat

- schlimmstenfalls erfolgt ein Rundumschlag über alle Seiten, wenn die Referenzbits aller betrachteten Seiten (auf 1) gesetzt waren
    - Strategie „entartet“ dann zu FIFO
- unterscheidet nicht lesende und schreibende Seitenzugriffe

LRU: Funktionsweise der *Clock Policy*



E aktiv, Referenzbit auf Null setzen, Seite im Hauptspeicher behalten
C aktiv, Referenzbit auf Null setzen, Seite im Hauptspeicher behalten
O inaktiv, Seite ist ersetzbar, Seitenrahmen für andere Seite nutzen

Schreibzugriffe stärker gewichten als Lesenzugriffe
enhanced second change prüft zusätzlich, ob eine Seite schreibend oder nur lesend referenziert wurde

- Grundlage dafür ist ein Modifikationsbit (schreibezugriff: 1)
- mit Referenzbit gibt es vier Paarungen (R, D)
- kann für jeden Prozess(adressraum) zwei Umläufe erwirken

	Bedeutung	Entscheidung
(0, 0)	ungenutzt	beste Wahl
(0, 1)	beschrieben	keine schlechte Wahl
(1, 0)	kürzlich gelesen	keine gute Wahl
(1, 1)	kürzlich beschrieben	schlechteste Wahl

Freiseitenpuffer
Alternative zur Seitenersetzung, die den Vorabruf von Seiten nutzt

- Steuerung der Seitneüberlagerung über Schwellwerte (low: Seitenrahmen als frei markieren, Seiten ggf. auslagern. high: Seiten ggf. einlagern, Vorausladen)
- die Auswahl greift z.B., auf Referenz-/Modifikationsbits zurück

Freiseiten gelangen in einen Zwischenspeicher (cache)

- die Zuordnung von Seiten zu Seitenrahmen bleibt jedoch erhalten
- vor ihrer Ersetzung doch noch benutzte Seiten werden zurückgeholt
- sog. *Reclaiming* von Seiten durch Prozesse

vergleichweise effizient: die Ersetzungszeit entspricht der Ladezeit

Seitenanforderung

Verteilung von Seitenrahmen auf Prozessadressräume
Prozessen ist mindestens die kritische Masse von Seitenrahmen zur Verfügung zu stellen, um in ihrer Ausführung voranschreiten zu können.

- Rechnerausstattung/-architektur geben „harte“ Begrenzungen vor:
    - Obergrenze: die Größte des Arbeitsspeichers
    - Untergrenze: definiert durch den komplexesten Maschinenbefehl
- innerhalb der Grenzen, ist die Zuordnung…
    - gleichverteilt: in Abh. von Prozessanzahl und/oder
    - größenabh.: bedingt durch den (statischen) Programmumfang
- „weiche“ Begrenzungen ergeben sich z.B. durch die gegenwärtige Systemlast und den gewünschten Grad an Mehrprogrammbetrieb

Einzugsbereiche: Wirkungskreis der Seitenüberlagerung
lokal nur Seitenrahmen des von der Seitenersetzung betroffenen Prozessadressraums nutzen.

- Seitenfehlerrate ist von einem Prozess selbst kontrolliernar (Trap-Eigenschaften)
- statische Zuordnung von Seitenrahmen zum Adressraum

global Seitenrahmen aller Adressräume nutzen: dyn. Zuordnung

- Verdrängung/Ersetzung von Seitenrahmen ist unvorhersehbar und auch nicht bzw. nur schwer reproduzierbar (Interrupt-Eig.)
- zeitl. Ablauf einer Programmausführung ist abh. von der in anderen Adressräumen vorgehenden Aktivitäten

Kombination beider Ansätze möglich: Prozess-/Adressraumklassen

Arbeitsmenge

Seitenflattern – trashing

Kritisches Systemverhalten, wenn die durch Seitenein-/auslagerungen verursachte E/A die gesamten Systemaktivitäten dominiert

- eben erst ausgelagerte Seiten werden sofort wieder eingelagert
- ein mögl. Phänomen der globalen Seitenersetzung
- verschwindet ggf. so plötzlich von allein, wie es aufgetreten ist

Lokalität der Speicherzugriffe **von Prozessen** beachten/bestimmen

Umlagerung (Swapping) **von** Adressräumen bzw. **Arbeitsmengen** (Vermeidung weggelassen)

Menge residenter Seiten (engl. *resident set*)

Beachte: Arbeitsmenge ⊂ „*resident set*"

- die Menge der residenten Seiten ist der im Hauptspeicher gehaltene Anteil eines virtuellen Adressraums eines Prozesses
  - also die Seiten eines Prozesses, die gegenwärtig geladen sind
- diese Menge kann Seiten beinhalten, die der Prozess nicht mehr aktiv in Benutzung hat
  - Seiten, die er früher aber nicht im letzten Zeitfenster referenziert hat
- zu dieser Menge zählen allerdings nicht jene Seiten, die der Prozess demnächst wahrscheinlich in Benutzung haben wird
  - Seiten, die er im nächsten Zeitfenster referenzieren könnte

Arbeitsmengen von Prozessen können sehr viel kleiner sein als die Mengen residenter Seiten eben dieser Prozesse

- in Abhängigkeit von der Güte der Approximation der Arbeitsmenge und der Lokalität des jeweils betrachteten Prozesses
- der quantitave Unterschied kann mehrere Größenordnungen betragen

3. Zusammenfassung

- die **Ladestrategie** sorgt für die Einlagerung von Seiten (Segmenten)
  - auf Anforderung oder im Voraus
- eingelagerte Seiten unterliegen der **Ersetzungsstrategie**
  - Ersetzungsverfahren: OPT, FIFO, LFU, MFU, LRU (*clock*)
    - alternativer Ansatz ist der Freiseitenpuffer
  - Verdrängung arbeitet adressraumlokal oder systemglobal
- **Arbeitsmengen** auseinanderreißen kann zum **Seitenflattern** führen
  - $W(t, r)$ umfasst die aktiven Seiten im Fenster der Breite r zur Zeit t
    - beschreibt damit die zur Zeit t gegebene **Lokalität** eines Prozesses
  - Approximation der Arbeitsmenge mittels Referenzbits und Seitenalter
  - die Berechnung erfolgt auf Basis periodischer Unterbrechungen
- **Umlagerung** bzw. **Vorausladen** immer kompletter Arbeitsmengen
  - Prozesse mit unvollständigen Arbeitsmengen werden ausgesetzt
  - sie unterliegen der langfristigen Einplanung (engl. *long-term scheduling*)

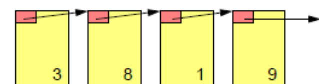
XIII. Dateisystem

1. Medien

Kontinuierliche Speicherung

- Datei wird in Blöcken mit aufsteigenden Blocknummern gespeichert (Nummer des ersten Blocks und Anzahl der Folgeblöcke muss gespeichert werden)
- Vorteile:
  - Zugriff auf alle Blöcke mit minimaler Positionierzeit des Schwenkarms
  - Schneller direkter Zugriff auf bestimmter Dateiposition
  - Einsatz z.B. bei Systemen mit Echtzeitanforderungen
- Probleme: Finden freien Platzes, Fragmentierungsproblem, Größe bei neuen Dateien oft nicht im Voraus bekannt. Erweitern problematisch
- Variation: Unterteilung einer Datei in Folge von Blöcken. Blockfolgen werden kontinuierlich gespeichert. Pro Datei muss erster Block und Länge jedes einzelnen Chunks gespeichert werden
- Problem: Verschnitt innerhalb einer Folge

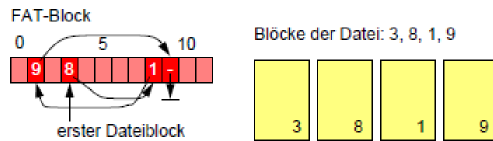
Verkettete Speicherung



z. B. Commodore Systeme (CBM 64 etc.)

- Blöcke einer Datei sind verkettet
- Datei kann wachsen und verlängert werden
- Probleme:
  - Speicherung für Verzweigerung geht von Nutzdaten im Block ab
  - Fehleranfälligkeit: Datei ist nicht restaurierbar, falls einmal Verzeigerung fehlerhaft
  - schlechter direkter Zugriff auf bestimmte Dateiposition
  - häufiges Positionieren des Schreib-, Lesekopfs bei verstreiten Datenblöcken
- Verkettung wird in speziellen Plattenblocks gespeichert

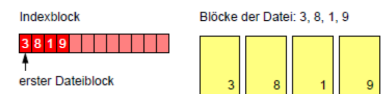
◆ FAT-Ansatz *(FAT: File Allocation Table)*, z. B. MS-DOS, Windows 95



- Vorteile: kompletter Inhalt des Datenblocks ist nutzbar (günstig bei Paging). mehrfache Speicherung der FAT möglich: Einschränkung der Fehleranfälligkeit
- Probleme:
    - mind. ein zusätzlicher Block muss geladen werden (Caching der FAT zur Effizienzsteigerung)
    - FAT enthält Verkettung für alle Dateien: das Laden der FAT-Blöcke lädt auch nicht benötigte Informationen
    - aufwändige Suche nach dem zugehörigen Datenblock bei bekannter Position in der Datei
    - häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

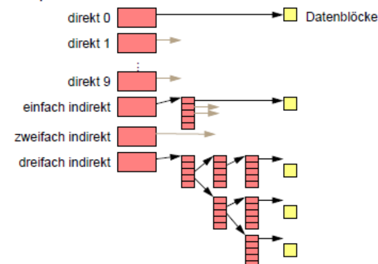
2. Speicherung von Dateien

Indiziertes Speichern



- Spezieller Plattenblock enthält Blocknummern der Datenblocks einer Datei
- Problem: feste Anzahl von Blöcken im Indexblock (Verschnitt bei kleinen Dateien, Erweiterung nötig für große Dateien)

Beispiel UNIX Inode



- Vorteil: Einsatz von mehreren Stufen der Indizierung
    - Inode benötigt sowieso einen Block auf der Platte
    - durch mehrere Stufen der Indizierung auch große Dateien adressierbar
- Nachteil: mehrere Blöcke müssen geladen werden

Freispeicherverwaltung

Prinzipiell ähnlich wie Verwaltung von freiem Hauptspeicher

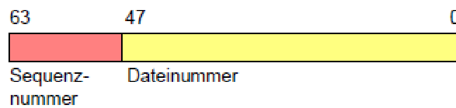
- Bitvektoren zeigen für jeden Block Belegung an
- verkettet Listen repräsentieren freie Blöcke
    - Verkettung kann in den freien Blöcken vorgenommen werden
    - Optimierung: aufeinanderfolgende Blöcke werden nicht einzeln aufgenommen, sondern als Stück verwaltet
    - Optimierung: ein freier Block enthält viele Blocknummern weitere freier Blöcke und evtl. die Blocknummer eines weiteren Blocks mit den Nummern freier Blöcke

Windows NT (NTFS)

- Datei
    - beliebiger Inhalt; für das BS ist der Inhalt transparent
    - Rechte verknüpft mit NT-Benutzern und -Gruppe
    - Datei kann automatisch komprimiert und verschlüsselt werden
    - große Dateien bis zu 2^{64} Bytes lang
    - Hard links: mehrere Einträge derselben Datei in verschiedenen Katalogen möglich
- Dateiinhalt: Sammlung von Streams
    - Stream: einfache, unstrukturierte Folge von Bytes
    - „normaler Inhalt“ = unbekannter Stream (default stream)
    - dynamisch erweiterbar
    - Syntax: dateiname:streamname

Dateiverwaltung

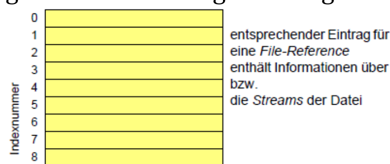
- Basiseinheit „Cluster“
    - 512 Bytes bis 4 Kilobytes (beim Formatieren festgelegt)
    - wird auf eine Menge von hintereinanderfolgenden Blöcken abgebildet
    - logische Vluster-Nummer als Adresse (LCN)
- Basiseinheit „Strom“
    - jede Datei kann mehrere (Daten-)Ströme speichern
    - einer der Ströme wird für die eigentlichen Daten verwendet
    - Dateiname, MS-DOS Dateiname, Zugriffsrechte, Attribute und Zeitstempel werden jeweils in eigenen Datenströmen gespeichert
- File-Reference
    - Bezeichnet eindeutig eine Datei oder einen Katalog



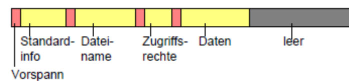
    - Dateinummer ist Index einer globalen Tabelle (MFT: Master File Table)
    - Sequenznummer wird hochgezählt, für jede neue Datei mit gleicher Dateinummer

Master-File Table

- Rückgrat des gesamten Systems
    - große Tabelle mit gleich langen Elementen. kann dyn. erweitert werden



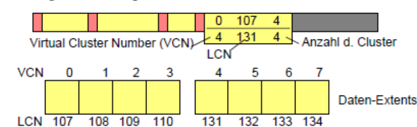
    - Index einer Tabelle ist Teil der File-Reference
- Eintrag für kurze Datei



Streams

- Standard-Information (immer in der MFT)
    - enthält Länge, Standard-Attribute, Zeitstempel, Anzahl der Hard links, Sequenznummer der gültigen File-Reference
    - Dateiname (immer in der MFT). kann mehrfach vorkommen (Hard links)
    - Zugriffsrechte (Security Descriptor)
    - Eigentliche Daten
- mögl. weitere Streams (Attribute)
    - Index, Indexbelegung, Attributliste, Streams mit bel. Daten

Eintrag für eine längere Datei



◆ *Extents* werden außerhalb der MFT in aufeinanderfolgenden Clustern gespeichert
◆ Lokalisierungsinformationen werden in einem eigenen Stream gespeichert

Alle Metadaten werden in Dateien gehalten

Indexnummer		
0	MFT	Feste Dateien in der MFT
1	MFT Kopie (teilweise)	
2	Log File	
3	Volume Information	
4	Attributtabelle	
5	Wurzelkatalog	
6	Clusterbelegungstabelle	
7	Boot File	
8	Bad Cluster File	
...		
16	Benutzerdateien u. -kataloge	
17		
...		

Metadaten

Bedeutung:

- MFT und MFT Kopie: MFT wird sich selnst als Datei erhalten. die Kopie enthält die ersten 16 Einträge der MFT (Fehlertoleranz)
- Log File: enthält protokollierte Änderungen am Dateisystem
- Volume Information: Name, Größe und ähnl. Attribute des Volumens
- Attributtabelle: definiert mögliche Ströme in den Einträgen
- Wurzelkatalog
- Clusterbelegungstabelle: Bitmap für jeden Cluster des Volumens
- Boot File: enthält initiales Programm zum Laden, sowie ersten Cluster der MFT
- Bad Cluster File: enthält alle nicht lesbaren Cluster der Platte NFTS markiert automatisch alle schlechten Cluster und versucht die Daten in einen anderen Cluster zu retten

3. Dateisysteme mit Fehlererholung

Fehlererholung

- NFTS ist ein Journal-File-System
  - Änderungen an der MFT und an Dateien werden protokolliert
  - Konsistenz der Daten und Metadaten kann nach einem Systemausfall durch Abgleich des Protokolls mit den Daten wieder hergestellt werden
- Nachteile:
  - etwas ineffizient
  - nur für Volumes > 400MB geeignet

Dateisysteme mit Fehlererholung

- Metadaten und aktuell genutzte Datenblöcke geöffneter Dateien werden im Hauptspeicher gehalten (Dateisystem-Cache)
  - effizienter Zugriff
  - Konsistenz zw. Cache und Platte muss regelmäßig hergestellt werden
    - synchrone Änderungen: Operation kehrt erst zurück, wenn Änderungen auf der Platte gespeichert wurden
    - asynchrone Änderungen: Änderungen erfolgen nur im Cache, Operation kehrt danach sofort zurück, Synchronisation mit der Platte erfolgt später
- Mögliche Fehlerursachen: Stromausfall, Systemabsturz

Konsistenzprobleme

Fehlerursache & Auswirkungen auf das Dateisystem

- Cache-Inhalte und aktuelle E/A-Operationen gehen verloren
- inkonsistente Metadaten

Reparaturprogramme: Programme wie chkdsk, scandisk oder fsck können inkonsistente Metadatetn reparieren

Nachteil:

- Datenverluste bei Reparatur möglich
- Große Platten bedeuten lange Laufzeiten der Reparaturprogramme

Journaling-File-Systems

Zusätzlich zum schreiben der Daten und Meta-Daten (z.B. Inodes) wird ein Protokoll der Änderungen geführt

- Grundidee: Log-based Recovery bei Datenbanken
- alle Änderungen treten als Teil von Transaktionen auf
- Bsp für Transaktion: Erzeugen, Löschen Erweitern, Verkürzen von Dateien. Dateiattribute verändern. Datei umbenennen
- Protokollieren aller Änderungen am Dateisystem zusätzlich in einer Protokolldatei (Log File)
- beim Bootvorgang wird Protokolldatei mit den aktuellen Änderungen abgeglichen und damit werden Inkonsistenzen vermieden

Protokollierung

- für jeden Einzelvorgang einer Transaktion wird zunächst ein Logeintrag erzeugt und danach die Änderung am Dateisystem vorgenommen
- dabei gilt: der Logeintrag wird immer vor der eignetlichen Änderung auf Platte geschrieben. wurde etwas auf Platte geändert, steht auch der Protokolleintrag dazu auf der Platte

Fehlererholung

- beim Bootvorgang wird überprüft, ob die protokollierten Änderungen vorhanden sind:
  - Transaktion kann wiederholt bzw. abgeschlossen werden (Redo) falls alle Logeinträge vorhanden sind
  - angefangene, aber nicht beendete Transaktionen werden rückgängig gemacht (Undo)

Beispiel: Löschen einer Datei im NTFS

*Vorgänge der Transaktion

- Beginn der Transaktion
- Freigeben der Extents durch Löschen der entsprechenden Bits in der Belegungstabelle
- Freigeben des MFT-Eintrags der Datei
- Löschen des Katalogeintrags der Datei (evtl. Freigeben eines Extents aus dem Index)
- Ende der Transaktion

*Alle Vorgänge werden unter der File-Reference im Log-File protokolliert, danach jeweils durchgeführt.

- Protokolleinträge enthalten Informationen zum Redo und zum Undo

*Log vollständig (Ende der Transakt. wurde protokolliert und steht auf Platte): Redo der Transaktion: alle Operationen werden wiederholt, falls nötig
*Log unvollständig (Ende der Transaktion steht nicht auf Platte): Undo der Transaktion: in umgekehrter Reihenfolge werden alle Operationen rückgängig gemacht

Checkpoints:

- Log-File kann nicht beliebig groß werden
- gelegentlich wird für einen konsistente Zustand auf Platte gesorgt (Checkpoint) und dieser Zustand protokolliert (alle Protokolleinträge von voher können gelöscht werden)
- ähnlich verfährt NTFS, wenn Ende des Log-Files erreicht wird

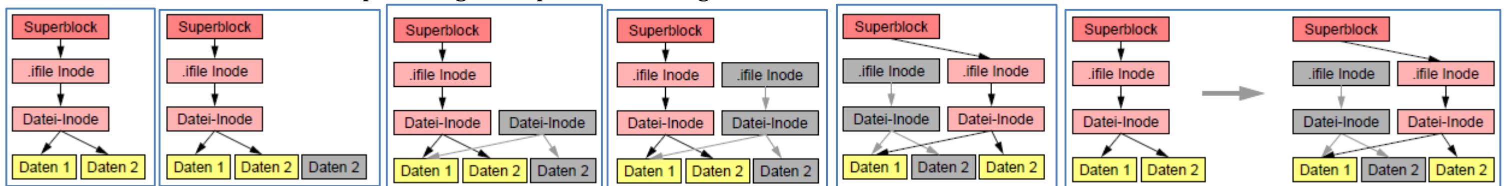
Ergebnis:

- eine Transaktion ist entweder vollständig durchgeführt oder gar nicht
- Benutzer kann ebenfalls Transaktionen über mehrere dateizugriffe definieren, wenn diese ebenfalls im Log erfasst werden
- keine inkonsistenten Metadaten möglich
- Hochfahren eines abgestürzten Systems benötigt nur den relativ kurzen Durchgang durch das Log-File (Alternative chkdsk benötigt viel Zeit auf gr. Pl.)

Nachteil: ineffizienter, da zusätzliches Log-File geschrieben wird
Bsp: NTFS, EXT3, EXT4, ReiserFS

Copy-on-Write-/Log-Structured-File-System

- Alternatives Konzept zur Realisierung von atomaren Änderungen
- Alle Änderungen im Dateisystem erfolgen auf Kopien
  - Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben
  - Bsp: LinLogFS: Superblock einziger nicht ersetzter Block



- Vorteile: Datenkonsistenz bei Systemausfällen (ein atomare Änderung macht alle zusammengehörigen Änderungen sichtbar). Schnappschüsse/Checkpoints einfach realisierbar
- Nachteil: Gesamtperformanz geringer
- Unterschied zw. Copy-on-Write- und Log-Structured-File-Systems
  - L-S-F-Systems schreiben kontinuierlich an's Ende des belegten Plattenspeichers und geben vorne die Blöcke wieder frei (gute Schreibeffizienz. Annahme: Lesen kann primär im Cache erfolgen)
- Bsp:
  - Log-Structured-File-Systems: LinLogFS, BSD LFS, AIX XFS
  - Copy-on-Write: Btrfs (Oracle)

Fehlerhaft Plattenblöcke

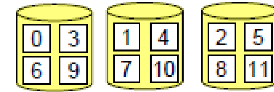
- Blöcke, die beim Lesen Fehlermeldungen erzeugen (z.B. Prüfsummenfehler)
- Hardwarelösung:
  - Platte und Plattencontroller bemerken selbst fehlerhafte Blöcke und maskieren diese aus
  - Zugriff auf den Block wird vom Controller automatisch auf einen „gesunden" Block umgeleitet
- Softwarelösung: File-System bemerkt fehlerhafte Blöcke und markiert diese auch als belegt

4. Datensicherung

- Schurz vor dem Totalausfall von Platten (z.B. durch Head-Crash oder andere Fehler)
- Sichern der Daten auf Tertiärspeicher (Bänder, WORM-Speicherplatten)
- Sichern großer Datenbestände: Total-Backups benötigen lange Zeit. Inkrementielle Backups sichern nur Änderungen ab einem bestimmten Zeitpunkt. Mischen von Total-Backups mit inkr. Backups

Einsatz mehrerer (redundanter) Platten

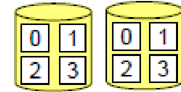
Gestreifte Platten (*Striping*, **RAID0**)



- Daten werden über mehrere Platten gespeichert
- Datentransfers sind nun schneller, da mehrere Platten gleichzeitig angesprochen

- Nachteil: keinerlei Datensicherung: Ausfall einer Platte lässt Gesamtsys. ausfallen

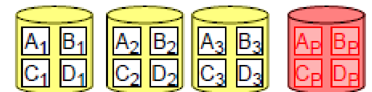
Gespiegelte Platten (*Mirroring*; **RAID1**)



- Daten werden auf zwei Platten gl. gespeichert
- Implementierung durch Software oder Hardware
- eine Platte kann ausfallen
- schnelleres Lesen (da zwei Platten unabh. voneinandern beauftragt werden können)

- Nachteil: doppelter Speicherbedarf
- wird langsameres Schreiben durch Warten auf zwei Plattentransfers
- Verknüpfung von RAID 0 und 1 möglich (**RAID 0+1**)

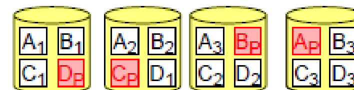
Paritätsplatte (**RAID 4**)



- Daten werden über mehrere Platten gespeichert, eine Platte enthält Parität
- Paritätsblock enthält byteweise XOR-Verknüpfungen von den zugehörigen Blöcken aus den anderen Streifen
- eine Platte kann ausfallen
- schnelles Lesen
- prinzipiell bel. Plattenanzahl (ab 3)

- Nachteil:
  - jeder Schreibvorgang erfodert auch das Schreiben des Paritätsblocks
  - Erzeugung des Paritätsblocks durch Speichern des vorherigen Blockinhalts möglich: $P_{neu} = P_{alt} \oplus B_{alt} \oplus B_{neu}$ (P=Parity, B=Block)
  - Schreiben eines kompletten Streifens benötigt nur einmaliges Schreiben des Paritätsblocks
  - Paritätsplatte ist hoch belastet

Verstreuter Paritätsblock (**RAID5**)



- Paritätsblock wird über alle Platten verstreut
- zusätzliche Belastung durch Schreiben des Paritätsblocks wird auf alle Platten verteilt
- heute gängigstes Verfahren redundanter Platten
- Vor- und Nachteile wie RAID4