Systemprogrammierung

C - Kurzeinführung

Struktur eines C - Programms

```
<globale Var-Definitionen>

<Funktionen>

int main(int argc, char *argv[]) {
    <Var-Definitionen>
    <Anwesiungen>
}
```

Datentypen und Variablen

- Datentyp = Menge von Werten + Menge von Operationen
- Literal, Konstante, Variablen und Funktion haben einen DT

Primitive DT in C

- Ganzzahlen: `char, short, int, long, long long`
- Fließkommazahlen: `float, double, long double`
- Leerer DT: `void`
- (Boolescher DT: `bool`) eigentlicht int, 0 = false, !0 = true

Typ-Modifier: `signed, unsigned, const`

Anweisungen

- Kann Ausdruck, Block oder Kontrollstruktur sein
- Ausdruck = Kombination von Operatoren, Werten und Vars

Operatoren

- (spezielle) Zuweisungsoperatoren
- (bitweise) logische Operatoren
- arithmetische Operatoren
- Vergleichsoperatoren
- logische Shiftoperatoren

Funktionen

- Schnittstelle = Ergebnistyp, Name, (formale) Parameter
- verringern Komplexität
- erlauben Wiederverwendung

- verbergen Implementierungsdetails (Black-Box-Prinzip)

Funktionsaufruf

- Format: `<Funktionsname> ( <Ausdruck>, <Ausdruck>, ...)`
- Ausdrücke in Param-Liste werden ausgewertet BEVOR in die Funktion gesprungen wird
- keine Ausführungsreihenfolge für Parameter definiert

Regeln

- Funktionen werden global definiert
- Funktionen müssen vor erstem Aufruf deklariert werden
- Funktionsdeklaration: `<Typ> <Name> ( <Liste formaler Parameter> );`

Parameterübergabe an Funktionen

- findet in C durch `call by value` statt
- d.h. Funktion erhält nur eine Kopie des übergebenen Params

C - Präprozessor

- bearbeitet C-Quelle bevor sie dem C-Compiler übergeben wird
- Anweisungen werden durch `#` gekennzeichnet (unabh. Syntax, kein `;`)
- wichtigste Funktionen: `#define` und `#include`

Programmstruktur und Module

- Modulierung: Gliederung von Datenbeständen und den Funktionen die darauf operieren
- Teile eines C-Programms können auf meherer .c Dateien verteilt werden
- Zwischenergebnis einer Übersetzung wird in .o Dateien gespeichert
- Komando `cc -o` kann mehrere .c und .o Dateien übersetzen und binden
- Header Dateien enthalten Deklarationen, keine Implementierungen

Gültigkeitsbereiche

- Stufen: Global (Programm), Global (Modul), Lokal (Funktion), Lokal (Block)
- "lokal überdeckt global"
- Einschränkung des Zugriffs auf das Modul durch `static`
- Funktionen sind generell global
- Hilfsfunktionen, die nicht teil der Schnittstelle sind, sollten immer `static` sein

Zeiger(-variablen)

- enthält als Wert die Adresse einer anderen Variablen
- übergeben von Zeigern ermöglicht `call by reference`
- Definition: `<Typ> *<Name>;`

Beispiele

```
int x = 5;
int *ip;
```

```
int y;
ip = &x;    // ip = Adresse von x
y = *ip;    // y = Inhalt auf den ip Zeigt = 5
```

Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
- `int (*fptr) (int, char *)`

Felder

- fasst eine Reihe von Daten des selben Typs zusammen
- seit C99 sind zur Laufzeit berechnete Werte als Größenangabe gültig
- Char Felder werden mit `\0` abgeschlossen

Zeiger und Felder

- ein Feldname ist konstanter Zeiger auf das erste Element des Feldes
- Beispiel: `int array[5]; int *ip = array;`
- In C lässt sich jede Feldoperation als Zeigeroperation abbilden

```
int array[N];
int *ip = array;

array == &array[0] == ip == &ip[0]
*array == array[0] == *ip == ip[0]
*(array + i) == array[i] == *(ip + i) == ip[i]
array++ != ip++ // Array ist konstant!
```

Felder als Funktionsparameter

- ganze felder können nicht by-value übergeben werden
- nur der zeiger auf den beginn des feldes wird by value übergeben
- information über die größe des Feldes geht verloren

Zeiger, Felder und Zeichenketten

- subtrahiert man zwei Zeiger voneinander, erhält man die Anzahl der Elemente zwischen den Beiden
- der compiler merkt sich wo konstante string literale gespeichert wurden und verwendet diese wieder
- oft werden zeichenketten in nicht-modifizerbare speicherbereiche geschrieben -> verändern über deref. Zeiger schlägt fehl

Dynamische Speicherverwaltung

- speicherbereich kann dynamisch vom OS angefordert werden mit `void *malloc(size_t size)`
- mangelnder Speicherplatz führt zur NullPointerException
- angeforderter Speicher wird mit `void free(void *ptr)` freigegeben

```
int *feld;
feld = malloc(groesse * sizeof(*feld));
if (feld == NULL) {
    perror("malloc feld");
    exit(1);
}
```

Strukturen

- werden mit `struct <Name> {...}` initialisiert
- Übergabe als Funktionsparameter erfolgt als `call by value` -> Speicherintensiv
- `foo->bar` === `(*foo).bar`

Standart Ein-/Ausgabe

- jedes C Programm erhält bei Start 3 E-/A-Kanäle:
  - stdin: mit Tastatur verbunden, EOF durch `CTRL-D`
    - auf Datei umlenkbar `./meinprog < meinedatei`
  - stdout: mit Terminal verbunden, auch umlenkbar `./meinprog > meindatei`
  - stderr: Ausgabekanal für Fehlermeldungen, mit Terminal verbunden
- Pipes verbinden stdout von Prog1 mit stdin von Prog2
- Ein Programm kann selbst weitere E-/A-Kanäle öffnen
  - Einbinden von `stdio.h`
  - Funktion `FILE *fopen(char *name, char *mode);`
    - name = pfad zur Datei
    - mode: r, w, a, rw
    - Fehlerfall liefert `NULL` Pointer

Zeichenweises Lesen und Schreiben

- Einzelnes Zeichen lesen: `int getchar()`, `int getc(FILE *fp)`
- Einzelnes Zeichen schreiben: `int putchar(int c)`, `int putc(int c, FILE *fp)`

Zeilenweises Lesen und Schreiben

- Lesen: `char *fgets(char *s, int n, FILE *fp)`
  - liest bis n-1 Zeichen gelesen wurde oder `\n` oder `EOF`
  - `s` wird mit `\0` abgeschlossen, `\n` bleibt erhalten
  - für `fp` kann `stdin` angegeben werden
- Schreiben: `int fputs(char *s, FILE *fp)`

Ausführung von Programmen

Übersetzen und Binden

1. Schritt: Präprozessor
   - fügt includes ein
   - expandiert Makros

    - entfernt Makro-abhängige Code Abschnitte
    - Zwischenergebnis kann mit `cc -P datei.c` als `datei.i` gespeichert werden
2. Schritt: Compiler
    - übersetzt C in Assembler falls gefragt, sonst Übergang zu 3.
    - Zwischenergebnis kann mit `cc -S datei.c` als `datei.s` gespeichert werden
3. Schritt: Assemblieren
    - Erzeugt Maschinencode (Objekt-Datei)
    - standardisiertes Objekt-Dateiformat: ELF
        - Maschinencode
        - Informationen über static Variablen
        - Symboltabelle: wo stehen globale Vars und Funcs
        - Relokierungsinformationen. wo werden "nicht gefundenene" Vars/Funcs referenziert
    - Zwischenergebnis mit `cc -c datei.c` als `datei.o` speichern
4. Schritt: Binden
    - Programm `ld`: erzeugt executable
    - bindet Objekt-Dateien zusammen
    - alle nicht erfüllten Referenzen werden gebunden (Relokation)
    - nach fehlenden Funktionen wird in Bibs gesucht

Programme und Prozesse

- Programm: Folge von Anweisungen
- Prozess: Programm das sich in Ausführung befindet + seine Daten (abstrakt)
- Prozessinstanz: Ausführungsumgebung für ein Programm (konkret)

Speicherorganisation eines Programms (ELF-Format)

- ELF Header: Verwaltungsinformationen und Identifikator
- text: Programmcode
- initialized data: initialisierte globale und static Variablen
- symbol table: Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen

Speicherorganisation eines Prozesses

- bss: nicht initialisierte globale und static Variablen (mit 0 vorbelegt)
- heap: dynamische Erweiterung des bss-Segment (malloc(3))
- stack: lokale Vars, Funktionsparameter, Speicher für Registerinhalte, dynamisch Erweiterbar

Prozesserzeugung, Operationen auf Prozessen

Prozesszustände

1. Erzeugt
2. Bereit
3. Laufend (Unterbrechung -> 2)
4. Beendet/Blockiert

Prozesserzeugung

- Duplizieren des gerade laufenden Prozesses mittels `fork();`
- `fork` gibt PID zurück (0 = Childprozess)
- "perfekte Kopie"

Prozessausführung

- `execve(const char *path, char *const argv[], char *const envp[]);`
- Inhalt vom Parent Prozess wird komplett gelöscht
- Inhalte vom neuen Programm werden in den Prozess geladen

Einleitung

Deutung des Begriffs Betriebssystem

Lehrbuch Definitionen

```
Eine Softwareschicht die alle Teile des Systems verwaltet und dem Benutzer eine
Schnittstelle anbietet die einfacher zu verstehen und programmieren ist.
```

```
Eine Menge von Programmen, die die Ausführung von Benutzerprogrammen und die
Benutzung von Betriebsmitteln steuern.
```

```
Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften
der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen
Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und
überwachen.
```

Zusammenfassung

Betriebssystem ist eine Menge von Programmen, die

- Programme, Anwendungen oder Benutzern assistieren sollen
- die Ausführung von Programmen überwachen und steuern
- den Rechner für eine Anwendungsklasse betreiben
- eine abstrakte Maschine implementieren
- die Betriebsmittel verwalten
- definiert sich über Funktionen, nicht Architektur

Rechnerorganisation

Schichtenstruktur und semantische Lücke, Mehrebenenmaschinen

- Semantische Lücke = Verschiedenheit zwischen Quell(hohe Abstraktion)- und Zielsprache(niedrige Abstr.)
- Abstraktionshierarchie

    - Modellsprache: Formel
    - Programmiersprache: Komplexschritte
    - Assembliersprache: Elementarschritte
    - Maschinensprache: Programmtext/Bits

Hierarchie virtueller Maschinen

- jede Ebene wird durch einen spezifischen (Software-)Prozessor implementiert
    - Kompilierer transformiert Programme in Quellsprache zu Zielsprache
    - Interpreter kann Hard-, Firm- oder Softwareprozessor sein der Programme direkt ausführt

Schichtenfolge

```
5. problemorientierte Programmiersprachenebene  Übersetzung (Kompilierer)
4. Assembliersprachenebene                      Übersetzung (Assemblierer),
Bindung
3. Maschinensprachenebene                       Teilinterpretation
(Betriebssystem)
2. Befehlssatzebene                             Interpretation (Mikroprogramm)
1. Mikroarchitekturebene                        Ausführung
0. digitale Logikebene
```

- Schichten von 0-2 sind meist Hardware, können aber simuliert werden
- Schichten von 3-5 sind virtuelle Maschinen

Entvirtualisierung

- 5 -> 4: Ein Befehl zu n Assembliersprachenbefehlen
- 4 -> 3: Quellmodul wird in Objektmodul umgewandelt, mit Bibliotheken zusammengengebunden
- 3 -> 2: Ausführen durch auslösung einer trap oder durchreichen
- 2 -> 1: Fetch-execute-cycle der CPU

Abweichung vom normalen Programmablauf

- Exception die die unterbrechung oder Abruch der Ausführung bedeutet
- Wiederaufnahme der Ausführung oder Termination, je nach schwere des Fehlers
- z.B. Java bietet Konstrukte zum Umgang mit Ausnahmen

Sonderfallbehandlung

- Ausgangssituation: Maschinenbefehl [3] und Operation und Operanden [2]
- Maschinenprogramm "benutzt" Befehlssatzebene
- Bei Interpretation tritt Ausnahmesituation auf -> raise -> trap/interrupt -> Ausnahmebehandlung (Betriebssystem)

Signalbehandlung

- Maschinenprogramm macht Systemaufruf -> Systemfunktion wird ausgeführt (OS)
- dabei tritt Ausnahmesituation auf -> raise -> signal -> Signalbehandlung [3]

Anhang zur Virtualisierung

- Ebene [2,3] werden durch reale/virtuelle Maschinen (Interpreter) implementiert
    - virtuell: spezifische Software
        - [2] Virtualisierungssystem (totale/partielle interpr.)
        - [3] Betriebssystem (partielle interpr.)
    - total: Als Emulator "complete software interpreter machine"
    - partiell: Als Virtual machine monitor (VMM), fängt sensitive Befehle ab
        - Typ 1: läuft auf "nackter" Wirtsmaschine, auf keinem Betriebssystem [2']
        - Typ 2: läuft auf dem Wirtsbetriebssystem [3']

Sensitive Befehle

- direkte Ausführung nicht durch VMM tolerierbar
- priviligierte Befehle im unpriviligierten Modus/ Befehle mit kritischen Seiteneffekten

Anforderungen an eine virtuelle Befehlssatzebene

- äquivalente Ausführung unpriviligierter Befehle
- Schutz von Systemmodus-Programmen
- Abfangen sensitiver Befehle

Maschinenprogramme

Hybridsoftware

- Maschinenprogramme bilden hybride Schicht (Instruktionen für CPU und für OS)
- OS interpretiert Systemaufrufe partiell, im Ausnahmefall auch Maschinenbefehle
- CPU interpretiert Maschinenbefehle
- OS ist quasi ein Programm der Befehlssatzebene[2], es implementiert die Maschinenprogrammebene [3]
    - es ist kein Maschinenprogramm
    - es ruft nicht seine eigenen Systemaufrufe auf
    - es unterbricht sich nie von selbst
- OS lässt Unterbrechen von Programmen durch Traps/Interrupts zu

Programmhierarchie

- Maschinenprogramme sind ohne Übersetzung vom Prozessor ausführbar
    - müssen durch Übersetzung generiert werden
    - sind technisch als Lademodul repräsentiert
    - werden vom OS geladen, verarbeitet und entsorgt
    - werden in Hochsprachen entwickelt
- Maschinenprogramm wird aufgeteilt in
    - Anwendungsprogrammebene
    - Lauftzeitsystemebe
- Ausführungsplattform = Betriebssystem + CPU
- CPU kann im user mode oder system mode laufen
- Maschinenprog läuft im User Mode -> Aufruf priviligierter Aktionen führen zum Systemaufruf (Moduswechsel)

Beispiel

```
void echo() {
    char c;
    while (read(0, &c, 1 ) == 1) {
        write(1, &c, 1);
    }
}
```

- read & write sind Systemaufrufe -> laufen in anderem Addressraum und system mode
  - ausgelöst durch `int $0x80`
  - operationscode in %eax
  - parameter in %ebx, %ecx und %edx
  - resultate in %eax zurück
- Parameter werden mittels Stack übergeben
- read & write sind zunächst unaufgelöste Referenzen
- Binder wird die Referenz später durch binden der libc.c auflösen
- Systemaufrufzuteiler (system call dispatcher) empfängt systemaufruf und wertet operationscode aus
- Systenaufruf ist adressraumübergreifender Prozeduraufruf

Organisationsprinzipien

- Domänenübergreifende Aufrufhierarchie
  - Anwendungsprogramm (z.B. `echo`): `main()` -> Prozeduraufruf
  - Laufzeitsystem (libc): `write()` -> Systemaufruf (Trap)
  - Betriebssystem (Linux): `system_call` -> Prozeduraufruf -> `sys_write()`

Laufzeitkontext

- Kontextwechsel müssen Konsistenz des Prozessorstatus wahren
  - flüchtige Register: unbeständiger Inhalt, kann geändert werden (caller saved)
  - nichtflüchtige Register: beständiger Inhalt, unverändert (callee saved)
- Aktivierungsblock
  - lokale Variablen
  - nichtflüchtige Registerinhalte
  - lokale basis
  - Rücksprungsadresse
  - tats. Parameter
- Prozessorregister der Unterprogrammverwaltung
  - stack pointer: Oberseite des Stapels
  - frame pointer: lokale Basis des umgebenden Unterprogramms
- Merkmale der Befehlssatzebene[2]
  - Expansionsrichtung des Stacks
  - der stack pointer (letztes abgelegtes oder nächster freier platz)
  - eine Adresse auf eine Speicherzelle im Stapel

Systemaufruf mittels Unterbrechungsbefehl (interrupt/trap)

- CPU durchläuft Unterbrechungszyklus
    - minimalen CPU Status sichern
    - Befehlszählerregister vom Ausnahmevektor laden
    - priviligierten Modus aktivieren
- sozusagen Ausnahme ohne Ausnahmesituation
- sind "teuer" und bieten keine konsistente Statussicherung

Systemaufrufbeschleunigung

- `syscall` und `sysret`
- Sicherung von `%ecx` (Stack Pointer), `%edx` (Programm Counter) und `%ebp`
- verringert Aufruflatenz signifikant

Betriebssystemmaschine

Teilinterpretation

- ausführende Instanz ist immer die CPU [2]
- reine Ebene [3] Befehle werden wargenommen und an das OS "hochgereicht"
- das OS bildet eine logische Maschine die Befehle von der CPU entgegennimmt
- während der Interpretation durch das OS können weitere Ausnahmen auftreten (reentrant)

Programmunterbrechung

- Trap
    - Abfangen einer Ausnahme mit interner Ursache
    - Synchrone Ausnahme: z.B. unbekannter Befehl, Adressraumverletzung
    - reproduzierbar
- Interrupt
    - Unterbrechung für eine Ausnahme externer Ursache
    - Asynchrone Ausnahme: I/O-Operation
- Behandlung ist immer Prozessorabhängig

Laufzeitkontext

- Unterbrechungen sind nicht-lokale Sprünge -> Wechsel des Laufzeitkontext
- CPU führt Zustandssicherung durch
    - minimal: statusregister und Befehlszähler (PC)
    - maximal: kompletter Registersatz

Nichtsequentialität

- asynchronität von Interrupts birgt die Gefahr von "race conditions"
- kritische Abschnitte die nicht unterbrochen werden dürfen müssen entsprechend definiert werden
    - inline assembler
    - basic block: `enter(INTERRUPT_LOCK);` und `leave(INTERRUPT_LOCK);`
    - interrupts deaktivieren [asm] `cli` und aktivieren `sti`

Betriebssystemkonzepte

Prozesse

Prozessorvirtualisierung

- alle Prozesse sind Implementierungen einer abstrakten Prozessdefinition
- ermöglicht simultane Programmabläufe im Multiplexverfahren
- Prozess einlasten = dispatching
- Prozessablauf planen = scheduling
- meist kommt Zeitteilverfahren zum Einsatz (Prozesse merken nichts von Virtualisierung)
- Prozessablauf benötigt Betriebsmittel
    - wiederverwendbar: CPU, RAM, Ein-/Ausgabe, Variablen
    - konsumierbar: Signale, Datenstrom

Prozessplanung

- Betriebsmittel müssen auf simultan laufende Prozesse verteilt werden
- Scheduler erstellt eine "Strategie" und Reihenfolge für Prozesse
- `scheduling`: Übergang in Zustand "bereit" oder "blockiert"
- `dispatching`: Übergang in den Zustand "laufend"
- Fehlt einem Prozessor an Betriebsmitteln nurnoch die CPU, kommt er auf die `ready list`
- Elemente auf der liste sind Prozesskontrollblöcke

Synchronisation

- ist die Koordination der Kooperation und Konkurrenz zwischen Prozessen
- unteilbare Betriebsmittel: multilateral
    - kann sich blockierend auf Prozesse auswirken
    - bei nichtblockierung kann Zustandsänderung fehlschlagen (Transactions)
- konsumierbare Betriebsmittel: unilateral
    - logisch: z.B. Producer-Consumer Rollen
    - bedingt: Fallunterscheidung
- Prozesse die gleichzeitig auftreten interagierten zwingend -> Interferenz

Semaphor

- fundamentale Primitiven für Erwerb/Abgabe von Betriebsmittel
- P (prolaag/down/wait/aquire)
    - verringert Semaphor `s` um 1
    - wenn `s` == 0 dann blockiert der Prozess -> Warteliste
- V (verhoog/up/signal/release)
    - erhöht `s` um 1
    - stellt einen Prozess von der Warteliste bereit
- beides sind logisch unteilbare Operationen

Implementierungsaspekte

- Prozess: Ausführung eines Programms, isolierter Adressraum
    - unter der Maschinenprogebene: im Betriebssystem (kernel thread) oder partiell virtualisiert
    - auf der Maschinenprogebene: im Laufzeit oder Anwendungssystem (user thread)
- "Faden": sequentieller Prozess

- der Prozessor weiß nicht dass er partiell virtualisiert wird
- Prozessor -> abstrahiert von n _kernel threads _ -> abstrahiert von n *user threads*
- Prozesswechsel
    - *feather weight*: selber Kern Adressraum
    - *light weight*: selber Anwendungs- und Kernadressraum
    - *heavy weight*: anderer Anwendungsadressraum, selber Kernadressraum
- Prozesskontrollblock (PCB) enthält alle relevanten Attribute eines Prozesses
- OS verwaltet pro CPU einen Prozesszeiger
- nach außen werden Prozessinkarnationen über PIDs identifiziert

Speicher

Speicherorganisation

- Speicherpyramide
    - Register (primär, verwaltet vom Kompilierer/Maschinenprog)
    - Zwischenspeicher (primär, von der CPU)
    - Notizblockspeicher (primär, vom Maschinenprog)
    - Hauptspeicher (primär, virtueller Speicher, vom OS)
    - Ablagespeicher (sekundär, teil-virtuell, vom OS)
    - Archivspeicher (tertiär, vom OS)
- Referenzfolge
    - laufender Prozess generiert Adressen auf Haupt-/Arbeitsspeicher
    - initial minimaler Vorrat an Adressen ist dynamisch erweiterbar
- Adressraumlehre
    - realer Adressraum: durch die CPU (64 bit) definierter Wertevorrat [0, 2^64-1], enthält Lücken
    - logischer Adressraum: im Programm definierter Wertevorrat, keine Lücken
    - virtueller Adressraum: logischer Adressraum, nicht jedes addressierbare Datum existiert zur Laufzeit im Hauaptspeicher -> trap -> OS lagert das Datum ein -> try again

Speicherverwaltung

- zentrale Aufgabe ist die Speicherzuteilung an einen Prozess zu überwachen
- zusätzl. Speichervirtualisierung, wenn Hauptspeicher nicht ausreicht
- Ladestrategie: wann muss ein Datum im Hauptspeicher liegen?
- Ersetzungsstrategie: welches Datum im Hauptspeicher ist ersetzbar?
- Platzierungsstrategie: wo im Hauptspeicher ist noch Platz?
    - segmentierte Speicherverwaltung: Verschiedene Granularität, Adressen und Länge
    - gekachelte Speicherverwaltung: Nur unterschiedliche Adressen
- Maschinenprog verwaltet seinen Adressraum selbst über Hochsprachenkonstrukte
- OS verwaltet den gesamten Haupt-/Arbeitsspeicher nach systemorientierten Kriterien

Speicherzuteilung

- Ziel ist die Minimierung von Verschnitt (waste)
- Granularität der Speicherzuteilung je nach Problem
    - Wort: Einheit für Hauptspeicher (n Byte groß)
    - Zelle: Einheit für Halden- und Hauptspeicher (16/32B groß)
    - Kachel: Einheit für Arbeitsspeicher (typisch 4 KiB groß)

    - Adressen entsprechen einem vielfachen der Einheitsgröße
- "Löcherliste": Liste freier Speicherbereiche
- Sortierkriterien der Löcherliste:
    - best-fit: kleinster Platz zuerst, Verschnitt minimieren
    - worst-fit: größter Platz zuerst, Suchaufwand minimieren

Speicherrückgabe

- eigentlich kann das Betriebssystem ungenutzten, virtuellen Speicher selbst rückgewinnen
- Virtueller Speicher kann nich vorrausgesetzt werden, also mittels `free()` freigeben

Speichervirtualisierung

- ermöglicht laufende Prozesse deren Maschinenprog nicht komplett im Hauptspeicher liegt
- Einheit Seite wird im logischen/virtuellen Adressraum verwendet = eine Kachel
- Seiten werden mittels Abbildungstabelle (page table) auf Kacheln abgebildet
- Jede Stelle im virtuellen Speicher lässt sich mittels Seitennummer `p` und Offset `o` referenziert werden

Adressbindung

Speicheradressen

- Adressen könnnen real, logisch oder virtueller Natur sein
- Adressen gelten immer nur für einen Namensraum
- Namensraum wird mittels Seitenadressierung und Seitentabelle organisiert
- Segmentierung mittels Segmenttabelle (zweidimensionaler Adressraum)
- Segmentierung (segmentation) und Seitenadressierung (paging) können kombiniert werden

Namensbindung- und Auflösung

- Verzeichniseintrag speichert Abbildung von einem Namen auf eine Informationsstruktur (Datei)
- Eine Informationsstruktur
    - hat mehrere Attribute (Eigentümer, Gruppenzugehörigkeit, Rechte, ..)
    - besitzt eine im Namensraum eindeutige numerische Adresse
- zentrale Datenstruktur ist die Indexknotentabelle
- Verzeichnis ist eine Abbildungstabelle (Name -> Indexknotennummer)
- Namensbindung ist der erste Schritt: Pfadname wird mit Indexknoten assoziiert
- Namensauflösung ist der zweite Schritt: Indexknoten anhand eines Pfadnames finden

Symbolauflösung

- Kompiler verteilt Programm auf Segmente und generiert Pseudobefehle
- Assemblierer ordnet Symbolen Werte und Attribute zu, generiert Symbol- und Verlagerungstabellen
- Binder platziert dsa gebundene Programm im Adressraum, produziert das Lademodul

Betriebsarten

Stapelverarbeitung

Manueller Rechenbetrieb

- Früher: Lochkarten
- Urlader (Bootloader) musste manuell einprogrammiert werden

Automatisierter Rechenbetrieb

- Dienstprogramme sind aufrufbereit im Rechensystem gespeichert
- Lochkarten wurden erweitert um Steuerkarten
- Betriebsmittelbedarf musste vor Ausführung beschrieben werden
- Residentes Steuerprogramm im Hauptspeicher (Heute BIOS)

Adressraumschutz

- Residentes Steuerprogramm durfte nicht von Ausserhalb zum Absturz gebracht werden
- Schutzgatter trennt den Hauptspeicherbereich des transienten Programms ab -> keine Übergriffe auf Speicherbereiche des Steuerprogramms
- Schutzgatterregister musste geschützt werden -> Einführung von Arbeitsmodi

Maßnahmen zur Leistungssteierung

- Ein/Ausgabe getrennt von Berechnungen
    - Satellitenrechner übernimmt Ein/Ausgabe
    - Hauptrechner übernimmt Berechnungen
    - Problem: Verbindung (Datenband) zwischen den Rechnern ist sequentiell

Mehrprogrammbetrieb

- Unterschiedung Mehrprogrammbetrieb (multi-programming, räumlich) und Simultanverarbeitung (multiprocessing, zeitlich)
- Aktives Warten
    - überlappte Ein-/Ausgabe = parallele E/A- und CPU-Stöße
    - CPU wartet evtl. auf Rückkehr des E/A Strangs -> idle
- Idee: Wartezeit eines Programms als Rechenzeit eines zweiten Programms verwenden -> passives Warten
- Umschaltmechanismus muss generisch implementiert werden um für alle Programme zu funktionieren -> Einführung des Prozesses

Systemprogrammierung 2

Prozessverwaltung

Einplanungsgrundlagen

Programmfäden

- planungseinheit zur Einplanung von Prozessen
- laufbereitschaft ist abhg. von Verfügbarkeit aller nötigen Betriebsmittel
- planung erfolgt zeitgesteuert oder ereignisbedingt
- einplanung und Einlastung nicht verwechseln!
- durchläuft abwechselnd CPU-burst und I/O-burst (blockiert evtl.)
- prozesszustand ist *Phasenuneindeutig*

Arbeitsweise des Prozessplaners

- Ebenen
    - langfristige Planung [s-min]: Lastkontrolle
        - erzeugt
        - gestoppt
        - beendet
    - mittelfristige Planung [ms-s]: Umlagerungsfunktion (swapping, hintergrund <-> vordergrund)
        - schwebend bereit
        - schwebend blockiert
    - kurzfristige Planung [<ms]: Einlastungsreihenfolge
        - bereit
        - laufend
        - blockiert
- Verdrängung
    - prozesse können zur abgabe der CPU gedrängt werden
    - einplanungslatenz: Zeit von Einplanung bis zur Bereitstellung
    - einlastungslatenz: Zeitspanne zwischen erfolgter Einplanung und Prozessorzuteilung
    - prozessbasierter OS lässt einlastung jederzeit zu (voll verdrängend)

Gütemerkmale

- Kriterien zur Aufstellung einer Einlastungsreihenfolge
- benutzerorientierte Kriterien: fokussiert auf Benutzerdienlichkeit
    - Antwortzeit
    - Durchlaufzeit
    - Termineinhaltung
    - Vorhersagbarkeit
- systemorientierte Kriterien: fokussiert auf Systemperformanz
    - Durchsatz
    - Prozessorauslastung
    - Gerechtigkeit

    - Dringlichkeiten: höchste prio bevorzugen
    - Lastausgleich

Einplanungsverfahren

- Wahl eines Verfahrens ist immer Kompromiss zwischen oder Priorisierung von Gütemerkmalen
- Einplanungsalgorithmen verwalten betriebsmittelgebundene Warteschlangen
- Warteschlangentheorie: keine absolute Lösung möglich, immer abhg. von Anforderungen

Einordnung

- Kooperativ: voneinander abhg. Prozesse, kein Entzug der CPU, CPU-Monopolisierung möglich
- Präemtiv: unabhg. Prozesse, CPU kann entzogen werden, CPU Schutz
- Synergie: kooperative user threads, präemptive kernel threads
- Deterministisch: alle Prozesse/Termine sind bekannt, genaue Vorhersage der Auslastung, Zeitgarantien gelten unabhg. von Systemlast
- Probabilistisch: unbekannte Prozesse, abschätzung der CPU Auslastung, Zeitgarantien durch die Anwendung
- Statische Planung: vor Prozessausführung, hohe Berechnungskomplexität -> nicht dynamisch Planbar, vollständiger Ablaufplan berechnet
- Dynamische Planung: läuft während der Prozessausführung, Stapelsysteme
- Asymmetrisch: je nach Prozessoreigenschaft auf Maschinenprogebene (CPU, GPU)
- Symmetrisch: je nach Prozessoreigenschaft der Befehlssatzebene, Lastausgleich möglich

Verfahrensweisen

- Kooperativ (gerecht)
    - *First come, first served* (FCFS)
    - Einplanung nach Ankunftszeit
    - höhere Antwortzeit, niedriger E/A Durchsatz
    - schlecht bei großen Unterschieden der Rechenstoßdauer
    - Konvoieffekt: kurze Rechenstöße folgen einem langen
- Verdrängend (reihum)
    - *round robin* (RR)
        - verdrängendes FCFS
        - periodische Umplanung nach *timer*
        - Länge zwischen Interrupts bestimmt die Effektivität
        - Konvoieffekt wird nur geschwächt
        - E/A Intensive Prozesse schöpfen Zeitslot selten aus -> geben CPU selbst ab
        - CPU intensive Prozesse schöpfen Zeitslot voll aus -> werden verdrängt
    - *virtual round robin* (VRR)
        - RR mit Vorzugswarteschlange
        - E/A intensive Prozesse werden bevorzugt eingeplant
        - kein voll-verdrängendes Verfahren
- Probabilistisch (priorisierend)
    - *shortest process next* (SPN)
        - prozess wird entsprechend der max erwarteten Bedienzeit eingeplant
        - Grundlage für Wissen über Prozesslaufzeiten

        - Stapelbetrieb: Programmierte Frist
        - Produktionsbetrieb: Statistisch
        - Dialogbetrieb: Abschätzung zur Laufzeit
      - verhungern längerer Prozesse ist möglich
      - kein Konvoi-Effekt
      - Bei Berechnung der mittleren Laufzeit sollten jüngste Messungen höhre gewichtet werden
    - *highest response ration next*
      - hungerfreies SPN
      - Einplanung nach erwarteter Bedienzeit
      - periodische Umplanung nach Wartezeit
    - *shortest remaining time first*
      - verdrängendes SPN
      - Hungergefahr
      - Einplanung nach erwarteter Bedienzeit
      - Umplanung spontan und unregelmäßig
- Mehrstufig
  - *multileven queue* (MLQ)
    - Typisierung von Prozessen (Dialog-, System-, Stapelprozesse)
    - Jede Typ Liste wird nach lokaler Strategie geplant
    - globale Einplanungsstrategie statisch oder dynamisch definieren
  - *multilevel feedback queue* (MLFQ)
    - begünstigt kurze, interaktive Prozesse
    - Einplanung nach Ankunftszeit, periodische Umplanung
    - mehrere Bereitlisten, nach Priorität sortiert
    - je nach Liste, verschiedene Strategien
    - **Bestrafung** lange Rechenstöße -> nach unten und vice versa

Prozesssynchronisation

Nichtsequentialität

- Nebenläufigkeit ist das Verhältnis von nicht kausal abhängigen Ereignissen
- Die Aktionen benötigen nicht das Resultat der jeweils anderen

Kausalitätsprinzip

- jede Aktion hat Ursache und Wirkung
- Wirkung einer Aktion kann Ursache einer Anderen sein, dann keine Nebenläufigkeit!

Gleichzeitige Prozesse

- Mehrere Prozesse deren Aktionen sich in Raum und Zeit überlappen
- notwendige Bedingung: OS ist Fähig zur Simultanaverarbeitung
- hinreichende Bedingung: ein nichtseq. prog. oder meherere seq. prog.

Gekoppelte Prozesse

- gleichzeitige Prozesse die min. eine Variable, Ressource gemeinsam nutzen
- erzeugen zeitliche Interferenz

- entscheidend ist logische Bedeutung der Var/Ressource (z.B. Kommunikationsmedium)
- Es bestehen Datenabhängigkeiten
- Prozesse besitzen Rollen, z.b.: Produzent/Konsument, Sender/Empfänger

Koordination von Konkurrenz

- gleichzeitige, komplexe Aktionen überlappen in Raum und Zeit
- reihenfolge von kausalen aktionen muss bestehen bleiben
- offline einplanung ungeeignet
- dynamische einplanung nötig (on-line)
- grundlage ist explizite Synchronisation durch Programmanweisungen
- explizite sync kann wettstreit erzeugen!

Atomare Aktionen

- aktion deren einzelschritte nach außen scheinbar gleichzeitig stattfinden
- wechselseitiger ausschluss schützt kritische abschnitte, gewährleistet atomarität

Synchronisationsarten und Verfahrensweisen

- unilateral (unterdrückend): Unterberchung, Verdrängung
- multilateran
    - blockierend: Semaphor, Monitor
    - nicht blockierend: atomares L/S, transaktionale Programme

Einseitige Synchronisation

- wirkt sich nur auf einen der beteiligten Prozesse aus, die anderen schreiten fort
- Bedingungssynchronisation: Fortschritt is abhg von einer Bedingung in einem nichtseq. Programm
- logische Synchronisation: Logische Abfolge, v.a. Rollenspiele
- Laufende Prozesse müssen die Wartebedingung wieder auflösen

Mehrseitige Synchronisation

- auswirkung auf alle beteiligten prozesse
- welcher prozess fortschreitet ist unbestimmt
- blockierend: pessimistisch, wahrscheinliche, häufige Konkurrenz
- nicht blockierend: optimistisch, unwahrscheinline, seltene Konkurrenz

Monitore

- auch *kritischer Bereich*
- meint Prozeduren mit einer gemeinsamen Variablen
- Ermöglicht wechselseitigen Ausschluss & Prüfung der Operationen auf der Var
- *condition* ist eine Var zur Anzeige und Steuerung des Wartezustands
- ist Mittel zur Synchronisation mittels einseitiger oder mehrseitiger Sync

Vergleich

- Hansen: blockierende Bedingungsvar (Signalnehmer hat Vorrang)
- Hoare: blockierende Bedingungsvar, wählt atomar genau einen Signalnehmer zur Fortsetzung aus

- Mesa: nichtblockierende Bedingungsvar (Signalgeber hat Vorrang), Nachfolger müssen sich "bewerben"

Semaphor

Definition

- spezielle ganzzahlige Variable
- bietet zwei operationen: P (prolaag, down, wait, aquire) und V (verhoog, up, signal, release)
- beides logisch unteilbare Operationen
- urspürunglich binärer Semaphor (s=[0,1])
- allgemeiner Semaphor (s=[n,m], m > 0 und n <= m)

Implementierung

- P und V müssen selbst atomar implementiert werden
- pessimistischer Ansatz: gleichzeitige Aktionen sind wahrscheinlich
- Implementierung als Monitor schließt wechselseitigen ausschluss aus
- Verwendung von lock-variable

Ablaufunterbrechung

- Schlafenlegen eines Prozesses
- als blockiert vermerken und auf warteliste setzen
- anderem Prozess den Prozessor zuteilen

Mutex

- "mutual exclusion"
- Unterschiede zum Semaphor
    - Semaphor kann von jedem Prozess freigegeben werden
    - Mutex kann nur vom Besitzerprozess freigegeben werden
    - Prüfung der Berechtigung zur Freigabe ist beim Mutex notwendig, nicht beim (allg.) Semaphor
    - Abbruch bei unebrechtigter Freigabe

Sperre

- verhindert Prozessauslösung (Unterbrechung, Fortsetzung, Verdrängung)
- Sperrt aber auch kausal unabhängige Prozesse -> Leistungsfähigkeit beschnitten
- funktionieren nur für Einkernprozessoren

Kreiseln und Spezialbefehle

Umlaufsperre

- `aquire`: verzögert aktuellen Prozess solange Sperre besteht, setzt sie nach Eintritt wieder
- `release`: hebt Sperre auf, ohne Verzögerung
- dient der Synchronisation gleichzeitiger Prozesse auf verschiedenen Prozessoren
- Synchronisation auf dem selben Prozessor erfordert zusätliche unilaterale Sperre

Spezialbefehle

- `__sync_lock_test_and_set` (TAS): testen, Ergebnis merken und setzen (atomar)

- `__sync_bool_compare_and_swap` (CAS): schreibt nur wenn Gleichheitsbedingung erfüllt, atomar

Schlossalgorithmen

- Kreiseln mit TAS: schädlich für Pufferspeicher, schließt kausal unabhängige Prozesse nahezu anhaltend aus
- Kreiseln mit CAS: unbedenklich für Pufferspeicher, blockiert dennoch Speicherbuszugriffe
- Kreiseln mit Ablesen: schwächt Wettstreit beim Buszugriff ab (kritischer Abschnitt darf nicht zu lang und nicht zu kurz sein)
- Kreiseln mit Zurückhaltung (*backoff*: statisch oder dynamische Verweilzeit) prozessorweise abgestuft
- Alle oberen Algorithmen können Prozesse im worst case verhungern lassen!
- Kreiseln mit proportionaler Zurückhaltung
    - Menge der angestauten Prozesse bestimmt der Wettstreitgrad
    - Implementierung: *ticket spin lock*

Transaktion

- Motivation
    - blockierende Sync beeinträchtigt Performance paralleler Systeme
    - single point of failure: prozess erhält lock und stürzt ab ohne ihn frei zu geben
    - Inteferenz mit dem Planer: Setzt sich über Planungsentscheidungen hinweg
    - Lebendigkeit: Gefahr von Verhungern oder Verklemmung
- optimistischer Ansatz
    - keine wettlaufkritischen Aktionen
    - wenn doch dann nachträglich erkennbar
    - Optimistic Concurrency Controll: "relying on transaction backup"
- Definition: Transaktion ist eine Konsistenzeinheit die Aktionsfolgen eines Prozesses gruppiert
- meist durch fußgesteuerte schleife mit Bestätigungsbedingung implementiert (z.B. CAS)

Betriebsmittelverwaltung

Betriebsmittel

- werden angefordert, zugeteilt, belegt, benutzt und freigegeben
- wiederverwendbar: persistent, nicht flüchtig
- konsumierbar: transient, flüchtig, Freigabe führt zur Entsorgung

Ziele der Verwaltung

- Konfliktefreie Auftragsabwicklung
- korrekte Bearbeitung in endlicher Zeit
- gleichmäßig, maximale Auslastung
- hoher Durchsatz, kurze Durchlaufzeit, hohe Ausfallsicherheit
- kein Verhungern/Verklemmung

Aufgaben

- Buchführung
- Steuerung der Anforderungen
- Betriebsmittelentzug

Verfahrensweisen

- statisch: vor Laufzeit, ggf. lange vor tats. Benutzung, Anforderung und Freigabe aller Betriebsmittel
- dynamisch: zur Laufzeit, Anforderung und Freigabe bei Bedarf, Verklemmungsrisiko gegeben

Systemblockade

Definition (deadly embrace)

- gekoppelte Prozesse warten gegenseitig auf die Aufhebung einer Wartebedingung
- Wartebedingungen müssen von ebendiesen Prozessen aufgehoben werden
- Warten kann aktiv oder inaktiv stattfinden

Inaktives Warten (deadlock)

- Befehlszähler bleiben konstant
- Prozess gibt den Prozessor ab
- Außnahme: Leerlaufprozess
- relativ leicht erkennbar und abgrenzbar

Aktives Warten (livelock)

- Befehlszähler wachsen
- Prozess im Zustand laufend oder bereit

Voraussetzungen für Verklemmung

- Ausschließlichkeit
- Nachforderung (mutual exclusion)
- Unentziehbarkeit (hold and wait)
- zirkulares Warten (no preemption)

Gegenmaßnahmen

- Aufhebung EINER der oben gelisteten Voraussetzungen
- Verklemmungsvorbeugung
    - indirekte Methoden
        - nichtblockierende Synchronisation
        - alle Anforderungen unteilbar platzieren
        - Virtualisierung von Betriebsmitteln
    - direkte Methoden
        - lineare Ordnung von Betriebsmittelklassen definieren
- Verklemmungsvermeidung: Vorabwissen nötig!
    - Überwachung von Prozessen
    - Überprüfung jeder Anforderung von Betriebsmitteln--
    - Zuteilung erfolgt nur im sicheren Zustand
    - Vermeidung ist immer analytisch und zur Laufzeit!

Speicherverwaltung

Adressräume

Referenzfolge

- laufender Prozess generiert Adressfolge auf den Arbeitsspeicher
- Wertevorrat ist gemäß Programm begrenzt, anfangs statisch, später dynamisch erweiterbar
- Adressraum wird dem Prozess zugebilligt, kann nicht überschritten werden

Real

- Adressraumbelegungsplan mappt Hardwareeinheiten (Speicher und Peripherie) auf Adressen
- Adressraum enthält (große) Lücken, Zugriff darauf ist undefiniert oder liefert Busfehler
- Bereiche mit speziellem Verwendungszweck sind reserviert (residenter Monitor)

Logisch

- Beschreibt den Hauptspeicher eines Prozesses
- ist linear Adressierbar, auch wenn er real fragmentiert ist
- enthält mindestens Text- und Datensegment eines Programmes
- festgelegt durch Adressraummodell
    - gekachelt (Eindimensional): Adressierung mit Seitennummer und Versatz
    - segmentiert (Zweidimensional): Adressierung mit Segmentname und Verschiebung
- Übersetzung der Tabellen wird im Übersetzungspuffer gecached -> Latenzverbergung

Virtuell

- ist eine logische Adresse, enkoppelt von der Lokaität des Arbeitsspeichers
- erlaubt Zugriff auf Adressen die logisch/real garnicht vorhanden sind
- erlaubt Seitenumlagerung (*paging*) aktiver Prozesse
- ermöglicht Ausführung von Programmen die nicht vollständig im Hauptspeicher liegen
- Präsenzbit macht Angabe über die Dereferenzierbarkeit der virtuellen Adresse

Mehradressraumsysteme

- Virtualisierung des realen Adressbereichs, nicht nur des Hauptspeichers
- Illusion eines eigenen physischen Adressraum für das Betriebssystem und Maschinenprog.
- Datenaustausch zwischen Programmer erfordert Spezialbefehle
- Ausbruch aus Adressraum wird von Spezialhardware verhindert (MMU, MPU)

Platzierungsstrategie

Freispeicherorganisation

- freie Speicherbereiche werden in der Lochliste (variable Größe)/Bitkarte (feste Größe) dokumentiert
- für jede Speicheranforderung wird die passende Stelle in der Liste gesucht
- Sortierung der Lochliste ist also Performancefaktor
- Hohe Anforderung an die Effizienz der Freispeicherverwaltung
- Identische Abbildung: eine logisch/virtuelle Adresse ist identisch zur realen Adresse
- Speicherung der Lochliste in den Löchern selbst erfodert evtl. Schutzkonzepte für Zugriffsfehler

Verfahrensweisen

- linear, Best-fit: Lochliste aufsteigend sortiert

    - kleinstes passendes Loch wird gewählt
    - beste Zuteilung, aber langsam
- linear, Worst-fit: Lochliste absteigend sortiert
    - größtes passendes Loch wird gewählt
    - schnelle Zuteilung, begünstigt Zerstückelung
- linear, First-fit: Lochliste aufsteigend nach Adresswert sortiert
    - erstes passendes Loch wird gewählt
    - schnelle Zuteilung, begünstigt Verschwendung
- linear, Next-fit
    - Suche beginnt bei letzter Zuteilung
    - fördert gleichverteilung, im mittel abnehmender Suchaufwand
- halbierungsverfahren, buddy: Lochliste ist nach Zweierpotenzgröße aufsteigend sortiert
    - kleinstes passendes loch, Größe 2^i
    - `i` ist Index des Lochs, wird fortsetzend dekrementiert

Speicherverschnitt

Fragmentierung

- intern: angeforderter Speicher ist kleiner als das zugeteilte Stück
- extern: angeforderter Speicher ist zu groß für jedes einzelne Loch

Verschmelzung

- Löcher vereinigen bei Zurücknahme von Speicher
- immer möglich wenn der Speicher an ein anderes Loch grenzt

Kompaktifizierung

- Verschiebung der Gebrauchsstücke im Speicher
- Ziel ist ein einziges großes Loch zu bilden
- Speicheradressen ändern sich hierbei NUR im realen Adressraum

Ladestrategie

- Einzelanforderung: *on demand*
    - Datum ist präsent: wird geliefert
    - Datum ist nicht präsent: MMU löst partielle Interpretation durch OS aus -> trap
- Vorausladen: anticipatory
    - Zuvorkommen der einzelanforderung
    - Heuristiken liefern Hinweise über zukünftige Zugriffe
- ggf. müssen geladene Daten verdrängt werden

Zugriffsfehler

- tritt auf wenn *present bit* der gefragten Adresse nicht gesetzt ist
- Behandlung kann je nach OS Leistunsverlust bedeuten
- Alignment von Data und Text Segment im logischen Adressraum reduziert die Anzahl von Seitenfehlern

Ersetzungsstrategie

- Grundvoraussetzung für Speichervirtualisierung
- jeder virtuelle Adressraum ist einem Prozessexemplar zugeordnet
- jedes Prozessexemplar erhält eine gewisse Anzahl an Seitenrahmen
- wenn beim Laden einer Seite alle Seitenrahmen belegt sind, muss eine Ersetzung geprüft werden
- zwei Dimensionen der Strategie: fest oder variabel, lokal oder global
- Feste/variable Zuteilung: Anzahl der Seitenrahmen ist fest/dynamisch

Globale Verfahren

- Versuch der Vorhersage welche Seiten in Zukunft (nicht) angefordert werden
- LFU (least frequently used)
    - ersetzt wird die am seltensten referenzierte Seite
    - Alternativ: MFU
- LRU (last recently used): am längsten nicht referenzierte Seite wird ersetzt

Lokale Verfahren

- Freiseitenpuffer (page buffering)
    - Arbeitet nach FIFO + Zwischenspeicher für zu ersetzende Seiten
    - zwei Listen: modifizierte und unmodifizierte Seiten
    - modifizierte Seiten werden vor Erstezung gecached (logisch Abwesend/kein Präsenzbit)
- Arbeitsmenge (working set)
    - minimale Menge an Daten und Textsegmenten die der Prozess zur Ausführung benötigt
    - Arbeitsmengengröße hängt von Prozesslokalität ab

Dateisystem

Medien

- Festplatten (Hard Disk Drive, HDD)
- Solid State Disk (SSD)
- CD-ROM/DVD

Kontinuierliche Speicherung

- speicherung der nummer des ersten blocks + anzahl folgeblöcke
- Vorteil: schneller direkter zugriff auf dateiposition
- Problem: Fragmentierungsproblem, Erweitern erfordert evtl umkopieren

Verkettete Speicherung

- blöcke einer datei sind verkettet, kann wachsen
- Probleme: Speicherbedarf für Verzeigerung, nicht restaurierbar bei fehler, schlechter direkter zugriff, häufiges positionierung des Lesekopfs
- FAT-Ansatz: File Allocation Table
- Vorteile: kompletter Datenblock nutzbar, FAT schränkt fehleranfälligkeit an, bei kleinen speichergrößen nützlich

Indiziertes Speichern

- Prinzip: eine FAT pro Datei
- Problem: feste Anzahl von Einträgen im Indexblock, verschnitt bei kleinen Dateien
- Idee: kleine Indexblöcke, bei größeren dateien, mehrere indexblöcke verketten

UNIX Dateisysteme

- System V File System: Boot und Superblock, Inodes und Datenblöcke
- Berkley Fast File System
  - Bootblock
  - Zylindergruppe: Superblock, Cylinder Group Block, Inodes, Datenblöcke
  - kürzere Positionierungszeit
- Linux EXT 2/3/4: ähnlich zu BSD, "Blockgruppen"

Windows NTFS

- Datei: beliebiger Inhalt, Rechte, Komprimierung/Verschlüsselung
- Dateiinhalt
  - Stream von Bytes
  - dynamisch erweiterbar
  - Syntax dateiname:streamname
- Basiseinheit "Cluster": Menge von Blöcken mit logischer Cluster Nummer
- Basiseinheit "Strom": Jede Datei kann mehrere Streams speichern. Dateiname, Rechte , usw. werden in einzelnen Streams gespeichert
- File Reference: Bezeichnet eindeutig eine Datei oder einen Katalog (Sequenz- und Dateinummer)
- Master File Table
  - große Tabelle mit gleich großen Elementen
  - dynamisch erweiterbar
  - Inhalt: Standart Info, Dateiname, Zugriffsrechte, Eigentlich Daten
  - Für größere Dateien: Virtual Cluster Number, Logical Cluster Number
- Verzeichnisse
  - Index of Files: File Reference, Dateiname, Dateilänge
  - Extends bei längerem Katalog

Journaling File System

- Beispiel NTFS: Änderungen an MFT und Dateien werden protokolliert
- Konsistenz von (Meta)daten kann nach Systemausfall mithilfe des Protokolls wiederhergestellt werden
- "Log based recovery"
- alle Änderungen treten als Teil von Transaktionen auf
- Logdatei wird beim Booten mit dem Dateisystem abgeglichen
- Protokollierung: schreiben des Log eintrags VOR eigentlicher Änderung
- Fehlererholung: Transaktion kann wiederholt bzw. abgeschlossen werden
- angefangene aber nicht beendete transaktionen werden rückgängig gemacht
- Beispiele: NFTS, EXT3/4

Datensicherung und RAID

- RAID 0: "Gestreifte Platten"

    - verteilung der Daten über mehrere Platten
    - keine redundanz
    - schnellerer datentransfer wegen parallelität
    - ausfall einer platte -> ausfall gesamtsystem
- RAID 1: "Gespiegelte Platten"
    - Daten werden auf zwei platten gleichzeitig gespeichert
    - Software oder Hardwareimplementierung
    - Schnelles lesen, langsameres schreiben
    - eine Platte kann komplett ausfallen
- RAID 4
    - Daten werden über mehrere Platten gespeichert
    - Paritätsblock: byteweise XOR verknüpfung der zugehörigen blöcken anderer streifen
    - beliebige anzahl platten (mindestens 3)
    - eine platte kann komplett ausfallen
    - hohe belastung der Paritätsplatte
- RAID 5
    - Paritätsblock über alle Platten verteitl (-> last ausgleich)
    - sonst wie RAID 4
- RAID 6
    - zwei paritätsblöcke
    - zwei festplatten können ausfallen
    - min 4 platten