Betriebssysteme - Das ultimative Cheat Sheet

Basierend auf Kurs 01670 - FernUniversität in Hagen

1 KE 1: Einführung & Grundlagen

1.1 Was ist ein Betriebssystem (BS)?

Definition: Menge von Programmen, die es ermöglichen, den Rechner zu betreiben und Anwendungsprogramme auf ihm auszuführen.

Zwei Hauptsichten:

- **Abstrakte/Virtuelle Maschine:** Verbirgt Hardware-Komplexität, bietet einfache Schnittstelle (API)
- **Ressourcen-Manager:** Verwaltet & verteilt Ressourcen (CPU, Speicher, Geräte) fair und effizient

1.2 Aufgaben eines Betriebssystems

Klassische Aufgaben:

- **Gerätesteuerung:** Verbergen der Hardware-Besonderheiten, Anbieten von Diensten
- **Schutz:** Speicherschutz, Zugriffschutz zwischen Benutzern
- **Fehlerbehandlung:** Division durch 0, illegale Adressen, Hardware-Defekte
- **Mehrprogrammbetrieb:** Parallele Ausführung mehrerer Programme
- **Prozess-Synchronisation/-Kommunikation:** Nachrichtenaustausch, Synchronisation
- **Ressourcenverwaltung:** CPU, E/A-Geräte, Hauptspeicher, Kommunikationsverbindungen
- **Kommandosprache:** Textuelle/grafische Schnittstelle zum System
- **Administration:** Datensicherung, Systemkonfiguration, Leistungsüberwachung

1.3 Systemarchitektur & Ebenenmodell

Ebenenmodell (von unten nach oben):

1. **Digitale Logikebene:** Gatter, Boole'sche Funktionen
2. **Mikroprogramm-Ebene:** Mikrobefehle, Mikroprogramme
3. **Konventionelle Maschinenebene:** Maschinenbefehle des Prozessors
4. **Betriebssystem-Ebene:** Systemaufrufe erweitern Maschinenebene
5. **Assembler-Sprachen:** Lesbare Namen für Maschinenbefehle
6. **Höhere Programmiersprachen:** Hardware-unabhängig

Betriebssystem-Komponenten:

- **Kern (Kernel):** Programme, die immer im Hauptspeicher sind
- **Standard-Bibliotheken:** Häufig benötigte Funktionen
- **Dienstprogramme (Utilities):** Administration, Textverarbeitung

1.4 Hardware-Grundlagen

Unterbrechungen (Interrupts):

- **Hardware-Interrupts:** Asynchrone Signale von Geräten (E/A-Abschluss)
- **Software-Interrupts (Traps):** Synchron durch Programmfehler oder Systemaufrufe
- **Ablauf:** Signal → CPU unterbricht Programm → Unterbrechungsroutine → Programmfortsetzung
- **Unterbrechungsvektor:** Tabelle mit Adressen der Unterbrechungsroutinen

Speicherschutz:

- **Grenzregister:** Trennt Benutzer- und Betriebssystem-Bereich
- **Zweck:** Schutz des BS vor fehlerhaften/bösartigen Programmen

System- und Benutzermodus:

- **Benutzermodus (User Mode):** Eingeschränkte Befehle, Speicherschutz aktiv
- **Systemmodus (Kernel Mode):** Alle Befehle erlaubt, Speicherschutz deaktiviert
- **Supervisor Call (SVC):** Kontrollierter Übergang User→Kernel für Systemaufrufe

1.5 Mehrprogrammbetrieb

Motivation:

- **Auslastungsverbesserung:** CPU arbeitet während E/A-Wartezeiten anderer Prozesse
- **Parallelität:** Mehrere Benutzer/Programme gleichzeitig
- **Virtueller Prozessor:** Jeder Prozess hat Eindruck einer eigenen CPU

Zeitscheiben:

- **Zeitgeber (Timer):** Hardware-Komponente für regelmäßige Unterbrechungen
- **Zeitscheibenablauf:** Unterbrechung nach Ablauf der zugeteilten Zeit
- **Prozesswechsel:** Umschaltung zwischen Prozessen

1.6 Betriebsarten

Interaktiver Betrieb (Dialog):

- Sofortige Programmausführung, direkte Benutzer-Programm-Kommunikation
- Time-Sharing: Mehrbenutzer-Dialogsysteme
- Optimierungsziel: Kurze Antwortzeiten

Stapelbetrieb (Batch):

- Jobs werden in Warteschlange eingereiht, keine direkte Kommunikation
- Optimierungsziel: Maximale Ressourcenauslastung
- Höhere Durchsatzraten, längere Wartezeiten akzeptabel

Hintergrundausführung:

- Programme laufen parallel zu interaktiven Prozessen
- Keine direkte Benutzerinteraktion während der Ausführung

Realzeitbetrieb:

- Harte Zeitgrenzen müssen eingehalten werden
- Zeitkritische Prozesse haben höchste Priorität
- Erfordert speziell konstruierten Betriebssystemkern

1.7 Systemstart (Bootstrap)

Ladevorgang:

- **Firmware/BIOS:** In ROM/EPROM gespeichert
- **Urlader (Bootstrap Loader):** Lädt Betriebssystem von Festplatte
- **Master Boot Record (MBR):** Enthält Startinformationen
- **Boot Manager:** Auswahl zwischen mehreren Betriebssystemen

1.8 Historisches Beispiel: CP/M

Komponenten:

- **BIOS:** Hardware-abhängige Gerätetreiber
- **BDOS:** Hardware-unabhängige Dateiverwaltung
- **CCP:** Kommandointerpreter (Shell)
- **TPA:** Transient Program Area (Benutzerbereich)

2 KE 2: Prozesse & Scheduling

2.1 Programm vs. Prozess

- **Programm:** Statische Formulierung eines Algorithmus (Programmtext)
- **Prozess:** Ablaufendes Programm inklusive aktueller Stand des Befehlszählers, Registerinhalte und Hauptspeicherbereich mit Variablenbelegungen

2.2 Prozessmerkmale

Prozesszustände:

- **erzeugt:** Datenstrukturen werden erstellt, Adressraum zugewiesen
- **bereit:** Rechenbereit, wartet auf Prozessorzuteilung
- **rechnend:** Prozessor ist zugeteilt, führt Anweisungen aus
- **blockiert:** Wartet auf Ereignis (z.B. E/A)
- **beendet:** Programmausführung ist beendet

Speicherbereich eines Prozesses:

- **Programmsegment:** Ausführbarer Code (ändert sich nicht)
- **Stacksegment:** Programmstack mit Aktivierungsblöcken
- **Datensegment:** Daten des Programms

Prozesskontrollblock (PCB):

- **Prozessidentifikation:** Eindeutige Prozess-ID
- **Prozessorstatus:** Programmzähler, alle Register
- **Prozesskontrollinformationen:** Zustand, Priorität, Speicherbereich, geöffnete Dateien, Buchhaltung, Besitzer

2.3 Zustandsübergänge & Prozesswechsel

Wichtige Übergänge:

- erzeugt → bereit (2): Ressourcen zugeteilt
- bereit → rechnend (3): Prozessor zugeteilt
- rechnend → blockiert (5): Warten auf Ereignis
- blockiert → bereit (6): Ereignis eingetreten
- rechnend → bereit (4): Zeitscheibe abgelaufen oder freiwillige Abgabe
- rechnend → beendet (7): Prozess terminiert

Dispatcher: Führt Prozesswechsel durch

- Sichert Prozessorzustand (Register, Programmzähler)
- Übergibt PCB an Scheduler
- Stellt Zustand des neuen Prozesses wieder her

Präemptiv vs. Nicht-präemptiv:

- **Nicht-präemptiv:** Nur Prozess selbst gibt Prozessor ab
- **Präemptiv:** Betriebssystem kann Prozessor entziehen (Timer-Interrupt)

2.4 Scheduling-Strategien

Qualitätsmaßstäbe:

- Prozessorauslastung, Durchlaufzeit, Durchsatz, Antwortzeit, Fairness
- CPU burst: Zeit, die Prozess den Prozessor am Stück behalten will

Nicht-präemptive Verfahren:

Verfahren	Vorteile	Nachteile
FCFS	Einfach, fair, geringer Verwaltungsaufwand	Kurze Prozesse warten lange (Convoy-Effekt)
SJF	Minimale mittlere Wartezeit	Verhungern langer Prozesse, Bedienzeit unbekannt
Priority	Wichtige Aufgaben schnell	Verhungern bei statischen Prioritäten

Präemptive Verfahren:

Verfahren	Vorteile	Nachteile
Round Robin	Sehr fair, gut für interaktive Systeme	Overhead durch Kontextwechsel
SRTF	Optimal für bekannte Zeiten	Verhungern, viele Unterbrechungen
Priority (präemptiv)	Flexible Prioritäten	Komplexer

Quantum-Wahl bei Round Robin:

- Zu klein: Hoher Verwaltungsaufwand
- Zu groß: Schlechte Antwortzeiten
- Optimal: Etwas größer als typische Interaktionszeit

2.5 Kombinierte Strategien

Feedback Scheduling:

- Berücksichtigt Vergangenheit des Prozesses
- Aging: Priorität steigt mit Wartezeit → verhindert Verhungern
- Rechenzeitabhängig: Neue Prozesse hohe Priorität + kleine Zeitscheibe
- Bei Quantumverbrauch: Niedrigere Priorität + größere Zeitscheibe

Multiple Queues:

- Verschiedene Klassen (System-, Dialog-, Hintergrundprozesse)
- Jede Klasse eigene Warteschlange + Scheduler
- Prozessorzeit-Verteilung zwischen Klassen (z.B. 60%/30%/10%)

Linux-Scheduler:

- **O(1)-Scheduler:** Konstante Laufzeit, Prioritäten 0-139, aktive/abgelaufene Gruppen
- **CFS (Completely Fair):** Virtuelle Zeit pro Prozess, perfekte Fairness angestrebt

2.6 Threads (Leichtgewichtige Prozesse)

Konzept:

- Mehrere Ausführungspfade pro Prozess
- **Gemeinsam:** Code, Daten, geöffnete Dateien
- **Separat:** Programmzähler, Register, Stack

Realisierungen:

Benutzer-Threads	Kernel-Threads
+ Einfache Realisierung	+ Echte Parallelität auf Multiprozessoren
+ Schnelles Umschalten	+ Ein blockierender Thread stoppt nicht alle
- Blockierung stoppt alle	- Aufwändige Realisierung
- Keine echte Parallelität	- Langsameres Umschalten

Anwendungsgebiete:

- Mehrprozessor-Maschinen (Parallelisierung)
- Gerätetreiber (parallele Anfragen-Bearbeitung)
- Verteilte Systeme (Server mit mehreren Clients)

3 KE 3: Hauptspeicherverwaltung

3.1 Grundlagen

Logische vs. Physische Adressen:

- **Physische Adresse:** Reale Adresse einer Speicherzelle im RAM
- **Logische/Virtuelle Adresse:** Vom Programm erzeugte Adresse, hardware-unabhängig
- **MMU (Memory Management Unit):** Hardware zwischen CPU und Hauptspeicher für Adressumsetzung

Übersetzer, Binder und Lader:

- **Quellprogramm-Modul → Compiler → Bindemodul → Binder → Lademodul → Lader → Geladenes Programm**
- **Absolute Adressen:** Binder kennt logischen Adressraum bereits
- **Relative Adressen:** Lader addiert Startadresse zu relativen Adressen
- **Basisregister-Adressierung:** (Registernummer, Offset) für verschiebbare Programme
- **Dynamisches Binden:** Bindung zur Laufzeit, Module werden bei Bedarf nachgeladen

3.2 Einfache zusammenhängende Speicherzuweisung

Konzept: Ein Prozess = ein zusammenhängender Speicherbereich

- Betriebssystemkern ab Adresse 0
- Anwenderprogramm ab Adresse a
- **Speicherschutz:** Grenzregister verhindert Zugriff auf Adressen $< a$
- **Swapping:** Kompletter Prozess wird auf Sekundärspeicher aus-/eingelagert

3.3 Mehrfache zusammenhängende Speicherzuweisung

MFT (Multiprogramming with Fixed Tasks):

- Feste Segmentgrößen beim Systemstart
- **Interne Fragmentierung:** Zugewiesener Speicher > Bedarf
- **Best-available-fit:** Kleinstes ausreichendes Segment wählen
- **Best-fit-only:** Nur Segmente verwenden, die nicht wesentlich größer sind

MVT (Multiprogramming with Variable Tasks):

- Variable Segmentgrößen je nach Bedarf
- **Externe Fragmentierung:** Viele kleine, nicht nutzbare Lücken
- **Lückenmanagement** erforderlich

Speicherplatzvergabestrategien:

- **First Fit:** Erste ausreichend große Lücke
- **Next Fit:** Wie First Fit, aber ab letzter Zuteilung suchen
- **Best Fit:** Kleinste ausreichend große Lücke (erzeugt kleine Restlücken)
- **Worst Fit:** Größte Lücke wählen (große Restlücke bleibt nutzbar)
- **Buddy-Verfahren:** Nur 2er-Potenzen, interne Fragmentierung, aber effiziente Verwaltung

Kompaktifizierung (Garbage Collection):

- Verschieben belegter Bereiche zur Zusammenlegung freier Bereiche
- Hoher Aufwand, daher selten verwendet

3.4 Nichtzusammenhängende Speicherzuweisung

Segmentierung:

- **Trennung von Programm und Daten** in separate Segmente
- Logische Adresse = (Segmentnummer, Relativadresse)
- **Basisregister pro Segment** für Adressumsetzung
- **Wiedereintrittsfähige Programme (Reentrant Code):** Ein Programmsegment für mehrere Prozesse
- **Shared Libraries:** Gemeinsam benutzte Bindemodule

3.5 Paging (Seitenorientierte Speicherverwaltung)

Grundkonzept:

- **Seiten (Pages):** Logischer Adressraum in gleichgroße Blöcke (typisch 4 KB)
- **Seitenrahmen (Page Frames):** Physischer Speicher in gleichgroße Blöcke
- **Seitentabelle:** Bildet Seiten auf Seitenrahmen ab
- Logische Adresse = (Seitennummer s, Offset d) mit s = v div p, d = v mod p

Adressumsetzung:

- Physische Adresse r = ST[s] × p + d
- **TLB (Translation Lookaside Buffer):** Schneller Assoziativspeicher für häufig benutzte Seitentabellen-Einträge
- **TLB Hit:** Eintrag im TLB gefunden (schnell)
- **TLB Miss:** Zugriff auf Seitentabelle im Hauptspeicher (langsam)

Zusätzliche Funktionen:

- **Individueller Speicherschutz:** Protection-Bits pro Seite (read/write/execute)
- **Shared Memory:** Seiten in mehreren Adressräumen einblenden
- **Memory-Mapped Files:** Dateiseiten in virtuellen Speicher einblenden

Implementierung:

- **Mehrstufige Seitentabellen:** Bei großen Adressräumen (z.B. 32 Bit)
- **Seitenrahmentabelle:** Globale Tabelle über Zustand aller Frames

3.6 Virtueller Hauptspeicher

Demand Paging:

- **Konzept:** Seiten werden erst bei Bedarf geladen, nicht alle Seiten eines Prozesses sind im RAM
- **Seitenfehler (Page Fault):** Zugriff auf nicht geladene Seite → Trap → Einlagerung
- **Present-Bit:** Zeigt an, ob Seite im Hauptspeicher liegt
- **Dirty Bit:** Zeigt an, ob Seite seit Einlagerung verändert wurde
- **Swap-Bereich:** Reservierter Festplattenbereich für ausgelagerte Seiten

Kosten eines Seitenfehlers:

- Festplattenzugriff $\approx 10^5$ CPU-Instruktionen
- Seitenfehler dürfen nur sehr selten auftreten ($< 10^{-4}$)

3.7 Seitenauslagerungsstrategien

Optimale Strategie (theoretisch):

- Lagere Seite aus, die am weitesten in der Zukunft benutzt wird
- Nicht implementierbar (Hellsehen unmöglich)
- Dient als Vergleichsmaßstab

LRU (Least Recently Used):

- Lagere am längsten unbenutzte Seite aus
- Beste praktische Approximation der optimalen Strategie
- **Problem:** Aufwändige exakte Implementierung

Approximation von LRU:

- **Zugriffsbits (Referenced Bits):** Hardware setzt Bit bei jedem Zugriff
- **Schieberegister:** Sammelt Zugriffsmuster über mehrere Perioden
- **Second Chance:** FIFO + Zugriffsbits, gibt Seiten zweite Chance
- **Clock-Algorithmus:** Zirkuläre Liste + Uhrzeiger, effiziente Implementierung von Second Chance

Einfachere Strategien:

- **FIFO:** Längste Zeit im Speicher → Problem: kann auch aktive Seiten auslagern
- **Nachteile:** Belady-Anomalie möglich (mehr Frames → mehr Seitenfehler)

3.8 Zuweisungsstrategien

Lokalitäten:

- **Lokalitätsprinzip:** Programme greifen zeitlich/räumlich konzentriert auf Speicher zu
- **Lokalität:** Menge von Seiten, die über kurzen Zeitraum häufig benutzt werden
- Beispiele: Schleifen, sequentielle Datenverarbeitung, zusammenhängende Datenstrukturen

Arbeitsmengenstrategie (Working Set):

- **Arbeitsmenge:** Seiten, die in letzter Zeit"benutzt wurden
- **Fenster:** Betrachteter Zeitraum ($\approx$ 10.000 Zugriffe)
- **Strategie:** Jedem Prozess so viele Frames zuteilen, wie Arbeitsmenge groß ist
- Bei Seitenfehler: Weiteren Frame zuteilen
- Seite fällt aus Arbeitsmenge: Frame entziehen

Seitenfehler-Frequenz-Algorithmus (PFF):

- Misst Zeit t zwischen Seitenfehlern eines Prozesses
- Wenn t $\leq$ T: Prozess braucht mehr Frames
- Wenn t $>$ T: Alle Seiten mit R-Bit = 0 auslagern

3.9 Scheduling bei virtuellem Speicher

Thrashing (Seitenflattern):

- System produziert nur noch Seitenfehler, kaum produktive Arbeit
- **Ursache:** Summe aller Arbeitsmengen > physischer Speicher
- **Lösung:** Weniger Prozesse parallel ausführen

Scheduling-Aspekte:

- Nur so viele Prozesse im Zustand "bereit", dass ihre Arbeitsmengen in den RAM passen
- Bei Speichermangel: Prozesse komplett stilllegen (alle Seiten auslagern)
- Vor Reaktivierung: Arbeitsmenge wieder einlagern (Prepaging)

Benutzergesteuerte Speicherverwaltung:

- Programme können Hinweise auf Zugriffsmuster geben
- Beispiel: Sequentielle Verarbeitung → Seiten schnell wieder auslagern
- Wichtige Datenstrukturen → bevorzugt im Speicher halten

4 KE 4: Prozesskommunikation

4.1 Grundlagen der Prozesskommunikation

Warum Kommunikation?

- **Gemeinsame Betriebsmittel:** Hardware/Software gleichzeitig nutzen
- **Kosteneffizienz:** Shared Libraries, gemeinsame Datenbasen
- **Wiedereintrittsinvariante Programme:** Ein Programm für mehrere Prozesse

Prozessarten:

- **Disjunkte Prozesse:** Keine gemeinsamen veränderbaren Daten
- **Überlappende Prozesse:** Gemeinsame veränderbare Daten → Race Conditions

Race Conditions: Ergebnis hängt von zeitlicher Reihenfolge der Operationen ab

4.2 Kritische Abschnitte

Definition:

- **Kritischer Abschnitt:** Programmteil mit Zugriff auf gemeinsame Daten
- **Unkritischer Abschnitt:** Kein Zugriff auf gemeinsame Daten

Wechselseitiger Ausschluss (Mutual Exclusion):

- Nur ein Prozess darf gleichzeitig im kritischen Abschnitt sein
- Abstrakte Form: `enter_critical_section` → kritischer Abschnitt → `leave_critical_section`

4.3 Anforderungen an wechselseitigen Ausschluss

5 Grundanforderungen:

1. **Mutual Exclusion:** Höchstens ein Prozess im kritischen Abschnitt
2. **Deadlock Freedom:** Entscheidung in endlicher Zeit
3. **Fairness:** Kein Prozess verhungert (Starvation-frei)
4. **Unabhängigkeit:** Gestoppter Prozess außerhalb kritischem Abschnitt blockiert andere nicht
5. **Geschwindigkeitsunabhängigkeit:** Keine Annahmen über relative Prozessgeschwindigkeiten

4.4 Synchronisationsvariablen

Einfache Ansätze (unzureichend):
Alternating Turn:

```
s := 1;
Prozess 1: if s=1 do {...; s:=2}
Prozess 2: if s=2 do {...; s:=1}
```

Problem: Strikte Alternierung, Selbstbehinderung
Flag-Variablen:

```
flag[0] := false; flag[1] := false;
Prozess i: flag[i] := true;
          while flag[1-i] do {};
          {...kritisch...}; flag[i] := false;
```

Problem: Deadlock möglich

Peterson-Algorithmus (korrekt):

```
flag[0]:=false; flag[1]:=false; turn:=0;
Prozess i: flag[i] := true; turn := i;
          while (flag[1-i] and turn=i) do {};
          {...kritisch...}; flag[i] := false;
```

4.5 Hardware-Unterstützung

Test-and-Set-Lock (TSL):

- **Atomare Operation:** Liest Wert und setzt ihn auf 1
- **Bus wird gesperrt** während TSL-Ausführung
- **Funktionsweise:** LOCK=0 (frei), LOCK=1 (belegt)

```
enter_critical_section:
    TSL RX, LOCK    ; Kopiere LOCK nach RX, setze LOCK=1
    CMP RX, #0      ; War LOCK vorher 0?
    JNE enter_critical_section ; Nein → wiederholen
    RET             ; Ja → weiter

leave_critical_section:
    MOVE LOCK, #0   ; LOCK freigeben
    RET
```

Problem: Aktives Warten (Busy Waiting) verschwendet CPU-Zeit

4.6 Semaphore

Konzept (Dijkstra 1968):

- **Semaphor:** Ganzzahlige Variable mit speziellen Operationen
- **Initialisierung:** s := k (k gleichzeitige Zugriffe erlaubt)

Atomare Operationen:

- **down(s) / P(s) / wait(s):**
  - s := s - 1
  - if s ¡ 0 then blockiere Prozess
- **up(s) / V(s) / signal(s):**
  - s := s + 1
  - if s $\leq$ 0 then wecke einen wartenden Prozess

Implementierung:

```
down(s):
    s := s - 1;
    if s < 0 then begin
        füge Prozess in Warteschlange ein;
        blockiere Prozess;
    end;

up(s):
    s := s + 1;
    if s <= 0 then begin  (* Warteschlange nicht leer *)
        wähle Prozess aus Warteschlange;
        versetze in "bereit"-Zustand;
    end;
```

Semaphor-Typen:

- **Binäre Semaphore:** s $\in$ 0,1, für Mutual Exclusion
- **Allgemeine Semaphore:** s $\geq$ 0, für Ressourcenzählung

Interpretation des Semaphor-Werts:

- s $\geq$ 0: Anzahl verfügbarer Ressourcen
- s $<$ 0: —s— = Anzahl wartender Prozesse

4.7 Klassische Synchronisationsprobleme

4.7.1 Erzeuger-Verbraucher-Problem

Szenario: Erzeuger produziert Daten, Verbraucher konsumiert sie über Puffer

Unbegrenzter Puffer:

```
var inhalt : semaphor;
inhalt := 0;

Erzeuger:                    Verbraucher:
repeat                       repeat
  Erzeuge Ware;               down(inhalt);
  Bringe in Puffer;           Hole aus Puffer;
  up(inhalt);                 Verbrauche Ware;
until false;                 until false;
```

Begrenzter Puffer (Kapazität n):

```
var voll, leer, mutex : semaphor;
voll := 0; leer := n; mutex := 1;

Erzeuger:                    Verbraucher:
repeat                       repeat
  Erzeuge Ware;               down(voll);
  down(leer);                 down(mutex);
  down(mutex);                Hole aus Puffer;
  Bringe in Puffer;           up(mutex);
  up(mutex);                  up(leer);
  up(voll);                   Verbrauche Ware;
until false;                 until false;
```

4.7.2 Philosophen-Problem

Szenario: n Philosophen an rundem Tisch, n Gabeln, jeder braucht 2 Gabeln zum Essen

Naive Lösung (deadlock-anfällig):

```
var gabel : array[0..n-1] of semaphor;
for i := 0 to n-1 do gabel[i] := 1;

Philosoph i:
repeat
  denken;
  down(gabel[i]);        (* linke Gabel *)
  down(gabel[(i+1) mod n]); (* rechte Gabel *)
  essen;
  up(gabel[(i+1) mod n]);
  up(gabel[i]);
until false;
```

Korrekte Lösung:

```
var ausschluss : semaphor; privat : array[0..n-1] of semaphor;
var c : array[0..n-1] of (denken, hungrig, essen);

procedure teste(i):
  if (c[i] = hungrig and
      c[links(i)] <> essen and
      c[rechts(i)] <> essen) then
  begin
    c[i] := essen;
    up(privat[i]);
  end;

procedure gabel_nehmen(i):
  down(ausschluss);
  c[i] := hungrig;
  teste(i);
  up(ausschluss);
  down(privat[i]);  (* warte falls nötig *)

procedure gabel_weglegen(i):
  down(ausschluss);
  c[i] := denken;
  teste(links(i));  (* Nachbarn prüfen *)
  teste(rechts(i));
  up(ausschluss);
```

4.7.3 Leser-Schreiber-Problem

Szenario: Mehrere Leser gleichzeitig OK, aber nur ein Schreiber exklusiv

Priorität für Leser:

```
var readcount : integer; db, readsem : semaphor;
readcount := 0; db := 1; readsem := 1;

Leser:                       Schreiber:
repeat                       repeat
  down(readsem);              down(db);
  readcount := readcount+1;    Schreibe Daten;
  if readcount=1 then          up(db);
    down(db);                 until false;
  up(readsem);
  Lies Daten;
  down(readsem);
  readcount := readcount-1;
  if readcount=0 then
    up(db);
  up(readsem);
until false;
```

Problem: Schreiber können verhungern

4.8 Nachrichtenaustausch

Konzept: Kommunikation ohne gemeinsamen Speicher

Ringpuffer-Implementierung:

- **Sender:** Erzeugt Nachrichten, legt sie in Puffer
- **Empfänger:** Entnimmt Nachrichten aus Puffer
- **Synchronisation:** Semaphore für leer", "voll", SZugriffschutz"

Briefkasten-Prinzip:

- **Einweg-Kommunikation:** A → B (unidirektional)
- **Zweiweg-Kommunikation:** A ↔ B mit Bestätigung (acknowledgment)

4.9 Monitore

Konzept (Hoare 1974):

- **Kapselung:** Daten + Prozeduren in einem Modul
- **Automatischer Mutex:** Nur ein Prozess zur Zeit im Monitor
- **Bedingungsvariablen:** Für komplexere Synchronisation

Grundstruktur:

```
monitor monitorname
  Datendeklarationen;

  procedure prozedur1(...) { ... }
  procedure prozedur2(...) { ... }

  begin
    Initialisierung;
  end monitor;
```

Bedingungsvariablen:

- **wait(c):** Blockiert Prozess, gibt Monitor frei
- **signal(c):** Weckt einen wartenden Prozess (falls vorhanden)

Erzeuger-Verbraucher mit Monitor:

```
monitor erzeuger_verbraucher;
  buffer : array[0..N-1] of item;
  in, out, count : integer;
  notfull, notempty : condition;

  procedure insert(x);
  begin
    if count = N then wait(notfull);
    buffer[in] := x;
    in := (in + 1) mod N;
    count := count + 1;
    signal(notempty);
  end;

  procedure remove(x);
  begin
    if count = 0 then wait(notempty);
    x := buffer[out];
    out := (out + 1) mod N;
    count := count - 1;
    signal(notfull);
  end;

begin
  in := 0; out := 0; count := 0;
end monitor;
```

4.10 Deadlocks (Systemverklemmungen)

Definition: Eine Menge von Prozessen befindet sich im Deadlock, wenn jeder Prozess auf ein Ereignis wartet, das nur von einem anderen Prozess der Menge ausgelöst werden kann.

Vier notwendige Bedingungen (Coffman et al.):

1. **Wechselseitiger Ausschluss:** Ressourcen können nur exklusiv benutzt werden
2. **Hold-and-Wait:** Prozesse halten Ressourcen und warten auf weitere
3. **Keine Unterbrechung:** Ressourcen können nicht entzogen werden
4. **Zyklisches Warten:** Geschlossene Kette von Prozessen und Ressourcen

Deadlock-Strategien:

1. Erkennung und Beseitigung (Detection):

- **Zustandsdarstellung:** Matrizen für Allokation, Anforderung, Verfügbarkeit
- **Erkennungsalgorithmus:** Suche nach beendbaren Prozessen
- **Beseitigung:** Prozesse abbrechen oder Ressourcen entziehen

Erkennungsalgorithmus:

```
for i := 1 to n do beendbar[i] := false;
repeat
  noch_einer_beendbar := false;
  for i := 1 to n do
    if not beendbar[i] then
      if Anforderung[i] <= Verfügbar then
      begin
        beendbar[i] := true;
        Verfügbar := Verfügbar + Allokation[i];
        noch_einer_beendbar := true;
      end;
until not noch_einer_beendbar;

deadlock := exists i: not beendbar[i];
```

2. Vermeidung (Avoidance):

- **Banker-Algorithmus:** Basiert auf maximalen Anforderungen
- **Sichere Zustände:** Existiert Ausführungsreihenfolge ohne Deadlock
- **Problem:** Maximale Anforderungen meist unbekannt

3. Verhinderung (Prevention):

- **Wechselseitiger Ausschluss aufheben:** Meist unmöglich
- **Hold-and-Wait verhindern:** Alle Ressourcen auf einmal anfordern
- **Unterbrechung erlauben:** Ressourcenentzug (nicht immer möglich)
- **Lineare Ordnung:** Ressourcen nur in fester Reihenfolge anfordern

4. Ignorieren ("Vogel-Strauß-Strategie"):

- Problem wird ignoriert (Windows, Linux)
- Benutzer muss selbst eingreifen
- Begründung: Deadlocks sehr selten

Übergreifende Strategie:

- **Interne Ressourcen:** Verhinderung durch lineare Ordnung
- **Hauptspeicher:** Verhinderung durch Swapping
- **Prozessressourcen:** Vermeidung mit Voranmeldung
- **Swap-Bereich:** Verhinderung durch Vorausallokation

5 KE 5: Geräteverwaltung & Dateisysteme

5.1 Ein-/Ausgabe-Geräte

Gerätetypen nach Zweck:

- **Sekundärspeicher:** Permanente Speicherung (Festplatten, Bänder, optische Speicher)
  - Arbeitsdaten: Täglich benötigt, schneller Zugriff
  - Sicherungskopien: Backup-Daten, langsamerer Zugriff OK
  - Archivierte Daten: Langzeitspeicherung (Jahrzehnte)
- **Ausgabegeräte:** Bildschirme, Drucker, Lautsprecher
- **Eingabegeräte:** Tastaturen, Mäuse, Scanner
- **Kommunikationsgeräte:** Ethernet, Modems, WLAN

Gerätetypen nach Übertragungseinheit:

- **Block-Geräte:** Übertragung in Blöcken fester Größe (Festplatten, Bänder)
- **Zeichen-Geräte:** Byteweise Übertragung ohne Blockstruktur (Terminals, Drucker)

5.2 Ein-/Ausgabe-Kommunikationstechniken

Controller (Geräte-Steuereinheit):

- Hardware zwischen CPU und Geräten
- Mehrere Geräte pro Controller möglich (z.B. USB: bis 127 Geräte)
- Entlastet CPU von elementaren Steuerungsaufgaben
- Eigene Prozessoren für Prüfsummen, Fehlerkorrektur

CPU-Controller-Kommunikation:

- **I/O-Ports:** Spezielle Befehle `IN/OUT` für Registeradressen
- **Memory-mapped I/O:** Controller-Register im Hauptspeicher-Adressraum

Drei E/A-Techniken:

1. Programmgesteuerte E/A:

- CPU fragt Statusregister ab (Polling, Busy Waiting)
- Synchrone Abarbeitung
- **Nachteil:** CPU komplett belegt bis E/A beendet

2. Interrupt-gesteuerte E/A:

- **Ablauf:** CPU gibt Auftrag → wird blockiert → Interrupt bei Fertigstellung → CPU fortgesetzt
- Ermöglicht überlappende E/A-Operationen
- CPU kann andere Prozesse bearbeiten

3. DMA (Direct Memory Access):

- Controller kann direkt auf Hauptspeicher zugreifen

- **DMA-Register:** Quelle, Ziel, Anzahl Bytes, Richtung
- CPU nur für Initialisierung und Abschluss involviert
- **Ablauf:** CPU initialisiert → DMA überträgt → Interrupt bei Fertigstellung

5.3 E/A-Software-Schichtenmodell

Von unten nach oben:

1. **Interrupt-Handler:** Verarbeitet alle Unterbrechungen
2. **Gerätetreiber:** Geräteabhängige Funktionen
   - Ein Treiber pro Controller-Typ
   - Übersetzt abstrakte Befehle in konkrete Hardware-Operationen
   - Beispiel: Blockadresse → Zylinder, Sektor, Oberfläche
3. **Geräteunabhängige E/A-Software:**
   - Einheitliche Schnittstelle für alle Geräte
   - Pufferung, Fehlerbehandlung, Gerätezuteilung
4. **Benutzer-E/A-Software:** Systemaufrufe, Libraries

Wichtige Funktionen der geräteunabhängigen Schicht:

- **E/A-Auftragslistenverwaltung:** Warteschlangen pro Gerät
- **Gerätezustandstabelle:** Status aller angeschlossenen Geräte
- **Pufferung:** Double Buffering, Ringpuffer für Effizienz
- **Spooling:** Simulation exklusiver Geräte (Drucker)
- **Geräteallokation:** Zuteilung und Freigabe von Geräten

5.4 Festplatten (Magnetplatten)

Physischer Aufbau:

- **Scheiben:** Mehrere übereinander (2-10), gemeinsame Achse
- **Oberflächen:** Beide Seiten jeder Scheibe beschichtet
- **Arme:** Bewegliche Lese-/Schreibköpfe zwischen Scheiben
- **Spuren:** Kreisförmige Linien auf Oberflächen
- **Zylinder:** Spuren gleichen Radius übereinander
- **Sektoren:** Unterteilung der Spuren (0,5-4 KB)

Sektoradressierung:

- **Sektoradresse:** (Zylinder z, Oberfläche o, Sektor s)
- **Sektornummer:** $(z \times O + o) \times S + s$
- Wo O = Anzahl Oberflächen, S = Sektoren pro Spur

Zugriffszeiten:

- **Suchzeit:** Positionierung der Köpfe (1-10 ms)
- **Latenzzeit:** Warten auf richtigen Sektor (durchschnittlich halbe Umdrehung, 2,8-7 ms)
- **Übertragungszeit:** Datentransfer ($T = \frac{b}{\omega n}$ Sekunden für b Bytes)
- **Gesamte Zugriffszeit = Suchzeit + Latenzzeit + Übertragungszeit**

Optimierungsverfahren:

- **SSTF (Shortest Seek Time First):** Nächster Auftrag mit geringster Suchzeit
  - Problem: Starvation möglich
- **SCAN (Elevator):** Arme wandern alternierend nach außen/innen
  - Verhindert Starvation, faire Bedienung
- **Interleaving:** Sektoren überspringen beim Schreiben
  - Grund: Zeit für Datenübertragung zum Hauptspeicher
  - Interleave Factor bestimmt Anzahl übersprungener Sektoren

Partitionierung:

- **Master Boot Record (MBR):** Sektor 0 mit Bootcode und Partitionstabelle
- **Partitionstypen:** Primär (max. 4), erweitert mit logischen Laufwerken
- **High-Level-Formatierung:** Anlegen des Dateisystems pro Partition

5.5 Optische Speicher & Flash

Optische Platten:

- **CD-ROM:** Nur lesbar, 650 MB
- **CD-R/DVD±R:** Einmal beschreibbar
- **CD-RW/DVD±RW:** Wiederbeschreibbar
- **Blu-ray:** Bis 25 GB pro Layer
- Langsamere Übertragung als Festplatten, aber wechselbar

Flash-Speicher (SSD):

- **NAND vs. NOR:** NAND für Massenspeicher, NOR für Code
- **Erase Blocks:** 128-256 KB, nur blockweise löschbar
- **Pages:** Kleinste Lese-/Schreibeinheit innerhalb Block
- **Wear Leveling:** Gleichmäßige Abnutzung aller Blöcke
- **Garbage Collection:** Aufräumen ungültiger Daten

Flash Translation Layer (FTL):

- **Adressübersetzung:** Logische → physische Blöcke
- **Bad Block Management:** Defekte Blöcke ausblenden
- **Wear Leveling:** Hot/Cold Data auf Young/Old Blocks verteilen

5.6 Dateisysteme - Grundlagen

Datei-Konzept:

- **Datei:** Sammlung zusammengehöriger Informationen
- **Dateisystem:** Menge von Dateien + Verzeichnissen + Hilfsdaten
- **Transparenz:** Details der Speicherung verborgen

Dateierungen:

- Beliebig viele Dateien verschiedener Größe
- Sinnvolle Organisation in Verzeichnissen
- Strukturierte Zugriffe (Zeichen, Sätze)
- Zugriffskontrolle zwischen Benutzern

5.7 Dateiverzeichnisse

Hierarchische Dateisysteme:

- **Wurzelverzeichnis:** Einstiegspunkt ins Dateisystem
- **Pfadnamen:**
  - **Absolut:** Beginnend mit `/` oder `\`
  - **Relativ:** Bezogen auf aktuelles Arbeitsverzeichnis
- **Navigation:** `cd` für Verzeichniswechsel, `..` für Elternverzeichnis

Dateiattribute:

- **Größe:** Aktuelle Dateigröße in Bytes
- **Besitzer:** Kontrolle über Zugriffsrechte
- **Zugriffsrechte:** Lesen, Schreiben, Ausführen
- **Zeitstempel:** Erstellung, letzter Zugriff, letzte Änderung
- **Dateityp:** Anwendbare Zugriffsmethode
- **Speicherort:** Verweise auf Datenblöcke

5.8 Verwaltung der Dateisektoren

1. File Allocation Table (FAT):

- **Prinzip:** Zentrale Tabelle mit einem Eintrag pro Sektor
- **Verkettung:** Sektoren einer Datei als verknüpfte Liste
- **Startsektor:** Steht im Verzeichniseintrag
- **Vorteile:** Einfach, gut für sequentiellen Zugriff
- **Nachteile:** Gesamte FAT muss im RAM sein (bei 20 GB ≈ 80 MB)

Beispiel FAT-Eintrag:

```
Datei "spiel" in Sektoren 3,27,19:
Verzeichnis: spiel → Startsektor 3
FAT[3] = 27, FAT[27] = 19, FAT[19] = nil
```

2. i-nodes (Index-Nodes):

- **Prinzip:** Separate Datenstruktur pro Datei
- **Inhalt:** Dateimetadaten + Zeiger auf Datenblöcke
- **Mehrstufige Indirektion:**
  - 10 direkte Zeiger (Blöcke 0-9)
  - 1 einfach indirekter Zeiger (Blöcke 10 bis 9+x)
  - 1 doppelt indirekter Zeiger (Blöcke 10+x bis $9+x+x^2$)
  - 1 dreifach indirekter Zeiger (Blöcke $10+x+x^2$ bis $9+x+x^2+x^3$)
- **Vorteile:** Nur benötigte i-nodes im RAM, gut für kleine/große Dateien
- **Beispiel:** x=256 → max. Dateigröße ≈ 16 GB

3. NTFS (NT File System):

- **MFT (Master File Table):** Array von 1 KB-Einträgen
- **Sektorfolgen:** Kompakte Darstellung zusammenhängender Blöcke
- **Format:** (Startblock, Länge) statt einzelner Blockadressen
- **Journaling:** Protokollierung aller Operationen für Crash-Recovery
- **Eindeutige Nummern:** 48 Bit Position + 16 Bit Sequenznummer

5.9 Verwaltung freier Sektoren

1. Verkettete Liste:

- Alle freien Sektoren als verknüpfte Liste
- **Bei FAT:** Nutzung der FAT-Struktur selbst
- **Bei Direktverwaltung:** Folgezeiger im Sektor selbst (I/O-intensiv)
- **Verbesserung:** Gruppierung (x Nummern pro Indexblock)

2. Bitmap:

- **Ein Bit pro Sektor:** 1=frei, 0=belegt
- **Vorteile:** Einfaches Löschen, wortweise Suche möglich
- **Nachteile:** Bei fast voller Platte langsame Suche
- **Speicherbedarf:** Bei 20 GB ≈ 2,5 MB für Bitmap

5.10 Optimierungen

E/A-Puffer:

- **Zweck:** Zwischenspeicherung für mehrere Prozesse
- **Double Buffering:** Ein Puffer wird gefüllt, anderer geleert

- **Cache-Strategien:** LRU, FIFO für Pufferersetzung

Asynchrones Schreiben:

- **Synchron:** Warten bis Daten auf Platte (sicher)
- **Asynchron:** Rückkehr nach Pufferung (schnell)

Vorausschauendes Lesen:

- Bei sequentiellem Zugriff: Nächste Blöcke vorab laden
- Read-ahead basierend auf Zugriffsmuster

Suchzeitreduktion:

- **i-node-Lokalität:** i-nodes nahe bei Datenblöcken
- **Reorganisation:** Zusammenhängende Speicherung
- **Zylindergruppen:** Aufteilen der Platte in Bereiche

5.11 Zugriffsmethoden

Zugriffsstrukturen:

- **Sequentiell:** Wie verknüpfte Liste, nur Anhängen möglich
- **Direktzugriff statisch:** Array-artig, direkter Zugriff über Index
- **Direktzugriff dynamisch:** Index-sequentiell mit Schlüsseln

Speichereinheiten:

- **Zeichen:** Byteweise, linearer Adressraum
- **Sätze:** Strukturierte Einheiten
  - Feste vs. variable Länge
  - Satzende-Markierung oder Längenfeld

5.12 Magnetbänder

Eigenschaften:

- **Sequentieller Zugriff:** Nur vorwärts/rückwärts
- **Variable Blockgrößen** möglich
- **Gaps:** Lücken zwischen Blöcken nötig
- **Wechselbare Datenträger**

Anwendung:

- **Backup:** Günstig pro MB, langsamer Zugriff OK
- **Archivierung:** Langzeitspeicherung
- **Tape Libraries:** Automatische Bandwechsler

Operationen:

- `read/write next/previous block`
- `skip n blocks forward/backward`
- `rewind to beginning`

6 KE 6: Sicherheit

6.1 Grundlagen der Sicherheit

Taxonomie der Sicherheit:

- **Schäden:** Verlust der Vertraulichkeit, Integrität, Verfügbarkeit; finanzieller Verlust; Missbrauch
- **Bedrohungen:** Unerwünschte/unerlaubte Vorgänge (unzulässige Aktionen, Fehlverhalten, Materialfehler, Eindringlinge)
- **Sicherheitsmaßnahmen:** Verhindern, begrenzen oder entdecken von Schäden

Sicherheit im engeren Sinn: Schutz vor ungewollter/unerlaubter Benutzung eines Rechners (nicht: Programmierfehler, Hardware-Defekte)

Primäre vs. Sekundäre Bedrohungen:

- **Primäre Bedrohung:** Unabhängig vom Rechner (z.B. Diebstahl)
- **Sekundäre Bedrohung:** Entstehen durch Rechnereinsatz (z.B. schwache Authentisierung)

6.2 Sicherheitsstrategien

Organisatorische vs. Automatisierte Sicherheitsstrategie:

- **Organisatorische:** Gesetze, Vorschriften, Regeln, Praktiken
- **Automatisierte:** Durch Rechner realisierte Sicherheitsmaßnahmen
- **Wichtig:** Automatisierte ergänzt organisatorische, ersetzt sie nicht!

Schichtenmodell der Sicherheit:

1. Hardware (Speicherschutz, privilegierte Modi)
2. Betriebssystem (Systemaufrufe, Dateischutz)
3. Datenbanksysteme/Middleware
4. Anwendungsprogramme

6.3 Sicherheitsklassifikationen

Orange Book (TCSEC85):

- **Klasse D:** Keine Sicherheit
- **Klasse C:** Diskretionärer Schutz (C1, C2)
- **Klasse B:** Mandatory Access Control (B1, B2, B3)
- **Klasse A:** Mathematisch beweisbar sicher

ITSEC (Information Technology Security Evaluation Criteria):

- **F1-F2:** Entspricht C1-C2 (die meisten Unix/Linux/Windows-Systeme)
- **F3-F5:** Entspricht B1-B3 (Zugriffskontrolle mit Sicherheitsstufen)
- **F6-F10:** Integrität, Verfügbarkeit, Netzwerksicherheit

6.4 Funktionsbereiche der Sicherheit

Identifikation und Authentisierung:

- **Identifikation:** Wer bist du? (Benutzername)
- **Authentisierung:** Beweise deine Identität!
- **Drei Faktoren:**
  - Wissen (Passwort)
  - Besitz (Schlüssel, Karte)
  - Eigenschaften (Biometrie)

Zugriffskontrollen:

- Rechteverwaltung und Rechteprüfung
- Kontrolle von Informationsfluss, Zugang, Ressourcennutzung
- **Least Privilege Principle:** Nur minimal benötigte Rechte gewähren

Beweissicherung/Audit:

- Protokollierung sicherheitsrelevanter Ereignisse
- Erkennung von Missbrauch und Penetrationsversuchen
- Parametrisierbar nach Art und Umfang

Speicherschutz:

- Initialisierung von Speicherbereichen vor Wiederverwendung
- Verhindert unzulässigen Informationsfluss

6.5 Benutzerverwaltung und -identifikation

Drei Stufen der Zugriffskontrolle:

1. **Zulassung als Benutzer**
2. **Benutzerprofile** (generelle Beschränkungen)
3. **Zugriffskontrollen** (objektspezifische Rechte)

Benutzerverwaltung:

- **Gespeicherte Daten:** Name, interne ID, Authentisierungsdaten, Profil, Zeitstempel
- **UID (User ID):** Eindeutige Nummer (0-65535)
- **Superuser/root:** UID 0, kann alle Schutzmechanismen umgehen

Benutzergruppen:

- **GID (Group ID):** Eindeutige Gruppennummer
- **Mitgliedschaft:** Durch Systemadministrator oder Authentisierung
- **Aktivierung:** Nicht alle Gruppen gleichzeitig aktiv
- **Strategien:** Eine Gruppe (UNIX System V), mehrere Gruppen, hierarchische Gruppen

Authentisierung:

- **Passwort-Regeln:**
  - Mindestens 8 Zeichen
  - Klein- und Großbuchstaben
  - Sonderzeichen/Zahlen
  - Nicht in Wörterbüchern
  - Regelmäßige Änderung
- **Verschlüsselte Speicherung:** Auch Superuser kann Passwörter nicht lesen

Programme als Subjekte:

- **Problem:** Zu grobkörnige Rechte für Benutzer
- **Lösung:** Programme erhalten eigene Rechte
- **SETUID-Bit:** Programm läuft mit Rechten des Besitzers, nicht des Aufrufers
- **Gefahren:** Alle Rechte des Besitzers, nicht nur benötigte!

6.6 Benutzerprofile

Generelle Benutzungsbeschränkungen:

- **Kommandosprache:** Einschränkung verfügbarer Befehle
- **Quota:** Quantitative Beschränkungen (CPU-Zeit, Speicher, Druckerpapier)
- **Sicherheitsfunktionen:** Verbot von Break-Taste, Subprozessen, Passwort-Änderung
- **Rollen:** Gast/Benutzer/Operator/Administrator mit vordefinierten Privilegien
- **Zeit/Ort:** Beschränkung auf Bürozeiten, bestimmte Terminals/Netzwerke

6.7 Zugriffskontrollen - Grundlagen

Subjekte und Objekte:

- **Subjekt:** Aktive Einheit (Benutzer, Gruppen, Programme, Prozesse)

- **Objekt:** Passive Einheit (Dateien, Verzeichnisse, Geräte, Speicherseiten)
- **Zugriffsmodi:** Gruppierung von Operationen (r, w, x, a, n, d, o)

Geschlossene vs. Offene Systeme:

- **Geschlossen:** Erlaubt ist nur, was explizit erlaubt ist (Standard)
- **Offen:** Verboten ist nur, was explizit verboten ist (unsicher)

Persistenter vs. Transienter Rechtezustand:

- **Persistent:** Überdauert Systemabschaltung (Dateien)
- **Transient:** Prozessspezifisch, vergänglich (geöffnete Dateien)

6.8 Diskretionäre Zugriffskontrollen (DAC)

Konzept:

- Zugriffskontrolle basierend auf Identität von Subjekten/Gruppen
- **Diskretionär:** Besitzer entscheidet über Rechte
- **Schwäche:** Kann Informationsfluss nicht kontrollieren!

Abstrakter Rechtezustand:

- Abbildung: `zugriff: S × O × M → W`
- S = Subjekte, O = Objekte, M = Modi, W = erlaubt, verboten
- **Zugriffskontrollmatrix:** Zeilen=Subjekte, Spalten=Objekte

6.8.1 Granulatorientierte Implementierungen

Zugriffskontrolllisten (ACL):

- **Pro Objekt:** Liste von (Subjekt, Modi)-Paaren
- **ACL-Auswertung:**
  - **Sequenz:** Erste passende Regel entscheidet
  - **Menge:** Rechte werden addiert (erlaubt, wenn mindestens ein Subjekt erlaubt)
- **Dreiwertige Logik:** erlaubt/nicht explizit erlaubt/explizit verboten
- **Benannte ACLs:** Mehrere Objekte teilen sich eine ACL

Schutzbits (UNIX):

- **User/Group/Others:** 3×3 Bits für rwx-Rechte
- **Beispiel:** `rwxr-xr--` = Besitzer: rwx, Gruppe: rx, Others: r
- **SETUID-Bit:** Ausführung mit Rechten des Dateibesitzers
- **Einschränkung:** Nur 3 Subjekttypen, begrenzte Modi

Besitzer-Konzept:

- **Owner-Modus:** Berechtigung zur Änderung der ACL
- **Exklusiver Besitzer:** Genau ein Besitzer pro Objekt
- **Delegierung:** Weiterübertragung von Rechten in Ketten

6.8.2 Subjektorientierte Implementierungen

Profile:

- **Pro Subjekt:** Liste von (Objekt, Modi)-Paaren
- **Nachteile:** Hohe Anzahl Objekte, schwierige Objektverwaltung
- **Einsatz:** Hauptsächlich für transiente Rechte

Capabilities:

- **Konzept:** SSchlüsselfür direkten Objektzugriff
- **Capability-Ticket:** Berechtigung + Objektreferenz
- **Schutz vor Manipulation:**
  - Hardware: Zusätzliches Bit pro Speicherwort
  - Betriebssystem: Capability-Listen im Kernel
  - Verschlüsselung: Geschützte Capabilities im Benutzerraum
- **Beispiel:** UNIX-Dateideskriptoren (Zeiger in globale Dateitabelle)

6.9 Informationsflusskontrolle (MAC)

Problem mit DAC: Benutzer können Informationen unkontrolliert weitergeben

Grundidee MAC: System kontrolliert Informationsfluss automatisch

6.9.1 Bell-LaPadula-Modell (Vertraulichkeit)

Sicherheitsklassen: Jedes Subjekt/Objekt hat Vertraulichkeitsstufe (Zahl)

Zugriffsregeln:

- **Einfache Geheimhaltung:** Prozess darf nur Objekte lesen, die nicht höher klassifiziert sind (No Read Up")
- ***-Eigenschaft:** Prozess darf nur in Objekte schreiben, die nicht niedriger klassifiziert sind (No Write Down")
- **Ruhe-Prinzip:** Klassifikation von Objekten kann nicht verändert werden

Kommunikationsregel: Prozess P1 darf P2 nur Daten senden, wenn Klasse(P1) $\leq$ Klasse(P2)

6.9.2 Biba-Modell (Integrität)

Dual zu Bell-LaPadula:

- **Einfache Integritätsbedingung:** Lesen nur von höherer/gleicher Integritätsstufe
- ***-Eigenschaft für Integrität:** Schreiben nur in niedrigere/gleiche Integritätsstufe

Beispiel: Leutnant kann Befehl des Generals nicht verändern

6.9.3 Erweiterte Konzepte

Gleitende Klassen:

- Automatische Anhebung der Prozess-/Objektklasse bei Bedarf
- **Problem:** Daten werden tendenziell immer geheimer
- **Lösung:** Abschaltbares Gleiten

Vertrauenswürdiges Ändern (Trusted Downgrade/Upgrade):

- Spezielle Subjekte dürfen Klassifikationen ändern
- Durchbricht automatisierte Strategie, aber nicht organisatorische
- Begrenzte Anzahl vertrauenswürdiger Personen

6.10 Sicherheitsfunktionen in der Praxis

Implementierung erfordert Hardware-Unterstützung:

- Speicherschutzmechanismen
- Privilegierte/nicht-privilegierte Modi
- Schutz von Geräte-Schnittstellen

Verteilte Spezifikation:

- Spezielle Systemaufrufe (chmod, chgrp)
- Fehlercodes in allen relevanten Systemaufrufen
- Sicherheitsaspekte in der Semantik vieler Operationen

Typische Kombination:

- **Dateien:** Diskretionäre Kontrolle (ACLs/Schutzbits)
- **Sensible Systeme:** Zusätzlich mandatory Kontrolle
- **Netzwerke:** Spezielle Übertragungssicherung

7 KE 7: Kommandosprachen

7.1 Das Starten von Prozessen

Warum Kommandosprachen?

- **Kommandosprache (Command Language):** Sprache zur Formulierung von Aufträgen an den Rechner
- **Hauptaufgabe:** Programme laden, Prozesse kreieren und starten
- Nicht nur durch Kommandointerpreter möglich, sondern durch entsprechende Systemaufrufe von jedem Prozess aus

Aufgaben beim Prozessstart:

- Neuen Prozess erzeugen und ladbares Programm in Arbeitsspeicher laden
- Parameter mitgeben (wie Prozeduraufruf)
- Rückgabewerte/Erfolg der Ausführung übertragen
- Standard-Ein-/Ausgabegeräte zuordnen
- Initiale Umgebung zur Verfügung stellen

7.1.1 Systemaufrufe zum Starten und Steuern von Prozessen

Prozesserzeugung - Zwei Ansätze:

1. Direkter Ansatz (z.B. PCTE):

```
process_create(name_ladb_prog, ...) : pid
process_start(pid, ...)
```

- Erzeugt neuen Prozessdeskriptor und logischen Hauptspeicher
- Lädt Programm, führt Initialisierungen durch
- Trennung von Erzeugen und Starten ermöglicht Änderungen vor Start

2. POSIX-Ansatz (fork + exec):

```
fork() : pid  // Erzeugt vollständige Kopie des aufrufenden Prozesses
exec(name_ladb_prog, ...)  // Lädt neues Programm in Prozess
```

- **fork():** Kopiert vollständigen logischen Hauptspeicher und Befehlszähler
- Unterschiedliche Rückgabewerte: Kindprozess erhält 0, Elternprozess erhält Kindprozess-ID
- **exec():** Ersetzt vorhandenes Programm durch neues
- Verschiedene exec-Varianten: execl, execv, execve, execvp

Weitere wichtige Systemaufrufe:

- **wait(...) : pid** - Warten auf Ende eines Kindprozesses

- **getuid() : uid** - Besitzer des Prozesses abfragen
- **getgroups() : grouplist** - Aktivierte Gruppen abfragen
- **kill(pid, ...)** - Prozess von außen abbrechen
- **exit(status)** - Prozess selbst beenden mit Rückgabewert

Prozesshierarchien:

- **Elternprozess:** Prozess, der anderen erzeugt hat
- **Kindprozess:** Erzeugter Prozess
- Baumartige Struktur durch Schachtelung möglich
- Frage: Sollen Kindprozesse automatisch beendet werden, wenn Elternprozess terminiert?

7.1.2 Die Umgebung eines Prozesses

Komponenten der Prozessumgebung:

1. Parameter:

- Wie Prozedurparameter in Programmiersprachen
- **POSIX:** Parameter als Array von Strings übergeben
- Beispiel: Übersetzer erhält Dateinamen für Quellcode, Bindemodul, Listing

2. Umgebungsvariablen:

- **Name-Wert-Paare** vom Typ String
- **Unterschied zu Parametern:** Allgemeine Daten/Einstellungen, selten verändert
- Beispiele: Benutzeridentifikation, Arbeitsverzeichnis
- **Zwei Ansätze:**
  - **Global:** Änderungen sind systemweit sichtbar (MS-DOS, problematisch)
  - **Prozess-spezifisch:** Kopiert bei Prozesserzeugung, unabhängig änderbar (empfohlen)

3. Offene Dateien:

- **Dateikontrollblock:** Interne BS-Datenstruktur für offene Datei
  - Öffnungsmodus (lesen, schreiben, anhängen)
  - Aktuelle Position des Dateizeigers
  - Nicht im logischen Adressraum, zentral verwaltet
- **Dateiidentifizierer:** Prozess-lokale Nummern (0, 1, 2, ...)
- **Vererbung:** Tabelle wird kopiert, nicht der Dateikontrollblock selbst
- Gemeinsamer Dateizeiger zwischen Eltern- und Kindprozess

Standard-Ein-/Ausgabegeräte:

- **Konzept:** Jedem Prozess zugeordnete virtuelle Geräte
- Automatisch vorhanden (müssen nicht geöffnet werden)
- **POSIX/MS-DOS Konvention:**
  - 0: Standard-Eingabegerät (stdin)
  - 1: Standard-Ausgabegerät (stdout)
  - 2: Standard-Fehlermeldungen (stderr, nur POSIX)
- **Umlenkung:** Dateien statt realer Geräte verwenden

Umlenkung von Standard-E/A-Geräten:

- **Problem:** Gleiches Programm soll mit Geräten oder Dateien arbeiten
- **Lösung:** Gleiche Systemaufrufe für Standard-E/A und Dateien
- **Beispiel UNIX:**
  - `sort < eindatei` - Eingabe aus Datei
  - `sort > ausdatei` - Ausgabe in Datei
  - `sort < eindatei > ausdatei` - Beides umleiten
- **dup2(di_alt, di_neu):** Kopiert Dateiidentifizierer

7.2 Generelle Merkmale von Kommandosprachen

Vielfalt der Kommandosprachen - Ursachen:

- Abhängigkeit von Betriebssystemkern-Funktionen
- Hardware-Eigenschaften der E/A-Geräte
- Architektur des Betriebssystems
- Unterschiede zwischen textuellen und graphischen Sprachen

Grundbegriffe:

- **Kommando:** Aufforderung an BS, bestimmten Dienst zu verrichten
- **Kommandoprozedur:** Folge von Kommandos mit Ablaufsteuerung
- **Textuelle Form:** `<kommandoverb> <Parameterliste>`
- **UNIX-Format:** `kommandoverb option argumente`

UNIX-Konventionen:

- Kommandos in Kleinbuchstaben
- Optionen ändern Wirkung (z.B. `ls -l`)
- Argumente sind Dateinamen oder Parameter
- Beispiel: `ls -l file1` - Lange Ausgabe für file1

Make-Kommando (Beispiel für komplexe Kommandoprozeduren):

- **Problem:** Abhängigkeiten zwischen Dateien in großen Programmen
- **Lösung:** Makefile beschreibt Abhängigkeiten und Aktionen
- **Funktionsprinzip:** Vergleich der Modifikationsdaten
- Nur notwendige Aktionen werden durchgeführt

Makefile-Beispiel:

```
all: forkdemo beeper

forkdemo: forkdemo.c
    cc -o forkdemo forkdemo.c

beeper: beeper.c
    cc -o beeper beeper.c
```

Betriebsarten:

- **Interaktiver Betrieb:** Ein Kommando → Eingabe → Ausführung → Ergebnis
- **Stapelbetrieb:** Kommandoprozedur als Dateiinhalt
- Moderne Systeme: Einheitliche Kommandosprache für beide Modi

Anforderungen an Kommandosprachen:

- **Effizienz:** Vollständige und effiziente Nutzung der Rechnerleistung
- **Unabhängigkeit:** So weit möglich unabhängig von Betriebsart
- **Benutzungsfreundlichkeit:** Leicht erlernbar, konsistent, Hilfesystem
- **Adaptierbarkeit:** Anpassung an individuelle Bedürfnisse und Benutzertypen

Architektonische Aspekte:

- **Auswechselbare Kommandointerpreter:** Nicht Teil des BS-Kerns
- **Shell:** UNIX-Bezeichnung für Kommandointerpreter (SSchaleüm Kern)
- **Shell-Skript:** UNIX-Bezeichnung für Kommandoprozedur
- Komplette Funktionalität muss über Systemaufrufe verfügbar sein

Betriebssysteme ohne Kommandosprache:

- Prinzipiell möglich: BS lädt immer dasselbe Programm
- Anwendung: Spezialrechner (Kopierer, Heizkessel, etc.)
- Für Mehrzweck-Rechner: Kommandosprache unverzichtbar

7.3 Kommandos

7.3.1 Phasen der Kommandoverarbeitung

Vier Phasen:

1. **Eingabe**
2. **Vorverarbeitung des Kommandoverbs**
3. **Vorverarbeitung der Parameterliste**
4. **Ausführung des Kommandos**

7.3.2 Ausführung von Kommandos

Interne vs. Externe Kommandos:

Intern:

- Ausführung innerhalb des Kommandointerpreters
- Beispiele: `cd`, `history`
- Notwendig für Kommandos, die Interpreter-Zustand ändern

Extern (als Kindprozess):

- Eigenständiger Kindprozess
- Beispiel: `date`, `ls`
- Höherer Aufwand durch Prozesserzeugung
- Aber: Isolierter Prozesszustand

Ausführung von Kommandoprozeduren:

- **Extern:** Kindprozess mit Kommandointerpreter, Datei als stdin
- **Intern:** UNIX `source` Kommando
- Extern ermöglicht Rekursion und andere Interpreter
- Intern notwendig für Änderungen am aufrufenden Interpreter

7.3.3 Eingabe von Kommandos

Textuelle vs. Graphische Eingabe:

Textuelle Kommandos:

- Alphanumerische Zeichen über Tastatur
- Beispiel: `rm a.out`
- Editiervorgang mit Cursor-Bewegung, Einfügen, Löschen

Graphische Kommandos:

- Erfordern Graphikfähiges Ausgabemedium und Zeigegeräte
- Beispiel: Drag & Drop zum Löschen
- Escape-Sequenzen für Maus-/Tastatureingaben
- Komplexere Zustandsverwaltung erforderlich

Hilfen bei interaktiver Eingabe:

Abkürzungen:

- Ausreichend langes eindeutiges Präfix
- Automatische Ergänzung bei Eindeutigkeit
- Kurz- und Langformen möglich

Wiederholung früherer Kommandos:

- Cursor-Tasten für Kommando-History
- `!n` für n-tes Kommando (C-Shell)
- `history` Kommando listet vergangene Kommandos

7.3.4 Vorverarbeitung des Kommandoverbs

Erweiterbare Kommandointerpreter:

- Drei Mechanismen: Ladbares Programm, Kommandoprozedur, direkt im Interpreter
- **Moderne Systeme:** Einheitliche Syntax für alle drei Arten
- Unterscheidung durch Dateiname-Suffix oder Dateiinhalt-Inspektion
- **Vorteil:** Leicht erweiterbar durch neue Programme/Skripte

Suchpfade:

- **Problem:** Programme in verschiedenen Verzeichnissen organisiert
- **Lösung:** PATH-Variable mit Liste von Verzeichnissen
- Suche in angegebener Reihenfolge
- **Optimierung:** Pufferung in Zuordnungstabelle (rehash notwendig)

Aliase:

- **Zweck:** Einfache Umbenennung oder Default-Optionen
- Textueller Ersatz vor Programmstart
- Schachtelung möglich (Zyklenerkennung notwendig)
- **Beispiel:** `alias dir=ls`

7.3.5 Verarbeitung der Parameter

Parameterübergabeverfahren:

Stellungsparameter:

- Position bestimmt Zuordnung (wie in Programmiersprachen)
- i-ter aktueller Parameter → i-ter formaler Parameter
- Gut für kurze Parameterlisten

Namensparameter:

- Explizite Benennung: `option7=yes`
- Beliebige Reihenfolge möglich
- Nicht angegebene Parameter haben Default-Werte
- Gut bei vielen optionalen Parametern

Mengen von Dateien als Parameter:

- **Beispiel:** `lpr file1.pr file2.pr file3.pr`
- Variable Parameterlisten-Länge erforderlich
- **Wild Card Characters** für Pattern-Matching:
  - `*` - beliebige Zeichenfolge
  - `?` - ein beliebiges Zeichen
  - `[ ]` - eines der angegebenen Zeichen
- **Beispiel:** `lpr *.pr` für alle .pr-Dateien

Expansion der Sonderzeichen:

- **UNIX-Shells:** Expansion im Kommandointerpreter
  - Vorteil: Einheitliche Bedeutung, einmalige Implementierung
  - Nachteil: Probleme mit noch nicht existierenden Dateien
- **MS-DOS:** Expansion im Anwendungsprogramm
  - Vorteil: Flexibler für spezielle Anwendungen
  - Nachteil: Inkonsistente Implementierungen

Kommandosubstitution:

- Ersetzen eines Kommandos durch seine Ausgabe
- **Notation:** `` `command` `` oder `$(command)`
- **Beispiel:** `set prompt = \$username@`hostname`:`
- Ermöglicht sehr mächtige Textverarbeitung in Skripten

7.4 Variablen und Kontrollstrukturen

7.4.1 Variablen

Shell-Variablen:

- **Typ:** Meist nur Strings (Text über bestimmtem Alphabet)
- Deklaration jederzeit möglich
- Ganzzahl-Operationen oft möglich wenn Wert numerisch
- **Array-ähnlich:** Zugriff auf n-tes Wort mit `variablenname[n]`

Wertzuweisung:

- **Beispiel:** `set variablenname = neuerwert`
- Intern ausgeführt (kein Kindprozess)

Wertverwendung:

- **C-Shell:** `$variablenname`
- Textueller Ersatz bei Kommandoverarbeitung
- **Beispiel:**

```
set XY = /usr/local/XY
$XY/load $XY/fonts $XY/init
```

Vordefinierte Variablen (C-Shell):

- **argv:** Parameter beim Shell-Aufruf
- **cwd:** Aktuelles Arbeitsverzeichnis
- **filec:** Schalter für automatische Dateinamen-Ergänzung
- **home:** Heimverzeichnis des Benutzers
- **path:** Suchpfad für ausführbare Dateien

Shell-Variablen vs. Umgebungsvariablen:

- **Shell-Variablen:** Nur lokal im Kommandointerpreter
- **Umgebungsvariablen:** Werden an Kindprozesse vererbt
- Kopieren zwischen beiden möglich

7.4.2 Ablaufsteuerung

Kontrollstrukturen wie in Programmiersprachen:

- Schleifen, Verzweigungen, Prozeduraufrufe
- Oft modifiziert für Kommando-spezifische Bedürfnisse

For-Schleife (Bourne-Shell):

```
for i in *
do
  cp $i $i.alt
done
```

- Kopiert jede Datei im aktuellen Verzeichnis
- `*` wird in Dateinamen expandiert
- Hinter `in` kann auch explizite Liste stehen

Erweiterte For-Schleife mit Kommandosubstitution:

```
for i in `grep kap namensliste`
do
  cp $i $i.alt
done
```

- Nur Dateien, deren Name "kapënthält und in namensliste steht
- `grep` findet Zeilen mit "kap"
- Kommandosubstitution liefert Dateinamen-Liste

Die Mächtigkeit der Kommandosprachen:

- UNIX-Shells erreichen Mächtigkeit konventioneller Programmiersprachen
- Kommandosubstitution ermöglicht Integration beliebiger Programme
- **Sprachverbund:** Kommandosprache + C + Textprozessoren (grep, awk, sed)
- Entwicklung großer Systeme möglich (besonders Prototypen)

7.4.3 Softwaretechnische Aspekte

Entwicklung von Kommandoprozeduren:

- Prinzipiell wie normale Programmentwicklung
- Softwaretechnik-Methoden anwendbar
- **Problem:** Meist kleine Prozeduren (wenige hundert Zeilen)
- Aber moderne Kommandosprachen ermöglichen große Systeme

Entwicklung der Kommandoprozeduren:

- Früher nur kleine, lineare Skripte
- Heute: Vollwertige Entwicklungsumgebung möglich
- BS wird zur Software-Entwicklungsumgebung
- Integration verschiedener Sprachen in einem System

Performance-Überlegungen:

- Interpretative Abarbeitung langsamer als übersetzte Programme
- Aber: Sehr schnelle Entwicklung und einfache Änderungen
- Gut für Prototypen und nicht performance-kritische Anwendungen

7.5 Zusammenfassung

Kernpunkte der Kommandosprachen:

- **Prozesserzeugung:** Systemaufrufe wie fork/exec oder process_create
- **Prozessumgebung:** Parameter, Umgebungsvariablen, offene Dateien
- **I/O-Umlenkung:** Standard-E/A auf Dateien umleiten
- **Kommandoverarbeitung:** Eingabe → Vorverarbeitung → Ausführung
- **Erweiterbarkeit:** Suchpfade, Aliase, externe Programme als Kommandos
- **Textverarbeitung:** Wild Cards, Kommandosubstitution

- **Programmiersprachen-Features:** Variablen, Kontrollstrukturen

Moderne Kommandosprachen:

- Erreichen Mächtigkeit konventioneller Programmiersprachen
- Ermöglichen Entwicklung komplexer Systeme
- Integration verschiedener Werkzeuge und Sprachen
- Wichtige Komponente moderner Betriebssysteme