1 Einführung in UNIX

Mehrbenutzer- und Mehrprozess-Betriebssystem

1.1 Aufbau eines UNIX-Systems

Aufbau eines UNIX-Systems:

- Hardware: Computersystem + Peripheriegeräte
- Software
  - Betriebssystem: **Kernel** (von spezifischer Hardware unabhängige Systemsicht) + Gerätetreiber, stellt Funktionalität durch **Systemcalls** zur Verfügung:
    - Filesystem
    - Prozessor- und Speicherverwaltung
    - Netzwerkkommunikationssystem
    - Ein-/Ausgabesystem
  - Applikationsprogramme: **Shell**, Compiler, Browser, …

1.2 Unix-Shell

Aufgaben:

- Benutzer-Kommandos verarbeiten → entsprechende Programme ausführen
- Shellscripts (Programme in spezieller Programmiersprache) ausführen

Gebräuchliche Shells:

- Bourne-Shell (sh): erste Shell
- **C-Shell (csh):** C-ähnliche Syntax
- Korn-Shell (ksh)
- Bourne-Again-Shell (bash)

Shell-Prompt:

- meist Zeichen $, zeigt Warten auf Kommandoeingabe
- Befehle: $ Befehl [Optionen beginnen mit -] [Files], Abbruch mit STRG + C

Shell-Variable:

- durch erste Zuweisung erzeugt: FN=/usr/useres/01/s8225607
- immer vom Typ String, kein Leerzeichen vor und nach =
- Wert abfragen: ${FN}
- vordefiniert:
  - HOME: Home-Verzeichnis
  - PATH: Suchpfad für Programme
  - USER: Benutzername
  - ?: letzter Return-Wert

Expansion von Datei- und Kommandoname:

- Anfangsbuchstaben + Tab
- Auflistung aller möglichen Varianten oder Auto-Auswahl der eindeutigen Auflösung

Ein- und Ausgabeumleitung:

- stdin = 0, stdout = 1, stderr = 2
- $ Kommando < infile > outfile
- Umleitung der Fehlermeldungen: $ Kommando 2> Fehlerfile
- Anhängen der Ausgabe an ein File: >>
- Umleitung Standardausgabe → Standardfehlerausgabe: $ Kommando 1>&2

(Unnamed) Pipes:

- Vermeiden von temporären Zwischenfiles
- $Kommando_1$ | $Kommando_2$ | … | $Kommando_n$

Signale:

- asynchrone Ereignisse, Unterbrechung des Programmablaufs
- Reaktion:
  - benutzerdefinierte Aktion
  - ignorieren
  - Standardaktion
- Beenden von Prozessen:
  - kill –TERM pid
  - kill –KILL pid
- Erzeugen mittels Tastatur:
  - STRG + Z: stoppt aktuellen Vordergrundprozess
  - STRG + C: terminiert aktuellen Vordergrundprozess
  - STRG + D: erzeugt EOF (kein Signal)

Anführungszeichen:

- doppelte “” (Double Quotes): String, Variablensubstitution aber keine Generierung von Dateinamen
- einfache ‘‘ (Single Quotes): String, keine Variablensubstitution oder Generierung von Dateinamen
- verkehrte ´´ (Grave Quotes): Kommandoersetzung, Kommando ausgeführt, dann Konstrukt durch Ausgabe ersetzt

Mehrere Kommandos:

- com1 ; com2: hintereinander
- com1 && com2: com2 nur wenn com1 erfolgreich
- com1 || com2: com2 nur wenn com1 Fehler
- com1 &: Hintergrund
- com1 & com2: com1 im Hintergrund, com2 im Vordergrund
- (com1 ; com2): beide in einer Shell

1.3 Dateisystem

- baumförmig
- Dateinamen bis zu 255 Zeichen, keine Dateierweiterungen
- **Root-Directory:** /, Pfade mit /, nicht \
- **Navigation: cd** Filename (absolut/relativ)

Zugriffskontrolle:

- Usergruppen:
  - user
  - group
  - others
- Zugriffsarten:
  - read
  - write
  - execute
- [d = directory, - = File]rwxrwxrwx
- **chmod** permissions file
  - **chmod** 777 file
  - **chmod** u=rwx, g+wx, o-rwx file

Links:

- Verweis/Link auf Datei erzeugen
- **hardlinks:**
  - unterscheiden sich nicht vom Original, wenn Original gelöscht bleibt Datei erhalten solang noch mindestens ein hardlink existiert
  - **ln** datei1 datei1.lnk
- **symbolic links:**
  - eigene kleine Dateien, Inhalt = Pfadname der Zieldatei, ungültig wenn Ziel umbenannt/gelöscht
  - **ln -s** datei2 datei2.lnk

Wildcards:

- * beliebige (auch leere) Zeichenfolge
- ? beliebiges einzelnes Zeichen
- [abx] beliebiges Zeichen aus der Liste
- [a-z] Zeichenintervall
- escapen mit \

1.4 Wichtige Kommandos

- Dateien kopieren: **cp**
- Datei löschen: **rm**
- Zeilen einer Datei sortieren: **sort**
- Datei nach Muster durchsuchen: **grep**
- Dateien verschieben/umbenennen: **mv** source destination
- Verzeichnis erzeugen: **mkdir**
- Verzeichnisinhalte auflisten: **ls**
- leeres Verzeichnis löschen: **rmdir**
- Files kompilieren: **gcc**
- Managen mehrerer Sourcefiles:
  - **make –f** makefile
  - zum Übersetzen: **gcc –ansi –pedantic –Wall –c** file.c
  - Target: Dependencies
    <tab> Command
  - all: program
  - clean: **rm –f** program module1.o module2.o
- Manualpages: **man** befehl

Packen, Entpacken, Archivieren:

- Komprimieren einzelner Dateien:
  - **gzip** archiv.tar: speichert archiv.tar komprimiert als archiv.tar.gz
  - **gunzip** archiv.tar.gz
  - bzip2 archiv.tar: speichert archiv.tar komprimiert als archiv.tar.bz2 (effizienter, aber langsamer)
  - **bunzip2** archiv.tar.bz2
- Zusammenfassen mehrerer Dateien zu einer einzigen:
  - **tar -cvvf** archiv.tar datei1 datei2 …: packen
  - **tar -xvvf** archiv.tar: entpacken
  - **zip** archiv.zip datei1 datei2 …
  - **unzip** archiv.zip

1.5 Shell-Programme

müssen ausführbar und lesbar sein (normale Binärprogramme nur ausführbar)

2 Mechanismen von UNIX/Linux

2.1 Filesystem

- Directory ist Sonderform eines Files
- **Special Files:**
  - **Named Pipes (FIFOs):** verhalten sich nach dem Öffnen wie Pipes
  - **Sockets:** Interprozesskommunikation über Computer hinweg, zB mittels TCP/IP
  - **Symbolic Links**
  - **Mounting:** Einbinden anderer Dateisysteme (eingebundenes Filesystem ersetzt Directory)

2.1.1 Virtuelles Dateisystem

Abstraktion von physikalischem Dateisystem (ext, ext2, ext3, ntfs, …)

VFS Superblock:

- jedes gemountete Filesystem wird durch einen repräsentiert
- Device-Nummer
- Zeiger auf ersten i-node
- Zeiger auf i-node des Directorys in das das Filesystem gemountet wird
- Blockgröße des Filesystems
- Typ des Dateisystems

VFS i-node:

- jedes File durch einen repräsentiert
- Device-Nummer + (innerhalb des Filesystems eindeutige) i-node-Nummer
- Filetyp (normales File, Directory, Special File)
- Zugriffsrechte, Besitzer, Gruppe des Besitzers
- Anzahl der Referenzen auf das File
- Größe, Anzahl der Blocks
- letzter Zugriff, letzter Schreibzugriff, letzte Änderung

2.1.2 Second Extended Filesystem (EXT2FS)

Plattenorganisation:

- jede Platte in mehrere **Zylindergruppen** (Gruppe nebeneinander liegender Zylinder) unterteilt
- **Zylinder** besteht aus allen übereinanderliegenden Spuren der Platte (Zugriff auf alle Daten ohne Bewegung des Lesekopfs)
- Zylindergruppe enthält:
  - eigene redundante Kopie des Superblocks (kritische Daten)
  - Zylindergruppenblock (beschreibt Struktur der einzelnen Zylindergruppen)
  - i-nodes
  - Datenblöcke
- Motivation für Zylindergruppen: i-nodes und Datenblöcke möglichst knapp beieinander

Auswahl der Blockgröße:

- große Blöcke: schneller Datentransfer (keine zeitraubenden Bewegungen des Lesekopfs), aber größere Platzverschwendung (im Schnitt halber Block pro Datei ungenutzt)
- Lösung große Blöcke + **Fragmente**:
    - Datenblock entweder ohne Unterteilung einem File zugeordnet oder in verschiedenen Files zugeordnete Fragmente vordefinierter Größe aufgespalten
    - nur Ende einer Datei kann in Fragment liegen
    - Nachteil: höhere CPU-Belastung

Allokation von Plattenblöcken:

- Globale Algorithmen:
    - i-nodes aller Files eines Directorys möglichst in derselben Zylindergruppe wie das Directory (neues Directory in Zylindergruppe mit möglichst vielen freien i-nodes und möglichst wenigen Directorys angelegt)
    - Datenblöcke eines Files möglichst in derselben Zylindergruppe wie i-node des Files
    - sehr große Daten über mehrere Zylindergruppen verteilt
    - fordern teilweise bereits belegte Blöcke an (sonst zu langsam)
- Lokale Algorithmen:
    - tatsächliche Zuteilung eines Plattenblocks an eine Datei (möglichst nahe an angeforderten)
    - eventuell Durchsuchen der ganzen Platte (Filesystem sollte nur zu bestimmtem Prozentsatz gefüllt sein)

EXT2FS-Superblock:

- beschreibt Größe und Struktur des Filesystems, in jedem Zylindergruppenblock repliziert vorhanden
- Revisionsnummer (von Filesystemversion unterstützte Dienste)
- Zylindergruppen-Nummer
- Blockgröße in Bytes, Anzahl der Blöcke pro Zylindergruppe, Anzahl der freien Blöcke
- Anzahl der unbenutzten i-nodes, i-node-Nummer des ersten i-nodes des Filesystems

EXT2FS-i-node: zusätzlich zu VFS-i-node Zeiger auf Datenblöcke (erste zwölf direkt, 13. auf einfach indirekten Block, 14. auf zweifach indirekten Block)

EXT2FS-Zylindergruppenblock:

- redundant in jeder Zylindergruppe
- Anzahl der freien Blöcke, i-nodes und benützten Directorys in der Zylindergruppe
- Blocknummer der Blöcke zur Speicherung der Block-Bitmap, der i-node-Bitmap und des Starts der i-node-Tabelle

Directorystruktur:

- einzige Informationen über ein File im Directory: Filename, Länge des Filenames, Länge des gesamten Directoryeintrags , Nummer des zugeordneten i-node
- Löschen eines Filenames: i-node-Nummer auf 0 gesetzt

2.1.3 Interne Verwaltung von Files

Tabellen zum Zugriff:

- **File-Descriptor-Tabelle:** liegt im Prozesskontext, enthält Eintrag für jedes offene File des Prozesses, Systemcalls open() und dup() liefern Index
- File-Descriptor enthält Zeiger auf Eintrag in **File-Tabelle** im Kernel-Kontext:
- open() erzeugt neuen Eintrag
- Zugriffsmodus
- Position des Schreib- bzw. Lesezeigers
- Eintrag in File-Tabelle zeigt auf Eintrag in **i-node-Tabelle:** resident im Kernel, genau einen Eintrag für jedes von mindestens einem Prozess geöffnete File

Verwaltung der i-nodes:

- i-nodes aller geöffneten Files resident in Hashtabelle im Kernel
- nicht mehr aktive bleiben in Liste, bis Platz für neuen i-node benötigt
- Referenzzähler

Systemprozeduren zum Dateizugriff:

- **namei():** wandelt Pfadnamen in i-node-Nummer um

2.2 Prozessverwaltung

Prozess:

- Ausführung eines Programms
- **Task-List: ps**

Prozesskontext: über Prozess gespeicherte Information

- auf Benutzerebene: für Programm selbst wichtig
- auf Betriebssystemebene: Parameter für Scheduling, Informationen über belegte Ressourcen, Speicherbelegung, Identifikation des Prozesses

Prozesszustände:

- **Waiting:** wartet auf Ereignis, durch Signal unterbrechbar? → interruptable/non-interruptable waiting
- **Running:** lauffähig
- **Swapping:** wird gerade erzeugt
- **Zombie:** hat terminiert, Status noch nicht vom Elternprozess empfangen
- **Stopped:** gestoppt

2.2.1 Datenstrukturen zur Prozessverwaltung

Prozesstabelle:

- ein Eintrag pro laufendem Prozess
- besteht aus Einträgen der task_struct-Struktur
- immer im Kernel resident
- verkettet Prozesse in Queues (**Run Queue**) und Eltern-Kind-Hierarchien

tasc_struct-Struktur:

- **Scheduling:** Priorität (Real-Time-Prozesse – Non-Real-Time-Prozesse), CPU-Zeit, …
- **Identifikation:** PID, Owner, Gruppe, …
- **Speicherverwaltung**
- **Synchronisation: Wait Queue** gibt Bedingung an, auf die Prozess wartet
- **Signale:** an Prozess geschickte Signale, auszuführende Aktionen
- **Berechtigungen**
- **Resource Accouting:** Informationen über Betriebsmittelverwendung, Plattenlimits (**Disk Quota**)
- **Timer-Verwaltung**
- **Process Control Block (PCB):**
  - Prozesszustand im User- bzw. Kernelmode wird bei Kontextwechsel im PCB gespeichert
  - prozessorabhängiger Teil, Page Tables für virtuelle Speicherverwaltung
- **Deskriptorentabelle:** verwendete Filedeskriptoren
- **Kernelstack:**
  - Betriebssystemstack für Abarbeitung von Systemcalls und Traps (**Top Half**)
  - eigener Interrupt Stack für Abarbeitung von Interruptroutinen (**Bottom Half**), die keinem Prozess zugeordnet werden können

Process Groups: Zusammenfassung zB für Signalbehandlung → Signale an alle Prozesse der Gruppe

Speicherverwaltung:

- **Page Tables:** Welche Teile des virtuellen Speicherraums des Prozesses befinden sich gerade im physikalischen Hauptspeicher und wo?
- **Text-Structure:** beschreibt Aufbau des geladenen Programmcodes, stellt ihn mehreren Prozessen gleichzeitig zur Verfügung

2.2.2 Schedulingmechanismen

Kontextwechsel (Context switch):

- Umschaltung zwischen zwei Prozessen
  - freiwillig: Prozess ruft System-Call **sleep_on()** auf
  - unfreiwillig: Ende der Zeitscheibe
- CPU zur Behandlung eines asynchronen Ereignisses gebraucht (zB Interrupt von einem Device)
- Austausch der task_struct-Struktur

Kontextsicherung:

bei Kontextwechsel Sicherung der Zustandsinformation des alten Prozesses in der task_struct-Struktur:

- User-Mode-Prozessorstatus: auf Kernel-Stack bei Eintritt in Kernel-Mode
- Kernel-Mode-Prozessorstatus: im **PCB** der task_struct-Struktur bei freiwilligem Kontextwechsel (kann sonst nicht unterbrochen werden)

Synchronisation:

- locked und wanted-Flag
- Prozess will auf Ressource zugreifen, wenn **locked** nicht gesetzt, setzt er es und verwendet die Ressource
- ist es gesetzt, setzt er **wanted** und ruft **sleep_on()** mit einer Wait Queue auf, die die Ressource beschreibt
- benötigt ein Prozess eine Ressource nicht mehr, löscht er **locked** und wenn **wanted** gesetzt ruft er **wake_up()** um wartende Prozesse zu wecken

Schedulingalgorithmen:

- Prozessen wird abhängig von ihrer Priorität eine fixe Zeitscheibe zugeteilt
- Handling des aktuellen Prozesses:
  - verbleibende Zeitscheibe = 0 → Zeitscheibe = Priorität, ans Ende der Run Queue
  - Status des aktuellen Prozesses = interruptable waiting, erhält Signal → Status = running
  - Status != running → aus Run Queue entfernt
- Selektion des nächsten Prozesses
  - Prozess mit höchster goodness aus Run Queue gewählt
  - Run Queue leer oder kein Prozess mit goodness > 0 → **idle process**
- Aktivierung des selektierten Prozesses: falls selektierter Prozess != laufendem → Context Switch
- **goodness:**
  - Real-Time-Prozess: 1000 + Priorität
  - gerade laufender Prozess: noch verbleibende Zeitscheibe + 1
  - alle anderen: noch verbleibende Zeitscheibe

2.2.3 Erzeugen von Prozessen

- System-Call **fork()**: Kindprozess ist genaue Kopie des Elternprozesses, liefert im Elternprozess Prozessnummer des Kindprozesses, im Kindprozess 0
- Anlegen einer neuen task_struct-Struktur in der Taskliste
- Duplizieren der task_struct-Struktur des Elternprozesses (außer PID)
- Erzeugen einer Referenz auf die Ressourcen des Elternprozesses
- Anlegen einer Wait Queue um auf Terminierung eigener Kindprozesse warten zu können
- Scheduling des Kindprozesses

2.2.4 Beenden von Prozessen

- Freigabe aller Ressourcen, Speichern des Exit-Status in task_struct, Status = Zombie, Elternprozess wird mittels Signal informiert und mittels **wake_up()** aufgeweckt
- Kindprozess terminiert, Elternprozess wird mittels **wake_up()** aufgeweckt, erhält gesamte Information über Status des Kindprozesses

2.2.5 Threads

- mehrere Threads in einem Prozess, teilen sich Betriebsmittel, haben aber eigenen Stack
- billiger als Prozesse + Shared Memory, keine explizite Shared Memory-Erzeugung nötig
- Bug in einem Thread kann ganzen Prozess zum Absturz bringen

2.3 Speicherverwaltung

- Aufgabe: verschiedene Speichertypen (Cache, Hauptspeicher, Plattenspeicher, Remote Fileserver) als homogenen **virtuellen Speicher** erscheinen lassen
- Anforderungen:
  - größer als physikalischer Hauptspeicher, nicht wesentlich langsamer
  - mehr Prozesse gleichzeitig als im Hauptspeicher Platz haben
  - Speicherbereich eines Prozesses vor Zugriff anderer Prozesse geschützt
  - Erweiterung des Speicherraums unsichtbar für Benutzer

virtueller Adressraum:

- Kernel: evtl. physikalischer Adressmodus
- Userprozess: virtueller Adressmodus

Adressumsetzung:

- von der **Memory Management Unit (MMU)** mittels Page Tables durchgeführt
- prozessorabhängig → abstrahiert durch **Makros** (verdecken Anzahl der Seitentabellen, Struktur eines Seitentabelleneintrags und einer virtuellen Adresse)

Mechanismen:

- **Swapping:**
  - Aus- (**Swap Out**) bzw. Einlagerung (**Swap In**) ganzer Prozesse Hauptspeicher → Sekundärspeicher
  - zeitaufwendig, Prozesse müssen zur Abarbeitung gänzlich im Hauptspeicher sein
- **Paging:**
  - Spezialhardware (**MMU: Memory Management Unit**) übersetzt logische in physikalische Adressen = **Paging**
  - physikalischer Speicher in Seitenrahmen gleicher Größe unterteilt (kann eine Seite aufnehmen)
  - nicht unbedingt alle Seiten eines Prozesses gleichzeitig im Hauptspeicher vorhanden → angeforderte Seite von speziellem Bereich des Sekundärspeichers (**Swap Space**) gelesen, nicht benötigte zurück geschrieben

Seitenfehler (Page Fault): bei Adressumsetzung festgestellt, dass Seite nicht im Hauptspeicher vorhanden → Programmabarbeitung unterbrochen, Seite in Hauptspeicher geladen

2.3.1 Verwaltung des Hauptspeichers

mem_map-Struktur: Seitenrahmen des physikalischen Speichers

- Anzahl der Referenzen auf diesen Seitenrahmen
- Alter des Seitenrahmens (Zeit seit letztem Zugriff)
- Nummer des physikalischen Eintrags

free_area-Struktur:

- Blöcke (freier) Seitenrahmen
- 1. Eintrag: Blöcke der Größe 2^0, 2. Eintrag: Blöcke der Größe 2^1, …
- doppelt verkettete Listen von Blöcken der jeweiligen Größe

Buddy-Verfahren: Allokation freier Seitenrahmen

- wenn vorhanden Block der gewünschten Größe zur Verfügung gestellt
- ansonsten Block der nächst größeren Größe geteilt → free_area-Struktur aktualisiert
- bei Freigabe wenn möglich benachbarte kleine Blöcke zusammen gefasst

2.3.2 Speicherverwaltung beim Erzeugen von Prozessen

- **fork():** identische Prozesskopie erzeugt:
  - Anlegen des **Swap Space** für Kindprozess
  - neuer **task_struct**-Tabelleneintrag
  - Anlegen der Seitentabellen
  - gesamter Adressraum dupliziert → zeitaufwendig
- **copy-on-write-Verfahren:**
  - nur Seitentabelleneinträge kopiert → Prozesse greifen auf selbe Seiten zu
  - erst wenn ein Prozess auf eine Seite schreiben möchte, wird sie dupliziert

2.3.3 Austauschstrategie

- nicht mehr gebrauchte und veränderte (**dirty-Bit** gesetzt) Seiten in Bereich des Sekundärspeichers (**Swap Space**) zurück geschrieben
- **Pagedaemon** überprüft periodisch Seitenrahmen, lagert vermutlich in nächster Zeit nicht benötigte Seiten aus, nur aktiv wenn freier Speicher Schwellwert unterschreitet
- **LRU (Last Recently Used):**
  - am längsten nicht verwendete Seite ausgelagert
  - in der Praxis nur gespeichert, ob überhaupt zugegriffen wurde (**Page Reference Bit**) → Clock-Algorithmus
- **Clock-Algorithmus:**
  - benötigt Referenzbit (**age-Bit**)
  - zyklische Untersuchung des physikalischen Speichers, beim ersten Umlauf alle Referenzbits gelöscht, beim zweiten alle die noch immer gelöscht sind, ausgelagert
  - bis zum Entfernen vergeht mindestens eine Umlaufzeit → **Two Handed Clock-Algorithmus:** zwei Zeiger in konstantem Abstand, erster löscht Referenzbit, zweiter lagert aus

2.3.4 Demand Paging

- Systeme ohne virtuelle Speicherverwaltung müssen beim Programmstart gesamtes Programm in Hauptspeicher laden
- Seiten erst geladen, wenn gebraucht
- bei Prozesserzeugung im Swap Space Platz für gesamten Prozess reserviert
- bei Zugriff Seite direkt vom ausführbaren File in physikalischen Speicher eingelagert

2.3.5 Speicherverwaltung beim Ausführen von Programmfiles

Austausch des gesamten Adressraums des aufrufenden Prozesses gegen einen anderen (**exec()**):

- Platz im Swap Space angelegt
- Adressraum des Prozesses mit Ausnahme der **task_struct**-Struktur freigegeben
- Seitentabellen auf richtige Größen für neuen Adressraum gebracht
- Text-Struktur des Files gesucht und falls nicht vorhanden neu angelegt

2.4 Verwaltung der Peripherie

- Kernel hat Liste aller Geräte vom Typ **device_struct**
- jedes Gerät hat Tabelle mit Sprungadressen von Treiberfunktionen

Zugriff über **Device Driver:**

- **Character Device Driver:** operieren auf Datenstrom
- **Block Device Driver:** operieren auf einem/mehreren Datenblöcken
- **Network Interface Driver:** Transaktion von Datenpaketen

Device:

- Zugriff über **Special File** (normalerweise in **/dev** abgelegt)
- **Major Device Number:** Typ eines Devices → zu verwendenden **Device Driver**
- **Minor Device Number:** genaues Gerät (Teile eines Geräts, zB Festplattepartion), Treiber als Parameter übergeben

2.4.1 Device Driver

- Sammlung von Routinen zum Zugriff auf ein Peripheriegerät
- muss abhängig von der Art (Block Device Driver oder Character Device Driver) gewisse Funktionalität bereitstellen
- statische (bei Initialisierung des Kernels) – dynamische Registrierung
- Struktur:
    - **Top Half:** im Kontext des aufrufenden Prozesses, durch System Call aktiviert, initiiert Datentransfer, bei Lese- oder synchronen Schreiboperationen auf Beendigung gewartet
    - **Bottom Half:** in eigenem Kontext asynchron zum Benutzerprozess, behandelt Interrupt (nach Beendigung des Datentransfers vom Gerät ausgelöst), weckt Top Half wieder auf

Block Device Driver:

- wahlweiser Zugriff auf Blöcke fixer Größe über Block Buffer Cache (zB Festplatten)
- wichtigste Routinen:
    - **open(), release()**
    - **strategy():** Ordnung der Lese-/Schreibzugriffe
    - **ioctl():** Einstellen Device-spezifischer Parameter
- Lesen und Schreiben von Blöcken in sinnvoller und effizienter Reihenfolge → Strategie:
    - **Evelvator-Algorithmus:** Request, der in momentaner Richtung als nächstes folgt, als nächstes bearbeitet
    - **C-Scan-Algorithmus:** Requests werden nur bei einer festgelegten Richtung bearbeitet → innere und äußere Blöcke nicht benachteiligt, Requestdichte am Anfang und Ende einer Kopfbewegung gleich groß

Character Device Driver:

- **Raw Devices:**
  - wahlfreier Zugriff auf Peripheriegeräte (ohne Block Buffer Cache)
  - keine Organisation in fixe Blockgrößen notwendig, oft auch raw Zugriffe über Block Devices (zB Harddisk)
- **Character Oriented Devices:**
  - zeichen- bzw. zeilenorientierter Zugriff
  - zB Drucker
- wichtigste Funktionen:
  - **open(), release(), ioctl()**
  - **read(), write():** einzelnes oder blockweises Lesen/Schreiben
  - **select():** Überprüfung ob ausreichend Platz zum Schreiben bzw. Daten zum Lesen vorhanden
- **Line Disciplines:**
  - Implementierung genereller Routinen, weder vom Device Driver noch vom Programm abhängig, zB Behandlung des Meta-Zeichens kill bei Terminals
  - einstellen mittels **ioctl()**

2.4.2 Streams

- Character Devices behandeln Datenstrom zeichenweise, Netzwerkkommunikation nachrichtenorientiert → beeinträchtigt Performance
- im Filesystem: **Character Special Files**
- bidirektionale Verbindung Prozess – Treiber
- besteht aus Anzahl von Modulen (Stream Head – n Module – Device Driver), jedem zwei Queues (Eingabe, Ausgabe) zugeordnet, Daten wandern durch alle Module
- Prozeduren:
  - **open(), close()**
  - **put():** Übergabe einer Nachricht an ein Modul
  - **service():** Bearbeitung der Nachrichten in der lokalen Queue des Moduls
- Nachrichten können Prioritäten zugewiesen werden → Vorreihung in der Queue

2.5 Interprozesskommunikation

- Synchronisation, Datenaustausch
- Möglichkeiten:
  - Signale
  - Unnamed Pipes, Named Pipes
  - Semaphore
  - Shared Memory
  - Message Queues
- verwandte Prozesse – Prozesse auf gleichem Rechner/Prozessor – Prozesse auf verschiedenen Rechnern/Prozessoren

2.5.1 Signale

- Behandlung von Ausnahmezuständen
- Reaktionen:
  - ignorieren
  - Prozess terminieren
  - Erzeugen eines Speicherabbildes
  - Prozess anhalten
  - benutzerdefinierte Signalbehandlungsroutine
  - Signale können blockiert werden → empfangen, gespeichert, nach Aufhebung der Blockierung ausgeführt
- Implementierung:
  - task_struct-Struktur: Bitfelder **signal** und **blocked**, Array von Zeigern auf benutzerdefinierte Signalbehandlungsroutinen
  - Ausführen bei Prozessscheduling: Check auf eingetroffene und nicht ignorierte Signale, Festlegen der auszuführenden Aktion

2.5.2 System V IPC (Semaphore, Shared Memory, Message Queues)

- Synchronisation und Kommunikation zwischen unabhängigen Prozessen
- nicht netzwerkfähig
- **Semaphore:**
  - binäre, counting Semaphore
  - Semaphorefelder werden in **Atomic Actions** gesetzt/abgefragt
- **Shared Memories:** Mapping in Adressbereich der Prozesse
- **Message Queues:** Nachrichtentypen zur Filterung oder für Prioritäten

2.5.3 Sockets

- Kommunikation zwischen unabhängigen Prozessen
- **Transparenz:** auch auf verschiedenen Computern (allgemeines Interface zu Netzwerkprotokollen)
- **Multicasting:** IP-Multicasting bietet keine zuverlässigen Verbindungen
- **ISO-Referenzmodell:** Session Layer
- Parameter:
  - Domain (UNIX, Internet)
  - Typ (Stream, Datagram: einzelne Datenpakete, Raw)
  - Protokoll (TCP/IP, UDP/IP)

2.6 Networking

Application Layer	Benutzerprogramme	
Presentation Layer		
Session Layer	Sockets	
Transport Layer	Netzwerk-Protokolle	TCP, UDP
Network Layer		IP
Data Link Layer	Netzwerk-Interfaces	
Physical Layer	Netzwerk-Hardware	

Netzwerk-Interface:

- real (Netzwerkkarte) oder rein softwaretechnisch (Loopback)
- liest und schreibt Datenpakete ohne über Transfer Bescheid zu wissen
- viele Netzwerkverbindungen stream-orientiert, Treiber kennt nur Pakete, keine Verbindungen

2.6.1 TCP/IP

Transmission Control Protocol:

- verbindungsorientiertes, zuverlässiges Protokoll
- baut auf (evtl. unzuverlässigem) Datagram-Service (**IP**) auf
- **Multiplexen:** gleichzeitiger Netzwerkzugriff durch mehrere Prozesse → **Port** entspricht **Socket**

User Datagram Protocol:

- ungesichert, verbindungslos, paketorientiert
- **Ports** mit **Multiplexing** und **Demultiplexing**
- Prüfsumme

Internet Protocol:

- Senden von Paketen an Hosts, ungesichert
- baut auf untergelagerten Netzwerkprotokollen auf (Ethernet, Token Ring) → unabhängig von physikalischer Struktur des Netzwerks
- Hauptaufgaben Routing, Fragmentierung (**Pakete**)
- IPv6: Adressraum 32 → 128 Bit

2.6.2 Netzwerkdateisysteme

Network File System (NFS):

- baut auf UDP/IP auf
- genormte Datenrepräsentation
- **Stateless Server:**
  - speichert möglichst keine Informationen über Client-Zustand
  - kein Zugriff auf **Special Files**
  - Datenintegrität nicht sichergestellt
- arbeitet auf Filesystem-Ebene → kann nur ganzes Filesystem exportieren, keine **Multi-Hop-Verbindungen** (über mehrere Rechner)

Remote File Sharing (RFS):

- baut auf TCP/IP auf
- Server halten Zustandsinformationen, Zugriff auch auf **Special Files**
- arbeitet auf Directory-Ebene → Server kann beliebige Directory-Bäume zur Verfügung stellen

2.6.3 Packet Filter

- Netzwerkprotokolle im Kernel implementiert → muss Mechanismen zum Zugriff auf Netzwerkhardware für Applikationsprogramme anbieten
- Welche Pakete an welches Programm weiterleiten?