PARALLELISIERUNG

Kindprozesse erzeugen

```
pid_t tid = fork();
for-loop: i<MAX_PROC
if(pid < 0) {
  //perror + continue
} else if(pid == 0) {
  //child
  :
  exit(0);
} else {
  //parent
  :
}
```

Kindprozess sendet SIGCHLD nach terminieren

Threads starten

```
pthread_t tid;                          arg
while(1){ P();
errno = pthread_create(&tid, NULL, run, NULL);
if(errno != 0) die("...");
}

pthread_join(tid, NULL);   <- auf Thread warten + Ress. freigeben
  oder errno =
pthread_detach(&tid);      pthread_self im Thread selber
if(errno != 0) perror("pthread_detach");
```

Zombie

```
int tmp = errno;
pid_t pid;                   NULL
while((pid = waitpid(0, &event, WNOHANG)) > 0) {
  //pid verwendbar
}
errno = tmp;
```

// beliebigen Kindprozess beh.: (errno!)

```
while(waitpid(-1, NULL, WNOHANG) < 0)
errno = tmp;
```

warten auf Signal:

```
//do someth.:
while(!condition) {
  sigsuspend(&old);
}
```

```
int *feld;
feld = (int*) malloc(42 * sizeof(int));
if(feld == NULL) perror();
calloc(42, sizeof(int));
realloc(feld, 73 * sizeof(int));
NULL-Abfrage
free(feld);
```

SIGNALE

z.B. SIGCHLD

```
// signal blockieren
sigset_t old, new;
if(sigemptyset(&new) == -1) die("...");
if(sigemptyset(&old) == -1) die("...");
if(sigaddset(&new, <Signal>) == -1) die("...");
if(sigprocmask(SIG_BLOCK, &new, NULL) == -1) die;
//.. do someth., danach Maske ändern
sigprocmask(SIG_SETMASK, &old, NULL);
```

beinhaltet SIG, die in Wartezust. vers. werden, während Sigbeh. läuft. Eine Maske pro Proz. aktiv, Kindproz. erbt Sighdl. v. Eltern

```
// Signalmaske (Bsp. ign. Zombie)
struct sigaction act = {
  .sa_handler = SIG_DFL, / SIG_IGN
  .sa_flags = SA_NOCLDWAIT,
};
sigemptyset(&act.sa_mask);
sigaction(SIGCHLD, &act, NULL);
        SIGPIPE: keine sa_flags + IGN

// kill sendet Signale
kill(pid_t pid, int sig); //ret: -1 & errno
```

// SIGNALE:

- SIGINT: strg + C
- SIGKILL: tötet Prozess, nicht behandelbar
- SIGPIPE: schreibe trotz geschl. Gegenseite
- SIGTERM: standardmäßig v. kill(1)
- SIGCHLD: Statusänd. Kindprozess (Zombie)

SERVER

```
int l    sock = socket(AF_INET6, SOCK_STREAM, 0);
if(sock == -1) die("socket");
struct sockaddr_in6 addr = {
  .sin6_family = AF_INET6,                   Byteorder
  .sin6_port = htons(<PORT>),
  .sin6_addr = in6addr_any,
};
int b = bind(sock, (struct sockaddr *) &addr,
                          sizeof(addr));
if(b == -1) die("bind");
int l = listen(sock, SOMAXCONN);
if(l == -1) die("listen");

while(1) {
  int fd = accept(sock, NULL, NULL);
  if(fd == -1) {
    perror("accept");
    continue;
  }
  //... FILE etc.....
  close(fd);
}
```

CLIENT

TCP: verbindungsorientiert, Schutz, Reihenf. & Duplikate
UDP: Paketorientiert, Bitfehlerschutz, kein Folgeverlustschutz

```
char *host, *port;
struct addrinfo hints = {
  .ai_socktype = SOCK_STREAM,
  .ai_family = AF_UNSPEC,
  .ai_flags = AI_ADDRCONFIG,
};
struct addrinfo *head;
int err = getaddrinfo("xhod.com", "80", &hints, &head);
if(err != 0) {
  if(err == EAI_SYSTEM) {
    perror("getaddrinfo");
  } else {
    fprintf(stderr, getaddrinfo: %s,
            gai_strerror(err));
int sock;
for(curr = head; curr != NULL; curr = curr->ai_next)
{ sock = socket(curr->ai_family,
         curr->ai_socktype,
         curr->ai_protocol);
  if(sock == -1) continue;
  if(connect(sock, curr->ai_addr,
             curr->ai_addrlen)
      == 0) break;
  close(sock);
}  // FILE *stream = fdopen(sock, "a+");
             mit Fehlerbeh. usw.
freeaddrinfo(head);
// Kommunikation mit Host
fprintf(stream, "%s", <bla>);
if(fclose(stream)) perror("fclose");
```

FILE

```
FILE *rx = fdopen(fd, "r");
if(!rx) {
  close(fd); (oder die,
  continue;   je nachdem...)
}
int fd2 = dup(fd);
if(fd2 < 0) {
  close(fd);
  continue;
}
FILE *tx = fdopen(fd2, "w");
if(!tx) {
  if(fclose(rx)) perror("fclose");
  close(fd2);
  close(fd);
  continue;
}
//... erfolgreich, do work
if(fclose(rx)) perror("fclose");
if(fclose(tx)) perror("fclose");
close(fd);  WICHTIG: close!
close(fd2);
```

MAKEFILE

```
CC = gcc
.PHONY: all clean
all: rush
clean: rm -rf rush *.o
rush: rush.o plist.o shellutils.o
	$(CC) $(CFLAGS) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c $<
```

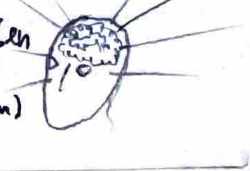
CHECKLIST

- return values ☐
- Fehler Behandl.: ☐ fgets -> ferror
- fflush ☐
- signalbehandlung ☐
- errno gesichert? ☐

AuD ÜBERLEBT + BESTANDEN

PFP ÜBERLEBT + BESTANDEN

SP ... geschrieben ?
+ ... ?
(i can exit vim)



SEM

```
struct SEM { - Einseitige & Mehrseitige Synchr.
  volatile int value;   - krit. Abschnitt mit wechsels. Ausschluss sichern
  pthread_mutex_t m;    - Logisch & phys. unteilbare (atomare) Operationen
  pthread_cond_t c;
};  ganzzahlige Zähl-variable  - Unterst. auf Befehlsatz Ebene
SEM *semCreate(int initVal) {
  SEM *sem = malloc(sizeof(*sem));
  if(sem != NULL) {
    if(pthread_mutex_init(&sem->m, NULL) == 0) {
      if(pthread_cond_init(&sem->c, NULL) == 0) {
        sem->value = initVal;
        return sem;
      }
      pthread_mutex_destroy(&sem->m);
    }
    free(sem);
  }
  return NULL;
}
void semDestroy(SEM *sem) {
  if(SEM == NULL) return;
  pthread_mutex_destroy(&sem->m);
  pthread_cond_destroy(&sem->c);
  free(sem);
}
void P(SEM *sem) { get/prolaag
  pthread_mutex_lock(&sem->m);
  while(sem->value <= 0) {
    pthread_cond_wait(&sem->c, &sem->m); }
  sem->value--;  dekrement
  pthread_mutex_unlock(&sem->m);
}
void V(SEM *sem) { release/signal
  pthread_mutex_lock(&sem->m);
  sem->value++;  inkrement
  pthread_cond_broadcast(&sem->c);
  pthread_mutex_unlock(&sem->m);
}
```

Ringpuffer

```
static volatile int read, write;
static SEM *full, empty, readL;
static int BB[<size>];
void bbCreate() {
  read = 0;
  write = 0;
  full = semCreate(0);
  empty = semCreate(<size>);
  readL = semCreate(1);
}
void bbDestroy() {
  semDestroy(readL);
  semDestroy(empty);
  semDestroy(full);
}
void bbPut(int value) {
  P(empty);
  BB[write] = value;
  write = (write+1) % <size>;
  V(full);
}
int bbGet() {
  P(readL);
  P(full);
  int result = BB[read];
  read = (read+1) % <size>;
  V(empty);
  V(readL);
  return result;
}
```

ABA-Problem

1. T1 liest Variable mit Wert A
2. T1 bekommt CPU entzog., T2 läuft
3. T2 ändert var von A -> B -> A
4. T1 bekommt CPU, prüft A
   -> T1 hat Änderung nicht gesehen

VERZEICHNIS DURCHSUCHEN

```
DIR *dir = opendir(path);
if(!dir) die("opendir");
struct dirent *de;
while(1) {
  errno = 0;
  de = readdir(dir);
  if(de == NULL) break;               ausgeschl.
  if((de->d_name)[0] == '.') continue;  ign.
  char name = [strlen(path)+strlen(...
        ...(entry->d_name) + 2];
  if(sprintf(name, "%s/%s", path,
        entry->d_name) < 0) {
    perror("sprintf");
    continue;
  }
  struct stat buf;
  if((stat(name, &buf) != 0) perror + cont.;
  if(S_ISDIR(buf.st_mode)) crawl(name);
  if(S_ISLNK(buf.st_mode)) evtl. ausgeben/ignore
  if(S_ISREG(buf.st_mode)) {  USER PRÜFUNG
    if(getuid() != buf.st_uid) continue; !
    //Datei: öffnen, Thread übergeb., ...
    continue;
  }
}
if(errno != 0) perror("readdir");
if(closedir(dir)) perror("closedir");
```

DATEI SCHREIBEN/SENDEN

```
// FILE *client
// Datei öffnen: file
int *c;
while((c = getc(file)) != EOF) {
  if(putc(c, client) == EOF) {
    break;
  }
} //fclose(file);
```

AUSGABE SCHREIBEN

```
p = printf(*format, *param);
p = fprintf(FILE*, *format, *param);
p = fputc(int c, FILE *stream);
p = fputs((const char *s, FILE *stream);
if(p < 0) perror("printf");
if(fflush(stdout)) die("...");
```

EINGABE LESEN

```
long max = sysconf(_SC_LINE_MAX) + 2;
        8192
char *tmp = (char*) calloc(max, sizeof(char));
if(tmp == NULL) die("...");
fgets(tmp, max, stdin);
if(tmp == NULL) die("...");
free(tmp); if(ferror(stdin)) die("fgets");
```

```
strcpy(const char *dest, const char *src);
char*       strncpy mit size_t n
strchr(const char *s, int c);
-> Pointer char * auf ersten Eintrag == c
strrchr(filepath, '/') + 1
-> char * auf letzten          ohne "/"
   Eintrag == c                vor filename
```

FEOF: 0: Daten noch; 1: EOF

```
while(feof(filestream) == 0) {...}
int strcmp(const char *s1, const char *s2);
   0: s1 == s2, -1: s1 vor s2, 1: s1 nach s2
strncmp -> vergleiche n Zeilen
size_t strlen(const char *s);
```

AUSNAHMEBEHANDLUNG

Trap
- interne Ursache
- synchron, vorhersagbar, reproduzierb.
- Behandlung: Beendigung / Wiederaufnahme!
- Bsp.: Syscall, Adressraumverletz., Seitenfehler, x/0

Interrupt (IRQ)
- externe Ursache
- asynchron, unvorhersagbar, nicht reproduzierbar
- Behandlung: nur Wiederaufnahme
- Bsp.: Eingabe/Ausgabe Oper., DMA operation

BETRIEBSARTEN

STAPELbetrieb
Ziel: Durchsatz, Durchlaufzeit, Prozessorauslastung
→ Abgesetzter Betrieb: Satellitenrechner (Eingabe) sendet Eingabe an Hauptrechner, dieser verarbeitet (schickt Erg. an Satellitr. → Entlastung
→ Überlappte E/A: Durch Interrupts gleichz. Proz., Speicherdirektzugriff (DMA) statt progr. I/O → überl. Progr. ausf. Problem: Leerlauf bei Auftragswechsel
→ überlappte Auftragsverarb.: Verarbeitungsstrom aufeinanderf. Programme, Technik: Prefetching oder Auftragseinplanung. Probleme: CPU Konsp., IO Leerlauf
→ Abgesetzte E/A: Spooling CPU-Stoß & IO-Stoß gleichzeitig, Entkopplung durch Pufferspeicher
→ Mehrprogrammbetrieb: CPU Multiplexen, mehrere Programme im HS, Akt./Pass. Warten. wichtig: Adr. Raum Schutz! Laufzeit: Eingrenz. Zur Laufzeit: Schutz durch Segmentierung
→ Dynamisches Laden: Program größer als Arbeitssp. zerteilen, Nachladen. Nachteil: Programmierer impl.!

Echtzeitbetrieb (→ Gerechtigkeit / Lastausgleich)
Ziel: Termineinhaltung, Vorhersagbarkeit, Dringlichkeit
Terminvorgabe:
→ weich: nutzbar, wird mit Zeit wertloser, Terminverletz. tolerierbar
→ fest: Erg. wertlos, verwerfen, Term. verl. toler.
→ hart: kein Erg. bed. großes Problem, nicht tol.

Mehrzugangsbetrieb
→ nur sinnvoll mit CPU und Speicherschutz!
Dialogbetrieb → Antwortzeit
→ mehrere Nutzer, Eingabe & Verarbeitung abwechselnd
Problem: Monopolisierung mögl.
Dialogorient. Monitorsys. (Hintergrundbetrieb)
→ Proz. im Vordergrund start, und im Hintergr. vollziehen
Aufgaben parallel bearbeiten
Problem: Größe HS
Teilnehmerbetrieb
→ eigene Dialogproz. werden interaktiv gestartet, Zeitscheibe verbergen CPU-Monopolis.
Teilhaberbetrieb:
→ Client-Server-System

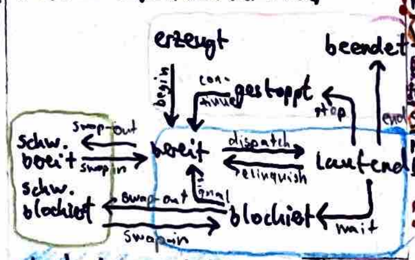
MONITOR abstrakte Datentypen

Konzept: mehrere WS, Prozesse warten außerhalb d. Mon.
MonitorWS: P warten auf Mon. Eintritt
EreignisWS: P warten auf Wartebedingungsende
verwendet bei blockierung Bedingungsvariablen
nicht blockierend: Vorrang Signalgeber
blockierend: Vorrang Signalnehmer
Hansen (blockierend): wait() / broadcast()
→ Signalisierung lässt Signalgeber d. Mon. verlassen

Hoare (blockierend): signal()

Mesa (nicht blockierend): signal

PROZESSVERWALTUNG



Short-term-scheduling:
→ wichtig Mehrprozessbetrieb
→ bereit: Proz. in Bereitliste
→ laufend: Proz. vollz. CPU-Stoß
→ block.: Proz. erw. BM-Zuteil.

medium-term-scheduling:
→ Umlagerung komplette Progr.
→ schwebend bereit: ausgelagert, Einlastung außer Kraft
→ schw. blockiert: ausgelagert, Proz. ereigniswartend

Long-term-scheduling:
→ erzeugt: fertig & Progr. Verarb.
→ gestoppt: erwartet Forts. / Beend.
→ beendet: Prozess erwartet Entsorgung

benutzerorientierte Kriterien:
(→ benutzerdienlichkeit)
Antwortzeit, Durchlaufz., Termineinhaltung, Vorhersagbark.

systemorientierte Kriterien:
Durchsatz, Prozessorauslastung, Gerechtigkeit, Dringlichkeit, Lastausgleich

User-Threads (federgewichtig)
→ nichtprivilegierter Code
→ blockierende Syscalls blockieren andere User-Threads ⊖
→ Schedulingstrategie v. Programmierer → vgl. kernel
→ effiziente Umschaltung ⊕
→ nicht für Multiprozessorbetr. ausgel.
→ KEIN eig. Adressraum, Ausführungsfaden!

Kernel-Threads (Leichtgewicht)
→ Threads mit privileg. Code
→ Erzeugung teuer als User-Thrd. ⊖
→ keine block. bei Syscalls ⊕
→ Scheduling durch BS
→ Umschalten im Kern → langsam ⊖
→ evtl. Sinn für Multiprozessorbetr.
→ teilt Adr. Raum mit anderen Proz.

Prozess (Schwergewichtig)
→ Prozess, der als Systemkernfaden allein in speicherschutzisolierten Adressraum stattfindet

Verklemmungen:
notwendige
1. wechselseitiger Ausschl.
2. Ein oder mehr Betr.Mittel nachford.
3. Betriebsmittel unentziehbar
notwendige & hinreichende
4. zirkuläres warten

vorbeugung: notw. Bed. werden aufgehoben (vor Laufzeit)
verhinderung/meidung: gieriger Algorithmus (Bankier) zur Laufzeit

FCFS Verarb. nach Ankunftszeit
RR regelmäßige Umplanung
VRR variable Zeitscheibe, Vorzugsliste
SPN Einplanung nach Bedienzeit
HRRN SPN + Altersberücksichtig.
SRTF SPN + spontane Umplanung
MLQ Mischbetrieb
MLFQ FCFS + Verdr. + Alter
kooperativ verdrängend
!Konvoi! verhungern

kooperativ (FCFS)	vs.	präemptiv (RR, VRR, SRTF)
→ abhängige Proz.		→ unabhängige Proz.
→ kein CPU-Entzug, keine Verdrängung, Monopolisierung!		→ CPU-Entzug, Ereignisbedingte Verdrängung

deterministisch	vs.	probabilistisch (SPN, SRTF, HRRN)
→ Alle Proz. zeiten sind bekannt		→ Proz. zeiten approx.
→ Einhaltung von Zeitgarantien		→ keine Zeitgarantie
→ CPU-Auslastung vorhersagbar		→ CPU-Auslastung nicht vorhersagbar

statisch	vs.	dynamisch
→ vor Betrieb, keine Planung in Betrieb		→ während Betrieb
→ kompletter Ablaufplan		→ Stapelsysteme, interaktive Systeme, verteilte Systeme
→ strikte Echtzeitsysteme		→ schwache & feste Echtzeitsysteme

assymetrische	vs.	symmetrische
→ assym. Multiproz. Sys.		→ identische Prozessoren
→ optional auf Symmetr. Multipr. Sys.		→ Lastausgleich von Prozessen auf CPUs
→ ungleiche Verteil. mehrerer Prozesse auf CPUs		

PROZESSSYNCHRONISATION

Gleichzeitige Prozesse:
→ vertikal: time sharing (multiplexing)
→ horizontal: multiprocessing
gekoppelte Prozesse: Zugr. auf gl. Daten
Sequentialisierung: Koord. durch atom. Oper.
unsicherer Zustand: Verkl. unvermeidb.

einseitige Synchronisation
→ erfolgt logisch oder bedingt
Erzeuger-Verbraucher-Beziehung
Warten auf Betriebsmittel, das noch nicht bereit ist

mehrseitige Synchronisation
→ blockierend / nichtblockierend
→ alle Prozesse betroffen, gegenseitiger Ausschluss

blockierende Synchronisation
z.B. Semaphore: block. krit. Abschn.
Nachteile: Leistung, Robustheit, Interferenz, Lebendigkeit ⊖

nicht-block. Synchronisation
⊕ keine krit. Abschn., Verwendung von atomaren Operationen, CAS
Nebenläufigkeitsfreie Vorgehen ohne Locks
⊖ Komplex, verhungern mögl.

BETRIEBSMITTEL

wiederverwendbar / konsumierbar
(unteilbar (zeitw. exkl.)
teilbar (präemptiv)
begrenzt, z.B. CPU, RAM, GPU
unbegrenzt: produz. & zerst. z.B. Signal, Trap

Ziele: Optimale Durchsetzung der Betriebsstrategien, kein Verhungern / Verklemmung! Buchführung über Betr. Mittel, Betr. Mittel entzug bei Verletzungen

PLATZIERUNGSSTRAT.:

→ Bitkarte: Hohlr. fester Größe
→ Lochliste: Hohlräume var. Größe
Zuteilungsverfahren:
worst-fit langsamere Verschm. als first fit
→ absteigend nach Lochgröße, größtes Loch gesucht
⊕ Suchaufw. klein, ⊖ Verschnitt
u.U. muss ganze Liste durchl. werd.
best-fit:
→ Lochgröße aufsteigend kleinstes pass. Loch gesucht
⊕ kleiner Verschnitt, ⊖ Suchaufwand
first-fit: hoch. Verschm. Aufwand
→ Erstes pass. Loch nutzen
⊕ kleiner Suchaufw., ⊖ Verschnitt
next-fit:
→ RR von first-fit, beginnt bei letztem Loch

FRAGMENTIERUNG

→ intern: Speicherblock nur teilweise gefüllt; buddy: entsteht durch aufrunden
→ extern: Zerstückelung des Speicherraumes, angeforderte Größe > einzelne Löcher
Verhinderung: Verschmelzung

LADESTRATEGIEN:

Demand-Paging: auslagern nichtbenutzter Seiten

PAGEFAULT

1. Adresszugriff
2. MMU Trap: nicht present
3. BS-Trap-Handler
4. a) Seite ungültig? → SIGSEGV
b) gültig, nicht anwesende Seite
→ im Freiseitenpuffer (kurz vorher ausgelagert) → einlagern & anwesend mach.
→ Seite nicht mehr da → einlagern.
→ Freier Seitenrahmen? → einlagern anstoßen, blockieren
→ kein freier Seitenrahmen? → Seite suchen, auslagern anstoßen, block. d. Prozess, Seite einlagern anst., block.
als anwesend markieren
5. Zugriff wiederholen

TLB: Logisch → phys zw.speich.

ERSETZUNGSSTRAT.

lokal: Suchraum Menge residente Seiten d. Proz. Seitenfehler vorhers./repr.
global: Suchraum Menge residenter Seiten aller Prozesse Seitenfehler unvorhers./unrepr.
Lokalitätsprinzip: Proz. referenziert Stelle in sein Adr. Raum → wahrsch. Stelle in direkter Umgeb.
Seitenflattern: Ein-/Aus-Lagern dominiert Systemaktivität verschwindet meist von alleine

Freiseitenpuffer
→ freien Seitenrahmen schnell finden. FIFO Cache für potenzielle Seiten

FIFO: First in First Out
Ersetzt wird zuletzt eingel. Seite. Impl. als verkett. Liste
Mehr Seitenrahmen führen zu mehr Seitenfehlern

LFU (Least Frequ. used):
→ ersetze seltensten referenzierte Seite, impl. eventuell mit Zähler

LRU: (Least recently used):
→ ersetze längsten nicht referenzierte Seite
periodische Unterbrechung (Hintergrundrauschen)

SECOND CHANCE
(ähnlich FIFO) nutzt Zeitintervall zur period. Unterbrechung, Ersetzt erste Seite mit Referenz-Bit = 0; wenn alle Ref Bit = 1: FIFO!

DRITTE CHANCE
SECOND CHANCE + Modifikationsbit
(0,0) unbenutzt → beste
(1,0) beschrieben → naja
(0,1) gelesen → naja
(1,1) gelesen kürzlich → schlecht
→ zwei Umläufe / Seite

ADRESSRÄUME

Programmadresse → Logische Adressraum → virt. Adr. → Phys. Adds.
Reale Adressen: lückenhafter, realer HS, reale Adresse, von HW
Logische Adressen: lückenlose, wirkliche HS
→ getrennte Adressbereiche
Sicherheit, Virtualisierung, bessere Verwaltung
virtuelle Adressen: lückenlose, scheinbarer HS
Paging, entkoppelt von RAM Lokalitätsprinzip

eindimensional / Zweidimensional → segmente

Seitendeskriptor
MMU: reale Adresse, presentbit,

Filedeskriptor
prozesslokale Int von einem Prozess

Adressraumschutz
FENCING Begrenzungsregister legen Adressbereich im physikalischen Speicher fest

Segmentdeskriptor
MMU: Basis (Segmentanfang)
Limit (Segmentlänge)
Typ (Text, Daten, Stapel)
Rechte (r/w/run)
Expansionsrichtung / presentbit

```
static int a = 2022
void f1(const int *y) {
  static int b;
  int c;
  char *d = malloc(0x802);
  void (*e)(const int *) = f1;
  y++;
  //...
}
```

y, c, d, e auf	stack
	↓
<0x802>	heap
b = 0	BSS
a = 2022	Daten
f1...	Text
0x0	

DATEISYSTEME

kontinuierliche Speicherung: Dateispeicherung in kont. nummerierten Blöcken. Identif. über Anfangsblock + Größe
⊕ schnelles Lesen, keine externe Fragm., geeignet für Echtzeitsysteme
⊖ freien Speicher finden schwer, interne Fragm., dyn. Erweiterung sehr aufwendig!

verkettete Speicherung: Speicherung von Daten in verketteten Blöcken (Liste) ⊕ dyn. Erweiterung einfach
⊖ hohe Fehleranfälligkeit / externe Fragmentierung

indiziertes Speichern: Spezieller Platten enthält Blocknummern (wie Adressen), Datenblock einer Datei über Blocknummer identifizierbar, Bei großen Dateien mehrere Datenblöcke. Extents → größerer Datenblock, kontinuierlich teils mögl. ⊖ interne Fragmentierung

Freispeicherverwaltung: Speicher in gl. große Blöcke, Bitvektor / Block: belegt / nicht belegt. verkettete Liste speichert freie Blöcke, besser: folgende Blöcke zusammenf. besser: statt Liste: Datenblöcke zeigen auf freien Speicher

UNIX Blockstruktur: | Bootbl. | Superbl. | Inodeliste | Datenblock |

Boot: Infos z. laden einer initialen Progr.
Super: Anzahl Blöcke / Inodes und Attribute
Inode: Dateiart, Userbl, User Recht, Erstellung, Zugriff, Änderungszeit, Dateigröße, Linkzähler HL, Blockverw. auflisten

Journaling File System: Transaktionsänderungen in Log-File dokumentiert, vor transaktion Log schreiben. Inkonsistenz beim boot vermeiden, undo nach erkanntem Problem. Optimiert: Checkpoint (→ wiederherstellen)

BINDEN statisch: Adressauflösung: Bindezeitpunkt, zum Ladezeitpunkt an bel. Adr. platzieren
dynamisch: Adressauflösung zur Laufzeit / beim Laden
Übersetzung: Präproz. → compile → assemble → Binden

RAID 0		RAID 1		RAID 4				RAID 5		
A1	A2	A1	A1	A1	A2	A3	Ap	A1	A2	Ap
A3	A4	A2	A2	B1	B2	B3	Bp	B1	Bp	B2
A5	A6	A3	A3	C1	C3	C3	Cp	Cp	C1	C2