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

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

Zombie

```
int tmp = errno;
pid_t pid;
while((pid = waitpid(0, &event, WNOHANG|WUNTRACED) > 0) {   // NULL
  //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-übergabe!
free(feld);
```

SIGNALE

z.B. SIGCHLD

```
// signal, abfangen
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);
```

Speichert SIG, die in Warteryst. vers. wurden, während Sigbeh. läuft. Eine Maske pro Prozess aktiv, Kindprozess erbt Sigbeh. 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: kein 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 sock_addr*) &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, sicher
UDP: Paketorientiert, kein Pkt-Verlust, Bitfehlerschutz, Pakete können verloren gehen

```
char *host, *port;
struct addrinfo hints = {
  .ai_socktype = SOCK_STREAM,
  .ai_family   = AF_UNSPEC,
  .ai_flags    = AI_ADDRCONFIG,
};
struct addrinfo *head;
int err = getaddrinfo("xhad.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", <blabla>);
if(fclose(stream)) perror("fclose");
```

FILE

```
FILE *rx = fdopen(fd, "r");
if(!rx) {
  close(fd);   // oder die, je nachdem...
  continue;
}
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 list.o shellutils.o
	$(CC) $(CFLAGS) -o $@ $^

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

CHECKLIST

- *return values ☐
- *Fehlerbehandl.: ☐ fgets → ferror
- *fflush ☐
- *Signalbehandlung ☐
- *errno gerichtet? ☐

Struct SEM

```
struct SEM {
  volatile int value;
  pthread_mutex_t m;
  pthread_cond_t c;
};
```

- Einsatz: 2 mehrere Kop. Synch. Abschnitt mit wechselS. Ausschluss sichern
- logisch & physs. Wert, niemals negativ
- garantiert atomar (atomare Operationen)
- Unterbrechungen

```
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/prolong
  pthread_mutex_lock(&sem->m);
  while(sem->value <= 0) {
    pthread_cond_wait(&sem->c, &sem->m);
  }
  sem->value--;   // decrement
  pthread_mutex_unlock(&sem->m);
}
void V(SEM *sem) {   // release/signal
  pthread_mutex_lock(&sem->m);
  sem->value++;
  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 entzogen, 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;   // ausgeb. ign.
  if((de->d_name)[0] == '.') continue;
  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(lstat(name, &buf) != 0) perror+cont.;
  if(S_ISDIR(buf.st_mode)) crawlen(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 übergeben, ...
    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;
  }
}
if(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;
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);   // strncpy zusätzl. size_t n
strchr(const char *s, int c);
→ Pointer char* auf ersten Eintrag == c
strrchr(filepath, "/") + 1
→ char* auf letzten Eintrag == c   ← ohne "/" 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 → vergleicht n Zeichen
size_t strlen(const char *s);
```

AUSNAHMEBEHANDLUNG

Trap
- interne Ursache
- synchron, vorhersagbar, reproduzierbar
- Behandlung: Beendigung / Wiederaufnahme!
- Bsp.: Syscall, Adr.raumverlet., Seitenfehler, ÷/0

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

BETRIEBSARTEN

STAPELbetrieb
Ziel: Durchsatz, Durchlaufzeit, Prozessorauslastung

→ Klassischer Betrieb: Satellitenrechner (Eingabe) sendet Eingaben an Hauptrechner, dieser verarbeitet C schickt Erg. an Satellitr. → Entlastung

→ Überlappte E/A: Durch interruptgesteuerte Proz., spezielle Hardware (DMA) stellt Programmier, Technik: Pufferung oder Auftragseinplanung Problem: CPU Overhead, Deadlock

→ Überlappte Auftragsverarb.: verarbeitungsstrom aus Auftrag, Programme, Technik: Pufferung oder Auftragseinplanung Problem: CPU Overhead, Deadlock

→ Abgekoppelte E/A: Spooling CPU-Stoß + IO-Stoß gleichmäßig, Entkoppelung durch Pufferbereiche

→ mehrprogrammbetrieb: CPU Multiplexen, mehrere Programme im HS, Akt./Pass. warten - wichtig: Adr. Raum Schutz! Laufzeit: Eingepf. Zur Laufzeit: Schedule Sequenzierung

→ Dynamisches Laden: Programm größer als Arbeitsspeicher, 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. tolerabel
→ hart: kein Erg. bed. großes Problem, nicht tol.

Mehrzugangsbetrieb
→ nur sinnvoll mit CPU und Speicherschutz!

Dialogbetrieb → Antwortzeit
→ mehrere Nutzer, Eingabe/verarbeitung abwechselnd
Problem: Monopolisierung
Dialogorient. Hintergr. (Hintergrundbetrieb)
→ Proz. im Vordergrund steht, und im Hintergr. vollziehen Aufgaben parallel bearbeitet
Problem: Größe HS

Teilnehmerbetrieb
→ eigene Dialogproz. werden interaktiv gesteuert, Zeitscheibe verbraucht CPU-Leistung gerecht verteilt

Teilhaberbetrieb:
→ Client-Server-System

MONITOR abstrakte Datentypen

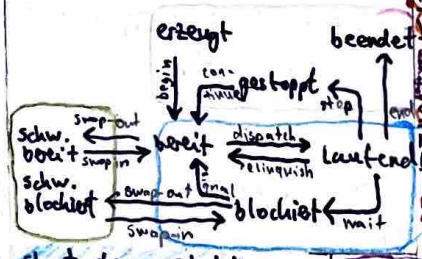
Konzept: mehrere WS, Prozesse werden außerhalb des Mon. Monitor WS: P warten auf Mon. Eintritt
Ereignis VS: P warten auf Wahrheitsbedingung, wenn verändert bei Änderung Bedingungsvariablen nicht blockierend: Vorrang Signalgeber
blockierend: Vorrang Signalnehmer

Hansen (blockierend): wait() / broadcast()
→ Signalisierung löst Signalgeber d. Mon. verlassen

Hoare (blockierend): signal()

Mesa (nicht blockierend): signal

PROZESSVERWALTUNG



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

medium-term-scheduling:
→ Umlagerung komplette Progr.
→ Schwerbed 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:
(→ benutzerverlichkeit)
Antwortzeit, Durchlaufz., Termineinhaltung, Vorhersagbark.

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

User-Threads (fadegewichtig)
→ nichtprivilegierte Code
→ blockierende Syscalls blockieren andere User-Threads ⊖
→ Schedulingstrategie v. Anwendung ⊕ entwickelt
→ effiziente Umschaltung ⊕
→ nicht für Multiprozessorbetr. ausgel.
→ VAS eig. Adressraum, Ausführungsfad.

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

Prozess (schwergewichtig)
→ Prozess, der als Systemhintergrundfaden allein in speicherschutzisolierten Adressraum stattfindet

Verklemmungen:
notwendige
1. wechselseitige Ausschl.
2. Einzelanforderung Betr. Mittel nachfrad
3. Betriebsmittel unentziehbar
notwendige & hinreichende
4. zirkuläres warten

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

FCFS
Verarb. nach Ankunftszeit
RR
regelmäßige Umplanung
VRR
variable Zeitscheibe, Vorzugsliste
SPN
Einplanung nach erwarteter Bedienzeit
HRRN
SPN + Altersberücksichtigung
SRTF
SPN + spontane Umplanung
MLQ
Mischbetrieb
FEEDBACK
FCFS + Verd., + 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
→ alle Proz.zeiten sind bekannt → Einhaltung von Zeitgarantien → CPU-Auslastung vorhersagbar
SPN, SRTF, HRRN → proz.zeiten approx. → keine Zeitgarantie → CPU-Auslastung nicht vorhersagbar

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

asymetrisch vs. symmetrisch
→ asymm. Multiproz. Sys. → optimal auf symmetr. Multiproz.sy. → gleiche verteilungsbedürftige Prozesse auf CPUs
→ identische Prozessoren → Lastausgleich von Prozessen auf CPUs

PROZESSSYNCHRONISATION

Gleichzeitige Prozesse:
→ vertikal: timesharing (multiplexing)
→ horizontal: multiprocessing
gekoppelte Prozesse: Zus. auf gl. Daten
Sequentialisierung: Koord. durch atom. oper.
unsicherer Zustand: Verkl. vermeidb.

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

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

blockierende Synchronisation
z.B. Semapher: block. bei Akt. Anforderung
Nachteile: Leistg, Robustheit ⊖, Interferenz, Lebendigkeit

nicht-block. Synchronisation
⊕ keine hit. Akt., verwendung von atomaren operationen, CAS
Lebendigkeit & freie Wahrscheinl. ohne Locks
⊖ Komplex, verhungern möglich

BETRIEBSMITTEL

wiederverwendbar (konsumierbar / unteilbar (nutzbar entl.)
teilbar (präemptiv)

beigenutzt, z.B. CPU, RAM, BPV
unbegenutzt: produz. & zerst. z.B. Signal, Imp

Ziele: Optimale Durchsetzung der Betriebsstrategien, kein Verhungern (Verklemmung), Buchführung über Betr.mittel, Betr.mittel entzug bei Verletzung

PLATZIERUNGSSTRAT.:

→ Bitkarte: Hdtr. feste Größe
→ Lochliste: Hdtrräume var. Größe

Zuteilungsverfahren:

worst-fit (ungenauer Verschn. als first fit)
→ absteigend nach Lochgröße, größtes Loch gesucht
⊕ Suchaufw klein, ⊖ Verschnitt u.U. mit ganzer Liste durchl. nötig

best-fit:
→ Lochgröße aufsteigend, erstes passt Loch gesucht
⊕ kleine Verschn., ⊖ Suchaufwand

first-fit: hoher Verschn. Aufwand
→ Erstes passt. Loch nutzen
⊕ kleine Suchaufw., ⊖ Verschnitt

next-fit:
→ RR von first-fit, beginnt bei letzt

FRAGMENTIERUNG

→ intern: speicherblock nur teilweise befüllt; buddy: aufgeteilt durch aufrunden
→ extern: zerstückelung des Speicherraumes, angeforderte Größe > einzelne Löcher
Vermeidung: Verschiebung

LADESTRATEGIEN:

Demand-Paging: auslagern nichtbenutzte Seiten

PAGEFAULT
1. Adresszugriff
2. MMU Trap: nicht present
3. BS-Trap-Handler
4. a) Seite ungültig? → SIGSEGV
   b) gültig, nicht-anwesende Seite
   → im freien Seitenpuffer (hard) woher angelogert) → einlagern & anmelden
   → Seite nicht mehr da → einlagern.
   → freier Seitenrahmen? → einlagern auslagern, blockieren
   → kein freier Seitenrahmen? → Seite suchen auslagern auslagern block. d. Prozess Seite einlagern anst. block
   als auszuführen markieren
5. Zugriff wiederholen

TLB: Logisch → plus zw. Speich.

ERSETZUNGSSTRAT.

Lokal: Suchraum menge residente Seiten d. Proz. Seitenfehler verursacht / repr.
global: Suchraum menge residenter Seiten aller Prozesse Seitenfehler verursacht / unrepr.

Lokalitätsprinzip:
Proz. referenziert Stelle in sein Adr.-Raum → wahrsch. Stelle in direkter Umgeb.

Seitenflattern: Ein-/Auslagern dominiert Systemaktivität → verschwendet Zeit

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

FIFO: First in First out
Ersetzt wird zuletzt eingel. Seite. Implem. als verkett. Liste, mehr Seitenrahmen führen zu mehr Seitenfehlern

LFU (least frequ used):
→ ersetzt seltenst referenzierte Seite, implementiert mit Zähler

LRU: (least recently used):
→ ersetze (ungefähr) nicht referenzierte Seite
periodische Unterbrechung (Hintergrundroutine)

SECOND CHANCE
(einfach FIFO) mit Zeitintervall zu prüfd. Unterbrechung. Ersetzt Seite mit Ref.bit = 0, wenn alle Ref.Bit = 1: FIFO!

DRITTE CHANCE
SECOND CHANCE + Modifikationsbit
(0,0) unbenutzt → beste
(1,0) benutzt → naja
(0,1) gelesen → naja
(1,1) benutzt + schlecht → zwei Umläufe bis Seite

ADRESSRÄUME

Programmadresse → Logische Adresse
→ virt. Adr. → Phys. Adr.

Reale Adresse:
Leitungsadr. realer HS, reale Adresse, von HW

Logische Adressen:
lückenlose, virtuelle HS
↳ getrennte Adressräume Sicherheit, Virtualisierung, eigene Verwaltung

virtuelle Adressen:
lückenlose, reelle HS
Paging, Verdoppelung von RAM (Lokalität)

eindimensional / zweidimensional
↳ segmente

Seitendeskriptor
MMU: reale Adresse, Präsentit.

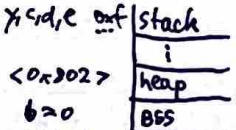
Filedeskriptor
prozesslokale Int von einem Prozess

Adressraumschutz
FENCING: Begrenzungsregister legen Adressbereich im physikalischen Speicher fest

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

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

x, c, d, e auf ...
<0x2022>
b = 0
a = 2022
f1 ...
0x0



DATEISYSTEME

konturielle/lückenlose Speicherung:
Dateispeicherung in kont. nummerierten Blöcken. Identif. über Anfangsblock + Größe
⊖ schneller 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)

indizierter Speicherung:
Spezieller Platten enthält Blocknummern (wie Adressen), Datenblock einer Datei über Blocknummern identifizierbar, bei großen Dateien mehrere Datenblöcke. Extents → größere Datenblöcke, kontinuierlich teils möglich. ⊖ interne Fragmentierung

Freispeicherverwaltung:
Speicher in gl. große Blöcke, Bitvektor / Block: belegt / nicht belegt. verkettete Liste: speichert freie Blöcke, besser: folgende Blöcke zusammenfassen besser: statt Liste: Datenblöcke belegen auf freien Speicher
UNIX Aufbau Partition: Bootbl. | Superbl. | Inodeliste | Datenblock

Boot: Lesen z. Laden einer initialen Progr.
Super: Anzahl Blöcke / Inodes und Attribute
Inode: Dateiart, Userbl, Usergr, Rechte, Inhalte, 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 echten Problem, optimiert: Checkpoint (→ wiederherstellen)

BINDEN
statisch: Adressauflösung: Binderzeitpunkt, Zwischenspeicherung bei Lade-Adr. phys.
dynamisch: Adressauflösung zur Laufzeit / beim Laden
Übersetzung: Äquivalent → compile → assemble → Binden