Server

```
int s = socket(AF_INET6, SOCK_STREAM, 0);
if(s==-1) die ...
struct sockaddr_in6 ad={
 .sin6_family = AF_INET6,
 .sin6_addr = in6addr_any,
 .sin6_port = htons(port)
};
if(bind(s, (struct sockaddr *)&ad, sizeof(ad))==-1) die...
if(listen(s, SOMAXCONN)==-1) die ...;
while(1){
 int a = accept(s, NULL, NULL);
 if(a==-1){ /* errno check */ continue;
 if(close(a)==-1) ...; }
```

Client

```
struct addrinfo *res;
getaddrinfo("as.cc", "123", &(struct addrinfo){
  .ai_socktype= SOCK_STREAM,
  .ai_family = AF_UNSPEC,
  .ai_flags= AI_ADDRCONFIG}, &res);
int r
if(r!=0){ if r== EAI_SYSTEM) die ...;
 else { fprint(stderr, "getaddrinfo: %s\n",
        gai_strerror(ret)); exit(EXIT_FAILURE);}
}
int sock=-1; struct addrinfo *iter= res;
do{ sock= socket(iter->ai_family, iter->ai_socktype, iter->ai_protocol);
   if(sock==-1) die ...;
   if(connect(sock, iter->ai_addr, iter->ai_addrlen)==0) break;
   else { if (close(sock)==-1) die..; sock=-1;}
   iter = iter->ai_next;
 } while (iter != NULL);
free addrinfo(res);
if(sock==-1){ exit(EXIT_FAILURE);
FILE *rx= fdopen(sock, "r");
if(rx==NULL) die ...;   -> fdopen(sc, "w")   tx!
int sc= dup(sock); if(sc==-1) die ...;
```

Kommunikation: `if(fprintf(tx, "msg")<0) die...` `if(fflush(tx)== EOF) die..; if(fclose(tx) == EOF) die...`

Datei senden:

```
int fd= open(path, O_RDONLY | O_CLOEXEC);
if(fd==-1) die..;
FILE *f = fdopen(fd, "r"); if(f==NULL) die...
int c; while((c=getc(f))!= EOF){
 if( putc(c,tx)== EOF){die /break .. }}
if(ferror(f) die...; fclose(f)==-1);
```

Exec / fork `execlp(char *path, char *argv0, .., NULL)`
`execvp(char *path, char *argv[]);`
argv0 muss Programmname sein!
return nur im Fehlerfall, mit -1 und errno gesetzt

Close on Exec: 1. `int fd= open("..", O_CLOEXEC)`

```
2. int f = fcntl(fd, F_GETFD);
if(f==-1) die..; f|= FD_CLOEXEC;
if(fcntl(fd, F_SETFD, f)==-1) die...
```

Signale — Blockieren:

```
sigset_t new, old;
if(sigemptyset(&new) ...
if(sigaddset (&new, (signal)) ==-1) die..
if(sigprocmask(SIG_BLOCK, &new, &old)
 ==-1) die ...; // do sth useful
if(sigprocmask(SIG_SETMASK, &old, NULL)== ...
```

+ bei SIG_IGN nicht nötig, weil schon

Handler

```
struct sigaction act={
  .sa_handler = SIG_DFL / SIG_IGN / function
  .sa_flags =SA_RESTART // bei ...handler
}; if(sigaction(signal, &act, NULL)==-1 die..
```

bei SIG_IGN: sigemptyset &act.sa_mask

Handler-Fkt: `(static) void handler(int signal){ ..}`

kill: `kill(pid_t pid, int signal)==-1) die..`

Signale: SIGKILL: tötet Prozess, nicht behandelbar
SIGINT ^C SIGPIPE: Pipe kaputt SIGTERM: std. beenden
SIGCHLD: Kind ändert Status -> Zombie

SIGPIPE bei Server: vor Thread-Erstellung
`sigemptyset..; sigaddset; if(pthread_sigmask(SIG_BLOCK, &new, NULL)!=0) die..`

pass. warten - sigsuspend

```
SIGCHLD blockieren
while(! condition){
 if(sigsuspend(&old)==-1) &&
  errno != EINTR) die.. }
```

SIGCHLD freigeben

Semaphore

```
struct SEM{
pthread_mutex_t mut;
pthread_cond_t cond;
int counter;
};
SEM *semCreate(int init){
struct SEM *sem= malloc(sizeof(struct SEM));
if(sem==NULL) ...
pthread_mutex_t m;    r) ==0 -> kein Fehler
                         !=0 Fehler
int ret= pthread_mutex_init(&m, NULL);
if(ret!=0){free(sem); ...}
sem->mut=m; sem->counter=init; return sem;}
void P(SEM *sem){
int ret= pthread_mutex_lock(&sem->mut);
if(ret!=0) ..;
while(sem->counter<1) pthread_cond_wait(
 &sem->cond, &sem->mut);
sem->counter--; ret= pthread_mutex_unlock(
 &sem->mut); if(ret!=0) .. }
void V(SEM *sem){ int ret= pthread_mutex_lock(&sem->mutex); if(ret!=0)..;
sem->counter++; ret= pthread_cond_broadcast(
&sem->cond); ret= pthread_mutex_unlock(..);}
void semDestroy(SEM *sem){
 if(sem==NULL) return
 int ret= pthread_mutex_destroy(&sem->mut);
 ret= pthread_cond_destroy(..);
 free(sem);}
```

Threads

```
pthread_t tid;   pthread_exit(status); im Thread
for(..){
 if((errno= pthread_create(&tid, NULL, work, NULL|&new))!=0) die..;
static void *work(void *arg){}
 if ((errno= pthread_join (&tid, NULL))!=0)
if((errno=pthread_detach(&tid))!=0) ..
```

Ovarieren wenn mit im Thread
Signal blockieren, dann erst im Thr. setzen

fork/Zombies

```
pid_t pid = fork();
 if(pid==-1){perror(..); ..}
 else if(pid==0){ //kind .. exit(..)}
 else{ //parent}
```

Zombie (fork)

```
int e=errno; pid_t pid; int wstatus;
while((pid= waitpid (0, &wstatus, WNOHANG))
>0){ //do sth with pid}
errno=e;
```

bei resistent Kindprozess   .. oder &wstatus
`while(waitpid(-1, NULL, WNOHANG))<0);`
`errno=e; WIFEXITED(wstatus) WEXITSTATUS(wstatus)`

Ringpuffer

```
struct BB{ size_t size;
size_t wi; size_t ri; SEM *es;
SEM *ws; int *buffer;};
BB *bbCreate (size_t size){
BB *b = malloc(sizeof..)); if(b==NULL)..
b->buffer =calloc(size, sizeof(int));
if(b->buffer==NULL){free(b); ..}
 b->size =size; b->wi=0; b->ri=0;
b->rs= semCreate (0); if(b->rs==NULL)
{free(b->buffer); free(b); ..}
b->ws= semCreate(size);
if(b->ws==NULL){free (b->buffer);
semDestroy(b->rs); free(b); ..}
return b;}
void bbPut(BB *b, int val){
P(b->ws);
b->buffer[b->wi]=val;
b->wi=(b->wi+1)%b->size;
V(b->rs);}
int bbGet(BB *b){
P(b->rs);
  size_t desired, readIdx;
int val; do{
  readIdx= bb->ri %bb->size
val= bb->buffer[readIdx];
desired =(bb->ri +1)% bb->size;
while( atomic_compare_exchange_strong(&bb->ri, &readIdx, desired)
 == false);
V(bb->ws); ret val;}
void bbDestroy(BB *b){
if(b==NULL) return;
semDestroy(b->rs); semDestroy(b->ws);
free(b->buffer); free(b);}
```

Stdin/out weiterleiten

```
int in =open(infd, O_RDONLY | O_CLOEXEC)
if(in==-1) die..;
if(dup2(in, STDIN_FILENO)==-1)die..
int out =open(outfd, O_RDWR |
 O_CREAT | O_CLOEXEC | O_TRUNC |
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
if(out==-1)die..
if( dup2(out, STDOUT_FILENO)==-1)
                             die...
```

No Zombies

```
struct sigaction ac={
  .sa_handler= SIG_DFL,
  .sa_flags =SA_NOCLDWAIT
}; if(sigaction(SIGCHLD, &ac, NULL) ..   ==-1
```

Strtol

```
errno=0; char *end; char *str;
long x = strtol(str, &end, 10);
if(errno!=0) die..
if(str==end || *end != '\0') fprint..
if(x>INT_MAX) fprintf
if(x<INT_MIN) fprintf
//cast to int
```

Verzeichnisse lesen

```
DIR *dir; struct dirent *dp;
 if ((dir=opendir(path))==NULL) die..
while((errno=0, dp =readdir(
dir)) != NULL){
if(strcmp(dp->d_name, ".."))..;
//do sth }
if(errno !=0) die "readdir";
if(closedir(dir)==-1) die..
```

Scandir (für threads)

```
struct dirent **entries;
int numEntr= scandir(path,
  &entries, filterCallback | NULL,
  alphasort);
if(numEntr==-1) die..;
for(int i=0; i<numEntr; i++){
  entries[i]; //do sth free(entries[i]);
} free(entries);
int callback(const struct dirent *entry){
 return 0->filter !=0 dont filter }
struct dirent{
  ino_t d_ino; //inode nr   nur die beiden
  off_t d_off;              sind POSIX
  unsigned short d_reclen; //record len
  unsigned char d_type; //file type
  char d_name[256]; //filename
}
```

Dateiinfos - Stat

```
struct stat s;
int r= stat(path, &s);   auch bei fdopen
if(r==-1){
  if(errno==ENOENT) //404
  else if(errno== EACCES) //403
  else die(..);}
lstat() -> folgt Symlinks nicht
S_ISREG(s.st_mode)
S_ISDIR(s.st_mode);
```

Strtok

Bsp mit Stack-Buffer

```
char buf[LINE_LENGTH]={0,};
//readbuffer
1. Aufruf: char *token= strtok(buf,
  "delimiters");
2+Aufruf token= strtok(NULL, "..");
token==NULL -> keiner mehr gefunden
->Threading: strtok_r
char *s= buffer;
strtok_r(buf, "..", &state);
```

qsort

```
qsort(array, #Elemente,
#Element-Größe, comparator);
static int comparator(const void
*a, const void *b){
//cast pointer!
return strcmp(*(const char **)a,
*(const char **)b); }
```

print strings

```
int sprintf(const char *s, format, ..);
int snprintf( -"- , size_t n, -"- ,..);
-> <0 errno
```

ABA-Problem 1. T_1 liest Variable mit Wert A
2. T_2 wird CPU entzogen, T_2 läuft
3. T_2 ändert Wert von A->B->A
4. T_1 bekommt CPU, prüft A
-> T_1 hat Änderung nicht gesehen

String functions

```
int strcmp(const char *s1, const char *s2)
int strncmp   + size_t n   als 3. Arg
0: s1=s2  1: s1>s2  -1: s1<s2
char *strchr(const char *s, int c);
char *strrchr( -  // -);
-> erste/letzte occurence vom char
  im String oder NULL falls nicht gefunden
char *strcpy(char *dst, const char *src)
char *strcat(const char *dst, const char *src);
return: pointer auf dst   -> dst+src (cat)
! Größe (Nullbyte nicht vergessen)
char *strdup(const char *s)
char *strndup( -"- , size_t size);
     -> nur size+1 chars kopieren
-> return pointer auf neuen Str / NULL + errno
! pointer free'en
```

Makefile

```
.PHONY: all clean
 CC=gcc
CFLAGS= -std=c11 -pedantic -Wall -Werror
-D_XOPEN_SOURCE=700
all: rush
rush: rush.o plist.o shellutils.o
clash: clash.o plist.o shellutils.o
%: %.o
    $(CC) $(CFLAGS) -o $@ $^
rush.o: rush.c plist.h shellutils.h
clash.o: clash.c plist.h shellutils.h
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<
clean:
    rm -f rush clash rush.o clash.o
```

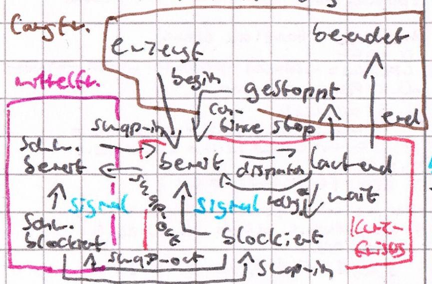
dynamic: -fPIC

```
libjbuffer.a: jbuffer.o sem.o
    ar -rcs $@ $^
jbuffer: ...o libjbuffer.a
    $(CC) $(CFLAGS) -o $@ $< -L. -ljbuffer
```

Prozessverwaltung

- langf. Planung: Lastkontrolle, erzeugt/fern bis zur Programmausführung
  - gestoppt: einheitl. Forts./Beendig.
  - beendet: evtl. Entsorgung
- mittelf. Pl.: Umlagerung von Prog.
  - schwebend bereit: ausgelagert, Erfüllung außer Kraft
  - schwebend blockiert: ausgelagert, ereigniswartend
- kurzfr. Pl. → Mehrprozessbetrieb
  - bereit: in Bereitliste
  - laufend: vollzieht CPU-Stoß
  - blockiert: erw. Betr.mittel-Zuteilung

Langfr. / mittelfr.



Benutzerorient. Kriterien: Antw.zeit / Durchlaufz. / Termineinhaltung / Vorhersagbark.
systemorient. Krit.: Durchsatz, CPU-Auslastung / Gerechtigkeit / Dringlichkeit

kooperativ (FCFS)	präemptiv (VRR)
- abhäng. Prozesse	- unabh. Prozesse
- kein CPU-Entzug	- CPU-Entzug
↳ Monopolisierung	Ereignisbed. Verdrängung

deterministisch	probabilistisch (SPN, SRTF, HRRN)
- alle P.zeiten + Termine bekannt	- P.zeiten unbek.
- Einhaltung von Zeitgarantien	- keine Zeitgarantie
- CPU-Ausl. vorhersagbar	- CPU-Ausl. nur abschätzbar

statisch	dynamisch
- vor Betrieb d. Proz.	- während Betrieb
- vollst. Ablaufplan	- Stapelsys, interakt. / verteilte Sys, Echtz.
- strikte Echtz. Sys	- schwache / feste Sys

asymmetrisch	symmetrisch
- asymm. Multiprozessorsys	- identische Prozessoren
- ungleiche Verteilung der Proz. auf CPUs	- Lastausgleich: gleichm. Verteilung auf CPUs

- **FCFS:** Reihenfolge der Ankunft
- **RR:** Regelmäßige Umplanung
- **VRR:** Vorrangige Zeitscheibe, Vorzugsliste
- **SPN:** nach erwarteter Bedienzeit (kürzester zuerst)
- **HRRN:** SPN + Wartezeit berücksicht.
- **SRTF:** SPN + spontane Umplanung
- **MLQ:** Arbeitsbetrieb, nach Typ
- **MLFQ:** FCFS + Umplanung + altert Priorität

anteilmäßig

Betriebsmittel

wiederverwendbar / konsumierbar (dauerhaft / vorübergehend)
teilbar (präemptiv) / unteilb. (begrenzt) / verteilbar (unbegrenzt) / abtrennbar
Beispiele z.B. CPU, RAM, GPU
Ursache: Signal, Trap

Betriebsstrategien: Zielopt. Durchsatz ohne Verklemmungen / Verhungern
Bm.-Entzug bei Verteilung

Prozesssynchronisation

gleichz. Prozesse:
- vertikal: Multiplexen eines Prozessors
- horizontal: mehrere Prozessoren

einseitige Synchronisation
- ein beteiligter Prozess
- erfolgt logisch/bedingt
- Erzeuger/Verbraucher: warten auf (konsumierbares) nicht existierendes Bm.

mehrseitige Sync: - blockierend/nicht blockierend
- alle beteiligten Prozesse betroffen / gegens. Ausschluss

blockierende Sync
- blockieren im krit. Abschn. z.B. Semaphore
- Nachteil: Leistung, Robustheit, Interferenz

nichtblock. Sync. Lebendigkeit
- Vor: kein krit. Abs., atomare Operationen
- CAS; Notwendigkeit bzw.: Lesen/ändern/schreiben
- NT: komplex, Verhungern möglich

Monitor: ADT mit impliziten Syncregeln.
- Konzept: mehrere ... , Proz. warten außerhalb
- Mutexbed.: P warten auf M.-Eintritt
- Ereignisbed.: P warten auf hohe Bed.-Ende
- Blockierung durch Bed.-Variablen
- blockierend: vorrang. Signalnehmer
- nicht block.: vorrang. Signalgeber
- Hansen (blockierend): Signalgeber verlässt Mon., notify() / broadcast()
- Hoare (blockierend): SG verl. Mon. → signal() → genau ein Signalnehmer fährt fort
- Mesa (nicht block.): SG im Mon. signal()

Mutex spezielle Semaphore
- Freigabe nur von Besitzerprozess
- für ein/mehrs. Sync geeignet / Sem. fehlen

Sperre
- Unterbrechung: asynchr. zu laufenden P und OS → Fehlerlevel-Interrupt-Handler
- Fortsetzung: Synchron zu Sys.kern. Sequencer
- Verdrängung: ... Slice Synchron zu Planer

Verklemmungen: notw. Bed.
1. wechsels. Ausschluss
2. ein/mehrere Bm nach fordern
3. Bm unentziehbar
- nicht hinreichend: zirkuläres Warten

Vorbeugung: Abhängig. notw. Bed. (Vorlaufzeit)
Vermeidung/Umgehung: gieriger Algo zur Laufzeit, Bankiersalgo oder Bm.graph
Lsg: Bm.-entzug bei einem P oder Prozess zerstören

Adressräume

Progadr → log. A → virt. A → phys. A.
real: realer HS, lückenhaft, Verteilen von HW gegeben (unsegmentiert)
logisch: lückenlos, durch MMU abgebildet, gebrauchte Adressbereiche
virtuell: wie logisch, Adr nicht an Ort gebunden (von Lokalität im HS entkoppelt) → Paging, MMU Präsenzbit

eindimensional → Segmente
Seitendeskriptor: von MMU gespeichert: reale Adresse + Präsenzbit + Kachelnr.
TLB (Buffer/Cache) für Adr.-Übersetzungen → Latenzverbesserung

Adressumsetzer (Fetch): Bestimmungspunkte legen Arbeitsbereich in physikal. Speicher fest
Segmentdeskriptor von MMU gespeichert an Daten
- MMU: Basis (Segmentanfang)
- LIM: Segm.-Länge × Rechte + Expansionsrichtung
- Attribute: typ (Text/Daten/Stapel) (aufw./abwärts)

Pagefault

1. Adresszugriff
2. MMU: Präsenzbit nicht gesetzt
3. BS-Trap-Handler
4. a) Seite ungültig → SIGSEGV
   b) gültig (aber nicht anwesend):
   ↳ im Freiseitenpuffer (kürzlich ausgelagert) → einlagern
   ↳ nicht im Puffer? → einlagern
   ↳ freier Seitenrahmen vorhanden: Einlagerung anstoßen, blockieren
   ↳ kein freier Seitenrahmen: Seite suchen, auslagern anstoßen, blockieren des Proz., Einlagern anstoßen, blockieren
   → als anwesend markieren
5. letzten Befehl wiederholen

Platzierungsstrategien

- Bitkarte: Hohlräume fester Größe
- Lochliste: Hohlr. variabler Größe

Verfahren (für Lochliste)
- **best-fit:** Lochgröße aufsteigend, kleinstes passendes Loch suchen
  - ⊕ minimaler Verschnitt
  - ⊖ steigender Suchaufwand
- **worst-fit:** Lochgröße absteigend, größtes passendes Loch suchen
  - ⊕ konstanter Suchaufwand → schnell
  - ⊖ großer Verschnitt, begünstigt Zerstückelung
- **first-fit:** aufsteigend nach Adresse, erstes passendes Loch nehmen
  - ⊕ schnelle Zuteilung ⊖ Verschnitt, steigender Suchaufwand
- **next-fit:** first-fit rotierend, beginnt bei zuletzt zugeteilten Loch
  - ⊕ im Mittel gleichgroße Löcher, abnehmender Suchaufwand
- **buddy:** Aufteilung mit 2^n Potenzgröße

Fragmentierung

intern: angeforderte Größe < zugeteiltes Stück
Buddy: durch aufrunden
kein Zugriffsfehler auf ungenutzten Bereich
- Verschwendung durch OS vermeidbar

extern: Zerstückelung des Adressraums
Löcher in Summe groß genug für angeforderte Größe, aber einzelne Löcher nicht
- nicht vermeidbar → Verschnitt/Zug.

Ladestrategien

Einzelanforderung (on demand): laden nach Ansprechen der Adresse
Vorausladen: Heuristiken + OS kommt Anforderung zuvor

Ersetzungsstrategien

Lokal: Menge residenter Seiten des Prozesses als Schranke; Seitenfehler vorhersagbar / reproduzierbar
Global: Menge resid. Seiten ALLER Proz. als Schranke; Seitenfehler unvorhersehb. / nicht rep.-bar
Lokalitätsprinzip Proz. referenziert Stellen nahe/einander, aus benachbarte/dieselbe Referenzzeit

globale Verfahren
- **FIFO** zuletzt eingelag. S. ersetzen; verkettete Liste, mehr Rahmen → mehr Seitenfehler
- **LRU** am längsten nicht ref. Seite; Impl: Zeitstempel (Statistiken), 2te Chance durch Referenz-Bits
- **LFU** am seltensten ref. Seite; Impl: Zähler
- Second Chance (Clock) / Referenzbits

Seitenflattern (Trashing)
Ein/Auslagern dominiert Systemaktivität (verschwindet meist von alleine)

lokale Verfahren
- **Freiseitenpuffer** Pool/Cache für potentiell zu ersetzende Seiten → schnelles Finden von Seitenrahmen
- **Arbeitsmenge** kleinste Sammlung an Progr.texten/daten, um effiziente Programmausführung zuzusichern

Speicherung

kontinuierlich: in aufsteigend nummerierten kontinuierl. Blöcken
Identif.: Anfangsblock + Größe
- ⊕ schnelles Lesen, keine ext. Fragm., geeignet für Echtzeitsys.
- ⊖ finden freier Blöcke, int. Fragm., dyn. Erweiterung aufwendig

verkettet: verkettete Blöcke (Liste)
- ⊕ dyn. Erw. einfach
- ⊖ Metadaten brauchen Platz, Liste erh. Fragm., Zeigerfehler → unsicher

indiziert: pro Datei ein spezieller Block → Indexblöcke (oder pro Platte); enthält Blocknummern einer Datei; mehrere Stufen möglich → „Inode"; „Indizierung Adressen"

Freispeicherverwaltung

- gleich große Blöcke mit Bitvektoren (belegt?)
- Verweisliste mit freien Blöcken
- Optimierung: aufeinanderf. Blöcke zusammenfassen, Daten sl. auf freiem Speicher statt Liste

UNIX-Dateisystem

- **Boot:** Infos zum Laden von initialem Programm
- **Superblock:** Anz. Blöcke, #Inodes und Attribute
- **Inodes:** Dateiattr., UID, Zeitstempel, Hard Links, Rechte, Blöcke
- **Datenblöcke:** Dateien, Kataloge, Indexbl.
- **Ext2:** gruppierte Blöcke + Kopie des Superblocks / Grp

Journaling-FS: Änderungen als TAs → Protokollierung in Log-File
↳ Eintrag vor Daten schreiben
- bei Boot: Abgleich Lf und Undo/Redo
- Optimierung: Checkpoints

Copy-on-write / Log-Structured FS
- alle Änderungen auf Kopien → atomare Umschaltung
- ⊕ gute Schreibeffizienz
- ⊖ starke Fragmentierung

RAID

- **0:** Daten über mehrere Platten
  - ⊕ schnelle Transfers
  - ⊖ keine Redundanz
- **1: Mirror:** D gespiegelt auf 2 Pl.
  - ⊕ schnelles Lesen, eine P kann ausfallen ⊖ Speicherbedarf ×2
- **4: Paritätspl.** D über mehr. Pl + 1 Parit.pl. (XOR)
  - ⊕ schnelles Lesen, 1 Ausfall, ≥ 3 Pl.
  - ⊖ Paritätspl. stark belastet
- **5: verstreute Parität** Par.blöcke über mehrere Pl. verteilt
  - +/- wie 4
- **6: doppelte Par.-Blöcke** wie 5 + 2 Par.bl. → Ausfall ≤ 2 Pl

User-Threads (federgewicht)

- → nicht privileg. Code
- → Scheduling durch Programmierer
- + effiziente Umschaltung
- - blockierende Syscalls blockieren andere UThreads
- → nicht für Multiprozessorbetr. ausgelegt
- → kein eig. Adr.raum, Ausführungsstrang

Kernel-Threads (leichtgew.)

- → Threads mit privileg. Code
- → Scheduling durch OS
- → teilt Adr.raum mit anderem Proz.
- → teilw. Schnell für mehrproz. Betr.
- ⊕ keine Blockierung bei Syscalls
- - Erzeugung teurer als UThreads
- - Umschalten im Kern → langsam

Prozess (schwergew.)

→ Prozess bildet allein als Systemkern-Instanz im Speicherschutz isolierten Adr.raum statt

Ausnahmebehandlung

Trap: interne Ursache	Interrupt: externe Ursache
- synchron, vorhersehbar, reproduzierbar	- asynchr., unvorhersehbar, nicht reproduzierbar
- z.B. unbekannter Befehl, DIV/0	- z.B. durch anderen Switch Prozess E/A
Behandlung: Beendigung/Wiederaufnahme	Beh.: Status des unterbr. Proz. sichern, Wiederaufnahme

Betriebsarten

Einprogramm-/Stapelbetrieb
Ziel: Durchsatz, Durchlaufzeit, Prozessorauslastung
- manueller Betrieb (volle Kontrolle bei Betriebs-/Programmierer)

abgesetzter Betr.
- Satellitenrechner für I/O
- Schneller Hauptrechner nur für Berechnung

überlappte I/O
gleichz. Prozesse durch Prog. unterbrechungen / Speicherdirektzugriff DMA → Nebenläufigkeit
⊖ Leerlauf bei Auftragswechsel

überlappte Auftr.verarb.
Verarbeitungsstrom ausgewählter Progr.
Einplanung im Hintergrund, Vorgriff (Prefetching)
⊖ I/O Leerlauf, CPU-Monopol

abgesetzte I/O (Spooling) wechselnd CPU/IO-Stoß (entkoppelt durch Puffer)
⊖ Leerlauf im Wartezustand

↳ **Mehrprogrammbetrieb**
CPU-Multiplexen → Nebenläufigkeit
↳ Adressraum-Schutz, Segmentierung
- dyn. Laden: Ausführ. von Prog, das größer als HS ist → nachladen, aber Programmieren muss unterstützen

Mehrzugangsbetr.
- nur mit CPU/Speicherschutz sinnvoll

Dialogbetrieb
Fokus: Antw.zeit
- mehrere Nutzer, interakt.: Eingabe/Verarbeitung
- ⊖ CPU-Monopol

Dialogorient. Monitorsystem
- Proz.-Start im Vordergrund
- Vollziehung im Hintergrund, parallele Verarb.
- ⊖ Größe HS

Teilnehmer-Betr.
- eigene, interakt. Dialogprozesse
- Zeitscheibe gegen CPU-Monopol.
- ⊖ Sicherheit, Ein-Ausgang...

Teilhaber-Betr.
- gemeinsame, mehrfache Dialogprozesse → Client-Server-Systeme

Echtzeitbetrieb
Ziel: Termineinhaltung, Vorhersagbarkeit, Dringlichkeit

Terminvorgabe
- weich/soft: Erg. nutzbar → Wert verlust, Verletzung tolerierbar
- fest: Erg. wertlos, verwerfen, Verl. tolerierbar
- hart: verspätetes Erg kann zu Katastrophe führen → Verl. nicht tolerierbar

```
static int a=1;        code
void f(const int *i){
  static int s;        → BSS
  int c;
  char *d=malloc(2);   → Heap
  void (*e)(const int*) = f;
  4+i;
}
```
→ 3 ... (uninitialisiert) → mit 0 belegt