Übersetzung: ① Präprozessor (Includes, Makros, Kommentare)
② Compiler → ASM → Assembler
③ Assemblieren → in Maschinencode
④ Binden: dyn vs statisch (Laufzeit/Ladezeit)

Programm: Folge von Anweisungen
Prozess: Programm in Ausführung

Mehrebenenmaschine
⑤ Problemorient. Programmierebene
④ Assemblierebene ← 2 → 3+4
③ Maschinenprogrammebene
② Befehlssatzebene ← Interpretation → Übersetzung
① Mikroarchitekturebene
⓪ digitale Logikebene

Maschinenprogramm: Anwendungsprog + Laufzeitsystem
↳ Maschinenbefehle d. ① & Systemaufrufe
↳ Ausführungsplattform: Betriebssystemabstraktion
unprivilegierte Befehle: CPU direkt ausführen
privilegierte Befehle: von BS ausgeführt als Trap
↳ Ablauf: 1. CPU interpretiert Maschinencode
2. Bei Trap startet BS, interpretiert
3. Programm d. BS schrittweise
4. CPU wird zum weitermachen verleitet

Trap: interne Ursache, synchron, vorhersagbar, reproduzierbar, zB IO, Systemaufruf
Interrupt: externe Ursache, asynchron, ...
→ Behandlung nebeneffektfrei
→ E/A-Operation, Seitenfehler global

USER-THREADS, federgewichte
- Realisierung auf Anwendungsebene,
- Systemkern sieht nur 1 Kontrollfluss
⊕ Erzeugung billig, effizient umschalten
⊖ Systemkern kein Wissen → Scheduling durch Programmierer
↳ Multiprozessorsystem keine Parallelisierung
→ blockiert 1 Thread → alle Threads blockiert
→ kein eigener Adressraum und Ausführungsstacks

KERNEL-THREADS
- Leichtgew.: teilt sich Adressraum
- ~~[illegible]~~
- eigene Ausführungsstacks
⊕ nutzen gem. Betriebsmittel 1 Prozesses
⊕ Betriebssystemkern kennt Faden: Multiprozessor
⊖ Erzeugung teurer als USER-THREAD

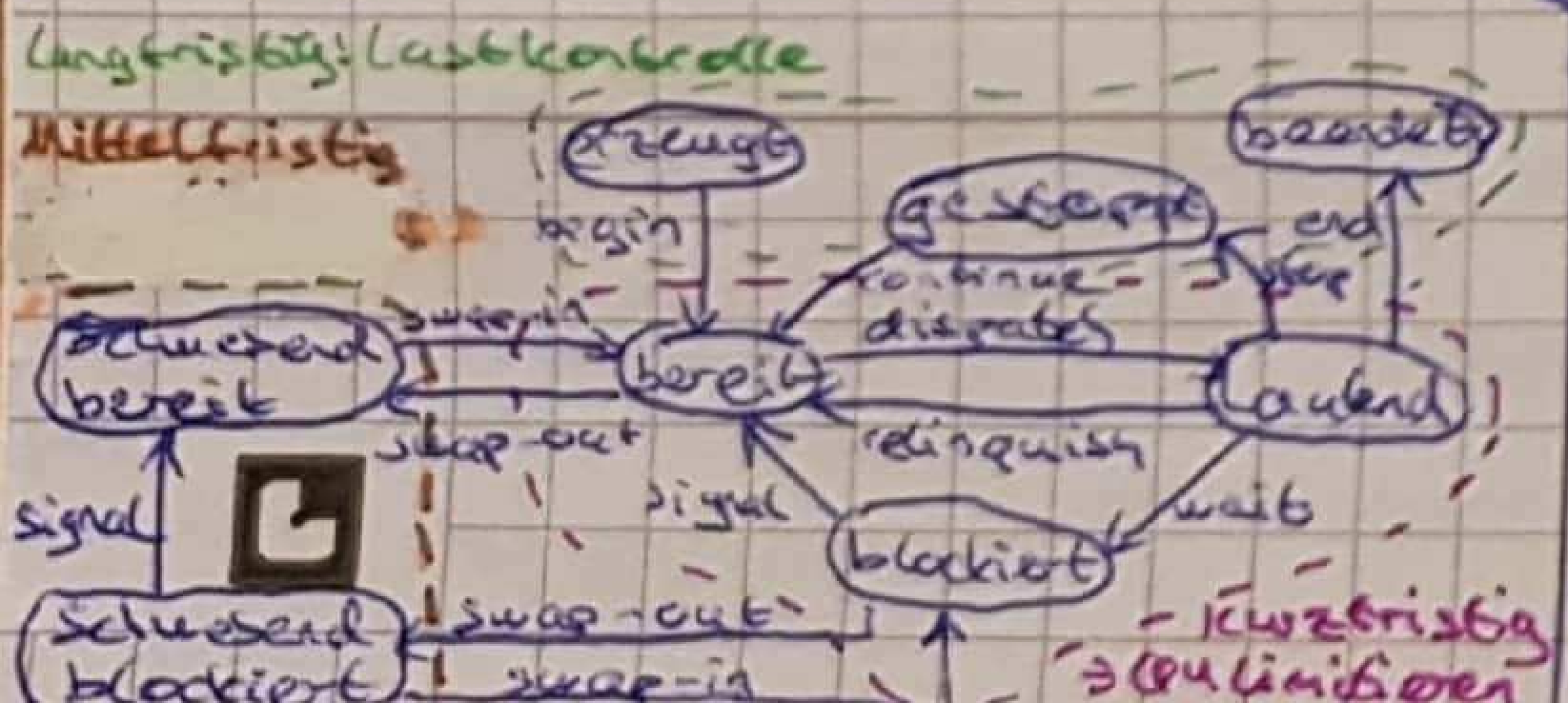
PROZESS: Ausführungsumg. mit Aktivitätsträger

→ Speicherbereiche

Stack	- lokale Variablen, Argumente, alte Rücksprungadr
Heap	von malloc reserviert
BSS	nicht init. globale/statische Var
Data	init globale/statische Var
Text	Programmcode

Adressraumschutz: → phys Speicherschutz
1 Abteilung: Schutzgatter trennen BS und Programme
2 Eingrenzung: Begrenzungsregister pro Programm → Überlappen möglich
3 Segmentierung: log. Adressraumschutz, Basis + Länge

Adressraum: reell: lückenhaft, echte Hauptspeicher
logisch: lückenlos; virtuell: scheinbar echt, pro Prozess



Prozessverwaltung
Gütemerkmale
allg: Gerechtigkeit, Lastausgleich
Stapelbetrieb: Durchsatz, Prozessorauslastung
Dialogbetrieb: Antwortzeit
Echtzeitbetrieb: Pünktlichkeit, Vorhersagbarkeit
Benutzer vs System (Betriebsauslastung)

Klassifikation
1 kooperativ (FCFS): alle Prozesse
- kein CPU-Entzug
- laufende Prozess gibt durch Systemaufruf CPU ab
⊖ CPU-Monopolisierung
1 preemptiv (RR, VRR, SRTF):
- unabh. Prozesse
- CPU-Entzug, ereignisbedingte Verdrängung laufender Prozesse
2 deterministische Planung
- alle Prozesszeiten bekannt
- Einhaltung von Zeitgarantien sicherstellen
- CPU-Auslastung vorhersagbar
2 probabilistische Planung (SPN, SRTF)
- Prozesszeit unbekannt → nur approximiert
- keine Zeitgarantie, CPU-Auslastung nicht vorhersehbar
3 statische Planung vor Betrieb Prozess
→ strikte Echtzeitsysteme [offline]
3 dyn Planung: während Betrieb Prozess [on-line]
4 asym Planung
→ ungleiche Verteilung Prozesse auf CPUs
→ wichtig für asym. Multiprozessorsys
4 sym Planung
→ identische Prozessoren ⇒ Lastausgleich

FCFS (First Come First Serve)
- Einplanung nach Reihenfolge Ankunft
→ Begünstigung langer Rechenstöße
ℰ Konvoieffekt: kurze Prozesse folgen einem langen

2 **RR und VRR** (Round Robin)
- verdrängende Variante von FCFS
↳ Zeitscheiben oder period. Unterbrechungen ⇒ CPU-Schutz
ℰ Konvoieffekt
- **VRR:** variable Zeitscheiben
Vorzugsliste für IO

3 **SPN** (Shortest Process Next)
- schnellster als nächstes
ℰ Prozessverhungern möglich
a priori Wissen über Prozesslaufzeit nötig → Schätzung nötig

4 **HRRN** (Hungerfreies SPN durch Alterungsgewichtung)

5 **SRTF** (Shortest remaining time first)
- verdrängendes SPN
- Verdrängung wenn neuer Prozess kürzere Bedienzeit als laufende Restdauer ⇒ Umplanung
→ nach Restdauer in Bereitliste einsortiert

6 **MLQ:** Mischbetrieb
- lokale Einplanung nach Typ & Ankunftszeit
- globale Einplanung

7 **MLFQ**
- nach FCFS mit Verdrängung
- Bestrafung: lange Rechenstöße
- Alterung: Begünstigungsfrist

Einplanung: Reihenfolgebildung Aufträge
Einlastung: Zuteilung von Betriebsmittel

Synchronisation
- **einseitig:** → logisch oder zeitlich; Auswirkung auf nur 1 beteiligten Prozess (zB Signal/bedingt)
- **mehrseitig:** auf alle beteiligten Prozesse zB pthread
→ blockierend vs nicht blockierend

Verhungern: andauernde Benachteiligung von Prozessen

Verklemmung: irreversible gegenseitige Blockierung von Prozessen
↳ notwendigen Bedingungen
1 wechselseitiger Ausschluss
2 Nachfordern von Betriebsmittel
3 Unentziehbarkeit zugeteilter Betriebsmittel
hinreichend: 4 zirkuläre Warten
→ Virtualisierung: Entzug phys Betriebsmittel, virtuelles bleibt

blockierend:
⊖ Leistung (seq. Anteil)
⊖ Robustheit (Absturz im krit Bereich)
⊖ Interferenz (Planungsentscheidung nicht umsetzen)
⊖ Lebendigkeit (Gefahr Verklemmung)
→ pessimistischer Ansatz, wechselseitiger Ausschluss, System aufrufe

nicht-blockierend: optimistisch,
⊕ effizienter keine Verklemmung
⊖ schwieriger zu programmieren, Verhungern
zB CAS: lesen, lokal arbeiten, Versuch Änderung zurückzuschreiben
→ Transaktion (komplett oder garnicht)

SPIN LOCK: Lucksperre, warten bis Lock auf 0 gesetzt wird

SEMAPHORE: Variable mit 2 atomaren Operation (ganzzahlig)
- P: dekrem. Wert um 1, blockiert wenn 0
- V: inkrementiert um 1, benachrichtigt ggf blockierte Prozesse

Mutex: Semaphorspezialisierung
↳ prüft Eigentümerschaft bei Freigabe

Monitor: abstrakter Datentyp, dessen Zugriffsoperationen implizit synchronisiert sind
- Prozess warten außerhalb Monitor
↳ mehrseitige Sync
↳ Monitor WS: auf Monitoreintritt
- Ereignis WS: auf Wartebedingungsende
↳ einseitige Sync
→ nicht blockierend: Signalgeber bleibt im Monitor → Mesa, Java
→ blockierend: Signalgeber verlässt Monitor nach Signalisierung
→ Hansen: signalisiert alle
→ Hoare: nur einen

Koordinierung
- on-line: explizite Sync durch Anweisungen
kein Vorabwissen, konstr. Ansatz
- off-line: implizite Sync, analyt. Ansatz

Fortschrittsgarantien (Lebendigkeit)
- behinderungsfrei: isolierter Prozess beendet nach endl. Schritten
- sperrfrei: systemweiter Fortschritt
- wartefrei: frei von Aushungerung

Betriebsmittel: Dinge die Programme brauchen
- wiederverwendbar: teilbar/unteilbar, begrenzte #, Anforderung & Freigabe, zB Prozessor, Speicher, kritischer Abschnitt
- konsumierbar: unbegrenzte #, Erzeugung & Zerstörung, zB Signale, Unterbrechung, Nachrichten

---

Platzierungsstrategien
- Bitkarte: Halträume fester Größe
- Lochliste: variable Größe
1) **worst-fit:**
→ absteigend nach Lochgröße sortiert
↳ größte passende Loch
⊕ kleiner Suchaufwand
⊖ großer Speicherverschnitt
2) **best-fit**
→ aufsteigend nach Lochgröße sortiert
↳ kleinste passende Loch
⊕ kleiner Speicherverschnitt
⊖ größerer Suchaufwand
3) **first-fit:** nach Adr sortiert
- erste passende Loch
⊕ kleiner Suchaufwand
⊖ großer Speicherverschnitt
4) **next-fit:** nach Adr sortiert
→ wie first-fit, Suche startet bei zuletzt zugeteilten Loch

Interne Fragmentierung (Paging)
↳ angefordertes Speicher < zugeteilter
↳ Zugriff nicht angef. Bereich löst kein Trap

Externe Fragmentierung:
Speicherblöcke zsm ausreichend, doch einzelnd zu klein + nicht zsmhängend

Verschmelzung: nebeneinanderliegende Blöcke
Kompaktifizierung: Umsortierung
↳ Kopieren, Adr ändert sich
5) Buddy-Verfahren: nächste 2er Potenz

Ladestrategie
Umsetzung durch MMU
- Demand Paging: laden beim Ansprechen in HM
- Anticipatory: Vorausladen

Pagefault
1. MMU löst Trap aus
2. BS löst Handler für pagefault aus
3. Freier Platz in Hauptspeicher gesucht evtl. auslagern
4. Einlagerung wird angestoßen
5. Prozesszustand blockiert
6. Present Bit auf 1
7. Interrupt, Einlagerung abgeschlossen
8. Prozess bereit
9. Nach Einlastung: erneuter Speicherzugriff
Prozess laufend

Ersetzungsstrategien
- lokal: Suchraum Seiten des Prozess
↳ Seitenfehler reproduzierbar
- global: Seiten aller Prozesse
↳ Seitenfehler nicht vorhersehbar

Lokalitätsprinzip: Referenzierung auf gleiche oder direkte Umgebung im Anschluss

Seitenflattern: sofortige Einlagern kürzlich ausgelagerter Seiten
↳ dominiert Systemaktivität

Freiseitenpuffer: FIFO Cache für potenziell zu ersetzenden Seiten
→ phys anwesend, Present Bit entfernt

Arbeitsmenge: kleinste Menge an Seiten für effiziente Programmausführung

Strategien:
1 **MIN:** Seite auf die am längsten nicht mehr referenziert wird
→ optimal, nicht implementierbar
2 **FIFO:** zuletzt eingelagerte
→ mehr Seitenrahmen, ggf mehr Seitenfehler (mehr Liste)

→ implementiert durch Zähler
3 **LFU:** am seltensten referenziert
4 **LRU:** am längsten nicht referenziert
→ period. Unterbrechungen & Hintergrundrauschen
- Altersstruktur: Schieberegister
- 2. Chance: Referenzbit
- 3. Chance: Referenz + Modifikationsbit

Dateisysteme
1 **kontinuierliche Speicherung**
↳ aufsteigend num. Blöcke
→ Anfangsblock + Größe zur Identifik.
⊕ schnelles Lesen, keine ext Fragmentierung
↳ Echtzeitsys
⊖ interne Fragm, dyn Erweiterung schwierig, freien Speicher finden schwierig

2 **Verkettete Speicherung:**
↳ verkettete Blöcke → Liste
⊕ einf. dyn Erweiterung
⊖ ext Fragm, hohe Fehleranfälligkeit

3 **Indizierte Speicherung**
↳ spez. Platte mit Blocknummern
→ Blocknummern ident. Datenblock
⊖ interne Fragm.
Extends: kontinuierliche Speicherung für größere Datenblöcke

4 **Freispeicherverwaltung**
↳ Bitvektor ob Block belegt oder Liste verketteter freier Blöcke

Unix Blockstruktur
1 Bootblock: ~~[illegible]~~ laden initiales Program
2 Superblock: Anzahl Blöcke/Inodes ~~[illegible]~~
3 Inode: systeminterne Verwaltungsstr. die Dateiattribute enthält
4 Datenblock

FAT-Ansatz: | 9 | | 8 | | | | | | -1 | 3, 8, 1, 9
⊕ Datenblock komplett nutzbar
⊕ mehrfache Speicherung von FAT

Journaling-File-System
- vor jeder Transaktion geloggt
- bei Systemabsturz inkonsistent → konsistenten Zustand durch redo/undo
- checkpoint: konsistent → Logs löschen

Copy-on-Write:
Änderungen auf Kopie
⊕ Schreibeffizient, Datenkonsistent
⊖ starke Fragmentierung

Raid 0: Daten über mehrere Platten gespeichert
⊖ keine Fehlertoleranz
Raid 1: n Platten mit gleichen Daten
⊕ n-1 Platten können ausfallen
Raid 4: wie 0 + Paritätsplatte P
⊖ P hoch belastet, mind 3 Platten
Raid 5: Paritätsblöcke verteilt
Raid 6: 2 Paritätsblöcke pro Platte

Verzeichnis

```
DIR *dir = opendir(path)
if(!dir) { die }
struct dirent *d
while(errno=0, (d=readdir(dir))){
    struct stat sb;
    char newPath[strlen(dir) + 1
    + strlen(d->d_name) + 1];
    sprintf(newPath, "%s/%s",
    dir, d->d_name);
    if((stat(newPath, &sb)){
      perror(), continue;}
    if(d->d_name[0]=='.'){
      continue;}
    if (S_ISREG(sb.st_mode)){
      ...}
  }}
if(errno!=0){die(readdir...);}
if(closeDir(dir)!=0){die}
```

CHDIR

```
if(chdir(path)!=0) die
```

FGETS

```
char buf[MAXLEN+1];
FILE *fp = fopen(path, "r");
if(!fp) die
if(fgets(buf, sizeof(buf), fp)==NULL)
  {if(ferror(fp)) die}
if (fclose(fp)!=0) die
```

ZEICHENWEISE:

```
int c;
while((c=fgetc(fp))!=EOF){
  if (putc(c, cl)==EOF) break;}
if (ferror(fp)||ferror(cl)) die
```

```
if(fprintf(fp, "%s&%d, string2)<0){
  -> die}
if (fflush(fp)==EOF) die
```

dup2

```
int fd=open(path, rights)
       [oder fileno(fp)]
if (fd==-1) die
if(dup2(fd, STDIN/STDOUT_FILENO)
  ==-1) die
```

```
                  argv[0] = Prog
int main(int argc, char *argv[]
```

```
static void die(const char *m){
  perror(m); exit(1);
```

```
int *feld=calloc(len, sizeof(int));
if(feld==NULL) die
int *feld2=malloc(len*sizeof(int));
if(feld2==NULL) die
feld=realloc(feld, 2*len*sizeof(int));
free(feld); free(feld2);
```

MAKEFILE

```
.PHONY: all clean
all: name
clean: rm -f name, name.o
name: name.o          [-pthread]
    gcc -o name name.o
name.o: x.h name.c
    gcc -c name.c
```

```
int sock = socket(AF_INET6, SOCK_STREAM, 0)
if(sock==-1) die
struct sockaddr_in6 ad={
  .sin6_family=AF_INET6,
  .sin6_port =htons(PORT),
  .sin6_addr =in6addr_any};
if(bind(sock, (const struct sockaddr *)
  &ad, sizeof(ad)) !=0) die
if(listen(sock, SOMAXCONN)!=0) die
while(1){
  int cl=accept(sock, NULL, NULL);
  if (cl==-1){perror, continue;}
  handleConnection(cl);
  close(cl)
} close(sock);
```

Server

Client

```
struct addrinfo hints ={
  .ai_socktype= SOCK_STREAM,
  .ai_family = AF_UNSPEC,
  .ai_flags = AI_ADDRCONFIG};
struct addrinfo *result;
int r=getaddrinfo(HOSTNAMESERVER,
   "PORT", &hints, &result);
if(r!=0) die
struct addrinfo *tmp; int sock;
for(tmp=result; tmp!=NULL; tmp=tmp->ai_next){
  sock=socket(tmp->ai_family, tmp->ai_socktype,
         tmp->ai_protocol);
  if(sock==-1)die
  if (connect(sock, tmp->ai_addr, tmp->ai_addrlen)
    ==0) {break;}
  close(sock);}
if(tmp==NULL){//keine Adr gefunden}
freeaddrinfo(result);
FILE *s = fdopen(sock, "r");
if(!s) die
```

Dup

```
int sock2=dup(sock);
if(sock2==-1) die
FILE *rx=fdopen(sock, "r");
if(!rx) {perror, close(sock),
        close(sock2), exit(1);}
FILE *tx= fdopen(sock, "w");
if(!tx){perror, fclose(rx), close(sock2),
        exit(1)}
```

```
struct dirent **namelist; int n;
n=scandir(path, &namelist, &filter, alphasort);
if(n==-1) die
while(n--){ //do; free(namelist[n]);
free(namelist);
int *filter(const struct dirent *d){
  //0 Filter, 1 sonst}
```

```
TYPE array[len];
qsort(array, sizeof(TYPE), len, cmp);
static int cmp(const void*p1, const void*p2){
  const struct TYPE *a1=(const TYPE *)p1;
  ->analog a2= p2;
  return strcmp(a1->x, a2->x);}
```

```
pid_t p= fork();
if(p==-1) die
if(p==0) {//Kind! EXIT!
                -> kein return
else {//Eltern
    int stat;
  pid_t x=waitpid(p, &stat, 0);
  if(x==-1){ die}
```

```
int event, pid_t p;
while((p=waitpid(-1, &event,
        WNOHANG)) >= 1) {...}
if(errno!=0) die
```

```
pthread_t tid[#THREADS];
for(int i=0; i<#THREADS; i++){
  errno=pthread_create(&tid[i],
       NULL, fkt, void *args);
  if(errno!=0) die
} } SIGNALE IGNORIEREN }
  pthread_detach(tid);
void *fkt(void *arg) {
  pthread_detach(pthread_self());
  struct args *a=(struct
       args *) arg;
```

```
errno=pthread_join(&tid, NULL);
if(errno!=0) die
```

STRTOL: *char => int

```
int x; long lx, char *endptr;
errno=0;
lx=strtol(s, &endptr, 10);
if(!(*s!='\0' && *endptr=='\0')
  die}
if(lx>=INT_MIN && lx<=INT_MAX){
  x=lx;}
=> s wird verändert => Kopie
```

Kopie von s

```
char dest[strlen(s)+1];
char *dest=strcpy(dest, s);
```

STRTOK_R zB GET <name>\n

```
char *saveptr;
char *get =strtok_r(l, " ", &saveptr);
if(get==NULL || strcmp(get, "GET")!=0)
  -> Fehler
char *name=strtok_r(NULL, "\n",
                  &saveptr);
if(name==NULL) ->Fehler
=> _r -> Threadsafe, strtok
   sonst ohne Letzter
```

Signalbehandlung:

```
struct sigaction sig ={
  .sa_handler= SIG_IGN |
          SIG_DFL | handler,
  .sa_flags = SA_RESTART |
          SA_NOCLDWAIT};
sigemptyset(&sig.sa_mask);
if(sigaction(SIGNAL, &sig, &oldSig)
  ==-1) die
 SIGNAL: SIGCHLD, SIGPIPE
        SIGINT, SIGALARM
->Zombies aufsammeln,
  Pipe ignorieren
```

```
static void handler(int sig){
  int backup = errno;
      ⋮   kein die!
  errno = backup;}
```

Signal blockieren

```
sigset_t mask, oldMask;
sigemptyset(&mask);
if(sigaddset(&mask, SIGNAL)==-1)
  die;
if(sigprocmask(SIG_BLOCK,
  &mask, &oldMask)==-1)
   die
->Blockiert
if(sigprocmask(SIG_SETMASK,
  &oldMask, NULL) == -1)
   die;
```

Passives Warten bsp Childs

```
->SIGCHLD blockieren
                  volatile
while(childcounter >= #maxChilds){
  sigsuspend(&oldMask);}
sigprocmask(SIG_SETMASK, &oldMask,
        NULL);
```

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

```
void P(SEM *sem){
  pthread_mutex_lock(&sem->m);
  while(sem->value<=0) {
    pthread_cond_wait
     (&sem->c, &sem->m);}
  sem->value--;
  pthread_mutex_unlock(&sem->m)
```

```
void V(SEM *sem){
  pthread_mutex_lock(&sem->m)
  sem->value++;
  pthread_cond_broadcast
     (&sem->c);
  pthread_mutex_unlock(&sem->m)
}
```

```
struct BB {
  int volatile read, write;
  SEM *full, *empty, *krit;
  int values[];};
```

```
BB *create(size_t size){
 if(size==0||size>INT_MAX){
   return NULL;}
 BB *bb=malloc(size*sizeof(int)
             +sizeof(BB));
 if(bb==NULL) return NULL;
 bb->read =0;
 bb->write=0;
 bb->size = size;
 bb->full=semCreate(size);
 bb->empty=semCreate(0);
 bb->krit=semCreate(1);
 return bb; }
```

```
void bbPut(BB *bb, int value){
  P(bb->full);
  P(bb->krit);
   bb->values[bb->write]=value;
  bb->write=(bb->write+1)%
         bb->size;
  V(bb->krit);
  V(bb->empty);
```

```
int bbGet(BB *bb){
  P(bb->empty);
  P(bb->krit);
  int r= bb->values[bb->read];
  bb->read=(bb->read+1)%bb->size;
  V(bb->krit);
  V(bb->full);}
=>Alternativ mit CAS ohne krit
  int lr, lrn, ret;
  do{
    lr=bb->read;
    lrn=(bb->read+1)%bb->size;
    ret=bb->values[lr];
  while(!CAS(&(bb->read, lr, lrn));
```

```
SEM *semCreate(int initValue){
  SEM*sem = malloc(sizeof(SEM));
  if(sem == NULL) return NULL;
  sem->value = initValue;
  if(pthread_mutex_init(&sem->m,
    NULL)!=0) { free(sem); return NULL;}
  if(pthread_cond_init(&sem->c,
  NULL)!=0) {
    pthread_mutex_destroy(&sem->m);
    free(sem); return NULL;}
  return sem;}
```

```
int execl(char *path, char *arg1, ...)
int execlp(char *file, char *arg, ...)
=>wenn zurück immer Fehler
  => die
```