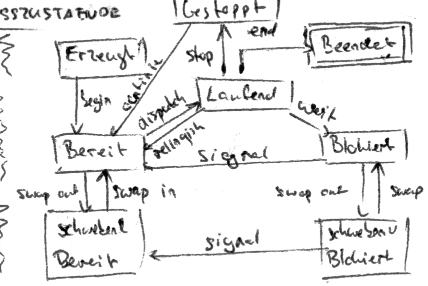
PROZESSZUSTÄNDE

LANG (M -> h)

KURZ (μs -> ms)

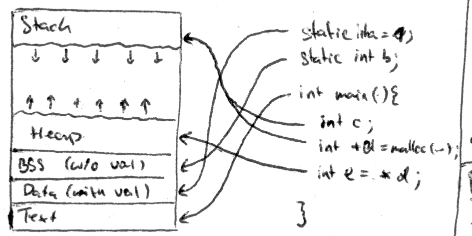
MITTEL (ms -> s)



EINPLANUNGSVERFAHREN

Name	Koop	Verd.	Pros.	Vorh.	Descr.
FCFS	X			X	Abarbeiten nach Queue
RR		X			FCFS mit Zeitscheiben
VRR		X			RR mit Vorzugsliste
SPN	(x)		X	X	Planen nach kürzeste erwarteter Laufzeit
HRRN	(x)		X		SPN mit berücksicht. von Wartez.
SRTF		X			SPN mit spontanem Umplanung
MLQ		X			Prioritätsebenen mit untersch. Zeitscheiben größe

SPEICHERRAUM



RAID-System

- RAID 0 -> Mehrere Festplatten
- RAID 1 -> Mirror-Storage
- RAID 4 -> 3-Platten + Parität (xor)
- RAID 5 -> 3-Platten + verteilte Parität
- RAID 6+ -> Mehr Platten und Parität

SEMAPHORE

P: -1 (prüfen)

V: +1 (freilassen)

-> von jedem Prozess freigebbar

<=> Mutex!

BETRIEBSSYSTEM UNTERBRECHUNGEN

trap -> Ausnahme interner Ursache (synchron, reproduzierbar)

interrupt -> Ausnahme externer Ursache (asynchron, unvorhersehbar)

BETRIEBSMITTEL sind

- wiederverwendbar und…
  - teilbar (präemptiv)
  - unteilbar (zeitweise exclusiv)
- Konsumierbar

FORTSCHRITTSGARANTIEN

- wait-free: + systemweiter Fortschritt – Aushungern
- lock-free: + systemweiter Fortschritt + Aushungern
- obstruction-free: allein -> wait-free

MONITORE

Kritischer Bereich, mehrseitig Synchro nach außen, einseitig innen.

-> Monitor warteschlange: PE warten auf Eintritt in Monitor

-> Ereigniswarteschlange: PE warte auf Aufhebung einer Wartebedingung

- Hansen: blockierend Bedingungsvar (-> Vorrang Signalnehmer) alle Signalnehmer bereit.
- Hoare: blockierend Beding. variable, ein Signaln. auf bereit
- Mesa: nicht-blockierend Bedingungsvar (-> Vorrang Signalgeber) ein o. alle Signalnehmer auf Bereit.

EINPLANUNGS-ALGORITHMEN

cooperativ: Abhängig von Prozessen

präemptiv: Unabhängig …

determ.: alle Proc. bekannt, Zeitgarantie.

probab.: nicht bekannt …

off-line: vor Betrieb alles Bekannt

on-line: nicht bekannt.

asymet.: verschiedene Programme haben nicht gleiche Ausstat.

symetrisch: alle Programme haben gleiche ISA

ADDRESSRÄUME

- real: lückenhaft, echte Hauptspeich
- logisch: lückenlos, jede Addr. gültig
- virtuell: scheinbare Echtheit pro Prozess

SPEICHER PLATZIERUNGSSTRATEGIE

- best-fit: kleinstes Passend Loch (langsam)
- worst-fit: größtes Passend Loch (langsam)
- first-fit: erstes passende Loch (schnell)
- next-fit: fit nach letztem Loch (mittel)
- buddy: halbierung des Speicher (einfach)

DATEISYSTEME

- kontinuierliche Sp.: nacheinander folgend Block
- Verkettete Speic.: Blöcke verweisen auf nächsten (extern -> FAT)
- Indiziertes Speic.: Plattenblock mit Blocknummern, welche Blöcke müssen gelesen werden

VERZEICHNIS LESEN

<dirent.h, sys/types.h, /stat.h>

```
DIR *dp = opendir("⊗");
if (dp == NULL) → error
struct dirent *ent;
while (errno = 0, ent = readdir(dp)) {
    → process ent
if (errno) → readdir error
if (closedir(dp)) → error
```

ent->d_name ← Dateiname
ent->d_type ← Typ (DT_DIR, DT_LNK, DT_REG)

ZEILEN LESEN

<stdio.h>, N=N+1

```
char buf[N];
FILE *fp = fopen("⊗", "r");
if (fp == NULL) → error;
while (fgets(buf, N, fp))
    → process buf
if (ferror(fp)) → fgets error
if (fclose(fp)) → error
```

getline(char**, size_t*, FILE*)
*NULL → must free
-1 for error

SIGNALE BEHANDELN

<signal.h>

```
struct sigaction sa = {
    .sa_handler = ⊗
    .sa_flags = SA_RESTART|
                SA_NOCLDWAIT}
if (sigaction(SIG..., &sa, NULL) == -1)
    → fehler
```

```
sigset_t new, old;
sigemptyset(&new);
sigaddset(&new, ⊗);
sigprocmask(SIG_BLOCK, &new, &old);
while (...) sigsuspend(&old);
sigprocmask(SIG_SETMASK, &old, NULL);
```

SEMAPHORE <sys/sem.h>, <pthread.h>

```
typedef struct {
    volatile int count;
    pthread_mutex_t mut;
    pthread_cond_t cond; } SEM;
SEM *semCreate(int val) {
    SEM *s = malloc(sizeof(SEM));
    if (!s) → error
    s->count = val;
    if (pthread_mutex_init(&s->mutex,
                           NULL)
        → error
    if (pthread_cond_init(&s->cond,
                          NULL)
        → error
    return s
}
```

PIPE <unistd.h>

```
int fd[2];
char send = "...";
if (pipe(fd)) → error
pid_t pid = fork();
if (pid == 0) {
    close(fd[0]);
    write(fd[1],
          send,
          1 + sizeof(send));
if (-1 == pid) → error
else { close(fd[1]);
    char buf[100] = {0};
    read(fd[0], buf, 99); }
```

SERVER INIT <sys/socket.h>

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) → error
struct sockaddr_in6 sin = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(PORT),
    .sin6_addr = in6addr_any, }
if (-1 == bind(sock, (struct sockaddr*)&sin,
               sizeof(sin))) → error
if (-1 == listen(sock, SOMAXCONN)) → error
while (1) { int conn = accept(sock, NULL, NULL)
    if (conn == -1) → error
    // use conn
    close(conn); }
close(sock);
```

SERVER KLIENT <unistd.h, stdio.h>

```
int dconn = dup(conn);
if (conn < 0) → error
FILE *rx, *tx;
if (!(rx = fdopen(conn, "r"))) → error
if (!(tx = fdopen(dconn, "w"))) → error
// use rx and tx
if (fclose(tx) || fclose(rx)) → error
```

INSPECT FILE <sys/stat.h>, <unistd.h>

```
struct stat *buf;
stat(⊗, &buf); // filename
fstat(⊗, &buf); // file-pointer
lstat(⊗, &buf); // symb-link
```

```
void semDestroy(SEM *s) {
    if (s) {
        pthread_mutex_destroy(&s->mut);
        pthread_cond_destroy(&s->cond);
        free(s); } }
```

```
void P(SEM *s) {
    pthread_mutex_lock(&s->mut);
    while (s->count <= 0)
        pthread_cond_wait(&s->cond, &s->mut);
    s->count--;
    pthread_mutex_unlock(&s->mut); }
```

```
void V(SEM *s) {
    pthread_mutex_lock(&s->mut);
    if (s->count == 0)
        pthread_cond_broadcast(&s->cond);
    s->count++;
    pthread_mutex_unlock(&s->mut); }
```

THREAD STARTEN <pthread.h>

```
pthread_t tid;
pthread_create(&tid, NULL, f, arg)
    → error
pthread_detach(tid)
// or, wait
void *val
pthread_join(tid, &val) == 0;
// and
pthread_exit(val);
```

FORK PROCESS <unistd.h>

```
pid_t pid = fork();
if (pid == -1) → error
if (pid == 0) → child
else → adult
```

CLEAN UP <sys/wait.h>

```
if (waitpid(pid, &status, 0))
    → error
if (WIFEXITED(status)) {
    → ok (WEXITSTATUS(...))
else if (WIFSIGNALED(status))
    → ok (WTERMSIG(...))
else → ???
```

```
while ((pid = waitpid(-1, &stat,
                      WNOHANG)
    if (pid == -1) {
        if (errno == ECHILD) break
        → error } }
```

BOUNDED BUFFER <stdint.h, limits.h>

```
struct BND {
    size_t size;
    volatile size_t r, w;
    SEM *full, *free;
    int tail[]; };
BND *bbCreate(size_t len) {
    BND *bnd =
    malloc(sizeof(BND) +
           len * sizeof(int));
    if (!bnd) → error
    bnd->len = len;
    bnd->r = bnd->w = 0;
    bnd->full = semC(0);
    if (!bnd->full) → error
    bnd->free = semC(len);
    if (!bnd->free) → error
    return bnd; }
```

```
void bbPut(BND *bnd,
           int val) {
    P(bnd->free);
    bnd->tail[bnd->w]
    = value;
    bnd->w++;
    bnd->w %= bnd->size;
    V(bnd->full); }
```

```
int bbGet(BND *bnd) {
    int pos, val;
    P(bnd->full);
    do { pos = bnd->r;
         val = bnd->tail[pos % bnd->size];
    } while (!CAS(&bnd->r, &pos, pos+1));
    V(bnd->free);
    return val; }
```

DNS <sys/socket.h>

```
struct addrinfo hints = {
    .ai_socktype = SOCK_STREAM,
    .ai_family = AF_UNSPEC,
    .ai_flags = ADDRCONFIG}, *head, *curr
struct int sock, error;
```

```
error = getaddrinfo("⊗", "1234", &hints, &head);
if (error == EAI_SYSTEM) → error
if (error) → error (gai_strerror(error))
```

```
for (curr = head; curr != NULL; curr = curr->ai_next) {
    sock = socket(curr->ai_family, curr->ai_socktype, curr->protocol);
    if (!connect(sock, curr->ai_addr, curr->ai_addrlen)) break;
    close(sock); }
if (sock == NULL) → error connect
freeaddrinfo(head);
```