Echtzeitbetrieb
Einhaltung von Terminen
Rechtzeitigkeit anstelle von Geschwindigkeit
Überwachung/Interaktion mit phys. Welt
Überschreiten von Terminvorgaben:
weich: Ergebnis weiterhin nutzbar
Terminverletzung tolerierbar
transparent für die Anwendung
fest: Abbruch der Berechnung(Ergebnis wertlos)
Start der Nächsten Berechnung durch BS
transparent für die Anwendung
hart: BS löst Ausnahmesituation aus
Ausnahmebehandlung führt System in sicheren Zustand
intransparent für die Anwendung
bevorzugte Kriterien: Dringlichkeit, Vorhersagbarkeit, Termineinhaltung
entstehende Konflikte: Gerechtigkeit, Lastausgleich

Stapelbetrieb Kriterien:
Durchsatz, Durchlaufzeit, Prozessorauslastung

Verklemmungen:
irreversible gegenseitige Blockierung von Prozessen
das Prozesssystem macht keinen Fortschritt, steht still
Notwendige Bedingungen:
1. Ausschließlichkeit der Betriebsmittelnutzung
2. Nachforderung von Betriebsmitteln
3. Unentziehbarkeit der Betriebsmittel
notwendig und hinreichend:
4. Zirkuläres Warten auf die Betriebsmittel
Vorbeugung: verhindert auftreten der Bedingungen
Vermeidung: durch Analyse der Prozesse usw. wird ein unsicherer Zustand (zirk. Warten) vermieden

Fragmentierungsarten
interne Fragmentierung: Blöcke haben vorgegebene Größe (bspw. Paging)
innerhalb der Blöcke bleibt Speicher ungenutzt
externe Fragmentierung: Blockgröße variabel (bspw. Segmentierung)
zwischen den Blöcken entsteht Verschnitt
Effekt bei interner Fragmentierung:
Eigentlich ungültige Zugriffe auf geschenkten Verschnitt werden nicht erkannt, weil die Adressen im Adressraum doch gültig sind
-> Programmfehler werden verschleiert

Prozess greift auf ausgelagerte Seite zu:
1. MMU erkennt fehlendes Present Bit → Trap (Prozess in blockiert)
2. MMU fordert Seite vom Hintergrundspeicher an und lagert sie ein
3. Prozess darf wieder in bereit dann in laufend und wiederholt Zugriff

Kriterien bei der Einplanung von Prozessen
benutzerorientiert: Das vom Nutzer wahrgenommene Verhalten
Antwortzeit: Minimierung Syscall -> Antwort
systemorientiert: effektive Auslastung von Betriebsmitteln
Durchsatz: Maximierung der Anzahl vollendeter Prozesse

Einplanungsstrategien:
kooperativ: Prozessen wird die CPU nicht zugunsten anderer Prozesse entzogen
Beispiel: FCFS, Anwendungsfall: Batch-Betrieb
präemptiv: Prozessen kann die CPU zugunsten anderer Prozesse entzogen werden
Beispiel: Round Robin; Anwendungsfall: Mehrbenutzerbetrieb

Freiseitenpuffer
speichert Seiten die ausgelagert werden sollen noch zwischen
falls man sie doch wieder braucht

Journaling File Systems
log-based recovery von Daten
alle Änderungen an Daten werden in log file gespeichert
bei einem Systemabsturz oder ähnlichem ist mit den log files eine Wiederherstellung eines konsistenten Zustands möglich

Nicht Blockierende Synchronisation (CAS)
Optimistischer Ansatz zur Synchronisation bei dem von wenigen Konflikten ausgegangen wird. Es wird durch Hardware ein wechselseitiger Ausschluss ermöglicht. Bei Konflikten wird der Änderungsversuch wiederholt.

Vorteile
keine teuren Systemaufrufe
in vielen Fällen keine Wiederholung notwendig
keine Verklemmungen
Nachteile
Kompliziert zu Implementieren
Verhungern von Prozesen mgl.
Hardware muss Funktionen wie CAS bereitstellen

Prozess/ Threadarten

Schwergewichtige	Leichtgewichtige(Kernel)	Federgewichtige(User)
eigener Adressraum	geteilter Adressraum	geteilter Adressraum
Umschaltung von BS	Umschaltung von BS	BS kennt sie nicht
echt parallel zueinander	echt parallel zueinander	nicht echt parallel
fork()	pthread_create()	kein Bsp. bekannt

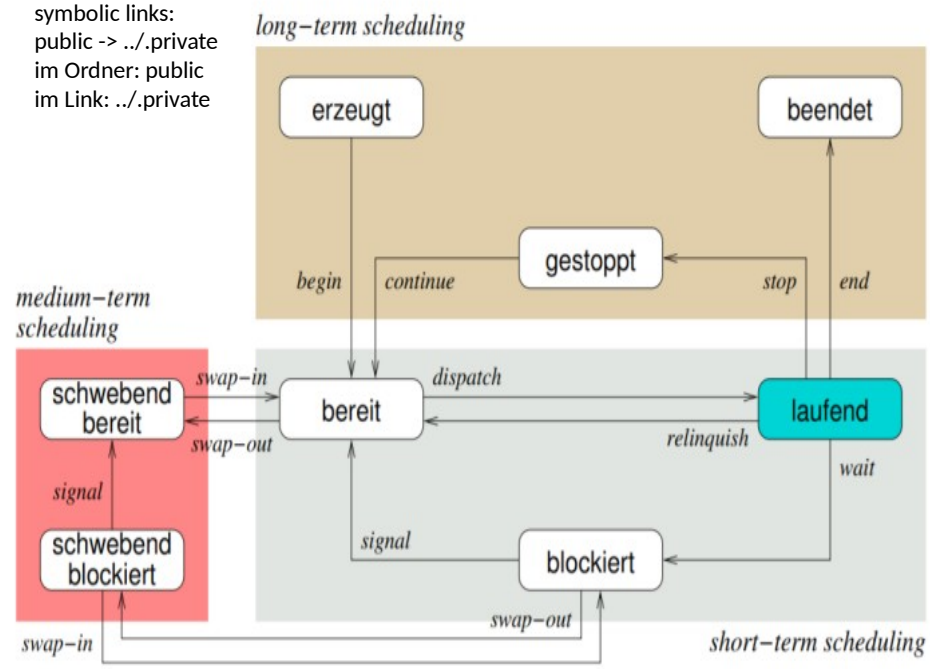
Betriebsmittelarten

Wiederverwendbar:	Konsumierbar:
dauerhaft, begrenzt	Vorübergehend,
persistent(nicht flüchtig)	unbegrenzt
Anforderung durch	transient (flüchtig)
mehrseitige Synchron.	Anforderung durch
Wiederverwendung	einseitige Synchronisation
nach Freigabe	Zerstoerung nach Freigabe

Semaphor:
Koordinationsmittel zur Synchronisation vom Zugriff auf Betriebsmittel
P(SEM*): verringert den Wert des Semaphors um 1,
ist der Wert vor dieser Aktion <= 0 blockiert der Prozess
V(SEM*): erhöht den Wert des Semaphors um 1
hebt ggf. die Blockierung eines wartenden Prozesses auf

Einseitige Synchronisation: ein Prozess synchronisiert und blockiert ggf.
Mehrseitige Synchronisation: mehrere Prozesse an Synchronisation beteiligt

symbolic links:
public -> ../.private
im Ordner: public
im Link: ../.private



- Aufteilung des Hauptspeichers eines Prozesses in Segmente
  - Vgl. Vorlesung A-III, Seite 7f.

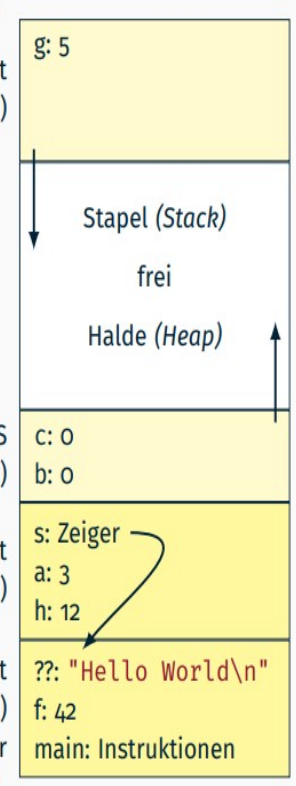
```
static int a = 3; static int b;
static int c = 0; const int f = 42;
const char *s = "Hello World\n";

int main(void) {
    int g = 5;
    static int h = 12;
}
```

- Compiler-Fehler

```
s[1] = 'a';
f = 2;
```

- Segmentation Fault

```
((char *) s)[1] = 'a';
*((int *) &f) = 2;
```



Checkliste falls fertig:
Alles gecloset und gefreet und destroyt (auch bei Fehlerbehandlung)
Bei void* Methoden return null? Alles static?
Semaphor Aufrufe nochmal durchgehen (in Threads auch im Fehlerfall V aufrufen)
Am Anfang argc geprueft? Am Ende ggf stdout gefflusht?
Noch irgendwas unnötiges dastehen lassen?

Pthreads:
```
static void *worker(void *args) {}
pthread_t pthread; // für join array
errno = pthread_create(&pthread, NULL,
&worker, args);
if (errno)
  die("pthread_create");
// join
errno = pthread_join(ausgabeTid, NULL);
if (errno)
  die("pthread_join")
// detach
errno = pthread_detach(pthread_self());
if (errno)
  perror("pthread_detach"); // nicht fatal
```

Fork:
```
pid_t p = fork();
if (p == -1) {
die("fork");
} else if (p == 0) { //Kindprozess
  // hier oft exit sonst lauft man
weiter
} else { //Elternprozess, p ist id
vom Kind
}
```

Funktionen mit Fehler
```
errno = pthread_{mutex, cond}_init(m/c*, NULL);
pthread_m/c_destroy(m/c*);
realloc(oldp, size) // NULL
fputc(c, tx) // EOF
fprintf(tx, "%d...\n“, 3); // < 0
strcpy(dest, src);
strcat(dst, src);
strcmp("abc", "abc") = 0
strncmp(s1, s2, n);
strdup(str) // NULL
close(fd) // -1
fclose(rx) // EOF
dup(fd) // -1
dup2(fd, STDOUT_NUM) // -1
fdopen(fd, "{r/w}") // NULL
```

Strtok_r
```
char *args[strlen(input) / 2 + 1];
char* saveptr; // nur bei _r notwendig
args[0] = strtok_r(input, " ", &saveptr);
int index = 1;
while ((args[index++] = strtok_r(NULL, " ", &saveptr)) !=
NULL);
// und index liegt bei Anzahl "!= NULL" Einträge + 2
```

Fgets Schleife:
```
char buf[LINE_LEN+ 1]; // LINE_LEN includes \n
while (fgets(buf, sizeof(buf),rx) != NULL) {
  size_t len = strlen(buf);
  if (len > sizeof(buf) - 2 && buf[len - 1] != '\n‘) {
    int c; // Überlange Zeile -> Alles weglesen
    while ((c = fgetc(rx)) != EOF) {
      if (c == '\n‘) break;
    }
    if (ferror(rx)) die("fgetc");
     continue;
  }
  if (buf[len - 1] == '\n')
    buf[len - 1] = '\0‘;
  // Zeile verarbeiten
}
if (ferror(rx))
    die("fgets");
```

Strtol
```
errno = 0;
char *endptr; // base 10:
long x = strtol(str, &endptr, 10);
if (errno)
    die("strtol");
if (str == endptr || *endptr != '\0‘) {
    fprintf(stderr, "invalid number\n");
    exit(EXIT_FAILURE);
}
```

Server Code
```
int sock= socket(AF_INET6, SOCK_STREAM, 0);
if (sock == -1)
    die("socket");
// hier ggf setsockopt
struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(port), // Port-Nummer
    .sin6_addr = in6addr_any,
};
if (bind(sock, (struct sockaddr *)&name,
sizeof(name)) == -1)
    die("bind");
if (listen(sock, SOMAXCONN) == -1)
    die("listen");
while (1) {
    int csock= accept(sock, NULL, NULL);
    if (csock == -1) {
        perror("accept"); // hier nie sterben
        continue;
    }
    handleConnection(csock, sock);
}
```

Directory durchgehen
```
char* path = ".";
DIR *dir = opendir(path);
if (dir == NULL)
    die("opendir");
struct dirent *dirent;
while (errno = 0, (dirent = readdir(dir)) != NULL){
    if(strcmp(dirent->d_name, "..") != 0)
        continue;
    // aktuellen Pfad zusammenbauen
    char npath[strlen(path) + strlen(dirent->d_name) + 2];
    if(sprintf(npath,"%s/%s", path, dirent->d_name) < 0){
        perror("sprintf");
        continue;
    }
    struct stat info;
    if(lstat(npath, &info) == -1){
        perror("lstat");
        continue;
    }
    if(S_ISREG(info.st_mode)){
        // do something
    } // auch S_ISDIR und mehr
}
if (errno)
    die("readdir");
 if (closedir(dir) == -1)
    die("closedir");
```

Signal Handler
```
static void handler(int signal){
  int tmp_errno = errno;
  // do something, kein perror!
  errno = tmp_errno
}
```

SigPipe Ignorieren
```
struct sigaction action = {
    .sa_handler = SIG_IGN,
    //.sa_flags = SA_RESTART,
}; // ggf. SIG_DFL oder &handler
sigemptyset(&action.sa_mask);
sigaction(SIGPIPE, &action,
NULL);
```

Makefile
```
.PHONY: all clean
all: clash
clean:
    rm -f clash clash.o plist.o
clash: clash.o plist.o
    gcc -o clash clash.o plist.o
clash.o: clash.c plist.h
    gcc -c clash.c
plist.o: plist.c plist.h
    gcc -c plist.c
```

Semaphor Implementierung:
```
void P() {
    pthread_mutex_lock(&m);
    while (counter <= 0) {
        pthread_cond_wait(&c,
&m);
    }
    counter--;
    pthread_mutex_unlock(&m)
}
void V() {
    pthread_mutex_lock(&m);
    counter++;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m)
}
```

CAS mit stdatomic.h:
```
volatile atomic_int x = 5;
int expected, desired;
do{
  expected = x; // oldvalue
  desired = expected * 2;
} while(!atomic_compare_strong
(&x, &expected,  desired));
```

Signale blockieren (Ignore siehe ServerCode)
```
sigset_t oldmask, mask;
sigemptyset(&mask);
sigaddset(&mask, SIGNAL); // SIGCHLD oderso
sigprocmask(SIG_BLOCK, &mask, &oldmask);
while (event == 0) { // ggf warten, event
volatile
  sigsuspend(&oldmask);
}  // sigsuspend wartet auf alle nicht in oldmask
event = 0;
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```