Definitionen:

Progr.: Folge von Anweisungen, kann von mehreren Prozessen gleichzeitig ausgeführt werden.

Prozess: Ein (!) in Ausführung befindliches Progr.

Compiler: Erzeugt aus mehreren Programmteilen (Modulen) ein Progr.

Binder: Erzeugt aus Objekt-Dateien ein Progr

Multi-Threading

Prozesse: Ausführungsumgebung mit einem Aktivitätsträger
fork();

Kernel-Level-Threads:
- Teurer als User-Level,
- Schedulingstrategie vom BS vorgegeben
- Systemkern sieht jeden Thread als eigenen Aktivitätsträger → Multi-Threading
- Gruppe v. Threads nutzt gemeinsame Betriebsmittel eines Prozesses pthread

User-Level-Threads:
- Billig
- Schedulingstrategie vom Programmierer vorgegeben
- Realisierung auf Anwendungsebene
- Systemkern sieht nur einen Kontrollfluss
  ↳ Kein (echtes) Multi-Threading
  ↳ Ein blockierender User-Thread blockiert alle User-Threads

Verklemmung

Eine irreversible Blockierung von Prozessen, bei der das Prozesssystem keinen Fortschritt mehr macht, bzw. still steht.

Notwendige Bed.:
- Ausschließlichkeit der Betriebsmittelnutzung
- Nachforderung von Betriebsmitteln
- Unentziehbarkeit der Betriebsmitteln

Hinreichende & notw. Bed.:
- Zirkulläres Warten auf Betriebsmittel

Verklemmungsvorbeugung

- Nicht blockierende Synchronisation
- Alle benötigten Betriebsmittel auf einmal atomar anfordern
- Betriebsmittel virtualisieren (→ reale können entzogen werden)
- Betriebsmittel in fester Reihenfolge zuteilen

Verklemmungsvermeidung

- Nur zirkuläres Warten wird durch laufende Analyse verhindert
- Vor jeder Betriebsmittelanforderung wird geprüft, ob System in unsicherem Zustand
  ↳ Bankiersalgorithmus, falls ja wird Anforderung zurückgewiesen

Betriebsmittelgraph: Zyklus → Verklemmung

Nicht-blockierende Sync.

Vorteile:
- Rein auf Anwendungsebene, keine teuren Systemaufrufe
- Geringere Mehrkosten als bei Locking, wenn CAS auf anhieb funktioniert
- Konkurrierende Threads werden vom Scheduler nach dessen Kriterien eingeplant
- Keine Abhängigkeit vom Halter d. Locks
- Verklemmungsfrei

Nachteile:
- Schwere algorithmische Umsetzung
- Verhungern (wenn lange Berechnung)
- Hardware-Unterstützung

Optimistischer Ansatz:
- Wenig Konkurrenz
- Falls doch Konflikt wird mit hardware-gesteuerten wechselseitigen Ausschluss behandelt (CAS)

Pessimistischer Ansatz:
- Viel Konkurrenz → Block. Sync.

Bei Monoprozessor ohne Verdrängung keine Synchronisation

Betriebsmittel:

Wiederverwendbar: Begrenzte Zugriff
Existieren dauerhaft (persistent)
Sind entweder teilbar e. unteilbar
Bsp. in Hardware: CPU, RAM, GPU
Bsp. in Software: Krit. Abschnitt

Konsumierbar: Unbegrenzter Zugriff
Existieren vorübergehend (flüchtig)
Werden erzeugt & wieder zerstört
Bsp. in Hardware: Signale, Traps
Bsp. in Software: Datenstrom

Prozesseinplanung

Kooperativ	Präemptiv
Kein CPU-Entzug ↳ CPU-Monopolisierung FCFS, Batch-Betrieb	CPU-Entzug RR, Mehrprgr.-Betrieb
Deterministisch	Probabilistisch
Prozesszeiten bekannt ↳ Zeitgarantien	Prozesszeiten unbek. Shortest Process Next
Statisch	Dynamisch
Kompletter Ablaufplan Echtzeitsysteme	Stapelbetrieb, schwache & feste Echtzeitsysteme
Asymetrisch	Symmetrisch
	Identische Prozessoren ↳ Lastausgleich

Echtzeitsysteme

- Einhaltung vorgegebener Termine
- Rechtzeitigkeit anstelle von Geschwindigkeit
- Vorhersagbarkeit
- Überwachung/Interaktion mit physikal. Welt

Weiche Vorgaben (soft):
- Ergebnis weiterhin nutzbar
- Terminverletzung tolerierbar
- Transparent f. Anwendung

Feste Vorgaben (firm):
- Terminverletzung tolerierbar
- Abbruch der Berechnungen (durch BS)
  ↳ Ergebnis wertlos + Start der nächsten Berechnung (durch BS)
- Transparent für Anwendung

Harte Vorgaben (hard):
- Terminverletzung nicht tolerierbar
- BS löst Ausnahmesituation aus
- Ausnahmebehandlung führt System in sicheren Zustand
- Intransparent für Anwendung

Kriterien f. Prozesseinplanung

Benutzerorientiert: Vom Nutzer wahrgenommene Verhalten
- Mgl. kurze Antwortzeit (bei Sysfalls)
- " Durchlaufzeit (Start bis Ende)
- Termineinhaltung
- Vorhersagbarkeit

Systemorientiert: Mgl. effektive Auslastung von Betriebsmitteln
- Mgl. hoher Durchsatz an vollendeten Prozessen
- Gleichmäßige Prozessauslastung
- Gerechtigkeit
- Dringlichkeit
- Lastausgleich

Prioritäten nach Betriebsart

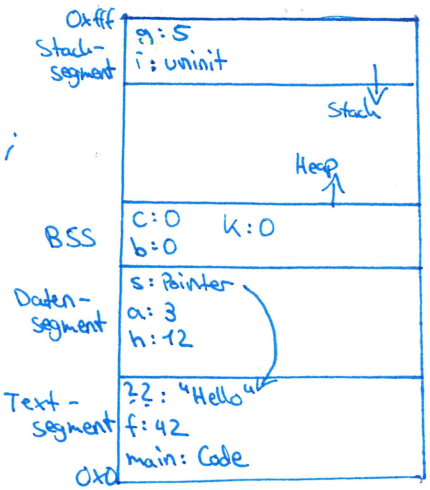
Allgemein: Gerechtigkeit, Lastausgleich
Stapelbetrieb: Durchsatz, Durchlaufzeit, Prozessorauslastung
Dialogbetrieb: Antwortzeit
Echtzeitbetrieb: Dringlichkeit, Termineinhaltung, Vorhersagbarkeit
Konflikt: Gerechtigkeit, Lastausgleich

Round Robin: FCFS mit Zeitscheibenverdrängung + period. Umplanung
VRR: Bevorzugung kürzerer Prozesse
Shortest Process Next: Verhungern möglich

```
Static int a = 3;
static int b;
static int c = 0;
const int f = 42;
const char *s = "Hello";
int main(void) {
    int g = 5;
    static int h = 12;
    int i;
    static int k;
}
```

Stack-Frame:

Alles was beim Fkt.-Aufruf und durch die Fkt. selbst auf den Stack gelegt wird.
Beim Rücksprung wird dieser Stack-Frame entfernt.



Adressraumschutz

→ Bei einem Prgr.

Abgrenzung/Abteilung: Ein Schutzgatterregister oberhalb erlaubten Bereich
Einfriedung/Eingrenzung: 2 Begrenzungsregister f. gültiges Intervall
↳ Bei beiden ext. Fragmentierung
Segmentierung: Basis- & Längenregister (↯ Pro Prgr. nur 1 Segment)

Fragmentierung:

Interne: Alle Blöcke gl. groß
Innerhalb der Blöcke ist der Speicher nur teilw. befüllt
↳ Kein Zugriffsfehler bei Zugriff auf ungenutzten Bereich

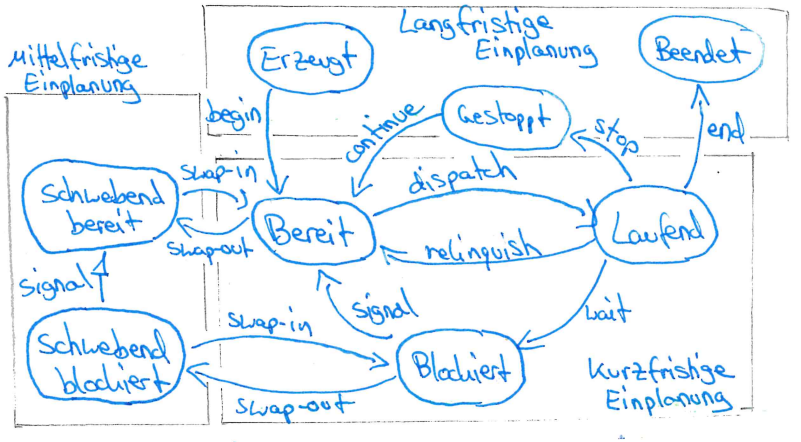
Externe: Blöcke unterschiedl. groß
Zwischen den Blöcken ist Speicher ungenutzt

Ersetzungsstrategien

FIFO, LRU: Am längsten
LFU: Am seltensten
2nd Chance: FIFO + period. use Bit
3rd Chance: ↳ + dirty-Bit bei mit 4. Zugriff Wiederholen

Page Fault

1. MMU erkennt nicht-gesetztes present bit → Trap
2. Wenn Seite ungültig: SIGSEGV (beendet)
3. Wenn Seite gültig: Seite wird angefordert und eingelagert (blockiert)
   - Falls in Freiseitenpuffer, nicht vom Hintergrundspeicher geladen werden
   - Falls keine freien Rahmen im Arbeitsspeicher: Ungenutzte Seite auslagern (nach Seitenersetzungsstrategie)
   - Seite in freien Rahmen einlagern und als present markieren (bereit)
4. Zugriff Wiederholen



Server-Socket

SIGPIPE ignorieren

```
struct sigaction a= {.sa_handler = SIG_IGN,};
sigemptyset(&a.sa_mask);
sigaction(SIGPIPE, &a, NULL);
```

```
int sock = socket(AF_INET6, SOCK_STREAM,0);
if(sock ==-1) die("socket");
struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port = htons(port),
    .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 client = accept(sock, NULL, NULL);
    if(client == -1){ perror("accept");
        continue;}
    ...
}
```

fgets:

```
char buf[n+1];
while(fgets(buf, sizeof(buf), stream)!=NULL){
 size_t len = strlen(buf);
 if(len > sizeof(buf)-2 && buf[len-1] != '\n'){
    int c;
    while((c=fgetc(stream)) != EOF){
       if(c == '\n') break;
    }
    if(ferror(stream)) die("fgetc");
    continue;
 }
 if(buf[len-1] == '\n')
    buf[len-1] = '\0';
 ....
}
if(ferror(stream)) die("fgets");
```

Funktionen

realloc(a, sizeof(a)+10*sizeof(int));

strcpy(dest, src); strcat(dest, src);

```
char *saveptr;
strtok_r(str, "/", &saveptr);
```

fgetc(rx) != EOF; fgets(buf,size,rx) != NULL

fputc(rx) != EOF; fputs => fprintf ∇

Signale

Signal-Handler

```
struct sigaction a = {
    .sa_handler = &signalHandler,
    .sa_flags = SA_RESTART,};
sigemptyset(&a.sa_mask);
sigaction(SIGUSR1, &a, NULL);
ooo
static void signalHandler(int s){
    int tmperrno = errno;
    ...
    errno = tmperrno;
}
```

Signal blockieren

```
static volatile e =0;
...
e=0;
sigset_t oldmask, mask;
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigprocmask(SIG_BLOCK, &mask,
    &oldmask);
```

Auf Signal warten

```
while(e==0)
    sigsuspend(&oldmask);
```

Signal deblockieren

```
sigprocmask(SIG_SETMASK,
    &oldmask, NULL);
```

strtol

```
errno = 0;
char *endptr;
int x = strtol(str, &endptr, 10);
if(errno !=0) die("strtol");
if(str==endptr || *endptr != '\0'){
    fprintf(stderr, "invalid nb.\n");
    exit(EXIT_FAILURE);
}
```

Makefile

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

Waitpid

```
int status;
pid_t child = waitpid(-1, &status, WNOHANG);
if(child == -1) perror("waitpid");
else if(child ==0) //no childs terminated
else{...
  if(!WIFEXITED(status)){
     fprintf(stderr, "%d\n",
        WEXITSTATUS(status));
  }
}
```

Ganzes Directory

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

Mutex (Semaphore)

```
static volatile int i = 0;
static pthread_mutex_t m;
static pthread_cond_t c;

void PC(){
    pthread_mutex_lock(&m);
    while( i <=0)
       pthread_cond_wait(&c, &m);
    i--;
    pthread_mutex_unlock(&m);
}

void VC(){
   pthread_mutex_lock(&m);
   i++;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
}
```

Semaphor

Koordinationsmittel zur Synchronisation von Betriebsmitteln. Atomarer Zähler:
PC(): Blockiert bei ≤0 (vorher dekr.)
VC(): Hebt Blockierung evtl. auf & erhöht

dup2

```
int fd = creat("path", 0);
if(fd == -1) die("creat");
if(fflush(stdout) == EOF)
   die("fflush");
if(dup2(fd, STDOUT_FILENO)
     == -1) die("dup2");
if(close(fd) == -1) perror("close");
```

Int-Overflow

```
if(x+y >
   INT_MAX)..
```

Atomics

```
atomic_int a = ATOMIC_VAR_INIT(10);
int d = atomic_load(&a);
atomic_store(&a, 8);
atomic_fetch_add(&a, 1); //a++;

int old, local;
do{ old = atomic_load(&a);
    local = old *2 //f(old)
}while(!atomic_compare_exchange_strong(
      &a, &old, local));
```

Funktionen (2):

```
pthread_t p;
errno = pthread_create(&p, NULL, f, args);
pthread_detach(pthread_self());
errno = pthread_join(p, NULL);
errno = pthread_mutex_init(&m, NULL);
errno = pthread_cond_init(&c, NULL);
```

FILE *fopen("path", "r"); fclose(FILE *) //-1
FILE *fdopen(fd, "r"); close(fd) //-1
open("path",0) //-1; fileno(FILE *)

Journaling-FS:

Alle Änderungen an Daten & Metadaten werden vor Änderung als Transaktion (mit Infos zu Undo & Redo) protokolliert. Erfolgr. Ausführung wird auch protokolliert. Nach Absturz kann System wieder in einen konsistenten Zustand gebracht werden. Nach Erreichen best. Checkpoints wird Log-File gelöscht.

Platzierungsstrategien

Nach Größe sortiert:

Worst-	Best-Fit
schnelle	langsame Suche
größer	kleiner Verschnitt

Nach Adresse sortiert:

First-	Next-Fit
schnelle Suche	Round Robin
größer Verschnitt	v. First-Fit

Checkliste

- fflush stdout
- perror bei printf<0
- Argc immer prüfen!
- Alles static!

Inode:

Eigentümer, Dateigr., T4p, Letzter Zugriff
Page: Rechte, Present

RAID:

RAID 0: Verteilt
1: Kopien
4: XOR
5: XOR verteilt

Trap: Synchron, vorhersagbar
Interne Ursache
Beendigung / Wiederaufnahme
Interrupt: Asynchron, unvorhersagbar
Ext. Ursache
Wiederaufnahme